RLPA2_actor_critic_cartpole

March 31, 2022

[2]:
```
'''
Installing packages for rendering the game on Colab
'''

!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
```

```
Requirement already satisfied: setuptools in /usr/local/lib/python3.7/dist-
packages (61.2.0)
```

[3]:
```
!pip install tensorflow-gpu
```

```
Collecting tensorflow-gpu
  Downloading tensorflow_gpu-2.8.0-cp37-cp37m-manylinux2010_x86_64.whl (497.5
MB)
     |                 | 497.5 MB 22 kB/s
Requirement already satisfied: opt-einsum>=2.3.2 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (3.3.0)
Requirement already satisfied: numpy>=1.20 in /usr/local/lib/python3.7/dist-
packages (from tensorflow-gpu) (1.21.5)
Requirement already satisfied: wrapt>=1.11.0 in /usr/local/lib/python3.7/dist-
packages (from tensorflow-gpu) (1.14.0)
Requirement already satisfied: protobuf>=3.9.2 in /usr/local/lib/python3.7/dist-
packages (from tensorflow-gpu) (3.17.3)
Requirement already satisfied: h5py>=2.9.0 in /usr/local/lib/python3.7/dist-
packages (from tensorflow-gpu) (3.1.0)
Requirement already satisfied: tensorflow-io-gcs-filesystem>=0.23.1 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (0.24.0)
Requirement already satisfied: absl-py>=0.4.0 in /usr/local/lib/python3.7/dist-
packages (from tensorflow-gpu) (1.0.0)
Requirement already satisfied: gast>=0.2.1 in /usr/local/lib/python3.7/dist-
packages (from tensorflow-gpu) (0.5.3)
```

```
Requirement already satisfied: google-pasta>=0.1.1 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (0.2.0)
Requirement already satisfied: termcolor>=1.1.0 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (1.1.0)
Collecting tf-estimator-nightly==2.8.0.dev2021122109
  Downloading tf_estimator_nightly-2.8.0.dev2021122109-py2.py3-none-any.whl (462
kB)
     |                  | 462 kB 43.9 MB/s
Requirement already satisfied: six>=1.12.0 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (1.15.0)
Requirement already satisfied: typing-extensions>=3.6.6 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (3.10.0.2)
Requirement already satisfied: tensorboard<2.9,>=2.8 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (2.8.0)
Requirement already satisfied: keras<2.9,>=2.8.0rc0 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (2.8.0)
Requirement already satisfied: flatbuffers>=1.12 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (2.0)
Requirement already satisfied: libclang>=9.0.1 in /usr/local/lib/python3.7/dist-
packages (from tensorflow-gpu) (13.0.0)
Requirement already satisfied: grpcio<2.0,>=1.24.3 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (1.44.0)
Requirement already satisfied: setuptools in /usr/local/lib/python3.7/dist-
packages (from tensorflow-gpu) (61.2.0)
Requirement already satisfied: astunparse>=1.6.0 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (1.6.3)
Requirement already satisfied: keras-preprocessing>=1.1.1 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (1.1.2)
Requirement already satisfied: wheel<1.0,>=0.23.0 in
/usr/local/lib/python3.7/dist-packages (from astunparse>=1.6.0->tensorflow-gpu)
(0.37.1)
Requirement already satisfied: cached-property in /usr/local/lib/python3.7/dist-
packages (from h5py>=2.9.0->tensorflow-gpu) (1.5.2)
Requirement already satisfied: google-auth-oauthlib<0.5,>=0.4.1 in
/usr/local/lib/python3.7/dist-packages (from tensorboard<2.9,>=2.8->tensorflow-
gpu) (0.4.6)
Requirement already satisfied: tensorboard-plugin-wit>=1.6.0 in
/usr/local/lib/python3.7/dist-packages (from tensorboard<2.9,>=2.8->tensorflow-
gpu) (1.8.1)
Requirement already satisfied: google-auth<3,>=1.6.3 in
/usr/local/lib/python3.7/dist-packages (from tensorboard<2.9,>=2.8->tensorflow-
gpu) (1.35.0)
Requirement already satisfied: requests<3,>=2.21.0 in
/usr/local/lib/python3.7/dist-packages (from tensorboard<2.9,>=2.8->tensorflow-
gpu) (2.23.0)
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.7/dist-
packages (from tensorboard<2.9,>=2.8->tensorflow-gpu) (3.3.6)
Requirement already satisfied: werkzeug>=0.11.15 in
```

```
/usr/local/lib/python3.7/dist-packages (from tensorboard<2.9,>=2.8->tensorflow-
gpu) (1.0.1)
Requirement already satisfied: tensorboard-data-server<0.7.0,>=0.6.0 in
/usr/local/lib/python3.7/dist-packages (from tensorboard<2.9,>=2.8->tensorflow-
gpu) (0.6.1)
Requirement already satisfied: cachetools<5.0,>=2.0.0 in
/usr/local/lib/python3.7/dist-packages (from google-
auth<3,>=1.6.3->tensorboard<2.9,>=2.8->tensorflow-gpu) (4.2.4)
Requirement already satisfied: rsa<5,>=3.1.4 in /usr/local/lib/python3.7/dist-
packages (from google-auth<3,>=1.6.3->tensorboard<2.9,>=2.8->tensorflow-gpu)
(4.8)
Requirement already satisfied: pyasn1-modules>=0.2.1 in
/usr/local/lib/python3.7/dist-packages (from google-
auth<3,>=1.6.3->tensorboard<2.9,>=2.8->tensorflow-gpu) (0.2.8)
Requirement already satisfied: requests-oauthlib>=0.7.0 in
/usr/local/lib/python3.7/dist-packages (from google-auth-
oauthlib<0.5,>=0.4.1->tensorboard<2.9,>=2.8->tensorflow-gpu) (1.3.1)
Requirement already satisfied: importlib-metadata>=4.4 in
/usr/local/lib/python3.7/dist-packages (from
markdown>=2.6.8->tensorboard<2.9,>=2.8->tensorflow-gpu) (4.11.3)
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.7/dist-
packages (from importlib-
metadata>=4.4->markdown>=2.6.8->tensorboard<2.9,>=2.8->tensorflow-gpu) (3.7.0)
Requirement already satisfied: pyasn1<0.5.0,>=0.4.6 in
/usr/local/lib/python3.7/dist-packages (from pyasn1-modules>=0.2.1->google-
auth<3,>=1.6.3->tensorboard<2.9,>=2.8->tensorflow-gpu) (0.4.8)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.7/dist-packages (from
requests<3,>=2.21.0->tensorboard<2.9,>=2.8->tensorflow-gpu) (2021.10.8)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in
/usr/local/lib/python3.7/dist-packages (from
requests<3,>=2.21.0->tensorboard<2.9,>=2.8->tensorflow-gpu) (1.24.3)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-
packages (from requests<3,>=2.21.0->tensorboard<2.9,>=2.8->tensorflow-gpu)
(2.10)
Requirement already satisfied: chardet<4,>=3.0.2 in
/usr/local/lib/python3.7/dist-packages (from
requests<3,>=2.21.0->tensorboard<2.9,>=2.8->tensorflow-gpu) (3.0.4)
Requirement already satisfied: oauthlib>=3.0.0 in /usr/local/lib/python3.7/dist-
packages (from requests-oauthlib>=0.7.0->google-auth-
oauthlib<0.5,>=0.4.1->tensorboard<2.9,>=2.8->tensorflow-gpu) (3.2.0)
Installing collected packages: tf-estimator-nightly, tensorflow-gpu
Successfully installed tensorflow-gpu-2.8.0 tf-estimator-
nightly-2.8.0.dev2021122109
```

[4]:

```
'''
A bunch of imports, you don't have to worry about these
```

```
'''
import collections
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 import Monitor
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
from typing import Any, List, Sequence, Tuple
from tensorflow.keras import layers
import statistics
import tqdm
from tqdm import tqdm
```

[ ]:
```
'''
Please refer to the first tutorial for more details on the specifics of␣
↪environments
We've only added important commands you might find useful for experiments.
'''

'''
List of example environments
(Source - https://gym.openai.com/envs/#classic_control)

'Acrobot-v1'
'CartPole-v0'
'MountainCar-v0'
'''

env = gym.make('CartPole-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("----")

'''
# Understanding State, Action, Reward Dynamics

The agent decides an action to take depending on the state.

The Environment keeps a variable specifically for the current state.
- Everytime an action is passed to the environment, it calculates the new state␣
↪and updates the current state variable.
- It returns the new current state and reward for the agent to take the next␣
↪action

'''

state = env.reset()
''' This returns the initial state (when environment is reset) '''

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

action = env.action_space.sample()
''' We take a random action now '''

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

next_state, reward, done, info = env.step(action)
''' env.step is used to calculate new state and obtain reward based on old␣
↪state and action taken  '''

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

```
4
2
0
----
[-0.04456399  0.04653909  0.01326909 -0.02099827]
```

```
----
1
----
[-0.04363321  0.24146826  0.01284913 -0.30946528]
1.0
False
{}
----
```

[12]:
```python
class ActorCriticModel(tf.keras.Model):
    """
    Defining policy and value networkss
    """
    def __init__(self, action_size, n_hidden1=512, n_hidden2=512):
        super(ActorCriticModel, self).__init__()

        #Hidden Layer 1
        self.fc1 = tf.keras.layers.Dense(n_hidden1, activation='relu')
        #Hidden Layer 2
        #self.fc2 = tf.keras.layers.Dense(n_hidden2, activation='relu')

        #Output Layer for policy
        self.pi_out = tf.keras.layers.Dense(action_size, activation='softmax')
        #Output Layer for state-value
        self.v_out = tf.keras.layers.Dense(1)

    def call(self, state):
        """
        Computes policy distribution and state-value for a given state
        """
        layer1 = self.fc1(state)
        #layer2 = self.fc2(layer1)

        pi = self.pi_out(layer1)
        v = self.v_out(layer1)

        return pi, v
```

[13]:
```python
class Agent:
    """
    Agent class
    """
    def __init__(self, action_size, lr=0.0001, gamma=0.99, seed = 85):
        self.gamma = gamma
        self.ac_model = ActorCriticModel(action_size=action_size)
        self.ac_model.compile(tf.keras.optimizers.Adam(learning_rate=lr))
        np.random.seed(seed)
```

```python
    def sample_action(self, state):
        """
        Given a state, compute the policy distribution over all actions and␣
↪sample one action
        """
        pi,_ = self.ac_model(state)

        action_probabilities = tfp.distributions.Categorical(probs=pi)
        sample = action_probabilities.sample()

        return int(sample.numpy()[0])

    def actor_loss(self, action, pi, delta):
        """
        Compute Actor Loss
        """
        return -tf.math.log(pi[0,action]) * delta

    def critic_loss(self,delta):
        """
        Critic loss aims to minimize TD error
        """
        return delta**2

    @tf.function
    def learn(self, state, action, reward, next_state, done):
        """
        For a given transition (s,a,s',r) update the paramters by computing the
        gradient of the total loss
        """
        with tf.GradientTape(persistent=True) as tape:
            pi, V_s = self.ac_model(state)
            _, V_s_next = self.ac_model(next_state)

            V_s = tf.squeeze(V_s)
            V_s_next = tf.squeeze(V_s_next)


            #### TO DO: Write the equation for delta (TD error)
            ## Write code below
            delta = reward + self.gamma*V_s_next-V_s

            loss_a = self.actor_loss(action, pi, delta)
            loss_c =self.critic_loss(delta)
            loss_total = loss_a + loss_c
```

```
        gradient = tape.gradient(loss_total, self.ac_model.trainable_variables)
        self.ac_model.optimizer.apply_gradients(zip(gradient, self.ac_model.
 ↪trainable_variables))
```

[15]:

```
env = gym.make('CartPole-v1')

#Initializing Agent
agent = Agent(lr=1e-4, action_size=env.action_space.n)
#Number of episodes
episodes = 1800
tf.compat.v1.reset_default_graph()

reward_list = []
average_reward_list = []
nsteps_list = []
begin_time = datetime.datetime.now()

for ep in range(1, episodes + 1):
    state = env.reset().reshape(1,-1)
    done = False
    ep_rew = 0
    nsteps = 0
    while not done:
        action = agent.sample_action(state) ##Sample Action
        next_state, reward, done, info = env.step(action) ##Take action
        next_state = next_state.reshape(1,-1)
        ep_rew += reward  ##Updating episode reward
        agent.learn(state, action, reward, next_state, done) ##Update Parameters
        state = next_state ##Updating State
        nsteps +=1
    nsteps_list.append(nsteps)
    reward_list.append(ep_rew)

    if ep % 10 == 0:
        avg_rew = np.mean(reward_list[-10:])
        print('Episode ', ep, 'Reward %f' % ep_rew, 'nsteps %f' % nsteps,␣
 ↪'Average Reward %f' % avg_rew)

    if ep % 100:
        avg_100 =  np.mean(reward_list[-100:])
        if avg_100 > 475.0:
            print('Stopped at Episode ',ep)
            break

time_taken = datetime.datetime.now() - begin_time
print(time_taken)
```

```
Episode  10 Reward 16.000000 nsteps 16.000000 Average Reward 31.500000
Episode  20 Reward 83.000000 nsteps 83.000000 Average Reward 38.700000
Episode  30 Reward 43.000000 nsteps 43.000000 Average Reward 44.000000
Episode  40 Reward 94.000000 nsteps 94.000000 Average Reward 66.300000
Episode  50 Reward 41.000000 nsteps 41.000000 Average Reward 51.300000
Episode  60 Reward 69.000000 nsteps 69.000000 Average Reward 59.200000
Episode  70 Reward 163.000000 nsteps 163.000000 Average Reward 67.900000
Episode  80 Reward 110.000000 nsteps 110.000000 Average Reward 96.200000
Episode  90 Reward 139.000000 nsteps 139.000000 Average Reward 103.300000
Episode  100 Reward 265.000000 nsteps 265.000000 Average Reward 170.400000
Episode  110 Reward 134.000000 nsteps 134.000000 Average Reward 165.300000
Episode  120 Reward 89.000000 nsteps 89.000000 Average Reward 144.500000
Episode  130 Reward 123.000000 nsteps 123.000000 Average Reward 132.100000
Episode  140 Reward 76.000000 nsteps 76.000000 Average Reward 129.800000
Episode  150 Reward 123.000000 nsteps 123.000000 Average Reward 118.100000
Episode  160 Reward 81.000000 nsteps 81.000000 Average Reward 98.900000
Episode  170 Reward 47.000000 nsteps 47.000000 Average Reward 70.800000
Episode  180 Reward 62.000000 nsteps 62.000000 Average Reward 56.700000
Episode  190 Reward 52.000000 nsteps 52.000000 Average Reward 82.000000
Episode  200 Reward 45.000000 nsteps 45.000000 Average Reward 60.900000
Episode  210 Reward 60.000000 nsteps 60.000000 Average Reward 59.500000
Episode  220 Reward 48.000000 nsteps 48.000000 Average Reward 70.300000
Episode  230 Reward 63.000000 nsteps 63.000000 Average Reward 47.500000
Episode  240 Reward 116.000000 nsteps 116.000000 Average Reward 67.600000
Episode  250 Reward 117.000000 nsteps 117.000000 Average Reward 70.200000
Episode  260 Reward 131.000000 nsteps 131.000000 Average Reward 120.100000
Episode  270 Reward 76.000000 nsteps 76.000000 Average Reward 93.300000
Episode  280 Reward 191.000000 nsteps 191.000000 Average Reward 196.000000
Episode  290 Reward 500.000000 nsteps 500.000000 Average Reward 189.800000
Episode  300 Reward 161.000000 nsteps 161.000000 Average Reward 255.100000
Episode  310 Reward 144.000000 nsteps 144.000000 Average Reward 152.300000
Episode  320 Reward 251.000000 nsteps 251.000000 Average Reward 139.700000
Episode  330 Reward 212.000000 nsteps 212.000000 Average Reward 261.900000
Episode  340 Reward 81.000000 nsteps 81.000000 Average Reward 129.900000
Episode  350 Reward 422.000000 nsteps 422.000000 Average Reward 210.000000
Episode  360 Reward 256.000000 nsteps 256.000000 Average Reward 272.200000
Episode  370 Reward 138.000000 nsteps 138.000000 Average Reward 186.400000
Episode  380 Reward 88.000000 nsteps 88.000000 Average Reward 197.000000
Episode  390 Reward 275.000000 nsteps 275.000000 Average Reward 195.800000
Episode  400 Reward 160.000000 nsteps 160.000000 Average Reward 245.100000
Episode  410 Reward 220.000000 nsteps 220.000000 Average Reward 326.200000
Episode  420 Reward 189.000000 nsteps 189.000000 Average Reward 276.100000
Episode  430 Reward 150.000000 nsteps 150.000000 Average Reward 163.400000
Episode  440 Reward 242.000000 nsteps 242.000000 Average Reward 274.600000
Episode  450 Reward 199.000000 nsteps 199.000000 Average Reward 178.900000
Episode  460 Reward 193.000000 nsteps 193.000000 Average Reward 274.500000
Episode  470 Reward 303.000000 nsteps 303.000000 Average Reward 268.300000
Episode  480 Reward 210.000000 nsteps 210.000000 Average Reward 375.400000
```

```
Episode  490 Reward 218.000000 nsteps 218.000000 Average Reward 235.100000
Episode  500 Reward 168.000000 nsteps 168.000000 Average Reward 204.800000
Episode  510 Reward 315.000000 nsteps 315.000000 Average Reward 266.800000
Episode  520 Reward 215.000000 nsteps 215.000000 Average Reward 281.700000
Episode  530 Reward 500.000000 nsteps 500.000000 Average Reward 451.600000
Episode  540 Reward 361.000000 nsteps 361.000000 Average Reward 375.300000
Episode  550 Reward 500.000000 nsteps 500.000000 Average Reward 382.900000
Episode  560 Reward 252.000000 nsteps 252.000000 Average Reward 303.700000
Episode  570 Reward 117.000000 nsteps 117.000000 Average Reward 142.200000
Episode  580 Reward 97.000000 nsteps 97.000000 Average Reward 98.600000
Episode  590 Reward 124.000000 nsteps 124.000000 Average Reward 110.900000
Episode  600 Reward 228.000000 nsteps 228.000000 Average Reward 160.200000
Episode  610 Reward 280.000000 nsteps 280.000000 Average Reward 251.200000
Episode  620 Reward 207.000000 nsteps 207.000000 Average Reward 198.000000
Episode  630 Reward 269.000000 nsteps 269.000000 Average Reward 214.800000
Episode  640 Reward 229.000000 nsteps 229.000000 Average Reward 190.400000
Episode  650 Reward 500.000000 nsteps 500.000000 Average Reward 350.700000
Episode  660 Reward 187.000000 nsteps 187.000000 Average Reward 399.000000
Episode  670 Reward 267.000000 nsteps 267.000000 Average Reward 220.400000
Episode  680 Reward 115.000000 nsteps 115.000000 Average Reward 138.200000
Episode  690 Reward 111.000000 nsteps 111.000000 Average Reward 111.600000
Episode  700 Reward 124.000000 nsteps 124.000000 Average Reward 117.300000
Episode  710 Reward 106.000000 nsteps 106.000000 Average Reward 123.500000
Episode  720 Reward 131.000000 nsteps 131.000000 Average Reward 122.900000
Episode  730 Reward 130.000000 nsteps 130.000000 Average Reward 139.600000
Episode  740 Reward 120.000000 nsteps 120.000000 Average Reward 122.200000
Episode  750 Reward 75.000000 nsteps 75.000000 Average Reward 92.900000
Episode  760 Reward 107.000000 nsteps 107.000000 Average Reward 106.300000
Episode  770 Reward 132.000000 nsteps 132.000000 Average Reward 110.100000
Episode  780 Reward 114.000000 nsteps 114.000000 Average Reward 102.600000
Episode  790 Reward 109.000000 nsteps 109.000000 Average Reward 112.500000
Episode  800 Reward 114.000000 nsteps 114.000000 Average Reward 117.200000
Episode  810 Reward 93.000000 nsteps 93.000000 Average Reward 102.600000
Episode  820 Reward 126.000000 nsteps 126.000000 Average Reward 109.800000
Episode  830 Reward 111.000000 nsteps 111.000000 Average Reward 93.100000
Episode  840 Reward 61.000000 nsteps 61.000000 Average Reward 80.600000
Episode  850 Reward 89.000000 nsteps 89.000000 Average Reward 87.600000
Episode  860 Reward 120.000000 nsteps 120.000000 Average Reward 164.400000
Episode  870 Reward 149.000000 nsteps 149.000000 Average Reward 132.200000
Episode  880 Reward 68.000000 nsteps 68.000000 Average Reward 120.100000
Episode  890 Reward 92.000000 nsteps 92.000000 Average Reward 72.800000
Episode  900 Reward 122.000000 nsteps 122.000000 Average Reward 105.700000
Episode  910 Reward 146.000000 nsteps 146.000000 Average Reward 142.200000
Episode  920 Reward 175.000000 nsteps 175.000000 Average Reward 165.900000
Episode  930 Reward 411.000000 nsteps 411.000000 Average Reward 257.400000
Episode  940 Reward 181.000000 nsteps 181.000000 Average Reward 356.700000
Episode  950 Reward 467.000000 nsteps 467.000000 Average Reward 238.600000
Episode  960 Reward 500.000000 nsteps 500.000000 Average Reward 482.900000
```

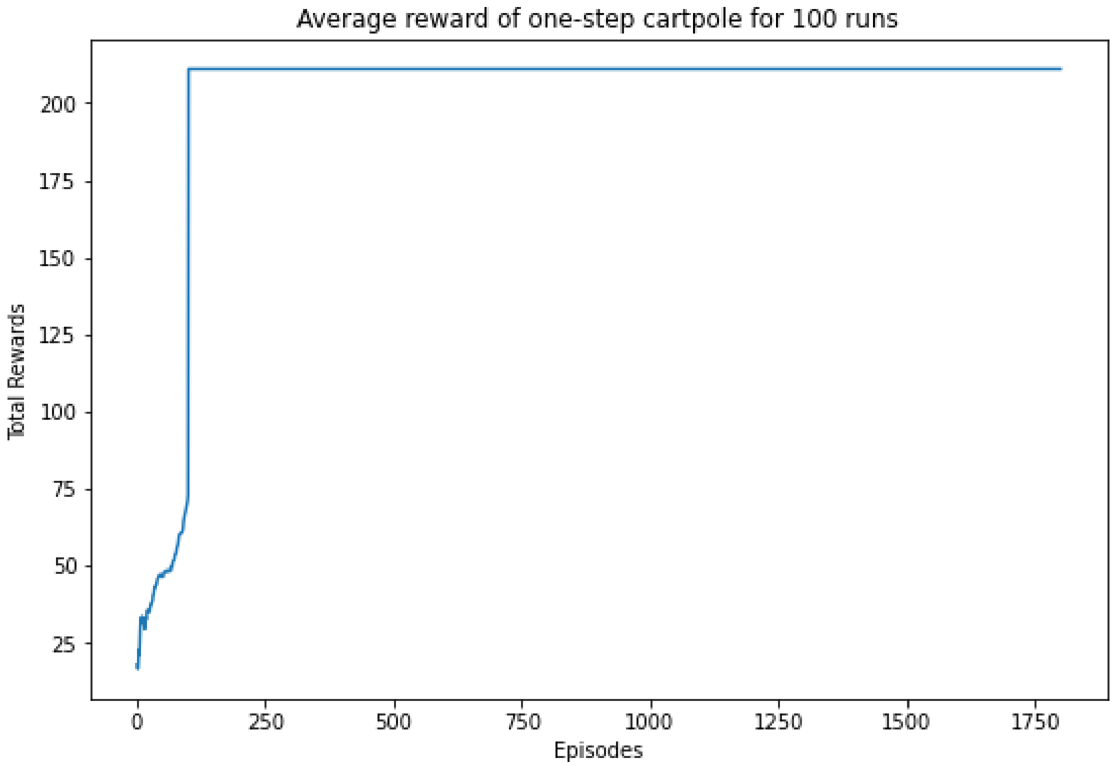
```
Episode  970 Reward 500.000000 nsteps 500.000000 Average Reward 500.000000
Episode  980 Reward 424.000000 nsteps 424.000000 Average Reward 492.400000
Episode  990 Reward 306.000000 nsteps 306.000000 Average Reward 345.900000
Episode  1000 Reward 500.000000 nsteps 500.000000 Average Reward 298.800000
Episode  1010 Reward 387.000000 nsteps 387.000000 Average Reward 465.400000
Episode  1020 Reward 475.000000 nsteps 475.000000 Average Reward 360.400000
Episode  1030 Reward 397.000000 nsteps 397.000000 Average Reward 346.500000
Episode  1040 Reward 500.000000 nsteps 500.000000 Average Reward 401.800000
Episode  1050 Reward 500.000000 nsteps 500.000000 Average Reward 500.000000
Episode  1060 Reward 500.000000 nsteps 500.000000 Average Reward 500.000000
Episode  1070 Reward 251.000000 nsteps 251.000000 Average Reward 323.200000
Episode  1080 Reward 174.000000 nsteps 174.000000 Average Reward 221.200000
Episode  1090 Reward 387.000000 nsteps 387.000000 Average Reward 264.500000
Episode  1100 Reward 500.000000 nsteps 500.000000 Average Reward 436.000000
Episode  1110 Reward 203.000000 nsteps 203.000000 Average Reward 318.700000
Episode  1120 Reward 500.000000 nsteps 500.000000 Average Reward 387.400000
Episode  1130 Reward 333.000000 nsteps 333.000000 Average Reward 475.400000
Episode  1140 Reward 384.000000 nsteps 384.000000 Average Reward 391.100000
Episode  1150 Reward 500.000000 nsteps 500.000000 Average Reward 451.900000
Episode  1160 Reward 354.000000 nsteps 354.000000 Average Reward 360.300000
Episode  1170 Reward 208.000000 nsteps 208.000000 Average Reward 217.800000
Episode  1180 Reward 214.000000 nsteps 214.000000 Average Reward 238.700000
Episode  1190 Reward 145.000000 nsteps 145.000000 Average Reward 150.800000
Episode  1200 Reward 207.000000 nsteps 207.000000 Average Reward 160.700000
Episode  1210 Reward 333.000000 nsteps 333.000000 Average Reward 281.500000
Episode  1220 Reward 222.000000 nsteps 222.000000 Average Reward 237.300000
Episode  1230 Reward 180.000000 nsteps 180.000000 Average Reward 183.900000
Episode  1240 Reward 208.000000 nsteps 208.000000 Average Reward 199.500000
Episode  1250 Reward 143.000000 nsteps 143.000000 Average Reward 157.800000
Episode  1260 Reward 148.000000 nsteps 148.000000 Average Reward 138.600000
Episode  1270 Reward 144.000000 nsteps 144.000000 Average Reward 138.000000
Episode  1280 Reward 126.000000 nsteps 126.000000 Average Reward 139.700000
Episode  1290 Reward 139.000000 nsteps 139.000000 Average Reward 134.100000
Episode  1300 Reward 123.000000 nsteps 123.000000 Average Reward 124.700000
Episode  1310 Reward 118.000000 nsteps 118.000000 Average Reward 118.200000
Episode  1320 Reward 118.000000 nsteps 118.000000 Average Reward 121.600000
Episode  1330 Reward 134.000000 nsteps 134.000000 Average Reward 133.700000
Episode  1340 Reward 141.000000 nsteps 141.000000 Average Reward 132.000000
Episode  1350 Reward 199.000000 nsteps 199.000000 Average Reward 165.400000
Episode  1360 Reward 236.000000 nsteps 236.000000 Average Reward 213.000000
Episode  1370 Reward 132.000000 nsteps 132.000000 Average Reward 170.800000
Episode  1380 Reward 140.000000 nsteps 140.000000 Average Reward 127.900000
Episode  1390 Reward 120.000000 nsteps 120.000000 Average Reward 128.100000
Episode  1400 Reward 109.000000 nsteps 109.000000 Average Reward 124.600000
Episode  1410 Reward 114.000000 nsteps 114.000000 Average Reward 114.400000
Episode  1420 Reward 125.000000 nsteps 125.000000 Average Reward 121.000000
Episode  1430 Reward 112.000000 nsteps 112.000000 Average Reward 115.200000
Episode  1440 Reward 108.000000 nsteps 108.000000 Average Reward 108.900000
```

