[[1]](#footnote-1)

A Computational Framework for Efficient Implementation of Neural Networks on Multicore 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 network training on multithread and multicore machines. GPNN has three components, parallelization of training process, abstraction modeling of network components, and generic programming to make neural network systems to change architectural configurations. The penalization component distributes training data to multicores to calculate errors, which are then summarized and used for weight update based on gradient descent the multithread programming to take advantage of a multicore computer architecture. The abstraction component models input, bias, weight, and neuron all as nodes, which make the training process in a parallel computer architecture much more efficient. The generic programming component will make a neural network system easy to change its configurations and easily transported among different operating systems or computer hardware systems. The GPNN was applied to four different neural learning algorithms, Classic Back-Propagation (BP), Quick Propagation (QP), Resilient Propagation (RP) and Levenberg-Marquardt Algorithm (LM), and experiments are conducted to evaluate the efficiency of these neural network algorithms implemented under the GPNN framework.

*Index Terms*—Algorithm, Back-Propagation, Neural Network, Parallel Computing, Generic Programming

# Introduction

A

rtificial neural networks are used in wide range of research and applications over past three decades. The network topology and propagation algorithms often vary with different application scenarios. Researchers usually spend much effort and time struggling to try different network architectures, applying them to training data, and analyzing training results. Many important issues, for example, over-fitting, network size, memory space need to be considered across the whole training procedure. This may shifts researchers from their original topics to too much network reliability considerations.

Benefited from generic programming, researchers can easily configure their own neural networks or try among different configurations through a design pattern, as known as the policy pattern, which makes everything instantiable modules. This library currently provides four types of networks, which are 1, 2, 3-hidden layer networks and recurrent network [1], four types of weight update algorithms, classic back-propagation (BP), quick propagation (QP) [2], resilient propagation (RP) [3] and Levenberg-Marquardt algorithms (LM) [4]. Besides these build-in learning models, researchers can connect or prune perceptron (using neuron instead of perceptron in rest of the article for convenience) and weights to customize any type of network topology.

With the increase number of cores or processors, appropriate parallelization and data partition can maximum training speed. The GPNN library provides the interface for fast local file access using memory map and an interface for Map-Reduce application. All of these operations and inner data flows during training are implemented on stack memory to avoid wasting time on dynamic allocation and access. The only restriction of network size depends on the stack pre-allocation of compiler, which normally can be adjusted by compilers.

Different from previous works, for example, FANN, OpenNN and tnnlib, which take only partial advantages of generic programming, algorithm and topology diversity or parallel execution ability (see Appendix), the GPNN library possesses all of these attractive characteristics.

# A computational framework for Implementing Neural Learning Algorithms on Multicore Machines

The proposed computational framework consists of three major components, parallelization of training process using a multithread scheme, abstraction of network components, and general programming on transfer functions, error functions and network topologies. First we present a formal description of neural network structures and back-propagation neural learning algorithm from the perspective of software implementation.

In software implementation development, 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.

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 the neural networks have a unique forward path.

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 chain rule.

If error function is chosen as the mean squared error in batch training mode,

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.

## Parallelization

According to Moore’s law, the density of circuits doubling at every new generation [5]. In the recent years, computer systems have increased number of cores to support parallel computing. In big data applications, it is important to implement neural learning algorithms in high level of parallelization on multicore CPUs with shared memory to achieve significant increases in CPU performance. We propose an approach of multithread implementation of batch training BP[6]. The approach is illustrated in Figure 1, where it is shown in a K-thread parallel computing structure, where an neural network is copied K times, and the training data are distributed to these K threads. The input are processed.

Simultaneously along the feed-forward paths of these K threads, and errors are calculated at the end of each thread. Then the errors at all K threads are combined for weight update using gradient descent (back-propagation path), which is represented as follows,

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



Change to:

Calculate errors at all nodes through back propagation

Change to:

Network weights using back-propogation

Change to:

Make K copies of neural network

Change to:

Merge error

Fig. 1. Illustration of multithread implementation of neural learning process.

## Abstraction of Weight, Neuron, Bias, Input and Target

Abstraction is a very critical and powerful concept in object-oriented programming which means to abstract as objects of similar functions to 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 that 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 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 back-propagation for 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 . [I am here]

Contrarily, a target node can have one input axon connected to the output of a neural network but no output axon. There is no feed-forward for target node.

|  |  |
| --- | --- |
| back-propagation: |  |

where is the target value of this output node, and is the network predicted value.

