# jb45dmyc9

March 30, 2023

**#Programming Assignment - 2**

## 0.1 ##Part 1: DQN

[ ]:
```
!pip install gym pyvirtualdisplay > /dev/null 2>&1
!apt-get install -y xvfb python-opengl ffmpeg > /dev/null 2>&1
!apt-get update > /dev/null 2>&1
!apt-get install cmake > /dev/null 2>&1
!pip install --upgrade setuptools 2>&1
!pip install ez_setup > /dev/null 2>&1
!pip install gym[atari] > /dev/null 2>&1
!pip install git+https://github.com/tensorflow/docs > /dev/null 2>&1
!pip install gym[classic_control]
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Requirement already satisfied: setuptools in /usr/local/lib/python3.9/dist-
packages (67.6.0)
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Requirement already satisfied: gym[classic_control] in
/usr/local/lib/python3.9/dist-packages (0.25.2)
Requirement already satisfied: numpy>=1.18.0 in /usr/local/lib/python3.9/dist-
packages (from gym[classic_control]) (1.22.4)
Requirement already satisfied: gym-notices>=0.0.4 in
/usr/local/lib/python3.9/dist-packages (from gym[classic_control]) (0.0.8)
Requirement already satisfied: cloudpickle>=1.2.0 in
/usr/local/lib/python3.9/dist-packages (from gym[classic_control]) (2.2.1)
Requirement already satisfied: importlib-metadata>=4.8.0 in
/usr/local/lib/python3.9/dist-packages (from gym[classic_control]) (6.1.0)
Requirement already satisfied: pygame==2.1.0 in /usr/local/lib/python3.9/dist-
packages (from gym[classic_control]) (2.1.0)
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.9/dist-
packages (from importlib-metadata>=4.8.0->gym[classic_control]) (3.15.0)
```

[ ]:
```
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
plt.style.use('classic')
```

[ ]:
```
'''
### Q Network & Some 'hyperparameters'

QNetwork1:
Input Layer - 4 nodes (State Shape) \
Hidden Layer 1 - 64 nodes \
Hidden Layer 2 - 64 nodes \
Output Layer - 2 nodes (Action Space) \
Optimizer - zero_grad()

QNetwork2: Feel free to experiment more
'''

import torch
import torch.nn as nn
import torch.nn.functional as F




class QNetwork1(nn.Module):

    def __init__(self, state_size, action_size, seed, fc1_units=128,␣
 ↪fc2_units=64):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
```

```
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(QNetwork1, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)

    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

[ ]:
```
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):
        """Add a new experience to memory."""
        e = self.experience(state, action, reward, next_state, done)
```

```
        self.memory.append(e)

    def sample(self):
        """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)
```

[ ]:
```
class Agent():

    def __init__(self, state_size, action_size,␣
↪seed,buffer_size,batch_size,gamma,lr,update_every):

        ''' Agent Environment Interaction '''
        self.state_size = state_size
        self.action_size = action_size
        self.seed = random.seed(seed)
        self.buffer_size=buffer_size
        self.batch_size=batch_size
        self.gamma=gamma
        self.lr=lr
        self.update_every=update_every

        ''' Q-Network '''
        self.qnetwork_local = QNetwork1(state_size, action_size, seed).
↪to(device)
        self.qnetwork_target = QNetwork1(state_size, action_size, seed).
↪to(device)
        self.optimizer = optim.Adam(self.qnetwork_local.parameters(), lr=self.
↪lr)
```

```
        ''' Replay memory '''
        self.memory = ReplayBuffer(action_size, self.buffer_size, self.
↪batch_size, seed)

        ''' Initialize time step (for updating every UPDATE_EVERY steps)       ␣
↪    -Needed for Q Targets '''
        self.t_step = 0

    def step(self, state, action, reward, next_state, done):

        ''' Save experience in replay memory '''
        self.memory.add(state, action, reward, next_state, done)

        ''' If enough samples are available in memory, get random subset and␣
↪learn '''
        if len(self.memory) >= self.batch_size:
            experiences = self.memory.sample()
            self.learn(experiences)

        """ +Q TARGETS PRESENT """
        ''' Updating the Network every 'UPDATE_EVERY' steps taken '''
        self.t_step = (self.t_step + 1) % self.update_every
        if self.t_step == 0:

            self.qnetwork_target.load_state_dict(self.qnetwork_local.
↪state_dict())

    def act(self, state, eps=0.):

        state = torch.from_numpy(state).float().unsqueeze(0).to(device)
        self.qnetwork_local.eval()
        with torch.no_grad():
            action_values = self.qnetwork_local(state)
        self.qnetwork_local.train()
        ''' Epsilon-greedy action selection (Already Present) '''
        if random.random() > eps:
            return np.argmax(action_values.cpu().data.numpy())
        else:
            return random.choice(np.arange(self.action_size))

    def learn(self, experiences):
        """ +E EXPERIENCE REPLAY PRESENT """
        states, actions, rewards, next_states, dones = experiences

        ''' Get max predicted Q values (for next states) from target model'''
        Q_targets_next = self.qnetwork_target(next_states).detach().max(1)[0].
↪unsqueeze(1)
```

```
        ''' Compute Q targets for current states '''
        Q_targets = rewards + (self.gamma * Q_targets_next * (1 - dones))

        ''' Get expected Q values from local model '''
        Q_expected = self.qnetwork_local(states).gather(1, actions)

        ''' Compute loss '''
        loss = F.mse_loss(Q_expected, Q_targets)

        ''' Minimize the loss '''
        self.optimizer.zero_grad()
        loss.backward()

        ''' Gradiant Clipping '''
        """ +T TRUNCATION PRESENT """
        for param in self.qnetwork_local.parameters():
            param.grad.data.clamp_(-1, 1)

        self.optimizer.step()
```

[ ]:
```
''' Defining DQN Algorithm '''



def dqn(n_episodes=10000, max_t=1000, eps_start=1.0, eps_end=0.01, eps_decay=0.
 ↪995, req_score=200):

    scores_point = []
    scores_running=[]
    ''' list containing scores from each episode '''

    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 '''
    step_his=[]

    eps = eps_start
    ''' initialize epsilon '''

    for i_episode in range(1, n_episodes+1):
        state = env.reset()
        score = 0
        for t in range(max_t):
            action = agent.act(state, eps)
```

```
            next_state, reward, done, _ = env.step(action)
            agent.step(state, action, reward, next_state, done)
            state = next_state
            score += reward
            if done:
                break
        step_his.append(t+1)

        scores_window.append(score)
        scores_window_printing.append(score)
        scores_point.append(score)
        scores_running.append(np.mean(scores_window_printing))

        ''' save most recent score '''

        eps = max(eps_end, eps_decay*eps)
        ''' decrease epsilon '''
        if i_episode%100==0:
          print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.
↪mean(scores_window)))


        if np.mean(scores_window)>=req_score:
           print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.
↪2f}'.format(i_episode, np.mean(scores_window)))
            break
    return [np.array(scores_point),np.array(scores_running),np.array(step_his)]
```

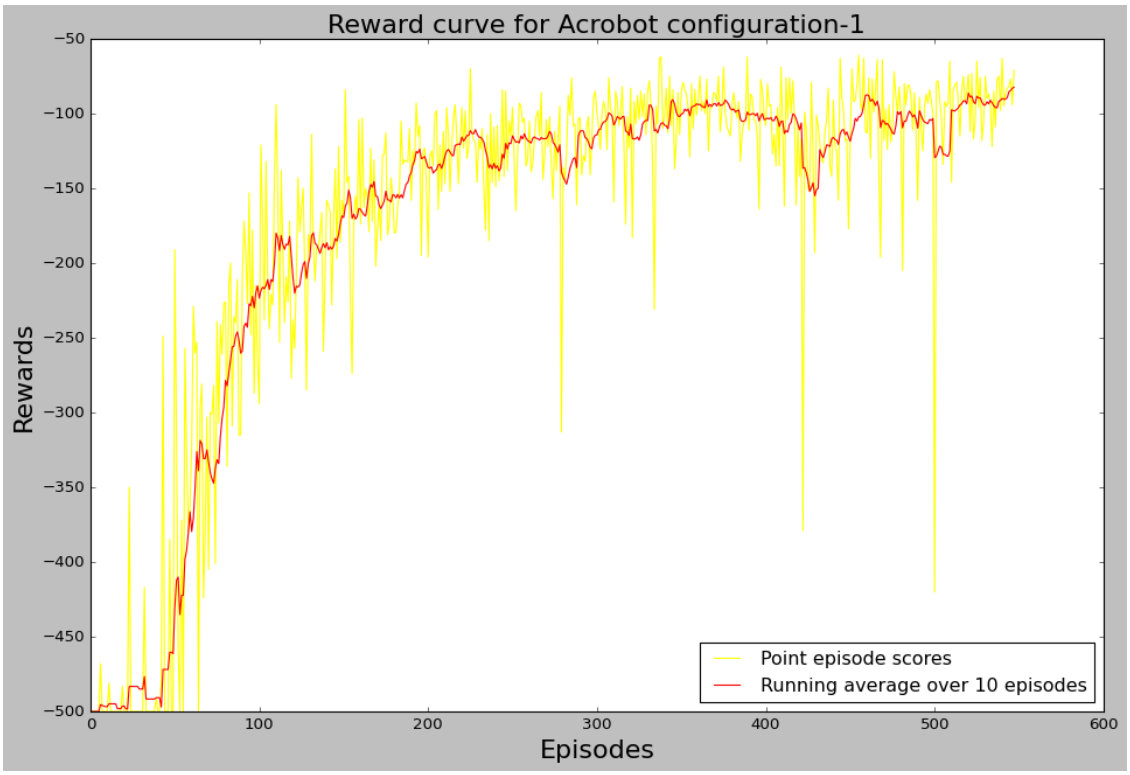
# 1 *Acrobot Configuration — 1*

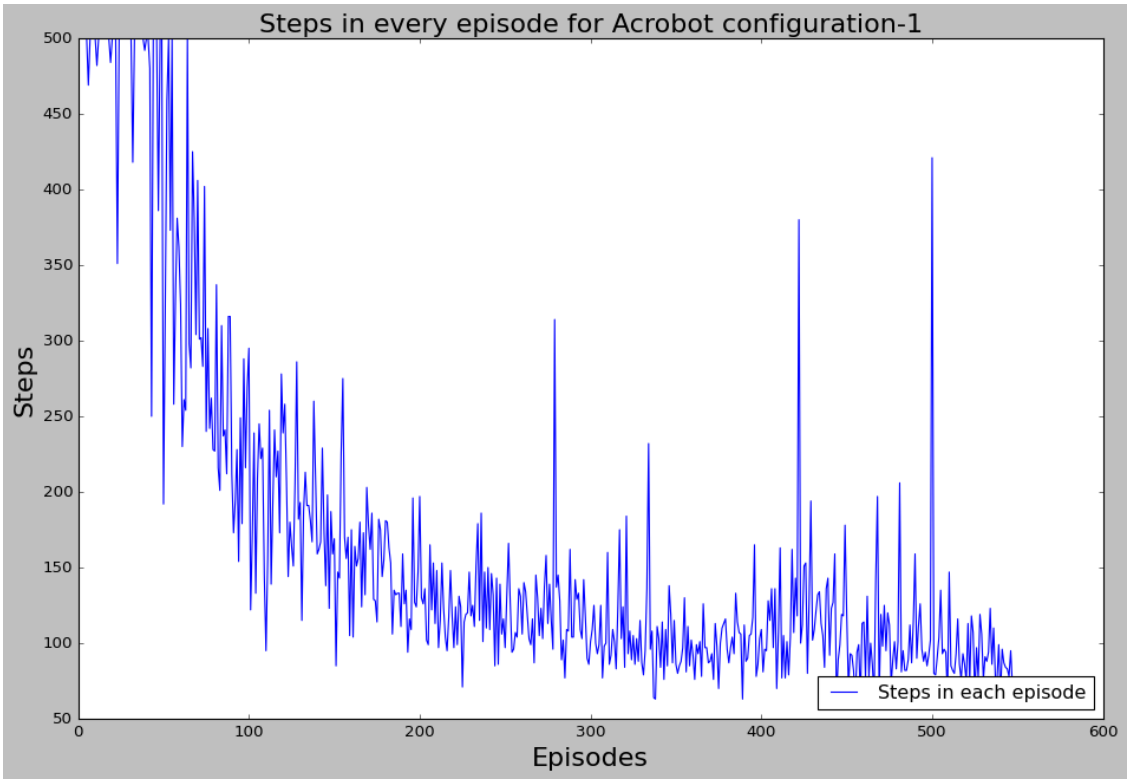
[ ]:
```
begin_time = datetime.datetime.now()

env = gym.make('Acrobot-v1')
env.seed(0)
state_shape = env.observation_space.shape[0]
action_shape = env.action_space.n


BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 64         # minibatch size
GAMMA = 0.99            # discount factor
LR = 5e-4               # learning rate
UPDATE_EVERY = 20       # how often to update the network (When Q target is␣
↪present)
```

```
agent = Agent(state_shape, action_shape,␣
 ↪0,BUFFER_SIZE,BATCH_SIZE,GAMMA,LR,UPDATE_EVERY)


scores_point,scores_running,steps=dqn(req_score=-100)


time_taken = datetime.datetime.now() - begin_time

print(time_taken)
```

```
Episode 100     Average Score: -389.36
Episode 200     Average Score: -170.15
Episode 300     Average Score: -122.49
Episode 400     Average Score: -101.57
Episode 500     Average Score: -110.04

