Tutorial_4_DQN_and_AC

March 8, 2022

1 Tutorial 4 - DQN and Actor-Critic

Please follow this tutorial to understand the structure (code) of DQNs & get familiar with Actor Critic methods.

1.0.1 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.

1.1 Part 1: DQN

[1]:
```
'''
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 (57.4.0)
Collecting setuptools
  Downloading setuptools-60.9.3-py3-none-any.whl (1.1 MB)
     || 1.1 MB 4.3 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
the packages that are installed. This behaviour is the source of the following
dependency conflicts.
tensorflow 2.8.0 requires tf-estimator-nightly==2.8.0.dev2021122109, which is
not installed.
datascience 0.10.6 requires folium==0.2.1, but you have folium 0.8.3 which is
incompatible.
Successfully installed setuptools-60.9.3
```

[1]:
```
!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 28 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: gast>=0.2.1 in /usr/local/lib/python3.7/dist-
packages (from tensorflow-gpu) (0.5.3)
Requirement already satisfied: astunparse>=1.6.0 in /usr/local/lib/python3.7
/dist-packages (from tensorflow-gpu) (1.6.3)
Requirement already satisfied: six>=1.12.0 in /usr/local/lib/python3.7/dist-
packages (from tensorflow-gpu) (1.15.0)
Requirement already satisfied: wrapt>=1.11.0 in /usr/local/lib/python3.7/dist-
packages (from tensorflow-gpu) (1.13.3)
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: 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: setuptools in /usr/local/lib/python3.7/dist-
packages (from tensorflow-gpu) (60.9.3)
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: opt-einsum>=2.3.2 in /usr/local/lib/python3.7
/dist-packages (from tensorflow-gpu) (3.3.0)
Requirement already satisfied: flatbuffers>=1.12 in /usr/local/lib/python3.7
/dist-packages (from tensorflow-gpu) (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: h5py>=2.9.0 in /usr/local/lib/python3.7/dist-
packages (from tensorflow-gpu) (3.1.0)
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 51.1 MB/s
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: 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: grpcio<2.0,>=1.24.3 in /usr/local/lib/python3.7
/dist-packages (from tensorflow-gpu) (1.44.0)
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: 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: 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: 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: 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-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: 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: 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: 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) (4.11.2)
```

```
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: 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: 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: 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
```

[2]:

```python
'''
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
```

[3]:
```
'''
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
1
----
[-0.04456399  0.04653909  0.01326909 -0.02099827]
----
0
----
[-0.04363321 -0.14877061  0.01284913  0.2758415 ]
1.0
False
{}
----
```

1.2 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.

1.2.1 Q-Network:

The neural network used as a function approximator is defined below

[8]:
```
'''
### 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(1e5)  #''' replay buffer size '''
BATCH_SIZE = 64         #''' minibatch size '''
GAMMA = 0.99            #''' discount factor '''
LR = 5e-4               #''' learning rate '''
UPDATE_EVERY = 20       #''' how often to update the network (When Q target is␣
↪present) '''


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)
```

1.2.2 Replay Buffer:

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

[9]:
```
import random
import torch
import numpy as np
from collections import deque, namedtuple

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

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

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

        Params
        ======
            action_size (int): dimension of each action
            buffer_size (int): maximum size of buffer
            batch_size (int): size of each training batch
            seed (int): random seed
        """
        self.action_size = action_size
        self.memory = deque(maxlen=buffer_size)
        self.batch_size = batch_size
        self.experience = namedtuple("Experience", field_names=["state",␣
 ↪"action", "reward", "next_state", "done"])
        self.seed = random.seed(seed)

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

    def sample(self):
        """Randomly sample a batch of experiences from memory."""
        experiences = random.sample(self.memory, k=self.batch_size)

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

```
        dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is␣
↪not None]).astype(np.uint8)).float().to(device)

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

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

1.3 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.

