Locally Weighted Learning

Peter Englert
Department of Computer Science
TU Darmstadt
englert.peter@gmx.de

Abstract

Locally Weighted Learning is a class of function approximation techniques, where a prediction is done by using an approximated local model around the current point of interest. This paper gives an general overview on the topic and shows two different solution algorithms. Finally some successful applications of LWL in the field of Robot Learning are presented.

1 Introduction

The goal of function approximation and regression is to find the underlying relationship between input and output. In a supervised learning problem training data, where each input is associated to one output, is used to create a model that predicts values which come close to the true function. One basic approach for solving this problem is to build a global model out of labeled training data. Examples for global methods are Support Vector Machine, Neural Networks and Gaussian Process Regression. All of these methods have in common that they use the complete training data for creating a global function. This approximated function is used to predict values that come close to the corresponding true value of the original function. A disadvantage of global methods is that sometimes no parameter values can provide a sufficient good approximation. Furthermore, the computational costs are for some tasks very high, i.e., if the task needs many predictions in short time or the model is extended incrementally. An alternative to global function approximation is Locally Weighted Learning (LWL) [2]. LWL methods are non-parametric and the current prediction is done by local functions which are using only a subset of the data. The basic idea behind LWL is that instead of building a global model for the whole function space, for each point of interest a local model is created based on neighboring data of the query point (cf. figure 1). For this purpose each data point becomes a weighting factor which expresses the influence of the data point for the prediction. In general, data points which are in the close neighborhood to the current query point are receiving a higher weight than data points which are far away. LWL is also called lazy learning, because the processing of the training data is shifted until a query point needs to be answered. This approach makes LWL a very accurate function approximation method where it is easy to add new training points.

In this paper, I will describe the general LWL solution method that can solve the function approximation problem accurately. Two different widely used algorithms are described in more detail. Afterwards, two applications in the field of robot learning are presented where LWL algorithms have been successfully applied.

2 Locally Weighted Learning

In the following paper a standard regression model like

$$y = f(x) + \epsilon \tag{1}$$

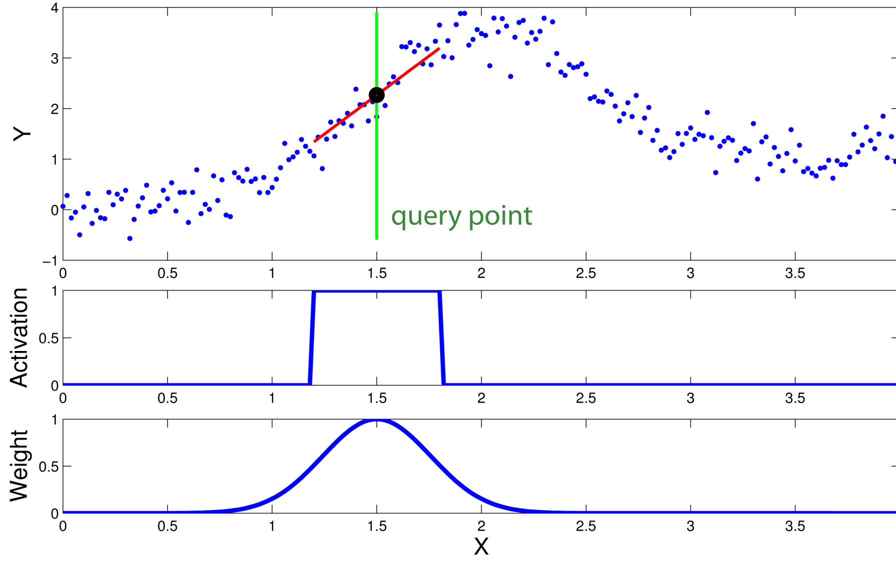

Figure 1: **Example of Locally Weighted Learning**, containing in the upper graphic the set of data points (x,y) (blue dots), query point (green line), local linear model (red line) and prediction (black dot). The graphic in the middle shows the activation area of the model. The corresponding weighting kernel (receptive field) is shown in the bottom graphic.

