Project 2 – Continuous Control

May 14, 2020

1 Introduction and objectives

For this project, An agent is trained to move a double-jointed arm to target locations. Considering the *first version*, the primary objective is to **get an average score of** $+30$ **over** 100 **consecutive episodes**. The *Deep Deterministic Policy Gradient (DDPG)* and *Twin Delayed Deep Deterministic (TD3)* are utilized for the task. The project is completed in the *Udacity Workspaces*.

2 Environment

An agent is trained to move a double-jointed arm 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 should be a number between -1 and $+1$.

3 Deep Deterministic Policy Gradient

The DDPG algorithm is performed to train the agent as follows:

Algorithm 1: Deep Deterministic Policy Gradient

Initialize critic network $Q(s, a|\theta)$with weight θ.
Initialize actor network $\pi(s|\phi)$ with weight ϕ.
Initialize corresponding target network $\theta' \leftarrow \theta$, $\phi' \leftarrow \phi$.
Initialize replay buffer B.
Initialize an empty list $scores = [\,]$.
for $t \leftarrow 1$ **to** *number of episode+1* **do**
    $score \leftarrow 0$
    for $t \leftarrow 0$ **to** *number of step-1* **do**
        Execute action $A_t \leftarrow \pi(S_t|\phi) + \epsilon$ based on current policy and exploration noise
        Observe reward R_t and state S_{t+1}.
        Store transition (S_t, A_t, R_t, S_{t+1}) in B.
        Sample minibatch of N transition (S_t, A_t, R_t, S_{t+1}) from B
        Evaluate target $\tilde{Q} \leftarrow R_t + \gamma\left[Q'\left(S_{t+1}, \pi'\left(S_{t+1}|\phi'\right)|\theta'\right)\right]$.
        Compute gradient descent by minimizing $N^{-1}\sum\left(\tilde{Q} - Q(S_t, A_t)\right)^2$.
        Update critic network θ.
        Update ϕ by deterministic policy gradient
        $\nabla_\phi J = N^{-1}\sum \nabla_a Q_1\left(S_t, A_t|\theta_1\right)|_{a=\pi_\phi(S_t)} \nabla_\phi \pi_\phi\left(S_t\right)$.
        Update target network $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$, $\phi' \leftarrow \tau\phi + (1-\tau)\phi$.
        $score \leftarrow score + R_t$ and append to the list $scores$.
    end
    Noise decay $\nu = \max(\nu_{\text{end}}, \nu_{\text{decay}}\epsilon)$
end
return *scores*

The local and target of actor network and critic network are constructed using 2 hidden fully connected layer neural network. The loss is minimized by using the Adaptive Moment Estimation (Adam) optimizer:

Code Listing 1: Actor network and critic network

```
# 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)
```

The model architecture for the actor networks which consist of 2 hidden fully connected layer neural network with Relu activation function, a batch normalization, and hyperbolic tangent at the output layer:

Code Listing 2: actor network architecture

```
def __init__(self, state_size, action_size, seed, fc1_units=128, fc2_units=128):
  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)

def forward(self, state):
  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))
```

The model architecture for the critic networks which consist of 2 hidden fully connected layer neural network with Relu activation function, a batch normalization, and linear activation function at the output layer:

Code Listing 3: critic network architecture

```
def __init__(self, state_size, action_size, seed, fcs1_units=128, fc2_units=128):
  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 forward(self, state, action):
  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 action is evaluated based on policy perturbed by exploration noise with decay ν:

Code Listing 4: Action based on policy with exploration noise

