```python
!pip install wandb
import gym
import torch
import numpy as np
import torch.nn as nn
from random import sample, shuffle
from torch.optim import Adam
import torch.nn.functional as F
from collections import namedtuple, deque
import matplotlib.pyplot as plt
import wandb

torch.manual_seed(0)
wandb.login()
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Running on: {device}")

# some fixed parameters
ENV = None
ENV_NAME = None
STATE_SIZE = None
ACTION_SIZE = None
EPISODES = 2000
MAX_T = 200

class DQNetwork(nn.Module):
    def __init__(self, state_size, action_size, num_hidden_units=128):
        super(DQNetwork, self).__init__()
        self.fc1 = nn.Linear(state_size, num_hidden_units)
        self.fc2 = nn.Linear(num_hidden_units, num_hidden_units*2)
        self.fc3 = nn.Linear(num_hidden_units*2, action_size)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

class ReplayBuffer:
    def __init__(self, action_size, max_buffer_size=int(1e5),
batch_size=64):
        self.memory = deque(maxlen=max_buffer_size)
        self.max_buffer_size = max_buffer_size
        self.action_size = action_size
        self.batch_size = batch_size
        self.experience = namedtuple("Experience",
field_names=["state", "action", "reward", "next_state", "done"])

    def add(self, state, action, reward, next_state, done):
        self.memory.append(self.experience(state, action, reward,
next_state, done))
```

```python
    def sample(self):
        experiences = 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 len(self.memory)

default = (5e-4, 0.99, int(1e5), 64, 128, 20, 0.995)

sweep_config = {
    "method": "bayes",
    "metric": {
        'name': 'avg_score',
        'goal': 'maximize'
    },
    "parameters": {
        "lr": {
            "values": [1e-2, 1e-3, 5e-4]
        },
        "gamma": {
            "values": [0.9, 0.99, 0.999]
        },
        "batch_size": {
            "values": [128, 256, 512, 1024]
        },
        "num_hidden_units": {
            "values": [128, 256, 512]
        },
        "update_freq": {
            "values": [5, 10, 20, 50, 75, 100],
        },
        "epsilon_decay": {
            "values": [0.9, 0.99, 0.999]
        }
    }
}

class TutorialAgent():
```

```
    def __init__(self,
                 state_size,
                 action_size,
                 learning_rate=1e-4,
                 gamma=0.99,
                 batch_size=64,
                 dqn_hidden_units=128,
                 buffer_size=int(1e5),
                 update_freq=1000):

        self.state_size = state_size
        self.action_size = action_size
        self.memory = ReplayBuffer(action_size, buffer_size,
batch_size)
        self.qnetwork_local = DQNetwork(state_size,
                                        action_size,

num_hidden_units=dqn_hidden_units).to(device)
        self.qnetwork_target = DQNetwork(state_size,
                                         action_size,

num_hidden_units=dqn_hidden_units).to(device)
        self.optimizer = Adam(self.qnetwork_local.parameters(),
lr=learning_rate)
        self.batch_size = batch_size
        self.update_freq = update_freq
        self.gamma = gamma
        self.t_step = 1

    def step(self, state, action, reward, next_state, done):
        self.memory.add(state, action, reward, next_state, done)
        if len(self.memory) >= self.batch_size:
            experiences = self.memory.sample()
            self.learn(experiences, self.gamma)
        if not (self.t_step % self.update_freq):

self.qnetwork_target.load_state_dict(self.qnetwork_local.state_dict())
        self.t_step += 1

    def act(self, state, eps=0.01):
        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()
        return np.argmax(action_values.cpu().data.numpy()) if
np.random.uniform() > eps else np.random.choice(self.action_size)

    def learn(self, experiences, gamma):
```

```
        states, actions, rewards, next_states, dones = experiences
        Q_targets_next =
self.qnetwork_target(next_states).detach().max(1)[0].unsqueeze(1)
        Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))
        Q_expected = self.qnetwork_local(states).gather(1, actions)
        loss = F.huber_loss(Q_expected, Q_targets)
        self.optimizer.zero_grad()
        loss.backward()
        for param in self.qnetwork_local.parameters():
            param.grad.data.clamp_(-10, 10)
        self.optimizer.step()

def dqn(agent, n_episodes=EPISODES, max_t=MAX_T, eps_start=1.0,
eps_end=0.01, eps_decay=1e-6):
    scores, steps = [], []
    eps = eps_start
    scores_window = deque(maxlen=100)
    for _ in range(1, n_episodes+1):
        score, step = 0, 0
        state = env.reset()
        done = False
        while not done and step < 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
            step += 1

        scores.append(score)
        steps.append(step)
        scores_window.append(score)
        wandb.log({"avg_score": np.mean(scores_window)})
        eps = max(eps_end, eps*eps_decay)
        if np.mean(scores_window) > env.spec.reward_threshold:
            break

    return scores, steps

def train(config=None):
    run = wandb.init(config=config)
    config = wandb.config

    wandb.run.name = (
        f"{ENV_NAME}"
        f"_lr_{config.lr}"
        f"_gamma_{config.gamma}"
        f"_batch-size_{config.batch_size}"
        f"_hidden-units_{config.num_hidden_units}"
        f"_update-freq_{config.update_freq}"
```

```
            f"_epsilon_decay_{config.epsilon_decay}"
        )


        all_scores, all_steps, all_episodes = [], [], []
        for i in range(0, 10):
            rl_agent = TutorialAgent(STATE_SIZE,
                                     ACTION_SIZE,
                                     gamma=config.gamma,
                                     learning_rate=config.lr,
                                     batch_size=config.batch_size,
                                     update_freq=config.update_freq,

    dqn_hidden_units=config.num_hidden_units)

            scores, steps = dqn(rl_agent, eps_decay=config.epsilon_decay)
            all_scores.append(scores)
            all_steps.append(steps)

        for i, y in enumerate(all_scores):
          plt.plot(range(1, len(y)+1), y, label=f"run_{i+1}")
        plt.xlabel("num_episodes")
        plt.ylabel("scores")
        plt.legend(loc="best")
        plt.title("scores vs episodes")
        plt.show()

        for i, y in enumerate(all_steps):
          plt.plot(range(1, len(y)+1), y, label=f"run_{i+1}")
        plt.xlabel("num_episodes")
        plt.ylabel("steps")
        plt.legend(loc="best")
        plt.title("steps vs episodes")
        plt.show()

        avg_score = np.mean(np.array(all_scores, dtype=object))
        print(f"average score for this config {avg_score:.2f}")
        print("-"*100)

    env = gym.make('CartPole-v1')
    env.seed(0)
    ENV_NAME = "CARTPOLE"
    STATE_SIZE = env.observation_space.shape[0]
    ACTION_SIZE = env.action_space.n

    sweep_id = wandb.sweep(sweep_config, project="CS6700-PA2",
    entity="varungumma")
    wandb.agent(sweep_id, function=train, count=12)
```

```python
env = gym.make('Acrobot-v1')
env.seed(0)
ENV_NAME = "ACROBOT"
STATE_SIZE = env.observation_space.shape[0]
ACTION_SIZE = env.action_space.n

sweep_id = wandb.sweep(sweep_config, project="CS6700-PA2",
entity="varungumma")
wandb.agent(sweep_id, function=train, count=12)
```

```python
env = gym.make('MountainCar-v0')
env.seed(0)
ENV_NAME = "MOUNTAIN_CAR"
STATE_SIZE = env.observation_space.shape[0]
ACTION_SIZE = env.action_space.n

sweep_id = wandb.sweep(sweep_config, project="CS6700-PA2",
entity="varungumma")
wandb.agent(sweep_id, function=train, count=12)
```

```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
```

Initializing Actor-Critic Network

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

```
class Agent1:
    """
    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_next = tf.stop_gradient(V_s_next)

            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))
```

```
#Storing variables as pickle files in the Google drive
#Helpful when the environment gets dosconnected and rebooted
#Used in plotting the graphs

from google.colab import drive
drive.mount('/content/drive')
```

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

Extracting variables from pickle files in the Google Drive

```
def get_variance(x):
    z = [i for i in x if i is not None]
    #print(z)
    return np.var(z)

import itertools as it
variance_of_total_episode_reward_across_runs_for_each_episode =
list(map(get_variance, it.zip_longest(*runs_reward_list) ))
```

Average function

```
def get_average(x):
    #print('x before: ', x)
    y = [i for i in x if i is not None]
    #print(y)
    return sum(y, 0.0) / len(y)
```

Mounting Google Drive to store variables as pickle files

```
#Storing variables as pickle files in the Google drive
#Helpful when the environment gets dosconnected and rebooted
#Used in plotting the graphs

from google.colab import drive
drive.mount('/content/drive')
```

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

CartPole-v1

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

#Initializing Agent
#agent = Agent1(lr=1e-4, action_size=env.action_space.n)
#Number of episodes
episodes = 1000
runs = 3
#tf.compat.v1.reset_default_graph() #Should this be inside the run for
loop??


runs_reward_list = []
runs_steps_list = []
required_episodes_to_solve_list = [episodes]*runs
#variance_list = []
step_count = 0
begin_time = datetime.datetime.now()

for run in range(runs):
    #Initializing Agent
    print('Starting run no: ', run )
    agent = Agent1(lr=1e-4, action_size=env.action_space.n)
    tf.compat.v1.reset_default_graph()
    episodes_reward_list = []
    episodes_steps_list = []
    environment_solved = False
    for ep in range(1, episodes + 1):
        step_count = 0
        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
            step_count += 1
        episodes_reward_list.append(ep_rew)
        episodes_steps_list.append(step_count)

        if ep % 10 == 0:
            avg_rew = np.mean(episodes_reward_list[-10:])
            print('Episode ', ep, 'Reward %f' % ep_rew, 'Average
Reward %f' % avg_rew, 'Step count %f' % step_count)

        if ep % 100 and environment_solved == False:
            avg_100 =  np.mean(episodes_reward_list[-100:])
            if avg_100 > env.spec.reward_threshold:
                print('Environment solved at Episode ',ep)
                print('Average Reward for last 100 episodes: ',
avg_100 )
                print('Threshold of the environment: ',
env.spec.reward_threshold)
                #required_episodes_to_solve_list.append(ep-100)
                required_episodes_to_solve_list[run] = ep
                environment_solved = True
                break
    runs_reward_list.append(episodes_reward_list)
    runs_steps_list.append(episodes_steps_list)

