#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 (https://www.nature.com/articles/nature14236)](https://www.nature.com/articles/nature14236) 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.

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

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

```
Requirement already satisfied: setuptools in /usr/local/lib/python3.11/dist-packages (76.0.0)
Requirement already satisfied: gym[classic_control] in /usr/local/lib/python3.11/dist-packages (0.26.2)
Requirement already satisfied: numpy>=1.18.0 in /usr/local/lib/python3.11/dist-packages (from gym[classic_control])
(1.26.4)
Requirement already satisfied: cloudpickle>=1.2.0 in /usr/local/lib/python3.11/dist-packages (from gym[classic_contr
ol]) (3.1.1)
Requirement already satisfied: gym_notices>=0.0.4 in /usr/local/lib/python3.11/dist-packages (from gym[classic_contr
ol]) (0.0.8)
Collecting pygame==2.1.0 (from gym[classic_control])
  Using cached pygame-2.1.0.tar.gz (5.8 MB)
  error: subprocess-exited-with-error

  × python setup.py egg_info did not run successfully.
  │ exit code: 1
  ╰─> See above for output.

  note: This error originates from a subprocess, and is likely not a problem with pip.
  Preparing metadata (setup.py) ... error
error: metadata-generation-failed

× Encountered error while generating package metadata.
╰─> See above for output.

note: This is an issue with the package mentioned above, not pip.
hint: See above for details.
```

In [3]:
```
'''
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 gymnasium as 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
```

In [4]:

```
'''
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')

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 var
- 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
----
(array([-0.0410322 , -0.04988823,  0.01525484,  0.01642654], dtype=float32), {})
----
0
----
[-0.04202996 -0.2452256   0.01558337  0.31388324]
1.0
False
{}
----
```

## DQN

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

### Q-Network:

The neural network used as a function approximator is defined below

In [5]:
```python
'''
### 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.

In [6]:
```
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().t
        dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is not None]).astype(np.uint8)).floa

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

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

## Tutorial Agent Code:

In [7]:
```
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.

In [7]:

```python
''' 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 '''

    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 = 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
           break
    return True

''' 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: 36.88
Episode 200     Average Score: 160.92
Episode 300     Average Score: 60.54
Episode 400     Average Score: 11.77
Episode 500     Average Score: 40.99
Episode 600     Average Score: 13.68
Episode 700     Average Score: 117.94
Episode 800     Average Score: 25.25
Episode 900     Average Score: 147.36
Episode 1000    Average Score: 167.90
Episode 1100    Average Score: 164.52
Episode 1200    Average Score: 162.53
Episode 1300    Average Score: 174.43
Episode 1354    Average Score: 197.40
Environment solved in 1354 episodes!    Average Score: 197.40
0:07:48.237709
```

### 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 ($\epsilon$-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 $\epsilon$-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

The Exploration strategy used in the given TutorialAgent class is Epsilon greedy.

### Task 2 Solution

In [8]:

```
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()

        ''' Implementing Softmax Exploration Strategy'''
        actions = action_values.cpu().data.numpy().flatten()

        Qmax = np.max(actions) # For Numerical Stability
        # To prevent overflow of values(e^(large value)) which would result in 'NaN' when computed probabilities
        actions = actions - Qmax
        expQ = np.exp(actions/ 0.3)
        probQ = expQ / np.sum(expQ) # Softmax Probabilities

        action = np.random.choice(np.arange(self.action_size), p=probQ)

        return action # Action selected based on computed softmax probabilities

    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()
```

In [9]:
```python
''' 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 '''

    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 = 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
           break
    return True

