C++搭建多层 BP 神经网络

装

订

线

摘 要

反向传播（Backpropagation,BP）是一种与最优化方法结合使用的，用来训练人工神经网络的常见方法。本实验将以反向传播算法为在神经网络中的原理和实现为主要研究方向，使用C++从网络最基础的神经元开始搭建，最终实现包含输入层，层数和节点可选的隐藏层、输出层整个完整的神经网络的搭建。本实验将详细阐述各层之间激励前向传播和后向传播的数学原理和推导过程、以及神经网络中每个突触上权重更新的过程，并给出相应的代码实现。报告中将详细展现实验过程中对算法的思考过程和创新的内容。为了验证实现神经网络的效果，本实验使用 Minist 手写数字数据集，对模型进行训练和预测。实验中实现了对书写数字数据的预处理以转化为神经网络可以接受的输入。同时，还选取测试集中部分手写数字数据进行由像素点阵绘制拼接成的图片展示，并与预测结果进行对比，进行可视化展示。实验还实现了对搭建的神经网络的可视化展示。最终可以实现指定隐藏层数和隐藏层节点数、指定训练轮次、指定学习率、指定 batch_size，指定查看训练模型节点的偏置值和突触权重等功能。

关键词：BP 算法，多层神经网络，前向传播，后向传播，Minist 数据集，C++

C++ build multilayer BP neural network

ABSTRACT

Backpropagation (BP) is a common method used in combination with optimization methods to train artificial neural networks. In this experiment, the principle and implementation of back propagation algorithm in neural network will be the main research direction, and c++ will be used to build from the most basic neuron of the network. Finally, the whole neural network including input layer, layer number and node optional hidden layer and output layer will be built. This experiment will elaborate the mathematical principle and derivation process of forward and backward propagation of excitation between layers, as well as the process of weight update on each synapse in the neural network, and give the corresponding code implementation. The report will show in detail the thinking process and innovation of the algorithm during the experiment. In order to verify the effect of neural network, this experiment uses Minist handwritten numeral data set to train and predict the model. In the experiment, the preprocessing of written digital data is realized to convert it into the acceptable input of neural network. At the same time, some handwritten digital data in the test set are selected for the picture display which is drawn and spliced by pixel matrix, and compared with the prediction results for visual display. The experiment also realized the visual display of the neural network. Finally, it can specify the number of hidden layers and hidden layer nodes, specify training rounds, specify learning rate, and specify batch_ Size to specify functions such as viewing the offset value and synaptic weight of the training model node.

Key words： BP algorithm, multilayer neural network, forward propagation, backward propagation, Minist dataset, C++

装 订 线

目录

1 实验总述

1.1 实验目的

本项目的目的是使用 C++架构出一个使用 BP 算法的多层神经网络，其中可以设置多个隐藏层，并且每个隐藏层节点可以单独选择。而神经网络训练的轮次，学习率，batch_size 等可以手动修改调参。本项目可以帮助我们深入理解 BP 神经网络的内部原理和实现以及优化的方案，建立对人工智能神经网络这一重要的方法的认识和见解。为验证模型的效果，我们使用 Minist 手写数字数据集进行训练和预测，并对预测结果和训练的网络进行可视化展示。

1.2 项目功能需求描述

（1）建立神经网络中基础的神经元
（2）建立输入层和输出层
（3）建立参数可调的隐藏层
（4）实现激励的正向传播和反向传播算法，以及权值和偏置值更新的过程
（5）实现对 Minist 数据集的读入和预处理
（6）由数据对网络进行训练，建立模型并对测试集进行预测，展示输出训练和预测的过程
（7）可视化展示手写数字识别的预测结果，可视化展示网络的训练结果，帮助深入理解神经网络的建立过程和预测结果

