[[1]](#footnote-1)

A Computational Framework for Efficient Implementation of Neural Networks on  
Multi-Core Machines

Wenduo Wang, *University of Michigan-Dearborn*Yi Murphey, *University of Michigan-Dearborn*

*Abstract*—This paper presents a computational framework, GPNN, for efficient implementation of Back-Propagation based neural learning algorithms running on multicore machines. GPNN has three components, parallelization of neural learning, abstraction of network components, and compile-time generalization. The parallelization component decomposes the back-propagation learning algorithm into two stages, forward propagation and back-propagation. During the forward propagation, training data are partitioned and distributed to K-threads, which run simultaneously multiple images of a neural network system on a multi-core machine. The back-propagation process, which runs on a single thread, collects the errors generated from the K-threads and uses it to update weights. The abstraction component models input, bias, weight, and neuron as abstract nodes and abstract layers. The compiler-time generalization component uses a generic programming technique to make neural network components instantiated at compiler time, which further reduces execution time. Together these computational components make GPNN an efficient framework for fast implementation of back-propagation based neural learning algorithms, and providing flexibility and reusability for modifying neural network topologies. The GPNN was applied to four different neural learning algorithms, classic back-propagation (BP), quick propagation (QP), resilient propagation (RP) and Levenberg-Marquardt (LM) algorithm. Experiments are conducted to evaluate the effectiveness of GPNN, and results show that the neural learning algorithms implemented in GPNN are more efficient than their respective functions provided by Matlab.

*Index Terms*—Back-Propagation, Neural Network, Parallel Computing, Multi-Core Machine, Generic Programming

# Introduction

Artificial neural networks are widely used in a wide range of research areas and shown being effective in solving many applications problems. In recent years, many applications face the challenges of discovering and extracting knowledge from large and complex data. However, as pointed by many researchers, our current machine learning methodologies and software tool are not sufficient to handle the volume, variability, and velocity of big data [FaB13, LaJ12]. Computational efficient algorithms are key areas of research in dealing with big data. The most noticeable software for processing big data is Apache Hadoop, a set of algorithms for distributed storage and distributed processing of very large data sets on computer clusters built from commodity hardware. Hadoop was initially designed to focus on running massive MapReduce jobs to process a web crawl. Recently Hadoop has been expanded to broad applications and ubiquitous usage, which has exposed some problems [VMD13]. There are also research activities focusing on fast implementation of neural learning algorithms such as FANN, OpenNN and tnnlib [7, 9], which take advantages of generic programming, algorithm and topology diversity and parallel execution. This research focuses on exploring the potential of using creative software technology to maximize parallelism in implementation of back-propagation based neural learning algorithms, and create a programming environment that is efficient for research in neural learning from big data, which has strong need for flexibility in making changes in neural network topology and optimizing learning parameters used in back-propagation based neural networks through repeated experiments on big data. Many important machine learning issues, for example, over-fitting, network size, memory space need for the end product, require the analysis of data generated from extensive experiments.

In this paper, we present a computational framework, GPNN, designed for efficient implementation of the back-propagation based neural networks. The GPNN consists of three computational components, parallelization of neural learning,abstraction of neural network components, and compile-time generalization. The parallelization component decomposes the back-propagation learning algorithm into two stages, forward propagation and back-propagation. During the forward propagation path, the training data are partitioned and distributed to threads, which all have identical images of the neural network. The errors generated from the threads are combined and used for weight update through the back-propagation, which is running on a single thread. This parallelization technique can improve training time considerably when training data are big. The abstraction component provides techniques for representing neural network components input, bias, weight, neuron and target in abstract nodes and abstract layers. The abstraction process provides an efficient architecture for implementing parallel computation in neural learning. For example since weights are the only data needed in forward propagation and, under the abstraction process, they are represented in separate abstract layers, they can be easily copied or shared among multiple threads as network images. The remaining parts of the network, i.e. input nodes, biases, neurons and target nodes, can stay local without being distributed over the multi-cores. Therefore the abstraction process helps to reduce communication time and required memory size, which could be significant improvement over efficiency when processing big data using large systems. The Compile-Time Generalization component uses generic programming technique to instantiate data types such as learning algorithms, transfer functions, error functions and network topologies at compiler time, which eliminates the run-time used in each loop to determine the running type of these objects. Together these computational components in GPNN provides mechanisms for fast implementation of back-propagation based neural learning algorithms, flexibility and reusability for neural network research in studying neural network topologies and optimizing learning parameters.

