Realizing Target-Directed Throwing With a Real Robot Using Machine Learning Techniques

Malte Wirkus[1] and **José de Gea Fernández**[2] and **Yohannes Kassahun**[3]

Abstract. This paper presents a practical application of machine learning techniques in real-world robotics. Our goal was to make a anthropomorphic robot throw a ball into a bin that is placed at an arbitrary position in front of the robot. We use evolutionary machine learning to optimize a cost function based on a simulation model to aim at the target and generate the necessary motion to throw the ball at the position that is estimated by the simulation model. In order to compensate for the error in the simulation model, we trained an artificial neural network based on data from real-world task executions. We show that a simple simulation model can already result in good throwing performance if machine learning is applied to compensate for the resulting simulation error.

1 Introduction

When machine learning is applied to robotic manipulation tasks, a common approach is to perform learning in a simulation environment. The simulation models can be very different in terms of complexity and accuracy in modeling real world behavior, ranging from very reduced models to very complex dynamic simulations. Especially for simpler models, but also for the more complex ones, it comes to problems when learned tasks are transferred to the real robot. These occur from restrictions of the real-world robot hardware that have not been considered in the simulation, or from inaccuracies that results from imprecise modeling of the robot, the task, physical phenomena and so on. Building a perfect simulation in a way that a seaming less transfer to a real robot is possible is very difficult and hardly possible for complex tasks that for example include dynamic interaction with the environment.

Different approaches to improve the transfer from simulation to the real system are given in literature. In [8] this problem is approached by constraining the search space to regions where the simulation is more likely to be correct. To realize this it is proposed to define a metric for the similarity of behaviors to compare simulation with real-world performance. An estimation of the likeliness for a successful transfer of a simulated to a real-world behavior is derived from this. An evolutionary algorithm is then used to optimizes multiple objectives that on the one hand optimize the primary target function in simulation, and on the other hand prefers solutions that are expected to perform well on the real system.

In [2], a forward model is learned that maps motor commands of a controller to expected sensor data (e.g. a compass) in simulation.

[1] Robotics Innovation Center, DFKI Bremen, Germany, email: Malte.Wirkus@dfki.de
[2] Robotics Innovation Center, DFKI Bremen, Germany, email: Jose.de_Gea_Fernandez@dfki.de
[3] Universität Bremen, Germany, email: Kassahun@informatik.uni-bremen.de

Figure 1. We used the robotic platform *AILA* in our experiments. This image sequence shows the execution of the throwing task.

During real-world execution, a simulation error is composed based on the forward model and actual sensor data. A correction function that modifies the behavior is then learned online to compensate for the simulation error.

In [3] walking optimization of a small two-legged robot was performed with no external simulation environment, instead a surrogate optimization scheme is used. By performing real-world experiments, data is collected to learn a smooth surrogate function that is used as a replacement for a simulation environment. Simultaneously the current estimation of the surrogate function is used to optimize the main objective (walking distance) in order to generate a new parameter set to be evaluated on the robot hardware. Although the surrogate function is not able to approximate the real-world performance of the robot appropriate on the whole domain, the number of experiments to be performed is rather low (≈ 50) to learn a good performing gait.

On the control area, learning strategies have been long using neural networks to implement dynamic controllers. At first they were used for learning the whole inverse dynamic model of the robot which was found later on to be too complex. Therefore, the use of neural networks was in recent years mainly to support an adaptive control scheme rather than trying to approximate the whole inverse dynamics. That is, there is available a certain inaccurate inverse model which is used by the main dynamic controller and the learning strategy utilizes neural networks to compensate for inaccuracies or unforeseen changes of that dynamic model [5], an idea we adopt in

our work in order to compensate for an imprecise simulation model.

In this paper we validate this approach on the problem of making a real robot throw a ball into a bin that is placed at an arbitrary position in a bounded area in front of the robot. An incomplete physical formulation simulates the problem of predicting how far a ball, placed in the hand of the anthropomorphic robot (see Figure 1), would be thrown, when executing a trajectory. A cost function is designed and minimized using an evolutionary learning technique to find trajectories and joint configurations to throw the ball at the desired target. Shortcomings in the simulation are compensated by supervised training of an artificial neural network in order to enable for real world application which is also trained using evolutionary learning. To increase stability of the approach a library of parameter presets for the initialization of the function minimization algorithm is used. An overview of the framework is given in Figure 2.

