

# Reinforcement Learning

## Coursework 2: CartPole Environment

Charlize Yang

CID: 01666113

Department of Computing

MSc Artificial Intelligence

November 29, 2021

## 1 Implementing a functional DQN

### 1.1 Implementing three features of DQN

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

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

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

## 1.2 Design decisions of the deep network

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

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

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

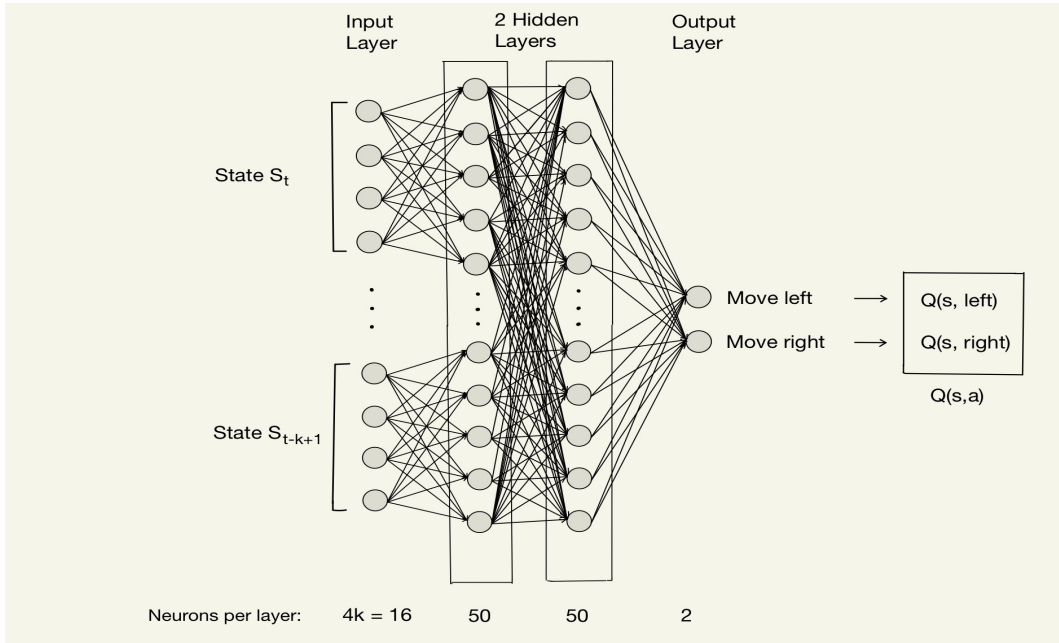


Figure 1: The architecture of the deep network.

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

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

### 1.3 Learning curve of the DQN

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

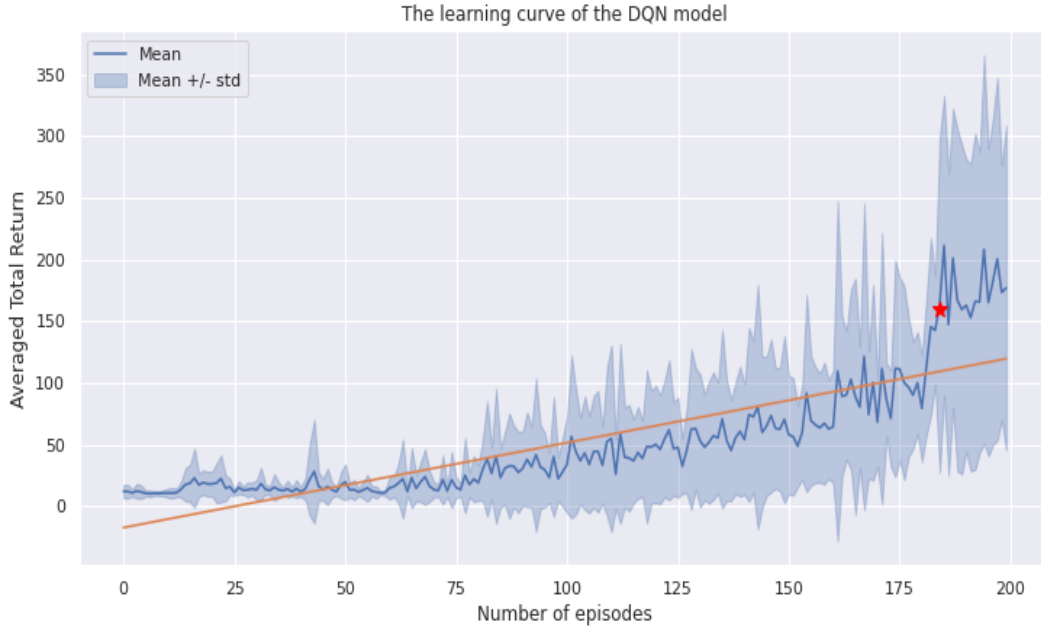


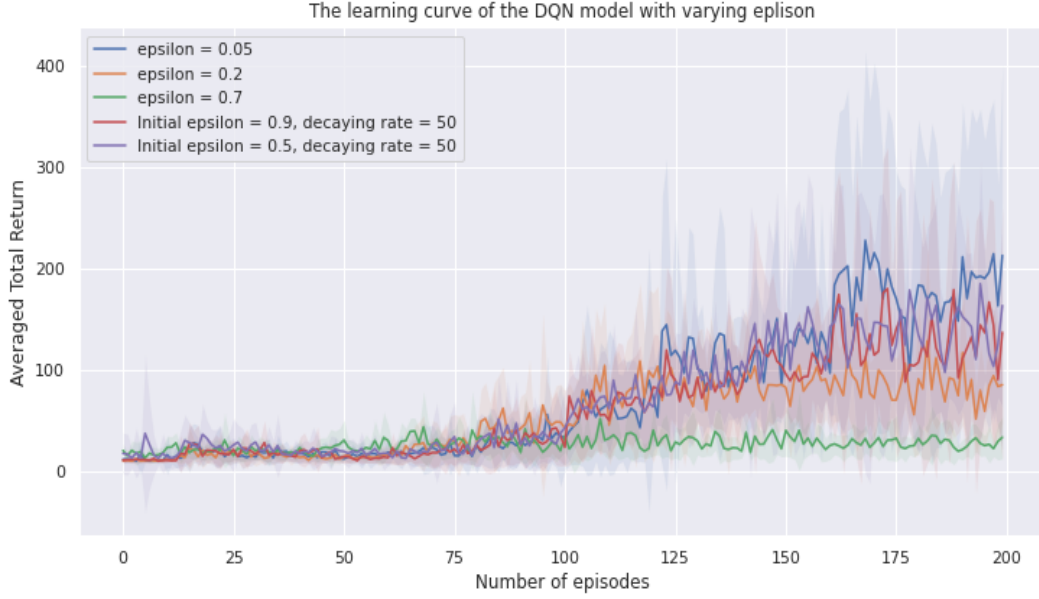
Figure 2: The learning curve of the DQN with (hyper)parameters in Table 1.

