[[1]](#footnote-1)

A Multi-Threaded Neural Networks Template Library with Back-Propagation Algorithms

Wenduo Wang, *University of Michigan-Dearborn*

*Abstract*—This report is related to a newly developed neural network programming library (BeefNet), which take the advantage of multi-thread in multicore machine and the comparison among different learning algorithms. In past three decades, learning algorithms for neural network have been evolving to be more accurate in searching for local minima between error function versus weight following gradient descent rules. On the other side, parallel computing is getting mature with state-of-the-arts distributed architectures such as Cloud Computing, Map-Reduce, etc. This brings an opportunity to reduce time consumption for learning algorithms. BeefNet library takes the advantage of generic programming in choosing various network configurations and makes it flexible in being transplanted among different operating systems or architectures.

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

# Introduction

A

rtificial neural networks are widely used in research and application over past three decades. The network topology and propagation algorithms are evolving in order to adapt with different application scenarios. Researchers usually spend much time struggling on preparing network architectures and waiting for training results. Huge numbers of other aspects, 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 4 types of networks, which are 1, 2, 3-hidden layer networks and recurrent network [1], 4 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 perceptrons (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 BeefNet library provides the interface for fast local file access using memory map and a potential 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.

# Propagation Algorithms

Back-Propagation is the most popular algorithm for supervised learning not only applied in multi-layered feed-forward networks but also in recurrent 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 any 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.

## 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.

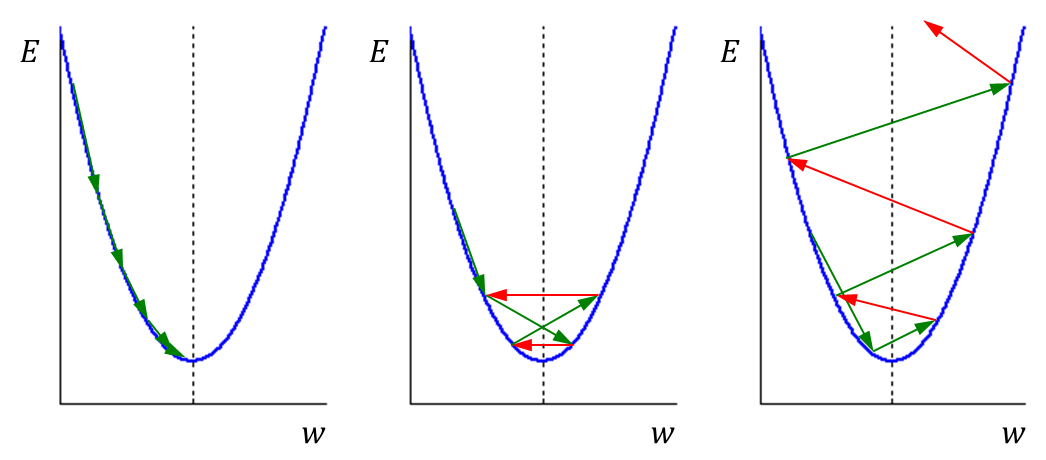


Fig. 1. Possible weight update trends, includes convergence (left), oscillation (middle) and divergence (right). The solid curve represents error vs. weight, local minimum is at the intersection between the solid curve and the dash line, red arrows represent weight update with positive gradient, and 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.