```
Episode  1450 Reward 99.000000 nsteps 99.000000 Average Reward 103.000000
Episode  1460 Reward 37.000000 nsteps 37.000000 Average Reward 49.700000
Episode  1470 Reward 37.000000 nsteps 37.000000 Average Reward 46.900000
Episode  1480 Reward 36.000000 nsteps 36.000000 Average Reward 35.200000
Episode  1490 Reward 28.000000 nsteps 28.000000 Average Reward 31.100000
Episode  1500 Reward 36.000000 nsteps 36.000000 Average Reward 30.300000
Episode  1510 Reward 34.000000 nsteps 34.000000 Average Reward 36.900000
Episode  1520 Reward 88.000000 nsteps 88.000000 Average Reward 61.400000
Episode  1530 Reward 103.000000 nsteps 103.000000 Average Reward 68.500000
Episode  1540 Reward 100.000000 nsteps 100.000000 Average Reward 102.900000
Episode  1550 Reward 37.000000 nsteps 37.000000 Average Reward 80.500000
Episode  1560 Reward 23.000000 nsteps 23.000000 Average Reward 32.400000
Episode  1570 Reward 38.000000 nsteps 38.000000 Average Reward 32.000000
Episode  1580 Reward 34.000000 nsteps 34.000000 Average Reward 34.200000
Episode  1590 Reward 30.000000 nsteps 30.000000 Average Reward 30.400000
Episode  1600 Reward 34.000000 nsteps 34.000000 Average Reward 31.200000
Episode  1610 Reward 92.000000 nsteps 92.000000 Average Reward 37.500000
Episode  1620 Reward 100.000000 nsteps 100.000000 Average Reward 97.700000
Episode  1630 Reward 109.000000 nsteps 109.000000 Average Reward 101.200000
Episode  1640 Reward 99.000000 nsteps 99.000000 Average Reward 115.000000
Episode  1650 Reward 105.000000 nsteps 105.000000 Average Reward 104.200000
Episode  1660 Reward 122.000000 nsteps 122.000000 Average Reward 108.900000
Episode  1670 Reward 118.000000 nsteps 118.000000 Average Reward 111.000000
Episode  1680 Reward 102.000000 nsteps 102.000000 Average Reward 108.400000
Episode  1690 Reward 113.000000 nsteps 113.000000 Average Reward 113.500000
Episode  1700 Reward 107.000000 nsteps 107.000000 Average Reward 118.100000
Episode  1710 Reward 116.000000 nsteps 116.000000 Average Reward 117.000000
Episode  1720 Reward 135.000000 nsteps 135.000000 Average Reward 125.800000
Episode  1730 Reward 150.000000 nsteps 150.000000 Average Reward 131.500000
Episode  1740 Reward 149.000000 nsteps 149.000000 Average Reward 140.800000
Episode  1750 Reward 186.000000 nsteps 186.000000 Average Reward 177.000000
Episode  1760 Reward 331.000000 nsteps 331.000000 Average Reward 343.400000
Episode  1770 Reward 500.000000 nsteps 500.000000 Average Reward 500.000000
Episode  1780 Reward 185.000000 nsteps 185.000000 Average Reward 227.700000
Episode  1790 Reward 132.000000 nsteps 132.000000 Average Reward 184.300000
Episode  1800 Reward 193.000000 nsteps 193.000000 Average Reward 163.800000
1:04:02.761438
```

[16]:
```
### Plot of total reward vs episode
## Write Code Below
mov_avg=[]
for i in range(1,len(reward_list)):
  if i > 100:
    mov_avg.append(np.mean(reward_list[-100:]))
  else:
    mov_avg.append(np.mean(reward_list[:i]))
```

```
plt.figure(figsize=(9,6))
plt.xlabel('Episodes')
plt.ylabel('Total Rewards')
plt.title('Average reward of one-step cartpole for 100 runs')
plt.plot(np.arange(1,len(mov_avg)+1),mov_avg)
```

[16]: [<matplotlib.lines.Line2D at 0x7f96a5dc9310>]



[17]:
```
episodes = 1000
env = gym.make('CartPole-v1')
agent = Agent(lr=1e-4, action_size=env.action_space.n)
#avg_reward_eps = np.zeros(episodes)
reward_eps = []
for run in range(10):
  print('Run:%f' % run)
#Initializing Agent

  tf.compat.v1.reset_default_graph()

  reward_list = []
  average_reward_list = []
  nsteps_list = []
```

```python
begin_time = datetime.datetime.now()

for ep in tqdm(range(1, episodes + 1)):
    state = env.reset().reshape(1,-1)
    done = False
    ep_rew = 0
    nsteps = 0
    while not done:
        action = agent.sample_action(state) ##Sample Action
        next_state, reward, done, info = env.step(action) ##Take action
        next_state = next_state.reshape(1,-1)
        ep_rew += reward  ##Updating episode reward
        agent.learn(state, action, reward, next_state, done) ##Update␣
↪Parameters
        state = next_state ##Updating State
        nsteps +=1
    nsteps_list.append(nsteps)
    reward_list.append(ep_rew)


reward_eps.append(np.array(reward_list))
    #if ep % 10 == 0:
    #    avg_rew = np.mean(reward_list[-10:])
    #    print('Episode ', ep, 'Reward %f' % ep_rew, 'nsteps %f' % nsteps,␣
↪'Average Reward %f' % avg_rew)

    #if ep % 100:
    #    avg_100 =  np.mean(reward_list[-100:])
    #    if avg_100 > 475.0:
    #        print('Stopped at Episode ',ep)
    #        break
```

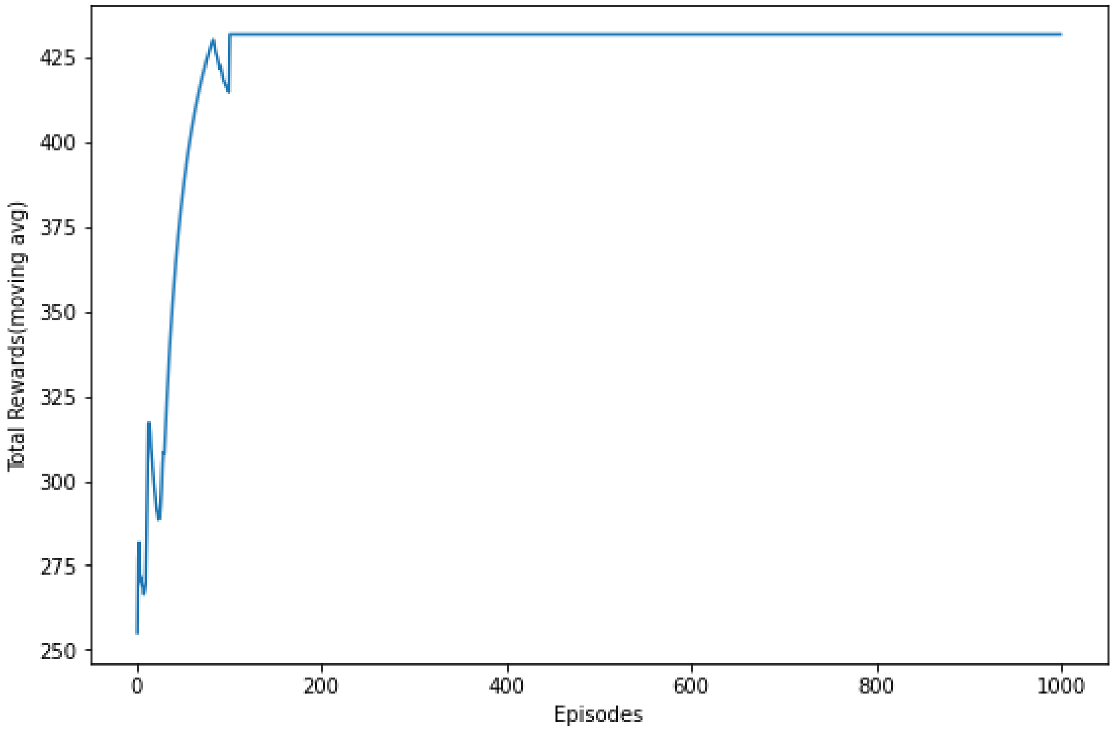
```
Run:0.000000

100%|      | 1000/1000 [50:01<00:00,  3.00s/it]

Run:1.000000

100%|      | 1000/1000 [17:39<00:00,  1.06s/it]

Run:2.000000

100%|      | 1000/1000 [16:16<00:00,  1.02it/s]

Run:3.000000

100%|      | 1000/1000 [05:47<00:00,  2.88it/s]

Run:4.000000

100%|      | 1000/1000 [22:15<00:00,  1.34s/it]
```

```
Run:5.000000

100%|      | 1000/1000 [43:54<00:00,  2.63s/it]

Run:6.000000

100%|      | 1000/1000 [1:28:21<00:00,  5.30s/it]

Run:7.000000

100%|      | 1000/1000 [1:32:34<00:00,  5.55s/it]

Run:8.000000

100%|      | 1000/1000 [1:26:14<00:00,  5.17s/it]

Run:9.000000

100%|      | 1000/1000 [1:28:12<00:00,  5.29s/it]
```

Lets plot and see some of the individual runs

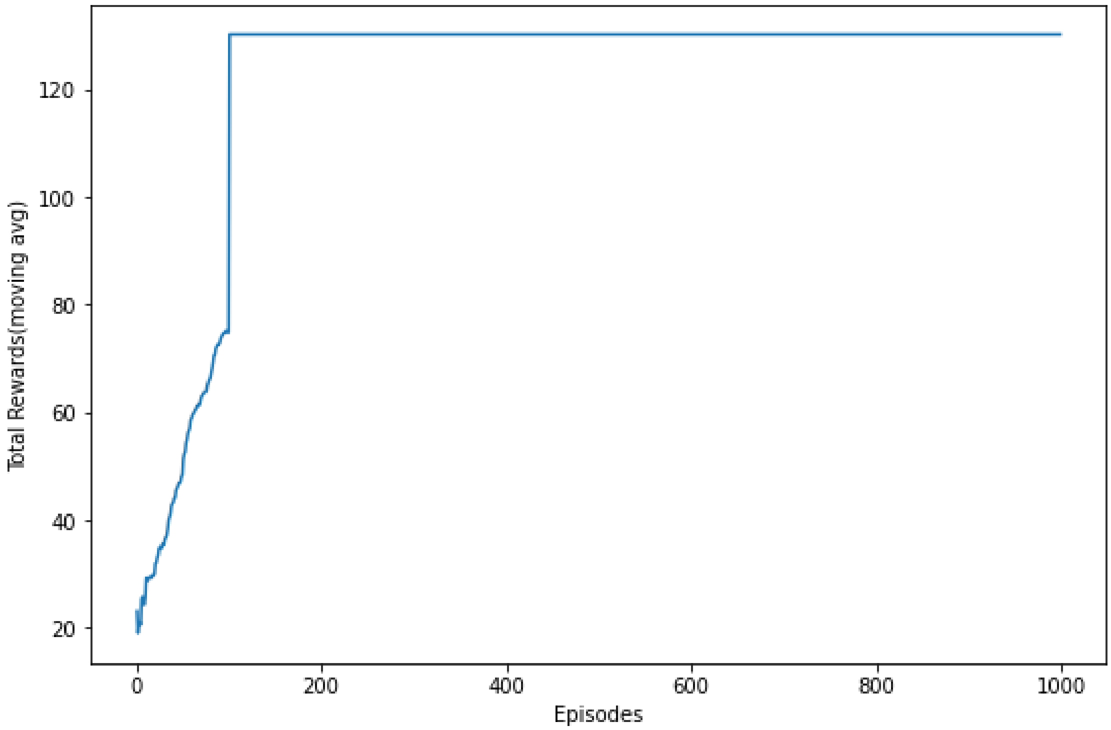
[29]:
```
mov_avg=[]
for i in range(1,len(reward_eps[-3])):
  if i > 100:
    mov_avg.append(np.mean(reward_eps[-2][-100:]))
  else:
    mov_avg.append(np.mean(reward_eps[-2][:i]))

plt.figure(figsize=(9,6))
plt.xlabel('Episodes')
plt.ylabel('Total Rewards(moving avg)')
plt.plot(np.arange(1,len(mov_avg)+1),mov_avg)
```

[29]:
```
[<matplotlib.lines.Line2D at 0x7f96a427e450>]
```



[27]:
```
mov_avg=[]
for i in range(1,len(reward_eps[0])):
  if i > 100:
    mov_avg.append(np.mean(reward_eps[0][-100:]))
  else:
    mov_avg.append(np.mean(reward_eps[0][:i]))

plt.figure(figsize=(9,6))
plt.xlabel('Episodes')
plt.ylabel('Total Rewards(moving avg)')
plt.plot(np.arange(1,len(mov_avg)+1),mov_avg)
```

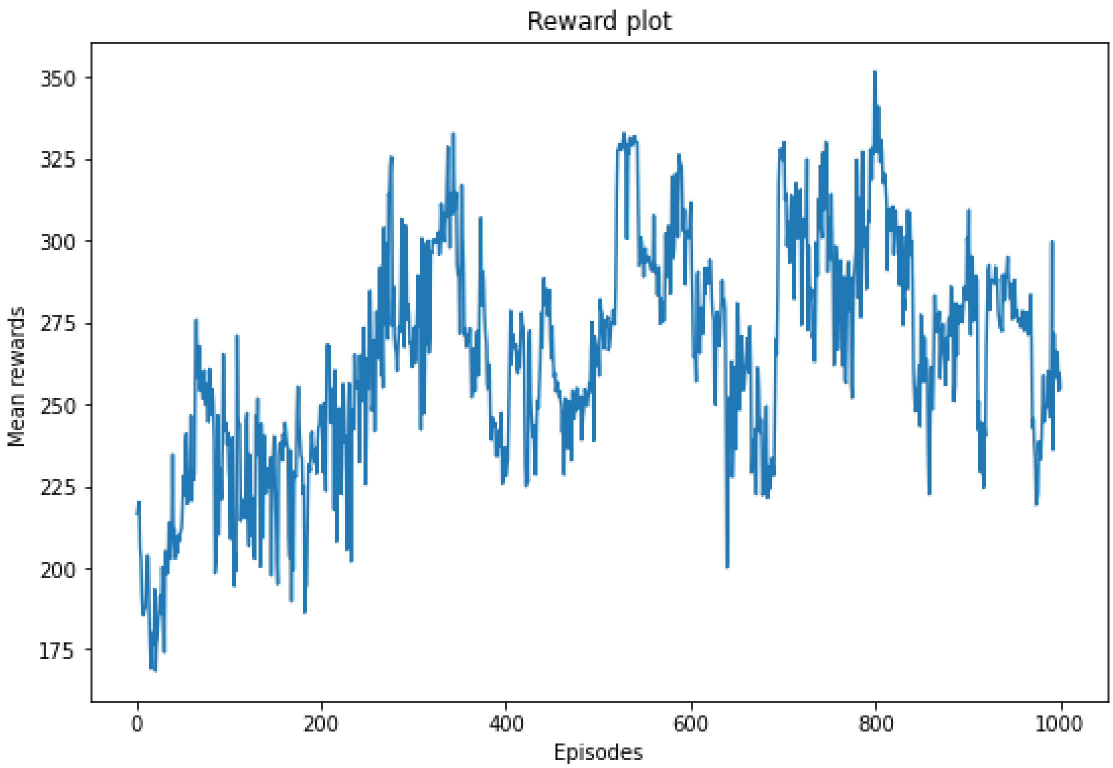
[27]:
```
[<matplotlib.lines.Line2D at 0x7f96a439dd50>]
```



[19]:
```
reward_eps = np.array(reward_eps)
mean=np.array([np.mean(reward_eps, axis=0) for i in range(10)])
var = np.sqrt(np.sum((reward_eps-mean)**2,axis=0))/10
```

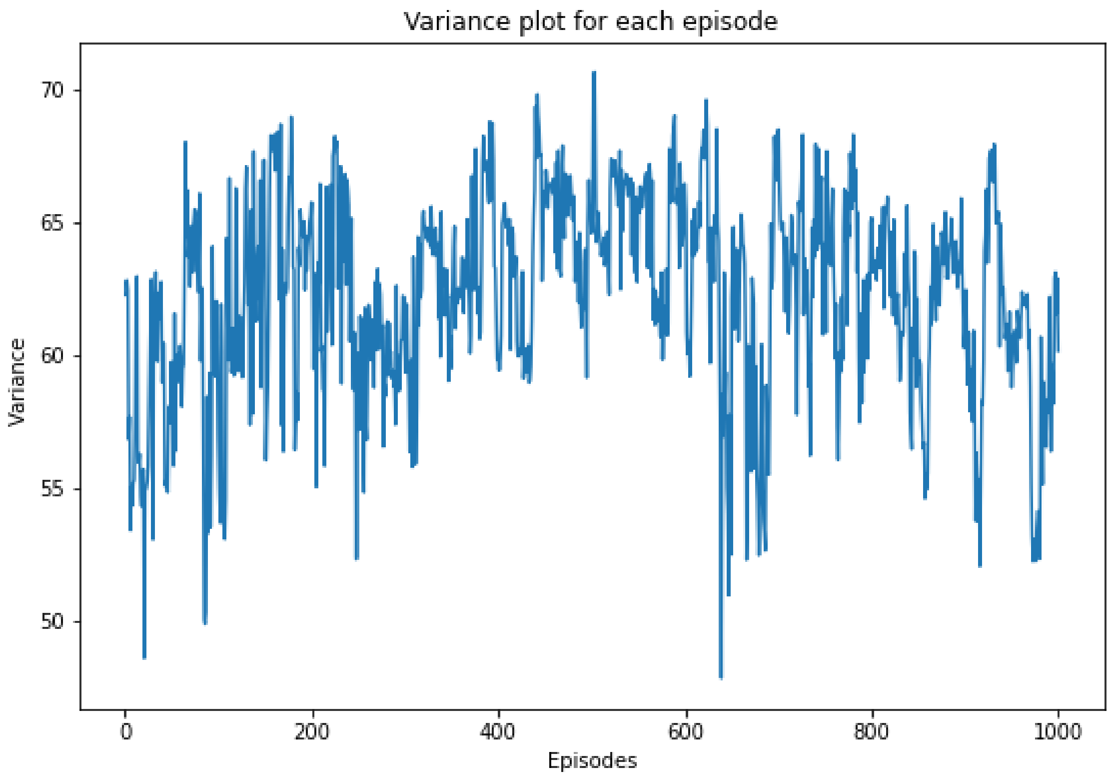
[20]:
```
plt.figure(figsize=(9,6))
plt.xlabel('Episodes')
plt.ylabel('Mean rewards ')
plt.title('Reward plot')
plt.plot(np.arange(1,len(mean[1])+1),mean[1])
```

[20]:
```
[<matplotlib.lines.Line2D at 0x7f96a5d94890>]
```



[22]:
```
plt.figure(figsize=(9,6))
plt.xlabel('Episodes')
plt.ylabel('Variance ')
plt.title('Variance plot for each episode')
plt.plot(np.arange(1,len(var)+1),var)
print(np.mean(var))
```

62.211973963979894



[ ]:

```

```

[ ]:

```

```

[ ]:

```

```

Full return

[ ]:

```
# Create the environment
env = gym.make("CartPole-v1")

# Set seed for experiment reproducibility
seed = 42
env.seed(seed)
tf.random.set_seed(seed)
np.random.seed(seed)

# Small epsilon value for stabilizing division operations
eps = np.finfo(np.float32).eps.item()
```

[ ]:

```
class ActorCritic(tf.keras.Model):
  """Combined actor-critic network."""
```

```
  def __init__(
      self,
      num_actions: int,
      num_hidden_units: int):
    """Initialize."""
    super().__init__()

    self.common = layers.Dense(num_hidden_units, activation="relu")
    self.actor = layers.Dense(num_actions)
    self.critic = layers.Dense(1)

  def call(self, inputs: tf.Tensor) -> Tuple[tf.Tensor, tf.Tensor]:
    x = self.common(inputs)
    return self.actor(x), self.critic(x)
```

[ ]:
```
num_actions = env.action_space.n  # 2
num_hidden_units = 128

model = ActorCritic(num_actions, num_hidden_units)
```

[ ]:
```
# Wrap OpenAI Gym's `env.step` call as an operation in a TensorFlow function.
# This would allow it to be included in a callable TensorFlow graph.

def env_step(action: np.ndarray) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
  """Returns state, reward and done flag given an action."""

  state, reward, done, _ = env.step(action)
  return (state.astype(np.float32),
          np.array(reward, np.int32),
          np.array(done, np.int32))


def tf_env_step(action: tf.Tensor) -> List[tf.Tensor]:
  return tf.numpy_function(env_step, [action],
                           [tf.float32, tf.int32, tf.int32])
```

[ ]:
```
def run_episode(
    initial_state: tf.Tensor,
    model: tf.keras.Model,
    max_steps: int) -> Tuple[tf.Tensor, tf.Tensor, tf.Tensor]:
  """Runs a single episode to collect training data."""

  action_probs = tf.TensorArray(dtype=tf.float32, size=0, dynamic_size=True)
  values = tf.TensorArray(dtype=tf.float32, size=0, dynamic_size=True)
  rewards = tf.TensorArray(dtype=tf.int32, size=0, dynamic_size=True)
```

```
  initial_state_shape = initial_state.shape
  state = initial_state

  for t in tf.range(max_steps):
    # Convert state into a batched tensor (batch size = 1)
    state = tf.expand_dims(state, 0)

    # Run the model and to get action probabilities and critic value
    action_logits_t, value = model(state)

    # Sample next action from the action probability distribution
    action = tf.random.categorical(action_logits_t, 1)[0, 0]
    action_probs_t = tf.nn.softmax(action_logits_t)

    # Store critic values
    values = values.write(t, tf.squeeze(value))

    # Store log probability of the action chosen
    action_probs = action_probs.write(t, action_probs_t[0, action])

    # Apply action to the environment to get next state and reward
    state, reward, done = tf_env_step(action)
    state.set_shape(initial_state_shape)

    # Store reward
    rewards = rewards.write(t, reward)

    if tf.cast(done, tf.bool):
      break

  action_probs = action_probs.stack()
  values = values.stack()
  rewards = rewards.stack()

  return action_probs, values, rewards
```

[ ]:
```
def get_expected_return(
    rewards: tf.Tensor,
    gamma: float,
    standardize: bool = True) -> tf.Tensor:
  """Compute expected returns per timestep."""

  n = tf.shape(rewards)[0]
  returns = tf.TensorArray(dtype=tf.float32, size=n)

  # Start from the end of `rewards` and accumulate reward sums
  # into the `returns` array
```

```
  rewards = tf.cast(rewards[::-1], dtype=tf.float32)
  discounted_sum = tf.constant(0.0)
  discounted_sum_shape = discounted_sum.shape
  for i in tf.range(n):
    reward = rewards[i]
    discounted_sum = reward + gamma * discounted_sum
    discounted_sum.set_shape(discounted_sum_shape)
    returns = returns.write(i, discounted_sum)
  returns = returns.stack()[::-1]

  if standardize:
    returns = ((returns - tf.math.reduce_mean(returns)) /
               (tf.math.reduce_std(returns) + eps))

  return returns
```

[ ]:
```
huber_loss = tf.keras.losses.Huber(reduction=tf.keras.losses.Reduction.SUM)

def compute_loss(
    action_probs: tf.Tensor,
    values: tf.Tensor,
    returns: tf.Tensor) -> tf.Tensor:
  """Computes the combined actor-critic loss."""

  advantage = returns - values

  action_log_probs = tf.math.log(action_probs)
  actor_loss = -tf.math.reduce_sum(action_log_probs * advantage)

  critic_loss = huber_loss(values, returns)

  return actor_loss + critic_loss
```

[ ]:
```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)


@tf.function
def train_step(
    initial_state: tf.Tensor,
    model: tf.keras.Model,
    optimizer: tf.keras.optimizers.Optimizer,
    gamma: float,
    max_steps_per_episode: int) -> tf.Tensor:
  """Runs a model training step."""

  with tf.GradientTape() as tape:
```

```
    # Run the model for one episode to collect training data
    action_probs, values, rewards = run_episode(
        initial_state, model, max_steps_per_episode)

    # Calculate expected returns
    returns = get_expected_return(rewards, gamma)

    # Convert training data to appropriate TF tensor shapes
    action_probs, values, returns = [
        tf.expand_dims(x, 1) for x in [action_probs, values, returns]]

    # Calculating loss values to update our network
    loss = compute_loss(action_probs, values, returns)

  # Compute the gradients from the loss
  grads = tape.gradient(loss, model.trainable_variables)

  # Apply the gradients to the model's parameters
  optimizer.apply_gradients(zip(grads, model.trainable_variables))

  episode_reward = tf.math.reduce_sum(rewards)

  return episode_reward
```

[ ]:
```
%%time

min_episodes_criterion = 100
max_episodes = 1800
max_steps_per_episode = 500


reward_threshold = 475
running_reward = 0


gamma = 0.99

reward_list = []
# Keep last episodes reward
episodes_reward: collections.deque = collections.
 ↪deque(maxlen=min_episodes_criterion)

with tqdm.trange(max_episodes) as t:
  for i in t:
    initial_state = tf.constant(env.reset(), dtype=tf.float32)
    episode_reward = int(train_step(
        initial_state, model, optimizer, gamma, max_steps_per_episode))
```

```
    episodes_reward.append(episode_reward)
    reward_list.append(episode_reward)
    running_reward = statistics.mean(episodes_reward)

    t.set_description(f'Episode {i}')
    t.set_postfix(
        episode_reward=episode_reward, running_reward=running_reward)

    # Show average episode reward every 10 episodes
    if i % 10 == 0:
      pass # print(f'Episode {i}: average reward: {avg_reward}')

    if running_reward > reward_threshold and i >= min_episodes_criterion:
        break

print(f'\nSolved at episode {i}: average reward: {running_reward:.2f}!')
```

```
Episode 102:   6%|          | 102/1800 [00:23<06:24,  4.42it/s,
episode_reward=500, running_reward=478]


Solved at episode 102: average reward: 478.22!
CPU times: user 33.5 s, sys: 1.51 s, total: 35 s
Wall time: 23.1 s
```

[ ]:
```
rewards_ep = []

for run in range(10):

  optimizer = tf.keras.optimizers.Adam(learning_rate=0.01)


  @tf.function
  def train_step(
      initial_state: tf.Tensor,
      model: tf.keras.Model,
      optimizer: tf.keras.optimizers.Optimizer,
      gamma: float,
      max_steps_per_episode: int) -> tf.Tensor:
    """Runs a model training step."""

    with tf.GradientTape() as tape:

      # Run the model for one episode to collect training data
      action_probs, values, rewards = run_episode(
```

```
        initial_state, model, max_steps_per_episode)

    # Calculate expected returns
    returns = get_expected_return(rewards, gamma)

    # Convert training data to appropriate TF tensor shapes
    action_probs, values, returns = [
        tf.expand_dims(x, 1) for x in [action_probs, values, returns]]

    # Calculating loss values to update our network
    loss = compute_loss(action_probs, values, returns)

  # Compute the gradients from the loss
  grads = tape.gradient(loss, model.trainable_variables)

  # Apply the gradients to the model's parameters
  optimizer.apply_gradients(zip(grads, model.trainable_variables))

  episode_reward = tf.math.reduce_sum(rewards)

  return episode_reward


env = gym.make("CartPole-v1")

model = ActorCritic(num_actions, num_hidden_units)

print(run)
# Keep last episodes reward
episodes_reward: collections.deque = collections.
↪deque(maxlen=min_episodes_criterion)

reward_list = []
with tqdm.trange(max_episodes) as t:
  for i in t:
    initial_state = tf.constant(env.reset(), dtype=tf.float32)
    episode_reward = int(train_step(
        initial_state, model, optimizer, gamma, max_steps_per_episode))

    episodes_reward.append(episode_reward)
    reward_list.append(episode_reward)
    running_reward = statistics.mean(episodes_reward)

    t.set_description(f'Episode {i}')
    t.set_postfix(
        episode_reward=episode_reward, running_reward=running_reward)
```

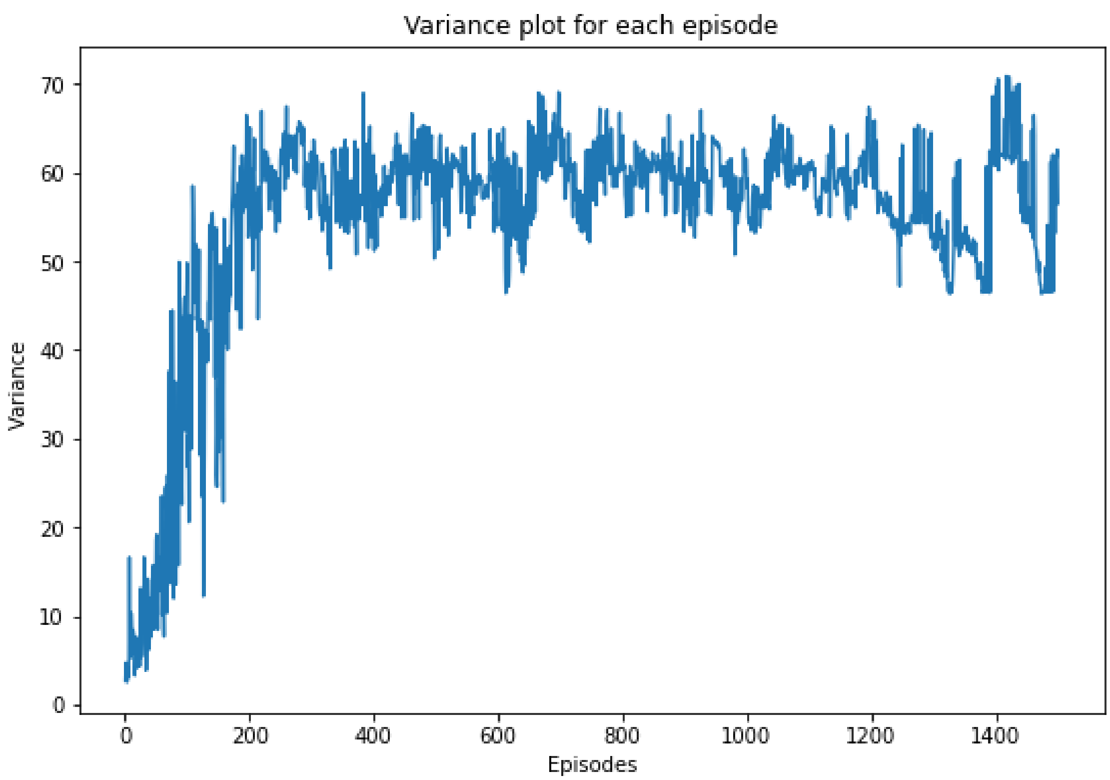
```
      # Show average episode reward every 10 episodes
      if i % 10 == 0:
        pass # print(f'Episode {i}: average reward: {avg_reward}')

  rewards_ep.append(reward_list)


#print(f'\nSolved at episode {i}: average reward: {running_reward:.2f}!')
```

```
0

Episode 1499: 100%|      | 1500/1500 [04:12<00:00,  5.95it/s,
episode_reward=500, running_reward=493]

1

Episode 1499: 100%|      | 1500/1500 [00:19<00:00, 76.31it/s,
episode_reward=9, running_reward=9.39]