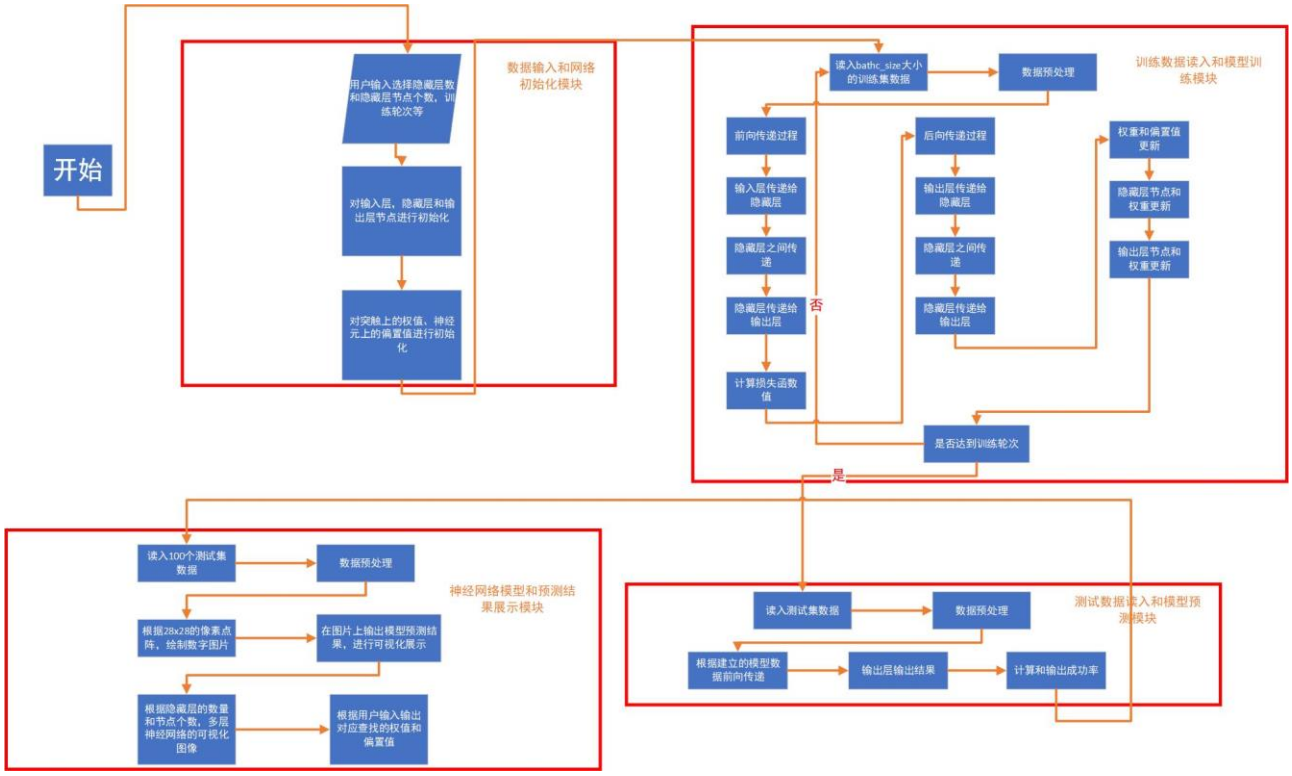
1.3 项目开发环境

1）IDE : Visual Studio2022

2）库依赖：<iostream>，<cmath>，<cstring>，<fstream>，<iomanip>，<graphics.h>，<conio.h>，<vector>

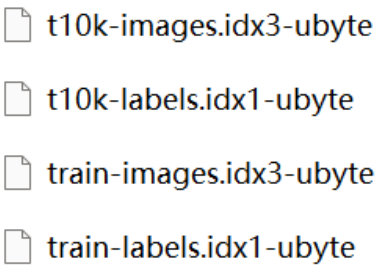
1.4 项目系统结构框图

将整个项目系统分为四大部分结构：
（1）数据输入和网络初始化模块
（2）训练数据输入和模型训练模块
（3）测试数据输入和模型预测模块
（4）神经网络模型和预测结果可视化展示模块
框图展示如下：



1.5 所用数据描述

本项目中使用的数据集为 MNIST 数据集（Mixed National Institute of Standards and Technology database），该数据集是美国国家标准与技术研究院收集整理的大型手写数据库，包含 600000 个示例的训练集以及 10000 个示例的测试集。该数据集中包含的文件如下：



其中，train-images-idx3-ubyte 的内容为 55000 张训练集和 5000 张验证集，train-labels-idx1-ubyte 的内容为训练集图片对应的标签，t10k-images-idx3-ubyte 的内容为 10000 张测试集，t10k-labels-idx1-ubyte 的内容为测试机图片对应的标签。

训练集和测试集的文件数据格式如下：

第 0~3 字节，是 32 位整型数据，取值为 0x00000801，用于记录文件数据格式。

第 4~7 字节，是 32 位整型数据，取值为 60000（训练集时）或 10000（测试集时），用于记录标签数据的个数。

第 8 字节~)，是一个无符号整型的数，取值为 0-9 之间的标签数字，用于记录样本的标签。

训练集和测试集的图像文件格式如下：

第 0~3 字节，是 32 位整型数据，取值为 0x00000803，用于记录文件的格式。

第 4~7 字节，是 32 位整型数据，取值为 60000（训练集时）或 10000（测试集时），用于记录图片数据的个数。

第 8~11 字节，是 32 位整型数据，取值位 28，用来记录图片数据的高度。

第 12~15 字节，是 32 位整型数据，取值为 28，用来记录图片数据的宽度。

第 16 字节~)，是一个无符号型的数，取值为 0-255 之间的灰度值，用来记录图片按行展开后得到的灰度值数据，其中 0 表示背景（白色），255 表示前景（黑色）。

2 采用算法原理及步骤

2.1 算法总体流程

本项目中采用的算法为反向传播（Backpropagation，缩写为 BP）算法。反向传播是“误差反向传播”的简称，是一种与最优化方法结合使用的，用来训练人工神经网络的常见方法。

BP 算法由信号的正向传播和误差的反向传播两个过程组成。

正向传播时，输入样本从输入层进入网络，经隐藏层逐层传递至输出层，如果输出层的实际输出与期望输出不同，则转至误差反向传播；如果输出层的实际输出与期望输出相同，则结束学习算法。

反向传播时，将输出误差（期望输出与实际输出之差）按原通路反传计算，通过隐藏层反向，直至输入层，在反传的过程中将误差分摊给各层的各个单元，获得各层各单元的误差信号，并将其作为修正各单元权值的根据。这一计算过程使用梯度下降法完成，在不停地调整各层神经元的权值和阈值后，使误差信号减小到最低限度。

权值和阈值不断调整的过程，就是网络的学习与训练的过程，经过信号正向传播与误差反向传播，权值和阈值的调整反复进行，一直进行到预先设定的学习训练次数，或输出误差减小到允许的程度。

2.2 算法推导过程

算法中采用 Sigmoid 函数作为激活函数，其表达式如下：

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

Sigmoid 函数的输出范围为 0 到 1。由于输出值限定在 0 到 1，因此它可以对每个神经元的输出进行归一化。

Sigmoid 函数的导数为：

$$\frac{d}{dx}\sigma(x) = \frac{d}{dx}\left(\frac{1}{1+e^{-x}}\right) = \frac{e^{-x}}{(1+e^{-x})^2} = \sigma(x) - \sigma(x)^2 = \sigma(1-\sigma)$$

