Reinforcement Learning - Autumn '21
Assignment 2
Department of Computing
MSc Advanced Computing

Aaron Hoffman - CID: 02129114

29th of November 2021

1 Implementing a functional DQN

Describe briefly how you implemented each of these three features

To implement the replay buffer, I used the already given *ReplayBuffer* class (line 53), since it already implemented the required functionality. The only necessary addition was increasing its size in the constructor.

Regarding the target network, to implement it I make a deep copy of *policy_net* before starting the training by using the *deep_copy* (line 196) function from python. I also added a parameter J that tells the training loop every how many iterations to update the target network (*copy_net*) (line 224). Moreover, in the function *optimize_model*, when evaluating the next state values and actions, I use the *copy_net* instead of *policy_net* (line 154).

Finally, to implement the multiple k frames, I first added *gym.wrappers.FrameStack* to the environment, which takes k frames for each step we take, returning $(s_{t-k}, ..., s_{t-1}, s_t)$ states, and the same for the rewards, actions and next states before adding them to the replay buffer. Then, I had to modify *ReplayBuffer.sample* so it added each element of a transition to a different array in the correct size (line 64). Finally, in the *optimize_model* function, I flattened all the states so they become of shape $(4 * k)$. I also tried a different implementation by using a 1D CNN (line 103) to process the inputs stacked as $(k, action_dim)$, but the results were worse.

Discuss briefly your design decisions as well as choices of hyper parameters for the learning task.

The parameters we need to consider are: ϵ, γ the decay of ϵ, the width and depth of the neural network, k, the size of the replay buffer, the optimizer we are using and its learning rate.

The choice of k used is 4, since it is the same used in [2] and [3] and also because after testing $k = 1$, $k = 2$ and $k = 4$, $k = 4$ resulted sometimes in a better curve. However, as will be evidenced in question 3, the difference between the values of k was not really high, and definitely not consistent except for extreme values, such as $k = 8$.

In the case of ϵ, I assigned it to $\epsilon = 0.2$ with a decay of 0.99. This is because we don't want very high values (ie: $\epsilon > 0.5$) as this would lead to an extremely random agent that, although it would explore quite a lot, it would not reach high rewards in our environment. However, with a number of epochs big enough and decay this exploration could yield better results in the long run, but because of the large run time this would mean, I decided to test lower values. After testing $0.1 < \epsilon < 0.4$, the better result seemed to be achieved with 0.2. The decay was added after attempting different values in the range of $[0.999, 0.9]$ so that $\epsilon = \epsilon * decay$, and obtained the best results with 0.99. However, this results were only marginally better.

Regarding the size of the network, I started using a small network (depth of 2, with 100 neurons in each layer), but as soon as k became bigger, it became necessary to scale up the width up to 100 (**Figure 1**). This makes sense since increasing k creates more relationships between the inputs, and thus we need a more complex function to model them. The value of γ was selected by testing, and fixed at 0.995. The testing range was $[0.995, 0.8]$. As to why this range, we know that in this specific environment, the model can fall in scenarios where it simply cannot recover no matter the actions we take (for example, a fast x-axis velocity and about to reach a terminal state). Thus, we want a model that considers the long-term rewards, which is what γ represents. The model seemed very sensible to this value since a small change on it caused it to perform considerably worse.

For the size of the replay buffer I tested powers of 10. From $10,000$ and above I saw no difference, and achieved a better performance with that value than with lower ones.

Finally, when choosing the optimizer I only had two choices in mind: RMSProp and Adam.

RMSProp was the default one, and the one used in [1]. However, from my experience, Adam tends to perform really well on many applications. After testing both of them, Adam gave me better results as can be seen in **Figure 2** and **3**.