1.4 Tutorial Agent Code:

[10]:
```
class TutorialAgent():

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

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

        ''' 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 for Q Targets '''
        self.t_step = 0

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

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

        ''' If enough samples are available in memory, get random subset and␣
↪learn '''
```

```
        if len(self.memory) >= 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.):

        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()
```

1.4.1 Here, we present the DQN algorithm code.

[11]:
```
''' Defining DQN Algorithm '''

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

def dqn(n_episodes=10000, 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
        for t in range(max_t):
            action = agent.act(state, eps)
            next_state, reward, done, _ = env.step(action)
            agent.step(state, action, reward, next_state, done)
            state = next_state
            score += reward
            if done:
                break

        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)), end="")
        if i_episode % 10 == 0:
            scores.append(np.mean(scores_window_printing))
        if i_episode % 100 == 0:
           print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.
 ↪mean(scores_window)))
        if np.mean(scores_window)>=195.0:
           print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.
 ↪2f}'.format(i_episode-100, np.mean(scores_window)))
           break
    return [np.array(scores),i_episode-100]

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

begin_time = datetime.datetime.now()
agent = TutorialAgent(state_size=state_shape,action_size = action_shape,seed =␣
 ↪0)


dqn()


time_taken = datetime.datetime.now() - begin_time

print(time_taken)
```

```
Episode 100     Average Score: 38.69
Episode 200     Average Score: 111.40
Episode 300     Average Score: 133.89
Episode 400     Average Score: 16.51
Episode 500     Average Score: 108.32
Episode 600     Average Score: 29.27
Episode 700     Average Score: 93.15
Episode 800     Average Score: 100.44
Episode 900     Average Score: 132.83
Episode 1000    Average Score: 114.17
Episode 1100    Average Score: 74.54
Episode 1200    Average Score: 84.73
Episode 1300    Average Score: 164.76
Episode 1400    Average Score: 109.28
Episode 1500    Average Score: 105.75
Episode 1600    Average Score: 85.55
Episode 1700    Average Score: 145.35
Episode 1800    Average Score: 169.25
```

```
Episode 1900	Average Score: 43.73
Episode 2000	Average Score: 9.35
Episode 2100	Average Score: 9.39
Episode 2200	Average Score: 9.32
Episode 2300	Average Score: 9.26
Episode 2400	Average Score: 9.43
Episode 2500	Average Score: 9.61
Episode 2600	Average Score: 9.44
Episode 2700	Average Score: 9.48
Episode 2800	Average Score: 9.45
Episode 2900	Average Score: 68.77
Episode 3000	Average Score: 10.47
Episode 3100	Average Score: 10.64
Episode 3200	Average Score: 10.44
Episode 3300	Average Score: 10.81
Episode 3400	Average Score: 63.23
Episode 3500	Average Score: 10.24
Episode 3600	Average Score: 10.00
Episode 3700	Average Score: 9.97
Episode 3800	Average Score: 9.79
Episode 3900	Average Score: 9.80
Episode 4000	Average Score: 9.58
Episode 4100	Average Score: 9.61
Episode 4200	Average Score: 9.64
Episode 4300	Average Score: 9.76
Episode 4400	Average Score: 9.51
Episode 4500	Average Score: 9.72
Episode 4600	Average Score: 9.48
Episode 4700	Average Score: 9.50
Episode 4800	Average Score: 9.39
Episode 4900	Average Score: 9.32
Episode 5000	Average Score: 9.48
Episode 5100	Average Score: 9.43
Episode 5200	Average Score: 9.44
Episode 5300	Average Score: 9.49
Episode 5400	Average Score: 9.56
Episode 5500	Average Score: 9.31
Episode 5600	Average Score: 9.27
Episode 5700	Average Score: 9.47
Episode 5800	Average Score: 9.48
Episode 5900	Average Score: 9.62
Episode 6000	Average Score: 9.36
Episode 6100	Average Score: 9.37
Episode 6200	Average Score: 9.39
Episode 6300	Average Score: 9.35
Episode 6400	Average Score: 9.32
Episode 6500	Average Score: 9.26
Episode 6600	Average Score: 9.36
```

```
Episode 6700    Average Score: 9.48
Episode 6800    Average Score: 9.41
Episode 6900    Average Score: 9.35
Episode 7000    Average Score: 9.55
Episode 7100    Average Score: 9.28
Episode 7200    Average Score: 9.45
Episode 7300    Average Score: 9.33
Episode 7400    Average Score: 9.38
Episode 7500    Average Score: 9.41
Episode 7600    Average Score: 9.42
Episode 7700    Average Score: 9.46
Episode 7800    Average Score: 9.39
Episode 7900    Average Score: 9.41
Episode 8000    Average Score: 9.31
Episode 8100    Average Score: 9.48
Episode 8200    Average Score: 9.43
Episode 8300    Average Score: 9.36
Episode 8400    Average Score: 9.48
Episode 8500    Average Score: 9.27
Episode 8600    Average Score: 9.49
Episode 8700    Average Score: 9.30
Episode 8800    Average Score: 9.44
Episode 8900    Average Score: 9.42
Episode 9000    Average Score: 9.28
Episode 9100    Average Score: 9.41
Episode 9200    Average Score: 9.38
Episode 9300    Average Score: 9.43
Episode 9400    Average Score: 9.63
Episode 9500    Average Score: 9.47
Episode 9600    Average Score: 9.40
Episode 9700    Average Score: 9.49
Episode 9800    Average Score: 9.55
Episode 9900    Average Score: 9.36
Episode 10000   Average Score: 9.43
0:19:51.070185
```