算法中采用均方差损失函数，其表达式为：

$$L = \frac{1}{2}\sum_{k=1}^{K}(y_k - o_k)^2$$

其中y_k为真实值，o_k为输出值。

损失函数的偏导数可以展开为：

$$\frac{\partial L}{\partial o_i} = \frac{1}{2}\sum_{k=1}^{K}\frac{\partial}{\partial o_i}(y_k - o_k) = \frac{1}{2}\sum_{k=1}^{K}\cdot 2\cdot(y_k - o_k)\cdot\frac{\partial(y_k - o_k)}{\partial o_i}$$

化简结果为

$$\frac{\partial L}{\partial o_i} = (o_i - y_i)$$

在全连接层中，最后一个隐藏层到输出层之间的权重和损失函数的关系如下：

全连接层中，有多个输出结点$o_1^1, o_2^1, o_3^1, \ldots, o_K^1$，每个输出结点对应不同真实标签 $t_1, t_2, t_3, \ldots t_K$，均方误差可以表示为

$$L = \frac{1}{2}\sum_{i=1}^{K}(o_i^1 - t_i)^2$$

由于 $\partial L/\partial w_{jk}$ 只与 o_k^1 有关联，

$$\frac{\partial L}{\partial w_{jk}} = (o_k - t_k)\frac{\partial o_k}{\partial w_{jk}}$$

将$o_k = \sigma(z_k)$带入

$$\frac{\partial L}{\partial w_{jk}} = (o_k - t_k)\frac{\partial \sigma(z_k)}{\partial w_{jk}}$$

将$\sigma' = \sigma(1-\sigma)$带入

$$\frac{\partial L}{\partial w_{jk}} = (o_k - t_k)\sigma(z_k)\big(1-\sigma(z_k)\big)\frac{\partial z_k^1}{\partial w_{jk}}$$

将$\sigma(z_k)$记为o_k

$$\frac{\partial L}{\partial w_{jk}} = (o_k - t_k) = (o_k - t_k)o_k(1-o_k)\frac{\partial z_k^1}{\partial w_{jk}}$$

最终可得

$$\frac{\partial L}{\partial w_{jk}} = (o_k - t_k) = (o_k - t_k)o_k(1-o_k)\cdot x_j$$

对该式令$\delta_k = (o_k - t_k)o_k(1-o_k)$，则$\partial L/\partial w_{jk}$可以表述为

$$\frac{\partial L}{\partial w_{jk}} = \delta_k \cdot x_j$$

继续向前推导，可得倒数第三层到倒数第二层的权重和损失函数的关系如下：

$$\frac{L}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\frac{1}{2}\sum_k (o_k - t_k)^2$$

将$o_k = \sigma(z_k)$和$\sigma' = \sigma(1-\sigma)$带入

$$\frac{\partial L}{\partial w_{ij}} = \sum_k (o_k - t_k)\, o_k(1-o_k)\frac{\partial z_k}{\partial w_{ij}}$$

对于$\partial z_k/\partial w_{ij}$可以应用链式法则分解为

$$\frac{z_k}{\partial w_{ij}} = \frac{\partial z_k}{o_j}\cdot\frac{\partial o_j}{\partial w_{ij}}$$

根据$z_k = o_j \cdot w_{jk} + b_k$

$$\frac{\partial L}{\partial w_{ij}} = \sum_k (o_k - t_k)\, o_k(1-o_k)w_{jk}\cdot\frac{\partial o_j}{\partial w_{ij}}$$

根据$\sigma' = \sigma(1-\sigma)$和$z_j = o_i \cdot w_{ij} + b_j$

$$\frac{\partial L}{\partial w_{ij}} = o_j\big(1-o_j\big)o_i \cdot \sum_k (o_k - t_k)\, o_k(1-o_k)w_{jk}$$

令$\delta_k^K = (o_k - t_k)o_k(1-o_k)$，则

$$\frac{\partial L}{\partial w_{ij}} = o_j(1-o_j)o_i \cdot \sum_k \delta_k^K \cdot w_{jk}$$

若令

$$\delta_j^J = o_j(1-o_j) \cdot \sum_k \delta_k^K \cdot w_{jk}$$

则

$$\frac{\partial L}{\partial w_{ij}} = \delta_j^J \cdot o_i^I$$

根据以上推导，可以得出以下结论

输出层

$$\frac{\partial L}{\partial w_{jk}} = \delta_k \cdot o_j$$

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

倒数第二层

$$\frac{\partial L}{\partial w_{ij}} = \delta_j^J \cdot o_i$$

$$\delta_j^J = o_j(1-o_j) \cdot \sum_k \delta_k^K \cdot w_{jk}$$

倒数第三层

$$\frac{\partial L}{\partial w_{ni}} = \delta_i^I \cdot o_n$$

$$\delta_i^I = o_i(1-o_i) \cdot \sum_j \delta_j^J \cdot w_{ij}$$

根据以上规律，只需要循环迭代计算每一层每个结点的$\delta_k^K, \delta_j^J, \delta_i^I, \ldots$，从而可以求得当前层的偏导数，得到每层各个权值的梯度。

3 实验结果可视化展示

3.1 训练过程可视化

3.1.1 用户输入和神经网络配置

在本项目中，隐藏层的个数和每个隐藏层的结点数和训练过程中训练的轮数都是可以任意设置的，因此将其设置为用户输入的方式。

运行程序后，先由用户输入这三项内容。

```
请输入隐藏层的层数：2
请输入第1个隐藏层的结点数：20
请输入第2个隐藏层的结点数：10
请输入训练轮次：3
```

