Report

August 13, 2019

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.

1.0.1 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
```

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]:
```
env = UnityEnvironment(file_name='Reacher_Windows_x86_64/Reacher.exe')
```

```
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]
```

1.0.2 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]:
```python
# 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: 20
Size of each action: 4
There are 20 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
  5.55726624e+00  0.00000000e+00  1.00000000e+00  0.00000000e+00
 -1.68164849e-01]
```

1.0.3 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!

[6]:
```
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 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
    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 if episode␣
↪finished
        break
print('Total score (averaged over agents) this episode: {}'.format(np.
↪mean(scores)))
```

```
Total score (averaged over agents) this episode: 0.07699999827891588
```

When finished, you can close the environment.

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

1.0.4 4. It's Your Turn!

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:

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

1.0.5 Implementation details

`experiences = random.choices(self.memory, k=self.batch_size)` - randomness with replacement is a key for success of DDPG implementation.

4.1 Importing libraries

[1]:
```
import datetime
import time
import random
import copy
from collections import namedtuple, deque
from unityagents import UnityEnvironment
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

4.2 Starting environment

[2]:
```
env = UnityEnvironment(file_name='Reacher_Windows_x86_64/Reacher.exe')
```

```
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: , , ,
```

4.3 Setting parameters `BUFFER_SIZE = int(5e5)` - a buffer should be big to accumulate maximum experience for training.

`BATCH_SIZE = 256` - big batch size mean more generic update during training. I also want to mention that batch size is used in random sampling and big batch size get more samples from `class ReplayBuffer`

`GAMMA = 0.99` - discount factor. I have made this number big to slightly discount distant Q values.

`TAU = 1e-3` - controlling how much two neural networks should be similar. And they are similar according to TAU ratio.

`LR_ACTOR = 1e-3` - learning rate of `1e-3` is one of the optimal learning rates for `Adam` optimizer.

`LR_CRITIC = 1e-3` - learning rate of `1e-3` is one of the optimal learning rates for `Adam` optimizer.

`WEIGHT_DECAY = 0` - No weight decay is performing better in a DDPG implementation.

`device = "cpu"` - torch 0.4.0 is compiled for CUDA 8.0 and in my case it causes a 3 minutes wait period for a PyTorch to understand that my GPU card is not supported.

[3]:
```
BUFFER_SIZE = int(5e5)  # replay buffer size
BATCH_SIZE = 256        # minibatch size
GAMMA = 0.99            # discount factor
TAU = 1e-3              # for soft update of target parameters
LR_ACTOR = 1e-3         # learning rate of the actor
LR_CRITIC = 1e-3        # learning rate of the critic
WEIGHT_DECAY = 0        # L2 weight decay

device = "cpu"
```

4.4 Getting data about an environment

[4]:
```
# get the default brain
brain_name = env.brain_names[0]
brain = env.brains[brain_name]
```

4.5 Creating `class Actor` that will act on behalf of the agent Too shallow model and algorithm do not learn well and results are inconsistent. Too deep model and algorithm takes long time to converge. I have choosen this model because it has quite an optimal performance.

[5]:
```python
def hidden_init(layer):
    fan_in = layer.weight.data.size()[0]
    lim = 1. / np.sqrt(fan_in)
    return (-lim, lim)

class Actor(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed, fc1_units=128,␣
 ↪fc2_units=128):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(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, action_size)
        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_(-3e-3, 3e-3)

    def forward(self, state):
        """Build an actor (policy) network that maps states -> actions."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return F.tanh(self.fc3(x))
```

4.5 Creating `class Critic` which "criticize" our `class Actor` This model gives a mediocre result. I stopped on that.

The most important part is `torch.cat`:

```
def forward(self, state, action):
    """Build a critic (value) network that maps (state, action) pairs -> Q-values."""
    xs = F.relu(self.fcs1(state))
    x = torch.cat((xs, action), dim=1)
    x = F.relu(self.fc2(x))
```

```
        return self.fc3(x)
```

It includes actions into `class Critic` to better forecast Q values.

