Report for the Continuous Control Project

Setting up the environment and DDPG algorithm

Import required libraries

In [1]:
```
from unityagents import UnityEnvironment
import numpy as np
import datetime
import torch
from collections import deque
import matplotlib.pyplot as plt
%matplotlib inline
from ddpg_agent import Agents
```

Unity environment configuration

In [2]:
```
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: , , ,
```

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

State and action space and DDPG configuration

The objective of Unity's [Reacher environment (https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Examples.md#reacher)](https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Examples.md#reacher) is to move the 20 double-jointed arms to reach and maintain a target location for as long as possible.

Action space: For each of the 20 arms, the action space is a vector with four numbers, corresponding to torque applicable to two joints. Every entry in the action vector should be a number between -1 and 1.

State space: The observation space consists of 33 variables corresponding to position, rotation, velocity, and angular velocities for each of the 20 arms.

Reward function: The reward is given to each agent individually: A reward of +0.1 is given each step the agent's arm is in its individual target location.

DDPG structure: Similar to Google DeepMind's paper, ["Continuous Control with Deep Reinforcement Learning" (https://arxiv.org/abs/1509.02971)](https://arxiv.org/abs/1509.02971), the adopted learning algorithm is a DDPG algorithm. DDPG is a model-free policy-based reinforcement learning algorithm where agents learn by observing state spaces with no prior knowledge of the environment. Learning improves by using policy gradient optimization.

DDPG is an Actor-Critic model:

- The Actor is a policy-based algorithm with high variance, taking relatively long to converge.
- The Critic is a value-based algorithm with high bias instead

In this approach, Actor and Critic work together to reach better convergence and performance.

Actor model

Neural network with 3 fully connected layers:

- Fully connected layer 1: with input = 33 (state spaces) and output = 400
- Fully connected layer 2: with input = 400 and output = 300
- Fully connected layer 3: with input = 300 and output = 4 (for each of the 4 actions)

Tanh is used in the final layer that maps states to actions. Batch normalization is used for mini batch training.

Critic model

Neural network with 3 fully connected layers:

- Fully connected layer 1: with input = 33 (state spaces) and output = 400
- Fully connected layer 2: with input = 404 (states and actions) and output = 300
- Fully connected layer 3: with input = 300 and output = 1 (maps states and actions to Q-values)

Parameters used in the DDPG algorithm:

- Replay buffer size: BUFFER_SIZE = int(1e5)
- Minibatch size: BATCH_SIZE = 128
- Discount factor: GAMMA = 0.99

- Soft update of target parameters: TAU = 1e-3
- Learning rate of the actor: LR_ACTOR = 1e-4
- Learning rate of the critic: LR_CRITIC = 1e-3
- L2 weight decay: WEIGHT_DECAY = 0

In [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: 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 -4.0000
0000e+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]
```

Taking Random Actions in the Environment

In [5]:
```
env_info = env.reset(train_mode=False)[brain_name]     # reset the
environment
states = env_info.vector_observations                  # get the cu
rrent 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 action
s 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 s
tate (for each agent)
    rewards = env_info.rewards                         # get reward
(for each agent)
    dones = env_info.local_done                        # see if epi
sode 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: 0.113999997451901
44
```

DDPG training and results:

In [6]:
```python
agents = Agents(state_size=state_size,
                action_size=action_size,
                num_agents=num_agents,
                random_seed=0)
print(agents.actor_local)
print(agents.critic_local)
```