Environment solved in 548 episodes!     Average Score: -99.83
0:06:15.688342
```

[ ]:
```
plt.figure(figsize=(14,9))
plt.plot(np.arange(len(scores_point)),scores_point,label='Point episode␣
 ↪scores',color='yellow')
plt.plot(np.arange(len(scores_running)),scores_running,label='Running average␣
 ↪over 10 episodes',color='red')
plt.xlabel('Episodes',fontsize=20)
plt.ylabel('Rewards',fontsize=20)
plt.title('Reward curve for Acrobot configuration-1',fontsize=20)
plt.legend(loc='lower right')
plt.figure(figsize=(14,9))
plt.plot(np.arange(steps.shape[0]),steps,label='Steps in each episode')
plt.xlabel('Episodes',fontsize=20)
plt.ylabel('Steps',fontsize=20)
plt.title('Steps in every episode for Acrobot configuration-1',fontsize=20)
plt.legend(loc='lower right')
plt.show()
```


Reward curve for Acrobot configuration-1
Rewards
Episodes
Point episode scores
Running average over 10 episodes
−50
−100
−150
−200
−250
−300
−350
−400
−450
−500
0
100
200
300
400
500
600


Steps in every episode for Acrobot configuration-1
Steps
Episodes
Steps in each episode
500
450
400
350
300
250
200
150
100
50
0
100
200
300
400
500
600

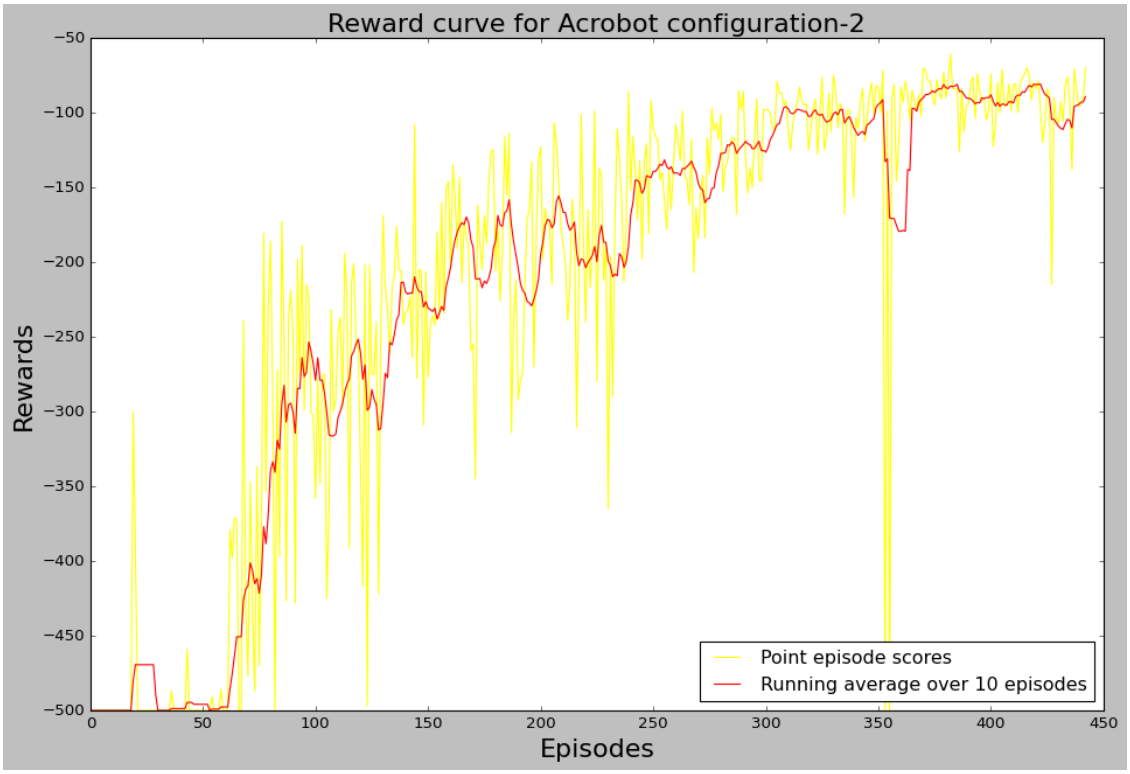
## 2 *Acrobot Configuration – 2*

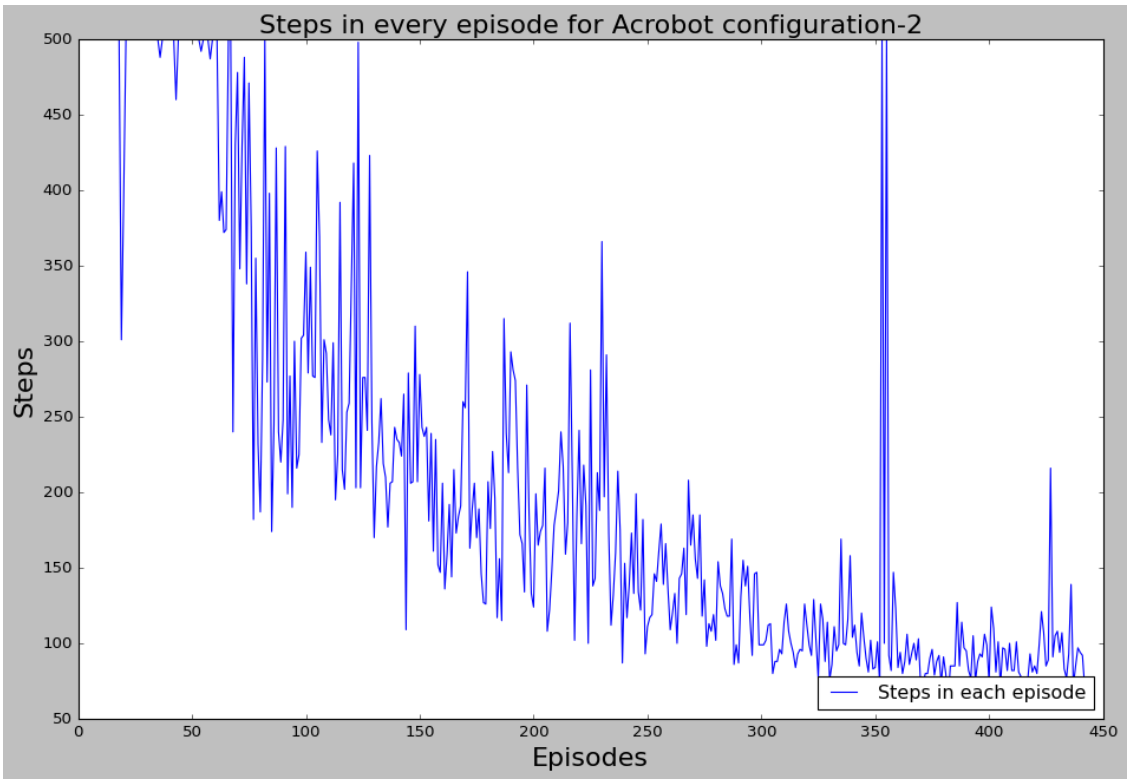
[ ]:
```
# Increasing the lr to 1e-3

begin_time = datetime.datetime.now()

env = gym.make('Acrobot-v1')
env.seed(0)
state_shape = env.observation_space.shape[0]
action_shape = env.action_space.n


BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 64         # minibatch size
GAMMA = 0.99            # discount factor
LR = 1e-3               # learning rate
UPDATE_EVERY = 20       # how often to update the network (When Q target is␣
 ↪present)

agent = Agent(state_shape, action_shape,␣
 ↪0,BUFFER_SIZE,BATCH_SIZE,GAMMA,LR,UPDATE_EVERY)


scores_point,scores_running,steps=dqn(req_score=-100)


time_taken = datetime.datetime.now() - begin_time

print(time_taken)
```

```
Episode 100     Average Score: -431.17
Episode 200     Average Score: -231.09
Episode 300     Average Score: -154.30
Episode 400     Average Score: -104.17

Environment solved in 443 episodes!     Average Score: -99.85
0:05:41.557047
```

[ ]:
```
plt.figure(figsize=(14,9))
plt.plot(np.arange(len(scores_point)),scores_point,label='Point episode␣
 ↪scores',color='yellow')
plt.plot(np.arange(len(scores_running)),scores_running,label='Running average␣
 ↪over 10 episodes',color='red')
plt.xlabel('Episodes',fontsize=20)
```

```
plt.ylabel('Rewards',fontsize=20)
plt.title('Reward curve for Acrobot configuration-2',fontsize=20)
plt.legend(loc='lower right')
plt.figure(figsize=(14,9))
plt.plot(np.arange(steps.shape[0]),steps,label='Steps in each episode')
plt.xlabel('Episodes',fontsize=20)
plt.ylabel('Steps',fontsize=20)
plt.title('Steps in every episode for Acrobot configuration-2',fontsize=20)
plt.legend(loc='lower right')
plt.show()
```






## 3 *Acrobot Configuration – 3*

[ ]:
```
class QNetwork1(nn.Module):

    def __init__(self, state_size, action_size, seed, fc1_units=128,␣
↪fc2_units=64):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(QNetwork1, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc2_units)
        self.fc2 = nn.Linear(fc2_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)
```

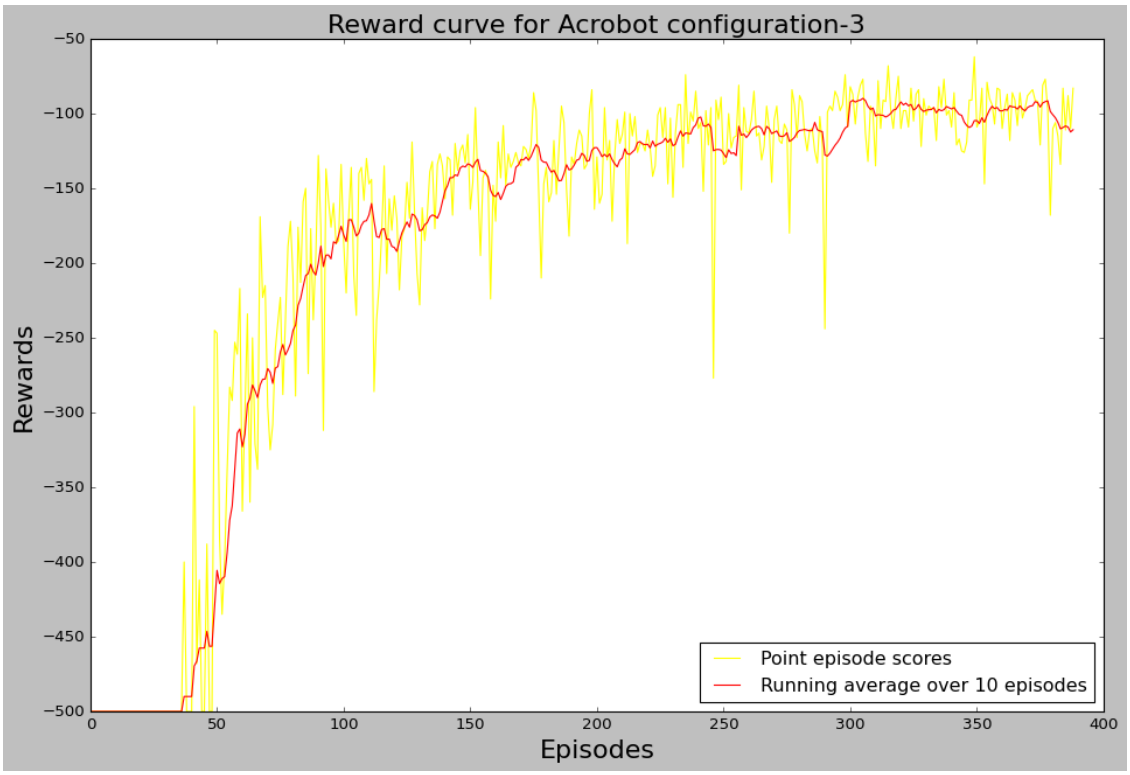
```
    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        #x=F.relu(self.fc3(x))
        return self.fc3(x)
```

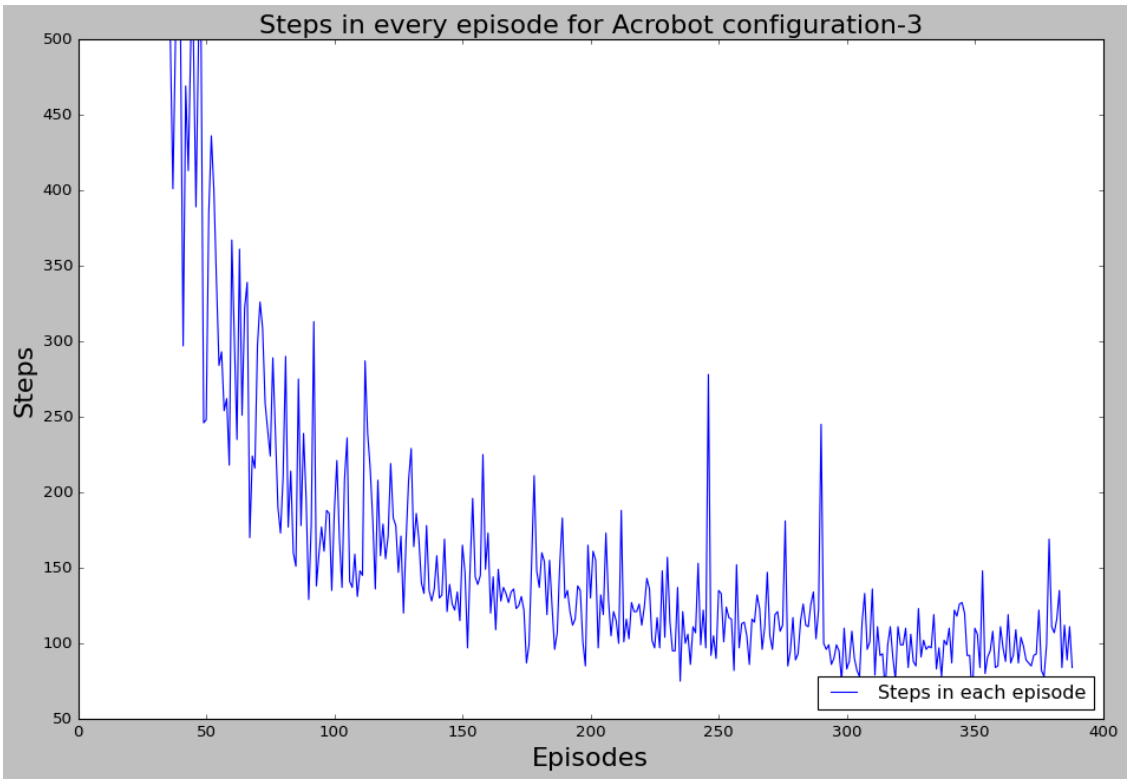
[ ]:
```
begin_time = datetime.datetime.now()

env = gym.make('Acrobot-v1')
env.seed(0)
state_shape = env.observation_space.shape[0]
action_shape = env.action_space.n


BUFFER_SIZE = int(1e4)  # replay buffer size
BATCH_SIZE = 64         # minibatch size
GAMMA = 0.99            # discount factor
LR = 1e-3               # learning rate
UPDATE_EVERY = 20       # how often to update the network (When Q target is␣
 ↪present)

agent = Agent(state_shape, action_shape,␣
 ↪0,BUFFER_SIZE,BATCH_SIZE,GAMMA,LR,UPDATE_EVERY)


scores_point,scores_running,steps=dqn(req_score=-100)


time_taken = datetime.datetime.now() - begin_time

print(time_taken)
```

```
Episode 100     Average Score: -364.67
Episode 200     Average Score: -150.61
Episode 300     Average Score: -116.39

Environment solved in 389 episodes!     Average Score: -99.99
0:03:22.209157
```

[ ]:
```
plt.figure(figsize=(14,9))
plt.plot(np.arange(len(scores_point)),scores_point,label='Point episode␣
 ↪scores',color='yellow')
plt.plot(np.arange(len(scores_running)),scores_running,label='Running average␣
 ↪over 10 episodes',color='red')
plt.xlabel('Episodes',fontsize=20)
plt.ylabel('Rewards',fontsize=20)
```

