Artificial Intelligence for Games (2019)
Prof. Dr. Köthe

Training robots with machine learning: Juggling & Throwing

Seminar report by

Stephen Schaumann

Submitted on July 15, 2019

Abstract

This report looks at the paper “Robot Juggling: An Implementation of Memory-based Learning” by Stefan Schaal and Christopher G. Atkeson [1], released 1994, describing their realization of a robot learning the task of juggling via machine learning. Basic concepts and techniques used in the paper will be introduced. At the end there will be also be a short summary of the more recent paper “TossingBot: Learning to Throw Arbitrary Objects with Residual Physics” by A. Zeng, S. Song, J. Lee, A. Rodriguez, and T. Funkhouser [2], in which they show how a robot arm can learn to throw objects by combining physical models and machine learning.

Contents

1 Introduction **3**
1.1 Training robots with machine learning . . . 3
1.2 A general description of robot-tasks . . . 3
1.3 Selecting a suited model . . . 4

2 Robot juggling with memory-based learning (1994) **5**
2.1 Locally Weighted Regression . . . 5
2.2 Shifting Setpoint Exploration Algorithm . . . 6
2.3 Devil sticking . . . 8
2.3.1 Task overview . . . 8
2.3.2 Results . . . 10

3 TossingBot (2019) **13**
3.1 Task and approach overview . . . 14
3.2 Results . . . 15

4 Conclusion **16**

Bibliography **17**

Chapter 1

Introduction

1.1 Training robots with machine learning

Robots are a highly valuable asset with many useful applications. Training robots using machine learning to improve their efficiency and range of skills is obviously quite desirable, but has been a hard problem for many years now. Unlike many digital applications of machine learning, robots are confined in the physical world with all it's limitations. Executing trial runs takes much, *much* slower compared to digital tasks. Resetting a setup is also not trivial, often consuming a lot of time, while being essential for many learning methods. Decisions also have to be made fast enough. For example, a self-driving car can't take a couple of seconds to decide whether it is about to run over a pedestrian.
Despite being hard to achieve, the gain from teaching robots via machine learning is big enough that several methods have evolved during the last decades. We will take a closer look at two of them in the following sections.

1.2 A general description of robot-tasks

The general form of a problem for a robot to solve is the following:

- Some input is given to the robot from the environment, either continually or in discrete time steps. This will most likely come from a camera or some kind of sensor (or a combination of multiple sources)

- Based on the received input, the robot issues one or several commands. This could be signals to turn a motor into a certain position, throttle/iincrease some thrust, etc.

- The commands issued should lead towards the goal. This can be simple, direct goal, but more often it can be very vague. Defining the goal of the robot can be a challenging task in itself.

1.3 Selecting a suited model

Often, when a robot learns a task, it has to gain some understanding of the world it operates in. This can be in the form of a machine learning model that, after training for a while, can predict the outcome of actions following some input state. Machine learning offers many possible ways to approach this. We can broadly divide them into two types of models: *parametric* and *non-parametric* models.
A *parametric* model is a mathematical function with a finite set of free parameters. It fits a *global* function, meaning this one function is fitted to the whole input-space. Data collected during training is not kept. All information is contained in the free parameters, which are fitted during training. Examples for this kind of model are *Linear regression* and *Neural networks.*
On the other hand there are *non-parametric* models. They have a potentially unlimited set of free parameters. This means that during training the parameters can not only be improved, but also extended by new ones. Unlike a *global* function fit, a *local* function fitting does not try to approximate the whole space at once. Instead, it depends on the location one is looking at. These methods can remember data gathered during training. Examples are *(N-)Nearest Neighbor* and *Kernel Regression.*
Remembering past data can be advantageous, as no information can be lost. The downside is that this will get more resource-intensive as the training progresses, as more and more space is needed to save the data. A program has to run fast enough with a lot of data and also needs to learn fast enough before too much data is gathered for it to be actually useful.

Chapter 2

Robot juggling with memory-based learning (1994)

2.1 Locally Weighted Regression

The main method introduced in [1] is the *Locally Weighted Regression* (LWR). It is non-parametric and memory-based. By using previous data it estimates a local linear model for a point at lookup. It offers various statistical tools to assess the reliability of lookups, optimize the quality of lookup, and handle noise and corrupted data.