[6]:
```
class Critic(nn.Module):
    """Critic (Value) Model."""

    def __init__(self, state_size, action_size, seed, fcs1_units=128,␣
↪fc2_units=128):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fcs1_units (int): Number of nodes in the first hidden layer
            fc2_units (int): Number of nodes in the second 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, 1)
        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_(-3e-3, 3e-3)

    def forward(self, state, action):
        """Build a critic (value) network that maps (state, action) pairs ->␣
↪Q-values."""
        xs = F.relu(self.fcs1(state))
        x = torch.cat((xs, action), dim=1)
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

4.6 Creating `class Agent` that will represent our agent The most important component is a method `learn`:

Instanstiating actors.

```
self.actor_local = Actor(state_size, action_size, random_seed).to(device)
self.actor_target = Actor(state_size, action_size, random_seed).to(device)
self.actor_optimizer = optim.Adam(self.actor_local.parameters(), lr=LR_ACTOR)
```

Instanstiating critics.

```
self.critic_local = Critic(state_size, action_size, random_seed).to(device)
self.critic_target = Critic(state_size, action_size, random_seed).to(device)
self.critic_optimizer = optim.Adam(self.critic_local.parameters(), lr=LR_CRITIC, weight_decay=W
```

`actions_next = self.actor_target(next_states)` - getting target actions.

`Q_targets_next = self.critic_target(next_states, actions_next)` - getting target Q value.

`Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))` - getting updated target Q value.

`Q_expected = self.critic_local(states, actions)` - getting current Q value.

`critic_loss = F.mse_loss(Q_expected, Q_targets)` - decreasing difference between current Q value and updated target Q value.

`actions_pred = self.actor_local(states)` - getting current actions.

`actor_loss = -self.critic_local(states, actions_pred).mean()` - updating `critic_local`. Increasing Q values using gradient ascent.

Making target closer to local with coefficient TAU.

```
self.soft_update(self.critic_local, self.critic_target, TAU)
self.soft_update(self.actor_local, self.actor_target, TAU)
```

`self.epsilon -= self.epsilon_decay` - decreasing exploration exploitation rate.

[7]:
```
class Agent():
    """Interacts with and learns from the environment."""

    def __init__(self, state_size, action_size, random_seed, epsilon=1.0,␣
 ↪epsilon_decay=1e-6):
        """Initialize an Agent object.

        Params
        ======
            state_size (int): dimension of each state
            action_size (int): dimension of each action
            random_seed (int): random seed
        """
        self.epsilon = epsilon
        self.epsilon_decay = epsilon_decay

        self.state_size = state_size
        self.action_size = action_size
        self.seed = random.seed(random_seed)

        # Actor Network (w/ Target Network)
        self.actor_local = Actor(state_size, action_size, random_seed).
 ↪to(device)
        self.actor_target = Actor(state_size, action_size, random_seed).
 ↪to(device)
        self.actor_optimizer = optim.Adam(self.actor_local.parameters(),␣
 ↪lr=LR_ACTOR)
```

```
        # Critic Network (w/ Target Network)
        self.critic_local = Critic(state_size, action_size, random_seed).
 ↪to(device)
        self.critic_target = Critic(state_size, action_size, random_seed).
 ↪to(device)
        self.critic_optimizer = optim.Adam(self.critic_local.parameters(),␣
 ↪lr=LR_CRITIC, weight_decay=WEIGHT_DECAY)
        # Noise process
        self.noise = OUNoise(action_size, random_seed)

        # Replay memory
        self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE,␣
 ↪random_seed)
#         self.cache = ReplayBuffer(action_size, int(6e4), BATCH_SIZE,␣
 ↪random_seed)


        for target_param, param in zip(self.actor_target.parameters(), self.
 ↪actor_local.parameters()):
            target_param.data.copy_(param.data)
        for target_param, param in zip(self.critic_target.parameters(), self.
 ↪critic_local.parameters()):
            target_param.data.copy_(param.data)


    def step(self, state, action, reward, next_state, done, epoch):
        """Save experience in replay memory, and use random sample from buffer␣
 ↪to learn."""
        # Save experience / reward
#         self.memory.add(state, action, reward, next_state, done)
#         for states, actions, rewards, next_states, dones in zip(state,␣
 ↪action, reward, next_state, done):
#             self.memory.add(states, actions, rewards, next_states, dones)
        self.memory.add(state, action, reward, next_state, done)