```
plt.title('Reward curve for Acrobot configuration-3',fontsize=20)
plt.legend(loc='lower right')
plt.figure(figsize=(14,9))
plt.plot(np.arange(steps.shape[0]),steps,label='Steps in each episode')
plt.xlabel('Episodes',fontsize=20)
plt.ylabel('Steps',fontsize=20)
plt.title('Steps in every episode for Acrobot configuration-3',fontsize=20)
plt.legend(loc='lower right')
plt.show()
```






## 4 *Acrobot Configuration – 4*

[ ]:
```
class QNetwork1(nn.Module):

    def __init__(self, state_size, action_size, seed, fc1_units=128,␣
↪fc2_units=64):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(QNetwork1, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc2_units)
        self.fc2 = nn.Linear(fc2_units, fc2_units)
       # self.fc3=nn.Linear(fc1_units,fc2_units)     # NEW LAYER IN␣
↪CONFIGURATION-3
```

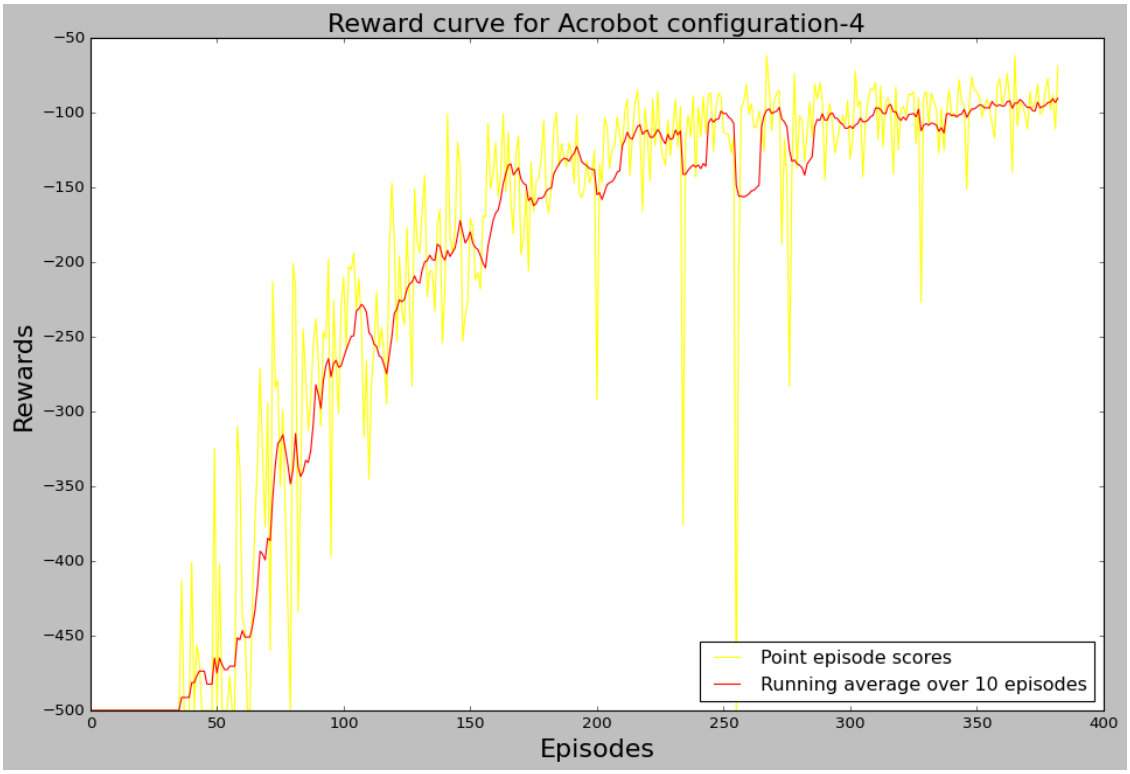
```
        self.fc3 = nn.Linear(fc2_units, action_size)

    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        #x=F.relu(self.fc3(x))
        return self.fc3(x)
```

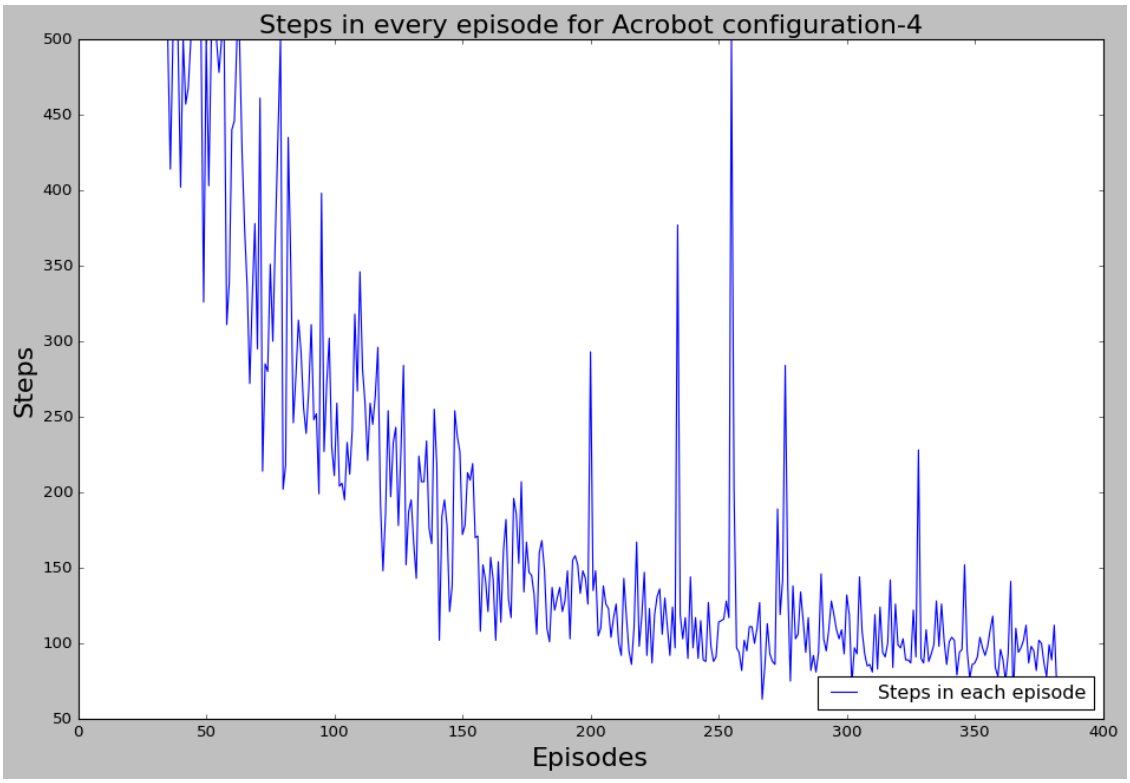
[ ]:
```
begin_time = datetime.datetime.now()

env = gym.make('Acrobot-v1')
env.seed(0)
state_shape = env.observation_space.shape[0]
action_shape = env.action_space.n


BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 64         # minibatch size
GAMMA = 0.99            # discount factor
LR = 1e-3               # learning rate
UPDATE_EVERY = 20       # how often to update the network (When Q target is␣
 ↪present)

agent = Agent(state_shape, action_shape,␣
 ↪0,BUFFER_SIZE,BATCH_SIZE,GAMMA,LR,UPDATE_EVERY)


scores_point,scores_running,steps=dqn(req_score=-100)


time_taken = datetime.datetime.now() - begin_time

print(time_taken)
```

```
Episode 100     Average Score: -420.82
Episode 200     Average Score: -181.42
Episode 300     Average Score: -120.64

Environment solved in 383 episodes!     Average Score: -99.89
0:03:53.234243
```

[ ]:
```
plt.figure(figsize=(14,9))
plt.plot(np.arange(len(scores_point)),scores_point,label='Point episode␣
 ↪scores',color='yellow')
plt.plot(np.arange(len(scores_running)),scores_running,label='Running average␣
 ↪over 10 episodes',color='red')
```

```
plt.xlabel('Episodes',fontsize=20)
plt.ylabel('Rewards',fontsize=20)
plt.title('Reward curve for Acrobot configuration-4',fontsize=20)
plt.legend(loc='lower right')
plt.figure(figsize=(14,9))
plt.plot(np.arange(steps.shape[0]),steps,label='Steps in each episode')
plt.xlabel('Episodes',fontsize=20)
plt.ylabel('Steps',fontsize=20)
plt.title('Steps in every episode for Acrobot configuration-4',fontsize=20)
plt.legend(loc='lower right')
plt.show()
```






## 5 *Acrobot Configuration – 5*

[ ]:
```
class QNetwork1(nn.Module):

    def __init__(self, state_size, action_size, seed, fc1_units=128,␣
↪fc2_units=64):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(QNetwork1, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc2_units)
        self.fc2 = nn.Linear(fc2_units, fc2_units)
       # self.fc3=nn.Linear(fc1_units,fc2_units)      # NEW LAYER IN␣
↪CONFIGURATION-3
```

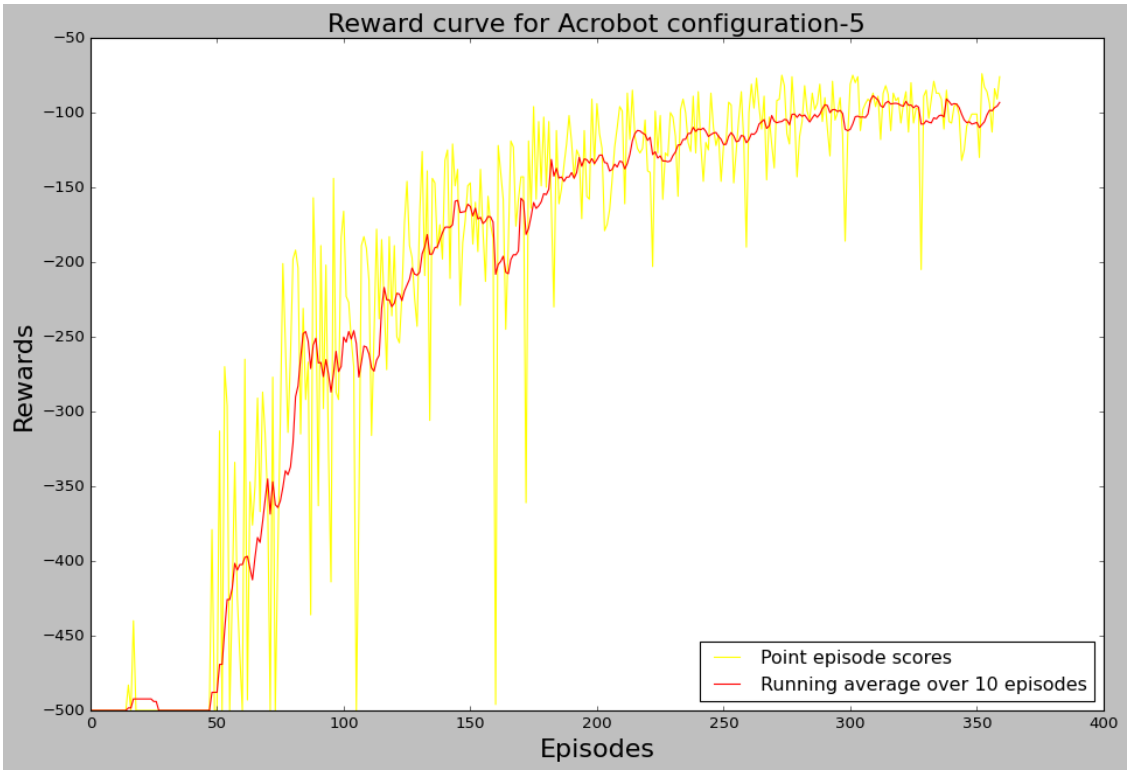
```
        self.fc3 = nn.Linear(fc2_units, action_size)

    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        #x=F.relu(self.fc3(x))
        return self.fc3(x)
```

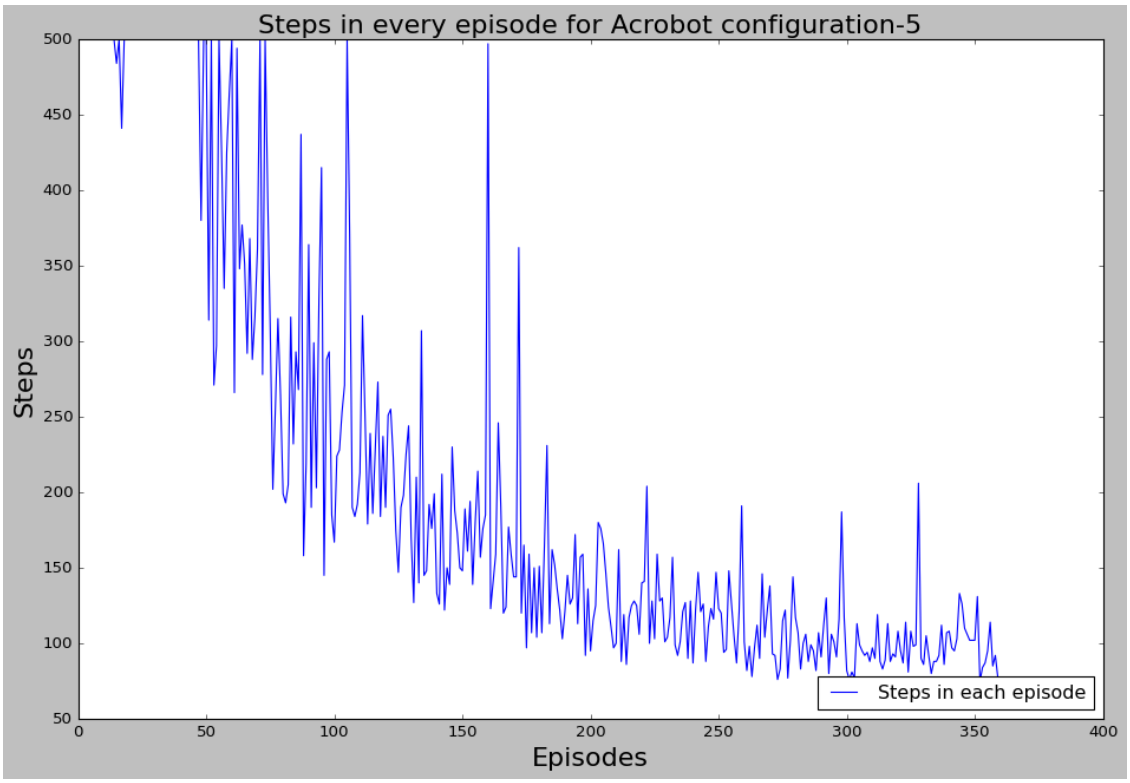
[ ]:
```
begin_time = datetime.datetime.now()

env = gym.make('Acrobot-v1')
env.seed(0)
state_shape = env.observation_space.shape[0]
action_shape = env.action_space.n


BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 128         # minibatch size
GAMMA = 0.99            # discount factor
LR = 1e-3               # learning rate
UPDATE_EVERY = 20       # how often to update the network (When Q target is␣
 ↪present)

agent = Agent(state_shape, action_shape,␣
 ↪0,BUFFER_SIZE,BATCH_SIZE,GAMMA,LR,UPDATE_EVERY)


scores_point,scores_running,steps=dqn(req_score=-100)


time_taken = datetime.datetime.now() - begin_time

print(time_taken)
```

```
Episode 100     Average Score: -409.97
Episode 200     Average Score: -183.45
Episode 300     Average Score: -114.48

Environment solved in 360 episodes!     Average Score: -99.81
0:04:41.785370
```

[ ]:
```
plt.figure(figsize=(14,9))
plt.plot(np.arange(len(scores_point)),scores_point,label='Point episode␣
 ↪scores',color='yellow')
plt.plot(np.arange(len(scores_running)),scores_running,label='Running average␣
 ↪over 10 episodes',color='red')
```

```
plt.xlabel('Episodes',fontsize=20)
plt.ylabel('Rewards',fontsize=20)
plt.title('Reward curve for Acrobot configuration-5',fontsize=20)
plt.legend(loc='lower right')
plt.figure(figsize=(14,9))
plt.plot(np.arange(steps.shape[0]),steps,label='Steps in each episode')
plt.xlabel('Episodes',fontsize=20)
plt.ylabel('Steps',fontsize=20)
plt.title('Steps in every episode for Acrobot configuration-5',fontsize=20)
plt.legend(loc='lower right')
plt.show()
```






[ ]:
```
display = Display(visible=0, size=(400, 300))
display.start()


def render_episode(env: gym.Env, max_steps: int):
  screen = env.render(mode='rgb_array')
  im = Image.fromarray(screen)

  images = [im]

  state = env.reset()#tf.constant(env.reset(), dtype=tf.float32)
  for i in range(1, max_steps + 1):
    action = agent.act(state, 0)
    state, reward, done, _ = env.step(action)

    # Render screen every 10 steps
    if i % 10 == 0:
      screen = env.render(mode='rgb_array')
      images.append(Image.fromarray(screen))

    if done:
      break
```

```
  return images


# Save GIF image
images = render_episode(env, 200)
image_file = 'acrobot-v1.gif'
# loop=0: loop forever, duration=1: play each frame for 1ms
images[0].save(
    image_file, save_all=True, append_images=images[1:], loop=0, duration=1)
```

```
/usr/local/lib/python3.9/dist-packages/gym/core.py:43: DeprecationWarning:
WARN: The argument mode in render method is deprecated; use render_mode

during environment initialization instead.

