Assignment-6

In [9]:
```python
import gym
import numpy as np
import tensorflow as tf

# Create and initialize a TensorFlow constant tensor
tensor = tf.constant([[1, 2], [3, 4]])

# Print the tensor
print("Tensor:")
print(tensor)

# Create a TensorFlow variable and initialize it with zeros
variable = tf.Variable(tf.zeros([2, 2]))

# Print the variable
print("\nVariable:")
print(variable)

# Update the variable
variable.assign_add(tf.ones([2, 2]))

# Print the updated variable
print("\nUpdated Variable:")
print(variable)

# Create an environment from OpenAI Gym
env = gym.make('CartPole-v1')

# Initialize the environment
state = env.reset()
```

```python
# Initialize the environment
state = env.reset()

# Run the environment for 100 steps
for _ in range(100):
    # Render the environment
    env.render()

    # Take a random action
    action = env.action_space.sample()

    # Perform the action in the environment
    result = env.step(action)

    # Unpack the first four values
    next_state, reward, done, info = result[:4]

    # If episode is finished, reset the environment
    if done:
        state = env.reset()

# Close the environment
env.close()
```

```
Tensor:
tf.Tensor(
[[1 2]
 [3 4]], shape=(2, 2), dtype=int32)

Variable:
<tf.Variable 'Variable:0' shape=(2, 2) dtype=float32, numpy=
array([[0., 0.],
       [0., 0.]], dtype=float32)>

Updated Variable:
<tf.Variable 'Variable:0' shape=(2, 2) dtype=float32, numpy=
array([[1., 1.],
       [1., 1.]], dtype=float32)>
```

Assignment- 7

```
[3] import os
    # Keep using keras-2 (tf-keras) rather than keras-3 (keras).
    os.environ['TF_USE_LEGACY_KERAS'] = '1'
```

```
from __future__ import absolute_import, division, print_function

import base64
import imageio
import IPython
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import PIL.Image
import pyvirtualdisplay
import reverb

import tensorflow as tf

from tf_agents.agents.dqn import dqn_agent
from tf_agents.drivers import py_driver
from tf_agents.environments import suite_gym
from tf_agents.environments import tf_py_environment
from tf_agents.eval import metric_utils
from tf_agents.metrics import tf_metrics
from tf_agents.networks import sequential
from tf_agents.policies import py_tf_eager_policy
from tf_agents.policies import random_tf_policy
from tf_agents.replay_buffers import reverb_replay_buffer
from tf_agents.replay_buffers import reverb_utils
```

```
[5] # Set up a virtual display for rendering OpenAI gym environments.
    display = pyvirtualdisplay.Display(visible=0, size=(1400, 900)).start()
```

```
[6] tf.version.VERSION

    '2.15.0'
```

```
num_iterations = 20000 # @param {type:"integer"}

initial_collect_steps = 100  # @param {type:"integer"}
collect_steps_per_iteration =   1# @param {type:"integer"}
replay_buffer_max_length = 100000  # @param {type:"integer"}

batch_size = 64  # @param {type:"integer"}
learning_rate = 1e-3  # @param {type:"number"}
log_interval = 200  # @param {type:"integer"}

num_eval_episodes = 10  # @param {type:"integer"}
eval_interval = 1000  # @param {type:"integer"}
```

num_iterations:	20000
initial_collect_steps:	100
collect_steps_per_iteration:	1
replay_buffer_max_length:	100000
batch_size:	64
learning_rate:	1e-3
log_interval:	200
num_eval_episodes:	10
eval_interval:	1000

```
[8] env_name = 'CartPole-v0'
    env = suite_gym.load(env_name)
```

You can render this environment to see how it looks. A free-swinging pole is attached to a cart. The goal is to move the cart right or left in order to keep the pole pointing up.

```
#@test {"skip": true}
env.reset()
PIL.Image.fromarray(env.render())
```