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


#Storing variables as pickle files
import pickle

with open('/content/drive/My Drive/RL-PA2/cartpole1-one-step-hidden-
1024-1024-lr-1e4-episodes-1000-runs-steps.pickle', 'wb') as f:
  pickle.dump(runs_steps_list,f)


with open('/content/drive/My Drive/RL-PA2/cartpole1-one-step-hidden-
1024-1024-lr-1e4-episodes-1000-runs-rewards.pickle', 'wb') as f:
  pickle.dump(runs_reward_list,f)

with open('/content/drive/My Drive/RL-PA2/cartpole1-one-step-hidden-
1024-1024-lr-1e4-episodes-1000-required-episodes.pickle', 'wb') as f:
  pickle.dump(required_episodes_to_solve_list,f)

# Load the pickled variable saved in Drive.
with open('/content/drive/My Drive/RL-PA2/cartpole1-one-step-hidden-
```

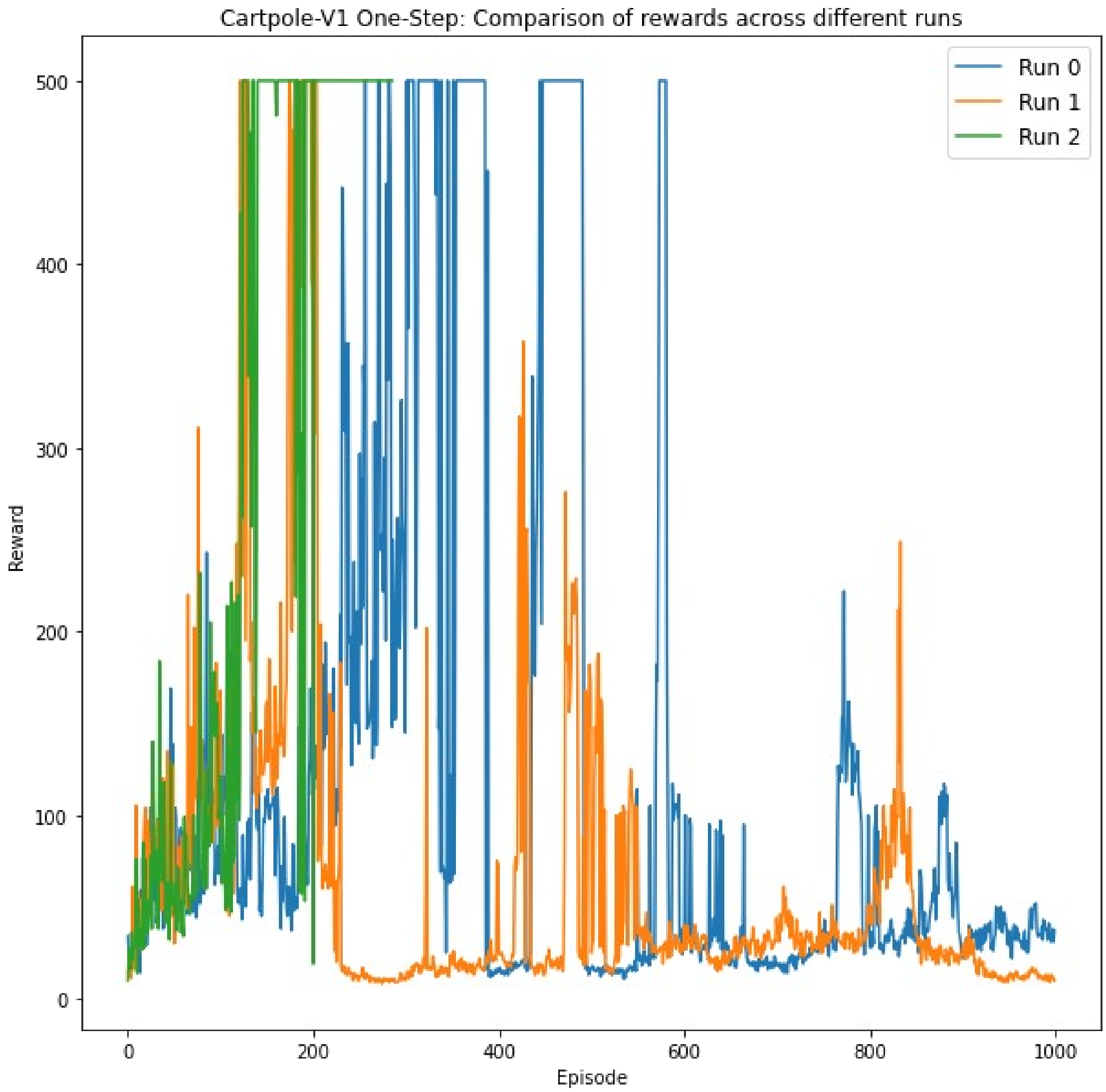
```
1024-1024-lr-1e4-episodes-1000-runs-steps.pickle', 'rb') as f:
  runs_steps_list = pickle.load(f)


with open('/content/drive/My Drive/RL-PA2/cartpole1-one-step-hidden-
1024-1024-lr-1e4-episodes-1000-runs-rewards.pickle', 'rb') as f:
  runs_reward_list = pickle.load(f)

with open('/content/drive/My Drive/RL-PA2/cartpole1-one-step-hidden-
1024-1024-lr-1e4-episodes-1000-required-episodes.pickle', 'rb') as f:
  required_episodes_to_solve_list = pickle.load(f)

print(len(runs_reward_list))
print(len(runs_reward_list[0]))
print(required_episodes_to_solve_list)
print(np.average(required_episodes_to_solve_list))
```

```
3
1000
[1000, 1000, 286]
762.0
```

Plots

Plotting reward curves

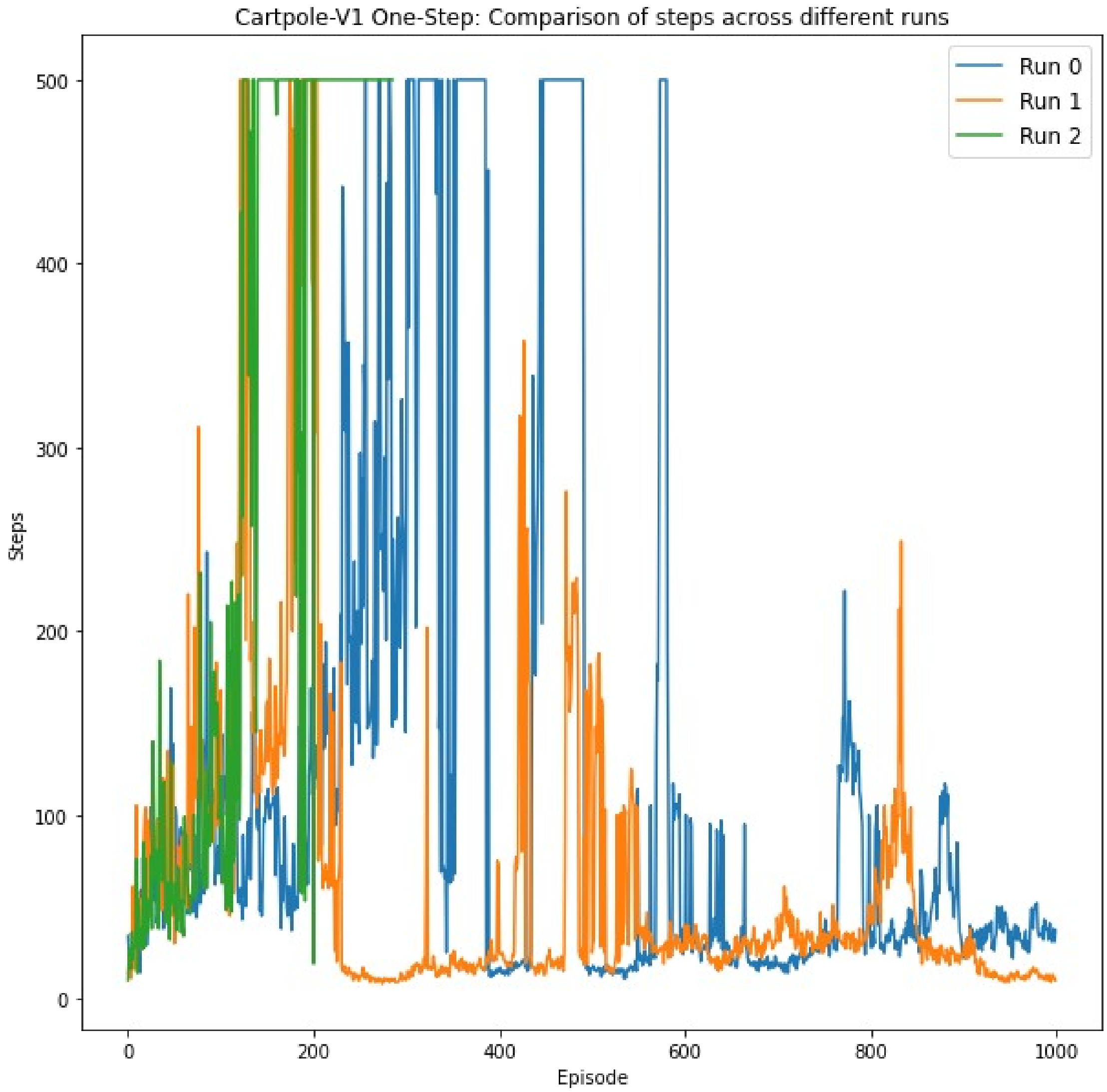
```
# Create subplot
fig, ax = plt.subplots(figsize=(10,10))

x0 = np.arange(len(runs_reward_list[0]))
y0 = runs_reward_list[0]
ax.plot(x0,y0, label='Run 0')

x1 = np.arange(len(runs_reward_list[1]))
y1 = runs_reward_list[1]
ax.plot(x1,y1, label='Run 1')

x2 = np.arange(len(runs_reward_list[2]))
y2 = runs_reward_list[2]
ax.plot(x2,y2, label='Run 2')

legend = ax.legend(loc='upper right', fontsize='large')

plt.xlabel('Episode')
plt.ylabel('Reward')

plt.title('Cartpole-V1 One-Step: Comparison of rewards across
different runs')

plt.show()
```



Plotting Number of Steps to reach goal in each episode

```
# Create subplot
fig, ax = plt.subplots(figsize=(10,10))

x0 = np.arange(len(runs_steps_list[0]))
y0 = runs_steps_list[0]
ax.plot(x0,y0, label='Run 0')

x1 = np.arange(len(runs_steps_list[1]))
y1 = runs_steps_list[1]
ax.plot(x1,y1, label='Run 1')