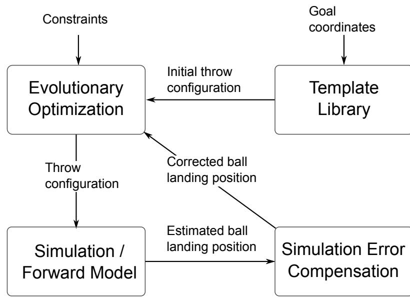


Figure 2. The proposed learning framework.

The ball throwing is performed on a human-scaled anthropomorphic robot, equipped with a five-fingered tendon driven hand which is used to hold the ball and release it while performing the throw movement. The given task is challenging and difficult to simulate precisely. Beside an accurate calibration of the robots kinematics and description of the dynamics of its movements, also the dynamic interaction between the ball and the multi-fingered, inherently compliant hand must be modeled to fully describe the simulation physically. We show that an incomplete simulation in terms of accuracy of the modeled quantities and the fact that there are unmodeled dynamic phenomena can already lead to good results. The inaccuracies of the simulation can be compensated by a function that maps the simulation results to corrected values, so that the task can be successfully fulfilled on real robot hardware.

In Section 2 we show how the robot's throw movements are represented. This forms the *Simulation* or *Forward Model* (Section 3) that is used to estimate the expected position where the ball will land. To find the proper movement to throw the ball at an arbitrary target, we formulate a parameter optimization problem that is described in Section 4. In Section 5 we discuss the shortcomings of the forward model and introduce a simple method to overcome these. We apply a correction function that was learned using supervised learning using data collected from multiple throw performances. We made a number of experiments in order to investigate the problem we are dealing with on the robot, collected training data and evaluated the results from learning the simulation error compensation function. In Section 7 we present these experiments along with their result. A discussion of the results and concluding comments are given in Section 8.

2 Throw configuration

To be able to throw the ball to arbitrary positions, we need the possibility for a computer program to modify the movements the robot should execute. A parametric description of movements provides this possibility and will be discussed in this section. While in Section 4, we will go more into detail on the modification of the throw parameters, here we only want to state that for parameter optimization the number of adjustable parameters plays a crucial role in the optimization performance. For this reason we where looking for a compact parametric representation of joint movement. Also, the joint movement should satisfy some criteria:

- The trajectory is bounded within defined joint limits.
- It should be smooth and non-oscillating.
- It should start and end with zero velocity.

To keep the number of parameters in a reasonable size, we decided to distinguish between joints that move (*active joints*) and joint that keep their position during the throw but are still used for aiming (*inactive joints*). The combination of both build the set of adjustable parameters to make the robot throw at different targets. We call this set of parameters *throw configuration*.

As a representation for joint movements we adopted a concept from the computer graphics community, the Bézier curve that is used to encode a sequence of joint angles. A Bézier curve

$$\mathbf{q}(u) = \sum_{i=0}^{n-1} \mathbf{p}_i B_i(u) \tag{1}$$

is described in terms of a parameter u, that varies in the interval of 0 to 1. The parameter is used to blend a set of control points $\mathbf{p}_i = [p_i, t_i],\ \ i \in \mathbb{Z}|0 \leq i < n$ by evaluating n basis functions B dependent on u.

The basis functions are the Bernstein polynomials, a set functions that are all always positive and together add up to 1 at each point in the interval of u. The resultant curve is a spline of degree $n-1$. Using fifth degree Bernstein polynomials as we do in this paper, Equation 1 can be written as

$$\begin{aligned}\mathbf{q}(u) &= [\mathbf{p}_0\ \mathbf{p}_1\ \mathbf{p}_2\ \mathbf{p}_3\ \mathbf{p}_4\ \mathbf{p}_5] \begin{bmatrix} (1-u)^5 \\ 5u(1-u)^4 \\ 10u^2(1-u)^3 \\ 10u^3(1-u)^2 \\ 5u^4(1-u) \\ u^5 \end{bmatrix} \\ &= \mathbf{P} \cdot \mathbf{b} \end{aligned} \tag{2}$$

The resulting curve starts at p_0, ends in p_5 and is always enclosed within the convex hull of the control polygon that is spanned between the control points, so no strong oscillations occur. By constraining the p-coordinate of the control points to the operating range of the corresponding joint, it is ensured that the trajectory is within the boundaries of the joint limits.