## 2 Hyperparameters of the DQN

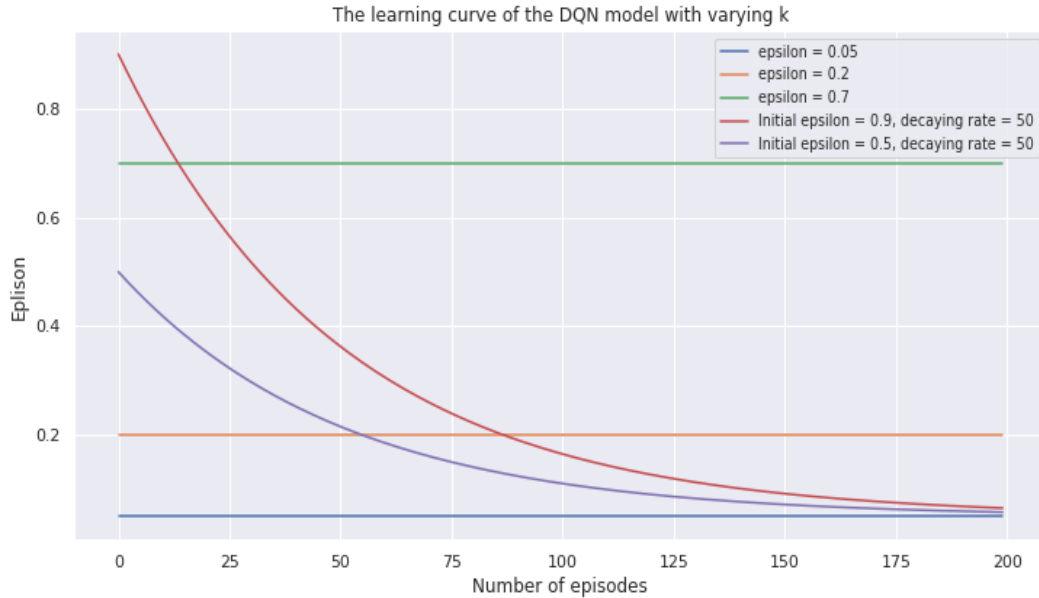
### 2.1 Constant and Variable Eplisons

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

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



(a)



(b)

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

## 2.2 Size of replay buffer

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

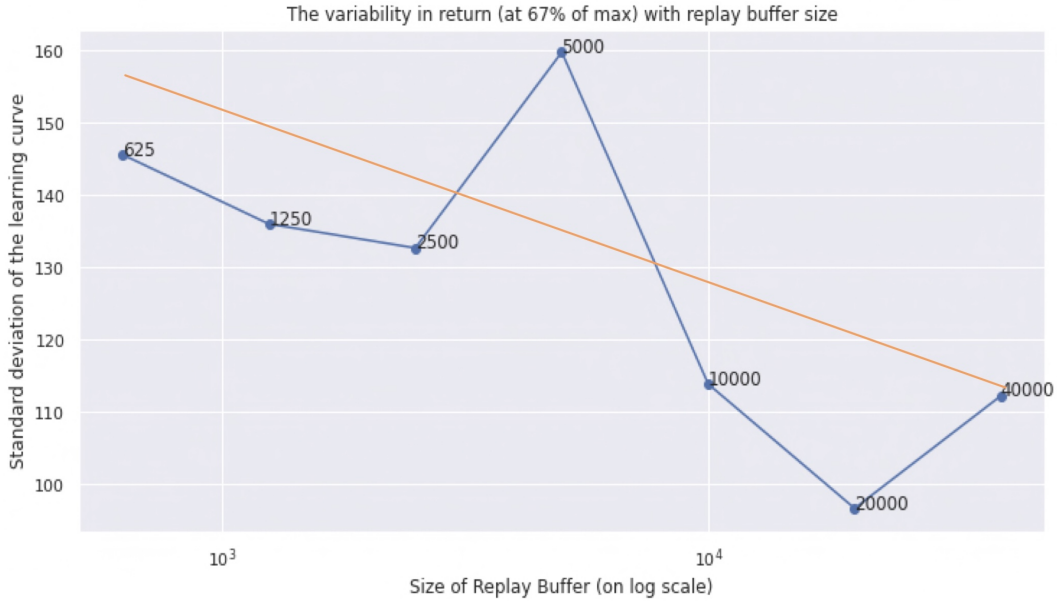


Figure 4: The variability in reward with varying replay buffer size (on the logarithmic scale). The standard deviation of the learning curve is taken at the 67% of the maximum total return.

## 2.3 Values of $k$

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

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

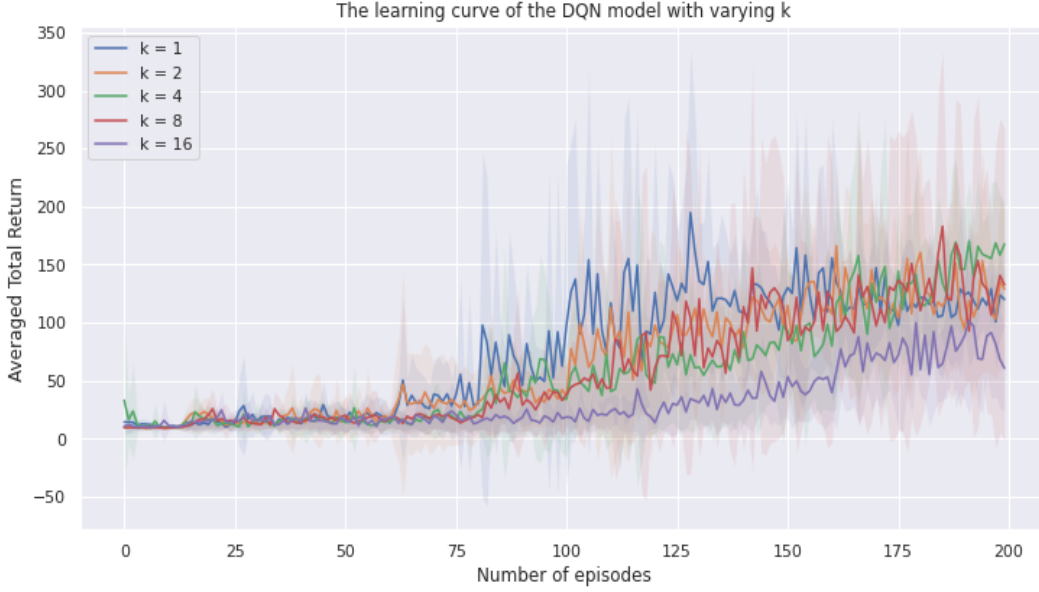


Figure 5: The learning curve of the DQN with varying values of  $k$ .