To sum up above abstraction, input, bias, weight, neuron and target will be aliased as node in following context. Each kind of node will be grouped by an abstracted layer and these abstracted layer will be connected to each other instead of directly connecting classic layer illustrated in Fig. 4. The connection among nodes and abstracted layer therefore can be equivalently looked upon while programming.

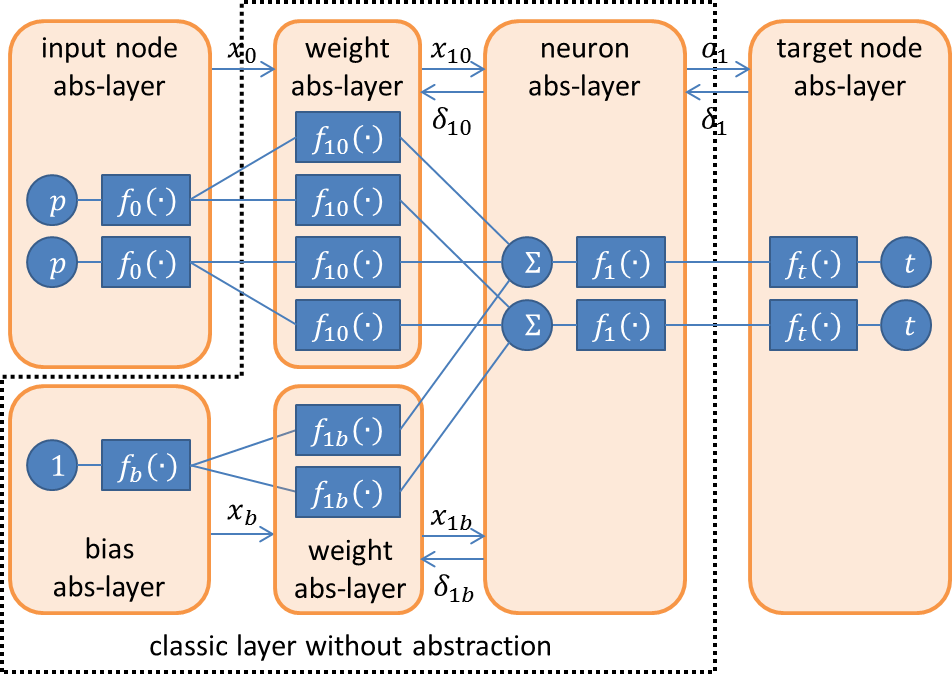


Fig. . 1-layer neural network topology with abstracted layer (abs-layer) interpretation. The abstracted layers inside the dash-line compose a classic layer.

Considering the convenience provided for implementing parallel computing algorithm, only weight abstracted layers need to be copied or shared among network images. The rest part of the network, such as input nodes, biases, neurons and target nodes, can remain local on distributed systems. This brings less communicational memory cost, which highly correlated to time consumption in parallelized implementation, will improve multithread efficiency.

Current input and output of a node will be stored for each input pattern in order to provide any convenience in processing weight update algorithm. Other values are prepared during feed-forward or back-propagation for intermediate calculation of critical variables, for example, and .

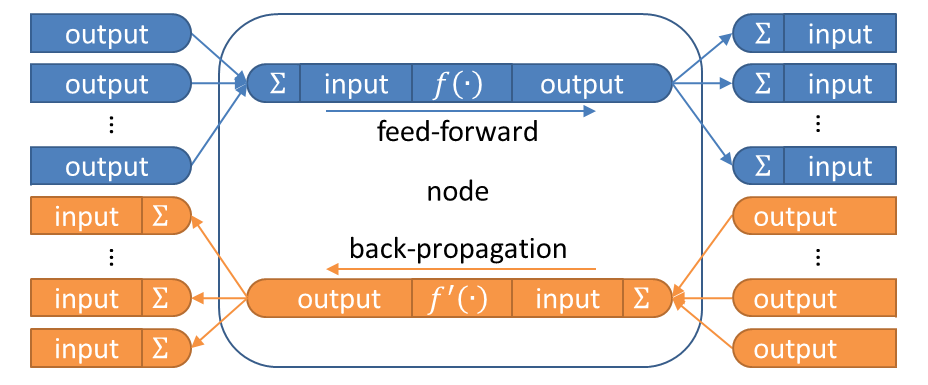


Fig. . A microscopic view of node. It connects the outputs and inputs of other nodes.

