哈爾濱工業大學（威海）

Harbin Institute of Technology at Weihai

程序设计实践报告

题目：CNN_MNIST 类实现

院　　系：计算机科学与技术暨软件学院

角色	学号	姓名	组内评分	小组成绩
组长	**181110315**	王少博		
组员	**181110318**	徐日升		
组员	**181110310**	刘凯华		
组员				

哈尔滨工业大学（威海）

二零一九年七月

哈尔滨工业大学（威海）程序设计实践任务书

姓　　名：　王少博　　徐日升　　刘凯华

学　　号：181110315 181110318 181110110

院（系）：　计算机科学与技术暨软件学院　　专　　业：　　软件工程

任务起止日期：2019 年　7　月　1　日 至　2019 年　7　月　7　日

课程设计题目：CNN_MINIST 类实现

系统可行性分析及主要功能：

在人工智能早期，那些对人类智力来说非常困难、但对计算机来说相对简单的问题得到迅速解决，比如，那些可以通过一系列形式化的数学规则来描述的问题。人类往往对抽象和形式化的工作感到困难，而人工智能的真正挑战在于解决那些对人来说很容易执行、但很难形式化描述的任务，如识别人们所说的话或图像中的脸。对于这些问题，人类往往可以凭借直觉轻易地解决。计算机更擅长解决线性关系而人更擅长解决非线性关系。但是随着人工智能的强势崛起，许多新型算法比如深度学习等方法给计算机解决非线性关系问题提供一种解决方法，也让我们看到了计算机解决相关问题的希望。因此我们也运用深度学习卷积网络中的方案，让计算机从经验中学习，增加计算机解决非线性关系的能力，并根据层次化的概念体系来理解世界，而每个概念则通过与某些相对简单的概念之间的关系来定义。从经验获取知识可以避免由人类来给计算机形式化地指定它需要的知识，而层次化让计算机构建较简单的概念来学习复杂概念。在诸多最新的人工智能成果中，我们所做的也许只是九牛一毛，但我们希望我们现在做的程序可以成为我们以后更好地学习相关知识的基础，为以后作出更好的成果做铺垫！

基于 C++神经网络的不规则手写图形的识别正是一种该方案的实践，它是实现计算机视觉的一种方法，具有很高的应用价值。该方案在普通的 BP 算法全连接神经网络基础上，给出一种基于卷积神经网络（CNN）深度学习的能够稀疏交互、参数共享和等变表示的识别不规则手写图形的方法。他的意义在于相比于普通 BP 算法全连接神经网络，拥有对图像处理的旋转不变性、位移不变性、扭曲不变性、降低网络复杂度、减少来自隐藏层的计算量、减少所需储存空间、减少权值的数量、降低了数据的维度，识别性能更高，更类似真正的生物神经网络，从而真正地实现从千变万化的图片到我们所要的真正信息的转变。

所要解决的主要问题有：

（1）规划 CNN 的层数、卷积层的特征图像数、池化层的特征图像数、卷积层卷积核大小、池化层池化邻域的大小、卷积核内置权值与算法、池化层内置算法、隐藏层间特征图像的连接方法、最高隐藏层与输出层的全连接方法。

（2）计算每层特征图像大小，设计计算大小的算法。

（3）选取每个神经元映射所使用的激活函数，并找出最优的。

（4）设计前向传播的过程，设计生成随机权重与阈值的均匀分布。

（5）设计反向传播的过程，设计梯度下降的算法与构造损失函数，设计合理的学习率防止高原反应或深陷泥潭。

（6）设计训练方法，选取逐个样例训练（EET）或批量样例训练（BT）或随机训练（ST），并找出最优的训练方法。

（7）给出目标正确率，给出迭代数范围，通过测试测试集打分不断向目标正确率靠拢 ，同时应该防止正确率下降。

（8）如果学习失败，出现一次迭代后正确率下降，或是在某些低正确率处出现正确率上升幅度非常缓慢时，对程序做相应修改或 debug。

其他边缘问题：

（1）文件操作：保存达到目标正确率时的网络全部的权值和阈值到文件中，同时在进行测试时读取文件内容作为网络前向传播的权值和阈值。

（2）读取 MNIST 数据集作为训练样本和测试样本。

（3）通过利用 openCV 和电脑摄像头实现现场手写图形的识别。

主要功能：

（1）实现对任何手写数字的准确识别。

（2）实现对任何不规则手写图形的识别。

工作量：

1. 卷积神经网络的搭建：

   由于三人对于相关知识了解甚少，计划先通过简单的 BP 神经网络学习来打下基础，再接触更为强大的卷积神经网络来实现。

2. MINIST 数据集使用和测试：

   我们需要海量的数据来训练，通过梯度下降等反馈机制让识别结果误差越来越小，然而这些数据又是我们自己再短期时间内无法提供的，因此必须借助强大的 MINIST 数据集。同时，测试工作作为项目的基本工作是必不可少的关键部分。

3. 交互：

   将底层代码通过 GUI 编程实现窗口界面化，增强用户粘合度。同时若时间允许我们还将增加摄像头的及时识别功能（openCV 库的应用）

4. 调试测试：

   对于已完成的部分进行全面的测试调试工作

目　录

CNN_MNIST 类实现详解

第 1 章 卷积神经网络(CNN)的基本组成

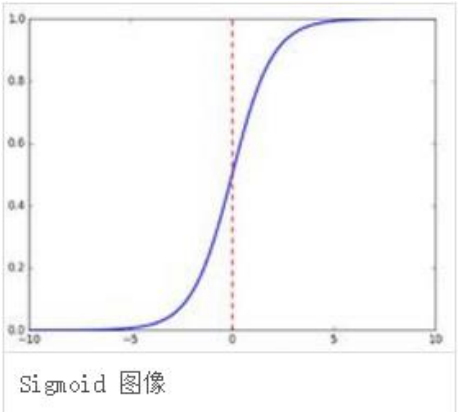
1.1 普通全连接神经网络的组成

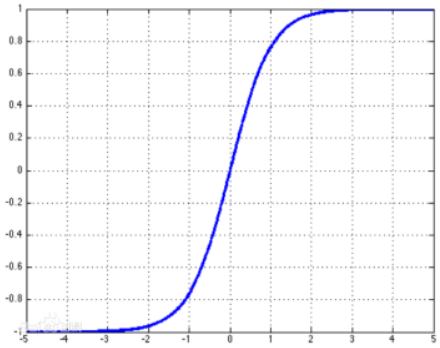
1.1.1 人工神经元（节点）

人工神经元是神经网络的基本组成个体，它的数学意义其实就是一个映射的过程。每个神经元的输入为一至多个，记为列向量 X。每个神经元的映射法则由激活函数给定，如果无给定激活函数则输出为输入 X 与参数间的线性组合。每个神经元的输出结果就是映射的结果，并且每个神经元将这个结果传递给与之相关（连接的）的每个下个神经元。计算的过程实际上是线性组合后经过映射法则并输出的过程。

1.1.2 激活函数（Activation Function）

所谓激活函数，就是在人工神经网络的神经元上运行的函数，负责将神经元的输入映射到输出端。常用的有 sigmoid 函数、tanh 函数、ReLu 函数。





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

图 1-1：sigmoid 函数

$$tanhx=\frac{sinhx}{coshx}=\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$

图 1-2：tanh 函数

由是观之，设一个神经元接受的全部输入是一个 i*1 的列向量$(x_1,\dots,x_i)^T$，输入的权重为 j*i 的矩阵，则输出即为一个 j*1 的列向量$(y_1,\dots,y_j)^T$。为了给网络增加“非线性因素”使之能够

拟合更多的非线性情况，才使用激活函数。它用来加入非线性因素，解决线性模型所不能解决的问题，提供网络的非线性建模能力。如果没有激活函数，那么该网络仅能够表达线性映射，此时即便有再多的隐藏层，其整个网络跟单层神经网络也是等价的。因此也可以认为，只有加入了激活函数之后，深度神经网络才具备了分层的非线性映射学习能力。其中激活函数的参数即为线性组合部分。

神经元的输出为$f(XW+B)$。（大写字母都是矩阵，B 是用来优化线性组合的阈值，是神经元的偏置，W 是权重，直观作用见下图）

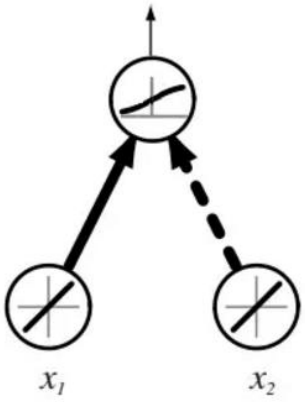
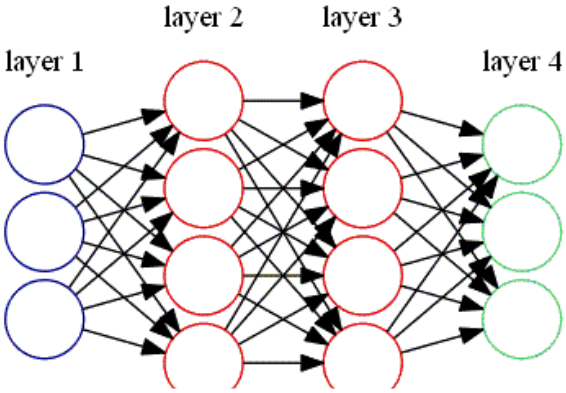


图 1-3 神经元输出

图 1-3 可以看出，两个过原点的线性函数经过人工神经元的激活函数后，变为非线性的不经过原点的函数。这样可以增加网络非线性因素，解决更多线性模型不能解决的问题。

1.1.3 前向传播（Forward Propagation）

前向传播强调的是层与层之间的递进关系。表示这一层是如何给下一层传入数据的，同时这一层接受到了上一层什么样的输出。在神经网络中输入输出为输入层（可见层）和输出层，其他层统称为隐藏层，称为第 1 隐藏层、第 2 隐藏层以此类推。由此观之，可以得到如下结果：**（每层的输出用$\mathbf{H_i}$表示）**



1-4 前向传播

每层输出的关系如下：

$$\boldsymbol{H_1 = f(XW_1 + B_1)}$$
$$\boldsymbol{H_2 = f(H_1W_2 + B_2)}$$

$$\cdots\ \cdots\ \cdots\ \cdots\ \cdots$$
$$\boldsymbol{H_i = f(H_{i-1}W_i + B_i)}$$
$$\boldsymbol{Y = f(H_iW_{i+1} + B_{i+1})}$$

由此每一层之间的关系就清晰了很多。

1.1.4 反向传播（Backward Propagation）

反向传播主要计算每层权值和阈值的误差以及每层神经元的误差。反向传播强调的是想办法更新参数以达到“学习”的目的，如果参数不更新，则每次输出都是一样的。那么如何更新参数呢？这就需要构造**损失函数**（又叫代价函数、误差函数）。损失函数通常作为学习准则与优化问题相联系，即通过最小化损失函数求解和评估模型。例如在统计学和机器学习中被用于模型的参数估计。我们把要最小化的函数称为**目标函数**（准则），目标函数和损失函数其实是同一个东西，在确定要对这个函数最小化时叫目标函数，最小化的过程中叫损失函数。（就好比染色质和染色体）下文将损失函数记为 E。

另外，如果想最大化一个目标函数，就通过最小化-E(x)来实现。

本文用到的损失函数是均方误差损失函数 MSE（mean squared error）。通过将 MSE 最小化可以无限近似得到 MSE->0，即 Y->Y*（实际输出趋于预期输出，Y*常数）

MSE 的表达式为：$\boldsymbol{MSE = \sum_{i=1}^{n}(y_i - y_i^p)^2}$

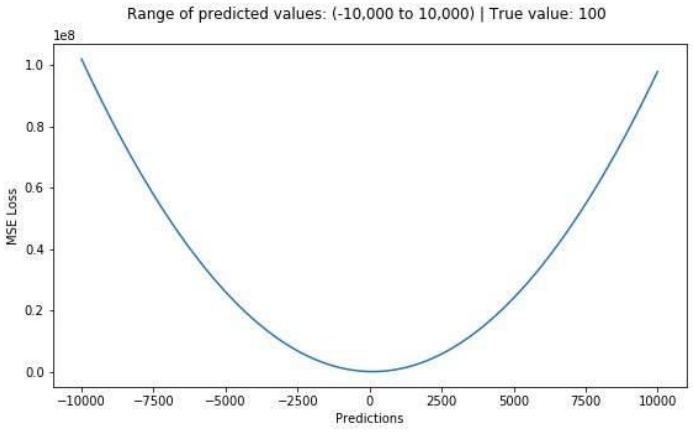


图 1-5 MSE 图像

当然，更新参数的最常用最核心的办法就是**梯度下降（Gradient Descent）**。何为梯度下降？即沿着梯度的反方向移动一小步来减小 E。对于 MSE 我们可以看出它的图像类似$\boldsymbol{y = x^2}$的图像，为了让 E 趋于 0，那么 x 也要趋于 0，可以看出 x<0 时 L’<0，x>0 时 L’>0，所以只需要沿着导数反方向移动就有可能到达最低点。其他多维度复杂模型可以参考二维简单模型类比。梯度的求法和工数学的一样 $\boldsymbol{gradE = (E_x^{'}, E_y^{'}, E_z^{'}, \ldots)}$沿梯度方向下降是最快的。当然梯度下降建议每次迭代后的新的点为：

$$\mathbf{x' = x - \alpha E_x^{'}}$$
$$\mathbf{y' = y - \alpha E_{y,}^{'}}$$
$$\mathbf{z' = z - \alpha E_z^{'}}$$
$$\cdots\ \cdots\ \cdots\ \cdots$$

