#Tutorial 5 - DQN

Please follow this tutorial to understand the structure (code) of DQN algorithm.

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 of the core concepts. Contact the TAs for further resources if needed.

```
'''
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]
```

```
Requirement already satisfied: setuptools in
/usr/local/lib/python3.10/dist-packages (69.1.1)
Requirement already satisfied: gym[classic_control] in
/usr/local/lib/python3.10/dist-packages (0.25.2)
Requirement already satisfied: numpy>=1.18.0 in
/usr/local/lib/python3.10/dist-packages (from gym[classic_control])
(1.25.2)
Requirement already satisfied: cloudpickle>=1.2.0 in
/usr/local/lib/python3.10/dist-packages (from gym[classic_control])
(2.2.1)
Requirement already satisfied: gym-notices>=0.0.4 in
/usr/local/lib/python3.10/dist-packages (from gym[classic_control])
(0.0.8)
Requirement already satisfied: pygame==2.1.0 in
/usr/local/lib/python3.10/dist-packages (from gym[classic_control])
(2.1.0)
```

```
'''
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
```

```python
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
```

```
/usr/local/lib/python3.10/dist-packages/tensorflow_probability/
python/__init__.py:57: DeprecationWarning: distutils Version classes
are deprecated. Use packaging.version instead.
  if (distutils.version.LooseVersion(tf.__version__) <
```

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

```
/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283:
DeprecationWarning: `should_run_async` will not call `transform_cell`
automatically in the future. Please pass the result to
`transformed_cell` argument and any exception that happen during
```

```
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
/usr/local/lib/python3.10/dist-packages/gym/core.py:317:
DeprecationWarning: WARN: Initializing wrapper in old step API which
returns one bool instead of two. It is recommended to set
`new_step_api=True` to use new step API. This will be the default
behaviour in future.
  deprecation(
/usr/local/lib/python3.10/dist-packages/gym/wrappers/step_api_compatib
ility.py:39: DeprecationWarning: WARN: Initializing environment in old
step API which returns one bool instead of two. It is recommended to
set `new_step_api=True` to use new step API. This will be the default
behaviour in future.
  deprecation(
/usr/local/lib/python3.10/dist-packages/gym/core.py:256:
DeprecationWarning: WARN: Function `env.seed(seed)` is marked as
deprecated and will be removed in the future. Please use
`env.reset(seed=seed)` instead.
  deprecation(
/usr/local/lib/python3.10/dist-packages/gym/utils/passive_env_checker.
py:241: DeprecationWarning: `np.bool8` is a deprecated alias for
`np.bool_`.  (Deprecated NumPy 1.24)
  if not isinstance(terminated, (bool, np.bool8)):
```

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 - 128 nodes \
Hidden Layer 2 - 64 nodes \
Output Layer - 2 nodes (Action Space) \
Optimizer - zero_grad()
'''

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

```
'''
Bunch of Hyper parameters (Which you might have to tune later)
'''
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:

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", "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)
```

Tutorial Agent Code:

```
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()
```

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=1000, eps_start=1.0, eps_end=0.01, 
eps_decay=0.995):

    scores_window = deque(maxlen=100)
    ''' last 100 scores for checking if the avg is more than 195 '''

    eps = eps_start
    ''' initialize epsilon '''

    eps_reward = []
    ''' episode reward'''

    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)
        eps_reward.append(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 % 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, np.mean(scores_window)))
           break
    return True, eps_reward

''' Trial run to check if algorithm runs and saves the data '''
''' Analysis of this agent done with the solutions'''

'''
begin_time = datetime.datetime.now()

agent = TutorialAgent(state_size=state_shape,action_size =
action_shape,seed = 0)
_, episode_rewards = dqn()

time_taken = datetime.datetime.now() - begin_time

print(time_taken)
'''

{"type":"string"}
```

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 1 Solution

We observe from the flow of the code, ϵ **greedy policy** has been implemented. Once the Q values are updated using the target Q network and local Q network the action taken by the agent follows this policy (ϵ greedy). The ϵ greedy policy allows the agent to explore by generating a random number and comparing it with the ϵ, when greater the agent performs a greedy action and when lesser it takes a random action distributed uniformly.

We also observe that the ϵ decays as we train the network. We decay the value of ϵ to reduce exploration as the network gets trained.

Task 2 Solution

The **softmax policy is not used**. The following code cell implements softmax policy incorporated in the 'Tutorial Agent Code'.