is assumed with a continuous function f(x) and noise ϵ. The basic cost function of LWL is defined as

$$J = \frac{1}{2}\sum_{i=1}^{n} w_i(x_q)(y_i - x_i\beta_q)^2 \tag{2}$$

with the components:

- **Labelled training data** $\mathcal{D} = \{(x_i, y_i)|i = 1, 2, ..., n\}$ where each data point x_i belongs to a corresponding output value y_i.
- **Point of interest** x_q (also called query point), which is the position where we want a prediction $\hat{y}_q$.
- **Weights** w_i describe the relevance of the corresponding training set (x_i, y_i) for the current prediction. They are dependent on the query point and are computed by a weighting function.
- **Regression coefficient** β_q of our linear model, which we want to obtain for doing the prediction.

The goal is to find a β_q that minimizes equation (2) for the current query point x_q. An important difference to global least square methods is that β_q is dependent of the current query point. One of the most important part of LWL is the way how the weights w_i are computed. The computation of a weight can be separated in to two steps [2]:

I **Distance function** $d(x_i, x_q)$: Measures the relevance of training points for the current prediction. The distance function needs two input objects and returns a number (i.e. euclidean distance $d = \sqrt{(x-q)D(x-q)}$ with distance metric D). The distance metric D is a very important parameter that describes the size and shape of the receptive field.

II **Weighting function** (Kernel function) $K(d)$: Computes for each distance value a corresponding weight w_i (i.e. $K(d) = \exp(-d^2)$). The smoothness of the used kernel will influence the smoothness of the output function. Some kernel functions will converge completely to zero. This property can can be used for decreasing the computational costs by ignoring all points with zero weight.

One advantage of LWL is the possibility to switch very easily between different weighting functions. There are two major categories in which you can split up LWL algorithms. The first category are memory-based LWL methods where all training data is kept in memory. The second category are purely incremental LWL methods that do not need to remember any data explicitly. One method of each category is described in the following sections.

2.1 Memory-Based Locally Weighted Regression

Locally Weighted Regression (LWR) is the classic approach to solve the function approximation problem locally [2]. It is also called Memory-Based Learning, because all training data is kept in memory to calculate the prediction. The single steps of LWR are outlined in algorithm 1 [4]. This algorithm has a complexity of $O(n^2)$ where n are the number of training points. The column of

---

Algorithm 1 Memory-Based Locally Weighted Regression

Given:

- query point x_q
- n training points $\{x_i, y_i\}$

Prediction:

- Build matrix $X = (\hat{x}_1, \hat{x}_2, ..., \hat{x}_n)^T$ where $\hat{x}_i = [x_i^T \quad 1]^T$
- Build vector $y = (y_1, y_2, ..., y_n)^T$
- Compute diagonal weight matrix W:

$$w_{i,i} = \exp\left(-\frac{1}{2}(x_i - x_q)^T D(x_i - x_q)\right)$$

- Calculate Regression coefficient:

$$\beta_q = (X^T W X)^{-1} X^T W y \tag{3}$$

- Predict

$$\hat{y}_q = [x_q^T \quad 1]\beta_q$$

---

ones is added to the Matrix X due to the offset parameter of the linear regression. The regression coefficient (3) is obtained in a closed form solution by setting the first derivative of equation (2) to zero. Algorithm 1 has to be executed once for each query point and the local model is discarded after each prediction. The only open parameter remaining is the distance metric D, which describes the size and shape of the receptive field. It is usually chosen as a diagonal matrix. To reduce the number of open parameters, D can be constructed as $D = hdiag([n_1, n_2, ..., n_n])$ with a scaling parameter h. The n_i are normalizing the range of the corresponding input space. Then there is only one open parameter h to estimate. This can be done by leave-one-out cross-validation, as described in algorithm 2 [4]. Figure 1 shows an example of one prediction with LWR. The advantages of LWR are the high accuracy due to the local model and the few open parameters that have to be obtained. Extensions of LWR can include a ridge regression formulation to increase stability, an outlier removal technique for higher accuracy and the arrangement of the training data in k-d trees for lower computational costs [2].