其余输入层的结点数、输出层的结点数、学习率、训练时每一批的数据个数通过配置文件的方式进行设置。

```
const int INPUT_NODE_NUM = 784; //输入层的结点数
const int OUTPUT_NODE_NUM = 10; //输出层的结点数
const double LEARNING_RATE = 0.35; //学习率
const int BATCH_SIZE = 1000; //训练时每一批的数据个数
```

3.1.2 训练即时数据可视化

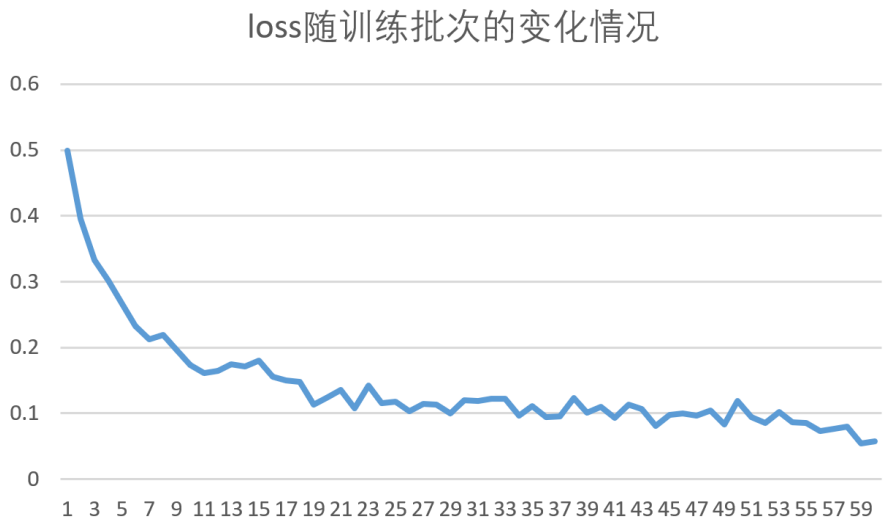
训练过程中，每一轮训练中分批次对神经网络进行训练，其中，每一批数据中包含 1000 个数据。

每一批数据训练完成后，对该批次中所有数据的平均 loss 值进行计算并输出。

```
#epoch 1 Training peocess:
cnt: 1000  loss:0.447251
cnt: 2000  loss:0.374471
cnt: 3000  loss:0.295668
cnt: 4000  loss:0.250174
cnt: 5000  loss:0.219882
cnt: 6000  loss:0.19193
cnt: 7000  loss:0.170199
cnt: 8000  loss:0.186691
cnt: 9000  loss:0.179295
cnt: 10000 loss:0.136614
cnt: 11000 loss:0.131497
cnt: 12000 loss:0.142357
cnt: 13000 loss:0.136705
cnt: 14000 loss:0.138246
cnt: 15000 loss:0.158291
cnt: 16000 loss:0.127885
cnt: 17000 loss:0.124712
cnt: 18000 loss:0.132856
cnt: 19000 loss:0.0998159
cnt: 20000 loss:0.101881
cnt: 21000 loss:0.11744
cnt: 22000 loss:0.0925679
cnt: 23000 loss:0.11122
```

```
cnt: 47000 loss:0.0936529
cnt: 48000 loss:0.102678
cnt: 49000 loss:0.0849448
cnt: 50000 loss:0.113323
cnt: 51000 loss:0.0911728
cnt: 52000 loss:0.0657265
cnt: 53000 loss:0.10515
cnt: 54000 loss:0.0836806
cnt: 55000 loss:0.0801077
cnt: 56000 loss:0.0712641
cnt: 57000 loss:0.0705078
cnt: 58000 loss:0.0722979
cnt: 59000 loss:0.0564801
cnt: 60000 loss:0.0567291
#epoch 1 --final_loss: 0.0567291
```

根据一轮的训练过程，将 loss 值的变化绘制成图表，其结果如下：



训练完成后，用测试集对训练出的最终网络模型进行测试，测试同样分批次进行，每批训练完成后对当前预测正确的数量进行统计并输出，最后根据最终的测试结果计算准确率并输出。

