In [ ]:

```
import gym
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import pybullet as p
import pybullet_data
import time
import random
import cv2
from decimal import Decimal
import os
from IPython.display import clear_output
import tensorflow as tf
tf.compat.v1.disable_eager_execution()
```

In [ ]:

```
# Bicycle and its environment

cv2.destroyAllWindows()

class CycleBalancingEnv(gym.Env):
    metadata = {'render.modes': ['human']}

    def __init__(self):
        # Out cycle has only 1 action spaces i.e. The position of the handlebar
        self.action_space = gym.spaces.box.Box(
            low=-1 * np.ones(1, dtype=np.float32),
            high=1 * np.ones(1, dtype=np.float32))
        # Obervation space
        self.observation_space = gym.spaces.box.Box(
            low=-1 * np.ones(6, dtype=np.float32),
            high=1 * np.ones(6, dtype=np.float32))
        self.np_random, _ = gym.utils.seeding.np_random()

        if not p.isConnected():
            self.client = p.connect(p.GUI)
        else:
            self.client = 1
        #self.client = p.connect(p.SHARED_MEMORY)
        #self.client = p.connect(p.DIRECT)

        self.n_target = 0
        self.min_target_dist = 10
        self.target_span = 100
        self.sphere_dist = 1.5
        self.pole = []
        p.resetSimulation(self.client)
        p.setRealTimeSimulation(0)
        p.setAdditionalSearchPath(pybullet_data.getDataPath())
        self.plane=p.loadURDF("plane.urdf",[0,0,0], useFixedBase=True)
        self.bike = 0
        self.angle_span = 20
        self.n_episodes = 0
        self.rays_distance = 30
        self.z_balance = -0.25
        self.z_target = -1

        self.make_obstacles()
        self.reset()
        #self.show_img()

    # Fuction to show the Distance traveled by rays in an image.
    def show_img(self):
        self.img = np.zeros((800,800,3), dtype='float32')
        shift = 400
```

```
        multiply = 400
        ls = p.getBasePositionAndOrientation(self.bike)
        bike_x = ls[0][0]
        bike_y = ls[0][1]
        handlebar_rotation = p.getEulerFromQuaternion( p.getLinkState(self.bike, 0)[1] )[2]
        mini = 1000
        for deg in range(1, 361, 1):
            mini = min(mini, self.dist[deg-1])
            if deg%self.angle_span==0:
                rad = Decimal( Decimal(deg * np.pi/180 + handlebar_rotation)%Decimal(2*np.pi) + Decimal(2*np.pi))%Decimal(2*np.pi)
                rad = float(rad)
                start = (int(shift + bike_x + self.sphere_dist*np.cos(rad)), int(shift + bike_y + self.sphere_dist*np.sin(rad)))
                end = (int(shift + bike_x + mini*multiply*np.cos(rad)), int(shift + bike_y + mini*multiply*np.sin(rad)))
                cv2.ellipse(self.img, start, (int(mini*multiply),int(mini*multiply)), 0, (rad*180/np.pi)-self.angle_span, (rad*180/np.pi), (0,0,255), -1)
                mini = 1000
        cv2.imshow('img', cv2.rotate(cv2.transpose(self.img), cv2.ROTATE_180))
        cv2.waitKey(1)

    # Step Function which take action as input and performs that action and returns the reward for that action as well as the next observation state
    def step(self, action):
        p.setJointMotorControl2(self.bike, 0, p.POSITION_CONTROL, targetPosition=action[0], maxVelocity=5) # Apply Position control to Handlebar
        for i in range(3):
            p.setJointMotorControl2(self.bike,1,p.TORQUE_CONTROL , force=(2.5+0)*10000) # Apply Toruqe to Back Wheel
            p.setJointMotorControl2(self.bike,2,p.TORQUE_CONTROL , force=(2.5+0)*10000) # Apply Toruqe to Front Wheel
            ls = p.getBasePositionAndOrientation(self.bike) # ls[0]=Postion of cycle, ls[1] = Orientation of cycle
            val = p.getEulerFromQuaternion(ls[1])[0] - 1.57 # Calculating inclination of cycle from vertical
            p.applyExternalTorque(self.bike, -1, [-1000000*val, 0, 0], flags=p.WORLD_FRAME)
            p.stepSimulation()

        ls = p.getBasePositionAndOrientation(self.bike) # ls[0]=Postion of cycle, ls[1] = Orientation of cycle
        val = p.getEulerFromQuaternion(ls[1])[0] - 1.57 # Calculating inclination of cycle from vertical
        z = ls[0][2] + self.z_balance

        tmp = ls[0]
        self.bike_x = tmp[0]
        self.bike_y = tmp[1]

        obs = [] # Observation Space
        obs.append(np.arctan( (self.target_x-self.bike_x)/(self.target_y-self.bike_y) ))
        ls = p.getBasePositionAndOrientation(self.bike)
        obs += p.getEulerFromQuaternion(ls[1])
        obs.append((ls[0][0] - self.target_x)/self.target_span)
        obs.append((ls[0][1] - self.target_y)/self.target_span)

        bike_x = ls[0][0]
        bike_y = ls[0][1]
        reward_2 = 0
        #cnt = 0
        ray_from = []
        ray_to = []