## Compile-Time Generalization to Learning Algorithms, Transfer Functions, Error Functions and Network Topologies

Generic programming is one of the best implementation approaches to generalize any type of replaceable functional node in neural networks, in which architecture is written in terms of types to-be-specified-later [7] that are then instantiated when needed for specific types provided as parameters. Thanks to template mechanism in C++, it is a good candidate for coping with combinatorial behavior data types, which refers to neural learning algorithms, transfer functions, network topologies, error functions and other training factors here because behavior data types can be deduced statically during compiling period [8]. This compile-time generalization technique avoids extra time consumption during each loop to determine the running type of an object through looking up its virtual table, which a run-time generalization usually implements through. For example, weight will provide forward, backward, update, copy and merge interfaces. User can easily specify an appropriate update strategy of weight during programming without modifying rest part of the code. The compiler will compile a made-to-order target file related to developer customized behavior data types. In addition, since the software is tailored with specific behavior data types at compiling period, it greatly reduces irrelevant code to be compiled, thus, reduces software size.

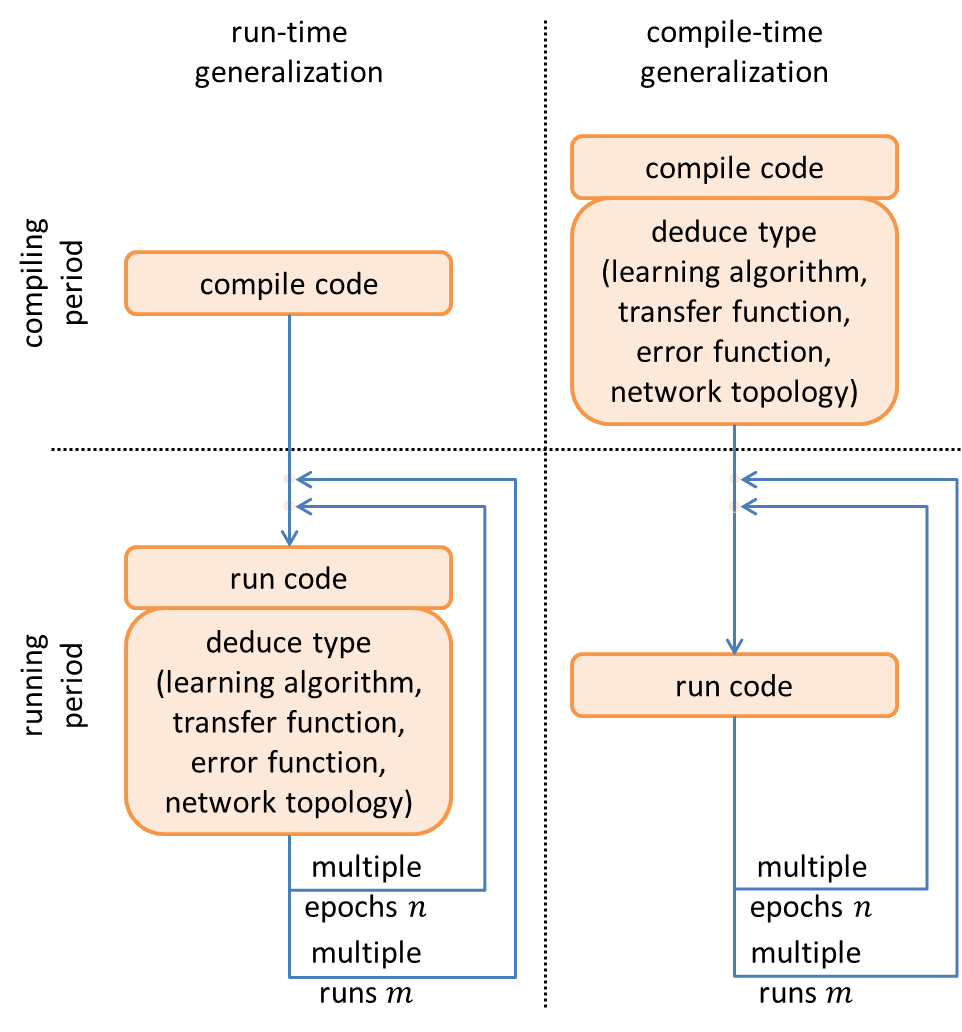


Fig. . Run-time generalization versus compile-time generalization from compiling code to running code. is epoch number and is experiment running times.

