基于 MSVL 的神经网络框架

Framework for Neural Networks Based on MSVL

冯哲

西安电子科技大学 ● ICTT

符 号 对 照 表

数相关符号

符号	含义
x	标量
$\boldsymbol{x}$	向量
$\boldsymbol{X}$	矩阵
X	张量

操作符相关符号

符号	含义
$(\cdot)^T$	向量或者矩阵的转置
$\odot$	按元素相乘
$\|\cdot\|_p$	L_p 范数
$\|\cdot\|$	L_2 范数
$\sum$	连加
$\prod$	连乘

函数相关符号

符号	含义
$f(\cdot)$	函数
$\log(\cdot)$	自然对数函数
$\exp(\cdot)$	指数函数

导数和梯度相关符号

符号	含义
$\frac{dy}{dx}$	y 关于 x 的导数
$\frac{\partial y}{\partial x}$	y 关于 x 的偏导
$\nabla_{\boldsymbol{x}} y$	y 关于向量 $\boldsymbol{x}$ 的梯度

概率统计相关符号

符号	含义
$\cdot \sim N(\mu, \sigma^2)$	随机变量 $\cdot$ 服从高斯分布
$\cdot \sim U(-\alpha,\ \alpha)$	随机变量 $\cdot$ 服从均匀分布
$E(\cdot)$	随机变量 $\cdot$ 的数学期望
$Var(\cdot)$	随机变量 $\cdot$ 的方差

目 录

第一部分 算法理论基础 **1**

第一章 神经网络介绍 **3**

1.1 神经元 3

1.2 单层神经网络 4

1.3 多层神经网络 6

第二章 神经网络理论基础 **9**

2.1 神经网络激活函数 9

2.1.1 sigmoid [5] 9

2.1.2 tanh 9

2.1.3 relu [14] 10

2.1.4 leaky relu 10

2.1.5 softmax [18] 10

2.2 神经网络损失函数 11

2.2.1 Mean Square Error 11

2.2.2 Cross Entropy Loss [18] 12

2.3 神经网络初始化方法 12

2.3.1 全零初始化 12

2.3.2 随机初始化 13

2.3.3 Xavier 初始化 13

2.3.4 凯明初始化 14

第三章 神经网络的正反向传播 **15**

3.1 神经网络正向传播 17

3.2 神经网络反向传播 [6] 17

3.3 神经网络张量流 19

第四章 神经网络优化方法 **21**
4.1 梯度下降优化方法 21
4.1.1 批量梯度下降优化方法 21
4.1.2 随机梯度下降优化方法 22
4.1.3 小批量梯度下降优化方法 23
4.2 Adam 优化方法 23

第二部分 框架文档 **25**

第五章 框架结构 **27**

第六章 编码说明 **29**
6.1 对象结构体说明 29
6.1.1 矩阵结构体 29
6.1.2 神经网络单层结构体 29
6.1.3 神经网络结构体 30
6.1.4 数据集结构体 30
6.1.5 用户自定义参数结构体 32
6.1.6 Adam 优化参数结构体 32
6.2 基础函数说明 33
6.2.1 辅助函数 33
6.2.2 二维矩阵操作函数 33
6.2.3 激活函数相关函数 37
6.2.4 损失函数相关函数 39
6.2.5 权值初始化函数 42
6.3 开发流程及其函数说明 44
6.3.1 用户自定义参数录入 45
6.3.2 数据集构建 46
6.3.3 神经网络初始化 47
6.3.4 前向传播构建 51
6.3.5 反向传播构建 53
6.3.6 优化算法构建 57
6.3.7 神经网络训练构建 59
6.3.8 神经网络测试构建 59
6.4 项目目录说明 61
6.5 矩阵运算的优化 61
6.5.1 原始矩阵乘操作 61
6.5.2 寄存器优化矩阵乘操作 61

6.5.3 稀疏矩阵优化矩阵乘操作 61
6.5.4 多级缓存优化矩阵乘操作 61

第三部分 应用示例 **63**

第七章 Minst 手写字体识别 **65**
7.1 数据集介绍 65
7.1.1 数据集来源 65
7.1.2 文件说明 65
7.1.3 数据可视化 67
7.2 代码样例解析 67
7.2.1 数据录入 67
7.2.2 主函数 69
7.2.3 输出 72

参考文献 **74**

PART I 第一部分

算法理论基础

第一章

神经网络介绍

人工神经网络（Artificial Neural Network，ANN），简称神经网络（Neural Network，NN）或类神经网络，在机器学习和认知科学领域，是一种模仿生物神经网络的结构和功能的数学模型或计算模型，用于对函数进行估计或近似 [2]。神经网络由大量的神经元联结进行计算。大多数情况下神经网络能在外界信息的基础上改变内部结构，是一种自适应系统，通俗的讲就是具备学习功能 [23]。现代神经网络是一种非线性统计性数据建模工具，神经网络通常是通过一个基于数学统计学类型的学习方法得以优化，所以也是数学统计学方法的一种实际应用，通过统计学的标准数学方法我们能够得到大量的可以用函数来表达的局部结构空间，另一方面在人工智能学的人工感知领域，我们通过数学统计学的应用可以来做人工感知方面的决定问题，一般这种方法比起正式的逻辑学推理演算更具有优势 [19]。