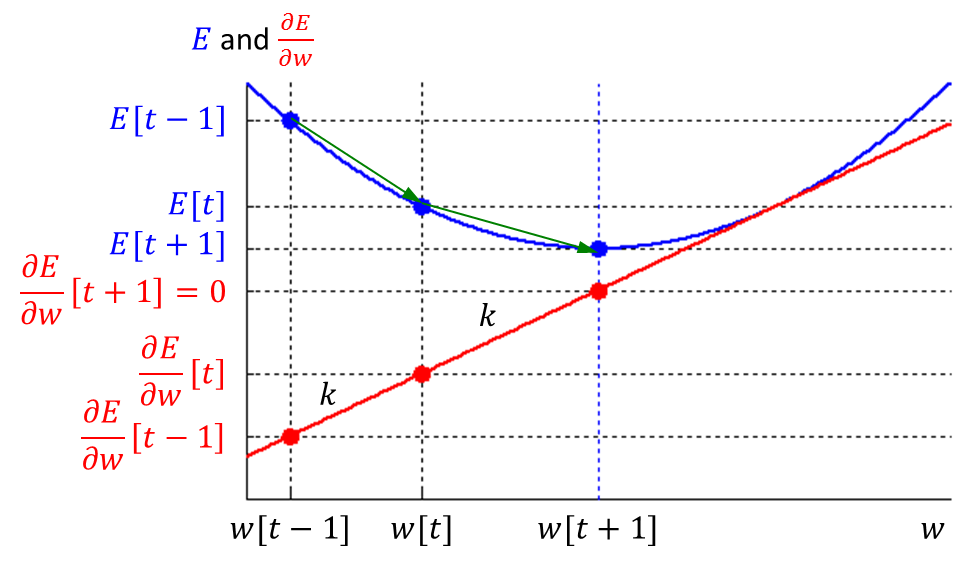


Fig. 2. Gradient (blue solid parabola) and its first derivative (red solid line). Minimum error is reached at which the first derivative equals to zero.

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.

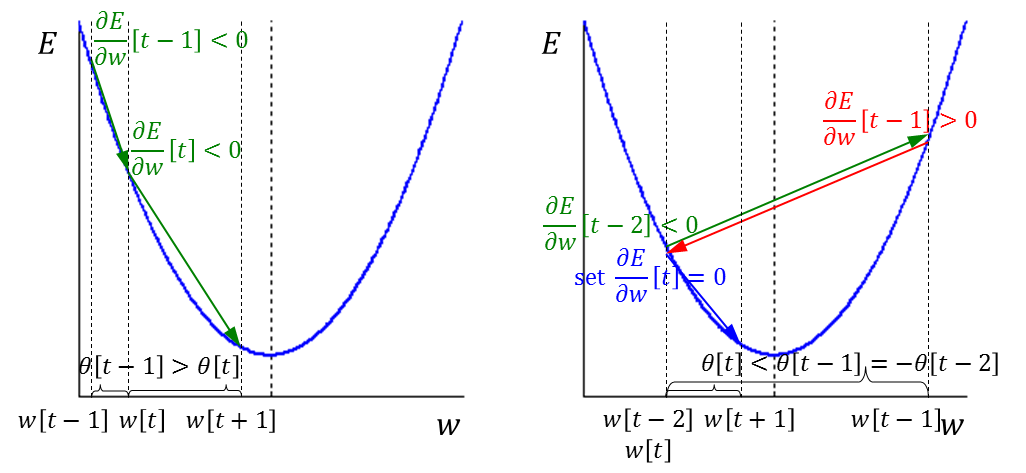


Fig. 3. When gradient doesn’t change its sign (left), weight takes a larger step by a ratio to update. When gradient changes its sign (right), weight doesn’t update at this epoch but will take a smaller step by a ratio to update at next epoch.

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].

Theoretically, if a weight is updated by a very small value, the network function can be approximately described as following.

The purpose is to minimize the mean squared error after weight being updated. Thus, error function is defined as,

Chain rule applied to the calculation of ,

Levenberg and Marquardt’s contribution is to modified above equation in order to make a larger movement along the direction where gradient is smaller.

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.

# Parallelization

According to Moore’s law, the density of circuits doubling every generation [5]. With the increase number of processing cores on a fixed size chip and fixed frequency, if the algorithm can be parallelized, its processing speed can be theoretically doubled. As an application, neural networks running on multithreaded and multicore CPUs with shared memory is the architecture of obtaining significant increases in CPU performance, especially for very large training datasets. The most common approach of parallelization is applied while training in batch mode, that is assigning a part of training dataset to each thread and train them simultaneously [6]. Weight will be updated after all threads finish.

