Journal of Machine Learning Research 9 (2008) 623-626 Submitted 10/07; Revised 3/08; Published 4/08

A Library for Locally Weighted Projection Regression

Stefan Klanke S.KLANKE@ED.AC.UK
Sethu Vijayakumar SETHU.VIJAYAKUMAR@ED.AC.UK
School of Informatics
University of Edinburgh
Edinburgh, EH9 3JZ, UK

Stefan Schaal SSCHAAL@USC.EDU
Dept. of Computer Science
University of Southern California
Los Angeles, CA 90089-2520, USA

Editor: Soeren Sonnenburg

Abstract

In this paper we introduce an improved implementation of locally weighted projection regression (LWPR), a supervised learning algorithm that is capable of handling high-dimensional input data. As the key features, our code supports multi-threading, is available for multiple platforms, and provides wrappers for several programming languages.

Keywords: regression, local learning, online learning, C, C++, Matlab, Octave, Python

1. Introduction

Locally weighted projection regression (LWPR) is an algorithm that achieves nonlinear function approximation in high dimensional spaces even in the presence of redundant and irrelevant input dimensions (Vijayakumar et al., 2002). At its core, it uses locally linear models, spanned by a small number of univariate regressions in selected directions in input space. This nonparametric local learning system learns rapidly with second order learning methods based on incremental training, using statistically sound stochastic cross validation.

The implementation of LWPR we present in this work is written in low-level C, requires no additional libraries, and comes with convenient wrappers for C++, Matlab and Python. Together with documentation, tutorials and additional supporting materials, it is freely available for download from `http://www.ipab.inf.ed.ac.uk/slmc/software/lwpr`.

2. The LWPR Algorithm

The goal of LWPR is to learn a regression function from training data that incrementally arrive as input-output tuples $(\mathbf{x}_i, y_i)$, where we assume univariate output data for now. The LWPR regression function is constructed by blending local linear models $\psi_k(\mathbf{x})$ in the form

$$f(\mathbf{x}) = \frac{1}{W(\mathbf{x})} \sum_{k=1}^{K} w_k(\mathbf{x}) \psi_k(\mathbf{x}), \quad W(\mathbf{x}) = \sum_{k=1}^{K} w_k(\mathbf{x}). \tag{1}$$

Here, $w_k(\mathbf{x})$ is a locality kernel that defines the area of validity of the local models (also termed "receptive field"), which is usually modeled by a Gaussian

$$w_k(\mathbf{x}) = \exp\left(-\frac{1}{2}(\mathbf{x}-\mathbf{c}_k)^T\mathbf{D}_k(\mathbf{x}-\mathbf{c}_k)\right), \tag{2}$$

where $\mathbf{c}_k$ is the centre of the k^{th} linear model and $\mathbf{D}_k$ is its distance metric. During training, all updates to the local models are weighted by their activation $w_k(\mathbf{x})$, facilitating fully localised and inpedendent learning. If no existing local model yields an activation above a certain threshold (e.g., $w_{gen} = 0.1$), a new local model is created with its center $\mathbf{c}_k$ set to the input datum. Thus, the number K of local models is adapted automatically.

For learning the linear models $\psi_k(\mathbf{x})$ themselves, LWPR employs an online formulation of weighted partial least squares (PLS) regression. In particular, within each local model[1] the input data $\mathbf{x}$ is projected along selected directions $\mathbf{u}_i$, yielding "latent" variables s_i with

$$s_i = \mathbf{u}_i^T\mathbf{x}_{i-1}, \qquad \mathbf{x}_i = \mathbf{x}_{i-1} - s_i\mathbf{p}_i = \mathbf{x}_{i-1} - \mathbf{p}_i\mathbf{u}_i^T\mathbf{x}_{i-1}, \qquad \mathbf{x}_0 = \mathbf{x} - \bar{\mathbf{x}},$$

