Locally Weighted Projection Regression (***LWPR***) - *a users manual*

Sethu Vijayakumar

Computational Learning and Motor Control Laboratory, Dept. of Computer Science & Neuroscience
University of Southern California, Los Angeles, 90089-2520

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. At its core, it uses locally linear models, spanned by a small number of univariate regressions in selected directions in input space. This non-parametric local learning system

- i) learns rapidly with second order learning methods based on incremental training
- ii) uses statistically sound stochastic cross validation to learn
- iii) adjusts its weighting kernels based on local information only
- iv) has a computational complexity that is linear in the number of inputs, and
- v) can deal with a large number of - possibly redundant & irrelevant - inputs

There are essentially three different ways in which one could use the LWPR algorithm:

1) **BATCH**: Using the **xlwpr** executable to do an incremental *batch* fitting of data (the training & testing data are drawn from a fixed *batch* pool; however, the algorithm works completely incrementally). You get continuous feedback about the learning progress through a trace. This is essentially an incremental regression fitting with a fixed data set.
2) **ONLINE**: Embedding the ***LWPR*** subroutines in real time control programs to achieve online learning. The training data is sampled in real time from a real data generating mechanism (for e.g., a moving robot). The learning happens incrementally as before. The querying (prediction) is also done in real time.
3) **DEBUG**: Using **xlwprstat** to check / decode a learned ***LWPR*** model. This debugging / visualization tool is completely menu driven.

2 Data Formats (for BATCH learning)

When using the ***LWPR*** algorithm for BATCH learning, the training as well as the test data should be input as *ASCII* files in the following format:

$x^{(1)}_{1}$ [space] $x^{(2)}_{1}$ … $x^{(n\_in)}_{1}$ [space] $y^{(1)}_{1}$ … $y^{(n\_out)}_{1}$ [\newline]
$x^{(1)}_{2}$ [space] $x^{(2)}_{2}$ … $x^{(n\_in)}_{2}$ [space] $y^{(2)}_{2}$ … $y^{(n\_out)}_{2}$ [\newline]

•

•

$x^{(1)}_{ndata}$ [space] $x^{(2)}_{ndata}$ … $x^{(n\_in)}_{ndata}$ [space] $y^{(1)}_{ndata}$ … $y^{(n\_out)}_{ndata}$ [\newline]

The file names are passed to the program through the script file. The program converts these *ASCII* data files into *binary* (BIG ENDIAN) after the first read and stores them in the same directory as .bin file for faster access.

3 Parameters & Script File

A script file is a convenient way to specify parameters of ***LWPR***. In this section, we describe the script file and the set of parameters that can be passed to the ***LWPR*** algorithm through them. It is parsed according to a keyword and some following arguments. Some of the script variables (as will be described in the later part of this section) are essential parameters of the learning algorithm like the learning rates etc. while the others are optional in the sense that they will be allocated a default value hard-coded (in the header file: lwpr.h) into the program, if not specified. In the script file, we also need to pass the training and testing data parameters specifying how and where to handle the training and test data sets. An example of the script file is shown in Section 3.2.

3.1 Script descriptors, Parameters & their interpretation

- *Necessary parameters of the script file are in normal bold font; the optional parameters (in italics, bold) have a default value hard coded in the program.*
- *Default values are specified in [ ] brackets (wherever appropriate). These default values can be changed by editing the main header file (lwpr.h) and recompiling the **LWPR** library*
- *The script file name (*.script) is passed as a run time parameter (an example of the script file is given later)*
- *We mark with an asterisk (*) those parameters that need to be set carefully, affect the learning performance considerably and might need a little bit of experimentation. We also insert comments that might be helpful in setting these parameters correctly.*

*Necessary **LWPR** variables (need to be specified in the script file)*

1. **n_in**	:	number of input dimensions in the regression
2. **n_out**	:	number of output dimensions in the regression
3. **diag_only**	:	1 – make the distance metric diagonal only 0 – allow full distance metric

(Initially try to start with diag_only – this computes much faster. In general, if your problem can be solved with diag_only option (non-skewed distance metric), stick to it!!)

4. **allow_meta_learning**	:	1 (0) – allow (disallow) second order learning. More details on second order learning in Section 4.2.6.
5. **meta_learning_rate**	:	To be used if second order learning is enabled
6. **penalty(γ) ***	:	Multiplication factor for the regularization penalty term in the optimization functional (See Eq. 1)
7. **init_all_alpha ***	:	Distance Metric learning rate initialization for gradient descent (currently implements only equal rates along all dimensions, i.e. initializes all the learning rates with the same parameters.

(*If you see lots of large update warnings or instability in convergence, you have too large a learning rate (**alpha**). If the Mean Squared Error is decreasing but the convergence is slow, you might try increasing the learning rate.)*

8. **norm ***	:	Normalization constant by which each input column is divided (one entry for every input dimension) - This tries to make the all the inputs dimensionless.

(*In general, it is dangerous to divide each dimension by it's variance without considering the physical properties of the input values since some input dimensions may be actually moving very little relative to it's range. Ideally, one should know the range of possible inputs in each dimension and try to normalize each input by that*)

9. **lwpr_name**	:	Filename to be used to store the learned model in binary(compact) format.

Optional LWPR variables (Specifying it overrides the default values)

10. ***max_rfs*** [100000]	:	Maximum number of local models or receptive fields (RFs) allowed (New local models are not added once this limit is reached)
11. ***init_all_diag_D*** [100.0] *******	:	Value to initialize ALL the diagonal elements of distance metric (isotropic distance metric initialization = same values on diagonal) Theoretically, normalization of the input data should allow to use isotropic distance metric D
12. ***init_diag_D*** [100 … 100] *******	:	Initial values of the diagonal elements of distance metric – used to specify non-isotropic but diagonal initialization of distance metric

(*The distance metric automatically adjusts itself if the distance metric learning is enabled. However, convergence properties and speed are strongly dependent on a good initialization. What can bad choices do :*

- *Too small value of D (large receptive fields) can lead to local minima and delay convergence*
- *Too large value of D (small receptive fields) can lead to allocation of too many receptive fields & overfitting*

Theoretically, the learning mechanism takes care of these problems. But, there is nothing like a good initialization to make things easier for the algorithm !!! One way of guessing a good initialization is to guess the Hessian of the function being approximated and put a conservatively big initialization of receptive field based on the curvature.