As illustrated in above figure, weight type will be deduced in every epoch during run-time generalization, which means that the processors spend time on deciding weight type in each loop through looking up virtual table. As a consequence, the accumulative time consumption from all loops is conspicuous. However, the compile-time generalization only deduces type onetime at compiling period. There is no extra time consumption during each loop. Moreover, if the neural networks software is required to be run multiple times for different experimental purposes, run-time generalization cannot avoid taking time on type deduction during each loop and each run. Contrarily, because there’s no need to compile the same neural network software again for these ex

periments, the compile-time generalization could absolutely get rid of running time consumption. To sum up, the time complexity for run-time generalization is , but there’s no extra cost while running after compile-time generalization.

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

## Scalability and Reusability

Scalability is an important measurement of a software library. A high scalable software library could provide freedom space for further development or maintenance. Benefit from abstraction and compile-time generalization, researchers can easily customize their own neural networks by simply connecting or pruning nodes without re-design the most part of the network architecture. For example, one would like to implement a recurrent neural network without bias using LM algorithm and log-sigmoid transfer function based on an existing biased 1-layer neural network using BP algorithm and linear transfer function. It is simple to modify following 3 steps.

1. Detach a bias abstracted layer and a weight abstracted layer.
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.

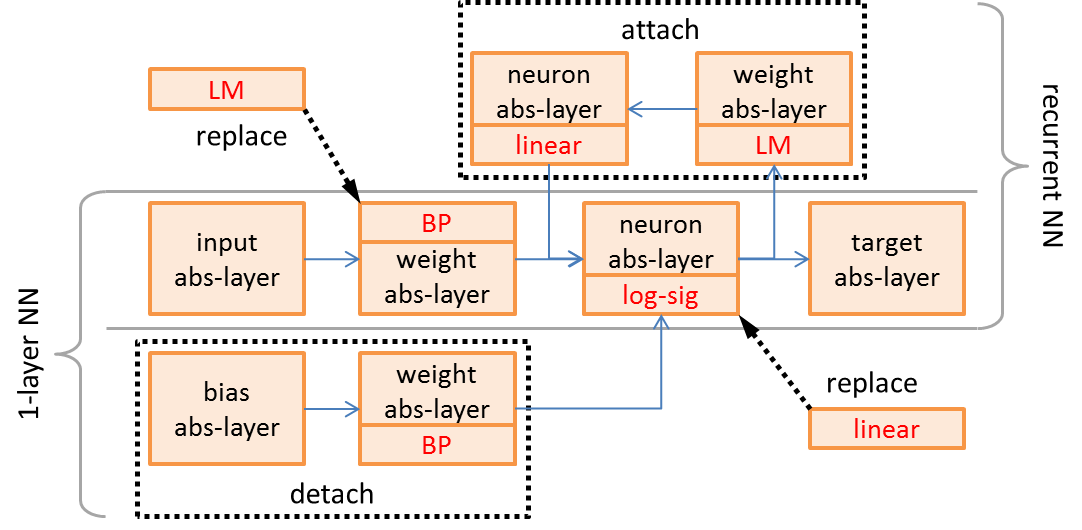


Fig. . Modification from a 1-layer neural network to a recurrent neural network.

Under the application of template programming and policy pattern, multiple potential scalable possibilities are listed in TABLE I, The library also emphasizes on the reusability of code applied to as many as future peripherals. Benefit from abstraction, there’s no need to replicate each function in further development.

TABLE   
Scalable Possibilities

|  |  |  |
| --- | --- | --- |
| Generic Data Type | Implemented | Pending Implementation |
| neural learning algorithm | BP, QP, RP, LM | Quasi-Newton, adaptive learning,  conjugate algorithm,  momentum, etc. |
| transfer function | log-sig, tan-sig, linear | asymmetric,  saturated, etc. |
| error function | MAE, MSE, RMSE | similarity,  distance, etc. |
| network topology | 1,2,3-layer, recurrent | n-layer,  Kohonen, etc. |
| input and output utility | neural network | control system,  deep network,  fuzzy system,  decision tree, etc. |

# Application with Neural Learning Algorithms

Current multithread architecture can be applied together with several neural learning algorithms reviewed as following.

