Learning Optimal Robot Ball Catching Trajectories Directly from the Model-based Trajectory Loss

Arne Hasselbring[1], Udo Frese[1,2] and Thomas Röfer[1,2]

[1] *Deutsches Forschungszentrum für Künstliche Intelligenz, Cyber-Physical Systems, Bremen, Germany*

[2] *Universität Bremen, Fachbereich 3 – Mathematik und Informatik, Bremen, Germany*

Keywords: Trajectory Optimization, Machine Learning, Robot Dynamics.

Abstract: This paper is concerned with learning to compute optimal robot trajectories for a given parametrized task. We propose to train a neural network directly with the model-based loss function that defines the optimization goal for the trajectories. This is opposed to computing optimal trajectories and learning from that data and opposed to using reinforcement learning. As the resulting optimization problem is very ill-conditioned, we propose a preconditioner based on the inverse Hessian of the part of the loss related to the robot dynamics. We also propose how to integrate this into a commonly used dataflow-based auto-differentiation framework (TensorFlow). Thus it keeps the framework's generality regarding the definition of losses, layers, and dataflow. We show a simulation case study of a robot arm catching a flying ball and keeping it in the torus shaped bat. The method can also optimize "voluntary task parameters", here the starting configuration of the robot.

1 INTRODUCTION

Motion planning (LaValle, 2006) is one of the most central issues for moving autonomous systems, let it be robot manipulators (Singh and Leu, 1991), autonomous vehicles (Lim et al., 2018) or spacecraft (Nikolayzik et al., 2011). In some scenarios, obstacle geometry is the most difficult part, e.g. when a robot mounts a bulky part in a motor compartment. Sometimes it is closed-loop reactivity, e.g. when balancing. And sometimes the dynamics of the system pose the largest challenge, e.g. when the system is operated close to its technical and physical limits and motion planning shall get the best performance out of these limits. Our long term goal is to perform ball sports tasks with a two-arm robot (Fig. 1), such as catching, throwing, or juggling. These require very dynamic movements, as "gravity is not waiting for you", and bring typical industrial robots to their limit.

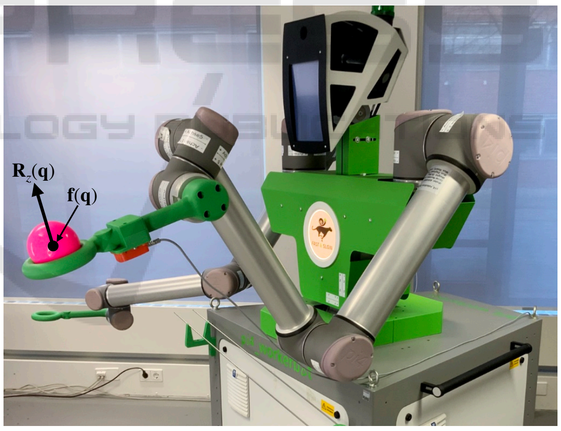

Figure 1: Our long term goal is to perform ball sports tasks, like throwing, catching, juggling, with this two-arm robot. The abstract and passive design of the torus-shaped bat imposes constraints to avoid losing the ball.

1.1 An Optimization View

Often such a planning task is formulated as an optimization problem:

$$\mathbf{q}^*(\mathbf{a}) = \underset{\mathbf{q}_{1\ldots T}\in\mathbb{R}^{T\times D}}{\arg\min}\ \mathcal{L}(\mathbf{a},\mathbf{q}) \qquad (1)$$

Here $\mathbf{q}_{1\ldots T}$ is the trajectory, defining a D-dimensional (joint) position vector per discretized timestep and $\mathcal{L}(\mathbf{a},\mathbf{q})$ is the loss function that defines the optimization goal. The loss function includes a model of the system (here robot dynamics) and a formalization of the task to be achieved, e.g. catching a ball. It also includes constraints, e.g. position, velocity, acceleration, or torque limits, abstractly by setting $\mathcal{L}(\mathbf{a},\mathbf{q}) = \infty$ for infeasible $\mathbf{q}$, or concretely by including a barrier function. The loss depends on some task parameters $\mathbf{a}$ describing the concrete task instance, e.g. a goal pose or – in our case – the trajectory of an incoming ball.

Hasselbring, A., Frese, U. and Röfer, T.
Learning Optimal Robot Ball Catching Trajectories Directly from the Model-based Trajectory Loss.
DOI: 10.5220/0011279000003271
In *Proceedings of the 19th International Conference on Informatics in Control, Automation and Robotics (ICINCO 2022)*, pages 201-208
ISBN: 978-989-758-585-2; ISSN: 2184-2809

If the task input $\mathbf{a}$ is not available beforehand, this optimization creates a delay before the motion can start and is computationally challenging despite modern algorithms and libraries, e.g. CasADI (Andersson et al., 2019) and WORHP (Nikolayzik et al., 2011). Because of this and the recent success of machine learning, it is promising to learn a mapping from $\mathbf{a} \mapsto \mathbf{q}^*$, e.g. with a neural network. One could use supervised learning with computed $(\mathbf{a}, \mathbf{q}^*(\mathbf{a}))$ pairs:

$$\theta^* = \arg\min_{\theta} \mathbb{E}_{\mathbf{a}} \left[\left\| \mathbf{q}_\theta(\mathbf{a}) - \mathbf{q}^*(\mathbf{a}) \right\|^2 \right] \quad (2)$$

Here θ are the learned parameters of a neural network $\mathbf{q}_\theta(\mathbf{a})$ that maps $\mathbf{a} \mapsto \mathbf{q}^*$. The $\mathbb{E}_{\mathbf{a}}$ is an expected value with respect to a distribution (practically a set) of training data for $\mathbf{a}$. Alternatively, one could also run reinforcement learning with $\mathcal{L}(\mathbf{a}, \mathbf{q})$ as reward.

