Learning physical Models of Robots

Jochen Mück
Technische Universität Darmstadt
jochen.mueck@googlemail.com

Abstract

In robotics good physical models are needed to provide appropriate motion control for different types of robots. Classical robotics falls short, when models are hand-crafted and unmodeled features as well as noise cause problems and inaccuracy. Based on the paper **Scalable Techniques from Nonparametric Statistics for Real Time Robot Learning** [4] by *Stefan Schaal, Christopher G. Atkeson and Sethu Vijayakumar* this review paper discusses the model learning problem, describes algorithms for locally weighted learning and presents some real world applications.

1 Introduction

Motion-Control is one of the most important basic tasks in robotics. The classical robotics approach, where a physical model of the robot is hand-crafted by an engineer before the robot is actually used, quickly comes to its limits because of the lack of accuracy in the model and noise caused by unmodeled features and change in characteristics due to environmental influence. In classical robotics simple control laws (e.g. PID-Controller) are used to compensate these errors.
In contrast the robot learning approach is to constantly learn and improve a physical model of the robot using recorded data from the robots joints. The concept of self-improvement has the flexibility to handle noise and unmodeled features. Hence, using learned models can help generating better control-laws, thus motion-control tasks can be performed with higher accuracy.
Challenges in model learning for robots are the high dimensionality of the problem (e.g up to 90 dimensions for a humanoid robot) and the need to calculate and improve the model in real-time given a continuous stream of data. Furthermore using on-line robot learning on autonomous systems requires fast algorithms since high performance hardware usually is not available.
There are many different learning methods to solve a learning problem. But since large amounts of data has to be handled and inexpensive algorithms are preferred for on-line model learning, not all of these can be applied to model learning for robots.
In this paper locally weighted learning methods (LWL) are described, which are suitable for robot learning problems. Before the algorithms are presented in section 3, some foundations of model learning are discussed in the next section which will also give a short overview about statistics, different types of models and learning architectures. Finally real-world model learning applications are described in section 4.

2 Foundations of Model Learning

This sections gives an overview about model learning. First the Model Learning Problem and a solution using regression methods and gradient descent search are shown. Afterwards different types of models are explained. At last learning architectures, which make use of models, are described.

2.1 The Model Learning Problem

In robot control, generally different model learning problems are interesting.
These are:

- Forward Kinematics

$$\begin{aligned} \mathbf{x} &= f(\mathbf{q}) \\ \dot{\mathbf{x}} &= \mathbf{J}(\mathbf{q})\dot{\mathbf{q}} \\ \ddot{\mathbf{x}} &= \dot{\mathbf{J}}(\mathbf{q})\dot{\mathbf{q}} + \mathbf{J}(\mathbf{q})\ddot{\mathbf{q}} \end{aligned} \tag{1}$$

- Inverse Kinematics

$$\mathbf{q} = f^{-1}(x) \tag{2}$$

- Forward Dynamics

$$\ddot{\mathbf{q}} = \mathbf{M^{-1}}(\mathbf{q})(\mathbf{u} - \mathbf{c}(\dot{\mathbf{q}}, \mathbf{q}) - \mathbf{g}(\mathbf{q})) \tag{3}$$

- Inverse Dynamics

$$\mathbf{u} = \mathbf{M}(\mathbf{q})\ddot{\mathbf{q}}_\mathbf{d} + \mathbf{c}(\dot{\mathbf{q}}, \mathbf{q}) + \mathbf{g}(\mathbf{q}) \tag{4}$$

All of these model learning problems except the inverse kinematics can be solved using regression methods. The forward dynamics equation 3 describes the joint accelerations $\ddot{\mathbf{q}}$ given the applied torque $\mathbf{u}$ using the physical parameters $\mathbf{M^{-1}}$, $\mathbf{c}(\dot{\mathbf{q}}, \mathbf{q})$ and $\mathbf{g}(\mathbf{q})$. In contrast the inverse dynamics equation 4 describes which torques have to be applied to the actuators to achieve the desired joint accelerations $\ddot{\mathbf{q}}_\mathbf{d}$. Hence, the task to learn a physical model of a robot is to determine the parameters $\mathbf{M}$, $\mathbf{c}(\dot{\mathbf{q}}, \mathbf{q})$ and $\mathbf{g}(\mathbf{q})$.
More generally the model can be written as:

