Project 2: Continuous Control

The project demonstrates how policy-based methods can be used to learn the optimal policy in a model-free Reinforcement Learning setting using a Unity environment, in which 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 the 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 is a number between -1 and 1. An agent choosing actions randomly can be seen in motion below:

The following report is written in four parts:

- **Implementation**
- **Results**
- **Ideas for improvement**

Implementation

The basic algorithm lying under the hood is an actor-critic method. Policy-based methods like REINFORCE, which use a Monte-Carlo estimate, have the problem of high variance. TD estimates used in value-based methods have low bias and low variance. Actor-critic methods marry these two ideas where the actor is a neural network which updates the policy and the critic is another neural network which evaluates the policy being learned which is, in turn, used to train the actor.

In vanilla policy gradients, the rewards accumulated over the episode is used to compute the average reward and then, calculate the gradient to perform gradient ascent. Now, instead of the reward given by the environment, the actor uses the value provided by the critic to make the new policy update.

Deep Deterministic Policy Gradient (DDPG) lies under the class of Actor Critic Methods but is a bit different than the vanilla Actor-Critic algorithm. The actor produces a deterministic policy instead of the usual stochastic policy and the critic evaluates the deterministic policy. The critic is updated using the TD-error and the actor is trained using the deterministic policy gradient algorithm.

$$\begin{aligned}\nabla_{\theta^\mu} J &\approx \mathbb{E}_{s_t\sim\rho^\beta}\left[\nabla_{\theta^\mu} Q(s,a|\theta^Q)|_{s=s_t,a=\mu(s_t|\theta^\mu)}\right]\\ &= \mathbb{E}_{s_t\sim\rho^\beta}\left[\nabla_a Q(s,a|\theta^Q)|_{s=s_t,a=\mu(s_t)}\nabla_{\theta_\mu}\mu(s|\theta^\mu)|_{s=s_t}\right]\end{aligned}$$

Since we are dealing with 20 agents, I went ahead with updating the weights after every 20 steps and for every such step, updating the weights 10 times. There are also a few techniques which contributed significantly towards stabilizing the training:

- **Fixed targets**: Originally introduced for DQN, the idea of having a fixed target has been very important for stabilizing training. Since we are using two neural networks for the actor and the critic, we have two

targets, one for actor and critic each.
- **Soft Updates**: In DQN, the target networks are updated by copying all the weights from the local networks after a certain number of epochs. However, in DDPG, the target networks are updated using soft updates where during each update step, 0.01% of the local network weights are mixed with the target networks weights, i.e. 99.99% of the target network weights are retained and 0.01% of the local networks weights are added.
- **Experience Replay**: This is the other important technique used for stabilizing training. If we keep learning from experiences as they come, then we are basically observed a sequence of observations each of which are linked to each other. This destroys the assumption of the samples being independent. In ER, we maintain a Replay Buffer of fixed size (say N). We run a few episodes and store each of the experiences in the buffer. After a fixed number of iterations, we sample a few experiences from this replay buffer and use that to calculate the loss and eventually update the parameters. Sampling randomly this way breaks the sequential nature of experiences and stabilizes learning. It also helps us use an experience more than once.

All of the above mentioned techniques were incorporated. The entire implementation was done in PyTorch.

Also, in my experience, I have found Batch normalization to have always improved training and hence, I added one Batch normalization layer in both actor and critic. Upon trying out both ReLU and Leaky ReLU, I found the latter to work better and hence,

Hyperparameters

You can find the actor-critic logic implemented as part of the `Agent()` class in `python/ddpg_agent.py` of the source code. The actor-critic models can be found via their respective `Actor()` and `Critic()` classes in `python/model.py`.

```
# 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)
```

There were many hyperparameters involved in the experiment contributes significantly towards getting the right results:

Learning Interval

In the first few versions of my implementation, the agent performed the learning step at every timestep. This made training very slow, and there was no apparent benefit to the agent's performance. So, I implemented an interval in which the learning step is only performed every 20 timesteps. As part of each learning step, the algorithm samples experiences from the buffer and runs the `Agent.learn()` method 10 times.

```
Update interval = 20        # learning timestep interval
Update times per interval = 10          # number of learning passes
```

You can find the learning interval implemented in the `Agent.step()` method in `python/ddpg_agent.py` of the source code.

Gradient Clipping

You can find gradient clipping implemented in the "update critic" section of the `Agent.learn()` method, within `python/ddpg_agent.py` of the source code.

