Continuous_Control

July 21, 2021

1 Continuous Control

In this notebook, you will learn how to use the Unity ML-Agents environment for the second project of the Deep Reinforcement Learning Nanodegree program.

1.1 Start the Environment

We begin by importing the necessary packages. If the code cell below returns an error, please revisit the project instructions to double-check that you have installed Unity ML-Agents and NumPy.

[1]:
```
from unityagents import UnityEnvironment
import numpy as np
import copy
import random
from collections import deque, namedtuple
from tqdm import trange

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

from platform import python_version
print('Python version:', python_version())
print('Torch  version:', torch.__version__)
```

```
Python version: 3.6.3
Torch  version: 1.8.1
```

Next, we will start the environment! Before running the code cell below, change the `file_name` parameter to match the location of the Unity environment that you downloaded.

- Mac: "path/to/Reacher.app"
- Windows (x86): "path/to/Reacher_Windows_x86/Reacher.exe"
- Windows (x86_64): "path/to/Reacher_Windows_x86_64/Reacher.exe"
- Linux (x86): "path/to/Reacher_Linux/Reacher.x86"
- Linux (x86_64): "path/to/Reacher_Linux/Reacher.x86_64"
- Linux (x86, headless): "path/to/Reacher_Linux_NoVis/Reacher.x86"
- Linux (x86_64, headless): "path/to/Reacher_Linux_NoVis/Reacher.x86_64"

For instance, if you are using a Mac, then you downloaded Reacher.app. If this file is in the same folder as the notebook, then the line below should appear as follows:

```
env = UnityEnvironment(file_name="Reacher.app")
```

[2]:
```python
env = UnityEnvironment(file_name='Reacher.app')
```

```
INFO:unityagents:
'Academy' started successfully!
Unity Academy name: Academy
        Number of Brains: 1
        Number of External Brains : 1
        Lesson number : 0
        Reset Parameters :
                goal_speed -> 1.0
                goal_size -> 5.0
Unity brain name: ReacherBrain
        Number of Visual Observations (per agent): 0
        Vector Observation space type: continuous
        Vector Observation space size (per agent): 33
        Number of stacked Vector Observation: 1
        Vector Action space type: continuous
        Vector Action space size (per agent): 4
        Vector Action descriptions: , , ,
```

Environments contain brains which are responsible for deciding the actions of their associated agents. Here we check for the first brain available, and set it as the default brain we will be controlling from Python.

[3]:
```python
# get the default brain
brain_name = env.brain_names[0]
brain = env.brains[brain_name]
```

1.2 Examine the State and Action Spaces

In this environment, a double-jointed arm can move to target locations. A reward of +0.1 is provided for each step that the agent's hand is in the goal location. Thus, the goal of your agent is to maintain its position at the target location for as many time steps as possible.

The observation space consists of 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm. Each action is a vector with four numbers, corresponding to torque applicable to two joints. Every entry in the action vector must be a number between -1 and 1.

Run the code cell below to print some information about the environment.

[4]:
```
# 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)

# 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])
```

```
Number of agents: 1
Size of each action: 4
There are 1 agents. Each observes a state with length: 33
The state for the first agent looks like: [ 0.00000000e+00 -4.00000000e+00
0.00000000e+00  1.00000000e+00
 -0.00000000e+00 -0.00000000e+00 -4.37113883e-08  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00 -1.00000000e+01  0.00000000e+00
  1.00000000e+00 -0.00000000e+00 -0.00000000e+00 -4.37113883e-08
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  5.75471878e+00 -1.00000000e+00
  5.55726671e+00  0.00000000e+00  1.00000000e+00  0.00000000e+00
 -1.68164849e-01]
```

1.3 Take Random Actions in the Environment

In the next code cell, you will learn how to use the Python API to control the agent and receive feedback from the environment.

Once this cell is executed, you will watch the agent's performance, if it selects an action at random with each time step. A window should pop up that allows you to observe the agent, as it moves through the environment.

Of course, as part of the project, you'll have to change the code so that the agent is able to use its experience to gradually choose better actions when interacting with the environment!