1.4.2 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 (-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.

1.4.3 Task 1d (Optional)

Take a look at the official submissions page on OpenAI gym's CartPole v-0 evaluations

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.

1.4.4 Task 1a

The exploration stratagy used is epsilon greedy where epsilon decays over time, that is as the agent learn the exploration is decreased

1.4.5 Task 1b

[18]:
```
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))
```

1.4.6 Task 1c

[19]:
```
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
```

[39]:
```
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 = 0)
  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
Episode 400     Average Score: 30.41
Episode 500     Average Score: 11.56
Episode 600     Average Score: 42.04
Episode 700     Average Score: 12.31
Episode 800     Average Score: 9.72
Episode 900     Average Score: 11.54
Episode 1000    Average Score: 20.48
Episode 1100    Average Score: 30.80
```

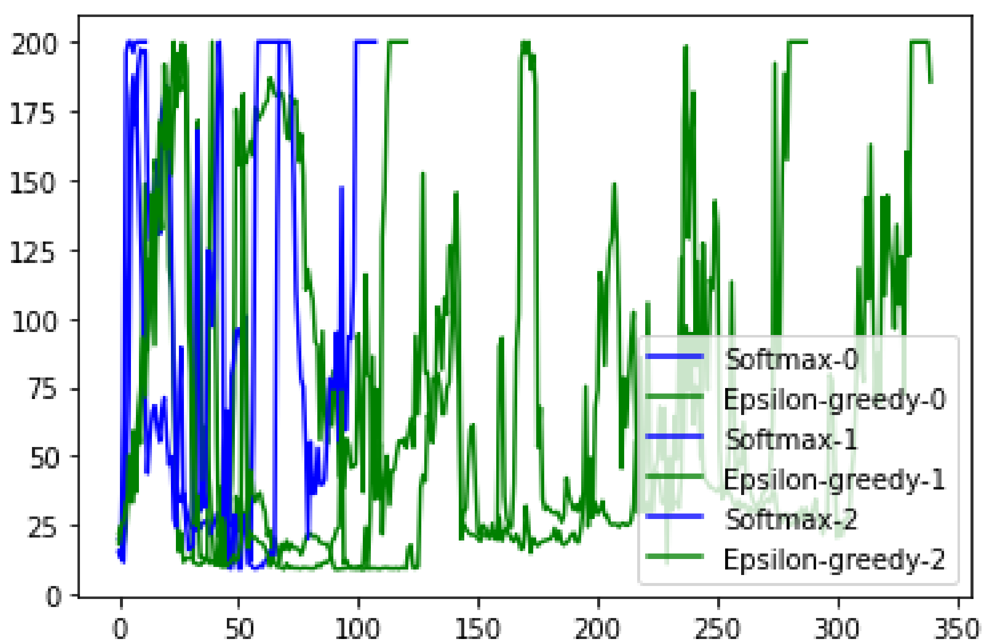
```
Episode 1200    Average Score: 45.32
Episode 1300    Average Score: 78.93
Episode 1400    Average Score: 92.41
Episode 1500    Average Score: 56.65
Episode 1600    Average Score: 22.77
Episode 1700    Average Score: 20.15
Episode 1800    Average Score: 20.45
Episode 1900    Average Score: 20.40
Episode 2000    Average Score: 36.09
Episode 2100    Average Score: 113.26
Episode 2200    Average Score: 66.47
Episode 2300    Average Score: 52.05
Episode 2400    Average Score: 72.38
Episode 2500    Average Score: 91.84
Episode 2600    Average Score: 62.10
Episode 2700    Average Score: 38.17
Episode 2800    Average Score: 31.79
Episode 2900    Average Score: 27.66
Episode 3000    Average Score: 39.92
Episode 3100    Average Score: 43.92
Episode 3200    Average Score: 104.81
Episode 3300    Average Score: 118.64
Episode 3400    Average Score: 190.82
Episode 3404    Average Score: 195.53
Environment solved in 3304 episodes!    Average Score: 195.53
Episode 100     Average Score: 111.40
Episode 200     Average Score: 66.37
Episode 300     Average Score: 44.66
Episode 400     Average Score: 78.82