$$y = \mathbf{f}_\theta(\mathbf{x}) + \epsilon \tag{5}$$

The function $\mathbf{f}_\theta(\mathbf{x})$ can also be written as $\phi(\mathbf{x})^\mathbf{T}\theta$ with parameters θ and features ϕ. ϵ denotes Gaussian distributed noise.
The goal of model learning is to find parameters θ, so that a cost-function $\mathbf{J}$ is minimal. $\mathbf{J}$ usually is defined as a least squares cost-function:

$$\mathbf{J} = \frac{1}{2}\sum_{i=1}^{N}(y_i - \mathbf{f}_\theta(\mathbf{x_i}))^2 \tag{6}$$

Which can also be written as:

$$\mathbf{J} = \frac{1}{2}(\mathbf{Y} - \phi\theta)^T(\mathbf{Y} - \phi\theta) \tag{7}$$

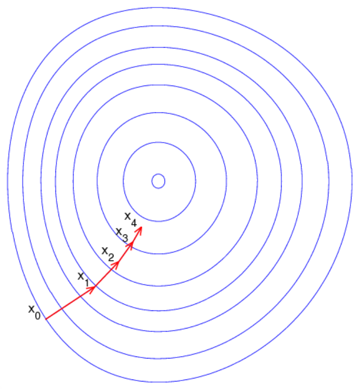
To find parameters θ with minimum cost-function $\mathbf{J}$ can be done with gradient descent search (see section 2.1.1). A closed form for the solution can be derived from gradient descent search and can be written as:

$$\theta = (\phi^\mathbf{T}\phi)^{\mathbf{-1}}\phi^\mathbf{T}\mathbf{Y} \tag{8}$$

Equations 5 to 8 show how a model learning problem looks like and how it can be solved using basic linear regression methods.

2.1.1 Gradient Descent Search

Figure 1: Gradient Descent Search



Gradient descent search is an iterative method to find a local minima of a multi-variable function $\mathbf{F}(\mathbf{x})$. The idea is to iteratively take small steps into the direction of the function's gradient at a certain point $\mathbf{x_k}$, given a random starting point. The step-size can be modified with a parameter γ. The iteration-step can be written as:

$$\mathbf{x_{k+1}} = \mathbf{x_k} - \gamma \nabla \mathbf{F}(\mathbf{x_k}) \tag{9}$$

Figure 1 shows how the gradient descent search methods works in a quadratic function.

2.2 Types of Models

The goal of model learning is to learn the behavior of the system given some observed quantities. Therefore missing information has to be predicted. Depending on what kind of information is missing, different types of models can be defined (see [1] section 2.1).

Forward Models predict a future state of the system $\mathbf{s_{k+1}}$ given the current state and action $\mathbf{s_k}$ and $\mathbf{a_k}$. Since the forward model directly describes the physical properties of the system, which represents a causal relationship between states and actions, learning a forward model is a well-defined problem.

Inverse Models in contrast are used to compute an action $\mathbf{a_k}$ which is needed to get from state $\mathbf{s_k}$ to $\mathbf{s_{k+1}}$. Since this mapping in general is not unique, learning an inverse model is not always a well-defined problem. But for a robot's inverse dynamics it is well-defined, thus the robot's inverse dynamics model can be learned using regression methods.

Forward and inverse models are the most important types of models. Although a combination of both models can be useful. The forward model can help to create a unique mapping for an inverse model learning problem. These models are called *Mixed Models.*
Furthermore in some applications it is important to know not just the next state of the system, but several states in the future. Models that provide this information are called *Multi-Step Predicton Models.*

2.3 Learning Architectures

Depending on the type of model and the learning problem different learning architectures can be defined. The learning architecture describes what quantities are observed to learn the model (see [1] section 2.2).

Using *Direct Modeling* a model is learnt by observing the system's inputs and outputs. The idea is to obtain the systems behavior and therefore the model by observing an input action and the resulting output state of the system. This learning architecture can basically be used to learn all types of models if the learning problem is well-defined.

An *Indirect Modeling* learning approach is feedback error learning, where a feedforward controller is being learned from a feedback controllers error signals. If the learned forward model is perfect, the feedback controller will not influence the control system any more, otherwise the error signal is used to update the model. Thus this learning architecture always tries to minimize the error. Although it can deal with ill-posed problems such as inverse kinematics learning. One drawback of this architecture is that it has to be applied on-line in order to get the actual feedback controller's error signals. Indirect Modeling can be used to learn inverse models and mixed models.