We have implemented four neural learning algorithms using the GPNN framework, classic back-propagation (BP) [1], quick propagation (QP) [2], resilient propagation (RP) [3] and Levenberg-Marquardt (LM) algorithms [4]. Experiments are conducted to evaluate the effectiveness of GPNN by training these learning algorithm to solve a real-world application problem, predicting humidity based on weather data such as visibility, wind direction, dew point pressure, etc. The run-time complexity of these algorithms are compared with their respective algorithms provided by Matlab.

This paper is organized as follows. Section II introduces the three computational components in GPNN, Section III presents the experiment results and Section IV concludes the paper.

# A computational framework for Implementing Back-propagation based Neural Learning Algorithms

The proposed computational framework, GPNN, is developed to optimize the implementation of back-propagation based neural networks using creative software techniques. Back-Propagation is the most popular neural learning algorithm for supervised learning in multi-layered feed-forward networks as well as in many recurrent neural networks. Most of these neural networks have a unique forward path and back-propagation path. The forward path is used to propagate the input from the input layer to the output layer, which can be represented as follows,

where is the outputs from the neuron in previous layer, which is regarded as the inputs of neuron , is the weight from all previous neuron to neuron , is calculated as the weighted sum of all the inputs, and is the output after filtered by the transfer function , which is also regarded as one of the input of next layer.

The backward path follows gradient descent calculated by the chain rule,

If the error function is chosen as the mean squared error in batch training mode, then it has the following expression,

where contributes to each input training pattern, represents the total number of training pattern, is target, and is predicted output (same as the output of last layer). The gradient can be calculated as follows,

For every hidden neuron, its gradient is affected by all of its successors. To consistently express the gradient of output nodes and hidden nodes, let

where contributes to next layer.

In software implementation, a classic neural network layer is normally modeled as a container, which holds a weight matrix, a bias node and multiple neurons. Additionally, it specifies the propagation order of these components. In a global view, layer can be connected with each other.

The following introduces the three major components in GPNN, parallelization of neural learning, abstraction of neural network components, and compile-time generalization. We will also show, using an example, the flexibility of reconstruct a new neural network from an existing one in the GPNN framework.

## Parallelization of Neural Learning Algorithm

According to Moore’s law, the density of circuits doubling at every new generation [5]. In the recent years, computer systems have increased the number of cores significantly to support parallel computing. Particularly in big data applications, it is important to implement neural learning algorithms in high level of parallelization on multi-core CPUs with shared memory to achieve significant increases in CPU performance. We propose an approach of multi-thread implementation of batch training BP [6], which is illustrated in Fig. 1. It is a -thread parallel computing structure, in which a neural network is copied times, and the training data are partitioned and distributed to these threads. Therefor the inputs are processed simultaneously along the feed-forward paths of these K threads, and errors are collected at the end of each thread. Then the errors at all threads are combined for weight update using gradient descent (back-propagation path), which is represented as follows,

Weights are then all updated through the back-propagation process. The updated neural network is made copies, and the same training process is repeated until user defined stop criteria are satisfied.