2

Episode 1499: 100%|      | 1500/1500 [03:36<00:00,  6.91it/s,
episode_reward=500, running_reward=500]

3

Episode 1499: 100%|      | 1500/1500 [02:50<00:00,  8.80it/s,
episode_reward=500, running_reward=478]

4

Episode 1499: 100%|      | 1500/1500 [04:11<00:00,  5.96it/s,
episode_reward=125, running_reward=362]

5

Episode 1499: 100%|      | 1500/1500 [04:37<00:00,  5.40it/s,
episode_reward=500, running_reward=500]

6

Episode 1499: 100%|      | 1500/1500 [04:44<00:00,  5.28it/s,
episode_reward=500, running_reward=485]

7

Episode 1499: 100%|      | 1500/1500 [03:36<00:00,  6.92it/s,
episode_reward=500, running_reward=202]

8

Episode 1499: 100%|      | 1500/1500 [03:56<00:00,  6.34it/s,
episode_reward=250, running_reward=489]

9
```

```
Episode 1499: 100%|      | 1500/1500 [02:57<00:00,  8.46it/s,
episode_reward=500, running_reward=453]
```

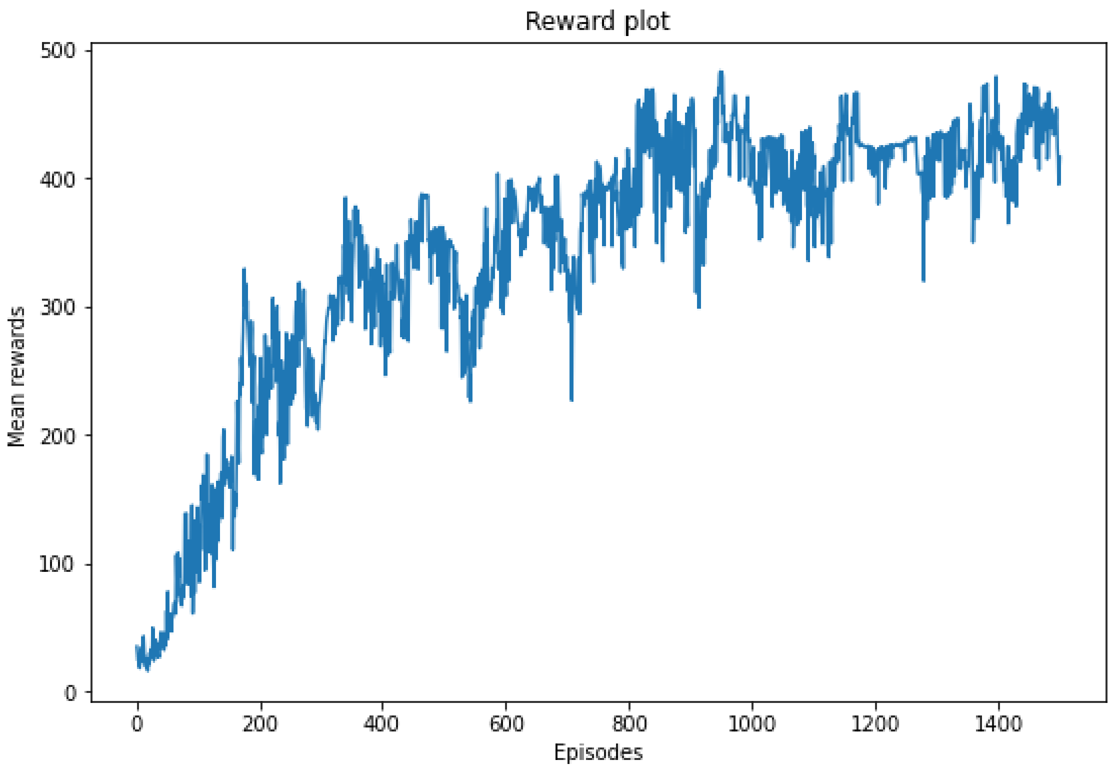
[ ]:
```
rewards_ep = np.array(rewards_ep)
mean=np.array([np.mean(rewards_ep, axis=0) for i in range(10)])
var = np.sqrt(np.sum((rewards_ep-mean)**2,axis=0))/10

plt.figure(figsize=(9,6))
plt.xlabel('Episodes')
plt.ylabel('Variance ')
plt.title('Variance plot for each episode')
plt.plot(np.arange(1,len(var)+1),var)
print(np.mean(var))
```

```
54.87694951735066
```



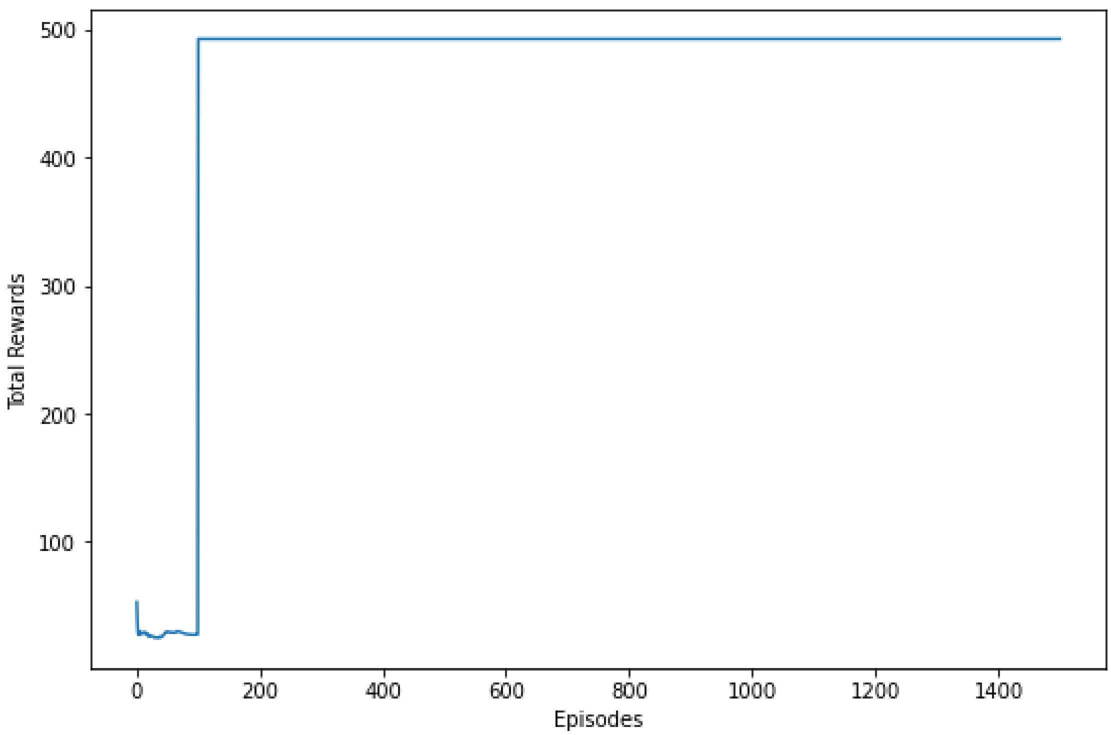
[ ]:
```
plt.figure(figsize=(9,6))
plt.xlabel('Episodes')
plt.ylabel('Mean rewards ')
plt.title('Reward plot')
plt.plot(np.arange(1,len(mean[1])+1),mean[1])
```

[ ]:
```
[<matplotlib.lines.Line2D at 0x7f5c4cd74f10>]
```



[ ]:
```
mov_avg=[]
for i in range(1,len(rewards_ep[0])):
  if i > 100:
    mov_avg.append(np.mean(rewards_ep[0][-100:]))
  else:
    mov_avg.append(np.mean(rewards_ep[0][:i]))

plt.figure(figsize=(9,6))
plt.xlabel('Episodes')
plt.ylabel('Total Rewards')
plt.plot(np.arange(1,len(mov_avg)+1),mov_avg)
```

[ ]:
```
[<matplotlib.lines.Line2D at 0x7f7ef2e1f890>]
```




```
[ ]:
```

n-step

```
[ ]: # Create the environment
env = gym.make("CartPole-v1")

# Set seed for experiment reproducibility
seed = 44
env.seed(seed)
tf.random.set_seed(seed)
np.random.seed(seed)

# Small epsilon value for stabilizing division operations
eps = np.finfo(np.float32).eps.item()
```

```
[ ]: class ActorCritic(tf.keras.Model):
  """Combined actor-critic network."""

  def __init__(
      self,
      num_actions: int,
      num_hidden_units: int):
```

```
    """Initialize."""
    super().__init__()

    self.common = layers.Dense(num_hidden_units, activation="relu")
    self.actor = layers.Dense(num_actions)
    self.critic = layers.Dense(1)

  def call(self, inputs: tf.Tensor) -> Tuple[tf.Tensor, tf.Tensor]:
    x = self.common(inputs)
    return self.actor(x), self.critic(x)
```

[ ]:
```
num_actions = env.action_space.n  # 2
num_hidden_units = 256

model = ActorCritic(num_actions, num_hidden_units)
```

[ ]:
```
# Wrap OpenAI Gym's `env.step` call as an operation in a TensorFlow function.
# This would allow it to be included in a callable TensorFlow graph.

def env_step(action: np.ndarray) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
  """Returns state, reward and done flag given an action."""

  state, reward, done, _ = env.step(action)
  return (state.astype(np.float32),
          np.array(reward, np.int32),
          np.array(done, np.int32))


def tf_env_step(action: tf.Tensor) -> List[tf.Tensor]:
  return tf.numpy_function(env_step, [action],
                           [tf.float32, tf.int32, tf.int32])
```

[ ]:
```
def run_episode(
    initial_state: tf.Tensor,
    model: tf.keras.Model,
    max_steps: int) -> Tuple[tf.Tensor, tf.Tensor, tf.Tensor]:
  """Runs a single episode to collect training data."""

  action_probs = tf.TensorArray(dtype=tf.float32, size=0, dynamic_size=True)
  values = tf.TensorArray(dtype=tf.float32, size=0, dynamic_size=True)
  rewards = tf.TensorArray(dtype=tf.int32, size=0, dynamic_size=True)

  initial_state_shape = initial_state.shape
  state = initial_state

  for t in tf.range(max_steps):
    # Convert state into a batched tensor (batch size = 1)
```

```python
    state = tf.expand_dims(state, 0)

    # Run the model and to get action probabilities and critic value
    action_logits_t, value = model(state)

    # Sample next action from the action probability distribution
    action = tf.random.categorical(action_logits_t, 1)[0, 0]
    action_probs_t = tf.nn.softmax(action_logits_t)

    # Store critic values
    values = values.write(t, tf.squeeze(value))

    # Store log probability of the action chosen
    action_probs = action_probs.write(t, action_probs_t[0, action])

    # Apply action to the environment to get next state and reward
    state, reward, done = tf_env_step(action)
    state.set_shape(initial_state_shape)

    # Store reward
    rewards = rewards.write(t, reward)

    if tf.cast(done, tf.bool):
      break

  action_probs = action_probs.stack()
  values = values.stack()
  rewards = rewards.stack()

  return action_probs, values, rewards
```

[ ]:
```python
def get_expected_return(
    rewards: tf.Tensor,
    values: tf.Tensor,
    step: int,
    gamma: float,
    standardize: bool = True) -> tf.Tensor:
  """Compute expected returns per timestep."""

  n = tf.shape(rewards)[0]
  returns = tf.TensorArray(dtype=tf.float32, size=n)

  # Start from the end of `rewards` and accumulate reward sums
  # into the `returns` array
  rewards = tf.cast(rewards, dtype=tf.float32)
  discounted_sum = tf.constant(0.0)
  discounted_sum_shape = discounted_sum.shape
```

```
  for i in tf.range(n):

    if i+step >= n:
      dsc = tf.constant(0.0)
      dsc_shape = dsc.shape
      for j in tf.range(i,n):
        dsc = tf.math.pow(gamma,float(j-i))*rewards[j] + dsc
        dsc.set_shape(dsc_shape)
      discounted_sum = dsc
    else:
      dsc = tf.constant(0.0)
      dsc_shape = dsc.shape
      for j in tf.range(i,i+step):
        dsc = tf.math.pow(gamma,float(j-i))*rewards[j] + dsc
        dsc.set_shape(dsc_shape)
      discounted_sum = dsc + tf.math.pow(gamma,float(step))*values[i+step]
    #discounted_sum.set_shape(discounted_sum_shape)


    #if i+step < n:
    #  discounted_sum += np.power(gamma,step)*values[i+step]

    #if i+step < n+1:
    #  discounted_sum += np.power(gamma,step-1)*rewards[i+step-1]

    discounted_sum.set_shape(discounted_sum_shape)
    returns = returns.write(i, discounted_sum)
  returns = returns.stack()

  if standardize:
    returns = ((returns - tf.math.reduce_mean(returns)) /
               (tf.math.reduce_std(returns) + eps))

  return returns
```

[ ]:
```
huber_loss = tf.keras.losses.Huber(reduction=tf.keras.losses.Reduction.SUM)

def compute_loss(
    action_probs: tf.Tensor,
    values: tf.Tensor,
    returns: tf.Tensor) -> tf.Tensor:
  """Computes the combined actor-critic loss."""

  advantage = returns - values

  action_log_probs = tf.math.log(action_probs)
  actor_loss = -tf.math.reduce_sum(action_log_probs * advantage)
```

```
  critic_loss = huber_loss(values, returns)

  return actor_loss + critic_loss
```

[ ]:
```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)


@tf.function
def train_step(
    initial_state: tf.Tensor,
    model: tf.keras.Model,
    optimizer: tf.keras.optimizers.Optimizer,
    gamma: float,
    step: int,
    max_steps_per_episode: int) -> tf.Tensor:
  """Runs a model training step."""

  with tf.GradientTape() as tape:

    # Run the model for one episode to collect training data
    action_probs, values, rewards = run_episode(
        initial_state, model, max_steps_per_episode)

    # Calculate expected returns
    returns = get_expected_return(rewards,values,step, gamma)

    # Convert training data to appropriate TF tensor shapes
    action_probs, values, returns = [
        tf.expand_dims(x, 1) for x in [action_probs, values, returns]]

    # Calculating loss values to update our network
    loss = compute_loss(action_probs, values, returns)

  # Compute the gradients from the loss
  grads = tape.gradient(loss, model.trainable_variables)

  # Apply the gradients to the model's parameters
  optimizer.apply_gradients(zip(grads, model.trainable_variables))

  episode_reward = tf.math.reduce_sum(rewards)

  return episode_reward
```

[ ]:
```
import tqdm
```

[ ]:
```
%%time

min_episodes_criterion = 100
max_episodes = 700
max_steps_per_episode = 500


reward_threshold = 475
running_reward = 0

step = 10

gamma = 0.99

# Keep last episodes reward
episodes_reward: collections.deque = collections.
 ↪deque(maxlen=min_episodes_criterion)

with tqdm.trange(max_episodes) as t:
  for i in t:
    initial_state = tf.constant(env.reset(), dtype=tf.float32)
    episode_reward = int(train_step(
        initial_state, model, optimizer, gamma,step, max_steps_per_episode))

    episodes_reward.append(episode_reward)
    running_reward = statistics.mean(episodes_reward)

    t.set_description(f'Episode {i}')
    t.set_postfix(
        episode_reward=episode_reward, running_reward=running_reward)

    # Show average episode reward every 10 episodes
    if i % 10 == 0:
      pass # print(f'Episode {i}: average reward: {avg_reward}')

    if running_reward > reward_threshold and i >= min_episodes_criterion:
        break

print(f'\nSolved at episode {i}: average reward: {running_reward:.2f}!')
```

[ ]:
```
min_episodes_criterion = 100
max_episodes = 700
max_steps_per_episode = 500


reward_threshold = 475
running_reward = 0
```

```
step = 10

gamma = 0.99

rewards_ep = []

for run in range(10):

  optimizer = tf.keras.optimizers.Adam(learning_rate=0.01)

  @tf.function
  def train_step(
      initial_state: tf.Tensor,
      model: tf.keras.Model,
      optimizer: tf.keras.optimizers.Optimizer,
      gamma: float,
      step: int,
      max_steps_per_episode: int) -> tf.Tensor:
    """Runs a model training step."""

    with tf.GradientTape() as tape:

      # Run the model for one episode to collect training data
      action_probs, values, rewards = run_episode(
          initial_state, model, max_steps_per_episode)

      # Calculate expected returns
      returns = get_expected_return(rewards,values,step, gamma)

      # Convert training data to appropriate TF tensor shapes
      action_probs, values, returns = [
          tf.expand_dims(x, 1) for x in [action_probs, values, returns]]

      # Calculating loss values to update our network
      loss = compute_loss(action_probs, values, returns)

    # Compute the gradients from the loss
    grads = tape.gradient(loss, model.trainable_variables)

    # Apply the gradients to the model's parameters
    optimizer.apply_gradients(zip(grads, model.trainable_variables))

    episode_reward = tf.math.reduce_sum(rewards)

    return episode_reward
```

```python
  env = gym.make("CartPole-v1")

  model = ActorCritic(num_actions, num_hidden_units)

  print(run)
  # Keep last episodes reward
  episodes_reward: collections.deque = collections.
↪deque(maxlen=min_episodes_criterion)

  reward_list = []
  with tqdm.trange(max_episodes) as t:
    for i in t:
      initial_state = tf.constant(env.reset(), dtype=tf.float32)
      episode_reward = int(train_step(
          initial_state, model, optimizer, gamma,step, max_steps_per_episode))

      episodes_reward.append(episode_reward)
      reward_list.append(episode_reward)
      running_reward = statistics.mean(episodes_reward)

      t.set_description(f'Episode {i}')
      t.set_postfix(
          episode_reward=episode_reward, running_reward=running_reward)

      # Show average episode reward every 10 episodes
      if i % 10 == 0:
        pass # print(f'Episode {i}: average reward: {avg_reward}')

  rewards_ep.append(reward_list)


#print(f'\nSolved at episode {i}: average reward: {running_reward:.2f}!')
```

```
0

Episode 699: 100%|      | 700/700 [01:08<00:00, 10.28it/s,
episode_reward=500, running_reward=360]

1

Episode 699: 100%|      | 700/700 [01:42<00:00,  6.80it/s,
episode_reward=500, running_reward=387]

2

Episode 699: 100%|      | 700/700 [01:49<00:00,  6.38it/s,
episode_reward=500, running_reward=479]

3
```

```
Episode 699: 100%|      | 700/700 [02:05<00:00,  5.57it/s,
episode_reward=500, running_reward=480]

4

Episode 699: 100%|      | 700/700 [01:08<00:00, 10.24it/s,
episode_reward=218, running_reward=183]

5

Episode 699: 100%|      | 700/700 [01:18<00:00,  8.89it/s,
episode_reward=98, running_reward=110]

6

Episode 699: 100%|      | 700/700 [01:44<00:00,  6.71it/s,
episode_reward=500, running_reward=356]

7

Episode 699: 100%|      | 700/700 [01:12<00:00,  9.63it/s,
episode_reward=15, running_reward=253]

8

Episode 699: 100%|      | 700/700 [01:23<00:00,  8.37it/s,
episode_reward=188, running_reward=149]

9

Episode 699: 100%|      | 700/700 [01:54<00:00,  6.10it/s,
episode_reward=14, running_reward=435]
```

[ ]:

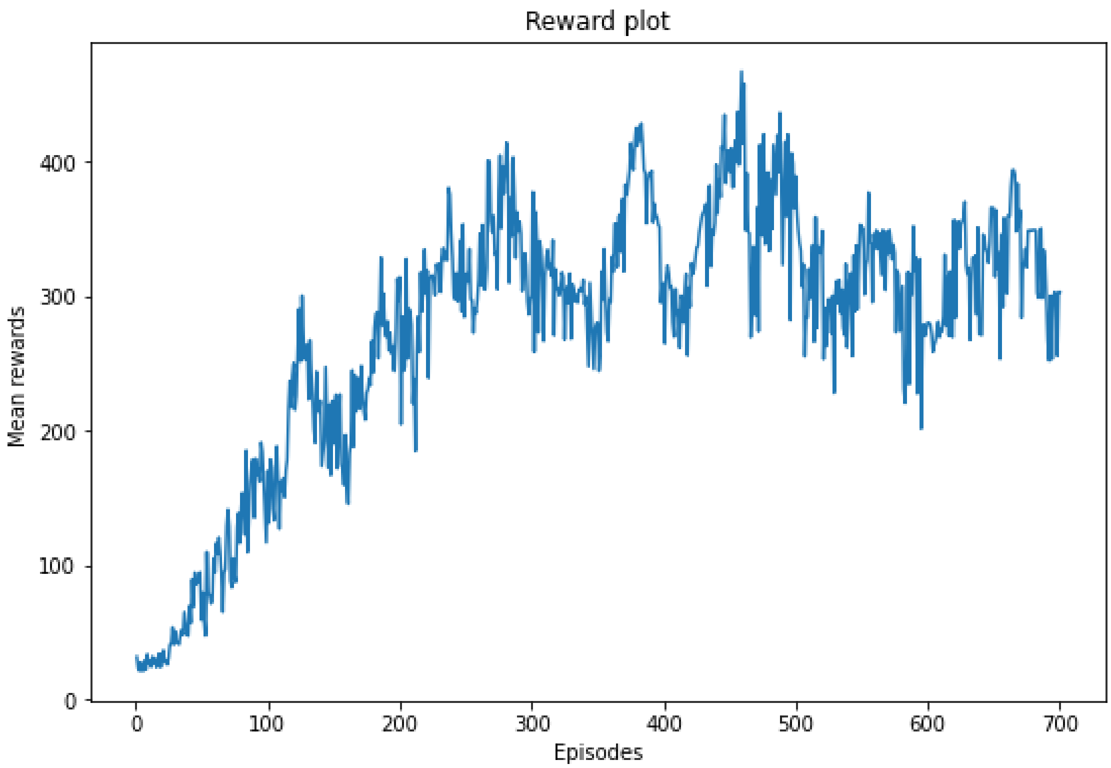
```
rewards_ep = np.array(rewards_ep)
mean=np.array([np.mean(rewards_ep, axis=0) for i in range(10)])
var = np.sqrt(np.sum((rewards_ep-mean)**2,axis=0))/10
```

[ ]:

```
plt.figure(figsize=(9,6))
plt.xlabel('Episodes')
plt.ylabel('Mean rewards ')
plt.title('Reward plot')
plt.plot(np.arange(1,len(mean[1])+1),mean[1])
```

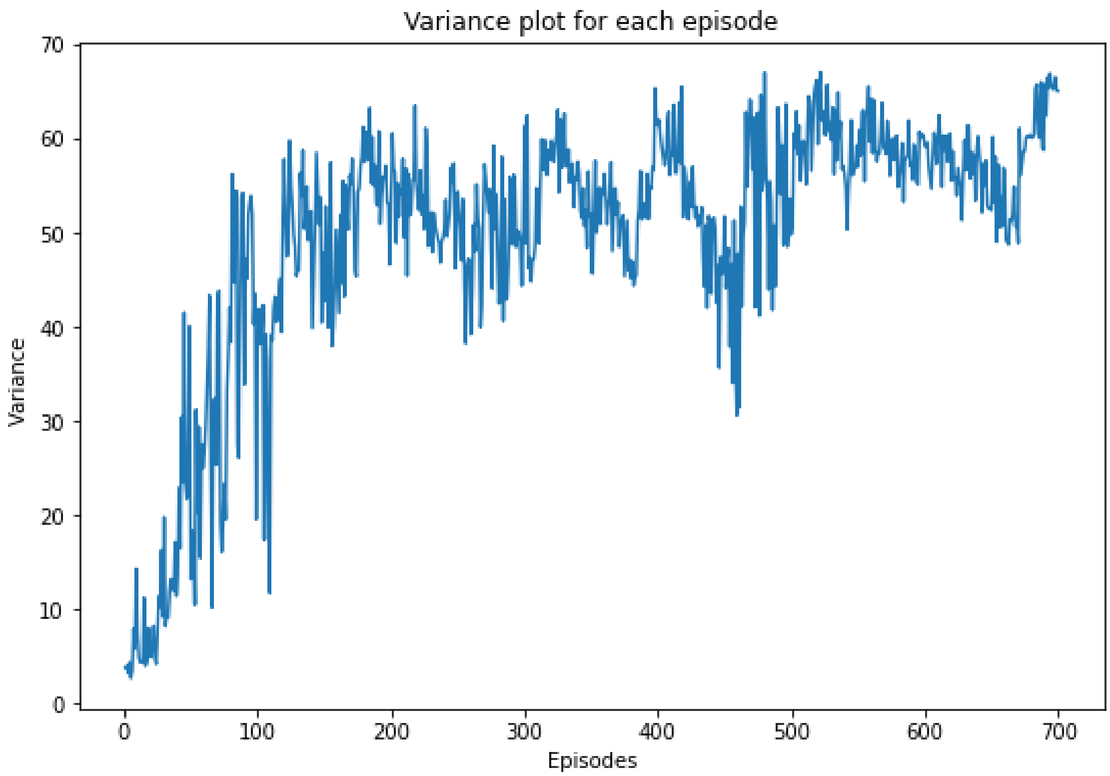
[ ]:

```
[<matplotlib.lines.Line2D at 0x7f7ef15ad510>]
```




```
[ ]: plt.figure(figsize=(9,6))
     plt.xlabel('Episodes')
     plt.ylabel('Variance ')
     plt.title('Variance plot for each episode')
     plt.plot(np.arange(1,len(var)+1),var)
```

```
[ ]: [<matplotlib.lines.Line2D at 0x7f7ef16eaa90>]
```


Variance plot for each episode
Variance
Episodes
0
10
20
30
40
50
60
70
0
100
200
300
400
500
600
700

RLPA2_actor_critic_acrobot

March 31, 2022

[2]:
```
'''
Installing packages for rendering the game on Colab
'''

!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
```

```
Requirement already satisfied: setuptools in /usr/local/lib/python3.7/dist-
packages (61.2.0)
```

[3]:
```
!pip install tensorflow-gpu
```

```
Collecting tensorflow-gpu
  Downloading tensorflow_gpu-2.8.0-cp37-cp37m-manylinux2010_x86_64.whl (497.5
MB)
     |                 | 497.5 MB 25 kB/s
Requirement already satisfied: absl-py>=0.4.0 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (1.0.0)
Requirement already satisfied: google-pasta>=0.1.1 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (0.2.0)
Requirement already satisfied: numpy>=1.20 in /usr/local/lib/python3.7/dist-
packages (from tensorflow-gpu) (1.21.5)
Requirement already satisfied: termcolor>=1.1.0 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (1.1.0)
Requirement already satisfied: libclang>=9.0.1 in /usr/local/lib/python3.7/dist-
packages (from tensorflow-gpu) (13.0.0)
Requirement already satisfied: grpcio<2.0,>=1.24.3 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (1.44.0)
Requirement already satisfied: six>=1.12.0 in /usr/local/lib/python3.7/dist-
packages (from tensorflow-gpu) (1.15.0)
Collecting tf-estimator-nightly==2.8.0.dev2021122109
  Downloading tf_estimator_nightly-2.8.0.dev2021122109-py2.py3-none-any.whl (462
```

```
kB)
     |                  | 462 kB 49.3 MB/s
Requirement already satisfied: typing-extensions>=3.6.6 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (3.10.0.2)
Requirement already satisfied: setuptools in /usr/local/lib/python3.7/dist-
packages (from tensorflow-gpu) (61.2.0)
Requirement already satisfied: keras-preprocessing>=1.1.1 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (1.1.2)
Requirement already satisfied: tensorflow-io-gcs-filesystem>=0.23.1 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (0.24.0)
Requirement already satisfied: keras<2.9,>=2.8.0rc0 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (2.8.0)
Requirement already satisfied: wrapt>=1.11.0 in /usr/local/lib/python3.7/dist-
packages (from tensorflow-gpu) (1.14.0)
Requirement already satisfied: astunparse>=1.6.0 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (1.6.3)
Requirement already satisfied: flatbuffers>=1.12 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (2.0)
Requirement already satisfied: h5py>=2.9.0 in /usr/local/lib/python3.7/dist-
packages (from tensorflow-gpu) (3.1.0)
Requirement already satisfied: opt-einsum>=2.3.2 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (3.3.0)
Requirement already satisfied: gast>=0.2.1 in /usr/local/lib/python3.7/dist-
packages (from tensorflow-gpu) (0.5.3)
Requirement already satisfied: protobuf>=3.9.2 in /usr/local/lib/python3.7/dist-
packages (from tensorflow-gpu) (3.17.3)
Requirement already satisfied: tensorboard<2.9,>=2.8 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (2.8.0)
Requirement already satisfied: wheel<1.0,>=0.23.0 in
/usr/local/lib/python3.7/dist-packages (from astunparse>=1.6.0->tensorflow-gpu)
(0.37.1)
Requirement already satisfied: cached-property in /usr/local/lib/python3.7/dist-
packages (from h5py>=2.9.0->tensorflow-gpu) (1.5.2)
Requirement already satisfied: tensorboard-plugin-wit>=1.6.0 in
/usr/local/lib/python3.7/dist-packages (from tensorboard<2.9,>=2.8->tensorflow-
gpu) (1.8.1)
Requirement already satisfied: tensorboard-data-server<0.7.0,>=0.6.0 in
/usr/local/lib/python3.7/dist-packages (from tensorboard<2.9,>=2.8->tensorflow-
gpu) (0.6.1)
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.7/dist-
packages (from tensorboard<2.9,>=2.8->tensorflow-gpu) (3.3.6)
Requirement already satisfied: requests<3,>=2.21.0 in
/usr/local/lib/python3.7/dist-packages (from tensorboard<2.9,>=2.8->tensorflow-
gpu) (2.23.0)
Requirement already satisfied: werkzeug>=0.11.15 in
/usr/local/lib/python3.7/dist-packages (from tensorboard<2.9,>=2.8->tensorflow-
gpu) (1.0.1)
Requirement already satisfied: google-auth-oauthlib<0.5,>=0.4.1 in
```

```
/usr/local/lib/python3.7/dist-packages (from tensorboard<2.9,>=2.8->tensorflow-
gpu) (0.4.6)
Requirement already satisfied: google-auth<3,>=1.6.3 in
/usr/local/lib/python3.7/dist-packages (from tensorboard<2.9,>=2.8->tensorflow-
gpu) (1.35.0)
Requirement already satisfied: cachetools<5.0,>=2.0.0 in
/usr/local/lib/python3.7/dist-packages (from google-
auth<3,>=1.6.3->tensorboard<2.9,>=2.8->tensorflow-gpu) (4.2.4)
Requirement already satisfied: rsa<5,>=3.1.4 in /usr/local/lib/python3.7/dist-
packages (from google-auth<3,>=1.6.3->tensorboard<2.9,>=2.8->tensorflow-gpu)
(4.8)
Requirement already satisfied: pyasn1-modules>=0.2.1 in
/usr/local/lib/python3.7/dist-packages (from google-
auth<3,>=1.6.3->tensorboard<2.9,>=2.8->tensorflow-gpu) (0.2.8)
Requirement already satisfied: requests-oauthlib>=0.7.0 in
/usr/local/lib/python3.7/dist-packages (from google-auth-
oauthlib<0.5,>=0.4.1->tensorboard<2.9,>=2.8->tensorflow-gpu) (1.3.1)
Requirement already satisfied: importlib-metadata>=4.4 in
/usr/local/lib/python3.7/dist-packages (from
markdown>=2.6.8->tensorboard<2.9,>=2.8->tensorflow-gpu) (4.11.3)
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.7/dist-
packages (from importlib-
metadata>=4.4->markdown>=2.6.8->tensorboard<2.9,>=2.8->tensorflow-gpu) (3.7.0)
Requirement already satisfied: pyasn1<0.5.0,>=0.4.6 in
/usr/local/lib/python3.7/dist-packages (from pyasn1-modules>=0.2.1->google-
auth<3,>=1.6.3->tensorboard<2.9,>=2.8->tensorflow-gpu) (0.4.8)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-
packages (from requests<3,>=2.21.0->tensorboard<2.9,>=2.8->tensorflow-gpu)
(2.10)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in
/usr/local/lib/python3.7/dist-packages (from
requests<3,>=2.21.0->tensorboard<2.9,>=2.8->tensorflow-gpu) (1.24.3)
Requirement already satisfied: chardet<4,>=3.0.2 in
/usr/local/lib/python3.7/dist-packages (from
requests<3,>=2.21.0->tensorboard<2.9,>=2.8->tensorflow-gpu) (3.0.4)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.7/dist-packages (from
requests<3,>=2.21.0->tensorboard<2.9,>=2.8->tensorflow-gpu) (2021.10.8)
Requirement already satisfied: oauthlib>=3.0.0 in /usr/local/lib/python3.7/dist-
packages (from requests-oauthlib>=0.7.0->google-auth-
oauthlib<0.5,>=0.4.1->tensorboard<2.9,>=2.8->tensorflow-gpu) (3.2.0)
Installing collected packages: tf-estimator-nightly, tensorflow-gpu
Successfully installed tensorflow-gpu-2.8.0 tf-estimator-
nightly-2.8.0.dev2021122109
```

[4]:
```
'''
A bunch of imports, you don't have to worry about these
```

```
'''
import collections
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 import Monitor
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
from typing import Any, List, Sequence, Tuple
from tensorflow.keras import layers
import statistics
import tqdm
```

[ ]:

```
'''
Please refer to the first tutorial for more details on the specifics of␣
↪environments
We've only added important commands you might find useful for experiments.
'''

'''
List of example environments
(Source - https://gym.openai.com/envs/#classic_control)

'Acrobot-v1'
'CartPole-v0'
'MountainCar-v0'
'''

env = gym.make('Acrobot-v1')
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("----")

'''
# Understanding State, Action, Reward Dynamics

The agent decides an action to take depending on the state.