1.1 神经元

对于神经元的研究由来已久，1904 年生物学家就已经知晓了神经元的组成结构。一个神经元通常具有多个树突，主要用来接受传入信息；而轴突只有一条，轴突尾端有许多轴突末梢可以给其他多个神经元传递信息。轴突末梢跟其他神经元的树突产生连接，从而传递信号。这个连接的位置在生物学上叫做“突触”[3]。

1943 年，心理学家 McCulloch 和数学家 Pitts 参考了生物神经元的结构，发表了抽象的神经元模型 MP。神经元模型是一个包含输入，输出与计算功能的模型。输入可以类比为神经元的树突，而输出可以类比为神经元的轴突，计算则可以类比为细胞核 [15]。

图 (1.1) 是一个典型的神经元模型：包含有 3 个输入 $[x_1, x_2, x_3]$，1 个输出 $[y]$，以及 2 个计算功能（求和 $\sum$ 以及非线性函数 sgn 见公式 (1.1)）。注意中间的连线。这些线称为“连接”。每个上有一个“权值”。其中的偏置 b 也可以看成是一个输入神经元值为 1 的一个“连接”上的“权值”。

$$\text{sgn}(x) = \begin{cases} 0, & \text{if } z < 0 \\ 1, & \text{otherwise.} \end{cases} \tag{1.1}$$

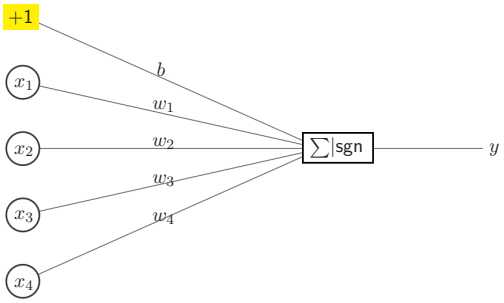

图 1.1: 神经元（MP）结构

将神经元结构图中的所有变量用符号表示，并且写出输出的计算公式的话，就是公式 (1.2)。

$$y = \text{sgn}(w_1 x_1 + w_2 x_2 + w_3 x_3 + b) \tag{1.2}$$

一个神经网络的训练算法就是让权重的值调整到最佳，以使得整个网络的预测效果最好。当用“神经元”组成网络以后，描述网络中的某个“神经元”时，我们更多地会用“单元”（unit）来指代。同时由于神经网络的表现形式是一个有向图，有时也会用“节点”（node）来表达同样的意思。

神经元模型的使用可以这样理解：存在一个数据，称之为样本。样本有五个属性，其中四个属性已知，一个属性未知。通过四个已知属性预测未知属性。具体办法就是使用神经元的公式进行计算。四个已知属性的值是 x_1, x_2, x_3, x_4，未知属性的值是 y。y 通过公式计算出来。这里，已知的属性称之为特征，未知的属性称之为目标。假设特征与目标之间确实是线性关系，并且得到表示这个关系的权值 w_1, w_2, w_3, w_4。那么，便可通过神经元模型预测新样本的目标。

起初由于计算机计算能力以及权值都是预先设置的等原因，神经元是不能学习的，直到接近 10 年后，第一个真正意义的神经网络才诞生。

1.2 单层神经网络

1958 年，计算科学家 Rosenblatt 提出了由两层神经元组成的神经网络。命名为“感知器”（Perceptron）感知器是当时首个可以学习的人工神经网络 [22]。Rosenblatt 现场演示了其学习识别简单图像的过程，在当时的社会引起了轰动。人们认为已经发现了智能的奥秘，许多学者和科研机构纷纷投入到神经网络的研究中。美国军方大力资助了神经网络的研究，并认为神经网络比“原子弹工程”更重要。这段时间直到 1969 年才结束，这个时期可以看作神经网络的第一次高潮。

在“感知器”中，有两个层次。分别是输入层和输出层。输入层里的“输入单元”只负责传输数据，不做计算。输出层里的“输出单元”则需要对前面一层的输入进行计算。把需要计算的层次称之为“计算层”，并把拥有一个计算层的网络称之为“单层神经网络”。假如要预测的目标不再是一个值，而是一个向量。那么可以在输出层再几个“输出单元”。