Classic (unweighted) regression works by finding the solution β to the equations $y = X\beta$, where X is a $m \times (n+1)$ matrix containing m data points with n input dimensions, respectively, and y is a vector holding the corresponding result/output for the entries in X. In practice, this is done by solving the equation $X^T X\beta = X^T y$. To get a prediction for a query point x_q one then simply inserts it to get $\hat{y_q} = x_q^T\beta$.
A problem in this is that each point is equally weighted. This means that for the query result a point in close proximity has as much influence as a point far off. When we are only interested in the prediction at x_q, we would like to give similar points more importance. Therefore, the influence of the points gets weighted by distance.

This leads to the LWR. For each stored data point i we calculate the squared distance to the query point: $d_i^2 = \sum_{j=1}^{n} s_j \left(X_{ij} - x_{q_j}\right)^2$. Each entry is then weighted by $w_i = f\left(d_i^2\right)$, with some weighting function f. A simple choice would be to use $w_i = \frac{1}{d_i^k}$. While this choice would succeed in giving close points more importance, when the distance goes towards zero the weight will go towards infinity. This would cause the LWR to exactly replicate the stored data point, which is undesired since the data is noisy. A better choice, which is also chosen in the paper, is $w_i = exp\left(\frac{-d_i^2}{2k^2}\right)$. Here, k scales the kernel size,

giving a control on how local the model should be.
Each row in X and y is then multiplied by the corresponding weight w_i before doing the regression. Additionally, they used *Ridge Regression* for better noise handling. Ridge Regression extends the classic regression with a diagonal matrix Λ with small positive entries. The equations to solve are then $\left(X^T X + \Lambda\right) \beta = X^T y$, with X and y already weighted by distance.
A comparison of LWR to other common non-parametric function approximation techniques can be seen in Figure 2.1

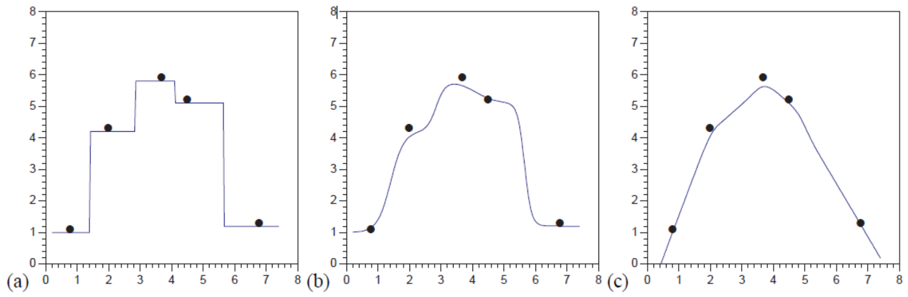


Figure 1: Characteristic performance of three different nonparametric function approximation techniques: (a) nearest neighbor; (b) weighted average; (c) locally weighted regression

Figure 2.1: Comparison of LWR to other non-parametric function approximation techniques. Taken from [1]

2.2 Shifting Setpoint Exploration Algorithm

To gain understanding about the world, the robot needs to explore the available space. This will collect data points, which will be used for further predictions. The problem with this is: the input space is usually very high-dimensional and the collected data is sparse. More inputs to the robot offer more information, but also increase the space that needs to be explored. Since the robot's speed is limited by the real world, it can only collect data slowly, making the progress even harder. It may also not be safe to explore all possible regions, or very costly (e.g. when the robot is damaged). For example, a self-driving car shouldn't need to run over pedestrians to learn how to stay on the road. Thus, random search is not a feasible option.

For this matter the authors introduce the *Shifting Setpoint Exploration Algorithm* (SSA). The idea is to approach exploration slow and steady, building up understanding in a region before shifting focus somewhere else. The task is broken down into two parts:

- Fast timescale: The system is kept under control at certain fixed points. These may not be the optimal solution at first
- Slow timescale: The fixed setpoints are slowly shifted towards the actual goal

By exploring around the fixed setpoints, the robot ensures confidence in that region. By shifting the setpoints towards the goal, the robot always knows what it's doing in the current region, learning along the way.
They then explain the algorithm in more detail using a simple example, a car driving along a mountain road (see Figure 2.2).