It can be shown that the first derivative at the first and last control point is given by

$$\dot{\mathbf{q}}(0) = n * (\mathbf{p}_1 - \mathbf{p}_0) \tag{3}$$

and

$$\dot{\mathbf{q}}(1) = n * (\mathbf{p}_{n-2} - \mathbf{p}_{n-1}). \tag{4}$$

To ensure zero velocity at the start and end of the trajectory we set $p_0 = p_1, p_{n-2} = p_{n-1}, t_0 \neq t_1$ and $t_{n-2} \neq t_{n-1}$ [10]. An example trajectory is shown in Figure 3.

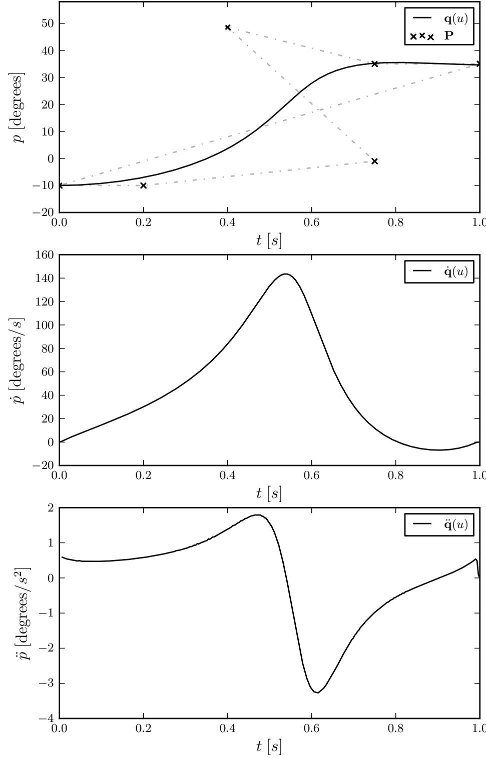


Figure 3. Joint trajectories are modeled as Bézier curves. By modifying the placement of the control points $\mathbf{P}$ the curve changes. The control polygon is shows as plotted line. Also joint velocity and acceleration are shown.

It is worth to mention that the parameter for the basis functions u lies within the range but is not equal to t. To sample the curve in discrete steps along the t-axis (which represents a scaled time axis), we sample $\dot{\mathbf{q}}(u)$ with a rate 4 times higher than the desired control frequency for the trajectory and linearly interpolate for the actual time steps.

3 Simulation / Forward Model

For our ball throwing task it is relevant, how the joint movements of the robot influence the movement of the ball, or stated differently: Given a throw configuration, where will the ball hit the ground. Actually, for throwing inside the bin we need to know when the ball passes a specified height, which is the height of the bin. We call a function that provides this information the forward model.

For a precise simulation, there are factors involved that are difficult to model. It starts with the question how the robot will be able to follow the joint space trajectories, which is dependent on the low-level controllers and mechanical parts like the motor, gears and so on. Also dynamic properties of each link have to be modeled if you want to be precise. In our case, the ball is held by the robot in a robotic hand. Here dynamic properties, involving the friction between hand and ball, as well as the exact shape of the fingers, should be considered. We decided not to try to provide a physical model that is very precise, but rather to use one that captures only the most relevant properties.

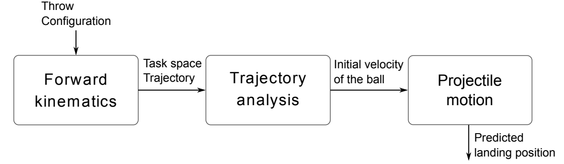


Figure 4. The Simulation / Forward model.

For the prediction where the ball will land we are especially interested in the movement of the robotic hand through space (*task space trajectory*) as the result from the throw configuration. Evaluating the task space trajectory then leads to an estimation of the ball trajectory in an ideal way. Figure 4 shows an overview about our forward model and will be explained in the following paragraphs.

The throw configuration contains fixed joint positions together with joint trajectory parameters that can be evaluated to joint space trajectories using Equation (2). That means all information from a throw configuration can be represented as a temporal sequence of joint configurations, when the fixed joint positions are considered as a trajectory of zero velocity. The task space trajectory results from the joint movements and the kinematic structure that connects the joint. The kinematic structure of the robot can for example be represented using the well-known Denavit-Hartenberg convention. Calculation of the forward kinematics (given the joint configuration at a particular time point) leads to the determination of the configuration of the end effector at this time. By repeating this for every sample from the joint trajectories we get the task space trajectory.