In the *Distal Teacher Approach* a unique forward model acts as a teacher to obtain an learned inverse models error and therefore help to update the inverse model. The goal is to minimize the error. The idea is that the inverse model results in a correct solution for a desired trajectory when the error between the output of the forward model and the input of the inverse model is minimized. The Distal Teacher Approach can only be applied for learning mixed models.

3 Locally Weighted Learning

Locally Weighted Learning (LWL) is one approach to learn models from training data. As described before the aim is to find the function $\mathbf{f}_\theta(\mathbf{x})$ in the model equation 5. The idea of LWL methods is to approximate this non-linear function by means of piecewise linear models. The main challenge is to find the region where the local model is valid (receptive field). Here, for each linear model the following receptive field is used:

$$w_k = \exp(-\frac{1}{2}(\mathbf{x}-\mathbf{c_k})^T\mathbf{D_k}(\mathbf{x}-\mathbf{c_k})) \tag{10}$$

The advantage of LWL methods is that we do not need a feature vector ϕ, which is usually hand-crafted.
The remainder of this section describes four different LWL methods.

3.1 Locally Weighted Regression

Locally Weighted Regression (LWR) extends the standard regression method as shown in equation 8 by a weight matrix which determines how much influence the data next to the query point has to the local linear model. This only requires the learning system to have sufficient training data in memory. Thus the model can easily be updated by adding new training data. An algorithm for a prediction $\hat{\mathbf{y}}_\mathbf{q}$ for a query point $\mathbf{x_q}$ is shown below (Algorithm 1). This algorithm seems quiet complex on a first view, but since the weigh-matrix $\mathbf{W}$ ensures that data-points which are not close to the query-point will be equal zero, the matrix multiplication and pseudo inverse simplifies a lot. Nevertheless the complexity rises with the dimensionality of the system.
The not yet defined variable in the above algorithm is the parameter $\mathbf{D}$ which is the distance matrix of the receptive field. Thus this parameter describes how big the region of validity around a query-point is. $\mathbf{D}$ can be optimized using Leave-One-Out Cross Validation (Algorithm 2), when sufficient training data is recorded. In purpose to reduce the number of parameters, $\mathbf{D}$ is assumed to be a global diagonal matrix multiplied by a scaling factor h. This scaling factor is now the only parameter to be optimized. Leave-One-Out Cross Validation predicts a value for a query-point which is left out of the training data and compares

Algorithm 1 Locally Weighted Regression

```
**Given:** $\mathbf{x_q}$ (Query Point), $\mathbf{p}$ (Training Points $\{x_i, y_i\}$)
**Compute weight-matrix W:**
$w_{ii} = \exp(-\frac{1}{2}(\mathbf{x_i} - \mathbf{x_q})^\mathbf{T}\mathbf{D}(\mathbf{x_i} - \mathbf{x_q}))$
**Build matrix X and vector y such that:**
$\mathbf{X} = (\tilde{\mathbf{x}}_\mathbf{1}, \tilde{\mathbf{x}}_\mathbf{2} \dots, \tilde{\mathbf{x}}_\mathbf{p})^\mathbf{T}$ where $\tilde{\mathbf{x}}_\mathbf{i} = [(\mathbf{x_i} - \mathbf{x_q})^\mathbf{T}\mathbf{1}]^\mathbf{T}$
$\mathbf{y} = (\mathbf{y_1}, \mathbf{y_2}, \dots, \mathbf{y_p})^\mathbf{T}$
**Compute locally linear model:**
$\beta = (\mathbf{X^T W X})^{-1}\mathbf{X^T W y}$
**Compute prediction $\mathbf{x_q}$:**
$\hat{\mathbf{y}}_\mathbf{q} = \beta_{\mathbf{n+1}}$
```

the prediction afterwards to the actual sample value, resulting in an error. This is repeated for all training points. The factor h is chosen to achieve a minimal error.