x2 = np.arange(len(runs_steps_list[2]))
y2 = runs_steps_list[2]
ax.plot(x2,y2, label='Run 2')
```

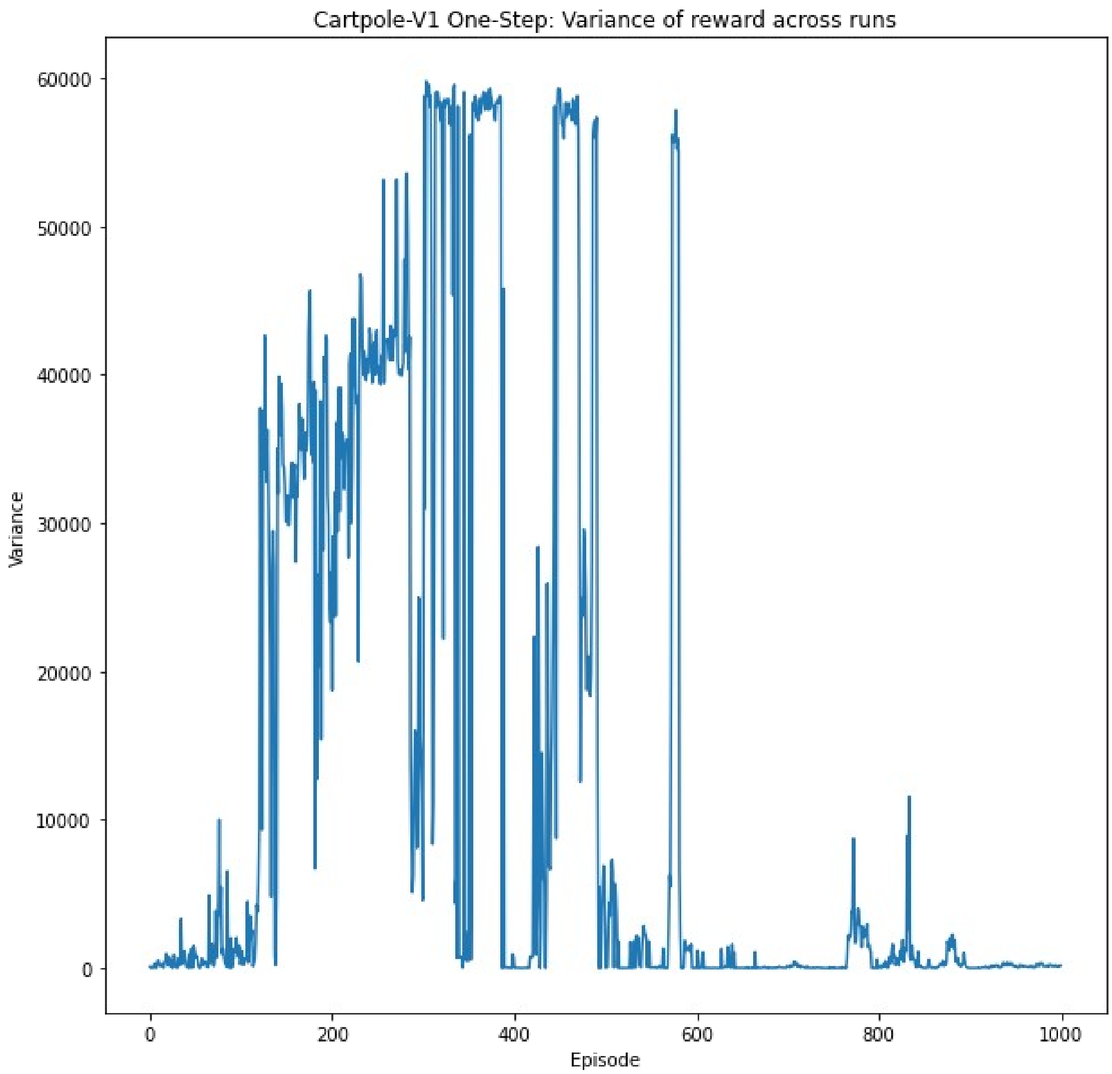
```
legend = ax.legend(loc='upper right', fontsize='large')

plt.xlabel('Episode')
plt.ylabel('Steps')

plt.title('Cartpole-V1 One-Step: Comparison of steps across different
runs')

plt.show()
```



Plotting variance

```
def get_variance(x):
    z = [i for i in x if i is not None]
    #print(z)
    return np.var(z)

import itertools as it
```

```
variance_of_total_episode_reward_across_runs_for_each_episode =
list(map(get_variance, it.zip_longest(*runs_reward_list) ))


plt.figure(figsize=(10,10))
plt.xlabel('Episode')
plt.ylabel('Variance')
plt.plot(np.arange(len(variance_of_total_episode_reward_across_runs_fo
r_each_episode)),
variance_of_total_episode_reward_across_runs_for_each_episode , 0)
plt.title('Cartpole-V1 One-Step: Variance of reward across runs')
plt.show()
```



Acrobot-v1

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

```
#Initializing Agent
#agent = Agent1(lr=1e-4, action_size=env.action_space.n)
#Number of episodes
episodes = 1000
runs = 3
#tf.compat.v1.reset_default_graph() #Should this be inside the run for
loop??


runs_reward_list = []
runs_steps_list = []
required_episodes_to_solve_list = [episodes]*runs
#variance_list = []
step_count = 0
begin_time = datetime.datetime.now()

for run in range(runs):
    #Initializing Agent
    print('Starting run no: ', run )
    agent = Agent1(lr=1e-4, action_size=env.action_space.n)
    tf.compat.v1.reset_default_graph()
    episodes_reward_list = []
    episodes_steps_list = []
    environment_solved = False
    for ep in range(1, episodes + 1):
        step_count = 0
        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
            step_count += 1
        episodes_reward_list.append(ep_rew)
        episodes_steps_list.append(step_count)

        if ep % 10 == 0:
            avg_rew = np.mean(episodes_reward_list[-10:])
            print('Episode ', ep, 'Reward %f' % ep_rew, 'Average
Reward %f' % avg_rew, 'Step count %f' % step_count)

        if ep % 100 and environment_solved == False:
            avg_100 =  np.mean(episodes_reward_list[-100:])
            if avg_100 > env.spec.reward_threshold:
```

```
                print('Environment solved at Episode ',ep)
                print('Average Reward for last 100 episodes: ',
avg_100 )
                print('Threshold of the environment: ',
env.spec.reward_threshold)
                #required_episodes_to_solve_list.append(ep-100)
                required_episodes_to_solve_list[run] = ep
                environment_solved = True
                break
    runs_reward_list.append(episodes_reward_list)
    runs_steps_list.append(episodes_steps_list)

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

#Storing variables as pickle files
import pickle

with open('/content/drive/My Drive/RL-PA2/Acrobot1-one-step-hidden-
1024-1024-lr-1e4-episodes-1000-runs-steps.pickle', 'wb') as f:
  pickle.dump(runs_steps_list,f)


with open('/content/drive/My Drive/RL-PA2/Acrobot1-one-step-hidden-
1024-1024-lr-1e4-episodes-1000-runs-rewards.pickle', 'wb') as f:
  pickle.dump(runs_reward_list,f)

with open('/content/drive/My Drive/RL-PA2/Acrobot1-one-step-hidden-
1024-1024-lr-1e4-episodes-1000-required-episodes.pickle', 'wb') as f:
  pickle.dump(required_episodes_to_solve_list,f)
```

Plots

Plotting reward curves

```
# Create subplot
fig, ax = plt.subplots(figsize=(10,10))

x0 = np.arange(len(runs_reward_list[0]))
y0 = runs_reward_list[0]
ax.plot(x0,y0, label='Run 0')

x1 = np.arange(len(runs_reward_list[1]))
y1 = runs_reward_list[1]
ax.plot(x1,y1, label='Run 1')

x2 = np.arange(len(runs_reward_list[2]))
y2 = runs_reward_list[2]
ax.plot(x2,y2, label='Run 2')
```

```
legend = ax.legend(loc='upper right', fontsize='large')

plt.xlabel('Episode')
plt.ylabel('Reward')

plt.title('Acrobot-v1 One-Step: Comparison of rewards across different
runs')

plt.show()
```

Plotting number of steps to reach goal in each episode

```
# Create subplot
fig, ax = plt.subplots(figsize=(10,10))

x0 = np.arange(len(runs_steps_list[0]))
y0 = runs_steps_list[0]
ax.plot(x0,y0, label='Run 0')

x1 = np.arange(len(runs_steps_list[1]))
y1 = runs_steps_list[1]
ax.plot(x1,y1, label='Run 1')

x2 = np.arange(len(runs_steps_list[2]))
y2 = runs_steps_list[2]
ax.plot(x2,y2, label='Run 2')

legend = ax.legend(loc='upper right', fontsize='large')

plt.xlabel('Episode')
plt.ylabel('Steps')

plt.title('Acrobot-v1 One-Step: Comparison of steps across different
runs')

plt.show()
```

Plotting variance

```
def get_variance(x):
    z = [i for i in x if i is not None]
    #print(z)
    return np.var(z)

import itertools as it
variance_of_total_episode_reward_across_runs_for_each_episode =
list(map(get_variance, it.zip_longest(*runs_reward_list) ))


plt.figure(figsize=(10,10))
plt.xlabel('Episode')
```

```
plt.ylabel('Variance')
plt.plot(np.arange(len(variance_of_total_episode_reward_across_runs_fo
r_each_episode)),
variance_of_total_episode_reward_across_runs_for_each_episode , 0)
plt.title('Acrobot-V1 One-Step: Variance of reward across runs')
plt.show()
```

```
'''
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
```

Mounting Google Drive to store variables as pickel files

```
#Storing variables as pickle files in the Google drive
#Helpful when the environment gets dosconnected and rebooted
#Used in plotting the graphs

from google.colab import drive
drive.mount('/content/drive')
```

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

Initializing Actor-Critic Network

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

```
class Agent1:
    """
    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
        """
        #print('state ',state)
        pi,_ = self.ac_model(state)
        #print('pi ', pi)

        action_probabilities = tfp.distributions.Categorical(probs=pi)
        #print('action_probabilities ', action_probabilities)
        sample = action_probabilities.sample()
        #print('sample ', sample)
        #print('return value',int(sample.numpy()[0]))

        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, n_step):
        """
        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_next = tf.stop_gradient(V_s_next)