#               self.cache.add(states, actions, rewards, next_states, dones)


        # Learn, if enough samples are available in memory
        if len(self.memory) > BATCH_SIZE and epoch % 20 == 0:
            for num in range(18):
                experiences = self.memory.sample()
                self.learn(experiences, GAMMA)
#             for num in range(3):
#                 experiences = self.cache.sample()
#                   self.learn(experiences, GAMMA)

    def act(self, state, add_noise=True):
        """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 += self.epsilon * self.noise.sample()
        return action

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

    def learn(self, experiences, gamma):
        """Update policy and value parameters using given batch of experience␣
↪tuples.
        Q_targets = r +  * 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

        # ---------------------------- update critic␣
↪---------------------------- #
        # 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()
        torch.nn.utils.clip_grad_norm_(self.critic_local.parameters(), 1)
        self.critic_optimizer.step()

        # ---------------------------- update actor␣
↪---------------------------- #
        # 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()

        # ----------------------- update target networks␣
↪----------------------- #
        self.soft_update(self.critic_local, self.critic_target, TAU)
        self.soft_update(self.actor_local, self.actor_target, TAU)
        self.epsilon -= self.epsilon_decay
        self.noise.reset()


    def soft_update(self, local_model, target_model, tau):
        """Soft update model parameters.
        _target = *_local + (1 - )*_target

        Params
        ======
            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)
```

4.7 Noise One of the key components of DDPG. Ornstein-Uhlenbeck makes exploration more effective.

[8]:
```
class OUNoise:
    """Ornstein-Uhlenbeck process."""

    def __init__(self, size, seed, mu=0., theta=0.15, sigma=0.05):
        """Initialize parameters and noise process."""
        self.mu = mu * np.ones(size)
        self.theta = theta
        self.sigma = sigma
        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.array([random.
↪random() for i in range(len(x))])
        self.state = x + dx
        return self.state
```

4.8 Replay buffer

[9]:
```
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."""
        for num in range(20):
            e = self.experience(state[num], action[num], reward[num],␣
↪next_state[num], done[num])
            self.memory.append(e)

    def sample(self):
        """Randomly sample a batch of experiences from memory."""

        experiences = random.choices(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)
```

4.9 Instanstiating class Agent

[10]:
```
agent = Agent(state_size=33, action_size=4, random_seed=2)
```

5.0 Training loop

[11]:
```
def ddpg(n_episodes=1000):
    print('Start time: ',datetime.datetime.now())
    all_agents = np.array([]).reshape(0,20)
    start_time = time.time()
    print('\rEP, Min, Max, Average, AV100, Time')
    for i_episode in range(1, n_episodes+1):
        env_info = env.reset(train_mode=True)[brain_name]      # reset the␣
↪environment
        states = env_info.vector_observations                  # get the␣
↪current state (for each agent)
        scores = np.zeros(20)                          # initialize the score␣
↪(for each agent)
        agent.reset()

        score_average = 0
        timestep = time.time()
        for t in range(100000000):
            actions = agent.act(states, add_noise=True)
            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
            agent.step(states, actions, rewards, next_states, dones,t)
            states = next_states                               # roll over␣
↪states to next time step
            scores += rewards                                  # update the␣
↪score (for each agent)
            if np.any(dones):                                  # exit loop if␣
↪episode finished
                break

        all_agents = np.concatenate((all_agents, scores.reshape(1,20)), axis=0)
        mean_100 = np.mean(all_agents[-101:-1], axis=0)
        print('{}, {:.2f}, {:.2f}, {:.2f}, {:.2f}, {:.2f}'\
              .format(str(i_episode).zfill(3), np.min(scores), np.max(scores),␣
↪np.mean(scores), float(np.mean(mean_100)),
                      time.time() - timestep), end="\n")
        if  np.all(mean_100 > 30.0):
            end_time = time.time()
            print('\nSolved in {:d} episodes!\tAvg Score: {:.2f}, time: {}'.
↪format(i_episode, float(np.mean(mean_100)),
                                                                               ␣
↪   end_time-start_time))
            for num, scr in enumerate(mean_100):
                print('Agent {} average: {:.2f} '.format(num+1, scr))
            torch.save(agent.actor_local.state_dict(), 'checkpoint_actor.pth')
            torch.save(agent.critic_local.state_dict(), 'checkpoint_critic.pth')
            print('End time: ',datetime.datetime.now())
            return all_agents
```

5.1 Training

[12]:
```
scores = ddpg()
```