where the vectors $\mathbf{p}_i$ ensure orthogonality of the projections, and $\bar{\mathbf{x}}$ is the weighted mean of the input data (as seen through the receptive field). The output of the local model is then formed by a linear combination of the latent variables (also called PLS factor loadings)

$$\psi_k(\mathbf{x}) = \beta_0 + \sum_{i=1}^{R}\beta_i s_i. \tag{3}$$

The number R of regression directions is automatically adapted to the local dimensionality of the training data, and the parameters $\mathbf{u}_i$, $\mathbf{p}_i$, and β_i can be robustly estimated from accumulating certain statistics of the training set (for details please see Vijayakumar et al., 2005). In a similar fashion, the distance metrics $\mathbf{D}$ can be adapted using stochastic cross-validation, such that the input space is covered by wide receptive fields in regions of low curvature, and narrow receptive fields where the curvature is high. The initial distance metric assigned to a newly created receptive field is a rather critical open parameter. If available, one should use an estimate of the Hessian of the function that is to be approximated. As a valuable feature of the algorithm, LWPR can optionally yield confidence bounds for its predictions (Vijayakumar et al., 2005).

There are two possible choices with regard to handling multivariate output data: First, the local models itself could be made multivariate, in which case only one layer of receptive fields is needed. Alternately, one can learn all output dimensions independently, effectively using univariate PLS regression (see, e.g., Garthwaite, 1994) within the local models. In the present implementation we use the latter approach, which is computationally more costly for many output dimensions, but usually exhibits superior prediction performance.

LWPR is an algorithm that is particularly suited (and recommended) for regression problems with a sufficiently large number of training examples. For use in small data set scenarios, the data should be presented to the algorithm multiple times in random order.

3. Details of the Implementation

This section describes several important aspects of our implementation, all of which are related to execution speed.

1. For notational convenience we drop the index k of the local model.

3.1 Data Structures and Memory Allocation

On modern computers, the speed at which many algorithms run does not only depend on the processor, but also critically on the speed of memory access. We designed our library so that memory access is as continuous as possible, thus minimising the chance of cache misses. In our library, all variables[2] of each local model are allocated together in a contiguous piece of memory. Moreover, we allocate "workspaces" as part of the LWPR model for storing intermediate results, such that no further allocations are needed during the computations.

3.2 Multithreaded Updates and Predictions

Since the LWPR algorithm is designed to be parallelisable (the local models learn independently), and nowadays even off-the-shelf mainstream computers are equipped with multi-core processors, we constructed our LWPR library such that it is capable of running the computations in multiple threads. The library currently supports POSIX threads on Linux/Unix machines and native threads on the Windows platform. The desired number of threads has to be defined at compile time, and threading can be deactivated altogether.

In order to balance the overhead of creating threads with the expected reduction in computation time, we chose to implement two complimentary strategies for distributing the work. *Predictions* of the LWPR model are split up on a per-output-dimension level, which implies that LWPR models for one-dimensional output data will not be accelerated by using multiple threads. The more costly *update* operations, however, are distributed across multiple threads on the receptive field level, so even single-output models may benefit (see Fig. 1).

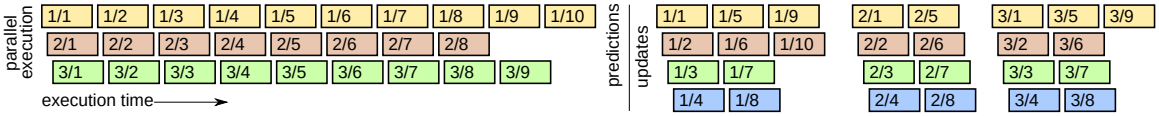

Figure 1: Illustration of our threading implementation for the case of (up to) 4 threads. The example LWPR model has 3 output dimensions with 10, 8, and 9 receptive fields, respectively. A label M/N denotes the N-th receptive field in the M-th output sub-model. For predictions, each thread handles a different output dimension. For updates, the workload of each output dimension is split up among threads, and outputs are handled one after another.

