Deep Reinforcement Learning Nanodegree

Project 2 Report - Continuous Control

Introduction

This project is the second one in Udacity's Deep Reinforcement Learning Nanodegree. In this project, The goal is to implement model based on Deep Deterministic Policy Gradient (DDPG) to control the movement of a group of 20 identical double-jointed robotic arms in a Unity ML-Agents environment Reacher.

Environment description

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 (for each agent) 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.

Task description

For successfully solving the environment, the agents must get an average score of +30 (over 100 consecutive episodes, and over all agents). Specifically,

- After each episode, we add up the rewards that each agent received (without discounting), to get a score for each agent. This yields 20 (potentially different) scores. We then take the average of these 20 scores.
- This yields an average score for each episode (where the average is over all 20 agents).

The environment is considered solved, when the average (over 100 episodes) of those average scores is at least +30.

Learning Algorithm

Deep Deterministic Policy Gradient (DDPG) is an algorithm which concurrently learns a Q-function and a policy using two networks in an (Actor-Critic) approach. It's also worth noting that each of these two networks contain two separate networks in its own, just as DQN where there is a local and a target network together making the DQN network. At each learning step, the Actor network is used to estimate the best action given the current state, and the Critic uses the current state and best action as in a DDQN to evaluate the optimal action value function which will be used back to tarin the Actor network.

Hyperparameters Tuning:

After careful monitoring of the training process using multiple sets of hyperparameters. The best set that was found was as follows:

Variable Name	Hyperparameters Description	Chosen Value
n_episodes	how many episodes to train for	300
BUFFER_SIZE	replay buffer size	1e5
BATCH_SIZE	sampling size from buffer	128
GAMMA	discount factor	0.99
TAU	Target networks soft update factor	1e-3
LR_ACTOR	Actor local network learning rate	1e-4
LR_CRITIC	Critic local network learning rate	1e-4
learn_every	time steps to update Actor and Critic networks	20

Network Architecture

For the Actor network, both the main and target networks were composed of 3 fully connected (dense) layers as follows:

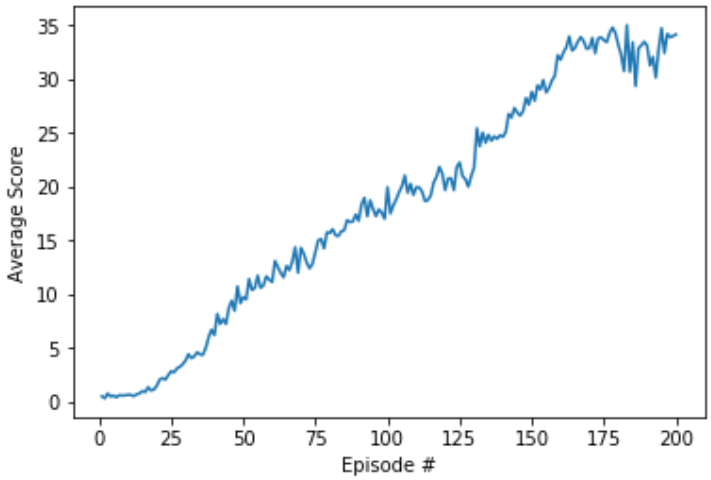
```
class Actor(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed, fc1_units=400, fc2_units=300):
        """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 torch.tanh(self.fc3(x))
```

For the Critic network, both the main and target networks were composed of 3 fully connected (dense) layers as follows:

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

    def __init__(self, state_size, action_size, seed, fcs1_units=400, fc2_units=300):
        """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.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.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."""
        xs = F.relu(self.fcs1(state))
        x = torch.cat((xs, action), dim=1)
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

Results

The training process lasted for 200 episodes, where the mean score of agents at the end of each episode has been recorded and is shown in the figure below:



The maximum average score over 100 consecutive episodes was recorded as 33 and the environment has been solved from episode number 61 to 161 with an average score of 30.32

The models weights of both Actor and Critic local networks at the max point of 33 have been saved in the *checkpoint_critic.pth* and *checkpoint_actor.pth* files in the root of the GitHub repository so it can be retrieved again after the end of training.

Future work to consider:

To improve on the current implementation, it is recommended to add following features to the above learning algorithm :

- Batch normalization.
- More tuning of hyperparameters and model architecture
- Other model such as PPO, A3C or D4PG could also achieve better results than DDPG.