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

begin_time = datetime.datetime.now()

print("Tutorial Agent with SoftMax Policy")
agent = TutorialAgent_SoftMax(state_size=state_shape,action_size = action_shape,seed = 0)
dqn()

time_taken = datetime.datetime.now() - begin_time

print(time_taken)
```

```
Tutorial Agent with SoftMax Policy
Episode 64	Average Score: 204.14
Environment solved in 64 episodes!	Average Score: 204.14
0:00:58.190546
```

## Task 3 Solution

## Modifying the DQN to return the rewards and episodes value

In [15]:

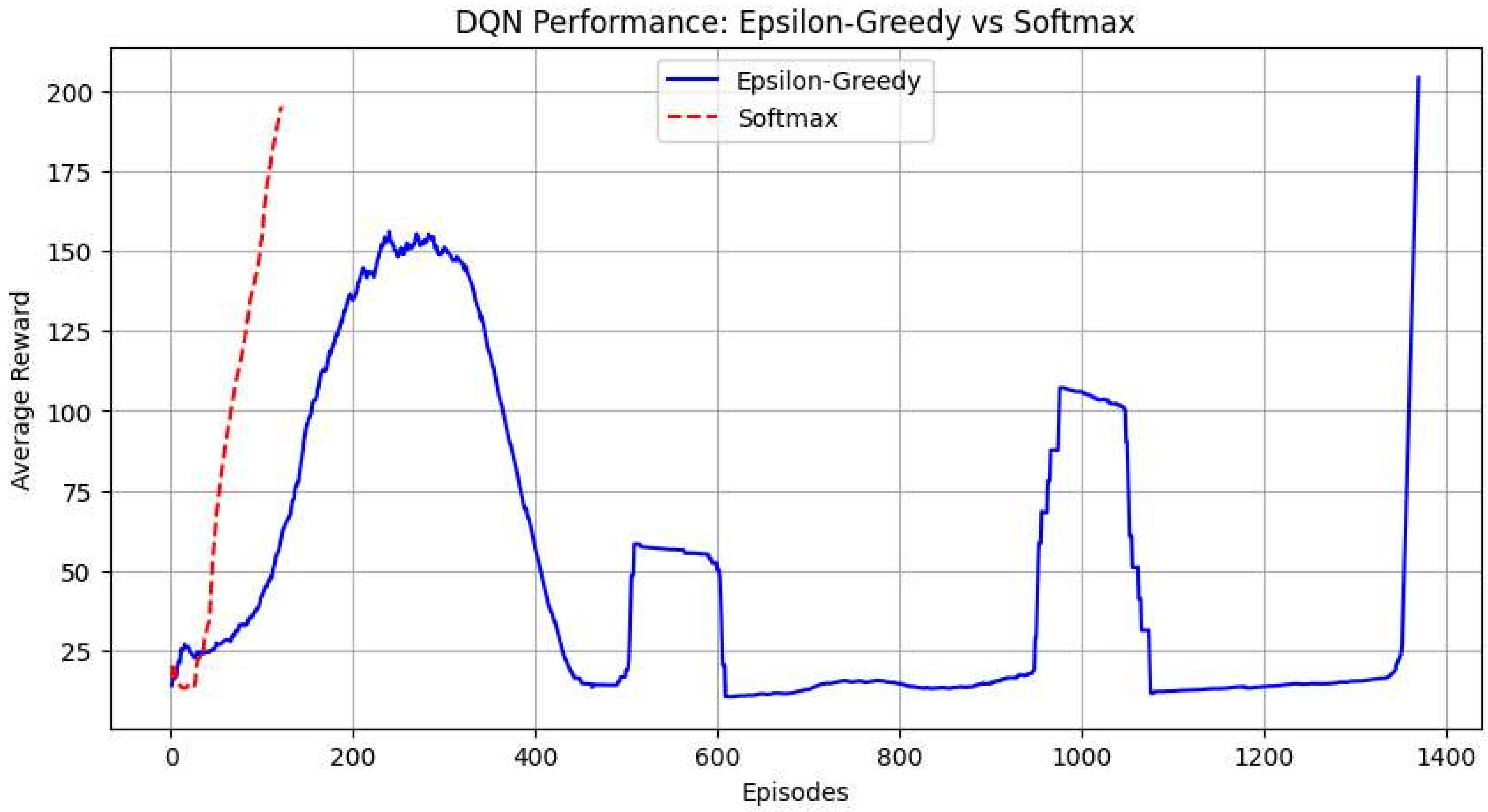
```
''' Defining DQN Algorithm '''

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 '''

    reward_value = []
    episode_num = []

    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)
        reward_value.append(np.mean(scores_window)) # Stores the Avg reward value
        episode_num.append(i_episode) # To Store the episode value

