2019

Derivative-Based Acceleration of General Vector Machine

Binbin Yong
Lanzhou University, yongbb14@lzu.edu.cn

Fucun Li
University of Wollongong, fl626@uowmail.edu.au

Qingquan Lv
Wind Power Technology Center of Gansu Electric Power Company

Jun Shen
University of Wollongong, jshen@uow.edu.au

Qingguo Zhou
Lanzhou University, zhouqg@lzu.edu.cn

Publication Details

Yong, B., Li, F., Lv, Q., Shen, J. & Zhou, Q. (2019). Derivative-Based Acceleration of General Vector Machine. Soft Computing, 23 987-995.

Derivative-Based Acceleration of General Vector Machine

Abstract
General vector machine (GVM) is one of supervised learning machine, which is based on three-layer neural network. It is capable of constructing a learning model with limited amount of data. Generally, it employs Monte Carlo algorithm (MC) to adjust weights of the underlying network. However, GVM is time-consuming at training and is not efficient when compared with other learning algorithm based on gradient descent learning. In this paper, we present a derivative-based Monte Carlo algorithm (DMC) to accelerate the training of GVM. Our experimental results indicate that DMC algorithm is faster than the original MC method. Specifically, the training time of our DMC algorithm in GVM for function fitting is also less than some gradient descent-based methods, in which we compare DMC with back-propagation neural network. Experimental results indicate that our algorithm is promising for training GVM.

Disciplines
Engineering | Science and Technology Studies

Publication Details
Yong, B., Li, F., Lv, Q., Shen, J. & Zhou, Q. (2019). Derivative-Based Acceleration of General Vector Machine. Soft Computing, 23 987-995.

Noname manuscript No.
(will be inserted by the editor)

Derivative Based Acceleration of General Vector Machine

Binbin Yong · Fucun Li · Qingquan Lv · Jun Shen · Qingguo Zhou*

Received: date / Accepted: date

Abstract General vector machine (GVM) is one of supervised learning machine, which is based on three-layer neural network (NN). It is capable of constructing a learning model with limited amount of data. Generally, it employs Monte Carlo algorithm (MC) to adjust weights of the underlying network. However, GVM is time-consuming at training and is not efficient when compared with other learning algorithm based on gradient descent learning. In this paper, we present a Derivative based Monte Carlo algorithm (DMC) to accelerate the training of GVM. Our experimental results indicate that DMC algorithm is faster than the original MC method. Specifically, the training time of our DMC algorithm in GVM for function fitting is also less than some gradient descent based methods, in which we compare DMC with back propagation neural network (BP). Experimental results indicate that our algorithm is promising for training GVM.

Keywords general vector machine · neural network · gradient descent · derivative · back propagation

* Corresponding Author
Binbin Yong · Qingguo Zhou
School of Information Science and Engineering, Lanzhou University, Lanzhou, Gansu, China
Tel.: +86-0931-8912025
Fax: +86-0931-8912025
E-mail: {yongbb14, zhouqg}@lzu.edu.cn

Fucun Li · Jun Shen
School of Computing and Information Technology, University of Wollongong, NSW, Australia
E-mail: fl626@uowmail.edu.au, jshen@uow.edu.au

Qingquan Lv
Wind Power Technology Center of Gansu Electirc Power Company, Lanzhou, China
E-mail: lvqingquanlzu@126.com

1 Introduction

Nowadays, neural network (NN), as an important model for machine learning and deep learning, has evolved at a fast pace (Wang and Raj, 2017). Meanwhile, NN based applications have been widely researched (Ji et al 2013; Krizhevsky et al 2012a,b; Lecun et al 1998) .

Recently, Zhao (2016) proposed a new supervised learning machine, which is entitled as general vector machine (GVM) based on mathematical statistics analysis. Considering GVM model with only one hidden layer, it successfully introduces Monte Carlo algorithm (MC) to randomly adjust weight matrices of GVM model. The basic idea behind GVM is that, the change of a weight would only be accepted while the cost function decreases. The MC algorithm introduces the strategy by randomly searching a promising change in the weight matrices. Meanwhile, this strategy makes the model more robust for small fluctuations of input vector. Previous experiments results have shown that the design and structure of GVM is effective in many cases. It is especially suitable for learning from limited training samples (Chen et al 2015; Wang et al 2016; Yong et al 2017; Zhou et al 2016). However, these studies also indicate that MC requires more time to train a GVM model compared with gradient descent based back propagation neural network (BP). Hence, in this paper, we design and implement a derivative based Monte Carlo algorithm (DMC) to accelerate the training of GVM models. DMC keeps the randomness of GVM. Meanwhile, we utilize the partial differential information of cost function to a weight, to increase or decrease the weight, so as to effectively reduce the overall cost faster.

The rest of the paper is organised as follows. In Section 2, we briefly introduce the basic concepts of GVM and the related work. Section 3 presents our design and implementation of DMC algorithm. The experiments are illustrated in Section 4. Also, the analysis on the effects of different parameters is discussed in Sect. 4. Section 5 concludes the paper and address the future work.

2 Preliminary and related work

2.1 Preliminary

In general, to construct a NN model, we firstly need to define the structure of the model. Next, we need to calculate the weight matrices of the NN model. Gradient descent algorithm is widely used for training NN models. Similarly, GVM applies a three-layer NN. The difference between GVM and gradient descent method is that GVM introduces MC algorithm to adjust weights of the network, which is consistent with the structural risk minimization (SRM). It randomly changes one weight of the weight matrices in a small range. Then, the GVM model will accept the weight change in the case that it leads the overall cost to converge, which ensures the outputs do not sensitive to small fluctuation of inputs. The GVM will gradually converge to a stable model.

2.1.1 Structure of GVM

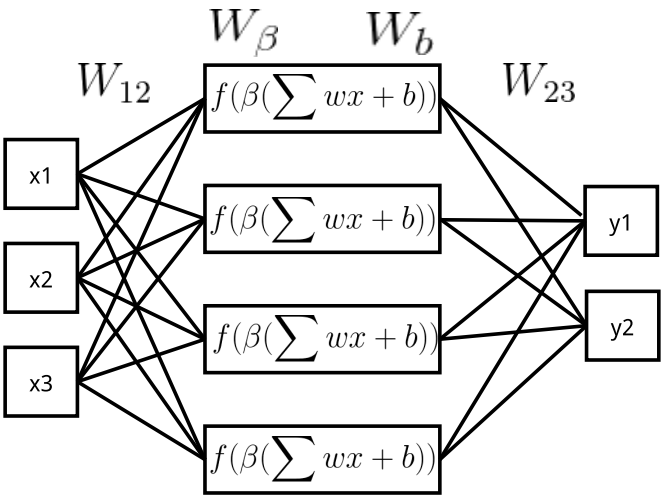

Fig. 1. The structure of GVM

As shown in Fig. 1, a GVM is composed of a fixed three-layer structure. It is proven that, with proper number of hidden nodes, a three-layer NN is able to fit any nonlinear and linear functions Kreinovich and Sirisaengtaksin (1993). In GVM, the outputs of hidden nodes are computed according to the following Eq. (1):

$$h_i = f(\beta(\sum wx + b)) \tag{1}$$

In Eq. (1), x denotes input vector, and w denotes weight vector from input nodes to a hidden node. Hence, $\sum wx$ is the weighted sum of the input nodes. b denotes the bias vector of hidden nodes. β is also an important parameter of each hidden node, which is used to control the smoothness of the model. f stands for the non-linear transfer function of the network. The popular transfer functions include *sigmoid*, *tanh* and *ReLu* etc. h_i is the output of the ith hidden node. These outputs are linearly connected to the output nodes by matrix W_{23}.

In general, the data structures of a GVM model are organized as arrays. The common used terminologies in GVM are listed below: The training samples are denoted as x (input) and y (output). We use W_{12} and W_{23} to represent the weights from input layer to hidden layer and the weights from hidden layer to output layer. The beta parameter is denoted as W_β. The bias parameter of hidden layer is denoted as W_b. In some cases, we use W to represent all the parameters. The cost function and the overall cost of a GVM model are all denoted as COST.

2.1.2 Design of GVM

In the design of GVM, there are five key components including: weight initialization, step, number of hidden nodes, transfer function and cost function.

(1) Weight initialization
According to experimental experiences, weight matrix W_{23} is randomly set as -1 or $+1$ at the beginning. Normally we will not further change W_{23} any more after initialization. Weight matrix W_{12} is initialized as decimals between -1 and $+1$, and weight vector β is initialized between -0.5 and $+0.5$. The range of bias b is not so important as long as it is not equal to 0.0. In general, we initialize it between -2 to $+2$ for function fitting task.

(2) Step
When changing a weight, the weight should be maintained in its range, as shown above. In general, the range of weight change is set as a small value, which is denoted as *step*. When training GVM, the *step* will decrease with the decrease of the cost. The original MC method of training GVM

gives an empirical expression as Eq. (2). In the specific implementation, *step* is set as 0.1 when the cost is relatively large. However, when the cost reduces to a small value, the *step* is updated according to Eq. (2). k is a control coefficient, which is normally set as 0.1.

$$step = k \cdot \sqrt{COST} \tag{2}$$

Empirically, a *step* decreasing with the cost will avoid sharp fluctuations of the GVM model. Meanwhile, it will ensure that the model smoothly evolves into an optimal state. Hence, a good *step* function is very useful for optimizing training time. In our design of DMC, we use a different *step* function to achieve a better training efficiency.

(3) Number of hidden nodes
Traditional BP use less hidden nodes to avoid over-fitting. However, GVM introduces the parameter β to control the smoothness of the fitting curve, so as to control over-fitting. In this paper, we can use more hidden nodes to enhance the learning power of GVM. For specific function fitting, such as *sin* function, BP uses only three hidden nodes while more hidden nodes will cause over-fitting. However, GVM is able to use 150 hidden nodes to achieve a better fitting result. Later we will show that GVM performs better than BP when fitting *sin* function with small samples.

(4) Transfer function
The choice of transfer function of GVM is very flexible. The basic principle is that the transfer function should be a non-linear function. The commonly used transfer functions include *sigmoid*, *tanh*, and Gaussian function (e^{-x^2}). In this paper, we mainly focus on function fitting experiments. In most cases, we use *tanh* as the transfer function, which is defined as the following equation:

$$f = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3}$$

(5) Cost function
In function fitting, Mean Square Error (MSE) is usually used as the cost function. For simplicity, we use a similar Eq. (4) as the cost function in this paper. Our experiment results prove that it is suitable for function fitting in GVM. In Eq. (4), $y_i^{'}$ represents the ith predicted output value. y_i is the ith actual value from the samples. In general, we denote $diff = y_i^{'} - y_i$.

$$COST = \frac{1}{2}\sum_{i=1}^{N}(y_i^{'} - y_i)^2 \tag{4}$$

2.1.3 MC training method

The parameters we need to adjust in GVM model are the parameter matrices W_{12}, W_β and W_b. The original method in GVM design is the MC method, which is not a fixed method, and it is very flexible according to different scenarios. Here we give a general method of using MC in training GVM for function fitting. We sort the adjustment of parameters in the sequence of W_β, W_{12} and W_b. The pseudo-code is shown below in Algorithm (1).

Algorithm 1 MC training algorithm

Function (1)
Input:
epochs number of epochs
N, includes $nW_{12}, nBeta, nBias$ dimension of matrix
W_{12} parameter matrix W_{12}
W_β parameter matrix W_{beta}
W_{bias} parameter matrix W_{bias}

Output: update parameter matrices of GVM

```
function MonteCarlo(epochs, N, W12, Wβ, Wbias)
    mincost ← COST()
    for i = 1 to epochs do
        select ← RANDOM(0, 2)
        if select == 0 then
            index ← RANDOM(0, nBeta)
            origin ← Wβ[index]
            Wβ[index] ← newBeta()
            cost ← COST()
            if cost < mincost then
                mincost ← cost
            else
                Wβ[index] ← origin
            end if
        else if select == 1 then
            index ← RANDOM(0, nW12)
            origin ← W12[index]
            W12[index] ← newWeight()
            cost ← COST()
            if cost < mincost then
                mincost ← cost
            else
                W12[index] ← origin
            end if
        else
            index ← RANDOM(0, nBias)
            origin ← Wbias[index]
            Wbias[index] ← newBias()
            cost ← COST()
            if cost < mincost then
                mincost ← cost
            else
                Wbias[index] ← origin
            end if
        end if
    end for
    return
end function
```

In our implementation of MC in Algorithm (1), the parameter matrices W_{12}, W_β, W_{bias} are organized as arrays. Next, we generate a random number *select* between 0, 1 and 2 to determine which weight array (weight matrix) is selected to change in each epoch. Then we generate another random number *index* between 0 and the length of the weight array. By *index* the corresponding weight $W_\beta[index]$ or $W_{12}[index]$ or $W_{bias}[index]$ is chosen. Next, a small change is tried on the weight to determine whether the cost will decrease. If the cost is decreased, a new cost value is acquired, otherwise the changed weight would be restored to its original value. The step of weight change is usually set according to Eq. (2).

2.2 Related work

In general, training a NN is to adjust the weight matrices to reduce the deviations between the output vector and the predict vector. The most frequently used method for training NN is back propagation algorithm developed by Hagan et al (1995), and the corresponding neural network is back propagation neural network. At the same time, they pointed out that random algorithms may be useful in finding out the best solution. Yet due to the lag of computational ability, they did not apply the random strategy in NN.

As a sophisticated algorithm, MC has been widely applied to many application fields. Abdalla and Buckley (2007, 2008) applied their fuzzy MC method in solving linear regression problems. By using MC method, Duygu and Cattaneo (2016) researched the way to determine the best parameters of fuzzy linear regression.

Freitas et al (2000) proposed a strategy for training NNs with MC algorithm. Then, Liang (2007) proposed the annealing stochastic approximation MC (ASAMC) algorithm for training NN. Zhao (2016) recently recalled this idea owing to the improvement of computing capability. In his experiments, applying MC in training GVM shows good performance, especially for training small dataset. Afterwards, as a new method of machine learning, GVM has been used in several different application scenarios.

Chen et al (2015) applied the same method to detect genetic features of cancers and obtained good classification results. Zhou et al (2016) has also applied this method in non-linear time series prediction. Yong et al (2017) used GVM to predict the electricity demand and achieved good prediction results. However, in these researches, the prediction results were considered as the primary tasks. The training efficiency is rarely studied.

Zhao (2016) has proposed a simple way to accelerate the training of GVM. That is, if we change a weight connected to the ith hidden node, the outputs of many nodes remain unchanged. Hence, by recording the outputs of these unchanged nodes, massive redundant computation is avoided, and we only need to recalculate the outputs connected to the ith hidden node. This method is very particularly useful when there are a large number of hidden nodes. This method of computing optimization is also advisable for our method, which will be also used in our DMC algorithm. The details will be presented in Sect. 3, and we will compare this combined method with the original method in the final part of our experimental section.

3 DMC Algorithm

In this section, we discuss the DMC algorithm for training GVM. We will describe our acceleration algorithm and present the detailed techniques of the algorithm in this section.

3.1 Overview

One of the most important aspects for training GVM is to utilize the randomness strategy of weight changing. However, with the convergence of overall cost, it appears to be more difficult to find a suitable weight to change by MC method. Considering the idea of traditional gradient descent algorithm, the gradient information is introduced in MC method, which is called DMC algorithm in this paper, to train GVM. The novel DMC algorithm keeps the randomness of weight changing. Meanwhile, it utilizes the gradient information combined with MC algorithm to adjust the weight matrices of GVM.

The DMC algorithm is depicted in Fig. 2. In general, the steps from 'Init samples' to 'Init minimum cost' are the initialization phase. The rest steps is the main part, which consists of many iterations to randomly find useful weight changes, so as to reduce the cost.

3.2 Logical equivalence

To guarantee the randomness, the original MC method does not use any extra strategy when finding suitable changes. Although the gradient information is introduced in DMC algorithm, we could still prove that DMC is logically equivalent to the original MC in randomness.

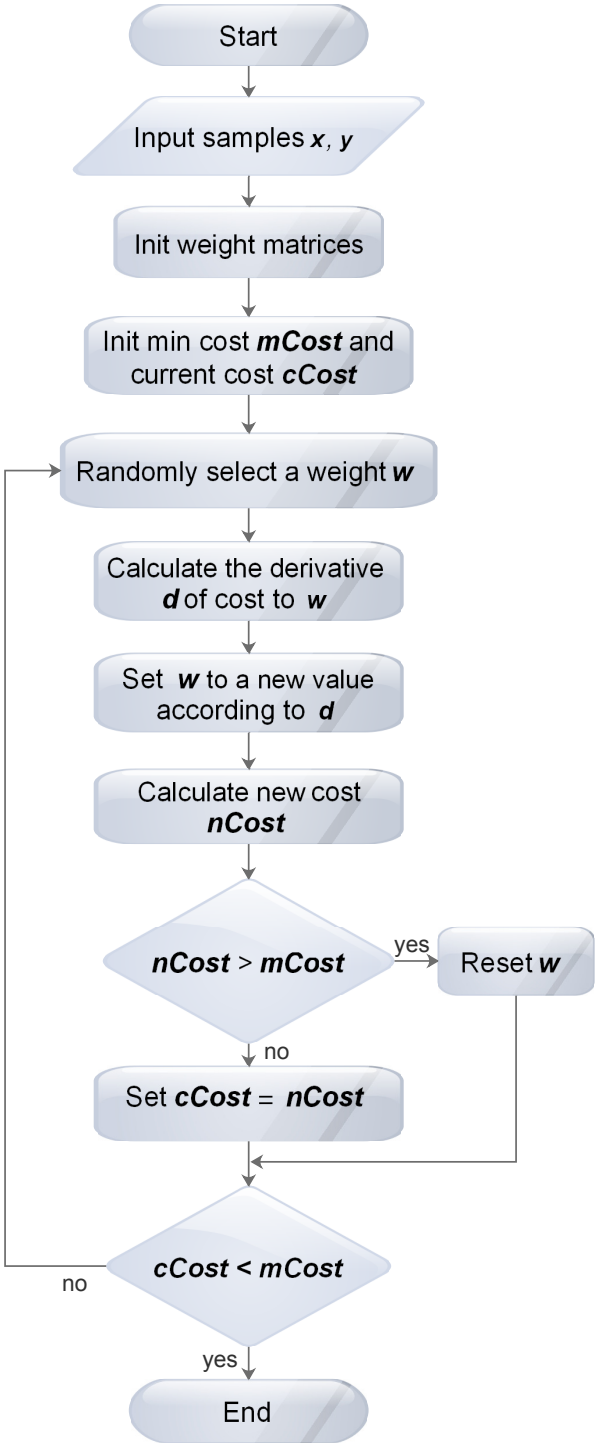

Fig. 2. The flowchart of DMC algorithm for training GVM

Suppose $R(x)$ is a random number that stands for the random index of W to change. $Y(R(x))$ denotes that the change of $R(x)$ is acceptable, and $N(R(x))$ denotes that the change of $R(x)$ is unacceptable. Now, we randomly generate an index $R(a)$. If $Y(R(a))$, the overall cost will decrease, and it equals that we use derivative based method to change a little of $W[R(a)]$. On the other hand, if $N(R(a))$, we will not adopt the change. We will then generate $R(b)$ until $W[R(b)]$ is acceptable. $R(x)$ is an independent random event for each x. Hence, the final $R(b)$ is equal to $R(a)$ in regard to randomness. While we use derivative based method, we do not change the randomness strategy of the original MC algorithm.

3.3 Partial derivative

In the application of function fitting by GVM, we set Eq. (4) as the cost function. The transfer function is set as *tanh* as Eq. (3). The derivative of function *tanh* is given in Eq. (5).

$$f^{'} = f'(x) = 1 - tanh^2(x) \tag{5}$$

The number of training samples is denoted as N. For simplicity, we use one input node and one single output node. The extensions to multi-input and multi-output are similar. In fact, some modern machine learning libraries, such as TensorFlow from Google, provide an application programming interface (API) (tensorflow.gradients) to calculate the derivative of cost with respect to its expression W parameter automatically. In the implementation of DMC, the derivative equations are as follows:

$$\frac{\partial COST}{\partial W_{12}[i]} = \sum_{j=1}^{N}(y_j^{'} - y_j) \cdot W_{23}[i] \cdot f^{'} \cdot W_{beta}[i] \cdot x_j \tag{6}$$

$$\frac{\partial COST}{\partial W_{beta}[i]} = \sum_{j=1}^{N}(y_j^{'} - y_j) \cdot W_{23}[i] \cdot f^{'} \cdot (W_{12}[i]x_j + W_{bias}[j]) \tag{7}$$

$$\frac{\partial COST}{\partial W_{bias}[i]} = \sum_{j=1}^{N}(y_j^{'} - y_j) \cdot W_{23}[i] \cdot f^{'} \cdot W_{beta}[j] \tag{8}$$

According to Eq. (6), (7) and (8) , we can easily achieve the partial derivative of cost function with respect to its each weight.

3.4 Weight change range

As discussed above, *step* is an important parameter in original MC and our DMC method. The basic principle is that the *step* should decrease with the iterative process. However, if *step* reduces to 0 or a relatively

small value, the GVM will no longer find any suitable change in the range of *step*. Hence, we should also keep the *step* larger than a minimum value. In our implementation, we use the following Eq. (9) to change *step*.

$$step = \alpha \cdot s1 + (1 - \alpha) \cdot s2 + s3 \tag{9}$$

Shown as above, α is initialized as 1.0 at the beginning. It is replaced with $0.99 \cdot \alpha$ after every 100 training epochs. $s1$ is a bigger value (10^{-1}). $s2$ is a smaller value (10^{-3}), and $s3$ is the smallest value (10^{-4}). By selecting the parameters $s1$, $s2$ and $s3$, we can control the descent speed of *step*, so as to control the convergence speed of overall cost.

3.5 The change of weights

After determining the partial derivative of cost with respect to a weight (denoted as w) and the change range *step*, the weight is changed as Eq. (10).

$$w = w + rand \cdot k \tag{10}$$

In which, *rand* is a float random number in the range of selected weight. k is set as -1 if the partial derivative is greater than 0, otherwise it is set as 1. By this formula, we can efficiently adjust the weights to converge to an ideal state.

3.6 Computing optimization method

As discussed before, we combine the original computing optimization method with our derivative based method as the final DMC algorithm.

To implement the optimization by avoiding the redundant computation, we allocate a temp array (denoted as $TEMP$) to store the temporary outputs of hidden nodes. The $TEMP$ array is initialized as the following pseudo-code Function (2):

The values of $TEMP$ array are based on training samples and weight parameters. Once a weight is changed, we need to update the $TEMP$ array. We also give the pseudo-code for updating $TEMP$ in Function (3). In fact, we only update the elements related to the changed weight (*index* th weight).

With $TEMP$ array, the *cost* is calculated according to Function (4). In this case, every time we change a weight, only the related nodes are activated to calculate the cost.

The computing optimization method is very useful for accelerating the training. However, it is independent of the derivative method. These two methods

Function (2)
Input:
index index of weights (include W_{12}, W_β, W_b) to change
$nSAMPLE$ number of samples
$TEMP$ temporary values array
$nHID$ number of hidden units
Output: initialize TEMP matrix
function INITTEMP($index, nSAMPLE, TEMP, nHID$)
  $size \leftarrow nSAMPLE * nHID$
  $TEMP \leftarrow malloc(size)$
  for $i = 0$ to $nSAMPLE$-1 **do**
    for $j = 0$ to $nHID$-1 **do**
      $z \leftarrow W_{12}[j] * x[i] + W_b[j]$
      $a \leftarrow tanh(W_\beta[j] * z)$
      $TEMP[i * nHID + j] \leftarrow W_{23}[j] * a$
    end for
  end for
  return
end function

Function (3)
Input:
index index of weights (include W_{12}, W_β, W_b) to change
$nSAMPLE$ number of samples
$TEMP$ temporary values array
$nHID$ number of hidden units
Output: update TEMP matrix
function UPDATE($index, nSAMPLE, TEMP, nHID$)
  for $k = 0$ to $nSAMPLE$-1 **do**
    $z \leftarrow W_{12}[index] * x[k] + W_b[index]$
    $a \leftarrow tanh(W_\beta[index] * z)$
    $TEMP[k * nHID + index] \leftarrow W_{23}[index] * a$
  end for
  return
end function

Function (4)
Input:
x input of samples
y output of samples
$nSAMPLE$ number of samples
$TEMP$ temporary values array
$nHID$ number of hidden units
Output: $COST$
function CALCCOST($x, y, TEMP, nHID$)
  $cost \leftarrow 0$
  for $i = 0$ to $nSAMPLE$-1 **do**
    $yp \leftarrow 0$
    for $j = 0$ to $nHid$-1 **do**
      $yp \leftarrow yp + TEMP[i * nHID + j]$
    end for
    $diff \leftarrow yp - y[i]$
    $cost \leftarrow diff * diff$
  end for
  $cost \leftarrow cost/2.0$
  return *cost*
end function

both can accelerate the training. Therefore, in the next experimental section, we will firstly test the derivative based method. Then we will test the combined DMC algorithm.

4 Experimental Results and Analysis

In the following subsections, we test the computational efficiency and accuracy of our DMC algorithm, and compare it with the original MC method and BP method. All the implementations of these algorithms are running on the same platform. The experimental environment is set as: a personal computer with Intel Core i3 processor and 4GB RAM with Ubuntu 15.04 installed. The programs are implemented in GNU environment with C programming language.

4.1 Function fitting accuracy

In order to test the accuracy of GVM, we use 7 training samples where the input values are evenly distributed to fit *sin* function. Meanwhile, DMC is used as the training method. In contrast, we also use a BP to fit *sin* function. The input values are limited between -2π and $+2\pi$. We tested a series of number of hidden nodes. Results show that when BP uses 3 hidden neuron nodes and GVM uses 150 hidden neuron nodes, we get the best fitting curves of *sin* function, respectively. The fitting curve of GVM is shown in Fig. 3 and Fig. 4 shows the fitting result of BP. We can easily conclude that GVM fits better than BP when using only 7 training samples. BP would require at least 10 training samples to achieve a relatively good fitting curve as GVM. This experiment indicates that GVM performs better than BP when using little training samples.

We also tested the role of β. If we keep β as 1, the GVM model is actually a traditional NN. Then we used MC to train the model with 7 training samples. The fitting result is shown in Fig. 5, which is pretty bad compared to GVM. Hence, without β, GVM will degenerate into an ordinary NN, which is lack of the advantages of GVM.

We also validate the importance of β. Fig. 5 is the fitting curve using 7 training samples while β is constantly set to 1. We could observe that this GVM performs worse than Fig. 3, which give us the evidence that β is one of the important parameters during optimization.

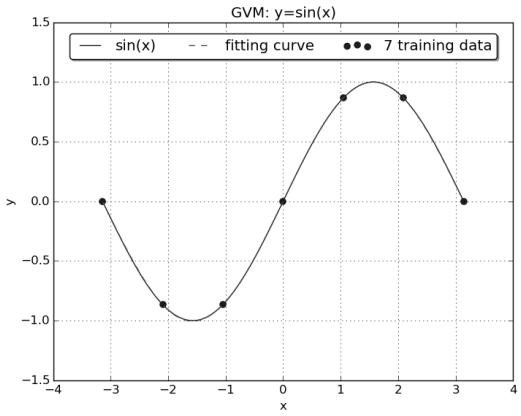

Fig. 3. The fitting curve by GVM with 7 training samples and 150 hidden nodes

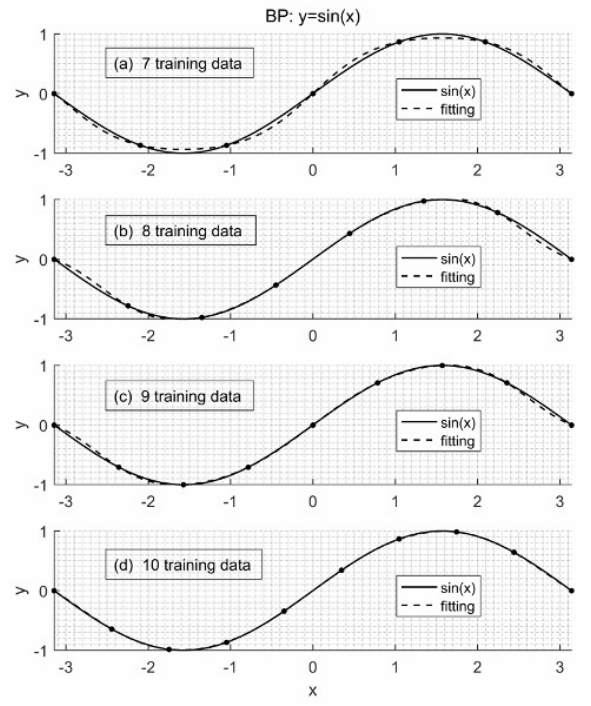

Fig. 4. The fitting curve by BP with 7, 8, 9 and 10 samples and 3 hidden nodes

4.2 Computational efficiency

In the implementation of fitting *sin* function, α is set as $0.99 * \alpha$ every 10 epochs as discussed in Section 3.4. $s1$ is set as 0.1, $s2$ is set as 0.01, and $s3$ is set as 0.0005. Then the *step* is calculated as Eq. (11).

$$step = \alpha * 0.1 + (1 - \alpha) * 0.001 + 0.0005; \tag{11}$$

All these compared methods use the same cost function, as Eq. (4). We compare two situations with 100 hidden nodes and 150 hidden nodes, respectively.

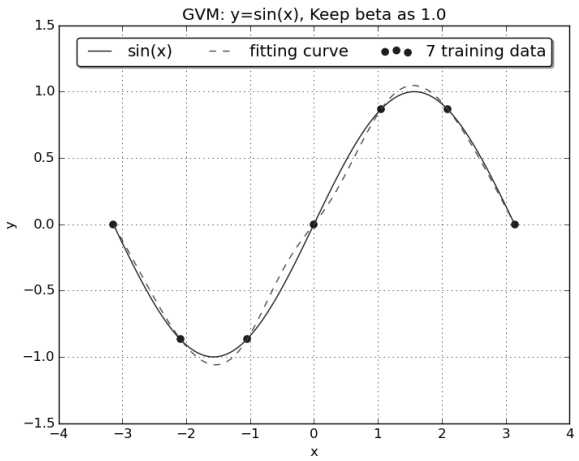

Fig. 5. The fitting curve by GVM with 7 training samples, 150 hidden nodes and a fixed beta 1

Meanwhile, we take the number of training samples as the variable of our experiments. Then we test the average training time in each scenario. The results are shown in Fig. 6. It can be observed that DMC has a similar training time compared to BP. In some cases, DMC is even faster than BP. Compared to original MC algorithm, DMC achieves a speed up rate of maximum 7.57 folds.

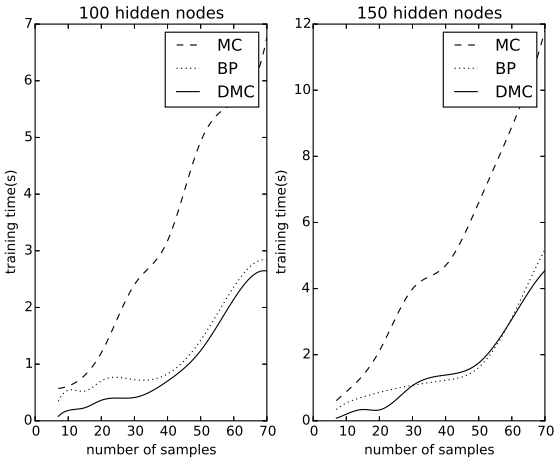

Fig. 6. The comparison of training time between MC and DMC

4.3 Combined DMC algorithm

We combine computation optimization method from and DMC to test the training time. The results are illustrated in Fig. 7. We can see that original MC and DMC are both accelerated by computation optimization method. Similar to Fig. 6, DMC is faster than MC and the maximum speed up rate is 7.4. In this scenario for function fitting, we can also see that both MC and DMC with computation optimization are faster than BP. Herein we only discuss the situation with small training samples, which is suitable for applying GVM. In fact, for large number of samples, BP trains faster than GVM.

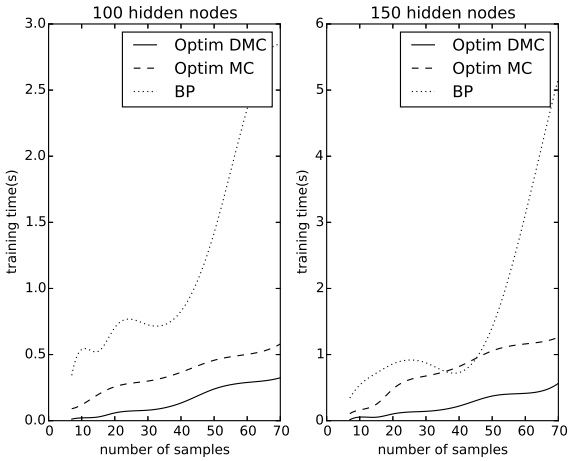

Fig. 7. The comparison of training time between optimized MC and optimized DMC

4.4 Complex function fitting

We also tested the computational efficiency of some complex functions. In these tests, the number of training samples is set as the same. The training times are listed in Table 1.

