Project 3: Continuous Control

Udacity Deep Reinforcement Learning Nanodegree Program

Bob Flagg

Figure 1: Designing a robotic arm. Source: SOLIDWORKS 2016

1 INTRODUCTION

In this project I'll implement a solution to the Reacher environment from the Unity Machine Learning Toolkit. The Reacher is a double-jointed arm that 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 for 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.

I'll use *Deep Deterministic Policy Gradient* [2] (DDPG) to solve the distributed version of the task which contains 20 identical agents, each with its own copy of the environment. Source code in Python, using PyTorch, is available on github in the repo Continuous-Control.

2 BACKGROUND

The Reacher is a *sequential decision making problem*, in which an agent interacts with an environment over discrete time steps and tries to find a *policy* to maximize the expected *discounted return*:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1},$$

where $\gamma \in [0, 1]$ is a discount factor that trades-off the importance of immediate and future rewards. See [3] for a general discussion of this sort of problem.

In this project I will use policy-based methods, which try to directly find an *optimal policy*, π^*, that an agent can use to decide what actions to take. The particular algorithm, *Deep Deterministic Policy Gradient* [2] (DDPG), is motivated by an important connection between the action selected by an optimal policy and the *optimal action-value function*

$$Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a).$$

Namely, if you know the optimal action-value function, then in any given state, s, an optimal action can be found by solving

$$\pi^*(s) = \arg\max_{a} Q^*(s, a).$$

DDPG concurrently learns an approximator to $Q^*(s, a)$ and an approximator to $\pi^*(s)$, and it does so in a way which is specifically adapted for environments with continuous action spaces.

Learning $Q^*(s, a)$

The starting point for approximating $Q^*(s, a)$ is the **Bellman Equation**:

$$Q^*(s, a) = \mathbb{E}\Big[R_{t+1} + \gamma \cdot \max_{a'} Q^*(S_{t+1}, a') | S_t = s, A_t = a\Big].$$

Suppose we are approximating $Q^*(s, a)$ with a neural network, $Q_\phi(s, a)$, and we have collected a set $\mathcal{D}$ of transitions (s, a, r, s', d), then the **mean-squared Bellman error** (MSBE)

$$L(\phi, \mathcal{D}) = \mathbb{E}_{\mathcal{D}}\Big[\Big(Q_\phi(s, a) - (r + \gamma \cdot (1 - d) \max_{a'} Q_\phi(s', a'))\Big)^2\Big]$$

tells us roughly how closely Q_ϕ comes to satisfying the Bellman equation and so can serve as the loss function in tuning ϕ.

Learning $\pi^*(s)$

Learning the optimal policy is pretty simple: we want to learn a deterministic policy $\pi^*(s)$ which gives the action that maximizes $Q^*(s, a)$. For this DDPG uses a neural network, $\pi_\theta(s)$, and loss function

$$L(\theta, \mathcal{D}) = -\mathbb{E}_{\mathcal{D}}\Big[Q_\phi(s, \pi_\theta(s))\Big].$$

3 DDPG FOR CONTINUOUS CONTROL

I've implemented DDPG for the Reacher in the notebook Continuous-Control-with-DDPG. The key components and hyper-parameter settings are outlined below.

- **Actor Network**: One fully connected linear hidden layer with 256 units. The input size is equal to the environment state-size and the output size is equal to the environment action-size. It uses ReLu activation functions and batch normalization after the hidden layer.
- **Critic Network**: Three fully connected linear hidden layers. The first hidden layer has 256 units and has input size equal to the environment state-size. The second hidden layer has 256 units and input size equal to 256 + the environment action-size. The third hidden layer has 128 units. ReLu activation functions are used throughout and batch normalization is applied after the first layer. The output of this network is one-dimensional.
- **Hyperparameter Settings**: I trained for 150 episodes with a batch size of 64, discount factor of $\gamma = 0.99$, soft update weight of $\tau = 0.001$, actor learning rate of 0.0001, and critic learning rate of 0.001. To get DDPG to learn in a multi-agent environment I had to separate updating network parameters from adding samples to the replay buffer so that I could interleave these two steps in an appropriate proportion. After a bit of experimentation, I modified slightly the suggestions from the "benchmark implimentation" discussion in the project description and updated the networks 15 times after every 20 timesteps.

With the above settings I achieved an average score of 34.26 for the final 100 episodes. The complete set of scores is plotted in Figure 2.

4 IMPROVING PERFORMANCE

Deep Deterministic Policy Gradient did well on this task. To improve performance futher, I would do grid search on the hyper-parameters and network architectures for this approach. Another interesting direction would be to try the *Soft Actor-Critic Algorithm* [1]. The key idea of this algorithm is to add an entropy constraint to the characterization of an optimal policy:

$$\pi^* = \arg\max_\pi \mathbb{E}_{\tau\sim\pi}\Big[\sum_{t=0}^{\infty}\gamma^t\Big(R_t + \alpha\mathcal{H}[\pi(\cdot|S_t)]\Big)\Big],$$

which can result in more stable learning.

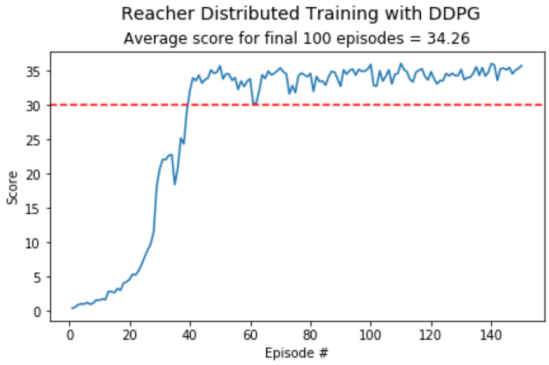

Figure 2: Training Scores

REFERENCES

[1] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. 2018. Soft Actor-Critic Algorithms and Applications. *CoRR* abs/1812.05905 (2018). arXiv:1812.05905 http://arxiv.org/abs/1812.05905

[2] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. 2014. Deterministic Policy Gradient Algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32 (ICML'14)*. JMLR.org, I–387–I–395. http://dl.acm.org/citation.cfm?id=3044805.3044850

[3] Richard S. Sutton and Andrew G. Barto. 1998. *Reinforcement learning - an introduction*. MIT Press. http://www.worldcat.org/oclc/37293240