Algorithm 2 Leave-One-Out Cross Validation

```
**Given: a set H of reasonable values $\mathbf{h_r}$**
**for all** $h_r \in H$ **do**
    $sse_r = 0$
    **for** $i = 1 : p$ **do**
        $\mathbf{x_q} = \mathbf{x_i}$
        Temporarily exclude $\{x_i, y_i\}$ from training data
        Compute LWR prediction $\hat{\mathbf{y}}_\mathbf{q}$ with reduced data
        $sse_r = sse_r + (y_i - \hat{y}_q)^2$
    **end for**
**end for**
Choose optimal $h_r^*$ such that $h_r^* = min\{sse_r\}$
```

3.2 Locally Weighted Partial Least Squares

Since the complexity of the LWR Algorithm (Algorithm 1) rises with the input dimensionality LWR can be slow for higher dimensions. Furthermore the matrix inversion step can become numerically unstable if there are redundant input dimensions. Locally Weighted Partial Least Squares (LWPLS) takes care of these problems. In this approach Partial Least Squares (PLS) is used to reduce the complexity of the problem. PLS is based on a linear transition from the high dimensionality of the input to a new variable space based on lower dimensional orthogonal factors. This means, that those orthogonal factors are independent linear combinations of the original input. These projections are used to calculate a prediction for a query-point (see Algorithm 3).
The only undefined parameter is the number of projections r. Since the squared error for each new projection should be reduced, adding new projections can be stopped if the rate of error reduction $\frac{res_i^2}{res_{i-1}^2} < \phi$ is not high enough any more. In [4] $\phi = 0.5$ was used for all learning tasks.

Algorithm 3 Locally Weighted Partial Least Squares

```
Given: $\mathbf{x_q}$ (Query Point), $\mathbf{p}$ (Training Points $\{x_i, y_i\}$)
Compute weight-matrix $\mathbf{W}$:
$w_{ii} = \exp(-\frac{\mathbf{1}}{\mathbf{2}}(\mathbf{x_i} - \mathbf{x_q})^{\mathbf{T}}\mathbf{D}(\mathbf{x_i} - \mathbf{x_q}))$
Build matrix X and vector y such that:
$\bar{\mathbf{x}} = \sum_{\mathbf{i=1}}^{\mathbf{p}} w_{ii}\mathbf{x_i} / \sum_{i=1}^{p} w_{ii}$
$\beta_{\mathbf{0}} = \sum_{\mathbf{i=1}}^{\mathbf{p}} w_{ii} y_i / \sum_{i=1}^{p} w_{ii}$
$\bar{\mathbf{X}} = (\tilde{\mathbf{x}}_{\mathbf{1}}, \tilde{\mathbf{x}}_{\mathbf{2}}, \ldots, \tilde{\mathbf{x}}_{\mathbf{p}})^{\mathbf{T}}$ where $\tilde{\mathbf{x}}_{\mathbf{i}} = (\mathbf{x_i} - \bar{\mathbf{x}})$
$\mathbf{y} = (\tilde{y}_1, \tilde{y}_2, \ldots, \tilde{y}_p)^T$ where $\tilde{y}_i = (y_i - \beta_0)$
Compute locally linear model:
Initialize: $\mathbf{Z_0} = \mathbf{X}, \mathbf{res_0} = \mathbf{y}$
for $i = 1 : r$ do
   $\mathbf{u_i} = \mathbf{Z_{i-1}^T W res_{i-1}}$
   $\mathbf{s_i} = \mathbf{Z_{i-1} u_i}$
   $\beta_{\mathbf{i}} = \frac{\mathbf{s_i^T W res_{i-1}}}{\mathbf{s^T W s_i}}$
   $\mathbf{p_i} = \frac{\mathbf{s_i^T W Z_{i-1}}}{\mathbf{s_i^T W s_i}}$
   $\mathbf{res_i} = \mathbf{res_{i-1}} - \mathbf{s_i}\beta_{\mathbf{i}}$
   $\mathbf{Z_i} = \mathbf{Z_{i-1}} - \mathbf{s_i p_i}$
end for
Compute prediction $\mathbf{x_q}$:
Initialize: $\mathbf{z_0} = \mathbf{x_q} - \bar{\mathbf{x}}, \tilde{y}_q = \beta_0$
for $i = 1 : r$ do
   $s_i = \mathbf{z_{i-1}^T u_i}$
   $\tilde{y}_q \leftarrow \tilde{y}_q + s_i\beta_i$
   $\mathbf{z_i} = \mathbf{z_{i-1}} - s_i \mathbf{p_i^T}$
end for
```

