pa2-dqn-mountain-car

March 29, 2023

[ ]:
```
import math
import numpy as np
import random
import torch
import torch.nn as nn
import torch.nn.functional as F
from collections import namedtuple, deque
import torch.optim as optim
import datetime
import gym
from gym.wrappers.record_video import RecordVideo
import glob
import io
import base64
import matplotlib.pyplot as plt
from IPython.display import HTML
from pyvirtualdisplay import Display
import tensorflow as tf
from IPython import display as ipythondisplay
from PIL import Image
import tensorflow_probability as tfp
import warnings
warnings.filterwarnings("ignore")
```

[ ]:
```
env = gym.make('MountainCar-v0')
env.seed(0)

state_shape = env.observation_space.shape[0]
no_of_actions = env.action_space.n

print(state_shape)
print(no_of_actions)
print(env.action_space.sample())
print("----")


state = env.reset()
```

```
print(state)
print("----")

action = env.action_space.sample()

print(action)
print("----")

next_state, reward, done, info = env.step(action)

print(next_state)
print(reward)
print(done)
print(info)
print("----")
```

```
2
3
1
----
[-0.47260767  0.        ]
----
1
----
[-4.7298861e-01 -3.8094356e-04]
-1.0
False
{}
----
```

[ ]:
```
sweep_config = {
    'method': 'bayes'
}
```

[ ]:
```
metric = {
    'name' : 'num_episodes_tosolve_cartpole',
    'goal' : 'minimize'
}
sweep_config['metric'] = metric
```

[ ]:
```
parameters_dict = {
    'NUM_NEURONS_EACH_LAYER' : {
        'values': [64,32,16]
      },
    'NUM_LAYERS' :{
        'values': [2,4,8]
```

```
    },
    }
sweep_config['parameters'] = parameters_dict
```

[ ]:

```
import math

parameters_dict.update({
    'LR': {

        'distribution': 'uniform',
        'min': (0.0001),
        'max': (0.01)
      },
    'BATCH_SIZE': {

        'distribution': 'q_log_uniform_values',
        'q'   : 1,
        'min': (64),
        'max': (512),
      },
    'UPDATE_EVERY': {

        'distribution': 'q_log_uniform_values',
        'q'   : 1,
        'min': (10),
        'max': (200),
      },
    'GAMMA': {

        'distribution': 'uniform',
        'min': 0.8,
        'max': 1,
      },
    'MAX_TRUNCATION': {

        'distribution': 'uniform',
        'min': (0.5),
        'max': (2),
      },
    'EPS_DECAY': {

        'distribution': 'uniform',
        'min': 0.9,
        'max': 0.995,
      },
    'a_TUNE': {
```

```
        'distribution': 'uniform',
        'min': 0,
        'max': 1,
      }

    })
```

[ ]:
```
import pprint

pprint.pprint(sweep_config)
```

```
{'method': 'bayes',
 'metric': {'goal': 'minimize', 'name': 'num_episodes_tosolve_cartpole'},
 'parameters': {'BATCH_SIZE': {'distribution': 'q_log_uniform_values',
                               'max': 512,
                               'min': 64,
                               'q': 1},
                'EPS_DECAY': {'distribution': 'uniform',
                              'max': 0.995,
                              'min': 0.9},
                'GAMMA': {'distribution': 'uniform', 'max': 1, 'min': 0.8},
                'LR': {'distribution': 'uniform', 'max': 0.01, 'min': 0.0001},
                'MAX_TRUNCATION': {'distribution': 'uniform',
                                   'max': 2,
                                   'min': 0.5},
                'NUM_LAYERS': {'values': [2, 4, 8]},
                'NUM_NEURONS_EACH_LAYER': {'values': [64, 32, 16]},
                'UPDATE_EVERY': {'distribution': 'q_log_uniform_values',
                                 'max': 200,
                                 'min': 10,
                                 'q': 1},
                'a_TUNE': {'distribution': 'uniform', 'max': 1, 'min': 0}}}
```