特别强调：这里的 x, y, z 表示的不是网络输入，而是损失函数 E 的全部参数。在梯度下降过程中的输入是不变的，也就是网络中的每一层之间的前向传播确定好的关系就不变了。而参

数 W 和 B 以及其他什么参数都是通过随机数生成法生成的，梯度下降是在更新参数，也就是更新诸如 W 和 B 一样的参数。使得最后输入的这一组 X 的实际输出 Y 能够趋于与其输出 Y*。这样最后保留下来的参数就会被留在网络中，这样的网络每层之间保存着训练好了的参数，当再遇到未知的输入时，就可以输出一个结果看它是不是与预期输出接近了。

表达式中的 **α 指学习率**。学习率的大小决定了每次迭代后 E 位移的距离，称为**步长**。学习率大步长大相对训练速度就越快，反之越慢。

我们知道在驻点的位置导数为 0（梯度为 0），在这时梯度就不能够提供移动方向的信息了，这意味着我们有可能训练成功了。但我们又知道驻点有两种极值点和鞍点。极值点不能保证它是全域最小的，鞍点就更不是我们想要的东西了。如果损失函数是一种拥有特别多的局部极小值的函数，那么训练的过程就像是被困在了一堆泥潭里根本出不去；如果损失函数有鞍点，那么我们的训练过程还有可能被困在鞍点，这些区域通常被误差（误差指 E 的值）相等的高原环绕，我们就陷入了高原反应。所以在被困住的时候我们极力希望能够大跨步走出困境，这时候就需要将 α 调大。但是相对的，如果我们的步子太大了，还会遇到其他尴尬就是是否会一大步直接把全域最小值迈过去了。学习率的选择需要慎重，因为我们初始化启动的位置是随机的（损失函数图像的坐标轴是它的参数表示的，参数值利用随机数法生成），干脆不知道前方会是什么样的境界。一般的学习率选取为小常数，比如 3, 1, 0.1, 0.01, 0.001, 0.0001, 0.00001。当然我们还可以用一种优化方法动态改变学习率，具体怎么动态改变学习率，可以先将这个设置为一个很小的值，如果我们检测到我们的误差确实是在减小则不断调大学习率（以减少训练时间），如果误差突然变大了，说明学习率太大了，即便是沿梯度反向步长过大也有可能移动到误差更大的地方。

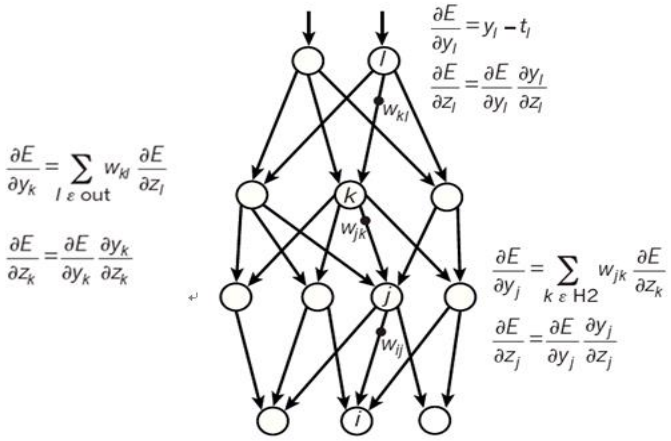
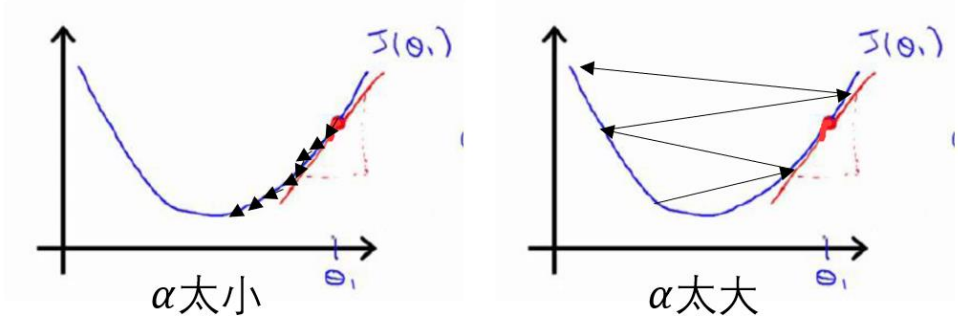


图 1-6 反向传播示意图

：



(a) small- α gradient descent

(b) big- α gradient descent

图 1-7 Gradient Descent

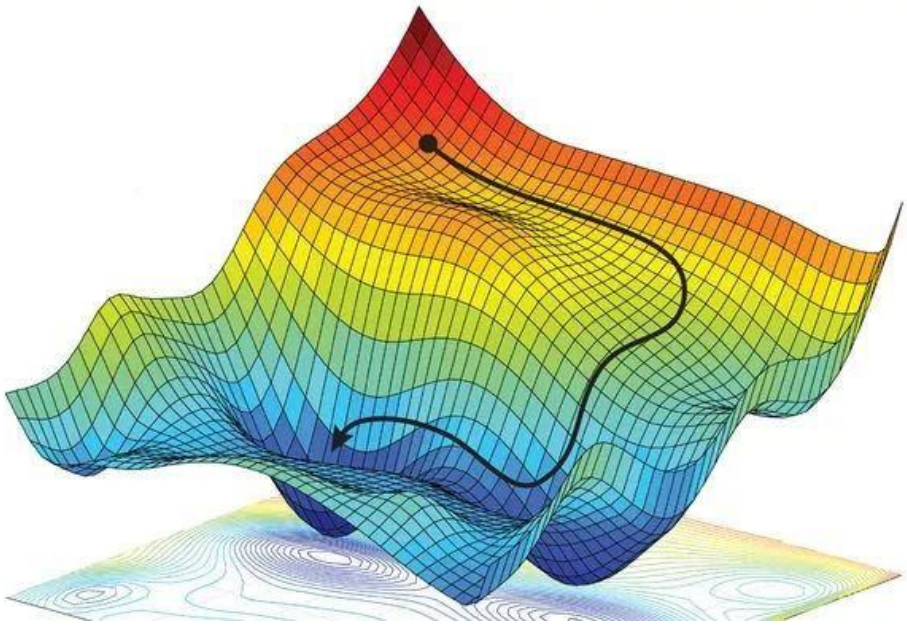


图 1-8　梯度下降示意图

1.2 卷积神经网络与普通全连接神经网络的区别

1.2.1 卷积神经网络简介

卷积神经网络（convolutional neural network，CNN），是一种专门用来处理具有类似网格结构的数据的神经网络。例如时间序列数据（可以认为是在时间轴上有规律地采样形成的一位网格）和图像数据（可以看作二维的像素网格）。CNN 在诸多应用领域表现优异。卷积一词表示该网络使用了卷积运算，卷积是一种特殊的线性运算，CNN 指那些至少在网络的一层中使用卷积运算来代替一般矩阵乘法运算的神经网络。

引例：

假设我们正在用激光传感器追踪一艘宇宙飞船的位置。我们的激光传感器给出一个单独的输出 x(t)，表示宇宙飞船在时刻 t 的位置。x 和 t 都是实值的，这意味着我们可以在任意时刻从传感器中读出飞船的位置。

现在假设我们的传感器受到一定程度的噪声干扰。为了得到飞船位置的低噪声估计，我们对得到的测量结果进行平均。显然，时间上越近的测量结果越相关，所以我们采用一种加权平均的方法，对于最近的测量结果赋予更高的权重。我们可以采用一个加权函数 w(a)来实现，其中 a 表示测量结果距当前时刻的时间间隔。如果我们对任意时刻都采用这种加权平均的操作，就得到了一个新的对于宇宙飞船未知的平滑估计函数 s：

$$s(t) = \int x(a)w(t-a)da$$

这种运算就叫作**卷积**。卷积也通常可以用星号表示：

$$s(t) = (x * w)(t)$$

在上述中，w 必须是一个有效概率密度，否则输出就不再是一个加权平均，另外在参数 a<0 时 w=0，否则它会预测未来，不符合我们的规定。

在 CNN 的术语中，卷积的第一个参数（在这个例子中，函数 x）通常叫作**输入**（input），第二个参数（函数 w）叫作**核函数**（kernel function）。输出有时被称作**特征映射**（feature map）

在本例中，激光传感器在每个瞬间反馈测量结果的想法是不切实际的。一般来讲，当我们用计算机处理数据时，时间会被离散化，传感器会定期地反馈数据。所以在我们的例子中，假设传感器每秒反馈一次测量结果是比较现实的。这样，时刻 t 只能取整数值。如果假设 x 和 w 都定义在整数时刻 t 上，就可以定义离散形式的卷积：

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)$$

在机器学习的应用中，输入通常是多维数组的数据，而核通常是由学习算法优化得到的多维数组参数。（卷积核 w 是我们训练的对象）

1.2.2 稀疏连接、参数共享与等变表示

在普通 BP 全连接神经网络中，每一层的神经元节点是一个线性一维排列结构，层与层各神经元节点之间是全连接的。卷积神经网络中，层与层之间（除最后两层）的神经元节点不再是全连接形式，利用层间局部空间相关性将相邻每一层的神经元节点只与和它相近的上层神经元节点连接，即局部连接。

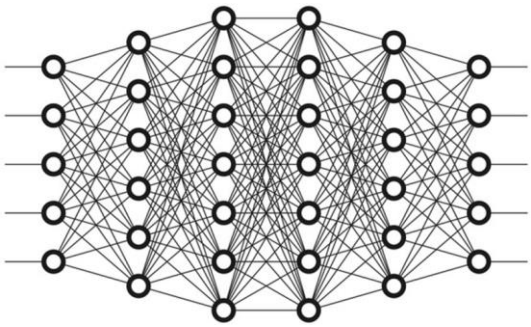


图 1-9 普通全连接神经网络

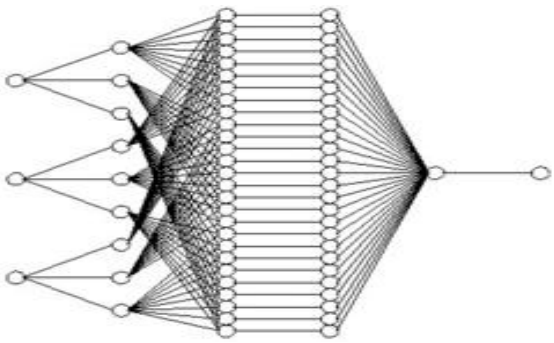


图 1-10 卷积神经网络稀疏连接

在卷积神经网络中，卷积层的每一个卷积滤波器（卷积核）重复的作用于整个感受野中，对输入图像进行卷积，卷积结果构成了输入图像的特征图，提取出图像的局部特征。每一个卷积滤波器共享相同的参数，包括相同的权重矩阵和偏置项。参数共享的好处是在对图像进行特征提取时不用考虑局部特征的位置，而且提供了一种有效的方式，使要学习的卷积神经网络模型参数数量大大降低。同一特征图的神经元参数共享，减少了网络参数，这也是卷积网络相对于全连接网络的一大优势。共享局部参数这一特殊结构更接近于真实的生物神经网络使 CNN 在图像处理、语音识别领域有着独特的优越性，另一方面多维输入信号（语音、图像）可以直接输入网络的特点避免了特征提取和分类过程中数据重排的过程。

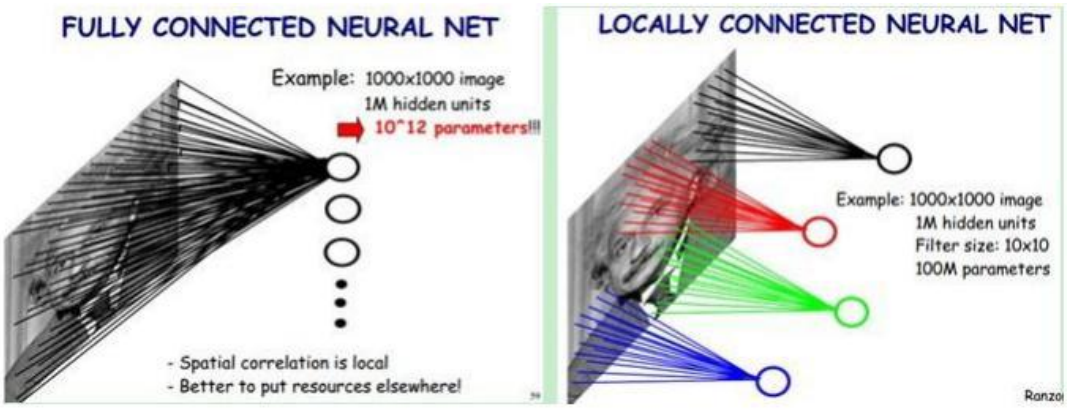


图 1-11　基于稀疏连接的参数共享图解

由于参数共享在对图像进行特征提取时不用考虑局部特征的位置，CNN 可以用来识别位移、缩放及其它形式扭曲不变性的二维或三维图像。局部区域（卷积核）感知能够发现数据的一些局部特征，比如图片上的一个角，一段弧，这些基本特征是构成动物视觉的基础。它只考虑这些特征和特征之间的相对位置而几乎不考虑特征的绝对位置。CNN 的特征提取层（卷积核）参数是通过训练数据学习得到的，所以其避免了人工特征提取，而是从训练数据中进行学习。

1.2.3 池化（利用最大池采样法）

池化是一种非线性降采样方法。在通过卷积获取图像特征之后是利用这些特征进行分类。可以用所有提取到的特征数据进行分类器的训练，但这通常会产生极大的计算量。所以在获取图像的卷积特征后，要通过最大池采样方法对卷积特征进行降维。将卷积特征划分为数个 n*n 的不相交区域，用这些区域的最大(有时利用平均)特征来表示降维之后的卷积特征。这些降维之后的特征更容易进行分类。