"@test" is not an allowed annotation - allowed values include [@param, @title, @markdown].

```
print('Observation Spec:')
print(env.time_step_spec().observation)

Observation Spec:
BoundedArraySpec(shape=(4,), dtype=dtype('float32'), name='observation', minimum=[-4.8000002e+00 -3.4028235e+38 -4.1887903e-01 -3.4028235e+38], maximum=[4.8000002e+00 3.4028235e+38 4.1
```

```
[11] print('Reward Spec:')
     print(env.time_step_spec().reward)

     Reward Spec:
     ArraySpec(shape=(), dtype=dtype('float32'), name='reward')
```

The `action_spec()` method returns the shape, data types, and allowed values of valid actions.

```
[12] print('Action Spec:')
     print(env.action_spec())

     Action Spec:
     BoundedArraySpec(shape=(), dtype=dtype('int64'), name='action', minimum=0, maximum=1)
```

```
time_step = env.reset()
print('Time step:')
print(time_step)

action = np.array(1, dtype=np.int32)

next_time_step = env.step(action)
print('Next time step:')
print(next_time_step)
```

```
Time step:
TimeStep(
{'step_type': array(0, dtype=int32),
 'reward': array(0., dtype=float32),
 'discount': array(1., dtype=float32),
 'observation': array([-0.04746315,  0.04750493, -0.02811875, -0.02446501], dtype=float32)})
Next time step:
TimeStep(
{'step_type': array(1, dtype=int32),
 'reward': array(1., dtype=float32),
 'discount': array(1., dtype=float32),
 'observation': array([-0.04651305,  0.24301861, -0.02860805, -0.32588542], dtype=float32)})
```

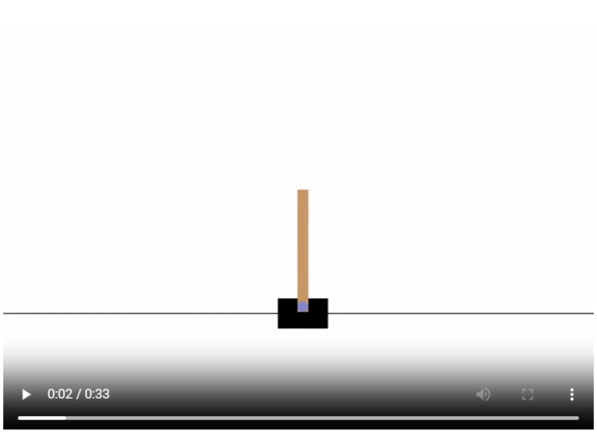
Usually two environments are instantiated: one for training and one for evaluation.

```
[14] train_py_env = suite_gym.load(env_name)
     eval_py_env = suite_gym.load(env_name)
```

```
[16] fc_layer_params = (100, 50)
     action_tensor_spec = tensor_spec.from_spec(env.action_spec())
     num_actions = action_tensor_spec.maximum - action_tensor_spec.minimum + 1

     # Define a helper function to create Dense layers configured with the right
     # activation and kernel initializer.
     def dense_layer(num_units):
       return tf.keras.layers.Dense(
           num_units,
           activation=tf.keras.activations.relu,
           kernel_initializer=tf.keras.initializers.VarianceScaling(
               scale=2.0, mode='fan_in', distribution='truncated_normal'))

     # QNetwork consists of a sequence of Dense layers followed by a dense layer
     # with `num_actions` units to generate one q_value per available action as
     # its output.
     dense_layers = [dense_layer(num_units) for num_units in fc_layer_params]
     q_values_layer = tf.keras.layers.Dense(
         num_actions,
         activation=None,
         kernel_initializer=tf.keras.initializers.RandomUniform(
             minval=-0.03, maxval=0.03),
         bias_initializer=tf.keras.initializers.Constant(-0.2))
     q_net = sequential.Sequential(dense_layers + [q_values_layer])
```

```
create_policy_eval_video(agent.policy, "trained-agent")
```

```
WARNING:root:IMAGEIO FFMPEG_WRITER WARNING: input image is not divisible by macro_block_size=16, resizing from (400, 600) to (400, 608) to ensure video compatibili
```