[ ]:
```
sweep_id = wandb.sweep(sweep_config, project="Hyper parameter tuning Cartpole -␣
 ↪v1")
```

```
Failed to detect the name of this notebook, you can set it manually with the
WANDB_NOTEBOOK_NAME environment variable to enable code saving.
```

```
Create sweep with ID: 96xha17k
Sweep URL: https://wandb.ai/me19b190/Hyper%20parameter%20tuning%20Cartpole%20-%2
0v1/sweeps/96xha17k
```

[ ]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
```

```python
class QNetwork1(nn.Module): #take state s and outputs Q(s,a)

  def __init__(self,seed, state_size, action_size,␣
↪num_layers,neurons_each_layer): #num_layers = number of hidden␣
↪layers,neurons_each_layer = number of neurons in hidden layers
    super(QNetwork1,self).__init__()
    self.seed = torch.manual_seed(seed)
    self.input_size = state_size
    self.output_size = action_size
    self.num_layers = num_layers
    self.neurons_each_layer = neurons_each_layer
    self.linears = nn.ModuleList([nn.Linear(self.input_size, self.
↪neurons_each_layer[0])])
    self.linears.extend([nn.Linear(self.neurons_each_layer[i-1], self.
↪neurons_each_layer[i]) for i in range(1, self.num_layers)])
    self.linears.append(nn.Linear(self.neurons_each_layer[-1], self.
↪output_size))

  def forward(self,state):
    h=None
    for i in range(self.num_layers+1):
      if i == 0:
        h = F.relu(self.linears[0](state))
      elif i<(self.num_layers):
        h = F.relu(self.linears[i](h))
      else:
        return self.linears[self.num_layers](h)
```

[ ]:
```python
import random
import torch
import numpy as np
from collections import deque, namedtuple

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

class ReplayBuffer:
    """Fixed-size buffer to store experience tuples."""

    def __init__(self, action_size, buffer_size, batch_size, seed):
        """Initialize a ReplayBuffer object.

        Params
        ======
            action_size (int): dimension of each action
            buffer_size (int): maximum size of buffer
            batch_size (int): size of each training batch
            seed (int): random seed
```

```
        """
        self.action_size = action_size
        self.memory = deque(maxlen=buffer_size)
        self.batch_size = batch_size
        self.experience = namedtuple("Experience", field_names=["state",␣
↪"action", "reward", "next_state", "done"])
        self.seed = random.seed(seed)

    def add(self, state, action, reward, next_state, done): #added TD
        """Add a new experience to memory."""
        e = self.experience(state, action, reward, next_state, done)
        self.memory.append(e)

    def sample(self,batch_size=None,a=1.0):
        """Randomly sample a batch of experiences from memory."""
        experiences = random.sample(self.memory, k=self.batch_size)

        states = torch.from_numpy(np.vstack([e.state for e in experiences if e␣
↪is not None])).float().to(device)
        actions = torch.from_numpy(np.vstack([e.action for e in experiences if␣
↪e is not None])).long().to(device)
        rewards = torch.from_numpy(np.vstack([e.reward for e in experiences if␣
↪e is not None])).float().to(device)
        next_states = torch.from_numpy(np.vstack([e.next_state for e in␣
↪experiences if e is not None])).float().to(device)
        dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is␣
↪not None]).astype(np.uint8)).float().to(device)

        return (states, actions, rewards, next_states, dones)


    def __len__(self):
        """Return the current size of internal memory."""
        return len(self.memory)
```

[ ]:
```
def calculate_mean(array):
    lens = [len(i) for i in array]
    arr = np.ma.empty((np.max(lens),len(array)))
    arr.mask = True
    for idx, l in enumerate(array):
        arr[:len(l),idx] = l
    return arr.mean(axis = -1), arr.std(axis=-1)
```

[ ]:
```
''' Defining DQN Algorithm '''
state_shape = env.observation_space.shape[0]
action_shape = env.action_space.n
```