The task is to drive the car at a constant horizontal speed $\dot{x}_{desired}$ (in this case $0.8m/s$) from left to right, while minimizing fuel consumption. Interaction takes place at discrete time steps with $5Hz$. There is noisy feedback of the position x and speed $\dot{x}$ of the car (in the following written as x, containing both position and speed). In return, the thrust F can be controlled.
SSA is initialized by executing the following steps:

1. Start at a random location (which is kept fixed)
2. Execute a few random trials
3. Search for the point with highest confidence (determined via prediction intervals. See [1]) and declare it as setpoint:
   $\left(x_{S,in}^T,\ F_S,\ x_{S,out}^T\right)^T$
4. Try to reach the setpoint from each new trial

To try to reach the setpoint they used a LQ-controller (Linear-quadratic regulator) which utilizes predictions from the LWR. After the initialization, the following steps are executed continuously:

1. Learn to reach the setpoint until a certain confidence (based on prediction intervals) is reached. This means starting always at the same point and trying to reach the setpoint, based on experiences so far
2. Take the derivative of $x_{desired} - x_{S,out}$ w.r.t. the command F_S
3. Calculate the correction ΔF_S and update the setpoint-thrust:
   $F_S = F_S - \Delta F_S$
   Calculate new setpoint by applying the updated thrust to the old setpoint.

4. Assess the fit at the updated setpoint. If the confidence is already high enough, terminate. Otherwise continue with step 1

The results can be seen in Figure 2.3. As can be seen, the car only explored a (mostly) connected region, and only where it is necessary. Inside the explored regions, data is quite dense, giving confident control.

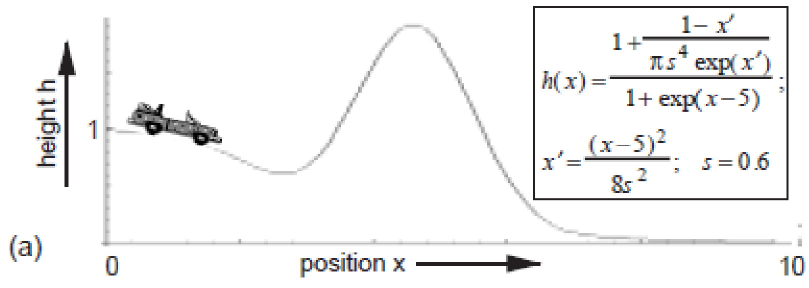

height h
1
(a)
0
position x
10
$h(x) = \frac{1 + \frac{1 - x'}{\pi s^4 \exp(x')}}{1 + \exp(x-5)}$;
$x' = \frac{(x-5)^2}{8s^2}$; $s = 0.6$

Figure 2.2: Overview of the example problem to showcase the SSA. The goal is for the car to drive with a constant horizontal speed to the right while minimizing fuel consumption. Taken from [1]

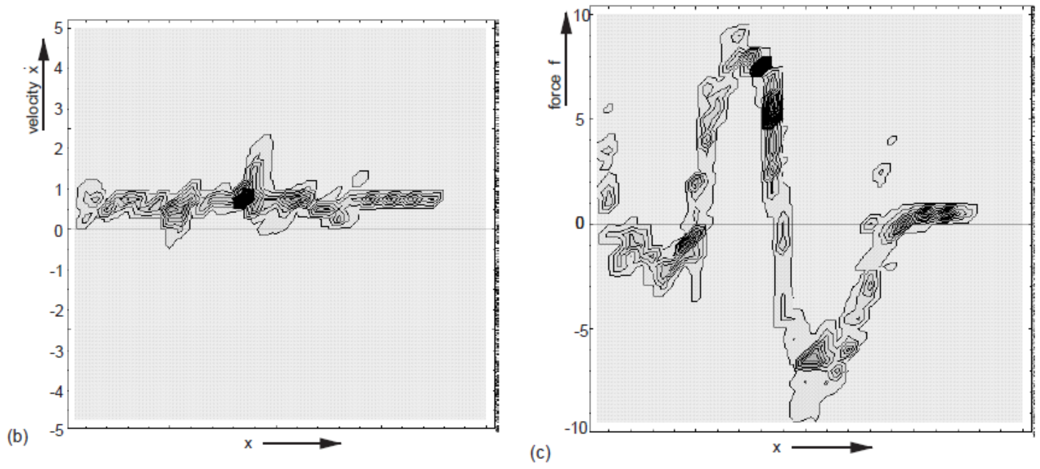


Figure 2.3: Results for the car-example using SSA. On the left is the explored region in phase space, on the right the explored thrust depending on the location. Taken from [1]

2.3 Devil sticking

2.3.1 Task overview

The main goal of this paper was for a robot to learn how to juggle. Slightly misleading, what is actually meant is so-called *Devil sticking*. Devil sticking has the person (or robot) holding two sticks, with the goal to throw a third stick back and forth between the two. An overview of this can be seen in Figure 2.4.

Controllable by the robot are three motors, which are shown by τ_1, τ_2, τ_3 in the diagram. For this paper they did not use the motor τ_3. The devil stick itself is mounted on a boom, causing it to move on the surface of a circle. Since the radius is big enough compared to the region of interest, it moves approximately on a plane.

A state vector of the task is a vector $x = \left(p, \theta, \dot{x}, \dot{y}, \dot{\theta}\right)^T$. It corresponds to the impact state of the devil stick with a hand stick's nominal position. The nominal position is an arbitrary, fixed position for the hand stick. In this case, the normal upright positions as seen in 2.4 are used as nominal positions. The components of the state vector are:

- p : Distance of the devil stick's center of mass to the impact point with nominal position
- θ: Angle of the devil stick at impact
- $\dot{x}, \dot{y}$: Horizontal and vertical speed of the devil stick at impact
- $\dot{\theta}$: Angular velocity at impact

After the stick is thrown, the impact with the other hand is estimated from the trajectory. This uses information from the boom holding the devil stick and classical trajectory equations. The estimated impact is then given as input to the other hand, which then computes what action to take. Both sticks act independently, learning only for themselves.

Commands are issued in the form of a vector $u = \left(x_h, y_h, \dot{\theta}_t, v_x, v_y\right)^T$. x_h and y_h define the displacement of the hand stick from it's nominal position at the end of it's movement. The center stick angular velocity threshold $\dot{\theta}_t$ controls when the movement of the hand stick should start. Lastly, v_x and v_y give the speed at which the hand should move.

Each throw generates an experience vector $\left(x_k^T, u_k^T, x_{k+1}^T\right)^T$, containing the state vector before and after the throw, as well as the performed action.