13. ***init_D*** [diag=100.0, offDiag = 0.01]	:	Initial values of distance metric (full) - specified only when we allow full distance metric
14. ***add_threshold*** [0.5]	:	The mean squared error of the current regression dimension is compared against the previous one. Only if the ratio of the $nMSE^{r-1} / nMSE^{r}$ < **add_threshold,** a new regression direction is added. This criterion is used in conjunction with some other checks to ensure that the decision is based on enough data support.
15. ***init_lambda*** [0.995]	:	Initial forgetting factor (Forgetting factor described in detail in Section 4.2.7)
16. ***final_lambda*** [0.99999]	:	Final forgetting factor
17. ***tau_lambda*** [0.9999]	:	Annealing constant for the forgetting factor

18. ***init_n_reg*** [min{2,n_in}]	:	Number of projections to start regression with (Usually start with 2 – one for regression and the other to compare whether an additional dimension should be used. Starting with 1 – implies use of LWPLS_1, a variation of LWPLS algorithm, which employs only one projection and adjusts the distance metric to compensate for this limited ability.) See reference [2] for details on this variation of the PLS.
19. ***init_S2*** [1.e-6]	:	Initial value for the covariance computation of the data (to handle the case when no data has been seen so far)
20. ***w_gen*** [0.1]	:	weight activation threshold. A new local model is generated if no existing model elicits response (activation) greater than **w_gen** for a training data
21. ***w_prune*** [0.9]	:	If a training data elicits responses greater than **w_prune** from 2 local models, then, the one with the greater error is pruned.
22. ***factor_prune*** [5.0]	:	All local models (RFs) have a variance in their error. If any RF exceeds the standard deviation (STD) of all the RFs by the **factor_prune**, it will be pruned. Normal values for **factor_prune** are 1.96, 2.57, 3.17, etc, the values for the area under a normal distribution
23. ***allow_addDim*** [TRUE]	**:**	1(0) - Allow(disallow) the addition of projection direction (to the initial number of projections specified by **init_n_reg**) if deemed necessary. This acts as a debugging variable.
24. ***allow_d_update*** [TRUE]	**:**	1(0) - Allow(disallow) the gradient descent up date of the Distance Metric D (initialization done by **init_D/init_diag_D/ init_all_diag_D)** This acts as a debugging variable.
25. ***write_rfs*** [TRUE]	:	1(0) – Enable(Disable) the warning messages during distance metric updates and the process of writing out (saving in a file) the details of receptive fields which are pruned as a result of the update.
26. ***life_decay*** [0.001]	:	a number in [0,1] which allows the life value of the receptive field (RF) to decay

27. ***die_threshold*** [0.9999]	:	if the life_value of the RF falls below this value and the RF has not seen enough data points, the RF is pruned
28. ***c_noise*** [0.01]	:	percentage noise to be added to the training data location when initializing the center of receptive field (to generate some stochasticity).
29. ***d_noise*** [0.01]	:	percentage noise to be added to the default distance metric initialization
30. ***use_nn*** [FALSE]	:	1(0) – Use (Do not use) nearest neighbor based lookup to speed training updates. Refer to Section 4.2.5 for details.
31. ***max_nn_comp*** [100000000]	:	Maximum number of nearest neighbors to be used in the lookup or update (if **use_nn** parameter is ON).

Parameters for handling evaluations (Training & Testing) data sets

32. **sampling_method** :	Specifies the method to sample from the data file (1 – random, 2- sequential , 3-generating function based, 4-pseudo random). Usually pseudo-random sampling (4) is a safe bet. It ensures that you sample randomly but still cover all the data points in one round. It corresponds to random sampling without replacement.
33. **index_function** :	Used in conjunction with *generated functions* kind of sampling. Refer to utility / data_sets.c for details – this requires preprogramming for usage. If using sampling methods 1,2 or 4, just assign some value (say 3) to this variable.
34. **max_iterations** :	Max. number of training iterations to run the algorithm (One iteration corresponds to presentation of one data point)
35. **eval_time** :	Evaluation of test statistics (testing error etc.) is done after **eval_time** iterations
36. ***cutoff*** [0.001] :	If none of the local models (receptive fields) elicits activation greater than **cutoff** for a query point, that query is labeled as not being well supported. We should not trust predictions for this data point (We get separate test statistics which include / exclude these queries). ***LWPR*** returns ZERO for this model – there is no unlimited extrapolation.
37. ***blending*** [TRUE] :	(1 / TRUE) : Allows weighted mixing of results from overlapping receptive fields. (0 / FALSE) : Winner-take-all prediction.
38. **file_name_training_data** :	File name containing the training data
39. **file_name_testing_data** :	File name containing the testing data
40. **n_training_data_columns** :	Number of columns in training data file
41. **n_testing_data_columns** :	Number of columns in testing data file
42. **index_train_input_columns**:	Indexes corresponding to input columns of the training data (allows reordering of columns)
43. **index_train_output_columns**:	Indexes corresponding to output columns of the training data
44. **index_test_input_columns**:	Indexes corresponding to the input columns of the testing data
45. **index_test_output_columns**:	Indexes corresponding to the output columns of the testing data