### 3 Ablation/Augmentation experiments

#### 3.1 Implementation of Double DQN

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

#### 3.2 Comparison of learning curves

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

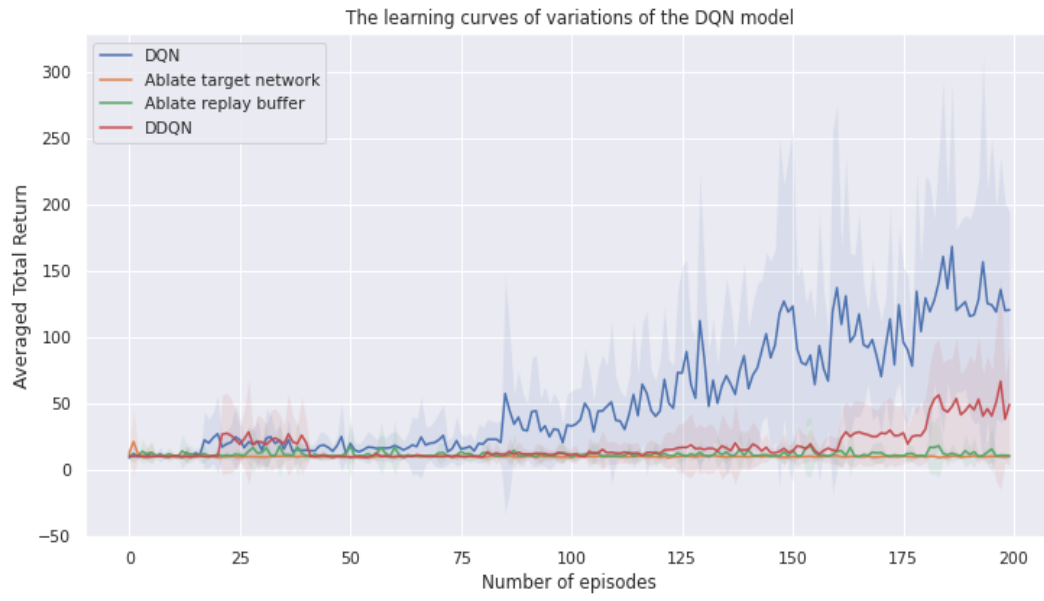


Figure 6: The learning curves of variations of the DQN model.

## 4 Code Appendix

```
1 # -*- coding: utf-8 -*-
2 """RL CW2.ipynb
3
4 Automatically generated by Colaboratory.
5
6 Original file is located at
7     https://colab.research.google.com/drive/1
8     Rs6DY0CHGwQjUEg5HmRL47z-khB5ob7a
9
10 # Initialisation
11 """
12
13 # This is the coursework 2 for the Reinforcement Learning course
14 # 2021 taught at Imperial College London (https://www.imperial.ac.
15 # uk/computing/current-students/courses/70028/)
16 # The code is based on the OpenAI Gym original (https://pytorch.org
17 # /tutorials/intermediate/reinforcement_q_learning.html) and
18 # modified by Filippo Valdetaro and Prof. Aldo Faisal for the
19 # purposes of the course.
20 # There may be differences to the reference implementation in
21 # OpenAI gym and other solutions floating on the internet, but
22 # this is the definitive implementation for the course.
23
24 # Installing in Google Colab the libraries used for the coursework
25 # You do NOT need to understand it to work on this coursework
26
27 # WARNING: if you don't use this Notebook in Google Colab, this
28 # block might print some warnings (do not mind them)
29
30 !pip install gym pyvirtualdisplay > /dev/null 2>&1
31 !apt-get install -y xvfb python-opengl ffmpeg > /dev/null 2>&1
32 !pip install colabgymrender==1.0.2
33 !wget http://www.atari-roms.com/roms/Roms.rar
34 !mkdir /content/ROM/
35 !unrar e /content/Roms.rar /content/ROM/
36 !python -m atari_py.import_roms /content/ROM/
37
38 from IPython.display import clear_output
39 clear_output()
40
41 # Importing the libraries
42
43 import gym
44 from gym.wrappers.monitoring.video_recorder import VideoRecorder
45     #records videos of episodes
46 import numpy as np
47 import matplotlib.pyplot as plt # Graphical library
48 import seaborn as sns
49 from sklearn.linear_model import LinearRegression
50
51 import torch
```



```

42 import torch.optim as optim
43 import torch.nn as nn
44 import torch.nn.functional as F
45 device = torch.device("cuda" if torch.cuda.is_available() else "cpu"
    ") # Configuring Pytorch
46
47 from collections import namedtuple, deque
48 from itertools import count
49 import math
50 import random
51
52 # WARNING: if you don't use this Notebook in Google Colab, comment
    out these two imports
53 from colabgymrender.recorder import Recorder # Allow to record
    videos in Google Colab
54 Recorder(gym.make("CartPole-v1"), './video') # Defining the video
    recorder
55 clear_output()
56
57 # Test cell: check ai gym environment + recording working as
    intended
58
59 env = gym.make("CartPole-v1")
60 file_path = 'video/video.mp4'
61 recorder = VideoRecorder(env, file_path)
62
63 observation = env.reset()
64
65 terminal = False
66 while not terminal:
67     recorder.capture_frame()
68     action = int(observation[2]>0)
69     observation, reward, terminal, info = env.step(action)
70     # Observation is position, velocity, angle, angular velocity
71
72 recorder.close()
73 env.close()
74
75 """# 1. Train the DQN model
76
77 """
78
79 Transition = namedtuple('Transition',
    ('state', 'action', 'next_state', 'reward')
80 ) # 'state' and 'next_state' should be k-dimensional
81
82
83 class ReplayBuffer(object):
84
85     def __init__(self, capacity):
86         self.memory = deque([],maxlen=capacity)
87
88     def push(self, *args):

```

```

89         """Save a transition"""
90         self.memory.append(Transition(*args))
91
92     def sample(self, batch_size):
93         return random.sample(self.memory, batch_size)
94
95     def __len__(self):
96         return len(self.memory)
97
98     class DQN(nn.Module):
99
100     def __init__(self, k, inputs, outputs, num_hidden, hidden_size)
101     :
102         super(DQN, self).__init__()
103         self.input_layer = nn.Linear(k*inputs, hidden_size) # The
104         input size should be k*state_dim
105         self.hidden_layers = nn.ModuleList([nn.Linear(hidden_size,
106         hidden_size) for _ in range(num_hidden-1)])
107         self.output_layer = nn.Linear(hidden_size, outputs)
108
109     def forward(self, x):
110         x.to(device)
111
112         x = F.relu(self.input_layer(x))
113         for layer in self.hidden_layers:
114             x = F.relu(layer(x)) # Apply the ReLU activation
115         function in all hidden layers
116
117         return self.output_layer(x)
118
119     def optimize_model(policy_net, target_net, optimizer, memory):
120
121     if len(memory) < BATCH_SIZE:
122         return
123     transitions = memory.sample(BATCH_SIZE)
124     # Transpose the batch (see https://stackoverflow.com/a/19343/3343043 for
125     # detailed explanation). This converts batch-array of
126     Transitions
127     # to Transition of batch-arrays.
128     batch = Transition(*zip(*transitions))
129
130     # Compute a mask of non-final states and concatenate the batch
131     elements
132     # (a final state would've been the one after which simulation
133     ended)
134     non_final_mask = torch.tensor(tuple(map(lambda s: s is not None
135     ,
136                                     batch.next_state))),
137     device=device, dtype=torch.bool)