```
def dqn(agent,n_episodes=5000, max_t=500, eps_start=1.0, eps_end=0.01,␣
↪eps_decay=0.999,a = 0.7):

    scores = []
    ''' list containing scores from each episode '''
    steps = []
    scores_window_printing = deque(maxlen=10)
    ''' For printing in the graph '''
    scores_window= deque(maxlen=100)
    ''' last 100 scores for checking if the avg is more than 195 '''
    scores_exit_bad_paramas = deque(maxlen=200)
    eps = eps_start
    ''' initialize epsilon '''

    for i_episode in range(1, n_episodes+1):
        state = env.reset()
        score = 0
        step = 0
        for t in range(max_t):
            state_adj = (state-env.observation_space.low)*np.array([10,50])
            state_adj = np.round(state_adj,0).astype(int)
            action = agent.act(state_adj, eps)
            next_state, reward, done, _ = env.step(action)
            next_state_adj = (next_state-env.observation_space.low)*np.
↪array([10,50])
            next_state_adj = np.round(next_state_adj,0).astype(int)
            agent.step(state_adj, action, reward, next_state_adj, done,a = a)
            state = next_state
            score += reward
            step += 1
            if done:
                break
        steps.append(step)
        scores_window.append(score)
        scores_window_printing.append(score)
        scores_exit_bad_paramas.append(score)

        ''' save most recent score '''

        eps = max(eps_end, eps_decay*eps)
        ''' decrease epsilon '''

        print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.
↪mean(scores_window)), end="")
        if i_episode % 10 == 0:
            scores.append(np.mean(scores_window_printing))
```

```
        if i_episode % 100 == 0:
           print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.
↪mean(scores_window)))
        if np.mean(scores_window)>=-110.0:
           print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.
↪2f}'.format(i_episode-100, np.mean(scores_window)))
           break
    return [np.array(steps),np.array(scores),i_episode,False]

''' Trial run to check if algorithm runs and saves the data '''
BATCH_SIZE= 64
EPS_DECAY= 0.995
GAMMA= 0.99
LR= 5e-4
MAX_TRUNCATION= 1
NUM_LAYERS= 2
NUM_NEURONS_EACH_LAYER= 64
UPDATE_EVERY= 20
a_TUNE= 0

agent_l = None
steps_over_10_runs  = []
rewards_over_10_runs = []
episodes = []
min_len = np.inf
for run in range(1):
      print("Run: ",run+1,"␣
↪------------------------------------------------------------")
      env = gym.make('MountainCar-v0',max_episode_steps=500)
      seed = run
      env.seed(seed)
      state_shape = env.observation_space.shape[0]
      action_shape = env.action_space.n

      begin_time = datetime.datetime.now()
      neurons_each_layer_ = [NUM_NEURONS_EACH_LAYER]*NUM_LAYERS
      agent = TutorialAgent(state_size= state_shape, action_size=action_shape,␣
↪seed=seed,num_layers=NUM_LAYERS,neurons_each_layer=neurons_each_layer_␣
↪,lr=LR,gamma=GAMMA,update_every=UPDATE_EVERY,batch_size=BATCH_SIZE,max_tRUNCATION=MAX_TRUNC
      steps_eps_greedy,reward_eps_greedy, episodes_eps_greedy,agent_l =␣
↪dqn(agent,eps_decay = EPS_DECAY, a = a_TUNE)
      steps_over_10_runs.append(steps_eps_greedy)
      rewards_over_10_runs.append(reward_eps_greedy)
      episodes.append(episodes_eps_greedy)
      if len(reward_eps_greedy) < min_len:
        min_len = len(reward_eps_greedy)
      time_taken = datetime.datetime.now() - begin_time
```