Reference to Map-Reduce architecture, the assigning procedure can be regarded as a map operation. To ensure consistent functionality among different mappers (copy of images), which requires the forward path of these parallel images should output same result whichever a single input pattern fed in, all weights in training network need to be copied to network images during map operation.

On the other bank, a reducer sums up all weight changes after gradient descent (backward path) in each network images.

Finally, weights will be updated in training network by choosing an appropriate algorithm. The whole procedure will be recursively conducted until satisfying user defined stop criteria exerted on the training network.

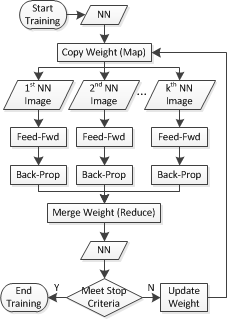


Fig. 4. Multi-Thread training procedure is a modification of traditional batch mode training. Weights will copy to several network images. Each image will feed-forward its training patterns and back-propagate gradients. Gradients of different network images will then merged together to be updated. The whole runs in a loop until stop criteria is met.

# Library Implementation

On the software development perspective, a neuron network layer can be regarded as a constraint, a virtual unit. It only specifies the propagation order of neurons in a global view. There’s no layer instance in this library, thus it is described as virtual. Neurons within a virtual layer can be operated in any order, whereas, neurons grouped by different virtual layers should be operated following specified order.

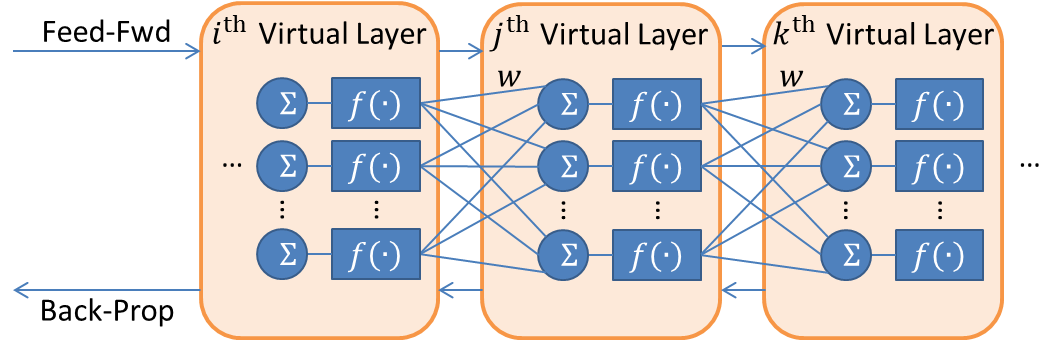


Fig. 5. Neurons are grouped by three virtual layers. Input patterns will be fed-forward with the order of th, th then th layer regardless of the order inside each of them. Gradients will be back-propagated with the order of th, th then th layer regardless of the order inside each of them. Weights can be updated regardless of any restriction.

## Abstraction

Abstraction is a very critical and powerful concept in object-oriented programming which means to abstract as much objects, whose has similar functions as possible. According to this motivation, weight can also be considered as similar as neuron which has only one input axon and one output axon, i.e., the input axon of a weight connects the output axon of previous neuron, and the output axon of a weight is connected by the input axon of next neuron. The transfer function of a weight is defined as follows.

As the same reason, an input feature to a neural network can also be treated similar as a neuron, whose has equal number of output axons to the first hidden layer but no input axon. What’s more, bias of a virtual layer can also be handled in this way.

Contrarily, an output feature can have one input axon connected to the output of a neural network but no output axon.

where is the target value of this output feature.

To sum up above abstraction, input, bias, weight, neuron and target will be aliased as node in following context. The connection among nodes therefore can be equivalently looked upon while programming.