3.2 Example script file

The following is a sample script file for a batch fitting of a data set with 21 dimensional input, one dimensional output. The training data is available in a file sample_inv_train.data and the testing data in sample_inv_test.data

```
#LWPR

/* general variables */

n_in                        = 21
n_out                       = 1
diag_only                   = 1
allow_meta_learning         = 1
penalty                     = 1.e-4
norm                        = 1 1 1 1 1 1 1 8 8 8 8 8 8 8 40 40 40 40 40 40 40
cutoff                      = 0.01
blending                    = 1
w_gen                       = 0.01
lwpr_name                   = sample
use_nn                      = 1
max_nn_comp                 = 5

init_lambda                 = 0.995
final_lambda                = 0.9995
tau_lambda                  = 0.99995



/* regression learning variables */

meta_learning_rate          = 5
init_all_alpha              = 1
init_all_diag_D             = 7
init_n_reg                  = 2
add_threshold               = 0.9

/* debugging variables */

write_rfs                   = 1
allow_d_update              = 1
allow_addDim                = 1
max_rfs                     = 1000
c_noise                     = 0
d_noise                     = 0
```

```
/* data set variables */

sampling_method             = 2
index_function              = 3
max_iterations              = 500000
eval_time                   = 20000
file_name_training_data     = sample_inv_train.data
file_name_testing_data      = sample_inv_test.data
n_training_data_columns     = 35
n_testing_data_columns      = 35
index_train_input_columns   = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
index_train_output_columns  = 22
index_test_input_columns    = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
index_test_output_columns   = 22
```

4 Basic elements of the program code

4.1 The event loop

The main ***LWPR*** program resides under functions located in the lwpr.c file. However, to run the algorithm on an a data set, we need to run an event loop which draws and adds data to the algorithm from the data file and does appropriate actions at the desired time.

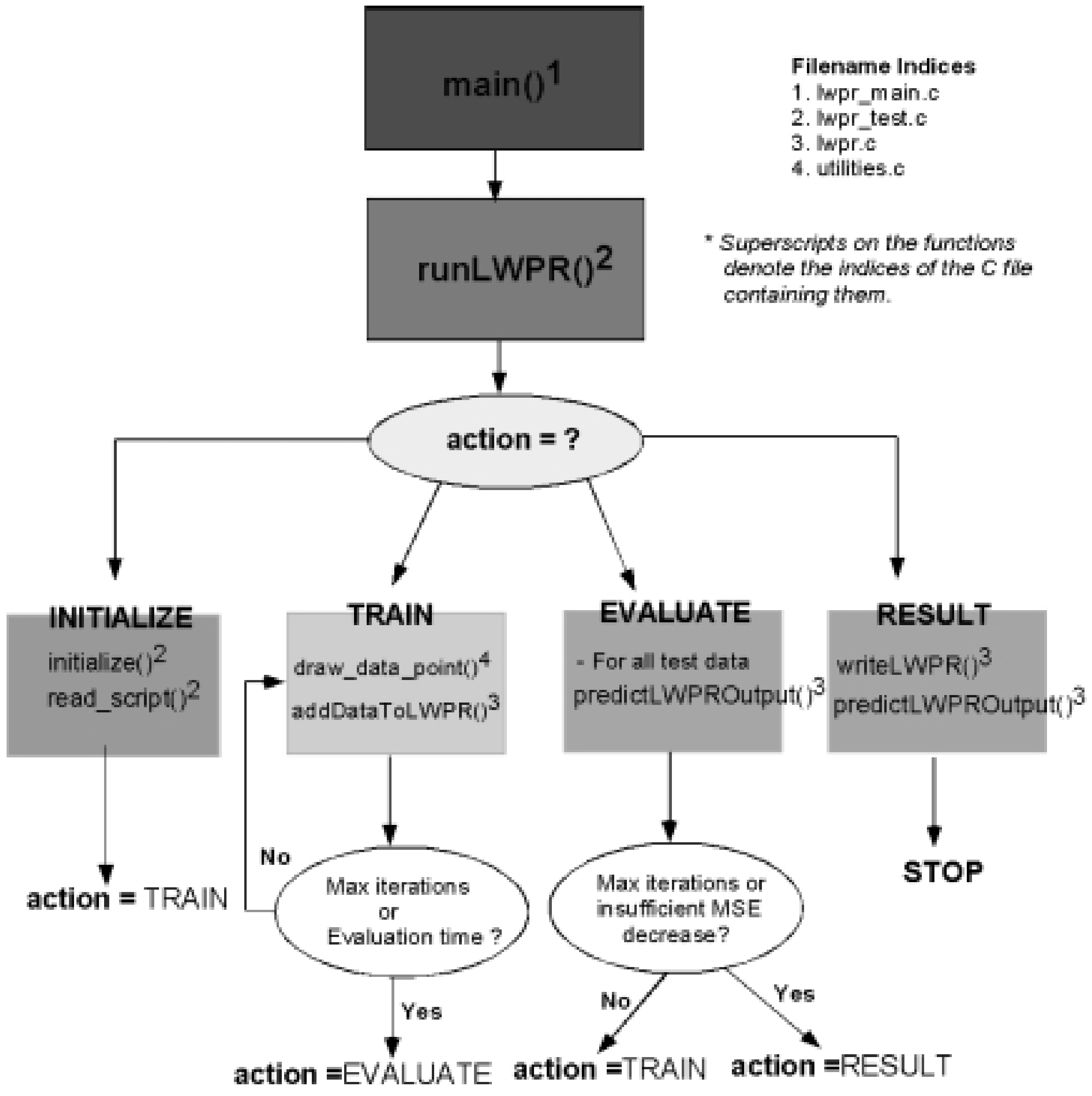

Figure 1: Schematic flowchart of ***LWPR*** algorithm

The structure of the event loop is shown in Fig.1. The algorithm is at one of the four action states at any given point of time. The INITIALIZE phase is used to initialize the ***LWPR*** and read in the script variables from the script file and fill in default values for those variables not specified in the script file.

The TRAIN phase of the algorithm draws data from the training data set file and trains the local model on it. The details of learning with addition of data (which constitute the major part of the program code) are described in detail in the next section.

After every **eval_time** iterations, the program goes into the EVALUATE phase where the learned model is tested against the novel (test) data set. Run time learning traces – which are described in detail in the section on tracing, evaluation & debugging – are generated to help track the progress of the algorithm.