            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**n_step) * 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))
```

CartPole-v1

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

#Initializing Agent
#agent = Agent1(lr=1e-4, action_size=env.action_space.n)
#Number of episodes
episodes = 1000
runs = 3
```

```
gamma = 0.99
n_step = 8
#tf.compat.v1.reset_default_graph() #Should this be inside the run for
loop??


runs_reward_list = []
runs_steps_list = []
required_episodes_to_solve_list = [episodes]*runs
#variance_list = []
step_count = 0
begin_time = datetime.datetime.now()

for run in range(runs):
    #Initializing Agent
    print('Starting run no: ', run )
    agent = Agent1(lr=1e-4, action_size=env.action_space.n)
    tf.compat.v1.reset_default_graph()
    episodes_reward_list = []
    episodes_steps_list = []
    environment_solved = False
    for ep in range(1, episodes + 1):
        step_count = 0
        state = env.reset().reshape(1,-1)
        done = False
        ep_rew = 0
        list_of_states_in_episode = []
        list_of_actions_in_episode = []
        list_of_step_rewards_in_episode = []
        list_of_step_rewards_in_episode.append(None) #Following
convention
        while not done:
            list_of_states_in_episode.append(state)
            action = agent.sample_action(state) ##Sample Action
            #print('before ',action)
            #print('after', action)
            next_state, reward, done, info = env.step(action) ##Take
action
            list_of_actions_in_episode.append(action)
            list_of_step_rewards_in_episode.append(reward)
            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
            step_count += 1
        episodes_reward_list.append(ep_rew)
        episodes_steps_list.append(step_count)

        for t in range(0,len(list_of_states_in_episode)):
```

```
          target = 0
          for t_dash in
range(t,min(n_step+t,len(list_of_states_in_episode))):
            target = target + gamma**(t_dash-t) *
list_of_step_rewards_in_episode[t_dash+1]

          if(n_step+t <  len(list_of_states_in_episode)):
            state_after_n_steps = list_of_states_in_episode[n_step+t]
          else:
            state_after_n_steps = list_of_states_in_episode[-1]


agent.learn(list_of_states_in_episode[t],list_of_actions_in_episode[t]
, target ,state_after_n_steps, done, n_step)


        if ep % 10 == 0:
            avg_rew = np.mean(episodes_reward_list[-10:])
            print('Episode ', ep, 'Reward %f' % ep_rew, 'Average
Reward %f' % avg_rew, 'Step count %f' % step_count)

        if ep % 100 and environment_solved == False:
            avg_100 =  np.mean(episodes_reward_list[-100:])
            if avg_100 > env.spec.reward_threshold:
                print('Environment solved at Episode ',ep)
                print('Average Reward for last 100 episodes: ',
avg_100 )
                print('Threshold of the environment: ',
env.spec.reward_threshold)
                #required_episodes_to_solve_list.append(ep-100)
                required_episodes_to_solve_list[run] = ep
                environment_solved = True
                break
    runs_reward_list.append(episodes_reward_list)
    runs_steps_list.append(episodes_steps_list)

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

#Storing variables as pickle files
import pickle

with open('/content/drive/My Drive/RL-PA2/cartpole1-n-step-8-hidden-
1024-1024-lr-1e4-episodes-1000-runs-steps.pickle', 'wb') as f:
  pickle.dump(runs_steps_list,f)


with open('/content/drive/My Drive/RL-PA2/cartpole1-n-step-8-hidden-
1024-1024-lr-1e4-episodes-1000-runs-rewards.pickle', 'wb') as f:
```

```
  pickle.dump(runs_reward_list,f)

with open('/content/drive/My Drive/RL-PA2/cartpole1-n-step-8-hidden-1024-1024-lr-1e4-episodes-1000-required-episodes.pickle', 'wb') as f:
  pickle.dump(required_episodes_to_solve_list,f)
```

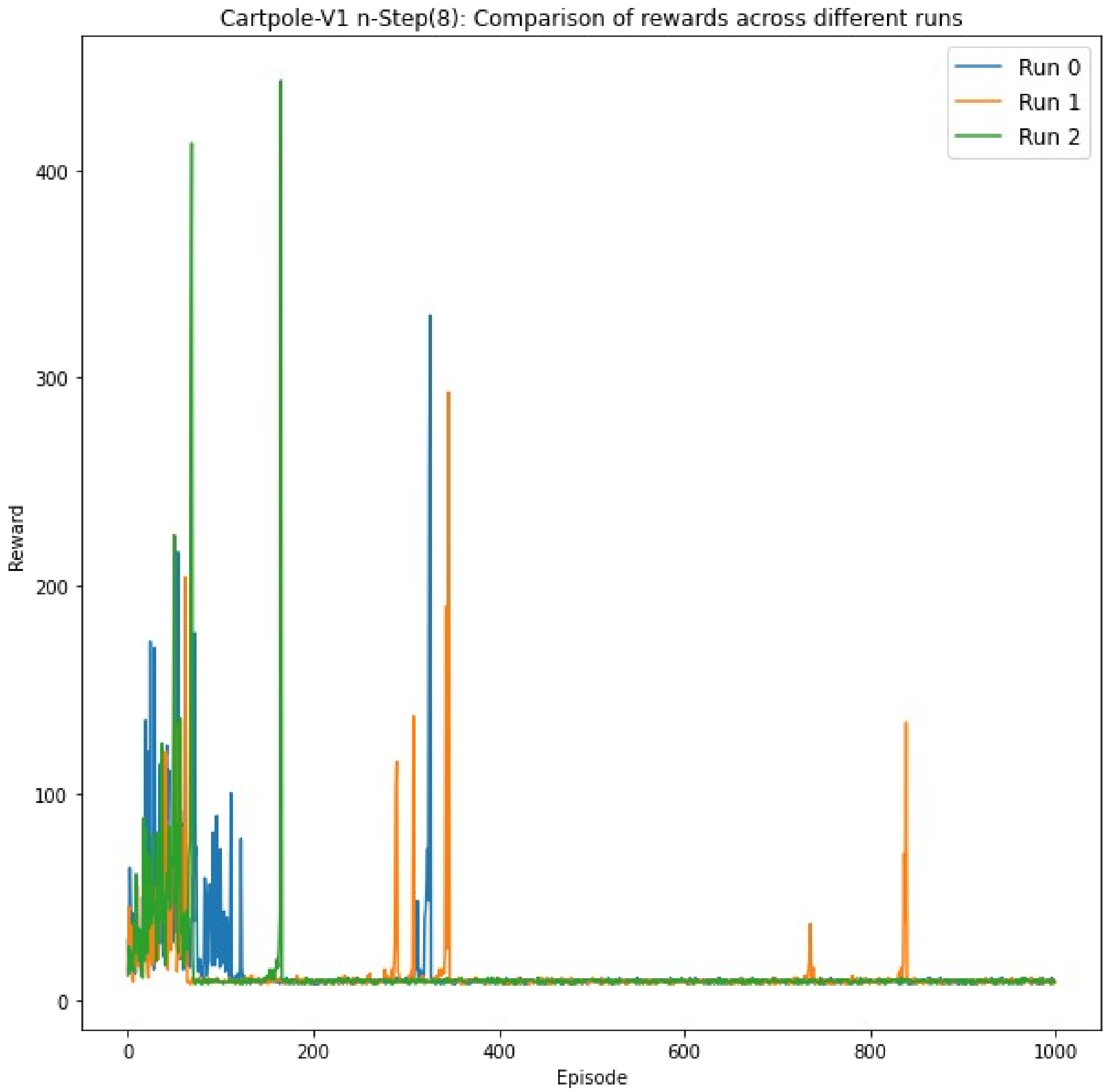
```
print(len(runs_reward_list))
print(len(runs_reward_list[0]))
print(required_episodes_to_solve_list)
print(np.average(required_episodes_to_solve_list))
```

```
3
1000
[1000, 1000, 1000]
1000.0
```

Plotting

Plotting reward curves

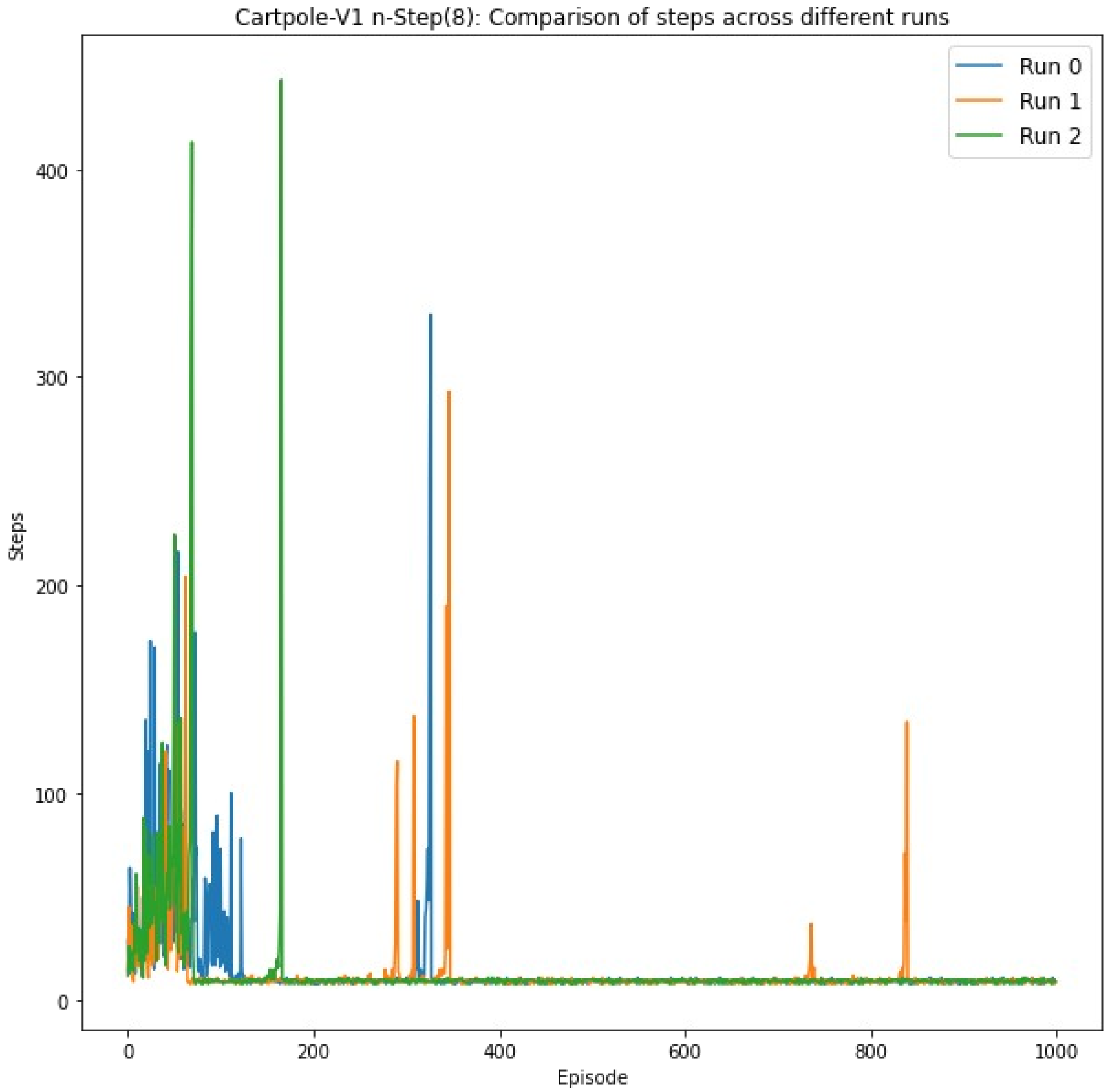
```
# Create subplot
fig, ax = plt.subplots(figsize=(10,10))

x0 = np.arange(len(runs_reward_list[0]))
y0 = runs_reward_list[0]
ax.plot(x0,y0, label='Run 0')

x1 = np.arange(len(runs_reward_list[1]))
y1 = runs_reward_list[1]
ax.plot(x1,y1, label='Run 1')

x2 = np.arange(len(runs_reward_list[2]))
y2 = runs_reward_list[2]
ax.plot(x2,y2, label='Run 2')

legend = ax.legend(loc='upper right', fontsize='large')

plt.xlabel('Episode')
plt.ylabel('Reward')

plt.title('Cartpole-V1 n-Step(8): Comparison of rewards across different runs')

plt.show()
```



