The “Reacher” Agent

Introduction

The agent uses the DDPG (deep deterministic policy gradient) algorithm for reinforcement learning. A thorough description of the algorithm’s principles can be found here:

- Deep Deterministic Policy Gradient
https://spinningup.openai.com/en/latest/algorithms/ddpg.html

- or in this publication: Continuous control with deep reinforcement learning
https://arxiv.org/abs/1509.02971.

Students could choose between a one-agent and a multi-agent environment. I have used the one-agent environment in my solution.

DDPG is an off-policy algorithm and can be used for environments with continuous action spaces. It can also be thought of being deep Q-learning for continuous action spaces. Instead of estimating an optimal Q value function for each state and action pair DDPG directly maps a state to an action through a deep neural network. DDPF also qualifies as an actor – critic algorithm. The actor is associated with the Q learning part and the critic is associated with the policy learning part. Overall four neural networks are used: a Q network, a target Q network, a deterministic policy network, and a target policy network.

The Q-learning side of DDPG: The goal is to find the optimal $Q_\phi(s,a)$ solution for a given state s and actions a. Q_ϕ is approximated by a deep neural network. It is assumed that a set of state-action-reward-new state pairs of the form (s,a,r,s’) have been accumulated in Ω. A mean-squared Bellman error function (MSBE) is set up to estimate how closely Q_ϕ comes to satisfying the Bellmann equation. The objective is to minimize MSBE.

Convergence for MSBE minimization is often difficult to achieve and two tricks can be employed: replay buffers and target networks. If one uses only the very-most recent data, overfitting will resilt; if one uses too much experience, learning will be slow. Minimizing MSBE through ϕ means that the optimization target depends on the same parameters ϕ which we try to train. This makes the process unstable. The solution is to use a set of parameters which comes close to ϕ, but with a time delay – this is achieved by a second network, called the target network, which lags the first.

The policy learning side of DDPG: a deterministic policy μ_θ (s) is determined which gives the action that maximizes $Q_\phi(s,a)$. Gradient ascent with respect to the policy parameters θ is performed by maximizing the expectation $\mathrm{E}(Q_\phi(s, \mu_\theta (s)))$ with respect to θ and leaving the Q parameters ϕ constant.

Implementation

(1) The Actor network
The Actor uses a three layer deep neural networks connected through RELUs and a final tanh function to map the network’s output to the continuous action space which accepts values from the interval [-1,+1]. The first network layer is the input layer with size equal the observations space (i.e. 33), the second layer has 256 units and the third layer has 128 units. The output is a single unit.

```
class Actor(nn.Module):
    """Actor (Policy) Model."""
    def __init__(self, state_size, action_size, seed, fc1_units=256, 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))
```

(2) The Critic network

The Critic uses a four layer deep neural networks connected through RELUs and a final linear output function. The first network layer, the input layer has the site of the observation space (i.e. 33). The second layer combines the output of the first layer (256 units) with the action space size (i.e. 4 units). The third layer has 256 units and the fourth 128. The output is a single linear unit. The admixing of the action space into the critic network at its second layer has the purpose of adding the policy μ_θ (s) into the network.

```
class Critic(nn.Module):
    """Critic (Value) Model."""
    def __init__(self, state_size, action_size, seed, fcs1_units=256, fc2_units=256, fc3_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
            fc3_units (int): Number of nodes in the third 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, fc3_units)
        self.fc4 = nn.Linear(fc3_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_(*hidden_init(self.fc3))
        self.fc4.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.leaky_relu(self.fcs1(state))
        x = torch.cat((xs, action), dim=1)
        x = F.leaky_relu(self.fc2(x))
        x = F.leaky_relu(self.fc3(x))
        return self.fc4(x)
```

(3) The DDPG loop
The objective is to run 1000 episodes with a maximum of 1000 steps per episode. Starting with a given state the proposed action from the Q network is calculated by ‘agent.act’ and the new status is observed through ‘env.step’, the rewards are calculated by ‘env_info.rewards’. With the call ‘env_info.local_done’ a check is performed if the episode is finished. Finally the agent performs a learning step by calling ‘agent.step’ which updates the Q network and the policy network through the (states, actions, rewards, next_state, done) information. While the update of the networks is done at every step the learning is only performed every hundredth step in order to avoid too much dependence on short-term noise.

```
agent = Agent(state_size=states.shape[1], action_size=brain.vector_action_space_size, random_seed=10)

def ddpg(n_episodes=1000, max_t=1001):
    scores_deque = deque(maxlen=100)
    scores = []
    max_score = -np.Inf
    for i_episode in range(1, n_episodes+1):
        env_info = env.reset(train_mode=True)[brain_name]
        states = env_info.vector_observations
        #state = env.reset()
        agent.reset()
        score = 0
        zeit=time.time()
        for t in range(max_t):
            actions       = agent.act(states,add_noise=False)
            env_info     = env.step(actions)[brain_name]
            next_states  = env_info.vector_observations
            rewards      = env_info.rewards
            dones        = env_info.local_done
            agent.step(states, actions, rewards, next_states, dones)
            states = next_states
            score += rewards[0]
            if t%100 == 0:
                agent.do_learn()
            if np.any(dones):
                break

        scores_deque.append(score)
        scores.append(score)
        zeit=time.time()-zeit
        if i_episode % 100 == 0:
            torch.save(agent.actor_local.state_dict(), 'checkpoint_actor.pth')
            torch.save(agent.critic_local.state_dict(), 'checkpoint_critic.pth')
            print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_deque)))
    return scores
```