1.2 Learning Directly from the Trajectory's Loss

However, in this paper we investigate to use the model based $\mathcal{L}(\mathbf{a}, \mathbf{q})$ directly as loss for the learning process:

$$\theta^* = \arg\min_{\theta} \mathbb{E}_{\mathbf{a}} \left[\mathcal{L}(\mathbf{a}, \mathbf{q}_\theta(\mathbf{a})) \right] \quad (3)$$

The system and task model are part of $\mathcal{L}$ and thus part of the learning optimization. Typically, such losses are differentiable as needed for gradient descent. This approach appears promising, because:

First, unlike (2), (3) considers which direction of deviation from the optimum affects the loss how much. So it aims for the *best* solution, not the one *closest to the optimum*. This is very relevant for loss terms depending on derivatives of $\mathbf{q}$ (Sec. 4.1).

Second, in reinforcement learning every roll-out produces – roughly speaking – a single total loss value. The distribution of the loss over timesteps is not that helpful, because of the delayed reward problem. In contrast, gradient descent on (3) produces one derivative with respect to every entry of $\mathbf{q}$. Thus the approach takes more information from the loss, utilizing that a model-based loss can be differentiated.

Third, sometimes the task input has a voluntary part $\mathbf{a}_v$ that can be chosen by the system but only before the remaining $\mathbf{a}_{\bar{v}}$ arrives ($\mathbf{a} = (\mathbf{a}_v, \mathbf{a}_{\bar{v}})$). For instance, we can choose the starting configuration $\mathbf{a}_v = \mathbf{q}_0$, but only before the ball approaches, i.e. independent of $\mathbf{a}_{\bar{v}}$, best on average. These parameters can be optimized along in (3) as $\arg\min_{\theta, \mathbf{a}_v} \ldots$, which is not possible in (1), that considers only a single task instance.

However, (3) is more difficult to optimize than (2), because it is conditioned worse (Sec. 4.1). Therefore, we propose an inverse Hessian preconditioning of the robot dynamics to make it tractable for common gradient optimizers. Overall, this paper contributes:

- A method to learn trajectory optimization problems directly from the trajectory's loss and optimizing voluntary task inputs along for loss functions involving robot dynamics.
- The concept and implementation, how this method can be integrated into a common dataflow auto-differentiation framework (TensorFlow).
- A simulation case study, for the task of catching flying balls, including full robot dynamics, choice of starting configuration, and the mechanical constraints for the ball to stay in the bat (Fig. 1).

The paper is structured as follows: After related work we introduce the generic parts of a robot trajectory loss refering to joint velocities and torques (Sec. 3), and propose the inverse Hessian preconditioner and its integration into TensorFlow (Sec. 4). Then, we study the case of ball catching (Sec. 5), and conclude.

2 RELATED WORK

Rao (2014) gives a survey on trajectory optimization. He distinguishes between indirect methods, where the control action is the variable to be optimized, and direct methods, where the trajectory itself is optimized. This part is relevant here, as we substitute the nonlinear trajectory optimizer by a neural network. As Rao says, "Typically, optimal trajectory generation is performed off-line, that is, such problems are not solved in real time nor in a closed-loop manner.", showing the need for efficient approximations.

Toussaint (2017) gives an excellent tutorial on the Newton method applied to trajectory optimization. He highlights relations to other fields, which all use minimization of quadratic functions as a common core, namely Simultaneous Localization and Mapping (SLAM), optimal control and probabilistic inference. In particular, he points out the bandmatrix structure of the Hessian, which we utilize in Section 4.1.

Kurtz et al. (2022) developed a controller that makes a falling Mini Cheetah robot always land on its feet, as cats do. They use an offline trajectory optimizer to either map the current state to joint torques or to map the initial state to a whole trajectory (the so-called *reflex*) and trained two neural networks on that. Experiments showed that the second network performed much better. This resembles the approach presented in this paper, but optimizing (2), not (3).

Mansard et al. (2018) confirm these findings. They use a neural network inside a probabilistic roadmap framework to learn a mapping from task input (start state, goal state, environment) to a) a control action and b) a trajectory. They use supervised learning with ground truth from a model-based optimizer and report that b) works better. Overall they aim at a good initial guess to speed up convergence.

Schüthe and Frese (2015) use full dynamic Model Predictive Control in real-time at 50 Hz to bat a ball. However, their robot only has three joints.

Bäuml et al. (2010) optimize a ball catching trajectory for a 7-DOF arm utilizing the redundancy both of the robot itself and the task in 10 ms on average (2.2 GHz quad-core Xeon). The optimization is kinematic, with joint acceleration instead of torque limits. Hence, they actually optimize the catching configuration using the optimal trapezoidal trajectory.

A large body of work considers kinematic motion planning, without taking dynamics into account but in geometrically complex environments. An example of this class is Schulman et al. (2013) and Ratliff et al. (2009), who showed how to integrate geometric obstacles into an optimization-based approach.