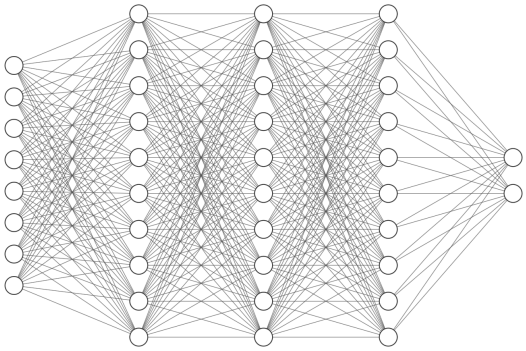


Figure 1: Real sizes: 16 for input layer, 100 for the three hidden layers, and two for the final layer

Show the learning curve for your DQN model

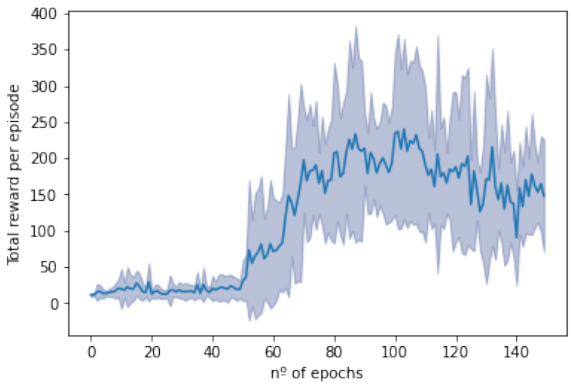


Figure 2: Mean reward and standard deviation with Adam

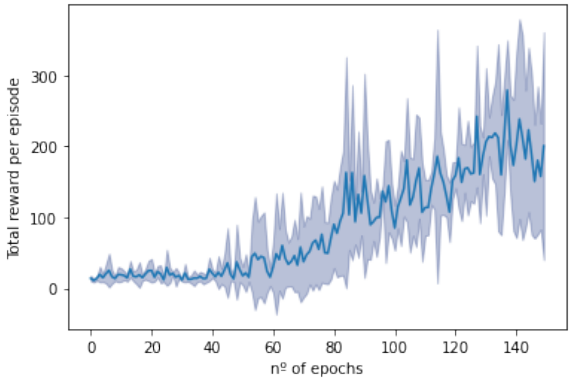


Figure 3: Mean reward and standard deviation with RMSProp

The mean and standard deviation was calculated with 10 runs of the model. This already took more than 30 minutes of computation, so a higher value starts getting harder to achieve. The range plot is the same as the number of epochs, and 150 was chosen because the curve simply plummeted at around that value. The range of the y-axis is just the maximum value of the model. The model achieves 90% of its final performance at around 70 epochs with a mean value of 120, and at 80 epochs if we look at the 90% of its best performance.

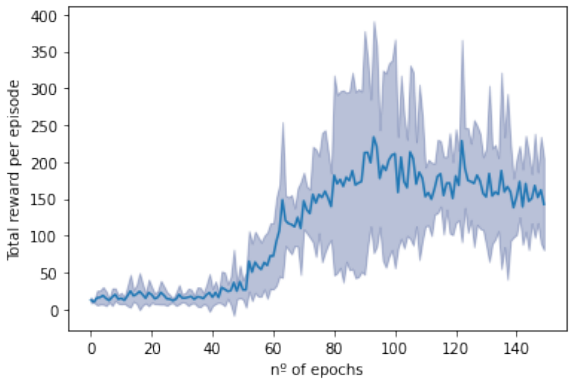


Figure 4: $\epsilon = 0.2, 0.99$ of decay

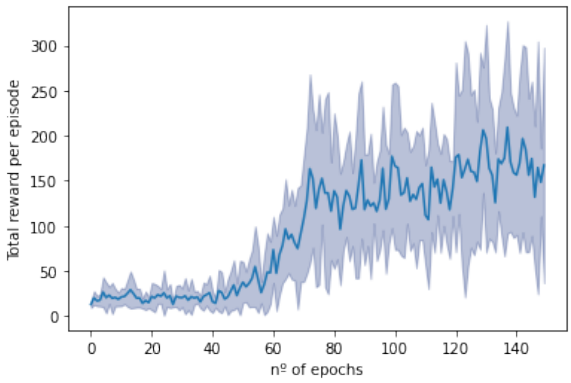
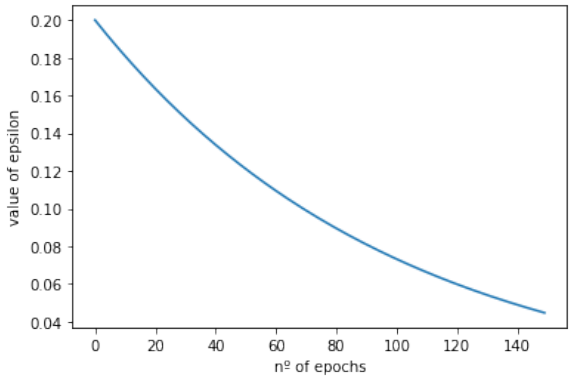
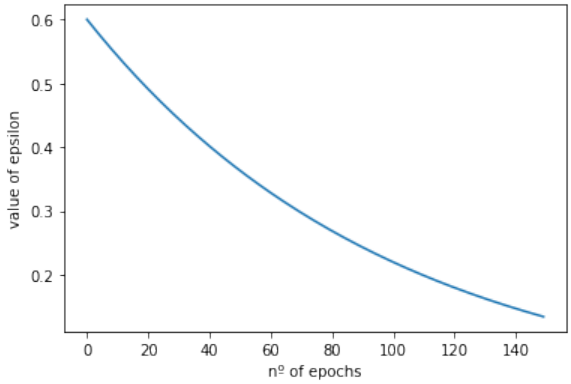


