Udacity Deep Reinforcemente Learning Nanodegree Continuous Control Project

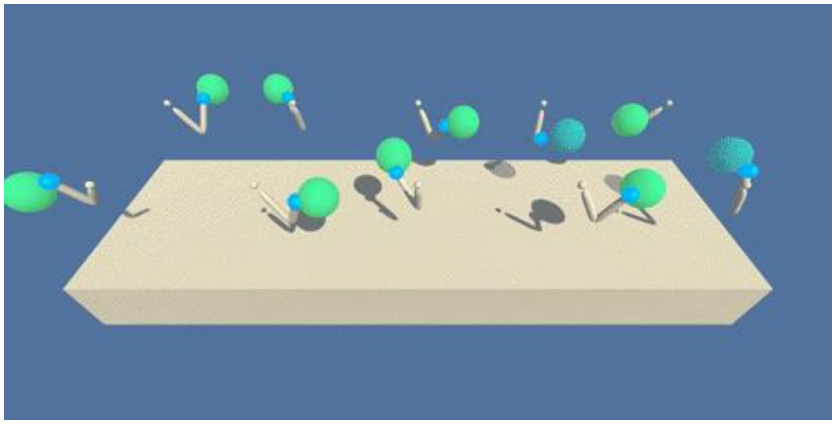
Udacityuser4
May 03rd, 2020

Abstract

The goal of this project is training a reinforcement learning agent that is a double-jointed arm to move to target locations. Thus, the goal of the agent is to maintain its position at the target location for as many time steps as possible.

1. DEFINITION

This project uses the Reacher Unity Environment and version 1 that contains only one agent.



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 should be a number between -1 and 1.

It is considered solved when the agent is able to achieve an average score of 30 or above during 100 consecutives episodes.

2. APPROACH

2.1 State and action space

The state space has 33 dimensions corresponding to the position, rotation, velocity and angular velocities of the robotic arm. There are two sections of the arm, analogous to those connecting the shoulder and elbow on a human body.

Each action is a vector containing for numbers, corresponding to the torque applied to the two joints. Every element in the action must be a number between -1 and 1, making the action space continuous.

There is a block of code testing an agent that selects actions randomly at every time step. Running this way the results are random around 0, so we need a better approach as we need a result of 30 score over 100 consecutives episodes.

2.1 Learning Algorithm

First of all I've chosen the alternative with one joint arm, thinking that it was easier. After working on that, I think the other option with 20 joint arms maybe it's faster, as it is like a big brain composed by several little collaborative brains.
We will use a policy –based method, as this is a continuous action space, with stochastic policies.
Also the policy-based methods learn the optimal policy without maintaining a separate function estimate.

So following the recommendations for this project, I have used the Deep Deterministic Policy Gradient (DDPG) (https://arxiv.org/abs/1509.02971)

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

2.1 Actor-Critic Method

Actor-Critic methods combine the strengths of policy-based and value-based methods.
Using a policy-based approach, the agent (actor) learns how to act by estimating the optimal policy and maximizing reward through gradient ascent.
With a value-based approach, the agent (critic) learns how to estimate the value. Combining those two methods, we can accelerate the learning process.
-->The Agent is implemented on the agent.py file.
-->The Actor and Critic are implemented on the model.py file.

At the beginning I followed exactly the numbers for the hyperparameters of the paper but the learning process was extremely slow. So after reading some tips from the mentors, I decided to change the network of the actor to 256 for the first layer and 128 for the second layer, with two ReLu activations for the forward step and finishing with a tanh, as the paper recommends.

For the Critic I used also two layers network with 256 for the first layer and 128 for the second with two Relu activations for the forward step.

2.3 Neural Networks

This is the network of the Actor:

```
class Actor(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed=0, 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."""
     #   if state.dim() == 1:
     #       state = torch.unsqueeze(state,0)
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return F.tanh(self.fc3(x))

```

This is the network for the critic

```
class Critic(nn.Module):
    """Critic (Value) Model."""

    def __init__(self, state_size, action_size, seed=0, 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
            fcs1_units (int): Number of nodes in the first hidden layer
            fc2_units (int): Number of nodes in the second hidden layer
        """
        super(Critic, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units+action_size, fc2_units)
        self.fc3 = nn.Linear(fc2_units, 1)
        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, 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.fc1(state))
        x = torch.cat((xs, action), dim=1)
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

2.4 Recommendations from Udacity mentors

Also I followed the recommendations of the mentors about:

1. Clean the notebook's kernel after each training, as I have experimented several strange errors and kernel disconnections.
2. Use the following parameters
    - UPDATE_EVERY = 5
    - UPDATE_TIMES = 20

3. Use 256x256/ 256x128 for the neural networks for the actor and critic.
4. Reducing the sigma parameter in OU noise to 0.1 or even a bit less.
5. Adding a check in the step function to only learn once every 5-10 steps, and then when it is time to learn, call the learn function several times (say, 4-8 times).
6. Add a call to pytorch's clip_grad_norm function to your learn method.

The recommendation number 5 has been critical for the model to be faster to learn.