```
        print(time_taken)

rewards = np.array(rewards_over_10_runs)
avg_episodes = int(np.array(episodes).mean())
steps = np.array(steps_over_10_runs)
```

```
Run:  1  ------------------------------------------------------------
Episode 100     Average Score: -500.00
Episode 200     Average Score: -500.00
Episode 300     Average Score: -492.10
Episode 400     Average Score: -403.26
Episode 500     Average Score: -251.92
Episode 600     Average Score: -169.75
Episode 700     Average Score: -147.15
Episode 800     Average Score: -142.20
Episode 900     Average Score: -142.17
Episode 1000    Average Score: -150.11
Episode 1100    Average Score: -138.30
Episode 1200    Average Score: -136.35
Episode 1300    Average Score: -133.42
Episode 1400    Average Score: -136.04
Episode 1500    Average Score: -134.69
Episode 1600    Average Score: -140.36
Episode 1700    Average Score: -140.18
Episode 1800    Average Score: -147.31
Episode 1900    Average Score: -147.55
Episode 2000    Average Score: -140.56
Episode 2100    Average Score: -141.59
Episode 2200    Average Score: -135.20
Episode 2300    Average Score: -144.09
Episode 2400    Average Score: -134.94
Episode 2500    Average Score: -132.15
Episode 2600    Average Score: -142.41
Episode 2700    Average Score: -146.01
Episode 2800    Average Score: -149.58
Episode 2900    Average Score: -148.96
Episode 3000    Average Score: -149.31
Episode 3100    Average Score: -149.58
Episode 3200    Average Score: -148.10
Episode 3300    Average Score: -154.22
Episode 3400    Average Score: -153.82
Episode 3500    Average Score: -154.79
Episode 3600    Average Score: -143.06
Episode 3700    Average Score: -139.03
Episode 3800    Average Score: -143.84
Episode 3900    Average Score: -153.15
```