Assignment-8

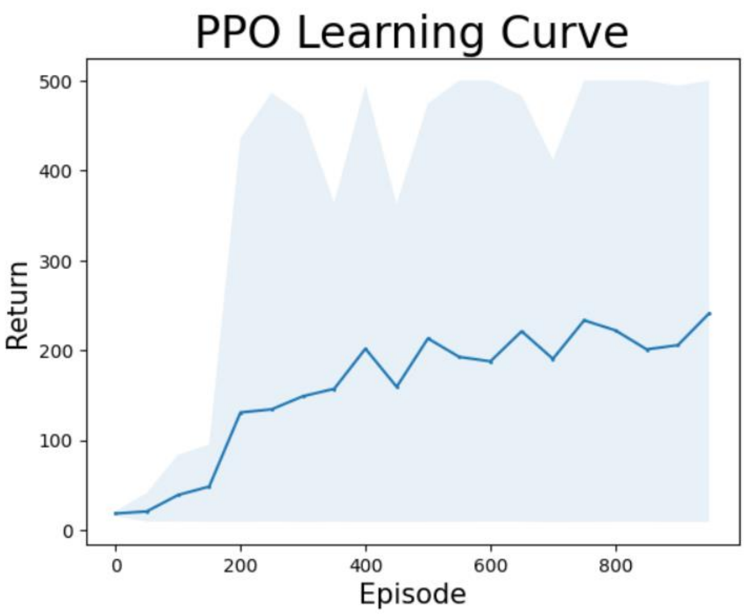
```
# Import packages
import sys
import os

import gymnasium as gym
import numpy as np
import matplotlib.pyplot as plt

import tqdm

import torch
import torch.optim as optim
from torch.autograd import Variable
import torch.nn.functional as F
import torch.nn as nn

from IPython.display import clear_output
from IPython import display

%matplotlib inline

# check and use GPU if available if not use CPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
# CODE from another notebook
class NeuralNet(torch.nn.Module):
    def __init__(self, input_size, output_size, activation, layers=[32,32,16]):
        super().__init__()

        # Define layers with ReLU activation
        self.linear1 = torch.nn.Linear(input_size, layers[0])
        self.activation1 = torch.nn.ReLU()
        self.linear2 = torch.nn.Linear(layers[0], layers[1])
        self.activation2 = torch.nn.ReLU()
        self.linear3 = torch.nn.Linear(layers[1], layers[2])
        self.activation3 = torch.nn.ReLU()

        self.output_layer = torch.nn.Linear(layers[2], output_size)
        self.output_activation = activation

        # Initialization using Xavier normal (a popular technique for initializing weights in NNs)
        torch.nn.init.xavier_normal_(self.linear1.weight)
        torch.nn.init.xavier_normal_(self.linear2.weight)
        torch.nn.init.xavier_normal_(self.linear3.weight)
        torch.nn.init.xavier_normal_(self.output_layer.weight)

    def forward(self, inputs):
        # Forward pass through the layers
        x = self.activation1(self.linear1(inputs))
        x = self.activation2(self.linear2(x))
        x = self.activation3(self.linear3(x))
        x = self.output_activation(self.output_layer(x))
        return x
```

```
def generate_single_episode(env, policy_net):
    """
    Generates an episode by executing the current policy in the given env
    """
    states = []
    actions = []
    rewards = []
    log_probs = []
    max_t = 1000 # max horizon within one episode
    state, _ = env.reset()

    for t in range(max_t):
        state = torch.from_numpy(state).float().unsqueeze(0)
        probs = policy_net.forward(Variable(state)) # get each action choice probability with the current policy network
        action = np.random.choice(env.action_space.n, p=np.squeeze(probs.detach().numpy())) # probablistic
        # action = np.argmax(probs.detach().numpy()) # greedy

        # compute the log_prob to use this in parameter update
        log_prob = torch.log(probs.squeeze(0)[action])

        # append values
        states.append(state)
        actions.append(action)
        log_probs.append(log_prob)

        # take a selected action
        state, reward, terminated, truncated, _ = env.step(action)
        rewards.append(reward)

        if terminated | truncated:
```

```python
# Define parameter values
env_name = 'CartPole-v1'
num_train_ite = 1000
num_seeds = 5 # fit model with 5 different seeds and plot average performance of 5 seeds
num_epochs = 10 # how many times we iterate the entire training dataset passing through the training
eval_freq = 50 # run evaluation of policy at each eval_freq trials
eval_epi_index = num_train_ite//eval_freq # use to create x label for plot
returns = np.zeros((num_seeds, eval_epi_index))
gamma = 0.99 # discount factor
clip_val = 0.2 # hyperparameter epsilon in clip objective

# Create the environment.
env = gym.make(env_name)
nA = env.action_space.n
nS = 4

policy_lr = 5e-4 # policy network's learning rate
baseline_lr = 1e-4

for i in tqdm.tqdm(range(num_seeds)):
    reward_means = []

    # Define policy and value networks
    policy_net = NeuralNet(nS, nA, torch.nn.Softmax())
    policy_net_optimizer = optim.Adam(policy_net.parameters(), lr=policy_lr)
    value_net = NeuralNet(nS, 1, torch.nn.ReLU())
    value_net_optimizer = optim.Adam(value_net.parameters(), lr=baseline_lr)
```

```python
    for m in range(num_train_ite):
        # Train networks with PPO
        policy_net, value_net = train_PPO(env, policy_net, policy_net_optimizer, value_net, value_net_optimizer, num_epochs, clip_val=clip_val, gamma=gamma)
        if m % eval_freq == 0:
            print("Episode: {}".format(m))
            G = np.zeros(20)
            for k in range(20):
                g = evaluate_policy(env, policy_net)
                G[k] = g

            reward_mean = G.mean()
            reward_sd = G.std()
            print("The avg. test reward for episode {0} is {1} with std of {2}.".format(m, reward_mean, reward_sd))
            reward_means.append(reward_mean)
    returns[i] = np.array(reward_means)

# Plot the performance over iterations
x = np.arange(eval_epi_index)*eval_freq
avg_returns = np.mean(returns, axis=0)
max_returns = np.max(returns, axis=0)
min_returns = np.min(returns, axis=0)

plt.fill_between(x, min_returns, max_returns, alpha=0.1)
plt.plot(x, avg_returns, '-o', markersize=1)

plt.xlabel('Episode', fontsize = 15)
plt.ylabel('Return', fontsize = 15)

plt.title("PPO Learning Curve", fontsize = 24)
```



