DELTA ARM TRAMPOLINE

Ruotong Jia, Yipeng Pan

Northwestern University

1. ABSTRACT

In this project, a delta robot learns how to bounce a ball with its top platform using Deep Deterministic Policy Gradient (DDPG). This is an interesting problem because conventionally, it is addressed through optimal control techniques. In that case, one needs to build a good dynamics model of the robot and the ball, and a cost function on control input given the system states. Then, one needs to optimize the cost function, which might be a complicated non-convex optimization problem. On the other hand, recent advancements in deep reinforcement learning are making robot motion planning and control an easier task. We are able to use GPUs to optimize a deep neural network that leans such a policy. The control input is joint torque, which comes directly from motors. The end result of the project is a PyTorch model that lets the robot bounce the ball 8 times. At the end of this paper, we also have a discussion on how to escape local minima and avoid diverging for DDPG models using Adam optimizer.

2. INTRODUCTION

A Delta Robot (see Fig 1) has now become a widely used robot platform in industrial and domestic applications. For example, due to its simple structure and robust position control, delta robots are great for handling light-weight objects in high speed motions. Currently, Delta robots are widely used in packaging, robotic surgery, electronics assembly, etc. In this project, a Delta robot in a simulated environment is going to work as "trampoline", which bounces a soccer ball as many times as possible. During each bounce, the maximum height difference between the ball and the "trampoline" must be over a threshold, H. The game is over when the ball falls to the ground.

An amazing thing about the Delta Arm is its simplicity in control. It only has 3 degrees of freedom (DoF) - x, y, z. Therefore, we only need to control three joints on the base to achieve the full control.

Thanks to Dr. Bryan Pardo and Dr. Matthew Elwin for their academic and hardware support

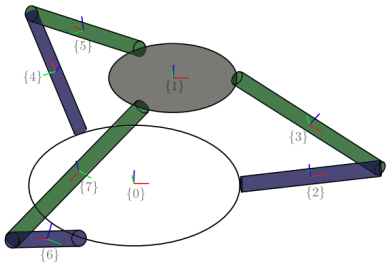

Fig. 1. Delta Arm Frames - red lines are x axes, blue lines are z axes

3. TERMINOLOGY

1. End-effector: the top platform of the Delta Arm that is in direct contact with the ball.

4. IMPLEMENTATION

The project is composed of the following major steps: building an OpenAI gym environment, connecting the gym to PyTorch and GPU, building a critic network, an actor network, make a copy of the critic and actor as a reference "target" network.

4.1. OpenAI Gym Environment

For the actual implementation, first, we build an OpenAI gym environment training and testing using the physics simulator PyBullet. The model has 3 joints, each joint is applied a torque, see Fig 2. our PyBullet model is able to provide dynamics information of each joint, link, and the ball, including 3 actuated joint angles, 3 actuated joint velocities, 3 Cartesian cooridnates of the ball.To simplify the problem, we assume we have direct access to the aforementioned information from sensors. The "labels" of each action is a reward. In order not to have too sparse rewards, we design our rewards of each time step as follows: +10 if the ball gets over H above the end-effector; 0 if the ball is above the platform but by less than H; -0.1 for every timstep that the robot is idle; -100 if the ball falls below the end-effector. Once the ball falls below the end-effector, the game is over. This information is sufficient for building an OpenAI gym [1].

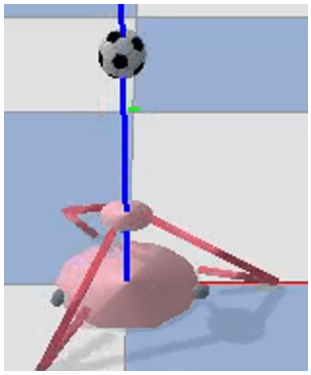

Fig. 2. Delta Robot and A Soccer Ball

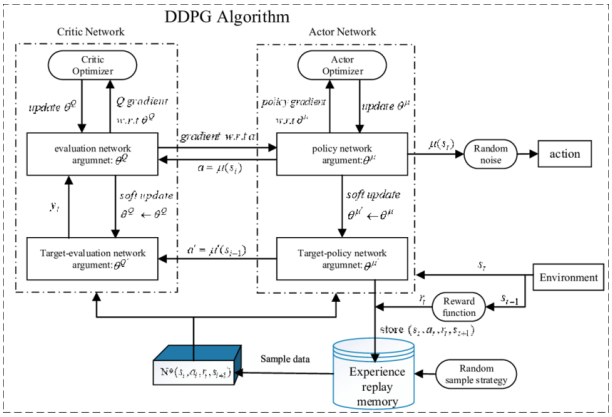

Fig. 3. High Level Overview of the DDPG Network Structure

The second step is to pipeline the state information and the score from the simulator to the neural networks. In our setup, the neural nets were trained on a Nvidia GeForce GTX 1080 GPU. This could be easily configured on PyTorch.

