Comparison of Different Learning Algorithms for Beer-Pong in SL

Andreas Wieland
Darmstadt University of Technology

David Hoppe
Darmstadt University of Technology

Abstract

In this paper we compare two different reinforcement learning algorithms for learning how to play the game of beer pong with a robotic arm. A robot arm with seven degrees of freedom learned to throw a ball into a cup after letting it bounce off a table. We use a gradient method as a first approach to solving the problem. There, we estimate the gradient and we use Finite Difference for updating. The second method used is a policy search method called PoWER. For trajectory generation we used fourth order splines. We applied our approach on a simulated robot. We used the SL Simulation environment for implementation and evaluation of the algorithms. Further the results of the two algorithms are described and compared.

1 Introduction

Robotic learning has become a very popular research area during the last decade due to the importance of robots who can adapt to the environment. To this end, robots are very difficult to be trained to anticipate every possible outcome, thus novel learning approaches have been developed. Robots capable of learning new behavior are necessary in order to be applicable to real world scenarios [5], where they have to constantly adapt to small changes in their environment. Thus, in order to achieve adaptive robots machine learning methods suitable to meet the requirements in that particular area have to be developed. As stated by Peters and Schaal [5] many reinforcement strategies do not provide convenient solutions due to their scalability. Also, without an initial solution a robot could break before it finds a useful solution. Therefore the optimization should only be achieved in small steps in term of the policy to prevent the robot from being damaged.

The goal of the project was to implement different learning algorithms in order to solve a game called beerpong. A robotic arm holding a ball is placed in front of a table. In our task a cup is placed in a fixed position on the table. The goal is to throw the ball on the table and then into the cup. A suitable trajectory for the ball is drawn as a black line in Figure 1. While the robotic arm has seven degrees of freedom, we decided for simplicity to reduce the problem to only two degrees of freedom. Additionally, not all seven degrees of freedom are needed to solve the task.

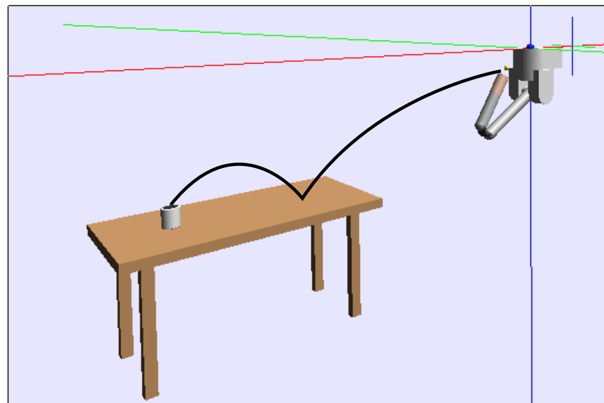


Figure 1: Simulation environment and task description

Reinforcement learning methods have been applied to solve the beer pong task in the past. Isaac and Kroenig [2] used policy-search methods (e.g., Finite Difference Gradient) in order to solve the problem. Further Wagner and Schmitt [7] extended the task by shifting the cup between iterations. The approach presented in this paper was implemented within the *SL Software Package* [6]. Figure 1 depicts our general set up of the task within the SL-Simulator. SL offers simulation and real-time control while being very close to programming a real robot. As SL is written C the implementation of the algorithms can be time consuming. It is particular not suitable for fast prototyping. Hence, we used a MATLAB [4] interface. The communication between SL and Matlab is realized through a shared memory [1].

This paper is structured as follows: First the model used to solve the beerpong problem is outline. We then shortly discuss the main approaches applied to the problem of learning the parameters of our model. In the subsequent section we present our results for each algorithm. Finally we discuss the differences of the algorithms and we compare the two approaches.