## Classic Back-Propagation (BP)

By selecting appropriate learning rate , the update equation of neuron from epoch to can be obtained. (neuron index and will be omitted in following equations except specified.)

In this weight update rule, learning rate is a fixed value, which scales weight update steps [3]. If it’s too small, more epochs need to be taken to reach local minima, if it’s too large, the error could oscillate or even diverge.

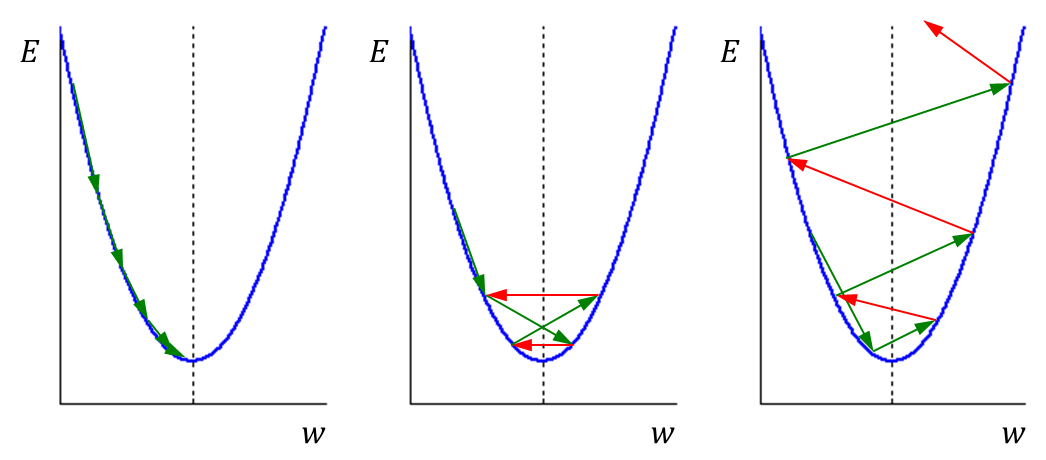


Fig. . Possible weight update trends, includes convergence (left), oscillation (middle) and divergence (right). The solid curve represents error vs. weight, local minimum is at the intersection between the solid curve and the dash line, red arrows represent weight update with positive gradient, and green arrows represent weight update with negative gradient.

To avoid complicate choices among learning rates, some local adaptive learning algorithms have been developed.

## Quick Propagation (QP)

The target of quick-propagation is to take the largest steps possible to local minima without overshooting. The basic idea is to directly jump to a local minimum closely enough. Risky assumption is made as the error versus weight curve for each weight can be approximated by a parabola whose arms open upward [2], which means its second derivative is approximate a line with positive slope . For a parabola curve, the minimum value is where its second derivative equals to 0.