2.2 Locally Weighted Projection Regression

Locally Weighted Projection Regression (LWPR) is a purely incremental LWL method [6][8]. It was developed to solve two major problems that exists with memory-based methods like LWR. One of them are the cost intensive computations of LWR with high dimensional data that increase quadratically. This makes the algorithm unusable for tasks that need many predictions in small time steps. Another problem of LWR is that the matrix inversion in algorithm 1 for obtaining the regression coefficient cannot handle redundant input dimensions and can become singular.

Instead of throwing away the model after each prediction like in LWR, LWPR is keeping each model for further predictions in memory. It uses multiple locally weighted linear models which are combined for approximating non-linear functions (cf. fig 2). Adding new data points requires only an update to the existing models or the creation of a new model if there is no trustworthy model

Algorithm 2 Leave-one-out cross-validation

Initialize:

- Define a set of m reasonable values for h: $H = (h_1, h_2, ..., h_m)$

Algorithm:

- **for** $k = 1 : m$ **do**
  $h_{act} = h_k$
  $se_k = 0$ {# squared error of current candidate}
  for $i = 1 : n$ **do**
    set query point $x_q = x_i$
    remove training point (x_i, y_i) temporally from training data
    compute LWR [alg. 1] with h_{act} on the remaining training data to obtain $\hat{y}_q$
    $se_k = se_k + (y_i - \tilde{y}_q)^2$
  end for
- **end for**
- choose the scaling factor h^* with the associated minimal squared error

available. This makes it unnecessary to save large training data in the memory, because the models are updated incrementally. Furthermore, LWPR is using an online version of the dimensionality reduction method Partial Least Squares (PLS) to handle redundant and irrelevant input data. The goal of PLS is to reduce the dimensionality locally to find optimal local projections and eliminate subspaces of the input space that minimally correlate with the output. This ensures that redundant and irrelevant input dimensions are ignored.

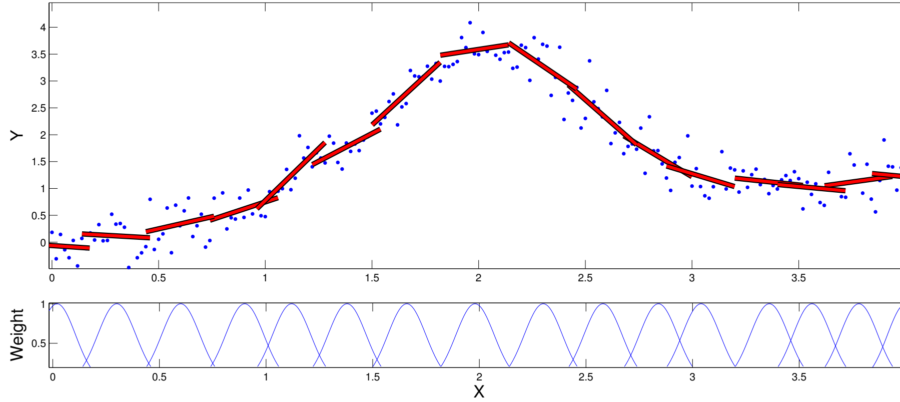

Figure 2: **Example for Locally Weighted Projection Regression (LWPR)**, containing in the upper graphic the data points (x,y) (blue dots) and the local linear models (red lines). The bottom graphic shows the Receptive Fields which show the activation weight of the corresponding model.