When the number of iterations has exceeded the **max_iterations** count or the change of normalized mean squared error (nMSE) between the last EVALUATE phase and the current is below a THRESHOLD (specified in lwpr_test.c), the program goes into the RESULT phase. In the RESULT phase, it dumps (saves) the learned ***LWPR***, saves the result of evaluation on the test set in a *.res file and PAUSES (stops).

4.2 Details of the Main C-Routine

The main routine in the C-code is the one that incorporates the new data into the current learnt model (addDataToLWPR()). Fig.2 describes the schematic of this routine. Most of the details are self-explanatory and the code itself has extensive comments which show what is being handled at which stage. During training, the main tasks to be carried out includes updating all the trace variables (running mean, variance etc.), updating the projection direction and the regressions along these directions and updating the distance metric according to the cost function gradient. In addition, one also needs to check whether a new projection direction needs to be added, whether a new RF (local model) needs to be added or whether one needs to be pruned. We will briefly describe each of these subroutines in the next subsections.

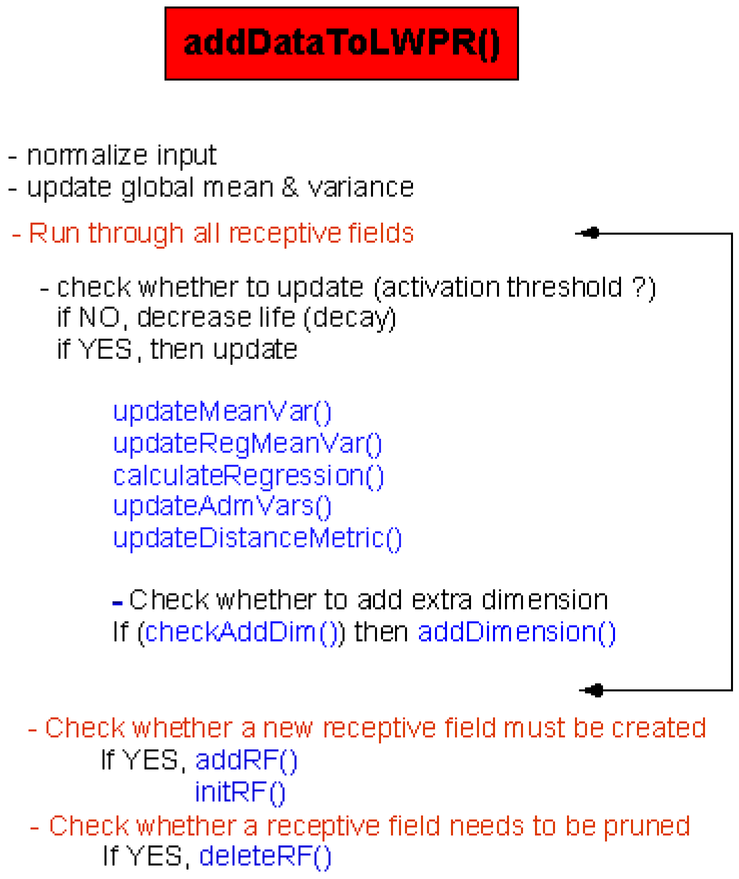
addDataToLWPR()
- normalize input
- update global mean & variance
- Run through all receptive fields
- check whether to update (activation threshold ?)
if NO, decrease life (decay)
if YES, then update
updateMeanVar()
updateRegMeanVar()
calculateRegression()
updateAdmVars()
updateDistanceMetric()
- Check whether to add extra dimension
If (checkAddDim()) then addDimension()
- Check whether a new receptive field must be created
If YES, addRF()
initRF()
- Check whether a receptive field needs to be pruned
If YES, deleteRF()

Figure 2: Main addData routine of the ***LWPR*** code

4.2.1 Updating the regression (and trace variables) – **calculateRegression()**

One of the key ingredients of the ***LWPR*** algorithm is the ability to do a linear regression on high dimensional input data by employing very few (local) projections. This is based on a modification of the Partial Least Squares (PLS) regression to perform incremental, locally weighted updates. The details of the method can be looked up in [2]. In Table 1, we provide the incremental version of the regression updates that are implemented in the ***LWPR*** code. Here, **x** is the new training input and the rest are trace variables that are maintained (updated) with each additional data. The regression itself turns out to be a simple division of two trace terms since the orthogonality ensures that we can perform the high dimensional regression as a series of univariate regression without the need for expensive matrix inversions.

Initialize: $\mathbf{z}_0 = \mathbf{x} - \mathbf{x}_0^{n+1}$, $res_0 = y - \beta_0^{n+1}$

For $i = 1:r$,

a) $\mathbf{u}_i^{n+1} = \lambda\, \mathbf{u}_i^n + w\, \mathbf{z}_{i-1}\, res_{i-1}$

b) $s_i = \mathbf{z}_{i-1}^T \mathbf{u}_i^{n+1}$

c) $SS_i^{n+1} = \lambda\, SS_i^n + w s_i^2$

d) $SR_i^{n+1} = \lambda\, SR_i^n + w s_i^2\, res_{i-1}$

e) $SZ_i^{n+1} = \lambda\, SZ_i^n + w\, \mathbf{z}_{i-1}\, s_i$

f) $\beta_i^{n+1} = SR_i^{n+1} / SS_i^{n+1}$

g) $\mathbf{p}_i^{n+1} = SZ_i^{n+1} / SS_i^{n+1}$

h) $\mathbf{z}_i = \mathbf{z}_{i-1} - s_i \mathbf{p}_i^{n+1}$

i) $res_i = res_{i-1} - s_i \beta_i^{n+1}$

j) $SSE_i^{n+1} = \lambda\, SSE_i^n + w\, res_i^2$

Table 1: Update equations for the regression & other trace variables

The other thing to note is that since we are working on orthogonal input (residual) spaces and also on residual outputs, the addition of an extra dimension to the regression does not affect the results that have already been learnt, making this ideally suited for an incremental growth of the number of projection directions (or latent variables, as one may like to think about the projections).