```
Episode 4000    Average Score: -146.50
Episode 4100    Average Score: -153.11
Episode 4200    Average Score: -169.81
Episode 4300    Average Score: -159.53
Episode 4400    Average Score: -154.84
Episode 4500    Average Score: -157.75
Episode 4600    Average Score: -154.27
Episode 4700    Average Score: -152.30
Episode 4800    Average Score: -142.60
Episode 4900    Average Score: -143.62
Episode 5000    Average Score: -142.63
1:08:07.755923
```

Hyperparameters: 1. Number of neurons in neural network 64,128,256,512 2. Number of layers in neural network 2,4,8,10 3. learning rate: log_uniform('1e-6' to '1e-1') 4. buffer_Size = log_uniform(10-300) 5. upate_frequency = log_uniform(20,200) 6. truncation_Limit = log_uniform(100,1000,10000) 7. discount factor = uniform(0,1) 8. epsilon decay = uniform(0.9,0.995) 9. a = uniform(0.01,1) 10.b = similar to epsilon decay of epsilon Try 12 different configs

[ ]:
```
rewards
```

[ ]:
```
array([[-500. , -500. , -500. , -500. , -500. , -500. , -500. , -500. ,
        -500. , -500. , -500. , -500. , -500. , -500. , -500. , -500. ,
        -500. , -500. , -500. , -500. , -497.8, -500. , -500. , -467.1,
        -496.4, -500. , -500. , -488.8, -471.5, -499.4, -489.6, -402.4,
        -436.6, -425.8, -379.5, -454.3, -404.2, -384.3, -339.4, -316.5,
        -315.8, -274.6, -290.7, -257.5, -268.7, -273.7, -206. , -218. ,
        -213.4, -200.8, -179.7, -175.2, -160.4, -177.1, -166.7, -169.4,
        -155.4, -183.2, -171.8, -158.6, -155.9, -168.8, -149.3, -133.5,
        -155.8, -146.9, -129.5, -144.1, -140. , -147.7, -142.7, -137.8,
        -150. , -144.1, -124.7, -152.4, -153.4, -141.3, -149. , -126.6,
        -142. , -146.8, -137.1, -141.4, -124.1, -142.7, -153. , -143. ,
        -142.2, -149.4, -140. , -144.6, -147.4, -154.1, -166.1, -140.1,
        -150.2, -148.4, -154.4, -155.8, -135. , -141. , -147.2, -151.6,
        -143. , -119.1, -137.7, -129.4, -139. , -140. , -145.7, -139.3,
        -122.6, -138.5, -137.9, -150.4, -138.5, -131. , -133.2, -126.4,
        -126.4, -126.3, -147.4, -139.5, -130.6, -128.5, -135.1, -137.6,
        -135. , -127.8, -130.3, -133.1, -123.2, -155.2, -136.4, -141.6,
        -132. , -114.5, -149.7, -144.4, -120.8, -124.2, -125.7, -128.3,
        -150.3, -146.1, -160.7, -129.4, -136.2, -125.2, -146.1, -151.1,
        -143.1, -139.2, -131. , -139.8, -140.1, -149.1, -141.2, -122.9,
        -128.1, -169.3, -131.7, -155. , -131.4, -140.4, -123.9, -133.9,
        -142.8, -145.3, -167. , -158.7, -142.2, -152.2, -138.6, -146. ,
        -143.1, -146. , -129.7, -149.6, -123.7, -153.3, -164.3, -143.5,