4.2. Construction of Neural Nets

In some previous work, Deep Q Net (DQN) was used heavily to evaluate the Q values of each discrete action. In this project, because a discrete action space might be too coarse, DQN is not an ideal solution.

The main technique used in this project is Deep Deterministic Policy Gradient (DDPG). A policy network will be trained with a reward function using policy gradient. The output layer of the network will be 3 values, each value represents a joint torque. We select our joint action to be joint torque τ applied on a joint by a motor, i.e, $a \in [\tau_{min}, \tau_{max}]$.

According to the DDPG network structure shown in Fig 3, we need two sets of actor and critic networks.

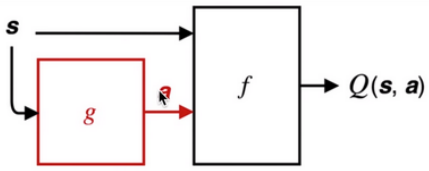

Fig. 4. Structure of a Simple Actor-Critic Framework. The actor is g, the critic is f, s is the current system state vector, a is the action vector

4.2.1. Actor network

The actor network's main purpose is to learn a "deterministic" policy function, such that given the current system states, it can output the correct action value in the continuous action space. For our actor network, the input of actor network is a $(1, 18)$ vector that represents the system states, s_t. These states are: 3 joint angles, 3 joint velocities, Cartesian ball position, Cartesian ball velocity, Cartesian end-effector position and Cartesian end-effector velocity.

Since we do not expect any obvious symmetry or any "deeper features" that could be identified by kernels, we do not use any weight sharing structures and use linear layers instead. ReLu will be used as activation layers. In total, there are 3 hidden layers, and the output layer is a tanh layer that maps joint action a, a $(1, 3)$ vector into the joint space properly.

The optimizer of choice is Adam, given its invariance to input scaling. We also set a weight decay factor to penalize large weights. One interesting to note is that weight decay is not the same as L2 regularization in Stochastic Gradient Descent (SGD), as the former will not penalize enough if gradients are large.

Since we intend to have the actor network maximize the critic's output $Q(s, a)$, we want to do gradient ascent on its parameters for training. Equivalently, we can do gradient descent by defining our loss function as $-Q(s, a)$. We can achieve the gradient $\nabla_\mu Q$ using PyTorch's back-propagation functionality. As a side note, the original paper decomposes the gradient $\nabla_\mu Q$ into

$$\mathbb{E}_{s_t \sim \rho^\beta}[\nabla_a Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s=s_t}]$$

4.2.2. Critic Network

The input of critic network is a $(1, 21)$ vector, including the output of the actor network a and the system states s_t. The structure of the network is three fully connected layers with a $(1 * 1)$ output. ReLu will be used after each hidden layer.

The loss function is the "TD Error", defined as the difference between its output and the target critic's output.

$$L(\phi, D) = \underset{(s,a,r,s') \sim D}{E}[(Q_\phi(s, a) - T)^2]$$

$$T = (r + \gamma(1-d) \max_{a'} Q_\phi(s', a'))$$

When the terminal state is reached, $d = 1$ so that there will be no future rewards, and the game is over. We will use gradient descent on the critic's parameters to minimize the TD Error.

4.2.3. Target Networks

The first set of networks is called the "training networks", with one actor and one critic, see Fig 4. The purpose of the target networks is to improve the convergence of the Q function learned by the critic **2**. The reasoning is without the target networks, updating the same "training" network that generates TD targets makes the Q function prone to divergence **2**. The target networks will provide a "stabilized" TD target, hence they get the name "target networks".

The target networks are composed of a "target" actor and a "target" critic. They have the same network structure as the "training" actor and the "training" critic. To provide a "stable" TD target, the target networks parameters, $\theta^{Q'}$, $\theta^{\mu'}$ will be updated at a "slower rate" than the training networks θ^Q, θ^μ by using Polyak Averaging.

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

where

$$\tau \ll 1$$

4.3. Training and Testing

We choose batch size as 32 and replay buffer size as 10000. These are hype-parameters that provides enough "randomness" but also will not make GPU too slow. Evaluation of the model is done every 10 episodes, and the average of 3 evaluation episodes is recorded as the evaluation reward. Therefore, the evaluation reward might be noisy given its small volume, we still expect it to reflect the overall model performance during training. Also, training takes place at every step after a number of initial "random draw episodes", using a random batch of experiences from the replay buffer

In our actual implementation, we noticed the great effect of "randomness" on the critic network. Without enough randomness on the action, the actor will always output the same locally optimal action, and the replay buffer is filled with highly correlated experience over and over. The way we add noise is "probablistic noise", as suggested by Lillicrap et al. [2] Matheron et al [3]., i.e, the optimal noise add a Gaussian noise $A(s) = a^*(s) + \mathcal{N}(\mu, \sigma^2)$.