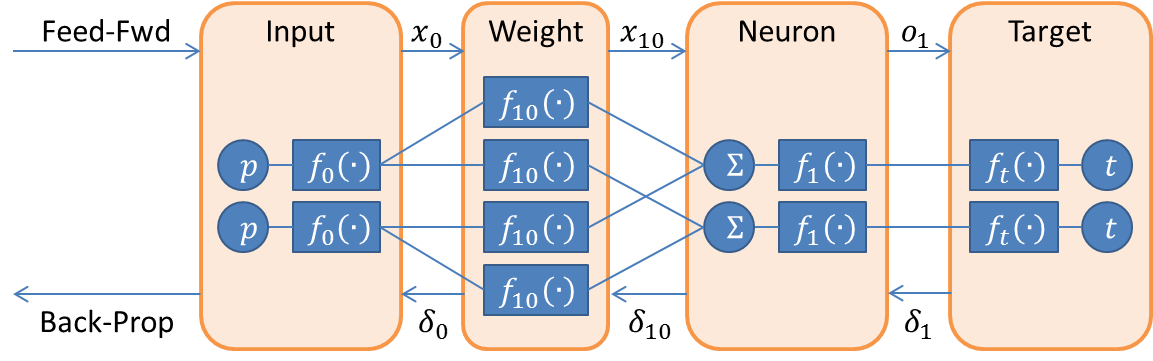


Fig. 6. Topology among nodes will helps further gets rid of the concept of layer, so as to simplify programming considerations.

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. Value is prepared for intermediate calculated variable, for example, or .

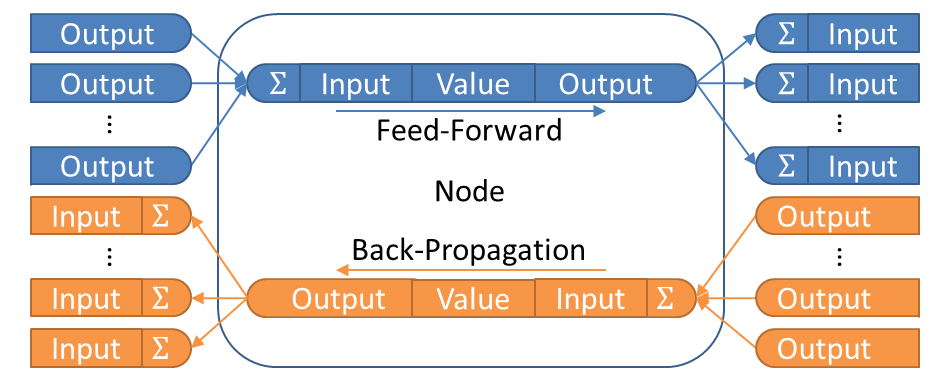


Fig. 7. A microscopic view of node, it connects to others’ outputs and inputs in above way.

## Generalization

Generic programming is one of the best implementation approaches to generalize any type of replaceable functional unit 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 behaviors, which corresponding to algorithms, neuron numbers, and error functions here, because behaviors can be deduced statically during compiling period [8]. This technique avoids extra time consumption during each loop to determine the running type of an object through looking up its virtual table. For example, weight will provide forward, backward, update, map and reduce interfaces. User can easily specify an appropriate update strategy of weight during coding without modifying rest part of the code. The compiler will compile a made-to-order target file related to customized weight.

In terms of design pattern, this generalization approach is as known as policy based class design [9]. In the library implementation, each update algorithm is defined as a kind of update policy, each error function is defined as a kind of error policy, even the number of neurons, input, target can also be considered as policies.

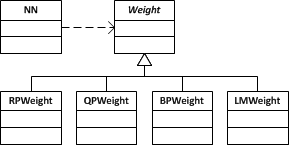


Fig. 8. Generalization UML example of weight component in a neural network class, RPWeight, QPWeight, BPWeight, LMWeight are different behaviors (algorithms) which could be replaced before compiling period.