2 Method

In order to apply the different algorithms we first modeled a goal trajectory in joint space. We chose to model the trajectory by dividing it into two fourth order splines.

$$A(w) = \begin{cases} f(t) & \text{if } t \leq t_{thres} \\ g(t) & \text{if } t > t_{thres} \end{cases} \tag{1}$$

where t_{thres} is denoting the time when the ball is thrown

$$f(t) = \eta_4 t^4 + \eta_3 t^3 + \eta_2 t^2 + \eta_1 t + \eta_0 \tag{2}$$

$$g(t) = \xi_4 t^4 + \xi_3 t^3 + \xi_2 t^2 + \xi_1 t + \xi_0 \tag{3}$$

In addition we introduced the following constraints for $f(t)$:

$$f(0) = x_{start} \tag{4}$$

$$f'(0) = 0 \tag{5}$$

$$f''(0) = 0 \tag{6}$$

$$f(t_{thres}) = x_{thres} \tag{7}$$

$$f'(t) = v_{thres} \tag{8}$$

We set a fix initial position to x_{start} and because the arm of the robot is not moving at the beginning. Therefore the initial velocity as well as the initial acceleration also concludes to zero. Finally the velocity at the point x_{thres} where the ball is thrown is set to v_{thres}. When applying these constraints to $f(t)$ we obtain

$$\boldsymbol{\eta} = \boldsymbol{A}^{-1} \begin{bmatrix} x_{start} \\ 0 \\ 0 \\ x_{thres} \\ v_{thres} \end{bmatrix} \tag{9}$$

where $\boldsymbol{\eta}$ is defined as follows

$$\boldsymbol{\eta} = \begin{bmatrix} \eta_4 \\ \eta_3 \\ \eta_2 \\ \eta_1 \\ \eta_0 \end{bmatrix} \tag{10}$$

After deriving $f(t)$ two times we optain $\boldsymbol{A}$ with

$$\boldsymbol{A} = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ t_{thres}^4 & t_{thres}^3 & t_{thres}^2 & t_{thres} & 1 \\ 4t_{thres}^3 & 3t_{thres}^2 & 2t_{thres} & 1 & 0 \end{pmatrix} \tag{11}$$

For the second part of the trajectory we defined the following constraints for $g(t)$

$$g(t_{end}) = x_{end} \tag{12}$$

$$g'(t_{end}) = 0 \tag{13}$$

$$g''(t_{end}) = 0 \tag{14}$$

$$g(t_{thres}) = x_{thres} \tag{15}$$

$$g'(t_{thres}) = v_{thres} \tag{16}$$

This leads to the following solution for $\boldsymbol{\xi}$

$$\boldsymbol{\xi} = \boldsymbol{B}^{-1} \begin{bmatrix} x_{end} \\ 0 \\ 0 \\ x_{thres} \\ v_{thres} \end{bmatrix} \tag{17}$$

where $\boldsymbol{\xi}$ is defined as follows

$$\boldsymbol{\xi} = \begin{bmatrix} \xi_4 \\ \xi_3 \\ \xi_2 \\ \xi_1 \\ \xi_0 \end{bmatrix} \tag{18}$$

After deriving $g(t)$ two times we can define $\boldsymbol{B}$ as follows

$$\boldsymbol{B} = \begin{pmatrix} t_{end}^4 & t_{end}^3 & t_{end}^2 & t_{end} & 1 \\ 4t_{end}^3 & 3t_{end}^2 & 2t_{end} & 1 & 0 \\ 12t_{end}^2 & 6t_{end} & 2 & 0 & 0 \\ t_{thres}^4 & t_{thres}^3 & t_{thres}^2 & t_{thres} & 1 \\ 4t_{thres}^3 & 3t_{thres}^2 & 2t_{thres} & 1 & 0 \end{pmatrix} \tag{19}$$

We set a fix end position x_{end} at which the velocity and the acceleration is set to zero. For a smooth transition at t_{thres} we set the position and the velocity at this point to the same value as in the first spline, which are x_{thres} and v_{thres}.