Each local linear model is described by

$$y_k = \beta_k s \tag{4}$$

where s is the lower dimensional projected input (also called latent variable) from x and β_k is the regression coefficient. The latent variable s is computed by mapping the input x along a projection directions u

$$s = u^T x \tag{5}$$

The prediction is done by combining multiple locally weighted linear models

$$\hat{y} = \frac{\sum_{k=1}^{K} w_k \hat{y}_k}{\sum_{k=1}^{K} w_k} \tag{6}$$

and the activation weight w_k of each model is computed using a Gaussian Kernel

$$w_k = exp\left(-\frac{1}{2}(x - c_k)^T D_k (x - c_k)\right) \tag{7}$$

The weight w_k describes the area of validity where the model is active, also known as receptive field (RF). The fixed center of the model is c_k and the size and shape is described by the distance metric D_k. The models are updated with each training example incrementally. Algorithm 3 describes the update for one local model, the complete LWPR is described in algorithm 4 and a prediction is done by executing algorithm 5 [7]. The value w_{gen} is a minimum activation threshold for a receptive

---

Algorithm 3 LWPR Model Update (for **one** local model)

Input:

- training point (x, y)

Update mean:

- $x_0^{n+1} = \frac{\lambda W^n x_0^n + wx}{W^{n+1}}$ $\qquad \beta_0^{n+1} = \frac{\lambda W^n \beta_0^n + wy}{W^{n+1}}$

  where $W^{n+1} = \lambda W^n + w \qquad x_0^0 = 0 \qquad u_r^0 = 0 \qquad \beta_0^0 = 0 \qquad W^0 = 0$

Update model:

- $z_0 = x - x_0^{n+1} \qquad res_0 = y - \beta_0^{n+1}$
- **for** $r = 1 : R$ **do**
  $u_r^{n+1} = \lambda u_r^n + w z_{r-1} res_{r-1}$ {# determine projection}
  $s_r = s_{r-1}^T u_r^{n+1}$ {# project input data}
  $SS_r^{n+1} = \lambda SS_r^n + w s_r^2$ {# memory term}
  $SR_r^{n+1} = \lambda SR_r^n + w s_r^2 res_{r-1}$ {# memory term}
  $SZ_r^{n+1} = \lambda SZ_r^n + w z_{r-1} s_r$ {# memory term}
  $\beta_r^{n+1} = SR_r^{n+1} / SS_r^{n+1}$ {# univariate regression}
  $p_r^{n+1} = SZ_r^{n+1} / SS_r^{n+1}$ {# regress input data against current projection}
  $z_r = z_{r-1} - s_r p_r^{n+1}$ {# reduce input space}
  $res_r = res_{r-1} - s_r \beta_r^{n+1}$
  $MSE_r^{n+1} = \lambda MSE_r^n + w res_i^2$ {# Mean Squared Error}
- **end for**

Algorithm 4 LWPR

Initialize LWPR with no Model
for each new training example (x,y) **do**
  for all Local Models **do**
    compute activation weight w from eq.7
    update model according to algorithm 3
    check if number of projections R needs to be increased
  end for
  if no local model was activated above w_{gen} **then**
    create new Receptive Field with $R = 2, c = x$
  end if
end for

Algorithm 5 LWPR Prediction

Input:

- query point x_q

Initialize:

- $x_q = x_q - x_0 \qquad \hat{y} = \beta_0$

Prediction:

- **for** $r = 1 : R$ **do**
  $s_r = u_r^T x_q$ {# compute latent variable}
  $\hat{y}_q = \hat{y}_q + s_r \beta_r$ {# update prediction}
  $x_q = x_q - s_r p_r^n$ {# reduce input space}
- **end for**

field to estimate if it is trustworthy for the current prediction. λ is a forgetting rate which weights training points earlier recorded down. This is useful for systems that change over time. R defines the number of projections that is used by PLS and is initialized with 2. The algorithm uses the mean squared error (MSE) to determine if the number of projections R has to be increased. If the factor $\frac{MSE_{r+1}}{MSE_r}$ is below a threshold, between 0 and 1, adding new projections is stopped. One of the major problems in LWL is to determine the region of validity D in which a local model can be trusted. In LWPR it is possible to optimize D_k for every local model individually with a gradient descent update of D based on stochastic leave-one-out cross-validation [3]. The complete LWPR algorithm reaches a complexity of $O(n)$.