```
def act(self, state, add_noise=True, nu = 1.0):
  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 += nu*self.noise.sample()
  return np.clip(action, -1, 1)
```

4 Twin Delayed Deep Deterministic

To alleviate the overestimation of the value function in DDPG. The TD3 algorithm is performed to train the agent as follows:

Algorithm 2: Twin Delayed Deep Deterministic

Initialize critic network $Q_1(s, a|\theta_1)$, $Q_2(s, a|\theta_2)$ with weight θ_1 and θ_2, respectively.
Initialize actor network $\pi(s|\phi)$ with weight ϕ.
Initialize corresponding target network $\theta_1' \leftarrow \theta_1$, $\theta_2' \leftarrow \theta_2$, $\phi' \leftarrow \phi$.
Initialize replay buffer B.
Initialize an empty list $scores = [\,]$.
for $t \leftarrow 1$ **to** *number of episode+1* **do**
    $score \leftarrow 0$
    for $t \leftarrow 0$ **to** *number of step-1* **do**
        Execute action $A_t \leftarrow \pi(S_t|\phi) + \epsilon$ based on current policy and exploration noise
        Observe reward R_t and state S_{t+1}.
        Store transition (S_t, A_t, R_t, S_{t+1}) in B.
        Sample minibatch of N transition (S_t, A_t, R_t, S_{t+1}) from B
        Evaluate target $\tilde{Q} \leftarrow R_t + \gamma \min\left[Q_1'\left(S_{t+1}, \pi'\left(S_{t+1}|\phi'\right)|\theta_1'\right), Q_2'\left(S_{t+1}, \pi'\left(S_{t+1}|\phi'\right)|\theta_2'\right)\right]$.
        Compute gradient descent by minimizing $N^{-1}\sum\left(\tilde{Q} - Q(S_t, A_t)\right)^2$.
        Update critic network θ_1 and θ_2.
        if $t \mod freq$ **then**
            Update ϕ by deterministic policy gradient
            $\nabla_\phi J = N^{-1}\sum \nabla_a Q(S_t, A_t|\theta_1)|_{a=\pi_\phi(S_t)} \nabla_\phi \pi_\phi(S_t)$.
            Update target network $\theta_1' \leftarrow \tau\theta_1 + (1-\tau)\theta_1'$, $\theta_2'\tau\theta_2 + (1-\tau)\theta_2'$, $\phi' \leftarrow \tau\phi + (1-\tau)\phi$.
        end
        $score \leftarrow score + R_t$ and append to the list $scores$.
    end
    Noise decay $\nu = \max(\nu_{\text{end}}, \nu_{\text{decay}}\epsilon)$
end
return *scores*

The local and target of actor network, critic network 1, and critic network 2 are constructed using 2 hidden fully connected layer neural network. The loss is minimized by using the Adaptive Moment Estimation (Adam) optimizer:

Code Listing 5: Actor network, critic network 1, and critic network 2

```
# 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_1 = Critic(state_size, action_size, random_seed).to(device)
self.critic_target_1 = Critic(state_size, action_size, random_seed).to(device)
self.critic_optimizer_1 = optim.Adam(self.critic_local_1.parameters(), lr=LR_CRITIC,
                                     weight_decay=WEIGHT_DECAY)
```

```
self.critic_local_2 = Critic(state_size, action_size, random_seed).to(device)
self.critic_target_2 = Critic(state_size, action_size, random_seed).to(device)
self.critic_optimizer_2 = optim.Adam(self.critic_local_2.parameters(), lr=LR_CRITIC,
                                     weight_decay=WEIGHT_DECAY)
```

The model architecture are identical to aforementioned DDPG. Delayed update of the target and actor networks is introduced to prevent the divergence of value estimation that subsequently jeopardizing the accuracy of policy evaluation:

Code Listing 6: Delayed update of the target and actor networks

```
if freq == 0:
  # ---------------------------- update actor ---------------------------- #
  # Compute actor loss
  actions_pred = self.actor_local(states)
  actor_loss = -self.critic_local_1(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_1, self.critic_target_1, TAU)
  self.soft_update(self.critic_local_2, self.critic_target_2, TAU)
  self.soft_update(self.actor_local, self.actor_target, TAU)
```

5 Training

After some hyperparameter searches, the following parameters are used to train the agent:

128, and 128 nodes of first, and second fully connected layer, respectively;

learning rate of all networks $= 2.5e-4$, noise decay $= 0.995$;

replay buffer size $= 100000$, minibatch size $= 128$, discount factor $= 0.99$;

soft update of target parameters $= 0.001$, maximum step of 1000;

Ornstein-Uhlenbeck noise ($\sigma = 0.1$, $\nu_{\text{end}} = 0.01$, $\nu_{\text{decay}} = 0.995$).

Code Listing 7: DDPG

```
agent_ddpg = DDPG_Agent(state_size=state_size, action_size=action_size, random_seed=7)
ddpg_scores = ddpg(agent_ddpg,n_episodes=500, max_t=1000)

Episode: 50, Score: 6.49, Average: 2.99, Max: 8.44,
Episode: 100, Score: 12.01, Average: 6.02, Max: 18.03,
Episode: 150, Score: 17.56, Average: 11.92, Max: 29.15,
Episode: 200, Score: 37.05, Average: 21.05, Max: 39.33,

Episode: 248, Score: 39.32, Average: 30.11, Max: 39.54,
Environment solved in 148 episodes! Average Score: 30.11
```

For TD3, in addition to delayed update (TD3) = 2, the identical parameters are applied:

Code Listing 8: Double deep Q-network training

```
agent_td3 = TD3_Agent(state_size=state_size, action_size=action_size, random_seed=7)
td3_scores = td3(agent_td3,n_episodes=500, max_t=1000,freq=2)