# Performance and Experiment Result

Briefly sum up previous techniques, this library implements three back-propagation algorithms, 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 2001 to 2013, year 2001 to 2010 as training samples and year 2011 to 2013 as testing samples. So total number of training samples is 87648, total number testing samples is 26304. Input features include hour of day, temperature, dew point, pressure, visibility, wind direction, wind speed and precipitation. The target feature is relative humidity. All features are normalized into .

## Multi-Thread 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 multi-threads 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 and same amount of data but different numbers of threads doubled from 1 to .

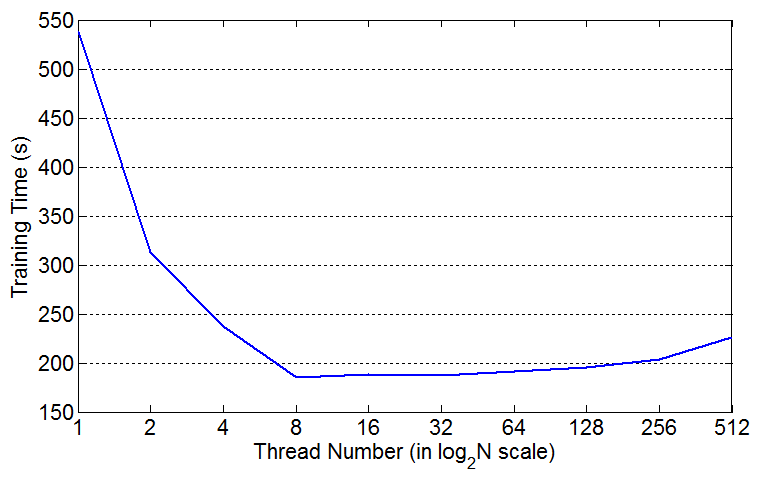


Fig. 9. The curve shows training time versus different numbers of threads. 2000 epochs are taken at each training task and 10 measurements have been taken and the average time consumption is plotted. Algorithm coefficients or factors will not affect the execution time.

TABLE I  
Neural Network Configuration  
for Multi-Thread Efficiency Experiment

|  |  |
| --- | --- |
| Parameter | Configuration |
| learning rate | 0.05 |
| input node number | 8 |
| hidden layer number | 2 |
| hidden node number | 10 |
| hidden layer transfer function | log-sigmoid |
| output node number | 1 |
| output layer transfer function | linear |
| back-propagation algorithm | BP |
| stop early | no |

The result demonstrates that the fastest thread configuration number is at 8, which equals to the thread number of CPU in this case. It is approximately 3-time faster than training with single thread but 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.

## Algorithms Efficiency

Converge epochs and converge error, as known as the error at converge epoch are two measurements of efficiency among different back-propagation algorithms. All algorithms participate in the comparison experiment here. Max training epoch is set to 2000. As soon as the mean absolute value of gradient is less than 10-6 or not changing for 6 epochs [10], training will stop which means that error converges. 10 measurements have been taken and the average values are presented.

TABLE II  
Converge Epochs and Errors of Different Algorithms

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | BP | QP | RP | LM |
|  | 1831 | 73 | 2000 | 6 |
| training error | 15.27% | 15.23% | 2.58% | 15.23% |
| testing error | 15.46% | 15.41% | 2.55% | 15.41% |

The result shows that QP and LM converge faster than BP and RP, especially LM. Although RP takes more epochs to converge, its training and testing errors are distinctly smaller than others’.