3 DYNAMIC ROBOT LOSS

Every trajectory that is executed on a robot should comply with the robot's physical limits and also minimize the required control effort. This is encoded in the dynamic robot loss defined in terms of torques and joint velocities. Besides the optimization variable $\mathbf{q}_{1\ldots T}$, this depends on boundary conditions as well. The trajectory is extended by $\mathbf{q}_{-1\ldots 0}$ at the start, which represent the current state of the robot (in this paper initial configuration, no velocity). The robot shall stop finally, so $\mathbf{q}_T$ is repeated in the end $\mathbf{q}_{T+1} = \mathbf{q}_T$ forcing the last velocity to zero. Given the now extended discrete trajectory $\mathbf{q}_{-1\ldots T+1}$, the continuous $\mathbf{q}(t)$ is defined by linear interpolation:

$$\mathbf{q}(t) = \mathrm{frac}(t)\mathbf{q}_{\lfloor t\rfloor+1} + (1-\mathrm{frac}(t))\mathbf{q}_{\lfloor t\rfloor} \tag{4}$$

$\dot{\mathbf{q}}(t)$ is then the difference quotient of the adjacent $\mathbf{q}_i$.

$$\dot{\mathbf{q}}(t) = \frac{\mathbf{q}_{\lfloor t\rfloor+1} - \mathbf{q}_{\lfloor t\rfloor}}{\Delta T} \tag{5}$$

and similarly, the accelerations (which are only needed at integer timesteps) are

$$\ddot{\mathbf{q}}(t) = \frac{\mathbf{q}_{t+1} + \mathbf{q}_{t-1} - 2\mathbf{q}_t}{\Delta T^2}. \tag{6}$$

τ: $\mathbb{R}^D \times \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}^D$ is the vector of torques per joint that can be obtained via inverse dynamics from the joint angles, velocities, and accelerations. In the following, $\tau^* \in \mathbb{R}^D$ and $\dot{\mathbf{q}}^* \in \mathbb{R}^D$ are the specified maximum joint torques and velocities, respectively.

The main loss component that smoothens the entire trajectory is the sum of squared torques, normalized by their maximum values:

$$\mathcal{L}_\tau(\mathbf{q}) = \frac{1}{2}\sum_{t=0}^{T}\sum_{j=1}^{D}\left(\frac{\tau_j(\mathbf{q}(t),\dot{\mathbf{q}}(t),\ddot{\mathbf{q}}(t))}{\tau_j^*}\right)^2 \tag{7}$$

The other two components address the hard limits on torques and velocity. Define the function

$$\mathrm{clip}(x,c) = \frac{\max(x-c,0)}{1-c} \tag{8}$$

that maps values below c to 0 and scales them to map 1 to 1. Then, the following loss component penalizes violations of the maximum torques and velocities:

$$\mathcal{L}_{\tau^*}(\mathbf{q}) = \frac{1}{2}\sum_{t=0}^{T}\sum_{j=1}^{D}\mathrm{clip}\left(\frac{|\tau_j(\mathbf{q}(t),\dot{\mathbf{q}}(t),\ddot{\mathbf{q}}(t))|}{\tau_j^*}, c_{\tau^*}\right)^2 \tag{9}$$

$$\mathcal{L}_{\dot{q}^*}(\mathbf{q}) = \frac{1}{2}\sum_{t=0}^{T}\sum_{j=1}^{D}\mathrm{clip}\left(\frac{|\dot{q}_j(t)|}{\dot{q}_j^*}, c_{\dot{q}^*}\right)^2 \tag{10}$$

The thresholds $c_{\tau^*}, c_{\dot{q}^*} < 1$ define a fraction of the actual limits at which the penalty starts, as the terms only define a soft, not a hard limit (cf. Sec. 5.2).

The complete dynamic loss is constructed as a weighted sum of its components:

$$\mathcal{L}_{\mathrm{dyn}}(\mathbf{q}) = \alpha_\tau\mathcal{L}_\tau(\mathbf{q}) + \alpha_{\tau^*}\mathcal{L}_{\tau^*}(\mathbf{q}) + \alpha_{\dot{q}^*}\mathcal{L}_{\dot{q}^*}(\mathbf{q}) \tag{11}$$

4 INVERSE HESSIAN PRECONDITIONING

4.1 The Conditioning Problem

The dynamic loss $\mathcal{L}_{\mathrm{dyn}}$ has directions of very large curvature due to the terms related to $\ddot{\mathbf{q}}$ (and τ), while typical task losses related to $\mathbf{q}$ create directions of much smaller curvature. To see this, consider as illustration the following simplified loss for just one joint:

$$\left(\frac{q_1}{1^\circ}\right)^2 + \left(\frac{q_{100}}{1^\circ}\right)^2 + \sum_{t=2}^{99}\left(\frac{\ddot{q}(t)}{100^\circ/s^2}\right)^2 \tag{12}$$

It demands q_1, q_{100} and $\ddot{q}$ to be zero and punishes 1° as much as $100\,^\circ\,\mathrm{s}^{-2}$. The optimum is clearly $q_t = 0$. If we change all q_t together by $\delta q/10$ (a change of norm δq), the loss grows by $2/100\ \delta q^2$. If we change one q_t in the middle by δq, the loss grows by $\frac{1^2+(-2)^2+1^2}{(100^\circ/s^2\,\Delta T^2)^2}\delta q^2 \approx 1.5\cdot 10^5\ \delta q^2$, with $\Delta T = 125\,\mathrm{s}^{-1}$. This is called bad conditioning and the ratio between the smallest and the largest curvature is the condition number $\approx 7 \cdot 10^6$. For our actual dynamic loss ($\mathcal{L}_{\text{dyn}}$ (11) with weights given in Section 5.4) the condition number is even $\approx 2 \cdot 10^{10}$.