        -146.6, -156.6, -139. , -144. , -147.9, -156.6, -128.1, -140.2,
        -141.9, -121.1, -158.1, -151.9, -127.4, -140.8, -149.3, -146.8,
        -138.8, -137.2, -150.2, -141.1, -140.3, -149. , -142.1, -150.8,
```

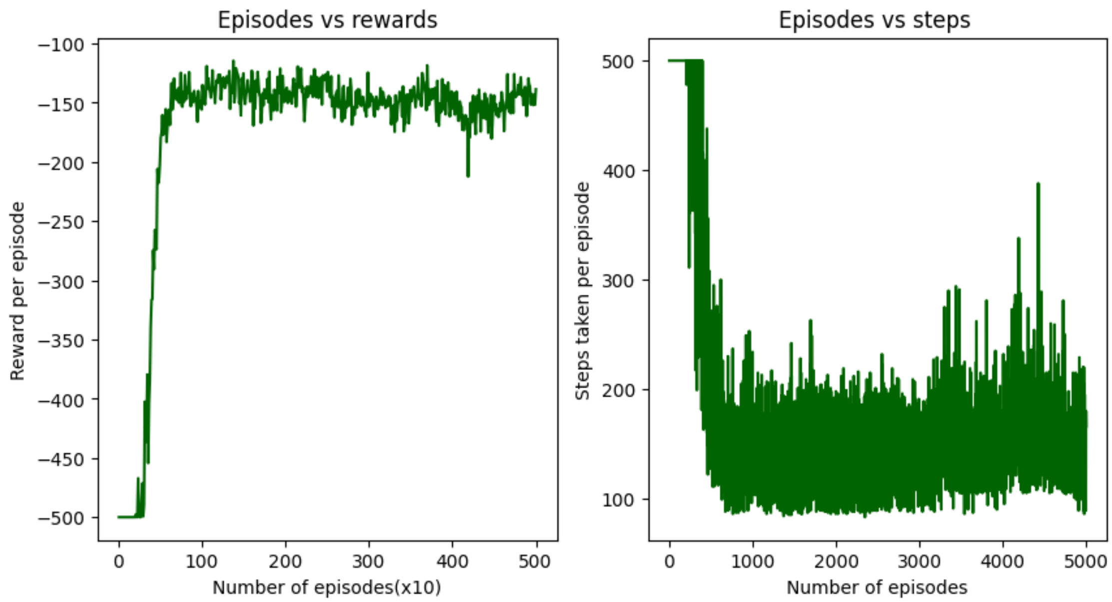
```
-136.8, -129.6, -144.6, -134.1, -153.3, -140.2, -119.4, -130.6,
-137.2, -130.7, -121. , -140.9, -137. , -148.7, -165.7, -152.1,
-139.7, -136.4, -145.1, -136.8, -137.3, -142.1, -127.7, -128.4,
-145.1, -136.1, -121.7, -136.2, -145.4, -128.3, -134.5, -146. ,
-130.9, -132.2, -128.4, -151.6, -126.9, -127.6, -130.8, -126.3,
-126.2, -140.6, -124. , -135. , -136.2, -145.4, -127.6, -153. ,
-145.6, -151.2, -162.5, -143.6, -143.3, -140. , -153.7, -126.9,
-145.9, -154.4, -156.3, -165.9, -137.9, -135.8, -163.2, -157.7,
-149.2, -153.4, -137.3, -146.8, -152.4, -129.7, -137.5, -168.6,
-153.5, -153.9, -160.3, -153.1, -140.2, -154.2, -138.7, -150.6,
-139. , -146.1, -152.7, -149.9, -145. , -149.3, -156.5, -156.7,
-161.7, -143.2, -124.5, -153.6, -152.5, -142.4, -148. , -144.7,
-148.4, -158.5, -161.7, -146.6, -144.8, -148.2, -155.6, -150.1,
-151.5, -137.6, -144.7, -151.9, -140.9, -144.6, -149.7, -154.4,
-147.8, -153.1, -152.7, -144.2, -151.5, -146. , -168.4, -159.4,
-167.1, -152. , -174.6, -143.8, -165.5, -159.7, -151.3, -148. ,
-140.8, -147.9, -144.8, -161.8, -136.2, -174.1, -165.3, -167.4,
-143.3, -156.4, -143. , -167.4, -155.6, -139.2, -148.7, -142.4,
-140. , -136.3, -146.9, -153.8, -146.1, -143.8, -129. , -143.6,
-147.4, -143.8, -152.9, -124.9, -131.3, -132.5, -140. , -142.8,
-156.3, -118.4, -140.9, -151.9, -138.7, -144.5, -147.6, -138.4,
-143.7, -146. , -137.6, -149.1, -165.9, -131.5, -158.8, -169.4,
-150.7, -158.4, -155.6, -130.1, -155.3, -155.8, -135. , -159.9,
-143.7, -147.5, -132.9, -144.6, -161.6, -157.1, -140.4, -142.3,
-154.9, -142.1, -160. , -151.3, -140.5, -150.3, -165.1, -165.2,
-148.9, -152.8, -153.6, -172.9, -160.7, -173.2, -164.7, -160.6,
-167.5, -163.5, -212.4, -169. , -179.2, -150. , -166.5, -165.7,
-144. , -153.9, -165.3, -176.5, -150.1, -144.1, -157.7, -138.9,
-150.9, -174.3, -149. , -148.8, -162.9, -160.1, -142.2, -163.6,
-146.1, -152.3, -175.4, -162.1, -154.8, -143.3, -180.3, -168.6,
-150.8, -143.8, -156.6, -151.1, -160.6, -154.8, -162.1, -161.7,
-150.2, -157. , -141.1, -147.5, -149.7, -174. , -156.2, -154.4,
-149.4, -125.9, -154.2, -157.6, -142.7, -158.9, -148. , -140.6,
-158.7, -125.8, -143.7, -145.2, -143.3, -143.2, -134.9, -142.6,
-135.5, -128.5, -143.7, -139.3, -152. , -146.1, -136.6, -144.8,
-161.2, -148.5, -129.4, -134.5, -136.4, -151.6, -142.6, -146.3,
-152. , -143.2, -151.7, -138.6]])
```

[ ]:

```
plt.figure(figsize = (10,5))
plt.subplot(121)
plt.plot(np.arange(len(rewards[0]))+1, rewards[0], color='darkgreen')
plt.title("Episodes vs rewards")
plt.xlabel("Number of episodes(x10)")
plt.ylabel("Reward per episode")

plt.subplot(122)
plt.plot(np.arange(len(steps[0]))+1, steps[0], color='darkgreen')
```

```
plt.title("Episodes vs steps ")
plt.xlabel("Number of episodes")
plt.ylabel("Steps taken per episode")
plt.show()
```



[ ]:
```
pprint.pprint(sweep_config)
```

```
{'method': 'random',
 'metric': {'goal': 'minimize', 'name': 'num_episodes_tosolve_cartpole'},
 'parameters': {'BATCH_SIZE': {'distribution': 'q_log_uniform_values',
                               'max': 512,
                               'min': 64,
                               'q': 1},
                'EPS_DECAY': {'distribution': 'uniform',
                              'max': 0.995,
                              'min': 0.9},
                'GAMMA': {'distribution': 'uniform', 'max': 1, 'min': 0.8},
                'LR': {'distribution': 'uniform', 'max': 0.01, 'min': 0.0001},
                'MAX_TRUNCATION': {'distribution': 'uniform',
                                   'max': 2,
                                   'min': 0.5},
                'NUM_LAYERS': {'values': [2, 4, 8]},
                'NUM_NEURONS_EACH_LAYER': {'values': [64, 32, 16]},
                'UPDATE_EVERY': {'distribution': 'q_log_uniform_values',
                                 'max': 200,
                                 'min': 10,
                                 'q': 1},
```

```
'a_TUNE': {'distribution': 'uniform', 'max': 1, 'min': 0}}
```

```
[ ]: def train(config = None):
       with wandb.init(config = config):
         config = wandb.config
         BUFFER_SIZE = int(1e5)  # replay buffer size
         BATCH_SIZE = config.BATCH_SIZE ## minibatch size
         GAMMA = config.GAMMA          ## discount factor
         LR = config.LR                ## learning rate
         UPDATE_EVERY = config.UPDATE_EVERY       ## how often to update the network␣
       ↪(When Q target is present)
         NUM_NEURONS_EACH_LAYER =  config.NUM_NEURONS_EACH_LAYER         ##number of␣
       ↪neurons in each hidden layer
         NUM_LAYERS =    config.NUM_LAYERS              ##number of layers for the␣
       ↪neural network
         MAX_TRUNCATION =  config.MAX_TRUNCATION                 ##max number of␣
       ↪steps in each episode(needed only for cartpole, acrobot and mountain car␣
       ↪these are already defined)
         EPS_DECAY =       config.EPS_DECAY                 ##epsilon decay for␣
       ↪epsilon exploration
         a_TUNE  =         config.a_TUNE

         steps_over_10_runs  = []
         rewards_over_10_runs = []
         episodes = []
         min_len = np.inf
         bad = False
         for run in range(10):
           print("Run: ",run,"␣
       ↪------------------------------------------------------------")
           env = gym.make('MountainCar-v0')

           seed = run
           env.seed(seed)
           state_shape = env.observation_space.shape[0]
           action_shape = env.action_space.n

           begin_time = datetime.datetime.now()
           agent = TutorialAgent(state_size=state_shape,action_size =␣
       ↪action_shape,seed =␣
       ↪seed,num_layers=NUM_LAYERS,neurons_each_layer=[NUM_NEURONS_EACH_LAYER]*NUM_LAYERS,lr␣
       ↪= LR,gamma= GAMMA,update_every=UPDATE_EVERY,batch_size =␣
       ↪BATCH_SIZE,max_tRUNCATION = MAX_TRUNCATION)
           steps_eps_greedy,reward_eps_greedy, episodes_eps_greedy,bad =␣
       ↪dqn(agent,eps_decay = EPS_DECAY, a = a_TUNE)
           steps_over_10_runs.append(steps_eps_greedy)
           rewards_over_10_runs.append(reward_eps_greedy)
```