```
Start time:  2019-08-12 21:10:10.629150
EP, Min, Max, Average, AV100, Time

C:\Users\andreiliphd\Anaconda3\envs\drlnd\lib\site-
packages\numpy\core\fromnumeric.py:3118: RuntimeWarning: Mean of empty slice.
  out=out, **kwargs)
C:\Users\andreiliphd\Anaconda3\envs\drlnd\lib\site-
packages\numpy\core\_methods.py:78: RuntimeWarning: invalid value encountered in
true_divide
  ret, rcount, out=ret, casting='unsafe', subok=False)
C:\Users\andreiliphd\Anaconda3\envs\drlnd\lib\site-
packages\ipykernel_launcher.py:31: RuntimeWarning: invalid value encountered in
greater
```

```
001, 0.00, 1.69, 0.47, nan, 12.37
002, 0.00, 0.21, 0.06, 0.47, 12.07
003, 0.00, 1.16, 0.17, 0.26, 12.21
004, 0.00, 0.65, 0.10, 0.23, 12.31
005, 0.00, 1.96, 0.28, 0.20, 12.38
006, 0.00, 0.90, 0.27, 0.22, 13.00
007, 0.00, 0.90, 0.38, 0.22, 13.01
008, 0.43, 2.27, 1.20, 0.25, 13.18
009, 0.72, 2.30, 1.24, 0.37, 13.35
010, 0.60, 2.59, 1.49, 0.46, 13.46
011, 0.61, 2.57, 1.42, 0.57, 13.47
012, 0.45, 4.85, 1.54, 0.64, 13.87
013, 0.00, 2.52, 1.28, 0.72, 14.03
014, 0.00, 3.81, 1.62, 0.76, 14.26
015, 0.16, 3.45, 1.69, 0.82, 14.52
016, 0.66, 4.14, 2.00, 0.88, 15.21
017, 0.47, 3.93, 2.18, 0.95, 15.48
018, 0.54, 3.93, 1.79, 1.02, 15.56
019, 1.11, 3.71, 2.51, 1.07, 15.83
020, 1.21, 4.15, 2.29, 1.14, 16.04
021, 0.95, 5.42, 3.38, 1.20, 16.13
022, 1.49, 4.41, 2.90, 1.30, 16.40
023, 0.81, 4.72, 2.73, 1.38, 16.37
024, 0.53, 6.13, 3.31, 1.43, 16.72
025, 0.35, 4.87, 2.52, 1.51, 17.22
026, 0.64, 6.78, 2.74, 1.55, 17.32
027, 1.14, 5.31, 3.21, 1.60, 17.36
028, 1.50, 7.55, 3.99, 1.66, 17.28
029, 0.65, 6.42, 3.20, 1.74, 17.57
030, 1.09, 8.87, 4.69, 1.79, 17.46
031, 2.03, 6.89, 4.16, 1.89, 17.56
032, 1.74, 6.62, 3.86, 1.96, 17.87
033, 2.63, 7.86, 5.08, 2.02, 17.83
034, 2.01, 8.74, 4.97, 2.11, 17.82
035, 2.02, 9.46, 6.21, 2.20, 17.70
036, 3.59, 10.80, 7.10, 2.31, 17.68
037, 1.44, 18.65, 7.28, 2.45, 17.81
038, 3.05, 11.94, 8.19, 2.58, 17.65
039, 2.60, 10.95, 6.57, 2.72, 17.85
040, 3.78, 15.61, 9.04, 2.82, 17.71
041, 3.50, 15.97, 8.43, 2.98, 17.83
042, 4.01, 16.52, 9.49, 3.11, 17.63
043, 2.19, 16.30, 10.47, 3.26, 17.58
044, 4.18, 15.49, 9.50, 3.43, 17.72
045, 3.91, 15.52, 10.84, 3.57, 17.82
046, 3.11, 20.63, 10.36, 3.73, 18.45
047, 7.19, 21.93, 12.95, 3.87, 17.63
048, 5.61, 18.83, 12.83, 4.07, 17.43
```