Plotting number of steps to reach goal in each episode

```
# Create subplot
fig, ax = plt.subplots(figsize=(10,10))

x0 = np.arange(len(runs_steps_list[0]))
y0 = runs_steps_list[0]
ax.plot(x0,y0, label='Run 0')

x1 = np.arange(len(runs_steps_list[1]))
y1 = runs_steps_list[1]
ax.plot(x1,y1, label='Run 1')

x2 = np.arange(len(runs_steps_list[2]))
y2 = runs_steps_list[2]
ax.plot(x2,y2, label='Run 2')
```

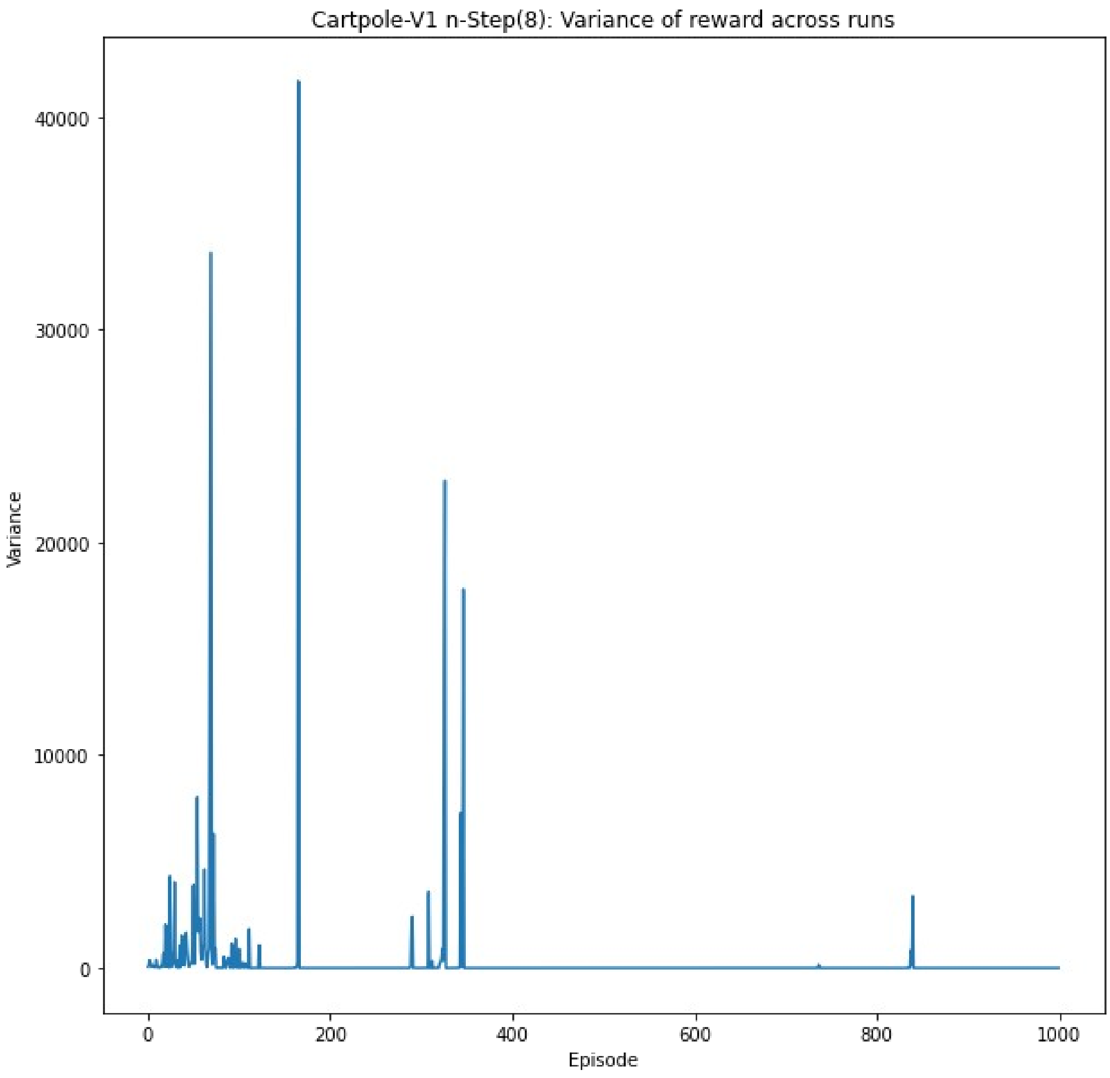
```
legend = ax.legend(loc='upper right', fontsize='large')

plt.xlabel('Episode')
plt.ylabel('Steps')

plt.title('Cartpole-V1 n-Step(8): Comparison of steps across different runs')

plt.show()
```



Plotting Variance

```
def get_variance(x):
    z = [i for i in x if i is not None]
    #print(z)
    return np.var(z)

import itertools as it
```

```
variance_of_total_episode_reward_across_runs_for_each_episode =
list(map(get_variance, it.zip_longest(*runs_reward_list) ))


plt.figure(figsize=(10,10))
plt.xlabel('Episode')
plt.ylabel('Variance')
plt.plot(np.arange(len(variance_of_total_episode_reward_across_runs_fo
r_each_episode)),
variance_of_total_episode_reward_across_runs_for_each_episode , 0)
plt.title('Cartpole-V1 n-Step(8): Variance of reward across runs')
plt.show()
```



Acrobot-v1

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

```
#Initializing Agent
#agent = Agent1(lr=1e-4, action_size=env.action_space.n)
#Number of episodes
episodes = 1000
runs = 3
gamma = 0.99
n_step = 8
#tf.compat.v1.reset_default_graph() #Should this be inside the run for
loop??


runs_reward_list = []
runs_steps_list = []
required_episodes_to_solve_list = [episodes]*runs
#variance_list = []
step_count = 0
begin_time = datetime.datetime.now()

for run in range(runs):
    #Initializing Agent
    print('Starting run no: ', run )
    agent = Agent1(lr=1e-4, action_size=env.action_space.n)
    tf.compat.v1.reset_default_graph()
    episodes_reward_list = []
    episodes_steps_list = []
    environment_solved = False
    for ep in range(1, episodes + 1):
        step_count = 0
        state = env.reset().reshape(1,-1)
        done = False
        ep_rew = 0
        list_of_states_in_episode = []
        list_of_actions_in_episode = []
        list_of_step_rewards_in_episode = []
        list_of_step_rewards_in_episode.append(None) #Following
convention
        while not done:
            list_of_states_in_episode.append(state)
            action = agent.sample_action(state) ##Sample Action
            #print('before ',action)
            #print('after', action)
            next_state, reward, done, info = env.step(action) ##Take
action
            list_of_actions_in_episode.append(action)
            list_of_step_rewards_in_episode.append(reward)
            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
```

```
            step_count += 1
        episodes_reward_list.append(ep_rew)
        episodes_steps_list.append(step_count)

        for t in range(0,len(list_of_states_in_episode)):
          target = 0
          for t_dash in
range(t,min(n_step+t,len(list_of_states_in_episode))):
            target = target + gamma**(t_dash-t) *
list_of_step_rewards_in_episode[t_dash+1]

          if(n_step+t <  len(list_of_states_in_episode)):
            state_after_n_steps = list_of_states_in_episode[n_step+t]
          else:
            state_after_n_steps = list_of_states_in_episode[-1]


agent.learn(list_of_states_in_episode[t],list_of_actions_in_episode[t]
, target ,state_after_n_steps, done, n_step)


        if ep % 10 == 0:
            avg_rew = np.mean(episodes_reward_list[-10:])
            print('Episode ', ep, 'Reward %f' % ep_rew, 'Average
Reward %f' % avg_rew, 'Step count %f' % step_count)

        if ep % 100 and environment_solved == False:
            avg_100 =  np.mean(episodes_reward_list[-100:])
            if avg_100 > env.spec.reward_threshold:
                print('Environment solved at Episode ',ep)
                print('Average Reward for last 100 episodes: ',
avg_100 )
                print('Threshold of the environment: ',
env.spec.reward_threshold)
                #required_episodes_to_solve_list.append(ep-100)
                required_episodes_to_solve_list[run] = ep
                environment_solved = True
                break
    runs_reward_list.append(episodes_reward_list)
    runs_steps_list.append(episodes_steps_list)

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


#Storing variables as pickle files
import pickle

with open('/content/drive/My Drive/RL-PA2/Acrobot1-n-step-8-hidden-
```

```
1024-1024-lr-1e4-episodes-1000-runs-steps.pickle', 'wb') as f:
  pickle.dump(runs_steps_list,f)


with open('/content/drive/My Drive/RL-PA2/Acrobot1-n-step-8-hidden-
1024-1024-lr-1e4-episodes-1000-runs-rewards.pickle', 'wb') as f:
  pickle.dump(runs_reward_list,f)

with open('/content/drive/My Drive/RL-PA2/Acrobot1-n-step-8-hidden-
1024-1024-lr-1e4-episodes-1000-required-episodes.pickle', 'wb') as f:
  pickle.dump(required_episodes_to_solve_list,f)

print(len(runs_reward_list))
print(len(runs_reward_list[0]))
print(required_episodes_to_solve_list)
print(np.average(required_episodes_to_solve_list))
```

Plotting

Plotting reward curves

```
# Create subplot
fig, ax = plt.subplots(figsize=(10,10))

x0 = np.arange(len(runs_reward_list[0]))
y0 = runs_reward_list[0]
ax.plot(x0,y0, label='Run 0')

x1 = np.arange(len(runs_reward_list[1]))
y1 = runs_reward_list[1]
ax.plot(x1,y1, label='Run 1')

x2 = np.arange(len(runs_reward_list[2]))
y2 = runs_reward_list[2]
ax.plot(x2,y2, label='Run 2')

legend = ax.legend(loc='upper right', fontsize='large')

plt.xlabel('Episode')
plt.ylabel('Reward')

plt.title('Acrobot-v1 n-Step(8): Comparison of rewards across
different runs')

plt.show()
```