Figure 5: $\epsilon = 0.6, 0.99$ of decay





2 Hyper-parameters of the DQN

Investigate through computer experiments how the exploration parameter ϵ affects the performance (and stability) of learning.

By looking at **Figure 4** and **Figure 5**, we can see how a lower value of ϵ leads to better results early on with a maximum of around 220, whereas the growth of the rewards with $\epsilon = 0.6$ is slower and plateaus at around 150. It makes sense that the growth of the rewards is slower with a bigger ϵ since even if the neural network learns, it will still make more mistakes during the training phase because of ϵ. Moreover, an ϵ decay isn't strictly necessary if ϵ is low enough. This is because with a value lower value of ϵ, like $\epsilon = 0.2$, there is only a 20% chance of the model taking random decisions, which does not impede good performance in this environment. Adding decay would lower that chance later on, which can improve performance like in this case. Decay does become necessary if we have a high ϵ value. If we think what would happen if we had $\epsilon = 0.5$, the model would be choosing random actions 50% of the time, during all epochs, and thus would make it very hard to learn what actions it should perform.

Investigate through computer experiments how the variability in reward during learning depends on the size of the replay buffer.

We can see that we have a lower variability with more elements in the replay buffer. This is in line with what is expected since with a larger replay buffer it becomes more likely that the model experiences the same states, whereas with a smaller one, since the states get removed faster while new states come in, the model has to adapt to new episodes, leading to more variability. The buffer sizes chosen were: $20, 200, 500, 1000, 10000, 100000$ and 1000000.

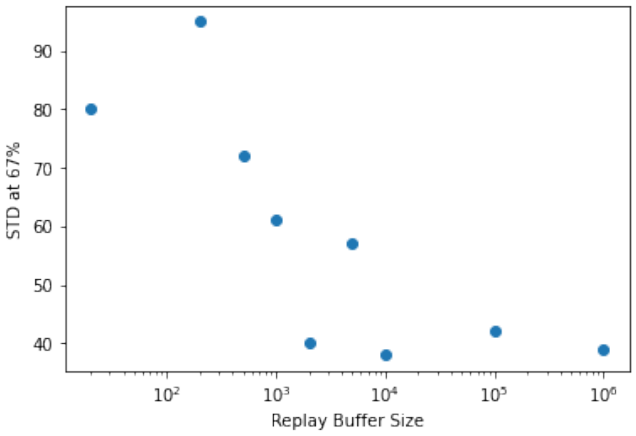


Figure 6: STD at 67% of the model performance for different buffer sizes

Investigate through computer experiments how the parameter $k = 1, 2, \ldots$ which sets how many frames are presented to the input, relates to overall learning and final performance.

The initial range of k that I wanted to explore ranged from $k = 1$ to $k = 5$. This was because it did not seem logical to have extremely big values of k, as it would be very hard to get a neural network capable of using the information of many different states. 5 states already seemed excessive. However, after looking at **Figure 7** and **Figure 8** generated from $k = 1$ up to $k = 5$, I saw little difference. To check if k really had an effect on learning, I added $k = 8$, where I saw a slightly worse performance than the rest.