LWPR is a method that is well suited for tasks with high-dimensional data, redundant input dimensions and continuous data streams. The biggest strength of LWPR is the combination of the high accuracy of the prediction and the low computational costs through the model structure and the dimensionality reduction with PLS. Another advantage is the adaption over time, which is useful when the system changes over time.

3 Applications

Locally Weighted Learning has been used in a wide range of application areas. In the following section two applications in the field of robot learning are described. LWL is well suited for robot tasks because it can handle continuous input streams very well and it approximates the input space with local models only on the positions that are relevant for the current task.

3.1 Billiards

One application where Locally Weighted Regression (sec. 2.1) has been applied to, is a billiard playing robot scenario [1]. Figure 3 shows the basic setup consisting of a small billiard table and a robot with a cue. The robot has a spring actuated cue and one rotatory joint that allows him to swivel around the table. They used two cameras as sensors, one on the ceiling looking down to the table and another one is positioned on the robot looking along the cue direction. Further parts of the setup are two billiard balls. The cue ball gets hit by the cue and the object ball has to be sank into one of the pockets. One restriction was that the cue ball has to be for each shot on the same initial position.

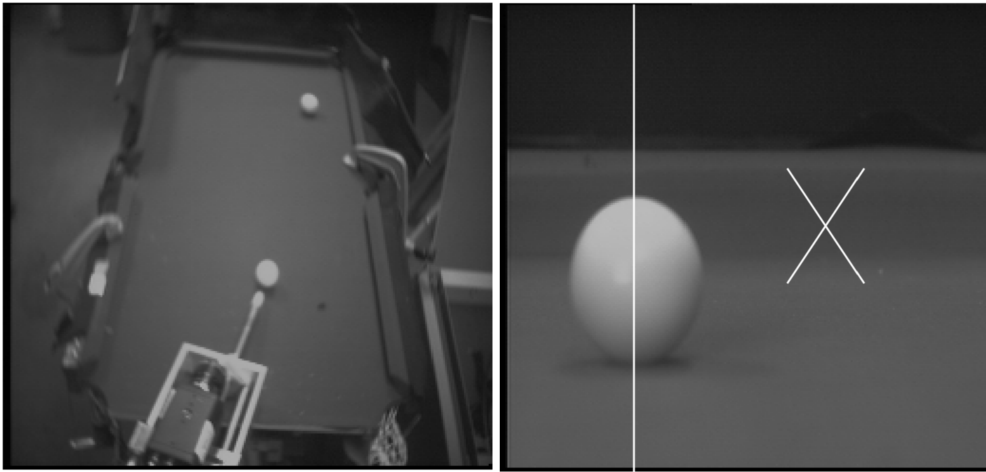

Figure 3: **Setup of the billiard task** (taken from [1]), containing the billiard table, the robot and two balls (left) and the aiming procedure of the camera mounted on the robot (right). The cross have to overlap with the line to reach the desired action.

As training data for LWR they have used the input state $x = (x^{above}_{object}, y^{above}_{object})$ which is the position of the object ball tracked by the camera from above. The action u is the starting position x^{cue}_{object} of

the cue at the beginning of the shot and is described by the view of the robots camera (see fig. 3 (right)). The output is the position b where the object ball hits the cushion first (cf. fig. 4) which is tracked by the top camera. This implies an update of the training data in memory with

$$(x_{object}^{above}, y_{object}^{above}, x_{object}^{cue}) \rightarrow b$$

where 3 input and 1 output parameter are used.

The standard procedure for performing one shot is:

1. Put the object ball to a random position on the one half of the table. Set the cue ball to its fixed start position.
2. Create an action u out of the position of the tracked object ball x with a combined inverse and forward model. The inverse model is used to receive a good initial starting point for the following search with steepest descent over the forward model.
3. Perform the shot with the selected u and track the trajectory of the ball with the camera.
4. Update the model with one training pair.