This impedes first-order methods, e.g. gradient descent, because the step rate is limited by the direction of highest curvature, making steps in the direction of lowest curvature extremely small.

While first-order methods are typical for deep learning (and hence well supported in prominent software frameworks), trajectory optimization therefore commonly uses second-order methods (Toussaint, 2017). Properly using Newton's method requires the full Hessian of the loss function w.r.t. the parameters. This can, in general, be intractable. Therefore, many quasi-Newton methods exist, such as L-BFGS (Liu and Nocedal, 1989), which estimate an approximate inverse Hessian from a sequence of gradients. Modifications (Schraudolph et al., 2007) are also applicable to stochastic problems.

4.2 Preconditioning

However, we can take a different approach: Recall that $\mathcal{L}_{\text{dyn}}$ is a sum of squares, so it can be written as

$$\mathcal{L}_{\text{dyn}}(\mathbf{q}) = \frac{1}{2}\|\mathbf{r}(\mathbf{q})\|^2 \tag{13}$$

with $\mathbf{r}(\mathbf{q}) \in \mathbb{R}^{3(T+1)D}$ being a "residual" vector, each element corresponding to one of the summands in Equations 7–10. This residual function has a Jacobian $\mathrm{J}_{\mathbf{r}}(\mathbf{q})$, which has to be calculated anyway for the gradient and we can construct the pseudo-Hessian

$$\mathbf{H} = \mathbf{J}_{\mathbf{r}}(\mathbf{q})^\top \mathbf{J}_{\mathbf{r}}(\mathbf{q}) \tag{14}$$

of $\mathcal{L}_{\text{dyn}}$ w.r.t. the trajectory $\mathbf{q}$ in the common Gauß-Newton approach. The idea is then to use this matrix as a preconditioner for the gradient of the loss w.r.t. $\mathbf{q}$ and propagate the preconditioned gradient further back through the neural network as in first-order gradient descent. Fortunately, $\mathbf{H}$ is symmetric banddiagonal with a $3D-1$ wide band, because each timestep only relates to its two neighbors (Toussaint, 2017, §2.3). This allows to solve for this matrix with a computation time linear in T.

The idea to only compute the pseudo-Hessian of the dynamics loss and not of the total loss corresponds to CHOMP's (Ratliff et al., 2009) idea to only include a smoothing loss in the pseudo-Hessian, however we refer to the torques instead of joint accelerations, i.e. consider the robot dynamics. As they say: "[..] It is useful to interpret the action of the inverse operator $[\mathbf{H}^{-1}]$ as spreading the gradient across the entire trajectory so that updating by the gradient decreases the cost [..] while retaining trajectory smoothness."

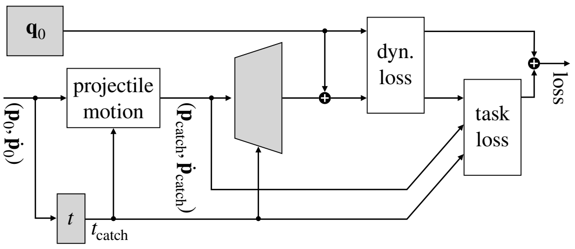
$\mathbf{q}_0$
projectile motion
$(\mathbf{p}_0, \dot{\mathbf{p}}_0)$
$(\mathbf{p}_{\text{catch}}, \dot{\mathbf{p}}_{\text{catch}})$
t
t_{catch}
dyn. loss
task loss
loss

Figure 2: Dataflow of the system during training. The shaded boxes contain learnable parameters while clear boxes are model-based. During execution, the loss boxes are removed.

Note that while only the dynamic part of the loss function $\mathcal{L}_{\text{dyn}}$ is used to construct the Hessian, the whole loss function ($\mathcal{L}_{\text{dyn}} + \mathcal{L}_{\text{task}}$) needs to be multiplied with the inverse Hessian, because

$$\begin{aligned} 0 = \nabla(\mathcal{L}_{\text{dyn}} + \mathcal{L}_{\text{task}}) &\Leftrightarrow 0 = \mathbf{H}^{-1}\nabla(\mathcal{L}_{\text{dyn}} + \mathcal{L}_{\text{task}}) \\ &\not\Leftrightarrow 0 = \mathbf{H}^{-1}\nabla\mathcal{L}_{\text{dyn}} + \mathcal{L}_{\text{task}}, \end{aligned} \tag{15}$$

i.e. optima can change otherwise.