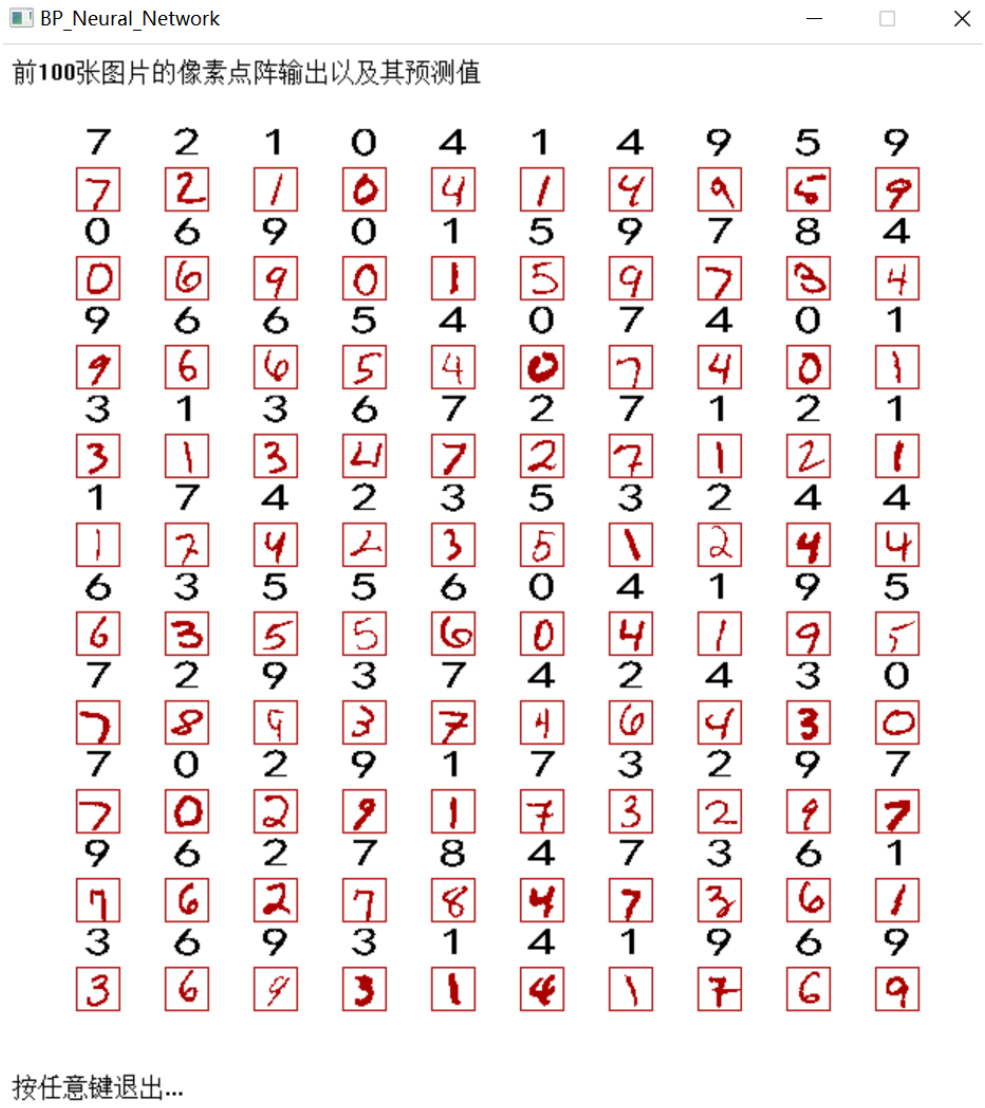
```
Predict result:
Test num: 1000 success : 914
Test num: 2000 success : 1779
Test num: 3000 success : 2688
Test num: 4000 success : 3582
Test num: 5000 success : 4471
Test num: 6000 success : 5415
Test num: 7000 success : 6353
Test num: 8000 success : 7300
Test num: 9000 success : 8260
Test num: 10000 success : 9176

Success rate: 0.9176
```

3.2 预测结果可视化

程序中对最终的训练结果进行了可视化的展示。

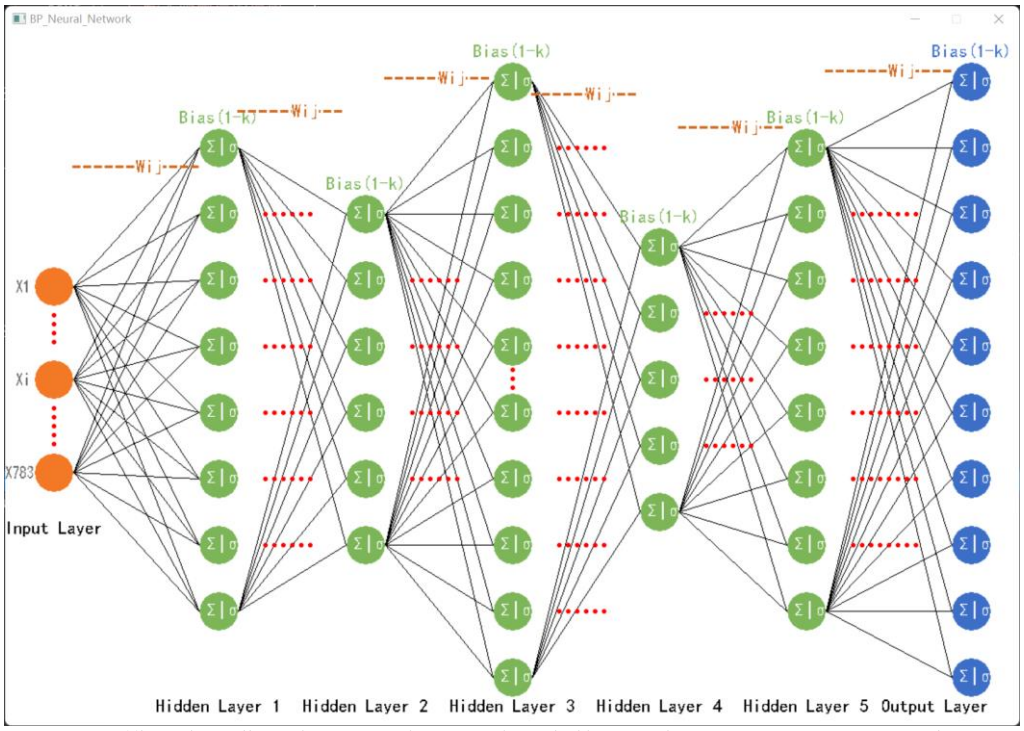
如图，程序对训练集的前 100 张图片进行了绘制，并输出了对应图片的预测结果。



本图通过 C++图形库 EasyX 进行绘制，红色图片根据数据集中 28*28 的矩阵数据进行绘制，每个数字图片的上方显示的数值为最终的神经网络模型对图片数据的预测结果。