```
    episodes.append(episodes_eps_greedy)
    if len(reward_eps_greedy) < min_len:
      min_len = len(reward_eps_greedy)
    time_taken = datetime.datetime.now() - begin_time

    print(time_taken)

  rewards = np.array(rewards_over_10_runs)
  avg_episodes = int(np.array(episodes).mean())
  steps = np.array(steps_over_10_runs)

  y, error = calculate_mean(rewards)
  plt.figure(figsize = (15,10))
  plt.subplot(121)
  plt.axvline(x = avg_episodes/10 , color = 'black', label ='Environment␣
↪solved')
  plt.scatter(avg_episodes/10,0, marker = 'x')
  for i in range(len(rewards_over_10_runs)):
    plt.plot(np.
↪arange(len(rewards_over_10_runs[i])),rewards_over_10_runs[i],label='run␣
↪'+str(i))
  plt.plot(np.arange(len(y))+1, y, color='darkblue',label='average')
  plt.fill_between(np.arange(len(y))+1, y-error, y+error,color =␣
↪'lightskyblue',label = 'standard deviation')
  plt.title("Episodes vs rewards")
  plt.xlabel("Number of episodes(x10)")
  plt.ylabel("Reward per episode")
  plt.legend()
  plt.subplot(122)
  y, error = calculate_mean(steps)
  for i in range(len(steps_over_10_runs)):
    plt.plot(np.
↪arange(len(steps_over_10_runs[i])),steps_over_10_runs[i],label='run '+str(i))
  plt.plot(np.arange(len(y))+1, y, color='darkblue',label='average')
  plt.fill_between(np.arange(len(y))+1, y-error, y+error,color =␣
↪'lightskyblue',label = 'standard deviation')
  plt.title("Episodes vs steps ")
  plt.xlabel("Number of episodes")
  plt.ylabel("Steps taken per episode")
  plt.legend()
  plt.savefig(str(CONFIGGG)+'runhyp'+str(LR)+'.jpg', dpi = 250)
  plt.show()
  for i in range(avg_episodes):
    print("Episode: ",i+1, 'Number of steps: ',round(y[i]))
  if bad == False:
    wandb.log({"num_episodes_tosolve_cartpole":avg_episodes-100})
  else:
```

```
    wandb.log({"num_episodes_tosolve_cartpole":5000})
```

[ ]:
```
wandb.agent(sweep_id, train, count=12)
```

```
wandb: Agent Starting Run: btvtcodk with config:
wandb:     BATCH_SIZE: 319
wandb:     EPS_DECAY: 0.902341235943596
wandb:     GAMMA: 0.8144181085986043
wandb:     LR: 0.0022755859727504983
wandb:     MAX_TRUNCATION: 1.074217920563914
wandb:     NUM_LAYERS: 2
wandb:     NUM_NEURONS_EACH_LAYER: 64
wandb:     UPDATE_EVERY: 15
wandb:     a_TUNE: 0.862186399649838
wandb: Currently logged in as: me19b190. Use `wandb
```

login --relogin` to force relogin

```
<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>
```

```
Run:  0 ------------------------------------------------------------
Episode 100     Average Score: -199.67
Episode 200     Average Score: -197.59
Episode 300     Average Score: -198.87
Episode 400     Average Score: -200.00
Episode 500     Average Score: -200.00
Episode 600     Average Score: -200.00
Episode 700     Average Score: -200.00
Episode 800     Average Score: -200.00
Episode 900     Average Score: -200.00
Episode 1000    Average Score: -200.00
Episode 1100    Average Score: -200.00
Episode 1200    Average Score: -200.00
Episode 1300    Average Score: -200.00
Episode 1400    Average Score: -199.25
Episode 1500    Average Score: -199.41
Episode 1600    Average Score: -200.00
Episode 1700    Average Score: -200.00
Episode 1737    Average Score: -200.00
```