        handlebar_rotation = p.getEulerFromQuaternion( p.getLinkState(self.bike, 0)[1] )[2]

        self.time += 1 # Adding 1 to the time for which the current episode has been running

        # Terminating the episode if the cycle covers less than 1 units distance in 200 timesteps
```

```
        dist_2 = np.sqrt((self.bike_x)**2 + abs(self.bike_y)**2)
        reward_3 = 0
        if dist_2 > (np.sqrt(self.target_x**2 + self.target_y**2) + 10):
            self.done = True
            reward_3 = -500
            print("Outside Range!")
        if self.time%10==0 and dist_2>self.distance: self.distance = dist_2

        if self.time>999:
            reward_3 = -500

        value = p.getEulerFromQuaternion(p.getLinkState(self.bike, 0)[1])[2] - p.getEule
rFromQuaternion(ls[1])[2]
        if value<-1: value += 2*np.pi
        if value>1: value -= 2*np.pi
        #print(value)
        if value < -0.5:
            self.left += 0.1
            self.right = 0
        elif value > 0.5:
            self.left = 0
            self.right += 0.1
        else:
            self.left = 0
            self.right = 0

        self.neg_reward = 0
        if self.left>10 or self.right>10:
            print("Slow!!!")
            self.neg_reward = -700
            self.done = True

        val = self.target_span
        reward_1 = 0
        dist_3 = np.sqrt( (self.bike_x - self.target_x)**2 + (self.bike_y - self.target_
y)**2 )
        if dist_3 < self.target_distance:
            reward_1 = 100 + self.target_reward
            self.target_distance -= 5
            self.target_reward = min(500, self.target_reward*2)

        self.completed = 0
        if dist_3 < 10:
            reward_1 = 500
            self.completed = 1
            self.done = True
            print("DONE!")
            self.make_obstacles()

        # Calculating the total reward
        reward = reward_1 - abs(ls[0][0] - self.target_x)/10. - abs(ls[0][1] - self.targ
et_y)/10. + self.neg_reward - self.left - self.right + reward_3
        #print(mini, end=" ")

        if self.done:
            print(self.left, self.right, dist_3, self.target_distance)

        obs = np.array(obs, dtype=np.float32)

#         if self.time%10==0 and not self.done:
#             self.show_img()

        return obs, reward/100, self.done, dict()


    def reset(self):

        self.n_episodes += 1
        if self.n_episodes==20:
            self.make_obstacles()
            self.n_episodes = 0
```

```
        if self.bike!=0:
            p.removeBody(self.bike)
        # Loading the cycle
        self.bike_x = 0 # random.randint(-5, 5) # X position of the cycle
        self.bike_y = 0 # random.randint(-5, 5) # Y position of the cycle
        self.bike=p.loadURDF("bike_2.urdf.xml",[self.bike_x, self.bike_y,0], p.getQuater
nionFromEuler([0,0, random.random()*2*np.pi]),  useFixedBase=False)

        for i in range(10):
            p.stepSimulation()


        # Adding friction and other dynamics
        p.changeDynamics(self.plane, -1, lateralFriction=5, angularDamping=1)
        p.changeDynamics(self.bike, 1, mass=100)
        p.changeDynamics(self.bike, -1, lateralFriction=5, angularDamping=1)

        p.setGravity(0, 0, -250) # Setting the gravity
        #p.setRealTimeSimulation(0)
        self.done=False
        self.time = 0
        self.distance = np.sqrt((self.bike_x)**2 + abs(self.bike_y)**2)
        self.neg_reward = 0

        obs = []
        obs.append( np.arctan( (self.target_x-self.bike_x)/(self.target_y-self.bike_y) )
)
        ls = p.getBasePositionAndOrientation(self.bike)
        obs += p.getEulerFromQuaternion(ls[1])
        obs.append((ls[0][0] - self.target_x)/self.target_span)
        obs.append((ls[0][1] - self.target_y)/self.target_span)

        self.cnt = 0
        self.left = 0
        self.right = 0
        self.target_distance = (np.sqrt( (self.bike_x - self.target_x)**2 + (self.bike_y
- self.target_y)**2))//10 * 10
        self.target_reward = 128
        self.completed = 0

        obs = np.array(obs, dtype=np.float32)
        return obs

    def make_obstacles(self):

        p.resetSimulation(self.client)
        p.setRealTimeSimulation(0)
        p.setAdditionalSearchPath(pybullet_data.getDataPath())
        #self.plane=p.loadURDF("plane.urdf",[0,0,0], useFixedBase=True)
        mul = 100
        height = 2
        visualShift = [0, 0, 0]
        shift = [0, 0, 0]
        meshScale=[0.1*mul, 0.1*mul, 0.1*mul*height]
        path = 'C:/Users/User/Documents/GitHub/bullet3/examples/pybullet/gym/pybullet_dat
a/'
        groundColId = p.createCollisionShape(shapeType=p.GEOM_MESH,
                                             fileName=os.path.join(path, "terrain.
obj"),
                                             collisionFramePosition=shift,