For clarity, I've kept out the standard deviations of $k = 2$ and $k = 3$. We can see that the value of k does not have much impact on the overall learning. This can be because our model already gets enough information from just the current states and stacking more frames is simply redundant.

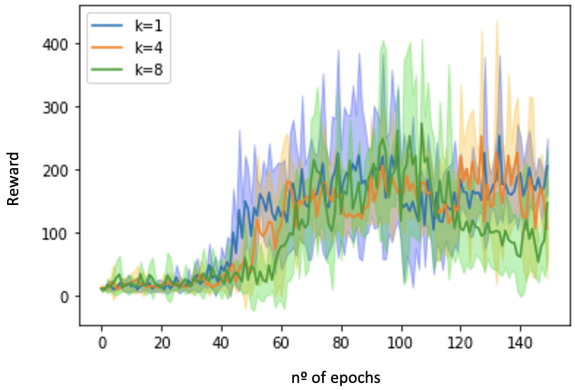


Figure 7: Mean reward and standard deviation of the model for $k = 1, 4, 8$

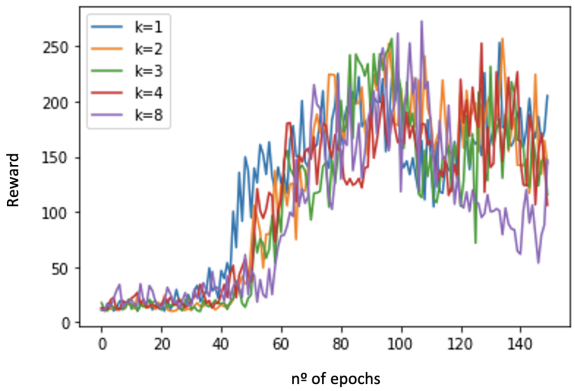


Figure 8: Mean reward of the model for each k value

3 Ablation/Augmentation experiments

Briefly discuss your DDQN implementation

Implementing DDQN is straightforward with the code we already have. All we need to do is modify how we compute $Q(S_{i+1}, a)$. In DQN with a target network, I used [3]:

$$Y_t = R_t + \gamma max_a Q(s_{t+1}, a, \theta')$$

to get the expected reward. When using DDQN however, this mutates into:

$$Y_t = R_t + \gamma Q(s_{t+1}, argmax_a Q(s_{t+1}, a, \theta), \theta')$$

Where θ' refers to the target network, and θ refers to the other network (line 335). This means we choose the action with one of the networks, but evaluate the value function using the other. Supposedly, using DDQN should achieve better results since DQN tends to overestimate the Q function. However, in my case, DDQN did not perform substantially better than DQN as can be seen in the graph. Only after increasing the value of k did I started seeing better results for the DDQN

Plot the learning curves for these 3 modifications against the original model (from Question 1) in a single plot and discuss your findings.

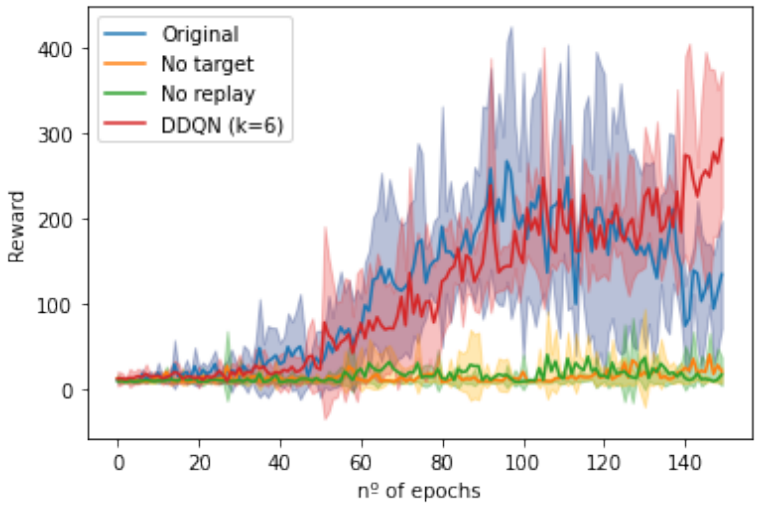


Figure 9: Network performance with ablation of different features

The network without replay was simply unable to learn. Even after changing different parameters, the network still did not perform well. The same happened without a target network except that, in this case, with a bigger neural network the model was able to learn yet it still performed considerably worse. It makes sense that the network performs worse without replay, not having it facilitates "catastrophic forgetting" [2]. Moreover, the target network is necessary because every time we update the network, we would be affecting the current state, but also the next state and so on. Why a bigger neural network mitigates this? It can be because with more parameters, we can model more specific states such that a change in one state does not alter others as much.