3.3 Receptive Field Weighted Regression

When training data is received constantly by the learning system like in on-line learning scenarios, the data set becomes very large. In this case LWR and LWPLS fall short because of high computational cost. Instead of computing a local model when a prediction has to be made, Receptive Field Weighted Regression (RFWR) iteratively builds new local models when training data is added. Thus the prediction for a query-point can be computed as the weighted average over the predictions of all local models:

$$\hat{y}_q = \frac{\sum_{k=1}^{K} w_k \hat{y_{q,k}}}{\sum_{k=1}^{K} w_k} \tag{11}$$

Algorithm 4 shows how the local models are updated. Like in LWL and LWPLS the only

Algorithm 4 Receptive Field Weighted Regression

```
Given: a training point $(\mathbf{x}, y)$
Update K local models:
$w_k = \exp(-\frac{1}{2}(\mathbf{x} - \mathbf{c_k})^{\mathbf{T}}\mathbf{D_k}(\mathbf{x} - \mathbf{c_k}))$
$\beta_k^{n+1} = \beta_k^n + w_k \mathbf{P_k^{n+1}} \tilde{\mathbf{x}} \mathbf{e_{cv,k}}$
where $\tilde{\mathbf{x}} = [(\mathbf{x} - \mathbf{c_k})^{\mathbf{T}}\mathbf{1}]$ and $\mathbf{P_k^{n+1}} = \frac{1}{\lambda}(\mathbf{P_n^k} - \frac{\mathbf{P_k^n}\tilde{\mathbf{x}}\tilde{\mathbf{x}}^{\mathbf{T}}\mathbf{P_k^n}}{\frac{\lambda}{\mathbf{w_k}}} + \tilde{\mathbf{x}}^{\mathbf{T}}\mathbf{P_k^n}\tilde{\mathbf{x}})$ and $e_{cv,k} = (y - \beta_k^{n^T}\tilde{\mathbf{x}})$
Compute prediction $\mathbf{x_q}$:
$\hat{y}_k = \beta_{\mathbf{k}}^{\mathbf{T}}\tilde{\mathbf{x}}_{\mathbf{1}}$
```

open parameter is the distance matrix **D**. Since RFWR uses several local models, we can use a different distance matrix **D** for each of these models. In [3] the following cost-function for a gradient descent update of $\mathbf{D}$ was used:

$$\mathbf{J} = \frac{\mathbf{1}}{\sum_{\mathbf{i=1}}^{\mathbf{k}} \mathbf{w_i}} \sum_{\mathbf{i=1}}^{\mathbf{k}} \frac{\mathbf{w_i}||\mathbf{y_i} - \hat{\mathbf{y}}_\mathbf{i}||^\mathbf{2}}{(\mathbf{1} - \mathbf{w_i}\tilde{\mathbf{x}}_\mathbf{i}^\mathbf{T}\mathbf{P}\tilde{\mathbf{x}}_\mathbf{i})^\mathbf{2}} + \gamma \sum_{\mathbf{i,j=1}}^{\mathbf{n}} \mathbf{D}_\mathbf{ij}^\mathbf{2} \tag{12}$$

Note that RWFR is kind of an incremental version of LWR, thus it also can not deal with very high dimensional problems.

3.4 Locally Weighted Projection Regression

For higher dimensional problems LWPLS was used to reduce the dimensionality and therefore the complexity of the problem. Using LWPLS on-line with huge data-sets relates to the problem of LWR which was solved there using incremental model updates leading to RWFR. The idea of Locally Weighted Projection Regression is to formulate an incremental version of LWPLS which can deal with high dimensionality and huge data-sets. Algorithm 5 shows how one local model is updated.