During the execution of the throw movement (i.e. the task space trajectory) we need to release the ball. The moment where the ball gets released should be well timed, so that it results in a nice trajectory of the ball through the air. The trajectory of the ball can be calculated using equations for projectile motion as shown in Equation (5). They depend on the initial velocity of the projectile (in our case the ball, which we assume has the velocity of the end-effector in direction of hand aperture) $\dot{\mathbf{p}}_0 = [\dot{x}_0 \ \dot{y}_0 \ \dot{z}_0]$, the gravity g, and the displacement from the origin frame at the moment of launch $\mathbf{p}_0 = [x_0 \ y_0 \ z_0]$. The origin frame $\mathbf{R}$ is placed centered to the robot on the ground (see Figure 5(b)).

$$\begin{aligned} x(t) &= x_0 + \dot{x}_0 t \\ y(t) &= y_0 + \dot{y}_0 t \\ z(t) &= z_0 + \dot{z}_0 t - 0.5 g t^2 \end{aligned} \tag{5}$$

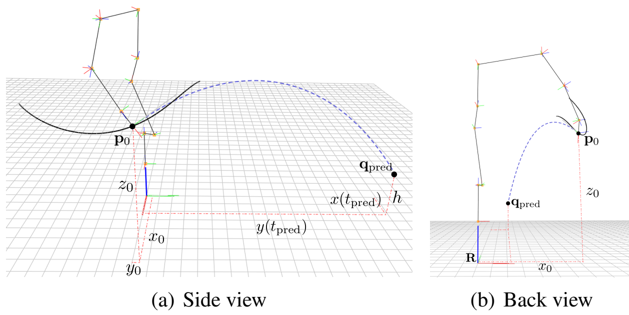
We are interested in the point where the ball passes a specified height h (the height of the bin). So we modify the last term from Equation (5) to include the height of the bin and solve the resulting quadratic equation for t

$$\begin{aligned} 0 &= z_0 - h + \dot{z}_0 t - 0.5 g t^2 \\ t_{\text{pred}} &= \frac{(\sqrt{2 g z_0 + \dot{z}_0^2 - 2 g h} - \dot{z}_0)}{g}. \end{aligned} \tag{6}$$

Now that we have the *time of flight* t_{pred}, we can insert the value in Equation (5) to calculate the predicted ball landing position

$$\mathbf{q}_{\text{pred}} = \left[\begin{array}{c} x(t_{\text{pred}}) \\ y(t_{\text{pred}}) \end{array} \right]. \tag{7}$$

To determine the end-effector velocity at each time point, we differentiate the task space trajectory with respect to time. We look for the point in the end-effector trajectory that has the highest velocity and use this for estimating where the ball will land using the Equations (6) and (7). The ball can leave the hand only in direction of the hand aperture, so the velocity along this vector is considered. In Figure 5(a) and 5(b) screenshots of a visualization of the simulation model are given.



(a) Side view (b) Back view

Figure 5. Two screenshots taken from the simulation for the same throw configuration from different views. The kinematic model is shown as black lines. The projectile trajectory is the dashed line. The end-effector trajectory is shown in thick black curve. We superimposed some of the quantities from the Equations (5 - 7).

It is clear that using this formulation alone as forward model will deliver faulty predictions. But in this paper we claim that an incomplete simulation can already be good enough to accomplish tasks in real world, if the error in the simulation is compensated as we will describe in Section 5.

4 Parameter optimization

In Section 2 we described how movement of the robot is modeled and in Section 3 how an estimation for the resulting ball landing position is calculated given a throw configuration. A throw configuration is a set of adjustable parameters and varying these will result in a different throw movement in terms of dynamics, position and orientation of the hand in task space. This finally results in a different estimated landing position of the ball. Modifying the values in the parametric description allows to aim for different targets to throw at.

Summarized the adjustable parameters $\mathbf{c}$ from a throw configuration are:

- The start and end position of each active joint represented by p_0 and p_5 (cf. Figure 3 and Equation (2)).
- The 2D-positions of the inner control points $p_1, \ldots, p_4$ for each active joint trajectory (also Equation (2)).
- The time for each active joint trajectory, which is the factor we scale the t axis from Figure 3 with.
- The angles of the inactive joints.