4.3 Integration into TensorFlow

There is a software engineering challenge in integrating the presented preconditioning into a dataflow driven auto-differentiation framework, such as TensorFlow (Abadi et al., 2015). We want to develop the neural network part $\mathbf{q}_\theta(\mathbf{a})$, define task loss functions with specific formulas, and use predefined optimizers as usual. Still, we want to use the established C++ library Pinocchio (Carpentier et al., 2021) to compute τ (inverse robot dynamics) and $\nabla\tau$ and use specialized C++ code to efficiently build and decompose the band matrix $\mathbf{H}$ for the preconditioner. The solution is a "robot dynamics" block (Fig. 2) that takes the trajectory $\mathbf{q}$ including initial conditions $\mathbf{q}_{-1}, \mathbf{q}_0$, along with some attributes such as the path to the URDF robot model. In the forward pass, the block just calculates the dynamic losses $\mathcal{L}_\tau$, $\mathcal{L}_{\tau^*}$, and $\mathcal{L}_{\dot{q}^*}$. However, it additionally outputs a copy of the input $\mathbf{q}$, which further loss calculations must use, such as $\mathcal{L}_{\text{task}}$. Because of this output, the block receives $\nabla\mathcal{L}_{\text{task}}$ during the gradient computation in the backward pass and can apply $\mathbf{H}^{-1}$ to it. The user must take care that the loss outputs of this function appear only linearly in the total loss function, as otherwise the pseudo-Hessian will be wrong. The block is technically a Python wrapper function with custom gradients for two TensorFlow C++ ops, one each for the forward and backward pass. The implementation is available at https://github.com/DFKI-CPS/tf-dynamics-op.

5 BALL CATCHING CASE STUDY

As a case study for our trajectory optimization approach, we consider a ball catching robot. This is a challenging task because several constraints have to be met: The ball must be hit at the right time, with the right velocity and rotation, and be kept from leaving the bat again. This is on top of the general dynamic feasibility and “elegance” of the trajectory.

Our scenario is episodic, i.e. the robot starts from a stationary initial joint configuration $\mathbf{q}_0$, which is the same for all attempts, and included in the optimization. As input, the system gets the initial ball position $\mathbf{p} \in \mathbb{R}^3$ and velocity $\dot{\mathbf{p}} \in \mathbb{R}^3$, e.g. from a tracking system not considered here. The output of the system is a trajectory, i.e. a sequence of T sets of joint angles $\mathbf{q}_{1\ldots T} \in \mathbb{R}^{T \times D}$ that catches the ball and subsequently balances the ball until stopping. The system can choose when to catch the ball, and therefore the state along the ball trajectory. For instance, a later time will make the ball be closer and lower and its velocity faster and steeper (Fig. 2).

5.1 Robot Model

Although this work is carried out entirely in simulation, we still model the setup after the real robot in Figure 1. It is a pi4 workerbot with two 6-DOF Universal Robots UR10 CB2 collaborative robot arms mounted on its “shoulders”. They can be controlled at 125 Hz ($\Delta T = 0.008\,\mathrm{s}$), with joint velocity commands. The robot has three types of motors: The first type is used for the two shoulder joints ($v^*_{1,2} = 120^\circ\,\mathrm{s}^{-1}$, $\tau^*_{1,2} = 330\,\mathrm{N\,m}$), the second for the elbow joint ($v^*_3 = 180^\circ\,\mathrm{s}^{-1}$, $\tau^*_3 = 150\,\mathrm{N\,m}$), and the third for the three wrist joints ($v^*_{4,5,6} = 180^\circ\,\mathrm{s}^{-1}$, $\tau^*_{4,5,6} = 56\,\mathrm{N\,m}$). A model of the mass distribution is available, but to match the torques the UR10 itself calculates, additional torques proportional to the accelerations have to be added, probably to account for motor inertia. We determined those factors experimentally to be $7\,\mathrm{kg\,m}^2$ for the shoulder joints, $2\,\mathrm{kg\,m}^2$ for the elbow joint, and $0.62\,\mathrm{kg\,m}^2$ for the wrist joints. Considering the torques instead of plain joint accelerations is necessary, because the upper arm joints need to lift a significantly higher weight. Keeping the torque limits is particularly important, because the real UR10, instead of just limiting the current, will enter an error state upon violating these.

The “hands” of the robot are torus-shaped with $r_{\mathrm{major}} = 47\,\mathrm{mm}$ and $r_{\mathrm{minor}} = 12.5\,\mathrm{mm}$.

Given a joint configuration $\mathbf{q} \in \mathbb{R}^D$, $\mathbf{f}$, $\mathbf{R}_z$ and $\mathbf{J}$ represent the forward kinematics of the manipulator in the global frame, specifically $\mathbf{f}\colon \mathbb{R}^D \to \mathbb{R}^3$ is the ball’s center on the bat, $\mathbf{R}_z\colon \mathbb{R}^D \to \mathbb{R}^3$ is the direction vector orthogonal to the torus’ plane, and $\mathbf{J}\colon \mathbb{R}^D \to \mathbb{R}^{3\times D}$ is the Jacobian of $\mathbf{f}$ (Fig. 1).

5.2 Task Loss

The task loss’ inputs are the trajectory $\mathbf{q}_{1\ldots T}$, the desired catch time point t_{catch}, and the ball’s predicted state at the catch time $(\mathbf{p}_{\mathrm{catch}}, \dot{\mathbf{p}}_{\mathrm{catch}}) \in \mathbb{R}^3 \times \mathbb{R}^3$. It has the following sum-of-squares-type components:

Position: The ball must be on the bat at the desired catch time.

$$\mathcal{L}_p(\mathbf{q}) = \|\mathbf{f}(\mathbf{q}(t_{\mathrm{catch}})) - \mathbf{p}_{\mathrm{catch}}\|^2 \tag{16}$$

Velocity: The velocity of the ball must match the velocity of the bat at the catch time, up to a given factor $c_v < 1$ (we assume that some kinetic energy can be absorbed on impact, making it an inelastic collision).

$$\mathcal{L}_{\dot{p}}(\mathbf{q}) = \|\mathbf{J}(\mathbf{q}(t_{\mathrm{catch}}))\dot{\mathbf{q}}(t_{\mathrm{catch}}) - c_v\dot{\mathbf{p}}_{\mathrm{catch}}\|^2 \tag{17}$$