DDQN outperformed the original network but only after increasing k. It also seems that it takes more epochs to reach a maximum as can be seen in **Figure 10**. The reason we may need

more epochs for DDQN could rely on the explanation given in [3]. Here, the author states that DQN tends to overestimate $Q(s, a)$, and that can be beneficial depending on how the network overestimates. In this case, that overestimation may be leading to a faster convergence to an optimal value, but that optimal value can easily be worse.

The necessity of increasing k became apparent after a first attempt with $k = 4$, in which DDQN did worse than DQN, which is plotted in **Figure 11**.

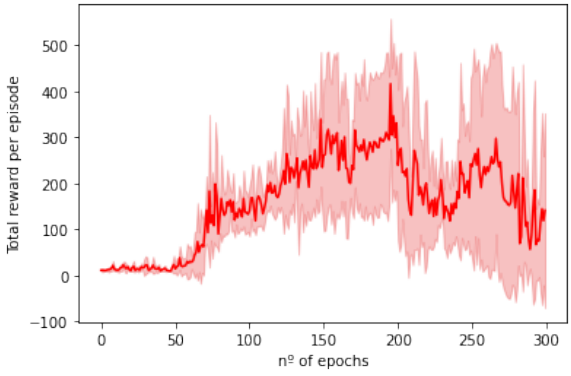


Figure 10: DDQN with 300 epochs and $k = 6$

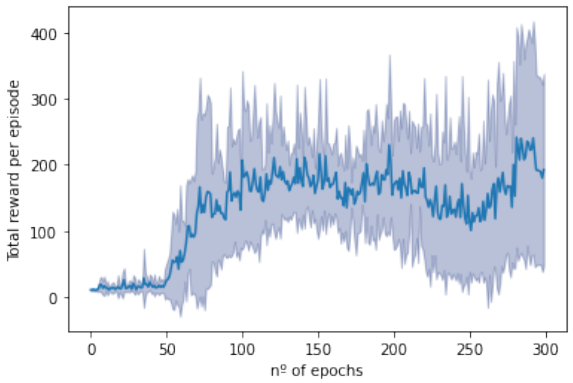


Figure 11: DDQN with $k = 4$

Bibliography

[1] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *nature* 518.7540 (2015), pp. 529–533.

[2] Melrose Roderick, James MacGlashan, and Stefanie Tellex. "Implementing the deep q-network". In: *arXiv preprint arXiv:1711.07478* (2017).

[3] Hado Van Hasselt, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 30. 1. 2016.

```
# -*- coding: utf-8 -*-
"""CW2_RL.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1uKNLxgvXMILhLXGVqyhheyFC9o-t8ZIl
"""

# This is the coursework 2 for the Reinforcement Leaning course 2021 taught at Imperia
# The code is based on the OpenAI Gym original (https://pytorch.org/tutorials/intermed
# There may be differences to the reference implementation in OpenAI gym and other sol


# 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

from IPython.display import clear_output
clear_output()

# Importing the libraries

import gym
from gym.wrappers.monitoring.video_recorder import VideoRecorder    #records videos of
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 Pyt

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


# 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)

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

```
class ReplayBuffer(object):

    def __init__(self, capacity, k_frames=1):
        self.memory = deque([],maxlen=capacity)
        self.k_frames = k_frames
    def push(self, *args):
        """Save a transition"""
        self.memory.append(Transition(*args))
    def __len__():
        return len(self.memory)

    def sample(self, batch_size):
        result = []
        actions, states, rewards, next_states = [], [], [], []
        for i in range(batch_size):
            random_index = np.random.randint(len(self.memory))

            state, action, next_state, reward = zip(self.memory[random_index])

            states.append(state[0][0])
            actions.append(action[0][0])
            rewards.append(reward[0][0])
            if next_state[0] != None:
                next_states.append(next_state[0][0])
            else:
                next_states.append(None)


        return states, actions, rewards, next_states

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

class DQN(nn.Module):