Note that this function is applied after the backward pass, but before the optimization step.

```
# 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()
```

Batch Normalization

You can find batch normalization implemented for the actor, and for the critic, within `python/model.py` of the source code.

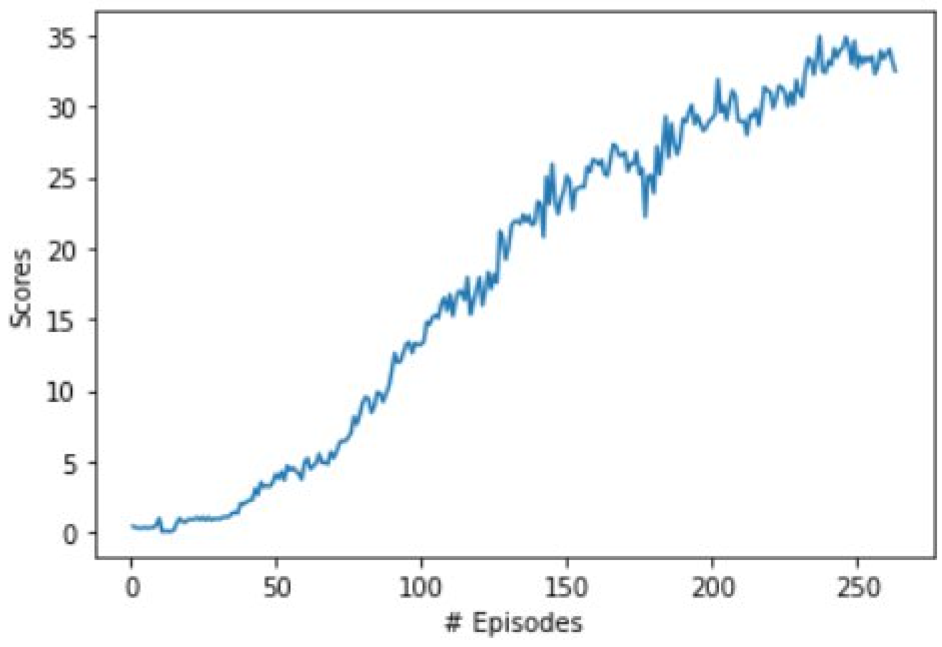
```
def forward(self, state):
    """Build an actor (policy) network that maps states -> actions."""
    state = self.bn(state)
    x = F.leaky_relu(self.fc1(state), negative_slope=self.leak)
    x = F.leaky_relu(self.fc2(x), negative_slope=self.leak)
    x =  torch.tanh(self.fc3(x))
    return x
```

```
def forward(self, state, action):
    """ Build a critic (value) network that maps (state, action) pairs -> Q-
values."""
    state = self.bn(state)
    x = F.leaky_relu(self.fcs1(state), negative_slope=self.leak)
    x = torch.cat((x, action), dim=1)
    x = F.leaky_relu(self.fc2(x), negative_slope=self.leak)
    x = F.leaky_relu(self.fc3(x), negative_slope=self.leak)
    x =  self.fc4(x)
    return x
```

Result

The best performance was achieved by **DDPG** where the reward of +30 was achieved in **163** episodes with hyperparameter and rewards across episodes below:

Hyperparameter	**Value**
Replay buffer size	1e6
Batch size	1024
$\gamma$ (discount factor)	0.99
$\tau$	1e-3
Actor Learning rate	1e-4
Critic Learning rate	3e-4
Update interval	20
Update times per interval	10
Number of episodes	500
Max number of timesteps per episode	1000
Leak for LeakyReLU	0.01



Ideas for improvement

- **Add *prioritized* experience replay** — Rather than selecting experience tuples randomly, prioritized replay selects experiences based on a priority value that is correlated with the magnitude of error. This can improve learning by increasing the probability that rare and important experience vectors are sampled.

- **Experiment with other algorithms** — Tuning the DDPG algorithm required a lot of trial and error. Perhaps another algorithm such as Trust Region Policy Optimization (TRPO), [Proximal Policy Optimization (PPO)](Proximal Policy Optimization Algorithms), or Distributed Distributional Deterministic Policy Gradients (D4PG) would be more robust.
- The Q-prop algorithm, which combines both off-policy and on-policy learning, could be good one to try.
- General optimization techniques like cyclical learning rates and warm restarts could be useful as well.