The Environment keeps a variable specifically for the current state.
- Everytime an action is passed to the environment, it calculates the new state␣
↪and updates the current state variable.
- It returns the new current state and reward for the agent to take the next␣
↪action

'''

state = env.reset()
''' This returns the initial state (when environment is reset) '''

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

action = env.action_space.sample()
''' We take a random action now '''

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

next_state, reward, done, info = env.step(action)
''' env.step is used to calculate new state and obtain reward based on old␣
↪state and action taken  '''

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

```
6
3
1
----
[ 0.99603073 -0.08901003  0.99567135  0.09294385  0.02653819 -0.04199653]
----
```

```
2
----
[ 0.9964048  -0.08472003  0.99483904  0.10146566  0.01598951  0.12664371]
-1.0
False
{}
----
```

[ ]:
```python
class ActorCriticModel(tf.keras.Model):
    """
    Defining policy and value networkss
    """
    def __init__(self, action_size, n_hidden1=512, n_hidden2=512):
        super(ActorCriticModel, self).__init__()

        #Hidden Layer 1
        self.fc1 = tf.keras.layers.Dense(n_hidden1, activation='relu')
        #Hidden Layer 2
        self.fc2 = tf.keras.layers.Dense(n_hidden2, activation='relu')

        #Output Layer for policy
        self.pi_out = tf.keras.layers.Dense(action_size, activation='softmax')
        #Output Layer for state-value
        self.v_out = tf.keras.layers.Dense(1)

    def call(self, state):
        """
        Computes policy distribution and state-value for a given state
        """
        layer1 = self.fc1(state)
        layer2 = self.fc2(layer1)

        pi = self.pi_out(layer2)
        v = self.v_out(layer2)

        return pi, v
```

[ ]:
```python
class Agent:
    """
    Agent class
    """
    def __init__(self, action_size, lr=0.1, gamma=0.99, seed = 85):
        self.gamma = gamma
        self.ac_model = ActorCriticModel(action_size=action_size)
        self.ac_model.compile(tf.keras.optimizers.Adam(learning_rate=lr))
        np.random.seed(seed)
```

```python
    def sample_action(self, state):
        """
        Given a state, compute the policy distribution over all actions and␣
↪sample one action
        """
        pi,_ = self.ac_model(state)

        action_probabilities = tfp.distributions.Categorical(probs=pi)
        sample = action_probabilities.sample()

        return int(sample.numpy()[0])

    def actor_loss(self, action, pi, delta):
        """
        Compute Actor Loss
        """
        return -tf.math.log(pi[0,action]) * delta

    def critic_loss(self,delta):
        """
        Critic loss aims to minimize TD error
        """
        return delta**2

    @tf.function
    def learn(self, state, action, reward, next_state, done):
        """
        For a given transition (s,a,s',r) update the paramters by computing the
        gradient of the total loss
        """
        with tf.GradientTape(persistent=True) as tape:
            pi, V_s = self.ac_model(state)
            _, V_s_next = self.ac_model(next_state)

            V_s = tf.squeeze(V_s)
            V_s_next = tf.squeeze(V_s_next)


            #### TO DO: Write the equation for delta (TD error)
            ## Write code below
            delta = reward + self.gamma*V_s_next-V_s

            loss_a = self.actor_loss(action, pi, delta)
            loss_c =self.critic_loss(delta)
            loss_total = loss_a + loss_c

        gradient = tape.gradient(loss_total, self.ac_model.trainable_variables)
```

```
        self.ac_model.optimizer.apply_gradients(zip(gradient, self.ac_model.
 ↪trainable_variables))
```

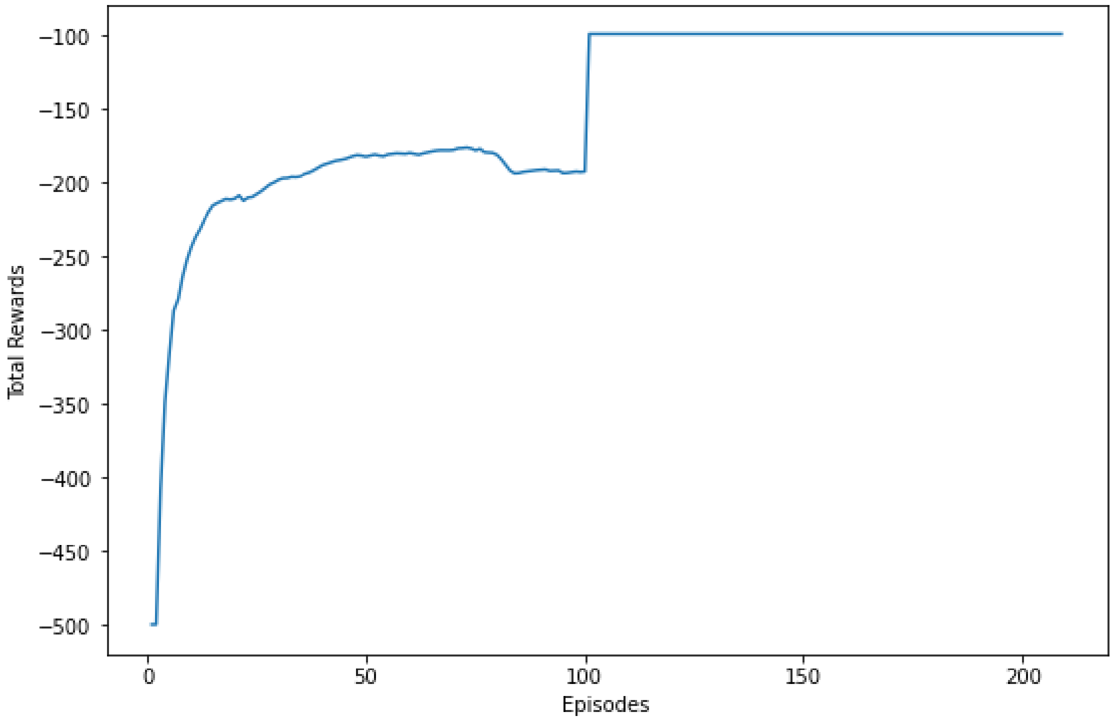
[ ]:
```
env = gym.make('Acrobot-v1')

#Initializing Agent
agent = Agent(lr=1e-4, action_size=env.action_space.n)
#Number of episodes
episodes = 300
tf.compat.v1.reset_default_graph()

reward_list = []
average_reward_list = []
begin_time = datetime.datetime.now()

for ep in range(1, episodes + 1):
    state = env.reset().reshape(1,-1)
    done = False
    ep_rew = 0
    while not done:
        action = agent.sample_action(state) ##Sample Action
        next_state, reward, done, info = env.step(action) ##Take action
        next_state = next_state.reshape(1,-1)
        ep_rew += reward  ##Updating episode reward
        agent.learn(state, action, reward, next_state, done) ##Update Parameters
        state = next_state ##Updating State
    reward_list.append(ep_rew)

    if ep % 10 == 0:
        avg_rew = np.mean(reward_list[-10:])
        print('Episode ', ep, 'Reward %f' % ep_rew, 'Average Reward %f' %␣
 ↪avg_rew)

    if ep % 100:
        avg_100 =  np.mean(reward_list[-100:])
        if avg_100 > -100.0:
            print('Stopped at Episode ',ep)
            break

time_taken = datetime.datetime.now() - begin_time
print(time_taken)
```

```
Episode  10 Reward -164.000000 Average Reward -243.900000
Episode  20 Reward -198.000000 Average Reward -178.600000
Episode  30 Reward -148.000000 Average Reward -173.000000
Episode  40 Reward -120.000000 Average Reward -157.800000
Episode  50 Reward -217.000000 Average Reward -159.600000
```

```
Episode  60 Reward -144.000000 Average Reward -167.400000
Episode  70 Reward -157.000000 Average Reward -165.600000
Episode  80 Reward -290.000000 Average Reward -205.200000
Episode  90 Reward -177.000000 Average Reward -272.700000
Episode  100 Reward -157.000000 Average Reward -202.200000
Episode  110 Reward -368.000000 Average Reward -198.000000
Episode  120 Reward -176.000000 Average Reward -175.300000
Episode  130 Reward -71.000000 Average Reward -131.700000
Episode  140 Reward -83.000000 Average Reward -90.700000
Episode  150 Reward -90.000000 Average Reward -78.700000
Episode  160 Reward -77.000000 Average Reward -83.000000
Episode  170 Reward -74.000000 Average Reward -82.000000
Episode  180 Reward -78.000000 Average Reward -90.200000
Episode  190 Reward -108.000000 Average Reward -87.300000
Episode  200 Reward -70.000000 Average Reward -82.400000
Episode  210 Reward -72.000000 Average Reward -91.800000
Stopped at Episode  210
0:05:06.372756
```

[ ]:
```
mov_avg=[]
for i in range(1,len(reward_list)):
  if i > 100:
    mov_avg.append(np.mean(reward_list[-100:]))
  else:
    mov_avg.append(np.mean(reward_list[:i]))

plt.figure(figsize=(9,6))
plt.xlabel('Episodes')
plt.ylabel('Total Rewards')
plt.plot(np.arange(1,len(mov_avg)+1),mov_avg)
```

[ ]:
```
[<matplotlib.lines.Line2D at 0x7f466292a610>]
```



[ ]:
```
from tqdm import tqdm

episodes = 300
env = gym.make('Acrobot-v1')
agent = Agent(lr=1e-4, action_size=env.action_space.n)
#avg_reward_eps = np.zeros(episodes)
reward_eps = []
for run in range(10):
  print('Run:%f' % run)
#Initializing Agent

  tf.compat.v1.reset_default_graph()

  reward_list = []
  average_reward_list = []
  nsteps_list = []
  begin_time = datetime.datetime.now()

  for ep in tqdm(range(1, episodes + 1)):
      state = env.reset().reshape(1,-1)
      done = False
      ep_rew = 0
      nsteps = 0
```

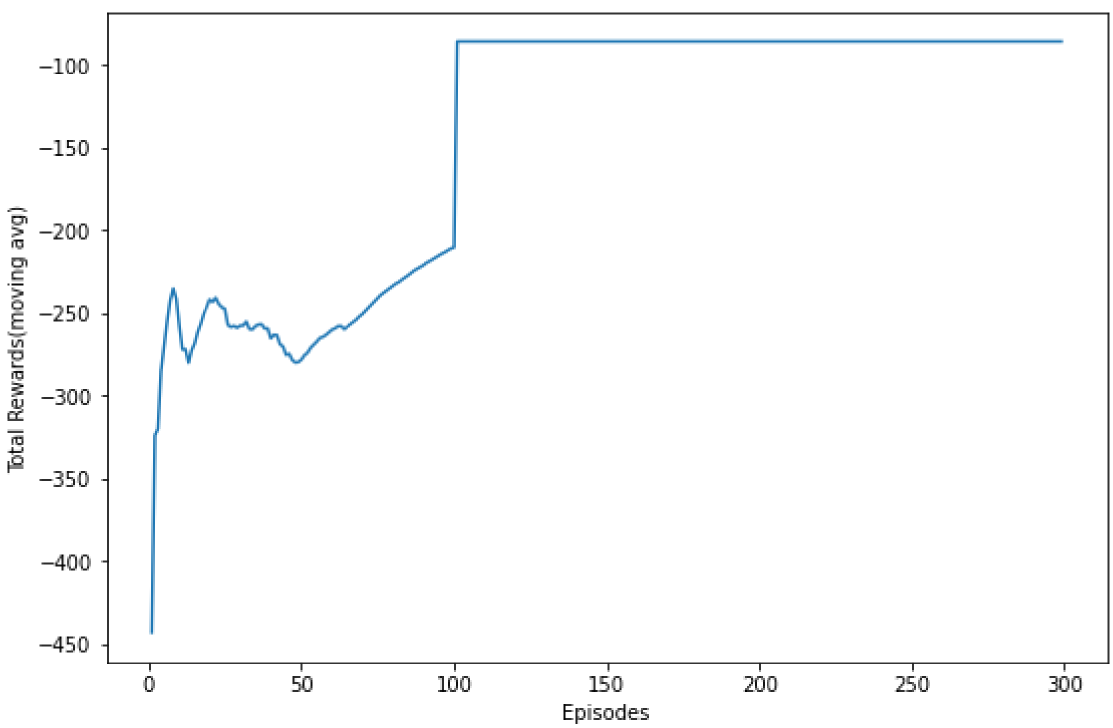
```python
    while not done:
        action = agent.sample_action(state) ##Sample Action
        next_state, reward, done, info = env.step(action) ##Take action
        next_state = next_state.reshape(1,-1)
        ep_rew += reward  ##Updating episode reward
        agent.learn(state, action, reward, next_state, done) ##Update␣
↪Parameters
        state = next_state ##Updating State
        nsteps +=1
    nsteps_list.append(nsteps)
    reward_list.append(ep_rew)



print(np.mean(reward_list[-100:]))
reward_eps.append(np.array(reward_list))
```

```
Run:0.000000

100%|      | 300/300 [06:59<00:00,  1.40s/it]

-85.76
Run:1.000000

100%|      | 300/300 [04:29<00:00,  1.11it/s]

-88.81
Run:2.000000

100%|      | 300/300 [04:32<00:00,  1.10it/s]

-89.89
Run:3.000000

100%|      | 300/300 [04:25<00:00,  1.13it/s]

-86.86
Run:4.000000

100%|      | 300/300 [04:14<00:00,  1.18it/s]

-79.3
Run:5.000000

100%|      | 300/300 [04:26<00:00,  1.12it/s]

-92.05
Run:6.000000

100%|      | 300/300 [04:29<00:00,  1.11it/s]

-93.57
Run:7.000000

100%|      | 300/300 [08:26<00:00,  1.69s/it]
```

```
-317.61
Run:8.000000

100%|      | 300/300 [24:40<00:00,  4.93s/it]

-500.0
Run:9.000000

100%|      | 300/300 [24:54<00:00,  4.98s/it]

-500.0
```

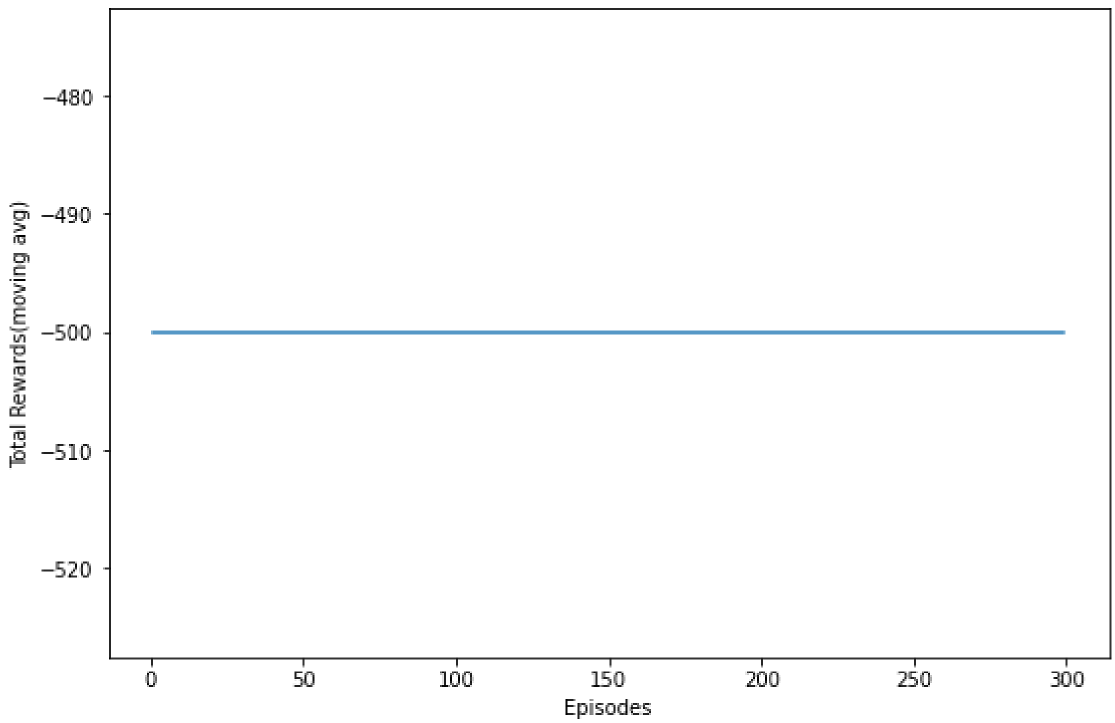
[ ]:
```python
mov_avg=[]
for i in range(1,len(reward_eps[-3])):
  if i > 100:
    mov_avg.append(np.mean(reward_eps[0][-100:]))
  else:
    mov_avg.append(np.mean(reward_eps[0][:i]))

plt.figure(figsize=(9,6))
plt.xlabel('Episodes')
plt.ylabel('Total Rewards(moving avg)')
plt.plot(np.arange(1,len(mov_avg)+1),mov_avg)
```

[ ]:
```
[<matplotlib.lines.Line2D at 0x7f46615b5090>]
```



[ ]:
```
mov_avg=[]
for i in range(1,len(reward_eps[-3])):
  if i > 100:
    mov_avg.append(np.mean(reward_eps[-2][-100:]))
  else:
    mov_avg.append(np.mean(reward_eps[-2][:i]))

plt.figure(figsize=(9,6))
plt.xlabel('Episodes')
plt.ylabel('Total Rewards(moving avg)')
plt.plot(np.arange(1,len(mov_avg)+1),mov_avg)
```

[ ]:
```
[<matplotlib.lines.Line2D at 0x7f4661533b50>]
```

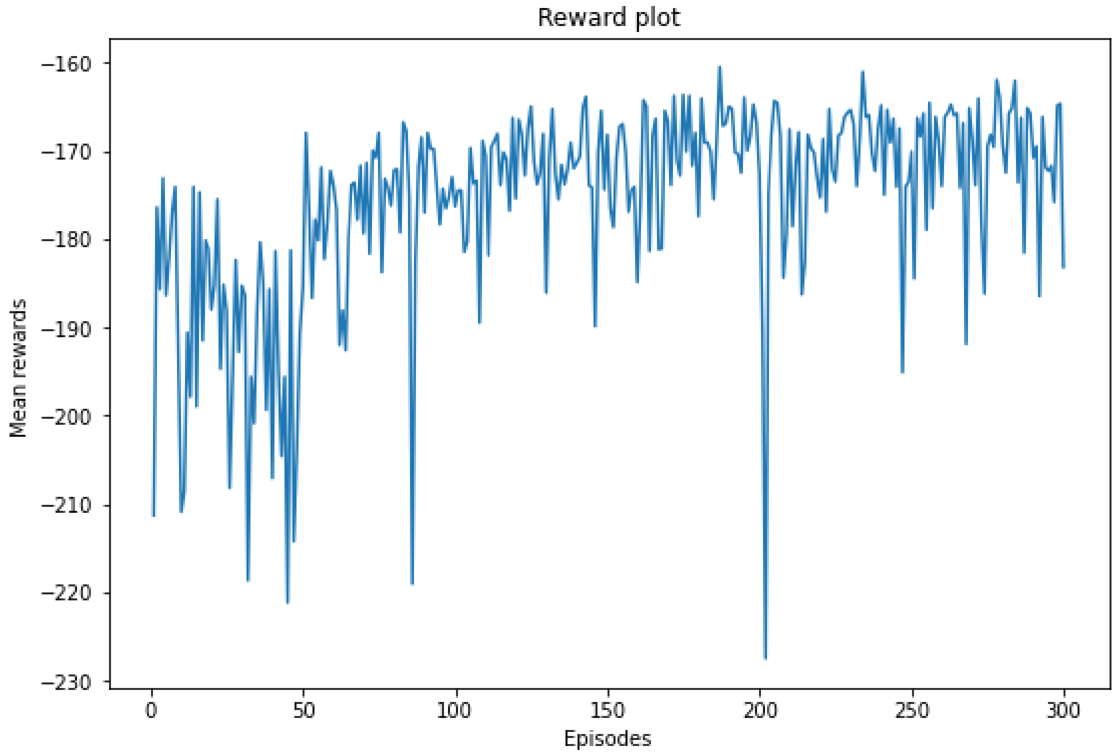


[ ]:
```
reward_eps = np.array(reward_eps)
mean=np.array([np.mean(reward_eps, axis=0) for i in range(10)])
var = np.sqrt(np.sum((reward_eps-mean)**2,axis=0))/10
```

[ ]:
```
plt.figure(figsize=(9,6))
plt.xlabel('Episodes')
plt.ylabel('Mean rewards ')
plt.title('Reward plot')
```

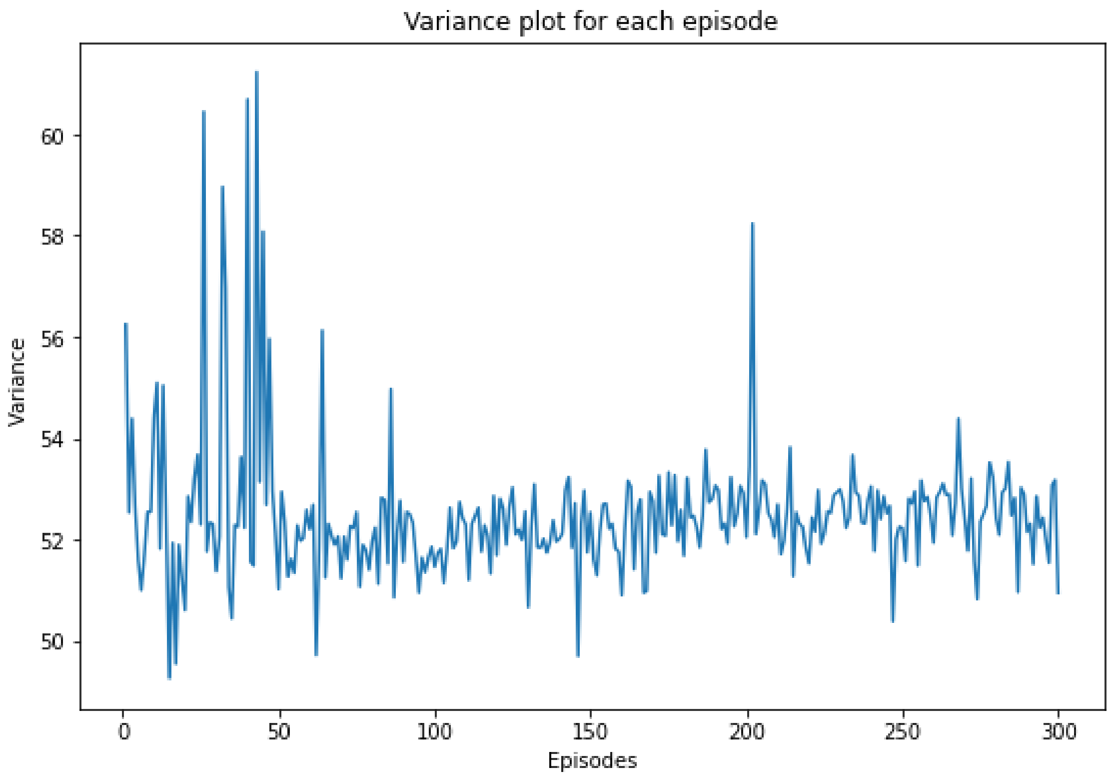
```
plt.plot(np.arange(1,len(mean[1])+1),mean[1])
```

[ ]: [<matplotlib.lines.Line2D at 0x7f46628504d0>]



[ ]:
```
plt.figure(figsize=(9,6))
plt.xlabel('Episodes')
plt.ylabel('Variance ')
plt.title('Variance plot for each episode')
plt.plot(np.arange(1,len(var)+1),var)
```

[ ]: [<matplotlib.lines.Line2D at 0x7f466271dd50>]



Full return

[ ]:

[114]:
```
# Create the environment
env = gym.make("Acrobot-v1")

# Set seed for experiment reproducibility
seed = 52
env.seed(seed)
tf.random.set_seed(seed)
np.random.seed(seed)

# Small epsilon value for stabilizing division operations
eps = np.finfo(np.float32).eps.item()
```

[115]:
```
class ActorCritic(tf.keras.Model):
  """Combined actor-critic network."""

  def __init__(
      self,
      num_actions: int,
```

```
      num_hidden_units: int,
      num_hidden_units1: int,
      num_hidden_units2: int,
      num_hidden_units3: int,
      num_hidden_units4: int):
    """Initialize."""
    super().__init__()

    self.common1 = layers.Dense(num_hidden_units, activation="relu")
    self.common2 = layers.Dense(num_hidden_units1, activation="relu")
    #self.common3 = layers.Dense(num_hidden_units2, activation="relu")
    self.actor = layers.Dense(num_actions)
    self.critic = layers.Dense(1)

  def call(self, inputs: tf.Tensor) -> Tuple[tf.Tensor, tf.Tensor]:
    x0 = self.common1(inputs)
    x1 = self.common2(x0)
    #x2 = self.common3(x1)
    return self.actor(x1), self.critic(x1)
```

[116]:
```
num_actions = env.action_space.n  # 2
num_hidden_units = 512
num_hidden_units1 = 1024
num_hidden_units2 = 1024
num_hidden_units3 = 1024
num_hidden_units4 = 512
model = ActorCritic(num_actions, num_hidden_units, num_hidden_units1,␣
 ↪num_hidden_units2, num_hidden_units3, num_hidden_units4)
```

[117]:
```
# Wrap OpenAI Gym's `env.step` call as an operation in a TensorFlow function.
# This would allow it to be included in a callable TensorFlow graph.

def env_step(action: np.ndarray) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
  """Returns state, reward and done flag given an action."""

  state, reward, done, _ = env.step(action)
  return (state.astype(np.float32),
          np.array(reward, np.int32),
          np.array(done, np.int32))


def tf_env_step(action: tf.Tensor) -> List[tf.Tensor]:
  return tf.numpy_function(env_step, [action],
                           [tf.float32, tf.int32, tf.int32])
```