4.2.2 *Gradient descent on the Distance Metric* – updateDistancMetric()

The distance metric specifies the area in which a local model (RF) is valid. The size and shape of the RF is constantly modified by performing a gradient descent update of the metric D based on a leave-one-out cross validation cost function :

$$\mathbf{D}^{n+1} = \mathbf{D}^{n} - \alpha \frac{\delta J_{\cos t}}{\delta \mathbf{D}},$$

$$where \quad J_{\cos t} = \frac{1}{W} \sum_{i=1}^{M} \sum_{k=1}^{r} \frac{w_i y_{\mathbf{res},k+1,i}^2}{(1 - w_i \frac{s_{k,i}^2}{s_k^T W s_k})^2} + \gamma \sum_{i,j=1}^{N} D_{ij}^2$$

Equation 1: LOO-CV based cost function for distance metric updates

The above update can be computed in an incremental manner with the addition of each training data. We need to maintain certain number of trace terms to this end. The update equations (details of which can be found in [1]) that are implemented in the code are as follows:

$$\mathbf{M}^{n+1} = \mathbf{M}^{n} - \alpha \frac{\partial J}{\partial \mathbf{M}} \quad \text{with } \mathbf{D} = \mathbf{M}^T\mathbf{M}, \text{where } \mathbf{M} \text{ is upper triangular}$$

$$W^{n+1} = \lambda W^{n} + w$$

$$E_k^{n+1} = \lambda E_k^{n} + w\, res_{k-1}^2$$

$$\mathbf{H}_k^{n+1} = \lambda \mathbf{H}_k^{n} + \frac{w s_k\, res_{k-1}}{1 - h_k}, \quad \text{where} \quad h_k = w \frac{s_k^2}{SS_k^{n+1}}$$

$$\mathbf{R}_k^{n+1} = \lambda \mathbf{R}_k^{n} + \frac{w^2\, res_{k-1}^2 s_k^2}{1 - h_k}$$

$$\frac{\partial J}{\partial \mathbf{M}} \approx \frac{\partial w}{\partial \mathbf{M}} \sum_{i=1}^{p} \sum_{k=1}^{r} \frac{\partial J_{1,r,i}}{\partial w} + \frac{w}{n W^{n+1}} \frac{\partial J_2}{\partial \mathbf{M}}$$

where:

$$\frac{\partial w}{\partial M_{rl}} = -\frac{1}{2} w (\mathbf{x} - \mathbf{c})^T \frac{\partial \mathbf{D}}{\partial M_{rl}} (\mathbf{x} - \mathbf{c}), \quad \frac{\partial J_2}{\partial M_{rl}} = 2\gamma \sum_{i,j=1}^{n} D_{ij} \frac{\partial D_{ij}}{\partial M_{rl}}$$

$$\frac{\partial D_{ij}}{\partial M_{rl}} = \delta_{lj} M_{ri} + \delta_{il} M_{rj} \quad (\delta \text{ is the Kronecker operator})$$

$$\sum_{i=1}^{p} \sum_{k=1}^{r} \frac{\partial J_{1,r,i}}{\partial w} \approx \sum_{k=1}^{r} \left[-\frac{E_k^{n+1}}{\left(W^{n+1}\right)^2} + \frac{1}{W^{n+1}} \left(res_{k-1}^2 - 2 \frac{res_k s_k}{SS_k^{n+1}} \mathbf{H}_k^{n} - 2 \left(\frac{s_k}{SS_k^{n+1}} \right)^2 \mathbf{R}_k^{n} \right) \right]$$

Table 2: Gradient descent update of the Distance Metric

4.2.3 Adding an extra projection dimension

When we start with **init_n_reg**=1, then, we interpret this as a special case of one projection ***LWPR*** where the number of local projection employed by each local model is restricted to one. The distance metric adjusts in order to accommodate for this restrictions. However, this may take a long time to converge for regression problems in which the inherent dimensionality is high.

Usually, if we have no prior knowledge of the intrinsic local dimensionality, we should start with **init_n_reg**=2. This implies that we actually employ only one dimension for the regression and the additional dimension we use to check whether to add an extra projection or not. The mean squared error of the current regression dimension is compared against the previous one. Only if the ratio $\mathrm{nMSE}^{r-1} / \mathrm{nMSE}^{r}$ < **add_threshold** (i.e. we get substantial reduction of error by adding an extra dimension), a new regression direction is added. This criterion is used in conjunction with a few more checks to ensure that the decision is based on enough data support before deciding to add a new dimension.

4.2.4 Pruning an existing RF (local model)

Usually, if there is a local model that elicits substantial activation in response to a training data, then, it prevents the allocation of an additional local model for that training data. However, since the distance metric (and hence, the range of validity of the local model) is changing with the gradient descent updates, there can arise cases in which there is a considerable overlap between two local models. Due to this and a few other reasons listed below, we might need to prune the local model and update the necessary book keeping variables.

The pruning could be due to

(i) too much overlap with another one (If 2 receptive fields elicits response greater than **w_prune** to a training data), - one with the larger error is pruned

(ii) too high variance in the error compared to the STD of all the RFs (determined by the variable **factor_prune**)

(iii) excessive error in one of the RFs (not yet implemented)

(iv) lost RFs (those local models which have seen very few training data and have not been updated for a long time - not yet implemented)

The details of the pruned models are dumped for verification purposes. In general, one should be ‘worried’ if there is too much pruning. This implies that your initialization of the distance metric (local RF range) is bad – usually too local or that the **w_gen** threshold is too low resulting in excessive number of local models in places where it is not needed.

4.2.5 Maintaining the local nearest neighbor (nn) list

When using the regression analysis in applications where the input values changes smoothly (i.e., do not jump around) – typically in movement systems like robots – it is useful to keep a neighborhood list and perform training by looking at only the neighboring local models which are close to each other or have a substantial overlap in their activation profiles. This saves a lot of computing resources as opposed to going through all the local models and finding out those that have enough activation to be updated.