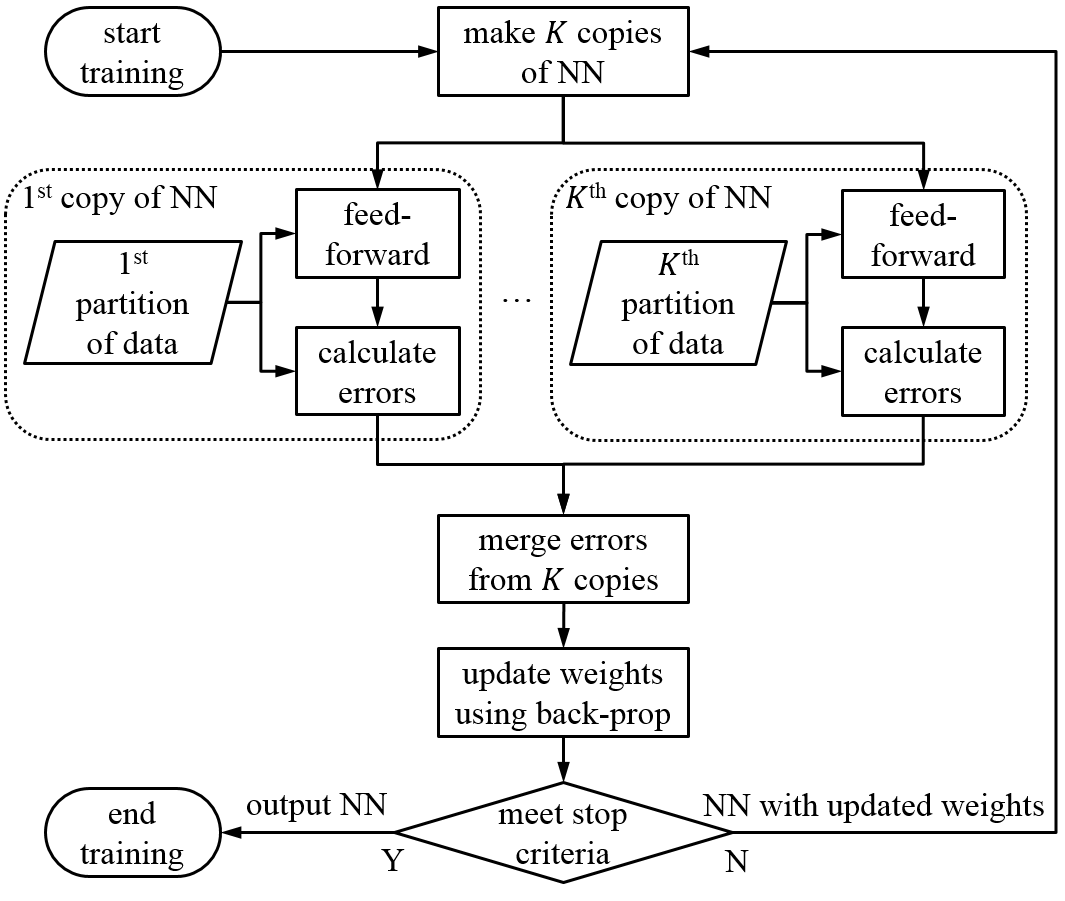


Fig. 1. Illustration of multi-thread implementation of neural learning process.

## Abstraction of neural network components

Abstraction is a very critical and powerful concept in object-oriented programming. It treats objects with similar functions as the same module. Based on this concept, weights between neurons can be considered as similar to neurons, since a neuron has one input axon and one output axon, and a weight connecting two neurons can also be considered to have an input, which is the output axon of the preceding neuron, and the weight can be considered as its output axon, which is connected to the input axon of following neuron. The transfer function of a weight is defined as follows,

|  |  |
| --- | --- |
| feed-forward: |  |
| back-propagation: |  |

For the same reason, an input node to a neural network can also be treated as similar to a neuron, which has the equal number of output axons to the first hidden layer but no input axon. Similarly, bias of a neuron layer can also be handled in this way. There is no need for back-propagation from any of these input nodes.

|  |  |
| --- | --- |
| feed-forward for input: |  |
| feed-forward for bias: |  |

where is the value of input feature, is the output of the input node, and the output of the bias node is always .

The output of a neuron is represented as a target node, which has one input axon connected to the output of a neural network, but no output axon, i.e there is no feed-forward from a target node.

|  |  |
| --- | --- |
| Target node: |  |

where is the target value of this output node, and is the network layer predicted value. Target nodes are used in back propagation process.

Based on the above discussion, input, bias, weight, neuron and target are all considered as nodes in an abstraction context. Nodes of the same type are grouped into the same abstraction layer, and these abstracted layers are connected to each other as illustrated in Fig. 2, in which a one-layer of neurons in a neural network is represented by input abs-layer (abstraction layer), bias abs-layer, weight abs-layer, neuron abs-layer and target node abs-layer. The connection among the nodes and the abstracted layer are considered equivalent in the context of programming.