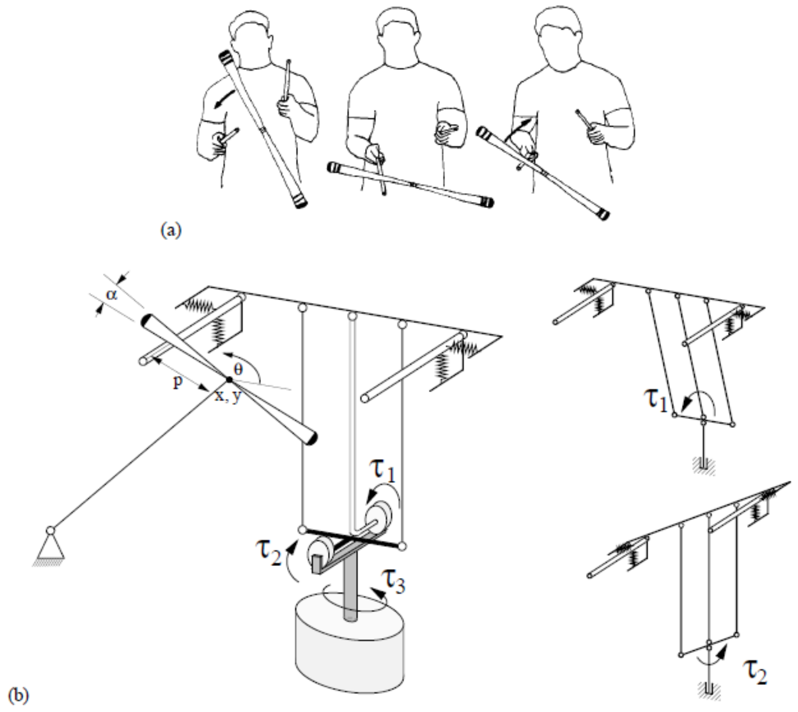


Figure 2.4: Schematic overview of (a) Devil sticking and (b) the setup used to implement it. The possible movements of the robot are shown by τ_1, τ_2, τ_3. Both hand sticks are dampened, so that the stick does not bounce off them, but needs to be actively thrown. Taken from [1]

2.3.2 Results

Training the juggling robot worked similarly to the car-example from the previous chapter. A simplified visualization of the gathered experiences can be seen in Figure 2.5.

(a) Initially, the robot starts with some random throws. As is to expect, the stick does not end up in the initial position after the second hand throws it back. There is only sparse data in a small region.

(b) The robot continues with the "bad" throws, but gathers more experience in the corresponding regions. It's not at the goal, but in the region where it operates, it knows what it's doing

(c) When confidence is high enough, the setpoints shift to match each other. The robot explores the new region, extending it's previous knowledge.

(d) Both setpoints match each other: the target of one setpoint is the starting position of the other and vice versa. At this point, the juggling becomes continuous.

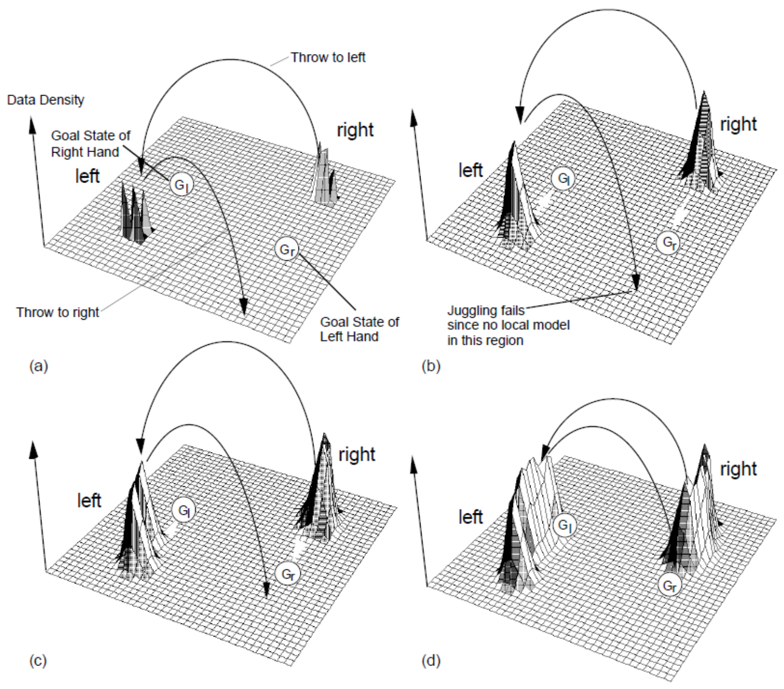


Figure 2.5: Simplified visualization of gathered experiences in phase space durin juggling training. The model reinforces knowledge in a region before moving the setpoints towards each other. Over time, via the SSA, the setpoints match up and the juggling is continuous. Taken from [1]

The learning curves for different runs can be seen in Figure 2.6. First they tested the approach in a simulation. For most of the training, the robot only makes a couple of hits per trial. This makes sense, since it builds up it's confidence in the region it randomly started at. Shortly after 40 trials the setpoints have reached each other, meaning the robot can succesfully throw the stick to a position the other hand knows. When this point in training is reached, the robot suddenly performs hundreds of hits per trial. The simulation was quickly stopped after this, since it always reached the maximum limit of 200 hits from there on.
Then a real robot was trained on the task. The start is again completely flat, with hardly any hits. But there is no sudden change when the robot masters the task. The authors claim this is due to too sparse data. When the robot

knows how to reach a certain state, it repeats *exactly* the same steps. If there is a slight change, it does not know how to react appropriately. The result is therefore not stable.
To fix this they added some small noise to the commands issued by the robot. This lead to more exploration (though still confined to a small region), which made the robot's actions more stable. The resulting curve looks almost exactly like in the simulation, seemingly mastering the task suddenly after only a few trials more.