[118]:
```
def run_episode(
    initial_state: tf.Tensor,
```

```
    model: tf.keras.Model,
    max_steps: int) -> Tuple[tf.Tensor, tf.Tensor, tf.Tensor]:
  """Runs a single episode to collect training data."""

  action_probs = tf.TensorArray(dtype=tf.float32, size=0, dynamic_size=True)
  values = tf.TensorArray(dtype=tf.float32, size=0, dynamic_size=True)
  rewards = tf.TensorArray(dtype=tf.int32, size=0, dynamic_size=True)

  initial_state_shape = initial_state.shape
  state = initial_state

  for t in tf.range(max_steps):
    # Convert state into a batched tensor (batch size = 1)
    state = tf.expand_dims(state, 0)

    # Run the model and to get action probabilities and critic value
    action_logits_t, value = model(state)

    # Sample next action from the action probability distribution
    action = tf.random.categorical(action_logits_t, 1)[0, 0]
    action_probs_t = tf.nn.softmax(action_logits_t)

    # Store critic values
    values = values.write(t, tf.squeeze(value))

    # Store log probability of the action chosen
    action_probs = action_probs.write(t, action_probs_t[0, action])

    # Apply action to the environment to get next state and reward
    state, reward, done = tf_env_step(action)
    state.set_shape(initial_state_shape)

    # Store reward
    rewards = rewards.write(t, reward)

    if tf.cast(done, tf.bool):
      break

  action_probs = action_probs.stack()
  values = values.stack()
  rewards = rewards.stack()

  return action_probs, values, rewards
```

[119]:
```
def get_expected_return(
    rewards: tf.Tensor,
    gamma: float,
```

```
    standardize: bool = True) -> tf.Tensor:
  """Compute expected returns per timestep."""

  n = tf.shape(rewards)[0]
  returns = tf.TensorArray(dtype=tf.float32, size=n)

  # Start from the end of `rewards` and accumulate reward sums
  # into the `returns` array
  rewards = tf.cast(rewards[::-1], dtype=tf.float32)
  discounted_sum = tf.constant(0.0)
  discounted_sum_shape = discounted_sum.shape
  for i in tf.range(n):
    reward = rewards[i]
    discounted_sum = reward + gamma * discounted_sum
    discounted_sum.set_shape(discounted_sum_shape)
    returns = returns.write(i, discounted_sum)
  returns = returns.stack()[::-1]

  if standardize:
    returns = ((returns - tf.math.reduce_mean(returns)) /
               (tf.math.reduce_std(returns) + eps))

  return returns
```

[120]:
```
huber_loss = tf.keras.losses.Huber(reduction=tf.keras.losses.Reduction.SUM)

def compute_loss(
    action_probs: tf.Tensor,
    values: tf.Tensor,
    returns: tf.Tensor) -> tf.Tensor:
  """Computes the combined actor-critic loss."""

  advantage = returns - values

  action_log_probs = tf.math.log(action_probs)
  actor_loss = -tf.math.reduce_sum(action_log_probs * advantage)

  critic_loss = huber_loss(values, returns)

  return actor_loss + critic_loss
```

[121]:
```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)


@tf.function
def train_step(
    initial_state: tf.Tensor,
```

```
    model: tf.keras.Model,
    optimizer: tf.keras.optimizers.Optimizer,
    gamma: float,
    max_steps_per_episode: int) -> tf.Tensor:
  """Runs a model training step."""

  with tf.GradientTape() as tape:

    # Run the model for one episode to collect training data
    action_probs, values, rewards = run_episode(
        initial_state, model, max_steps_per_episode)

    # Calculate expected returns
    returns = get_expected_return(rewards, gamma)

    # Convert training data to appropriate TF tensor shapes
    action_probs, values, returns = [
        tf.expand_dims(x, 1) for x in [action_probs, values, returns]]

    # Calculating loss values to update our network
    loss = compute_loss(action_probs, values, returns)

  # Compute the gradients from the loss
  grads = tape.gradient(loss, model.trainable_variables)

  # Apply the gradients to the model's parameters
  optimizer.apply_gradients(zip(grads, model.trainable_variables))

  episode_reward = tf.math.reduce_sum(rewards)

  return episode_reward
```

[122]:
```
import tqdm
```

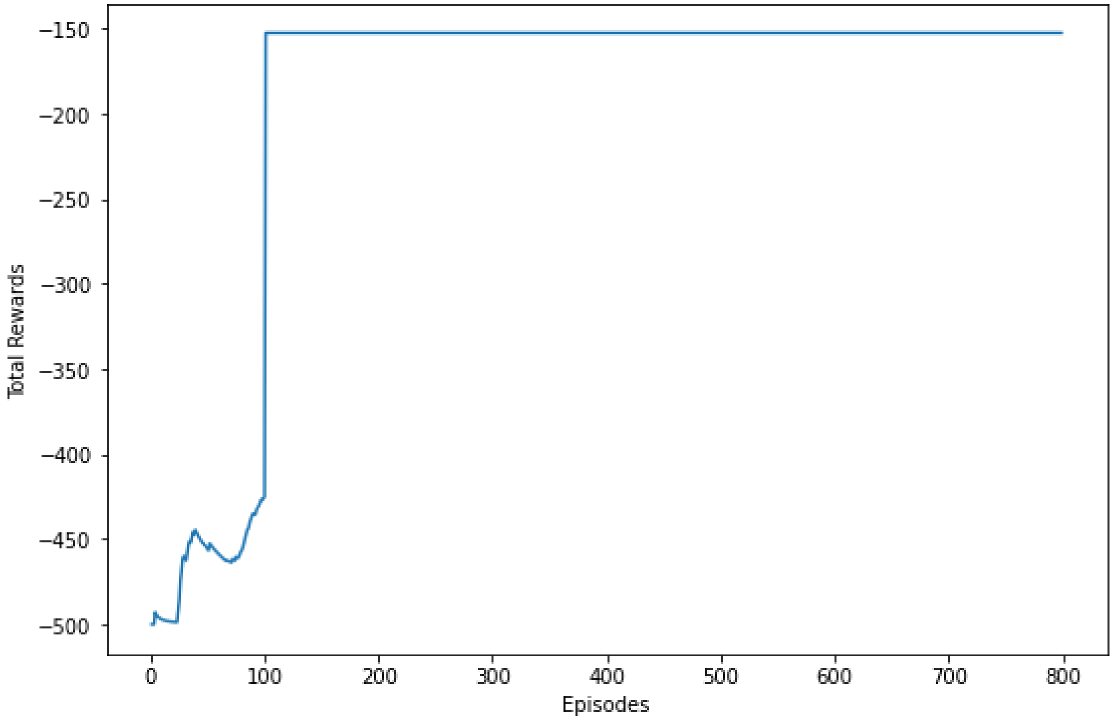
[123]:
```
%%time

min_episodes_criterion = 100
max_episodes = 800
max_steps_per_episode = 500


reward_threshold = -100
running_reward = 0


gamma = 0.99
```

```python
# Keep last episodes reward
episodes_reward: collections.deque = collections.
↪deque(maxlen=min_episodes_criterion)
reward_list= []
with tqdm.trange(max_episodes) as t:
  for i in t:
    initial_state = tf.constant(env.reset(), dtype=tf.float32)
    episode_reward = int(train_step(
        initial_state, model, optimizer, gamma, max_steps_per_episode))

    episodes_reward.append(episode_reward)
    reward_list.append(episode_reward)
    running_reward = statistics.mean(episodes_reward)

    t.set_description(f'Episode {i}')
    t.set_postfix(
        episode_reward=episode_reward, running_reward=running_reward)

    # Show average episode reward every 10 episodes
    if i % 10 == 0:
      pass # print(f'Episode {i}: average reward: {avg_reward}')

    if running_reward > reward_threshold and i >= min_episodes_criterion:
        break

print(f'\nSolved at episode {i}: average reward: {running_reward:.2f}!')
```

```
Episode 799: 100%|      | 800/800 [04:35<00:00,  2.90it/s,
episode_reward=-132, running_reward=-153]


Solved at episode 799: average reward: -152.56!
CPU times: user 7min 33s, sys: 22.3 s, total: 7min 55s
Wall time: 4min 35s
```

[124]:
```python
mov_avg=[]
for i in range(1,len(reward_list)):
  if i > 100:
    mov_avg.append(np.mean(reward_list[-100:]))
  else:
    mov_avg.append(np.mean(reward_list[:i]))

plt.figure(figsize=(9,6))
plt.xlabel('Episodes')
plt.ylabel('Total Rewards')
plt.plot(np.arange(1,len(mov_avg)+1),mov_avg)
```

[124]: [<matplotlib.lines.Line2D at 0x7fd40678d9d0>]



[ ]:
```
rewards_ep = []

for run in range(10):

  optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)


  @tf.function
  def train_step(
      initial_state: tf.Tensor,
      model: tf.keras.Model,
      optimizer: tf.keras.optimizers.Optimizer,
      gamma: float,
      max_steps_per_episode: int) -> tf.Tensor:
    """Runs a model training step."""

    with tf.GradientTape() as tape:

      # Run the model for one episode to collect training data
      action_probs, values, rewards = run_episode(
          initial_state, model, max_steps_per_episode)
```

```
    # Calculate expected returns
    returns = get_expected_return(rewards,values, gamma)

    # Convert training data to appropriate TF tensor shapes
    action_probs, values, returns = [
        tf.expand_dims(x, 1) for x in [action_probs, values, returns]]

    # Calculating loss values to update our network
    loss = compute_loss(action_probs, values, returns)

  # Compute the gradients from the loss
  grads = tape.gradient(loss, model.trainable_variables)

  # Apply the gradients to the model's parameters
  optimizer.apply_gradients(zip(grads, model.trainable_variables))

  episode_reward = tf.math.reduce_sum(rewards)

  return episode_reward

env = gym.make("Acrobot-v1")

model = ActorCritic(num_actions, num_hidden_units, num_hidden_units1,␣
↪num_hidden_units2, num_hidden_units3, num_hidden_units4)


print(run)
episodes_reward: collections.deque = collections.
↪deque(maxlen=min_episodes_criterion)
rewards_list = []
with tqdm.trange(max_episodes) as t:
  for i in t:
    initial_state = tf.constant(env.reset(), dtype=tf.float32)
    episode_reward = int(train_step(
        initial_state, model, optimizer, gamma, max_steps_per_episode))

    episodes_reward.append(episode_reward)
    rewards_list.append(episode_reward)
    running_reward = statistics.mean(episodes_reward)

    t.set_description(f'Episode {i}')
    t.set_postfix(
        episode_reward=episode_reward, running_reward=running_reward)

    # Show average episode reward every 10 episodes
    if i % 10 == 0:
```

```
        pass # print(f'Episode {i}: average reward: {avg_reward}')

  rewards_ep.append(rewards_list)
```

[ ]:
```
rewards_ep = np.array(rewards_ep)
mean=np.array([np.mean(rewards_ep, axis=0) for i in range(10)])
var = np.sqrt(np.sum((rewards_ep-mean)**2,axis=0))/10
```

[ ]:
```
plt.figure(figsize=(9,6))
plt.xlabel('Episodes')
plt.ylabel('Variance ')
plt.title('Variance plot for each episode')
plt.plot(np.arange(1,len(var)+1),var)
```

n-step return

[ ]:
```

```

[ ]:
```
# Create the environment
env = gym.make("Acrobot-v1")

# Set seed for experiment reproducibility
seed = 42
env.seed(seed)
tf.random.set_seed(seed)
np.random.seed(seed)

# Small epsilon value for stabilizing division operations
eps = np.finfo(np.float32).eps.item()
```

[ ]:
```
class ActorCritic(tf.keras.Model):
  """Combined actor-critic network."""

  def __init__(
      self,
      num_actions: int,
      num_hidden_units: int,
      num_hidden_units1: int,
      num_hidden_units2: int,
      num_hidden_units3: int):
    """Initialize."""
    super().__init__()

    self.common1 = layers.Dense(num_hidden_units, activation="relu")
    self.common2 = layers.Dense(num_hidden_units1, activation="relu")
    #self.common3 = layers.Dense(num_hidden_units2, activation="relu")
    #self.common4 = layers.Dense(num_hidden_units3, activation="relu")
```

```
    self.actor = layers.Dense(num_actions)
    self.critic = layers.Dense(1)

  def call(self, inputs: tf.Tensor) -> Tuple[tf.Tensor, tf.Tensor]:
    x0 = self.common1(inputs)
    x1 = self.common2(x0)
    #x2 = self.common3(inputs)
    #x3 = self.common4(x2)
    return self.actor(x1), self.critic(x1)
```

[ ]:
```
num_actions = env.action_space.n  # 2
num_hidden_units = 512
num_hidden_units1 = 1024
num_hidden_units2 = 512
num_hidden_units3 = 1024

model = ActorCritic(num_actions, num_hidden_units,␣
 ↪num_hidden_units1,num_hidden_units2,num_hidden_units3)
```

[ ]:
```
# Wrap OpenAI Gym's `env.step` call as an operation in a TensorFlow function.
# This would allow it to be included in a callable TensorFlow graph.

def env_step(action: np.ndarray) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
  """Returns state, reward and done flag given an action."""

  state, reward, done, _ = env.step(action)
  return (state.astype(np.float32),
          np.array(reward, np.int32),
          np.array(done, np.int32))


def tf_env_step(action: tf.Tensor) -> List[tf.Tensor]:
  return tf.numpy_function(env_step, [action],
                           [tf.float32, tf.int32, tf.int32])
```

[ ]:
```
def run_episode(
    initial_state: tf.Tensor,
    model: tf.keras.Model,
    max_steps: int) -> Tuple[tf.Tensor, tf.Tensor, tf.Tensor]:
  """Runs a single episode to collect training data."""

  action_probs = tf.TensorArray(dtype=tf.float32, size=0, dynamic_size=True)
  values = tf.TensorArray(dtype=tf.float32, size=0, dynamic_size=True)
  rewards = tf.TensorArray(dtype=tf.int32, size=0, dynamic_size=True)

  initial_state_shape = initial_state.shape
  state = initial_state
```

```
  for t in tf.range(max_steps):
    # Convert state into a batched tensor (batch size = 1)
    state = tf.expand_dims(state, 0)

    # Run the model and to get action probabilities and critic value
    action_logits_t, value = model(state)

    # Sample next action from the action probability distribution
    action = tf.random.categorical(action_logits_t, 1)[0, 0]
    action_probs_t = tf.nn.softmax(action_logits_t)

    # Store critic values
    values = values.write(t, tf.squeeze(value))

    # Store log probability of the action chosen
    action_probs = action_probs.write(t, action_probs_t[0, action])

    # Apply action to the environment to get next state and reward
    state, reward, done = tf_env_step(action)
    state.set_shape(initial_state_shape)

    # Store reward
    rewards = rewards.write(t, reward)

    if tf.cast(done, tf.bool):
      break

  action_probs = action_probs.stack()
  values = values.stack()
  rewards = rewards.stack()

  return action_probs, values, rewards
```

[ ]:
```
def get_expected_return(
    rewards: tf.Tensor,
    values: tf.Tensor,
    step: int,
    gamma: float,
    standardize: bool = True) -> tf.Tensor:
  """Compute expected returns per timestep."""

  n = tf.shape(rewards)[0]
  returns = tf.TensorArray(dtype=tf.float32, size=n)

  # Start from the end of `rewards` and accumulate reward sums
  # into the `returns` array
```

```
  rewards = tf.cast(rewards, dtype=tf.float32)
  discounted_sum = tf.constant(0.0)
  discounted_sum_shape = discounted_sum.shape
  for i in tf.range(n):

    if i+step >= n:
      dsc = tf.constant(0.0)
      dsc_shape = dsc.shape
      for j in tf.range(i,n):
        dsc = tf.math.pow(gamma,float(j-i))*rewards[j] + dsc
        dsc.set_shape(dsc_shape)
      discounted_sum = dsc
    else:
      dsc = tf.constant(0.0)
      dsc_shape = dsc.shape
      for j in tf.range(i,i+step):
        dsc = tf.math.pow(gamma,float(j-i))*rewards[j] + dsc
        dsc.set_shape(dsc_shape)
      discounted_sum = dsc + tf.math.pow(gamma,float(step))*values[i+step]
    #discounted_sum.set_shape(discounted_sum_shape)


    #if i+step < n:
    #  discounted_sum += np.power(gamma,step)*values[i+step]

    #if i+step < n+1:
    #  discounted_sum += np.power(gamma,step-1)*rewards[i+step-1]

    discounted_sum.set_shape(discounted_sum_shape)
    returns = returns.write(i, discounted_sum)
  returns = returns.stack()

  if standardize:
    returns = ((returns - tf.math.reduce_mean(returns)) /
               (tf.math.reduce_std(returns) + eps))

  return returns
```

[ ]:
```
huber_loss = tf.keras.losses.Huber(reduction=tf.keras.losses.Reduction.SUM)

def compute_loss(
    action_probs: tf.Tensor,
    values: tf.Tensor,
    returns: tf.Tensor) -> tf.Tensor:
  """Computes the combined actor-critic loss."""

  advantage = returns - values
```

```
  action_log_probs = tf.math.log(action_probs)
  actor_loss = -tf.math.reduce_sum(action_log_probs * advantage)

  critic_loss = huber_loss(values, returns)

  return actor_loss + critic_loss
```

[ ]:
```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)


@tf.function
def train_step(
    initial_state: tf.Tensor,
    model: tf.keras.Model,
    optimizer: tf.keras.optimizers.Optimizer,
    gamma: float,
    step: int,
    max_steps_per_episode: int) -> tf.Tensor:
  """Runs a model training step."""

  with tf.GradientTape() as tape:

    # Run the model for one episode to collect training data
    action_probs, values, rewards = run_episode(
        initial_state, model, max_steps_per_episode)

    # Calculate expected returns
    returns = get_expected_return(rewards,values,step, gamma)

    # Convert training data to appropriate TF tensor shapes
    action_probs, values, returns = [
        tf.expand_dims(x, 1) for x in [action_probs, values, returns]]

    # Calculating loss values to update our network
    loss = compute_loss(action_probs, values, returns)

  # Compute the gradients from the loss
  grads = tape.gradient(loss, model.trainable_variables)

  # Apply the gradients to the model's parameters
  optimizer.apply_gradients(zip(grads, model.trainable_variables))

  episode_reward = tf.math.reduce_sum(rewards)

  return episode_reward
```

[ ]:
```
import tqdm
```

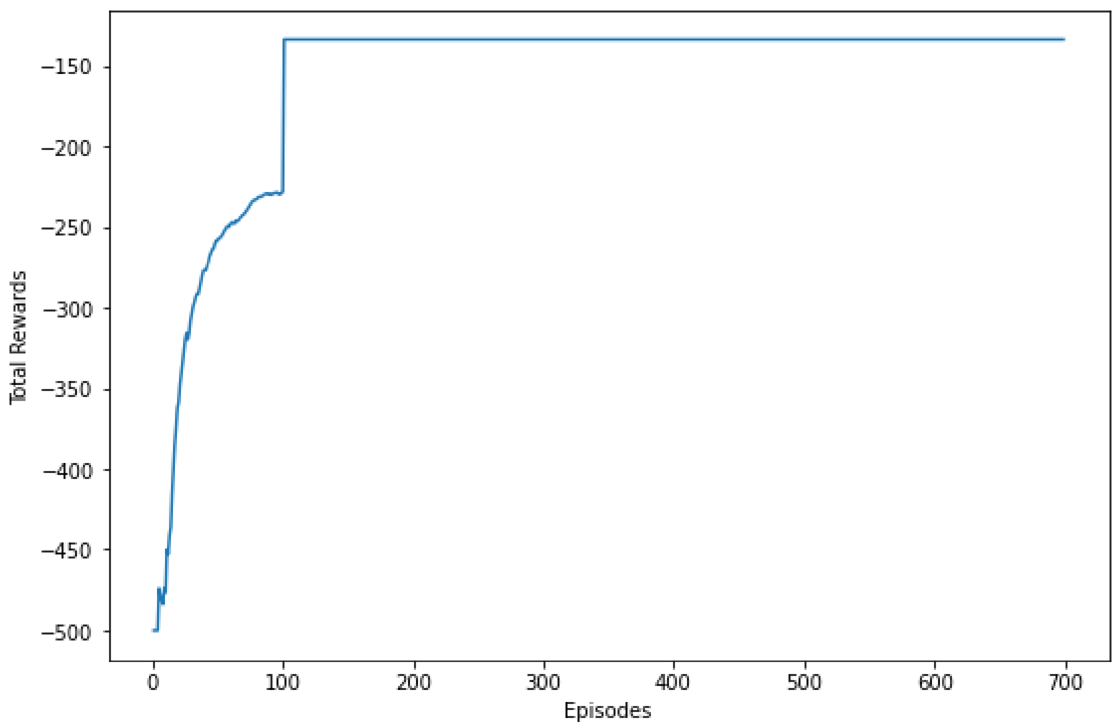
[ ]:
```
%%time

min_episodes_criterion = 100
max_episodes = 700
max_steps_per_episode = 500


reward_threshold = -100
running_reward = 0

step = 10

gamma = 0.99

# Keep last episodes reward
episodes_reward: collections.deque = collections.
 ↪deque(maxlen=min_episodes_criterion)
rewards_list = []
with tqdm.trange(max_episodes) as t:
  for i in t:
    initial_state = tf.constant(env.reset(), dtype=tf.float32)
    episode_reward = int(train_step(
        initial_state, model, optimizer, gamma,step, max_steps_per_episode))

    episodes_reward.append(episode_reward)
    rewards_list.append(episode_reward)
    running_reward = statistics.mean(episodes_reward)

    t.set_description(f'Episode {i}')
    t.set_postfix(
        episode_reward=episode_reward, running_reward=running_reward)

    # Show average episode reward every 10 episodes
    if i % 10 == 0:
      pass # print(f'Episode {i}: average reward: {avg_reward}')

    if running_reward > reward_threshold and i >= min_episodes_criterion:
        break

print(f'\nSolved at episode {i}: average reward: {running_reward:.2f}!')
```

```
Episode 699: 100%|      | 700/700 [04:47<00:00,  2.43it/s,
episode_reward=-179, running_reward=-134]


Solved at episode 699: average reward: -133.73!
```

```
CPU times: user 6min 46s, sys: 24 s, total: 7min 10s
Wall time: 4min 47s
```

[ ]:
```python
mov_avg=[]
for i in range(1,len(rewards_list)):
  if i > 100:
    mov_avg.append(np.mean(rewards_list[-100:]))
  else:
    mov_avg.append(np.mean(rewards_list[:i]))

plt.figure(figsize=(9,6))
plt.xlabel('Episodes')
plt.ylabel('Total Rewards')
plt.plot(np.arange(1,len(mov_avg)+1),mov_avg)
```

[ ]: [<matplotlib.lines.Line2D at 0x7f2292cf9710>]



[ ]:
```python
rewards_ep = []

for run in range(10):

  optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
```

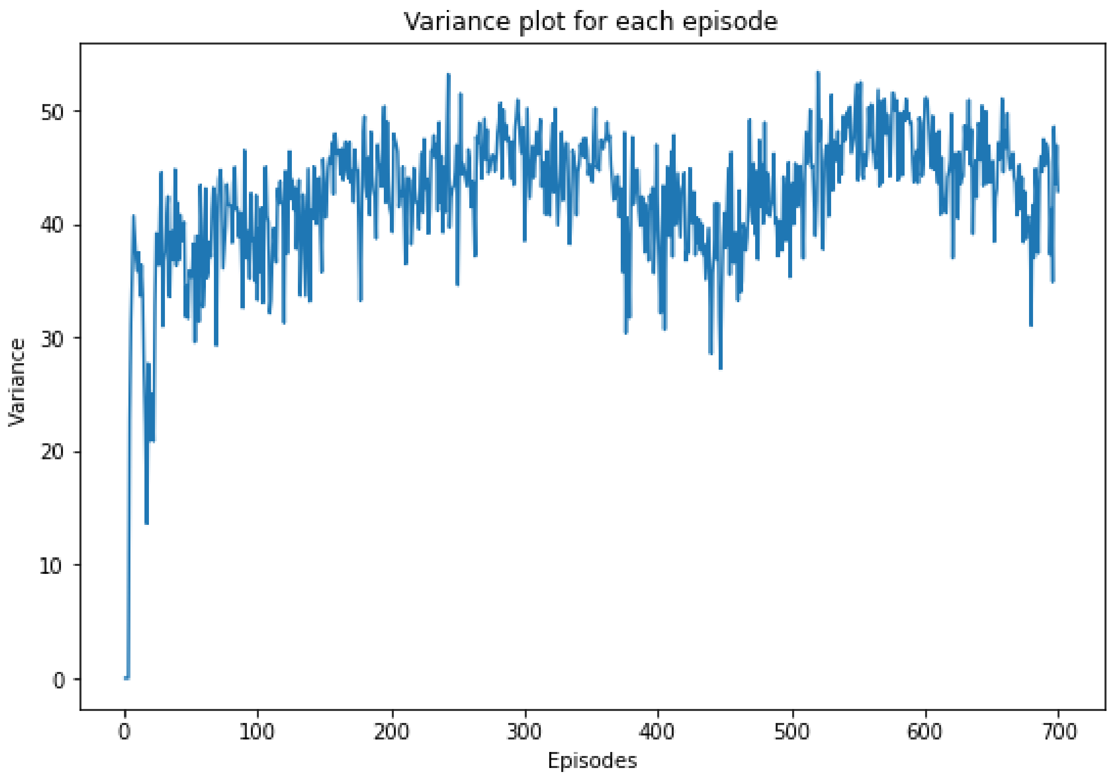
```python
@tf.function
def train_step(
    initial_state: tf.Tensor,
    model: tf.keras.Model,
    optimizer: tf.keras.optimizers.Optimizer,
    gamma: float,
    step: int,
    max_steps_per_episode: int) -> tf.Tensor:
  """Runs a model training step."""

  with tf.GradientTape() as tape:

    # Run the model for one episode to collect training data
    action_probs, values, rewards = run_episode(
        initial_state, model, max_steps_per_episode)

    # Calculate expected returns
    returns = get_expected_return(rewards,values,step, gamma)

    # Convert training data to appropriate TF tensor shapes
    action_probs, values, returns = [
        tf.expand_dims(x, 1) for x in [action_probs, values, returns]]

    # Calculating loss values to update our network
    loss = compute_loss(action_probs, values, returns)

  # Compute the gradients from the loss
  grads = tape.gradient(loss, model.trainable_variables)

  # Apply the gradients to the model's parameters
  optimizer.apply_gradients(zip(grads, model.trainable_variables))

  episode_reward = tf.math.reduce_sum(rewards)

  return episode_reward

env = gym.make("Acrobot-v1")

model = ActorCritic(num_actions, num_hidden_units,␣
↪num_hidden_units1,num_hidden_units2,num_hidden_units3)


print(run)
episodes_reward: collections.deque = collections.
↪deque(maxlen=min_episodes_criterion)
```

```python
  rewards_list = []
  with tqdm.trange(max_episodes) as t:
    for i in t:
      initial_state = tf.constant(env.reset(), dtype=tf.float32)
      episode_reward = int(train_step(
          initial_state, model, optimizer, gamma,step, max_steps_per_episode))

      episodes_reward.append(episode_reward)
      rewards_list.append(episode_reward)
      running_reward = statistics.mean(episodes_reward)

      t.set_description(f'Episode {i}')
      t.set_postfix(
          episode_reward=episode_reward, running_reward=running_reward)

      # Show average episode reward every 10 episodes
      if i % 10 == 0:
        pass # print(f'Episode {i}: average reward: {avg_reward}')

  rewards_ep.append(rewards_list)
```

[ ]:
```python
rewards_ep = np.array(rewards_ep)
mean=np.array([np.mean(rewards_ep, axis=0) for i in range(10)])
var = np.sqrt(np.sum((rewards_ep-mean)**2,axis=0))/10
```

[ ]:
```python
plt.figure(figsize=(9,6))
plt.xlabel('Episodes')
plt.ylabel('Variance ')
plt.title('Variance plot for each episode')
plt.plot(np.arange(1,len(var)+1),var)
```

[ ]:
```
[<matplotlib.lines.Line2D at 0x7f228f099e90>]
```


Variance plot for each episode
Variance
Episodes
0
10
20
30
40
50
0
100
200
300
400
500
600
700

[ ]:

RLPA2_actor_critic_mountaincar

March 31, 2022

[ ]:
```
'''
Installing packages for rendering the game on Colab
'''

!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
```

```
Requirement already satisfied: setuptools in /usr/local/lib/python3.7/dist-
packages (61.2.0)
```