Hence, it suffices to look at the neighborhood list to check for activations that are above the threshold and need to be updated.

4.2.6 Second order updates

The gradient descent updates of the distance metric is speeded up – faster convergence - and is more efficient if we use Newton's second order gradient information (meta learning) and incorporate this into the learning updates.

If the **allow_meta_learning** variable is TRUE, then the second order learning is switched on.

4.2.7 Forgetting factor

The forgetting factor is a variable that is used to discount the effects of the statistics computed at an earlier stage (when we had seen very few data points) and give more weight to the recent statistics - which are a result of having experienced more data points. It can be thought of as a sliding window over which the stochastic sufficient statistics are accumulated.

The forgetting factor (lambda) takes a value [0,1] where 0 corresponds to using only the current point and 1 corresponding to not `forgetting ' anything. Here, we use an annealed forgetting rate which forgets more at the start (to account for unsettled learning dynamics) – specified by **init_lambda** and anneals towards a value closer to one (**final_lambda**) - not forgetting anything based on annealing factor **tau_lambda**.

5 Unpacking, Compiling and Creating Executables

Unpack the ***LWPR*** distribution (lwpr.tar.gz) using the gzip –d option and then, tar –xvf option under your home directory or wherever you want to install it (here, called /MYHOME for explanation purposes). This procedure will create a directory /MYHOME/lwpr and unpack all the files under this.

The /MYHOME/lwpr/src directory contains all the source files of the algorithm (like lwpr.c, lwpr.h etc.). There are machine specific subdirectories (like /lwpr/sparc, /lwpr/mac etc.) for putting the appropriate object files and executables for execution on different platforms. This installation of the executable to the appropriate machine specific directory is handled by the Makefile. Then, there is an /lwpr/Imakefile which is used for generating the Makefile for compiling the code. It is much easier and intuitive to make changes in the Imakefile and automatically generate the Makefile from it as compared to directly trying to modify the Makefile.

6 Running the *LWPR* algorithm in BATCH mode (xlwpr)

Let us assume that we are running a learning/generalization test for a *sample* dataset. Usually, it is a good idea to create a subdirectory under the lwpr directory like MYHOME/lwpr/sample and put all the scripts and test & train data sets specific to that problem (in our case *sample*) in that subdirectory. This is not necessary as long as you provide the complete path for accessing these files but in our example here, we will follow this organization.

Checklist before running the program for *sample* dataset:

2. Ensure that there exists a script file sample.script with the necessary run time parameters for the problem in /MYHOME/lwpr/sample.
3. Ensure that the directory /MYHOME/lwpr/sample contains the testing and training data sets and their names are specified in the sample.script file.

Change directory to /MYHOME/lwpr/sample to run ***LWPR***. This is useful since all the program trace, learned models and other outputs are written in the directory where the program is started. After making sure that the executable **xlwpr** is in your path, type the following command at the prompt.

```
➢ xlwpr sample.script
```

You are prompted

```
➢ Generate new LWPR =1; Read from file = 0 [0]:
```

If you are starting a fresh learning job, enter 1. If you want to continue learning using a previously stored model, enter 0 (default). In the latter case, it will look for a stored version of the model based on the name specified in the script file. If it doesn't find the stored model, it will initialize a new model and start learning.

The storing of the learned model is done at completion of learning - either after convergence or at the end of the maximum number of iterations (**max_iterations**) specified in the script file.

7 Tracing and Evaluation in batch mode

In this section, we will concentrate on the features offered by ***LWPR*** for tracing the progress of learning and evaluation mechanisms. We will systematically look at the outputs produced by the algorithm and lay out methods of interpreting the results.

7.1 Runtime trace outputs

At runtime, the ***LWPR*** algorithm generates learning traces on the screen after every **eval_time** iterations, which look like this:

```
iter=150000:    nMSE =  0.069462 ( 0.069462)  MSE =  2.517709 ( 2.517709)
#rf=119  #tr=1  #reg=2.924433  #pre=0  #pro=3  #mu=0  #lo=0
```

7.1.1 How to interpret the runtime outputs

The interpretation of each variable is as follows:

1. **iter** : Number of data points the algorithm has seen so far.
2. **nMSE**: Normalized mean squared error. This is the mean squared error normalized by the output variance. The number in the brackets is the nMSE after EXCLUDING the poorly supported query points. Poorly supported query points are those data points which did not elicit an activation above certain **cutoff** threshold (specified in lwpr_test.c) from any of the existing local receptive fields, i.e. no coverage of input space for these points.
3. **MSE**: Mean squared error. The interpretation of the numbers in the bracket is the same as for nMSE.
4. **#rf**: The number of receptive fields (local models) allocated so far.
5. **#tr**: Number of the untrustworthy(without enough data support) receptive fields - this happens if a receptive fields has not seen more than **2*n_in** (a default value hard coded in the program) data points.
6. **#reg**: The average number of projection directions employed by ***LWPR*** (average taken over all the local models)
7. **#pre**: Number of receptive fields pruned due to excessive error (not used currently).
8. **#pro**: Number of receptive fields pruned due to high overlap error
9. **#mu**: Number of mutations among the receptive fields error (not used currently)
10. **#lo**: Number of lost receptive fields that were pruned error (not used currently).