                                             meshScale=meshScale,
                                             flags=p.GEOM_FORCE_CONCAVE_TRIMESH)
        groundVisID = p.createVisualShape(shapeType=p.GEOM_MESH,
                                          fileName=os.path.join(path, "terrain.obj"),
                                          rgbaColor=[0.7,0.3,0.1,1],
                                          specularColor=[0.4,.4,0],
                                          visualFramePosition=visualShift,
                                          meshScale=meshScale)
        self.plane = p.createMultiBody(baseMass=0,
                                       baseInertialFramePosition=[0,0,0],
                                       baseCollisionShapeIndex=groundColId,
                                       baseVisualShapeIndex=groundVisID,
```

```
                                              basePosition=[0,0,0],
                                              useMaximalCoordinates=True)
        self.bike = 0

        self.bike_x = 0
        self.bike_y = 0
        self.pole = []
        for i in range(self.n_target):
            target_x = self.bike_x
            target_y = self.bike_y
            while (np.sqrt( (self.bike_x - target_x)**2 + (self.bike_y - target_y)**2 ))
< self.min_target_dist:
                target_x = random.randint(int(self.bike_x) - self.target_span, int(self.
bike_x) + self.target_span)
                target_y = random.randint(int(self.bike_y) - self.target_span, int(self.
bike_y) + self.target_span)
            self.pole.append( p.loadURDF("C:/Users/User/Documents/GitHub/bullet3/example
s/pybullet/gym/pybullet_data/cube.urdf",[target_x, target_y, 4], [0,0,0,1], useFixedBase
=True, globalScaling=1.0) )
            p.changeDynamics(self.pole[i], -1, mass=1000)

        # Loading the target
        self.pole = []
        min_target_range = 90
        for i in range(1):
            self.target_x = 0
            self.target_y = 0
            while (np.sqrt( (self.bike_x - self.target_x)**2 + (self.bike_y - self.targe
t_y)**2 )) < 90:
                self.target_x = random.randint(int(self.bike_x) - self.target_span, int(
self.bike_x) + self.target_span)
                self.target_y = random.randint(int(self.bike_y) - self.target_span, int(
self.bike_y) + self.target_span)
            self.pole.append( p.loadURDF("C:/Users/User/Documents/GitHub/bullet3/example
s/pybullet/gym/pybullet_data/cube.urdf",[self.target_x, self.target_y, 2], [0,0,0,1], us
eFixedBase=True, globalScaling=5.0) )
        self.target_distance = (np.sqrt( (self.bike_x - self.target_x)**2 + (self.bike_y
- self.target_y)**2))//10 * 10
        self.target_reward = 128

    def make_sphere(self):
        for i in self.sphere:
            p.removeBody(i)

        ls = p.getBasePositionAndOrientation(self.bike)
        z = ls[0][2] + self.z_balance
        bike_x = ls[0][0]
        bike_y = ls[0][1]
        self.sphere = []
        handlebar_rotation = p.getEulerFromQuaternion( p.getLinkState(self.bike, 0)[1] )
[2]
        for deg in range(1, 361, 10):
            rad = Decimal( Decimal(deg * np.pi/180 + handlebar_rotation)%Decimal(2*np.pi
) + Decimal(2*np.pi))%Decimal(2*np.pi)
            rad = float(rad)
            #self.sphere.append(p.loadURDF('sphere_small.urdf', [bike_x + self.sphere_di
st*np.cos(rad), bike_y + self.sphere_dist*np.sin(rad), z], [0,0,0,1]))
            #p.loadURDF('sphere_small.urdf', [bike_x + (self.sphere_dist+1*rad)*np.cos(r
ad), bike_y + (self.sphere_dist+1*rad)*np.sin(rad), 1], [0,0,0,1], useFixedBase=True, glo
balScaling=deg/10)
            self.sphere.append(p.loadURDF('sphere_small.urdf', [bike_x + self.rays_dista
nce*np.cos(rad), bike_y + self.rays_distance*np.sin(rad), z+(abs(deg-180)*(-self.rays_dis
tance/90.)+self.rays_distance)*np.tan(p.getEulerFromQuaternion(ls[1])[1])], [0,0,0,1], u
seFixedBase=True))
        print(ls[1], p.getEulerFromQuaternion(ls[1]))

    def render(self, mode='human'):

        #p.stepSimulation()

        distance=5
        yaw = 0
```

```
        humanPos, humanOrn = p.getBasePositionAndOrientation(self.bike)
        humanBaseVel = p.getBaseVelocity(self.bike)
        #print("frame",frame, "humanPos=",humanPos, "humanVel=",humanBaseVel)
        camInfo = p.getDebugVisualizerCamera()
        curTargetPos = camInfo[11]
        distance=camInfo[10]
        yaw = camInfo[8]
        pitch=camInfo[9]
        targetPos = [0.95*curTargetPos[0]+0.05*humanPos[0],0.95*curTargetPos[1]+0.05*hum
anPos[1],curTargetPos[2]]

        p.resetDebugVisualizerCamera(distance,270 ,pitch,targetPos)

    def close(self):
        p.disconnect(self.client)

    def seed(self, seed=None):
        self.np_random, seed = gym.utils.seeding.np_random(seed)
        return [seed]
```