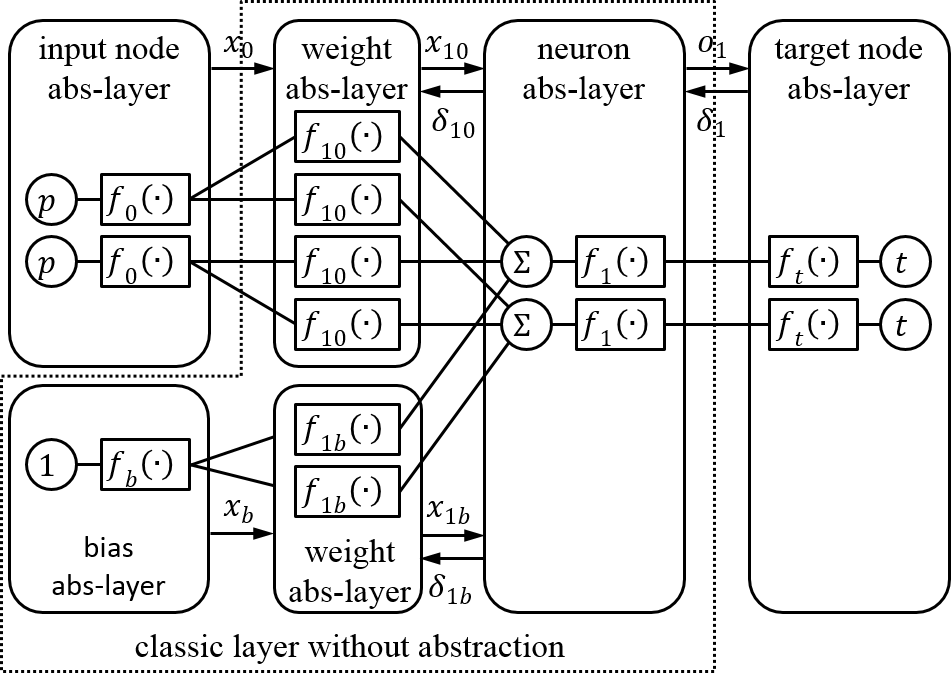


Fig. 2. One-layer of neural network topology represented by the abstraction layers (abs-layer). The abstracted layers inside the box in dash-line is the neural network layer.

The abstraction process provides an efficient architecture for implementing parallel computation in neural learning, since only weights need to be copied or shared among multiple threads as network images. In the abstraction process, weights are represented in separate abstract layers so they can be easily copies or shared without touching the other parts of the network, e.g., input nodes, biases, neurons and target nodes, which can stay local without being distributed over the multi-cores. Therefore the abstraction approach can reduce communication cost, and reduce the need for memory. This improvement could be significant in applications involving neural learning from big data.

Current input and output of a node are stored for each input pattern for weight update processing during the back-propagation. Other values are prepared during feed-forward or back-propagation for intermediate calculation of critical variables, e.g., and shown in Fig. 3.

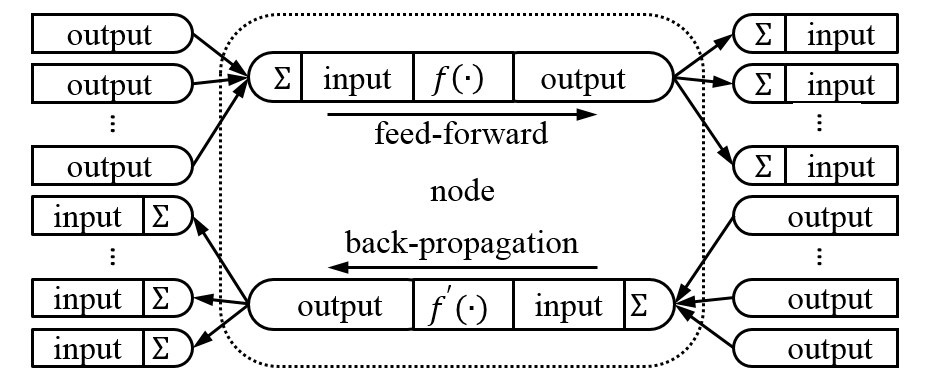


Fig. 3. Communication between nodes during feed-forward and back-propagation.

## Compile-Time Generalization

Generic programming is a software approach that can be used to generalize replaceable functional nodes in neural networks. In a generic programming architecture variables can be written in terms of types to-be-specified-later [7], and then instantiated when needed. In neural networks, transfer functions, network topologies, and weight update functions can be type-deduced so functional call can be determined during compile time [8].

We proposed to implement weights using compile-time generalization, since they used in different data types including forward, backward, update and copy processes. User can specify an appropriate update strategy of weights during programming without modifying rest part of the code. The compiler then generate a made-to-order target file related to the developer customized behavior data types. In addition, since the software is tailored to specific behavior data types at compiling period, this implementation greatly reduces irrelevant code, thus, reduces the software size. The compile-time generalization of weights eliminates the run-time otherwise needed in every loop to determine the running type of weight object through looking up its virtual table.

Fig. 4 illustrates the differences between run-time and compile-time implementation of a neural learning process. In the run-time generation implementation, weight type is deduced in every epoch, which means that the processors need to spend time on deciding weight type in each loop through looking up a virtual table. As a consequence, the accumulative time consumed in all loops is conspicuous. If the compile-time generalization implemented, data type deduction is calculated only once at the compiling process. Moreover, if the neural network software is required to be run multiple times for different experimental purposes, run-time generalization time on type deduction can be more significant. Contrarily, because there’s no need to compile the same neural network software again for repeated experiments, the compile-time generalization does not need extra time for data type deduction. In summary, the time complexity for run-time generalization of weights is , which can be saved if data type deduction for weights are carried out at the compiling process.