Assignment-9

In [ ]:
```
from mlagents_envs.environment import UnityEnvironment
import numpy as np
from collections import deque

import matplotlib.pyplot as plt
from MADDPG_Agent import MADDPG
import torch

import random
```

In [ ]:
```
env = UnityEnvironment(file_name=r"C:\Users\anike\CL_4\Tennis_Windows_x86_64\Tennis_Windows_x86_64\Tennis.exe")
```

In [ ]:
```
# get the default brain
brain_name = env.brain_names[0]
brain = env.brains[brain_name]
```

In [ ]:
```
# reset the environment
env_info = env.reset(train_mode=True)[brain_name]

# number of agents
num_agents = len(env_info.agents)
print('Number of agents:', num_agents)

# size of each action
action_size = brain.vector_action_space_size
print('Size of each action:', action_size)
```


```
# number of agents
num_agents = len(env_info.agents)
print('Number of agents:', num_agents)

# size of each action
action_size = brain.vector_action_space_size
print('Size of each action:', action_size)
```


```
# examine the state space
states = env_info.vector_observations
state_size = states.shape[1]
print('There are {} agents. Each observes a state with length: {}'.format(states.shape[0], state_size))
print('The state for the first agent looks like:', states[0])
```

In [ ]:
```
MADDPG_Agent = MADDPG(seed=2, noise_start=0.5, update_every=2, gamma=0.99, t_stop_noise=30000)

print(type(MADDPG_Agent))
```