Over the training process, the model demonstrated two common major problems in DDPG: convergence to local minima and divergence. Before A, the model converged to a local minimum. At A, to avoid too much repetition in the replay buffer, we increased the randomness

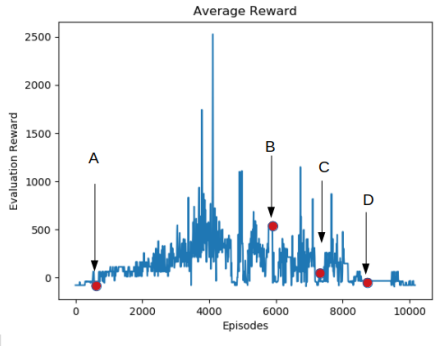

Fig. 5. The average reward throughout the entire training process. At first the model quickly converged to a local minimum, which outputs 0 actions. A: Increased standard deviation σ to 0.3. The model was able to learn but the model could not converge to a single parameter setting. B:Reduced noise standard deviation σ further to 0, however the model was still not converging. C: Enabled Adam Optimzer's Amsgrad [4], which provably converges to optimal solution in convex settings [4]. D: Reduced Learning rate of Adam to $\eta = 10^{-5}$
.

by with a standard deviation $\sigma = 0.3$. This successfully helped us escape the local minimum. However, this also contributed to the divergence of network parameters.

As shown in Fig5, after the large divergence, at B we decided to explore the effect reduced noise. We continued to use the same model, but with noise eliminated, the evaluation reward still fluctuates over a large range. At C we enabled Adam's amsgrad, which provably converges to optimal solution in convex settings [4]. At D, we reduced learning rate to $\eta = 10^{-5}$, the model converges to the local minimum reached before A. From this set of experiments, we rule out the effect of simply reducing noise and enabling amsgrad in Adam. We believe that the loss landscape itself very likely has fine-grained local minima in certain areas.

The convergence to local minima and divergence of the model triggers an interesting question: in DDPG, how to better escape local minima and converge to a reasonable result in a loss landscape with fine-grained local minima? Sutton et al. suggest the "deadly triad" can contribute to the divergence of deep RL models [5]: 1. Function Approximation (Q Function is not piece-wise continuous between states, instead, they are estimated by continuous functions that has unwanted local minima) 2.Bootstraping (estimating a value from another value, which is an essential technique used in TD learning) 3. Offline-Learning (Q Learning). These may be some leads to further research.

For testing purposes, we took a screenshot of the model at B, and in a simulated setting, the model is able to bounce the ball 8 times. There is a demonstration at: https://user-

images.githubusercontent.com/39393023/101517185-0e93e700-3946-11eb-9860-fd909cf052f6.gif

5. CONCLUSION AND FUTURE WORK

In this work, we explore the possibility and challenges of using deep reinforcement learning in a possible real-world optimal control problem. DDPG has demonstrated a promising capability in addressing such a complex problem, which may challenge the common optimal control techniques such as LQR, iLQR, etc. We also identify avoiding convergence to local minima and divergence in fine-grained DDPG's landscape as a topic for more future exploration.

References

[1] OpenAI, *Getting started with gym*, https://gym.openai.com/docs, 2017.

[2] S. Gu, E. Holly, T. Lillicrap, and S. Levine, *Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates*, 2016. eprint: arXiv:1610.00633.

[3] G. Matheron, N. Perrin, and O. Sigaud, *The problem with ddpg: Understanding failures in deterministic environments with sparse rewards*, https://arxiv.org/pdf/1911.11679.pdf, 2017.

[4] S. J. Reddi, S. Kale, and S. Kumar, *On the convergence of adam and beyond*, https://openreview.net/forum?id=ryQu7f-RZ, 2017.

[5] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction. the mit press, cambridge, ma, 2018.* 2017.

[6] A. Rajeswaran, V. Kumar, A. Gupta, G. Vezzani, J. Schulman, E. Todorov, and S. Levine, *Learning complex dexterous manipulation with deep reinforcement learning and demonstrations*, 2017. eprint: arXiv:1709.10087.

[7] Y. Duan, "Meta learning for control," https://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-233.pdf, Ph.D. dissertation, University of California at Berkeley, 2017.

[8] G. Anfu, X. Zheng, and D. Yanhua, *An autonomous path planning model for unmanned ships based on deep reinforcement learning*, https://www.researchgate.net/figure/Deep-Deterministic-Policy-Gradient-DDPG-algorithm-structure_fig3_338552761, 2020.

[9] OpenAI, *Getting started with gym*, https://papers.nips.cc/paper/1988/file/1c9ac0159c94d8d0cbedc973445af2da-Paper.pdf, 2017.

[2] [6] [7] [8] [1] [9] [3] [4] [5]