. Deep Deterministic Policy Gradients

1 Introduction

In the following report, I implement a minimalist version of the deep deterministic policy gradients algorithm (DDPG) and report my findings. The rest of this document is organized as follows: Section 2 provides a brief overview of the DDPG algorithm; Section 3 describes the frameworks used; Section 4 illustrates the issues encountered during training and the steps taken to alleviate them; Section 5 presents and illustrates the performance of the agent, and finally, Section 6 provides concluding remarks. Code for this implementation can be found at: github.com/Niwhskal/ddpg

2 DDPG

The DDPG algorithm is based on the widely successful actor-critic variant of policy gradients in Reinforcement learning. In conventional policy gradients the objective is plagued by high variance in gradient estimates. A way to reduce this variance is by introducing a baseline. In actor-critic methods, this baseline is nothing but the Q-value estimated by a neural network. Therefore, in actor-critic methods we have two neural networks, an actor (or the policy network) which predicts the action given a state, and a critic (or the q-value network) which is tasked with telling the actor how good a particular state and action is.

DDPG is an extension of the actor-critic method to continuous state and action spaces. Because of this, the actor directly maps states to actions instead of outputting the probability distribution across a discrete action space.

Pseudo-code for implementing DDPG is as described in Fig. 1

3 Frameworks used

The following libraries were used for implementation:

1 Jax: for it's AutoGrad and XLA support.

2 Haiku: for the ability to easily compose modules into pure functions.

3 Gym: provides a straightforward way to interact with environments.

4 Optax: for easy of managing optimizers.

5 Rlax: for the l2-loss function.

6 Numpy: for support with jax

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

Figure 1: The DDPG algorithm

7 Matplotlib: for plotting and debugging.

8 Mujoco-py: to allow gym to make use of environments defined in the Mujoco simulator.

My implementation is organized into the following sections (in order):

1 Creating the Actor and Critic networks.

2 Implementing Experience Replay.

3 Creating the DDPG agent.

4 Training loop.

5 Evaluation loop.

All parts are composed in a single .py file. This was done to enable straightforward debugging and to provide an implementation where every module was defined in the same file. Personally, I have found this way of organizing code have better readability. I hope this resonates with the evaluators as well.

In addition, unit tests were carried out for each module to ensure expected behavior.

4 Training Methodology

All experiments were run in OpenAI's gym library. I ran these experiments in two mujoco-gym environemnts: InvertedPendulum-v2 and Reacher-v2. I considered state and action spaces to be low-dimensional outputs given by the respective environments. My initial plan was to extend the agent to train on raw pixel states, but constraints with resources and time have made it unable to do so.

On implementing DDPG according to the pseudocode given in Fig.1, I encountered an issue of vanishing gradients. I suspected it having noticed constant rewards during training. Upon debugging I learnt that the agent never changed it's parameters because of vanishing gradients. Changing two aspects resolved this problem: 1. The original paper used two hidden layers with 400 and 300 units respectively. I changed both layers to have 256 units. 2. The paper suggested using a learning rate of 10^{-4} for the actor and 10^{-3} for the critic. However, I found a learning rate of $3 * 10^{-4}$ for both networks to work best.

A second issue I encountered had to do with adding explorative noise to actions. The paper suggested using the Ornstein-Uhlenbeck process to do so, however I found that noise from the OU process wasn't contributing to learning, the network behaved in a way to counter the added noise. I remain unclear if this was a flaw in my implementation. I was able to resolve this issue by manually injecting noise from a gaussian distribution to the action given out by the policy.

Listed below are the hyper-parameters used while training:

- actor and critic weight initializations: sampled from a uniform distribution with a variance scaling initializer (scale = 2.0)
- actor and critic lr: $3 * 10^{-4}$
- replay buffer size: $2 * 10^{5}$
- Batch size: 256
- hidden layer size (actor and critic): 256
- gamma: 0.99
- actor output activation: tanh

- critic output activation: none
- actor and critic hidden activation: relu
- tau (target networks): 0.001
- noise: sampled from a unit gaussian
- timesteps run: $2 * 10^5$ steps

Finally, having learnt in class that delayed actor and target network updates can improve stabilization, I enforced the actor and target networks (target_actor and target_critic) to be updated once every 2 critic-updates. I did not notice a considerable improvement in stabilization during these runs.

5 Results

Having incorporated the changes mentioned in Section 4, I was able to successfully train the agent to perform maximally in both the InvertedPendulum-v2 and Reacher-v2 environments. Playbacks of the optimal policy are uploaded on github. Instructions to run the same is also documented.

5.1 InvertedPendulum-v2

On the InvertedPendulum-v2 environment, I noticed the agent learning rapidly after 30k timesteps. However, it's behaviour after was not consistent. I was able to achieve a maximum possible reward of 1000. A plot of the rewards in each episode is as shown in Fig.2.

Note: the X-axis records episodes and not timesteps. Each episode for InvertedPendulum-v2 can vary between 0-1000 timesteps.

Parameters for the best policy are uploaded on https://github.com/Niwhskal/ddpg.

5.2 Reacher-v2

Training on the Reacher-v2 environment was stable. Rewards were consistent and learning was drastic. A plot of the rewards accumulated during training is as shown in fig.3

Note: the X-axis records episodes and not timesteps. Each episode for Reacher-v2 can vary between 0-50 timesteps.

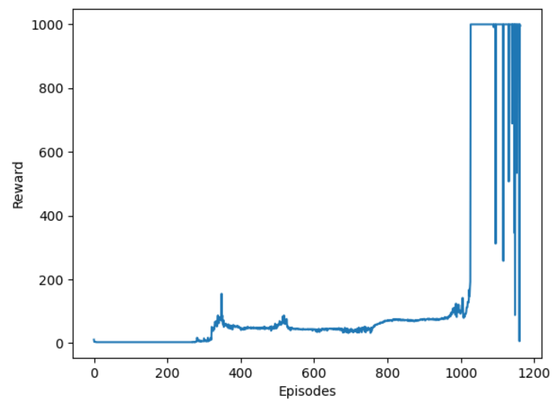

Figure 2: Rewards collected during training in the InvertedPendulum-v2 environment

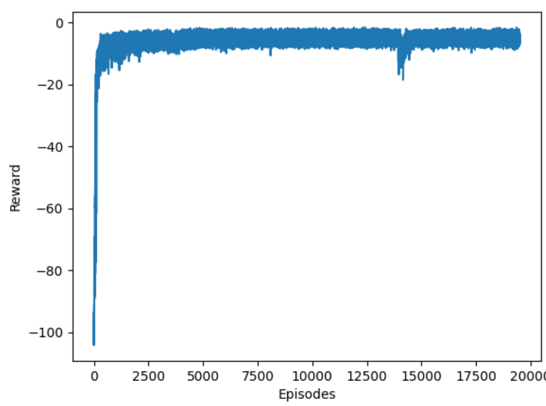

Figure 3: Rewards collected during training in the Reacher-v2 environment

Parameters for the best policy are uploaded on https://github.com/Niwhskal/ddpg.

6 Conclusion

Debugging and fine tuning training behaviour was a frustrating but enjoyable process. It suffices to say that DDPG is definitely reproducible albeit with some minor changes.

On a rather personal note, having been a beginner in jax, it was quite an experience trying to learn it's intricacies. However, in the end I was taken back by the incredibly boost in compute performance that the library offers, it has made me appreciate the library so much more.