TABLE III  
Neural Network Configuration  
for Algorithm Efficiency Experiment

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | BP | QP | RP | LM |
| learning rate | 0.05 | 0.05 | - | - |
| max grow factor | - | 1.75 | - | - |
| increase factor | - | - | 1.2 | - |
| decrease factor | - | - | 0.5 | - |
| initial update value | - | - | 0.1 | - |
|  | - | - | - | 10 |
|  | - | - | - | 10 |
| thread number | 8 | 8 | 8 | 8 |
| stop early | yes | yes | yes | yes |
| Other configurations are as the same as in TABLE I | | | | |

## Algorithm Complexity

For the same network structure, different algorithm will take various number of CPU instructions to run, thus it greatly affect training time among different algorithms. In this experiment, maximum epoch is set to 2000 and coefficients or factors will not affect the execution time if doesn’t stop early. 10 measurements are taken, mean values are presented and standard deviation is presented as reliability.

TABLE IV  
Execution Time of Different Algorithms

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | BP | QP | RP | LM |
| (s) | 177.6 | 229.1 | 181.5 | 335.7 |
| (s) | 4.09 | 2.96 | 3.78 | 10.68 |
| (s) | 162.6 | 8.4 | 181.5 | 1.0 |
| weight size (byte) | 96 | 120 | 120 | 120 |

This experiment shows that BP and RP have got least execution time, QP takes more time and LM takes far more time to run. However, with regards to converge epochs which means training stop early in TABLE II, the total execution time is approximately calculated as,

Still LM and QP holds the least execution time if stop early, especially LM algorithm.

Weights of different algorithms are allocated statically i. BP weight has the smaller size than other.

TABLE V  
Neural Network Configuration  
for Algorithm Efficiency Experiment

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | BP | QP | RP | LM |
| stop early | no | no | no | no |
| Other configurations are as the same as in TABLE III | | | | |

# Conclusion

Based on the architecture and design pattern of this neural networks library, the author developed multiple weight update algorithms one by one after review 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 come continuously.

The author prefers static allocation rather than dynamic allocation is because it takes less CPU instructions to access and overwrite, especially for neural networks training with tremendous number of samples and iterations. The time difference between static and dynamic allocation could be distinct.

Theoretically, the matrix notation could be a more unique way in interpreting learning algorithms. However, under both of the abstraction and generalization considerations, the single variable notation could be more implementable. In addition, considering the weight as an individual unit rather than in a matrix will produce benefits if a network structure is not symmetric but pruned or branched asymmetrically.

Last but most important, the multi-threaded architecture could easily be embedded into any kind of distribution systems by calling few functions this library provides.

The Multi-Threaded Neural Networks Template Library (BeefNet) is available at

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

Appendix

A simple code example of the neural networks template library in C++ is presented here.

//define input, output and hidden node number

const uint32 input\_num = 8;

const uint32 output\_num = 1;

const uint32 hidden\_num = 10;

// define network, err function,

// input and target file reader type

typedef CNN2Layer

<

CWeightLM<>, // default params for LM algorithm:

// lambda = 10, beta = 10.

input\_num,

hidden\_num, FXferLogSig, // log-sigmoid function

hidden\_num, FXferLogSig, // log-sigmoid function

output\_num, FXferLnr // linear function

> NN;

typedef FErrMAE ErrFunction; // mean absolute error

typedef CReaderBinary<input\_num> Input;

typedef CReaderBinary<output\_num> Target;

srand( (uint32)time(NULL) ); // set random seed

NN nn; // create network structure

// neural networks training

double train\_err[output\_num];

CTrainer< NN, Input, Target, ErrFunction > trainer;

trainer.open\_input( "train\_input.dat" );

trainer.open\_target( "train\_target.dat" );

trainer.train<true>( train\_err, nn ); // stop early

// neural networks testing

double test\_err[output\_num];

CTester< NN, Input, Target, ErrFunction > tester;

tester.open\_input( "../../data/test\_input.dat" );

tester.open\_target( "../../data/test\_target.dat" );

tester.test( test\_err, nn );

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. 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, Dearborn, MI 48128. This report is related to his ECE591 Directed Study. (e-mail: wwdxds@gmail.com) [↑](#footnote-ref-1)