通过本图片，可以较为直观地感知最终神经网络的训练效果。

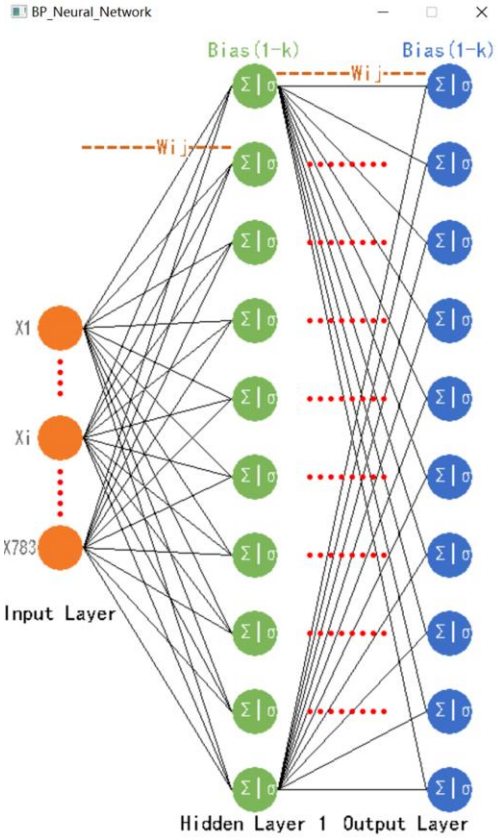
3.3 网络模型可视化

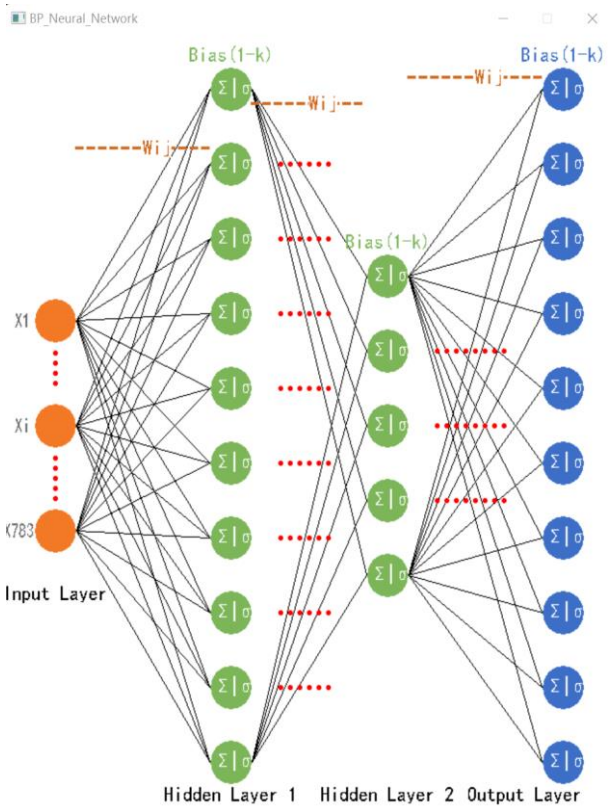


本图对应的模型为隐藏层有五层，每层的结点个数分别为 8、6、15、5、8 的神经网络。

图中，橙色的圆表示输入层结点，绿色的圆表示隐藏层结点，蓝色的圆表示输出层结点。受画布大小限制，未能画出的结点和线均通过省略号来表示。

程序进行绘图时图上的隐藏层数量会随着具体采用的模型的改变而改变。





如上图所示，左侧图片对应的是隐藏层有一层，隐藏层结点有 10 个的情形，而右侧图片对应的是隐藏层有两个，第一个隐藏层有 10 个结点，第二个隐藏层有 5 个结点的情形。
根据一开始输入的数据绘制相应的模型，可以较为直观地对当前采用地神经网络进行感知。

3.4 网络模型数据交互可视化

由于难以再图上标注出所有的层之间的权重和所有结点的阈值，可以采用**交互的方式来查看最终训练完成后神经网络上的各个权重和阈值。**

```
您可以通过输入来选择各节点之间数据传递的权重，或者某节点的偏置值
您是否要查看？（输入 1/0 表示 是/否）：1

您要查看权重/偏置值？（输入 1/0 表示 权重/偏置值）1

您要查看节点之间的权重，将输入层看作第0层，输出层看作最后一层，请输入要查看的层(该层和该层的下一层即为要选择的两层):1
请输入要查看的两个层数上节点的位置:3 5
权值:3.20261

您可以通过输入来选择各节点之间数据传递的权重，或者某节点的偏置值
您是否要查看？（输入 1/0 表示 是/否）：1

您要查看权重/偏置值？（输入 1/0 表示 权重/偏置值）0

您要查看节点的偏置值，将输入层看作第0层，输出层看作最后一层，请输入要查看节点所在的层:2
请输入您要查看的节点标号:6
偏置值:-6.50388

您可以通过输入来选择各节点之间数据传递的权重，或者某节点的偏置值
您是否要查看？（输入 1/0 表示 是/否）：0
```

3.5 可视化展示实现方案

可视化展示主要分为两部分：手写数字训练结果展示，训练网络模型展示。
可视化展示部分主要使用<graphics.h>库进行设计。

3.5.1 手写数字训练结果展示

对于训练的结果，我们希望不仅使用正确率的输出来展示模型和预测的效果，我们希望用更加直观的效果，让用户看到预测的图片和预测的结果是什么样，从感性和理性两方面了解到模型的预测效果。所以我们选择将图片和预测结果绘制在窗口上。

图片的数据来源是测试集，我们默认选取测试集前 100 张图片的数据进行展示。因为在测试集文件中，图片都是以一个一个像素的形式记录的，每张图片的大小是 28*28。所以，可以一次读取 28*28 共 784 字节的数据，再将其一个像素点一个像素点地绘制到屏幕上。