Episode 500     Average Score: 83.53
Episode 600     Average Score: 13.65
Episode 700     Average Score: 80.87
Episode 800     Average Score: 116.01
Episode 900     Average Score: 51.30
Episode 1000    Average Score: 99.08
Episode 1084    Average Score: 196.46
Environment solved in 984 episodes!     Average Score: 196.46
```

[41]:
```
avg =[]
for i in range(3):
  print(f"Epsilon greedy DQN Expriement {i}: \t Environment solved in␣
 ↪{e_scores_l[i][1]} episodes\n")
  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]} episodes\n")
  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
```

[47]:
```
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

1.5 Part 2: One-Step Actor-Critic Algorithm

Actor-Critic methods learn both a policy $\pi(a|s;\theta)$ and a state-value function $v(s;w)$ simultaneously. The policy is referred to as the actor that suggests actions given a state. The estimated value function is referred to as the critic. It evaluates actions taken by the actor based on the given policy. In this exercise, both functions are approximated by feedforward neural networks.

- The policy network is parametrized by θ - it takes a state s as input and outputs the probabilities $\pi(a|s;\theta)\ \forall\ a$
- The value network is parametrized by w - it takes a state s as input and outputs a scalar value associated with the state, i.e., $v(s;w)$
- The single step TD error can be defined as follows:

$$\delta_t = R_{t+1} + \gamma v(s_{t+1};w) - v(s_t;w)$$

- The loss function to be minimized at every step ($L_{tot}^{(t)}$) is a summation of two terms, as follows:

$$L_{tot}^{(t)} = L_{actor}^{(t)} + L_{critic}^{(t)}$$

where,

$$L_{actor}^{(t)} = -\log \pi(a_t|s_t;\theta)\delta_t$$
$$L_{critic}^{(t)} = \delta_t^2$$

- **NOTE: Here, weights of the first two hidden layers are shared by the policy and the value network**
  - First two hidden layer sizes: [1024, 512]
  - Output size of policy network: 2 (Softmax activation)
  - Output size of value network: 1 (Linear activation)

1.5.1 Initializing Actor-Critic Network

[48]:
```
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_hidden2, activation='relu')
```