```

```

131     # Can safely omit the condition below to check that not all
132     # states in the
133     # sampled batch are terminal whenever the batch size is
134     # reasonable and
135     # there is virtually no chance that all states in the sampled
136     # batch are
137     # terminal
138     if sum(non_final_mask) > 0:
139         non_final_next_states = torch.cat([s for s in batch.
140 next_state
141                                     if s is not
142 None])
143     else:
144         non_final_next_states = torch.empty(0, 4).to(device)
145
146     state_batch = torch.cat(batch.state)
147     action_batch = torch.cat(batch.action)
148     reward_batch = torch.cat(batch.reward)
149
150     # Compute Q(s_t, a) - the model computes Q(s_t), then we select
151     # the
152     # columns of actions taken. These are the actions which would've
153     # been taken
154     # for each batch state according to policy_net
155     state_action_values = policy_net(state_batch).gather(1,
156 action_batch)
157
158     # Compute V(s_{t+1}) for all next states.
159     # This is merged based on the mask, such that we'll have either
160     # the expected
161     # state value or 0 in case the state was final.
162
163     next_state_values = torch.zeros(BATCH_SIZE, device=device).to(
164 device) # Added to device
165
166     with torch.no_grad():
167         # Once again can omit the condition if batch size is large
168         # enough
169         if sum(non_final_mask) > 0:
170             # Expected values of actions for non_final_next_states
171             # are computed based on the target_net
172             next_state_values[non_final_mask] = target_net(
173 non_final_next_states).max(1)[0].detach()
174         else:
175             next_state_values = torch.zeros_like(next_state_values)
176             .to(device) # Added to device
177
178     # Compute the expected Q values
179     expected_state_action_values = (next_state_values * GAMMA) +
180 reward_batch
181
182     # Compute Huber loss
183     criterion = nn.SmoothL1Loss()

```

```

169     loss = criterion(state_action_values,
170                       expected_state_action_values.unsqueeze(1))
171
172     # Optimize the model
173     optimizer.zero_grad()
174     loss.backward()
175
176     # Limit magnitude of gradient for update step
177     for param in policy_net.parameters():
178         param.grad.data.clamp_(-1, 1)
179
180     optimizer.step()
181
182 NUM_EPISODES = 200
183 BATCH_SIZE = 128
184 GAMMA = 0.999 # The discount rate
185 TARGET_UPDATE = 20
186
187 num_hidden_layers = 2
188 size_hidden_layers = 50
189 lr = 0.01
190 # epsilon = 0.05
191 # buffer_size = 10000
192 # k = 4
193
194 def set_up_parameters(buffer_size, k):
195
196     # Get number of states and actions from gym action space
197     env = gym.make("CartPole-v1")
198     env = gym.wrappers.FrameStack(env, k) # Stack k frames together
199     env.reset()
200
201     state_dim = len(env.state) # x, x_dot, theta, theta_dot
202     n_actions = env.action_space.n # left, right
203     env.close()
204
205     policy_net = DQN(k, state_dim, n_actions, num_hidden_layers,
206                      size_hidden_layers).to(device)
207     target_net = DQN(k, state_dim, n_actions, num_hidden_layers,
208                      size_hidden_layers).to(device) # Initiate the target network
209     # print(policy_net.state)
210     target_net.load_state_dict(policy_net.state_dict())
211     target_net.eval()
212
213     optimizer = optim.RMSprop(policy_net.parameters(), lr = lr) # Use
214                       the Root Mean Squared Propagation
215     memory = ReplayBuffer(buffer_size) # Set the size of the replay
216                       buffer
217
218     return policy_net, target_net, optimizer, memory
219
220 def select_action(policy_net, state, current_eps):
221     sample = random.random()

```

```

217     eps_threshold = current_eps
218
219     if sample > eps_threshold:
220         with torch.no_grad():
221             # t.max(1) will return largest column value of each row
222             .
223             # second column on max result is index of where max
element was
224             # found, so we pick action with the larger expected
reward.
225             return policy_net(state).max(1)[1].view(1, 1)
226         else:
227             return torch.tensor([[random.randrange(2)]], device=device,
dtype=torch.long)
228
229 steps_done = 0
230
231 def select_action_decaying_eplison(policy_net, state, EPS_START,
EPS_END, EPS_DECAY):
232     global steps_done
233     sample = random.random()
234     eps_threshold = EPS_END + (EPS_START - EPS_END) * \
235         math.exp(-1. * steps_done / EPS_DECAY)
236     steps_done += 1
237
238     if sample > eps_threshold:
239         with torch.no_grad():
240             # t.max(1) will return largest column value of each row
241             .
242             # second column on max result is index of where max
element was
243             # found, so we pick action with the larger expected
reward.
244             return policy_net(state).max(1)[1].view(1, 1)
245         else:
246             return torch.tensor([[random.randrange(2)]], device=device,
dtype=torch.long)
247
248 def learning(epsilon, buffer_size = 10000, k = 4):
249     env = gym.make("CartPole-v1")
250     env = gym.wrappers.FrameStack(env, k) # Stack k frames together
env.reset()
251
252     policy_net, target_net, optimizer, memory = set_up_parameters(
buffer_size, k)
253
254     list_of_returns = []
255
256     for i_episode in range(NUM_EPISODES):
257         if i_episode % 20 == 0:
258             print("episode ", i_episode, "/", NUM_EPISODES)
259
260         # Initialize the environment and state

```

```

260     env.reset()
261     state = torch.tensor(env.frames).float().flatten().
unsqueeze(0).to(device)
262     # print(state.shape)
263
264     total_return_for_the_episode = 0
265
266     for t in count():
267         # Select and perform an action
268         action = select_action(policy_net, state, epsilon)
269         _, reward, done, _ = env.step(action.item())
270         total_return_for_the_episode += reward # Calculate the
sum of undiscounted rewards
271         reward = torch.tensor([reward], device=device)
272
273         # Observe new state
274         if not done:
275             next_state = torch.tensor(env.frames).float().
flatten().unsqueeze(0).to(device)
276         else:
277             next_state = None
278
279         # Store the transition in memory
280         memory.push(state, action, next_state, reward)
281
282         # Move to the next state
283         state = next_state
284
285         # Perform one step of the optimization (on the policy
network)
286         # Calculate the Q-value this time using the policy_net
287         optimize_model(policy_net, target_net, optimizer,
memory)
288         if done:
289             break
290
291         # Update the target network, copying all weights and
biases in DQN
292         if i_episode % TARGET_UPDATE == 0:
293             target_net.load_state_dict(policy_net.state_dict())
294
295         list_of_returns.append(total_return_for_the_episode)
296
297     print('Complete')
298
299     env.close()
300
301     return(list_of_returns)
302
303 def learning_decaying_epsilon(EPS_START = 0.9, EPS_END = 0.05,
EPS_DECAY = 50, buffer_size = 10000, k = 4):
304     env = gym.make("CartPole-v1")
305     env = gym.wrappers.FrameStack(env, k) # Stack k frames together

```

```

306     env.reset()
307
308     policy_net, target_net, optimizer, memory = set_up_parameters(
309         buffer_size, k)
310
311     list_of_returns = []
312
313     for i_episode in range(NUM_EPISODES):
314         if i_episode % 20 == 0:
315             print("episode ", i_episode, "/", NUM_EPISODES)
316
317             # Initialize the environment and state
318             env.reset()
319             state = torch.tensor(env.frames).float().flatten().
320                 unsqueeze(0).to(device)
321
322             total_return_for_the_episode = 0
323
324             for t in count():
325                 # Select and perform an action
326                 action = select_action_decaying_eplison(policy_net,
327                     state, EPS_START, EPS_END, EPS_DECAY)
328                 _, reward, done, _ = env.step(action.item())
329                 total_return_for_the_episode += reward # Calculate the
330                 sum of undiscounted rewards
331                 reward = torch.tensor([reward], device=device)
332
333                 # Observe new state
334                 if not done:
335                     next_state = torch.tensor(env.frames).float().
336                         flatten().unsqueeze(0).to(device)
337                 else:
338                     next_state = None
339
340                 # Store the transition in memory
341                 memory.push(state, action, next_state, reward)
342
343                 # Move to the next state
344                 state = next_state
345
346                 # Perform one step of the optimization (on the policy
347                 network)
348                 optimize_model(policy_net, target_net, optimizer,
349                     memory)
350                 if done:
351                     break
352
353                 # Update the target network, copying all weights and
354                 biases in DQN
355                 if i_episode % TARGET_UPDATE == 0:
356                     target_net.load_state_dict(policy_net.state_dict())