Algorithm 5 Locally Weighted Projection Regression

```
Given: a training point $(\mathbf{x}, y)$
Update the means of inputs and outputs:
$\mathbf{x_0^{n+1}} = \frac{\lambda W^n \mathbf{x_0^n} + w\mathbf{x}}{W^{n+1}}$
$\beta_\mathbf{0}^\mathbf{n+1} = \frac{\lambda W^n \beta_\mathbf{0}^\mathbf{n} + wy}{W^{n+1}}$
where $W^{n+1} = \lambda W^n + w$
Update the local model:
Initialize $\mathbf{z_0} = \mathbf{x} - \mathbf{x_0^{n+1}}, \mathbf{res_0} = \mathbf{y} - \beta_\mathbf{0}^\mathbf{n+1}$
for $i = 1 : r$ do
  $\mathbf{u_i^{n+1}} = \lambda \mathbf{u_i^n} + \mathbf{w z_{i-1} res_{i-1}}$
  $s_i = \mathbf{z_{i-1}^T u_i^{n+1}}$
  $SS_i^{n+1} = \lambda SS_i^n + ws_i^2$
  $SR_i^{n+1} = \lambda SR_i^n + ws_i^2 res_{i-1}$
  $SZ_i^{n+1} = \lambda SZ_i^n + w\mathbf{z_{i-1}} s_i$
  $\beta_i^{n+1} = \frac{SR_i^{n+1}}{SS_i^{n+1}}$
  $p_i^{n+1} = \frac{SZ_i^{n+1}}{SS_i^{n+1}}$
  $\mathbf{z_i} = \mathbf{z_{i+1}} - s_i \mathbf{p_i^{n+1}}$
  $res_i = res_{i-1} - s_i \beta_i^{n+1}$
  $SSE_i^{n+1} = \lambda SSE_i^n + wres_i^2$
end for
```

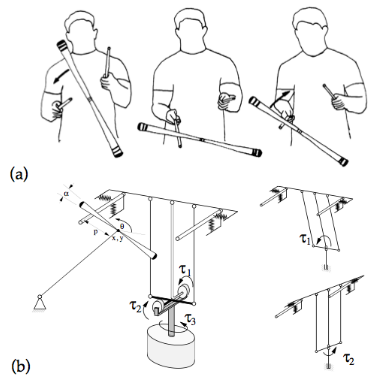
Like in RFWR the distance matrix $\mathbf{D}$ is updated using gradient descent for each local model (see [4]).

4 Model Learning Applications

In this section some very different model learning applications are shown. For every application information about the task that has to be learned, the type of model learning method used as well as the results are presented. The first three example applications use different LWL methods, while the last example uses Gaussian process regression (GPR) . This example intents to show that other model learning methods can be applied to specific problems but will not give an detailed look into GPR.

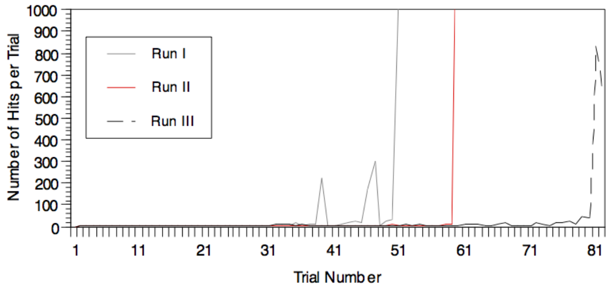
4.1 Learning Devil Sticking

Figure 2: Devil Sticking Task a) and Robot Layout b) (see [4])



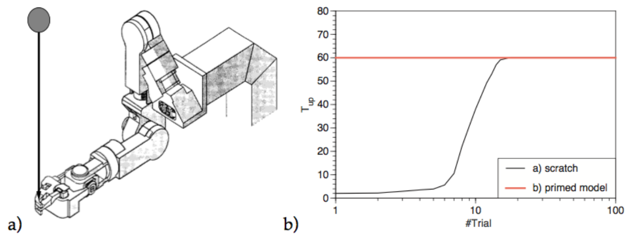
Task Description: Juggling a center stick between two control sticks. The center stick is lifted alternately by the two control sticks to joggle it. The task is to learn a continuous left-right-left pattern. This task is modeled as a discrete function that maps impact states from one hand to the other. A state is described as a vector $\mathbf{x} = (p, \theta, \dot{x}, \dot{y}, \dot{\theta})^T$ with impact position, angle, velocities of the center of the center stick and its angular velocity. The task command $\mathbf{u} = (x_h, y_h, \dot{\theta}, v_x, v_y)^T$ with a catch position, and angular trigger velocity and the two dimensional throw direction.
Type of Model Learning: The robot learns a forward model given the current state and action and predicting the next state. This problem has a 10 dimensional input and a five dimensional output and therefore is ideally suited for LWR methods. Furthermore training data is only generated with 1-2Hz, every time the center stick hits one of the control sticks.
Results: More than 1000 consecutive hits where counted as a successful trial, which was achieved in about 40-80 trials (see Figure 3). This is a remarkable result since even for humans devil sticking is a difficult task and a lot of trials are needed for an untrained human.