Plotting number of steps to rach goal in each episode

```
# Create subplot
fig, ax = plt.subplots(figsize=(10,10))
```

```
x0 = np.arange(len(runs_steps_list[0]))
y0 = runs_steps_list[0]
ax.plot(x0,y0, label='Run 0')

x1 = np.arange(len(runs_steps_list[1]))
y1 = runs_steps_list[1]
ax.plot(x1,y1, label='Run 1')

x2 = np.arange(len(runs_steps_list[2]))
y2 = runs_steps_list[2]
ax.plot(x2,y2, label='Run 2')

legend = ax.legend(loc='upper right', fontsize='large')

plt.xlabel('Episode')
plt.ylabel('Steps')

plt.title('Acrobot-v1 n-Step(8): Comparison of steps across different
runs')

plt.show()
```

Plotting variances

```
def get_variance(x):
    z = [i for i in x if i is not None]
    #print(z)
    return np.var(z)

import itertools as it
variance_of_total_episode_reward_across_runs_for_each_episode =
list(map(get_variance, it.zip_longest(*runs_reward_list) ))


plt.figure(figsize=(10,10))
plt.xlabel('Episode')
plt.ylabel('Variance')
plt.plot(np.arange(len(variance_of_total_episode_reward_across_runs_fo
r_each_episode)),
variance_of_total_episode_reward_across_runs_for_each_episode , 0)
plt.title('Acrobot-v1 n-Step(8): Variance of reward across runs')
plt.show()
```

```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

#tf.compat.v1.enable_eager_execution()

class ActorCriticModel(tf.keras.Model):
    """
    Defining policy and value networkss
    """
    def __init__(self, action_size, n_hidden1=64, n_hidden2=64):
        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

class Agent1:
    """
    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
        """
        #print('state ',state)
        pi,_ = self.ac_model(state)
        #print('pi ', pi)

        action_probabilities = tfp.distributions.Categorical(probs=pi)
        #print('action_probabilities ', action_probabilities)
        sample = action_probabilities.sample()
        #print('sample ', sample)
        #print('return value',int(sample.numpy()[0]))

        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

    def calculateDeltaAndPi(self, state, reward):
      pi, V_s = self.ac_model(state)
      V_s = tf.squeeze(V_s)
```

```
        delta = reward - V_s
        return delta, pi

    @tf.function
    def learn(self,list_of_states_in_episode,
list_of_actions_in_episode, list_of_targets_in_episode):
        """
        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:
          loss = tf.constant(0, dtype=tf.float32)
          for i in range(len(list_of_states_in_episode)):
            pi, V_s = self.ac_model(list_of_states_in_episode[i])
            V_s = tf.squeeze(V_s)
            delta = list_of_targets_in_episode[i] - V_s

            loss += self.actor_loss(list_of_actions_in_episode[i],
                                    pi,
                                    delta)
            loss += self.critic_loss(delta)

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

CartPole-v1

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

#Initializing Agent
#agent = Agent1(lr=1e-4, action_size=env.action_space.n)
#Number of episodes
episodes = 1000
runs = 1
gamma = 0.99
#n_step = 8
#tf.compat.v1.reset_default_graph() #Should this be inside the run for
loop??


runs_reward_list = []
runs_steps_list = []
required_episodes_to_solve_list = [episodes]*runs
#variance_list = []
step_count = 0
```

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

for run in range(runs):
    #Initializing Agent
    print('Starting run no: ', run )
    agent = Agent1(lr=1e-4, action_size=env.action_space.n)
    tf.compat.v1.reset_default_graph()
    episodes_reward_list = []
    episodes_steps_list = []
    environment_solved = False
    for ep in range(1, episodes + 1):
        step_count = 0
        state = env.reset().reshape(1,-1)
        done = False
        ep_rew = 0
        list_of_states_in_episode = []
        list_of_actions_in_episode = []
        list_of_step_rewards_in_episode = []
        list_of_deltas_in_episode = []
        list_of_targets_in_episode = []
        list_of_pi_in_episode = []
        list_of_step_rewards_in_episode.append(None) #Following
convention
        while not done:
            list_of_states_in_episode.append(state)
            action = agent.sample_action(state) ##Sample Action
            #print('before ',action)
            #print('after', action)
            next_state, reward, done, info = env.step(action) ##Take
action
            list_of_actions_in_episode.append(action)
            list_of_step_rewards_in_episode.append(reward)
            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
            step_count += 1
        episodes_reward_list.append(ep_rew)
        episodes_steps_list.append(step_count)

        n_step = len(list_of_states_in_episode)

        for t in range(0,len(list_of_states_in_episode)):
          target = 0
          for t_dash in
range(t,min(n_step+t,len(list_of_states_in_episode))):
            target = target + gamma**(t_dash-t) *
list_of_step_rewards_in_episode[t_dash+1]
          list_of_targets_in_episode.append(target)
```

```
agent.learn(list_of_states_in_episode,list_of_actions_in_episode,list_
of_targets_in_episode)

        if ep % 10 == 0:
            avg_rew = np.mean(episodes_reward_list[-10:])
            print('Episode ', ep, 'Reward %f' % ep_rew, 'Average
Reward %f' % avg_rew, 'Step count %f' % step_count)

        if ep % 100 and environment_solved == False:
            avg_100 =  np.mean(episodes_reward_list[-100:])
            if avg_100 > env.spec.reward_threshold:
                print('Environment solved at Episode ',ep)
                print('Average Reward for last 100 episodes: ',
avg_100 )
                print('Threshold of the environment: ',
env.spec.reward_threshold)
                #required_episodes_to_solve_list.append(ep-100)
                required_episodes_to_solve_list[run] = ep
                environment_solved = True
                break
    runs_reward_list.append(episodes_reward_list)
    runs_steps_list.append(episodes_steps_list)

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

#Storing variables as pickle files
import pickle

with open('/content/drive/My Drive/RL-PA2/cartpole1-full-return-
hidden-1024-1024-lr-1e4-episodes-1000-runs-steps.pickle', 'wb') as f:
  pickle.dump(runs_steps_list,f)


with open('/content/drive/My Drive/RL-PA2/cartpole1-full-return-
hidden-1024-1024-lr-1e4-episodes-1000-runs-rewards.pickle', 'wb') as
f:
  pickle.dump(runs_reward_list,f)

with open('/content/drive/My Drive/RL-PA2/cartpole1-full-return-
hidden-1024-1024-lr-1e4-episodes-1000-required-episodes.pickle', 'wb')
as f:
  pickle.dump(required_episodes_to_solve_list,f)
```

Mounting Google Drive

```
#Storing variables as pickle files in the Google drive
#Helpful when the environment gets dosconnected and rebooted
#Used in plotting the graphs
```

```
from google.colab import drive
drive.mount('/content/drive')
```

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

```
#Storing variables as pickle files
import pickle

with open('/content/drive/My Drive/RL-PA2/cartpole1-full-return-hidden-64-64-lr-1e4-episodes-1000-runs-steps.pickle', 'wb') as f:
  pickle.dump(runs_steps_list,f)


with open('/content/drive/My Drive/RL-PA2/cartpole1-full-return-hidden-64-64-lr-1e4-episodes-1000-runs-rewards.pickle', 'wb') as f:
  pickle.dump(runs_reward_list,f)

with open('/content/drive/My Drive/RL-PA2/cartpole1-full-return-hidden-64-64-lr-1e4-episodes-1000-required-episodes.pickle', 'wb') as f:
  pickle.dump(required_episodes_to_solve_list,f)
```

```
# Load the pickled variable saved in Drive.
import pickle
with open('/content/drive/My Drive/RL-PA2/cartpole1-full-return-hidden-64-64-lr-1e4-episodes-1000-runs-steps.pickle', 'rb') as f:
  runs_steps_list = pickle.load(f)


with open('/content/drive/My Drive/RL-PA2/cartpole1-full-return-hidden-64-64-lr-1e4-episodes-1000-runs-rewards.pickle', 'rb') as f:
  runs_reward_list = pickle.load(f)

with open('/content/drive/My Drive/RL-PA2/cartpole1-full-return-hidden-64-64-lr-1e4-episodes-1000-required-episodes.pickle', 'rb') as f:
  required_episodes_to_solve_list = pickle.load(f)
```

```
#2nd run
env = gym.make('CartPole-v1')

#Initializing Agent
#agent = Agent1(lr=1e-4, action_size=env.action_space.n)
#Number of episodes
episodes = 1000
runs = 1
gamma = 0.99
#n_step = 8
#tf.compat.v1.reset_default_graph() #Should this be inside the run for
```

```
loop??


runs_reward_list2 = []
runs_steps_list2 = []
required_episodes_to_solve_list2 = [episodes]*runs
#variance_list = []
step_count = 0
begin_time = datetime.datetime.now()

for run in range(runs):
    #Initializing Agent
    print('Starting run no: ', run )
    agent = Agent1(lr=1e-4, action_size=env.action_space.n)
    tf.compat.v1.reset_default_graph()
    episodes_reward_list = []
    episodes_steps_list = []
    environment_solved = False
    for ep in range(1, episodes + 1):
        step_count = 0
        state = env.reset().reshape(1,-1)
        done = False
        ep_rew = 0
        list_of_states_in_episode = []
        list_of_actions_in_episode = []
        list_of_step_rewards_in_episode = []
        list_of_deltas_in_episode = []
        list_of_targets_in_episode = []
        list_of_pi_in_episode = []
        list_of_step_rewards_in_episode.append(None) #Following
convention
        while not done:
            list_of_states_in_episode.append(state)
            action = agent.sample_action(state) ##Sample Action
            #print('before ',action)
            #print('after', action)
            next_state, reward, done, info = env.step(action) ##Take
action
            list_of_actions_in_episode.append(action)
            list_of_step_rewards_in_episode.append(reward)
            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
            step_count += 1
        episodes_reward_list.append(ep_rew)
        episodes_steps_list.append(step_count)

        n_step = len(list_of_states_in_episode)
```

```
        for t in range(0,len(list_of_states_in_episode)):
          target = 0
          for t_dash in
range(t,min(n_step+t,len(list_of_states_in_episode))):
            target = target + gamma**(t_dash-t) *
list_of_step_rewards_in_episode[t_dash+1]
          list_of_targets_in_episode.append(target)

agent.learn(list_of_states_in_episode,list_of_actions_in_episode,list_
of_targets_in_episode)

        if ep % 10 == 0:
            avg_rew = np.mean(episodes_reward_list[-10:])
            print('Episode ', ep, 'Reward %f' % ep_rew, 'Average
Reward %f' % avg_rew, 'Step count %f' % step_count)

        if ep % 100 and environment_solved == False:
            avg_100 =  np.mean(episodes_reward_list[-100:])
            if avg_100 > env.spec.reward_threshold:
                print('Environment solved at Episode ',ep)
                print('Average Reward for last 100 episodes: ',
avg_100 )
                print('Threshold of the environment: ',
env.spec.reward_threshold)
                #required_episodes_to_solve_list.append(ep-100)
                required_episodes_to_solve_list2[run] = ep
                environment_solved = True
                break
    runs_reward_list2.append(episodes_reward_list)
    runs_steps_list2.append(episodes_steps_list)