```
049, 6.00, 20.54, 12.99, 4.25, 17.75
050, 5.83, 17.45, 12.69, 4.43, 17.72
051, 9.59, 22.73, 15.28, 4.59, 17.43
052, 8.55, 22.84, 15.20, 4.80, 17.35
053, 8.35, 24.88, 16.45, 5.00, 17.48
054, 10.46, 22.23, 16.54, 5.22, 17.40
055, 10.68, 23.67, 18.08, 5.43, 17.61
056, 10.74, 23.39, 17.61, 5.66, 17.58
057, 12.37, 23.63, 17.85, 5.87, 17.76
058, 14.05, 23.23, 18.30, 6.08, 17.56
059, 14.89, 27.22, 18.22, 6.29, 17.55
060, 9.20, 23.53, 16.57, 6.50, 17.64
061, 12.12, 26.17, 20.31, 6.66, 17.76
062, 9.82, 27.78, 19.36, 6.89, 17.80
063, 11.09, 26.26, 19.62, 7.09, 17.73
064, 13.96, 25.83, 19.56, 7.29, 17.71
065, 11.82, 25.19, 18.82, 7.48, 17.74
066, 14.86, 30.43, 20.97, 7.65, 17.71
067, 11.29, 27.56, 18.65, 7.86, 17.78
068, 10.24, 25.60, 19.26, 8.02, 17.83
069, 11.39, 28.27, 20.14, 8.18, 17.63
070, 10.81, 28.18, 20.71, 8.36, 17.74
071, 13.92, 35.56, 22.13, 8.53, 17.97
072, 14.99, 28.90, 22.30, 8.72, 18.20
073, 9.39, 28.36, 21.65, 8.91, 17.88
074, 15.89, 27.23, 24.06, 9.09, 17.85
075, 9.31, 27.16, 21.96, 9.29, 17.82
076, 12.31, 33.18, 22.40, 9.46, 17.62
077, 7.84, 29.90, 19.48, 9.63, 17.65
078, 6.76, 31.16, 23.96, 9.76, 17.54
079, 8.41, 36.72, 23.50, 9.94, 17.52
080, 16.67, 30.35, 24.22, 10.11, 17.42
081, 13.67, 31.56, 24.84, 10.29, 17.48
082, 10.46, 32.97, 22.46, 10.47, 17.59
083, 12.17, 30.28, 22.04, 10.61, 17.69
084, 15.68, 28.89, 23.73, 10.75, 17.70
085, 15.41, 37.72, 24.55, 10.90, 17.49
086, 14.81, 29.72, 23.56, 11.06, 17.46
087, 18.26, 30.81, 25.26, 11.21, 17.49
088, 18.35, 32.07, 25.39, 11.37, 17.27
089, 20.66, 32.81, 26.34, 11.53, 17.58
090, 23.38, 36.71, 30.11, 11.70, 17.69
091, 17.01, 34.01, 29.07, 11.90, 17.47
092, 14.44, 35.39, 27.31, 12.09, 17.71
093, 16.32, 36.04, 29.12, 12.26, 17.48
094, 20.17, 35.51, 27.12, 12.44, 17.37
095, 20.47, 35.41, 28.76, 12.59, 17.32
096, 18.79, 35.24, 28.57, 12.76, 17.12
```