Figure 3: Devil Sticking Results (see [4])



4.2 Learning Pole Balancing

Figure 4: Pole Balancing a) and Results b) (see [4])

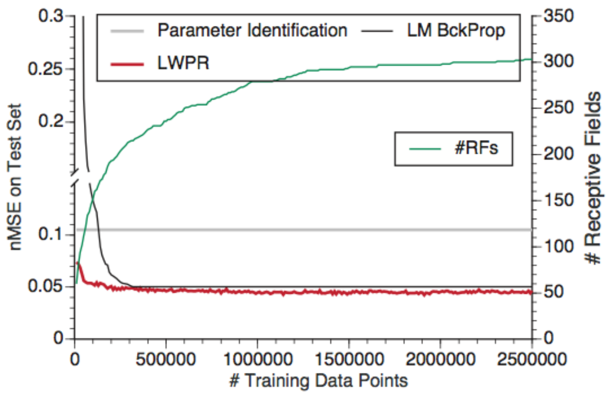


Task Description: In this application balancing a pole up-right on a robots finger is the learning task (see Figure 4a). The robot arm has 7 degree-of-freedom. Given the inverse dynamics model of the robot the goal of the learning problem was to learn task level commands like Cartesian accelerations of the robot's finger.
Type of Model Learning: On-line learning using RFWR was used with input data from a stereo camera system observing the pole. Thus input data is 12 dimensional: 3 positions of the lower pole end, 2 angular positions, the 5 corresponding velocities and two horizontal accelerations of the robot's finger. The output that predicts the next state of the pole therefore is 10 dimensional. So RFWR was used to learn a forward model of the pole to predict the next state.
Results: While learning the model from scratch took the system about 10-20 trials to keep the pole up-right for longer then a minute, observing a human teacher who demonstrated pole balancing for 30 seconds helped to extract the forward model so that the system fulfilled the task on a single trial (see Figure 4b). This could also be shown using different poles and demonstrations from different people.

4.3 Inverse Dynamics Learning

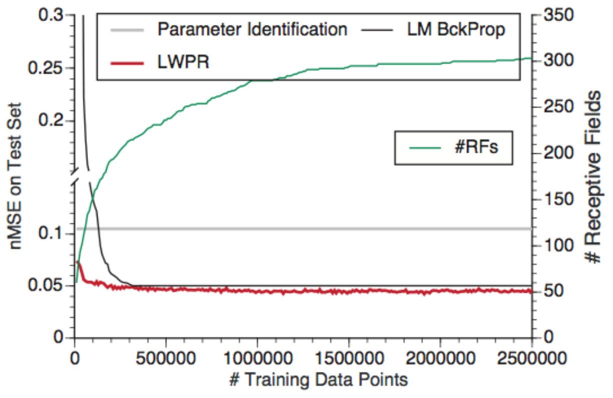
Task Description: Computing a inverse dynamics model 4 by hand from rigid-body dynamics does not lead to a satisfying solution since a lot of the systems properties can not be modeled accurately. Learning the inverse dynamics model of a 7-degree-of-freedom robot arm was accomplished in this example application (=21 input dimensions). In a second example the authors learned the inverse dynamics of a humanoid robots shoulder motor with 30 degrees-of-freedom (=90 input dimensions).
Type of Model Learning: In both inverse dynamics learning problems LWPR was applied. This fits the need of handling high dimensionality input and coping with big data sets.
Results: For the 7-degree-of-freedom robot arm Figure 5 shows the results compared to a parametric inverse dynamics model. The normalized Mean-Squared-Error (=nMSE) converges after about 500.000 training points to $nMSE = 0.042$. During learning LWPR created about 300 local models.

Figure 5: Inverse dynamics learning results with 7DOF Robot (see [4])



The results of the second example are shown in Figure 6. Here a lot more training points where needed, but LWPR outperformed the parametric model quickly. On this high dimensional learning problem LWPR created more than 2000 local models.