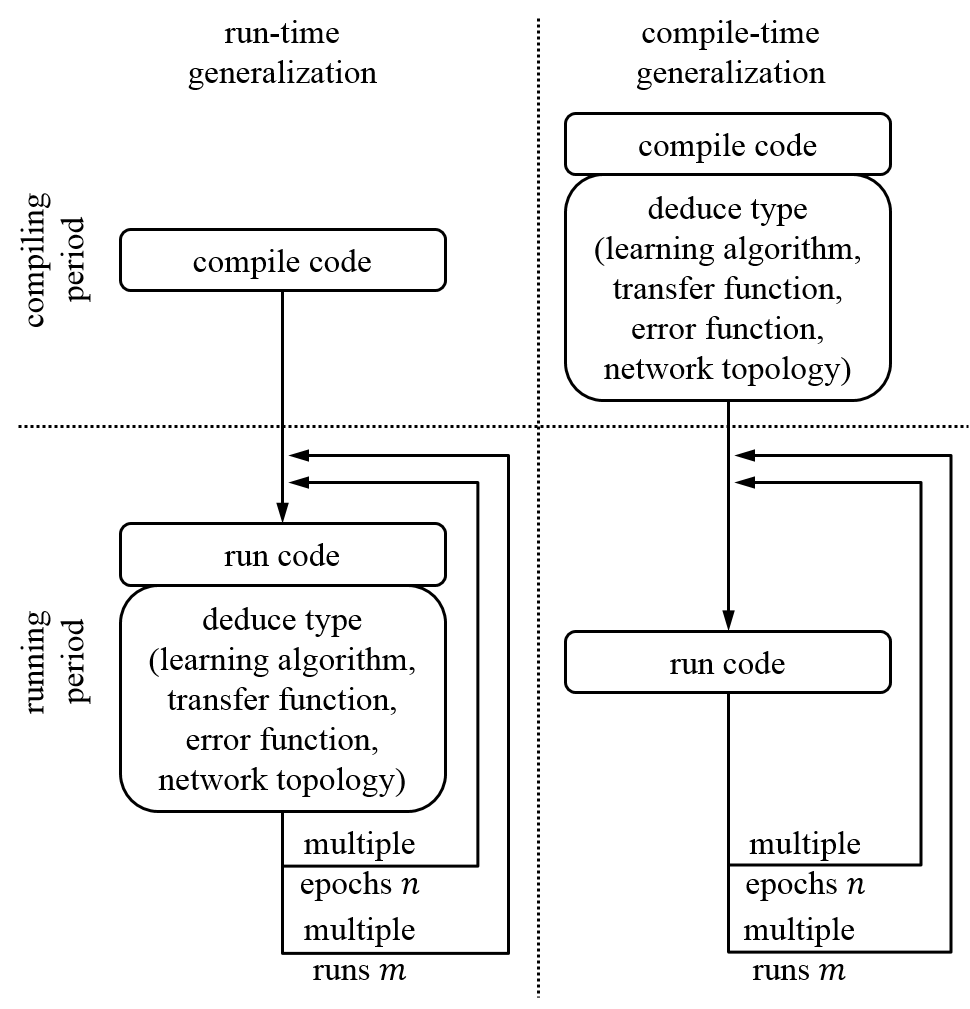


Fig. 4. Run-time generalization versus compile-time generalization during the processes of compiling code to running code for a neural learning algorithm, where is epoch number and is number of experiment repeated.

In terms of design pattern, the compile-time generalization approach is also known as policy based class design [9]. In our implementation, a learning algorithm is defined as a type of update policy, a transfer function is defined as a type of transfer policy, a network topology is defined as a kind of topology policy, an error function is defined as a kind of error policy, and even the number of neurons, input, target and other training factors can be considered as individual policies as well.

## Flexibility and Reusability of GPNN framework

In addition to the fast implementation of neural learning algorithms, the proposed GPNN framework also provides flexibility and reusability for neural network research. The abstraction and compile-time generalization processes in GPNN allow researchers to build new neural networks by simply connecting or pruning nodes, without the need of re-constructing the most part of an existing network architecture. The following algorithm shows the steps to build a recurrent neural network, which has no bias, and uses LM algorithm and log-sigmoid transfer function, based on an existing biased one-layer neural network, which uses the BP algorithm and a linear transfer function. Fig. 5 illustrates the algorithm steps.

*Building a recurrent neural network,*

1. Detach the bias abstracted layer and the weight abstracted layer in the given neural network
2. Attach a neuron abstracted layer using linear transfer function, and attach a weight abstracted layer using LM algorithm.
3. Replace the algorithm type of the weight abstracted layer from BP to LM, and replace the transfer function type of the neuron abstracted layer from log-sigmoid to linear.