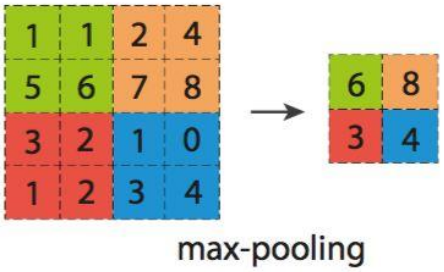


图 1-12　最大池采样法

它在计算机视觉中的价值体现在两个方面：
（1）减小了来自上层隐藏层的计算复杂度；
（2）这些池化单元具有平移不变性，即使图像有小的位移，提取到的特征依然会保持不变。由于增强了对位移的鲁棒性，最大池采样法是一个高效的降低数据维度的采样方法。

1.2.4 特征图（feature map，以下简称 map）

一个卷积核在这个图片上滚一下形成一个**特征图（map）**，多种卷积核则可以形成多个 map。卷积核往往代表一个特征，比如某个卷积核代表一段弧，那么把这个卷积核在整个图片上滚一下，map 中卷积值较大的区域就很有可能是一段弧。CNN 中每一层由多个 map 组成，每个 map 由多个神经元组成，同一个 map 的所有神经元共用一个卷积核。注意卷积核其实就是权重，我们并不需要单独去计算一个卷积，而是一个固定大小的权重矩阵去图像上匹配并生成 map，这个操作与卷积类似，因此我们称它为卷积神经网络。实际上，普通 BP 全连接神经网络也可以看作一种特殊的卷积神经网络，只是这个卷积核就是某层的所有权重，即感知区域（卷积核）是整个图像。参数共享策略使得卷积核缩小（远小于整张图片的大小），减少了需要训练的参数，使得训练出来的模型的泛化能力更强。
：

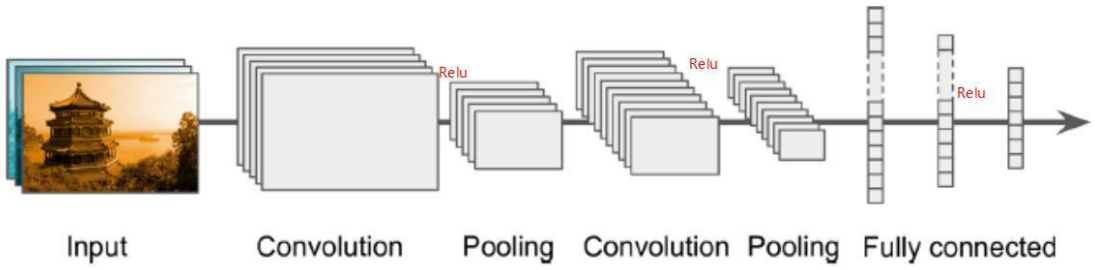


图 1-13　特征图图解（利用 Relu 函数，为一种激活函数）

1.2.5 卷积神经网络的注意事项

数据驱动的模型一般依赖于数据集的大小，CNN 和其它经验模型一样，能适用于任意大小的数据集，但用于训练的数据集应该足够大，能够覆盖问题域中所有已知可能出现的问题。设计 CNN 的时候，数据集中应该包含三个子集：训练集、测试集、验证集。训练集应该包含问题域中的所有数据，并在训练阶段用来调整网络权值。测试集用来在训练过程中测试网络对于训练集中未出现的数据的分类性能。根据网络在测试集上的性能情况，网络的结构可能需要做出调整，或者增加训练循环的次数。验证集中的数据同样应该包含在测试集合训练集中没有出现过的数据，用于在确定网络结构后能够更加好的测试和衡量网络的性能。推荐数据集中的 65%用于训练，25%用于测试，剩余的 10%用于验证。

为了加速训练算法的收敛速度，一般都会采用一些数据预处理技术，这其中包括：去除噪声、输入数据降维、删除无关数据等。数据的平衡化在分类问题中异常重要，一般认为训练集中的数据应该相对于标签类别近似于平均分布，也就是每一个类别标签所对应的数据量在训练集中是基本相等的，以避免网络过于倾向于表现某些分类的特点。为了平衡数据集，应该移除一些过度富余的分类中的数据，并相应的补充一些相对样例稀少的分类中的数据。还有一个办法就是复制一部分这些样例稀少分类中的数据，并在这些输入数据中加入随机噪声。

将数据规则化到一个统一的区间（如[0,1]）中具有很重要的优点：防止数据中存在较大数值的数据造成数值较小的数据对于训练效果减弱甚至无效化。一个常用的方法是将输入和输出数据按比例调整到一个和激活函数（sigmoid 函数等）相对应的区间。

CNN 的初始化主要是初始化卷积层和输出层的卷积核（权重）和阈值。网络权值初始化

就是将网络中的所有连接权值（包括阈值）赋予一个初始值。如果初始权值向量处在误差曲面的一个相对平缓的区域的时候，网络训练的收敛速度可能会异常缓慢。一般情况下，网络的连接权值和阈值被初始化在一个具有以 0 为平均值的相对小的区间内的均匀分布上，比如[-0.30, +0.30]这样的区间内。

如果学习率 α 选取的比较大则会在训练过程中较大幅度的调整权值 w，从而加快网络训练的速度，但这会造成网络在误差曲面上搜索过程中频繁抖动且有可能使得训练过程不能收敛，而且可能越过一些接近最优的 w。同样，比较小的学习速率能够稳定的使得网络逼近于全局最优点，但也有可能陷入一些局部最优区域。对于不同的学习速率设定都有各自的优缺点，但还有一种自适应的学习速率方法，即 α 随着训练算法的运行过程而自行调整。

有几个条件可以作为停止训练的判定条件，训练误差、误差梯度和交叉验证。一般来说，训练集的误差会随着网络训练的进行而逐步降低。

训练样例可以有两种基本的方式提供给网络训练使用，也可以是两者的结合：**逐个样例训练(EET)**、**批量样例训练(BT)**。在 EET 中，先将第一个样例提供给网络，然后开始应用 BP 算法训练网络，直到训练误差降低到一个可以接受的范围，或者进行了指定步骤的训练次数。然后再将第二个样例提供给网络训练。EET 的优点是相对于 BT 只需要很少的存储空间，并且有更好的随机搜索能力，防止训练过程陷入局部最小区域。EET 的缺点是如果网络接收到的第一个样例就是劣质（有可能是噪音或者特征不明显）的数据，可能使网络训练过程朝着全局误差最小化的反方向搜索。相对的，BT 方法是在所有训练样例都经过网络传播后才更新一次权值，因此每一次学习周期就包含了所有的训练样例数据。BT 方法的缺点也很明显，需要大量的存储空间，而且相比 EET 更容易陷入局部最小区域。而**随机训练(ST)**则是相对于 EET 和 BT 一种折衷的方法，ST 和 EET 一样也是一次只接受一个训练样例，但只进行一次 BP 算法并更新权值，然后接受下一个样例重复同样的步骤计算并更新权值，并且在接受训练集最后一个样例后，重新回到第一个样例进行计算。ST 和 EET 相比，保留了随机搜索的能力，同时又避免了训练样例中最开始几个样例若出现劣质数据对训练过程的过度不良影响。

第 2 章 CNN_MNIST 类的算法与代码实现

2.1 CNN 网络搭建预处理

2.1.1 利用 LeNet-5 模型搭建网络基本结构

LeNet-5 模型是一种专门设计用来实现手写数字识别的卷积神经网络模型。**我们的工作是按照其核心思想利用 C++程序语言设计并实现这个模型的基本功能。**神经元所使用的激活函数是 tanh。该模型一共有七层（定义层结构是这个层的输出）包括三个卷积层、两个采样层（池化）、一个全连接层、一个输出层（实际上最高卷积层也被设计为全连接层）。

为了方便，下文将卷积层定义为 C_i，采样层定义为 S_i，全连接层定义为 F_i，输出层定义为 OUTPUT（i 依次取 1,2,3,4,5,6）。此外，将输入的图片可以形象的定义为输入层。

定义输入层：

输入图像为 32*32 大小。这要比 MNIST 数据库中最大的字母还大。这样做的原因是希望潜在的明显特征如笔画断点或角点能够出现在最高层特征监测子感受野的中心。

定义 map：

卷积层的 map 个数是在网络初始化指定的，而卷积层的 map 的大小是由卷积核和上一层输入 map 的大小决定的，假设上一层的 map 大小是 n*n、卷积核的大小是 k*k，则可得到该层的 map 大小是(n-k+1)*(n-k+1)。

定义 C1 层：

表 2-1　C1 层定义

项目	规格	说明
输入图片大小与数量	(32*32)*1	
卷积核大小	5*5	
卷积核种类	6	
输出 map 数量	6	
输出 map 大小	28*28	(32-5+1)*(32-5+1)
神经元数量	4707	(28*28)*6
连接数	122304	(28*28*5*5*1*6)+(28*28*6)
可训练参数数	156	5*5*6+6

卷积过程是用一个可训练的卷积核 f_i 去卷积一个输入的图像（第一阶段是输入的图像，后面的阶段就是卷积 map 了），然后加一个偏置 B_i，得到卷积层 C_i。用 6 个 5*5 的卷积核进行卷积，结果在卷积层 C1 中，得到 6 张 map，map 的每个神经元与输入图片中的 5*5 的邻域相连，即用 5*5 的卷积核去卷积输入层，由卷积运算可得 C1 层输出的 map 大小为 (32-5+1)*(32-5+1)=28*28。

定义采样层：

采样层是对上一层 map 的一个采样处理，这里的采样方式是对上一层 map 的相邻小区域进行聚合统计，区域大小为 scale*scale，本设计采用 2*2 区域并取均值。注意，卷积层的卷积核计算部分是有重叠的，而采样的卷积核没有重叠，且卷积核是 2*2，每个元素都是乘权值再乘 1/4 得到的。最后去掉计算得到的卷积结果中有重叠的部分。

定义 S2 层：

表 2-2　S2 层定义

项目	规格	说明
输入图片大小与数量	(28*28)*6	
卷积核大小	2*2	
卷积核种类	6	
输出 map 数量	6	
输出 map 大小	14*14	(28/2)*(28/2)
神经元数量	1176	(14*14)*6
连接数	5880	(14*14*2*2*6)+(14*14*6)
可训练参数数	12	6+6

采样过程包括每邻域四个像素求和变为一个像素，然后通过标量 W_{i+1} 加权，再增加偏置 B_{i+1}，然后通过一个 tanh 激活函数，产生一个大概缩小四倍的 map，S_{i+1}。降采样后，降采样层 S2 的输出 map 大小为（28/2）*（28/2）=14*14。S2 层每个单元的 4 个输入相加，乘以一个可训练参数，再加上一个可训练偏置。结果通过 tanh 函数计算。可训练系数和偏置控制着 tanh 函数的非线性程度。如果系数比较小，那么运算近似于线性运算，下采样相当于模糊图像。如果系数比较大，根据偏置的大小下采样可以被看成是有噪声的“或”运算或者有噪声的“与”运

算。从一个平面到下一个平面的映射可以看作是卷积运算，S_i 层可看作是模糊滤波器，起到二次特征提取的作用。隐藏层与隐藏层之间空间分辨率递减，而每层所含的平面数递增，这样可用于检测更多的特征信息。每个单元的 2*2 感受野并不重叠，因此 S2 中每个 map 的大小是 C1 中 map 大小的 1/4（行和列各 1/2）。

定义 C3 层：

表 2-3　C3 层定义

项目	规格	说明
输入图片大小与数量	(14*14)*6	
卷积核大小	5*5	
卷积核种类	16	
输出 map 数量	16	
输出 map 大小	10*10	(14-5+1)* (14-5+1)
神经元数量	1600	(10*10)*16
连接数	151600	1516*10*10
可训练参数数	1516	6*(3*25+1)+6*(4*25+1)+3*(4*25+1)+1*(6*25+1)

C3 层也是一个卷积层，它同样通过 5*5 的卷积核去卷积 S2 层，然后得到的 map 就只有 10*10 个神经元，但是它有 16 种不同的卷积核，所以就存在 16 个 map 了。

特别强调：C3 中的每个 map 是连接到 S2 中的所有 6 个或者几个 map 的，表示本层的 map 是上一层提取到的 map 的不同组合。为什么不把 S2 中的每个 map 连接到每个 C3 的 map 呢？原因有两点：第一，不完全的连接机制将连接的数量保持在合理的范围内。第二，也是最重要的，其破坏了网络的对称性。由于不同的 map 有不同的输入，所以迫使他们抽取不同的特征。本设计使用的连接方法为表格的最后一格的长算式，它是 1998 年 Yann LeCun 提出的连接方法，具体的连接的判别标准如下（代码）：

```
#define O true
#define X false
static const bool tbl[6][16] = {
    O, X, X, X, O, O, O, X, X, O, O, O, O, X, O, O,
    O, O, X, X, X, O, O, O, X, X, O, O, O, O, X, O,
    O, O, O, X, X, X, O, O, O, X, X, O, X, O, O, O,
    X, O, O, O, X, X, O, O, O, O, X, X, O, X, O, O,
    X, X, O, O, O, X, X, O, O, O, O, X, O, O, X, O,
    X, X, X, O, O, O, X, X, O, O, O, O, X, O, O, O
};
// 连接（计算）的时候一旦遇到X则跳过，继续下一个连接。
#undef O
#undef X
```

定义 S4 层：

表 2-4　S4 层定义

项目	规格	说明
输入图片大小与数量	(10*10)*16	
卷积核大小	2*2	
卷积核种类	16	
输出 map 数量	16	
输出 map 大小	5*5	(10/2)*(10/2)
神经元数量	400	(5*5)*16
连接数	2000	(5*5*2*2*16)+(5*5*16)
可训练参数数	32	16+16

S4 层是一个采样层，由 16 个 5*5 大小的 map 构成。特征图中的每个单元与 C3 中相应 map 的 2*2 邻域相连接，类比 C1 和 S2 之间的连接一样。S4 层有 32 个可训练参数（每个 map 有 1 个因子和一个偏置）和 2000 个连接。

定义 C5 层：

表 2-5　C5 层定义

项目	规格	说明
输入图片大小与数量	(5*5)*16	
卷积核大小	5*5	
卷积核种类	120	
输出 map 数量	120	
输出 map 大小	1*1	(5-5+1)* (5-5+1)
神经元数量	120	1*120
连接数	48120	(1*1*5*5*16*120)+(120*1*1)
可训练参数数	48120	5*5*16*120+120

C5 层是一个卷积层，有 120 个 map。每个单元与 S4 层的全部 16 个单元的 5*5 邻域相连。由于 S4 层 map 的大小也为 5*5（同卷积核一样大），故 C5 的 map 的大小为 1*1。请注意，这构成了 S4 和 C5 之间的全连接。之所以仍将 C5 标示为卷积层而非全连接层，是因为如果 LeNet-5 的输入变大，而其他的保持不变，那么此时 map 的维数就会比 1*1 大。C5 层有 48120 个可训练参数。

定义 F6 层：

表 2-6　F6 层定义

项目	规格	说明
输入图片大小与数量	(1*1)*120	
卷积核大小	1*1	
卷积核种类	84	7*12
输出 map 数量	1	
输出 map 大小	7*12	
神经元数量	84	7*12
连接数	10164	120*84+84
可训练参数数	10164	120*84+84

设计 F6 层的意义在于 LeNet-5 模型旨在能够识别 ASCII 打印字符，而标准 ASCII 字符打印大小是 7*12=84。由于 ASCII 标准的打印字符是用 7*12 大小的位图表示的，所以希望以每一个神经元表示这个图上的一个位，来突出其每一个像素点的特性。实际上 F6 层输出的 map 一共有 84 个，而每个 map 都是 1*1 大小的，但因为要组合成一个 7*12 的 ASCII 标准的打印字符即 84 位比特图，所以全部的 map 组合成了一个新的 map（上述表格中表示的）。F6 层有 10164 个可训练参数。如同经典神经网络，F6 层计算输入向量和权重向量之间的点积，再加上一个偏置。然后将其传递给 tanh 函数产生每一个像素点的一个状态。每个像素点的取值范围被定义在(-1,1)（这是由激活函数的函数值域决定的）。最终希望通过 OUTPUT 层的欧式径向基函数（Euclidean Radial Basis Function，RBF）计算输出结果。RBF 的输出结果以特定的评判标准（实际是基于高斯分布的一个标准）决定当前识别到的数字是什么。（由于 RBF 和该高斯分布的评判标准过于复杂，且对我们的设计意图没有太大影响，我们决定使用其他简单的方法代替）

定义 OUTPUT 层：

按照上述 F6 层的设计，OUTPUT 层应该有如下定义：

表 2-7　OUTPUT 层定义

项目	规格	说明
输入图片大小与数量	(7*12)*1	
卷积核大小	RBF 运算，非卷积运算	
卷积核种类	RBF 运算，非卷积运算	
输出 map 数量	10	
输出 map 大小	1*1	是一个值
神经元数量	10	表示 10 个数字
连接数	840	84*10
可训练参数数	840	84*10

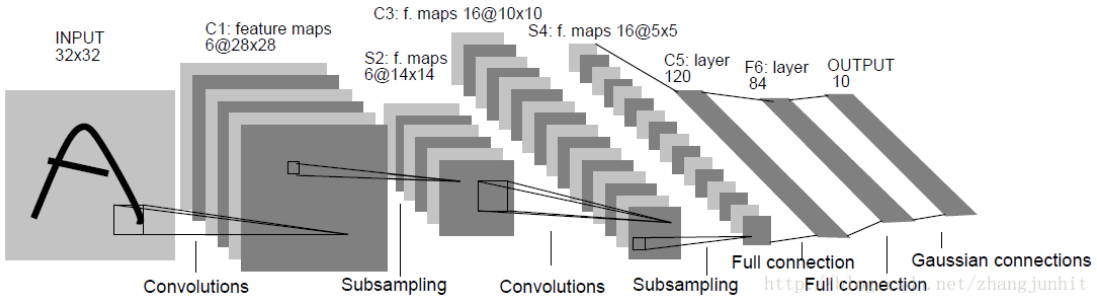


图 2-1 标准 LeNet-5 模型

但是遗憾的是，在我们的设计中并没有使用 F6 层。因为我们的目的是识别手写字符而非 ASCII 标准的打印字符。我们有更优化的策略使 C5 层与 OUTPUT 层全连接。即我们没有让 C5 映射成一个 ASCII 标准的打印字符 map，所以 OUTPUT 的输出过程也就没有使用 RBF 运算，而使用普通的卷积运算。得到的结果不能按照前文提到的基于 RBF 公式的特殊评判标准（高斯分布）评判识别到的是什么字符，而是使用一种简单的逐个比较 OUTPUT 层中每一个神经元输出值的大小的方法确定识别结果（后文代码实现部分将详细介绍）。所以重新定义 OUTPUT 层如下：

重新定义 OUTPUT 层：

表 2-8 重新定义的 OUTPUT 层

项目	规格	说明
输入图片大小与数量	(1*1)*120	
卷积核大小	1*1	
卷积核种类	10	
输出 map 数量	10	
输出 map 大小	1*1	是一个值
神经元数量	10	表示 10 个数字
连接数	1210	(1*1*1*1*120*10)+(1*1*10)
可训练参数数	1210	1*1*120*10+10

OUTPUT 层与 C5 层形成全连接，C5 层的 120 个输出依次与 OUTPUT 层的 10 个神经元连接，每个神经元再增加一个偏置。最后由 tanh 函数将结果映射到一个取值范围为（-1,1）的区间。

补充说明（对计算连接数和计算参数个数的说明）：

（1）对普通连接的卷积层：设该卷积层的 map 大小是 a*a、卷积核的大小是 k*k，输入图片数量为 m，特征图数量为 n，每个神经元还连接着一个偏置，所以连接数为：

$$(a*a*k*k*m*n)+(a*a*n)$$

可训练参数数为：

$$(k * k * m * n) + n$$

（2）对采样层（不考虑连接前向连接）：对于采样层有 m = n,连接数为：

$$(a * a * k * k * m) + (a * a * n)$$

可训练参数数为：

$$m + n$$

2.1.2 配置 OpenCV

OpenCV 是一个基于 BSD 许可（开源）发行的跨平台计算机视觉库，它主要是利用 C++编写的，主要接口也是 C++语言，实现了图像处理和计算机视觉的许多通用算法，利用它提供的类可以轻而易举的解决许多问题。

如果想使用 OpenCV 就必须在官网上下载。本作品在制作期间使用了当期的较新版本 OpenCV-4.0.1。下载完成后将它放在一个独立的文件夹中。如果想使用 OpenCV 的库和类就必须要能够预处理包含这些相应的文件。但是它不是 C++标准类库不能直接包含进来，必须要进行一些工作才可以。

首先我们使用 Visual Studio 创建一个新的 C++项目，创建好了之后打开“解决方案资源管理器”，右键单击项目名称，再左键单击最下面的“属性”，打开后如图所示：

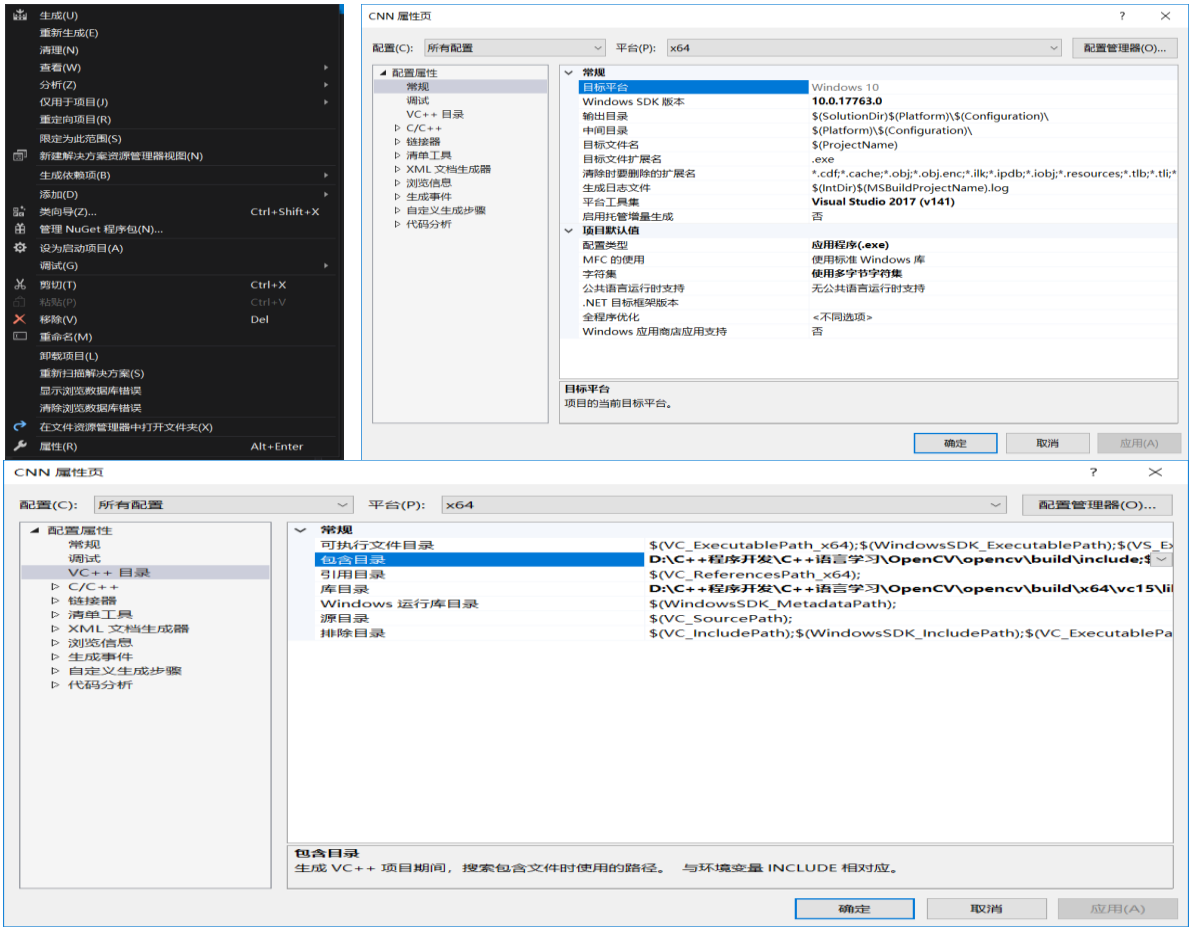


图 2-3 配置 OpenCV C++项目

单击左边的“VC++目录”，看到右边有第二行的“包含目录”和“库目录”，我们需要依次对这两个目录进行配置。于此同时我们发现窗口的右侧有一个小箭头图标，把它打开并且点击“编辑”，会得到如下图所示的内容：

点击“新行”图标，会出现如左图所示的小空白行，点击右侧的“...”，会出现选择文件夹路径的窗口。这时需要把 opencv\build\include 文件包含进去。这里有一个容易混淆的地方，就是 build 文件夹和另一个 sources 文件夹下都有 include，注意要讲 build 下的 include 包含进去才可以。包含好了之后点击确定。会发现左图中“计算的值”下出现了你包含的内容，说明操作成功了。操作完成之后关闭窗口。

之后利用同样的方法对库目录进行类似的操作，只不过包含的不是刚才的文件了。这里需要包含 opencv\build\x64\vc15\lib 文件。注意：我们用的是 Visual Studio 2017，如果是 2015 版本的需要将 x64 下的 vc14\lib 包含进去才能兼容。同样也是 build 下的而不是 sources 下的。包含好了之后点击确定。会发现左图中“计算的值”下出现了你包含的内容，说明操作成功了。操作完成之后关闭窗口。