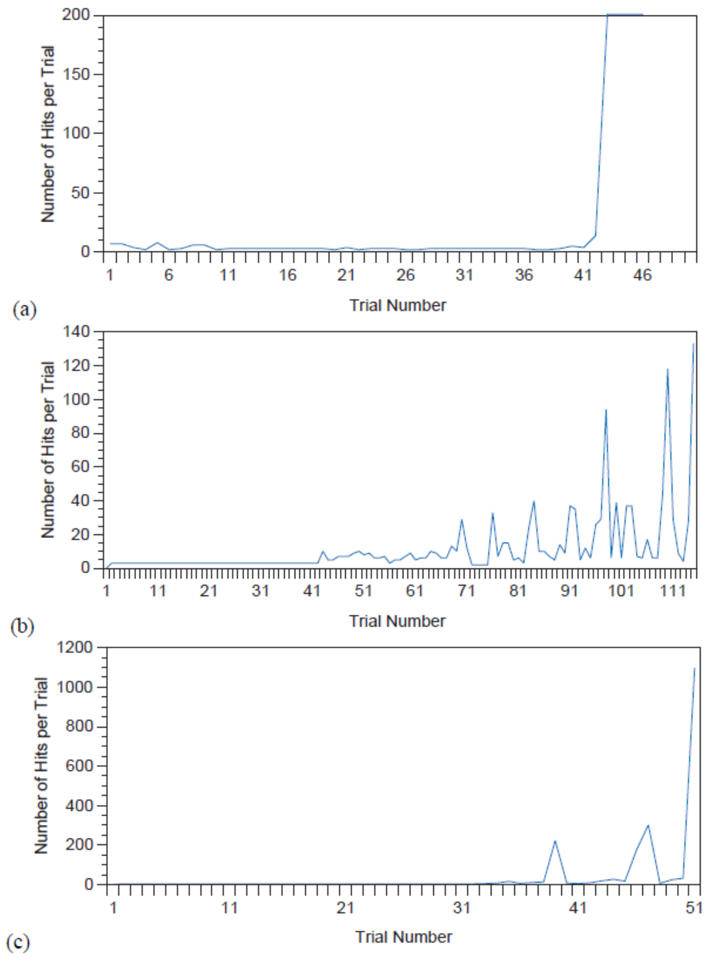


Figure 2.6: Learning curves of the devil-sticking robot when using a (a) Simulation (b) Real robot (c) Real robot with small random noise added to commands. Taken from [1]

Chapter 3

TossingBot (2019)

This section will give a brief overview over the work shown in [2].
TossingBot consists of a robot arm which is tasked to grab and throw arbitrary objects. As can be seen in Figure 3.1, this is a challenging task, since it heavily depends on the shape of the object and where it is grabbed. Previous work mostly focused on fixed shapes. TossingBot on the other hand generalizes to new objects it has never seen before. They do this by using an approach they call *Residual Physics*, which combines known Physics and machine learning.
To simplify the task, the robot arm does not actually learn the whole throw. Instead, the authors set a fixed throwing angle of 45° upwards, pointing towards the target. Only the release velocity needs to be found.