3.3 Fast Computation of Predictions and Their Gradients

For some applications of LWPR, it may be useful to compute analytic derivatives of the model, for example, to retrieve the Jacobian from a learned forward kinematics relation. The gradient of a single predicted output (1) is given by

$$\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} = \frac{1}{W}\sum_k \left(\frac{\partial w_k}{\partial \mathbf{x}}\psi_k + w_k \frac{\partial \psi_k}{\partial \mathbf{x}} \right) - \frac{1}{W^2}\sum_k w_k \psi_k \sum_l \frac{\partial w_l}{\partial \mathbf{x}},$$

2. Local models require no less than 27 variables, most of them vectors or matrices, for storing all the memory terms and sufficient statistics, etc.

where $\frac{\partial w_k}{\partial \mathbf{x}} = -w_k \mathbf{D}_k(\mathbf{x}-\mathbf{c}_k)$ for the Gaussian kernel (2). The local models $\psi_k(\mathbf{x})$ are computed by PLS recursions, and therefore also the gradients of (3) have to be calculated in this rather costly way:

$$\begin{aligned}\frac{\partial \psi}{\partial \mathbf{x}} &= \sum_{i=1}^{R} \beta_i \frac{\partial s_i}{\partial \mathbf{x}_0} = \sum_{i=1}^{R} \beta_i \left(\mathbf{u}_i^T \frac{\partial \mathbf{x}_{i-1}}{\partial \mathbf{x}_0} \right)^T = \sum_{i=1}^{R} \beta_i \left(\frac{\partial \mathbf{x}_{i-1}}{\partial \mathbf{x}_{i-2}} \cdots \frac{\partial \mathbf{x}_1}{\partial \mathbf{x}_0} \right)^T \mathbf{u}_i \\ &= \beta_1 \mathbf{u}_1 + \beta_2 (\mathbf{I} - \mathbf{u}_1 \mathbf{p}_1^T)\mathbf{u}_2 + \beta_3 (\mathbf{I} - \mathbf{u}_1 \mathbf{p}_1^T)(\mathbf{I} - \mathbf{u}_2 \mathbf{p}_2^T)\mathbf{u}_3 + \dots \ .\end{aligned}$$

However, between updates (for example, during prediction-only cycles) or after training has finished, the slopes $\frac{\partial \psi}{\partial \mathbf{x}}$ of the local models do not change. Our implementation exploits this by memorising the slopes once a gradient is calculated, and directly using these slopes for predictions without running through the PLS recursions.

3.4 Matlab Interface

Our library started its life as a Matlab-only implementation, and therefore the Matlab struct describing an LWPR model is practically identical to the data structure used within the C library. When calling the C functions from Matlab via MEX-wrappers, however, these data structures normally have to be converted back and forth, which is time-consuming. Therefore, we added a special storage mechanism to our MEX-wrappers, which allows us to transfer Matlab data to and from C-managed memory. Then, updates and predictions of an LWPR model can be computed by calling the "normal" MEX-functions, but passing a certain reference identifier instead of the complete Matlab struct. As an illustration of the performance gain, we trained an LWPR model on a 2D toy data set. For accomplishing 10,000 updates, the Matlab-only implementation took roughly 100 seconds, using the MEX wrappers alone took 11.3s, but with our storage mechanism the task is finished after only 0.8s. The Matlab implementation—including the MEX-wrappers and the storage scheme—is also compatible with recent versions of Octave.[3]

Acknowledgments

This work has been carried out with support from the EU FP6 SENSOPAC project to SK and SV, funded by the European Commission.

References

P. H. Garthwaite. An interpretation of partial least squares. *Journal of the American Statistical Association*, 89(425):122–127, 1994.

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

S. Vijayakumar, A. D'Souza, and S. Schaal. Incremental online learning in high dimensions. *Neural Computation*, 17:2602–2634, 2005.

3. Octave is a free Matlab clone available at `http://www.octave.org`. We tested our library against version 2.9.12.