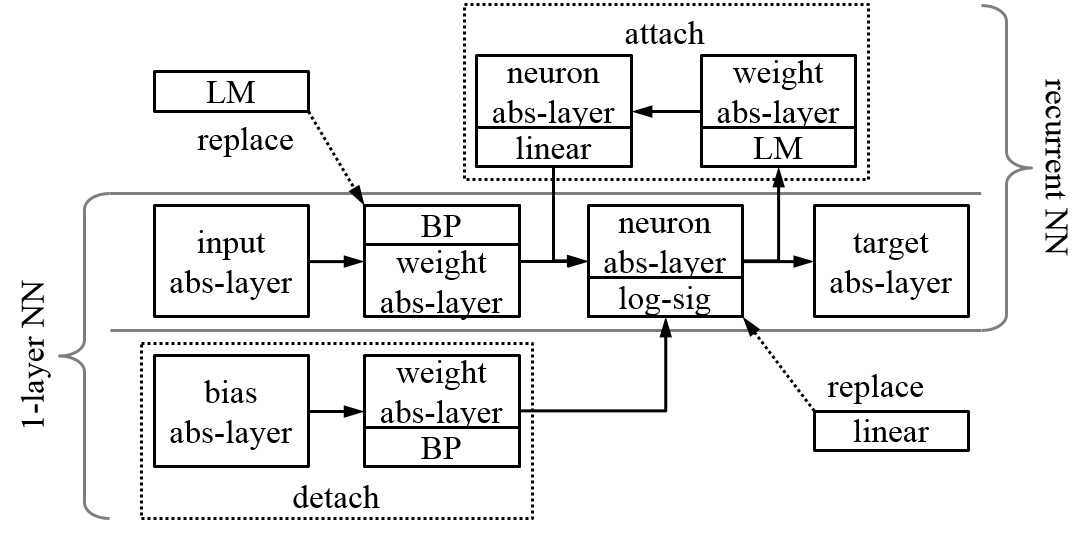


Fig. 5. Building a recurrent neural network by modifying a one-layer neural network.

# Performance and Experiment Result

We applied the proposed computational framework, GPNN, to four back-propagation based neural learning algorithms, classic back-propagation (BP), quick propagation (QP), resilient propagation (RP) and Levenberg-Marquardt (LM) algorithm.

The QP takes the largest steps possible to reach local minima without overshooting. The QP learning algorithm is based on the assumption that the error versus weight curve can be approximated by a parabola whose arms open upward [2], which means its second derivative is approximately linear with a positive slope . For a parabola curve, the minimum value is where its second derivative equals to 0. QP is efficient for large training data.

The basic idea of RP is that every time the gradient changes its sign, it indicates that the last update was so big that the error function skipped a local minimum. Thus, the weight update absolute value needs to be reduced by a factor , where . On the other hand, if the gradient remains the same sign as in the previous update, a larger weight update step can be taken, e.g. increasing weight change by a factor of , where . The detail of the RP algorithm can be found in [3].

The LM algorithm aims at solving non-linear least square problem. It interpolates between the Gauss–Newton algorithm (GNA) and the method of gradient descent. The LM algorithm is more robust than the GNA [4].

We implemented these four neural learning algorithms using the parallelization, abstraction and compiler-time generalization processes discussed in Section II.

In order to evaluate the performance of the proposed GPNN, we applied the following experiment data to the four neural learning algorithms implemented using GPNN. Experiment data are selected from hourly historical climate data of Ann Arbor, MI, USA and downloaded from [www.wunderground.com](http://www.wunderground.com) website from year 2010 to 2013. The data from year 2010 to 2012 were used as training data and year 2013 as testing data. The total number of training samples is 26304, and total number testing samples is 8760. The training samples are partitioned evenly and distributed to all the threads, the number of which is determined by the configuration of each experiment. Ten input features listed in TABLE I are input to each neural network system. The neural network output is the predicted humidity value. All input and output are normalized to zero mean () and unit standard deviation (

TABLE I  
Features in Climate Dataset

|  |  |  |  |
| --- | --- | --- | --- |
| Usage | Feature | Valid Range | Unit |
| input | month | 1~12 | - |
| hour | 0~23 | - |
| temperature | -50~150 |  |
| dew point | -50~150 |  |
| pressure | 28~31 | inHg |
| visibility | 0~10 | mile |
| wind direction | 0~359 |  |
| wind speed | 0~50 | mph |
| gust speed | 0~100 | mph |
| precipitation | 0~1.5 | in |
| target | humidity | 0~100 | % |

A computer with a 2.3GHz quad-core 8-thread CPU, 8G RAM, and a 64-bit operating system, was used to run all the experiments presented below.

## Evaluating Multi-Thread Efficiency

Theoretically, multiple processors and cores can support almost any number of threads running simultaneously. However, since the communication between threads is usually implemented by a pooling approach, it takes certain amount of time to synchronize all the image threads to main network thread. It is clear that the most efficient number of threads should be equal to the number of cores, since running time of multi-threads on the same core will add up more than the running time of single thread even with thread scheduling applied.