```

```

351         list_of_returns.append(total_return_for_the_episode)
352
353     print('Complete')
354
355     env.close()
356
357     return(list_of_returns)
358
359     ## run an episode with trained agent and record video
360     ## remember to change file_path name if you do not wish to
361     ## overwrite an existing video
362
363     k = 4
364     env = gym.make("CartPole-v1")
365     env = gym.wrappers.FrameStack(env, k) # Stack k frames together
366
367     file_path = 'video/video.mp4'
368     recorder = VideoRecorder(env, file_path)
369
370     observation = env.reset()
371     done = False
372
373     state = torch.tensor(env.frames).float().flatten().unsqueeze(0).to(
374         device)
375     state_dim = len(env.state) #x, x_dot, theta, theta_dot
376     n_actions = env.action_space.n
377
378     policy_net = DQN(k, state_dim, n_actions, num_hidden_layers,
379         size_hidden_layers).to(device)
380
381     duration = 0
382
383     while not done:
384         recorder.capture_frame()
385
386         # Select and perform an action
387         action = select_action(policy_net, state, current_eps = 0.05)
388         observation, reward, done, _ = env.step(action.item())
389         duration += 1
390         reward = torch.tensor([reward], device=device)
391         # Observe new state
392         state = torch.tensor(env.frames).float().flatten().unsqueeze(0)
393         .to(device)
394
395     recorder.close()
396     env.close()
397     print("Episode duration: ", duration)
398
399     # Plot the learning curve for DQN
400     sns.set()
401
402     multiple_lists = []

```



```

400
401 # Record the list of total returns of ten repetitions
402 for i in range(10):
403     list_of_returns = learning(epsilon = 0.05)
404     multiple_lists.append(list_of_returns)
405
406 arrays = [np.array(x) for x in multiple_lists]
407
408 # Compute the mean of returns over 10 repetitions
409 mean_return = np.array([np.mean(k) for k in zip(*arrays)])
410 # Compute the standard deviation of returns
411 std_return = np.array([np.std(k) for k in zip(*arrays)])
412
413 plt.figure(figsize=(12,6))
414 plt.plot(mean_return, label='Mean')
415 plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return,
416     std_return), np.add(mean_return, std_return), color='b', alpha
417     =.3, label='Mean +/- std')
418
419 reg = LinearRegression().fit(np.arange(len(mean_return)).reshape
420     (-1, 1), np.array(mean_return).reshape(-1, 1))
421 y_pred = reg.predict(np.arange(len(mean_return)).reshape(-1, 1))
422 plt.plot(y_pred)
423
424 final_mean = mean_return[-1]
425 print("90% of final performance of average total return", 0.9*
426     final_mean)
427
428 for i, value in enumerate(mean_return):
429     if value >= 0.9*final_mean:
430         break
431
432 plt.plot(i, 0.9*final_mean, '*', color = 'red', markersize = 10)
433 print("Episode number at which the DQN achives 90% final
434     performance:", i)
435
436 plt.ylabel('Averaged Total Return')
437 plt.xlabel('Number of episodes')
438 plt.title('The learning curve of the DQN model')
439 plt.legend(loc="upper left")
440
441 plt.show()
442
443 """# 2. Hyperparameter Tuning
444
445 ## 2.1 Epsilon
446 """
447
448 # Plot the learning curve for DQN with constant and variable
449     epsilons
450 sns.set()
451
452 multiple_lists_small = []

```

```

447 multiple_lists_medium = []
448 multiple_lists_large = []
449 multiple_lists_var1 = []
450 multiple_lists_var2 = []
451
452 # Record the list of total returns of ten repetitions
453 for i in range(10):
454     multiple_lists_small.append(learning(epsilon = 0.05))
455     multiple_lists_medium.append(learning(epsilon = 0.2))
456     multiple_lists_large.append(learning(epsilon = 0.7))
457     multiple_lists_var1.append(learning_decaying_epsilon(0.9, 0.05,
458     50, 10000, 4)) # EPS_START = 0.9
459     multiple_lists_var2.append(learning_decaying_epsilon(0.5, 0.05,
460     50, 10000, 4)) # EPS_START = 0.5
461
462 arrays_small = [np.array(x) for x in multiple_lists_small]
463 arrays_medium = [np.array(x) for x in multiple_lists_medium]
464 arrays_large = [np.array(x) for x in multiple_lists_large]
465 arrays_var1 = [np.array(x) for x in multiple_lists_var1]
466 arrays_var2 = [np.array(x) for x in multiple_lists_var2]
467
468 # Compute the mean and sd of returns over 10 repetitions
469 mean_return_small = np.array([np.mean(k) for k in zip(*arrays_small
470     )])
471 std_return_small = np.array([np.std(k) for k in zip(*arrays_small
472     )])
473
474 mean_return_medium = np.array([np.mean(k) for k in zip(*
475     arrays_medium)])
476 std_return_medium = np.array([np.std(k) for k in zip(*arrays_medium
477     )])
478
479 mean_return_large = np.array([np.mean(k) for k in zip(*arrays_large
480     )])
481 std_return_large = np.array([np.std(k) for k in zip(*arrays_large
482     )])
483
484 mean_return_var1 = np.array([np.mean(k) for k in zip(*arrays_var1
485     )])
486 std_return_var1 = np.array([np.std(k) for k in zip(*arrays_var1)])
487
488 mean_return_var2 = np.array([np.mean(k) for k in zip(*arrays_var2
489     )])
490 std_return_var2 = np.array([np.std(k) for k in zip(*arrays_var2)])
491
492 plt.figure(figsize=(12,6))
493
494 plt.plot(mean_return_small, label='epsilon = 0.05')
495 plt.plot(mean_return_medium, label='epsilon = 0.2')
496 plt.plot(mean_return_large, label='epsilon = 0.7')
497 plt.plot(mean_return_var1, label='Initial epsilon = 0.9, decaying
498     rate = 50')