[5]:
```python
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)
while True:
    actions = np.random.randn(num_agents,
                              action_size)        # select an action
                                                   # (for each agent)
    actions = np.clip(actions, -1, 1)             # all actions btw (-1,1)
    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
    scores += env_info.rewards                    # update the score
                                                   # (for each agent)
    states = next_states                          # roll over states
                                                   # to next time step
    if np.any(dones):                             # exit loop
        break                                     # if episode finished
print('Total score (averaged over agents) this episode: {}'\
      .format(np.mean(scores)))
```

```
Total score (averaged over agents) this episode: 0.19999999552965164
```

1.4 Define a neural network architecture

Now it's your turn to train your own agent to solve the environment! When training the environment, set `train_mode=True`, so that the line for resetting the environment looks like the following:

```python
env_info = env.reset(train_mode=True)[brain_name]
```

1.4.1 Actor (Policy) Model

Maps states to action values. The actor network has the following structure:

- fully connected layer (state size x 128)
- SELU-activation
- fully connected layer (128 x 64)
- SELU-activation
- fully connected layer (64 x 32)
- SELU-activation
- fully connected layer (32 x action size)

[6]:

```
def hidden_init(layer):
    '''Hidden layers initialization'''
    fan_in = layer.weight.data.size()[0]
    lim = 1. / np.sqrt(fan_in)
    return (-lim, lim)
```

[7]:

```
class Actor(nn.Module):

    def __init__(self, state_size, action_size, seed,
                 fc1_units = 128, fc2_units = 64, fc3_units = 32):
        '''
        Neural network parameters:
            state_size (int) : # parameters characterizing the environment state
            action_size (int): # possible actions
            seed (int)       : random seed
            fc1_units (int)  : # nodes in the first hidden layer
            fc2_units (int)  : # nodes in the second hidden layer
            fc3_units (int)  : # nodes in the third hidden layer
        '''
        super(Actor, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1  = nn.Linear(state_size, fc1_units)
        self.fc2  = nn.Linear(fc1_units,  fc2_units)
        self.fc3  = nn.Linear(fc2_units,  fc3_units)
        self.fc4  = nn.Linear(fc3_units,  action_size)

        self.selu1 = nn.SELU(fc1_units)
        self.selu2 = nn.SELU(fc2_units)
        self.selu3 = nn.SELU(fc3_units)

        self.reset_parameters()

    def reset_parameters(self):
        self.fc1.weight.data.uniform_(*hidden_init(self.fc1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(*hidden_init(self.fc3))
        self.fc4.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state):
        x = self.selu1(self.fc1(state))
        x = self.selu2(self.fc2(x))
        x = self.selu3(self.fc3(x))
        return torch.tanh(self.fc4(x))
```

1.4.2 Critic (Value) Model

Maps (state, action) pairs to Q-values. The critic network has the following structure: * fully connected layer (state size x 128) * SELU-activation * concatenate the action * fully connected layer (128 + action size x 64) * SELU-activation * fully connected layer (64 x 32) * SELU-activation * fully connected layer (32 x 1)

[8]:
```
class Critic(nn.Module):
    def __init__(self, state_size, action_size, seed,
                 fcs1_units = 128, fc2_units = 64, fc3_units = 32):
        '''
        Neural network parameters:
            state_size (int) : # parameters characterizing the environment state
            action_size (int): # possible actions
            seed (int)       : random seed
            fc1_units (int)  : # nodes in the first hidden layer
            fc2_units (int)  : # nodes in the second hidden layer
            fc3_units (int)  : # nodes in the third hidden layer
        '''
        super(Critic, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fcs1 = nn.Linear(state_size, fcs1_units)
        self.fc2  = nn.Linear(fcs1_units+action_size, fc2_units)
        self.fc3  = nn.Linear(fc2_units, fc3_units)
        self.fc4  = nn.Linear(fc3_units, 1)
        self.selu1 = nn.SELU(fcs1_units)
        self.selu2 = nn.SELU(fc2_units)
        self.selu3 = nn.SELU(fc3_units)
        self.reset_parameters()

    def reset_parameters(self):
        self.fcs1.weight.data.uniform_(*hidden_init(self.fcs1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(*hidden_init(self.fc3))
        self.fc4.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state, action):
        xs = self.selu1(self.fcs1(state))
        x = torch.cat((xs, action), dim=1)
        x = self.selu2(self.fc2(x))
        x = self.selu3(self.fc3(x))
        return self.fc4(x)
```