```
097, 20.61, 35.82, 27.94, 12.93, 17.33
098, 20.45, 35.18, 29.86, 13.08, 17.10
099, 16.72, 35.83, 28.90, 13.25, 17.15
100, 18.94, 36.16, 29.58, 13.41, 17.09
101, 17.49, 34.28, 27.47, 13.57, 17.13
102, 20.54, 39.04, 28.49, 13.84, 17.25
103, 26.74, 39.62, 31.55, 14.13, 16.95
104, 25.51, 38.98, 30.76, 14.44, 17.45
105, 25.22, 37.19, 30.68, 14.75, 17.36
106, 16.84, 36.14, 30.74, 15.05, 17.57
107, 18.66, 37.82, 27.59, 15.36, 17.35
108, 15.80, 37.80, 30.88, 15.63, 17.43
109, 15.92, 36.85, 28.09, 15.93, 17.59
110, 18.08, 37.15, 28.29, 16.19, 17.58
111, 18.66, 37.15, 28.67, 16.46, 17.75
112, 23.25, 37.02, 31.62, 16.74, 17.46
113, 21.29, 39.07, 34.32, 17.04, 17.53
114, 21.06, 39.54, 30.60, 17.37, 17.66
115, 25.24, 38.68, 32.54, 17.66, 17.44
116, 21.18, 37.86, 31.32, 17.96, 17.40
117, 20.61, 36.48, 30.04, 18.26, 17.77
118, 17.24, 36.48, 31.48, 18.54, 17.64
119, 25.56, 38.76, 31.85, 18.83, 17.29
120, 23.32, 39.53, 33.89, 19.13, 17.54
121, 21.67, 39.59, 33.53, 19.44, 17.40
122, 25.76, 39.05, 34.53, 19.74, 17.38
123, 15.50, 39.23, 33.93, 20.06, 17.55
124, 29.37, 39.39, 35.19, 20.37, 17.25
125, 21.09, 39.60, 34.61, 20.69, 17.54
126, 29.32, 39.49, 36.37, 21.01, 17.23
127, 35.35, 39.31, 37.83, 21.35, 17.15
128, 20.84, 39.56, 35.36, 21.69, 17.20
129, 26.26, 39.66, 36.93, 22.01, 17.07
130, 18.73, 39.63, 36.22, 22.35, 17.26
131, 28.73, 39.61, 37.78, 22.66, 17.24
132, 33.09, 39.67, 37.53, 23.00, 17.31
133, 25.13, 39.66, 36.66, 23.33, 17.31
134, 26.09, 39.38, 37.31, 23.65, 17.31
135, 28.21, 39.65, 36.85, 23.97, 17.41
136, 32.54, 39.62, 36.98, 24.28, 17.39
137, 30.32, 39.58, 35.67, 24.58, 17.56
138, 29.89, 39.66, 36.02, 24.86, 17.45
139, 29.52, 39.65, 35.36, 25.14, 17.51
140, 24.31, 39.57, 35.67, 25.43, 17.55
141, 31.41, 39.17, 36.89, 25.69, 17.51
142, 32.77, 39.54, 37.10, 25.98, 17.59
143, 32.01, 39.10, 36.81, 26.25, 17.36
144, 23.34, 39.30, 35.69, 26.52, 17.32
```

```
145, 31.75, 38.87, 36.44, 26.78, 17.26
146, 33.20, 39.02, 36.49, 27.04, 17.38
147, 33.21, 38.91, 36.36, 27.30, 17.48
148, 31.44, 39.14, 36.26, 27.53, 17.23
149, 28.54, 38.90, 35.77, 27.77, 17.38
150, 29.19, 38.85, 35.75, 27.99, 17.25
151, 33.54, 39.65, 37.39, 28.22, 17.37
152, 21.88, 39.51, 35.92, 28.45, 17.33
153, 29.00, 39.37, 36.58, 28.65, 17.07
154, 30.87, 39.61, 37.55, 28.85, 17.18
155, 29.71, 39.55, 37.33, 29.06, 17.17
156, 20.55, 39.38, 34.47, 29.26, 17.29
157, 21.84, 39.45, 35.95, 29.42, 17.24
158, 35.89, 39.53, 38.22, 29.61, 17.77
159, 33.55, 39.66, 37.60, 29.81, 17.78
160, 32.79, 39.13, 37.11, 30.00, 17.52
161, 32.65, 39.49, 37.12, 30.20, 17.71
162, 25.02, 39.63, 35.26, 30.37, 17.59
163, 25.97, 37.57, 35.04, 30.53, 17.48
164, 28.69, 39.43, 36.25, 30.69, 17.53
165, 30.66, 39.64, 36.21, 30.85, 17.39
166, 30.38, 39.47, 36.66, 31.03, 17.53