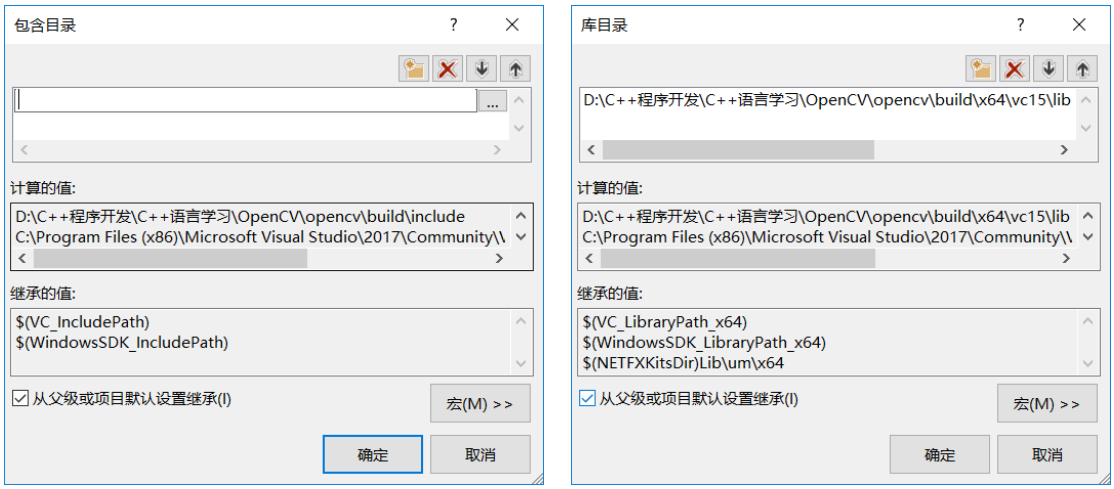


图 2-4 配置 OpenCV 目录

接着，打开“链接器”下的“输入”，单击“附加依赖项”后面的小箭头。打开“编辑”。

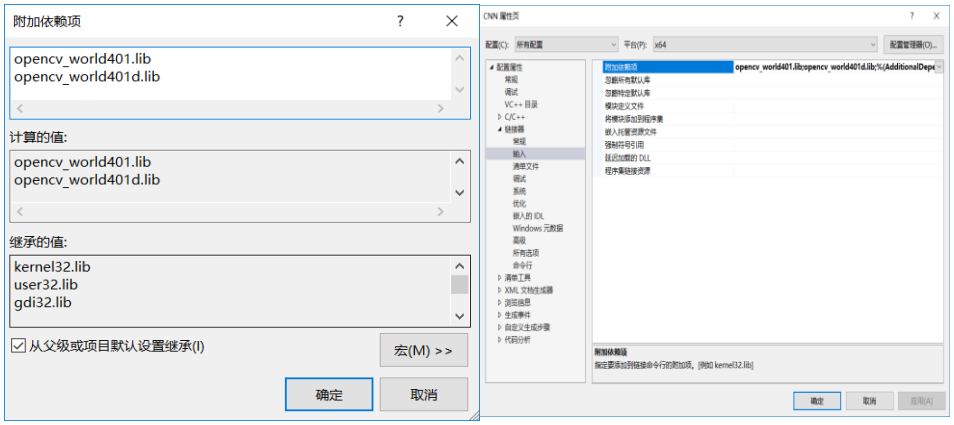


图 2-5 配置 OpenCV 附加依赖项

由于刚才已经包含了 opencv\build\x64\vc15\lib，所以依赖项可以直接将库名输入，为“opencv_world401.lib”和“opencv_world401d.lib”。包含好了在之后如图 2-2(e)所示，然后将这个窗口与上一个窗口都关闭。

接下来的一个较为关键的问题就是将调试器从 x86 改为 x64，否则程序就会莫名其妙的报错。因为我们的 OpenCV 使用的是 x64 的库。

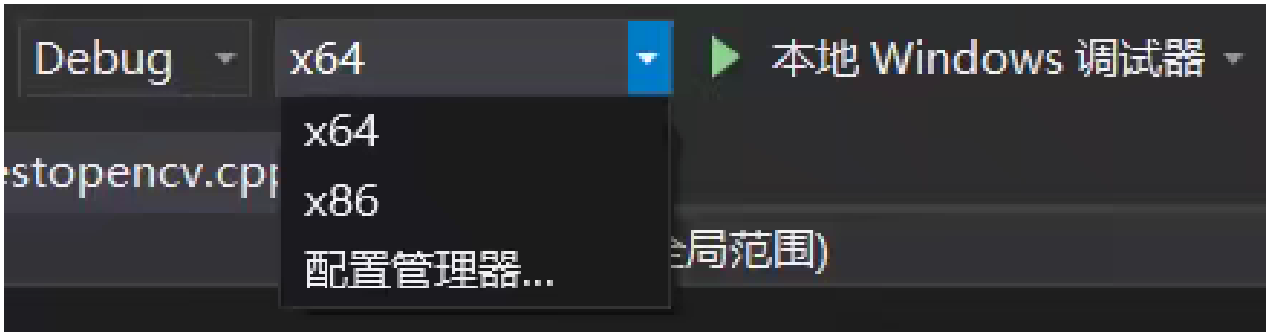


图 2-6 配置 OpenCV 调试器修改

下一步配置系统环境变量，在用户变量下的“Path”中，添加“opencv\build\x64\vc15\bin”如图 2-2(f)所示。

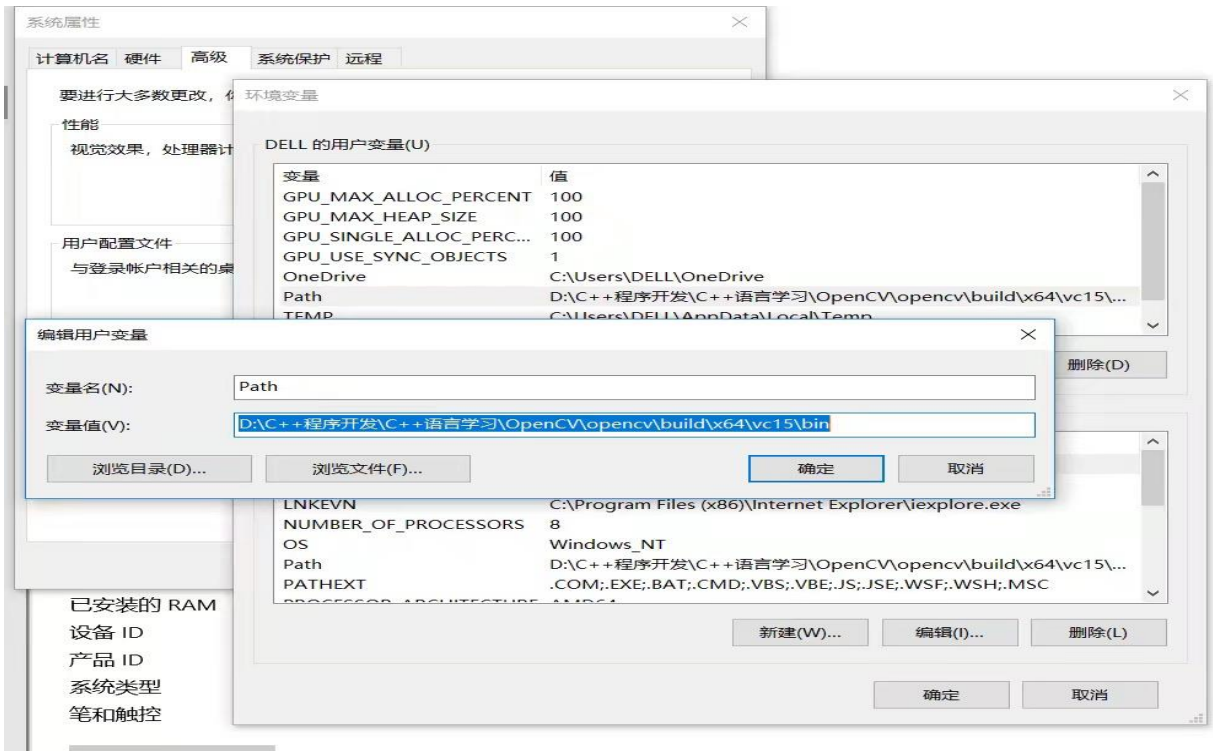


图 2-7 配置 OpenCV 环境变量

图 2-8 配置 OpenCV 成功开启摄像头

最后包含相关文件和调用该库的函数，以能够打开摄像机为成功标志，如图 2-2(g)所示

2.2 CNN_MNIST 类实现原理

2.2.1 常量的定义

根据前文定义的每层的数据表格，定义一些不变的常量方便使用。

定义每层输入图片的高与宽：

```
#define  width_image_input          32  // 输入图片宽
#define  height_image_input         32  // 输入图片高
#define  width_image_C1             28  // (32-5+1)*(32-5+1)
#define  height_image_C1            28  // (32-5+1)*(32-5+1)
#define  width_image_S2             14  // (28/2)*(28/2)
#define  height_image_S2            14  // (28/2)*(28/2)
#define  width_image_C3             10  // (14-5+1)*(14-5+1)
#define  height_image_C3            10  // (14-5+1)*(14-5+1)
#define  width_image_S4             5   // (10/2)*(10/2)
#define  height_image_S4            5   // (10/2)*(10/2)
#define  width_image_C5             1   // (5-5+1)*(5-5+1)
#define  height_image_C5            1   // (5-5+1)*(5-5+1)
#define  width_image_output         1   // 全连接
#define  height_image_output        1   // 全连接
```

定义两种卷积核的高与宽：

```
#define  width_kernel_conv          5  // 卷积层卷积核宽
#define  height_kernel_conv         5  // 卷积层卷积核高
#define  width_kernel_pooling       2  // 采样层卷积核宽
#define  height_kernel_pooling      2  // 采样层卷积核高
```

定义每层的 map 数量：

```
#define num_map_input               1    // 输入层map数
#define num_map_C1                  6    // C1层map数
#define num_map_S2                  6    // S2层map数
#define num_map_C3                  16   // C3层map数
#define num_map_S4                  16   // S4层map数
#define num_map_C5                  120  // C5层map数
#define num_map_output              10   // 输出层 map 数
```

定义训练和测试图片的数量：

```
#define num_train                   60000  // 训练图片数量
#define num_test                    10000  // 测试图片数量
```

定义每层不同参数的数量：

```
#define len_weight_C1           150    // C1层权值个数，5*5*6*1=150
#define len_bias_C1             6      // C1层阈值个数，6
#define len_weight_S2           6      // S2层权值个数,1*6=6
#define len_bias_S2             6      // S2层阈值个数,6
#define len_weight_C3           2400   // C3层权值个数，5*5*16*6=2400
#define len_bias_C3             16     // C3层阈值个数,16
#define len_weight_S4           16     // S4层权值个数，1*16=16
#define len_bias_S4             16     // S4层阈值个数，16
#define len_weight_C5           48000  // C5层权值个数，5*5*16*120=48000
#define len_bias_C5             120    // C5层阈值个数，120
#define len_weight_output       1200   // 输出层权值个数，120*10=1200
#define len_bias_output         10     // 输出层阈值个数，10
```

定义每层的神经元数量：

```
#define num_neuron_input        1024   // 32*32=1024
#define num_neuron_C1           4704   // 28*28*6=4704
#define num_neuron_S2           1176   // 14*14*6=1176
#define num_neuron_C3           1600   // 10*10*16=1600
#define num_neuron_S4           400    // 5*5*16=400
#define num_neuron_C5           120    // 1*120=120
#define num_neuron_output       10     // 1*10=10
```

其他所需定义：

```
#define num_epochs              100    // 最大迭代次数
#define accuracy_rate           0.985   // 需要达到的正确率
#define learning_rate           0.01    // 学习率
```

2.2.2 数组的定义

根据前文定义的每层的数据表格，定义一些矩阵。

定义前向传播神经元：

```
double neuron_input[num_neuron_input];
double neuron_C1[num_neuron_C1];
double neuron_S2[num_neuron_S2];
double neuron_C3[num_neuron_C3];
double neuron_S4[num_neuron_S4];
double neuron_C5[num_neuron_C5];
double neuron_output[num_neuron_output];
```

定义反向传播神经元误差：

```
double delta_neuron_output[num_neuron_output];
double delta_neuron_C5[num_neuron_C5];
double delta_neuron_S4[num_neuron_S4];
double delta_neuron_C3[num_neuron_C3];
double delta_neuron_S2[num_neuron_S2];
double delta_neuron_C1[num_neuron_C1];
double delta_neuron_input[num_neuron_input];
```

定义更新后的参数：

```
double weight_C1[len_weight_C1];
double bias_C1[len_bias_C1];
double weight_S2[len_weight_S2];
double bias_S2[len_bias_S2];
double weight_C3[len_weight_C3];
double bias_C3[len_bias_C3];
double weight_S4[len_weight_S4];
double bias_S4[len_bias_S4];
double weight_C5[len_weight_C5];
double bias_C5[len_bias_C5];
double weight_output[len_weight_output];
double bias_output[len_bias_output];
```

定义更新前的参数：

```
double E_weight_C1[len_weight_C1];
double E_bias_C1[len_bias_C1];
double E_weight_S2[len_weight_S2];
double E_bias_S2[len_bias_S2];
double E_weight_C3[len_weight_C3];
double E_bias_C3[len_bias_C3];
double E_weight_S4[len_weight_S4];
double E_bias_S4[len_bias_S4];
double* E_weight_C5;
double* E_bias_C5;
double* E_weight_output;
double* E_bias_output;
```

定义参数误差：

```
double delta_weight_C1[len_weight_C1];
double delta_bias_C1[len_bias_C1];
double delta_weight_S2[len_weight_S2];
double delta_bias_S2[len_bias_S2];
double delta_weight_C3[len_weight_C3];
double delta_bias_C3[len_bias_C3];
double delta_weight_S4[len_weight_S4];
double delta_bias_S4[len_bias_S4];
double delta_weight_C5[len_weight_C5];
double delta_bias_C5[len_bias_C5];
double delta_weight_output[len_weight_output];
double delta_bias_output[len_bias_output];
```

2.2.3 基本功能函数的定义

为了实现卷积神经网络的基本结构，必须整合出常用的算法并封装成函数。以下写出了这些功能函数的函数原型和实现。在 CNN_MNIST 类中选择双曲正切函数 tanh 作为激活函数。

激活函数 tanh：

```
double CNN_MNIST::activation_function_tanh(double x)
{
    double ep = std::exp(x);
    double em = std::exp(-x);
    return (ep - em) / (ep + em);
}
```

激活函数 tanh 的导数：

```
double CNN_MNIST::activation_function_tanh_derivative(double x)
{
    return (1.0 - x * x);
}
```

均方误差损失函数 MSE：

```
double CNN_MNIST::loss_function_mse(double y, double t)
{
    return (y - t) * (y - t) / 2;
}
```

均方误差损失函数 MSE 的导数：

```
double CNN_MNIST::loss_function_mse_derivative(double y, double t)
{
    return (y - t);
}
```

