Phase 3 Deep Reinforcement Nanodegree Scholarship from Facebook

Nanodegree Program by Udacity

8/30/2019

PROJECT
CONTINUOUS CONTROL

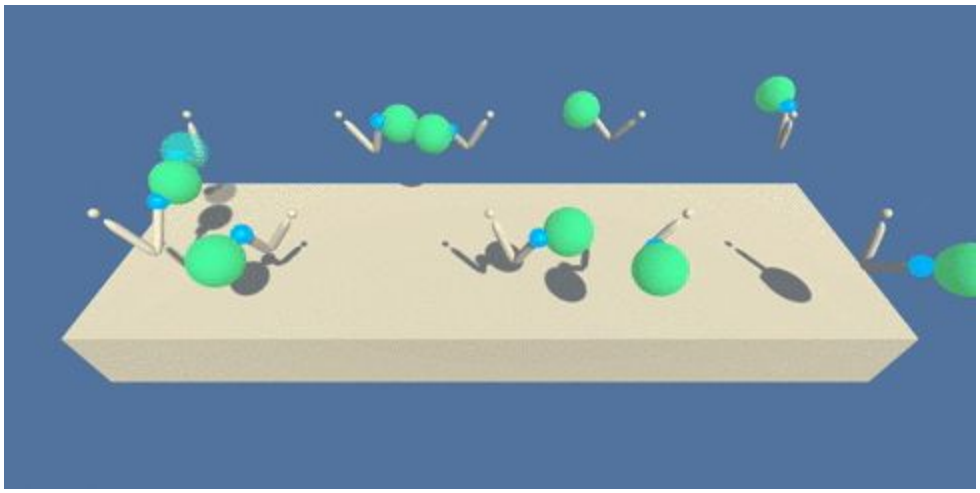
By Sabrina Palis

Project description and specifications by Udacity

Image Credit Sabrina Palis

For this project, we will work with the **Reacher Unity environment** in which 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 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.



Screenshot of the Unity Reacher Environment

Distributed Training

For this project, we will provide you with two separate versions of the Unity environment:

- The first version contains a single agent.

- The second version contains 20 identical agents, each with its own copy of the environment.

The second version is useful for algorithms like PPO, A3C, and D4PG that use multiple (non-interacting, parallel) copies of the same agent to distribute the task of gathering experience.

Solving the Environment - Only solve one for submission

Option 1: Solve the First Version

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

Option 2: Solve the Second Version

Your 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, yielding an **average score** for each episode.

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

Implementation: Multiple agent with DDPG

Rationale

The Continuous Control project is the second project of the Deep Reinforcement Learning Nanodegree, DRLND, by Udacity. It is the opportunity to **demonstrate an understanding of the relevancy of policy-based methods** in contrast to value-based methods in a complex environment.

In small state spaces, value-based methods are used to find an estimate of the optimal action value function q* which can be represented as a table with one row for each state and one column for each action - the optimal action just being the largent entry of each row. In larger spaces, the optimal action value function q* is represented by a neural network. It takes the environment state as an input and returns the value of each possible action as an output. This was illustrated by the deep Q-Network, DQN of the Navigation project of the DRLND.

In much more complex environment policy-based methods allow to find the optimal policy without having to first estimate the optimal action value function.

However policy-based agents have high variance, even though they learn faster, they can struggle to converge. A solution is to use a Temporal Difference critic to train baseline. **Actor-critic agents** have the best of policy-based and value-based agents: they are good at learning to act and they learn to estimate situations and actions. They are more stable and need fewer samples to learn.

An example of Actor-critic agent is The **Deep Deterministic Policy Gradient algorithm**, DDPG. It was conceived as the combination of the deterministic policy gradient with a DQN, therefore solving the limitations of a DQN agent in continuous action spaces.

Learning algorithm

The DDPG algorithm uses **two deep neural networks**, one is the actor and the other the critic. The **actor** is used to approximate the optimal policy deterministically by always outputting the best action for any given state. The **critic** learns to evaluate the optimal action value function by using the actor's best believed action argmaxa Q(s,a).

The DDPG includes **two copies of the network weights** for each network: a regular for the actor, a regular for the critic, a target for the critic.

In DDPG the target networks are updated using a **soft update strategy** which consists in slowly blending your regular network weights with your target network weights. Every time step, the target network is 99.99% target network weights and only 0.01% of the regular network weights. This soft update strategy helps to ensure the learning stability of the DDPG.

A **replay buffer** allows the DDPG agent to learn offline by gathering experiences.

Noise is added through the **Ornstein Uhlenbeck process** to get a better exploration, thus accelerating the convergence.

Pseudocode

DDPG algorithm. Reproduced from Lillicrap et al. (2015).

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:

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

        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

Network architecture

A simple architecture yielded good results with fully-connected linear layers and ReLu Activations.

Note that:

- the actor output is a tanh layer scaled to be between $[-b,+b], b \in R$.
- the critic network takes both the state and the action as inputs, and the action input skips the first layer. This is achieved using torch.cat.

Actor

Linear ➜ ReLu ➜ Linear ➜ ReLu ➜ Linear ➜ Tahn

Critic

Linear ➜ ReLu ➜ Linear ➜ ReLu ➜ Linear

Note the use of torch.cat to concatenate along an existing dimension.

