Back-propagation **Neural** Net

Tejpal Singh Chhabra, 29 Mar 2006

4.76 (38 votes) Rate:

A C++ class implementing a back-propagation algorithm **neural** net, that supports any number of layers/neurons.

Download demo project - 4.64 Kb

Introduction

The class `CBackProp` encapsulates a feed-forward **neural** network and a back-propagation algorithm to train it. This article is intended for those who already have some idea about **neural networks** and back-propagation algorithms. If you are not familiar with these, I suggest going through some material first.

Background

This is part of an academic project which I worked on during my final semester back in college, for which I needed to find the optimal number and size of hidden layers and learning parameters for different data sets. It wasn't easy finalizing the data structure for the **neural** net and getting the back-propagation algorithm to work. The motivation for this article is to save someone else the same effort.

Here's a little disclaimer... This article describes a simplistic implementation of the algorithm, and does not sufficiently elaborate on the algorithm. There is a lot of room to improve the included code (like adding exception handling :-), and for many steps, there is a lot more reasoning required than I have included, e.g., values that I have chosen for parameters, and the number of layers/the neurons in each layer are for demonstrating the usage and may not be optimal. To know more about these, I suggest going to:

- **Neural** Network FAQ

Using the code

Typically, the usage involves the following steps:

- Create the net using `CBackProp::CBackProp(int nl,int *sz,double b,double a)`
- Apply the back-propagation algorithm - train the net by passing the input and the desired output, to `void CBackProp::bpgt(double *in,double *tgt)` in a loop, until the *Mean square error*, which is obtained by `CBackProp::double mse(double *tgt)`, gets reduced to an acceptable value.
- Use the trained net to make predictions by *feed-forward*ing the input data using `void CBackProp::ffwd(double *in)`.

The following is a description of the sample program that I have included.

One step at a time...

Setting the objective:

We will try to teach our net to crack the binary **A *XOR* B *XOR* C**. XOR is an obvious choice, it is not linearly separable hence requires hidden layers and cannot be learned by a single perception.

A training data set consists of multiple records, where each record contains fields which are input to the net, followed by fields consisting of the desired output. In this example, it's three inputs + one desired output.

Hide Copy Code

```
// prepare XOR training data
double data[][4]={//    I  XOR  I  XOR  I   =   O
                  //--------------------------------
                        0,      0,      0,      0,
                        0,      0,      1,      1,
                        0,      1,      0,      1,
                        0,      1,      1,      0,
                        1,      0,      0,      1,
                        1,      0,      1,      0,
                        1,      1,      0,      0,
                        1,      1,      1,      1 };
```

Configuration:

Next, we need to specify *a suitable* structure for our **neural** network, i.e., the number of hidden layers it should have and the number of neurons in each layer. Then, we specify *suitable* values for other parameters: **learning rate** - `beta`, we may also want to specify **momentum** - `alpha` (this one is optional), and **Threshold** - `thresh` (target *mean square error*, training stops once it is achieved else continues for `num_iter` number of times).

Let's define a net with 4 layers having 3,3,3, and 1 neuron respectively. Since the first layer is the input layer, i.e., simply a placeholder for the input parameters, it has to be the same size as the number of input parameters, and the last layer being the output layer must be same size as the number of outputs - in our example, these are 3 and 1. Those other layers in between are called *hidden layers*.

Hide Copy Code

```
int numLayers = 4, lSz[4] = {3,3,3,1};
double beta = 0.2, alpha = 0.1, thresh = 0.00001;
long num_iter = 500000;
```

Creating the net:

Hide Copy Code

```
CBackProp *bp = new CBackProp(numLayers, lSz, beta, alpha);
```

Training:

Hide Copy Code

```
for (long i=0; i < num_iter ; i++)
{
    bp->bpgt(data[i%8], &data[i%8][3]);

    if( bp->mse(&data[i%8][3]) < thresh)
        break; // mse < threshold - we are done training!!!
}
```

Let's test its wisdom:

We prepare *test data*, which here is the same as *training data* minus the desired output.

