Project: Continuous Control

Introduction

In this environment, a double-jointed arm 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 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.

The task is episodic, and in order to solve the environment, agent must get an average score of +30 over 100 consecutive episodes.

Learning Algorithm

Environment was solved by implementing Deep Deterministic Policy Gradient (DDPG), an actor-critic, model-free algorithm based on the deterministic policy gradient that can operate over continuous action spaces.

Deep DPG (DDPG) can learn competitive policies using low-dimensional observations (e.g. Cartesian coordinates or joint angles) using the same hyper-parameters and network structure. In many cases, agent can learn good policies directly from pixels, again keeping hyperparameters and network structure constant [ref]

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

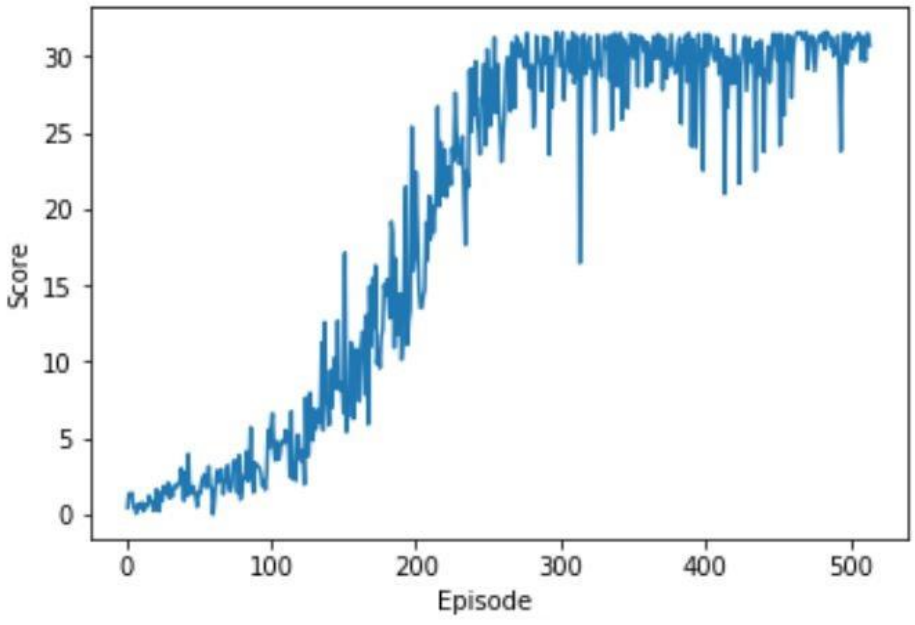
For this project, I used Udacity Reinforcement Learning Bipedal project [ref] DDPG implementation as starting point and adapted for project problem. DDPG algorithm requires time to train both Actor and Critic networks for agent to get an average score of +30 over 100 consecutive episodes, latest environment was solved in 8 hour with Udacity Workspace GPU instance. For efficient fine-tuning, I decided keep both networks simple and work with only two hidden layers, limiting size up to 1024 nodes. *max_t* (max number of timesteps per episode) is another parameter to limit for faster training. Experiments show that even slight

change in parameters or network architecture could stop training. Many times, training score peaked up and stopped converging. Below table to compare successful agents.

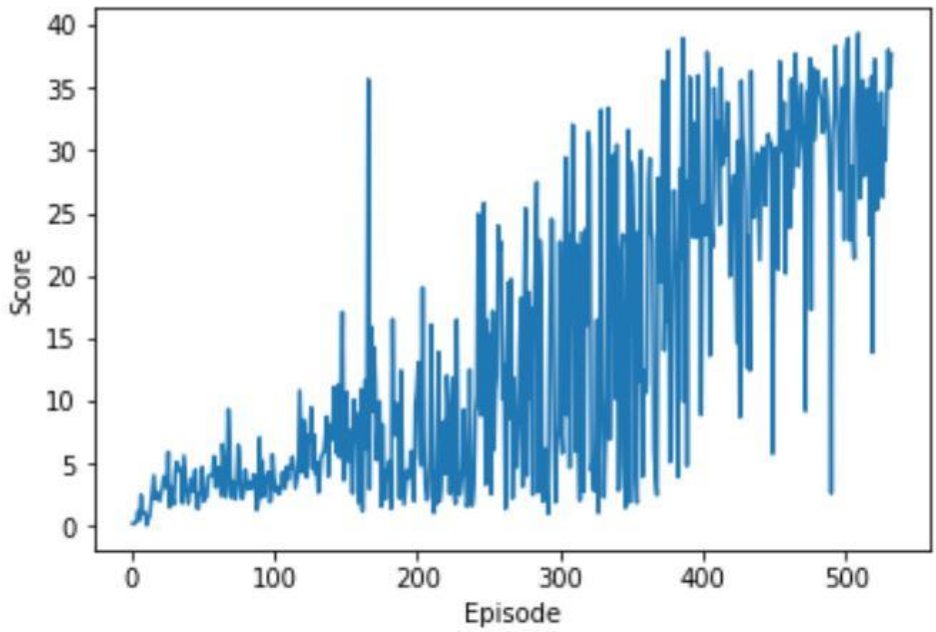
	Agent 1	**Agent 2**
Actor	fc1_units=256, fc2_units=256	fc1_units=512, fc2_units=256
	super(Actor, self).__init__() self.seed = torch.manual_seed(seed) self.fc1 = nn.Linear(state_size, fc1_units) self.bn1 = nn.BatchNorm1d(fc1_units) self.fc2 = nn.Linear(fc1_units, fc2_units) self.bn2 = nn.BatchNorm1d(fc2_units) self.fc3 = nn.Linear(fc2_units, action_size)	
Critic	fc1_units=256, fc2_units=256	fc1_units=512, fc2_units=256
	super(Critic, self). __init__() self.seed = torch.manual_seed(seed) self.fc1 = nn.Linear(state_size, fc1_units) self.bn1 = nn.BatchNorm1d(fc1_units) self.fc2 = nn.Linear(fc1_units+action_size, fc2_units) self.fc3 = nn.Linear(fc2_units, 1) self.reset_parameters()	
HP	BUFFER_SIZE = int(1e5) BATCH_SIZE = 512 max_t = 800	BUFFER_SIZE = int(1e6) BATCH_SIZE = 1024 max_t = 1000
	GAMMA = 0.99 TAU = 1e-3 LR_ACTOR = 1e-3 LR_CRITIC = 1e-3 WEIGHT_DECAY = 0	
Solved in	513 episodes	532 episodes
Time	5 hours	8 hours

Table 1: Agents comparison table

Plot of Rewards



Plot 1: Agent 1 plot of rewards



Plot 2: Agent 2 plot of rewards

Agent 1 shows rapid and smooth score increase from first episodes, but convergence stops before and could even start decreasing before reaching goal. Only increasing Batch, Buffer, max_t parameters adds more noise to training but average increase could go above 30.

Ideas for Future Work

- Investigate Ornstein-Uhlenbeck process parameters;
- Solve the Second Version with 20 identical agents, each with its own copy of the environment;
- Implement other algorithms like PPO, A3C, and D4PG;
- Find resources to increase training speed.