The problem to solve is to find the set of parameters, for which the result from the forward model corresponds to a given target position. We formulate this as a parameter optimization problem

$$\underset{\mathbf{c}}{\text{argmin}}\ f(\mathbf{c}). \tag{8}$$

The predicted ball landing position is given by Equation (7) of the forward model described in Section 3, which is a function $g(\mathbf{c})$ dependent on the throw configuration. To describe the quality of one particular throw configuration, we measure the squared Euclidean distance from the target $\mathbf{d}$:

$$f_{\text{pos}}(\mathbf{c}) = \|g(\mathbf{c}) - \mathbf{d}\|^2 . \tag{9}$$

Working on actual hardware we need to generate feasible trajectories that are executable on the robot. This requirement constraints the joint limits and the maximum velocity per joint, that is limited by the dynamic properties of the robot, its actuators and the low-level controllers. Also we want to enforce a principle characteristic in the throw movement, i.e. we have an acceleration and deceleration phase in the end-effector movement. By constraining the position of the inner control points of the joint space trajectory parameters this can be achieved. To induce these constraints $\mathbf{k}$, we formulate the objective function from Equation (8) as

$$f(\mathbf{c}) = f_{\text{pos}}(\mathbf{c}) + f_{\text{penalty}}(\mathbf{c}, \mathbf{k}, \alpha), \tag{10}$$

where $f_{\text{penalty}}(\mathbf{c}, \mathbf{k}, \alpha)$ calculates the absolute distance of each parameter value to the nearest boundary of the valid range if it exceeds it. For each parameter the constraint violation is weighted by a corresponding factor from the weight vector α.

From the different methods to choose for parameter optimization, we decided to use Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [1]. This method allows for minimization of non-linear functions without the need of a derivative. The algorithm already showed very good results in many different problem domains[4].

5 Simulation error compensation

Due to our rather simple simulation model, the estimation of the throwing distance lacks precision. For example, the point on the trajectory where the ball leaves the hand and the direction are only rough estimations. Friction between the ball and the robotic hand or collisions of the ball with the fingers is not considered at all. Also the kinematic model of the robot is not precisely calibrated and the response behavior of the low-level controllers is not modeled in the simulation.

A classical way would be to try to improve the simulation to model all involved phenomena using physical equations and to establish a complete dynamic simulation of the problem. The resulting software would be much more complex and slower, and also the success of this approach can be questioned, since most simulation software show a discrepancy as compared to real-world system it simulates [6]. Instead we decided to accept that the simulation is not perfect but gives an estimation that only reflects the most important characteristics of the problem.

We wanted to overcome these shortcomings by means of supervised machine learning and decided to train a feed-forward neural network that acts as an error compensation function. As depicted in Figure 6, the neural network maps from the predicted position by

[4] A list of applications of the algorithm can be found on the website of the author of CMA-ES (http://www.lri.fr/ hansen/cmaesintro.html)

the forward model to a corrected position and was trained on data collected in real world experiments (see Section 7 for a detailed description of the experiments).

We used a feed-forward artificial neural network with one hidden layer that consists of 5 nodes activated by $\tanh$ functions. The two output nodes are also implemented using $\tanh$ activation functions.

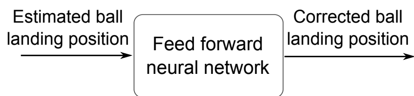


Figure 6. Simulation Error Compensation: A feed-forward artificial neural network is used to correct the simulation error.

Learning is performed using the RPROP algorithm [9] to minimize the mean square error of the output of the network activated by the predicted ball landing position from Equation (10) to the measured throwing distances $\mathbf{m} = [m_x \; m_y]$:

$$\underset{\mathbf{x}}{\operatorname{argmin}} \frac{1}{2k} \sum_{i=0}^{k-1} \left(n(\mathbf{q}_{\text{pred}}^{(i)}, \mathbf{x})_x - m_x^{(i)} \right)^2 + \left(n(\mathbf{q}_{\text{pred}}^{(i)}, \mathbf{x})_y - m_y^{(i)} \right)^2 \tag{11}$$

Here, $n(\mathbf{q}_{\text{pred}}^{(i)}, \; \mathbf{x})_{x,y}$ denotes the activation level of the two output nodes of the artifical neural network as a function of the estimated throw position for the ith out of k samples, given the weight vector $\mathbf{x}$. As a result the corrected estimation is now given by

$$\mathbf{q}_{\text{corr}} = n(\mathbf{q}_{\text{pred}}, \mathbf{x}_{\text{min}}), \tag{12}$$

where $\mathbf{x}_{\text{min}}$ are the weights for the neural network obtained from training.