From these formulas we can extract the parameters which the algorithms will be learning, these parameters are as follows:

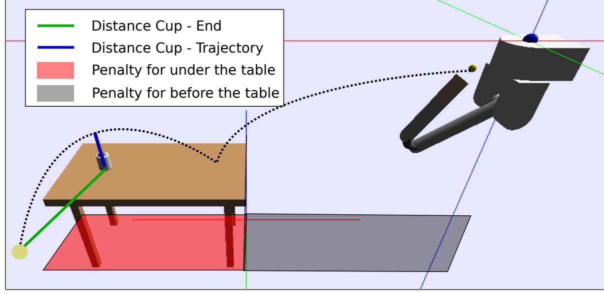


Figure 2: Visualization of the different parts of the reward function

$$\boldsymbol{\theta} = \begin{pmatrix} x_{thres} \\ v_{thres} \\ t_{thres} \end{pmatrix} \tag{20}$$

with $0 \leq t_{thres} \leq t_{end}$

2.1 Reward Function

For the reward function we combined different distance measures in order to obtain a suitable function.

$$\begin{aligned} r = & -\alpha \cdot d(x_{cup}, x_{ballend}) - \\ & \beta \cdot d(x_{cup}, x_{ballnearest}) + \gamma \end{aligned} \tag{21}$$

The distance $d(x, y)$ is the euclidean distance between x and y. And therefore $d(x_{cup}, x_{ballend})$ describes the euclidean distance between the ball and the cup after the ball is thrown and the time is up. The distance $d(x_{cup}, x_{ballnearest})$ describes the euclidean distance between ball and cup where $x_{ballnearest}$ is the nearest position of the ball to the cup in the entire trajectory of the ball. In order to prevent the reward from being high when the ball is thrown under the table γ is introduced as penalty. It also includes the penalty for a throw in front of the table which is a local maximum. Suitable values for α and β were chosen. Figure 2 depicts how the different parts of the reward function are measured.

$$\gamma = \begin{cases} -100 & \text{if the ball is thrown under the table} \\ -50 & \text{if the ball halts in front of the table} \\ 0 & \text{else} \end{cases}$$

2.2 Finite-Difference Gradient

In order to optimize the parameters of the trajectory a finite-difference gradient approach was used. The algorithm contains the following two steps: *Gradient estimation* and *parameter update*. After adding some random perturbations, the gradient is computed as follows

$$g_{FD} = (\Delta\Theta^T \Delta\Theta)^{-1} \Delta\Theta^T \Delta J \tag{22}$$

The gradient is used to properly update the parameters by

$$\theta_{t+1} = \theta_t + \alpha g_t \tag{23}$$

where α is the *learning rate*. We tried out different sample sizes N. New samples were drawn from a gaussian distribution with zero mean and covariance σI. We fine-tuned σ to 0.001. And obtain a set of samples Θ as follows:

$$\boldsymbol{\Theta} = \begin{pmatrix} \theta_t + \epsilon_{t,1} \\ \theta_t + \epsilon_{t,2} \\ \theta_t + \epsilon_{t,3} \\ \vdots \\ \theta_t + \epsilon_{t,n} \end{pmatrix} \tag{24}$$

with $\epsilon = \mathcal{N}(0, \sigma I)$

2.3 POWER algorithm

Policy learning by Weighting Exploration with the Returns (PoWER) [3] is another popular algorithm. The principle is that it calculates a distribution Model for the parameters and trys to tie down the variance in respect to the rewards of the current sample. In each step we calculate a random sample in respect to:

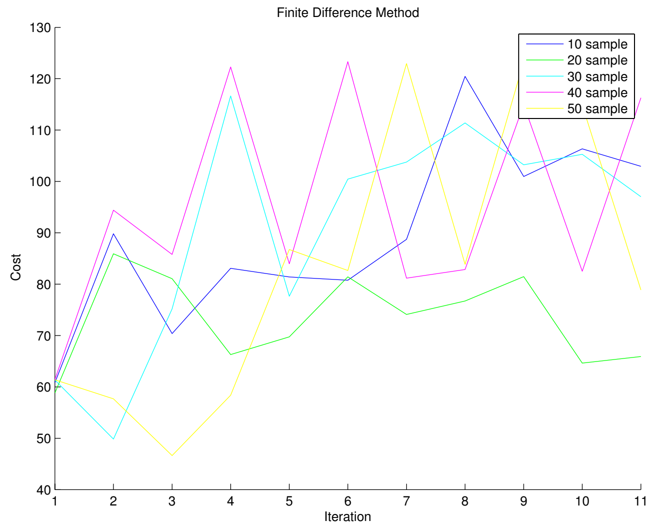
$$\boldsymbol{\Theta}_t = \mathcal{N}(\theta^*_{t-1}, \Sigma^*_{t-1}) \tag{25}$$

where θ^*_{t-1} is the weighted arithmetic mean calculated from the sample $\boldsymbol{\Theta}_{t-1}$:

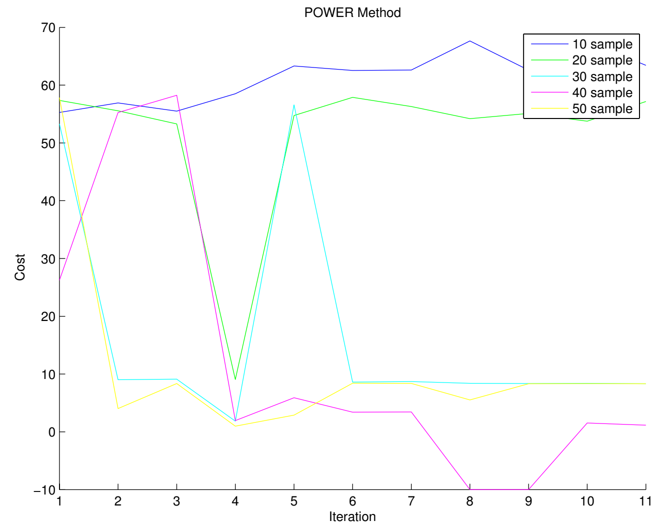
$$\theta^*_t = \frac{\sum_{i=1}^N w_i \theta_i}{\sum_{i=1}^N w_i} \tag{26}$$

And Σ^*_{t-1} is the the biased weighted covariance matrix calculated from the previous sample $\boldsymbol{\Theta}_{t-1}$

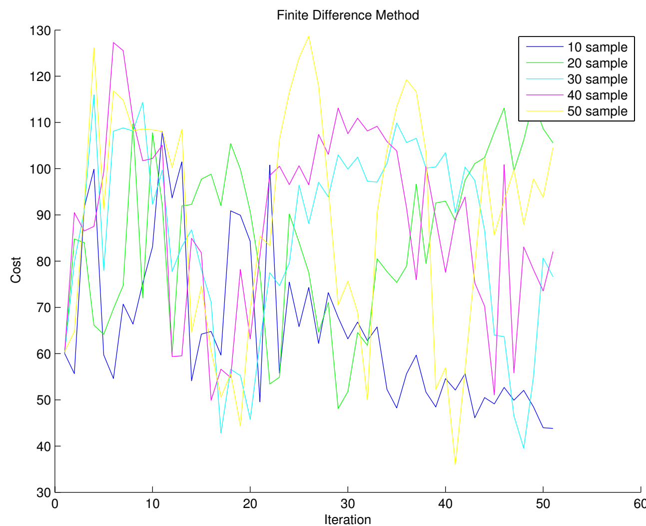
$$\begin{aligned} \Sigma^*_t = & \frac{\sum_{i=1}^N w_i}{(\sum_{i=1}^N w_i)^2 - \sum_{i=1}^N (w_i)^2} \cdot \\ & \sum_{i=1}^N w_i (x_i - \theta^*_t)^T (x_i - \theta^*_t) \end{aligned} \tag{27}$$