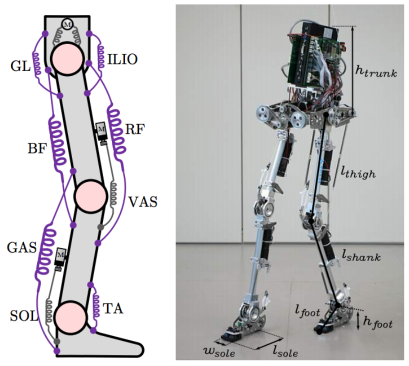
Figure 6: Inverse dynamics learning results with 30DOF Robot (see [4])



4.4 Bio-Inspired Motion Control Based on a Learned Inverse Dynamics Model

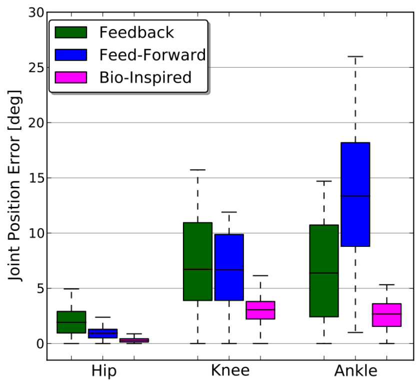
Task Description: The previous example shows that learning an inverse dynamics model leads to better results in comparison to manually computed models. This was shown on a rigid body robot with stiff joints. The problem of accurately modeling all system properties gets even worse when elastic (bio-inspired) joints are used. In this example a different model learning approach was used to learn an inverse dynamics model of the bio-inspired robot BioBiped1 (see Figure 7).
Type of Model Learning: The authors decided to use Gaussian process regression (GPR) to learn the inverse model off-line using recorded training data. As shown in [2] GPR is suited well for off-line model learning.

Figure 7: Bio-Inspired robot BioBiped1 (see [5])



Results: Using the inverse dynamics model a model-based feed-forward controller was implemented and compared to a standard PD-controller. Furthermore a combination of the feed-forward controller and the feed-back controller was implemented. All three approaches were evaluated in an experiment, where the robot is standing on the ground with both feet and performs a periodic up and down swinging motion using both legs. Figure 8 shows the joint error of all three approaches.

Figure 8: Bio-Inspired Motion Control Results (see [5])



5 Conclusion

Model learning can help to find properties of a system which is in this case related to a robot. In 1 the model learning problem was discussed, describing different types of models and a regression solution was shown by minimizing a cost function. Different types of models where categorized in 2 in more detail together with learning architectures which can be utilized to learn a specific model. Based on the regression solution to the model learning problem locally weighted learning (LWL) methods where discussed in section 3. Beginning with a very straightforward approach (see 1), which than was modified in 3 to reduce the systems dimensionality and in 4 and 5 to deal with big data-sets.
The first three examples show that applying LWL methods to different learning scenarios is successful for learning forward models, as well as inverse dynamics models. The last application example (see 4.4) shows that also other learning methods (here GPR) can be applied successful to a learning problem.
However, the challenge in model learning remains to deal with high dimensionality, complex algorithms and big data sets. On-line learning is preferred for self-improved while performing a certain task but needs very efficient algorithms (e.g. GPR can not applied on-line).

References

[1] Nguyen-Tuong, D., and Peters, J. Model learning in robotics: a survey. *Cognitive Processing*, 12(4) (2011), 319–340. Intelligent Autonomous Systems.

[2] Nguyen-Tuong, D., Peters, J., Seeger, M., and Schölkopf, B. Learning inverse dynamics: a comparison. In *Proceedings of the European Symposium on Artificial Neural Networks (ESANN)* (2008). Intelligent Autonomous Systems.

[3] Schaal, S., and Atkeson, C. G. Constructive incremental learning from only local information. *Neural Computation 10* (1997), 2047–2084.

[4] Schaal, S., Atkeson, C. G., and Vijayakumar, S. Scalable techniques from nonparametric statistics for real time robot learning. *Applied Intelligence 17* (2002), 49–60. 10.1023/A:1015727715131.

[5] Scholz, D., Kurowski, S., Radkhah, K., and von Stryk, O. Bio-inspired motion control of the musculoskeletal biobiped1 robot based on a learned inverse dynamics model. In *Proc. 11th IEEE-RAS Intl. Conf. on Humanoid Robots* (Bled, Slovenia, Oct. 26-28 2011).