Rotation: The bat’s z axis must be aligned with the ball velocity’s direction at the catch time (rotation around the axis through the bat is arbitrary), such that ideally, the ball touches the bat evenly.

$$\mathcal{L}_R(\mathbf{q}) = \|\mathbf{R}_z(\mathbf{q}(t_{\mathrm{catch}})) \times -\dot{\mathbf{p}}_{\mathrm{catch}}\|^2 \tag{18}$$

Note this theoretically allows two optimal solutions pointing in opposite directions, i.e. while $\mathbf{R}_z$ and $\dot{\mathbf{p}}_{\mathrm{catch}}$ should be pointing in opposite directions, they could as well point to the same. This is prevented by other loss components and initial conditions.

Balance: Once the ball has collided with the bat, the ball must be kept in position by a reaction force from the bat, which can only act from within the torus. Otherwise it would fall out. This imposes an upper bound

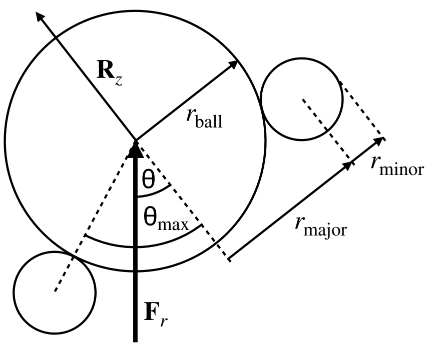

Figure 3: Cross-section of the tilted bat with a ball. The reaction force acting on the ball’s center (here only countering gravity) must stay within the $\theta_{\max}$ cone around $\mathbf{R}_z$.

$\theta_{\max}$ on the angle between the direction $\mathbf{R}_z(\mathbf{q}(t))$ and the reaction force $\mathbf{F}_r(t)$, where

$$\cos(\theta_{\max}) = \frac{\sqrt{(r_{\text{minor}} + r_{\text{ball}})^2 - r_{\text{major}}^2}}{r_{\text{minor}} + r_{\text{ball}}} \tag{19}$$

and (with $\mathbf{g}$ being the gravity vector):

$$\frac{\mathbf{F}_r(t)}{m} = \ddot{\mathbf{p}}(t) - \mathbf{g} \tag{20}$$

$\ddot{\mathbf{p}}(t)$ is calculated as second-order difference quotient in Cartesian space (to avoid needing the Hessian of the forward kinematics) and implicitly captures $\mathbf{q}(t)$, although we actually do not backpropagate the gradient through this:

$$\ddot{\mathbf{p}}(t) = \frac{\mathbf{f}(\mathbf{q}(t+1)) + \mathbf{f}(\mathbf{q}(t-1)) - 2\mathbf{f}(\mathbf{q}(t))}{\Delta T^2} \tag{21}$$

As hard constraint, the balance condition is (Fig. 3)

$$\cos(\theta) = \frac{\mathbf{F}_r(t)^\top \mathbf{R}_z(\mathbf{q}(t))}{\|\mathbf{F}_r(t)\|} \geq \cos(\theta_{\max}). \tag{22}$$

Again, $\mathbf{q}$ is implicitly captured in θ, although here it *does* receive a gradient. For differentiability, the loss is a penalty, i.e. it is 0 below a specified fraction (c_b) of the limit angle $\theta_{\max}$:

$$\mathcal{L}_b(\mathbf{q}) = \sum_{t=\lceil t_{\text{catch}} \rceil}^{T} \max\left(\frac{\|\mathbf{F}_r(t)\|}{m} (\cos(\theta_{\max}) - c_b \cos(\theta)), 0 \right)^2 \tag{23}$$

The complete task-specific loss function is:[1]

$$\mathcal{L}_{\text{task}} = \alpha_p \mathcal{L}_p + \alpha_{\dot{p}} \mathcal{L}_{\dot{p}} + \alpha_R \mathcal{L}_R + \alpha_b \mathcal{L}_b \tag{24}$$

5.3 Network Architecture

There are two neural networks in the system (Fig. 2): One produces the desired catch time point t_{catch} from the initial ball state, while the other produces the trajectory from the catch time point and the predicted ball state at that time.

By choosing a catch time, the network can utilize an additional degree of freedom, namely a shift along the trajectory to find the point where it is most convenient to catch the ball. It has to output this time explicitly and the loss assesses whether the ball and trajectory at t_{catch} are suitable for that.

The prediction of the catch time is subdivided into an analytic component that calculates a base time, and a neural network adding an offset. The base time is chosen as the time when the ball crosses the $z = 0$ plane (a bit below shoulder level).

$$t_{\text{base}} = -\frac{\dot{p}_{0,z} + \sqrt{\dot{p}_{0,z}^2 - 2 p_{0,z} g_z}}{g_z} \tag{25}$$

The initial ball state is then propagated to this time and fed into a neural network together with t_{base}. All values are shifted to lie around 0. The catch time network has three dense layers with 10, 5, and 1 units, respectively. The first two layers are activated by leaky ReLU, while the last one uses tanh. This offset is then scaled to a certain range (we used the factor 0.15) and added to t_{base} to obtain t_{catch}.

After t_{catch} has been determined, the actual ball state at catch time $(\mathbf{p}_{\text{catch}}, \dot{\mathbf{p}}_{\text{catch}})$ is calculated, which is used as input for the trajectory network and reference for the task loss.