        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
           break
    return True, reward_value, episode_num

```

In [20]:
```
print("Tutorial Agent with Epsilon Greedy Policy")
begin_time = datetime.datetime.now()

agent = TutorialAgent(state_size=state_shape,action_size = action_shape,seed = 0)
_, rewards_eps, episodes_eps = dqn()
#print("Rewards: ", rewards_eps)
#print("Episodes: ", episodes_eps)
time_taken_eps = datetime.datetime.now() - begin_time

print("Time taken by Agent with Epsilon Greedy Policy: ", time_taken_eps)

print("Tutorial Agent with SoftMax Policy")
begin_time = datetime.datetime.now()

agent = TutorialAgent_SoftMax(state_size=state_shape,action_size = action_shape,seed = 0)
_, rewards_softmax, episodes_softmax = dqn()
#print("Rewards: ", rewards_softmax)
#print("Episodes: ", episodes_softmax)
time_taken_softmax = datetime.datetime.now() - begin_time

print("Time taken by Agent with Softmax Policy: ", time_taken_softmax)
```

```
Tutorial Agent with Epsilon Greedy Policy
Episode 100	Average Score: 38.98
Episode 200	Average Score: 128.71
Episode 300	Average Score: 52.69
Episode 400	Average Score: 21.55
Episode 500	Average Score: 56.98
Episode 600	Average Score: 13.62
Episode 700	Average Score: 37.22
Episode 800	Average Score: 25.40
Episode 900	Average Score: 61.43
Episode 1000	Average Score: 87.57
Episode 1100	Average Score: 155.28
Episode 1200	Average Score: 158.14
Episode 1300	Average Score: 179.65
Episode 1400	Average Score: 176.57
Episode 1423	Average Score: 201.02
Environment solved in 1423 episodes!	Average Score: 201.02
Time taken by Agent with Epsilon Greedy Policy:  0:08:19.627216
Tutorial Agent with SoftMax Policy
Episode 100	Average Score: 171.35
Episode 107	Average Score: 204.01
Environment solved in 107 episodes!	Average Score: 204.01
Time taken by Agent with Softmax Policy:  0:01:20.224017
```

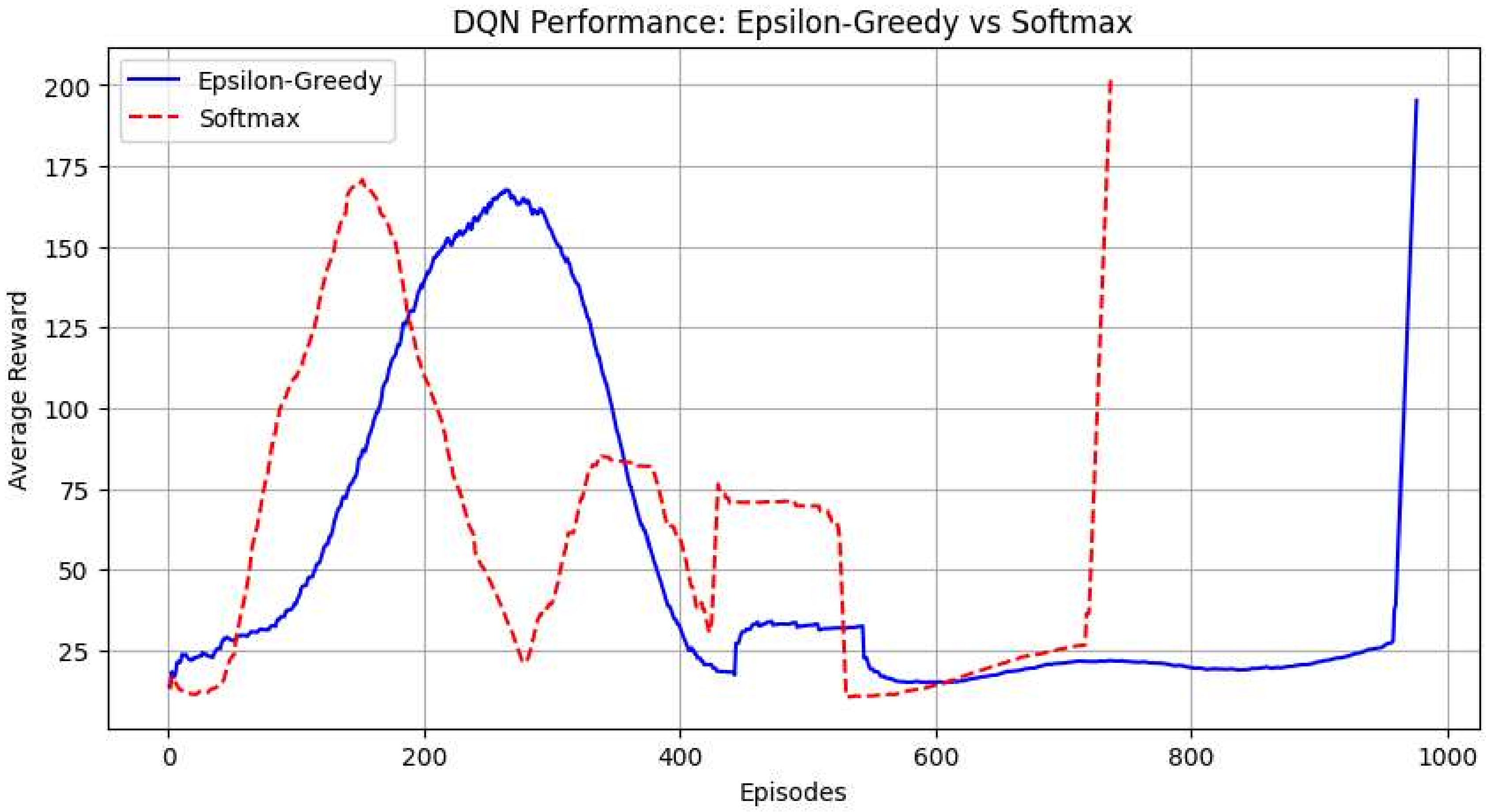
In [17]:

```
import matplotlib.pyplot as plt

# Trail 1
plt.figure(figsize=(10, 5))

plt.plot(episodes_eps, rewards_eps, label="Epsilon-Greedy", color='blue', linestyle='-')
plt.plot(episodes_softmax, rewards_softmax, label="Softmax", color='red', linestyle='--')

plt.xlabel("Episodes")
plt.ylabel("Average Reward")
plt.title("DQN Performance: Epsilon-Greedy vs Softmax")
plt.legend()
plt.grid()

plt.show()
```



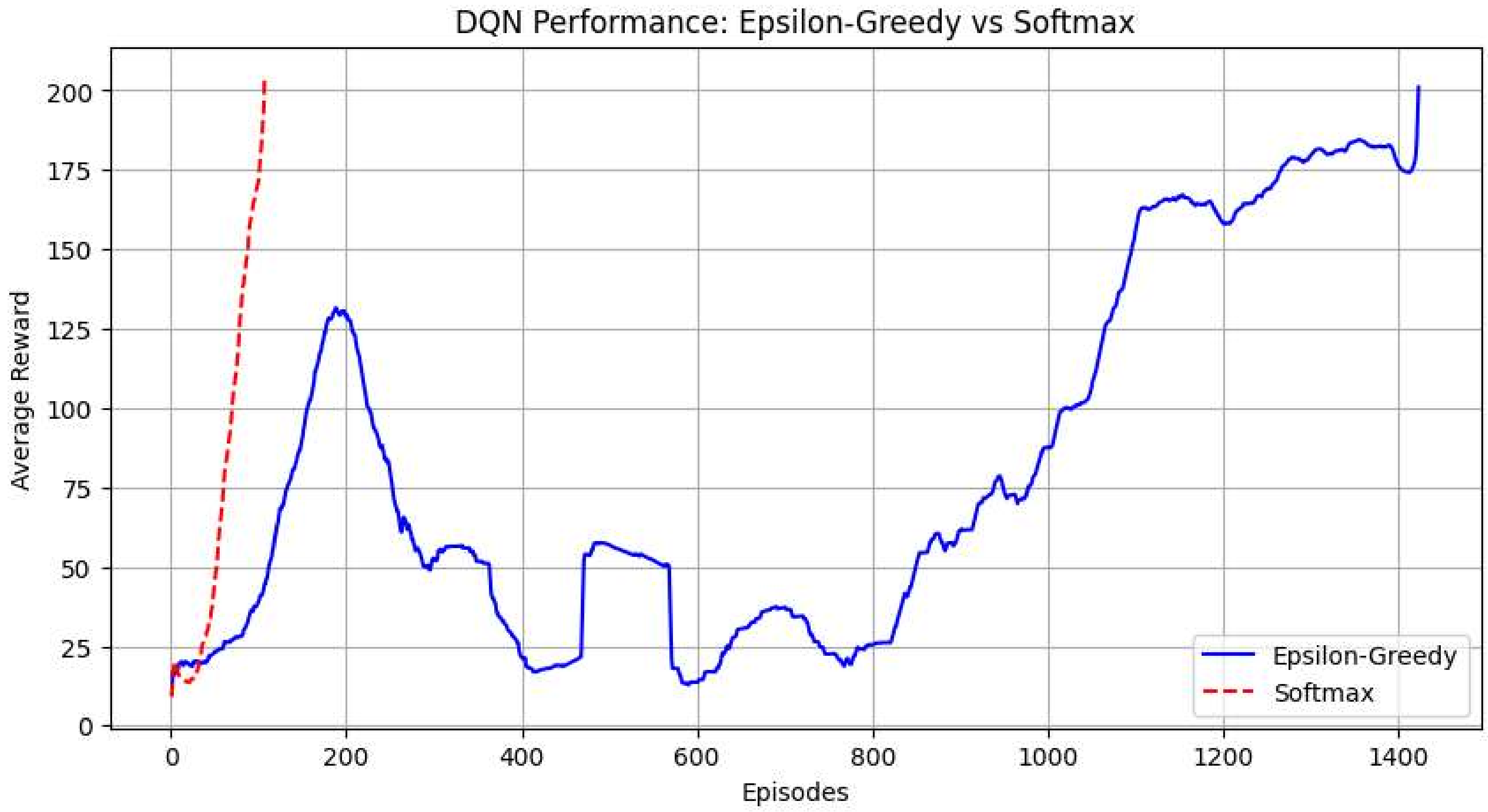
In [19]:

```
import matplotlib.pyplot as plt

# Trail 2
plt.figure(figsize=(10, 5))

plt.plot(episodes_eps, rewards_eps, label="Epsilon-Greedy", color='blue', linestyle='-')
plt.plot(episodes_softmax, rewards_softmax, label="Softmax", color='red', linestyle='--')

plt.xlabel("Episodes")
plt.ylabel("Average Reward")
plt.title("DQN Performance: Epsilon-Greedy vs Softmax")
plt.legend()
plt.grid()

plt.show()

```



In [21]:

```python
import matplotlib.pyplot as plt

# Trail 3
plt.figure(figsize=(10, 5))

plt.plot(episodes_eps, rewards_eps, label="Epsilon-Greedy", color='blue', linestyle='-')
plt.plot(episodes_softmax, rewards_softmax, label="Softmax", color='red', linestyle='--')

plt.xlabel("Episodes")
plt.ylabel("Average Reward")
plt.title("DQN Performance: Epsilon-Greedy vs Softmax")
plt.legend()
plt.grid()

plt.show()

```




DQN with a softmax strategy converges faster because it balances exploration and exploitation more effectively, leading to quicker identification of optimal actions. The controlled randomness from the temperature parameter ($\tau$=0.25−0.30) ensures steady learning without excessive exploration, improving stability.

The reward plots, compute time and the number of episode confirm the same that the DQN with softmax exploration strategy outperforms the DQN algorithm with Epsilon greedy strategy.

### Epsilon Greedy Strategy

| Episodes | Compute Time (min:sec) |
|---|---|
| 1378 | 7 : 55 |
| 976 | 4 : 14 |
| 1423 | 8 : 19 |

### Softmax Strategy

| Episodes | Compute Time (min:sec) |
|---|---|
| 121 | 1 : 45 |
| 737 | 3 : 58 |
| 107 | 1 : 20 |

The temperature parameter of the softmax is found by hypertuning, by running the DQN with different values of $\tau$.