Learning to Throw as a Humanoid

Han-Hsien Huang
Texas A&M University
hanhsien.huang@tamu.edu

Sanghyeon Lee
Texas A&M University
sanghyeonlee@tamu.edu

Abstract

Throwing a ball is difficult for a robot because throwing consists of cooperation of many body parts. In this work, we attempted to train a humanoid to throw a ball from scratch in simulations. We designed two training schemes. The first one is training the robot with only one reward function. The other scheme consists of three stages. We first train the robot to stand, and then to hold a ball, and finally to throw a ball. In experiments, we show that the former scheme outperforms the latter one significantly. We discuss the possible reasons and the future works for improvement. Our code is on GitHub[1] and video is on YouTube[2].

1 Introduction

Throwing is one of the most complicated body motions. A good throw requires a coordination of all body parts. Not only that the arm should swing like a whip, the torso should twist and the legs should push the ground. In that way, the body can put as much energy to the object as possible. Due to the complexity of the motion, it is very difficult for a humanoid robot to accomplish.

DeepMimic [1] is the only current model which has demonstrated leaning to throw. They have a demonstration video on YouTube[3]. However, their model is trained in imitation learning, which requires data of human motion by capturing actual human doing the task. Collecting data and connecting real-world data to simulation are highly non-trivial, and thus we want to avoid that. Some other works have experiments of training humanoid robots with pure reinforcement learning in simulations. But most of them are experimented on simple tasks, such as walking or standing. None of them tried to learn throwing from scratch.

In this work, we tried to train a humanoid robot to throw a ball from scratch. To do so, we first need to define the environment. Since there isn't a predefined reinforcement learning environment for throwing a ball, we construct an environment on our own. We adopted MuJoCo [2] as our physics simulation engine, which is a popular engine in reinforcement learning research. We used OpenAI Gym [3] as the framework of environment. We designed three reward functions, which respectively encourage standing, holding a ball and throwing a ball.

We defined two training schemes. One is directly training the robot with throwing reward function. The other one include three stages. The first stage is training the robot to stand with only the standing reward function. The other two stages have a slowly changing reward function in training process. The second stage is from standing to holding a ball, while the third stage is from holding a ball to throwing a ball. In experiments, we compare the results of the two training schemes.

[1]`https://github.com/hanhsienhuang/ReinforcementLearningProject`
[2]`https://youtu.be/NN1LUZjlwS4`
[3]`https://youtu.be/vppFvq2quQ0?t=279`

2 Related Work

2.1 Humanoid simulations

There are many physics engines that can be used to simulate the dynamics of humanoids. Mujoco [2] is a very popular physics engine to simulate robotics especially in reinforcement learning. It is widely used and considered the benchmark of reinforcement learning on robotics. OpenAI Gym [3] has two built-in environments for humanoid modeled in Mujoco, which are Humanoid and HumanoidStandUp. The tasks in these environments are walking and standing up. DeepMind Control Suite [1] also has two environment for humanoids modeled in Mujoco. The tasks are also simple tasks, such as leaning to walk. Another physics engine is Bullet Physics [4]. There are some predefined environments designed for humanoids, such as walking, following a target and backflipping. Although Bullet Physics has comparable number of environments with Mujoco, it is less popular than Mujoco in benchmarking reinforcement learning. OpenSim [5] is another physics engine but only specifically for musculoskeletal structure simulation. Its goal is to have a realistic muscle and skeleton simulation for humans.

2.2 Reinforcement learning

Since the action space of humanoids is continuous, it is more suitable to use policy-gradient-based methods than Q-learning-based methods. Below we list some state-of-the-art methods in reinforcement learning, which are frequently used as baselines. Advantage Actor-Critic (A2C) and Asynchronous Advantage Actor-Critic (A3C) [6] uses advantage function to guide the training of policy. The asynchronous version can be used to train in distributed systems. Proximal Policy Optimization (PPO) [7] updates the policy within a trusted region, to avoid having a large step size and harm the performance of policy. Deep Deterministic Policy Gradient (DDPG) [8] is an off-policy method, unlike most policy-gradient-based methods. DDPG has a deterministic policy and it uses Ornstein–Uhlenbeck process [9] to explore. Soft Actor-Critic (SAC) [10] adds the entropy of policy into the reward function, which improves the exploration strategy and yields better performance.