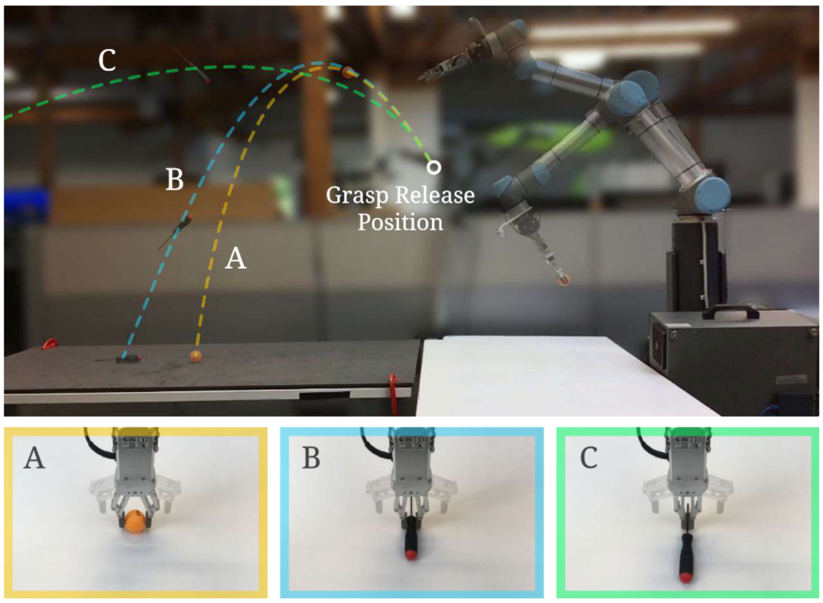
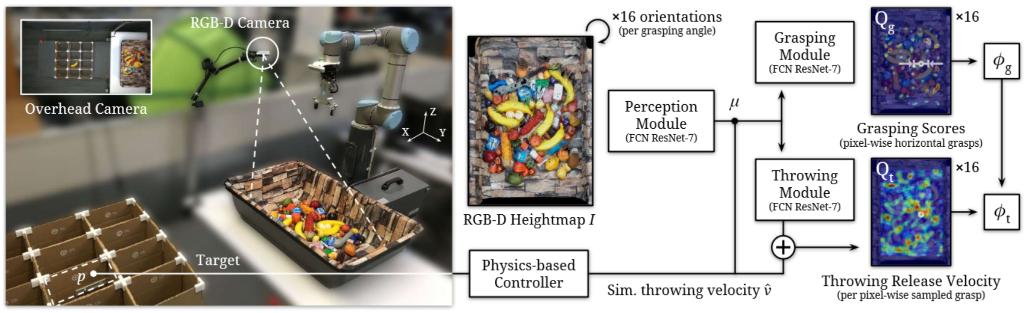


Figure 3.1: Flight trajectories of objects thrown with same force and direction. The trajectories massively change depending on the object shape and where it was grabbed. Taken from [2]

3.1 Task and approach overview

TossingBot's task is made up of two subtasks: grasping and throwing. First it has to grab an object out of a box with many possible items. Then it has to throw this object into one of multiple boxes.
The input is given in the form of a RGB-D picture from an overhead camera and the position of the target box in real world coordinates. Using the target position and a Physics-based Controller which implements a classical trajectory calculation, a throwing velocity is calculated. Instead of doing everything, the machine learning model only tries to find a correction for this physical prediction. This combination of Physics and machine learning is what the authors call *Residual Physics*.
To incorporate the possible rotations of the robot gripper, the image is given in 16 increasingly rotated orientations. The images are first fed through a 7-layer fully convolutional residual network (FCN ResNet) that includes two layers of 2×2 max-pooling to reduce the image size. This is called the *Perception Module*. From the Perception Module the information is passed on to both the *Grasping Module* and *Throwing Module*. Additionally, the Throwing Module is also given the calculated throwing velocity prediction. Both modules consist of a FCN ResNet with two upsampling layers, resulting in a picture with the same size as the initial input.
The Grasping Module returns a map with scores that indicate how good a grasp could be performed at this location and angle that covers the 16 rotations of the camera picture. Similarly, the Throwing Module also returns a map, corresponding to the throwing release velocity that should be used when a grasp is attempted at this location. The system chooses the grasp with the highest score and then throws it with the corresponding correction from the Throwing Module.