See here for more information: https://www.gymlibrary.ml/content/api/
  deprecation(
```

[ ]:
```
import tensorflow_docs.vis.embed as embed
embed.embed_file(image_file)
```

[ ]:
```
<IPython.core.display.HTML object>
```

## 6 *CartPole Configuration – 1*

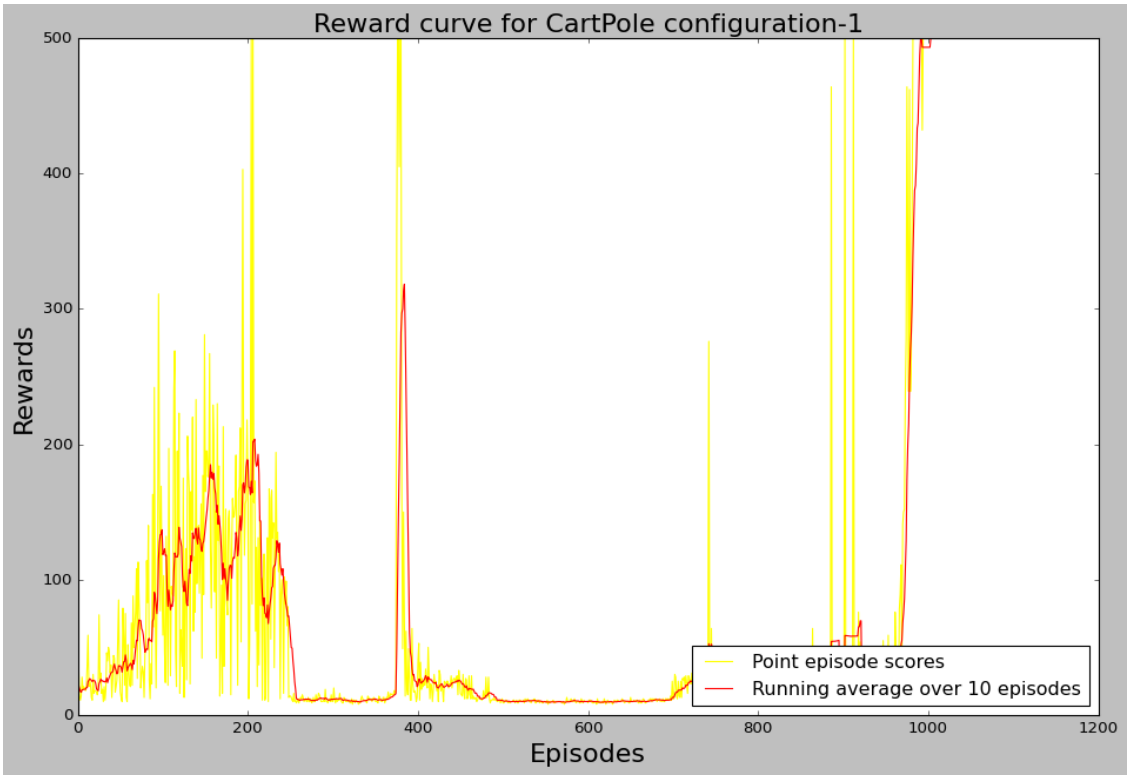
[ ]:
```
class QNetwork1(nn.Module):

    def __init__(self, state_size, action_size, seed, fc1_units=128,␣
 ↪fc2_units=64):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(QNetwork1, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)

    def forward(self, state):
        """Build a network that maps state -> action values."""
```

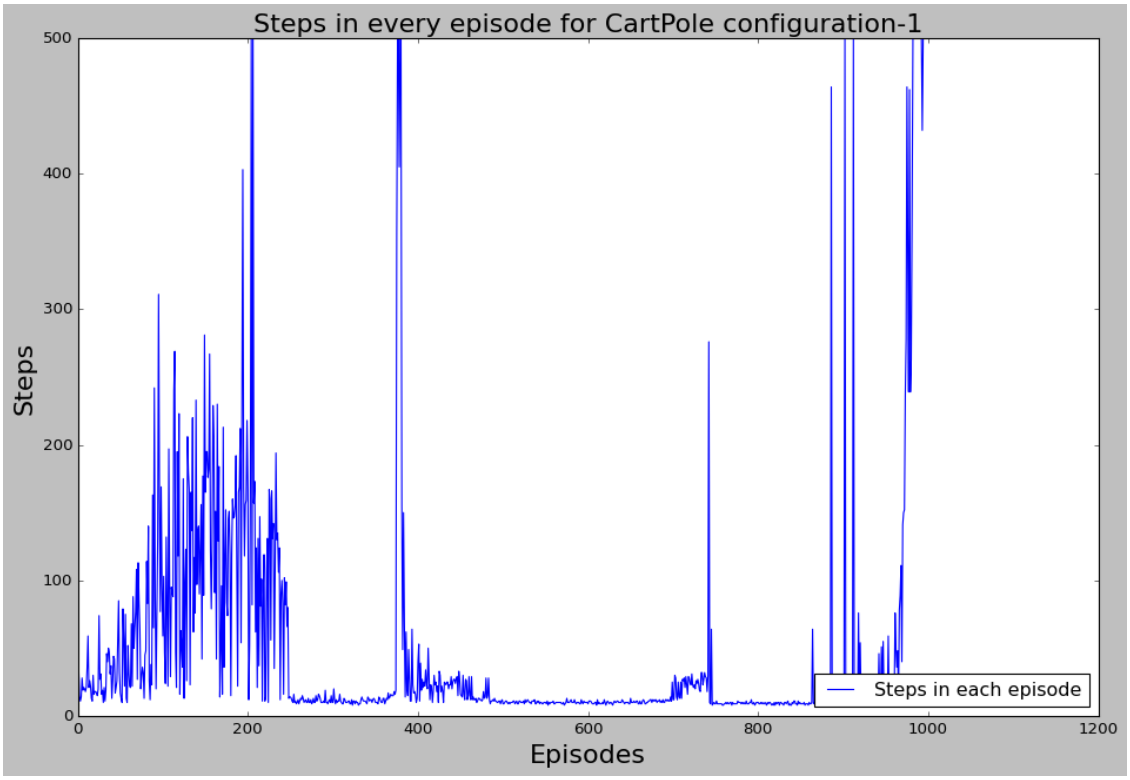
```
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return self.fc3(x)

begin_time = datetime.datetime.now()

env = gym.make('CartPole-v1')
env.seed(0)
state_shape = env.observation_space.shape[0]
action_shape = env.action_space.n


BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 64         # minibatch size
GAMMA = 0.99            # discount factor
LR = 5e-4               # learning rate
UPDATE_EVERY = 20       # how often to update the network (When Q target is␣
 ↪present)

agent = Agent(state_shape, action_shape,␣
 ↪0,BUFFER_SIZE,BATCH_SIZE,GAMMA,LR,UPDATE_EVERY)


scores_point,scores_running,steps=dqn(req_score=200)


time_taken = datetime.datetime.now() - begin_time

print(time_taken)
```

```
Episode 100     Average Score: 48.16
Episode 200     Average Score: 128.28
Episode 300     Average Score: 60.05
Episode 400     Average Score: 44.30
Episode 500     Average Score: 19.58
Episode 600     Average Score: 10.05
Episode 700     Average Score: 10.49
Episode 800     Average Score: 18.44
Episode 900     Average Score: 14.41
Episode 1000    Average Score: 146.67

Environment solved in 1012 episodes!    Average Score: 200.66
0:02:35.504611
```

[ ]:
```
plt.figure(figsize=(14,9))
plt.plot(np.arange(len(scores_point)),scores_point,label='Point episode␣
 ↪scores',color='yellow')
```

```
plt.plot(np.arange(len(scores_running)),scores_running,label='Running average␣
  ↪over 10 episodes',color='red')
plt.xlabel('Episodes',fontsize=20)
plt.ylabel('Rewards',fontsize=20)
plt.title('Reward curve for CartPole configuration-1',fontsize=20)
plt.legend(loc='lower right')
plt.figure(figsize=(14,9))
plt.plot(np.arange(steps.shape[0]),steps,label='Steps in each episode')
plt.xlabel('Episodes',fontsize=20)
plt.ylabel('Steps',fontsize=20)
plt.title('Steps in every episode for CartPole configuration-1',fontsize=20)
plt.legend(loc='lower right')
plt.show()
```






## 7 *CartPole Configuration – 2*

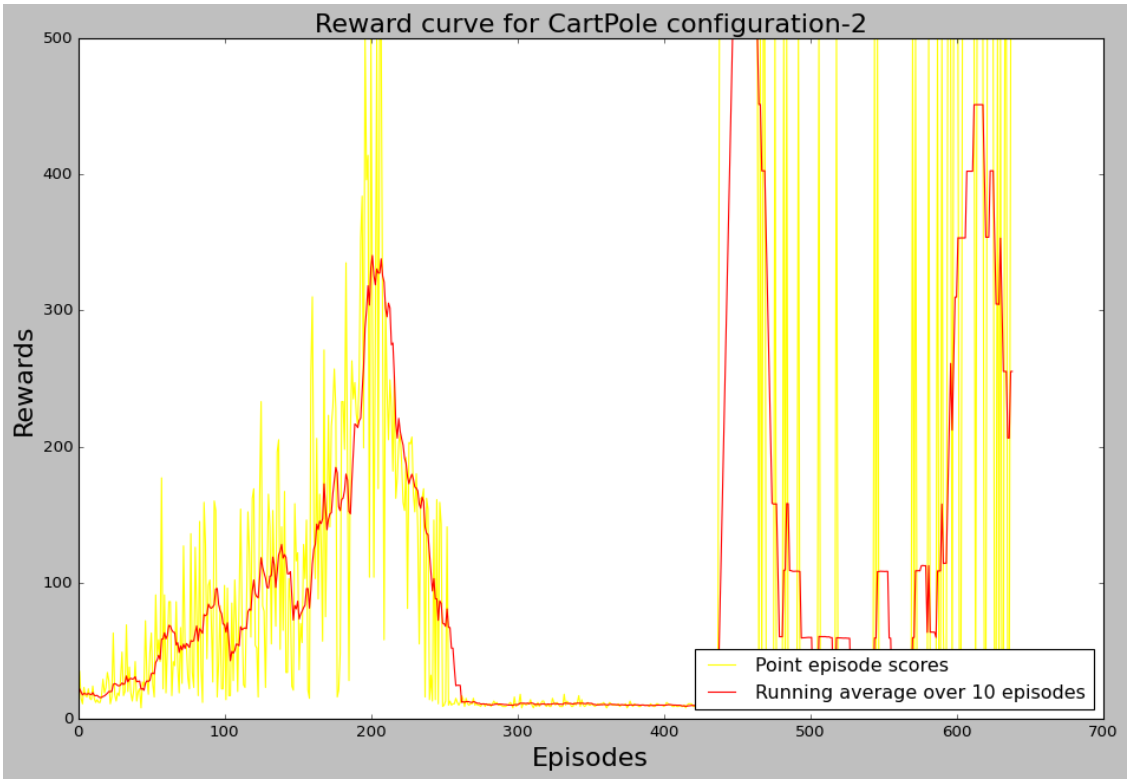
[ ]:
```
#decreaseing or increasing the learning rate doesn't seem to be doing any␣
↪better. Decreasing makes the changes slow and increasing makes it oscillating
```

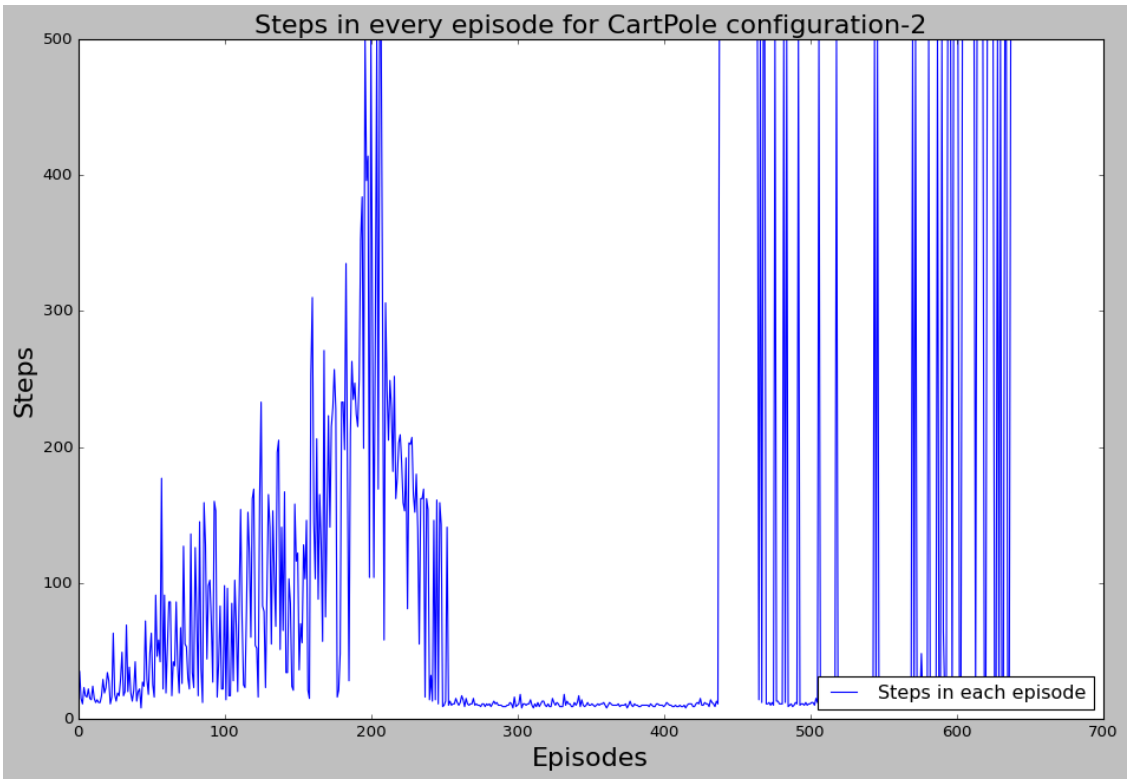
[ ]:
```
#increasing the batch_size to 128
class QNetwork1(nn.Module):

    def __init__(self, state_size, action_size, seed, fc1_units=128,␣
↪fc2_units=64):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(QNetwork1, self).__init__()
        self.seed = torch.manual_seed(seed)
```

```
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)

    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return self.fc3(x)

begin_time = datetime.datetime.now()

env = gym.make('CartPole-v1')
env.seed(0)
state_shape = env.observation_space.shape[0]
action_shape = env.action_space.n


BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 128         # minibatch size
GAMMA = 0.99            # discount factor
LR = 5e-4               # learning rate
UPDATE_EVERY = 20       # how often to update the network (When Q target is␣
 ↪present)

agent = Agent(state_shape, action_shape,␣
 ↪0,BUFFER_SIZE,BATCH_SIZE,GAMMA,LR,UPDATE_EVERY)


scores_point,scores_running,steps=dqn(req_score=200)


time_taken = datetime.datetime.now() - begin_time

print(time_taken)
```

```
Episode 100     Average Score: 44.20
Episode 200     Average Score: 137.69
Episode 300     Average Score: 99.12
Episode 400     Average Score: 10.58
Episode 500     Average Score: 177.07
Episode 600     Average Score: 80.03

