# A Regression Tool for Multivariate Nonlinear System Approximation

## Supervised Machine Learning Based on Stochastic Back Propagation

Jiawei Xia, Haining Zhou, Qicang Shen, Bennett Williams

~~NERS 590~~

~~December 9, 2016~~

November 10, 2017

travel in time :)

# Motivation

Regression is a widely used tool for modeling systems, e.g.

- Surrogate of complex systems for uncertainty quantification and sensitivity analysis
- Complex systems: nonparametric, multivariate, computationally expensive

### Trade-off in Modeling by Regression

- Parametric regression requires a known model to "start with"
- Nonparametric regression is considerably more flexible at a cost of higher time and space complexity

# Objectives

### Design of a Nonlinear Regression Suite

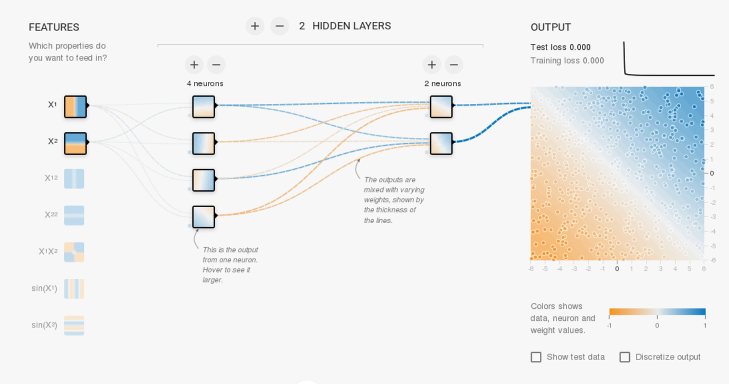
1. Implement an algorithm for performing nonparametric regression
2. Quantify the efficacy of algorithm design and parallelization

### Software Development

Incorporate concepts of software engineering

1. Version control
2. Efficient algorithm design
3. Unit testing and profiling
4. Code optimization

# Machine Learning and Multi-layer Perceptron



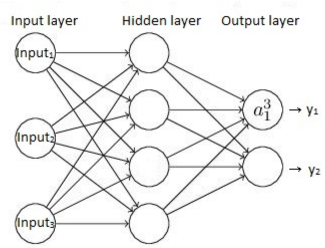

nodes, bias, weights, layers, and activation functions

# Principles of Supervised Machine Learning Nonparametric Regression

### Characterized by the use of nonlinear basis functions

Given a nonlinear activation function $f$ and a set of $M$ weights and inputs:

$$a_j^l = f\left(\sum_j^M w_{jk}^l \mathbf{a}_k^{l-1} + b_j^l\right)$$

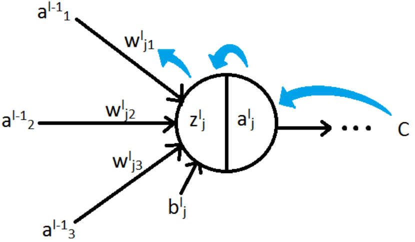


# Error Backpropagation

Minimizing the cost function using partial derivatives $\frac{\partial C}{\partial w}$, $\frac{\partial C}{\partial b}$

$$C = \frac{1}{2n} \sum_{x} \| y(\mathbf{x}) - a^L(\mathbf{x}) \|^2$$

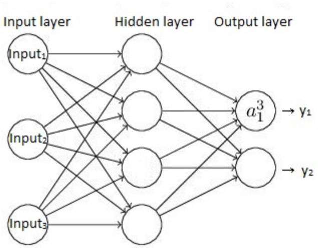
$$w_{jk}^{l}{}' = w_{jk}^l - \eta * \frac{\partial C}{\partial w_{jk}^l} = w_{jk}^l - \eta * a_k^{l-1} \sigma'\left(z_j^l\right) \frac{\partial C}{\partial a_j^l}$$



# Error Backpropagation

Minimizing the cost function using partial derivatives $\frac{\partial C}{\partial w}$, $\frac{\partial C}{\partial b}$

1. Feedforward: Calculate outputs for $l = 2, 3, ...L$.
2. Calculate the final output and the cost function using the expected output.
3. Backpropagate: Calculate $\Delta w^l_{jk}$ and $\Delta b^l_j$ for $l = L, L-1, ...2$.



# Format

## Based on C++ Programming Language

- Interfaced with CBLAS, linked with Intel's MKL
- Tested with IMPI (Windows), OpenMPI (CAEN login node), and MPICH(Ubuntu)
- User defined input specified and parsed with functions supplied by TinyXML.
- Provides weight matrix change, network structure summary, run time record, regression results.