1.5 Define the Agent

1.5.1 Fixed-size buffer to store experience tuples

[9]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
display(device)
```

```
device(type='cpu')
```

[10]:
```
class ReplayBuffer:
    '''Fixed-size buffer to store experience tuples'''

    def __init__(self, action_size, buffer_size, batch_size, seed):
        '''Initialize a ReplayBuffer object
        Params
        ======
            buffer_size (int): maximum size of buffer
            batch_size (int) : size of each training batch
        '''
        self.action_size = action_size
        self.memory = deque(maxlen=buffer_size)  # internal memory (deque)
        self.batch_size = batch_size
        self.experience = namedtuple("Experience",
                                     field_names=["state", "action", "reward",
                                                  "next_state", "done"])
        self.seed = random.seed(seed)

    def add(self, state, action, reward, next_state, done):
        '''Add a new experience to memory'''
        e = self.experience(state, action, reward, next_state, done)
        self.memory.append(e)

    def sample(self):
        '''Randomly sample a batch of experiences from memory'''
        experiences = random.sample(self.memory, k=self.batch_size)

        states = torch.from_numpy(
            np.vstack([e.state for e in experiences if e is not None]))\
            .float().to(device)
        actions = torch.from_numpy(
            np.vstack([e.action for e in experiences if e is not None]))\
            .float().to(device)
        rewards = torch.from_numpy(
            np.vstack([e.reward for e in experiences if e is not None]))\
            .float().to(device)
        next_states = torch.from_numpy(
            np.vstack([e.next_state for e in experiences if e is not None]))\
            .float().to(device)
```

```
        dones = torch.from_numpy(
            np.vstack([e.done for e in experiences if e is not None])\
            .astype(np.uint8)).float().to(device)

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

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

1.5.2 Algorithm

The algorithm implementation was adapted from the course repository:

https://github.com/udacity/deep-reinforcement-learning/tree/master/ddpg-pendulum

Agent parameters:

```
state_size  : environmen state size
action_size : action size
num_agents  : # learning agents
andom_seed  : random seed number (optional)
batch_size  : minibatch size for neural network training
lr_actor    : actor neural network learning rate
lr_critic   : critic neural network learning rate
noise_theta : parameter theta for Ornstein-Uhlenbeck noise process
noise_sigma : parameter sigma for Ornstein-Uhlenbeck noise process
actor_fc1   : # nodes in first hidden layer for actor
actor_fc2   : # nodes in second hidden layer for actor
actor_fc3   : # nodes in the third hidden layer for actor
critic_fc1  : # nodes in first hidden ayer for critic
critic_fc2  : # nodes in second hidden layer for critic
critic_fc3  : # nodes in the third hidden layer for critic
update_every: # time steps between each updating neural networks
num_updates : # times to update the networks at every update_every interval
buffer_size : buffer size for experience replay
```