Environment solved in 639 episodes!     Average Score: 202.46
0:04:09.985937
```

[ ]:
```
plt.figure(figsize=(14,9))
plt.plot(np.arange(len(scores_point)),scores_point,label='Point episode␣
 ↪scores',color='yellow')
plt.plot(np.arange(len(scores_running)),scores_running,label='Running average␣
 ↪over 10 episodes',color='red')
plt.xlabel('Episodes',fontsize=20)
plt.ylabel('Rewards',fontsize=20)
plt.title('Reward curve for CartPole configuration-2',fontsize=20)
plt.legend(loc='lower right')
plt.figure(figsize=(14,9))
plt.plot(np.arange(steps.shape[0]),steps,label='Steps in each episode')
plt.xlabel('Episodes',fontsize=20)
plt.ylabel('Steps',fontsize=20)
plt.title('Steps in every episode for CartPole configuration-2',fontsize=20)
plt.legend(loc='lower right')
plt.show()
```






## 8 *CartPole Configuration – 3*

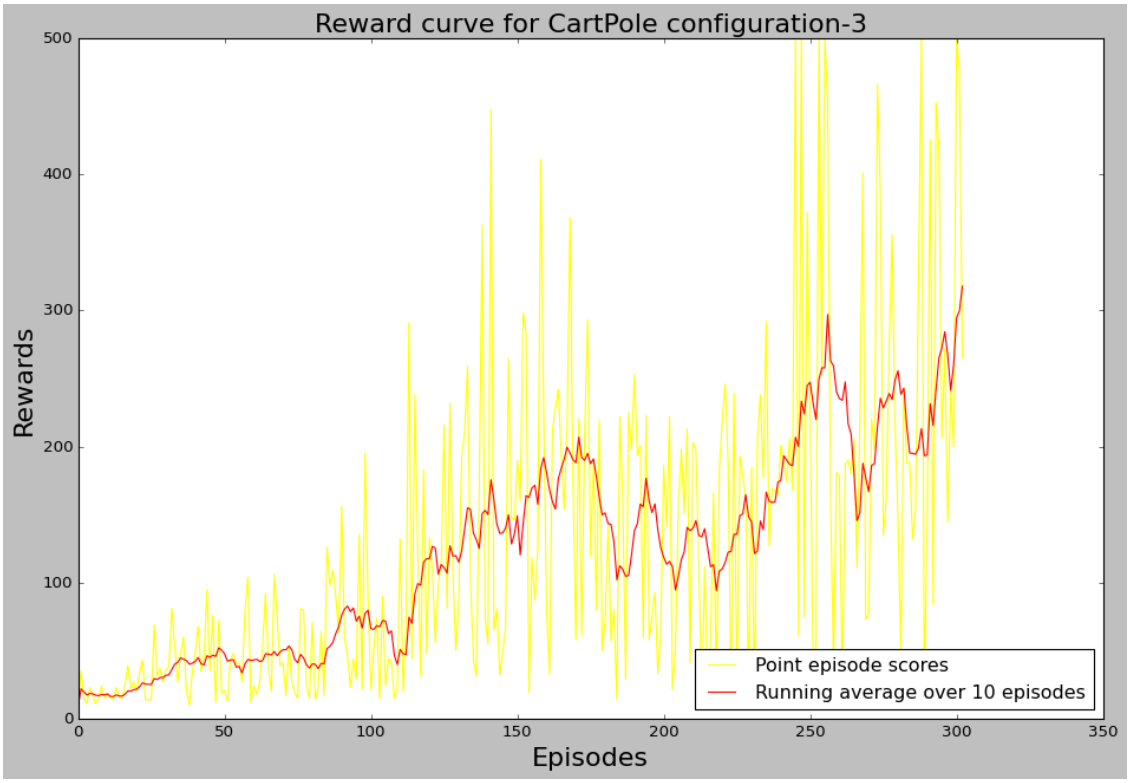
[ ]:
```
#Increasing the size of the of neural network
class QNetwork1(nn.Module):

    def __init__(self, state_size, action_size, seed, fc1_units=256,␣
 ↪fc2_units=128):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(QNetwork1, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)
```

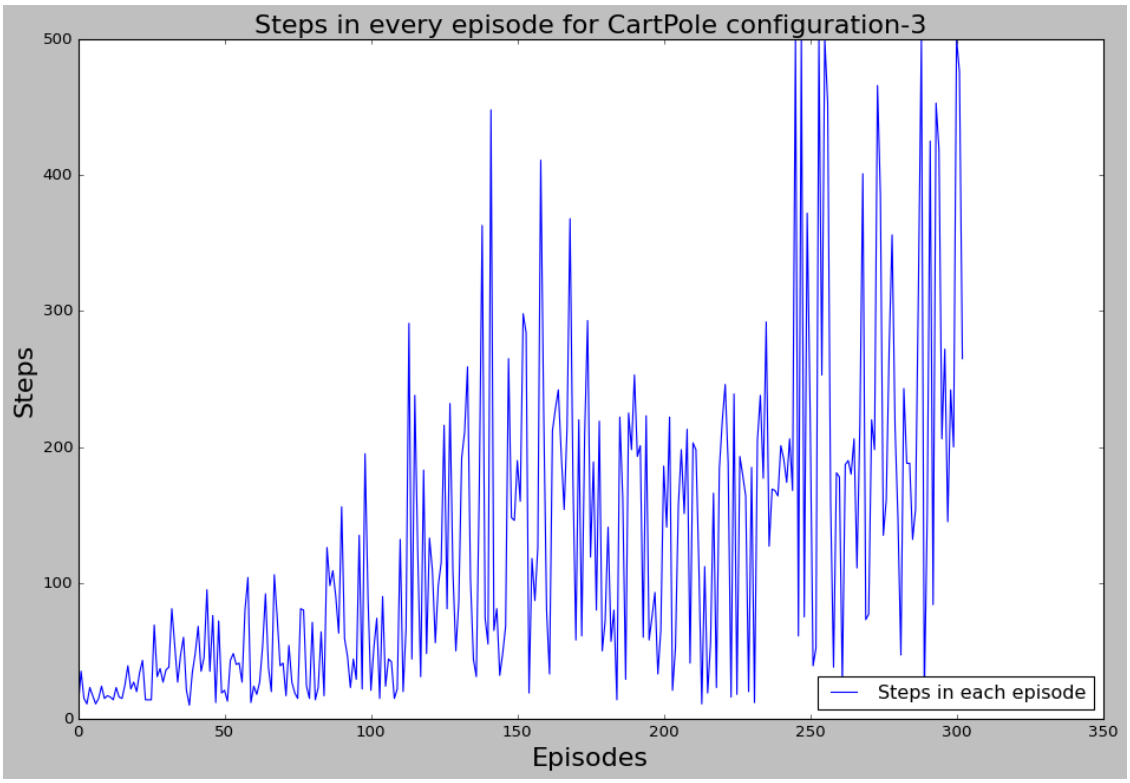
```
    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return self.fc3(x)

begin_time = datetime.datetime.now()

env = gym.make('CartPole-v1')
env.seed(0)
state_shape = env.observation_space.shape[0]
action_shape = env.action_space.n


BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 128         # minibatch size
GAMMA = 0.99            # discount factor
LR = 5e-4               # learning rate
UPDATE_EVERY = 20       # how often to update the network (When Q target is␣
 ↪present)

agent = Agent(state_shape, action_shape,␣
 ↪0,BUFFER_SIZE,BATCH_SIZE,GAMMA,LR,UPDATE_EVERY)


scores_point,scores_running,steps=dqn(req_score=200)


time_taken = datetime.datetime.now() - begin_time

print(time_taken)
```

```
Episode 100     Average Score: 43.71
Episode 200     Average Score: 134.36
Episode 300     Average Score: 193.21

Environment solved in 303 episodes!     Average Score: 200.13
0:02:54.901890
```

[ ]:
```
plt.figure(figsize=(14,9))
plt.plot(np.arange(len(scores_point)),scores_point,label='Point episode␣
 ↪scores',color='yellow')
plt.plot(np.arange(len(scores_running)),scores_running,label='Running average␣
 ↪over 10 episodes',color='red')
plt.xlabel('Episodes',fontsize=20)
plt.ylabel('Rewards',fontsize=20)
```

```
plt.title('Reward curve for CartPole configuration-3',fontsize=20)
plt.legend(loc='lower right')
plt.figure(figsize=(14,9))
plt.plot(np.arange(steps.shape[0]),steps,label='Steps in each episode')
plt.xlabel('Episodes',fontsize=20)
plt.ylabel('Steps',fontsize=20)
plt.title('Steps in every episode for CartPole configuration-3',fontsize=20)
plt.legend(loc='lower right')
plt.show()
```






## 9 *CartPole Configuration – 4*

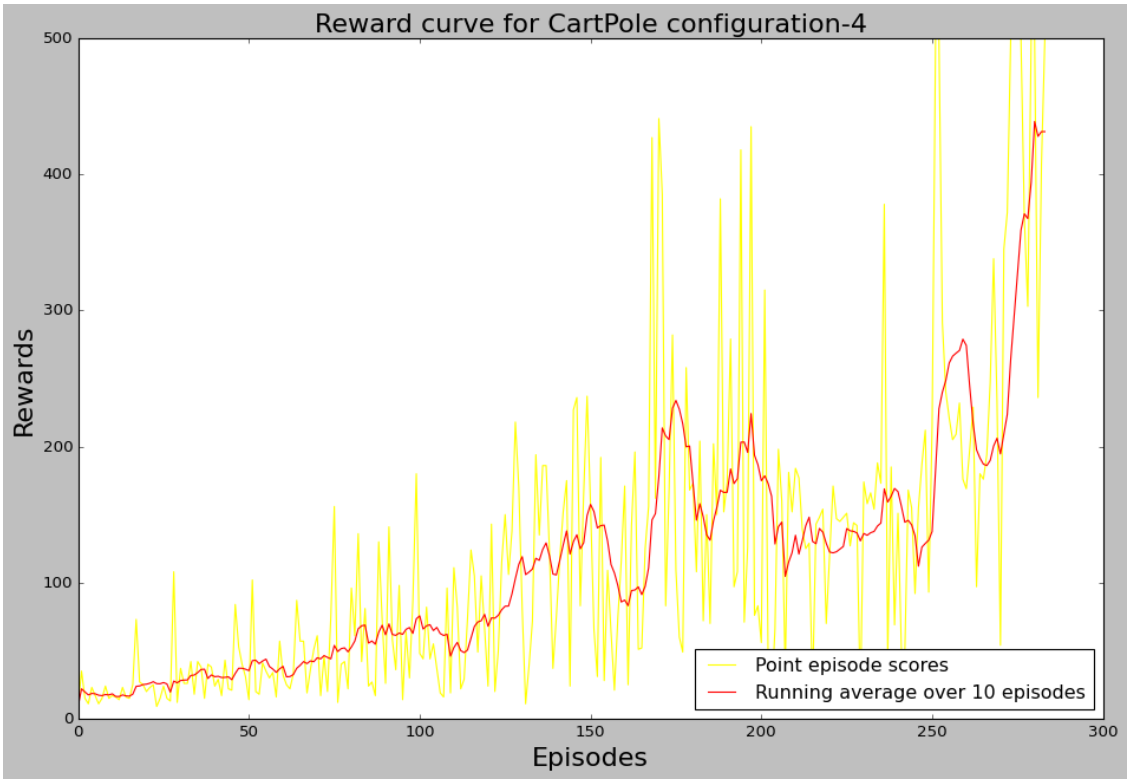
[ ]:
```
class QNetwork1(nn.Module):

    def __init__(self, state_size, action_size, seed, fc1_units=256,␣
 ↪fc2_units=256):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(QNetwork1, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)
```

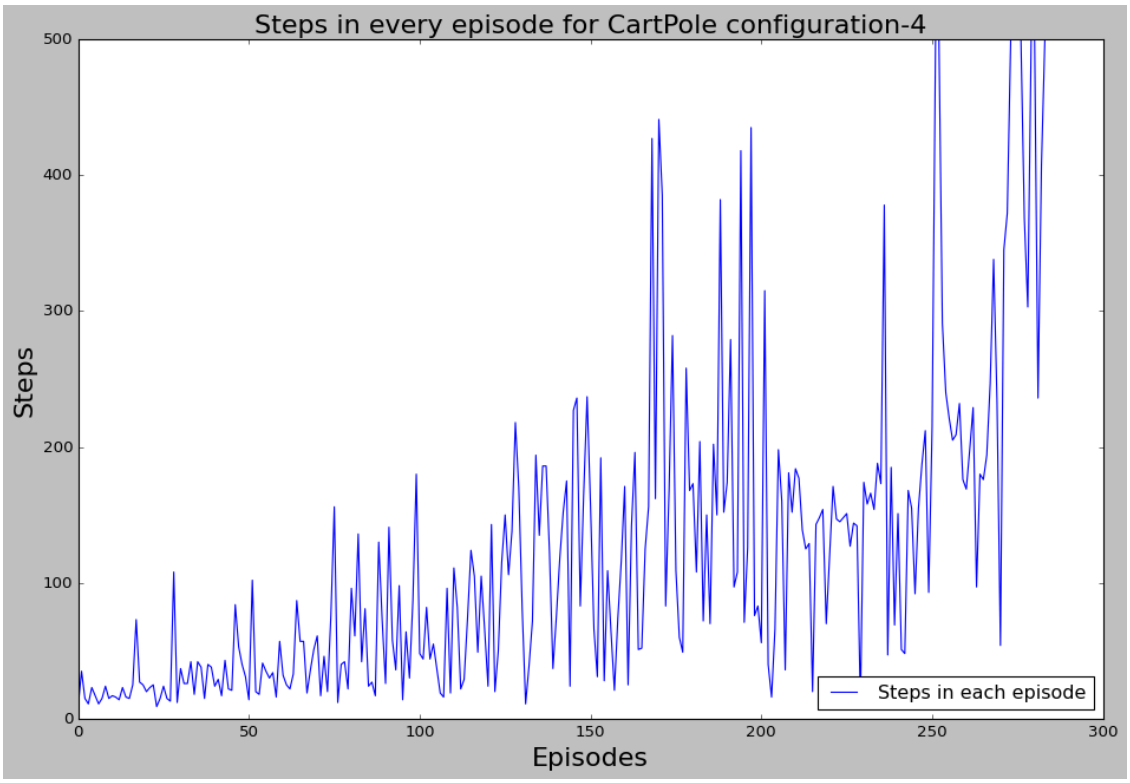
```
    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return self.fc3(x)

begin_time = datetime.datetime.now()

env = gym.make('CartPole-v1')
env.seed(0)
state_shape = env.observation_space.shape[0]
action_shape = env.action_space.n


BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 128        # minibatch size
GAMMA = 0.99            # discount factor
LR = 5e-4               # learning rate
UPDATE_EVERY = 20       # how often to update the network (When Q target is␣
 ↪present)

agent = Agent(state_shape, action_shape,␣
 ↪0,BUFFER_SIZE,BATCH_SIZE,GAMMA,LR,UPDATE_EVERY)


scores_point,scores_running,steps=dqn(req_score=200)


time_taken = datetime.datetime.now() - begin_time

print(time_taken)
```

```
Episode 100     Average Score: 40.75
Episode 200     Average Score: 128.16

Environment solved in 284 episodes!     Average Score: 202.28
0:03:18.777936
```

[ ]:
```
plt.figure(figsize=(14,9))
plt.plot(np.arange(len(scores_point)),scores_point,label='Point episode␣
 ↪scores',color='yellow')
plt.plot(np.arange(len(scores_running)),scores_running,label='Running average␣
 ↪over 10 episodes',color='red')
plt.xlabel('Episodes',fontsize=20)
plt.ylabel('Rewards',fontsize=20)
plt.title('Reward curve for CartPole configuration-4',fontsize=20)
plt.legend(loc='lower right')
```