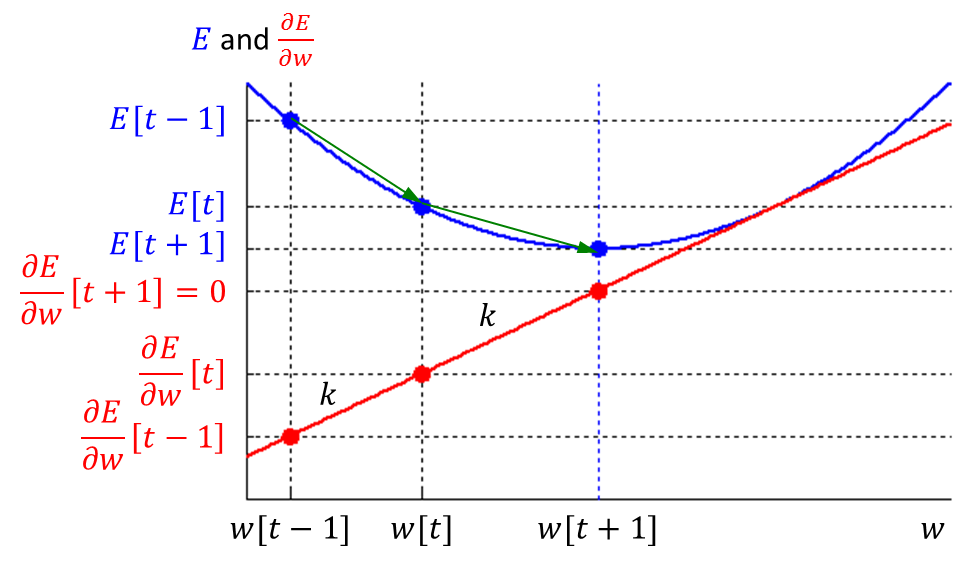


Fig. . Gradient (blue solid parabola) and its first derivative (red solid line). Minimum error is reached at which the first derivative equals to zero.

Considering the slope equation of both previous weight update and current weight update, it is a second-order method [2].

According to above equation, if is approximate the same as , will reach an infinite value, which leads to an infinite step or towards a local maximum. To restrain weight change, a maximum growth factor is defined in order that no weight step is allowed to be greater in magnitude than times the previous step. A fit-to-all value of [2].

## Resilient Propagation (RP)

The basic idea of resilient propagation is that every time the gradient changes its sign, which indicates the last update was so big that jumped over a local minimum. Thus, the weight update absolute value needs to be reduced by factor , where . Contrarily, if the gradient remains the same sign as previous, a larger step of can be increased by factor , where . The algorithm can be implemented in following approach [3].

|  |
| --- |
| for all weights of neurons and bias  {  if  {      }  else if  {        }  else  {    }      store current state variables to next epoch  } |

Intuitively, the algorithm makes it confident for the weights updated to reach local minima.

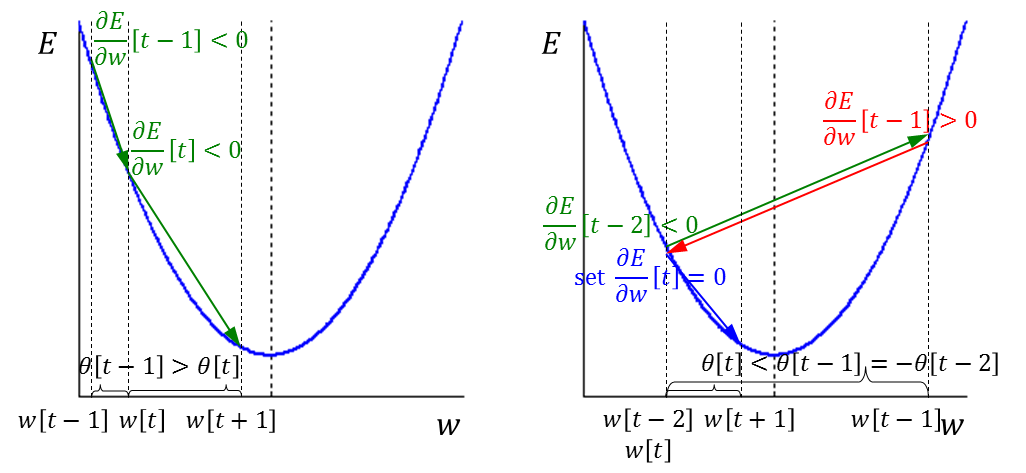


Fig. . When gradient doesn’t change its sign (left), weight takes a larger step by a ratio to update. When gradient changes its sign (right), weight doesn’t update at this epoch but will take a smaller step by a ratio to update at next epoch.

Concluded after some experiments [3], slight variation of or will neither improve nor deteriorate convergence time. These two factors are fixed to and . As a similar consequence, the initial value of all is set to 0.1.

## Levenberg-Marquardt Algorithm (LM)

Mathematically, Levenberg-Marquardt algorithm aims at solving out non-linear least square problem. It is much more efficient than other techniques applied to a neural network no more than hundreds of weights, even if the computation requirements are higher than other algorithms within iterations [4].

Gauss-Newton’s method gives out an update to the weight,

where is the Hessian matrix and is the gradient. The Hessian matrix can be approximated by,

and the gradient vector can be calculated as,

where is the Jacobian matrix with as the network output number and as the weight number connected to a single neuron.

Thus, the Gauss-Newton’s method updates weights as,

Levenberg-Marquardt algorithm modifies above equation to ensure the inversion of matrix always exists by,

where is a changeable damping parameter. will increase if squared error increases, will decrease if squared error reduced, that is,

A good try for initial value of could be 0.01 and the factor could be 10. In addition, if squared error increase, weight will be reverted to previous value.