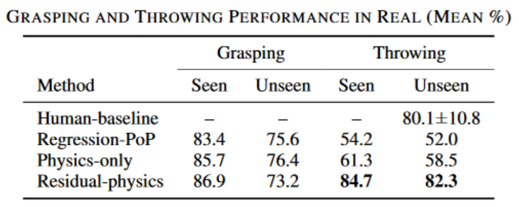


RGB-D Camera
Overhead Camera
Target
p
×16 orientations
(per grasping angle)
RGB-D Heightmap I
Perception Module (FCN ResNet-7)
μ
Grasping Module (FCN ResNet-7)
Throwing Module (FCN ResNet-7)
Physics-based Controller
Sim. throwing velocity v̂
Q_g
Q_t
×16
Grasping Scores (pixel-wise horizontal grasps)
Throwing Release Velocity (per pixel-wise sampled grasp)
φ_g
φ_t

Figure 3.2: Overview of TossingBot's task and structure for solving. Taken from [2]

3.2 Results

TossingBot showed to be able to learn very succesful. As can be seen in Figure 3.3, TossingBot (listed as *Residual-physics*) outperforms both it's single components, namely the Physics-only and the pure Regression approach. When it grasped an unseen object succesfully, it even outperformed humans at throwing (although humans could probably increase their accuracy with some practice). Throwing generalizes well to unseen objects, getting almost the same accuracy. In this subtask it greatly outperforms the other methods. Grasping is a bit worse, going from 86.9% grasping success on seen objects to 73.2% on unseen objects. Here, all methods perform similarly.


GRASPING AND THROWING PERFORMANCE IN REAL (MEAN %)

	Grasping		Throwing	
Method	Seen	Unseen	Seen	Unseen
Human-baseline	–	–	–	80.1±10.8
Regression-PoP	83.4	75.6	54.2	52.0
Physics-only	85.7	76.4	61.3	58.5
Residual-physics	86.9	73.2	**84.7**	**82.3**

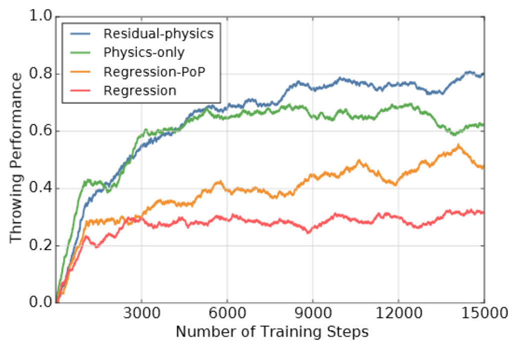


Figure 3.3: (Left) Performance of TossingBot on seen and unseen objects. (Right) Learning curves of different methods. Taken from [2]

Chapter 4

Conclusion

Teaching robots how to learn certain tasks has been a challenge for a long time, constrained additionally by physical limitations compared to purely digital machine learning. Over the years, various methods have emerged to tackle this problem with different strategies.
The first paper, released in 1994, showed a succesful approach using *Locally weighted regression*. Already at such early years, the method proved succesful to train a robot to perform devil-stick juggling. It still has it's limitations: the memory-based method gatheres more and more data during training. More data increases the lookup-time, limiting how much the robot can be trained before it gets too slow.
A more modern approach was shown with TossingBot, which uses *Deep Learning* in a hybrid method called *Residual Physics*. This combination of classical physics and machine learning proved to be more succesful than both of it's parts alone to train a robot arm to pick up arbitrary objects and throw them to certain locations. Perhaps this kind of hybrid model will be picked up more in the future, as it seems a promising approach to utilize modern machine learning techniques in an effective way, together with theoretical physical knowledge.

Bibliography

[1] Stefan Schaal and Christopher G. Atkeson. Robot juggling: Implementation of memory-based learning. *Control Systems, IEEE*, 14:57 – 71, 03 1994. doi: 10.1109/37.257895.

[2] Andy Zeng, Shuran Song, Johnny Lee, Alberto Rodriguez, and Thomas A. Funkhouser. Tossingbot: Learning to throw arbitrary objects with residual physics. *CoRR*, abs/1903.11239, 2019. URL `http://arxiv.org/abs/1903.11239`.