```
plt.figure(figsize=(14,9))
plt.plot(np.arange(steps.shape[0]),steps,label='Steps in each episode')
plt.xlabel('Episodes',fontsize=20)
plt.ylabel('Steps',fontsize=20)
plt.title('Steps in every episode for CartPole configuration-4',fontsize=20)
plt.legend(loc='lower right')
plt.show()
```






## 10 *CartPole Configuration – 5*

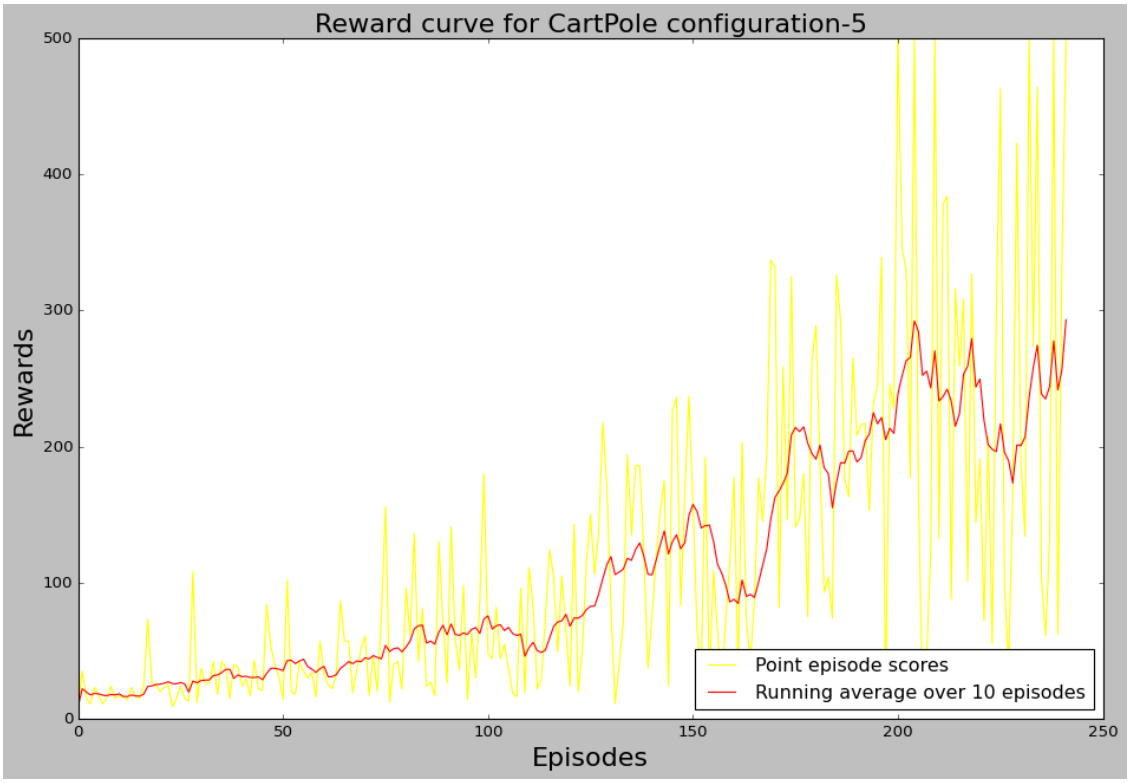
[ ]:
```
#Decreasing the buffer size
class QNetwork1(nn.Module):

    def __init__(self, state_size, action_size, seed, fc1_units=256,␣
 ↪fc2_units=256):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(QNetwork1, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)
```

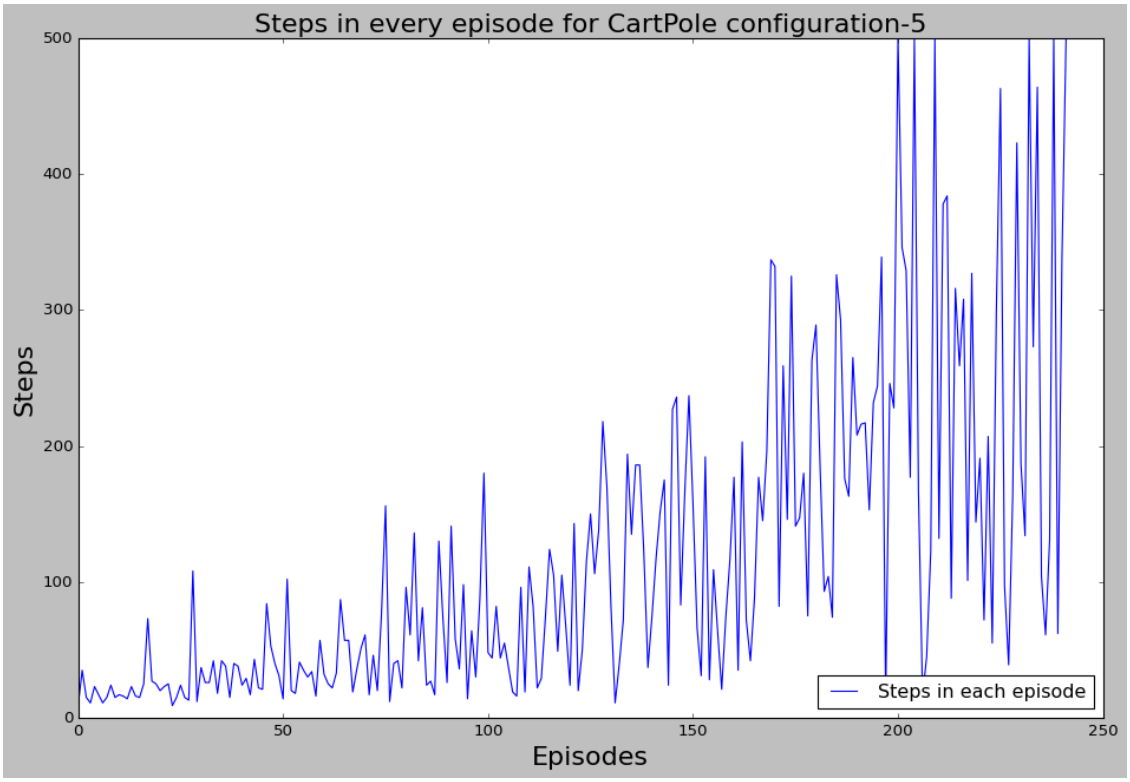
```
    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return self.fc3(x)

begin_time = datetime.datetime.now()

env = gym.make('CartPole-v1')
env.seed(0)
state_shape = env.observation_space.shape[0]
action_shape = env.action_space.n


BUFFER_SIZE = int(1e4)  # replay buffer size
BATCH_SIZE = 128        # minibatch size
GAMMA = 0.99            # discount factor
LR = 5e-4               # learning rate
UPDATE_EVERY = 20       # how often to update the network (When Q target is␣
 ↪present)

agent = Agent(state_shape, action_shape,␣
 ↪0,BUFFER_SIZE,BATCH_SIZE,GAMMA,LR,UPDATE_EVERY)


scores_point,scores_running,steps=dqn(req_score=200)


time_taken = datetime.datetime.now() - begin_time

print(time_taken)
```

```
Episode 100     Average Score: 40.75
Episode 200     Average Score: 132.62

Environment solved in 242 episodes!     Average Score: 200.49
0:02:43.332096
```

[ ]:
```
plt.figure(figsize=(14,9))
plt.plot(np.arange(len(scores_point)),scores_point,label='Point episode␣
 ↪scores',color='yellow')
plt.plot(np.arange(len(scores_running)),scores_running,label='Running average␣
 ↪over 10 episodes',color='red')
plt.xlabel('Episodes',fontsize=20)
plt.ylabel('Rewards',fontsize=20)
plt.title('Reward curve for CartPole configuration-5',fontsize=20)
```

```
plt.legend(loc='lower right')
plt.figure(figsize=(14,9))
plt.plot(np.arange(steps.shape[0]),steps,label='Steps in each episode')
plt.xlabel('Episodes',fontsize=20)
plt.ylabel('Steps',fontsize=20)
plt.title('Steps in every episode for CartPole configuration-5',fontsize=20)
plt.legend(loc='lower right')
plt.show()
```






## 11 *CartPole Configuration – 6*

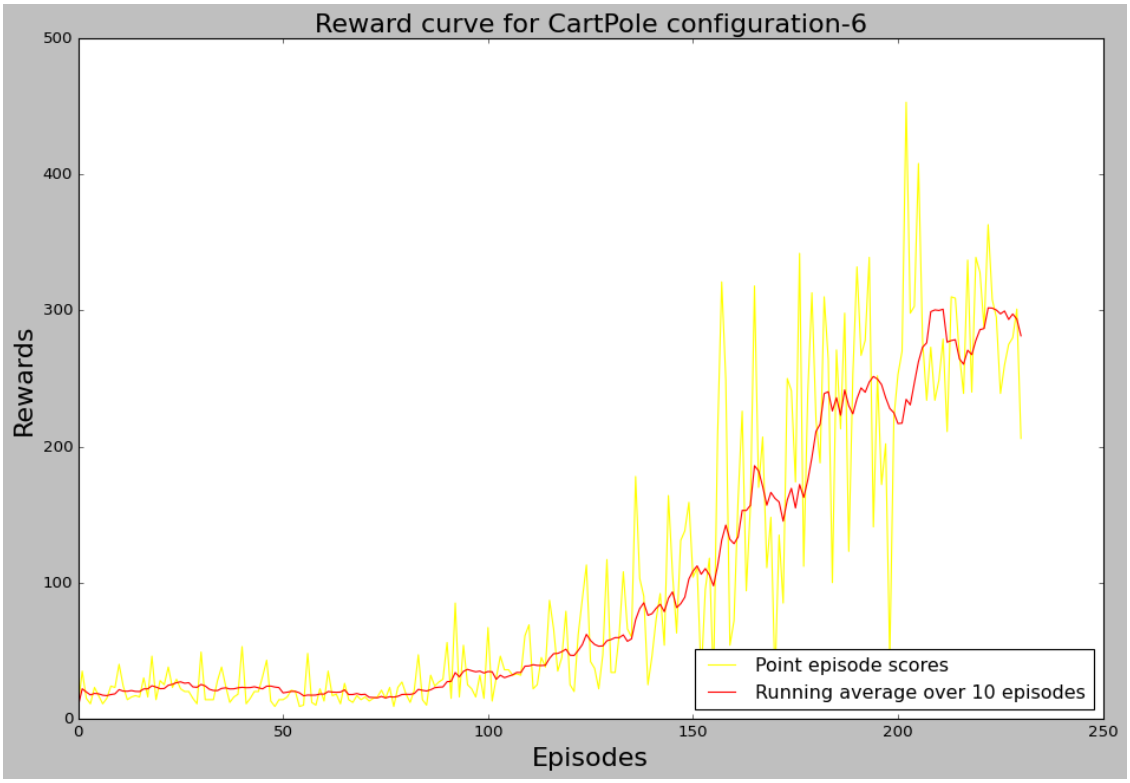
[ ]:
```
#Decreasing the learning rate to 5e-5
class QNetwork1(nn.Module):

    def __init__(self, state_size, action_size, seed, fc1_units=256,␣
 ↪fc2_units=256):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(QNetwork1, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)
```

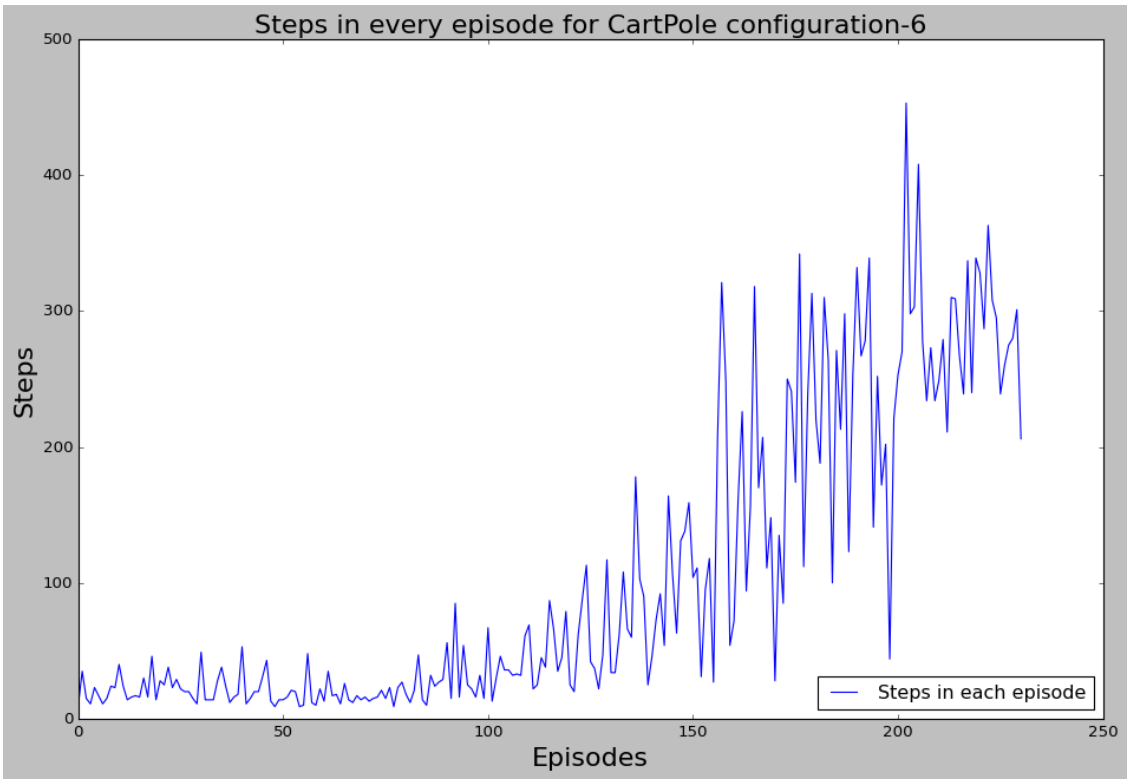
```
    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return self.fc3(x)

begin_time = datetime.datetime.now()

env = gym.make('CartPole-v1')
env.seed(0)
state_shape = env.observation_space.shape[0]
action_shape = env.action_space.n


BUFFER_SIZE = int(1e4)  # replay buffer size
BATCH_SIZE = 128        # minibatch size
GAMMA = 0.99            # discount factor
LR = 5e-5               # learning rate
UPDATE_EVERY = 20       # how often to update the network (When Q target is␣
 ↪present)

agent = Agent(state_shape, action_shape,␣
 ↪0,BUFFER_SIZE,BATCH_SIZE,GAMMA,LR,UPDATE_EVERY)


scores_point,scores_running,steps=dqn(req_score=200)


time_taken = datetime.datetime.now() - begin_time

print(time_taken)
```

```
Episode 100     Average Score: 22.10
Episode 200     Average Score: 126.45

Environment solved in 231 episodes!     Average Score: 200.67
0:02:04.963809
```

[ ]:
```
plt.figure(figsize=(14,9))
plt.plot(np.arange(len(scores_point)),scores_point,label='Point episode␣
 ↪scores',color='yellow')
plt.plot(np.arange(len(scores_running)),scores_running,label='Running average␣
 ↪over 10 episodes',color='red')
plt.xlabel('Episodes',fontsize=20)
plt.ylabel('Rewards',fontsize=20)
plt.title('Reward curve for CartPole configuration-6',fontsize=20)
```