# Performance and Experiment Result

We applied the proposed computational framework, GPNet, to four back-propagation based neural learning algorithms, Classic Back-Propagation (BP), Quick Propagation (QP), Resilient Propagation (RP) and Levenberg-Marquardt Algorithm (LM). The quick-propagation is to take the largest steps possible to local minima without overshooting. The basic idea is to directly jump to a local minimum closely enough. Risky assumption is made as the error versus weight curve for each weight can be approximated by a parabola whose arms open upward [2], which means its second derivative is approximate a line with positive slope . For a parabola curve, the minimum value is where its second derivative equals to 0.

…..

parallelization, abstraction and generalization. In order to evaluate the performance of such techniques, several experiments have been conducted by controlling variables. All following experiments are running on a 2.3GHz quad-core 8-thread CPU with 8G RAM machine installing 64-bit operating system.

Experiment data is selected from hourly historical climate data of Ann Arbor, MI, USA downloaded from [www.wunderground.com](http://www.wunderground.com) website from year 2010 to 2013, year 2010 to 2012 as training samples and year 2013 as testing samples. So total number of training samples is 26304, total number testing samples is 8760. Training samples are averagely distributed on each thread of which the number is determined by the configuration of each experiment. 10 input features listed in table II are fed into each experiment system and the humidity value is the only output. All of them are normalized to zero mean () and unit standard deviation ().[what does your system do? Need to explain what is system output. Why it is chosen in your experiment.]

TABLE   
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 | % |

## Multithread Efficiency

Theoretically, multiple processors and cores can simulate almost any number of threads running simultaneously regardless of very large system specified limit. However, the communication between threads is usually implemented by a pooling approach. As a result, it will consume certain amount of time to synchronize all the image threads to main network thread. Intuitively, the most efficient number of threads should be equal to the number of cores, since running time of multithreads on the same core will add up to no less than the running time of single thread even though any kind of thread scheduling applied.

Here is the experiment result using the same network configuration as shown in table IV and same amount of data but different numbers of threads doubled from 1 to . BP algorithm is used in this experiment, maximum epoch is set to 2000 and algorithm coefficients will not affect the execution time.

TABLE   
Training Time for Different Hidden Node Number  
using Different Thread Number

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| training time (s) | | hidden node number | | | | |
| 10 | 20 | 30 | 40 | 50 |
| thread number () | 1 | 132 | 428 | 850 | 1390 | 2148 |
| 2 | 80 | 269 | 539 | 883 | 1373 |
| 4 | 62 | 212 | 420 | 693 | 1045 |
| 8 | 46 | 161 | 311 | 576 | 953 |
| 16 | 49 | 163 | 316 | 614 | 973 |
| 32 | 50 | 165 | 313 | 590 | 965 |
| 64 | 53 | 165 | 315 | 595 | 976 |
| 128 | 59 | 172 | 323 | 608 | 975 |
| 256 | 75 | 186 | 344 | 630 | 1018 |
| 512 | 119 | 214 | 381 | 676 | 1054 |

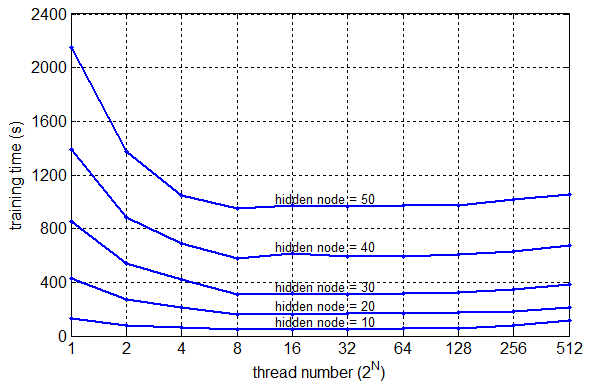


Fig. . Training time for different hidden node number configuration versus different numbers of threads. Curves from bottom to top corresponding to the network hidden node number equals to 10, 20, 30, 40 and 50.