均方误差损失函数 MSE 的梯度（对参数求偏导）：

```
void CNN_MNIST::loss_function_gradient(const double* y, const double* t, double* dst,
int len)
{
    for (int i = 0; i < len; i++)
    {
        dst[i] = loss_function_mse_derivative(y[i], t[i]);
    }
}
```

计算向量的点乘结果：

```
double CNN_MNIST::dot_product(const double* s1, const double* s2, int len)
{
    double result = 0.0;
    for (int i = 0; i < len; i++)
    {
        result += s1[i] * s2[i];
    }
    return result;
}
```

初始化函数（将长度为 len 的数组每项初始化为 c 的函数）：

```
void CNN_MNIST::init_variable(double* val, double c, int len)
{
    for (int i = 0; i < len; i++)
    {
        val[i] = c;
    }
}
```

计算矩阵先乘后加的函数（multiply -add）：

```
bool CNN_MNIST::muladd(const double* src, double c, int len, double* dst)
{
    for (int i = 0; i < len; i++)
    {
        dst[i] += (src[i] * c);
    }
    return true;
}
```

2.2.4 利用生成随机数初始化所有权重和偏置

人工神经网络的目的在于将杂乱无章的参数（权重和偏置）通过梯度下降的方法变成期望输出的参数。这里，我们必须要初始化一些参数。但是参数又不能漫无目的的初始化，如果不规定范围，或是不规定参数初始化时值的分布，就会导致参数过于杂乱无章而使得训练进程变得无限拉长。前文有提到，一般情况下，网络的连接权值和阈值被初始化在一个具有以 0 为平均值的相对小的区间内的均匀分布上。

那么 C++如何生成服从均匀分布的参数呢？这就要用到 C++11 的 random 库了，它主要包括随机数引擎类和随机数分布类。随机数引擎类是可以独立运行的随机数发生器，它以均匀的概率生成某一类型的随机数，但无法指定随机数的范围、概率等信息，因此，它也被称为“原始随机数发生器”，由于不能指定生成随机数的范围，它通常不会被单独使用。随机数分布类是一个需要有随机数引擎类的支持才能运行的类，而且它能根据用户的需求利用随机数引擎生成符合条件的随机数，例如在某一区间、服从某一分布的随机数。

而我们需要参数服从的连续型均匀分布，来自于 C++11 提供的均匀分布模板类，它们是 std::uniform_int_distribution 和 std::uniform_real_distribution。前者能生成指定范围的随机整数，后者可以生成指定范围的随机浮点数。因为它们都是模板类，所以需要一个类型来实例化它们。这里我们用 std::uniform_real_distribution，以等概率在一定范围内产生 double 类型的随机数，具体语法为:

```
std::uniform_real_distribution<double> dst(min, max);
```

min 和 max 是对象的两个参数，分别表示范围下限和上限。另外需要注意的是范围为左闭右开，即[min, max)。但是这句话只是声明了有一个上下限分别是 min 和 max 的 double 类型的 std::uniform_real_distribution 的对象 dst，并不是说 dst 现在就是随机数了，要想让 dst 成为随机数，还需要给它一个种子。我们可以利用 C++11 产生随机数的办法每次声明 dst 之前声明一个随机数 rd 作为 dst 的种子，再利用 std::uniform_real_distribution 生成以[0.0, 1.0)为范围的服从均匀分布的随机数 dst(rd)，语法如下：

```
std::random_device rd;
```

```
std::uniform_int_distribution<double> dst(0.0, 1.0);
std::cout << dst(rd) << std::endl;
```

std::random_device 是生成非确定随机数的均匀分布整数随机数生成器，可以理解为前文提到的随机数引擎，用它的对象 rd 作为 dst 的种子，即 dst(rd)即可实现以往 C++中 srand(time(NULL))和 rand()的功能了。只不过此时 dst(rd)虽是随机数，但是却是服从某一范围的均匀分布的，在这个范围内，所有 double 的类型数值出现的概率都是相等的。

我们还可以对这个过程进行改进，可以用 C++11 的一种全新的随机数算法，马特赛特旋转演算法，是伪随机数发生器之一，其主要作用是生成伪随机数，语法为 std::mt19937。它是一种随机数算法，用法与 rand()函数类似，但是具有速度快，周期长的特点(它的名字便来自周期长度：2^19937-1)。我们知道 rand()在 windows 下生成的数据范围为 0~32767，但是这个函数的随机范围大概在(-maxint,+maxint)，maxint 为 int 类型最大值。但是它的具体原理超出了我们的范围，这里我们直接拿来应用。同上一个道理，让 rd 做 std::mt19937 的种子，而让 std::mt19937 的对象做 dst 的种子。语法为：

```
std::random_device rd;
std::mt19937 gen(rd());
std::uniform_int_distribution<double> dst(0.0, 1.0);
std::cout << dst(gen) << std::endl;
```

所以有了以上内容我们就可以定义我们的随机数产生函数：

定义均匀分布随机数产生函数：

```
double CNN_MNIST::uniform_rand(double min, double max)
{
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<double> dst(min, max);
    return dst(gen);
}
```

注意到这个函数每次只能产生一个随机数 dst(gen)，所以为了给全部的权重和偏置进行初始化，我们还希望有一个能够进行循环赋值的函数。以目标参数向量、向量长度、均匀分布的上下限为参数。

重载均匀分布随机数产生函数：

```
bool CNN_MNIST::uniform_rand(double* src, int len, double min, double max)
{
    for (int i = 0; i < len; i++)
    {
        src[i] = uniform_rand(min, max);
    }
    return true;
}
```

补充说明：以上内容均需要预处理#include <random>。

现在我们有了初始化参数的两个函数。但是这两个函数的接口指定了三个条件，分别是目标参数数组、均匀分布下限、均匀分布上限。对区间来说，要求以 0 为平均值，所以应该把值为 0 的点作为区间的中点。下限和上限的点关于值为 0 的点成左右对称。

经过分析，由于卷积网络越深，深层的 map 能表达的信息就越少（但是越具体），为了弥补信息量，我们需要更多的卷积核去卷积前一层的不同特征图，这就意味着会有更多的连接参数，这在 LeNet-5 模型的设计中（前文表格）已经能直观的看到了。在浅层次中，我们为了更好地捕捉特定的特征，会将随机参数服从的均匀分布区间设置的更大一些；但是随着层次的深入，参数增多，然而特征已经提取得足够详细了，为了能够更快的训练，会将这个区间设置的更窄一点。

以上内容对权重向量生效，而对于偏置，它所起的作用并不是很大。为了训练更加节省时间，将全部的偏置向量初始化为 0，它们可以从 0 开始训练。但需要注意的是，我们不能让权重和偏置全都从 0 开始训练，那样就无法进行求梯度的反向传播了，因为求出的梯度也全部是 0，参数无法更新。所以我们只能采用让期望是 0 的均匀分布进行训练。

在这里解释一下为什么让期望的均值是 0：我们知道 MSE 采用的是最小化损失函数值的方式进行训练的，当 MSE 最小化时也就是 MSE 的值无限接近 0 的时候。我们分析 MSE 的自变量就可以发现它们其实都是我们上文正在初始化的权重和偏置，为了让这些权重和偏置参数的所指向的函数值能无限趋近于 0，所以才有了梯度下降和参数更新一说。不让全部的参数等于 0 也是因为虽然此时 MSE 确实等于 0，但是它是无效的，因为无论如何训练 MSE 从始至终都等于 0，而参数全为 0 的组合无论训练哪一张图片它输出的结果都是一样的，所以是无效的。所以我们必须要将所有的参数分布在一个以 0 为期望的均匀分布区间中，这样才能够产生训练，在特定的输入下训练出特定的参数组合，令这个组合让 MSE 的值无限趋于 0（不可能为 0），当达到要求的正确率的时候停止，此时的参数组合就会唯一的指向一个输出，那就是我们想要的期望输出。

为了达到以上目的，参考的代码如下**（黑体带下划线内容表示用中文做过的处理）**：

```
void CNN_MNIST::initWeightBias()
{
    const double scale = 一个比较小的数作为规模;

    double upper;
    double lower;

    upper = -std::sqrt(scale / 一个较小的数);
    lower = std::sqrt(scale /一个较小的数);
    uniform_rand(weight_C1, len_weight_C1, upper, lower);
    for (int i = 0; i < len_bias_C1; i++) {
        bias_C1[i] = 0.0;
    }

    upper = -std::sqrt(scale / 5.0);
    lower = std::sqrt(scale / 5.0);
    uniform_rand(weight_S2, len_weight_S2, upper, lower);
    for (int i = 0; i < len_bias_S2; i++) {
        bias_S2[i] = 0.0;
    }
```

```
    upper = -std::sqrt(scale /一个不大不小的数);
    lower = std::sqrt(scale /一个不大不小的数);
    uniform_rand(weight_C3, len_weight_C3, upper, lower);
    for (int i = 0; i < len_bias_C3; i++) {
        bias_C3[i] = 0.0;
    }

    upper = -std::sqrt(scale / 5.0);
    lower = std::sqrt(scale / 5.0);
    uniform_rand(weight_S4, len_weight_S4, upper, lower);
    for (int i = 0; i < len_bias_S4; i++) {
        bias_S4[i] = 0.0;
    }

    upper = -std::sqrt(scale /一个较大的数);
    lower = std::sqrt(scale /一个较大的数);
    uniform_rand(weight_C5, len_weight_C5, upper, lower);
    for (int i = 0; i < len_bias_C5; i++) {
        bias_C5[i] = 0.0;
    }

    upper = -std::sqrt(scale /一个较小的数);
    lower = std::sqrt(scale /一个较小的数);
    uniform_rand(weight_output, len_weight_output, upper, lower);
    for (int i = 0; i < len_bias_output; i++) {
        bias_output[i] = 0.0;
    }
}
```

补充说明：以上内容需要#include <cmath>。

2.2.5 前向传播前的一些预处理工作

这里我们要用到 C++11 中的一个叫做 std::pair 的数据类型。std::pair 这个数据类型可以用来表示连接端的两头，分别用 pair.first 和 pair.second 表示 pair 中的第一个数据和第二个数据的值。首先进行定义：

```
typedef vector<pair<int, int> > wi_connections;
typedef vector<pair<int, int> > wo_connections;
typedef vector<pair<int, int> > io_connections;
```

这个定义的含义是，connections 表示连接，前面的两个字母第一个表示 pair.first，第二个字母表示 pair.second。例如 wi 表示 weight_id 和 in_id。但是由于我们的 std::pair 的数据类型会用在连接中，是以矩阵表示的，所以必须要形成二维数组，所以还需要再定义一组基本的二维数组：

```
vector<wi_connections> out2wi_S2;    // out_id -> [(weight_id, in_id)]
```

```
vector<wi_connections> out2wi_S4;
vector<wo_connections> in2wo_C3;    // in_id -> [(weight_id, out_id)]
vector<io_connections> weight2io_C3;  // weight_id -> [(in_id, out_id)]
vector<wo_connections> in2wo_C1;
vector<io_connections> weight2io_C1;

vector<int> out2bias_S2;
vector<int> out2bias_S4;
vector<vector<int> > bias2out_C3;
vector<vector<int> > bias2out_C1;
```

上述的偏置部分有些奇特，采样层的 S2 和 S4 是一维的。这是因为他们用在前向传播中，在这个过程中他们会用作 bias 数组的元素从而形成矩阵。而后者的偏置矩阵中的每一个元素则会在反向传播中用来循环作为神经元误差的角标，以达到计算误差的效果。

计算前向传播中的 out2bias 一维数组的计算函数为如下，它的功能是扩展 out2bias 参数到前几个参数指定的大小，而 push_back 的值是什么却无所谓：

```
void CNN_MNIST::calc_out2bias(int width, int height, int depth, std::vector<int>&
out2bias)
{
    for (int i = 0; i < depth; i++) {
        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                out2bias.push_back(i);
            }
        }
    }
}
```

计算反向传播中的 bias2out 二维矩阵的计算函数为如下，它的功能是扩展 bias2out 参数到前几个参数指定的大小，而 push_back 的值是什么却无所谓，注意它是二维的，有二维的扩展方式：

```
void CNN_MNIST::calc_bias2out(int width_in, int height_in, int width_out, int
height_out, int depth_in, int depth_out, std::vector<std::vector<int> >& bias2out)
{
    int len = depth_in;
    bias2out.resize(len);
    for (int c = 0; c < depth_in; c++) {
        for (int y = 0; y < height_out; y++) {
            for (int x = 0; x < width_out; x++) {
                int index_out = get_index(x, y, c, width_out, height_out, depth_out);
                bias2out[c].push_back(index_out);
            }
        }
    }
}
```

我们还需要一类函数，它能够帮助指针找到在某一邻域内具体的区域。一般处理图片的指针都是以某一大小的邻域为单位进行移动的，为了让它们能够在该邻域内移动，需要一种获取地址值的 get_index 的实现方法：

```
int CNN_MNIST::get_index(int w, int h, int channel, int width, int height, int depth)
{
    assert(w >= 0 && w < width);
    assert(h >= 0 && h < height);
    assert(channel >= 0 && channel < depth);
    return (height * channel + h) * width + w;
}
```