The architecture of the trajectory network is described in Table 1. The input layer gets only the ball state, and t_{catch} is injected in later layers. The output of the last layer is added to the broadcasted start angles to form the trajectory that enters the loss function. This way, both the start angles and the trajectory network receive gradients. The length of the trajectory is fixed to $T = 225$ timesteps (1.8 s).

Table 1: Layers of the trajectory network. Residual layers add their input to their output. The last column states whether the catch time is concatenated to the layer's input.

Layer	Units	Activation	Residual	Time
1	20	leaky ReLU	no	no
2	100	leaky ReLU	no	yes
3	101	leaky ReLU	yes	yes
4	102	leaky ReLU	yes	yes
5	150	leaky ReLU	no	yes
6	$D \cdot T$	linear	no	no

5.4 Training Process

From the external perspective (ignoring the preconditioner hidden in the dynamics block), the entire computation graph (i.e. including both neural networks and the start configuration) is trained with plain stochastic gradient descent for 8000 epochs over batches of 16 elements. The inputs $(\mathbf{p}_0, \dot{\mathbf{p}}_0)$ are generated as follows: First a target position is sampled from a subregion of the reachable space of the bat $[0.8, 1.2] \times [0, 0.6] \times [-0.1, 0.2]$ [m]. Independently, a target velocity is uniformly sampled from the range $[-3.7, -2.7] \times [-0.5, 0.5] \times [-4.1, -3.9]$ [m s^{-1}]. These are traced back by a fixed time (1 s) according to projectile motion. Note that this does not exactly cancel out with the projectile

[1]To prevent self-collisions, we added another quadratic term limiting the shoulder lift angle.

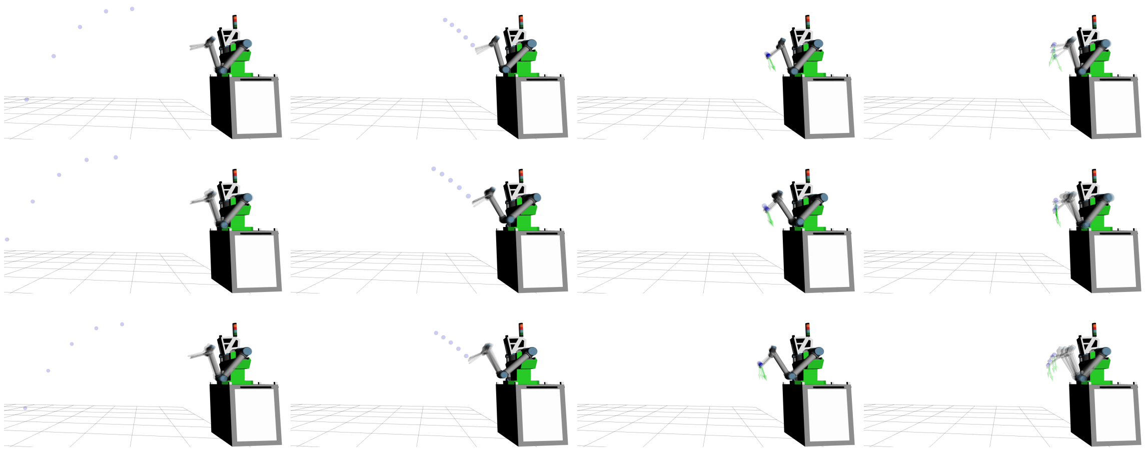
Figure 4: Kinematic visualization of three generated trajectories (from left to right) with different initial ball states. Contact dynamics are not simulated here, instead the ball is attached to the bat at the catch time. The balance constraint is visualized by the direction of the force ($-\mathbf{F}_r$) with a green arrow in the right two frames. A video of multiple trajectories can be found at https://tinyurl.com/e4dk7m6d.

motion block in the system, since the system can adjust the time point and *actual* catch position.

The learning rate starts at $5 \cdot 10^{-4}$ and decays every 100 epochs by a factor of 0.9. The training success is highly sensitive to these learning rate(s).

The weights of the velocity loss $\alpha_{\dot{p}}$ and balance loss α_b are gradually blended in during training. They start at 0 and increase by 0.01 in every epoch in which $\mathcal{L}_p$ is below 0.002, up to a maximum value of 10. The idea is that in the first epochs, the optimizer should mainly try to get the trajectory to a rougly right position and rotation (i.e. solve inverse kinematics first) before turning to velocity and balance, which could otherwise prevent the optimizer from getting there. Similarly, the learning rate for the catch time network is increased, so it starts learning only after the optimizer has roughly reached the right position.

The other loss weights are fixed at $\alpha_\tau = 1$, $\alpha_{\tau^*} = 1$, $\alpha_{v^*} = 5$, $\alpha_p = 1000$, $\alpha_R = 33$. c_{τ^*}, $c_{\dot{q}^*}$ and c_b are set to 0.9 (penalties starting at 90% of the respective limits). These values have been determined by trial and error.

5.5 Results

First of all, qualitative results in form of visualized example trajectories and a video link are given in Figure 4. The trajectories are generally smooth and the bat manages to arrive at the ball in time with suitable rotation and velocity. However, as this is just a visualization without contact dynamics, we can't draw conclusions about the real ball-bat interaction.

Quantitatively, we can examine some metrics on a "test set", i.e. a grid over the input space consisting of 50000 states. The mean loss (with $\alpha_{\dot{p}} = \alpha_b = 10$) over this set is 10.92. No trajectory violates the maximum torque/velocity constraints. However, the mean error in catch position is quite high with 17.3 mm. We assume that position errors larger than 10 mm would cause the ball to bump out of the bat.