In [ ]:

```
env = CycleBalancingEnv()
env.reset().shape
```

In [ ]:

```
print(env.observation_space.sample().shape)
print(env.action_space.sample())
```

In [ ]:

```
episodes = 1
for episode in range(1, episodes+1):
    state = env.reset()
    done = False
    score = 0

    while not done:
        env.render()
        action = env.action_space.sample()
        action = [0]
        state, reward, done, info = env.step(action)
        score+=reward
        time.sleep(1/24.)
        print(state)
        clear_output(wait=True)
    print('Episode:{} Score:{}'.format(episode, score))
#env.close()
```

In [ ]:

```
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import *
from tensorflow.keras.optimizers import Adam
import tensorflow as tf
```

In [ ]:

```
states = env.observation_space.shape # Shape of our observation space
nb_actions = env.action_space.shape[0] # shape of our action space
states, nb_actions
```

In [ ]:

```
del actor, critic
```

In [ ]:

```
# Defining our actor model for the DDPG algorithm
```

```
# Defining our actor model for the DDPG algorithm

actor = Sequential()
actor.add(Flatten(input_shape=(1,) + env.observation_space.shape))
# actor.add(LSTM(32, input_shape=(8,) + env.observation_space.shape))
# actor.add(Flatten())
actor.add(Dense(32, kernel_initializer='he_uniform'))
actor.add(Activation('relu'))
actor.add(Dense(32, kernel_initializer='he_uniform'))
actor.add(Activation('relu'))
actor.add(Dense(32, kernel_initializer='he_uniform'))
actor.add(Activation('relu'))
# actor.add(Reshape((1, -1)))
# actor.add(LSTM(32))
actor.add(Dense(nb_actions))
actor.add(Activation('tanh'))
print(actor.summary())
```