7.1.2 Runtime warnings and error messages

In addition to the learning trace described in the previous section, you may also encounter warning messages if the **write_rfs** parameter is enabled (TRUE). The messages are usually of the form:

```
WARNING:Large M[4][4]-Update: ID=1 rfID=3 n_data=54.902568
WARNING:Large M[16][16]-Update: ID=1 rfID=2 n_data=21.778989
WARNING:Large M[16][16]-Update: ID=1 rfID=2 n_data=54.172665
WARNING:Large M[16][16]-Update: ID=1 rfID=2 n_data=64.748642
WARNING:Large M[16][16]-Update: ID=1 rfID=2 n_data=68.102163
WARNING:Large M[16][16]-Update: ID=1 rfID=2 n_data=68.762921
WARNING:Large M[1][1]-Update: ID=1 rfID=1 n_data=196.255219
WARNING:Large M[4][4]-Update: ID=1 rfID=4 n_data=172.250223
WARNING:Large M[1][1]-Update: ID=1 rfID=1 n_data=186.160585
WARNING:Large M[4][4]-Update: ID=1 rfID=6 n_data=161.989528
```

This is a warning that triggers when the distance metric updates are too large. This usually happens when we select *too high* a learning rate or there is something *unstable in the convergence* process. Currently, we restrict the gradient descent updates of the distance metric **D** to be less than 10% of the largest **M** matrix element, where $\mathbf{D}=\mathbf{M}^T\mathbf{M}$ (**M** is the Cholesky decomposition of **D**). When this limit is exceeded, we output this warning and reduce the learning rate corresponding to that dimension by half.

If this persists for a long time, it is advisable to stop the learning, change the learning rate or check other parameters.

7.2 Results (at the end of run) and interpretation

In this section, we will list all the outputs from the program that are generated to help evaluate the performance of the algorithm and also to store away the learnt model. This happens at the termination of the learning cycle (either due to convergence - specified by the TOLERANCE parameter in lwpr_test.c or on exceeding the **max_iterations** number of data presentations).

7.2.1 The LWPR dump (.lwpr)*

The final learnt model is dumped in binary form using the name specified in the script file (e.g. sample.lwpr). If one wishes to continue to learn by adding more data to the model, the stored model can be read at run time. The receptive field (RF) dump is in normalized coordinates. It is not possible to read the contents of this dump directly. However, we can use **xlwprstat** to analyze this dump. Details of using this debugging tool are explained in detail in Section 8.

7.2.2 The trace file (.trace)*

The trace file (sample.trace) stores the trace details of the algorithm based on evaluations at every **eval_time** cycle. The outputs refer to the variables listed below, where nMSE(without) refers to the error computed taking only query points with enough model support into account.

```
[Iter; nMSE ; nMSE(without); MSE ; MSE(without); #rf; #reg]
20000    0.183908        0.183908 38.78562  38.78562  36   1.000000
                                            •
                                            •
                                            •
100000  0.083530  0.083530  3.027616  3.027616  982 2.824847
```

7.2.3 The final test data result (.res)*

The final regression result (or prediction) on the entire test set is output in ASCII form in the appropriate *.res file (sample.res) at the end of the learning cycle.

$x^{(1)}_{1}$ [space] $x^{(2)}_{1}$ … $x^{(n\_in)}_{1}$ [space] $y_{pred}{}^{(1)}_{1}$ … $y_{pred}{}^{(n\_out)}_{1}$ [\newline]
$x^{(1)}_{2}$ [space] $x^{(2)}_{2}$ … $x^{(n\_in)}_{2}$ [space] $y_{pred}{}^{(2)}_{2}$ … $y_{pred}{}^{(n\_out)}_{2}$ [\newline]

•

•

$x^{(1)}_{ndata}$ [space] $x^{(2)}_{ndata}$ … $x^{(n\_in)}_{ndata}$ [space] $y_{pred}{}^{(1)}_{ndata}$ … $y_{pred}{}^{(n\_out)}_{ndata}$ [\newline]

7.2.4 Pruned receptive filed dumps

If the **write_rfs** parameter is TRUE (default), then, whenever a receptive field is pruned, the details of the pruned local model are dumped into a file whose name is derived from the ***LWPR*** and the ID number of pruned receptive field. In addition to this, the details of the receptive field that caused the pruning are also dumped. The pruning could be due to various factors that are described in detail in Section 4.2.4.

In the case when RF number 99 of the `sample' ***LWPR*** is pruned due to excessive overlap, an ASCII (readable) dump by the name sample.99.rf is generated which has the following details:

```
Data of RF #99 of LWPR >sample<:
message                   = overlap prune due to 97
n_reg                     = 1
cost                      = -999.000000
trustworthy               = 1
1.sum_weights             = 42.785174
1.sum_error               = 32.868158
1.n_data                  = 293.242146
1.lambda                  = 0.996632
sum_D2                    = 0.000000
n_updates                 = 9009
life_value                = 1.000000
Vector >RF-center c< :
                          vector comes here
Matrix >Regression Coefficients B< :
                          1.4780
Matrix >Memory trace H< :
                           -0.4651
Vector >Memory trace r< :
                          0.4639
Matrix >Distance Metric D< :
                          Distance metric matrix comes here
Vector >Mean in Input Space mean_x< :
                          Weighted mean comes here
Vector >Var in Input Space mean_x< :
                          Variance of data seen so far comes here
Vector >Mean in Output Space mean_y< :
                          3.4468
Vector >Var in Output Space var_y< :
                          0.8093
Vector >Weighted Sum of Regression Var ss2< :
                          0.9879
```

```
Vector >Local Degree of freedom< :
                          0.4469
Matrix >Weighted Sum of Regression Cov SXresYres< :
       8.5767  -0.2318   0.6348  -8.8523  -0.1138  -0.1007  -1.4910   0.3410  -2.2303
       3.0709  -0.1804  -0.2661   1.1584  -0.1353  -3.4792   2.9247  -5.5803   3.0089  -
       3.2418   1.0030  -0.0712



Matrix >Weighted Sum of Regression Cov SSXres< :
       0.0400  -0.0488   0.0637  -0.1243   0.0048   0.0097  -0.0195  -0.0288  -0.0101
       0.0241   0.0109  -0.0167   0.0103   0.0011  -0.2646   0.6262  -1.4444   0.3835
      -0.4135  -0.0954   0.0037

Matrix >Weighted Sum of Regression Cov SSYres< :
          1.4600
Matrix >Distance Metric Learning Rate alpha< :
```
Matrix of distance metric learning rate comes here
```
Matrix >Projections W< :
       0.5502  -0.0149   0.0407  -0.5679  -0.0073  -0.0065  -0.0957   0.0219  -0.1431
       0.1970  -0.0116  -0.0171   0.0743  -0.0087  -0.2232   0.1876  -0.3580   0.1930  -
       0.2080   0.0643  -0.0046
Matrix >Projections U< :
       0.0405  -0.0494   0.0645  -0.1259   0.0048   0.0098  -0.0197  -0.0292  -0.0102
       0.0244   0.0110  -0.0169   0.0105   0.0011  -0.2679   0.6339  -1.4622   0.3882  -
       0.4186  -0.0966   0.0037
Vector >Nearest Neighbors nn< :
```
Vector of nearest neighbor list comes here
```
Vector >Nearest Neighbors nnw< :
```
Vector of nearest neighbor weights comes here