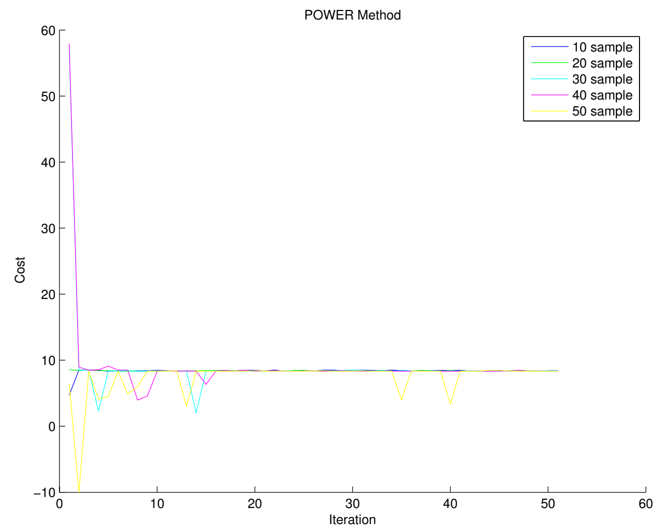
(a) Results of the Finite Diffrence Gradient Method for 11 Iterations and different samplessizes



(c) Most frequent values in array for the POWER Method over 11 Iterations and different samplessizes



(b) Results of the Finite Diffrence Gradient Method for 50 Iterations and different samplessizes



(d) Most frequent values in array for the POWER Method over 50 Iterations and different samplessizes

By using the reward r_i as the weights w_i we can assure that better throws have a higher influence on the next iteration so that the algorithm converge to maximze the Reward.

3 Results

In this section we will discuss our results for the two different learning methods and compare them with each other. For the purpose of comparing the two methods we took the most frequent values in the resulting array of the PoWER method. In Figure 3a you can see that for 11 iterations the finite difference gradient method does not converge for any sample size compared to the Power Method which converged for all sample sizes greater than 20 (see Figure 3c). For higher iterations we see that the finite difference gradient is converging but only for the smallest sample size. A bigger sample is in fact contra-productive and probably needs more iterations to converge. In addition to that more fine tuning of the start parameters and the learning rate could produce better results.

4 Discussion

In this paper we presented an approach to a robot competing in the game beerpong based on the finite difference method and the POWER algorithm. We implemented both algorithms on the SL Simulation Framework. In accordance with the results of Wagner and Schmitt [7] we observed the *Finite Difference Gradient* to converge slower and to show a higher vari-

ance with respect to the rewards. Our results are even more unstable compared to theirs. One reason for that could lie in the fact that our approach incorporated more degrees of freedom and therefore more parameters. We also observed Finite Difference to get more stable results when the sample size is low. Since PoWER converges to a local maximum there is still room for improvement. The simplifications made in our reward function could cause this convergence. It is also possible that further experimentation with the parameters α and β could improve the results. For this project we set the parameters to one. Hence both reward measures are equally weighted.

5 Limitations and Future Work

While we applied reinforcement methods to the beer-pong problem we made some simplifications to task. We only used two of the seven degrees of freedom. Although policy gradient methods scale well even beyond three degrees of freedom [5] by using only two degrees of freedom we could have chosen other reinforcement strategies as well. Another simplification we made is setting the cup's position fixed. If the cup's position changes during the learning of the policy transfer learning and therefore more complex algorithms are needed to complete the task [7].

The choice of a suitable reward function is crucial in reinforcement learning. We combined multiple rather simple to compute measures (e.g., Euclidean distance between the ball's end-position and the cup) and added a penalty for the ball landing under the table. Another more expensive with respect to implementation but rather convenient reward function is the position of the ball bouncing on the table for the second time.

References

[1] C. Daniel, G. Neumann, and J. Peters. Complacs robot arm evaluation scenarios.

[2] J. Isaak and M. Kroenig. *Beer pong*. PhD thesis, 2013.

[3] J. Kober and J. Peters. Policy search for motor primitives in robotics. (1-2):171–203, 2011.

[4] MATLAB. *version 8.1 (R2013a)*. The MathWorks Inc., Natick, Massachusetts, 2013.

[5] J. Peters and S. Schaal. Policy gradient methods for robotics. In *Proceedings of the IEEE International Conference on Intelligent Robotics Systems (IROS 2006)*, 2006.

[6] Stefan Schaal. The SL Simulation and Real-Time Control Software Package. Technical report, March 2006.

[7] F. Wagner and F. Schmitt. *Robot Beerpong: Model-Based Learning for Shifting Targets*. PhD thesis, 2013.