The first experiment was conducted to evaluate different number of threads applied to different number of hidden nodes while BP was implemented using GPNN as a neural learning algorithm. Fig. 6 shows the training time for these different neural network configurations running on different number of threads. In the experiments the maximum epoch is set to 2000, and every experiment was repeated 10 times and the average training time was presented.

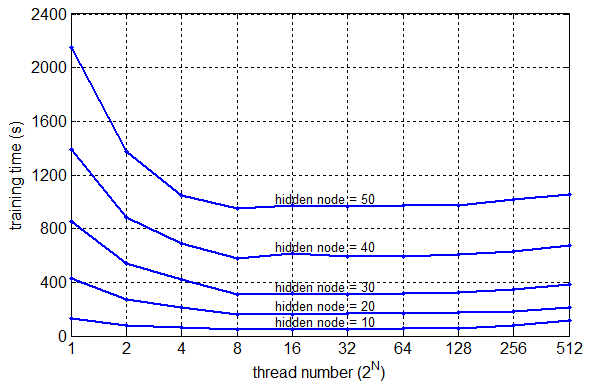


Fig. 6. Training time of BP for different hidden node numbers versus different numbers of threads.

The result demonstrates that the fastest thread configuration is indeed 8 threads, which equals to the hardware concurrency of CPU in the computer. It is about 3 times () faster than the training using a single thread to train a 10 hidden nodes network. As the number of threads grows larger than 8, the training time increases, which is due to the time cost for scheduling.

## Comparing GPNN Implementation with Conventional implementation

In this experiment, we compare the efficiency of the neural learning algorithms implemented using GPNN with their respective Matlab versions, except for the RP algorithm, which is not available in Matlab. In this experiment, the stop criterion is also 2000 epochs, every experiment was repeated 10 times and the mean training time is presented in TABLE II.

TABLE II  
Training Time at Different Thread Number

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Training Time (s) | BP | QP | RP | LM |
| 8-thread GPNN | 56.8 | 69.7 | 62.4 | 263.9 |
| 1-thread GPNN | 176.7 | 192.3 | 167.9 | 337.7 |
| 1-thread Matlab | 649.1 | 678.1 | - | 1264.0 |

It is no surprise that the 8-thread implementations of BP, QP and RP neural learning algorithms are much more computationally efficient than their respective 1-thread implementations. It is noted that the LM algorithm is more time consuming than the other three neural learning algorithms. This is because LM algorithm needs to calculate Hessian matrix inversion frequently during each iteration [10]. Even the 8-thread implementation of LM algorithm does not reduce execution time much, which is due to the added time cost for communication among threads. In the 1-thread implementation, the algorithms implemented in GPNN are much more efficient than their respective functions provided by Matlab, the speed up for the GPNN implemented BP is 3.6, the GPNN implemented QP is 3.5 times, and the GPNN LM is 3.7.

## Algorithm Efficiency based on Dynamic Stopping Criteria

In many practices, neural network training could be terminated based on a number of dynamic criteria, for example, gradient approximates to 0, gradient stops decrease, and etc. In this experiment, we train the GPNN implemented BP, QP, RP and LM learning algorithms using the following stopping criteria: when the mean error is less than 10-9 or error has not changed for 6 epochs [11]. The configuration of the four neural networks are shown in TABLE III. Each experiment is repeated 10 times and the average CPU times are presented in TABLE IV.

TABLE I  
Neural Network Configuration  
used in Algorithm Efficiency analysis Experiments

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | BP | QP | RP | LM |
| thread number | 8 | 8 | 8 | 8 |
| learning rate | 0.5 | 0.5 | - | - |
| max grow factor | - | 1.75 | - | - |
| increase factor | - | - | 1.2 | - |
| decrease factor | - | - | 0.5 | - |
| initial update value | - | - | 0.1 | - |
|  | - | - | - | 10 |
|  | - | - | - | 10 |
|  | | | | |

TABLE   
Training Time of Neural Learning Algorithms  
with Dynamic Stopping Criteria

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | BP | QP | RP | LM |
| training time (s) | 3.5 | 35.1 | 28.2 | 31.1 |
| converge epoch | 43.2 | 976.4 | 866.7 | 97.2 |
| training error (%) | 3.80 | 1.86 | 0.73 | 3.15 |
| testing error (%) | 3.66 | 1.88 | 0.76 | 3.08 |

The results show that LM algorithm converges in significantly less epochs than QP and RP. However, it takes more time to run per epoch. Although RP takes more epochs to converge, its training and testing errors are distinctly smaller than others. The BP has the least training time and the least number of epochs, but its training and test error are the highest. Based on these results we can deduce that if an application requires high accuracy, GPNN implemented RP algorithm is a good fit to find the local minima, and if big data are involved, LMA is a good learning algorithm to use since it converges in the least number of epochs.