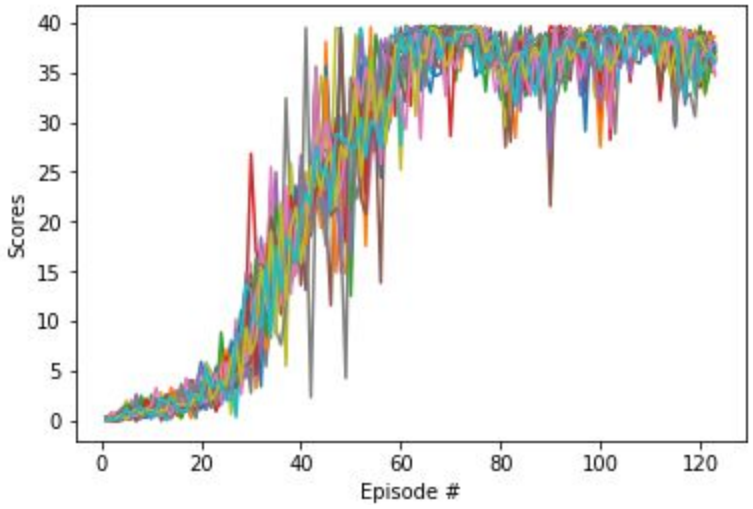
```
xs = F.relu(self.fcs1(state))
x = torch.cat((xs, action), dim=1)
```

Chosen hyperparameters

- BUFFER_SIZE = int(5e5) replay buffer size
- BATCH_SIZE = 256 minibatch size
- GAMMA = 0.99 discount factor
- TAU = 1e-3 for soft update of target parameters
- LR_ACTOR = 1e-3 learning rate of the actor
- LR_CRITIC = 1e-3 learning rate of the critic
- WEIGHT_DECAY = 0 L2 weight decay
- sigma=0.05 standard deviation for the OU noise
- FC1 = 128 input channels for the first hidden layer
- FC = 128 input channels for the second hidden layer

Plot of rewards

The following plot of rewards illustrate that the agent is able to receive an average reward (over 100 episodes, and over all 20 agents) of at least +30. The Reacher Environment Option 2 was solved in 123 episodes with an average score of 30.79.

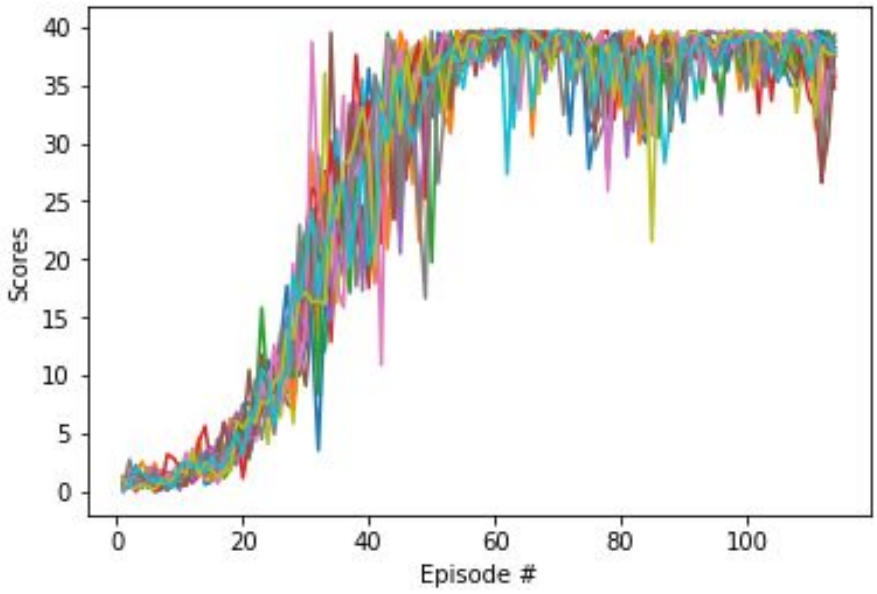


Plot of rewards DDPG

Further implementations

DDPG with Batch Normalization

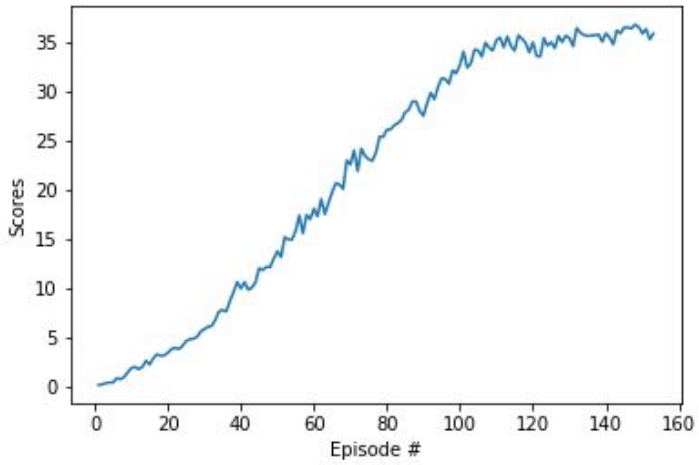
I initially thought that included batch normalization would yield better results, but probably because I kept the same hyperparameters, it gave a very similar plot of rewards as can be seen below . Solving the environment in 114 episodes with an average score of 30.68. (code available in this repo).



Plot of rewards DDPG with Batch Normalization

PPO

The PPO agent solved the environment solved in 53 episodes with an average score of 30.02. The code is available in this repo, note that it is originally by Shangtong Zhang and introduced during the DRLND course.



Plot of rewards PPO

Ideas for future work

- Trying to improve the stability of DDPG by playing with the learning rate
- Modifying the architecture
- Using mean-zero Gaussian noise instead of OU noise
- Implementation of A3C and D4PG in order to compare performances