The result demonstrates that the fastest thread configuration number start from 8, which equals to the thread number and hardware concurrency of CPU in this case. It is about 3-time () faster than training with single thread training a 10 hidden nodes network, but it is not able to reach an ideal 8-time because of communication problem mentioned before. With the increase number of threads more than 8, training time slightly goes up because of scheduling problem mentioned before.

TABLE   
Neural Network Configuration  
for Multithread Efficiency Experiment

|  |  |
| --- | --- |
| Affected Parameter | Configuration |
| input node number | 10 |
| hidden layer number | 2 |
| hidden node number | 10, 20, …, 50 |
| hidden layer transfer function | log-sigmoid |
| output node number | 1 |
| output layer transfer function | linear |
| neural learning algorithm | BP |
| stop early | no |

## Algorithm Complexity

For the same network structure, different algorithm will take various number of CPU instructions to execute, thus it greatly affect training time among different algorithms. In this experiment, maximum epoch is set to 2000 and algorithm coefficients in table VI will not affect the execution time. A baseline is set under the help of Matlab Neural Networks Toolbox with the same network parameters set. Another 10 measurements are taken independently from previous experiment, mean values are presented and standard deviation is presented as reliability.

TABLE   
Training Time at Different Thread Number

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Training Time (s) | BP | QP | QP | 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 |

Results in TABLE V show that the LM algorithm takes far more time to be trained than other algorithms. This is because Hessian matrix inversion needs to be calculated frequently even during each iteration [10]. The speed is gained by second-order approximation to the number of weights. Multithread technique does not accelerate the training process of LM algorithm because the time consumption of communication among threads is now a less distinct factor compared to that of matrix inversion.

TABLE   
Neural Network Configuration  
for Algorithm Complexity Experiment

|  |  |
| --- | --- |
| Affected Parameter | Configuration |
| thread number | 1 and 8 |
| neural learning algorithm | BP, RP, QP and LM |
| Other parameters are same as in TABLE III. | |

## Algorithms Efficiency

In reality, it is unnecessary to complete a training process up to specified maximum epoch. Training could be terminated if certain criteria meet, for example, sum of error or gradient approximates to 0, gradient doesn’t distinctly decrease for several epochs. Training time, converge epochs, converge error and at the termination are three measurements of efficiency among different back-propagation algorithms. All developed algorithms participate in the comparison experiment here as configured in table VIII. As soon as the mean absolute value of gradient is less than 10-9 or not changing for 6 epochs [11], training will stop which means that error converges. Another 10 measurements are taken independently from previous experiments and the average values are presented and standard deviation is presented as reliability.

TABLE   
Converge Epochs and Errors of Different Algorithms

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 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 in table VII show that LM converges faster than QP and RP, especially LM. However, it takes relatively more time to run because of matrix inversion time consumption discussed before. Although RP takes more epochs to converge, its training and testing errors are distinctly smaller than others’.

According to this experiment and under the application scenario, if high accuracy is required for prediction or classification, RP algorithm is a good fit to deep dig out the local minima. If big dataset applied and there’s no specific accuracy requirement, BP and QP algorithm could be applied. If no more than hundreds of hidden nodes are needed, LM could be efficient as well.

TABLE   
Neural Network Configuration  
for Algorithm Efficiency Experiment

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 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 |
| Other parameters are same as in TABLE V. | | | | |

# Conclusion

Based on the architecture and design pattern of this neural networks library, the author developed multiple weight update algorithms one by one after reviewing different back-propagation techniques without greatly modifying other components inside the architecture. In terms of this, the library could be regarded as easily extendable, especially facing the circumstance that future algorithms being developed continuously.

The author created this library under the consideration of both abstraction and generalization. It produces benefits if a network structure is not symmetric, which means user can customize the topology by pruning or branching the connections during programming.

The multithreaded architecture could easily be embedded into any kind of distributed systems by calling few functions the library provides.

The Multithreaded Neural Networks Template Library GPNN (BeefNet) under LGPL license is available at

<https://www.github.com/wenduow/BeefNet>

Appendix

TABLE   
Neural Network Characteristics

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Supports | FANN | OpenNN | tnnlib | GPNN |
| parallel computing interface | N | N | N | Y |
| neural learning algorithm diversity | Y | Y | Y | Y |
| generic programming (strong scalability) | N | N | Y | Y |

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. ([yilu@umich.edu](mailto:yilu@umich.edu)) [↑](#footnote-ref-1)