[ ]:
```
!pip install tensorflow-gpu
```

```
Collecting tensorflow-gpu
  Downloading tensorflow_gpu-2.8.0-cp37-cp37m-manylinux2010_x86_64.whl (497.5
MB)
     |                 | 497.5 MB 21 kB/s
Requirement already satisfied: absl-py>=0.4.0 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (1.0.0)
Requirement already satisfied: keras<2.9,>=2.8.0rc0 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (2.8.0)
Requirement already satisfied: h5py>=2.9.0 in /usr/local/lib/python3.7/dist-
packages (from tensorflow-gpu) (3.1.0)
Requirement already satisfied: astunparse>=1.6.0 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (1.6.3)
Requirement already satisfied: tensorboard<2.9,>=2.8 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (2.8.0)
Requirement already satisfied: typing-extensions>=3.6.6 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (3.10.0.2)
Requirement already satisfied: libclang>=9.0.1 in /usr/local/lib/python3.7/dist-
packages (from tensorflow-gpu) (13.0.0)
Requirement already satisfied: wrapt>=1.11.0 in /usr/local/lib/python3.7/dist-
packages (from tensorflow-gpu) (1.14.0)
```

```
Requirement already satisfied: numpy>=1.20 in /usr/local/lib/python3.7/dist-
packages (from tensorflow-gpu) (1.21.5)
Requirement already satisfied: flatbuffers>=1.12 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (2.0)
Requirement already satisfied: protobuf>=3.9.2 in /usr/local/lib/python3.7/dist-
packages (from tensorflow-gpu) (3.17.3)
Requirement already satisfied: gast>=0.2.1 in /usr/local/lib/python3.7/dist-
packages (from tensorflow-gpu) (0.5.3)
Requirement already satisfied: opt-einsum>=2.3.2 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (3.3.0)
Collecting tf-estimator-nightly==2.8.0.dev2021122109
  Downloading tf_estimator_nightly-2.8.0.dev2021122109-py2.py3-none-any.whl (462
kB)
     |                 | 462 kB 47.9 MB/s
Requirement already satisfied: six>=1.12.0 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (1.15.0)
Requirement already satisfied: grpcio<2.0,>=1.24.3 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (1.44.0)
Requirement already satisfied: keras-preprocessing>=1.1.1 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (1.1.2)
Requirement already satisfied: tensorflow-io-gcs-filesystem>=0.23.1 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (0.24.0)
Requirement already satisfied: setuptools in /usr/local/lib/python3.7/dist-
packages (from tensorflow-gpu) (61.2.0)
Requirement already satisfied: google-pasta>=0.1.1 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (0.2.0)
Requirement already satisfied: termcolor>=1.1.0 in
/usr/local/lib/python3.7/dist-packages (from tensorflow-gpu) (1.1.0)
Requirement already satisfied: wheel<1.0,>=0.23.0 in
/usr/local/lib/python3.7/dist-packages (from astunparse>=1.6.0->tensorflow-gpu)
(0.37.1)
Requirement already satisfied: cached-property in /usr/local/lib/python3.7/dist-
packages (from h5py>=2.9.0->tensorflow-gpu) (1.5.2)
Requirement already satisfied: werkzeug>=0.11.15 in
/usr/local/lib/python3.7/dist-packages (from tensorboard<2.9,>=2.8->tensorflow-
gpu) (1.0.1)
Requirement already satisfied: google-auth<3,>=1.6.3 in
/usr/local/lib/python3.7/dist-packages (from tensorboard<2.9,>=2.8->tensorflow-
gpu) (1.35.0)
Requirement already satisfied: requests<3,>=2.21.0 in
/usr/local/lib/python3.7/dist-packages (from tensorboard<2.9,>=2.8->tensorflow-
gpu) (2.23.0)
Requirement already satisfied: tensorboard-data-server<0.7.0,>=0.6.0 in
/usr/local/lib/python3.7/dist-packages (from tensorboard<2.9,>=2.8->tensorflow-
gpu) (0.6.1)
Requirement already satisfied: tensorboard-plugin-wit>=1.6.0 in
/usr/local/lib/python3.7/dist-packages (from tensorboard<2.9,>=2.8->tensorflow-
gpu) (1.8.1)
```

```
Requirement already satisfied: google-auth-oauthlib<0.5,>=0.4.1 in
/usr/local/lib/python3.7/dist-packages (from tensorboard<2.9,>=2.8->tensorflow-
gpu) (0.4.6)
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.7/dist-
packages (from tensorboard<2.9,>=2.8->tensorflow-gpu) (3.3.6)
Requirement already satisfied: rsa<5,>=3.1.4 in /usr/local/lib/python3.7/dist-
packages (from google-auth<3,>=1.6.3->tensorboard<2.9,>=2.8->tensorflow-gpu)
(4.8)
Requirement already satisfied: cachetools<5.0,>=2.0.0 in
/usr/local/lib/python3.7/dist-packages (from google-
auth<3,>=1.6.3->tensorboard<2.9,>=2.8->tensorflow-gpu) (4.2.4)
Requirement already satisfied: pyasn1-modules>=0.2.1 in
/usr/local/lib/python3.7/dist-packages (from google-
auth<3,>=1.6.3->tensorboard<2.9,>=2.8->tensorflow-gpu) (0.2.8)
Requirement already satisfied: requests-oauthlib>=0.7.0 in
/usr/local/lib/python3.7/dist-packages (from google-auth-
oauthlib<0.5,>=0.4.1->tensorboard<2.9,>=2.8->tensorflow-gpu) (1.3.1)
Requirement already satisfied: importlib-metadata>=4.4 in
/usr/local/lib/python3.7/dist-packages (from
markdown>=2.6.8->tensorboard<2.9,>=2.8->tensorflow-gpu) (4.11.3)
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.7/dist-
packages (from importlib-
metadata>=4.4->markdown>=2.6.8->tensorboard<2.9,>=2.8->tensorflow-gpu) (3.7.0)
Requirement already satisfied: pyasn1<0.5.0,>=0.4.6 in
/usr/local/lib/python3.7/dist-packages (from pyasn1-modules>=0.2.1->google-
auth<3,>=1.6.3->tensorboard<2.9,>=2.8->tensorflow-gpu) (0.4.8)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in
/usr/local/lib/python3.7/dist-packages (from
requests<3,>=2.21.0->tensorboard<2.9,>=2.8->tensorflow-gpu) (1.24.3)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.7/dist-packages (from
requests<3,>=2.21.0->tensorboard<2.9,>=2.8->tensorflow-gpu) (2021.10.8)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-
packages (from requests<3,>=2.21.0->tensorboard<2.9,>=2.8->tensorflow-gpu)
(2.10)
Requirement already satisfied: chardet<4,>=3.0.2 in
/usr/local/lib/python3.7/dist-packages (from
requests<3,>=2.21.0->tensorboard<2.9,>=2.8->tensorflow-gpu) (3.0.4)
Requirement already satisfied: oauthlib>=3.0.0 in /usr/local/lib/python3.7/dist-
packages (from requests-oauthlib>=0.7.0->google-auth-
oauthlib<0.5,>=0.4.1->tensorboard<2.9,>=2.8->tensorflow-gpu) (3.2.0)
Installing collected packages: tf-estimator-nightly, tensorflow-gpu
Successfully installed tensorflow-gpu-2.8.0 tf-estimator-
nightly-2.8.0.dev2021122109
```

[ ]:
```
'''
A bunch of imports, you don't have to worry about these
```

```
'''

import collections
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 import Monitor
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
from typing import Any, List, Sequence, Tuple
from tensorflow.keras import layers
import statistics
import tqdm
```

[ ]:
```
'''
Please refer to the first tutorial for more details on the specifics of␣
↪environments
We've only added important commands you might find useful for experiments.
'''

'''
List of example environments
(Source - https://gym.openai.com/envs/#classic_control)

'Acrobot-v1'
'CartPole-v0'
'MountainCar-v0'
'''

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

state_shape = env.observation_space.shape[0]
```

```python
no_of_actions = env.action_space.n

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

'''
# Understanding State, Action, Reward Dynamics

The agent decides an action to take depending on the state.

The Environment keeps a variable specifically for the current state.
- Everytime an action is passed to the environment, it calculates the new state␣
↪and updates the current state variable.
- It returns the new current state and reward for the agent to take the next␣
↪action

'''

state = env.reset()
''' This returns the initial state (when environment is reset) '''

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

action = env.action_space.sample()
''' We take a random action now '''

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

next_state, reward, done, info = env.step(action)
''' env.step is used to calculate new state and obtain reward based on old␣
↪state and action taken  '''

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

```
2
3
2
----
[-0.58912799  0.        ]
```

```
----
1
----
[-5.88639679e-01  4.88309600e-04]
-1.0
False
{}
----
```

[ ]:
```
class ActorCriticModel(tf.keras.Model):
    """
    Defining policy and value networkss
    """
    def __init__(self, action_size, n_hidden1=1024, n_hidden2=512):
        super(ActorCriticModel, self).__init__()

        #Hidden Layer 1
        self.fc1 = tf.keras.layers.Dense(n_hidden1, activation='relu')
        #Hidden Layer 2
        self.fc2 = tf.keras.layers.Dense(n_hidden1, activation='relu')

        self.fc3 = tf.keras.layers.Dense(n_hidden1, activation='relu')
        self.fc4 = tf.keras.layers.Dense(n_hidden1, activation='relu')

        self.fc5 = tf.keras.layers.Dense(n_hidden2, activation='relu')

        #Output Layer for policy
        self.pi_out = tf.keras.layers.Dense(action_size, activation='softmax')
        #Output Layer for state-value
        self.v_out = tf.keras.layers.Dense(1)

    def call(self, state):
        """
        Computes policy distribution and state-value for a given state
        """
        layer1 = self.fc1(state)
        layer2 = self.fc2(layer1)
        layer3 = self.fc3(layer2)
        layer4 = self.fc4(layer3)
        layer5 = self.fc5(layer4)

        pi = self.pi_out(layer5)
        v = self.v_out(layer5)

        return pi, v
```

[ ]:

```python
class Agent:
    """
    Agent class
    """
    def __init__(self, action_size, lr=0.01, gamma=0.99, seed = 85):
        self.gamma = gamma
        self.ac_model = ActorCriticModel(action_size=action_size)
        self.ac_model.compile(tf.keras.optimizers.Adam(learning_rate=lr))
        np.random.seed(seed)

    def sample_action(self, state):
        """
        Given a state, compute the policy distribution over all actions and sample one action
        """
        pi,_ = self.ac_model(state)

        action_probabilities = tfp.distributions.Categorical(probs=pi)
        sample = action_probabilities.sample()

        return int(sample.numpy()[0])

    def actor_loss(self, action, pi, delta):
        """
        Compute Actor Loss
        """
        return -tf.math.log(pi[0,action]) * delta

    def critic_loss(self,delta):
        """
        Critic loss aims to minimize TD error
        """
        return delta**2

    @tf.function
    def learn(self, state, action, reward, next_state, done):
        """
        For a given transition (s,a,s',r) update the paramters by computing the
        gradient of the total loss
        """
        with tf.GradientTape(persistent=True) as tape:
            pi, V_s = self.ac_model(state)
            _, V_s_next = self.ac_model(next_state)

            V_s = tf.squeeze(V_s)
            V_s_next = tf.squeeze(V_s_next)
```

```python
            #### TO DO: Write the equation for delta (TD error)
            ## Write code below
            delta = reward + self.gamma*V_s_next-V_s

            loss_a = self.actor_loss(action, pi, delta)
            loss_c =self.critic_loss(delta)
            loss_total = loss_a + loss_c

        gradient = tape.gradient(loss_total, self.ac_model.trainable_variables)
        self.ac_model.optimizer.apply_gradients(zip(gradient, self.ac_model.
 ↪trainable_variables))
```

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

#Initializing Agent
agent = Agent(lr=1e-3, action_size=env.action_space.n)
#Number of episodes
episodes = 1000
tf.compat.v1.reset_default_graph()

reward_list = []
average_reward_list = []
begin_time = datetime.datetime.now()

for ep in range(1, episodes + 1):
    state = env.reset().reshape(1,-1)
    done = False
    ep_rew = 0
    while not done:
        action = agent.sample_action(state) ##Sample Action
        next_state, reward, done, info = env.step(action) ##Take action
        next_state = next_state.reshape(1,-1)
        ep_rew += reward  ##Updating episode reward
        agent.learn(state, action, reward, next_state, done) ##Update Parameters
        state = next_state ##Updating State
    reward_list.append(ep_rew)

    if ep % 10 == 0:
        avg_rew = np.mean(reward_list[-10:])
        print('Episode ', ep, 'Reward %f' % ep_rew, 'Average Reward %f' %␣
 ↪avg_rew)

    if ep % 100:
        avg_100 =  np.mean(reward_list[-100:])
        if avg_100 > -110.0:
            print('Stopped at Episode ',ep)
```

```
            break

time_taken = datetime.datetime.now() - begin_time
print(time_taken)
```

```
Episode  10 Reward -200.000000 Average Reward -200.000000
Episode  20 Reward -200.000000 Average Reward -200.000000
Episode  30 Reward -200.000000 Average Reward -200.000000
Episode  40 Reward -200.000000 Average Reward -200.000000
Episode  50 Reward -200.000000 Average Reward -200.000000
Episode  60 Reward -200.000000 Average Reward -200.000000
Episode  70 Reward -200.000000 Average Reward -200.000000
Episode  80 Reward -200.000000 Average Reward -200.000000
Episode  90 Reward -200.000000 Average Reward -200.000000
Episode  100 Reward -200.000000 Average Reward -200.000000
Episode  110 Reward -200.000000 Average Reward -200.000000
Episode  120 Reward -200.000000 Average Reward -200.000000
Episode  130 Reward -200.000000 Average Reward -200.000000
Episode  140 Reward -200.000000 Average Reward -200.000000
Episode  150 Reward -200.000000 Average Reward -200.000000
Episode  160 Reward -200.000000 Average Reward -200.000000
Episode  170 Reward -200.000000 Average Reward -200.000000
Episode  180 Reward -200.000000 Average Reward -200.000000
Episode  190 Reward -200.000000 Average Reward -200.000000
Episode  200 Reward -200.000000 Average Reward -200.000000
Episode  210 Reward -200.000000 Average Reward -200.000000
Episode  220 Reward -200.000000 Average Reward -200.000000
Episode  230 Reward -200.000000 Average Reward -200.000000
Episode  240 Reward -200.000000 Average Reward -200.000000
Episode  250 Reward -200.000000 Average Reward -200.000000
Episode  260 Reward -200.000000 Average Reward -200.000000
Episode  270 Reward -200.000000 Average Reward -200.000000
Episode  280 Reward -200.000000 Average Reward -200.000000
Episode  290 Reward -200.000000 Average Reward -200.000000
Episode  300 Reward -200.000000 Average Reward -200.000000
Episode  310 Reward -200.000000 Average Reward -200.000000
Episode  320 Reward -200.000000 Average Reward -200.000000
Episode  330 Reward -200.000000 Average Reward -200.000000
Episode  340 Reward -200.000000 Average Reward -200.000000
Episode  350 Reward -200.000000 Average Reward -200.000000
Episode  360 Reward -200.000000 Average Reward -200.000000
Episode  370 Reward -200.000000 Average Reward -200.000000
Episode  380 Reward -200.000000 Average Reward -200.000000
Episode  390 Reward -200.000000 Average Reward -200.000000
Episode  400 Reward -200.000000 Average Reward -200.000000
Episode  410 Reward -200.000000 Average Reward -200.000000
Episode  420 Reward -200.000000 Average Reward -200.000000
```

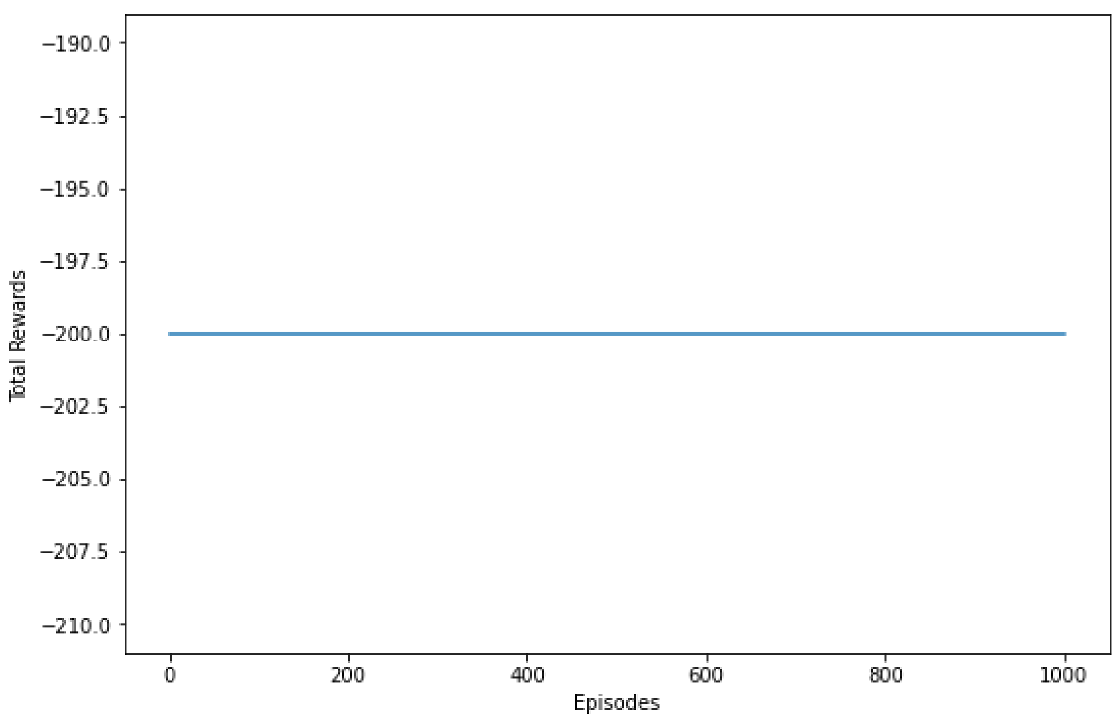
```
Episode  430 Reward -200.000000 Average Reward -200.000000
Episode  440 Reward -200.000000 Average Reward -200.000000
Episode  450 Reward -200.000000 Average Reward -200.000000
Episode  460 Reward -200.000000 Average Reward -200.000000
Episode  470 Reward -200.000000 Average Reward -200.000000
Episode  480 Reward -200.000000 Average Reward -200.000000
Episode  490 Reward -200.000000 Average Reward -200.000000
Episode  500 Reward -200.000000 Average Reward -200.000000
Episode  510 Reward -200.000000 Average Reward -200.000000
Episode  520 Reward -200.000000 Average Reward -200.000000
Episode  530 Reward -200.000000 Average Reward -200.000000
Episode  540 Reward -200.000000 Average Reward -200.000000
Episode  550 Reward -200.000000 Average Reward -200.000000
Episode  560 Reward -200.000000 Average Reward -200.000000
Episode  570 Reward -200.000000 Average Reward -200.000000
Episode  580 Reward -200.000000 Average Reward -200.000000
Episode  590 Reward -200.000000 Average Reward -200.000000
Episode  600 Reward -200.000000 Average Reward -200.000000
Episode  610 Reward -200.000000 Average Reward -200.000000
Episode  620 Reward -200.000000 Average Reward -200.000000
Episode  630 Reward -200.000000 Average Reward -200.000000
Episode  640 Reward -200.000000 Average Reward -200.000000
Episode  650 Reward -200.000000 Average Reward -200.000000
Episode  660 Reward -200.000000 Average Reward -200.000000
Episode  670 Reward -200.000000 Average Reward -200.000000
Episode  680 Reward -200.000000 Average Reward -200.000000
Episode  690 Reward -200.000000 Average Reward -200.000000
Episode  700 Reward -200.000000 Average Reward -200.000000
Episode  710 Reward -200.000000 Average Reward -200.000000
Episode  720 Reward -200.000000 Average Reward -200.000000
Episode  730 Reward -200.000000 Average Reward -200.000000
Episode  740 Reward -200.000000 Average Reward -200.000000
Episode  750 Reward -200.000000 Average Reward -200.000000
Episode  760 Reward -200.000000 Average Reward -200.000000
Episode  770 Reward -200.000000 Average Reward -200.000000
Episode  780 Reward -200.000000 Average Reward -200.000000
Episode  790 Reward -200.000000 Average Reward -200.000000
Episode  800 Reward -200.000000 Average Reward -200.000000
Episode  810 Reward -200.000000 Average Reward -200.000000
Episode  820 Reward -200.000000 Average Reward -200.000000
Episode  830 Reward -200.000000 Average Reward -200.000000
Episode  840 Reward -200.000000 Average Reward -200.000000
Episode  850 Reward -200.000000 Average Reward -200.000000
Episode  860 Reward -200.000000 Average Reward -200.000000
Episode  870 Reward -200.000000 Average Reward -200.000000
Episode  880 Reward -200.000000 Average Reward -200.000000
Episode  890 Reward -200.000000 Average Reward -200.000000
Episode  900 Reward -200.000000 Average Reward -200.000000
```

```
Episode  910 Reward -200.000000 Average Reward -200.000000
Episode  920 Reward -200.000000 Average Reward -200.000000
Episode  930 Reward -200.000000 Average Reward -200.000000
Episode  940 Reward -200.000000 Average Reward -200.000000
Episode  950 Reward -200.000000 Average Reward -200.000000
Episode  960 Reward -200.000000 Average Reward -200.000000
Episode  970 Reward -200.000000 Average Reward -200.000000
Episode  980 Reward -200.000000 Average Reward -200.000000
Episode  990 Reward -200.000000 Average Reward -200.000000
Episode  1000 Reward -200.000000 Average Reward -200.000000
1:41:35.534222
```

Tried for several hyperparameters but still doesn't converge

[ ]:
```
mov_avg=[]
for i in range(1,len(reward_list)):
  if i > 100:
    mov_avg.append(np.mean(reward_list[-100:]))
  else:
    mov_avg.append(np.mean(reward_list[:i]))

plt.figure(figsize=(9,6))
plt.xlabel('Episodes')
plt.ylabel('Total Rewards')
plt.plot(np.arange(1,len(mov_avg)+1),mov_avg)
```

[ ]:
```
[<matplotlib.lines.Line2D at 0x7f7ef7a89450>]
```



[ ]:

```

```

Full return

[ ]:

```
# Create the environment
env = gym.make('MountainCar-v0')

# Set seed for experiment reproducibility
seed = 42
env.seed(seed)
tf.random.set_seed(seed)
np.random.seed(seed)

# Small epsilon value for stabilizing division operations
eps = np.finfo(np.float32).eps.item()
```

[ ]:

```
class ActorCritic(tf.keras.Model):
  """Combined actor-critic network."""

  def __init__(
      self,
      num_actions: int,
      num_hidden_units: int,
      num_hidden_units1: int,
      num_hidden_units2: int,
      num_hidden_units3: int):
    """Initialize."""
    super().__init__()

    self.common1 = layers.Dense(num_hidden_units, activation="relu")
    self.common2 = layers.Dense(num_hidden_units1, activation="relu")
    self.common3 = layers.Dense(num_hidden_units2, activation="relu")
    self.common4 = layers.Dense(num_hidden_units3, activation="relu")
    self.actor = layers.Dense(num_actions)
    self.critic = layers.Dense(1)

  def call(self, inputs: tf.Tensor) -> Tuple[tf.Tensor, tf.Tensor]:
    x0 = self.common1(inputs)
    x1 = self.common2(x0)
    x2 = self.common3(x1)
    x3 = self.common4(x2)
    return self.actor(x1), self.critic(x1)
```

[ ]:

```
num_actions = env.action_space.n  # 2
num_hidden_units = 512
```

```
num_hidden_units1 = 1024
num_hidden_units2 = 1024
num_hidden_units3 = 1024

model = ActorCritic(num_actions, num_hidden_units,␣
 ↪num_hidden_units1,num_hidden_units2,num_hidden_units3)
```

[ ]:
```
# Wrap OpenAI Gym's `env.step` call as an operation in a TensorFlow function.
# This would allow it to be included in a callable TensorFlow graph.

def env_step(action: np.ndarray) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
  """Returns state, reward and done flag given an action."""

  state, reward, done, _ = env.step(action)
  return (state.astype(np.float32),
          np.array(reward, np.int32),
          np.array(done, np.int32))


def tf_env_step(action: tf.Tensor) -> List[tf.Tensor]:
  return tf.numpy_function(env_step, [action],
                           [tf.float32, tf.int32, tf.int32])
```

[ ]:
```
def run_episode(
    initial_state: tf.Tensor,
    model: tf.keras.Model,
    max_steps: int) -> Tuple[tf.Tensor, tf.Tensor, tf.Tensor]:
  """Runs a single episode to collect training data."""

  action_probs = tf.TensorArray(dtype=tf.float32, size=0, dynamic_size=True)
  values = tf.TensorArray(dtype=tf.float32, size=0, dynamic_size=True)
  rewards = tf.TensorArray(dtype=tf.int32, size=0, dynamic_size=True)

  initial_state_shape = initial_state.shape
  state = initial_state

  for t in tf.range(max_steps):
    # Convert state into a batched tensor (batch size = 1)
    state = tf.expand_dims(state, 0)

    # Run the model and to get action probabilities and critic value
    action_logits_t, value = model(state)

    # Sample next action from the action probability distribution
    action = tf.random.categorical(action_logits_t, 1)[0, 0]
    action_probs_t = tf.nn.softmax(action_logits_t)
```

```
    # Store critic values
    values = values.write(t, tf.squeeze(value))

    # Store log probability of the action chosen
    action_probs = action_probs.write(t, action_probs_t[0, action])

    # Apply action to the environment to get next state and reward
    state, reward, done = tf_env_step(action)
    state.set_shape(initial_state_shape)

    # Store reward
    rewards = rewards.write(t, reward)

    if tf.cast(done, tf.bool):
      break

  action_probs = action_probs.stack()
  values = values.stack()
  rewards = rewards.stack()

  return action_probs, values, rewards
```

[ ]:
```
def get_expected_return(
    rewards: tf.Tensor,
    gamma: float,
    standardize: bool = True) -> tf.Tensor:
  """Compute expected returns per timestep."""

  n = tf.shape(rewards)[0]
  returns = tf.TensorArray(dtype=tf.float32, size=n)

  # Start from the end of `rewards` and accumulate reward sums
  # into the `returns` array
  rewards = tf.cast(rewards[::-1], dtype=tf.float32)
  discounted_sum = tf.constant(0.0)
  discounted_sum_shape = discounted_sum.shape
  for i in tf.range(n):
    reward = rewards[i]
    discounted_sum = reward + gamma * discounted_sum
    discounted_sum.set_shape(discounted_sum_shape)
    returns = returns.write(i, discounted_sum)
  returns = returns.stack()[::-1]

  if standardize:
    returns = ((returns - tf.math.reduce_mean(returns)) /
               (tf.math.reduce_std(returns) + eps))
```

```
  return returns
```

[ ]:
```
huber_loss = tf.keras.losses.Huber(reduction=tf.keras.losses.Reduction.SUM)

def compute_loss(
    action_probs: tf.Tensor,
    values: tf.Tensor,
    returns: tf.Tensor) -> tf.Tensor:
  """Computes the combined actor-critic loss."""

  advantage = returns - values

  action_log_probs = tf.math.log(action_probs)
  actor_loss = -tf.math.reduce_sum(action_log_probs * advantage)

  critic_loss = huber_loss(values, returns)

  return actor_loss + critic_loss
```

[ ]:
```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.01)


@tf.function
def train_step(
    initial_state: tf.Tensor,
    model: tf.keras.Model,
    optimizer: tf.keras.optimizers.Optimizer,
    gamma: float,
    max_steps_per_episode: int) -> tf.Tensor:
  """Runs a model training step."""

  with tf.GradientTape() as tape:

    # Run the model for one episode to collect training data
    action_probs, values, rewards = run_episode(
        initial_state, model, max_steps_per_episode)

    # Calculate expected returns
    returns = get_expected_return(rewards, gamma)

    # Convert training data to appropriate TF tensor shapes
    action_probs, values, returns = [
        tf.expand_dims(x, 1) for x in [action_probs, values, returns]]

    # Calculating loss values to update our network
    loss = compute_loss(action_probs, values, returns)

```

```python
  # Compute the gradients from the loss
  grads = tape.gradient(loss, model.trainable_variables)

  # Apply the gradients to the model's parameters
  optimizer.apply_gradients(zip(grads, model.trainable_variables))

  episode_reward = tf.math.reduce_sum(rewards)

  return episode_reward
```

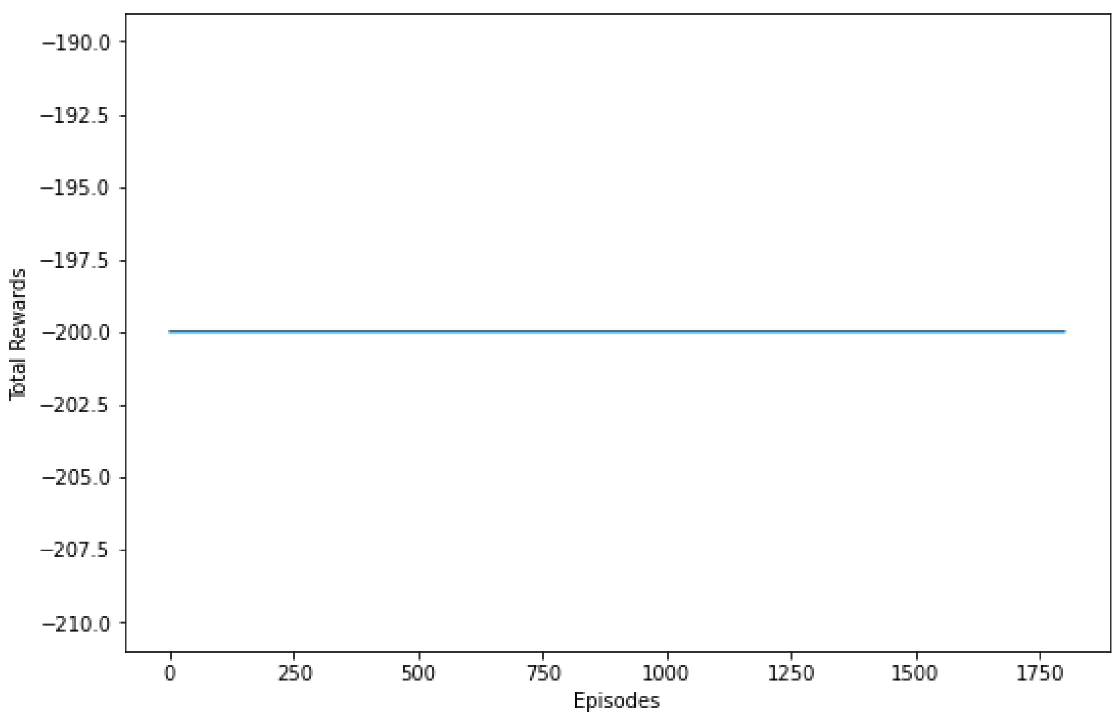
[ ]:
```python
%%time

min_episodes_criterion = 100
max_episodes = 1800
max_steps_per_episode = 500


reward_threshold = 475
running_reward = 0


gamma = 0.99

# Keep last episodes reward
episodes_reward: collections.deque = collections.
 ↪deque(maxlen=min_episodes_criterion)
rewards_list = []
with tqdm.trange(max_episodes) as t:
  for i in t:
    initial_state = tf.constant(env.reset(), dtype=tf.float32)
    episode_reward = int(train_step(
        initial_state, model, optimizer, gamma, max_steps_per_episode))

    episodes_reward.append(episode_reward)
    rewards_list.append(episode_reward)
    running_reward = statistics.mean(episodes_reward)

    t.set_description(f'Episode {i}')
    t.set_postfix(
        episode_reward=episode_reward, running_reward=running_reward)

    # Show average episode reward every 10 episodes
    if i % 10 == 0:
      pass # print(f'Episode {i}: average reward: {avg_reward}')

    if running_reward > reward_threshold and i >= min_episodes_criterion:
        break
```

```
print(f'\nSolved at episode {i}: average reward: {running_reward:.2f}!')
```

```
Episode 1799: 100%|      | 1800/1800 [09:48<00:00,  3.06it/s,
episode_reward=-200, running_reward=-200]


Solved at episode 1799: average reward: -200.00!
CPU times: user 14min 39s, sys: 48.3 s, total: 15min 27s
Wall time: 9min 48s
```