[11]:

```
WEIGHT_DECAY = 0 # L2 weight decay
class Agent():
    '''Interacts with and learns from the environment'''

    def __init__(self, state_size, action_size, num_agents, random_seed = 0,
                 batch_size = 128, lr_actor = 1e-4, lr_critic = 1e-3,
                 noise_theta = 0.15, noise_sigma = 0.2,
                 actor_fc1 = 128, actor_fc2 = 64, actor_fc3 = 32,
                 critic_fc1 = 128, critic_fc2 = 64, critic_fc3 = 32,
                 update_every = 1, num_updates = 1, buffer_size = int(2e6)):

        '''
            state_size (int) : state size
            action_size (int): action size
            num_agents (int) : number of agents
            random_seed (int): random seed
            batch_size (int) : minibatch size
            lr_actor (float) : the actor network learning rate
            lr_critic (float): the critic network learning rate
            noise_theta (float): theta for Ornstein-Uhlenbeck noise process
            noise_sigma (float): simga for Ornstein-Uhlenbeck noise process
            actor_fc1 (int)    : # nodes in first hidden layer for actor
            actor_fc2 = 64     : # nodes in second hidden layer for actor
            actor_fc3 = 32     : # nodes in the third hidden layer for actor
            critic_fc1 = 128   : # nodes in first hidden ayer for critic
            critic_fc2 = 64    : # nodes in second hidden layer for critic
            critic_fc3 = 32    : # nodes in the third hidden layer for critic
            update_every = 1   : # time steps between each
                                 updating neural networks
            num_updates = 1
            buffer_size = int(2e6): buffer size for experience replay
        '''
        self.state_size = state_size
        self.action_size = action_size
        self.num_agents = num_agents
        self.seed = random.seed(random_seed)
        self.batch_size = batch_size
        self.update_every = update_every
        self.num_updates = num_updates
        self.buffer_learn_size = min(batch_size * 10, buffer_size)

        # actor Network (Target Network)
        self.actor_local  = Actor(state_size, action_size, random_seed,
                                  actor_fc1, actor_fc2, actor_fc3).to(device)
        self.actor_target = Actor(state_size, action_size, random_seed,
                                  actor_fc1, actor_fc2, actor_fc3).to(device)
        self.actor_optimizer = optim.Adam(self.actor_local.parameters(),
```

```
                                     lr=lr_actor)

        # critic Network (Target Network)
        self.critic_local  = Critic(state_size, action_size,
                                    random_seed, critic_fc1, critic_fc2,
                                    critic_fc3).to(device)
        self.critic_target = Critic(state_size, action_size,
                                    random_seed, critic_fc1, critic_fc2,
                                    critic_fc3).to(device)
        self.critic_optimizer = optim.Adam(self.critic_local.parameters(),
                                           lr=lr_critic,
                                           weight_decay=WEIGHT_DECAY)

        # noise process
        self.noise = OUNoise((num_agents, action_size), random_seed,
                             theta=noise_theta, sigma=noise_sigma)

        # replay memory
        self.memory = ReplayBuffer(action_size, buffer_size,
                                   self.batch_size, random_seed)

        # initialize the time step counter
        # for updating each UPDATE_EVERY number of steps
        self.t_step = 0

    def step(self, states, actions, rewards, next_states,
             dones, gamma = 0.96, tau = 0.001):
        '''Save experience in replay memory,
        and use random sample from buffer to learn'''

        # Save experience / reward
        for state, action, reward, next_state, done in zip(states, actions,
                                                           rewards, next_states,
