CS6700: Reinforcement Learning

Tutorial 5 - DQN and Actor-Critic

Gautham Govind A, EE19B022

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

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

In [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
!pip install gym[classic_control]
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/publi
c/simple/
Requirement already satisfied: setuptools in /usr/local/lib/python3.8/dist-packages (67.4
.0)
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/publi
c/simple/
Requirement already satisfied: gym[classic_control] in /usr/local/lib/python3.8/dist-pack
ages (0.25.2)
Requirement already satisfied: importlib-metadata>=4.8.0 in /usr/local/lib/python3.8/dist
-packages (from gym[classic_control]) (6.0.0)
Requirement already satisfied: numpy>=1.18.0 in /usr/local/lib/python3.8/dist-packages (f
rom gym[classic_control]) (1.22.4)
Requirement already satisfied: cloudpickle>=1.2.0 in /usr/local/lib/python3.8/dist-packag
es (from gym[classic_control]) (2.2.1)
Requirement already satisfied: gym-notices>=0.0.4 in /usr/local/lib/python3.8/dist-packag
es (from gym[classic_control]) (0.0.8)
Requirement already satisfied: pygame==2.1.0 in /usr/local/lib/python3.8/dist-packages (f
rom gym[classic_control]) (2.1.0)
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.8/dist-packages (from
importlib-metadata>=4.8.0->gym[classic_control]) (3.15.0)
```

In [2]:

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

import numpy as np
import random
import torch
```

```python
import torch.nn as nn
import torch.nn.functional as F
from collections import namedtuple, deque
import torch.optim as optim
import datetime
import gym
from gym.wrappers.record_video import RecordVideo
import glob
import io
import base64
import matplotlib.pyplot as plt
from IPython.display import HTML
from pyvirtualdisplay import Display
import tensorflow as tf
from IPython import display as ipythondisplay
from PIL import Image
import tensorflow_probability as tfp
from scipy.special import softmax
```

```
/usr/local/lib/python3.8/dist-packages/tensorflow_probability/python/__init__.py:57: Depr
ecationWarning: distutils Version classes are deprecated. Use packaging.version instead.
  if (distutils.version.LooseVersion(tf.__version__) <
```

In [3]:

```python
'''
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-v1'
'MountainCar-v0'
'''

env = gym.make('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 updat
es 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 acti
on taken  '''

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

```
4
2
1
----
[ 0.01369617 -0.02302133 -0.04590265 -0.04834723]
----
0
----
[ 0.01323574 -0.21745604 -0.04686959  0.22950698]
1.0
False
{}
----
```

```
/usr/local/lib/python3.8/dist-packages/gym/core.py:317: DeprecationWarning: WARN: Initial
izing wrapper in old step API which returns one bool instead of two. It is recommended to
set `new_step_api=True` to use new step API. This will be the default behaviour in future
.
  deprecation(
/usr/local/lib/python3.8/dist-packages/gym/wrappers/step_api_compatibility.py:39: Depreca
tionWarning: WARN: Initializing environment in old step API which returns one bool instea
d of two. It is recommended to set `new_step_api=True` to use new step API. This will be
the default behaviour in future.
  deprecation(
/usr/local/lib/python3.8/dist-packages/gym/core.py:256: DeprecationWarning: WARN: Functio
n `env.seed(seed)` is marked as deprecated and will be removed in the future. Please use
`env.reset(seed=seed)` instead.
  deprecation(
```

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

In [4]:

```
'''
### 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)
```

Replay Buffer:

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

In [5]:

```
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
rd", "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 No
ne])).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)
```

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:

In [6]:

```
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)           -Need
ed 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()
```

Here, we present the DQN algorithm code.

In [7]:

```
''' 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 '''

    rewards = []

    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)
        rewards.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
dow)), 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, rewards]