# Conclusion

We have presented a computational framework, GPNN, for efficient implementation of back-propagation based neural learning algorithms for big data learning. In GPNN, the forward path of a neural network learning algorithm is distributed to K-threads, errors are then combined, and the weights updated by a back-propagation process running in a single thread. The abstraction component in GPNN implements the abstractions of weights, neurons, transfer functions, input and output in terms of abstraction nodes, and organize the nodes into abstraction layers according to node types. The abstraction process provides an efficient computational environment for parallel computation during neural learning. The GPNN also introduced a compile-time generalization technique to make as many neural network components being instantiated at compiler time as possible, which further reduces the execution time.

Experiments are conducted to evaluate the effectiveness of multi-thread implementation of GPNN implemented BP algorithm, and the comparison of computational time efficiency among the four neural learning algorithms, GP, QP, RP and LM, implemented using GPNN and the respective functions provided by Matlab library. The experiment results show that the algorithms implemented in GPNN are more efficient than their respective functions provided by Matlab. Specifically, the speed up for the GPNN implemented BP is 3.6, the GPNN implemented QP is 3.5 times, and the GPNN LM is 3.7. When implemented in 8-thread parallelization with a fixed number of epochs as the training stopping criterion, the GPNN BP has the largest speed up, 3.1, over 1-thread GPNN BP, and LM algorithm has the least speed up, 1.3. When the multiple stopping criteria of minimum error and error convergence are used, the results show that (1) although GPNN RP takes more epochs to converge, its training and testing errors are decidedly smaller than all others, (2) GPNN LM converges in less epochs than GPNN QP and GPNN RP, but it takes more time to run each epoch, and (3) the GPNN BP has the least training time, but its training and test error are the larger than all the other three.

Based on these results we conclude that GPNN RP algorithm is good to find minimum error, and GPNN LM is a good learning algorithm for big data applications.

In addition to efficient run-time, the proposed GPNN framework also provides flexibility and reusability for neural network research. The abstraction and compile-time generalization processes in GPNN allow researchers to build new neural networks by simply connecting or pruning nodes without re-design the most part of an existing network architecture.

We have put the four neural learning algorithms implemented using the GPNN in a library available under LGPL license at <https://www.github.com/wenduow/BeefNet>.

References

1. M. Bodén, “A guide to recurrent neural networks and backpropagation,” *The Dallas Project, SICS Technical Report*, 2002.
2. S. E. Fahlman, “An empirical study of learning speed in back-propagation networks,” *Technical Report*, CMU-CS-88-162, 1988.
3. M. Riedmiller and H. Braun, "A direct adaptive method for faster backpropagation learning: the RPROP algorithm," *IEEE International Conference on Neural Networks*, vol.1, 1993, pp.586-591.
4. M. T. Hagan and M. B. Menhaj, "Training feedforward networks with the Marquardt algorithm," *IEEE Transactions on Neural Networks*, vol.5, no.6, Nov 1994, pp.989-993.
5. C. Chu, S. K. Kim, Y. A. Lin, et al., “Map-reduce for machine learning on multicore,” *Advances in Neural Information Processing Systems*, 2007.
6. O. Schuessler and D. Loyola, “Parallel training of artificial neural networks using multithreaded and multicore CPUs,” *Adaptive and Natural Computing Algorithms*, Springer Berlin Heidelberg, 2011, pp.70-79.
7. Wikipedia, “Generic programming, ” Available: <http://en.wikipedia.org/wiki/Generic_programming>
8. A. Alexandrescu, “Preface,” in *Modern C++ Design: Generic Programming and Design Patterns Applied*, Feb 2001, pp.xi.
9. A. Alexandrescu, “Policy based class design,” in *Modern C++ Design: Generic Programming and Design Patterns Applied*, Feb 2001, pp.2-18.
10. M. T. Hagan, H. B. Demuth and M. Beale, “Levenberg-Marquardt Training” in *Neural Network Design*, Boston, MA: PWS Publishing, 1996, pp.12.1-12.16.
11. MathWorks, “Train and apply multilayer neural networks, ” Available: <http://www.mathworks.com/help/nnet/ug/train-and-apply-multilayer-neural-networks.html>

1. W. Wang is with the Electrical and Computer Engineering Department of University of Michigan-Dearborn. (e-mail: [wenduow@umich.edu](mailto:wenduow@umich.edu))

   Y. Murphey is a professor at the ECE department at the University of Michigan-Dearborn. (e-mail: [yilu@umich.edu](mailto:yilu@umich.edu)) [↑](#footnote-ref-1)