                                                           dones):
            self.memory.add(state, action, reward, next_state, done)

        # Learn every update_every time steps.
        self.t_step = (self.t_step + 1) % self.update_every
        if self.t_step == 0:
            # If enough samples are available in memory,
            # get a random subset from the
            # saved experiences (weighted if prior_replay = True) and learn
            if len(self.memory) > self.buffer_learn_size:
                for _ in range(self.num_updates):
                    experiences = self.memory.sample()
                    self.learn(experiences, gamma, tau)
```

```
    def act(self, state, add_noise=True, noise_scale=1.0):
        '''Returns actions for given state as per current policy'''
        state = torch.from_numpy(state).float().to(device)
        self.actor_local.eval()
        with torch.no_grad():
            action = self.actor_local(state).cpu().data.numpy()
        self.actor_local.train()
        if add_noise:
            action += (noise_scale * self.noise.sample())
        return np.clip(action, -1, 1)

    def reset(self):
        self.noise.reset()

    def learn(self, experiences, gamma, tau):
        '''
        Update policy and value parameters using
        given batch of experience tuples.
        Q_targets = r + gamma * critic_target(next_state,
                                              actor_target(next_state))
        where:
            actor_target(state) -> action
            critic_target(state, action) -> Q-value
        Params
        ======
            experiences (Tuple[torch.Tensor]): tuple of (s, a, r, s', done) tuples
            gamma (float): discount factor
        '''
        states, actions, rewards, next_states, dones = experiences

        # ************************* CRITIC UPDATE ***********************
        # Get predicted next-state actions and Q values from target models
        actions_next = self.actor_target(next_states)
        Q_targets_next = self.critic_target(next_states, actions_next)
        # Compute Q targets for current states (y_i)
        Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))
        # Compute critic loss
        Q_expected = self.critic_local(states, actions)
        critic_loss = F.mse_loss(Q_expected, Q_targets)

        # Minimize the loss
        self.critic_optimizer.zero_grad()
        critic_loss.backward()

        # Gradient clipping
        torch.nn.utils.clip_grad_norm_(self.critic_local.parameters(), 1)
```

```
        self.critic_optimizer.step()

        # ************************* ACTOR UPDATE *************************
        # Compute actor loss
        actions_pred = self.actor_local(states)
        actor_loss = -self.critic_local(states, actions_pred).mean()
        # Minimize the loss
        self.actor_optimizer.zero_grad()
        actor_loss.backward()
        self.actor_optimizer.step()

        # ********************* TARGET NETWORKS UPDATE *******************
        self.soft_update(self.critic_local, self.critic_target, tau)
        self.soft_update(self.actor_local, self.actor_target, tau)

    def soft_update(self, local_model, target_model, tau):
        '''
        Soft update model parameters.
        theta_target = tau*theta_local + (1 - tau)*theta_target
        Weighted average. Smaller tau means more of the updated target model is
            weighted towards the current target model.

        local_model : PyTorch model (weights will be copied from)
        target_model: PyTorch model (weights will be copied to)
        tau (float) : interpolation parameter
        '''
        for target_param, local_param in zip(target_model.parameters(),
                                             local_model.parameters()):
            target_param.data.copy_(tau*local_param.data +
                                    (1.0-tau)*target_param.data)
```