Solved in 166 episodes! Avg Score: 31.03, time: 2829.3051903247833
Agent 1 average: 30.92
Agent 2 average: 31.59
Agent 3 average: 31.73
Agent 4 average: 31.03
Agent 5 average: 30.52
Agent 6 average: 31.17
Agent 7 average: 31.70
Agent 8 average: 30.96
Agent 9 average: 30.93
Agent 10 average: 30.56
Agent 11 average: 31.01
Agent 12 average: 30.85
Agent 13 average: 31.57
Agent 14 average: 30.83
Agent 15 average: 30.92
Agent 16 average: 30.62
Agent 17 average: 30.15
Agent 18 average: 30.71
Agent 19 average: 31.66
Agent 20 average: 31.07
End time:  2019-08-12 21:57:19.940336
```

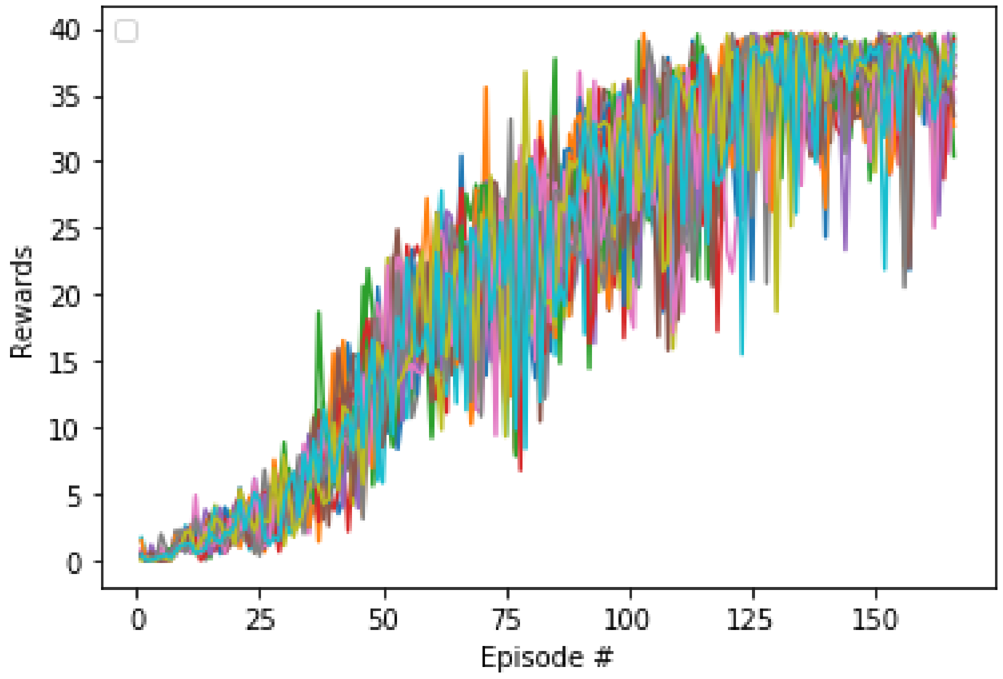
[13]:
```
env.close()
```

6. Visualizing rewards accross 20 workers

[29]:
```
plt.plot(np.arange(1, len(scores)+1), scores)
plt.ylabel('Rewards')
plt.xlabel('Episode #')
```

```
WARNING:matplotlib.legend:No handles with labels found to put in legend.
```

[29]:
```
<matplotlib.legend.Legend at 0x267ac514c18>
```



1.0.6 Improvements

1. Batch normalization could be added to improve numerical stability.
2. Sigma coefficient might be tuned for a faster convergence.
3. Neural network might be tuned for a better performance. Too many layers and two many neurons lead to a slower convergence.

DDPG looks for me very fragile. One step right or left and an implementation should be retuned.

[ ]:
```

```