```
plt.legend(loc='lower right')
plt.figure(figsize=(14,9))
plt.plot(np.arange(steps.shape[0]),steps,label='Steps in each episode')
plt.xlabel('Episodes',fontsize=20)
plt.ylabel('Steps',fontsize=20)
plt.title('Steps in every episode for CartPole configuration-6',fontsize=20)
plt.legend(loc='lower right')
plt.show()
```






## 12 $Mountaincar\ Configuration - 1 (Baseline)$

[ ]:
```
class QNetwork1(nn.Module):

    def __init__(self, state_size, action_size, seed, fc1_units=128,␣
 ↪fc2_units=64):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(QNetwork1, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)
```

```
    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return self.fc3(x)

begin_time = datetime.datetime.now()

env = gym.make('MountainCar-v0')
env.seed(0)
state_shape = env.observation_space.shape[0]
action_shape = env.action_space.n


BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 64         # minibatch size
GAMMA = 0.99            # discount factor
LR = 5e-4               # learning rate
UPDATE_EVERY = 20       # how often to update the network (When Q target is␣
 ↪present)

agent = Agent(state_shape, action_shape,␣
 ↪0,BUFFER_SIZE,BATCH_SIZE,GAMMA,LR,UPDATE_EVERY)


scores_point,scores_running,steps=dqn(req_score=-160)


time_taken = datetime.datetime.now() - begin_time

print(time_taken)
```

```
Episode 100     Average Score: -200.00
Episode 200     Average Score: -200.00
Episode 300     Average Score: -200.00
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
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-20-14019e30d7c5> in <module>
     40
     41
---> 42 scores_point,scores_running,steps=dqn(req_score=-80)
```

```
     43
     44

<ipython-input-16-6d65e37fbf20> in dqn(n_episodes, max_t, eps_start, eps_end,␣
 ↪eps_decay, req_score)
     25             action = agent.act(state, eps)
     26             next_state, reward, done, _ = env.step(action)
---> 27             agent.step(state, action, reward, next_state, done)
     28             state = next_state
     29             score += reward

<ipython-input-5-72faf8f30370> in step(self, state, action, reward, next_state,␣
 ↪done)
     32         if len(self.memory) >= self.batch_size:
     33             experiences = self.memory.sample()
---> 34             self.learn(experiences)
     35
     36         """ +Q TARGETS PRESENT """

<ipython-input-5-72faf8f30370> in learn(self, experiences)
     79             param.grad.data.clamp_(-1, 1)
     80
---> 81         self.optimizer.step()

/usr/local/lib/python3.9/dist-packages/torch/optim/optimizer.py in␣
 ↪wrapper(*args, **kwargs)
    138                 profile_name = "Optimizer.step#{}.step".format(obj.
 ↪__class__.__name__)
    139                 with torch.autograd.profiler.
 ↪record_function(profile_name):
--> 140                     out = func(*args, **kwargs)
    141                     obj._optimizer_step_code()
    142                     return out

/usr/local/lib/python3.9/dist-packages/torch/optim/optimizer.py in␣
 ↪_use_grad(self, *args, **kwargs)
     21         try:
     22             torch.set_grad_enabled(self.defaults['differentiable'])
---> 23             ret = func(self, *args, **kwargs)
     24         finally:
     25             torch.set_grad_enabled(prev_grad)

/usr/local/lib/python3.9/dist-packages/torch/optim/adam.py in step(self,␣
 ↪closure, grad_scaler)
    232                     state_steps.append(state['step'])
    233
```

```
--> 234             adam(params_with_grad,

    235                  grads,
    236                  exp_avgs,

/usr/local/lib/python3.9/dist-packages/torch/optim/adam.py in adam(params,␣
 ↪grads, exp_avgs, exp_avg_sqs, max_exp_avg_sqs, state_steps, foreach,␣
 ↪capturable, differentiable, fused, grad_scale, found_inf, amsgrad, beta1,␣
 ↪beta2, lr, weight_decay, eps, maximize)
    298         func = _single_tensor_adam
    299
--> 300     func(params,

    301          grads,
    302          exp_avgs,

/usr/local/lib/python3.9/dist-packages/torch/optim/adam.py in␣
 ↪_single_tensor_adam(params, grads, exp_avgs, exp_avg_sqs, max_exp_avg_sqs,␣
 ↪state_steps, grad_scale, found_inf, amsgrad, beta1, beta2, lr, weight_decay,␣
 ↪eps, maximize, capturable, differentiable)
    408                 denom = (max_exp_avg_sqs[i].sqrt() /␣
 ↪bias_correction2_sqrt).add_(eps)
    409             else:
--> 410                 denom = (exp_avg_sq.sqrt() / bias_correction2_sqrt).
 ↪add_(eps)
    411
    412             param.addcdiv_(exp_avg, denom, value=-step_size)

KeyboardInterrupt:
```

## 13 *Mountaincar Configuration – 2*

[ ]:
```
#Increasing lr to 1e-1
class QNetwork1(nn.Module):

    def __init__(self, state_size, action_size, seed, fc1_units=128,␣
 ↪fc2_units=64):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(QNetwork1, self).__init__()
        self.seed = torch.manual_seed(seed)
```

```
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)

    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return self.fc3(x)

begin_time = datetime.datetime.now()

env = gym.make('MountainCar-v0')
env.seed(0)
state_shape = env.observation_space.shape[0]
action_shape = env.action_space.n


BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 64         # minibatch size
GAMMA = 0.99            # discount factor
LR = 1e-1              # learning rate
UPDATE_EVERY = 20       # how often to update the network (When Q target is␣
 ↪present)

agent = Agent(state_shape, action_shape,␣
 ↪0,BUFFER_SIZE,BATCH_SIZE,GAMMA,LR,UPDATE_EVERY)


scores_point,scores_running,steps=dqn(req_score=-160)


time_taken = datetime.datetime.now() - begin_time

print(time_taken)
```

```
Episode 100     Average Score: -200.00
Episode 200     Average Score: -200.00
Episode 300     Average Score: -200.00
Episode 400     Average Score: -200.00
Episode 500     Average Score: -198.73
Episode 600     Average Score: -199.32
Episode 700     Average Score: -196.79
Episode 800     Average Score: -197.20
Episode 900     Average Score: -198.18
Episode 1000    Average Score: -195.52
Episode 1100    Average Score: -197.40
```

```
Episode 1200    Average Score: -198.44
Episode 1300    Average Score: -197.92
Episode 1400    Average Score: -197.97
Episode 1500    Average Score: -199.54
Episode 1600    Average Score: -199.20
Episode 1700    Average Score: -198.42
Episode 1800    Average Score: -197.58
Episode 1900    Average Score: -199.23
Episode 2000    Average Score: -198.39
Episode 2100    Average Score: -197.52
Episode 2200    Average Score: -198.82
Episode 2300    Average Score: -195.60
Episode 2400    Average Score: -198.94
Episode 2500    Average Score: -197.70
Episode 2600    Average Score: -196.97
Episode 2700    Average Score: -198.00
Episode 2800    Average Score: -196.66
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-21-2f837d5f7833> in <module>
     40
     41
---> 42 scores_point,scores_running,steps=dqn(req_score=-80)
     43
     44

<ipython-input-16-6d65e37fbf20> in dqn(n_episodes, max_t, eps_start, eps_end,␣
 ↪eps_decay, req_score)
     24         for t in range(max_t):
     25             action = agent.act(state, eps)
---> 26             next_state, reward, done, _ = env.step(action)
     27             agent.step(state, action, reward, next_state, done)
     28             state = next_state

/usr/local/lib/python3.9/dist-packages/gym/wrappers/time_limit.py in step(self,␣
 ↪action)
     58         """
     59         observation, reward, terminated, truncated, info =␣
 ↪step_api_compatibility(
---> 60             self.env.step(action),
     61             True,
     62         )

/usr/local/lib/python3.9/dist-packages/gym/wrappers/order_enforcing.py in␣
 ↪step(self, action)
     35         if not self._has_reset:
```

```
     36             raise ResetNeeded("Cannot call env.step() before calling env.
 ↪reset()")
---> 37         return self.env.step(action)
     38
     39     def reset(self, **kwargs):

/usr/local/lib/python3.9/dist-packages/gym/wrappers/step_api_compatibility.py in
 ↪step(self, action)
     50             (observation, reward, terminated, truncated, info) or␣
 ↪(observation, reward, done, info)
     51         """
---> 52         step_returns = self.env.step(action)
     53         if self.new_step_api:
     54             return step_to_new_api(step_returns)

/usr/local/lib/python3.9/dist-packages/gym/wrappers/env_checker.py in step(self
 ↪action)
     37             return env_step_passive_checker(self.env, action)
     38         else:
---> 39             return self.env.step(action)
     40
     41     def reset(self, **kwargs):

/usr/local/lib/python3.9/dist-packages/gym/envs/classic_control/mountain_car.py
 ↪in step(self, action)
    134         position, velocity = self.state
    135         velocity += (action - 1) * self.force + math.cos(3 * position)
 ↪(-self.gravity)
--> 136         velocity = np.clip(velocity, -self.max_speed, self.max_speed)
    137         position += velocity
    138         position = np.clip(position, self.min_position, self.
 ↪max_position)

/usr/local/lib/python3.9/dist-packages/numpy/core/overrides.py in clip(*args,
 ↪**kwargs)

/usr/local/lib/python3.9/dist-packages/numpy/core/fromnumeric.py in clip(a,
 ↪a_min, a_max, out, **kwargs)
   2150
   2151     """
-> 2152     return _wrapfunc(a, 'clip', a_min, a_max, out=out, **kwargs)
   2153
   2154

/usr/local/lib/python3.9/dist-packages/numpy/core/fromnumeric.py in
 ↪_wrapfunc(obj, method, *args, **kwds)
     55
     56     try:
```

```
---> 57         return bound(*args, **kwds)
     58     except TypeError:
     59         # A TypeError occurs if the object does have such a method in i s

/usr/local/lib/python3.9/dist-packages/numpy/core/_methods.py in _clip(a, min,␣
 ↪max, out, casting, **kwargs)
    123         return ufunc(*args, out=out, casting="unsafe", **kwargs)
    124
--> 125 def _clip(a, min=None, max=None, out=None, *, casting=None, **kwargs):
    126     if min is None and max is None:
    127         raise ValueError("One of max or min must be given")

KeyboardInterrupt:
```

## 14 *Mountaincar Configuration – 3*

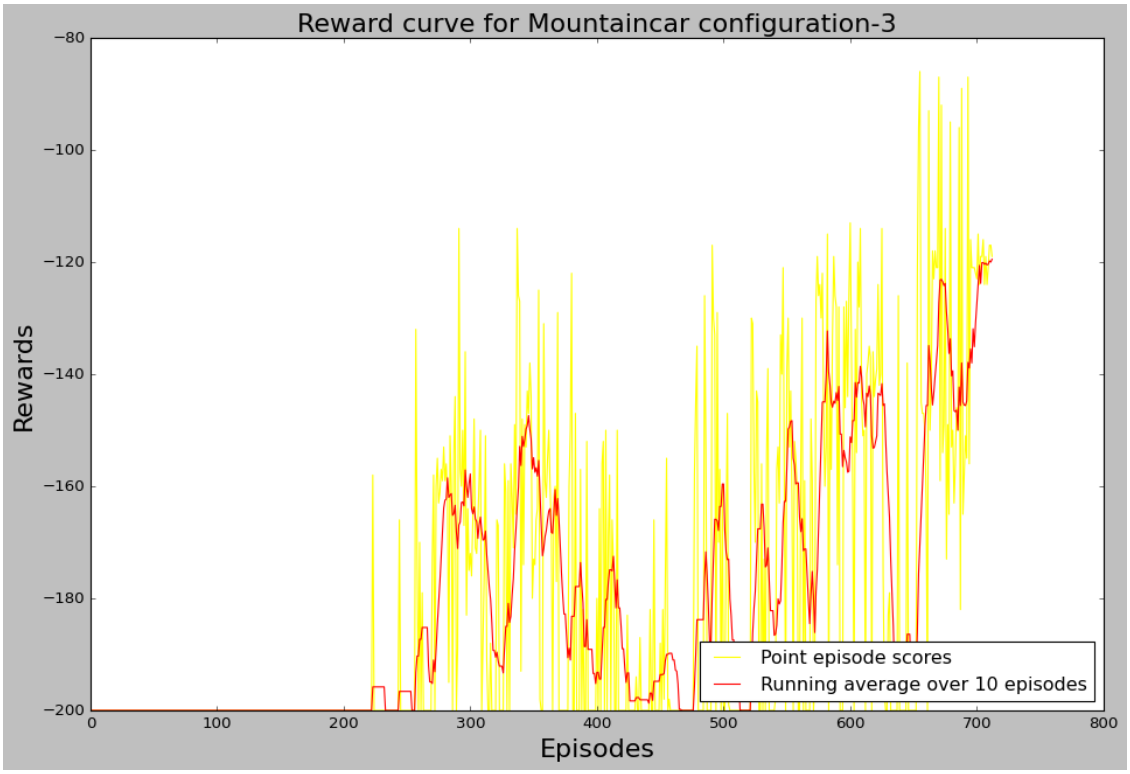
[ ]:
```
# Decreasing the lr to 1e-2 and increasing the buffer size to 1e6
class QNetwork1(nn.Module):

    def __init__(self, state_size, action_size, seed, fc1_units=128,␣
 ↪fc2_units=64):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(QNetwork1, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)

    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return self.fc3(x)

begin_time = datetime.datetime.now()