In [ ]:
```
numagents =2

agentslist = np.zeros(numagents)

for i in range(len(agentslist)):
    MADDPG_Agent.agents[i].actor_local.load_state_dict(torch.load('checkpoint_actor_agent_'+str(i)+'.pth'))
    MADDPG_Agent.agents[i].critic_local.load_state_dict(torch.load('checkpoint_critic_agent_'+str(i)+'.pth'))
```

```
env_info = env.reset(train_mode=False)[brain_name] # reset the environment
states = env_info.vector_observations           # get the current state (for each agent)
#scores = np.zeros(num_agents)                   # initialize the score (for each agent)
#scoreslist = []                        # list containing scores from each episode

#for i in range(1, 6):

scores = []
scores_deque = deque(maxlen=100)
scores_avg = []

num_episodes = 5

for i in range(1, num_episodes+1):                                    # play game for 30 episodes

    #env_info = env.reset(train_mode=False)[brain_name] # reset the environment
    #states = env_info.vector_observations           # get the current state (for each agent)
    scorestab = np.zeros(num_agents)                   # initialize the score (for each agent)
    #scoreslist = []                        # list containing scores from each episode

    rewardslist = []
    #env_info = env.reset(train_mode=False)[brain_name]   # reset the environment
    #states = env_info.vector_observations                # get the current state (for each agent)


    while True:
        actions = MADDPG_Agent.act(states,i)# select an action (for each agent)
        actions = np.clip(actions, -1, 1)                 # all actions between -1 and 1
        env_info = env.step(actions)[brain_name]          # send all actions to tne environment
        next_states = env_info.vector_observations        # get next state (for each agent)
        rewards = env_info.rewards                        # get reward (for each agent)
        dones = env_info.local_done                       # see if episode finished
        scorestab += env_info.rewards                        # update the score (for each agent)
```

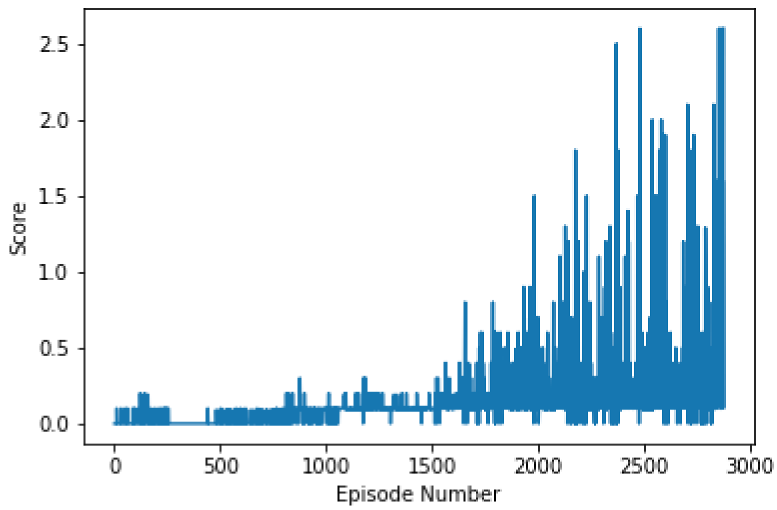
```
    episode_reward = np.max(np.sum(np.array(rewardslist),axis=0))

    scores.append(episode_reward)             # save most recent score to overall score array
    scores_deque.append(episode_reward)       # save most recent score to running window of 100 last scores
    current_avg_score = np.mean(scores_deque)
    scores_avg.append(current_avg_score)      # save average of last 100 scores to average score array

    print('\rEpisode {}\tAverage Score: {:.3f}'.format(i, current_avg_score),end="")

    print("\n")

    print('Score (max over agents) from episode {}: {}'.format(i, np.max(scorestab)))

    print("\n")

    #print('Score (max over agents) from episode {}: {}'.format(i, np.max(scores)))
```

When finished, you can close the environment.

```
env.close()
```

```
# plot the scores
#fig = plt.figure()
#ax = fig.add_subplot(111)
#plt.plot(np.arange(1,len(scoreslist)+1), scoreslist)
#plt.ylabel('Score')
#plt.xlabel('Episode #')
#plt.show()


fig = plt.figure()
ax= fig.add_subplot(111)
plt.plot(np.arange(1,len(scores)+1),scores)
```

```
Episode 2600	Average Score: 0.340

Score (max over agents) from episode 2600: 0.30000000447034836

Episode 2700	Average Score: 0.261

Score (max over agents) from episode 2700: 0.4000000059604645

Episode 2800	Average Score: 0.386

Score (max over agents) from episode 2800: 0.800000011920929

Episode 2877	Average Score: 0.501
Environment solved in 2877 episodes!	Average Score: 0.501
```