8 DEBUG-ing the learnt ***LWPR*** model (xlwprstat)

xlwprstat is a tool that helps us to visualize and debug the learned ***LWPR*** models. We can use this tool after training the model on a particular data set. At the end of training, the model is dumped as *.lwpr. This binary file cannot be read as it is. However, we can use the debug tool to access the details of the learned model and visualize or analyze it in various ways. A typical runtime output is shown below.

```
> xlwprstat

Input File Name:

filename [inv_dynamics2DOF]: sample.lwpr

Reading file >sample.lwpr< ...done

Data of LWPR >sample<:


        n_in                = 21
        n_out               = 1
        init_n_reg          = 2
        diag_only           = 1
        #n_data             = 100002
        #rfs                = 49
        max_rfs             = 1000
        #no_trust           = 0
        #pruned error       = 0
        #pruned overlap     = 0
        #pruned lost        = 0
        #mutations          = 0
        w_gen               = 0.010000
        w_prune             = 0.900000
        factor_prune        = 5.000000
        final_lambda        = 0.9995
        init_lambda         = 0.995
        tau_lambda          = 0.99995
        penalty             = 0.0001
        init_S2             = 1e-06
        meta_learn_rate     = 5.000000
        d_noise             = 0.000000
        c_noise             = 0.000000
```

```
      mean_error        = 0.000000
      std_error         = 0.000000
      mean_n_reg        = 4.400000
      allow meta learn. = 1
      allow D update    = 1
      allow PCA add dim = 1
      add threshold     = 0.900000

Vector >Normalization Factors norm< :
          Vector of normalization constant comes here
Vector >Mean Inputs mean_x< :
          Vector of input means comes here
Vector >Var Inputs var_x< :
          Vector of input variance comes here

FURTHER OPTIONS:
            Write RF                      --> 1
            Write RF with largest error   --> 2
            Write trust RF w.larg.error   --> 3
            Lookup query point            --> 4
            Delete a RF                   --> 5
            Save LWPR                     --> 6
            Mathematica Surface           --> 7
            Matlab Output                 --> 8
            RBF-like data                 --> 9
            Test                          --> t
            Quit                          --> q
                         Input [q]:
```

The analysis tool essentially prints out the basic parameters of the learned model, the modified (learned) quantities and all the representative values like average number of projections, mean error, mean variance etc. In addition to this, we can use a menu driven set of options to **write** a particular RF, **write** RF with **largest error**, directly **lookup a query** point, **delete** a RF, **save** the modified model, **generate Mathematica or Matlab** compatible data output for visualization of the result (useful for low dimensional data or low dimensional slices through high dimensional data only) or generate plots which give the **contours of the receptive field** spread.

9 Embedding *LWPR* in an ONLINE real time system

There are some key functions that one will need to embed in their real time learning system to utilize the algorithm as an online incremental learning scheme. We will briefly describe what each of these functions achieves.

1. **addDataToLWPR()** : This function is the main routine involved in the learning phase. As described in detail in Section 4.2, it is this function that incrementally incorporates the new training data into the model and updates the necessary learning parameters. In a real time system, one should send the training data (input–output pairs) to this module at a frequency that can be afforded by your real-time computational constraints.
2. **predictLWPROutput()** : This function generates a prediction (output) for a given input data using the current model. For some problems where you need to predict on the same data that you used for training, a more efficient way (than doing the training and prediction separately) is to use the **addDataToLWPRPredict()** function which combines the above two functions. Generally, in online time critical implementations, the lookup or prediction has to be done with a higher priority than adding data to the learning system.

In addition to these critical functions that needs to be run continuously for real time operations, there are additional initialization and bookkeeping operations that need to be done. Some essential functions that are run primarily either at the start or end of the learning / predictions are listed here:

3. **readLWPRScript()** : This function reads the script file and parses the inputs based on some keywords. It is through this file that we pass the learning parameters to the real time learning system. This script is read when a new LWPR model is initialized.
4. **writeLWPR()** : This function dumps the final learnt module in binary form for future use. Debugging on this can be done using the **xlwprstat** executable.
5. **readLWPR()** : Using this function, any previously generated model that has been saved can be loaded and used for further learning or just prediction.

By embedding these sets of functions in a real time system, one can achieve the necessary online learning performance within the limits of the computational capabilities of the system.

10 Bibliography

1. Stefan Schaal, Chris Atkeson & Sethu Vijayakumar, Scalable Locally Weighted Statistical Techniques for Real Time Robot Learning, Applied Intelligence: Special Issue on Scalable Robotic Applications of Neural Elsevier Science (in press).
2. Sethu Vijayakumar & Stefan Schaal, LWPR: An O(n) Algorithm for Incremental Real Time Learning in High Dimensional Space, *Proc. International Conference on Machine Learning (ICML2000),* Stanford, CA pp.1079-1086 (2000).
3. Sethu Vijayakumar & Stefan Schaal, Real Time Learning in Humanoids: A challenge for scalability of Online Algorithms, *Humanoids2000, First IEEE-RAS Intl. Conf. on Humanoid Robots* MIT, Cambridge, MA, USA (2000).
4. Schaal, S. & Atkeson, C. G. "Constructive incremental learning from only local information." Neural Computation, 10, 8, pp.2047-2084 (1998).