```python
        #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
```

1.5.2 Agent Class

1.5.3 Task 2a: Write code to compute δ_t inside the Agent.learn() function

[49]:
```python
class Agent:
    """
    Agent class
    """
    def __init__(self, action_size, lr=0.001, 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)


            #### 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))
```

1.5.4 Train the Network

[50]:
```
env = gym.make('CartPole-v0')

#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 = []
```

```python
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 > 195.0:
            print('Stopped at Episode ',ep-100)
            break

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

```
Episode  10 Reward 15.000000 Average Reward 26.700000
Episode  20 Reward 126.000000 Average Reward 76.300000
Episode  30 Reward 60.000000 Average Reward 72.800000
Episode  40 Reward 104.000000 Average Reward 90.300000
Episode  50 Reward 88.000000 Average Reward 106.500000
Episode  60 Reward 42.000000 Average Reward 63.100000
Episode  70 Reward 98.000000 Average Reward 77.600000
Episode  80 Reward 143.000000 Average Reward 103.800000
Episode  90 Reward 179.000000 Average Reward 162.300000
Episode  100 Reward 149.000000 Average Reward 160.900000
Episode  110 Reward 113.000000 Average Reward 164.800000
Episode  120 Reward 113.000000 Average Reward 119.200000
Episode  130 Reward 66.000000 Average Reward 119.700000
Episode  140 Reward 124.000000 Average Reward 147.200000
Episode  150 Reward 110.000000 Average Reward 115.000000
Episode  160 Reward 90.000000 Average Reward 121.300000
Episode  170 Reward 78.000000 Average Reward 150.700000
```

```
Episode  180 Reward 169.000000 Average Reward 138.600000
Episode  190 Reward 200.000000 Average Reward 187.200000
Episode  200 Reward 194.000000 Average Reward 178.100000
Episode  210 Reward 200.000000 Average Reward 190.000000
Episode  220 Reward 200.000000 Average Reward 196.100000
Episode  230 Reward 200.000000 Average Reward 200.000000
Episode  240 Reward 80.000000 Average Reward 150.500000
Episode  250 Reward 186.000000 Average Reward 105.600000
Episode  260 Reward 200.000000 Average Reward 184.300000
Episode  270 Reward 200.000000 Average Reward 200.000000
Episode  280 Reward 200.000000 Average Reward 200.000000
Episode  290 Reward 92.000000 Average Reward 124.800000
Episode  300 Reward 154.000000 Average Reward 87.200000
Episode  310 Reward 200.000000 Average Reward 182.600000
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 24.000000 Average Reward 142.800000
Episode  360 Reward 126.000000 Average Reward 145.600000
Episode  370 Reward 136.000000 Average Reward 151.700000
Episode  380 Reward 23.000000 Average Reward 77.600000
Episode  390 Reward 189.000000 Average Reward 133.300000
Episode  400 Reward 145.000000 Average Reward 183.100000
Episode  410 Reward 200.000000 Average Reward 173.700000
Episode  420 Reward 200.000000 Average Reward 188.900000
Episode  430 Reward 200.000000 Average Reward 200.000000
Episode  440 Reward 200.000000 Average Reward 197.600000
Episode  450 Reward 151.000000 Average Reward 189.100000
Episode  460 Reward 132.000000 Average Reward 148.500000
Episode  470 Reward 200.000000 Average Reward 175.000000
Episode  480 Reward 200.000000 Average Reward 199.400000
Episode  490 Reward 200.000000 Average Reward 199.400000
Episode  500 Reward 200.000000 Average Reward 199.800000
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 189.800000
Episode  540 Reward 200.000000 Average Reward 200.000000
Episode  550 Reward 44.000000 Average Reward 143.100000
Episode  560 Reward 33.000000 Average Reward 42.500000
Episode  570 Reward 31.000000 Average Reward 33.700000
Episode  580 Reward 31.000000 Average Reward 26.500000
Episode  590 Reward 22.000000 Average Reward 22.700000
Episode  600 Reward 20.000000 Average Reward 22.800000
Episode  610 Reward 30.000000 Average Reward 23.300000
Episode  620 Reward 28.000000 Average Reward 29.000000
Episode  630 Reward 30.000000 Average Reward 26.800000
Episode  640 Reward 200.000000 Average Reward 123.100000
Episode  650 Reward 135.000000 Average Reward 182.200000
```