2.3 Imitation learning with demonstration

To make the movement of robot more human-like or accelerate the training process, some works utilize data collected by motion capture to train the model. Ho and Ermon [11] and Merel et al. [12] used adversarial approach to train the agent to act like the demonstrated motions. DeepLoco [13] designed a hierarchical policy for humanoid. The low level policy is trained with motion captured data and the reward is defined as the difference between the agent's state and the demonstrated data. DeepMimic [14] is similar to DeepLoco but without hierarchical policy and with modified imitation reward function. The resulting method can train a diversity of robots with good quality. These methods require demonstrations made by capturing movement of real human or other simulations. Our goal is to train the model without any prior knowledge, so we don't apply any imitation learning techniques.

3 Method

3.1 Environment setup

We use MuJoCo [2] as our physics simulation engine. We adopt OpenAI Gym [3] as the framework of our environments. We modified the existing humanoid environment in OpenAI Gym to fit our tasks. For the humanoid model, we took the xml file[4] in DeepMind Control Suite [1], because the humanoid model has fingers so that it can hold a ball. We modified the initial posture of the humanoid and added a ball above its right hand. The initial state of the humanoid is shown in Figure 1. Our purpose is to train the robot to throw the ball to the right of the figure as far as possible.

We define three types of reward functions. First, the goal is to throw as far as possible, so we define the reward of throwing as follows.

$$r_{\text{throw}}(s_{t-1}, a, s_t) = x_b(t) - x_b(t-1), \tag{1}$$

[4]`https://github.com/deepmind/dm_control/blob/master/dm_control/suite/humanoid_CMU.xml`

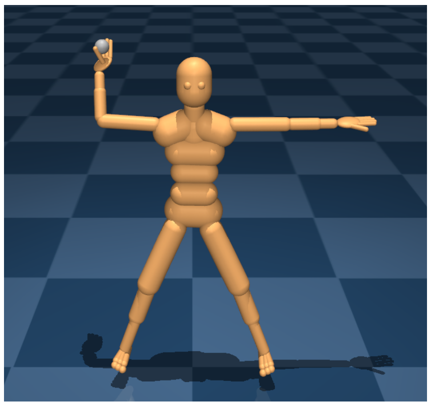


Figure 1: Initial state of the humanoid.

where $x_b(t)$ is the x value of the ball at time t. The direction of x-axis is to the left-hand-side of the humanoid, which is the to the right of Figure 1. In this reward function, we define the velocity of the ball in x direction. And thus the total reward is how far the ball is thrown. The terminal condition is when the ball drops on the ground.

Intuitively, directly learning to throw is difficult. Therefore, we also design a three-staged training scheme. First, we train the robot to stand stably. And then we train it to hold a ball at a fixed height. Finally, we train the robot to throw the ball. To do so, we define two additional reward functions, for standing and holding a ball. For standing, the reward is defined as

$$r_{\text{stand}}(s_{t-1}, a, s_t) = z_h(t), \tag{2}$$

where $z_h(t)$ is the height of the head at time t. In other words, we want to encourage the robot to stand straight and let its head as high as possible. The terminal condition is when the center of mass of the humanoid is below a certain height. For holding a ball, we have the reward function below,

$$r_{\text{hold}}(s_{t-1}, a, s_t) = z_b(0) - \left|z_b(0) - z_b(t)\right|, \tag{3}$$

where $z_b(t)$ is the height of the ball at time t. The reward function decreases both when the ball is higher and lower than its original height. That is, we encourage the robot to keep the ball at the initial height and don't let it fall down.