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


import pickle

with open('/content/drive/My Drive/RL-PA2/cartpole1-full-return-
hidden-64-64-lr-1e4-episodes-1000-runs-steps2.pickle', 'wb') as f:
  pickle.dump(runs_steps_list,f)


with open('/content/drive/My Drive/RL-PA2/cartpole1-full-return-
hidden-64-64-lr-1e4-episodes-1000-runs-rewards2.pickle', 'wb') as f:
  pickle.dump(runs_reward_list,f)

with open('/content/drive/My Drive/RL-PA2/cartpole1-full-return-
hidden-64-64-lr-1e4-episodes-1000-required-episodes2.pickle', 'wb') as
```

```
f:
  pickle.dump(required_episodes_to_solve_list,f)
```

```
print(len(runs_reward_list2))
print(len(runs_reward_list2[0]))
print(required_episodes_to_solve_list2)
print(np.average(required_episodes_to_solve_list2))
```

```
1
1000
[1000]
1000.0
```

```
runs_steps_list.append(runs_reward_list2[0])
runs_reward_list.append(runs_reward_list2[0])
required_episodes_to_solve_list.append(required_episodes_to_solve_list
2[0])
```

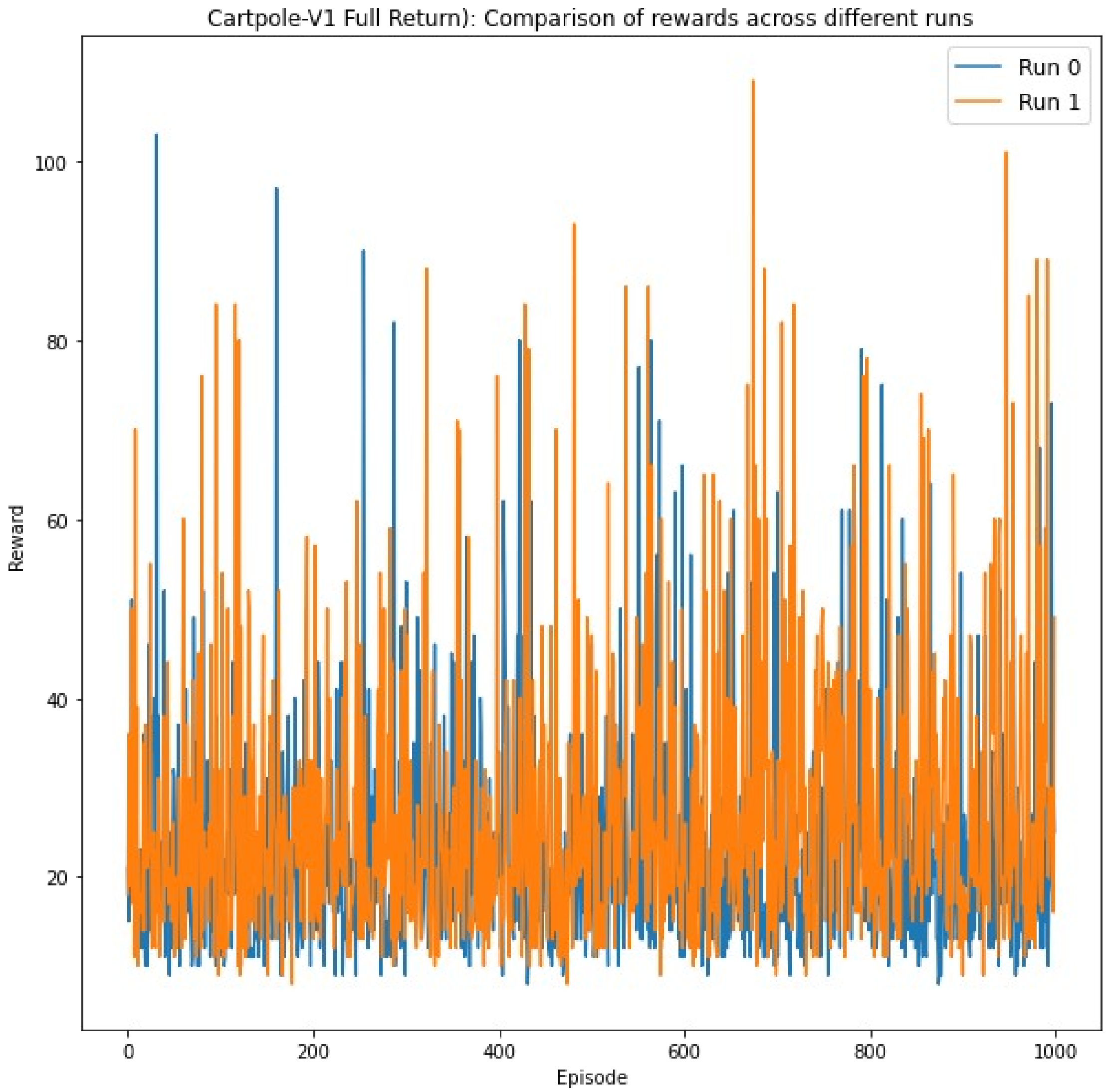
```
print(len(runs_steps_list))
```

```
2
```

Plotting

Plotting reward curves

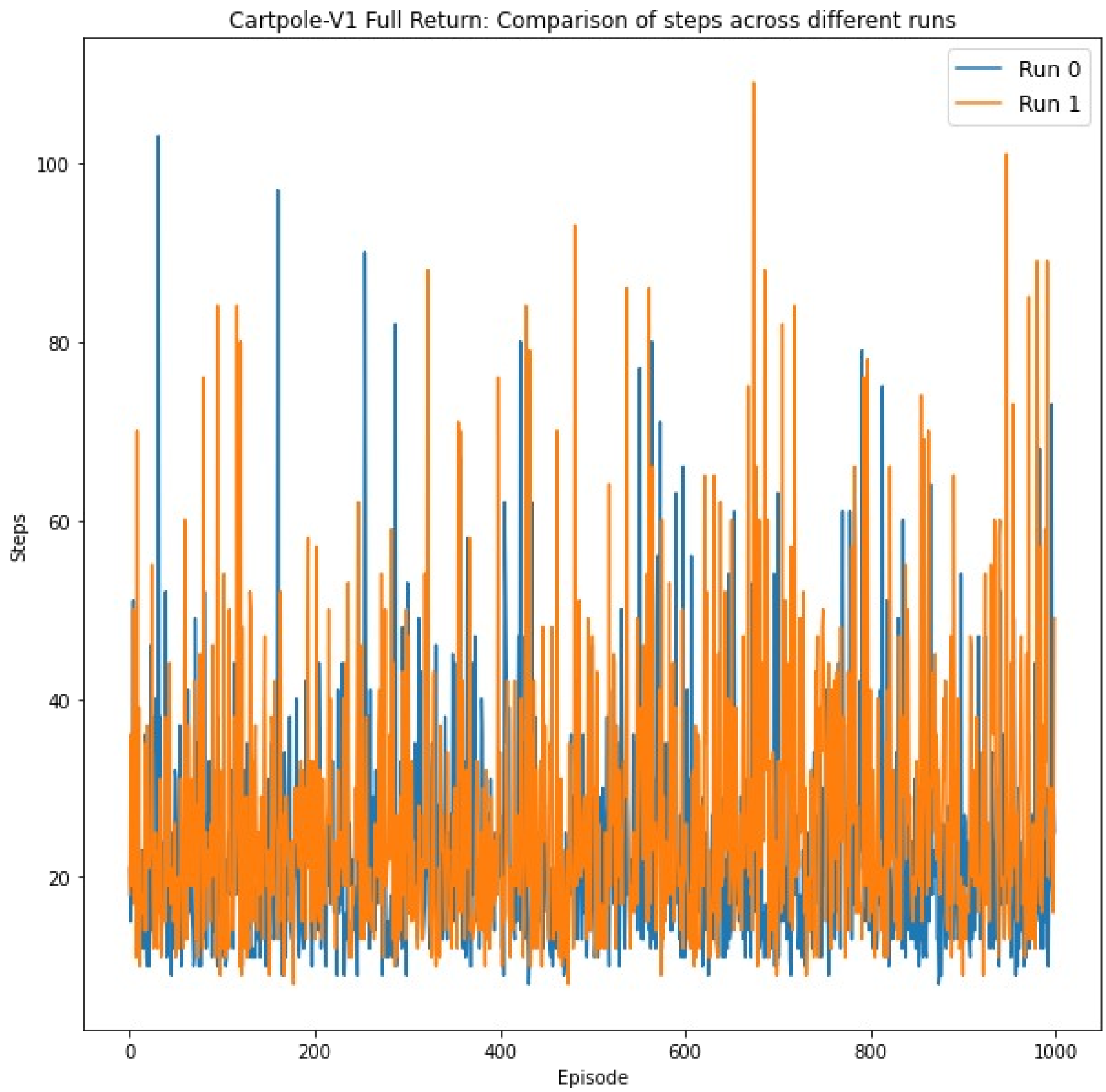
```
# Create subplot
fig, ax = plt.subplots(figsize=(10,10))

x0 = np.arange(len(runs_reward_list[0]))
y0 = runs_reward_list[0]
ax.plot(x0,y0, label='Run 0')

x1 = np.arange(len(runs_reward_list[1]))
y1 = runs_reward_list[1]
ax.plot(x1,y1, label='Run 1')

#x2 = np.arange(len(runs_reward_list[2]))
#y2 = runs_reward_list[2]
#ax.plot(x2,y2, label='Run 2')

legend = ax.legend(loc='upper right', fontsize='large')

plt.xlabel('Episode')
plt.ylabel('Reward')

plt.title('Cartpole-V1 Full Return): Comparison of rewards across 
different runs')

plt.show()
```



Plotting number of steps to reach the goal

```
# Create subplot
fig, ax = plt.subplots(figsize=(10,10))

x0 = np.arange(len(runs_steps_list[0]))
y0 = runs_steps_list[0]
ax.plot(x0,y0, label='Run 0')

x1 = np.arange(len(runs_steps_list[1]))
y1 = runs_steps_list[1]
ax.plot(x1,y1, label='Run 1')