最后是一类计算权重连接的函数了。它们大体上的思路与上述一致，基本上就是为了扩展。有三种，第一个是 out 数组指向 pair（weight，in）的，第二个是 in 数组指向 pair（weight，out）的，第三个是 weight 指向 pair（in，out）的。最主要的思路就是将最后一个参数扩展到前面几个参数指定的大小。例如如下的例子是上述第一个的实现过程：

```
void CNN_MNIST::calc_out2wi(int width_in, int height_in, int width_out, int
height_out, int depth_out, std::vector<wi_connections>& out2wi)
{
    for (int i = 0; i < depth_out; i++) {
        int block = width_in * height_in * i;

        for (int y = 0; y < height_out; y++) {
            for (int x = 0; x < width_out; x++) {
                int rows = y * width_kernel_pooling;
                int cols = x * height_kernel_pooling;

                wi_connections wi_connections_;
                std::pair<int, int> pair_;

                for (int m = 0; m < width_kernel_pooling; m++) {
                    for (int n = 0; n < height_kernel_pooling; n++) {
                        pair_.first = i;
                        pair_.second = (rows + m) * width_in + cols + n + block;
                        wi_connections_.push_back(pair_);
                    }
                }
                out2wi.push_back(wi_connections_);
            }
        }
    }
}
```

由于这类函数很长，剩余两种就不予举例了。

2.2.6 前向传播函数

LeNet-5 模型中的前向传播包括从输入图片到 C1、C1 到 S2、S2 到 C3、C3 到 S4、S4 到 C5、C5 到输出层的过程。前向传播主要计算神经元的输出值，其中卷积层 C1、C3、C5 过程类似，采样层 S2、S4 过程类似。

例如，C1 层的前向传播工作原理为首先初始化 C1 层神经元的值为 0，然后利用 get_index 得到三种类型的地址值 addr1、addr2、addr3。然后将权重数组和 addr1 配对赋给 pw，将输入图片和 addr2 配对赋值给 pi，将神经元的值和 addr3 配对赋值给 pa。接下来利用这三个指针操作。神经元数为(28*28)*6=4704，分别用每一个 5*5 的卷积图像去乘以 32*32 的图像，获得一个 28*28 的图像，即对应位置相加再求和，stride 长度为 1；一共 6 个 5*5 的卷积图像，然后对每一个神经元加上一个阈值，最后再通过 tanh 激活函数对每一神经元进行运算得到最终每一个神经元的结果。

其中 C1、C3、C5 的操作过程类似，S2、S4 的操作过程类似。如下举例为前向传播中 C1 层的函数实现。由于这类前向传播函数过长而且复杂，但大体实现方法又很类似，所以只举一个例子。

```
bool CNN_MNIST::Forward_C1()
{
    init_variable(neuron_C1, 0.0, num_neuron_C1);
    for (int i = 0; i < num_map_C1; i++) // create C1 maps, 6 C1 maps
    {
        for (int j = 0; j < num_map_input_; j++)
        {
            int addr1 = get_index(0, 0, num_map_input * i + j, width_kernel_conv,
height_kernel_conv, num_map_C1 * num_map_input);
            int addr2 = get_index(0, 0, j, width_image_input, height_image_input,
num_map_input);
            int addr3 = get_index(0, 0, i, width_image_C1, height_image_C1,
num_map_C1);

            const double* pw = &weight_C1[0] + addr1;
            const double* pi = data_single_image + addr2;
            double* pa = &neuron_C1[0] + addr3;

            for (int y = 0; y < height_image_C1; y++)
            {
                for (int x = 0; x < width_image_C1; x++)
                {
                    const double* ppw = pw;
                    const double* ppi = pi + y * width_image_input + x;
                    double sum = 0.0;

                    for (int wy = 0; wy < height_kernel_conv; wy++) {
                        for (int wx = 0; wx < width_kernel_conv; wx++) {
                            sum += *ppw++ * ppi[wy * width_image_input + wx];
                        }
                    }
```

```
                    pa[y * width_image_C1 + x] += sum;
                }
            }
        }

        int addr3 = get_index(0, 0, i, width_image_C1, height_image_C1, num_map_C1);
        double* pa = &neuron_C1[0] + addr3;
        double b = bias_C1[i];
        for (int y = 0; y < height_image_C1; y++)
        {
            for (int x = 0; x < width_image_C1; x++)
            {
                pa[y * width_image_C1 + x] += b;
            }
        }
    }

    // reflect every neuron with tanh
    for (int i = 0; i < num_neuron_C1; i++)
    {
        neuron_C1[i] = activation_function_tanh(neuron_C1[i]);
    }

    return true;
}
```

2.2.7 反向传播函数

主要计算每层权值和偏置的误差以及每层神经元的误差；其中输入层、S2 层、S4 层操作过程相同；C1 层、C3 层操作过程相同。

计算 C5 层神经元误差、输出层权值误差、输出层偏置误差；通过输出层神经元误差乘以输出层权值，求和，结果再乘以 C5 层神经元的 tanh 激活函数的导数(即 1-C5 层神经元值的平方)，获得 C5 层每一个神经元误差；通过输出层神经元误差乘以 C5 层神经元获得输出层权值误差；输出层偏置误差即为输出层神经元误差。

然后计算 S4 层神经元误差、C5 层权值误差、C5 层偏置误差；通过 C5 层权值乘以 C5 层神经元误差，求和，结果再乘以 S4 层神经元的 tanh 激活函数的导数(即 1-S4 神经元的平方)，获得 S4 层每一个神经元误差；通过 S4 层神经元乘以 C5 层神经元误差，求和，获得 C5 层权值误差；C5 层偏置误差即为 C5 层神经元误差。

然后计算 C3 层神经元误差、S4 层权值误差、S4 层偏置误差；通过 S4 层权值乘以 S4 层神经元误差，求和，结果再乘以 C3 层神经元的 tanh 激活函数的导数(即 1-S4 神经元的平方)，然后再乘以 1/4，获得 C3 层每一个神经元误差；通过 C3 层神经元乘以 S4 神经元误差，求和，再乘以 1/4，获得 S4 层权值误差；通过 S4 层神经元误差求和，来获得 S4 层偏置误差。

然后计算 S2 层神经元误差、C3 层权值误差、C3 层偏置误差；通过 C3 层权值乘以 C3 层神经元误差，求和，结果再乘以 S2 层神经元的 tanh 激活函数的导数(即 1-S2 神经元的平方)，获得 S2 层每一个神经元误差；通过 S2 层神经元乘以 C3 层神经元误差，求和，获得 C3 层权值误差；C3 层偏置误差即为 C3 层神经元误差和。

然后计算 C1 层神经元误差、S2 层权值误差、S2 层偏置误差；通过 S2 层权值乘以 S2 层神经元误差，求和，结果再乘以 C1 层神经元的 tanh 激活函数的导数(即 1-C1 神经元的平方)，然后再乘以 1/4，获得 C1 层每一个神经元误差；通过 C1 层神经元乘以 S2 神经元误差，求和，再乘以 1/4，获得 S2 层权值误差；通过 S2 层神经元误差求和，来获得 S4 层偏置误差。

最后计算输入层神经元误差、C1 层权值误差、C1 层偏置误差；通过 C1 层权值乘以 C1 层神经元误差，求和，结果再乘以输入层神经元的 tanh 激活函数的导数(即 1-输入层神经元的平方)，获得输入层每一个神经元误差；通过输入层层神经元乘以 C1 层神经元误差，求和，获得 C1 层权值误差；C1 层偏置误差即为 C1 层神经元误差和。

举一个 C5 层反向传播的例子，这类函数大体过长又很相似，所以只举一个例子：

```
bool CNN_MNIST::Backward_C5()
{
    init_variable(delta_neuron_C5, 0.0, num_neuron_C5);
    init_variable(delta_weight_output, 0.0, len_weight_output);
    init_variable(delta_bias_output, 0.0, len_bias_output);

    for (int c = 0; c < num_neuron_C5; c++) {
        delta_neuron_C5[c] = dot_product(&delta_neuron_output[0], &weight_output[c
* num_neuron_output], num_neuron_output);
        delta_neuron_C5[c] *= activation_function_tanh_derivative(neuron_C5[c]);
    }

    for (int c = 0; c < num_neuron_C5; c++) {
        muladd(&delta_neuron_output[0], neuron_C5[c], num_neuron_output,
&delta_weight_output[0] + c * num_neuron_output);
    }

    for (int i = 0; i < len_bias_output; i++) {
        delta_bias_output[i] += delta_neuron_output[i];
    }

    return true;
}
```

2.2.8 进行训练

训练之前先将连接用的数组进行清空，以方便之后填充数据：

```
out2wi_S2.clear();
out2bias_S2.clear();
out2wi_S4.clear();
out2bias_S4.clear();
in2wo_C3.clear();
weight2io_C3.clear();
bias2out_C3.clear();
in2wo_C1.clear();
weight2io_C1.clear();
bias2out_C1.clear();
```

然后在前向传播之前，把这些连接用的数组利用最一开始定义的常数进行初始化。

```
calc_out2wi(width_image_C1, height_image_C1, width_image_S2, height_image_S2,
num_map_S2, out2wi_S2);
calc_out2bias(width_image_S2, height_image_S2, num_map_S2, out2bias_S2);

calc_out2wi(width_image_C3, height_image_C3, width_image_S4, height_image_S4,
num_map_S4, out2wi_S4);
calc_out2bias(width_image_S4, height_image_S4, num_map_S4, out2bias_S4);

calc_in2wo(width_image_C3, height_image_C3, width_image_S4, height_image_S4,
num_map_C3, num_map_S4, in2wo_C3);
calc_weight2io(width_image_C3, height_image_C3, width_image_S4, height_image_S4,
num_map_C3, num_map_S4, weight2io_C3);
calc_bias2out(width_image_C3, height_image_C3, width_image_S4, height_image_S4,
num_map_C3, num_map_S4, bias2out_C3);

calc_in2wo(width_image_C1, height_image_C1, width_image_S2, height_image_S2,
num_map_C1, num_map_C3, in2wo_C1);
calc_weight2io(width_image_C1, height_image_C1, width_image_S2, height_image_S2,
num_map_C1, num_map_C3, weight2io_C1);
calc_bias2out(width_image_C1,height_image_C1,width_image_S2,height_image_S2,
num_map_C1, num_map_C3, bias2out_C1);
```