./generate_xml.o

```xml
<?xml version="1.0" ?>
<Network>
    <!--Setup the network, claim the number of layers/nodes to
be used.-->
    <Setup numHiddenLayers="1" learningRate="0.25"
convergeCriteria="1e-07" maxIterationTime="2000000"
numTrainSet="100" shuffleTrain="Yes" numTestSet="50" />
    <InputLayer numNodes="1" biasnodeWeight="0.4" />
    <hiddenLayer numNodes="4" activationFunction="TANH"
biasnodeWeight="0.5" />
    <outputLayer numNodes="1" activationFunction="TANH" />
    <weightMatrix>0.155012,0.299553,0.850973,0.3</weightMatrix>
    <weightMatrix>0.567048,0.543467,0.264478,0.4</weightMatrix>
    <trainingDataFile dataFileName="basic_test_4" />
</Network>
```

```
#######Finish Training#######
After 10 times of iteration, the error is:
0.000262568
The weight matrices are:
0.0851309 0.539286 0.525529 0.142742 0.342204
0.0763125 0.651254 0.109263 0.285546 0.346315
0.0551452 0.189294 0.129016 0.126839 0.411862
The machine is trained by 100 training set.
Numbers of hidden layer:  1Number of nodes in each layer
4
The activation function used for each layer is (from the
first hidden layer to the output layer):
2 2
 #######Tests Start#######
0.3546740
```

# Algorithm Design

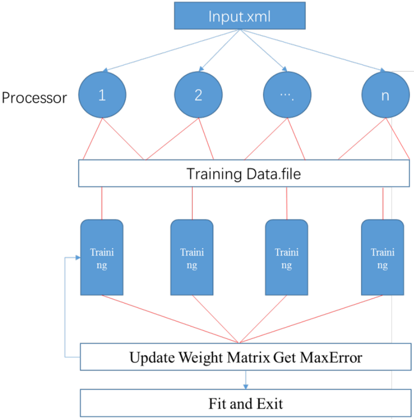
## Considerations

- Data class design: network structure flexibility
- Choice of learning rate: stability, convergence and accuracy
- Convergence criteria: weight matrix or global test results
- Loop structure: shuffle through all cases or stochastic training

## Implementation

- Shuffle training.✔
- Stochastic gradient descent.✔
- Learning rate modification.✔

# Parallelism

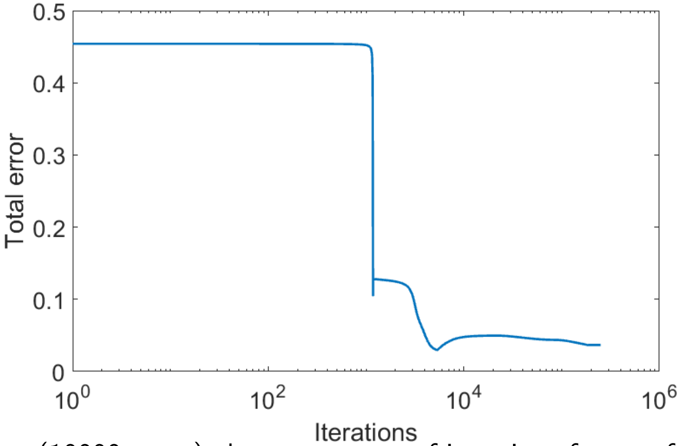



```
WHILE max_err > error_shreshold or nShuffle < total Shuffle
|   (For each processor)
|       RunTraing
|       Get the Weightbuffer and BiasNodes
|       Calculate the max_error
|   (Parallel Part)
|       Call MPI_AllReduce() for Weightbuffer and BiasNodes
|       Update the WeightMtx and BiaNodes
|     (For the master)
|       Call MPI_Gather() to gather the max_error
|       Call MPI_Bcast to sent the max_error to each slaver
ENDWHILE
```

# Results: Test Problem Description

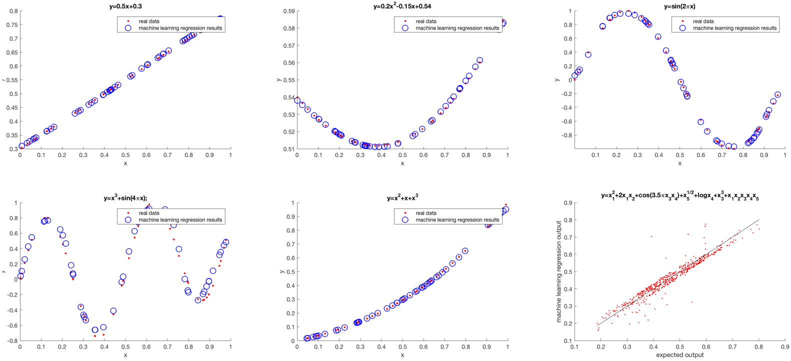
- Sample functions:
  - $y = 0.5x + 0.3$
  - $y = 0.2x^2 - 0.15x + 0.54$
  - $y = cos(2\pi x)$
  - $y = x^2 + sin(4\pi x)$
  - $y = x^3 + x^2 + x$
  - $y = x_1^2 + 2x_1x_2 + cos(3.5\pi x_3x_4) + \sqrt{x_5} + \log x_4 + x_3^3 + x_1x_2x_3x_4x_5$
- Data prepare:
  For each function,
  - generate random variables that are uniformly distributed in [0,1] as input values ($x_i$).
  - calculate their corresponding expected output values ($y$).
  - train the machine use some of the input-output pairs and test the machine using the input values which were not used during training.