1.5.3 Ornstein-Uhlenbeck process

[12]:
```python
class OUNoise:

    def __init__(self, size, seed, mu=0., theta=0.15, sigma=0.2):
        '''Initialize parameters and noise process'''
        self.mu = mu * np.ones(size)
        self.theta = theta
        self.sigma = sigma
        self.size = size
        self.seed = random.seed(seed)
        self.reset()

    def reset(self):
        '''Reset the internal state (= noise) to mean (mu)'''
        self.state = copy.copy(self.mu)

    def sample(self):
        '''Update internal state and return it as a noise sample'''
        x = self.state
        dx = (self.theta * (self.mu - x) +
              self.sigma * np.random.standard_normal(self.size))

        self.state = x + dx
        return self.state
```

1.6 DDPG (Deep Deterministic Policy Gradients)

Loop through each episode:

```
- get an action
- send the action to the environment
- get the next state
- get the reward
- add the state, action, reward, and next state to the experience replay buffer
- every # time steps in the environment,
  randomly sample the experience replay buffer and perform # learning steps
- updating the critic network for each learning step
- updating the actor network for each learning step
- add the reward to the score
- update the state to the next state and loop back to step 1
```

[13]:

```python
def ddpg(agent, prefix, n_episodes=2000, max_t=1500,
         gamma_initial = 0.9, gamma_final = 0.99, gamma_rate = 0.002,
         tau_initial = 0.01, tau_final = 0.001, tau_rate = 0.001,
         noise_factor = 1.0):
    '''
    Deep Deterministic Policy Gradients

    PARAMETERS:
    agent (object)        : the learning agent
    prefix (string)       : a prefix string for naming all of the checkpoints
                            of the actor and critic neural networks that are saved
    n_episodes (int)      : maximum number of training episodes
    max_t (int)           : maximum number of timesteps per episode
    gamma_initial (float): initial gamma discount factor (0 to 1);
                              higher values favor long term over current rewards
    gamma_final (float)   : final gamma discount factor (0 to 1)
    gammma_rate (float)   : a rate (0 to 1) for increasing gamma
    tau_initial (float)   : initial value for tau, the weighting factor
                              for soft updating the neural networks
    tau_final (float)     : final value of tau
    tau_rate (float)      : rate (0 to 1) for increasing tau each episode
    noise_factor          : the value for scaling the noise
                            every episode to gradually decrease it
    '''

    gamma = gamma_initial
    gamma_scale = 1.0 - gamma_rate

    tau = tau_initial
    tau_scale = 1.0 - tau_rate

    noise_scale = 1.0
    scores_deque = deque(maxlen=100)
    scores = []
    for i_episode in trange(1, n_episodes+1):

        # Reset environment
        env_info = env.reset(train_mode=True)[brain_name]

        # Get next state
        state = env_info.vector_observations

        # state = env.reset()
        agent.reset()

        score = np.zeros(agent.num_agents)
```

```python
        for t in range(max_t):

            # get action
            action = agent.act(state, noise_scale)
            # send action to environment
            env_info = env.step(action)[brain_name]
            # get next state
            next_state = env_info.vector_observations
            # get reward
            reward = env_info.rewards

            # check if episode is finished
            done = env_info.local_done

            agent.step(state, action, reward, next_state, done, gamma, tau)
            score += np.array(env_info.rewards) # update the score
            state = next_state

            if np.any(done):
                break # exit if episode is finished

        agent_avg = np.mean(score)
        scores_deque.append(agent_avg)
        scores.append(agent_avg)

        # Increase gamma discount factor. Limit to gamma_final.
        gamma = gamma_final - gamma_scale * (gamma_final - gamma)

        tau = tau_final - tau_scale * (tau_final - tau)

        noise_scale *= noise_factor

        if i_episode % 50 == 0:
            print('\rEpisode {}\tAverage Score: {:.2f}'.\
                format(i_episode, np.mean(scores_deque)))

    # saved Model Weights
    torch.save(agent.actor_local.state_dict() , prefix + '_actor.pth')
    torch.save(agent.critic_local.state_dict(), prefix + '_critic.pth')
    return scores
```

1.7 Train the Agent

[14]:
```
agent = Agent(state_size = state_size, action_size = action_size,
              num_agents = num_agents, random_seed = 0, batch_size = 1024,
              lr_actor = 0.0001, lr_critic = 0.001,
              noise_theta = 0.4, noise_sigma = 0.6,
              actor_fc1 = 128, actor_fc2 = 64, actor_fc3 = 32,
              critic_fc1 = 128, critic_fc2 = 64, critic_fc3 = 32,
              update_every = 20, num_updates = 5, buffer_size = int(2e6))
```

1.7.1 Learning parameters:

```
agent        : the agent
prefix       : a prefix string for naming all of the checkpoints
n_episodes   : maximum number of training episodes
max_t        : maximum number of timesteps per episode
gamma_initial: initial gamma discount factor (0 to 1)
gamma_final  : final gamma discount factor (0 to 1)
gammma_rate  : a rate (0 to 1) for increasing gamma
tau_initial  : initial value for tau, the weighting factor
               for soft updating the neural networks
tau_final    : final value of tau
tau_rate     : rate (0 to 1) for increasing tau each episode
noise_factor : the value for scaling the noise (<=1)
```