To transfer the reward function from one to another. We linearly interpolating two reward functions. When transferring, the reward function is defined as follows.

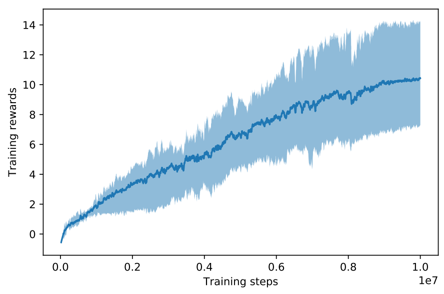
$$r_{1\to 2} = (1-\alpha)r_1 + \alpha r_2, \tag{4}$$

where α is gradually changing from 0 to 1 during training.

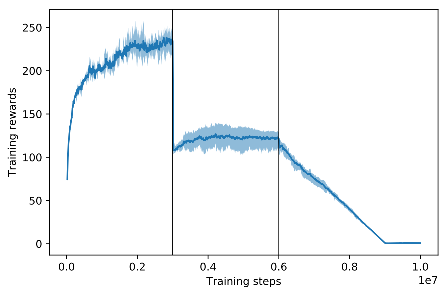
3.2 Training algorithm

At first, we tried Deep Deterministic Policy Gradient (DDPG) [8] to train our task. We used the code in the GitHub repository[5]. DDPG is a state-of-the-art method. However, we found that the performance wasn't very satisfying. We then adopted Proximal Policy Optimization (PPO) [7]. This is also a state-of-the-art method. We directly used the code in the GibHub repository[6] with some hyper-parameter tuning in our experiments.

We applied a technique called state normalization and reward normalization. In each training step, we calculate the exponential moving average of mean and standard deviation of states and rewards. And then normalize the states and rewards with the corresponding mean and std. We found that applying this technique enhanced the performance significantly. In addition, we don't have to adjust the scale of rewards.



(a) Training in throwing environment.



(b) Training in three environments.

Figure 2: The mean, minimum and maximum of training rewards with different types of training schemes. The experiments are run three times. The vertical lines in (b) represent the switches of environments.

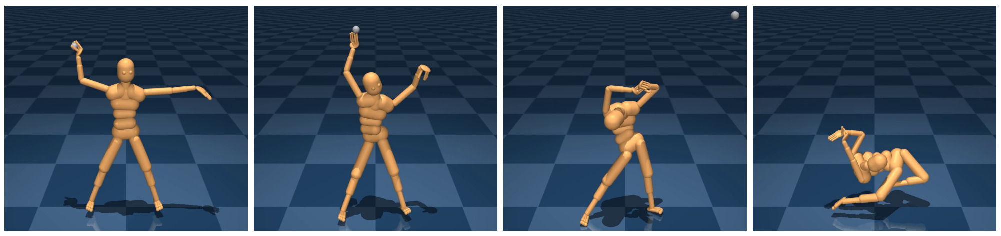


Figure 3: Snapshots of a trained model.

4 Experiments

We use PPO to train our model. The hyper-parameters of the training model are described as follows. The policy network and value network are separate networks. Both of them have two hidden layers. Each hidden layer has 500 dimensions. The activation function is ReLU. The output of policy network is a diagonal Gaussian distribution. We used Adam optimizer [15]. The initial learning rate is set to be 3e-4 and the learning rate is linearly decayed to zero at the end of each training.

We provide experiments of two training schemes. The first is that the agent is directly trained using only the throwing reward function r_{throw}. The total number of training steps is set to be 1e7. The other scheme is that we first train it to stand with r_{stand} for 3e6 steps. And then we linearly interpolate the reward function to r_{hold} in the next 3e6 steps. Finally, we linearly interpolate the reward function to r_{throw} in the next 3e6 steps, and continue training for 1e6 steps. As a result, the total number of training steps is also 1e7.