(4) The Agent

The agent initializes four neural networks: a Q and a Q-target network and a policy and a policy-target network. The Q networks are instantiated from class “Actor” and the policy networks are instantiated from class “Critic”. Also, a replay buffer is initiated upon creation of an agent.

```
def __init__(self, state_size, action_size, 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)

        # Replay memory
        self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE, random_seed)
```

The weights of the Q-network, i.e. the actor, are updated by calculating a loss function ‘actor_loss’ by means of the policy, aka critic, network:

```
# ---------------------------- 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()
```

The policy network is updated by calculating the MSBE and minimizing it:

```
        # ---------------------------- 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()
        self.critic_optimizer.step()
```

Finally, an update of the target networks is performed by mean of 'soft_update'.

```
        # ----------------------- update target networks ----------------------- #
        self.soft_update(self.critic_local, self.critic_target, TAU)
        self.soft_update(self.actor_local, self.actor_target, TAU)

    def soft_update(self, local_model, target_model, tau):
        """
        Soft update model parameters.
        θ_target = τ*θ_local + (1 - τ)*θ_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)
```

Results

A reward of +0.1 is provided for each timestep that the agent's hand is in the target sphere. The goal is to keep the agent's arm position at the target for as many time steps as possible.

The environment is considered solved, if the agent achieves a score of +30 averaged for 100 consecutive episodes.

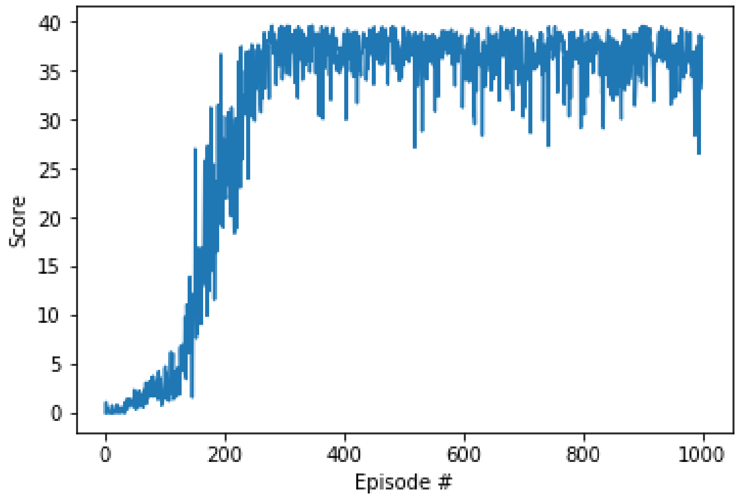
Hyperparameters used are:

- replay buffer size = 1e6
- minibatch size = 1024
- discount factor = 0.99
- tau for soft update of target parameters = 1e-3
- learning rate of the actor = 1e-3
- learning rate of the critic = 1e-3
- L2 weight decay = 0.0

The following scores were obtained:

```
Episode 100      Average Score: 1.29      Score: 4.47      Duration: 6.89
Episode 200      Average Score: 12.22     Score: 23.19     Duration: 7.29
Episode 300      Average Score: 33.00     Score: 37.14     Duration: 8.07
Episode 400      Average Score: 37.15     Score: 38.84     Duration: 9.03
Episode 500      Average Score: 37.20     Score: 39.46     Duration: 10.29
Episode 600      Average Score: 37.11     Score: 34.99     Duration: 11.13
Episode 700      Average Score: 36.30     Score: 30.27     Duration: 12.29
Episode 800      Average Score: 36.51     Score: 38.62     Duration: 13.42
Episode 900      Average Score: 36.04     Score: 39.60     Duration: 14.60
Episode 1000     Average Score: 36.41     Score: 38.45     Duration: 15.50
```

After 300 episodes the agent obtained already a score >30 and remained close to this performance thereafter. The following plot shows the full score history per episode.



Further improvements

Future improvements and a valuable learning experience would be to try out other algorithms:

- Trust region policy optimization (TRPO): https://arxiv.org/abs/1502.05477
- Proximal policy optimization (PPO): https://arxiv.org/abs/1707.06347
- Distributed distributional deterministic policy gradients (D4PG): https://arxiv.org/abs/1804.08617
- Asynchronous optimization methods (A2C): https://arxiv.org/abs/1602.01783
- Prioritized experience replay: https://arxiv.org/abs/1511.05952