6 Template library

Our simulation error compensation function maps from expected ball landing positions to corrected ones. This approach completely neglects special characteristics in the resulting ball trajectory due to distinct parameter constellations in the throw configurations. So the correction function is likely to work only for throw configurations that are very similar to each other.

To improve the results, we decided to not strictly use one source configuration as initial parameter set for the generation of new throw configurations, but created a database from the training data. We then used the configuration that results in the closest ball landing position to the given target (determined by nearest neighbor search) as initial parameter set.

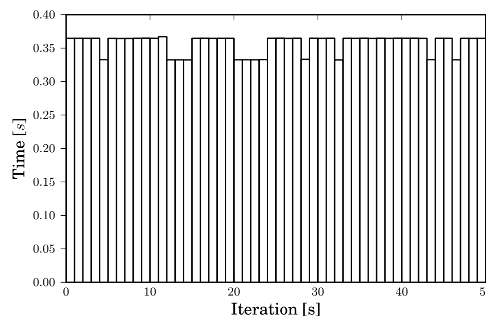
We expected to increase the performance of the algorithm by providing template throw configurations. But it is clear that it does not really solve the problem which arises from the fact that different throw configurations with the same target estimated by the forward model, might actually result in different ball landing positions.

7 Experiments and Results

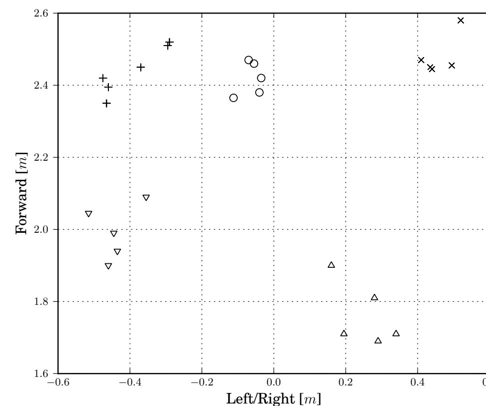
7.1 Initial tests of the robot hardware

As robotic platform to implement the framework, we use a human scaled mobile dual-arm robot named AILA (shown in Figure 1). It consists of a complete anthropomorphic upper body mounted on a mobile platform. The upper body consists of two arms, each of them with seven degrees of freedom, a torso with four joints, and a head with two degrees of freedom. Each torso and arm joint is equipped with position and velocity sensors. To the right arm a five fingered hand is mounted. The hand is underactuated which means that several of the 19 joints are coupled and only actuated by one motor. In total there are 9 active degrees of freedom for the hand. The design of the hand makes it in parts inherently compliant. The head of the robot is equipped with two cameras.

The task requires the robot hand to grasp and hold the ball and release it during the trajectory after accelerating at the time of highest task space velocity. We use predefined hold and release configurations that are position controlled by a PID-controller. To synchronize the hand and body movements, we determined the time for changing from the hold to the release configuration empirically.



(a)



(b)

Figure 7. (a) Difference in opening time of the hand over multiple repetitions. (b) Resulting ball landing position after execution of the same trajectory multiple times. Each symbol represents a different trajectory.

Beside other factors like the capabilities of the low-level controllers in the body and arms of the robot to follow a given trajectory in a similar way over multiple repetitions, the repeatability of the hand opening movement was identified as an important factor to influence the overall ball throwing performance. To evaluate this, we brought the hand into an open configuration, placed the ball and set the hold configuration as reference for the position controllers of the hand joints. Now we set the release configuration as reference and measured the time until the movement is completed using software timers. This was performed 50 times (see Figure 7 (a)). The mean

time needed for the trajectory was 0.357s with a standard deviation in the opening time is 0.014s.

Together with unmodeled dynamics inside the system, the repeatability of complete throw performances, measured as the mean standard deviation in the ball landing position is 0.08m. This was found out by performing the resulting trajectory from one throw configuration multiple times. For each trial the position where the ball landed was marked and measured. This resulted in the distributions as shown in Figure 7 (b).

7.2 Visual detection

In this work we did not focus on computer vision algorithms, but found using the measuring tape cumbersome. A simple method to detect the bin visually and estimate its position using the camera in the head of the robot was implemented. We detect the bin by finding and filtering blobs (according to size and position constrains) that result from the difference of the current camera image to a reference image taken earlier in an initialization step. For the reconstruction of the position of the bin we search for the pixel in the detected blob that has the highest y-value, since this is a relatively strong feature that in most poses corresponds to the same point on the rotational invariant bin (eg. the centre point of the frontal arc of the bins bottom). The resulting image coordinates point is reprojected into 3D space and the intersection with a virtual ground plane is calculated that was determined for a fixed pose of the torso and head in a calibration procedure.

