Project 2: Continuous Control Report

Name: Arun Bala Subramaniyan

Introduction:

In this report, the methodology used for the continuous control project is explained, where the goal is to train an agent to reach the goal for every time step in the environment. The environment is built with Unity and the complete environment information is provided below. There are two versions of the environment available, and version 1 with a single agent is used for this project.

```
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: , , ,
```

Learning Algorithm:

The DDPG algorithm explained in [1] is used to solve the problem to achieve an average score of +30 over 100 episodes. The actor and critic network are implemented in pytorch. The network is similar to the one provided from udacity's DDPG implementation, in addition to that, batch normalization [1] is added with modified network size as shown below.

Actor Network:

```
        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.bn1 = nn.BatchNorm1d(fc1_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_(-3e-3, 3e-3)

    def forward(self, state):
        """Build an actor (policy) network that maps states -> actions."""
        if state.dim() == 1:
            state = torch.unsqueeze(state,0)
        x = F.relu(self.fc1(state))
        x = self.bn1(x)
        x = F.relu(self.fc2(x))
        return F.tanh(self.fc3(x))
```

Critic Network:

```
        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.bn1 = nn.BatchNorm1d(fcs1_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_(-3e-3, 3e-3)

    def forward(self, state, action):
        """Build a critic (value) network that maps (state, action) pairs -> Q-values."""
        if state.dim() == 1:
            state = torch.unsqueeze(state,0)
        xs = F.relu(self.fcs1(state))
        xs = self.bn1(xs)
        x = torch.cat((xs, action), dim=1)
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

The value for fc1_units, fcs1_units and fc2_units are all chosen to be 128. The values of hyperparameters used for training are shown as follows:

```
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 128        # 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
EPS = 1
```

Algorithm Explanation:

At first, the algorithm starts by collecting samples from the environment using random actions. Initially, the network weights are randomly assigned and not updated till sufficient samples are available (ie., atleast 128 samples are required to update). There is an experience replay buffer with size of 100000 initialized. All the samples are stored in this buffer (deque) and will be used to update the network parameters. A batch size of 128 is passed to the network, randomly obtained from the experience replay buffer. This random sampling allows to eliminate the correlation between samples, the same strategy used by DQN algorithm.

We have two networks defined, i) the local network and ii) target network for both actor and critic. The neural network layers and the corresponding activation functions like RELU and tanh functions are shown in the previous figures (in page 2). The actor network is fed with the state space of 33 variables to input layer and then the last layer gives 4 outputs, corresponding to the action values. The critic network is fed with the state space of 33 variables as well as 4 actions, which provides the Q value for the corresponding action value pair. The idea is that the best possible action is determined by the actor network which in turn is used to estimate the Q value using the critic network. A major challenge of learning in continuous action spaces is exploration. An advantage of off-policy algorithms such as DDPG is that we can treat the problem of exploration independently from the learning algorithm. The exploration policy was constructed by adding noise sampled from Ornstein-Uhlenbeck process to the action values from actor network . This addition of noise to the process gets decayed by a factor of 0.000001 to balance the exploration-exploitation trade off.

Initially, the local network parameters were updated for every timestep and a soft update is made on the target network parameters, ie., {tau*local parameter data + (1.0-tau)*target parameter data}. The value of tau is chosen to be 0.001. But the learning was too slow and the algorithm was not stable. Hence, the number of updates per time step was changed to be less aggressive. In particular, instead of updating the actor and critic networks at every timestep, the code is amended to update the networks 5 times after every 30 timesteps. The corresponding code snippet is shown below.

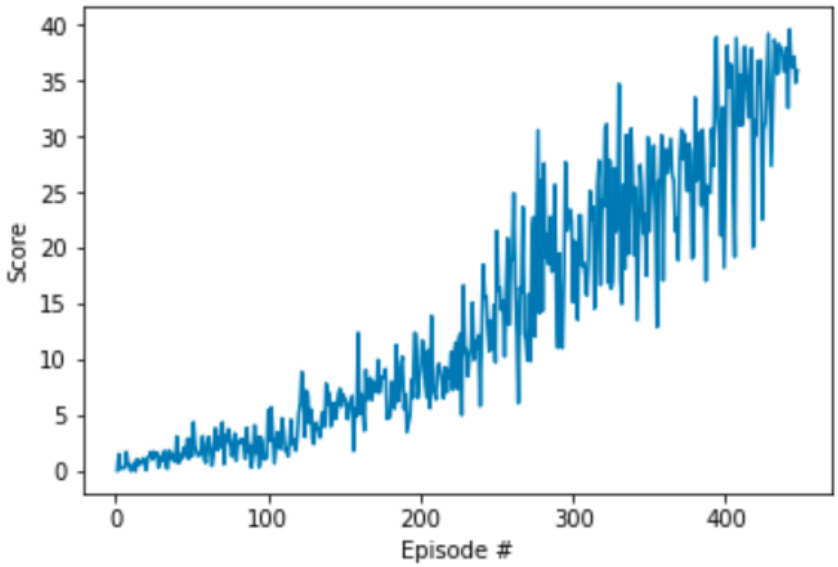
```
# Learn, if enough samples are available in memory
if len(self.memory) > BATCH_SIZE and time % 30 == 0:
    for _ in range(5):
        experiences = self.memory.sample()
        self.learn(experiences, GAMMA)
```

The optimizer used is Adam optimizer with learning rate of 0.001 for both actor and critic networks. The loss function used in this project is MSE (Mean Squared Error) for critic network, which is the difference between local and target network values and the objective is to minimize this loss. Similarly, the actor policy is updated using the sampled policy gradient as given in [1]. An small change for critic network is the use gradient clipping when training the critic network. The corresponding snippet of code is as follows:

```
# 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()
```

The problem is considered to be solved if the average reward over 100 episodes reaches a value of +30. Initially 1500 episodes were planned with maximum of 1000 timesteps. But using the hyperparameters and the given model, the problem is solved in 347 (447-100) episodes. Note that the hyperparameters can still be modified to obtain better convergence and the number of time steps for each episode can be varied to observe the learning difference. The plot of the episode number versus the agent's average score is given below.

```
Episode 100     Average Score: 1.67
Episode 200     Average Score: 5.76
Episode 300     Average Score: 14.24
Episode 400     Average Score: 24.35
Episode 447     Average Score: 30.01
```



Conclusion and Future Work:

The project solution is based on the version 1 of the environment with single agent, solved in 347 episodes. The same model was used to solve the version 2 problem with 20 agents. It was observed that the convergence was faster for multiple agents (solved within 25 episodes) but the computational effort was little bit higher. In future, a more challenging task with the crawler environment can be tried with the same model structure or modifying the existing model and hyperparameters. In addition, other algorithms like PPO, D4PG could be tried and the results can be compared, especially for multiple agents.

References:

[1] Lillicrap, T, P., et al. Continuous Control with Deep Reinforcement Learning, In arXiv:1509.02971v5 [cs.LG], 29 Feb 2016.