```python
from scipy.special import softmax

class TutorialAgent_Softmax():
    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()

        ''' Softmax action selection (Already Present) '''
        probs = softmax(action_values.cpu().data.numpy()[0])
        return np.random.choice(np.arange(self.action_size), p =
probs)

    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 3 Solution

Here is the code for running both the agents and comparing their results based on number of episodes to 'solve' the environment, reward plots and computation time.

DQN with ϵ greedy policy

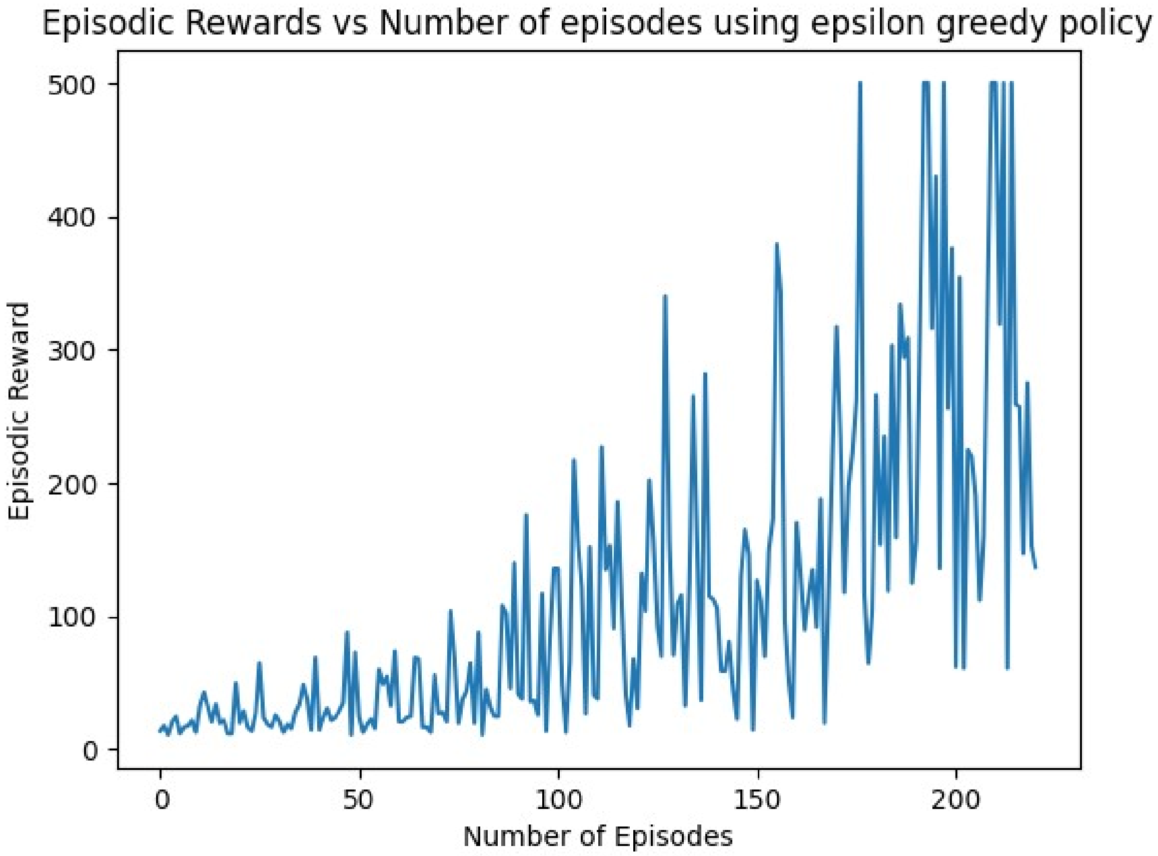
```
begin_time = datetime.datetime.now()

agent = TutorialAgent(state_size=state_shape,action_size =
action_shape,seed = 0)
_, episode_rewards_epsilon = dqn()

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

```
print(time_taken)
```

```
Episode 100	Average Score: 38.24
Episode 200	Average Score: 163.10
Episode 221	Average Score: 195.58
Environment solved in 221 episodes!	Average Score: 195.58
0:01:29.952288
```

```
import matplotlib.pyplot as plt

plt.plot(episode_rewards_epsilon)
plt.xlabel('Number of Episodes')
plt.ylabel('Episodic Reward')
plt.title('Episodic Rewards vs Number of episodes using epsilon greedy policy')
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: DeprecationWarning: `should_run_async` will not call `transform_cell` automatically in the future. Please pass the result to `transformed_cell` argument and any exception that happen during thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

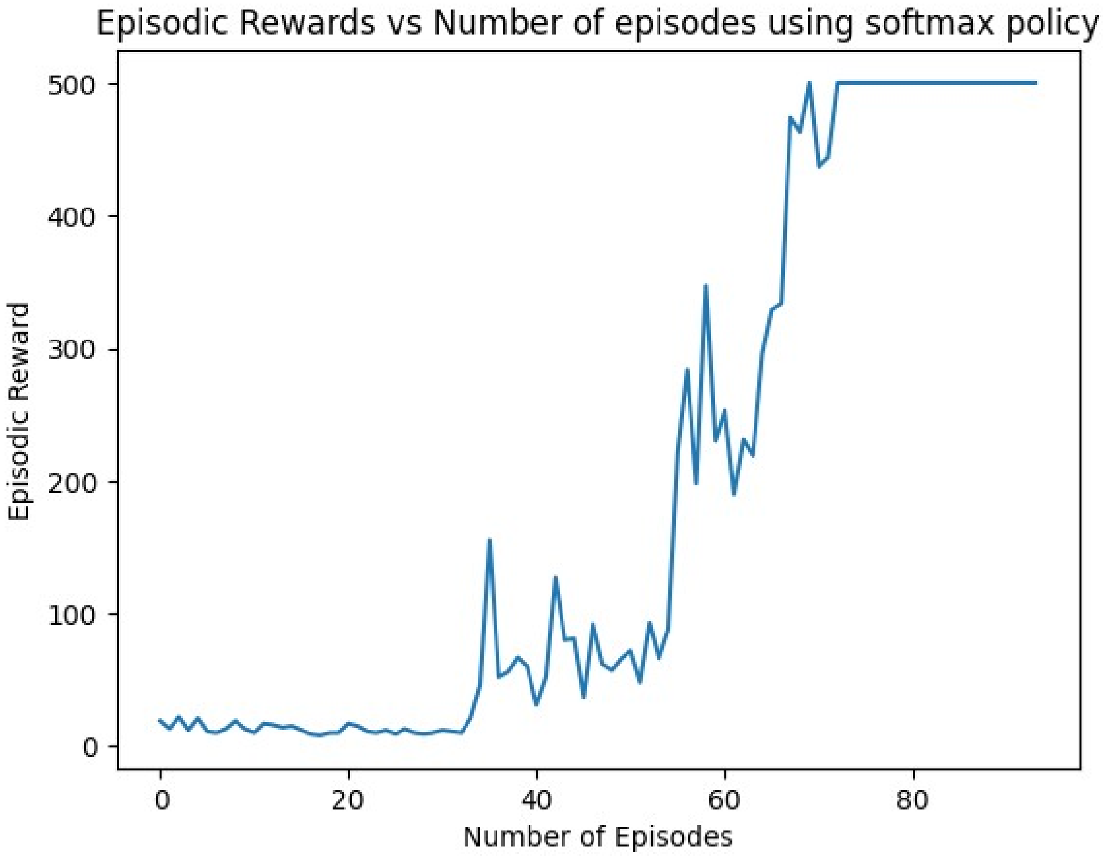


DQN with softmax policy

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

agent = TutorialAgent_Softmax(state_size=state_shape,action_size =
action_shape,seed = 0)
_, episode_rewards_softmax = dqn()

time_taken = datetime.datetime.now() - begin_time

print(time_taken)
```

```
Episode 94 Average Score: 195.56
Environment solved in 94 episodes!    Average Score: 195.56
0:01:00.943490
```

```
import matplotlib.pyplot as plt

plt.plot(episode_rewards_softmax)
plt.xlabel('Number of Episodes')
plt.ylabel('Episodic Reward')
plt.title('Episodic Rewards vs Number of episodes using softmax
policy')
plt.show()
```



Inferences

Comparing ϵ greedy policy and softmax policy, we observe that softmax policy performs better.

We can conclude this by observing the following:

- The number of episodes taken to converge to 'solve' the CartPole-v1 environment in the case of ϵ greedy is 221 episodes, whereas the softmax policy takes only 94 episodes. We clearly observe that softmax policy learns faster than ϵ greedy policy.
- From the reward plots, we can observe that there is clear upward trend in the case of softmax policy, whereas in ϵ greedy policy we observe a zig-zag nature of episodic rewards.
- We also observe that the computation time taken for the agent exploring using softmax policy (0:01:00.943490) is lesser than the time taken for the agent explogin using ϵ greedy policy (0:01:29.952288).

Therefore, softmax policy performs better than ϵ greedy policy in this case.