通过规划和计算，可以将 100 张图片以十行十列的形式整齐地排列在屏幕上。并将预测的结果进行输出。

主要的核心代码部分如下

```
void draw_pic(char mat[784], int pred, int row, int col) {
    int root_x = col * 28 * 2;
    int root_y = row * 28 * 2 + 28;
    for (int i = 0; i < 28; i++) {
        for (int j = 0; j < 28; j++) {
            if (i == 0 || i == 27 || j == 0 || j == 27 || mat[i * 28 + j] == char(1))
                putpixel(root_x + j + 50, root_y + i + 50, RED);
        }
    }

    setcolor(BLACK);
    settextstyle(25, 18, L"黑体");
    outtextxy(root_x + 55, root_y - 28 + 50, '0' + pred);
}
void Net::show_model(const char* filename) {
    /*
    此处省略相关文件操作
    */
    setcolor(BLACK);
    outtextxy(10, 10, L"前 100 张图片的像素点阵输出以及其预测值");
    outtextxy(10, 650, L"按任意键退出...");

    settextstyle(20, 20, L"黑体");
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 10; j++) {
            unsigned char buff[784];
            memset(buff, 0, 784);
            indata.read((char*)(buff), 784);

            char tran[784];
            memset(tran, 0, 784);
            for (int k = 0; k < 784; k++) {
                tran[k] = (unsigned int)(buff[k]) < 128 ? char(0) : char(1);
            }
            draw_pic(tran, predict_label[i * 10 + j], i, j);
        }
```

```
    }
    /*
    此处省略相关善后的操作
    */
}
```

3.5.2 训练网络模型展示

对于用户和数据集共同构建得到的神经网络，我们希望用户可以直观地看到网络的状况，包括不同层之间的突触连线，不同的神经元模型展示，让用户通过可视化地展示了解网络架构和训练的过程。
所以我们通过规划和计算展示窗口的坐标和布局，最终实现了网络的展示，效果将在下文呈现。这一部分代码较多，不方便在报告中展示，其对应的实现在 net.cpp 中的

4 优化与总结

4.1 算法的改进与优化

4.1.1 传递公式优化

不同层之间的权重和偏置值传递公式中存在可公用的部分，最初我们对这些可公用的部分多次计算，导致了代码的冗长和效率的降低。

举个例子：

输出层某一节点上偏置值的修正量为：

$$\delta\lambda = -\eta\,(y-\hat{y}) * \hat{y} * (1-\hat{y})$$

隐藏层某一到输出层某一节点的权重的修正量为：

$$\delta v = \eta(y-\hat{y}) * \hat{y} * (1-\hat{y})h$$

隐藏层某一节点上偏置值的修正量为：

$$\delta\beta = -\eta(y-\hat{y}) * \hat{y} * (1-\hat{y}) * v * h * (1-h)$$

输入层某一节点到隐藏层某一节点的权重的修正量为：

$$\delta w = \eta(y-\hat{y}) * \hat{y} * (1-\hat{y}) * v * h * (1-h)$$

可以看到，在以上四个公式中，$\eta(y-\hat{y}) * \hat{y} * \hat{y} * (1-\hat{y})$

我们完全可以在后向传播时，计算得到$\delta\lambda$的值后，直接将其传递给前一层调用，可以有效的避免重复数据的计算，并使程序更加简洁

对于这一部分，我们最初不够成熟的代码如下：从中可以看出，多次重复计算的情况是比较严重的，代码也比较冗长。

```
void BPNet::backward(int* label) {
    /*
    计算后向传播的 bias_delta，用来更正 bias
    更正的是输出层第 i 个节点的值
    */
    for (int i = 0; i < Config::OUTPUTNODE; i++) {
        double bias_delta = -(label[i] - outputLayer[i]->value) * outputLayer[i]->value *
(1.0 - outputLayer[i]->value);
        outputLayer[i]->bias_delta += bias_delta;
    }
    /*
    计算后向传播中的 weight_delta,用来更正 weight
    更正的是从第 j 个隐藏层节点到第 k 个输出层节点的权重值
    */
    for (int j = 0; j < Config::HIDENODE; j++) {
        for (int i = 0; i < Config::OUTPUTNODE; i++) {
            double weight_delta = (label[i] - outputLayer[i]->value) *
outputLayer[i]->value * (1.0 - outputLayer[i]->value) * hiddenLayer[j]->value;
            hiddenLayer[j]->weight_delta[i] += weight_delta;
        }
    }
    /*
    计算后向传播中 bias_delta，用来更正 bias
    更正的是隐藏层中第 i 个节点的 bias 值
    */
    for (int i = 0; i < Config::HIDENODE; i++) {
```

```
        double bias_delta = 0;
        for (int k = 0; k < Config::OUTPUTNODE; k++) {
            bias_delta += -(label[k] - outputLayer[k]->value) * (outputLayer[k]->value *
(1.0 - outputLayer[k]->value) * hiddenLayer[i]->weight[k]);
        }
        bias_delta *= hiddenLayer[i]->value * (1.0 - hiddenLayer[i]->value);
        hiddenLayer[i]->bias_delta += bias_delta;
    }
    /*
    计算 weight_bias，用来更正 weight 的值
    更正的是从第 j 个输入层节点到第 i 个隐藏层节点的 weight
    */
    for (int j = 0; j < Config::INPUTNODE; j++) {
        for (int i = 0; i < Config::HIDENODE; i++) {
            double weight_delta = 0;
            for (int k = 0; k < Config::OUTPUTNODE; k++) {
                weight_delta += (label[k] - outputLayer[k]->value) *
outputLayer[k]->value * (1 - outputLayer[k]->value) * hiddenLayer[i]->weight[k];

            }
            weight_delta *= hiddenLayer[i]->value * (1 - hiddenLayer[i]->value) *
inputLayer[j]->value;
            inputLayer[j]->weight_delta[i] += weight_delta;
        }
    }
}
```