Table 1 Training time (s) of different functions based on BP, MC and DMC

function	BP	MC	DMC
$sin(x)$	0.32	1.08	**0.19**
$sin(x^2)$	0.50	3.18	**0.53**
$e^{sin(x)}$	0.55	1.38	**0.22**
$e^{sin(x)} + x^2$	**0.22**	2.51	**1.28**
$sin^2(x)$	0.79	1.66	**0.63**

Generally, we can have the following conclusions from Table 1. In most cases, DMC performs faster than BP. Sometimes, it will be a little slower than BP (fitting function $e^{sin(x)} + x^2$). Overall, DMC has a great

improvement in training efficiency than the original MC.

5 Conclusion

GVM is a new type of NN, which is proved effective in classification and regression problems, especially for the scenarios lacking training samples. The original GVM adopts MC algorithm to achieve the optimum solution. However, it is inadequate sometimes as it may be inefficient in searching suitable weight changes. In this paper, we have presented an improved DMC algorithm to train GVM more effectively. Meanwhile, DMC keeps the randomness and the advantages of GVM. We also proved that back propagation algorithm is not suitable for training GVM. In the case of lacking training data, DMC performs faster than BP. In our paper, we also discussed the influence of *step*.

The experimental results prove that DMC algorithm is feasible and efficient for training GVM. However, we only tested it on function fitting. The complex applications in pattern recognition or other machine learning areas are still not tested. These experiments will be undertaken in the future.

References

Abdalla A, Buckley JJ (2007) Monte carlo methods in fuzzy linear regression. Soft Computing 11(10):991–996

Abdalla A, Buckley JJ (2008) Monte carlo methods in fuzzy linear regression ii. Soft Computing 12(5):463–468, doi: 10.1007/s00500-007-0179-6

Chen H, Zhao H, Shen J, Zhou R, Zhou Q (2015) Supervised machine learning model for high dimensional gene data in colon cancer detection. IEEE BigData Congress pp 134–141

Duygu, Cattaneo MEGV (2016) Different distance measures for fuzzy linear regression with monte carlo methods. Soft Computing pp 1–11, doi: 10.1007/s00500-016-2218-7

Freitas JFGD, Niranjan MA, Gee AH, Doucet A (2000) Sequential monte carlo methods to train neural network models. Neural Computation 12(4):955

Hagan MT, Demuth HB, Beale MH (1995) Neural Network Design. PWS Pub. Co.

Ji S, Xu W, Yang M, Yu K (2013) 3D convolutional neural networks for human action recognition. IEEE Transactions on Pattern Analysis and Machine Intelligence 35(1):221–231, doi: 10.1109/TPAMI.2012.59

Kreinovich V, Sirisaengtaksin O (1993) 3-layer neural networks are universal approximators for functionals and for control strategies. Neural Parallel & Scientific Computations 1(3)

Krizhevsky A, Sutskever I, Hinton GE (2012a) Imagenet classification with deep convolutional neural networks. In: Pereira F, Burges CJC, Bottou L, Weinberger KQ (eds) Advances in Neural Information Processing Systems 25, Curran Associates, Inc., pp 1097–1105

Krizhevsky A, Sutskever I, Hinton GE (2012b) Imagenet classification with deep convolutional neural networks. In: International Conference on Neural Information Processing Systems, pp 1097–1105

Lecun Y, Bottou L, Bengio Y, Haffner P (1998) Gradient-based learning applied to document recognition. Proceedings of the IEEE 86(11):2278–2324

Liang F (2007) Annealing stochastic approximation monte carlo algorithm for neural network training. Machine Learning 68(3):201–233

Wang H, Raj B (2017) On the origin of deep learning. arXiv preprint

Wang L, Shen J, Zhou Q, Shang Z, Chen H, Zhao H (2016) An evaluation of the dynamics of diluted neural network 9(6):1191–1199

Yong B, Xu Z, Shen J, Chen H, Tian Y, Zhou Q (2017) Neural network model with monte carlo algorithm for electricity demand forecasting in queensland. In: Australasian Computer Science Week Multiconference, p 47

Zhao H (2016) General vector machine. arXiv preprint

Zhou Q, Chen H, Zhao H, Shen J (2016) A local field correlated and monte carlo based shallow neural network model for nonlinear time series prediction. Scalable Information Systems 16(8)