Hide Copy Code

```
double testData[][3]={ //  I  XOR  I  XOR  I  =  ?
                       //----------------------
                           0,      0,      0,
                           0,      0,      1,
                           0,      1,      0,
                           0,      1,      1,
                           1,      0,      0,
                           1,      0,      1,
                           1,      1,      0,
                           1,      1,      1};
```

Now, using the trained network to make predictions on our test data....

Hide Copy Code

```
for ( i = 0 ; i < 8 ; i++ )
{
    bp->ffwd(testData[i]);
    cout << testData[i][0]<< "  "
         << testData[i][1]<< "  "
         << testData[i][2]<< "  "
         << bp->Out(0) << endl;
}
```

Now a peek inside:

Storage for the neural net

I think the following code has ample comments and is self-explanatory...

Hide Shrink Copy Code

```
class CBackProp{

//      output of each neuron
        double **out;

//      delta error value for each neuron
        double **delta;

//      3-D array to store weights for each neuron
        double ***weight;

//      no of layers in net including input layer
        int numl;

//      array of numl elements to store size of each layer
        int *lsize;
```

```
//      learning rate
        double beta;

//      momentum
        double alpha;

//      storage for weight-change made in previous epoch
        double ***prevDwt;

//      sigmoid function
        double sigmoid(double in);

public:

        ~CBackProp();

//      initializes and allocates memory
        CBackProp(int nl,int *sz,double b,double a);

//      backpropogates error for one set of input
        void bpgt(double *in,double *tgt);

//      feed forwards activations for one set of inputs
        void ffwd(double *in);

//      returns mean square error of the net
        double mse(double *tgt);

//      returns i'th output of the net
        double Out(int i) const;
};
```

Some alternative implementations define a separate class for layer / neuron / connection, and then put those together to form a **neural** network. Although it is definitely a cleaner approach, I decided to use `double ***` and `double **` to store weights and output etc. by allocating the exact amount of memory required, due to:

- The ease it provides while implementing the learning algorithm, for instance, for weight at the connection between $(i-1)^{th}$ layer's j^{th} Neuron and i^{th} layer's k^{th} neuron, I personally prefer `w[i][k][j]` (than something like `net.layer[i].neuron[k].getWeight(j)`). The output of the `i`th neuron of the `j`th layer is `out[i][j]`, and so on.
- Another advantage I felt is the flexibility of choosing any number and size of the layers.

Hide Shrink Copy Code

```
// initializes and allocates memory
CBackProp::CBackProp(int nl,int *sz,double b,double a):beta(b),alpha(a)
{

// Note that the following are unused,
//
// delta[0]
// weight[0]
// prevDwt[0]

//  I did this intentionally to maintain
//  consistency in numbering the layers.
//  Since for a net having n layers,
//  input layer is referred to as 0th layer,
//  first hidden layer as 1st layer
//  and the nth layer as output layer. And
//  first (0th) layer just stores the inputs
//  hence there is no delta or weight
//  values associated to it.


    //    set no of layers and their sizes
    numl=nl;
    lsize=new int[numl];

    for(int i=0;i<numl;i++){
        lsize[i]=sz[i];
    }
```

```
    //    allocate memory for output of each neuron
    out = new double*[numl];

    for( i=0;i<numl;i++){
        out[i]=new double[lsize[i]];
    }

    //    allocate memory for delta
    delta = new double*[numl];

    for(i=1;i<numl;i++){
        delta[i]=new double[lsize[i]];
    }

    //    allocate memory for weights
    weight = new double**[numl];

    for(i=1;i<numl;i++){
        weight[i]=new double*[lsize[i]];
    }
    for(i=1;i<numl;i++){
        for(int j=0;j<lsize[i];j++){
            weight[i][j]=new double[lsize[i-1]+1];
        }
    }

    //    allocate memory for previous weights
    prevDwt = new double**[numl];

    for(i=1;i<numl;i++){
        prevDwt[i]=new double*[lsize[i]];

    }
    for(i=1;i<numl;i++){
        for(int j=0;j<lsize[i];j++){
            prevDwt[i][j]=new double[lsize[i-1]+1];
        }
    }

    //    seed and assign random weights
    srand((unsigned)(time(NULL)));
    for(i=1;i<numl;i++)
        for(int j=0;j<lsize[i];j++)
            for(int k=0;k<lsize[i-1]+1;k++)
                weight[i][j][k]=(double)(rand())/(RAND_MAX/2) - 1;

    //    initialize previous weights to 0 for first iteration
    for(i=1;i<numl;i++)
        for(int j=0;j<lsize[i];j++)
            for(int k=0;k<lsize[i-1]+1;k++)
                prevDwt[i][j][k]=(double)0.0;
}
```

Feed-Forward

This function updates the output value for each neuron. Starting with the first hidden layer, it takes the input to each neuron and finds the output (o) by first calculating the weighted sum of inputs and then applying the Sigmoid function to it, and passes it forward to the next layer until the output layer is updated:

$$o = \sigma(\vec{w}\vec{x})$$

where:

$$\sigma(y) = \frac{1}{1+e^{-y}}$$

Hide Shrink ▲ Copy Code

```
// feed forward one set of input
void CBackProp::ffwd(double *in)
{
```

```
        double sum;

 // assign content to input layer

        for(int i=0;i < lsize[0];i++)
                out[0][i]=in[i];


// assign output(activation) value
// to each neuron usng sigmoid func

        // For each layer
        for(i=1;i < numl;i++){
                // For each neuron in current layer
                for(int j=0;j < lsize[i];j++){
                        sum=0.0;
                        // For input from each neuron in preceding layer
                        for(int k=0;k < lsize[i-1];k++){
                                // Apply weight to inputs and add to sum
                                sum+= out[i-1][k]*weight[i][j][k];
                        }
                        // Apply bias
                        sum+=weight[i][j][lsize[i-1]];
                        // Apply sigmoid function
                        out[i][j]=sigmoid(sum);
                }
        }
}
```

Back-propagating...

The algorithm is implemented in the function `void CBackProp::bpgt(double *in,double *tgt)`. Following are the various steps involved in back-propagating the error in the output layer up till the first hidden layer.

Hide Copy Code

```
void CBackProp::bpgt(double *in,double *tgt)
{
    double sum;
```

First, we call `void CBackProp::ffwd(double *in)` to update the output values for each neuron. This function takes the input to the net and finds the output of each neuron:

$$o = \sigma(\vec{w}\vec{x})$$

where:

$$\sigma(y) = \frac{1}{1+e^{-y}}$$

Hide Copy Code

```
ffwd(in);
```

The next step is to find the delta for the output layer:

$$\delta_k \leftarrow o_k(1-o_k)(t_k-o_k)$$

Hide Copy Code

```
for(int i=0;i < lsize[numl-1];i++){
   delta[numl-1][i]=out[numl-1][i]*
     (1-out[numl-1][i])*(tgt[i]-out[numl-1][i]);
}
```

then find the delta for the hidden layers...

$$\delta_h \leftarrow o_h(1-o_h)\sum_{k \in outputs} w_{kh}\delta_k$$

Hide Copy Code

```
for(i=numl-2;i>0;i--){
    for(int j=0;j < lsize[i];j++){
        sum=0.0;
        for(int k=0;k < lsize[i+1];k++){
                sum+=delta[i+1][k]*weight[i+1][k][j];
        }
        delta[i][j]=out[i][j]*(1-out[i][j])*sum;
    }
}
```

Apply momentum (does nothing if alpha=0):

Hide Copy Code

```
for(i=1;i < numl;i++){
    for(int j=0;j < lsize[i];j++){
        for(int k=0;k < lsize[i-1];k++){
                weight[i][j][k]+=alpha*prevDwt[i][j][k];
        }
        weight[i][j][lsize[i-1]]+=alpha*prevDwt[i][j][lsize[i-1]];
    }
}
```

Finally, adjust the weights by finding the correction to the weight.

$$\Delta w_{ji} = \eta \delta_j x_{ji}$$

And then apply the correction:

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

Hide Copy Code

```
for(i=1;i < numl;i++){
    for(int j=0;j < lsize[i];j++){
        for(int k=0;k < lsize[i-1];k++){
                prevDwt[i][j][k]=beta*delta[i][j]*out[i-1][k];
                weight[i][j][k]+=prevDwt[i][j][k];
        }
        prevDwt[i][j][lsize[i-1]]=beta*delta[i][j];
        weight[i][j][lsize[i-1]]+=prevDwt[i][j][lsize[i-1]];
    }
}
```

How learned is the net?

Mean square error is used as a measure of how well the **neural** net has learnt.

$$E(\vec{w}) = \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2$$

As shown in the sample XOR program, we apply the above steps until a satisfactorily low error level is achieved. `CBackProp::double mse(double *tgt)` returns just that.

History

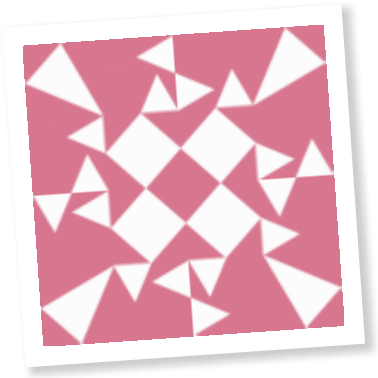
- Created date: 25th Mar'06.

License

Share

TWITTER FACEBOOK

About the Author



Tejpal Singh Chhabra

Software Developer

Australia

Follow this Member

A technology enthusiast with interest in all aspects of software development. I code in C, C++, Java and Go mostly on Unix/Linux.

You may also be interested in...

Neural Networks on C#

Get Started Turbo-Charging Your Applications with Intel® Parallel Studio XE

Neural Dot Net Pt 4 Neural Net Tester

Build an Autonomous Mobile Robot with the Intel® RealSense™ Camera, ROS, and SAWR

Optimizing Traffic for Emergency Vehicles using IOT and Mobile Edge Computing

SAPrefs - Netscape-like Preferences Dialog

Comments and Discussions

Add a Comment or Question

Search Comments

First Prev Next

to get reference generated current of three phase using back propagation control algorithm using matlab
Member 12111737 27-Nov-15 22:20

plz help to get this

Reply · Email · View Thread

Fine piece of code
SoothingMist 7-Jan-15 16:56

Found this work to be quite good. It is well commented and easy to follow.

A suggestion for the future would be to use spelled-out meaningful variable names and white space between components of executable lines.

Modifications for the application at hand were simple.

As the code appears to have been developed using an older version of Visual Studio and C++, it was necessary to create a completely new project and to make just a few modifications for Visual Studio Express v2013 and its use of C++ v11. Will see if it is possible to email the new code to the original author. He is welcome to post it.

Reply · Email · View Thread

Sigmoid derivative
Raphael Gruaz 2-Oct-12 18:12

Thanks a lot for this article. It really helped me to understand how to implement a neural network. However, there is something I still cannot understand from your code: I thought the derivative function of Sigmoid was something like DS(x) = S(x)(1-S(x)), but I cannot see it anywhere in your back-propagation function...

Did I miss something? Or did you use a simplified form?
Thanks again,

Raphael

Reply · Email · View Thread

Re: Sigmoid derivative
Annisa Kartikasari 7-Jul-15 13:35

same question with me, it just written in sigmoid, and there is no definition what is sigmoid in the code. Have you solve this by yourself?

Reply · Email · View Thread

Overflow
Miguel Tomas 6-Aug-12 1:06

on the traning code, the data array has only 8 elements how could be on a iteration with 200.000 steps
here's the code:

Hide Copy Code

```
for (long i=0; i < num_iter ; i++)
{
    bp->bpgt(data[i%8], &data[i%8][3]);

    if( bp->mse(&data[i%8][3]) < thresh)
        break; // mse < threshold - we are done training!!!
}
```

and also the second argument of bp->bpgt has to be an array and on the above code it isn't.

Can anyone help me with this?

regards

Reply · Email · View Thread

How to save the network?
tooym 21-Jan-10 11:05

Firstly,thanks to the author for providing me a very useful backpropagation neural network in C++. Regarding to the neural network as shown above, how can i save the network 'bp' permanently for the further testing with others input?
Thanks.

Reply · Email · View Thread

How to minimize Total Network Error
AjayIndian 1-Jan-10 17:21

Dear Sir,
I have developed the code for back-propagation algo.,but I m not able to minimize the Total Network Error(Global Error),How it can be done.
Actually after learning,when I m presenting the test pattern to the network,the network is giving the result as the last result given by the network for last training pattern.

looking for ur kind cooperation.

ajay

Reply · Email · View Thread

Understanding the basis for code
locuaz 17-Oct-09 16:34

Hi dude, your article is interesting and didactic, I'm focused on understanding your backpropagation algorithm, so I need you can tell me which books or sources you used to implement it into C++, please. I'm having serious problems to adopt/understand the equation tp update weigths, 'cause there are some books and papers using different representations for it, including you. For example, someone is using this formula:
weight(new) = weight(old) + learning rate * output error * output(neurons i) * output(neurons i+1) * (1 - output(neurons i+1))
And my doubt is, wtf he gets:
output(neurons i) * output(neurons i+1) * (1 - output(neurons i+1))
It doesnot appear in my books

Reference: Backpropagation
Thanks in advance.

I'm in love with Delphi

Billiardo

Reply · Email · View Thread

how can I get the source code
xumin1988 17-Oct-09 0:03

dear Sir,
I am a new user!I can find the demo file,but where is the source?Please help me.Thank you !

Reply · Email · View Thread

Bias value?
emrecaglar 13-Oct-09 3:30

I was wondering where the bias weights are set to 1 or -1? I think they are initialized to random values in constructor.Am i right? Many thanks

Reply · Email · View Thread

Out(0)
tangsu 1-Sep-09 20:10

Thank you for uploading your code for implementation of backpropagation neural networks.

I have a question about statement: ">> bp->Out(0) >> endl;"

Function "Out" has no reference anywhere in the code. Could you please clarify on this?

Tang Su

Reply · Email · View Thread

telecom billing
Sofritom 13-Jul-09 19:32

Hi Tejpal,

I wish to know about the telecom billing part - did you worked with FTS? what does is has to do with Neural network?

Reply · Email · View Thread

Error in the code [modified]
gpwr9k95 12-Jun-09 7:08

The code computes MSE for just one training value. Check these lines in main():

Hide Copy Code

```
for (i=0; i<num_iter ; i++)
{
    bp->bpgt(data[i%8], &data[i%8][3]);
    if( bp->mse(&data[i%8][3]) < Thresh) {
```

The learning iterations are arranged such that only one training value is checked at each iteration steps. As a result, for a net with one output, like in the article example, MSE simply becomes the squared error between the net output and the training value, i.e. (Net_Out - Target)^2. This is wrong because the iterations stop when MSE is less than Threshold for just one training value.
Instead, iterations should be arranged in epochs where one epoch contains the complete scan over all training values and MSE is calculated by summing squared errors between the net outputs and the corresponding training values for each epoch as in the example below (ntr - number of training samples):

Hide Copy Code

```
double MSE;
int ep;
for (ep=0;ep<nep;ep++)
```

```
{
    MSE=0.0;
    for(int i=0;i<ntr;i++)
    {
        bp->bpgt(data[i], &data[i][lSz[0]]);
        MSE+=bp->mse(&data[i][lSz[0]]);
    }
    MSE/=ntr;
    if(MSE<minErr)
    {
        cout << "Network trained in " << ep << " epochs. MSE: " << MSE << endl;
        break;
    }
```

Otherwise, the code is fairly simple and easy to understand.
Thanks.

modified on Friday, June 12, 2009 12:07 AM

Reply · Email · View Thread 5.00/5 (1 vote)

is this a bug?
Member 429450 19-Apr-09 9:18

for(i=1;i < numl;i++){
for(int j=0;j < lsize[i];j++){
for(int k=0;k < lsize[i-1];k++){
prevDwt[i][j][k]=beta*delta[i][j]*out[i-1][k];
weight[i][j][k]+=prevDwt[i][j][k];
}
prevDwt[i][j][lsize[i-1]]=beta*delta[i][j]; <----------here
weight[i][j][lsize[i-1]]+=prevDwt[i][j][lsize[i-1]];
}
}

i think should be..

for(i=1;i < numl;i++){
for(int j=0;j < lsize[i];j++){
for(int k=0;k <= lsize[i-1];k++) { <----------here
prevDwt[i][j][k]=beta*delta[i][j]*out[i-1][k];
weight[i][j][k]+=prevDwt[i][j][k];
}
}
}

I made similar modifications to the momentum code as well.

Another issue that i noticed is that iteration stops whenever the error for _one_ of the training data falls below the threshold.

Other than it works good.
Thanks

Reply · Email · View Thread

How about bias
KadirErturk 9-Mar-09 0:36

great code. Now I am trying to convert your code to complex number domain. I mean the element of net has complex number (i.e a+ib)

Would you correct me if am I wrong;

in your code
// Apply bias
sum+=weight[i][j][lsize[i-1]];
// Apply sigmoid function
....

so, is it mean that bias is constant for any layer. I think each neuron should have its own bias.

regards
Kadir

Reply · Email · View Thread

Normalized values
picand 6-May-08 20:09

Hello every one !
Does this algorithm works with values that are not between 0 and 1, or do I have to normalize them to implement them ?
Thank you.
PA

Reply · Email · View Thread

Feedforward Backpropagation Neural Network
raceng0585 7-Dec-07 23:41

In MatLab,

1st. we use purelin function p(x)=x where x is the summation of total weight multiple with bias that pass thought the output neuron in feedforward, but what is the dpurelin equation/formula that used in backpropagation?

2nd. we use logsig function logsig(x)=1/(1+e^(-x)) where x is the summation of total weight multiple with bias that pass thought the hidden neuron in feedforward method, but wat is the dlogsig equation/formula that used in backpropagation?

3rd. how to write the traingd or trainda function code in C style?

Reply · Email · View Thread

Trouble converting to C
pradeep swamy 19-Nov-07 16:37

This NN code is very fast. I'm converting this code to C, but having trouble while converting "***weight" variable to C. How to convert multiple pointer variable into C? Do you have a C equivalent code of this NN?

Reply · Email · View Thread

Re: Trouble converting to C
robiii 16-Jul-09 23:31

in c you cant use the new operator.
just use malloc

Reply · Email · View Thread

Scaling Input

ravenspoint 16-Oct-07 22:44

Although CBackProp will handle any range of input (unlike the outputs which must be in the range 0,1 - see my earlier post) I have found that if I also scale the inputs to the range 0,1 the results are improved, sometimes greatly.

The example below shows the results of experimenting with a time series generated from the function y = 0.5 sine(0.5 x)

Hide Expand Copy Code

```
Input       Actual       Pred       Error       %
  0.000000   0.479462   0.366899   0.112563  23.476872
  1.000000   0.488765   0.357821   0.130944  26.790794
  2.000000   0.378401   0.338155   0.040246  10.635803
  3.000000   0.175392   0.293070  -0.117679  67.094771
  4.000000  -0.070560   0.182579  -0.253139 -358.756813
  5.000000  -0.299236  -0.070652  -0.228584 -76.389083
  6.000000  -0.454649  -0.376426  -0.078223 -17.205125
  7.000000  -0.498747  -0.470585  -0.028163  -5.646709
  8.000000  -0.420735  -0.230091  -0.190645 -45.312249
  9.000000  -0.239713   0.271760  -0.511473 -213.368952
 10.000000   0.000000   0.328039  -0.328039   1.#INF00
 11.000000   0.239713   0.294158  -0.054445  22.712536
 12.000000   0.420735   0.164060   0.256676  61.006471
 13.000000   0.498747   0.069891   0.428856  85.986604
 14.000000   0.454649   0.018435   0.436214  95.945287
 15.000000   0.299236  -0.007738   0.306975 102.586070
 16.000000   0.070560  -0.020798   0.091358 129.475865
 17.000000  -0.175392  -0.027278  -0.148113 -84.447263
 18.000000  -0.378401  -0.030488  -0.347914 -91.943055
 19.000000  -0.488765  -0.032076  -0.456689 -93.437334
Maximum Absolute Error 0.511473


Input       Actual       Pred       Error       %
  0.000000   0.479462   0.469578   0.009884   2.061550
  0.052632   0.488765   0.447425   0.041340   8.458115
  0.105263   0.378401   0.384571  -0.006170   1.630557
  0.157895   0.175392   0.211022  -0.035630  20.314779
```

Reply · Email · View Thread 5.00/5 (1 vote)

MSE Calculation

ravenspoint 16-Oct-07 22:04

The calculation of mean square error looks strange to me.

Should we not divide by the number of outputs (instead of just 2 in every case) ?

I think the code in double CBackProp::mse(double *tgt) should be:

Hide Copy Code

```
double mse=0;
for(int i=0;i<lsize[numl-1];i++){
    mse+=(tgt[i]-out[numl-1][i])*(tgt[i]-out[numl-1][i]);
}
return mse/lsize[numl-1];
```

Reply · Email · View Thread

Re: MSE Calculation
brutjbro 23-Oct-07 21:25

I agree with you, "return mse/2;" should be replaced by "return mse/lsize[numl-1];".
File: BackProp.cpp, line 130.

btw It is a really good, simple and well written article.

Reply · Email · View Thread

Re: MSE Calculation
ravenspoint 23-Oct-07 22:48

Thank you for double checking my observation. It is always good to get somebody else to look at something like this, since I might well have been missing something. This is especially the case here, where the original author seems no longer to be around.

Yes, the article, and code, are a good, simple start to working with neural networks. However, the class CBackProp is really too simple to be useful by itself in serious work.

1. The undocumented scaling issues are a problem

2. The interface, which requires data to be passed in through pointers to elements of two dimensional arrays, is quite unfriendly.

3. The learning procedure in the sample application is extremely limited, and will not work with different input data.

4. There is no support for my own particular interest, time series analysis.

I am developing a wrapper class for CBackProp which addresses the above issues.

James

Reply · Email · View Thread

Re: MSE Calculation
brutjbro 25-Oct-07 3:57

Sorry, my previous statement is incorrect. MSE here is CORRECT.

I just realized, that this is not a standard MSE.
It was modified in Neural Networks theory to make it more simple to derive MSE function.
- simple example: (1/2 * (x)^2)' = 2 * 1/2 * x' = x'

Reply · Email · View Thread

Re: MSE Calculation
ravenspoint 29-Oct-07 3:16

I do not understand.

How can mean square error be anything other than the sum of errors squared, divided by total count?

What does x' mean in your simple example.

James

Reply · Email · View Thread 5.00/5 (1 vote)

Refresh 1 2 3 Next »

General News Suggestion Question Bug Answer Joke Praise Rant Admin