udacity Deep Learning Nano Degree Project 2
Continuous Control – Project Report

data.camp097@audi.de

April 21, 2020

Contents

1 Introduction

This report summarizes my implementation and results for the second project in the udacity Deep Reinforcement Learning Nanodegree. In the first section, the DDPG algorithm is briefly described, in the second section my implementation and necessary modifications are presented. The results are presented and in the third section and a brief outlook on possible future improvements is given in the fourth section.

2 Algorithm

For my solution I implemented the Deep Deterministic Policy Gradient (DDPG) algorithm [1].

DDPG extends the idea of Q-Learning to continuous action spaces. Q-Learning needs to calculate the max over the actions in $\max_a Q^*(s, a)$. This is feasible if the action space is discrete, but non-trivial if it is continuous. DDPG uses a policy network $\mu(s)$ to predict the best action and thus to approximate

$$\max_a Q(s,a) \approx Q(s,\mu(s)) \tag{1}$$

The policy $\mu(s)$ can be learned in a gradient-based fashion, because the action space is continuous and we assume the value function $Q^*(s,a)$ to be differentiable w.r.t. a.

As DDPG contains two networks, one value-network and one policy network, it can also be considered an actor-critic method.

3 Implementation

The code is based on the DDPG example in the udacity DRL Nanodegree repo [1] with only minimal modifications to run on the unity ML environment.

Finding an implementation that solves the environment turned out to be quite tricky and time consuming, so after some experimentation on my own I started to beg, borrow and steal ideas from other papers and blog posts to get a working solution. I also included the hints given in the Classroom (see 3.3).

3.1 Network Architecture

As the basic architecture I chose a basic fully connected network with 2 hidden layers. As the state space is a low dimensional vector and contains no visual observations, I assumed there is no need for a deeper and / or convolational architecture. I started with a vanilla MLP with ReLU activation without any additional features like batch normalization, dropout and hand-crafted weight initialization. To solve the environment I gradually added additional features, I tried leaky ReLU, batch normalization, different sizes for the hidden layers, weight initialization and two different architectures for the critic network (see 3.1.2)

3.1.1 Network Architecture Actor

I experimented with batch normnalization at different positions in the network and got best results with one batch normalization before the second layer. Regarding weight initialization, I implemented Kaiming-like initialization [2] which worked ok. I started with (512,256) hidden layers, but learning improved with smaller sizes, so currently the size is (128,128).

3.1.2 Network Architecture Critic

I first tried to concatenate state and action as the joint input for the first layer, but I got better results with the original architecture from [1], where only the state is used as input to the first layer and the action is concatenated to the output of the first layer.

Again, larger hidden layers were not helpful so currently I also use (128, 128). Weight initialization is identical to the actor network and batch normalization was used after the first layer before concatenating with the action vector.

[1] https://github.com/udacity/deep-reinforcement-learning

3.2 Hyperparameters

Finding a set of hyperparameters that solves the environment turned out to be tricky a quite time-consuming. One set that works, even if neither quick nor stable, is:

- Size of the replay buffer: 1e5
- Batch size: 128
- Discount factor γ: 0.99
- Soft update τ: 0.001
- Exploration (OU) Noise σ:
- Exploration (OU) Noise μ:
- Maximum number of time steps per episode: 10000
- Learning rate actor: 3e-4
- Learning rate critic: 3e-4
- Weight decay: 0

3.3 Training

To solve the environment, a few modifications hat to be made. To improve stability I implemented both modifications given in the classroom, gradient clipping

```
torch.nn.utils.clip_grad_norm_(self.critic.parameters(), 1)
```

and dispersed updates to the networks every 10th timestep

```
for i in range(max_t):
    if i%10==0:
        action = agent.act(state)
        env_info = env.step(action)[brain_name]
        next_state = env_info.vector_observations[0]
        reward = env_info.rewards[0]
        done = env_info.local_done[0]
        agent.step(state, action, reward, next_state, done)
        state = next_state
        score += reward
```

3.3.1 Exploration

At first I used simple Gaussian noise to do the exploration but changed to Ohrstein-Uhlenbeck-Noise to improve the results as decribed in the original paper [1]. Later publications claimed that this particular kind of correlated noise is not necessary or beneficial, but here it seemed to help improve the learning.

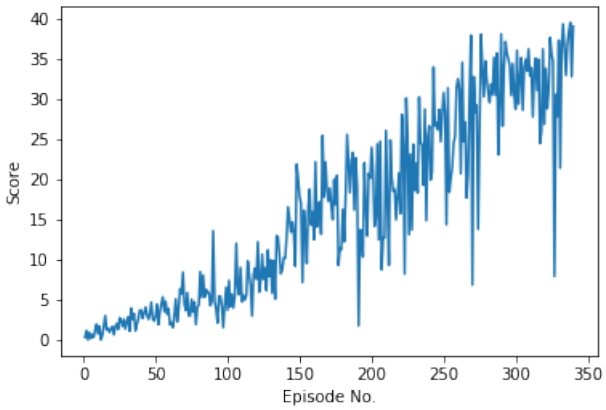

Figure 1: Learning Curve

4 Results

This section briefly shows the learning curve of the implemented algorithm. The learning curve (score as a function of episode number) is shown in figure 1. The training yields:

```
Episode 100 Average 3.26
Episode 200 Average 13.06
Episode 300 Average 24.98
Episode 340 Average 30.06

Environment solved in 340 episodes!
```

Learning does not seem to be very fast or stable. Future work should aim to improve both issues.

5 Outlook

There are many interesting topics for future explorastion and research:

- The logical next step would be to switch to the environment with multiple arms and try parallel training.
- Once we have a robust agent in the simulation environment it would be interesting to try out the agent on a real physical robotic arm.
- As suggested in the udacity classroom it could be fun to try an even more challanging task in continuous control as e.g. the unity ML Crawler.

To improve the performance of the agent, next steps could be:

- Priority Sampling. It might be beneficial to use Priority Sampling when picking samples from the replay buffer.

- It might stabilize the learning if we would decay the exploration as is often used in value basesd methods with ϵ-greedy exploratioin strategies. This would mean to modify the parameters of the Orstein–Uhlenbeck-Noise in our agent.
- We could switch from DDPG to other algorithms like PPO, A3C or Distributed Distributional Deep Deterministic Policy Gradients (D4PG) [3] to see if they offer advantages for this task.

References

[1] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," 2015.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," 2015.

[3] G. Barth-Maron, M. W. Hoffman, D. Budden, W. Dabney, D. Horgan, D. TB, A. Muldal, N. Heess, and T. Lillicrap, "Distributed distributional deterministic policy gradients," 2018.