''' 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)
```

Out[7]:

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

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-v1 defines "solving" as getting average reward of 195.0 over 100 consecutive trials)

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.

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-1 environment and add a new text cell below it with your inferences.

Task 1a:

From the code structure, it is evident that the exploration strategy used is defined in TutorialAgent() class. On examining the class, we see that ϵ-greedy exploration strategy is already defined in act() method of the class.

Task 1b:

The strategy that is not used is softmax exploration strategy, which is implemented in the class below. Note that the hyperparameter now is β and not ϵ.

In [8]:

```
class SoftMaxAgent():

    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)           -Need
ed 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, beta=1.):
```

```
        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 (Added newly) '''
        #print(action_values.cpu().data.numpy().shape)
        soft_prob = softmax(action_values.cpu().data.numpy().flatten()/beta)
        soft_prob /= np.sum(soft_prob)
        return np.random.choice(np.arange(self.action_size), p = soft_prob)


    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()
```

Task 1c:

We are required to try out both exploration strategies and analyze their performance. It is worthwhile to point out that an ϵ-decay strategy is being adopted in the training process. We adopt a similar β-decay strategy for softmax.

ϵ-greedy:

In [9]:

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

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

def epsilon_dqn(n_episodes=10000, max_t=1000, eps_start=1.0, eps_end=0.01, eps_decay=0.99
5):

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

    rewards = []

    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

        rewards.append(score)
        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}\tEpsilon:{}'.format(i_episode, np.mea
n(scores_window), eps), 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,rewards]

''' 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)

print("Running epsilon-greedy based exploration:")
print("---------------------------------------------------------------------------------------
--------------")

epsilon_avg_scores, epsilon_episodes, epsilon_scores = epsilon_dqn()


time_taken = datetime.datetime.now() - begin_time

print(time_taken)
```

```
Running epsilon-greedy based exploration:
---------------------------------------------------------------------------------------
-------
Episode 100 Average Score: 38.24
Episode 200 Average Score: 144.32
Episode 231 Average Score: 195.80 Epsilon:0.3141460853680822
Environment solved in 131 episodes! Average Score: 195.80
0:01:49.338487
```

SoftMax

In [10]:

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

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

```
action_shape = env.action_space.n

def soft_dqn(n_episodes=10000, max_t=1000, beta_start=10.0, beta_end=0.05, beta_decay=0.9
):

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

    rewards = []

    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 '''

    beta = beta_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, beta)
            next_state, reward, done, _ = env.step(action)
            agent.step(state, action, reward, next_state, done)
            state = next_state
            score += reward
            if done:
                break

        rewards.append(score)
        scores_window.append(score)
        scores_window_printing.append(score)
        ''' save most recent score '''

        beta = max(beta_end, beta_decay*beta)
        ''' decrease epsilon '''

        print('\rEpisode {}\tAverage Score: {:.2f}\t Beta: {}'.format(i_episode, np.mean
(scores_window), beta), 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, rewards]

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

begin_time = datetime.datetime.now()
agent = SoftMaxAgent(state_size=state_shape,action_size = action_shape,seed = 0)

print("Running softmax based exploration:")
print("-----------------------------------------------------------------------------
--------------")

softmax_avg_scores, softmax_episodes, softmax_scores = soft_dqn()


time_taken = datetime.datetime.now() - begin_time

print(time_taken)
```

```
Running softmax based exploration:
-----------------------------------------------------------------------------
-------
Episode 100 Average Score: 163.87
```

```
Episode 107 Average Score: 197.05  Beta: 0.05
Environment solved in 7 episodes! Average Score: 197.05
0:01:14.769149
```

In [11]:

```
print("No: of episodes for epsilon-greedy:", epsilon_episodes)
print("No: of episodes for softmax:", softmax_episodes)
```

```
No: of episodes for epsilon-greedy: 131
No: of episodes for softmax: 7
```

In [24]:

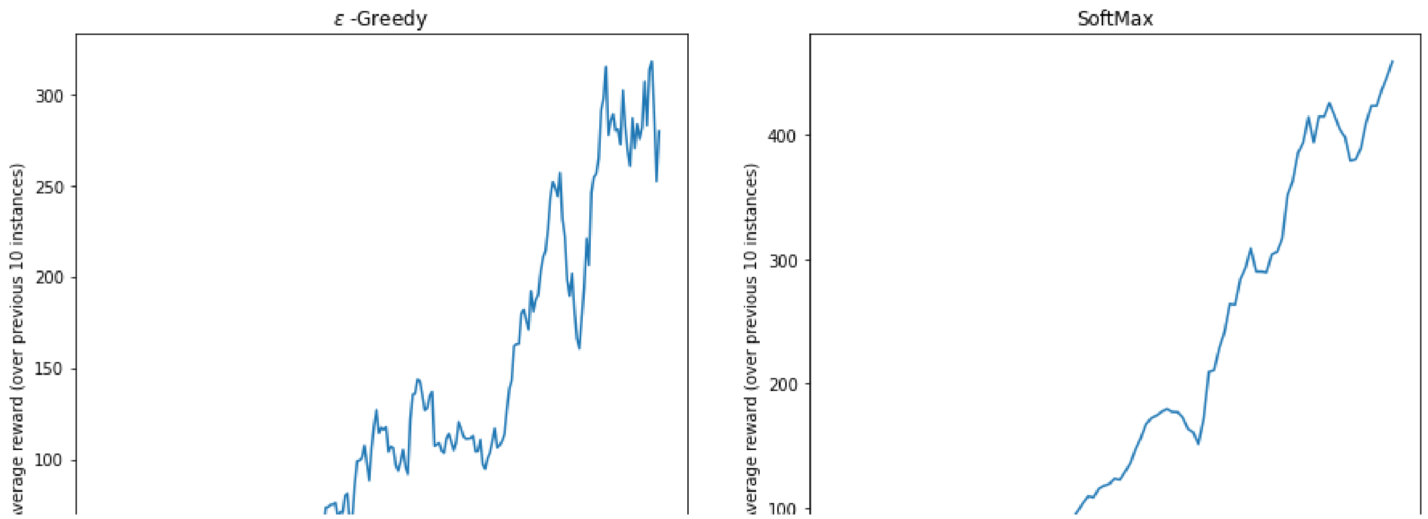
```
```

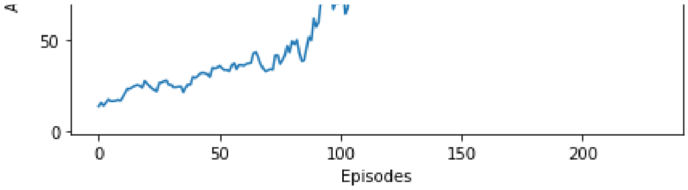
In [24]:

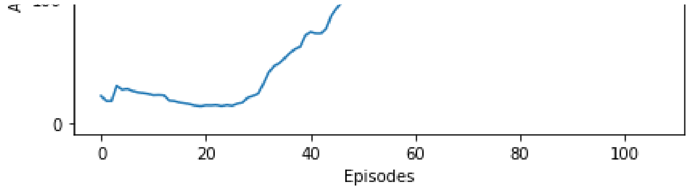
```
```

In [18]:

```
running_avg_epsilon = []

for i in range(len(epsilon_scores)):
  if i <= 9:
    running_avg_epsilon.append(np.mean(epsilon_scores[:i+1]))
  else:
    running_avg_epsilon.append(np.mean(epsilon_scores[i-9:i+1]))

running_avg_soft = []

for i in range(len(softmax_scores)):
  if i <= 9:
    running_avg_soft.append(np.mean(softmax_scores[:i+1]))
  else:
    running_avg_soft.append(np.mean(softmax_scores[i-9:i+1]))




fig, ax = plt.subplots(1, 2, figsize = (15, 7))
ax[0].plot(list(range(len(running_avg_epsilon))), running_avg_epsilon)
ax[0].set_xlabel("Episodes")
ax[0].set_ylabel("Average reward (over previous 10 instances)")
ax[0].set_title(r"$\epsilon$ -Greedy")


ax[1].plot(list(range(len(running_avg_soft))), running_avg_soft)
ax[1].set_xlabel("Episodes")
ax[1].set_ylabel("Average reward (over previous 10 instances)")
ax[1].set_title(r"SoftMax")

plt.show()
```