They have used locally weighted regression for both models with an outlier removal technique. The kernel width was determined with cross validation. Duration of the control choice was around 0.8 seconds on a Sun-4. Figure 4 (right) shows the learning rate of this task. It can be recognized that

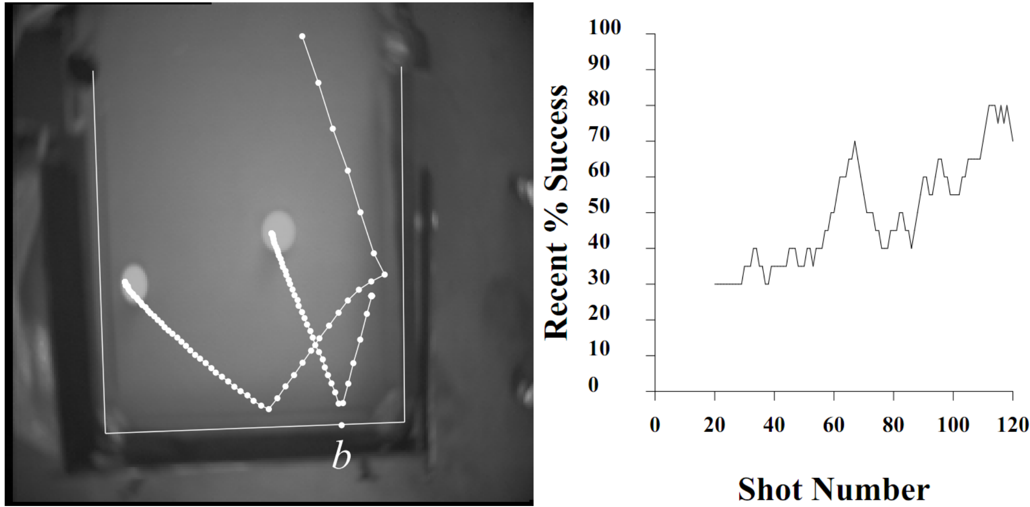

Figure 4: Trajectory of the balls (taken from [1]) recorded from the upper camera (left) and the success rate (right).

the highest success rate 80% was received after 115 experiences. This rate can be interpreted as a very good result for such an easy setup in a high precision task with only one output parameter. Causes for unsuccessful shots have been identified in outliers and visual tracking errors. Due to the random position of the object ball, there also have been some shots that were extremely difficult to perform.

3.2 Inverse Dynamics Learning

Another successful application of LWL is the online learning of an robots inverse dynamics model [3] [4] [7]. An Inverse dynamics model describes the mapping of the joint positions, velocities and accelerations to corresponding motor torque commands. LWPR (sec. 2.2) is well suited for learning the inverse dynamics, because it can handle high dimensions and it can predict values in real time which is necessary for the fast execution of commands. One example can be found in [7] where LWPR is used to learn the inverse dynamics of a 7 degree of freedom anthropomorphic robot arm. The learning was done with 21 input dimensions (position, velocity, acceleration) and 7 output

dimensions (torque). For each joint one LWPR system was used with a special neighbor structure of the local models to speed up the algorithm. The robot task was to perform a desired figure-8 pattern X_{des} in front of his body (fig. 5) with its end effector. X_{sim} is the trajectory performed in a simulation with a perfect inverse dynamics model, X_{param} is the performance obtained with a rigid body dynamics model and X_{lwpr} shows the results of LWPR after learning. Figure 5 (right) shows the development of the result after 10, 20, 30 and 60 seconds. The learning process started without any inverse dynamics model and converges after 60 seconds of learning very close to the desired pattern. This example shows the high accurate predictions and the fast learning rate of LWPR.