In the next sections we will evaluate the overall performance of the system while using the vision system.

7.3 Training the error compensation function

To train the compensation function, first we designed a exemplary throw configuration (source configuration). We defined an operating range (the range of possible targets) of 1-2.5m to the front of the robot and 1m to the left and right. The throwing distance of the source configuration was approximately 2m straight to the front. This source configuration defines the initial parameter set for the parameter optimization that is performed in order to generate a throw configuration for a new target as described in Section 4).

Now, we randomly defined new targets near the predicted distance of the source throw configuration and generated new throw configurations for these targets. We performed the throws for each target several times to find good optimal position to place the bin for this throw configuration. Finally we took the distance for the bin position using the computer vision system. In total data from 17 throws has been collected for training.

To train the feed forward neural network described in Section 5, each weight in the neural network was initialized with a small random value. Using the normalized predicted distances as input and the also normalized measurements from the vision system as reference output, after approximately 500 iterations of RPROP[5] we stopped optimization. Figure 8 shows the training data for the error compensation function (bigger arrows) as well as the correction function sampled in discrete steps. Here we can see that the internal model, as expected, gives a rather high deviance from the measured distances of 0.388m (RMS error) what implies the necessity of the correction function since with such a prediction error it's very unlikely to hit the target. Using the neural network the error is reduced a lot to 0.088m.

[5] For training we used the implementation given by the FANN library: http://leenissen.dk/fann/wp/

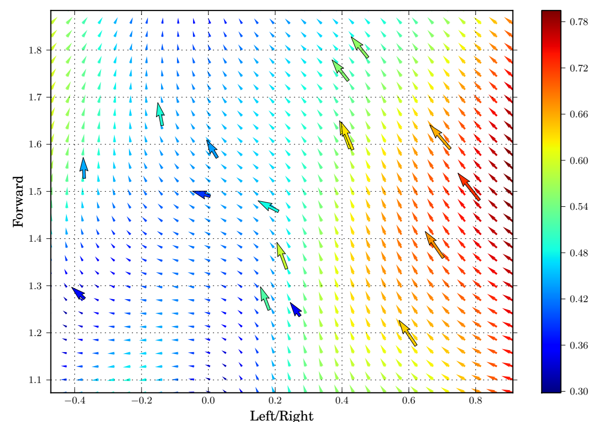


Figure 8. The learned correction function. The tail of the arrows show the predicted position $\mathbf{q}_{\mathrm{pred}}$. The direction points towards the corrected position $\mathbf{q}_{\mathrm{corr}}$ with the color encoding the magnitude of $\mathbf{q}_{\mathrm{corr}} - \mathbf{q}_{\mathrm{pred}}$.

Running the RPROP algorithm for more iterations could further decrease the error, but leads to overfitting to the few data points available what decreases the overall performance which is evaluated below.

7.4 Overall performance evaluation

For evaluating the overall performance, we placed the bin at 5 random positions inside the operating range. The position of the bin was measured using the vision system and for the resulting coordinates a throw configuration was generated with the error compensation enabled. Each throw was then performed 10 times. For every throw it was counted whether the ball landed inside the bin or missed it. Table 1 shows the results of this experiment, resulting in a final success rate of 0.46. The bin that was used in this experiment was cylinder shaped and has a height of 34cm and a diameter of 28cm.

Target	1	2	3	4	5
Success	7	6	1	4	6
Fail	4	4	9	6	4
Success rate	0.64	0.6	0.1	0.4	0.6
Avg. success rate			0.46		

Table 1. Evaluation of the overall performance. First, the bin was detected visually. A throw configuration was generated afterwards and executed multiple times. This was done for five different bin positions. *Success* is the number that indicates how often the ball landed inside the basked and *fail* the number how often it missed.

Table 1 shows a summary of the experiment. The success rate ($\frac{\text{Success}}{\text{Fail+Success}}$) shown there is influenced by the repeatability of executing one particular throw configuration similarly (cf. Section 7 and Figure 7(b)) together with the position error due to the visual bin detection and errors in the simulation error compensation. We can see in Table 1, that for target 3, there was only one successful throw out of 10 trials, what is an indication that the at this point the error compensation failed.