Episode: 50, Score: 4.13, Average: 1.99, Max: 10.96,
Episode: 100, Score: 20.26, Average: 6.50, Max: 35.09,
Episode: 150, Score: 36.44, Average: 21.58, Max: 39.38,

Episode: 180, Score: 39.55, Average: 30.14, Max: 39.59,
Environment solved in 80 episodes!  Average Score: 30.14
```

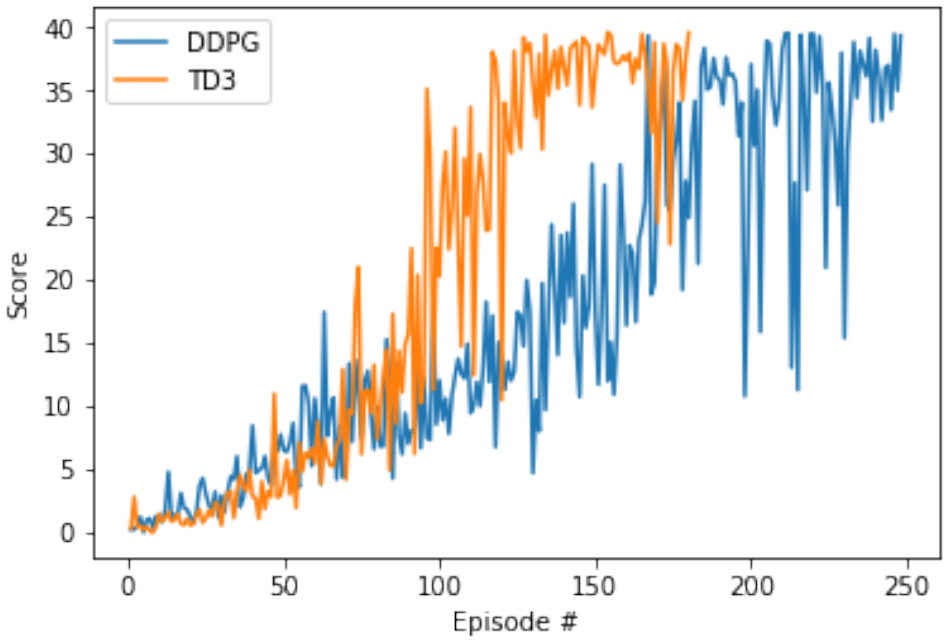

Figure 1: Average score comparison of DDPG and TD3

As a result, the DDPG **solved in 148 episodes with an average score of +30.11 over 100 consecutive episodes**. The TD3 performed better and **solved in 80 episodes with an average score of +30.14 over 100 consecutive episodes** as shown in Figure 1.

6 Test the agent

Finally, the TD3 agent is tested against the environment over 10 episodes:

Code Listing 9: Test the TD3 agent

```
scores = []
scores_window = deque(maxlen=100)
  for i_episode in range(1, 10+1):
  state = env.reset(train_mode=False)[brain_name].vector_observations[0]
  score = 0
  for t in range(1000):
    action = agent_td3.act(state)
    env_info = env.step(action)[brain_name]
    next_state = env_info.vector_observations[0]
    reward = env_info.rewards[0]
    done = env_info.local_done[0]
    state = next_state
    score += reward
    if done:
    break
  scores_window.append(score)
  scores.append(score)
  print('\rEpisode {}\tScore: {:.2f}'.format(i_episode, scores[-1]))

Episode 1 Score: 32.90
Episode 2 Score: 37.65
Episode 3 Score: 39.27
Episode 4 Score: 38.30
Episode 5 Score: 37.76
Episode 6 Score: 38.77
Episode 7 Score: 33.99
Episode 8 Score: 33.51
Episode 9 Score: 35.42
Episode 10  Score: 38.88
```

Overall, the performance is as expected (average over +30). We discuss some future works to improve the agent in the following section.

7 Future work

1. **Hyperparameter search**: There are still many combinations of hyperparameter to be explored. This study focused on tuning the number of nodes in neural network, replay buffer size and noise decayν. Other parameters remain unexplored are minibatch size, discount factor, soft update of target parameters, and delayed update of TD3 networks.

2. **Hindsight Experience Replay**: Dealing with sparse rewards is one of the biggest challenges in this project. Hindsight Experience Replay facilitate sample-efficient learning from rewards which are sparse and binary. It can be combined with the present off-policy algorithms.

3. **Distributed Distributional DDPG (D4PG)**: D4PG applies a set of improvements on DDPG to make it run in the distributional fashion. The key ingredients include Distributional Vritic, N-step returns, and Multiple Distributed Parallel Actors.