From the above analysis we make the following observations:

- **In this particular case, softmax converges in lesser number of episodes (7 v/s 131). However, it must be noted that this does not mean softmax is better in geenral. This is because the rate of convergence may vary because of the inherent stochasticity. Furthermore, the rate heavily depends on the decay hyperparameter for both methods. Hence, it might be possible that epsilon-greedy performs better for a different set of hyperparameters.**
- **Despite the number of episodes being considerably less, softmax still takes 1 min 14 seconds to run, whereas epsilon-greedy takes 1 min 49 seconds. This is to be expected because softmax is computationally much more expensive that epsilon-greedy and despite running for much lower number of episodes, still takes significant time.**
- **Finally, we also note from the avereage score plots that softmax has a relatively smoother curve. Again, this can be expected because softmax samples from a weiighted distribution whereas epsilon-greedy is more random sampling, hence causing a lot of jitters.**

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)

Initializing Actor-Critic Network

In [13]:

```
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')

        #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
```

Agent Class

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

In [14]:

```
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 ac
tion
        """
        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    # Single step TD error
            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_var
iables))
```

Train the Network

In [21]:

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

#Initializing Agent
agent = Agent(lr=0.5e-4, action_size=env.action_space.n)
#Number of episodes
episodes = 1800
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 > 195.0:
            print('Stopped at Episode ',ep-100)
            break

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

```
/usr/local/lib/python3.8/dist-packages/gym/core.py:317: DeprecationWarning: WARN: Initial
izing wrapper in old step API which returns one bool instead of two. It is recommended to
set `new_step_api=True` to use new step API. This will be the default behaviour in future
.
  deprecation(
```

```
  deprecation(
/usr/local/lib/python3.8/dist-packages/gym/wrappers/step_api_compatibility.py:39: DeprecationWarning: WARN: Initializing environment in old step API which returns one bool instead of two. It is recommended to set `new_step_api=True` to use new step API. This will be the default behaviour in future.
  deprecation(
```

```
Episode  10 Reward 15.000000 Average Reward 25.800000
Episode  20 Reward 46.000000 Average Reward 28.600000
Episode  30 Reward 46.000000 Average Reward 28.200000
Episode  40 Reward 35.000000 Average Reward 43.800000
Episode  50 Reward 46.000000 Average Reward 65.600000
Episode  60 Reward 63.000000 Average Reward 62.100000
Episode  70 Reward 60.000000 Average Reward 88.300000
Episode  80 Reward 60.000000 Average Reward 76.100000
Episode  90 Reward 50.000000 Average Reward 71.100000
Episode  100 Reward 174.000000 Average Reward 89.500000
Episode  110 Reward 74.000000 Average Reward 109.200000
Episode  120 Reward 87.000000 Average Reward 99.900000
Episode  130 Reward 63.000000 Average Reward 79.200000
Episode  140 Reward 39.000000 Average Reward 62.500000
Episode  150 Reward 64.000000 Average Reward 60.100000
Episode  160 Reward 38.000000 Average Reward 51.600000
Episode  170 Reward 61.000000 Average Reward 72.800000
Episode  180 Reward 47.000000 Average Reward 91.800000
Episode  190 Reward 118.000000 Average Reward 115.800000
Episode  200 Reward 143.000000 Average Reward 128.000000
Episode  210 Reward 126.000000 Average Reward 143.200000
Episode  220 Reward 161.000000 Average Reward 135.100000
Episode  230 Reward 122.000000 Average Reward 155.400000
Episode  240 Reward 186.000000 Average Reward 157.400000
Episode  250 Reward 112.000000 Average Reward 130.300000
Episode  260 Reward 97.000000 Average Reward 68.600000
Episode  270 Reward 43.000000 Average Reward 57.800000
Episode  280 Reward 62.000000 Average Reward 58.200000
Episode  290 Reward 114.000000 Average Reward 57.100000
Episode  300 Reward 86.000000 Average Reward 54.300000
Episode  310 Reward 39.000000 Average Reward 49.100000
Episode  320 Reward 82.000000 Average Reward 54.400000
Episode  330 Reward 51.000000 Average Reward 51.700000
Episode  340 Reward 52.000000 Average Reward 58.900000
Episode  350 Reward 56.000000 Average Reward 64.500000
Episode  360 Reward 53.000000 Average Reward 62.800000
Episode  370 Reward 49.000000 Average Reward 53.500000
Episode  380 Reward 195.000000 Average Reward 75.400000
Episode  390 Reward 76.000000 Average Reward 67.100000
Episode  400 Reward 377.000000 Average Reward 179.200000
Episode  410 Reward 212.000000 Average Reward 208.700000
Episode  420 Reward 176.000000 Average Reward 238.500000
Episode  430 Reward 144.000000 Average Reward 242.600000
Episode  440 Reward 500.000000 Average Reward 330.000000
Episode  450 Reward 253.000000 Average Reward 394.400000
Stopped at Episode  358
0:08:23.162835
```

Task 2b: Plot total reward curve

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

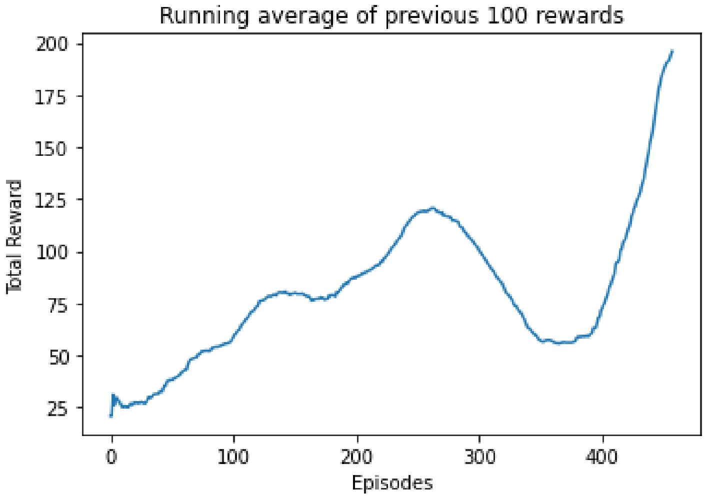
In [22]:

```
running_avg = []

for i in range(len(reward_list)):
  if i <= 99:
    running_avg.append(np.mean(reward_list[:i+1]))
  else:
    running_avg.append(np.mean(reward_list[i-99:i+1]))

plt.plot(np.arange(len(running_avg)), running_avg)
plt.xlabel('Episodes')
```

```
plt.ylabel('Total Reward')
plt.title('Running average of previous 100 rewards')
```

Out[22]:

```
Text(0.5, 1.0, 'Running average of previous 100 rewards')
```



Code for rendering (source)

In [23]:

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

```
/usr/local/lib/python3.8/dist-packages/gym/core.py:43: DeprecationWarning: WARN: The argu
ment mode in render method is deprecated; use render_mode during environment initializati
on instead.
See here for more information: https://www.gymlibrary.ml/content/api/
  deprecation(
```

In [24]:

```
import tensorflow_docs.vis.embed as embed
embed.embed_file(image_file)
```

Out[24]:

In [24]:

```

```