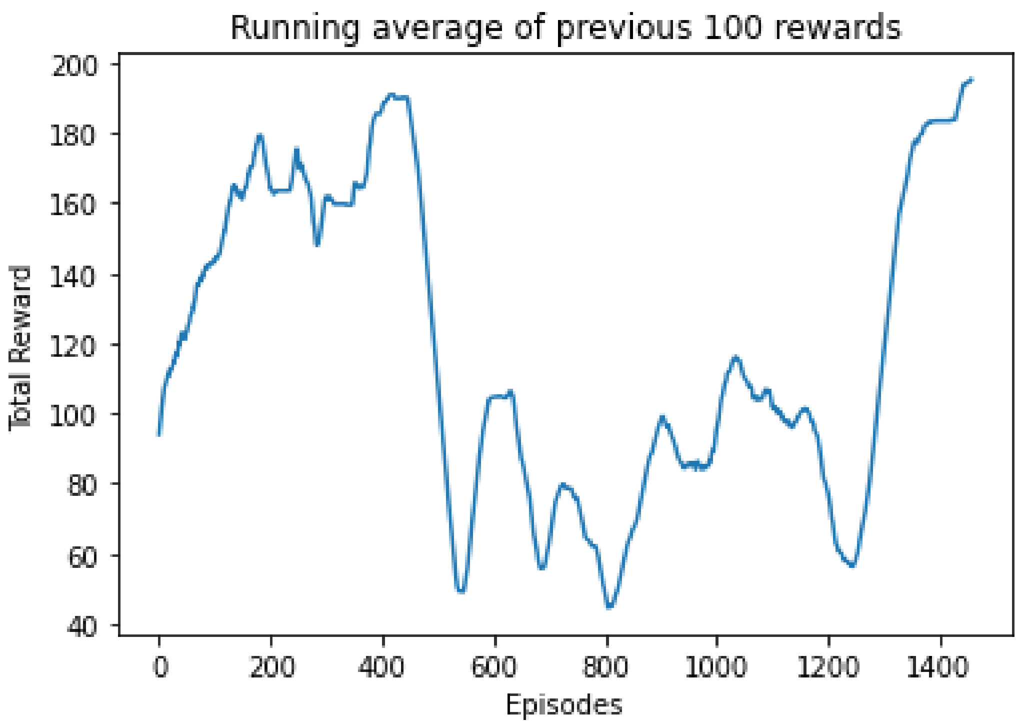
```
Episode  660 Reward 181.000000 Average Reward 174.000000
Episode  670 Reward 200.000000 Average Reward 188.100000
Episode  680 Reward 144.000000 Average Reward 149.300000
Episode  690 Reward 107.000000 Average Reward 107.200000
Episode  700 Reward 28.000000 Average Reward 43.800000
Episode  710 Reward 24.000000 Average Reward 25.000000
Episode  720 Reward 28.000000 Average Reward 24.300000
Episode  730 Reward 46.000000 Average Reward 46.800000
Episode  740 Reward 32.000000 Average Reward 37.000000
Episode  750 Reward 36.000000 Average Reward 72.700000
Episode  760 Reward 111.000000 Average Reward 104.200000
Episode  770 Reward 39.000000 Average Reward 74.100000
Episode  780 Reward 39.000000 Average Reward 48.900000
Episode  790 Reward 119.000000 Average Reward 84.200000
Episode  800 Reward 136.000000 Average Reward 116.400000
Episode  810 Reward 100.000000 Average Reward 128.900000
Episode  820 Reward 71.000000 Average Reward 75.400000
Episode  830 Reward 33.000000 Average Reward 45.000000
Episode  840 Reward 32.000000 Average Reward 35.900000
Episode  850 Reward 41.000000 Average Reward 44.000000
Episode  860 Reward 25.000000 Average Reward 33.800000
Episode  870 Reward 22.000000 Average Reward 26.400000
Episode  880 Reward 41.000000 Average Reward 31.700000
Episode  890 Reward 34.000000 Average Reward 37.200000
Episode  900 Reward 34.000000 Average Reward 28.500000
Episode  910 Reward 113.000000 Average Reward 93.200000
Episode  920 Reward 112.000000 Average Reward 110.000000
Episode  930 Reward 126.000000 Average Reward 115.000000
Episode  940 Reward 107.000000 Average Reward 104.800000
Episode  950 Reward 81.000000 Average Reward 90.100000
Episode  960 Reward 95.000000 Average Reward 78.800000
Episode  970 Reward 103.000000 Average Reward 110.900000
Episode  980 Reward 85.000000 Average Reward 105.200000
Episode  990 Reward 101.000000 Average Reward 88.800000
Episode  1000 Reward 97.000000 Average Reward 85.500000
Episode  1010 Reward 111.000000 Average Reward 77.300000
Episode  1020 Reward 70.000000 Average Reward 77.200000
Episode  1030 Reward 53.000000 Average Reward 61.200000
Episode  1040 Reward 75.000000 Average Reward 71.100000
Episode  1050 Reward 66.000000 Average Reward 94.900000
Episode  1060 Reward 70.000000 Average Reward 78.200000
Episode  1070 Reward 87.000000 Average Reward 114.500000
Episode  1080 Reward 75.000000 Average Reward 97.600000
Episode  1090 Reward 200.000000 Average Reward 117.500000
Episode  1100 Reward 200.000000 Average Reward 165.200000
Episode  1110 Reward 171.000000 Average Reward 171.600000