[ ]:
```
mov_avg=[]
for i in range(1,len(rewards_list)):
  if i > 100:
    mov_avg.append(np.mean(rewards_list[-100:]))
  else:
    mov_avg.append(np.mean(rewards_list[:i]))

plt.figure(figsize=(9,6))
plt.xlabel('Episodes')
plt.ylabel('Total Rewards')
plt.plot(np.arange(1,len(mov_avg)+1),mov_avg)
```

[ ]:
```
[<matplotlib.lines.Line2D at 0x7f7ef211dfd0>]
```



[ ]:

```

```

[ ]:

```

```

n-step return mountain-car

[ ]:

```
# Create the environment
env = gym.make("MountainCar-v0")

# Set seed for experiment reproducibility
seed = 42
env.seed(seed)
tf.random.set_seed(seed)
np.random.seed(seed)

# Small epsilon value for stabilizing division operations
eps = np.finfo(np.float32).eps.item()
```

[ ]:

```
class ActorCritic(tf.keras.Model):
  """Combined actor-critic network."""

  def __init__(
      self,
      num_actions: int,
      num_hidden_units: int,
      num_hidden_units1: int):
    """Initialize."""
    super().__init__()

    self.a1 = layers.Dense(num_hidden_units, activation="relu")
    self.a2 = layers.Dense(num_hidden_units1, activation="relu")
    self.c1 = layers.Dense(num_hidden_units, activation="relu")
    self.c2 = layers.Dense(num_hidden_units1, activation="relu")
    self.actor = layers.Dense(num_actions)
    self.critic = layers.Dense(1)

  def call(self, inputs: tf.Tensor) -> Tuple[tf.Tensor, tf.Tensor]:
    x = self.a1(inputs)
    y = self.c1(inputs)
    return self.actor(x), self.critic(y)
```

[ ]:

```
num_actions = env.action_space.n  # 2
num_hidden_units = 512
num_hidden_units1 = 128
model = ActorCritic(num_actions, num_hidden_units,num_hidden_units1)
```

[ ]:
```python
# Wrap OpenAI Gym's `env.step` call as an operation in a TensorFlow function.
# This would allow it to be included in a callable TensorFlow graph.

def env_step(action: np.ndarray) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
  """Returns state, reward and done flag given an action."""

  state, reward, done, _ = env.step(action)
  return (state.astype(np.float32),
          np.array(reward, np.int32),
          np.array(done, np.int32))


def tf_env_step(action: tf.Tensor) -> List[tf.Tensor]:
  return tf.numpy_function(env_step, [action],
                           [tf.float32, tf.int32, tf.int32])
```

[ ]:
```python
def run_episode(
    initial_state: tf.Tensor,
    model: tf.keras.Model,
    max_steps: int) -> Tuple[tf.Tensor, tf.Tensor, tf.Tensor]:
  """Runs a single episode to collect training data."""

  action_probs = tf.TensorArray(dtype=tf.float32, size=0, dynamic_size=True)
  values = tf.TensorArray(dtype=tf.float32, size=0, dynamic_size=True)
  rewards = tf.TensorArray(dtype=tf.int32, size=0, dynamic_size=True)

  initial_state_shape = initial_state.shape
  state = initial_state

  for t in tf.range(max_steps):
    # Convert state into a batched tensor (batch size = 1)
    state = tf.expand_dims(state, 0)

    # Run the model and to get action probabilities and critic value
    action_logits_t, value = model(state)

    # Sample next action from the action probability distribution
    action = tf.random.categorical(action_logits_t, 1)[0, 0]
    action_probs_t = tf.nn.softmax(action_logits_t)

    # Store critic values
    values = values.write(t, tf.squeeze(value))

    # Store log probability of the action chosen
    action_probs = action_probs.write(t, action_probs_t[0, action])

    # Apply action to the environment to get next state and reward
```

```
    state, reward, done = tf_env_step(action)
    state.set_shape(initial_state_shape)

    # Store reward
    rewards = rewards.write(t, reward)

    if tf.cast(done, tf.bool):
      break

  action_probs = action_probs.stack()
  values = values.stack()
  rewards = rewards.stack()

  return action_probs, values, rewards
```

[ ]:
```
def get_expected_return(
    rewards: tf.Tensor,
    values: tf.Tensor,
    step: int,
    gamma: float,
    standardize: bool = True) -> tf.Tensor:
  """Compute expected returns per timestep."""

  n = tf.shape(rewards)[0]
  returns = tf.TensorArray(dtype=tf.float32, size=n)

  # Start from the end of `rewards` and accumulate reward sums
  # into the `returns` array
  rewards = tf.cast(rewards[::-1], dtype=tf.float32)
  discounted_sum = tf.constant(0.0)
  discounted_sum_shape = discounted_sum.shape
  for i in tf.range(n-1,-1,-1):
    reward = rewards[i]
    if i+step >= n:
      discounted_sum = reward + gamma * discounted_sum
    else:
      discounted_sum = reward + gamma * discounted_sum + np.
↪power(gamma,step)*values[i+step]
    discounted_sum.set_shape(discounted_sum_shape)
    returns = returns.write(i, discounted_sum)

    if i+step < n:
      discounted_sum -= np.power(gamma,step)*values[i+step]

    if i+step < n+1:
      discounted_sum -= np.power(gamma,step-1)*rewards[i+step-1]
```

```
    discounted_sum.set_shape(discounted_sum_shape)

  returns = returns.stack()[::-1]

  if standardize:
    returns = ((returns - tf.math.reduce_mean(returns)) /
               (tf.math.reduce_std(returns) + eps))

  return returns
```

[ ]:
```
huber_loss = tf.keras.losses.Huber(reduction=tf.keras.losses.Reduction.SUM)

def compute_loss(
    action_probs: tf.Tensor,
    values: tf.Tensor,
    returns: tf.Tensor) -> tf.Tensor:
  """Computes the combined actor-critic loss."""

  advantage = returns - values

  action_log_probs = tf.math.log(action_probs)
  actor_loss = -tf.math.reduce_sum(action_log_probs * advantage)

  critic_loss = huber_loss(values, returns)

  return actor_loss + critic_loss
```

[ ]:
```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)


@tf.function
def train_step(
    initial_state: tf.Tensor,
    model: tf.keras.Model,
    optimizer: tf.keras.optimizers.Optimizer,
    gamma: float,
    step: int,
    max_steps_per_episode: int) -> tf.Tensor:
  """Runs a model training step."""

  with tf.GradientTape() as tape:

    # Run the model for one episode to collect training data
    action_probs, values, rewards = run_episode(
        initial_state, model, max_steps_per_episode)

    # Calculate expected returns
```

```python
    returns = get_expected_return(rewards,values,step, gamma)

    # Convert training data to appropriate TF tensor shapes
    action_probs, values, returns = [
        tf.expand_dims(x, 1) for x in [action_probs, values, returns]]

    # Calculating loss values to update our network
    loss = compute_loss(action_probs, values, returns)

  # Compute the gradients from the loss
  grads = tape.gradient(loss, model.trainable_variables)

  # Apply the gradients to the model's parameters
  optimizer.apply_gradients(zip(grads, model.trainable_variables))

  episode_reward = tf.math.reduce_sum(rewards)

  return episode_reward
```

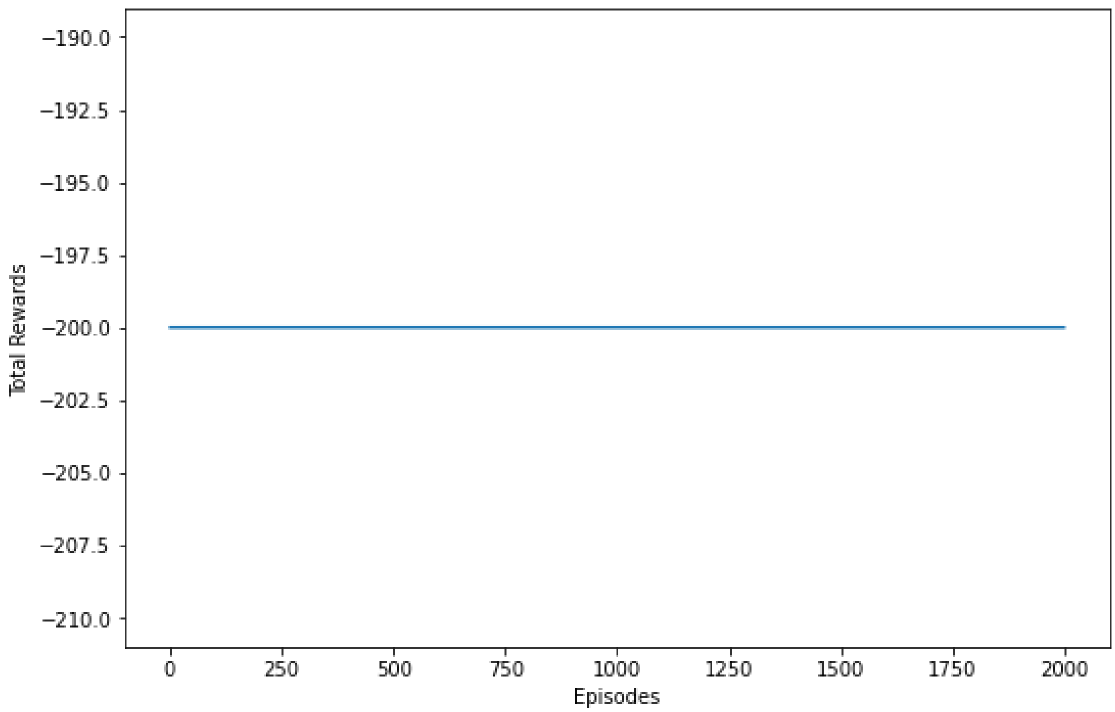
[ ]:
```python
%%time

min_episodes_criterion = 100
max_episodes = 2000
max_steps_per_episode = 1000


reward_threshold = 475
running_reward = 0

step = 10

gamma = 0.99

# Keep last episodes reward
episodes_reward: collections.deque = collections.
 ↪deque(maxlen=min_episodes_criterion)
reward_list =[]
with tqdm.trange(max_episodes) as t:
  for i in t:
    initial_state = tf.constant(env.reset(), dtype=tf.float32)
    episode_reward = int(train_step(
        initial_state, model, optimizer, gamma,step, max_steps_per_episode))

    episodes_reward.append(episode_reward)
    reward_list.append(episode_reward)
    running_reward = statistics.mean(episodes_reward)
```

```
    t.set_description(f'Episode {i}')
    t.set_postfix(
        episode_reward=episode_reward, running_reward=running_reward)

    # Show average episode reward every 10 episodes
    if i % 10 == 0:
      pass # print(f'Episode {i}: average reward: {avg_reward}')

    if running_reward > reward_threshold and i >= min_episodes_criterion:
        break

print(f'\nSolved at episode {i}: average reward: {running_reward:.2f}!')
```

```
Episode 1999: 100%|      | 2000/2000 [05:01<00:00,  6.64it/s,
episode_reward=-200, running_reward=-200]


Solved at episode 1999: average reward: -200.00!
CPU times: user 6min 42s, sys: 16.5 s, total: 6min 59s
Wall time: 5min 1s
```

[ ]:
```
mov_avg=[]
for i in range(1,len(reward_list)):
  if i > 100:
    mov_avg.append(np.mean(reward_list[-100:]))
  else:
    mov_avg.append(np.mean(reward_list[:i]))

plt.figure(figsize=(9,6))
plt.xlabel('Episodes')
plt.ylabel('Total Rewards')
plt.plot(np.arange(1,len(mov_avg)+1),mov_avg)
```

[ ]:
```
[<matplotlib.lines.Line2D at 0x7f7ef1983a90>]
```


−190.0
−192.5
−195.0
−197.5
−200.0
−202.5
−205.0
−207.5
−210.0
Total Rewards
0
250
500
750
1000
1250
1500
1750
2000
Episodes

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

Arcobat v1

Setup - DQN

References:

Please follow [Human-level control through deep reinforcement learning]() for the original publication as well as the psuedocode. Watch Prof. Ravi's lectures on moodle or nptel for further understanding the core concepts. Contact the TAs for further resources if needed.

Part 1: DQN

```
!pip install wandb

import wandb
wandb.login()
```

```
Collecting wandb
  Downloading wandb-0.12.11-py2.py3-none-any.whl (1.7 MB)
     |████████████████████████████████| 1.7 MB 5.3 MB/s
Requirement already satisfied: PyYAML in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: requests<3,>=2.0.0 in /usr/local/lib/python3
Collecting GitPython>=1.0.0
  Downloading GitPython-3.1.27-py3-none-any.whl (181 kB)
     |████████████████████████████████| 181 kB 32.9 MB/s
Collecting yaspin>=1.0.0
  Downloading yaspin-2.1.0-py3-none-any.whl (18 kB)
Requirement already satisfied: promise<3,>=2.0 in /usr/local/lib/python3.7/
Requirement already satisfied: six>=1.13.0 in /usr/local/lib/python3.7/dist
Collecting sentry-sdk>=1.0.0
  Downloading sentry_sdk-1.5.8-py2.py3-none-any.whl (144 kB)
     |████████████████████████████████| 144 kB 41.9 MB/s
Collecting docker-pycreds>=0.4.0
  Downloading docker_pycreds-0.4.0-py2.py3-none-any.whl (9.0 kB)
Requirement already satisfied: protobuf>=3.12.0 in /usr/local/lib/python3.7
Requirement already satisfied: Click!=8.0.0,>=7.0 in /usr/local/lib/python3
Collecting setproctitle
  Downloading setproctitle-1.2.2-cp37-cp37m-manylinux1_x86_64.whl (36 kB)
Requirement already satisfied: psutil>=5.0.0 in /usr/local/lib/python3.7/di
Collecting pathtools
  Downloading pathtools-0.1.2.tar.gz (11 kB)
Collecting shortuuid>=0.5.0
  Downloading shortuuid-1.0.8-py3-none-any.whl (9.5 kB)
Requirement already satisfied: python-dateutil>=2.6.1 in /usr/local/lib/pyt
Collecting gitdb<5,>=4.0.1
  Downloading gitdb-4.0.9-py3-none-any.whl (63 kB)
     |████████████████████████████████| 63 kB 1.4 MB/s
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib
Collecting smmap<6,>=3.0.1
```

```
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dis
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3
Requirement already satisfied: termcolor<2.0.0,>=1.1.0 in /usr/local/lib/py
Building wheels for collected packages: pathtools
  Building wheel for pathtools (setup.py) ... done
  Created wheel for pathtools: filename=pathtools-0.1.2-py3-none-any.whl si
  Stored in directory: /root/.cache/pip/wheels/3e/31/09/fa59cef12cdcfecc627
Successfully built pathtools
Installing collected packages: smmap, gitdb, yaspin, shortuuid, setproctitl
Successfully installed GitPython-3.1.27 docker-pycreds-0.4.0 gitdb-4.0.9 pa
wandb: You can find your API key in your browser here: https://wandb.ai/aut
wandb: Paste an API key from your profile and hit enter, or press ctrl+c to
wandb: Appending key for api.wandb.ai to your netrc file: /root/.netrc
True
```

```
'''
Installing packages for rendering the game on Colab
'''

!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
```

```
Requirement already satisfied: setuptools in /usr/local/lib/python3.7/dist-
Collecting setuptools
  Downloading setuptools-61.2.0-py3-none-any.whl (1.1 MB)
     |████████████████████████████████| 1.1 MB 4.9 MB/s
Installing collected packages: setuptools
  Attempting uninstall: setuptools
    Found existing installation: setuptools 57.4.0
    Uninstalling setuptools-57.4.0:
      Successfully uninstalled setuptools-57.4.0
ERROR: pip's dependency resolver does not currently take into account all t
tensorflow 2.8.0 requires tf-estimator-nightly==2.8.0.dev2021122109, which
datascience 0.10.6 requires folium==0.2.1, but you have folium 0.8.3 which
Successfully installed setuptools-61.2.0
```

```
!pip install tensorflow-gpu
```

Show hidden output

```
'''
A bunch of imports, you don't have to worry about these
'''
```

```
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 import Monitor
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
from scipy.special import softmax
seed = 42
rg = np.random.RandomState(seed)


'''
Please refer to the first tutorial for more details on the specifics of environments
We've only added important commands you might find useful for experiments.
'''

'''
List of example environments
(Source - https://gym.openai.com/envs/#classic_control)

'Acrobot-v1'
'CartPole-v0'
'MountainCar-v0'
'''

env = gym.make('Acrobot-v1')
ENV_NAME = 'Acrobot'
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("----")

'''
# Understanding State, Action, Reward Dynamics

The agent decides an action to take depending on the state.

The Environment keeps a variable specifically for the current state.
- Everytime an action is passed to the environment, it calculates the new state and updates
- It returns the new current state and reward for the agent to take the next action

'''
```

```
state = env.reset()
''' This returns the initial state (when environment is reset) '''

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

action = env.action_space.sample()
''' We take a random action now '''

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

next_state, reward, done, info = env.step(action)
''' env.step is used to calculate new state and obtain reward based on old state and action

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

```
6
3
2
----
[ 0.99603073 -0.08901003  0.99567135  0.09294385  0.02653819 -0.04199653]
----
2
----
[ 0.9964048  -0.08472003  0.99483904  0.10146566  0.01598951  0.12664371]
-1.0
False
{}
----
```

DQN

Using NNs as substitutes isn't something new. It has been tried earlier, but the 'human control' paper really popularised using NNs by providing a few stability ideas (Q-Targets, Experience Replay & Truncation). The 'Deep-Q Network' (DQN) Algorithm can be broken down into having the following components.

Q-Network:

The neural network used as a function approximator is defined below

```
'''
### Q Network & Some 'hyperparameters'

QNetwork1:
Input Layer - 4 nodes (State Shape) \
Hidden Layer 1 - 64 nodes \
Hidden Layer 2 - 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


'''
Bunch of Hyper parameters (Which you might have to tune later **wink wink**)
'''
BUFFER_SIZE = int(1e6)  #''' replay buffer size '''
BATCH_SIZE = 512         #''' minibatch size '''
GAMMA = 0.99          #''' discount factor '''
LR = 5.00e-04                #''' learning rate '''
UPDATE_EVERY = 20      #''' how often to update the network (When Q target is present) '''
POLICY= 'eps'
LAYERS = 3

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)
```

Replay Buffer:

This is a 'deque' that helps us store experiences. Recall why we use such a technique.

```
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",
        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])
        actions = torch.from_numpy(np.vstack([e.action for e in experiences if e is not None
        rewards = torch.from_numpy(np.vstack([e.reward for e in experiences if e is not None
        next_states = torch.from_numpy(np.vstack([e.next_state for e in experiences if e is
        dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is not None]).a

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

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

Truncation:

We add a line (optionally) in the code to truncate the gradient in hopes that it would help with the stability of the learning process.

Tutorial Agent Code:

```
class TutorialAgent():

    def __init__(self, state_size, action_size, policy,seed):

        ''' Agent Environment Interaction '''
        self.state_size = state_size
        self.action_size = action_size
        self.seed = random.seed(seed)
        self.policy = policy
```

```
        self.policy = policy

        ''' 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=LR)

        ''' Replay memory '''
        self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE, seed)

        ''' Initialize time step (for updating every UPDATE_EVERY steps)           -Needed f
        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) >= BATCH_SIZE:
            experiences = self.memory.sample()
            self.learn(experiences, GAMMA)

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

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

    def act(self, state, eps=0., beta=1):

        if self.policy == 'softmax':
          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()

          ''' Softmax action selection '''
          p = softmax(action_values.cpu().data.numpy())
          p = p.ravel()
          p /= p.sum()
          return rg.choice(  np.arange(self.action_size)  , p = p  )

        else :
          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, gamma):
        """ +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 + (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()
```

Here, we present the DQN algorithm code.

```
''' Defining DQN Algorithm '''

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

def dqn(n_episodes=2000, max_t=1000, eps_start=1.0, eps_end=0.01, eps_decay=0.995):

    scores = []
    ''' 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 '''

    eps = eps_start
    ''' initialize epsilon '''

    for i_episode in range(1, n_episodes+1):
        state = env.reset()
        score = 0
        steps = 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
            steps += 1
            if done:
                break
```

```
            break
        wandb.log({'score':score})
        wandb.log({'steps':steps})

        scores_window.append(score)
        scores_window_printing.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)
        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_wind
        if np.mean(scores_window)>=-80.0:
           print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.format(i_ep
           break
    return np.array(scores),i_episode-100

''' Trial run to check if algorithm runs and saves the data '''
```

```
' Trial run to check if algorithm runs and saves the data '
```

Run

[ ] ↳ *6 cells hidden*

Conventitional Testing

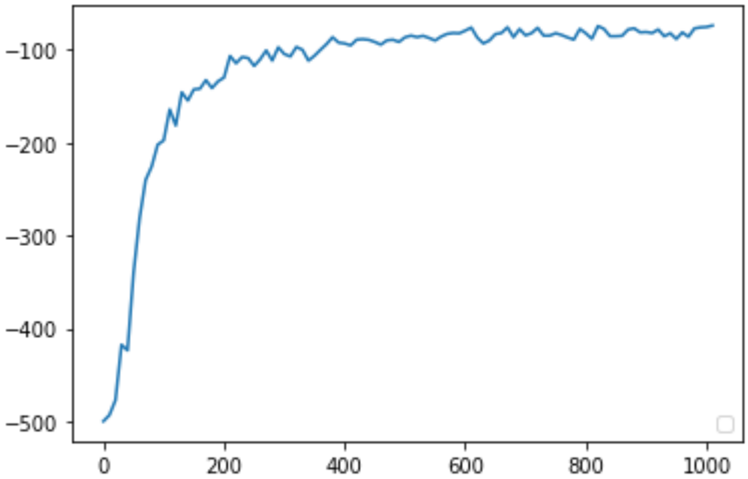
```
# from this website I got the followin information:

# BATCH_SIZE = 128
# GAMMA = 0.999
# EPS_START = 0.9
# EPS_END = 0.05
# EPS_DECAY = 200
# TARGET_UPDATE = 10
# num_episoeds = 50



# # Testing code
# POLICY= 'eps'

# agent = TutorialAgent(state_size=state_shape,action_size = action_shape, policy = POLICY,


# scores, epconv =  dqn()
# steps_in_exp.append(epconv)
# ##wandb.log({'Steps_taken':epconv})
```

```
# fig, ax = plt.subplots()
# x = [p*10 for p in list(range(len(scores)))]
# ax.plot(x,scores)
# ax.legend(loc='lower right')
# #wandb.log({'scores-v-epoch':fig})
# plt.show()
# #wandb.finish()


begin_time = datetime.datetime.now()

steps_in_exp = []
VAR = 7
for exp in range(0,2):

  config = {'buffer_size':BUFFER_SIZE, 'batch_size':BATCH_SIZE, 'gamma':GAMMA, 'lr':LR, 'Upc
  wandb.init(project = 'RLPA2', config = config, group = ENV_NAME, name = f'Var-{VAR}-Exp-{e

  agent = TutorialAgent(state_size=state_shape,action_size = action_shape, policy = POLICY,


  scores, epconv =  dqn()
  steps_in_exp.append(epconv)
  wandb.log({'Steps_taken':epconv})
  fig, ax = plt.subplots()
  x = [p*10 for p in list(range(len(scores)))]
  ax.plot(x,scores)
  ax.legend(loc='lower right')
  #wandb.log({'scores-v-epoch':fig})
  plt.show()
  wandb.finish()

config = {'buffer_size':BUFFER_SIZE, 'batch_size':BATCH_SIZE, 'gamma':GAMMA, 'lr':LR, 'Updat
wandb.init(project = 'RLPA2', config = config, group = ENV_NAME, name = f'Var-{VAR}', tags =

steps_in_exp = np.array(steps_in_exp)
wandb.log({'Average-Steps-Taken':np.mean(steps_in_exp)})

wandb.finish()
time_taken = datetime.datetime.now() - begin_time

print(time_taken)
```

```
wandb: Currently logged in as: sathvikjoel (use `wandb login --relogin` to
```

Tracking run with wandb version 0.12.11
Run data is saved locally in `/content/wandb/run-20220330_034440-z7vmi7co`
Syncing run **Var-7-Exp-0** to Weights & Biases (docs)

```
Episode 100   Average Score: -360.24
Episode 200   Average Score: -153.82
Episode 300   Average Score: -110.70
Episode 400   Average Score: -100.16
Episode 500   Average Score: -91.43
Episode 600   Average Score: -85.52
Episode 700   Average Score: -83.28
Episode 800   Average Score: -83.45
Episode 900   Average Score: -81.67
Episode 1000  Average Score: -81.94
Episode 1024  Average Score: -80.06No handles with labels found to put in le
Episode 1025  Average Score: -79.95
Environment solved in 925 episodes!  Average Score: -79.95
```



Waiting for W&B process to finish... **(success).**

0.009 MB of 0.009 MB uploaded (0.000 MB deduped)

Run history:

Steps_taken ▁

Run summary:

Steps_taken 925

Synced **Var-7-Exp-0**: https://wandb.ai/sathvikjoel/RLPA2/runs/z7vmi7co
Synced 5 W&B file(s), 0 media file(s), 0 artifact file(s) and 0 other file(s)
Find logs at: `./wandb/run-20220330_034440-z7vmi7co/logs`
Tracking run with wandb version 0.12.11
Run data is saved locally in `/content/wandb/run-20220330_040920-s9cm8o7s`
Syncing run **Var-7-Exp-1** to Weights & Biases (docs)

```
Episode 100   Average Score: -349.73
Episode 200   Average Score: -141.73
Episode 300   Average Score: -113.99
Episode 400   Average Score: -101.24
Episode 500   Average Score: -93.53
Episode 600   Average Score: -90.42
Episode 700   Average Score: -85.09
Episode 800   Average Score: -84.01
Episode 900   Average Score: -84.12
```

Mountain Car

```
!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 wandb

import wandb
wandb.login()
```

```
Failed to detect the name of this notebook, you can set it manually with th
wandb: Currently logged in as: sathvikjoel (use `wandb login --relogin` to
True
```

```
'''
A bunch of imports, you don't have to worry about these
'''

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 import Monitor
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
from scipy.special import softmax
seed = 42
rg = np.random.RandomState(seed)
'''
Please refer to the first tutorial for more details on the specifics of environments
We've only added important commands you might find useful for experiments.
'''

'''
List of example environments
```

```
'Acrobot-v1'
'CartPole-v0'
'MountainCar-v0'
'''

env = gym.make('MountainCar-v0')
ENV_NAME = 'MountainCar'
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("----")

'''
# Understanding State, Action, Reward Dynamics

The agent decides an action to take depending on the state.

The Environment keeps a variable specifically for the current state.
- Everytime an action is passed to the environment, it calculates the new state and updates
- It returns the new current state and reward for the agent to take the next action

'''

state = env.reset()
''' This returns the initial state (when environment is reset) '''

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

action = env.action_space.sample()
''' We take a random action now '''

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

next_state, reward, done, info = env.step(action)
''' env.step is used to calculate new state and obtain reward based on old state and action

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

```
2
3
1
----
[-0.47260767  0.        ]
----
2
----
```

```
[-0.47198862  0.00061906]
-1.0
False
{}
----
```

```
def run():
  with wandb.init(tags =['MountainCar-v0']):
    config = wandb.config
    '''
    ### 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


    '''
    Bunch of Hyper parameters (Which you might have to tune later **wink wink**)
    '''
    BUFFER_SIZE = config.buffer_size  #''' 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 tar
    POLICY = 'eps'

    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)
```

```
        return self.fc3(x)
```

Replay Buffer:

```
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", "rewa
        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 No
        actions = torch.from_numpy(np.vstack([e.action for e in experiences if e is not
        rewards = torch.from_numpy(np.vstack([e.reward for e in experiences if e is not
        next_states = torch.from_numpy(np.vstack([e.next_state for e in experiences if e
        dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is not None