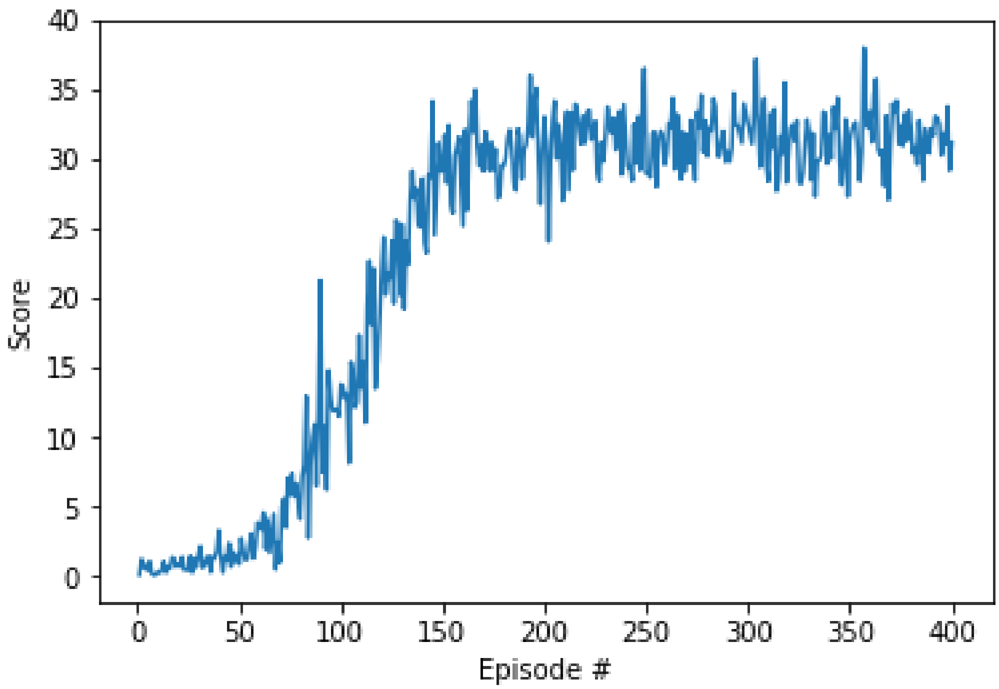
[15]:
```
scores = ddpg(
    agent,
    prefix = 'one_agent',
    n_episodes = 400,
    max_t = 10000,
    gamma_initial = 0.95,
    gamma_final = 0.99,
    gamma_rate = 0.01,
    tau_initial = 0.01,
    tau_final = 0.001,
    tau_rate = 0.1,
    noise_factor = 0.995)
```

```
Episode 100     Average Score: 3.70

Episode 150     Average Score: 13.99

Episode 200     Average Score: 26.04

Episode 250     Average Score: 30.97

Episode 300     Average Score: 31.47

Episode 350     Average Score: 31.35

Episode 400     Average Score: 31.50
```

1.7.2 A plot of rewards per episode

The agent receives an average reward (over 100 episodes) of at least +30

[16]:
```
%matplotlib inline
import matplotlib.pyplot as plt

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



1.8 Test the Agent

Load model weights for the actor and critic networks for the pre-trained agent. Run the agent for 100 episodes and average the scores

[18]:
```python
# env_info = env.reset(train_mode=False)[brain_name] # reset the environment
agent = Agent(
    state_size = state_size,
    action_size = action_size,
    num_agents = num_agents,
)
n_episodes = 100
prefix_name = "one_agent"

# load the weights from file
agent.actor_local.load_state_dict(torch.load(prefix_name + "_actor.pth"))
agent.critic_local.load_state_dict(torch.load(prefix_name + "_critic.pth"))

# Get the default brain
brain_name = env.brain_names[0]
brain = env.brains[brain_name]

total_score = 0.0
for i in range(n_episodes):
    # Reset the environment
    env_info = env.reset(train_mode=False)[brain_name]
    # Get the current state
    state = env_info.vector_observations
    # Initialize the scores
    score = np.zeros(agent.num_agents)

    while True:
        # Choose actions
        action = agent.act(state)
        # Send actions to the environment
        env_info = env.step(action)[brain_name]
        # Get the next state
        next_state = env_info.vector_observations
        # Get rewards
        reward = env_info.rewards
        # Check if the episode is finishised
        done = env_info.local_done
        # Add rewards to the scores
        score += reward
        # Replace the current state with the next state for the next timestep
        state = next_state

        # Exit the loop if the episode is finished
```

```
        if np.any(done):
            break

    print("\rEpisode {}\tAverage Score: {:.2f}".format(i + 1, np.mean(score)))
    total_score += np.mean(score)

print("Average score over {} episodes: {:.2f}"\
      .format(n_episodes, total_score / n_episodes))
```