```

```

488 plt.plot(mean_return_var2, label='Initial epsilon = 0.5, decaying
    rate = 50')
489
490 plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_small
    , std_return_small), np.add(mean_return_small, std_return_small)
    , alpha=.1)
491 plt.fill_between(range(NUM_EPISODES), np.subtract(
    mean_return_medium, std_return_medium), np.add(
    mean_return_medium, std_return_medium), alpha=.1)
492 plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_large
    , std_return_large), np.add(mean_return_large, std_return_large)
    , alpha=.1)
493 plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_var1,
    std_return_var1), np.add(mean_return_var1, std_return_var1),
    alpha=.1)
494 plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_var2,
    std_return_var2), np.add(mean_return_var2, std_return_var2),
    alpha=.1)
495
496 plt.ylabel('Averaged Total Return')
497 plt.xlabel('Number of episodes')
498 plt.title('The learning curve of the DQN model with varying eplison
    ')
499 plt.legend(loc="upper left")
500 plt.show()
501
502 # Plot the graph for showing values of constant/variable eplisons
503 plt.figure(figsize=(12,6))
504
505 plt.plot([0.05]*200, label='epsilon = 0.05')
506 plt.plot([0.2]*200, label='epsilon = 0.2')
507 plt.plot([0.7]*200, label='epsilon = 0.7')
508 y_var1 = []
509 for i in range(200):
510     new_y = 0.05 + (0.9 - 0.05) * math.exp(-1. * i / 50)
511     y_var1.append(new_y)
512
513 y_var2 = []
514 for i in range(200):
515     new_y = 0.05 + (0.5 - 0.05) * math.exp(-1. * i / 50)
516     y_var2.append(new_y)
517
518 plt.plot(range(200), y_var1, label='Initial epsilon = 0.9, decaying
    rate = 50')
519 plt.plot(range(200), y_var2, label='Initial epsilon = 0.5, decaying
    rate = 50')
520
521 plt.ylabel('Eplison')
522 plt.xlabel('Number of episodes')
523 plt.title('The learning curve of the DQN model with varying k')
524
525 plt.legend(loc="upper right", prop={'size': 9.5})
526 plt.show()

```

```

527
528 """## 2.2 Size of Replay Buffer"""
529
530 # Plot the learning curve for DQN with different buffer sizes
531 sns.set()
532
533 multiple_lists_625 = []
534 multiple_lists_1250 = []
535 multiple_lists_2500 = []
536 multiple_lists_5000 = []
537 multiple_lists_10000 = []
538 multiple_lists_20000 = []
539 multiple_lists_40000 = []
540
541 # Record the list of total returns of ten repetitions
542 for i in range(10):
543     multiple_lists_625.append(learning(epsilon = 0.05, buffer_size
544     = 625))
545     multiple_lists_1250.append(learning(epsilon = 0.05, buffer_size
546     = 1250))
547     multiple_lists_2500.append(learning(epsilon = 0.05, buffer_size
548     = 2500))
549     multiple_lists_5000.append(learning(epsilon = 0.05, buffer_size
550     = 5000))
551     multiple_lists_10000.append(learning(epsilon = 0.05,
552     buffer_size = 10000))
553     multiple_lists_20000.append(learning(epsilon = 0.05,
554     buffer_size = 20000))
555     multiple_lists_40000.append(learning(epsilon = 0.05,
556     buffer_size = 40000))
557
558
559
560 arrays_625 = [np.array(x) for x in multiple_lists_625]
561 arrays_1250 = [np.array(x) for x in multiple_lists_1250]
562 arrays_2500 = [np.array(x) for x in multiple_lists_2500]
563 arrays_5000 = [np.array(x) for x in multiple_lists_5000]
564 arrays_10000 = [np.array(x) for x in multiple_lists_10000]
565 arrays_20000 = [np.array(x) for x in multiple_lists_20000]
566 arrays_40000 = [np.array(x) for x in multiple_lists_40000]
567
568
569 # Compute the mean and sd of returns over 10 repetitions
570 mean_return_625 = np.array([np.mean(k) for k in zip(*arrays_625)])
571 std_return_625 = np.array([np.std(k) for k in zip(*arrays_625)])
572
573
574 mean_return_1250 = np.array([np.mean(k) for k in zip(*arrays_1250)
575 ])
576 std_return_1250 = np.array([np.std(k) for k in zip(*arrays_1250)])
577
578
579 mean_return_2500 = np.array([np.mean(k) for k in zip(*arrays_2500)
580 ])
581 std_return_2500 = np.array([np.std(k) for k in zip(*arrays_2500)])
582
583

```

```

570 mean_return_5000 = np.array([np.mean(k) for k in zip(*arrays_5000)
    ])
571 std_return_5000 = np.array([np.std(k) for k in zip(*arrays_5000)])
572
573 mean_return_10000 = np.array([np.mean(k) for k in zip(*arrays_10000)
    ])
574 std_return_10000 = np.array([np.std(k) for k in zip(*arrays_10000)
    ])
575
576 mean_return_20000 = np.array([np.mean(k) for k in zip(*arrays_20000)
    ])
577 std_return_20000 = np.array([np.std(k) for k in zip(*arrays_20000)
    ])
578
579 mean_return_40000 = np.array([np.mean(k) for k in zip(*arrays_40000)
    ])
580 std_return_40000 = np.array([np.std(k) for k in zip(*arrays_40000)
    ])
581
582
583 plt.figure(figsize=(12,6))
584
585 plt.plot(mean_return_625, label='buffer_size = 625')
586 plt.plot(mean_return_1250, label='buffer_size = 1250')
587 plt.plot(mean_return_2500, label='buffer_size = 2500')
588 plt.plot(mean_return_5000, label='buffer_size = 5000')
589 plt.plot(mean_return_10000, label='buffer_size = 10000')
590 plt.plot(mean_return_20000, label='buffer_size = 20000')
591 plt.plot(mean_return_40000, label='buffer_size = 40000')
592
593 plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_625,
    std_return_625), np.add(mean_return_625, std_return_625), alpha
    =.1)
594 plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_1250,
    std_return_1250), np.add(mean_return_1250, std_return_1250),
    alpha=.1)
595 plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_2500,
    std_return_2500), np.add(mean_return_2500, std_return_2500),
    alpha=.1)
596 plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_5000,
    std_return_5000), np.add(mean_return_5000, std_return_5000),
    alpha=.1)
597 plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_10000
    , std_return_10000), np.add(mean_return_10000, std_return_10000)
    , alpha=.1)
598 plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_20000
    , std_return_20000), np.add(mean_return_20000, std_return_20000)
    , alpha=.1)
599 plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_40000
    , std_return_40000), np.add(mean_return_40000, std_return_40000)
    , alpha=.1)
600
601 plt.ylabel('Averaged Total Return')

```

```

602 plt.xlabel('Number of episodes')
603 plt.title('The learning curve of the DQN model with varying replay
        buffer size')
604 plt.legend(loc="upper left")
605 plt.show()
606
607 mean_ep = [mean_return_625, mean_return_1250, mean_return_2500,
        mean_return_5000, mean_return_10000, mean_return_20000,
        mean_return_40000]
608 std_ep = [std_return_625, std_return_1250, std_return_2500,
        std_return_5000, std_return_10000, std_return_20000,
        std_return_40000]
609 mean_and_std_ep = list(zip(mean_ep, std_ep))
610
611 std_list = []
612 for mean_and_std_return in mean_and_std_ep:
613     mean_return, std_return = mean_and_std_return
614     two_thirds_return = np.max(mean_return) * 2/3
615
616     for j, reward in enumerate(mean_return): # j is the episode at
        which we measure the std
617         if reward >= two_thirds_return:
618             break
619
620     std = std_return[j]
621     std_list.append(std)
622
623 print(std_list)
624
625 # Plot the sd vs buffer size
626 plt.figure(figsize=(12,6))
627
628 buffer_size = [625, 1250, 2500, 5000, 10000, 20000, 40000]
629 plt.scatter(buffer_size, std_list)
630 plt.plot(buffer_size, std_list)
631 plt.xlabel("Size of Replay Buffer (on log scale)")
632 plt.ylabel("Standard deviation of the learning curve")
633 plt.xscale('log')
634 plt.title('The variability in return (at 67% of max) with replay
        buffer size')
635
636 for i, size in enumerate(buffer_size):
637     plt.annotate(size, (buffer_size[i], std_list[i]))
638
639 plt.show()
640
641 """## 2.3 Value of k"""
642
643 # Plot the learning curve for DQN with different k
644 sns.set()
645
646 multiple_lists_1 = []
647 multiple_lists_2 = []