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

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

Tutorial Agent Code:

```
class TutorialAgent():

    def __init__(self, state_size, action_size, policy,seed):

        ''' Agent Environment Interaction '''
```

```
        self.state_size = state_size
        self.action_size = action_size
        self.seed = random.seed(seed)
        self.policy = policy

        ''' 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=LR)

        ''' Replay memory '''
        self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE, seed)

        ''' Initialize time step (for updating every UPDATE_EVERY steps)           -Need
        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) >= BATCH_SIZE:
            experiences = self.memory.sample()
            self.learn(experiences, GAMMA)

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

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

    def act(self, state, eps=0., beta=1):

        if self.policy == 'softmax':
          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()

          ''' Softmax action selection '''
          p = softmax(action_values.cpu().data.numpy())
          p = p.ravel()
          p /= p.sum()
          return rg.choice(  np.arange(self.action_size)  , p = p  )

        else :
          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, gamma):
        """ +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(

        ''' Compute Q targets for current states '''
        Q_targets = rewards + (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()

### Here, we present the DQN algorithm code.

''' Defining DQN Algorithm '''

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

def dqn(n_episodes=10000, max_t=100000000, eps_start=1.0, eps_end=0.01, eps_decay=0.995)

    scores = []
    ''' 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 '''

    eps = eps_start
    ''' initialize epsilon '''

    for i_episode in range(1, n_episodes+1):
        state = env.reset()
        score = 0
        steps = 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
            steps += 1
            if done:
```

```
                break
            wandb.log({'score':score})
            wandb.log({'steps':steps})

            scores_window.append(score)
            scores_window_printing.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_win
            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_w
            if np.mean(scores_window)>=195.0:
              print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.format(i
              break
        return np.array(scores),i_episode-100

    ''' Trial run to check if algorithm runs and saves the data '''


    agent = TutorialAgent(state_size=state_shape,action_size = action_shape, policy = POLICY


    scores, epconv =  dqn()
    #np.save(f'/content/drive/MyDrive/{NUM}.npy', scores)
    print(epconv)



sweep_config = {
    "name" : f"{ENV_NAME}-config-sweep",
    "method": "random",
    "parameters": {
        "gamma": {
            "min": 0.990,
            "max": 0.999
        },
        "update_every": {
            "min": 10,
            "max": 50
        },
        "lr": {
            "min": 5e-4,
            "max": 5e-3
        },
        "buffer_size": {
            "values" : [int(1e4), int(1e5), int(1e6)]
        },
        "batch_size": {
            "values": [64, 128, 256]
        },
    }
}
```

```
sweep_id = wandb.sweep(sweep_config, project='RLPA2')
```

```
Create sweep with ID: oj1god07
Sweep URL: https://wandb.ai/sathvikjoel/RLPA2/sweeps/oj1god07
```

```
wandb.agent(sweep_id, run, count=8)
```

```
wandb: Agent Starting Run: vsr43gmu with config:
wandb:    batch_size: 128
wandb:    buffer_size: 10000
wandb:    gamma: 0.9983518582916392
wandb:    lr: 0.0015536694087040783
wandb:    update_every: 32
Failed to detect the name of this notebook, you can set it manually with th
```

Tracking run with wandb version 0.12.11
Run data is saved locally in /home/joel/Insync/cs19b025@smail.iitm.ac.in/Google Drive/Documents/Sem6-drive/RL/Assignments/2Assignment/wandb/run-20220330_031009-vsr43gmu
Syncing run **mild-sweep-1** to Weights & Biases (docs)
Sweep page: https://wandb.ai/sathvikjoel/RLPA2/sweeps/oj1god07

```
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: -199.91
Episode 900   Average Score: -200.00
Episode 1000  Average Score: -199.95
Episode 1100  Average Score: -191.57
Episode 1200  Average Score: -188.66
Episode 1300  Average Score: -177.16
Episode 1400  Average Score: -183.50
Episode 1500  Average Score: -160.84
Episode 1600  Average Score: -141.08
Episode 1700  Average Score: -154.44
Episode 1800  Average Score: -169.52
Episode 1900  Average Score: -196.62
Episode 2000  Average Score: -200.00
Episode 2100  Average Score: -200.00
Episode 2200  Average Score: -200.00
Episode 2300  Average Score: -200.00
Episode 2400  Average Score: -200.00
Episode 2500  Average Score: -200.00
Episode 2600  Average Score: -200.00
Episode 2700  Average Score: -200.00
Episode 2800  Average Score: -200.00
Episode 2900  Average Score: -200.00
Episode 3000  Average Score: -199.77
Episode 3100  Average Score: -200.00
Episode 3200  Average Score: -200.00
Episode 3300  Average Score: -200.00
Episode 3400  Average Score: -200.00
Episode 3500  Average Score: -200.00
Episode 3600  Average Score: -200.00
Episode 3700  Average Score: -200.00
Episode 3800  Average Score: -200.00
```

```
Episode 3800 Average Score: -200.00
Episode 3900 Average Score: -199.47
Episode 4000 Average Score: -200.00
Episode 4100 Average Score: -200.00
Episode 4200 Average Score: -200.00
Episode 4300 Average Score: -200.00
```

Cartpole-v0

Setup - DQN

References:

Please follow Human-level control through deep reinforcement learning for the original publication as well as the psuedocode. Watch Prof. Ravi's lectures on moodle or nptel for further understanding the core concepts. Contact the TAs for further resources if needed.

Part 1: DQN

```
!pip install wandb

import wandb
wandb.login()
```

```
Collecting wandb
  Downloading wandb-0.12.11-py2.py3-none-any.whl (1.7 MB)
     |████████████████████████████████| 1.7 MB 5.3 MB/s
Requirement already satisfied: promise<3,>=2.0 in /usr/local/lib/python3.7/di
Requirement already satisfied: Click!=8.0.0,>=7.0 in /usr/local/lib/python3.7
Requirement already satisfied: protobuf>=3.12.0 in /usr/local/lib/python3.7/d
Requirement already satisfied: python-dateutil>=2.6.1 in /usr/local/lib/pytho
Requirement already satisfied: PyYAML in /usr/local/lib/python3.7/dist-packag
Collecting sentry-sdk>=1.0.0
  Downloading sentry_sdk-1.5.8-py2.py3-none-any.whl (144 kB)
     |████████████████████████████████| 144 kB 45.3 MB/s
Collecting pathtools
  Downloading pathtools-0.1.2.tar.gz (11 kB)
Requirement already satisfied: six>=1.13.0 in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: psutil>=5.0.0 in /usr/local/lib/python3.7/dist
Requirement already satisfied: requests<3,>=2.0.0 in /usr/local/lib/python3.7
Collecting setproctitle
  Downloading setproctitle-1.2.2-cp37-cp37m-manylinux1_x86_64.whl (36 kB)
Collecting docker-pycreds>=0.4.0
  Downloading docker_pycreds-0.4.0-py2.py3-none-any.whl (9.0 kB)
```

```
'''
Installing packages for rendering the game on Colab
'''

!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
```

```
Requirement already satisfied: setuptools in /usr/local/lib/python3.7/dist-
Collecting setuptools
  Downloading setuptools-61.1.1-py3-none-any.whl (1.1 MB)
     |████████████████████████████████| 1.1 MB 5.4 MB/s
Installing collected packages: setuptools
  Attempting uninstall: setuptools
    Found existing installation: setuptools 57.4.0
    Uninstalling setuptools-57.4.0:
      Successfully uninstalled setuptools-57.4.0
ERROR: pip's dependency resolver does not currently take into account all t
tensorflow 2.8.0 requires tf-estimator-nightly==2.8.0.dev2021122109, which
datascience 0.10.6 requires folium==0.2.1, but you have folium 0.8.3 which
Successfully installed setuptools-61.1.1
```

```
!pip install tensorflow-gpu
```

Show hidden output

```
'''
```

```
A bunch of imports, you don't have to worry about these
'''

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 import Monitor
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
from scipy.special import softmax
seed = 42
rg = np.random.RandomState(seed)

'''
Please refer to the first tutorial for more details on the specifics of environme
We've only added important commands you might find useful for experiments.
'''

'''
List of example environments
(Source - https://gym.openai.com/envs/#classic_control)

'Acrobot-v1'
'CartPole-v0'
'MountainCar-v0'
'''

env = gym.make('CartPole-v0')
ENV_NAME = 'CartPole'
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("----")

'''
```

```
# Understanding State, Action, Reward Dynamics

The agent decides an action to take depending on the state.

The Environment keeps a variable specifically for the current state.
- Everytime an action is passed to the environment, it calculates the new state a
- It returns the new current state and reward for the agent to take the next acti

'''

state = env.reset()
''' This returns the initial state (when environment is reset) '''

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

action = env.action_space.sample()
''' We take a random action now '''

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

next_state, reward, done, info = env.step(action)
''' env.step is used to calculate new state and obtain reward based on old state

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

```
4
2
1
----
[-0.04456399  0.04653909  0.01326909 -0.02099827]
----
0
----
[-0.04363321 -0.14877061  0.01284913  0.2758415 ]
1.0
False
{}
----
```

DQN

Using NNs as substitutes isn't something new. It has been tried earlier, but the 'human control' paper really popularised using NNs by providing a few stability ideas (Q-Targets, Experience Replay & Truncation). The 'Deep-Q Network' (DQN) Algorithm can be broken down into having the following components.

Q-Network:

The neural network used as a function approximator is defined below

```
'''
### 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


'''
Bunch of Hyper parameters (Which you might have to tune later **wink wink**)
'''
BUFFER_SIZE = int(1e6)  #''' replay buffer size '''
BATCH_SIZE = 128         #''' minibatch size '''
GAMMA = 0.999            #''' discount factor '''
LR = 1.00e-04               #''' learning rate '''
UPDATE_EVERY = 30      #''' how often to update the network (When Q target is prese


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)
```

Replay Buffer:

This is a 'deque' that helps us store experiences. Recall why we use such a technique.

```
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",
        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 r
        actions = torch.from_numpy(np.vstack([e.action for e in experiences if e is
        rewards = torch.from_numpy(np.vstack([e.reward for e in experiences if e is
        next_states = torch.from_numpy(np.vstack([e.next_state for e in experiences
        dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is not

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

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

Truncation:

We add a line (optionally) in the code to truncate the gradient in hopes that it would help with the stability of the learning process.

Tutorial Agent Code:

```
class TutorialAgent():

    def __init__(self, state_size, action_size, policy,seed):

        ''' Agent Environment Interaction '''
        self.state_size = state_size
        self.action_size = action_size
        self.seed = random.seed(seed)
        self.policy = policy

        ''' 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=LR)

        ''' Replay memory '''
        self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE, seed)

        ''' Initialize time step (for updating every UPDATE_EVERY steps)
        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) >= BATCH_SIZE:
            experiences = self.memory.sample()
            self.learn(experiences, GAMMA)

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

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

    def act(self, state, eps=0., beta=1):

        if self.policy == 'softmax':
          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()

            ''' Softmax action selection '''
            p = softmax(action_values.cpu().data.numpy())
            p = p.ravel()
            p /= p.sum()
            return rg.choice(  np.arange(self.action_size)  , p = p  )

          else :
            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, gamma):
        """ +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].unsqu

        ''' Compute Q targets for current states '''
        Q_targets = rewards + (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()
```

Here, we present the DQN algorithm code.

```
''' Defining DQN Algorithm '''

state_shape = env.observation_space.shape[0]
```

```
action_shape = env.action_space.n

def dqn(n_episodes=7000, max_t=1000, eps_start=1.0, eps_end=0.01, eps_decay=0.995):

    scores = []
    ''' 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 '''

    eps = eps_start
    ''' initialize epsilon '''

    for i_episode in range(1, n_episodes+1):
        state = env.reset()
        score = 0
        steps = 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
            steps += 1
            if done:
                break
        wandb.log({'score':score})
        wandb.log({'steps':steps})

        scores_window.append(score)
        scores_window_printing.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(score
        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(sc
        if np.mean(scores_window)>=195.0:
           print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.fo
           break
    return np.array(scores),i_episode-100

''' Trial run to check if algorithm runs and saves the data '''
```

' Trial run to check if algorithm runs and saves the data '

```
begin_time = datetime.datetime.now()
POLICY= 'eps'
steps_in_exp = []
VAR = 10
for exp in range(0,4):

  config = {'buffer_size':BUFFER_SIZE, 'batch_size':BATCH_SIZE, 'gamma':GAMMA, 'lr'
  wandb.init(project = 'RLPA2', config = config, group = ENV_NAME, name = f'Var-{VA

  agent = TutorialAgent(state_size=state_shape,action_size = action_shape, policy =


  scores, epconv =  dqn()
  steps_in_exp.append(epconv)
  wandb.log({'Steps_taken':epconv})
  fig, ax = plt.subplots()
  x = [p*10 for p in list(range(len(scores)))]
  ax.plot(x,scores)
  ax.legend(loc='lower right')
  wandb.log({'scores-v-epoch':fig})
  plt.show()
  wandb.finish()

config = {'buffer_size':BUFFER_SIZE, 'batch_size':BATCH_SIZE, 'gamma':GAMMA, 'lr':L
wandb.init(project = 'RLPA2', config = config, group = ENV_NAME, name = f'Var-{VAR}

steps_in_exp = np.array(steps_in_exp)
wandb.log({'Average-Steps-Taken':np.mean(steps_in_exp)})

wandb.finish()
time_taken = datetime.datetime.now() - begin_time

print(time_taken)
```

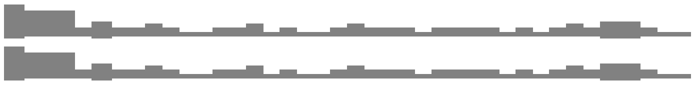
Finishing last run (ID:11rpy78n) before initializing another...

Waiting for W&B process to finish... **(success).**

0.009 MB of 0.009 MB uploaded (0.000 MB deduped)

Run history:

score
steps



Run summary:

score	9.0
steps	9

Synced **Var-9-Exp-1**: https://wandb.ai/sathvikjoel/RLPA2/runs/11rpy78n
Synced 5 W&B file(s), 0 media file(s), 0 artifact file(s) and 0 other file(s)
Find logs at: `./wandb/run-20220327_044543-11rpy78n/logs`
Successfully finished last run (ID:11rpy78n). Initializing new run:
Tracking run with wandb version 0.12.11
Run data is saved locally in `/content/wandb/run-20220327_044755-204pqffy`
Syncing run **Var-10-Exp-0** to Weights & Biases (docs)

```
Episode 100     Average Score: 20.15
Episode 200     Average Score: 57.67
Episode 300     Average Score: 160.35
Episode 356     Average Score: 194.37No handles with labels found to put in
Episode 357     Average Score: 195.11
Environment solved in 257 episodes!     Average Score: 195.11
/usr/local/lib/python3.7/dist-packages/plotly/matplotlib/mplexporter/expo

Legend element <matplotlib.offsetbox.HPacker object at 0x7f8533bd1b50> not

/usr/local/lib/python3.7/dist-packages/plotly/matplotlib/renderer.py:613:

I found a path object that I don't think is part of a bar chart. Ignoring.
```

Task 1a

Understand the core of the algorithm, follow the flow of data. Identify the exploration strategy used.

Task 1b

Out of the two exploration strategies discussed in class (ϵ-greedy & Softmax). Implement the strategy that's not used here.

Task 1c

How fast does the agent 'solve' the environment in terms of the number of episodes? (CartPole-v0 defines "solving" as getting average reward of 195.0 over 100 consecutive trials. SOURCE - https://gym.openai.com/envs/CartPole-v0/)

How 'well' does the agent learn? (reward plot?) The above two are some 'evaluation metrics' you can use to comment on the performance of an algorithm.

Please compare DQN (using ϵ-greedy) with DQN (using softmax). Think along the lines of 'no. of episodes', 'reward plots', 'compute time', etc. and add a few comments.

Task 1d (Optional)

Take a look at the official submissions page on [OpenAI gym's CartPole v-0 evaluations]

```
Episode 200	Average Score: 92.75
```

Submission Steps

Task 1: Add a text cell with the answer.

Task 2: Add a code cell below task 1 solution and use 'Tutorial Agent Code' to build your new agent (with a different exploration strategy).

Task 3: Add a code cell below task 2 solution running both the agents to solve the CartPole v-0 environment and add a new text cell below it with your inferences.

Task 1a

The exploration stratagy used is epsilon greedy where epsilon decays over time, that is as the agent learn the exploration is decreased

Run history:

Task 1b

score


```
from scipy.special import softmax
seed = 42
rg = np.random.RandomState(seed)

class MyTutorialAgent(TutorialAgent):
  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()

        ''' Softmax action selection '''
        p = softmax(action_values.cpu().data.numpy())
        p = p.ravel()
        p /= p.sum()
        return rg.choice(  np.arange(self.action_size)  , p = p  )


        # if random.random() > eps:
        #     return np.argmax(action_values.cpu().data.numpy())
```

```
    # else:
    #     return random.choice(np.arange(self.action_size))
```

Task 1c

```
agent = MyTutorialAgent(state_size=state_shape,action_size = action_shape,seed = 0
s_scores = dqn()
```

```
Episode 100     Average Score: 110.30
Episode 200     Average Score: 161.68
Episode 300     Average Score: 28.08
Episode 400     Average Score: 37.61
Episode 500     Average Score: 107.09
Episode 600     Average Score: 72.85
Episode 673     Average Score: 195.52
Environment solved in 573 episodes!     Average Score: 195.52
```

```
s_scores_l = []
e_scores_l = []

for i in range(3):
  agent = TutorialAgent(state_size=state_shape,action_size = action_shape,seed = 0
  e_scores = dqn()
  e_scores_l.append(e_scores)

  agent = MyTutorialAgent(state_size=state_shape,action_size = action_shape,seed =
  s_scores = dqn()
  s_scores_l.append(s_scores)
```

```
Episode 100     Average Score: 38.65
Episode 200     Average Score: 136.83
Episode 300     Average Score: 58.50
Episode 400     Average Score: 57.59
Episode 500     Average Score: 41.70
Episode 600     Average Score: 163.05
Episode 700     Average Score: 180.40
Episode 800     Average Score: 157.15
Episode 900     Average Score: 80.60
Episode 1000    Average Score: 54.46
Episode 1100    Average Score: 69.31
Episode 1200    Average Score: 185.66
Episode 1218    Average Score: 195.98
Environment solved in 1118 episodes!    Average Score: 195.98
Episode 100     Average Score: 150.03
Episode 126     Average Score: 195.20
Environment solved in 26 episodes!      Average Score: 195.20
Episode 100     Average Score: 40.82
Episode 200     Average Score: 124.14
Episode 300     Average Score: 173.90
Episode 400     Average Score: 50.38
Episode 500     Average Score: 34.62
Episode 600     Average Score: 33.04
Episode 700     Average Score: 20.58
Episode 800     Average Score: 17.69
Episode 900     Average Score: 18.61
Episode 1000    Average Score: 9.35
```

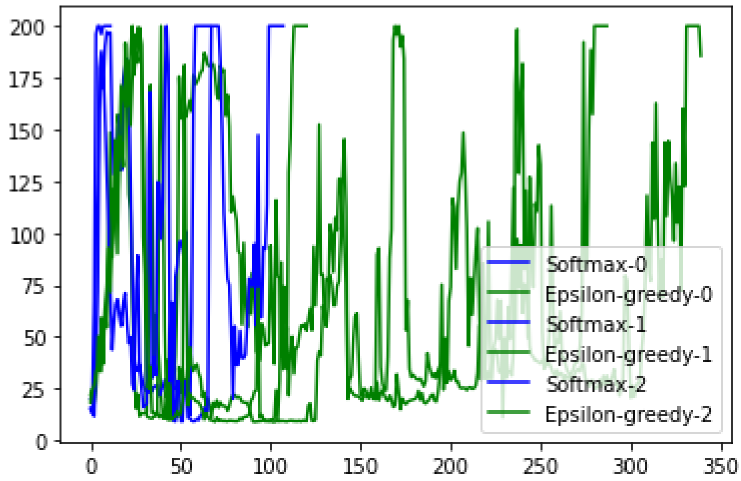
```
Episode 1100    Average Score: 9.33
Episode 1200    Average Score: 9.50
Episode 1300    Average Score: 26.08
Episode 1400    Average Score: 75.06
Episode 1500    Average Score: 53.95
Episode 1600    Average Score: 28.29
Episode 1700    Average Score: 82.77
Episode 1800    Average Score: 118.31
Episode 1900    Average Score: 31.38
Episode 2000    Average Score: 34.17
Episode 2100    Average Score: 27.64
Episode 2200    Average Score: 32.65
Episode 2300    Average Score: 36.08
Episode 2400    Average Score: 89.38
Episode 2500    Average Score: 69.61
Episode 2600    Average Score: 35.89
Episode 2700    Average Score: 32.01
Episode 2800    Average Score: 104.35
Episode 2884    Average Score: 195.01
Environment solved in 2784 episodes!    Average Score: 195.01
Episode 100     Average Score: 104.71
Episode 200     Average Score: 157.89
Episode 300     Average Score: 75.97
Episode 400     Average Score: 24.66
Episode 500     Average Score: 44.86
Episode 600     Average Score: 103.75
Episode 670     Average Score: 196.23
Environment solved in 570 episodes!     Average Score: 196.23
Episode 100     Average Score: 40.51
Episode 200     Average Score: 125.21
Episode 300     Average Score: 182.53
```

```
avg =[]
for i in range(3):
  print(f"Epsilon greedy DQN Expriement {i}: \t Environment solved in {e_scores_l[i
  avg.append(e_scores_l[i][1])
avg = np.array(avg)
print(f"Average is {np.mean(avg)}")

avg = []
for i in range(3):
  print(f"Softmax DQN Expriement {i}: \t Environment solved in {s_scores_l[i][1]} e
  avg.append(s_scores_l[i][1])
avg = np.array(avg)
print(f"Average is {np.mean(avg)}")
```

```
Epsilon greedy DQN Expriement 0:        Environment solved in 1118 episodes

Epsilon greedy DQN Expriement 1:        Environment solved in 2784 episodes

Epsilon greedy DQN Expriement 2:        Environment solved in 3304 episodes

Average is 2402.0
Softmax DQN Expriement 0:       Environment solved in 26 episodes

Softmax DQN Expriement 1:       Environment solved in 570 episodes

Softmax DQN Expriement 2:       Environment solved in 984 episodes
```

```
Average is 526.6666666666666
```

```
for i in range(3):
  plt.plot(s_scores_l[i][0], 'b', label = f"Softmax-{i}")
  plt.plot(e_scores_l[i][0], 'g', label = f"Epsilon-greedy-{i}")
plt.legend(loc='lower right')
plt.show()
```



Conclusion :

From the above expriments it is clear that softmax was able to achieve 195 score in lot less episodes than Epsilon greedy. So, softmax is prefarable in this settings

Cartpole v1

```
#!pip install wandb

import wandb
wandb.login()
```

```
Failed to detect the name of this notebook, you can set it manually with th
wandb: Currently logged in as: sathvikjoel (use `wandb login --relogin` to
True
```

```
'''
A bunch of imports, you don't have to worry about these
'''

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 import Monitor
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
from scipy.special import softmax
seed = 42
rg = np.random.RandomState(seed)
'''
Please refer to the first tutorial for more details on the specifics of environments
We've only added important commands you might find useful for experiments.
'''

'''
List of example environments
(Source - https://gym.openai.com/envs/#classic_control)

'Acrobot-v1'
'CartPole-v0'
'MountainCar-v0'
'''

env = gym.make('CartPole-v1')
ENV_NAME = 'CartPole-v1'
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("----")

'''
# Understanding State, Action, Reward Dynamics

The agent decides an action to take depending on the state.

The Environment keeps a variable specifically for the current state.
- Everytime an action is passed to the environment, it calculates the new state and updates
- It returns the new current state and reward for the agent to take the next action

'''

state = env.reset()
''' This returns the initial state (when environment is reset) '''

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

action = env.action_space.sample()
''' We take a random action now '''

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

next_state, reward, done, info = env.step(action)
''' env.step is used to calculate new state and obtain reward based on old state and action

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

```
4
2
0
----
[ 0.01369617 -0.02302133 -0.04590265 -0.04834723]
----
0
----
[ 0.01323574 -0.21745604 -0.04686959  0.22950698]
1.0
False
{}
----
```

```
def run():
  with wandb.init(tags =['CartPole-v0']):
    config = wandb.config
```

```
'''
### 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


'''
Bunch of Hyper parameters (Which you might have to tune later **wink wink**)
'''
BUFFER_SIZE = config.buffer_size  #''' 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 tar
POLICY = 'eps'

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)

### Replay Buffer:


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")
```

```
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", "rewa
        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 No
        actions = torch.from_numpy(np.vstack([e.action for e in experiences if e is not
        rewards = torch.from_numpy(np.vstack([e.reward for e in experiences if e is not
        next_states = torch.from_numpy(np.vstack([e.next_state for e in experiences if e
        dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is not None

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

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


## Tutorial Agent Code:


class TutorialAgent():

    def __init__(self, state_size, action_size, policy,seed):

        ''' Agent Environment Interaction '''
        self.state_size = state_size
        self.action_size = action_size
        self.seed = random.seed(seed)
        self.policy = policy

        ''' 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=LR)
```

```
        ''' Replay memory '''
        self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE, seed)

        ''' Initialize time step (for updating every UPDATE_EVERY steps)           -Need
        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) >= BATCH_SIZE:
            experiences = self.memory.sample()
            self.learn(experiences, GAMMA)

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

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

    def act(self, state, eps=0., beta=1):

        if self.policy == 'softmax':
          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()

          ''' Softmax action selection '''
          p = softmax(action_values.cpu().data.numpy())
          p = p.ravel()
          p /= p.sum()
          return rg.choice(  np.arange(self.action_size)  , p = p  )

        else :
          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, gamma):
        """ +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(

        ''' Compute Q targets for current states '''
        Q_targets = rewards + (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()

### Here, we present the DQN algorithm code.

''' Defining DQN Algorithm '''

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

def dqn(n_episodes=2000, max_t=100000000, eps_start=1.0, eps_end=0.01, eps_decay=0.995):

    scores = []
    ''' 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 '''

    eps = eps_start
    ''' initialize epsilon '''

    for i_episode in range(1, n_episodes+1):
        state = env.reset()
        score = 0
        steps = 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
            steps += 1
            if done:
                break
        wandb.log({'score':score})
        wandb.log({'steps':steps})

        scores_window.append(score)
        scores_window_printing.append(score)
        ''' save most recent score '''

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

```
            decrease epsilon

        print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_win
        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_w
        if np.mean(scores_window)>=475.0:
          print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.format(i
          break
    return np.array(scores),i_episode-100

''' Trial run to check if algorithm runs and saves the data '''


agent = TutorialAgent(state_size=state_shape,action_size = action_shape, policy = POLICY


scores, epconv =  dqn()
#np.save(f'/content/drive/MyDrive/{NUM}.npy', scores)
print(epconv)
```

```
sweep_config = {
    "name" : f"{ENV_NAME}-config-sweep",
    "method": "random",
    "parameters": {
        "gamma": {
            "min": 0.9970,
            "max": 0.9990
        },
        "update_every": {
            "min": 35,
            "max": 40
        },
        "lr": {
            "min": 0.029,
            "max": 0.040
        },
        "buffer_size": {
            "values" : [ int(1e5)]
        },
        "batch_size": {
            "values": [128]
        },
    }
}
```

```
sweep_id = wandb.sweep(sweep_config, project='RLPA2')
```

```
Create sweep with ID: 1kju6lmv
Sweep URL: https://wandb.ai/sathvikjoel/RLPA2/sweeps/1kju6lmv
```

```
wandb.agent(sweep_id, run, count=5)
```

```
wandb: Agent Starting Run: lq54w42z with config:
wandb:    batch_size: 128
```

```
wandb:     buffer_size: 100000
wandb:     gamma: 0.9970396370760792
wandb:     lr: 0.030557700330308843
wandb:     update_every: 36
Failed to detect the name of this notebook, you can set it manually with th
```

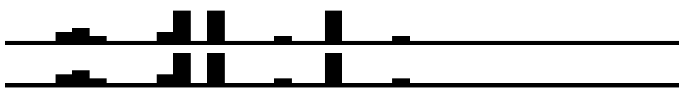
Tracking run with wandb version 0.12.11
Run data is saved locally in `/home/joel/Insync/cs19b025@smail.iitm.ac.in/Google Drive/Documents/Sem6-drive/RL/Assignments/2Assignment/wandb/run-20220330_233849-lq54w42z`
Syncing run **curious-sweep-1** to Weights & Biases (docs)
Sweep page: https://wandb.ai/sathvikjoel/RLPA2/sweeps/1kju6lmv

```
Episode 100   Average Score: 24.87
Episode 200   Average Score: 87.47
Episode 300   Average Score: 88.822
Episode 400   Average Score: 11.31
Episode 500   Average Score: 153.33
Episode 600   Average Score: 200.14
Episode 700   Average Score: 137.65
Episode 800   Average Score: 52.298
Episode 900   Average Score: 91.526
Episode 1000  Average Score: 140.67
Episode 1100  Average Score: 67.815
Episode 1200  Average Score: 45.93
Episode 1300  Average Score: 34.74
Episode 1400  Average Score: 11.35
Episode 1500  Average Score: 10.23
Episode 1600  Average Score: 10.24
Episode 1700  Average Score: 12.09
Episode 1800  Average Score: 41.91
Episode 1900  Average Score: 24.17
Episode 2000  Average Score: 15.97
1900
```

Waiting for W&B process to finish... **(success).**

Run history:

score
steps



Run summary:

score	14.0
steps	14

Synced **curious-sweep-1**: https://wandb.ai/sathvikjoel/RLPA2/runs/lq54w42z
Synced 6 W&B file(s), 0 media file(s), 0 artifact file(s) and 0 other file(s)
Find logs at: `./wandb/run-20220330_233849-lq54w42z/logs`

```
wandb: Sweep Agent: Waiting for job.
wandb: Job received.
wandb: Agent Starting Run: w4c9l64p with config:
wandb:     batch_size: 128
wandb:     buffer_size: 100000
wandb:     gamma: 0.99818720499242
wandb:     lr: 0.035067038865898195
wandb:     update_every: 37
Failed to detect the name of this notebook, you can set it manually with th
```

Tracking run with wandb version 0.12.11
Run data is saved locally in `/home/joel/Insync/cs19b025@smail.iitm.ac.in/Google`