# Results




Total error (10000 cases) change vs steps of iterations for test function 6

# Results




- Points plotted here contains only test expectation/results.
- Depends on the characteristic of these functions, network structure used for each of them may be slightly different from the others.

# Results

| Times of shuffle | Run time | test $R^2$ |
|---|---|---|
| 1 | 0.10s | 0.1491 |
| 100 | 30.97s | 0.6842 |
| 200 | 87.54s | 0.6898 |

Shuffle training results compare, performed with test function 6, 500 training set.

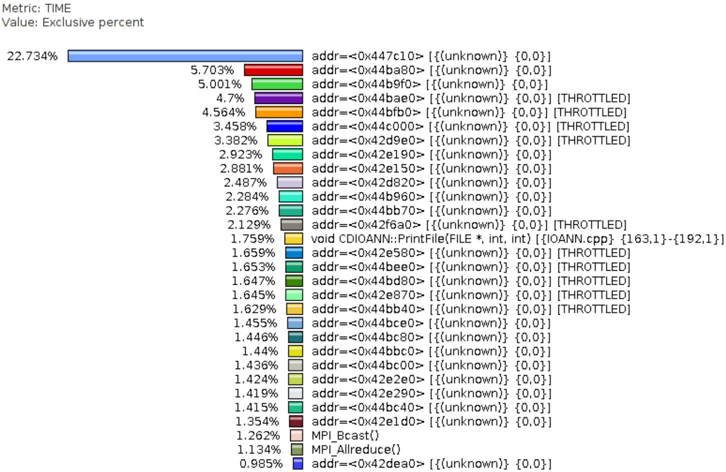
| function | Run time | network structure | Activation function | num training set | test $R^2$ |
|---|---|---|---|---|---|
| 1 | 43.46s | 1 hidden layer (3 nodes) | TanH | 50 | 0.9996 |
| 2 | 29.22s | 1 hidden layer (3 nodes) | TanH | 50 | 0.9977 |
| 3 | 28.72s | 1 hidden layer (3 nodes) | TanH | 50 | 0.9951 |
| 4 | 120.85s | 1 hidden layer (4 nodes) | TanH+Sinusoid | 100 | 0.9649 |
| 5 | 65.07s | 1 hidden layer (4 nodes) | TanH | 100 | 0.9987 |
| 6* | 3280.16s | 1 hidden layer (7 nodes) | TanH | 10000 | 0.9059 |

Stochastic gradient descent training results. *:trained in parallel, mpirun np=4
Run time recorded only for the training stage by clock_t in c++.

| | |
|---|---|
| function 6 training without mpirun**, mac | 19600s |
| function 6 training with mpirun**, CAEN linux 4 cores | 3280.16s |

**:These two tests were performed only to compare the run time between training with/without parallelism. The maximum iteration steps were limited to a small number.

# Profiling with TAU



# Documentation with doxygen

git clone https://git-ners590.aura.arc-ts.umich.edu/btwill/590_Project.git
branch:jiawei

UMRegress 0.9

Main Page | Related Pages | **Classes** | Files

Class List | Class Index | Class Hierarchy | Class Members

**Data Class Reference**

## Public Member Functions

| | |
|---|---|
| | **Data** (const double *inputs, int datalength, int activationFn=0, const double *outputs=NULL, const vector< **Layer** * > *layersVector=NULL) |
| const double * | **getInputs** () |
| void | **forwardPropagate** () |
| void | **calculateNodeError** () |

## Constructor & Destructor Documentation

```
Data::Data ( const double *              inputs,
             int                         datalength,
             int                         activationFn = 0,
             const double *              outputs = NULL,
             const vector< Layer * > *   layersVector = NULL
           )
```

Construct a data set with a vector of double precision numbers for: inputs data for the inputs outputs data for the outputs

If the data is a training set, outputs should not be input as NULL

# Summary

- The concepts of software development were implemented to design a nonlinear regression tool
- Algorithm design and MPI were implemented to accelerate the execution

## Future work

- Refine profiling results by using different TAU compile flag (-G).
- Organize the unit test files.
- Push our latest codes to gitLab.