```
Episode 1	Average Score: 37.84
Episode 2	Average Score: 37.56
Episode 3	Average Score: 39.15
Episode 4	Average Score: 35.63
Episode 5	Average Score: 34.56
Episode 6	Average Score: 39.28
Episode 7	Average Score: 39.26
Episode 8	Average Score: 36.60
Episode 9	Average Score: 39.09
Episode 10	Average Score: 36.73
Episode 11	Average Score: 37.27
Episode 12	Average Score: 37.41
Episode 13	Average Score: 37.39
Episode 14	Average Score: 38.83
Episode 15	Average Score: 38.01
Episode 16	Average Score: 37.13
Episode 17	Average Score: 39.42
Episode 18	Average Score: 38.85
Episode 19	Average Score: 38.28
Episode 20	Average Score: 39.49
Episode 21	Average Score: 36.51
Episode 22	Average Score: 39.54
Episode 23	Average Score: 36.92
Episode 24	Average Score: 38.39
Episode 25	Average Score: 37.98
Episode 26	Average Score: 38.66
Episode 27	Average Score: 37.75
Episode 28	Average Score: 37.27
Episode 29	Average Score: 38.16
Episode 30	Average Score: 37.38
Episode 31	Average Score: 35.98
Episode 32	Average Score: 37.56
Episode 33	Average Score: 38.33
Episode 34	Average Score: 38.09
Episode 35	Average Score: 37.66
Episode 36	Average Score: 38.57
Episode 37	Average Score: 38.19
Episode 38	Average Score: 39.62
Episode 39	Average Score: 37.67
```

```
Episode 40      Average Score: 37.67
Episode 41      Average Score: 37.47
Episode 42      Average Score: 39.17
Episode 43      Average Score: 39.06
Episode 44      Average Score: 37.94
Episode 45      Average Score: 36.59
Episode 46      Average Score: 38.66
Episode 47      Average Score: 37.03
Episode 48      Average Score: 39.66
Episode 49      Average Score: 39.21
Episode 50      Average Score: 38.10
Episode 51      Average Score: 38.68
Episode 52      Average Score: 39.38
Episode 53      Average Score: 37.28
Episode 54      Average Score: 38.66
Episode 55      Average Score: 38.66
Episode 56      Average Score: 36.28
Episode 57      Average Score: 38.19
Episode 58      Average Score: 38.32
Episode 59      Average Score: 37.34
Episode 60      Average Score: 39.53
Episode 61      Average Score: 38.54
Episode 62      Average Score: 38.67
Episode 63      Average Score: 38.09
Episode 64      Average Score: 36.22
Episode 65      Average Score: 38.68
Episode 66      Average Score: 39.57
Episode 67      Average Score: 39.67
Episode 68      Average Score: 37.23
Episode 69      Average Score: 38.50
Episode 70      Average Score: 38.12
Episode 71      Average Score: 38.13
Episode 72      Average Score: 36.28
Episode 73      Average Score: 38.29
Episode 74      Average Score: 34.87
Episode 75      Average Score: 39.45
Episode 76      Average Score: 38.09
Episode 77      Average Score: 37.47
Episode 78      Average Score: 36.92
Episode 79      Average Score: 37.88
Episode 80      Average Score: 38.28
Episode 81      Average Score: 36.74
Episode 82      Average Score: 38.80
Episode 83      Average Score: 39.42
Episode 84      Average Score: 38.90
Episode 85      Average Score: 38.45
Episode 86      Average Score: 38.08
Episode 87      Average Score: 36.24
```

```
Episode 88      Average Score: 37.77
Episode 89      Average Score: 39.21
Episode 90      Average Score: 38.22
Episode 91      Average Score: 38.83
Episode 92      Average Score: 38.32
Episode 93      Average Score: 38.20
Episode 94      Average Score: 37.76
Episode 95      Average Score: 38.15
Episode 96      Average Score: 39.55
Episode 97      Average Score: 39.11
Episode 98      Average Score: 37.57
Episode 99      Average Score: 38.54
Episode 100     Average Score: 37.53
Average score over 100 episodes: 38.05
```

When finished, you can close the environment.

[19]:
```
env.close()
```

1.9 Ideas for Future Work

1. Optimization of the existing code
2. Search for optimal hyperparameters (it is necessary to pay special attention to the values of theta and sigma, which have a significant impact on the learning process)
3. Implementation of the other algorithms such as PPO, A3C and D4PG