The results of training rewards are shown in Figure 2. The reason that Figure 2b has discontinuous training rewards is that the terminal conditions of the environments are different. At the end of training, the model trained in only throwing environment has a testing total rewards of 10.81 ± 3.04, while the model trained in three environment has testing total rewards of 0.31 ± 1.40. That is, training only in throwing environment significantly outperforms training with three stages. It is a little surprising. Intuitively, a complicated task should be learned gradually, and the three stages provide a good route to learn. However, in our experiments, it works very badly.

One possible reason of the bad result of training in three environments is the different scale of the reward functions. In Figure 2b, it can be seen that at the third training stage, which is from learning holding a ball to learning throwing a ball, the scale of the rewards differ largely. The scale of the rewards is dropped by about 10 times. As a result, the agent could possibly not learn the rewards of throwing. Initially, we thought that we have reward normalization so the scale is not needed to be carefully designed, However, it turns out designing a proper reward function is very important.

[5]`https://github.com/ghliu/pytorch-ddpg`

[6]`https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail`

To demonstrate the performance of the humanoid, we took several snapshots of a trained model throwing a ball in Figure 3. We took the model trained in only the throwing environment. In the figures, it can be seen that the robot actually throws the ball to its left. However, it is obvious that the body motion is not the optimal way to throw a ball. The robot clearly does not utilize its full strength of all body parts, such as torso and legs. The outcome is awkward and unlike an actually human. The reason is the lack of good exploration. The agent didn't find those kind of good throwing actions, so it couldn't learn anything like that. This shows the importance and difficulty of exploration. Without a good exploration strategy, the agent will not learn as expected.

5 Conclusion

We intended to train a humanoid robot to throw a ball like an actual human. We train the robot with two different strategies. For the first one, we design a throwing reward function and train the agent for 1e7 steps. For the other one, we designed two additional reward functions and train the agent in three stages. The experiments showed that training only with throwing reward function yielded better results. However, the motion of the robot looks nothing like an actual human. The future work would be trying to use other exploration methods in order to learn a better throwing motion.

References

[1] Yuval Tassa, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego de Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, Timothy Lillicrap, and Martin Riedmiller. DeepMind control suite. Technical report, DeepMind, January 2018. URL `https://arxiv.org/abs/1801.00690`.

[2] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[4] Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. `http://pybullet.org`, 2016–2019.

[5] Ajay Seth, Jennifer L Hicks, Thomas K Uchida, Ayman Habib, Christopher L Dembia, James J Dunne, Carmichael F Ong, Matthew S DeMers, Apoorva Rajagopal, Matthew Millard, et al. Opensim: Simulating musculoskeletal dynamics and neuromuscular control to study human and animal movement. *PLoS computational biology*, 14(7), 2018.

[6] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[7] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[8] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[9] George E Uhlenbeck and Leonard S Ornstein. On the theory of the brownian motion. *Physical review*, 36(5):823, 1930.

[10] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.

[11] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. In *Advances in neural information processing systems*, pages 4565–4573, 2016.

[12] Josh Merel, Yuval Tassa, Dhruva TB, Sriram Srinivasan, Jay Lemmon, Ziyu Wang, Greg Wayne, and Nicolas Heess. Learning human behaviors from motion capture by adversarial imitation. *arXiv preprint arXiv:1707.02201*, 2017.

[13] Xue Bin Peng, Glen Berseth, KangKang Yin, and Michiel van de Panne. Deeploco: Dynamic locomotion skills using hierarchical deep reinforcement learning. *ACM Transactions on Graphics (Proc. SIGGRAPH 2017)*, 36(4), 2017.

[14] Xue Bin Peng, Pieter Abbeel, Sergey Levine, and Michiel van de Panne. Deepmimic: Example-guided deep reinforcement learning of physics-based character skills. *ACM Transactions on Graphics (TOG)*, 37(4):1–14, 2018.

[15] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.