```

    def step(self, time_step, state, action, reward, next_state, done):
   # def step(self, state, action, reward, next_state, done):
        """Save experience in replay memory, and use random sample from buffer to learn."""
        # Save experience / reward
        self.memory.add(state, action, reward, next_state, done)

        # add time_step
        self.t += 1  # total time step counter

        # Learn, if enough samples are available in memory
        if len(self.memory) > BATCH_SIZE:
            if self.t % self.n_learn_updates == 0:
                for i in range(N_LEARN_UPDATES):
                    experiences = self.memory.sample()
                    self.learn(experiences, GAMMA)

```

2.5 Noise

The Ornstein-Uhlenbeck process adds an amount of noise to the action values at each timestep. This noise is correlated to previous noise and therefore stay in the same direction for longer duration without cancelling itself out. This allows the arm to keep speed and explore the action space with continuity.

2.6 Gradient Clipping

I have implemented gradient clipping with the corresponding torch function. I set the function to "clip" the norm of the gradients at 1, therefore placing an upper limit on the size of the parameter updates, and preventing them from growing exponentially. Once this change was implemented), my model became much more stable and my agent started to learn faster. I've implemented that function on the agent.py file, on the learning part of the critic.

```
111
112         # ---------------------------- update critic ---------------------------- #
113         # Get predicted next-state actions and Q values from target models
114         actions_next = self.actor_target(next_states)
115         Q_targets_next = self.critic_target(next_states, actions_next)
116         # Compute Q targets for current states (y_i)
117         Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))
118         # Compute critic loss
119         Q_expected = self.critic_local(states, actions)
120         critic_loss = F.mse_loss(Q_expected, Q_targets)
121         # Minimize the loss
122         self.critic_optimizer.zero_grad()
123         critic_loss.backward()
124         torch.nn.utils.clip_grad_norm_(self.critic_local.parameters(), 1)
125         self.critic_optimizer.step()
126
```

3. HYPERPARAMETERS

These are the rest of the hyperparameters used within this implementation:

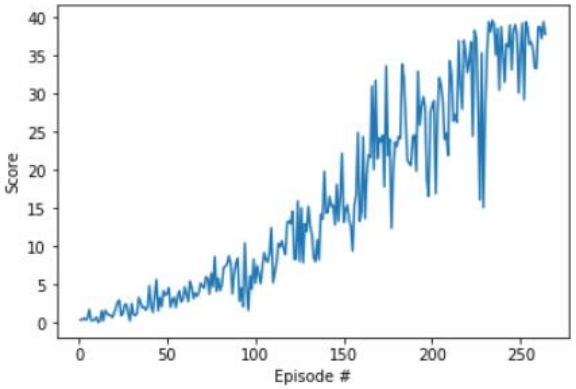
BUFFER SIZE	100.000
BATCH SIZE	128
GAMMA (discount factor)	0.99
TAU (soft update of target parameters)	0.001
LR_ACTOR	1e-4
LR_CRITIC	1e-4
LEARNING RATE	0.0005
UPDATE_RATE	Every 4 steps
TARGET SCORE	30
WEIGHT_DECAY	0
THETA	0.15
SIGMA	0.01

Table 1: Hyperparameters

4. PLOT OF REWARDS

These is the plot of rewards

```
Episode 100 (00m13s)    Average Score: 3.37
Episode 200 (00m13s)    Average Score: 16.97
Episode 264 (00m13s)    Average Score: 30.07
Environment solved in 264 episodes!     Average Score: 30.07    Elapsed time: 00h58m48s
```



So the agent finally reached the solution after **264 episodes**.

5. RESULTS

5.1Interpretation

After reading the paper suggested by the "Not sure where to start?" part, I started implementing all the hyperparameters exactly as the paper recommended. But when I launched the model was extremely slow and even the kernel changed to idle mode. After some tunning the model started to run but ended with 1000 episodes and not reaching the solution.

So I started researching and after reading the recommendation from the mentors on the "Mentor Help" section, I changed some hyperparameters and the model started to learn faster and reached the 30 average scores soon.

I really don't like to wait several hours to train an agent, as sometimes it ends up failing and I find this really frustrating.

But after doing some research and readings, I am really proud of the final Plot of Rewards my agent has reached. **My agent solved the environment in 264 episodes**.

6. IDEAS FOR IMPROVEMENT

There are several possibilities to evolution this project that I will list:

1. More tuning of hyperparameters. Although I have experimented changing the values of the hyperparameters, there is for sure more space for improvements (adding batch and drop layers to the networks of the actor/critic), changing the value for the noise, increasing the batch size or the frecuency update.

2. Experiment with negative reward when the agent takes an action against to keep in touch with the target location.

3. Add prioritize to the buffer replay: rather than select tuples randomly, use a priority value to select the best tuples with more TD error to improve.

4. Experiment with other algorithms like:

    - TRPO - Trust Region Policy Optimization (https://arxiv.org/abs/1502.05477)
    - GAE - Generalized Advantage Estimation (https://arxiv.org/abs/1506.02438)
    - A3C - Asynchronous Advantage Actor-Critic (https://arxiv.org/abs/1602.01783)
    - A2C - Advantage Actor-Critic
    - ACER - Actor Critic with Experience Replay (https://arxiv.org/abs/1611.01224)
    - PPO - Proximal Policy Optimization (https://arxiv.org/pdf/1707.06347.pdf)
    - D4PG - Distributed Distributional Deterministic Policy Gradients (https://arxiv.org/pdf/1804.08617.pdf)