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

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

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

```
        return self.output_layer(x)

class DQN_CNN(nn.Module):

    def __init__(self, inputs, k, outputs, batch_size,num_hidden, hidden_size):
        super(DQN_CNN, self).__init__()
        self.batch_size = batch_size
        self.input_layer = nn.Conv1d(in_channels=k, out_channels=32, kernel_size=3)
        self.post_conv_layer = nn.Linear(32*2, hidden_size)
        self.hidden_layers = nn.ModuleList([nn.Linear(hidden_size, hidden_size) for _ in
        self.output_layer = nn.Linear(hidden_size, outputs)

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

        x = F.leaky_relu(self.input_layer(x))
        x = F.leaky_relu(self.post_conv_layer(x.view(x.shape[0], -1)))
        for layer in self.hidden_layers:
            x = F.leaky_relu(layer(x))

        return self.output_layer(x)

def optimize_model(memory):
    if len(memory) < BATCH_SIZE:
        return
    state_batch, action_batch, reward_batch, next_state_batch = memory.sample(BATCH_SIZE

    # 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([i is not None for i in next_state_batch]), devi


    state_batch = torch.stack([torch.flatten(a) for a in state_batch])
    action_batch = torch.stack([torch.flatten(a) for a in action_batch])
    reward_batch = torch.stack([torch.flatten(a) for a in reward_batch])


    # 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

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

    # Compute V(s_{t+1}) for all next states.
    # This is merged based on the mask, such that we'll have either the expected
```

```
    # state value or 0 in case the state was final.
    next_state_values = torch.zeros(BATCH_SIZE, device=device)
    with torch.no_grad():
        next_state_values[non_final_mask] = copy_net(non_final_next_states).max(1)[0]

    # Compute the expected Q values
    expected_state_action_values = (next_state_values * GAMMA) + reward_batch.flatten()
    # Compute loss
    loss = ((state_action_values - expected_state_action_values.unsqueeze(1))**2).sum()

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

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

    optimizer.step()
    return loss

NUM_EPISODES = 150
BATCH_SIZE = 20
GAMMA = 0.995

epsilon = .3
num_hidden_layers = 3
size_hidden_layers = 100

MEM_SIZE = 10000

k = 4
J = 10
eps_decay = 0.995

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

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

policy_net = DQN(state_dim*k, n_actions, num_hidden_layers, size_hidden_layers).to(devic
copy_net = deepcopy(policy_net)

optimizer = optim.Adam(policy_net.parameters())
memory = ReplayBuffer(MEM_SIZE, k)
```

```
losses = []
rewards = []

def select_action(k_steps, k, current_eps=0):
    sample = random.random()
    if sample > current_eps:
        t = torch.cat(list(k_steps)).flatten()
        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(t).argmax().view(1,1)
    else:
        return torch.tensor([[random.randrange(n_actions)]], device=device, dtype=torch.

best_reward = 0
losses = []
rewards = []
for i_episode in range(NUM_EPISODES):
    if i_episode % 20 == 0:
        print("episode ", i_episode, "/", NUM_EPISODES, " best reward: ", best_reward)
    if i_episode % J == 0:
        copy_net.load_state_dict(policy_net.state_dict())

    # Initialize the environment and state

    state = torch.tensor(env.reset()).float().unsqueeze(0).to(device)
    episode_loss = []
    R = 0
    epsilon = max(epsilon * eps_decay, 0.1)
    for t in count():
        # Select and perform an action
        action = select_action(state, k, epsilon)
        next_state, reward, done, _ = env.step(action.item())
        reward = torch.tensor([reward], device=device)

        # Observe new 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

        R += reward
```

```
        # Perform one step of the optimization (on the policy network)
        loss = None

        if len(memory) >= k:
            loss = optimize_model(memory)
        if loss:
          losses.append(loss)
        if done:
            break
    if best_reward < R and R > 100:
        best_reward = R
        torch.save(policy_net.state_dict(), 'best_model.pth')
    rewards.append(R)
print('Complete')

env.close()

plt.plot([i for i in range(len(rewards))], rewards)

def train(rb):