In [ ]:

```
# Defining our critic network for the DDPG algorithm

action_input = Input(shape=(nb_actions,), name='action_input')
observation_input = tf.keras.Input(shape=(1,) + env.observation_space.shape, name='obser
vation_input')
flattened_observation = Flatten()(observation_input)
x = Concatenate()([action_input, flattened_observation])
x = Dense(32, kernel_initializer='he_uniform')(x)
x = Activation('relu')(x)
x = Dense(32, kernel_initializer='he_uniform')(x)
x = Activation('relu')(x)
x = Dense(32, kernel_initializer='he_uniform')(x)
x = Activation('relu')(x)
x = Dense(1)(x)
x = Activation('linear')(x)
critic = tf.keras.Model(inputs=[action_input, observation_input], outputs=x)
print(critic.summary())
```

In [ ]:

```
from rl.agents import DQNAgent, SARSAAgent, DDPGAgent
#from rl.agents.sarsa import SARSAAgent
from rl.policy import BoltzmannQPolicy, BoltzmannGumbelQPolicy, SoftmaxPolicy,  EpsGreedy
QPolicy, GreedyQPolicy, BoltzmannGumbelQPolicy
from rl.memory import SequentialMemory
from rl.random import OrnsteinUhlenbeckProcess
from rl.util import *
```

In [ ]:

```
episode_reward = []
```

In [ ]:

```
# Defining our DDPG agent

memory = SequentialMemory(limit=100000, window_length=1)
random_process = OrnsteinUhlenbeckProcess(size=nb_actions, theta= 0.1, mu=0, sigma=.2)
agent = DDPGAgent(nb_actions=nb_actions, actor=actor, critic=critic, critic_action_input
=action_input,
                  memory=memory, nb_steps_warmup_critic=20, nb_steps_warmup_actor=20,
                  random_process=random_process, gamma=0.99, target_model_update=1e-3)
```

In [ ]:

```
agent.compile([Adam(lr=.00001, clipnorm=1.0), Adam(lr=.001, clipnorm=1.0)], metrics=['ma
e'])
```

In [ ]:

```
history = agent.fit(env, nb_steps=10000, visualize=True, verbose=2, nb_max_episode_steps
```

```
=1000)
episode_reward += history.history['episode_reward']
```

In [ ]:

```
plt.plot(episode_reward)
```

In [ ]:

```
avg_reward = []
sum_reward = 0
span = 100
for i in range(len(episode_reward)):
    if i>=span: sum_reward -= episode_reward[i-span]
    sum_reward += episode_reward[i]
    if i>=span: avg_reward.append(sum_reward/span)
plt.plot(avg_reward)
```

In [ ]:

```
avg_reward = []
sum_reward = 0
span = 50
for i in range(len(episode_reward)):
    if i>=span: sum_reward -= episode_reward[i-span]
    sum_reward += episode_reward[i]
    if i>=span: avg_reward.append(sum_reward/span)
plt.plot(avg_reward)
```

In [ ]:

```
avg_reward = []
sum_reward = 0
span = 10
for i in range(len(episode_reward)):
    if i>=span: sum_reward -= episode_reward[i-span]
    sum_reward += episode_reward[i]
    if i>=span: avg_reward.append(sum_reward/span)
plt.plot(avg_reward)
```

In [ ]:

```
# agent.save_weights('ddpg_{}_weights.h5f'.format('32_3_rays_final'), overwrite=True)
# actor.save_weights('actor_32_3_rays_final.h5', overwrite=True)
# critic.save_weights('critic_32_3_rays_final.h5', overwrite=True)
```

In [ ]:

```
env.make_obstacles()
env.reset()
```

In [ ]:

```
#time.sleep(10.)
_ = agent.test(env, nb_episodes=10, visualize=True) #, nb_max_episode_steps=1000)
```

In [ ]:

```
# actor.save_weights('actor_32_3_rays_final.h5', overwrite=True)
# critic.save_weights('critic_32_3_rays_final.h5', overwrite=True)
```

In [ ]:

```
actor.load_weights('actor_32_3_rays_final.h5')
critic.load_weights('critic_32_3_rays_final.h5')
```

In [ ]:

```
agent.load_weights('ddpg_{}_weights.h5f'.format('32_3_rays_final'))
```