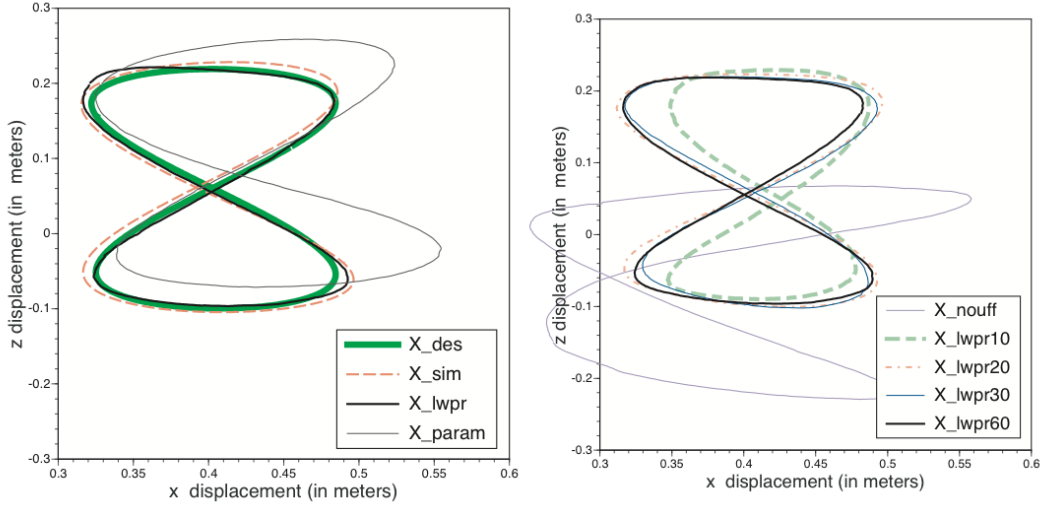

Figure 5: Result of LWPR (taken from [7]) in comparison to other methods (left) and the learning speed of LWPR (right) after 10, 20, 30 and 60 seconds.

4 Conclusion

This paper gave an general overview on algorithms and applications of Locally Weighted Learning. The classic method Locally Weighted Regression was introduced which is well suited for tasks that need very accurate predictions. Restrictions for this method are high dimensional and redundant input spaces. Incremental methods like Locally Weighted Projection Regression solve those restrictions. Through dimensionality reduction techniques and an incremental model update it is possible to use LWPR in real-time tasks with high dimensions. Nevertheless, some parameters still have to be selected manually. Current research focuses on a full Bayesian treatment of Locally Weighted Regression to avoid cross validation and manual parameter tuning [5]. Finally it can be said that Locally Weighted Learning provides some powerful methods that are well suited for many different tasks and the results are comparable to current state of the art global function approximation methods.

References

[1] C.G. Atkeson, A.W. Moore, and S. Schaal. Locally weighted learning for control. *Artificial Intelligence Review*, 11(1):75–113.

[2] C.G. Atkeson, A.W. Moore, and S. Schaal. Locally weighted learning. *Artificial intelligence review*, 11(1):11–73, 1997.

[3] S. Schaal, C.G. Atkeson, and S. Vijayakumar. Real-time robot learning with locally weighted statistical learning. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 1, pages 288–293. IEEE, 2000.

[4] S. Schaal, C.G. Atkeson, and S. Vijayakumar. Scalable techniques from nonparametric statistics for real time robot learning. *Applied Intelligence*, 17(1):49–60, 2002.

[5] J.-A. Ting, S. Vijayakumar, and S. Schaal. *Locally weighted regression for control*, pages 613–624. Springer, 2010.

[6] S. Vijayakumar, A. D'souza, and S. Schaal. Incremental online learning in high dimensions. *Neural Computation*, 17(12):2602–2634, 2005.

[7] S. Vijayakumar, A. D'souza, T. Shibata, J. Conradt, and S. Schaal. Statistical learning for humanoid robots. *Autonomous Robots*, 12(1):55–69, 2002.

[8] S. Vijayakumar and S. Schaal. Locally weighted projection regression: An o(n) algorithm for incremental real time learning in high dimensional space. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000)*, volume 1, pages 288–293. Citeseer, 2000.