    global epsilon
    global env
    global policy_net
    global copy_net
    global optimizer
    global memory
    global k
    global eps_decay
    global MEM_SIZE
    global BATCH_SIZE

    eps_decay = 0.99
    k = 4
    epsilon = 0.2
    best_reward = 0
    eps_values = []
    BATCH_SIZE = 20
    MEM_SIZE = 10_000
    # Get number of states and actions from gym action space
    env = gym.make("CartPole-v1")
    env = gym.wrappers.FrameStack(env, k)

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

    policy_net = DQN(state_dim*k, n_actions, num_hidden_layers, size_hidden_layers).to(d
    copy_net = deepcopy(policy_net)
```

```
optimizer = optim.RMSprop(policy_net.parameters())
memory = ReplayBuffer(MEM_SIZE, k)

losses = []
rewards = []
def optimize_model(memory):
    if len(memory) < BATCH_SIZE:
        return
    state_batch, action_batch, reward_batch, next_state_batch = memory.sample(BATCH_

    # 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([i is not None for i in next_state_batch]),


    state_batch = torch.stack([torch.flatten(a) for a in state_batch])
    action_batch = torch.stack([torch.flatten(a) for a in action_batch])
    reward_batch = torch.stack([torch.flatten(a) for a in reward_batch])


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

    # Compute V(s_{t+1}) for all next states.
    # This is merged based on the mask, such that we'll have either the expected
    # state value or 0 in case the state was final.
    next_state_values = torch.zeros(BATCH_SIZE, device=device)
    with torch.no_grad():
        actions = policy_net(non_final_next_states).max(1)[1]
        next_state_values[non_final_mask] = copy_net(non_final_next_states)[:, actio

    # Compute the expected Q values
    expected_state_action_values = (next_state_values * GAMMA) + reward_batch.flatte
    # Compute loss
    loss = ((state_action_values - expected_state_action_values.unsqueeze(1))**2).su

    # 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()
    return loss
for i_episode in range(NUM_EPISODES):
    if i_episode % 20 == 0:
        print("episode ", i_episode, "/", NUM_EPISODES)
    if i_episode % J == 0:
        copy_net.load_state_dict(policy_net.state_dict())

    # Initialize the environment and state

    state = torch.tensor(env.reset()).float().unsqueeze(0).to(device)
    episode_loss = []
    R = 0
    eps_values.append(epsilon)
    epsilon *= eps_decay
    for t in count():
        # Select and perform an action
        action = select_action(state, k, epsilon)
        next_state, reward, done, _ = env.step(action.item())
        reward = torch.tensor([reward], device=device)

        # Observe new 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

        R += reward
        # Perform one step of the optimization (on the policy network)
        loss = None

        if len(memory) >= k:
            loss = optimize_model(memory)
        if loss:
            losses.append(loss)
        if done:
            break
    if best_reward < R and R > 100:
        best_reward = R
        # torch.save(policy_net.state_dict(), 'best_model.pth')
    rewards.append(R)
print('Complete')
env.close()
```

```
    return rewards, eps_values

all_rewards = []
all_eps = []
for e in range(5):
    rewards, eps = train('')
    all_rewards.append(rewards)
    all_eps.append(eps)

all_rewards_1 = torch.stack([torch.cat(r) for r in all_rewards])

mean = all_rewards_1.mean(0).cpu()
std = all_rewards_1.std(0).cpu()

plt.plot([i for i in range(len(mean))], mean)
plt.fill_between([i for i in range(len(mean))],
      mean - std,
      mean + std, color=(0.1, 0.2, 0.5, 0.3))
plt.xlabel("nº of epochs")
plt.ylabel("Total reward per episode")
plt.show()

'''
k_values = [1, 2, 3, 4, 8]
for idx, mean in enumerate(means_storage):
    plt.plot([i for i in range(len(mean))], mean, label='k=' + str(k_values[idx]))
    std = std_storage[idx]

plt.legend(loc="upper left")
plt.show()
'''

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

observation = env.reset()
done = False

state = torch.tensor(env.state).float().unsqueeze(0).to(device)
duration = 0
policy_net.load_state_dict(torch.load('best_model.pth'))

while not done:
    recorder.capture_frame()

    # Select and perform an action
```

```
    action = select_action([state], 1)
    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).to(device)

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



len(rewards)
```