To improve the precision, we can calculate the a-posteriori error $\mathbf{p}_{\text{catch}} - \mathbf{f}(\mathbf{q}(t_{\text{catch}}))$ and run the trajectory network again with a modified target position $2\mathbf{p}_{\text{catch}} - \mathbf{f}(\mathbf{q}(t_{\text{catch}}))$. This assumes that the network behaves locally consistent. Indeed, this reduces the mean position error to 10.5 mm.

In order to assess the influence of letting the optimizer choose the catch time, we train the system (i.e. trajectory network and start configuration) without the catch time offset, i.e. setting $t_{\text{catch}} = t_{\text{base}}$. This results in a mean loss of 11.17. Similarly, we can disable optimization of the start configuration and leave it at a reasonable default guess. The resulting mean loss over the test set is 10.96, which seems insignificant compared to the value of 10.92 for the full problem. However, now some of the trajectories violate the velocity limits. This indicates that the optimizer actually utilizes the additional degrees of freedom.

The inference time using TensorFlow on a notebook CPU is about 2 ms including the refinement step.

6 CONCLUSIONS & OUTLOOK

We have shown that it is possible to learn a mapping from task input to optimal trajectory directly from the trajectory's loss, when the inverse Hessian of the robot dynamics loss is used as preconditioner. This can be implemented as a block in TensorFlow with a special bypass output that allows to apply the preconditioner also to the other parts of the loss defined outside that block. The simulated ball catching case study showed that this approach works in practice and can optimize voluntary task parameters along with learning the network, in our case the starting configuration of the robot arm.

Future work will be to improve spatial precision by a more general iteration and also to include a moving horizon scheme, which can handle "infinite" tasks such as juggling and can adapt to sensor input, e.g. a change in ball prediction.

ACKNOWLEDGEMENTS

This work is partially funded by the German BMBF – Bundesministerium für Bildung und Forschung project Fast&Slow (FKZ 01IS19072).

REFERENCES

Abadi, M. et al. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Andersson, J. A. E., Gillis, J., Horn, G., Rawlings, J. B., and Diehl, M. (2019). CasADi: a software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation*, 11(1):1–36.

Bäuml, B., Wimböck, T., and Hirzinger, G. (2010). Kinematically optimal catching a flying ball with a hand-arm-system. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2592–2599. IEEE.

Carpentier, J., Valenza, F., Mansard, N., et al. (2015–2021). Pinocchio: fast forward and inverse dynamics for poly-articulated systems. https://stack-of-tasks.github.io/pinocchio.

Kurtz, V., Li, H., Wensing, P. M., and Lin, H. (2022). Mini Cheetah, the falling cat: A case study in machine learning and trajectory optimization for robot acrobatics. In *2022 IEEE International Conference on Robotics and Automation (ICRA)*.

LaValle, S. M. (2006). *Planning Algorithms*. Cambridge University Press.

Lim, W., Lee, S., Sunwoo, M., and Jo, K. (2018). Hierarchical trajectory planning of an autonomous car based on the integration of a sampling and an optimization method. *IEEE Transactions on Intelligent Transportation Systems*, 19(2):613–626.

Liu, D. C. and Nocedal, J. (1989). On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45:503–528.

Mansard, N., del Prete, A., Geisert, M., Tonneau, S., and Stasse, O. (2018). Using a memory of motion to efficiently warm-start a nonlinear predictive controller. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*.

Nikolayzik, T., Büskens, C., and Wassel, D. (2011). Nonlinear optimization in space applications with WORHP. Technical Report Berichte aus der Technomathematik, 11-10, University of Bremen.

Rao, A. V. (2014). Trajectory optimization: A survey. In Waschl, H., Kolmanovsky, I., Steinbuch, M., and del Re, L., editors, *Optimization and Optimal Control in Automotive Systems*, volume 455 of *LNCIS*, pages 3–21. Springer.

Ratliff, N., Zucker, M., Bagnell, J. A., and Srinivasa, S. (2009). CHOMP: Gradient optimization techniques for efficient motion planning. In *2009 IEEE International Conference on Robotics and Automation (ICRA)*.

Schraudolph, N. N., Yu, J., and Günter, S. (2007). A stochastic quasi-Newton method for online convex optimization. In *11th International Conference on Artificial Intelligence and Statistics (AIstats)*, volume 2 of *PMLR*, pages 436–443.

Schulman, J., Ho, J., Lee, A., Awwal, I., Bradlow, H., and Abbeel, P. (2013). Finding locally optimal, collision-free trajectories with sequential convex optimization. In *Robotics: Science and Systems*, volume 9.

Schüthe, D. and Frese, U. (2015). Optimal control with state and command limits for a simulated ball batting task. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3988–3994. IEEE.

Singh, S. K. and Leu, M. C. (1991). Manipulator motion planning in the presence of obstacles and dynamic constraints. *The International Journal of Robotics Research*, 10(2):171–187.

Toussaint, M. (2017). A tutorial on Newton methods for constrained trajectory optimization and relations to SLAM, Gaussian process smoothing, optimal control, and probabilistic inference. In Laumond, J.-P., Mansard, N., and Lasserre, J.-B., editors, *Geometric and Numerical Foundations of Movements*, volume 117 of *STAR*, pages 361–392. Springer.