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

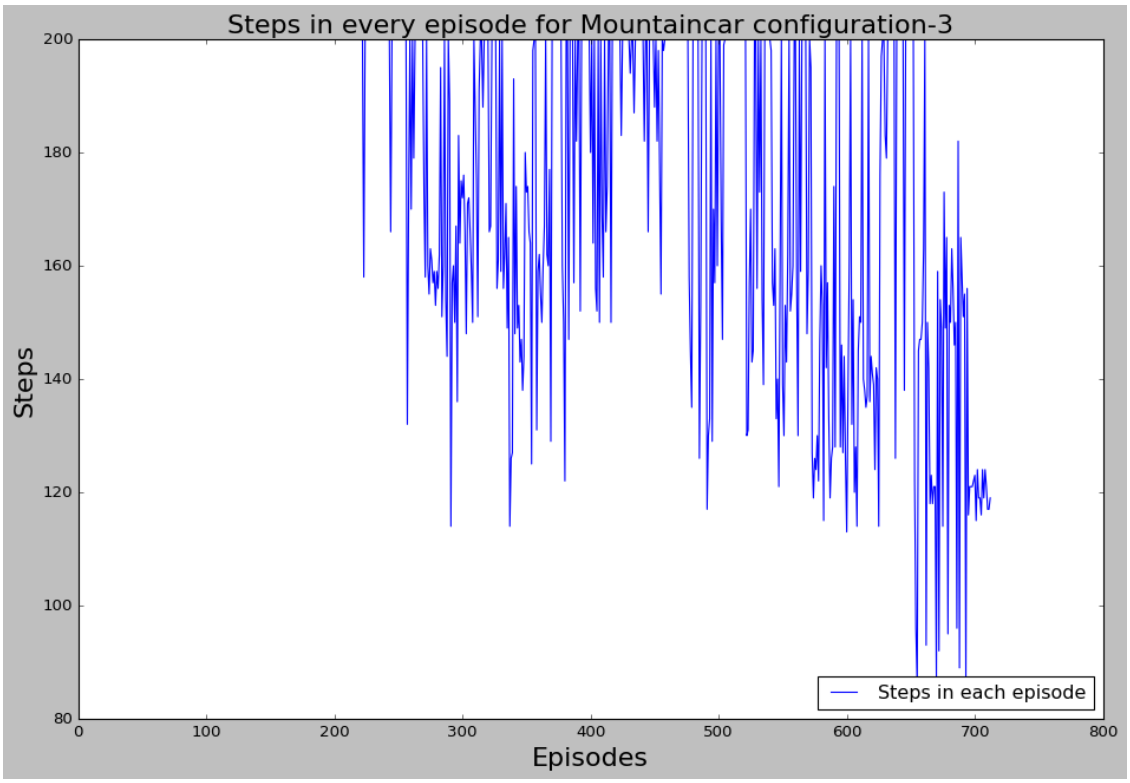
```
state_shape = env.observation_space.shape[0]
action_shape = env.action_space.n


BUFFER_SIZE = int(1e6)  # replay buffer size
BATCH_SIZE = 64         # minibatch size
GAMMA = 0.99            # discount factor
LR = 1e-2               # learning rate
UPDATE_EVERY = 20       # how often to update the network (When Q target is␣
 ↪present)

agent = Agent(state_shape, action_shape,␣
 ↪0,BUFFER_SIZE,BATCH_SIZE,GAMMA,LR,UPDATE_EVERY)


scores_point,scores_running,steps=dqn(req_score=-150)


time_taken = datetime.datetime.now() - begin_time

print(time_taken)
```

```
Episode 100     Average Score: -200.00
Episode 200     Average Score: -200.00
Episode 300     Average Score: -186.91
Episode 400     Average Score: -175.03
Episode 500     Average Score: -187.42
Episode 600     Average Score: -168.81
Episode 700     Average Score: -152.72

Environment solved in 713 episodes!     Average Score: -149.23
0:06:40.827713
```

[ ]:
```
plt.figure(figsize=(14,9))
plt.plot(np.arange(len(scores_point)),scores_point,label='Point episode␣
 ↪scores',color='yellow')
plt.plot(np.arange(len(scores_running)),scores_running,label='Running average␣
 ↪over 10 episodes',color='red')
plt.xlabel('Episodes',fontsize=20)
plt.ylabel('Rewards',fontsize=20)
plt.title('Reward curve for Mountaincar configuration-3',fontsize=20)
plt.legend(loc='lower right')
plt.figure(figsize=(14,9))
plt.plot(np.arange(steps.shape[0]),steps,label='Steps in each episode')
plt.xlabel('Episodes',fontsize=20)
plt.ylabel('Steps',fontsize=20)
plt.title('Steps in every episode for Mountaincar configuration-3',fontsize=20)
```

```
plt.legend(loc='lower right')
plt.show()
```






## 15 *Mountaincar*

Other Configurations which didn't give much improvement in results

[ ]:
```
#Increasing the lr to 1e-1 with replay buffer 1e6
class QNetwork1(nn.Module):

    def __init__(self, state_size, action_size, seed, fc1_units=128,␣
 ↪fc2_units=64):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(QNetwork1, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
```

```
        self.fc3 = nn.Linear(fc2_units, action_size)

    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return self.fc3(x)

begin_time = datetime.datetime.now()

env = gym.make('MountainCar-v0')
env.seed(0)
state_shape = env.observation_space.shape[0]
action_shape = env.action_space.n


BUFFER_SIZE = int(1e6)  # replay buffer size
BATCH_SIZE = 64         # minibatch size
GAMMA = 0.99            # discount factor
LR = 1e-1               # learning rate
UPDATE_EVERY = 20       # how often to update the network (When Q target is␣
 ↪present)

agent = Agent(state_shape, action_shape,␣
 ↪0,BUFFER_SIZE,BATCH_SIZE,GAMMA,LR,UPDATE_EVERY)


scores_point,scores_running,steps=dqn(req_score=-150)


time_taken = datetime.datetime.now() - begin_time

print(time_taken)
```

```
Episode 100     Average Score: -200.00
Episode 200     Average Score: -200.00
Episode 300     Average Score: -200.00
Episode 400     Average Score: -200.00
Episode 500     Average Score: -198.73
Episode 600     Average Score: -198.09
Episode 700     Average Score: -198.01
Episode 800     Average Score: -198.71
Episode 900     Average Score: -199.04
Episode 1000    Average Score: -197.27
Episode 1100    Average Score: -196.71
Episode 1200    Average Score: -197.17
Episode 1300    Average Score: -196.81
```

```
Episode 1400    Average Score: -198.15
Episode 1500    Average Score: -197.90
Episode 1600    Average Score: -199.48
Episode 1700    Average Score: -195.21
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-16-cd5553364ea4> in <module>
     40
     41
---> 42 scores_point,scores_running,steps=dqn(req_score=-150)
     43
     44

<ipython-input-6-5184d6847401> in dqn(n_episodes, max_t, eps_start, eps_end,␣
 ↪eps_decay, req_score)
     25             action = agent.act(state, eps)
     26             next_state, reward, done, _ = env.step(action)
---> 27             agent.step(state, action, reward, next_state, done)
     28             state = next_state
     29             score += reward

<ipython-input-5-72faf8f30370> in step(self, state, action, reward, next_state,
 ↪done)
     32         if len(self.memory) >= self.batch_size:
     33             experiences = self.memory.sample()
---> 34             self.learn(experiences)
     35
     36         """ +Q TARGETS PRESENT """

<ipython-input-5-72faf8f30370> in learn(self, experiences)
     79             param.grad.data.clamp_(-1, 1)
     80
---> 81         self.optimizer.step()

/usr/local/lib/python3.9/dist-packages/torch/optim/optimizer.py in␣
 ↪wrapper(*args, **kwargs)
    138                 profile_name = "Optimizer.step#{}.step".format(obj.
 ↪__class__.__name__)
    139                 with torch.autograd.profiler.
 ↪record_function(profile_name):
--> 140                     out = func(*args, **kwargs)
    141                     obj._optimizer_step_code()
    142                     return out

/usr/local/lib/python3.9/dist-packages/torch/optim/optimizer.py in␣
 ↪_use_grad(self, *args, **kwargs)
```

```
     21         try:
     22             torch.set_grad_enabled(self.defaults['differentiable'])
---> 23             ret = func(self, *args, **kwargs)
     24         finally:
     25             torch.set_grad_enabled(prev_grad)

/usr/local/lib/python3.9/dist-packages/torch/optim/adam.py in step(self,␣
 ↪closure, grad_scaler)
    232                     state_steps.append(state['step'])
    233
--> 234             adam(params_with_grad,

    235                  grads,
    236                  exp_avgs,

/usr/local/lib/python3.9/dist-packages/torch/optim/adam.py in adam(params,␣
 ↪grads, exp_avgs, exp_avg_sqs, max_exp_avg_sqs, state_steps, foreach,␣
 ↪capturable, differentiable, fused, grad_scale, found_inf, amsgrad, beta1,␣
 ↪beta2, lr, weight_decay, eps, maximize)
    298         func = _single_tensor_adam
    299
--> 300     func(params,

    301          grads,
    302          exp_avgs,

/usr/local/lib/python3.9/dist-packages/torch/optim/adam.py in␣
 ↪_single_tensor_adam(params, grads, exp_avgs, exp_avg_sqs, max_exp_avg_sqs,␣
 ↪state_steps, grad_scale, found_inf, amsgrad, beta1, beta2, lr, weight_decay,␣
 ↪eps, maximize, capturable, differentiable)
    362         # Decay the first and second moment running average coefficient
    363         exp_avg.mul_(beta1).add_(grad, alpha=1 - beta1)
--> 364         exp_avg_sq.mul_(beta2).addcmul_(grad, grad.conj(), value=1 -␣
 ↪beta2)
    365
    366         if capturable or differentiable:

KeyboardInterrupt:
```

[ ]:
```
#increasing the buffer size futher to 1e7
class QNetwork1(nn.Module):

    def __init__(self, state_size, action_size, seed, fc1_units=128,␣
 ↪fc2_units=64):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
```

```
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(QNetwork1, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)

    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return self.fc3(x)

begin_time = datetime.datetime.now()

env = gym.make('MountainCar-v0')
env.seed(0)
state_shape = env.observation_space.shape[0]
action_shape = env.action_space.n


BUFFER_SIZE = int(1e7)  # replay buffer size
BATCH_SIZE = 64         # minibatch size
GAMMA = 0.99            # discount factor
LR = 1e-2               # learning rate
UPDATE_EVERY = 20       # how often to update the network (When Q target is␣
 ↪present)

agent = Agent(state_shape, action_shape,␣
 ↪0,BUFFER_SIZE,BATCH_SIZE,GAMMA,LR,UPDATE_EVERY)


scores_point,scores_running,steps=dqn(req_score=-150)


time_taken = datetime.datetime.now() - begin_time

print(time_taken)
```

```
Episode 100     Average Score: -200.00
Episode 200     Average Score: -200.00
Episode 300     Average Score: -186.91
Episode 400     Average Score: -175.03
Episode 500     Average Score: -187.42
```

```
Episode 600     Average Score: -168.81
Episode 700     Average Score: -152.72

Environment solved in 713 episodes!     Average Score: -149.23
0:08:41.619336
```

[ ]:
```python
# Changing the lr to 1e-3 with buffer size 1e7
class QNetwork1(nn.Module):

    def __init__(self, state_size, action_size, seed, fc1_units=128,␣
 ↪fc2_units=64):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(QNetwork1, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)

    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return self.fc3(x)

begin_time = datetime.datetime.now()

env = gym.make('MountainCar-v0')
env.seed(0)
state_shape = env.observation_space.shape[0]
action_shape = env.action_space.n


BUFFER_SIZE = int(1e7)  # replay buffer size
BATCH_SIZE = 64         # minibatch size
GAMMA = 0.99            # discount factor
LR = 1e-3               # learning rate
UPDATE_EVERY = 20       # how often to update the network (When Q target is␣
 ↪present)
```

```
agent = Agent(state_shape, action_shape,␣
 ↪0,BUFFER_SIZE,BATCH_SIZE,GAMMA,LR,UPDATE_EVERY)


scores_point,scores_running,steps=dqn(req_score=-150)


time_taken = datetime.datetime.now() - begin_time

print(time_taken)
```

```
/usr/local/lib/python3.9/dist-packages/gym/core.py:317: DeprecationWarning:
WARN: Initializing wrapper in old step API which returns one bool instead
of two. It is recommended to set `new_step_api=True` to use new step API. This
will be the default behaviour in future.
  deprecation(
/usr/local/lib/python3.9/dist-
packages/gym/wrappers/step_api_compatibility.py:39: DeprecationWarning:
WARN: Initializing environment in old step API which returns one bool
instead of two. It is recommended to set `new_step_api=True` to use new step
API. This will be the default behaviour in future.
  deprecation(
/usr/local/lib/python3.9/dist-packages/gym/core.py:256: DeprecationWarning:
WARN: Function `env.seed(seed)` is marked as deprecated and will be removed
in the future. Please use `env.reset(seed=seed)` instead.
  deprecation(

Episode 100     Average Score: -200.00
Episode 200     Average Score: -200.00
Episode 300     Average Score: -200.00
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
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-8-dc05cffa57d0> in <module>
     40
     41
---> 42 scores_point,scores_running,steps=dqn(req_score=-150)
     43
```

```
     44

<ipython-input-7-6d65e37fbf20> in dqn(n_episodes, max_t, eps_start, eps_end,␣
 ↪eps_decay, req_score)
     25             action = agent.act(state, eps)
     26             next_state, reward, done, _ = env.step(action)
---> 27             agent.step(state, action, reward, next_state, done)
     28             state = next_state
     29             score += reward

<ipython-input-6-72faf8f30370> in step(self, state, action, reward, next_state,␣
 ↪done)
     31         ''' If enough samples are available in memory, get random subset␣
 ↪and learn '''
     32         if len(self.memory) >= self.batch_size:
---> 33             experiences = self.memory.sample()
     34             self.learn(experiences)
     35

<ipython-input-5-b93058745ec0> in sample(self)
     32     def sample(self):
     33         """Randomly sample a batch of experiences from memory."""
---> 34         experiences = random.sample(self.memory, k=self.batch_size)
     35
     36         states = torch.from_numpy(np.vstack([e.state for e in␣
 ↪experiences if e is not None])).float().to(device)

/usr/lib/python3.9/random.py in sample(self, population, k, counts)
    468                     j = randbelow(n)
    469                 selected_add(j)
--> 470                 result[i] = population[j]
    471         return result
    472

KeyboardInterrupt:
```

[ ]:
```
#increasing the size of the NN network
class QNetwork1(nn.Module):

    def __init__(self, state_size, action_size, seed, fc1_units=128,␣
 ↪fc2_units=128):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
```

```
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(QNetwork1, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)

    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return self.fc3(x)

begin_time = datetime.datetime.now()

env = gym.make('MountainCar-v0')
env.seed(0)
state_shape = env.observation_space.shape[0]
action_shape = env.action_space.n


BUFFER_SIZE = int(1e6)  # replay buffer size
BATCH_SIZE = 64         # minibatch size
GAMMA = 0.99            # discount factor
LR = 1e-2               # learning rate
UPDATE_EVERY = 20       # how often to update the network (When Q target is␣
 ↪present)

agent = Agent(state_shape, action_shape,␣
 ↪0,BUFFER_SIZE,BATCH_SIZE,GAMMA,LR,UPDATE_EVERY)


scores_point,scores_running,steps=dqn(req_score=-150)


time_taken = datetime.datetime.now() - begin_time

print(time_taken)
```

```
/usr/local/lib/python3.9/dist-packages/gym/core.py:317: DeprecationWarning:
WARN: Initializing wrapper in old step API which returns one bool instead
of two. It is recommended to set `new_step_api=True` to use new step API. This
will be the default behaviour in future.
  deprecation(
```

```
/usr/local/lib/python3.9/dist-
packages/gym/wrappers/step_api_compatibility.py:39: DeprecationWarning:
WARN: Initializing environment in old step API which returns one bool

instead of two. It is recommended to set `new_step_api=True` to use new step

API. This will be the default behaviour in future.
  deprecation(
/usr/local/lib/python3.9/dist-packages/gym/core.py:256: DeprecationWarning:
WARN: Function `env.seed(seed)` is marked as deprecated and will be removed

in the future. Please use `env.reset(seed=seed)` instead.
  deprecation(

Episode 100     Average Score: -200.00
Episode 200     Average Score: -200.00
Episode 300     Average Score: -199.48
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: -199.16
Episode 900     Average Score: -197.95
Episode 1000    Average Score: -200.00
Episode 1100    Average Score: -194.59
Episode 1200    Average Score: -198.68
Episode 1300    Average Score: -198.79
Episode 1400    Average Score: -195.61
Episode 1500    Average Score: -196.60
Episode 1600    Average Score: -197.65
Episode 1700    Average Score: -198.36
Episode 1800    Average Score: -196.39
Episode 1900    Average Score: -196.43
Episode 2000    Average Score: -197.26
Episode 2100    Average Score: -196.38
Episode 2200    Average Score: -197.97
Episode 2300    Average Score: -198.37
Episode 2400    Average Score: -197.98
Episode 2500    Average Score: -197.66
Episode 2600    Average Score: -195.44
Episode 2700    Average Score: -197.60
```