```
Actor(
  (fc1): Linear(in_features=33, out_features=400, bias=True)
  (fc2): Linear(in_features=400, out_features=300, bias=True)
  (fc3): Linear(in_features=300, out_features=4, bias=True)
)
Critic(
  (fcs1): Linear(in_features=33, out_features=400, bias=True)
  (fc2): Linear(in_features=404, out_features=300, bias=True)
  (fc3): Linear(in_features=300, out_features=1, bias=True)
)
```

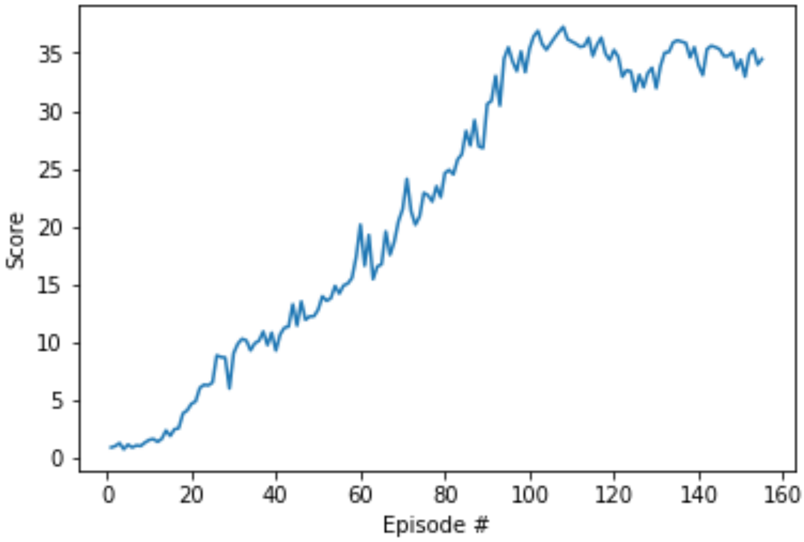
In [7]:
```
def ddpg(n_episodes=2000, max_t=1000):
    scores_deque = deque(maxlen=100)
    scores = []
    for i_episode in range(1, n_episodes+1):
        env_info = env.reset(train_mode=True)[brain_name]
        state = env_info.vector_observations
        agents.reset()
        score = np.zeros(num_agents)
        for t in range(max_t):
            action = agents.act(state)
            env_info = env.step(action)[brain_name]
            next_state = env_info.vector_observations
            rewards = env_info.rewards
            dones = env_info.local_done
            agents.step(state, action, rewards, next_state, dones)
            state = next_state
            score += rewards
            if np.any(dones):
                print('\tSteps: ', t)
                break
        scores_deque.append(np.mean(score))
        scores.append(np.mean(score))
        print('\rEpisode {}\tAverage Score: {:.2f}\tScore: {:.3f}'.\
              format(i_episode, np.mean(scores_deque), np.mean(score)), end="")
        avg_score = np.mean(scores_deque)
        if i_episode % 20 == 0 or avg_score > 30:
            print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, avg_score))
            torch.save(agents.actor_local.state_dict(), 'checkpoint_actor.pth')
            torch.save(agents.critic_local.state_dict(), 'checkpoint_critic.pth')
            if avg_score > 30:
                print('\nEnvironment solved in {:d} episodes!'.format(i_episode))
                break
    return scores
```

In [8]:
```
scores = ddpg()
```

```
Episode 20      Average Score: 1.87     Score: 4.662
Episode 40      Average Score: 5.23     Score: 9.2761
Episode 60      Average Score: 8.05     Score: 20.155
Episode 80      Average Score: 11.13    Score: 24.621
Episode 100     Average Score: 14.96    Score: 35.352
Episode 120     Average Score: 21.76    Score: 35.259
Episode 140     Average Score: 26.87    Score: 33.939
Episode 155     Average Score: 30.15    Score: 34.476

Environment solved in 155 episodes!
```

In [9]:
```
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()
```



Summary and further optimization proposal

By increasing the number of episodes (increasing *n_episodes* from *1000* to *2000*) and by finetuning the hyperparameters (reducing L2 weight decay to zero and reducing the actor learning rate *LR_Actor* to *1e-4*) the DDPG algorithm was eventually successful in solving the environment after 155 episodes.

To further enhance the accuracy of the DDPG agents, I would recommend implementing additional optimization techniques, such as Trust Region Policy Optimization (TRPO) and Truncated Natural Policy Gradient (TNPG), as discussed in the following paper on ["Benchmarking Deep Reinforcement Learning for Continuous Control"], as well as the recent Distributed Distributional Deterministic Policy Gradients (D4PG) algorithm.

In [10]:
```
env.close()
```

In [ ]: