基于 C++的多层感知机神经网络

JY921127970142 网络空间安全

921127970142 汪哲男

联系方式： ericangean@163.com

189-0610-5703

摘要与声明

本文记录了基于 C++实现的多层感知机神经网络 MLP 的搭构原理细节与经过。

本文使用的MLP为三层结构，其中第一层是输入层，用来获取输入，即与输入维度对应，第二层为隐藏层，通过隐层神经元获取数据之间的隐藏规律。第三层是输出层，目的是将隐藏层获取的规律映射到输出数据上。这三层网络之间以全连接的方式组合，并且在模型的最后，本文使用了 softmax()函数完成分类操作。

本文并未使用任何第三方框架或代码。

问题种类：分类问题

921127970142 汪哲男

2022-10-29

多层感知机神经网络

1 数据

(1) 数据说明

数据样例是自己的构造的，用来测试模型的功能正确性。首先输入数据的维度是 3，每各个项的取值为 0 或 1（例如[0,0,1]），输出数据的维度为 8，代表输入的数据可以分为八类。测试样例构造的思想是将三位二进制转化为十进制数，例如输入[0,0,1]对应的输出结果应该是 1，输入[1,1,1]对应的结果应该是 7.

(2) 图片展示

数据的训练集如图 1 中所示。

```
double train_X[trainNum][n_in] = {
        {1, 1, 1},
        {1, 1, 0},
        {1, 0, 1},
        {1, 0, 0},
        {0, 1, 1},
        {0, 1, 0},
        {0, 0, 1},
        {0, 0, 0}
};
```

图 1 数据样例图

2 网络结构

(1) 网络结构图示

本文使用的 MLP 为三层结构，其中第一层是输入层，用来获取输入，即与输入维度对应，第二层为隐藏层，通过隐层神经元获取数据之间的隐藏规律。第三层是输出层，目的是将隐藏层获取的规律映射到输出数据上。这三层网络之间以全连接的方式组合，并且在模型的最后，本文使用了 softmax()函数完成分类操作。

模型具体内容如图 2 所示。

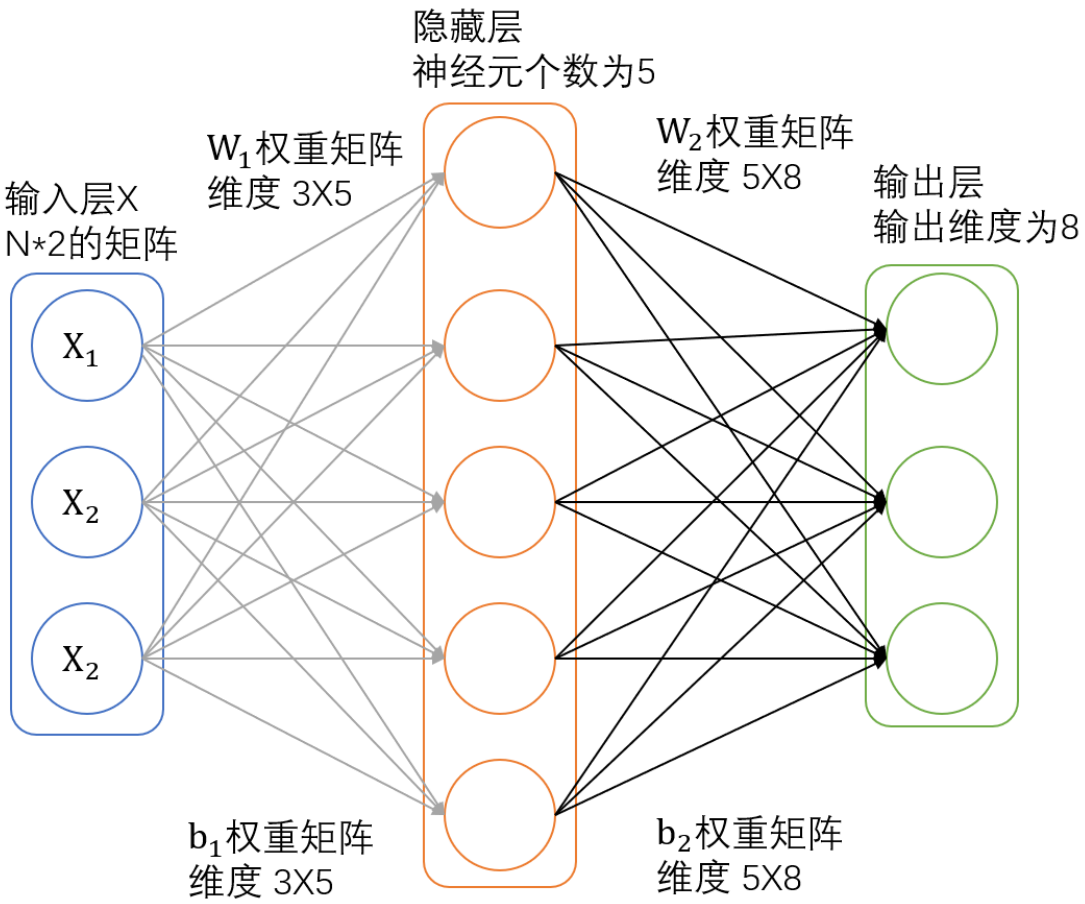

图 2 神经网络结构图

伪代码表示训练过程

Begin:

初始化 MLP 模型与超参数

TRAIN:

If 迭代次数 > 预设迭代值：

Break

Else:

读取待训练数据一条

前向传播：

第一层神经网络输入数据 1*3
放入激活函数 Relu
第二层隐层网络运算 1*3->1*5
放入激活函数 Relu
第三层输出层运算 1*5->1*8
Softmax 层概率运算
输出分类概率结果

反向传播：

计算全 softmax 函数局部梯度
计算第三层输出层梯度
修改第三层输出层权重与偏置
计算第二层隐藏层梯度
修改第二层隐藏层权重与偏置
计算第一层输入层梯度
修改第一层输入层权重与偏置
参数清零

TEST:

读取一条训练数据
将数据带入 MLP 神经网络
计算输出 8 个数字中，概率最大的下标
判断下标与标签是否相同
计算成功率

(2) 公式表示，输入、输出、参数、计算过程

该模型与一个有向无环图相关联，公式表示为：

$$f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$$

$f^{(1)}$是第一层网络，叫做输入层。$f^{(2)}$是第二层网络，叫做隐藏层。$f^{(3)}$是第三层网络，叫做输出层。

A、 输入层

在我们的例子中，输入层是输入的特征值，例如(1，1，1)、(1，1，0)、(0，0，1)，这是一个包含三个元素的数组，也可以看作是一个 1×3 的矩阵。输入层的元素维度与输入量的特征息息相关，如果输入的是一张 32×32 像素的灰度图像，那么输入层的维度就是 32×32。

B、 隐藏层

连接输入层和隐藏层的是W_1和b_1。由 X 计算得到 H 是线性矩阵运算，具体公式如下所示：

$$H = X \cdot W_1 + b_1$$

上面的公式类似于线性代数的公式，是线性矩阵运算，如图 2 中所示，在设定隐藏层为 50 维（也可以理解成 50 个神经元）之后，矩阵 H 的大小为 1X50 的矩阵。

连接隐藏层和输出层的是W_2和b_2。同样是矩阵运算生成的，只不过这次的输入是隐藏层的输出，故公式如下：

$$Y = H \cdot W_1 + b_1$$

从上述公式中可知，上述两个公式联立后可用一个线性方程表达。不仅对于两层的神经网络是这样，就算网络深度家到 100 层，也依然是这样，所以在这两层网络之间，需要一个中间的处理操作，使这两层网络有其单独特别的作用，所

以引入了激活层。在每个隐藏层计算（矩阵线性运算）之后，都需要加一层激活层，以此来实现两层网络的作用，本项目中，引入了 ReLU 激活函数，该函数的函数图像如图 3 所示。

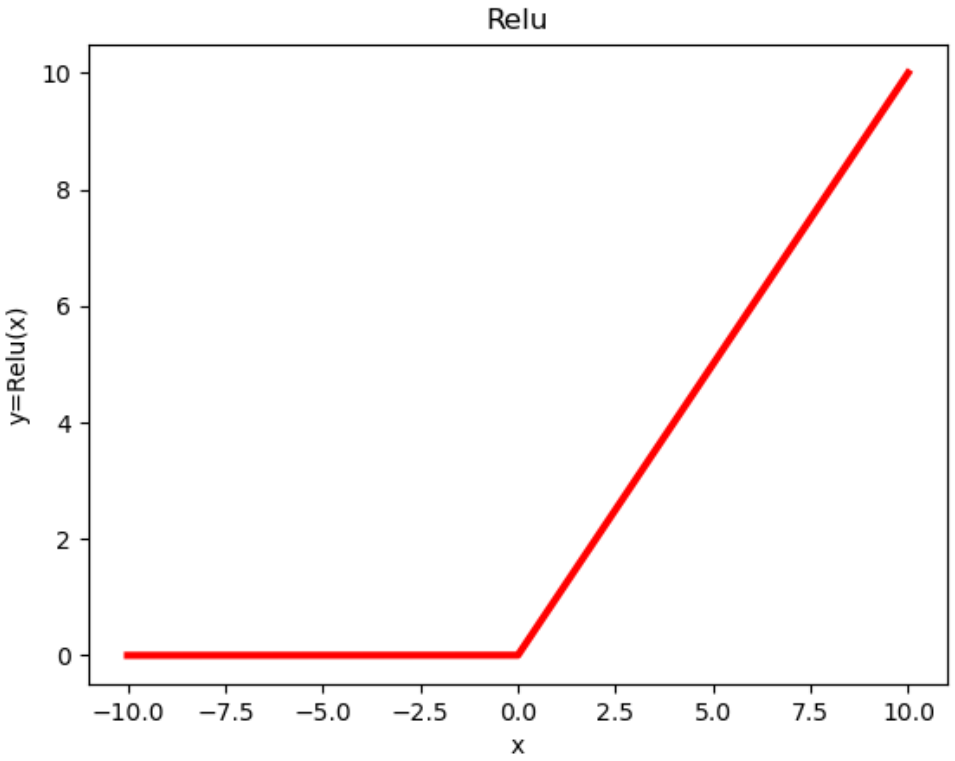

图 3 ReLU 函数图像

C、 输出层

输出的 Y 值是该项数据所属的类别可能性，数据所属的类别数据越高，该列属于当前类别的可能性越大。例如，输出的 Y 的值可能为(3,1,0.1,0.2,0,1,0.0,0.3)。可以直接从 Y 值中找到最大值，为 3，从而分到对应的类别，这在直观性和可操作性上差了很多。经过 softmax()函数最终的输出为概率，也就是说可以生成类似(86%,1%,5%,1%,3%,1%,1%,2%)的结果。其中 softmax()的公式如下：

$$S_i = \frac{e^i}{\sum_j e^j}$$

该公式可以分三步来说，一是以 e 为底对所有元素求指数幂；二时将所有指数幂数求和；三是分别将这些指数幂与该和做商。通过 softmax()函数处理之后输出结果的总和为 1，而且每个元素的结果可以代表概率值。Softmax()函数的图像如图 4 所示。

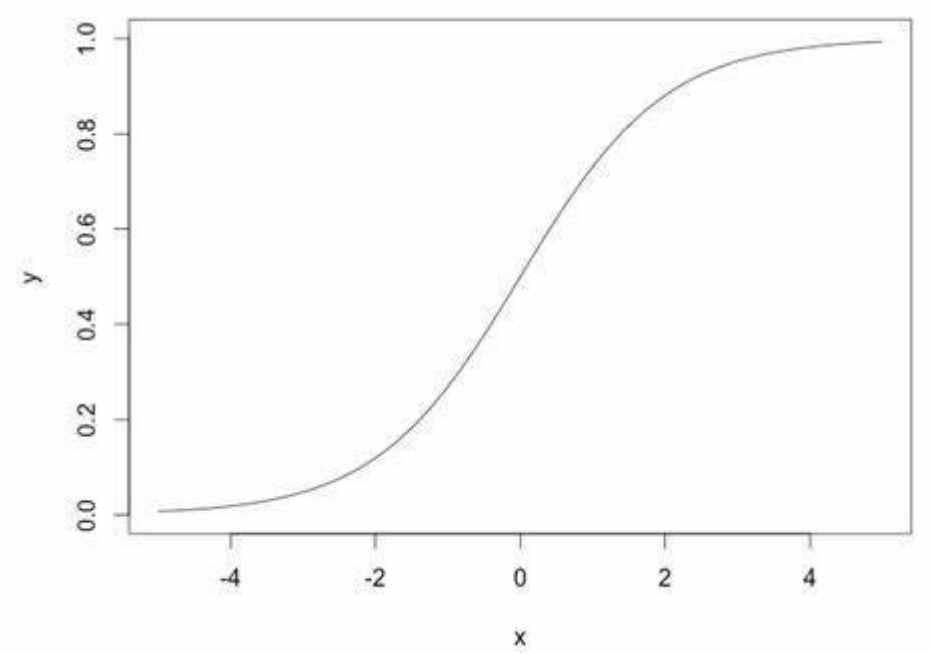

图 4 softmax 函数图像

但是在计算机中 Softmax 还要面对溢出问题，因为 Softmax 函数要进行指数运算而指数函数的值可能会变得非常大。比如，e^{100}的值就有 40 多个尾 0，e^{1000}的结果回直接返回表示无穷大的 inf。如果在这些大数间就行除法运算，很可能会出现错误的值。

所以 Softmax 的基本公式需要改进：

$$S_i = \frac{e^i}{\sum_j e^j} = \frac{C \cdot e^i}{C \cdot \sum_j e^j} = \frac{e^{i+\log C}}{\sum_j e^{j+\log C}}$$

$$= \frac{e^{i+C'}}{\sum_j e^{j+C'}}$$

首先，在原式的分子和分母上都乘上常数 C（C 为任意常数），再将 C 移到指数函数的幂上，记为$\log C$。最后将$\log C$替换为一个全新的符号C'。C'可以是任意常数，但为了防止溢出，我选择使用输入信号中的最大值。

D、 Softmax 层

Softmax 层的作用是将输入值进行正规化（其中输入中和调整为 1，反映其概率）后进行输出。Softmax()函数是分类任务通用的函数，本文将该任务看成分类任务，并使用 softmax()函数求其概率。最终得到的结果是某一项输入数据所命中的八个类别的概率。我们选择概率最大的一个类别作为模型分类的结果。

3　目标函数与优化

(1)　目标函数公式表示

Loss 层使用的是交叉熵误差（cross entropy error）这是一种适合分类任务的损失函数。该函数接收 softmax 的输出和监督标签，输出预测的损失误差。该函数的公式可以为：

$$\mathcal{L} = -\sum_{k} t_k log y_k$$

其中，y_k表示神经网络的输出，t_k则表示类别标签，k 表示数据的维度，log是以 e 为底数的自然对数。

(2)　传导原理及偏置导数推导

计算梯度下降则为计算$\mathcal{L}$对于每一个 W 和 b 中的每一个元素的偏导数。首先，其需求目标为：

$$\frac{\partial \mathcal{L}}{\partial W^k} = \frac{\partial \mathcal{L}}{\partial P^k} \cdot \frac{\partial P^k}{\partial \mathcal{W}^k}$$

$$\frac{\partial \mathcal{L}}{\partial b^k} = \frac{\partial \mathcal{L}}{\partial P^k} \cdot \frac{\partial P^k}{\partial b^k}$$

在公式中可以发现，两个等式之中均含有$\frac{\partial \mathcal{L}}{\partial P^k}$，应该先对其进行求偏导。假设：

$$\frac{\partial \mathcal{L}}{\partial P^k} = \delta^k = \left[\delta_1^k \delta_2^k \dots \delta_{D_k}^k\right]_{1\times D_k}$$

为了方便，只关注其中的第 i 个元素$\delta_i^k, 0 \le \mathrm{I} < \mathrm{n}$输出层的骗到将在后续单独进行讨论。

$$\delta_i^k = \frac{\partial \mathcal{L}}{\partial P_i^K}$$

$$= \frac{\partial \mathcal{L}}{\partial X_i^K} \cdot \frac{\partial X_i^K}{\partial P_i^K}$$

$$= \left(\frac{\partial \mathcal{L}}{\partial P_1^{K+1}} \cdot \frac{\partial P_1^{K+1}}{\partial X_i^K} + \frac{\partial \mathcal{L}}{\partial P_2^{K+1}} \cdot \frac{\partial P_2^{K+1}}{\partial X_i^K} + \cdots + \frac{\partial \mathcal{L}}{\partial P_{D_{k+1}}^{K+1}} \cdot \frac{\partial P_{D_{k+1}}^{K+1}}{\partial X_i^K}\right)\left(f'(P_i^K)\right)$$

$$= \left(\sum_{j=1}^{D_{k+1}} \frac{\partial \mathcal{L}}{\partial P_j^{K+1}} \cdot \frac{\partial P_j^{K+1}}{\partial X_i^K} \right) \left(f'(P_i^K) \right)$$

$$= \left(\sum_{j=1}^{D_{k+1}} \frac{\partial \mathcal{L}}{\partial P_j^{K+1}} \cdot \frac{\partial (\sum_{m=1}^{D_k} W_{j,m}^{k+1} \cdot X_m^k + b_j^{k+1}}{\partial X_i^K} \right) \left(f'(P_i^K) \right)$$

$$= \left(\sum_{j=1}^{D_{k+1}} \frac{\partial \mathcal{L}}{\partial P_j^{K+1}} \cdot \frac{\partial (W_{j,1}^{k+1} \cdot X_1^k + W_{j,2}^{k+1} \cdot + \cdots + W_{j,i}^{k+1} \cdot X_i^k + \cdots + W_{j,D_k}^{k+1} \cdot X_{D_k}^k + b_j^{k+1}}{\partial X_i^K} \right) \left(f'(P_i^K) \right)$$

$$= (\sum_{j=1}^{D_{k+1}} \delta_J^{k+1} \cdot W_{j,I}^{k+1}) \left(f'(P_i^K) \right)$$

$$= (\delta_J^{k+1} \cdot W_{:,I}^{k+1}) \left(f'(P_i^K) \right)$$

上式中$W_{:,I}^{k+1}$表示权重矩阵W^{k+1}的第 i 列，可以发现，元素δ_i^k是行向量δ^{k+1}和列向量$W_{:,I}^{k+1}$的点乘。结合 Element-wise 的传递函数，我们可以使用 Element-wise 符号 $\odot$ 对上式进行化简：

$$\delta^k = (\delta^{k+1} \cdot W^{k+1}) \odot f'(P^k)$$

从上式中已知的δ_i^k的表示方法，可以将δ_i^k视为已知，将其带入梯度中。接下来的的运算，仍然观察梯度矩阵中单个的元素：

$$\frac{\partial \mathcal{L}}{\partial \mathrm{W_{i,j}^k}} = \frac{\partial \mathcal{L}}{\partial \mathrm{P_i^k}} \cdot \frac{\partial P_i^K}{\partial W_{i,j}^K}$$

$$= \delta_i^k \cdot \frac{\partial (W_{i,1}^k \cdot X_1^{k-1} + W_{i,2}^k \cdot X_2^{k-1} + \cdots + W_{i,j}^k \cdot W_i^{k-1} + \cdots + W_{i,D_{k-1}}^k \cdot X_{D_{k-1}}^{k-1} + b_i^k)}{\partial W_{i,j}^K}$$

$$= \delta_i^k \cdot X_i^{k-1}$$

$$\frac{\partial \mathcal{L}}{\partial \mathrm{b_i^k}} = \frac{\partial \mathcal{L}}{\partial \mathrm{P_i^k}} \cdot \frac{\partial P_i^K}{\partial b_i^K}$$

$$= \delta_i^k \cdot \frac{\partial \left(W_{i,1}^k \cdot X_1^{k-1} + W_{i,2}^k \cdot X_2^{k-1} + \cdots + W_{i,j}^k \cdot W_i^{k-1} + \cdots + W_{i,D_{k-1}}^k \cdot X_{D_{k-1}}^{k-1} + b_i^k \right)}{\partial b_i^K}$$

$$= \delta_i^k$$

至此，以上即是完整表示 BP 过程中参数的反向更新过程。

4 程序说明

(1) 神经元、全连接层函数说明

隐藏层代码，主要是实现了隐藏层网络的全连接，并且定义了前向传播函数和反向传播函数，而且根据 MLP 公式实现了网络运行时的前向传播计算，而且实现了网络更新时的反向传播计算。具体代码如下所示：

```
#include <cmath>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include "HiddenLayer.h"
#include "util.h"

using namespace std;
HiddenLayer::HiddenLayer(int n_i, int n_o)
{
 n_in = n_i;
 n_out = n_o;

 w = new double* [n_out];
 for(int i = 0; i < n_out; ++i)
 {
  w[i] = new double [n_in];
 }
 b = new double [n_out];

 double a = 1.0 / n_in;

 srand((unsigned)time(NULL));
 for(int i = 0; i < n_out; ++i)
 {
  for(int j = 0; j < n_in; ++j)
   w[i][j] = uniform(-a, a);
  b[i] = uniform(-a, a);
 }

 delta = new double [n_out];
 output_data = new double [n_out];
}

HiddenLayer::~HiddenLayer()
{
```

```
37. for(int i=0; i<n_out; i++)
38.  delete []w[i];
39. delete[] w;
40. delete[] b;
41. delete[] output_data;
42. delete[] delta;
43. }
44. void HiddenLayer::forward_propagation(double* pdinputData)
45. {
46. for(int i = 0; i < n_out; ++i)
47.  {
48.  output_data[i] = 0.0;
49.  for(int j = 0; j < n_in; ++j)
50.  {
51.  output_data[i] += w[i][j]*pdinputData[j];
52.  }
53.  output_data[i] += b[i];
54.
55.  output_data[i] = sigmoid(output_data[i]);
56.  }
57. }
58.
59. void HiddenLayer::back_propagation(double *pdinputData, double *pdnextLaye
    rDelta,
60.       double** ppdnextLayerW, int iNextLayerOutNum, double dlr, int N)
61. {
62. /*
63. pdinputData     为输入数据
64. *pdnextLayerDelta  为下一层的残差值 delta,是一个大小为 iNextLayerOutNum
    的数组
65. **ppdnextLayerW   为此层到下一层的权值
66. iNextLayerOutNum  实际上就是下一层的 n_out
67. dlr          为学习率 learning rate
68. N          为训练样本总数
69. */
70.
71. //sigma 元素个数应与本层单元个数一致，而网上代码有误
72. //作者是没有自己测试啊，测试啊
73. //double* sigma = new double[iNextLayerOutNum];
74. double* sigma = new double[n_out];
75. //double sigma[10];
76. for(int i = 0; i < n_out; ++i)
77.  sigma[i] = 0.0;
78.
```

```
for(int i=0; i<iNextLayerOutNum; ++i)
{
 for(int j=0; j<n_out; ++j)
 {
  sigma[j] += ppdnextLayerW[i][j]*pdnextLayerDelta[i];
 }
}
//计算得到本层的残差 delta
for(int i=0; i<n_out; ++i)
{
 delta[i]=sigma[i]*output_data[i]*(1-output_data[i]);
}

//调整本层的权值 w
for(int i=0; i<n_out; ++i)
{
 for(int j=0; j<n_in; ++j)
 {
  w[i][j] += dlr*delta[i]*pdinputData[j];
 }
 b[i] += dlr*delta[i];
}
 delete[] sigma;
}


void HiddenLayer::writewb(const char *pcname)
{
 savewb(pcname, w, b, n_out, n_in);
}
long HiddenLayer::readwb(const char *pcname, long dstartpos)
{
 return loadwb(pcname, w, b, n_out, n_in, dstartpos);
}
```

有了全连接的隐藏层部分，我们只需要将已有的隐藏层部分与 softmax 激活函数相连接，构造完整的 MLP 网络即可，代码如下所示：

```
#include <iostream>
#include "NeuralNetwork.h"
#include "util.h"
//#include "HiddenLayer.h"
//#include "LogisticRegression.h"

```

```
using namespace std;

const int n_train = 8, innode = 3, outnode = 8;
NeuralNetwork::NeuralNetwork(int n, int n_i, int n_o, int nhl, int *hls)
{
 N = n;
 n_in = n_i;
 n_out = n_o;

 n_hidden_layer = nhl;
 hidden_layer_size = hls;

 //构造网络结构
 sigmoid_layers = new HiddenLayer* [n_hidden_layer];
 for(int i = 0; i < n_hidden_layer; ++i)
 {
  if(i == 0)
  {
   sigmoid_layers[i] = new HiddenLayer(n_in, hidden_layer_size[i]);//第一个隐层
  }
  else
  {
   sigmoid_layers[i] = new HiddenLayer(hidden_layer_size[i-1], hidden_layer_size[i]);//其他隐层
  }
 }

 log_layer = new LogisticRegression(hidden_layer_size[n_hidden_layer-1], n_out, N);//最后的 softmax 层
}

NeuralNetwork::~NeuralNetwork()
{
 //二维指针分配的对象不一定是二维数组
 for(int i = 0; i < n_hidden_layer; ++i)
  delete sigmoid_layers[i]; //删除的时候不能加[]
 delete[] sigmoid_layers;
 //log_layer 只是一个普通的对象指针，不能作为数组 delete
 delete log_layer;//删除的时候不能加[]
}

void NeuralNetwork::train(double** ppdinData, double** ppdinLabel, double dlr, int iepochs)
```

```
{
printArrDouble(ppdinData, N, n_in);


cout << "******label****" << endl;
printArrDouble(ppdinLabel, N, n_out);

//反复迭代样本 iepochs 次训练
for(int epoch = 0; epoch < iepochs; ++epoch)
{
 double e = 0.0;
 for(int i = 0; i < N; ++i)
 {
  //前向传播阶段
  for(int n = 0; n < n_hidden_layer; ++ n)
  {
   if(n == 0) //第一个隐层直接输入数据
   {
    sigmoid_layers[n]->forward_propagation(ppdinData[i]);
   }
   else //其他隐层用前一层的输出作为输入数据
   {
    sigmoid_layers[n]->forward_propagation(sigmoid_layers[n-
1]->output_data);
   }
  }
  //softmax 层使用最后一个隐层的输出作为输入数据
  log_layer->forward_propagation(sigmoid_layers[n_hidden_layer-
1]->output_data);

  //e += log_layer->cal_error(ppdinLabel[i]);

  //反向传播阶段
  log_layer->back_propagation(sigmoid_layers[n_hidden_layer-
1]->output_data, ppdinLabel[i], dlr);

  for(int n = n_hidden_layer-1; n >= 1; --n)
  {
   if(n == n_hidden_layer-1)
   {
    sigmoid_layers[n]->back_propagation(sigmoid_layers[n-1]->output_data,
     log_layer->delta, log_layer->w, log_layer->n_out, dlr, N);
   }
   else
```

```
88.    {
89.     double *pdinputData;
90.     pdinputData = sigmoid_layers[n-1]->output_data;
91.
92.     sigmoid_layers[n]->back_propagation(pdinputData,
93.      sigmoid_layers[n+1]->delta, sigmoid_layers[n+1]->w, sigmoid_layers[n+1]->n_out, dlr, N);
94.    }
95.   }
96.   //这里该怎么写?
97.   if (n_hidden_layer > 1)
98.    sigmoid_layers[0]->back_propagation(ppdinData[i],
99.    sigmoid_layers[1]->delta, sigmoid_layers[1]->w, sigmoid_layers[1]->n_out, dlr, N);
100.    else
101.     sigmoid_layers[0]->back_propagation(ppdinData[i],
102.     log_layer->delta, log_layer->w, log_layer->n_out, dlr, N);
103.   }
104.   //if (epoch % 100 == 1)
105.   //cout << "iepochs number is " << epoch << " cost function is " << e / (double)N << endl;
106.  }
107. }
108. void NeuralNetwork::predict(double** ppdata, int n)
109. {
110.  for(int i = 0; i < n; ++i)
111.  {
112.   for(int n = 0; n < n_hidden_layer; ++n)
113.   {
114.    if(n == 0) //第一个隐层直接输入数据
115.    {
116.     sigmoid_layers[n]->forward_propagation(ppdata[i]);
117.    }
118.    else //其他隐层用前一层的输出作为输入数据
119.    {
120.     sigmoid_layers[n]->forward_propagation(sigmoid_layers[n-1]->output_data);
121.    }
122.   }
123.   //softmax 层使用最后一个隐层的输出作为输入数据
124.   log_layer->predict(sigmoid_layers[n_hidden_layer-1]->output_data);
125.   //log_layer->forward_propagation(sigmoid_layers[n_hidden_layer-1]->output_data);
126. }
```

```
127. }
128.
129. void NeuralNetwork::writewb(const char *pcname)
130. {
131.  for(int i = 0; i < n_hidden_layer; ++i)
132.  {
133.   sigmoid_layers[i]->writewb(pcname);
134.  }
135.  log_layer->writewb(pcname);
136.
137. }
138. void NeuralNetwork::readwb(const char *pcname)
139. {
140.  long dcurpos = 0, dreadsize = 0;
141.  for(int i = 0; i < n_hidden_layer; ++i)
142.  {
143.   dreadsize = sigmoid_layers[i]->readwb(pcname, dcurpos);
144.   cout << "hiddenlayer " << i + 1 << " read bytes: " << dreadsize << endl;
145.   if (-1 != dreadsize)
146.    dcurpos += dreadsize;
147.   else
148.   {
149.    cout << "read wb error from HiddenLayer" << endl;
150.    return;
151.   }
152.  }
153.  dreadsize = log_layer->readwb(pcname, dcurpos);
154.  if (-1 != dreadsize)
155.   dcurpos += dreadsize;
156.  else
157.  {
158.   cout << "read wb error from sofmaxLayer" << endl;
159.   return;
160.  }
161. }
162. //double **makeLabelSample(double **label_x)
163. double **makeLabelSample(double label_x[][outnode])
164. {
165.  double **pplabelSample;
166.  pplabelSample = new double*[n_train];
167.  for (int i = 0; i < n_train; ++i)
168.  {
169.   pplabelSample[i] = new double[outnode];
170.  }
```

```
171.
172.  for (int i = 0; i < n_train; ++i)
173.  {
174.   for (int j = 0; j < outnode; ++j)
175.    pplabelSample[i][j] = label_x[i][j];
176.  }
177.  return pplabelSample;
178. }
179. double **maken_train(double train_x[][innode])
180. {
181.  double **ppn_train;
182.  ppn_train = new double*[n_train];
183.  for (int i = 0; i < n_train; ++i)
184.  {
185.   ppn_train[i] = new double[innode];
186.  }
187.
188.  for (int i = 0; i < n_train; ++i)
189.  {
190.   for (int j = 0; j < innode; ++j)
191.    ppn_train[i][j] = train_x[i][j];
192.  }
193.  return ppn_train;
194. }
195. void mlp()
196. {
197.  //输入样本
198.  double X[n_train][innode]= {
199.   {0,0,0},{0,0,1},{0,1,0},{0,1,1},{1,0,0},{1,0,1},{1,1,0},{1,1,1}
200.  };
201.  double Y[n_train][outnode]={
202.   {1, 0, 0, 0, 0, 0, 0, 0},
203.   {0, 1, 0, 0, 0, 0, 0, 0},
204.   {0, 0, 1, 0, 0, 0, 0, 0},
205.   {0, 0, 0, 1, 0, 0, 0, 0},
206.   {0, 0, 0, 0, 1, 0, 0, 0},
207.   {0, 0, 0, 0, 0, 1, 0, 0},
208.   {0, 0, 0, 0, 0, 0, 1, 0},
209.   {0, 0, 0, 0, 0, 0, 0, 1},
210.  };
211.  const int ihiddenSize = 2;
212.  int phidden[ihiddenSize] = {5, 5};
213.  //printArr(phidden, 1);
214.  NeuralNetwork neural(n_train, innode, outnode, ihiddenSize, phidden);
```

```
215.  double **train_x, **ppdlabel;
216.  train_x = maken_train(X);
217.  //printArrDouble(train_x, n_train, innode);
218.  ppdlabel = makeLabelSample(Y);
219.  neural.train(train_x, ppdlabel, 0.1, 3500);
220.  cout<<"trainning complete..."<<endl;
221.  //pcname 存放权值
222.  const char *pcname = "mlp55new.wb";
223.  neural.writewb(pcname);
224.  NeuralNetwork neural2(n_train, innode, outnode, ihiddenSize, phidden);
225.  cout<<"readwb start..."<<endl;
226.  neural2.readwb(pcname);
227.  cout<<"readwb end..."<<endl;
228.  neural.predict(train_x, n_train);
229.  cout << "----------after readwb_________" << endl;
230.  neural2.predict(train_x, n_train);
231.  for (int i = 0; i != n_train; ++i)
232.  {
233.   delete []train_x[i];
234.   delete []ppdlabel[i];
235.  }
236.  delete []train_x;
237.  delete []ppdlabel;
238.  cout<<endl;
239. }
```

(2)　主要步骤说明

前向传播，求取网络中值的传递：

```
1.  void HiddenLayer::forward_propagation(double* pdinputData)
2.  {
3.      for(int i = 0; i < n_out; ++i)
4.      {
5.          output_data[i] = 0.0;
6.          for(int j = 0; j < n_in; ++j)
7.          {
8.              output_data[i] += w[i][j]*pdinputData[j];
9.          }
10.         output_data[i] += b[i];
11.
12.         output_data[i] = sigmoid(output_data[i]);
13.     }
```

```
14.}
```

反向传播，调整权重：

```
1.      //计算得到本层的残差 delta
2.      for(int i = 0; i < n_out; ++i)
3.      {
4.          delta[i] = sigma[i] * output_data[i] * (1 - output_data[i
    ]);
5.      }
6.
7.      //调整本层的权值 w
8.      for(int i = 0; i < n_out; ++i)
9.      {
10.         for(int j = 0; j < n_in; ++j)
11.         {
12.             w[i][j] += dlr * delta[i] * pdinputData[j];
13.         }
14.         b[i] += dlr * delta[i];
15.     }
```

构造网络结构：

```
1.      sigmoid_layers = new HiddenLayer* [n_hidden_layer];
2.      for(int i = 0; i < n_hidden_layer; ++i)
3.      {
4.          if(i == 0)
5.          {
6.              sigmoid_layers[i] = new HiddenLayer(n_in, hidden_laye
    r_size[i]);//第一个隐层
7.          }
8.          else
9.          {
10.             sigmoid_layers[i] = new HiddenLayer(hidden_layer_size
    [i-1], hidden_layer_size[i]);//其他隐层
11.         }
12.     }
13.
14.     log_layer = new LogisticRegression(hidden_layer_size[n_hidden
    _layer-1], n_out, N);//最后的 softmax 层
```

5 实验分析

模型中定义了一些超参，这些参数可以调节模型的层数以及神经网络的个数和训练时的一些设置(例如训练次数，学习率等)。下面将进行实验，探究一下主

要超参数对结果的影响并在最后给出截图展示，首先我们将统一定义一下超参数的值为：

```
1.      const int ihiddenSize = 2;
2.      int phidden[ihiddenSize] = {5, 5};
3.      int lr = 0.1;
4.      int epochs = 3500;
```

(1) 学习率与网络性能的关系

学习率	正确率
5e-2	9.55%
3e-2	90.68%
1e-2	95.17%
3e-3	95.38%
3e-5	45.7%
3e-7	9.59%

本次测试中使用了一千个训练数据，每个训练数据的迭代次数为 100，分别设置不同的学习率看起效果。在模型中每层之键都使用 ReLU 函数作为激活函数，从以上的实验中可以发现当学习率在 0.01 到 0.003 之间时效果最好。

(2) 网络层数与正确率的关系

网络层数	正确率
1 输入层 1 隐藏层 1 输出层	99.21%
1 输入层 2 隐藏层 1 输出层	99.96%
1 输入层 3 隐藏层 1 输出层	99.96%

本次测试中使用了 1000 个训练数据，每个训练数据迭代的次数为 100，分别设置网络不同的隐藏层观察结果不同的表现。从图中的结果可以看出增加隐藏层的层数可以增加效果，但是隐藏层增加到一定层数之后效果不再有提升。这说明，一味的增加隐藏层的效果不能提升模型分类的能力。

(3) 模型准确率随训练次数的变化效果

本次测试中，使用了不同的训练次数，观察模型的准确率，可以看出来模型的准确率随训练次数不断提高，但训练次数在 80 次之后，效果提升不大。如图所示。

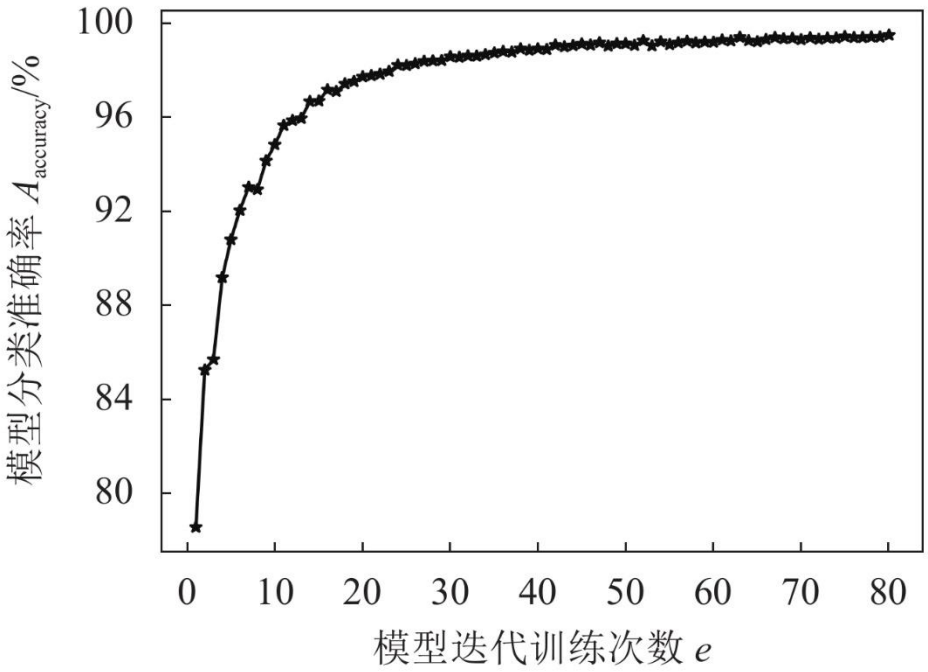

图 5 准确率随训练次数的变化

(4) 模型运行效果

```
****softmax****
before readwb ---------
***result is ***
5

***result is ***
1
****printArr****
0.0816572 0.343983 0.0613494 0.170706 0.0486719 0.16843 0.033765 0.0914372

after readwb ---------
***result is ***
7

***result is ***
6

***result is ***
5

***result is ***
4

***result is ***
3
```

图 6 运行截图 1

```
****printArr****
0.0975992 0.304031 0.0413843 0.201707 0.0417687 0.204063 0.0268654 0.0825823

***result is ***
0

*********
****mlp****
****printArrDouble****
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1
******label****
****printArrDouble****
1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0
```

图 7 运行截图 2