Episode  1120 Reward 75.000000 Average Reward 142.200000
Episode  1130 Reward 80.000000 Average Reward 98.400000
```

```
Episode  1140 Reward 63.000000 Average Reward 72.000000
Episode  1150 Reward 47.000000 Average Reward 47.200000
Episode  1160 Reward 85.000000 Average Reward 53.400000
Episode  1170 Reward 136.000000 Average Reward 83.000000
Episode  1180 Reward 86.000000 Average Reward 92.100000
Episode  1190 Reward 150.000000 Average Reward 139.200000
Episode  1200 Reward 129.000000 Average Reward 121.900000
Episode  1210 Reward 181.000000 Average Reward 156.500000
Episode  1220 Reward 87.000000 Average Reward 118.900000
Episode  1230 Reward 53.000000 Average Reward 86.200000
Episode  1240 Reward 62.000000 Average Reward 72.100000
Episode  1250 Reward 65.000000 Average Reward 77.000000
Episode  1260 Reward 61.000000 Average Reward 64.700000
Episode  1270 Reward 64.000000 Average Reward 49.500000
Episode  1280 Reward 40.000000 Average Reward 52.900000
Episode  1290 Reward 47.000000 Average Reward 43.100000
Episode  1300 Reward 54.000000 Average Reward 53.300000
Episode  1310 Reward 68.000000 Average Reward 53.600000
Episode  1320 Reward 72.000000 Average Reward 55.300000
Episode  1330 Reward 57.000000 Average Reward 63.300000
Episode  1340 Reward 61.000000 Average Reward 56.800000
Episode  1350 Reward 114.000000 Average Reward 91.600000
Episode  1360 Reward 126.000000 Average Reward 145.800000
Episode  1370 Reward 141.000000 Average Reward 137.500000
Episode  1380 Reward 169.000000 Average Reward 175.200000
Episode  1390 Reward 200.000000 Average Reward 196.200000
Episode  1400 Reward 200.000000 Average Reward 199.000000
Episode  1410 Reward 200.000000 Average Reward 200.000000
Episode  1420 Reward 200.000000 Average Reward 200.000000
Episode  1430 Reward 132.000000 Average Reward 182.200000
Episode  1440 Reward 152.000000 Average Reward 134.900000
Episode  1450 Reward 155.000000 Average Reward 172.600000
Episode  1460 Reward 116.000000 Average Reward 173.300000
Episode  1470 Reward 200.000000 Average Reward 171.100000
Episode  1480 Reward 200.000000 Average Reward 199.800000
Episode  1490 Reward 200.000000 Average Reward 200.000000
Episode  1500 Reward 200.000000 Average Reward 200.000000
Episode  1510 Reward 200.000000 Average Reward 200.000000
Episode  1520 Reward 200.000000 Average Reward 200.000000
Episode  1530 Reward 200.000000 Average Reward 200.000000
Episode  1540 Reward 200.000000 Average Reward 200.000000
Episode  1550 Reward 200.000000 Average Reward 200.000000
Stopped at Episode  1457
0:27:25.899065
```