```

```

648 multiple_lists_4 = []
649 multiple_lists_8 = []
650 multiple_lists_16 = []
651
652 # Record the list of total returns of ten repetitions
653 for i in range(10):
654     multiple_lists_1.append(learning(epsilon = 0.05, k = 1))
655     multiple_lists_2.append(learning(epsilon = 0.05, k = 2))
656     multiple_lists_4.append(learning(epsilon = 0.05, k = 4))
657     multiple_lists_8.append(learning(epsilon = 0.05, k = 8))
658     multiple_lists_16.append(learning(epsilon = 0.05, k = 16))
659
660 arrays_1 = [np.array(x) for x in multiple_lists_1]
661 arrays_2 = [np.array(x) for x in multiple_lists_2]
662 arrays_4 = [np.array(x) for x in multiple_lists_4]
663 arrays_8 = [np.array(x) for x in multiple_lists_8]
664 arrays_16 = [np.array(x) for x in multiple_lists_16]
665
666 # Compute the mean and sd of returns over 10 repetitions
667 mean_return_1 = np.array([np.mean(k) for k in zip(*arrays_1)])
668 std_return_1 = np.array([np.std(k) for k in zip(*arrays_1)])
669
670 mean_return_2 = np.array([np.mean(k) for k in zip(*arrays_2)])
671 std_return_2 = np.array([np.std(k) for k in zip(*arrays_2)])
672
673 mean_return_4 = np.array([np.mean(k) for k in zip(*arrays_4)])
674 std_return_4 = np.array([np.std(k) for k in zip(*arrays_4)])
675
676 mean_return_8 = np.array([np.mean(k) for k in zip(*arrays_8)])
677 std_return_8 = np.array([np.std(k) for k in zip(*arrays_8)])
678
679 mean_return_16 = np.array([np.mean(k) for k in zip(*arrays_16)])
680 std_return_16 = np.array([np.std(k) for k in zip(*arrays_16)])
681
682 plt.figure(figsize=(12,6))
683
684 plt.plot(mean_return_1, label='k = 1')
685 plt.plot(mean_return_2, label='k = 2')
686 plt.plot(mean_return_4, label='k = 4')
687 plt.plot(mean_return_8, label='k = 8')
688 plt.plot(mean_return_16, label='k = 16')
689
690 plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_1,
691     std_return_1), np.add(mean_return_1, std_return_1), alpha=.1)
691 plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_2,
692     std_return_2), np.add(mean_return_2, std_return_2), alpha=.1)
692 plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_4,
693     std_return_4), np.add(mean_return_4, std_return_4), alpha=.1)
693 plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_8,
694     std_return_8), np.add(mean_return_8, std_return_8), alpha=.1)
694 plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_16,
695     std_return_16), np.add(mean_return_16, std_return_16), alpha=.1)
695

```

```

696 plt.ylabel('Averaged Total Return')
697 plt.xlabel('Number of episodes')
698 plt.title('The learning curve of the DQN model with varying k')
699 plt.legend(loc="upper left")
700 plt.show()
701
702 """# 3. Ablation/Augmentation experiments
703
704 ## 3.1 Ablate the target network
705 """
706
707 def learning_ablate_target(epsilon, buffer_size = 10000, k = 4):
708     env = gym.make("CartPole-v1")
709     env = gym.wrappers.FrameStack(env, k) # Stack k frames together
710     env.reset()
711
712     policy_net, _, optimizer, memory = set_up_parameters(
713         buffer_size, k)
714
715     list_of_returns = []
716
717     for i_episode in range(NUM_EPISODES):
718         if i_episode % 20 == 0:
719             print("episode ", i_episode, "/", NUM_EPISODES)
720
721             # Initialize the environment and state
722             env.reset()
723             state = torch.tensor(env.frames).float().flatten().
724             unsqueeze(0).to(device)
725             # print(state.shape)
726
727             total_return_for_the_episode = 0
728
729             for t in count():
730                 # Select and perform an action
731                 action = select_action(policy_net, state, epsilon)
732                 _, reward, done, _ = env.step(action.item())
733                 total_return_for_the_episode += reward # Calculate the
734                 sum of undiscounted rewards
735                 reward = torch.tensor([reward], device=device)
736
737                 # Observe new state
738                 if not done:
739                     next_state = torch.tensor(env.frames).float().
740                     flatten().unsqueeze(0).to(device)
741                 else:
742                     next_state = None
743
744                 # Store the transition in memory
745                 memory.push(state, action, next_state, reward)
746
747                 # Move to the next state
748                 state = next_state

```



```

745
746         # Perform one step of the optimization (on the policy
network)
747         optimize_model(policy_net, policy_net, optimizer,
memory) # Calculate the Q-value using the policy_net
748         if done:
749             break
750
751         # Update the target network, copying all weights and
biases in DQN
752         # if i_episode % TARGET_UPDATE == 0:
753             # target_net.load_state_dict(policy_net.state_dict())
754
755         list_of_returns.append(total_return_for_the_episode)
756
757     print('Complete')
758
759     env.close()
760
761     return(list_of_returns)
762
763 """## 3.2 Ablate the replay buffer
764
765 """
766
767 def learning_ablate_buffer(epsilon, buffer_size = 1, k = 4): # Set
the buffer size to be 1
768     env = gym.make("CartPole-v1")
769     env = gym.wrappers.FrameStack(env, k) # Stack k frames together
770     env.reset()
771
772     policy_net, target_net, optimizer, memory = set_up_parameters(
buffer_size, k)
773
774     list_of_returns = []
775
776     for i_episode in range(NUM_EPISODES):
777         if i_episode % 20 == 0:
778             print("episode ", i_episode, "/", NUM_EPISODES)
779
780         # Initialize the environment and state
781         env.reset()
782         state = torch.tensor(env.frames).float().flatten().
unsqueeze(0).to(device)
783         # print(state.shape)
784
785         total_return_for_the_episode = 0
786
787         for t in count():
788             # Select and perform an action
789             action = select_action(policy_net, state, epsilon)
790             _, reward, done, _ = env.step(action.item())