#x2 = np.arange(len(runs_steps_list[2]))
#y2 = runs_steps_list[2]
#ax.plot(x2,y2, label='Run 2')
```

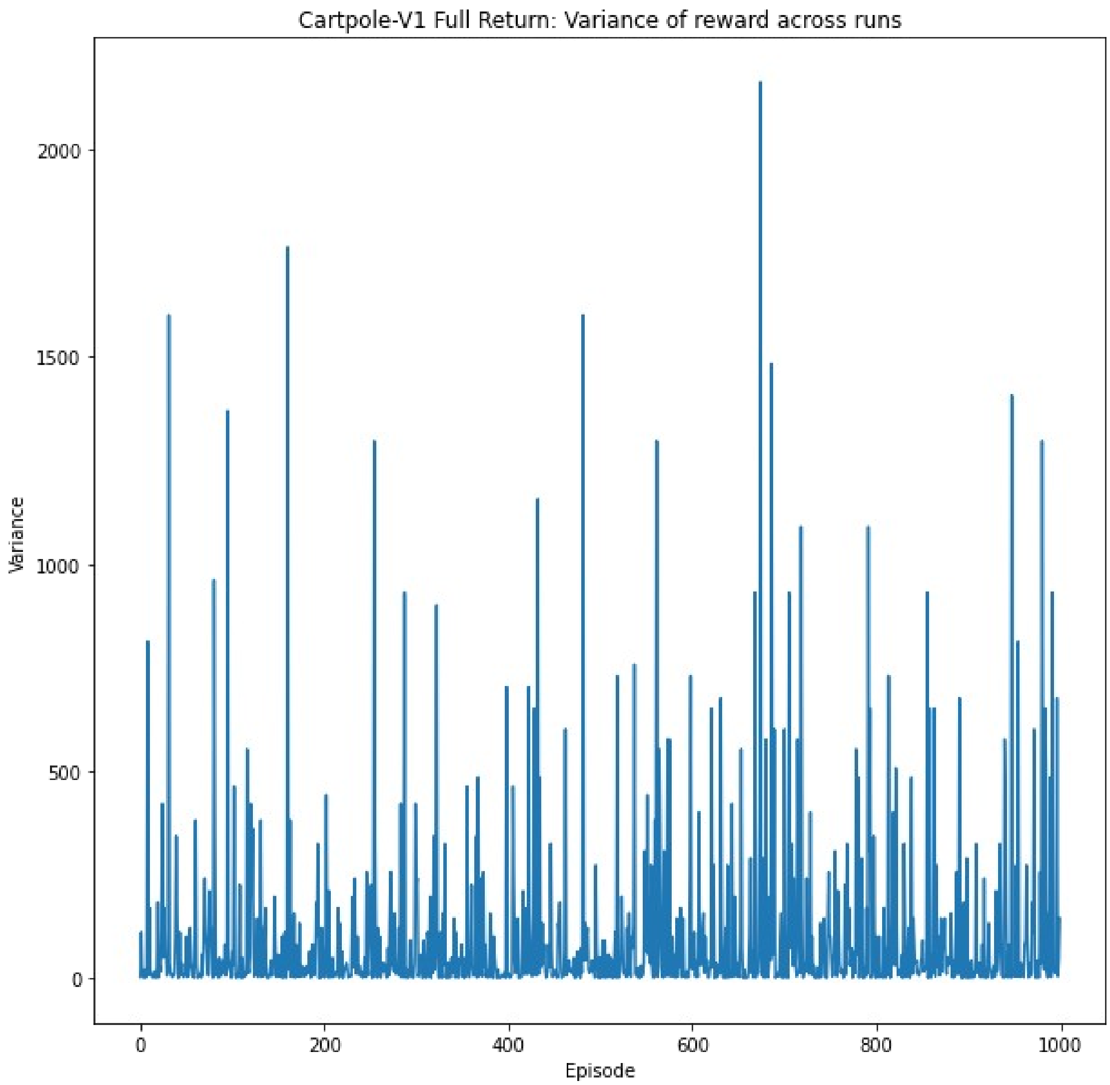
```
legend = ax.legend(loc='upper right', fontsize='large')

plt.xlabel('Episode')
plt.ylabel('Steps')

plt.title('Cartpole-V1 Full Return: Comparison of steps across
different runs')

plt.show()
```



Plotting variance

```
def get_variance(x):
    z = [i for i in x if i is not None]
    #print(z)
    return np.var(z)

import itertools as it
```

```
variance_of_total_episode_reward_across_runs_for_each_episode =
list(map(get_variance, it.zip_longest(*runs_reward_list) ))


plt.figure(figsize=(10,10))
plt.xlabel('Episode')
plt.ylabel('Variance')
plt.plot(np.arange(len(variance_of_total_episode_reward_across_runs_fo
r_each_episode)),
variance_of_total_episode_reward_across_runs_for_each_episode , 0)
plt.title('Cartpole-V1 Full Return: Variance of reward across runs')
plt.show()
```




```
#Acrobot-v1

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

```
#Initializing Agent
#agent = Agent1(lr=1e-4, action_size=env.action_space.n)
#Number of episodes
episodes = 1000
runs = 3
gamma = 0.99
#n_step = 8
#tf.compat.v1.reset_default_graph() #Should this be inside the run for
loop??


runs_reward_list = []
runs_steps_list = []
required_episodes_to_solve_list = [episodes]*runs
#variance_list = []
step_count = 0
begin_time = datetime.datetime.now()

for run in range(runs):
    #Initializing Agent
    print('Starting run no: ', run )
    agent = Agent1(lr=1e-4, action_size=env.action_space.n)
    tf.compat.v1.reset_default_graph()
    episodes_reward_list = []
    episodes_steps_list = []
    environment_solved = False
    for ep in range(1, episodes + 1):
        step_count = 0
        state = env.reset().reshape(1,-1)
        done = False
        ep_rew = 0
        list_of_states_in_episode = []
        list_of_actions_in_episode = []
        list_of_step_rewards_in_episode = []
        list_of_deltas_in_episode = []
        list_of_targets_in_episode = []
        list_of_pi_in_episode = []
        list_of_step_rewards_in_episode.append(None) #Following
convention
        while not done:
            list_of_states_in_episode.append(state)
            action = agent.sample_action(state) ##Sample Action
            #print('before ',action)
            #print('after', action)
            next_state, reward, done, info = env.step(action) ##Take
action
            list_of_actions_in_episode.append(action)
            list_of_step_rewards_in_episode.append(reward)
            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
            step_count += 1
        episodes_reward_list.append(ep_rew)
        episodes_steps_list.append(step_count)

        n_step = len(list_of_states_in_episode)

        for t in range(0,len(list_of_states_in_episode)):
          target = 0
          for t_dash in
range(t,min(n_step+t,len(list_of_states_in_episode))):
            target = target + gamma**(t_dash-t) *
list_of_step_rewards_in_episode[t_dash+1]
          list_of_targets_in_episode.append(target)

agent.learn(list_of_states_in_episode,list_of_actions_in_episode,list_
of_targets_in_episode)

        if ep % 10 == 0:
            avg_rew = np.mean(episodes_reward_list[-10:])
            print('Episode ', ep, 'Reward %f' % ep_rew, 'Average
Reward %f' % avg_rew, 'Step count %f' % step_count)

        if ep % 100 and environment_solved == False:
            avg_100 =  np.mean(episodes_reward_list[-100:])
            if avg_100 > env.spec.reward_threshold:
                print('Environment solved at Episode ',ep)
                print('Average Reward for last 100 episodes: ',
avg_100 )
                print('Threshold of the environment: ',
env.spec.reward_threshold)
                #required_episodes_to_solve_list.append(ep-100)
                required_episodes_to_solve_list[run] = ep
                environment_solved = True
                break
    runs_reward_list.append(episodes_reward_list)
    runs_steps_list.append(episodes_steps_list)

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

#Storing variables as pickle files
import pickle

with open('/content/drive/My Drive/RL-PA2/Acrobot1-full-return-hidden-
1024-1024-lr-1e4-episodes-1000-runs-steps.pickle', 'wb') as f:
  pickle.dump(runs_steps_list,f)
```

```
with open('/content/drive/My Drive/RL-PA2/Acrobot1-full-return-hidden-1024-1024-lr-1e4-episodes-1000-runs-rewards.pickle', 'wb') as f:
  pickle.dump(runs_reward_list,f)

with open('/content/drive/My Drive/RL-PA2/Acrobot1-full-return-hidden-1024-1024-lr-1e4-episodes-1000-required-episodes.pickle', 'wb') as f:
  pickle.dump(required_episodes_to_solve_list,f)
```

```
print(len(runs_reward_list))
print(len(runs_reward_list[0]))
print(required_episodes_to_solve_list)
print(np.average(required_episodes_to_solve_list))
```

Plotting

Plotting Reward Curves

```
# Create subplot
fig, ax = plt.subplots(figsize=(10,10))

x0 = np.arange(len(runs_reward_list[0]))
y0 = runs_reward_list[0]
ax.plot(x0,y0, label='Run 0')

x1 = np.arange(len(runs_reward_list[1]))
y1 = runs_reward_list[1]
ax.plot(x1,y1, label='Run 1')

x2 = np.arange(len(runs_reward_list[2]))
y2 = runs_reward_list[2]
ax.plot(x2,y2, label='Run 2')

legend = ax.legend(loc='upper right', fontsize='large')

plt.xlabel('Episode')
plt.ylabel('Reward')

plt.title('Acrobot-V1 Full Return): Comparison of rewards across different runs')

plt.show()
```

Plotting number of steps to reach the goal

```
# Create subplot
fig, ax = plt.subplots(figsize=(10,10))

x0 = np.arange(len(runs_steps_list[0]))
y0 = runs_steps_list[0]
```

```
ax.plot(x0,y0, label='Run 0')

x1 = np.arange(len(runs_steps_list[1]))
y1 = runs_steps_list[1]
ax.plot(x1,y1, label='Run 1')

x2 = np.arange(len(runs_steps_list[2]))
y2 = runs_steps_list[2]
ax.plot(x2,y2, label='Run 2')

legend = ax.legend(loc='upper right', fontsize='large')

plt.xlabel('Episode')
plt.ylabel('Steps')

plt.title('Acrobot-v1 Full Return: Comparison of steps across
different runs')

plt.show()
```

Plotting variance

```
def get_variance(x):
    z = [i for i in x if i is not None]
    #print(z)
    return np.var(z)

import itertools as it
variance_of_total_episode_reward_across_runs_for_each_episode =
list(map(get_variance, it.zip_longest(*runs_reward_list) ))


plt.figure(figsize=(10,10))
plt.xlabel('Episode')
plt.ylabel('Variance')
plt.plot(np.arange(len(variance_of_total_episode_reward_across_runs_fo
r_each_episode)),
variance_of_total_episode_reward_across_runs_for_each_episode , 0)
plt.title('Acrobot-v1 Full Return: Variance of reward across runs')
plt.show()
```