1.5.5 Task 2b: Plot total reward curve

In the cell below, write code to plot the total reward averaged over 100 episodes (moving average)

[53]:
```python
### Plot of total reward vs episode
## Write Code Below
def running_mean(x, N):
    cumsum = np.cumsum(np.insert(x, 0, 0))
    return (cumsum[N:] - cumsum[:-N]) / float(N)

y = running_mean(reward_list ,100)
plt.plot(y)
plt.title("Running average of previous 100 rewards")
plt.xlabel("Episodes")
plt.ylabel("Total Reward")
plt.show()
```



1.5.6 Code for rendering (source)

[ ]:
```python
# Render an episode and save as a GIF file

display = Display(visible=0, size=(400, 300))
display.start()


def render_episode(env: gym.Env, model: tf.keras.Model, max_steps: int):
  screen = env.render(mode='rgb_array')
```

```
  im = Image.fromarray(screen)

  images = [im]

  state = tf.constant(env.reset(), dtype=tf.float32)
  for i in range(1, max_steps + 1):
    state = tf.expand_dims(state, 0)
    action_probs, _ = model(state)
    action = np.argmax(np.squeeze(action_probs))
    state, _, done, _ = env.step(action)
    state = tf.constant(state, dtype=tf.float32)

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

    if done:
      break

  return images


# Save GIF image
images = render_episode(env, agent.ac_model, 200)
image_file = 'cartpole-v0.gif'
# loop=0: loop forever, duration=1: play each frame for 1ms
images[0].save(
    image_file, save_all=True, append_images=images[1:], loop=0, duration=1)
```

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

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

2 For downlading the file (IGNORE)

[57]:
```
!sudo apt-get install texlive-xetex texlive-fonts-recommended␣
↪texlive-plain-generic
```

```
Reading package lists... Done
Building dependency tree
Reading state information... Done
texlive-fonts-recommended is already the newest version (2017.20180305-1).
texlive-plain-generic is already the newest version (2017.20180305-2).
texlive-xetex is already the newest version (2017.20180305-1).
The following package was automatically installed and is no longer required:
```

```
  libnvidia-common-470
Use 'sudo apt autoremove' to remove it.
0 upgraded, 0 newly installed, 0 to remove and 78 not upgraded.
```

[ ]:
```
!jupyter nbconvert --to pdf /content/drive/MyDrive/Documents/Sem6-drive/RL/
↪Tutorial/4Tut/Tutorial_4_DQN_and_AC.ipynb
```

[56]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

[ ]:
```
```