在仔细比对公式之后，我们对可代换和重复计算的部分进行简化处理，使得算法的效率提高，代码也更加整齐和简洁。

4.1.2 权值和偏置值更新策略优化

在实验最初，我们选择每读入一个数据，求得一次 one_sample_bias_delta 和 one_sample_weight_delta，并将该单个数据的 delta 值加到另一个总的 batch_bias_delta 和 batch_weight_delta 之上，当完整读取了 batch_size 的数据之后，计算

$$average\_bias\_delta = \frac{batch\_bias\_delta}{batch\_size} \quad and \quad average\_weight\_delta = \frac{batch\_weight\_delta}{batch\_size}$$

再用两个平均值 average_bias_delta 和 average_weight_delta 进行神经网络权值和偏置值的更新。

但是经过实际的模型训练和预测过程后发现，这样设计的网络效果不好，在训练过程中损失函数下降的效果不好，最终预测的结果也处在百分之三十的正确率左右。

最开始我们得到百分之三十的正确率十分疑惑是网络的哪里出了问题，在仔细查阅资料后重新对 update 更新进行了修正。

我们选择没输入一个训练数据，就在后向传播的过程中计算出 bias_delta 和 weight_delta 的值，并直接对网络进行更新，这样大大提高了网络对于训练数据的敏感性，可以敏捷地修改网络的各部分内容，取得了非常好的效果。

对于这一部分，我们最初不够成熟的代码如下

```
double max_loss = 0;
int cnt = 0;
while (DataFile.peek()!=EOF && LabelFile.peek() != EOF) {

    for (int j = 0; j < Config::batch_size; j++) {
        memset(image_buf, 0, 784);
        memset(label_buf, 0, 10);
        DataFile.read(image_buf, 784);
        LabelFile.read(&label_temp, 1);
        for (int i = 0; i < 10; i++) {
            label_buf[i] = (unsigned int)(label_temp) == i ? 1 : 0;
        }
        for (int i = 0; i < Config::INPUTNODE; i++) {
            inputLayer[i]->value = ((unsigned int)(image_buf[i]) < 128 ? 0 : 1);
        }
        forward_temp();
        double loss = LossFun(label_buf);
        //cout << "loss:" << loss << endl;
        max_loss = max(max_loss, loss);
        backward_temp(label_buf);
    }
    update_temp(Config::batch_size);

    cout << "cnt:" << ++cnt << "max_loss:" << max_loss << endl;
}
```

4.1.3 多层隐藏层神经网络算法改进

实验中一开始采用了较为简单的神经网络模型，将神经网络设定为输入层、隐藏层和输出层三层，然后通过配置文件来调整各层的结点数，为了能够更好地感知不同层数的神经网络对训练效果的影响，我们对代码进行了重构，改成了通过输入来对神经网络中隐藏层进行设定的方式。

将隐藏层设定为多层主要需要对算法中的以下部分进行改进。

后向传播过程中原先的计算隐藏层的代码如下

```
for (int j = 0; j < Config::HIDE_NODE_NUM; j++) {
    double sigma = 0;
    for (int k = 0; k < Config::OUTPUT_NODE_NUM; k++) {
        sigma += hiddenLayer[j]->weight[k] * outputLayer[k]->bias_delta;
    }
    hiddenLayer[j]->bias_delta = (hiddenLayer[j]->value) * (1.0 - hiddenLayer[j]->value)
    * sigma;
}
```

该部分代码采用的公式为

$$\frac{\partial L}{\partial w_{jk}} = (o_k - t_k) = (o_k - t_k)o_k(1 - o_k) \cdot x_j$$

改进后的代码如下

```
for (int layer = hidden_layer_num - 1; layer >= 0; layer--) {
    for (int j = 0; j < hidden_layer_node_num[layer]; j++) {
        double sigma = 0;
        if (layer == hidden_layer_num - 1) {
            for (int k = 0; k < Config::OUTPUT_NODE_NUM; k++) {
                sigma += hiddenLayer[layer][j]->weight[k] * outputLayer[k]->bias_delta;
            }
            hiddenLayer[layer][j]->bias_delta = (hiddenLayer[layer][j]->value) * (1.0 -
hiddenLayer[layer][j]->value) * sigma;
        }
        else {
            for (int k = 0; k < hidden_layer_node_num[layer + 1]; k++) {
                sigma += hiddenLayer[layer][j]->weight[k] * hiddenLayer[layer +
1][k]->bias_delta;
            }
            hiddenLayer[layer][j]->bias_delta = (hiddenLayer[layer][j]->value) * (1.0 -
hiddenLayer[layer][j]->value) * sigma;
        }
    }
}
```

从上面公式推导部分的结论可以看到，倒数第二层和倒数第三层的权重和损失值的关系依次为

$$\frac{\partial L}{\partial w_{ij}} = \delta_j^J \cdot o_i$$

$$\delta_j^J = o_j(1-o_j) \cdot \sum_k \delta_k^K \cdot w_{jk}$$

和

$$\frac{\partial L}{\partial w_{ni}} = \delta_i^I \cdot o_n$$

$$\delta_i^I = o_i(1-o_i) \cdot \sum_j \delta_j^J \cdot w_{ij}$$

可以看出神经网络中从后往前递推的过程中，计算阈值变化量的公式在形式上是完全一样的。

因此，在有多个隐藏层存在的情况下，可以通过一个循环从后往前依次求出阈值的变化量，在计算当前的结点的阈值变化量时，只需要用到前面一层的阈值变化量，然后和当前位置原来的输出值一并带入同样的公式，即可实现求解，从而可以实现后向传播的过程。