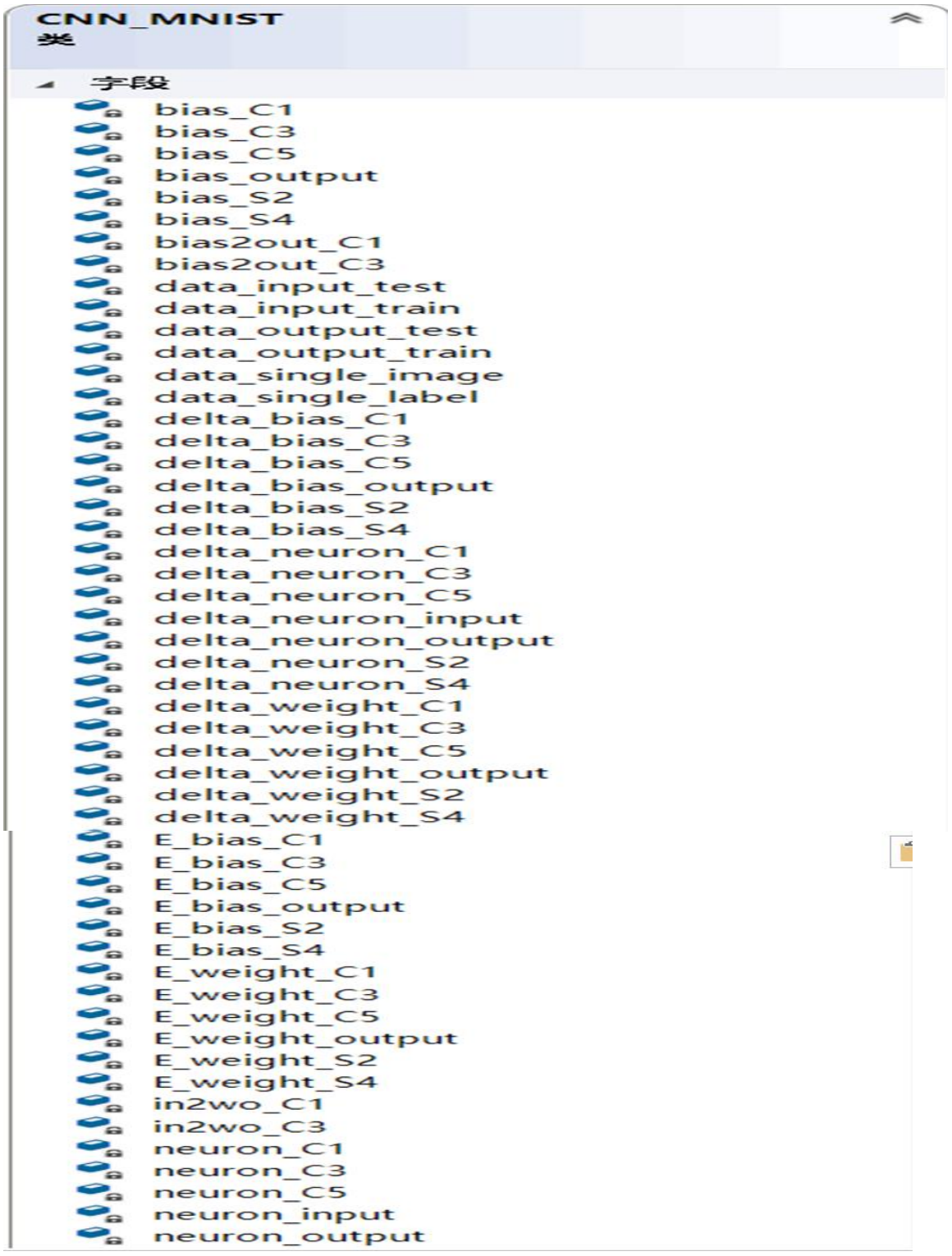
最后利用 ST 训练模式循环 60000 次如下代码：

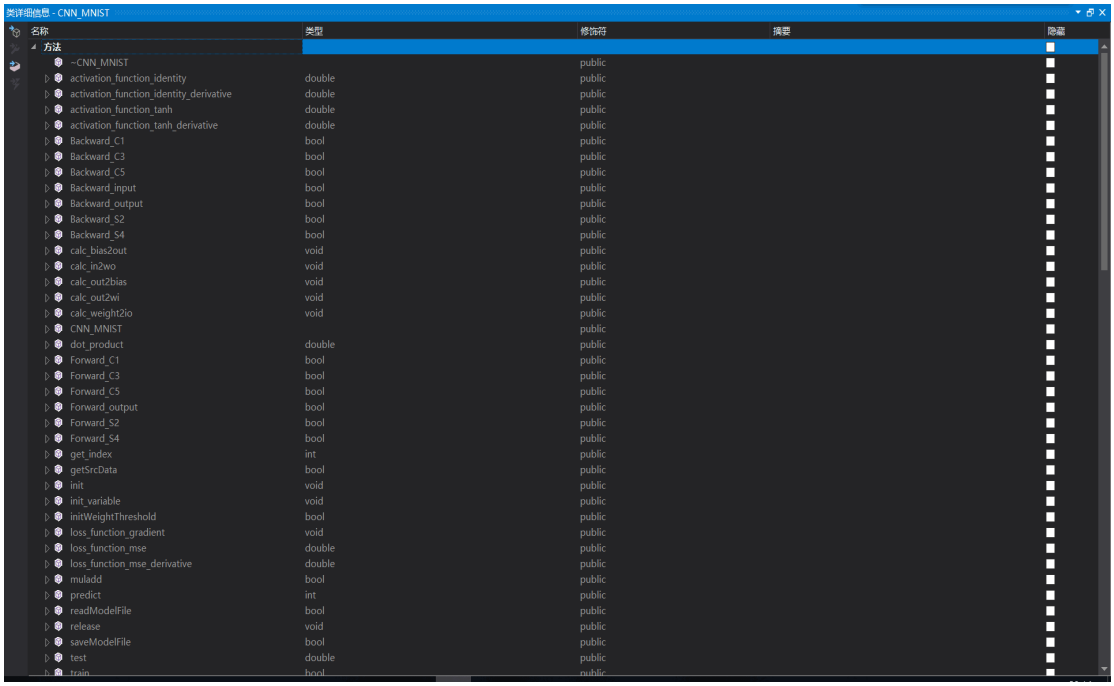
```
Forward_C1();
Forward_S2();
Forward_C3();
Forward_S4();
Forward_C5();
Forward_output();

Backward_output();
Backward_C5();
Backward_S4();
Backward_C3();
Backward_S2();
Backward_C1();
Backward_input();
UpdateWeights();
```

关于最后一个更新参数，前面已经有所描述了。

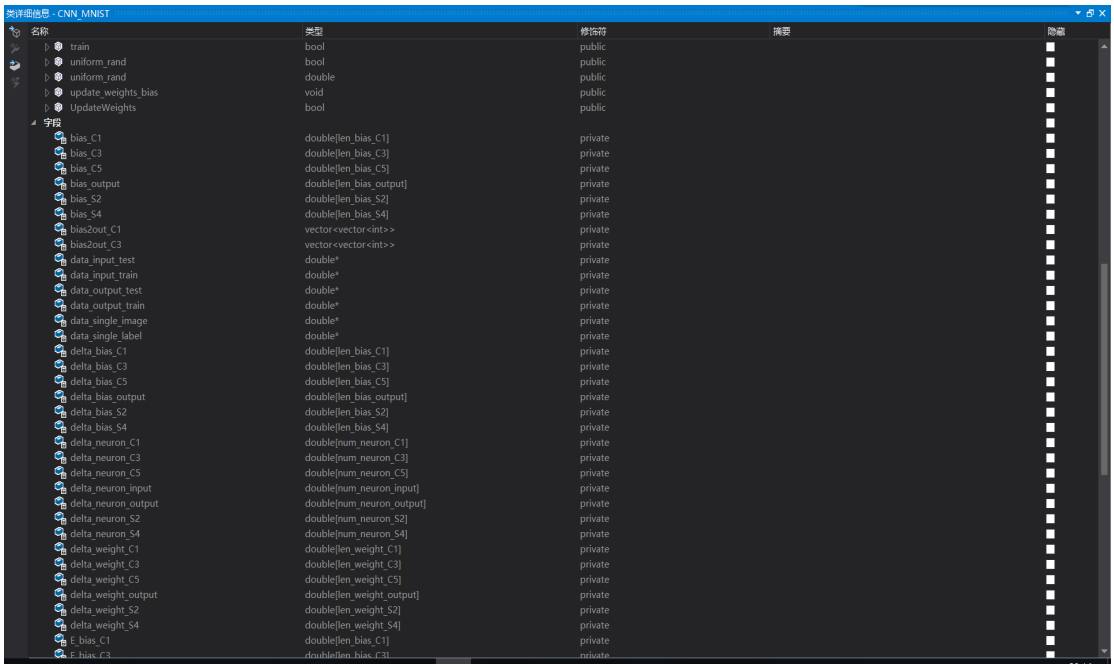
2.2.8 CNN_MINIST 类的类图：

如图 2-3 所示：




类详细信息 - CNN_MNIST

名称	类型	修饰符	摘要	隐藏
方法				
~CNN_MNIST		public		
activation_function_identity	double	public		
activation_function_identity_derivative	double	public		
activation_function_tanh	double	public		
activation_function_tanh_derivative	double	public		
Backward_C1	bool	public		
Backward_C3	bool	public		
Backward_C5	bool	public		
Backward_input	bool	public		
Backward_output	bool	public		
Backward_S2	bool	public		
Backward_S4	bool	public		
calc_bias2out	void	public		
calc_in2wo	void	public		
calc_out2bias	void	public		
calc_out2wi	void	public		
calc_weight2io	void	public		
CNN_MNIST		public		
dot_product	double	public		
Forward_C1	bool	public		
Forward_C3	bool	public		
Forward_C5	bool	public		
Forward_output	bool	public		
Forward_S2	bool	public		
Forward_S4	bool	public		
get_index	int	public		
getSrcData	bool	public		
init	void	public		
init_variable	void	public		
initWeightThreshold	bool	public		
loss_function_gradient	void	public		
loss_function_mse	double	public		
loss_function_mse_derivative	double	public		
muladd	bool	public		
predict	int	public		
readModelFile	bool	public		
release	void	public		
saveModelFile	bool	public		
test	double	public		
train	bool	public		


类详细信息 - CNN_MNIST

名称	类型	修饰符	摘要	隐藏
train	bool	public		
uniform_rand	bool	public		
uniform_rand	double	public		
update_weights_bias	void	public		
UpdateWeights	bool	public		
字段				
bias_C1	double[len_bias_C1]	private		
bias_C3	double[len_bias_C3]	private		
bias_C5	double[len_bias_C5]	private		
bias_output	double[len_bias_output]	private		
bias_S2	double[len_bias_S2]	private		
bias_S4	double[len_bias_S4]	private		
bias2out_C1	vector<vector<int>>	private		
bias2out_C3	vector<vector<int>>	private		
data_input_test	double*	private		
data_input_train	double*	private		
data_output_test	double*	private		
data_output_train	double*	private		
data_single_image	double*	private		
data_single_label	double*	private		
delta_bias_C1	double[len_bias_C1]	private		
delta_bias_C3	double[len_bias_C3]	private		
delta_bias_C5	double[len_bias_C5]	private		
delta_bias_output	double[len_bias_output]	private		
delta_bias_S2	double[len_bias_S2]	private		
delta_bias_S4	double[len_bias_S4]	private		
delta_neuron_C1	double[num_neuron_C1]	private		
delta_neuron_C3	double[num_neuron_C3]	private		
delta_neuron_C5	double[num_neuron_C5]	private		
delta_neuron_input	double[num_neuron_input]	private		
delta_neuron_output	double[num_neuron_output]	private		
delta_neuron_S2	double[num_neuron_S2]	private		
delta_neuron_S4	double[num_neuron_S4]	private		
delta_weight_C1	double[len_weight_C1]	private		
delta_weight_C3	double[len_weight_C3]	private		
delta_weight_C5	double[len_weight_C5]	private		
delta_weight_output	double[len_weight_output]	private		
delta_weight_S2	double[len_weight_S2]	private		
delta_weight_S4	double[len_weight_S4]	private		
E_bias_C1	double[len_bias_C1]	private		
E_bias_C3	double[len_bias_C3]	private		

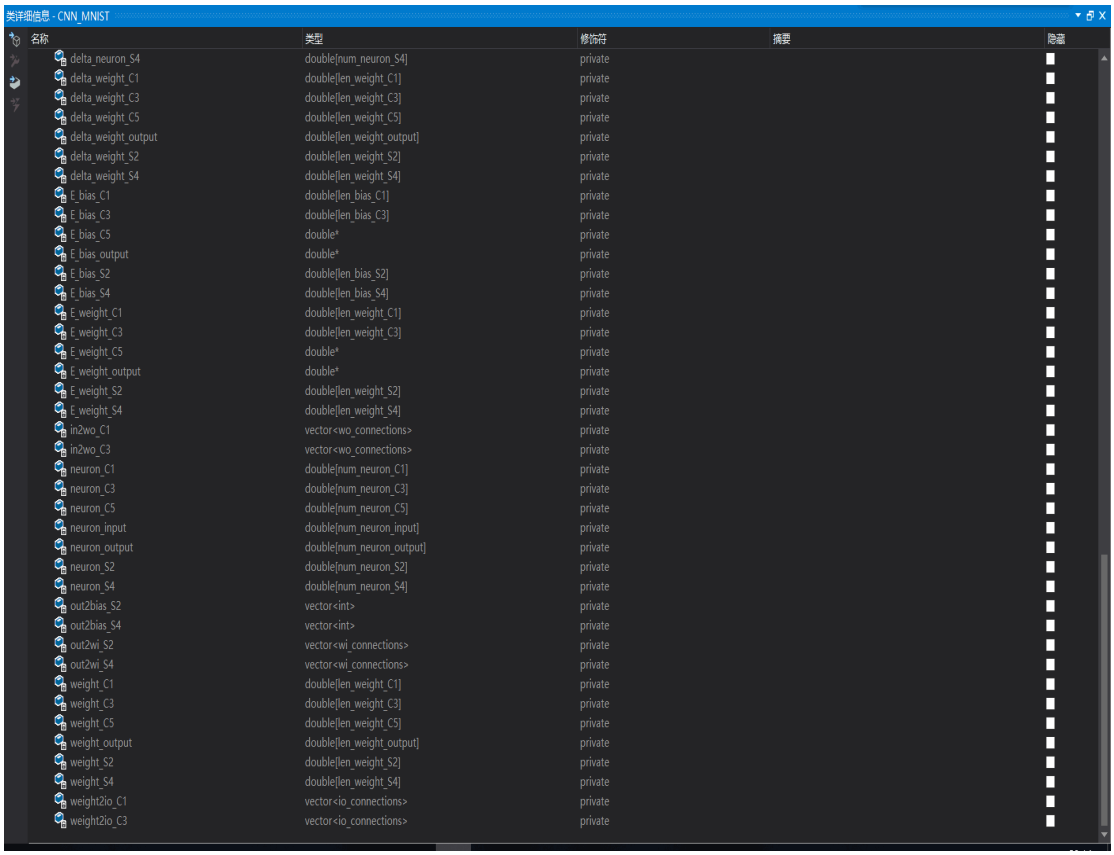


图 2-3 CNN_MINIST 类图

关于数据测试结果：详细内容参考 cnn.model 文件

总　结

整个要实现的类的功能已经全部达到，做到了在进行“训练”的时候不出现任何 BUG。由于电脑配置有限，而且即便是在 VS 的 release 下跑“训练”也需要大概一天的时间才能生成结果——cnn.model 文件，该文件中是训练好的参数模型，训练达标标志为训练中的验证集验证成功率达到 98.5%以上。当然，为了验证该模型的好坏还应该进行进一步的手写测试。但是由于在 openCV 库上的一些问题，导致读取图片失败。虽然导致了数据是否成功的不可视，但毕竟测试手写的内容在 main.cpp 中，所以并不影响整个神经网络训练时所要用到的这个“类”的功能。

整段代码的 95%以上的内容集中在类的实现上，整个类因为功能过于复杂导致相关函数和数据成员过多。在 VS 上跑程序的时候只需要在 main 中调用一个函数就可以了，这就意味着在类中的各函数互相调用的频率更高一些。对此提出的优化措施即应该利用继承和多态进行类的增加，并简化每一个类中的数据。对于一些经常被其他函数调用的函数应该放置在基类中。而且几乎所有常量或常用到的变量和数组应该放在基类中，因为它们是被所有函数共享的。

三个人首先对于深度学习神经网络方法有了比较深刻的理解，反向传播的 BP 算法和梯度下降大致也用 C++成功实现了。基于上文给出的问题，导致测试量比较少，因此还需扩大测试量和测试次数——由于时间有限未能实现。总之应用 ML DL 神经网络相关知识配合 C++这一效率极高的算法、openCV 库理论上实现了手写识别功能，收获颇丰。

参考文献

[1]　Paul Deitel　Harvey Deitel　C++大学教程　电子工业出版社

[2]　H.M.Deitel　P.J.Deitel　C++大学基础教程（第五版）　电子工业出版社

[3]　Stephen Prata　C++ Primer Plus(中文版)　人民邮电出版社

[4]　GoodFellow　深度学习 人民邮电出版社

[5]　周志华 机器学习 清华大学出版社