Assignment -10

```python
import numpy as np
import tensorflow as tf
import gym

# Define Actor and Critic networks
class ActorCritic(tf.keras.Model):
    def __init__(self, num_actions):
        super(ActorCritic, self).__init__()
        self.dense1 = tf.keras.layers.Dense(64, activation='relu')
        self.policy_logits = tf.keras.layers.Dense(num_actions)
        self.dense2 = tf.keras.layers.Dense(64, activation='relu')
        self.values = tf.keras.layers.Dense(1)

    def call(self, inputs):
        x = self.dense1(inputs)
        logits = self.policy_logits(x)
        v = self.dense2(inputs)
        values = self.values(v)
        return logits, values

# Actor-Critic Agent
class ACAgent:
    def __init__(self, num_actions):
        self.model = ActorCritic(num_actions)
        self.optimizer = tf.keras.optimizers.Adam(learning_rate=0.01)
```

```python
# Actor-Critic Agent
class ACAgent:
    def __init__(self, num_actions):
        self.model = ActorCritic(num_actions)
        self.optimizer = tf.keras.optimizers.Adam(learning_rate=0.01)

    def get_action(self, state):
        logits, _ = self.model(state[np.newaxis])
        action_probs = tf.nn.softmax(logits).numpy()[0]
        action = np.random.choice(len(action_probs), p=action_probs)
        return action

    def train(self, states, actions, rewards, next_states, dones):
        with tf.GradientTape() as tape:
            policy_logits, values = self.model(states)
            next_values = self.model(next_states)[1]
            advantages = rewards + (1 - dones) * 0.99 * next_values - values
            actor_loss = -tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(labels=actions, logits=policy_logits) * advantages)
            critic_loss = tf.reduce_mean(tf.square(advantages))
            total_loss = actor_loss + critic_loss

        grads = tape.gradient(total_loss, self.model.trainable_variables)
        self.optimizer.apply_gradients(zip(grads, self.model.trainable_variables))
```

```python
# Training loop
def train_ac_agent(env, agent, num_episodes):
    for episode in range(num_episodes):
        state = env.reset()
        episode_reward = 0
        done = False

        while not done:
            action = agent.get_action(state)
            next_state, reward, done, _ = env.step(action)
            agent.train(np.array([state]), np.array([action]), np.array([reward]), np.array([next_state]), np.array([done]))
            episode_reward += reward
            state = next_state

        print(f"Episode {episode + 1}: Total Reward = {episode_reward}")

# Create environment and agent
env = gym.make('Acrobot-v1')
num_actions = env.action_space.n
agent = ACAgent(num_actions)

# Train agent
train_ac_agent(env, agent, num_episodes=100)
```

```
/usr/local/lib/python3.10/dist-packages/gym/utils/passive_env_checker.py:241: DeprecationWarning: `np.bool8` is a deprecated alias for `
  if not isinstance(terminated, (bool, np.bool8)):
Episode 1: Total Reward = -500.0
Episode 2: Total Reward = -500.0
Episode 3: Total Reward = -500.0
Episode 4: Total Reward = -500.0
Episode 5: Total Reward = -500.0
Episode 6: Total Reward = -500.0
Episode 7: Total Reward = -500.0
Episode 8: Total Reward = -500.0
Episode 9: Total Reward = -500.0
Episode 10: Total Reward = -500.0
Episode 11: Total Reward = -500.0
Episode 12: Total Reward = -500.0
Episode 13: Total Reward = -500.0
Episode 14: Total Reward = -500.0
Episode 15: Total Reward = -500.0
Episode 16: Total Reward = -500.0
Episode 17: Total Reward = -500.0
Episode 18: Total Reward = -500.0
Episode 19: Total Reward = -500.0
Episode 20: Total Reward = -500.0
Episode 21: Total Reward = -500.0
Episode 22: Total Reward = -500.0
Episode 23: Total Reward = -500.0
```