8 Conclusion and Future Work

In this paper we showed an approach to make a anthropomorphic robot throw a ball into a bin that is placed randomly in a region in front of the robot. Instead of completely modeling the problem with an exact physical simulation we proposed a different approach, to use a simple simulation model and compensate the resulting error using machine learning techniques.

To accomplish this we represented parametric joint space trajectories as fifth order polynomials based on Bernstein basis functions. The joint space trajectories are analyzed to identify the task space motion. From the task space motions, based on an incomplete simulation model, an estimation of the position where the ball will land is given. By minimizing a constrained error function, throw movements can be generated, that lead to arbitrary ball landing positions. The deviation between simulation and the real world is compensated using a neural network that maps the estimated landing position of the ball to a corrected one that corresponds to real world positions. Dynamics that are unmodeled in the simulation and calibration errors are compensated by this function. Providing a mapping from estimated to corrected throw positions cascades the underlying parameters to generate the trajectories, but this error compensation only works reliably if the resulting movements are similar. We build a database of parameter configurations that stores example trajectories for distinct target positions. In order to generate similar trajectories, the function minimization is initialized using template configuration that results in a nearby target.

Our solution gives a simple and practical approach to the given problem, but suffers especially from one weakness. Different throw configurations that are predicted by the forward model to have the same ball landing position, but in reality result in different positions cannot be handled by the simulation error compensation in its current form. The correction is solely be made on the predicted target coordinates from the forward model. The underlying parametric configuration of the throw movement is not considered. Instead of learning a correction function, learning the forward model from examples would solve this. Even better would be to learn the inverse model, i.e. a function that directly maps goal coordinates to a proper throw configuration. We want to look at this issue and try different learning methods on this problem. Another possibility is to use reinforcement learning methods, so that the estimation can get gradually better over time with each trial. In [7] reinforcement learning was used to improve an initial control policy given by imitation learning [4] to solve the challenging *ball in a cup* task. One concern with this approach might be, that while with the approach shown in this paper only 17 training samples where it is likely that control policy improvement with reinforcement learning would need more data.

9 Acknowledgements

This work was performed within the iStruct project funded by the German Space Agency (DLR) with federal funds of the Federal Ministry of Economics and Technology (BMWi) in accordance with the parliamentary resolution of the German Parliament, grant no. 50RA1013 and grant no. 50RA1014.

REFERENCES

[1] N. Hansen and A. Ostermeier, ‘Completely derandomized self-adaptation in evolution strategies.’, *Evolutionary computation*, **9**(2), 159–95, (January 2001).

[2] Cedric Hartland and Nicolas Bredeche, ‘Evolutionary Robotics , Anticipation and the Reality Gap’, in *Proceedings of the 2006 IEEE Conference on Robotics and Biomimetics*, pp. 1640–1645, (2006).

[3] T. Hemker, M. Stelzer, O. von Stryk, and H. Sakamoto, ‘Efficient Walking Speed Optimization of a Humanoid Robot’, *The International Journal of Robotics Research*, **28**(2), 303–314, (February 2009).

[4] A. J. Ijspeert, J. Nakanishi, and S. Schaal, ‘Movement imitation with nonlinear dynamical systems in humanoid robots’, in *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No.02CH37292)*, pp. 1398–1403. Ieee, (2002).

[5] Akio Ishiguro, Takeshi Furuhashi, Shigeru Okuma, and Yoshiki Uchikawa, ‘A Neural Network Compensator for Uncertainties of Robotics’, *Industrial Electronics, IEEE Transactions on*, **39**(6), 565–570, (1992).

[6] N. Jakobi, ‘Evolutionary Robotics and the Radical Envelope-of-Noise Hypothesis’, *Adaptive Behavior*, **6**(2), 325–368, (September 1997).

[7] Jens Kober, Betty J. Mohler, and Jan Peters, ‘Imitation and reinforcement learning for motor primitives with perceptual coupling’, in *From Motor Learning to Interaction Learning in Robots*, 209–225, (2010).

[8] Sylvain Koos, Jean-Baptiste Mouret, and Stephane Doncieux, ‘Crossing the Reality Gap in Evolutionary Robotics by Promoting Transferable Controllers’, in *In Proc . of GECCO*, pp. 119–126, (2010).

[9] Martin Riedmiller and Heinrich Braun, ‘A direct adaptive method for faster backpropagation learning: the RPROP algorithm’, in *IEEE International Conference on Neural Networks*, pp. 586–591. Ieee, (1993).

[10] Alan Watt, *3D Computer Graphics*, Pearson Education, Essex, 3. edn., 2000.