```

```

791         total_return_for_the_episode += reward # Calculate the
sum of undiscounted rewards
792         reward = torch.tensor([reward], device=device)
793
794         # Observe new state
795         if not done:
796             next_state = torch.tensor(env.frames).float().
flatten().unsqueeze(0).to(device)
797         else:
798             next_state = None
799
800         # Store the transition in memory
801         memory.push(state, action, next_state, reward)
802
803         # Move to the next state
804         state = next_state
805
806         # Perform one step of the optimization (on the policy
network)
807         optimize_model(policy_net, target_net, optimizer,
memory)
808         if done:
809             break
810
811         # Update the target network, copying all weights and
biases in DQN
812         if i_episode % TARGET_UPDATE == 0:
813             target_net.load_state_dict(policy_net.state_dict())
814
815         list_of_returns.append(total_return_for_the_episode)
816
817         print('Complete')
818
819         env.close()
820
821         return(list_of_returns)
822
823 """## 3.3 Double DQN"""
824
825 def learning_ddqn(epsilon, buffer_size = 10000, k = 4):
826     env = gym.make("CartPole-v1")
827     env = gym.wrappers.FrameStack(env, k) # Stack k frames together
828     env.reset()
829
830     policy_net, target_net, optimizer, memory = set_up_parameters(
buffer_size, k)
831
832     list_of_returns = []
833
834     for i_episode in range(NUM_EPISODES):
835         if i_episode % 20 == 0:
836             print("episode ", i_episode, "/", NUM_EPISODES)
837

```

```

838     # Initialize the environment and state
839     env.reset()
840     state = torch.tensor(env.frames).float().flatten().
unsqueeze(0).to(device)
841     # print(state.shape)
842
843     total_return_for_the_episode = 0
844
845     for t in count():
846         # Select and perform an action
847         action = select_action(target_net, state, epsilon) #
Use the target net to select action
848         _, reward, done, _ = env.step(action.item())
849         total_return_for_the_episode += reward
850         reward = torch.tensor([reward], device=device)
851
852         # Observe new state
853         if not done:
854             next_state = torch.tensor(env.frames).float().
flatten().unsqueeze(0).to(device)
855         else:
856             next_state = None
857
858         # Store the transition in memory
859         memory.push(state, action, next_state, reward)
860
861         # Move to the next state
862         state = next_state
863
864         # Perform one step of the optimization (on the policy
network)
865         optimize_model(policy_net, target_net, optimizer,
memory)
866         if done:
867             break
868
869         # Update the target network, copying all weights and
biases in DQN
870         if i_episode % TARGET_UPDATE == 0:
871             target_net.load_state_dict(policy_net.state_dict())
872
873         list_of_returns.append(total_return_for_the_episode)
874
875     print('Complete')
876
877     env.close()
878
879     return(list_of_returns)
880
881 """## 3.4 Plot the learning curves"""
882
883 # Plot the learning curves for DQN without target network or replay
buffer

```

```

884 sns.set()
885
886 multiple_lists_no_target = []
887 multiple_lists_no_buffer = []
888
889 # Record the list of total returns of ten repetitions
890 for i in range(10):
891     multiple_lists_no_target.append(learning_ablate_target(epsilon
892     = 0.05))
893     multiple_lists_no_buffer.append(learning_ablate_buffer(epsilon
894     = 0.05))
895
896 arrays_no_target = [np.array(x) for x in multiple_lists_no_target]
897 arrays_no_buffer = [np.array(x) for x in multiple_lists_no_buffer]
898
899 # Compute the mean and sd of returns over 10 repetitions
900 mean_return_no_target = np.array([np.mean(k) for k in zip(*
901     arrays_no_target)])
902 std_return_no_target = np.array([np.std(k) for k in zip(*
903     arrays_no_target)])
904
905 mean_return_no_buffer = np.array([np.mean(k) for k in zip(*
906     arrays_no_buffer)])
907 std_return_no_buffer = np.array([np.std(k) for k in zip(*
908     arrays_no_buffer)])
909
910 plt.figure(figsize=(12,6))
911
912 plt.plot(mean_return_no_target, label='Ablate target network')
913 plt.plot(mean_return_no_buffer, label='Ablate replay buffer')
914
915 plt.fill_between(range(NUM_EPISODES), np.subtract(
916     mean_return_no_target, std_return_no_target), np.add(
917     mean_return_no_target, std_return_no_target), alpha=.1)
918 plt.fill_between(range(NUM_EPISODES), np.subtract(
919     mean_return_no_buffer, std_return_no_buffer), np.add(
920     mean_return_no_buffer, std_return_no_buffer), alpha=.1)
921
922 plt.ylabel('Averaged Total Return')
923 plt.xlabel('Number of episodes')
924 plt.title('The learning curves of variations of the DQN model')
925 plt.legend(loc="upper left")
926 plt.show()
927
928 # Plot the learning curves for variations of DQN
929 sns.set()
930
931 multiple_lists_dqn = []
932 multiple_lists_no_target = []
933 multiple_lists_no_buffer = []
934 multiple_lists_ddqn = []
935
936 # Record the list of total returns of ten repetitions

```

```

927 for i in range(10):
928     multiple_lists_dqn.append(learning(epsilon = 0.05))
929     multiple_lists_no_target.append(learning_ablate_target(epsilon
= 0.05))
930     multiple_lists_no_buffer.append(learning_ablate_buffer(epsilon
= 0.05))
931     multiple_lists_ddqn.append(learning_ddqn(epsilon = 0.05))
932
933 arrays_dqn = [np.array(x) for x in multiple_lists_dqn]
934 arrays_no_target = [np.array(x) for x in multiple_lists_no_target]
935 arrays_no_buffer = [np.array(x) for x in multiple_lists_no_buffer]
936 arrays_ddqn = [np.array(x) for x in multiple_lists_ddqn]
937
938 # Compute the mean and sd of returns over 10 repetitions
939 mean_return_dqn = np.array([np.mean(k) for k in zip(*arrays_dqn)])
940 std_return_dqn = np.array([np.std(k) for k in zip(*arrays_dqn)])
941
942 mean_return_no_target = np.array([np.mean(k) for k in zip(*
arrays_no_target)])
943 std_return_no_target = np.array([np.std(k) for k in zip(*
arrays_no_target)])
944
945 mean_return_no_buffer = np.array([np.mean(k) for k in zip(*
arrays_no_buffer)])
946 std_return_no_buffer = np.array([np.std(k) for k in zip(*
arrays_no_buffer)])
947
948 mean_return_ddqn = np.array([np.mean(k) for k in zip(*arrays_ddqn)
])
949 std_return_ddqn = np.array([np.std(k) for k in zip(*arrays_ddqn)])
950
951 plt.figure(figsize=(12,6))
952
953 plt.plot(mean_return_dqn, label='DQN')
954 plt.plot(mean_return_no_target, label='Ablate target network')
955 plt.plot(mean_return_no_buffer, label='Ablate replay buffer')
956 plt.plot(mean_return_ddqn, label='DDQN')
957
958 plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_dqn,
std_return_dqn), np.add(mean_return_dqn, std_return_dqn), alpha
=.1)
959 plt.fill_between(range(NUM_EPISODES), np.subtract(
mean_return_no_target, std_return_no_target), np.add(
mean_return_no_target, std_return_no_target), alpha=.1)
960 plt.fill_between(range(NUM_EPISODES), np.subtract(
mean_return_no_buffer, std_return_no_buffer), np.add(
mean_return_no_buffer, std_return_no_buffer), alpha=.1)
961 plt.fill_between(range(NUM_EPISODES), np.subtract(mean_return_ddqn,
std_return_ddqn), np.add(mean_return_ddqn, std_return_ddqn),
alpha=.1)
962
963 plt.ylabel('Averaged Total Return')
964 plt.xlabel('Number of episodes')

```

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