下图 (1.2) 显示了带有八个输出单元的单层神经网络，其中输出单元 z_1 的计算公式如 (1.3)。

$$z_1 = \text{sgn}(w_{01} + w_{11} x_1 + w_{21} x_2 + w_{31} x_3) \tag{1.3}$$

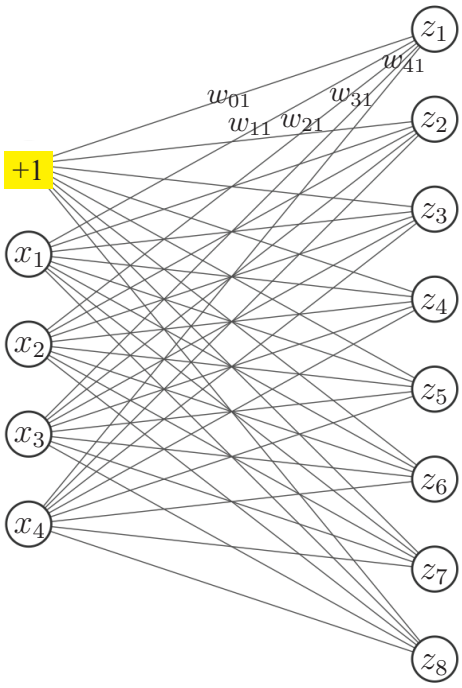

图 1.2: 单层神经网络结构

已知一个神经元的输出可以向多个神经元传递，因此 z_2 的计算公式 (1.4)。

$$z_2 = \text{sgn}(w_{02} + w_{12}x_1 + w_{22}x_2 + w_{32}x_3) \tag{1.4}$$

观察输出的计算公式，会发现这两个公式就是线性代数方程组。因此可以用矩阵乘法来表达这两个公式。例如，输入的变量是 $[1, x_1, x_2, x_3, x_4]$（代表由 $1, x_1, x_2, x_3, x_4$ 组成的行向量），用向量 $\boldsymbol{x^T}$ 来表示。方程的左边是 $[z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8]$，用向量 $\boldsymbol{z}$ 来表示。系数则是矩阵 $\boldsymbol{W}$（5 行 8 列的矩阵，排列形式与公式中的一样）。于是，输出公式可以改写成公式 (1.5)：

$$\boldsymbol{z^T} = \text{sgn}(\boldsymbol{x^T W}) \tag{1.5}$$

与神经元模型不同，感知器中的权值是通过训练得到的。感知器类似一个逻辑回归模型，可以做线性分类任务。可以用决策分界来形象的表达分类的效果。决策分界就是在二维的数据平面中划出一条直线，当数据的维度是 3 维的时候，就是划出一个平面，当数据的维度是 n 维时，就是划出一个 n-1 维的超平面。下图 (1.3) 显示了在二维平面中划出决策分界的效果，也就是感知器的分类效果。

感知器只能做简单的线性分类任务。但是当时的人们热情太过于高涨，并没有人清醒的认识到这点。于是，当人工智能领域的巨擘 Minsky 指出这点时，事态就发生了变化。Minsky 在 1969 年出版了一本叫《Perceptron》的书，里面用详细的数学证明了感知器的弱点，尤其是感知器对 XOR（异或）这样的简单分类任务都无法解决 [16]。Minsky 认为，如果将计算层增加到两层，计算量则过大，而且没有有效的学习算法。所以，他认为研究更深层的网络是没有价值的。由于 Minsky 的巨大影响力以及书中呈现的悲观态度，让很多学者和实验室纷纷放弃了神经网络的研究。神经网络的研究陷入了冰河期。这个时期又被称为“AI winter”。接近 10 年以后，对于两层神经网络的研究才带来神经网络的复苏。

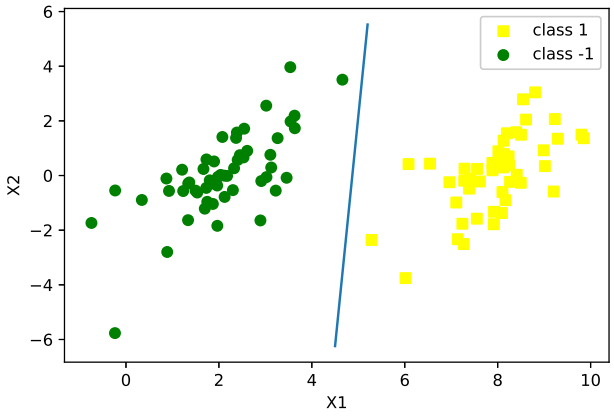

图 1.3: 单层神经网络（决策分界）

1.3 多层神经网络

Minsky 说过单层神经网络无法解决异或问题。但是当增加一个计算层以后，两层神经网络不仅可以解决异或问题，而且具有非常好的非线性分类效果。不过两层神经网络的计算是一个问题，没有一个较好的解法。

1986 年，Rumelhar 和 Hinton 等人提出了反向传播（Backpropagation，BP）算法，解决了两层神经网络所需要的复杂计算量问题，从而带动了业界使用两层神经网络研究的热潮 [6]。

在两层神经网络中，不再使用 sgn 函数，而是使用平滑函数 sigmoid。把此类函数也称作激活函数（active function）。事实上，神经网络的本质就是通过参数与激活函数来拟合特征与目标之间的真实函数关系。

与单层神经网络不同。理论证明，两层神经网络可以无限逼近任意连续函数。

机器学习模型训练的目的，就是使得参数尽可能的与真实的模型逼近。具体做法是这样的。首先给所有参数赋上随机值。我们使用这些随机生成的参数值，来预测训练数据中的样本。样本的预测目标为 y_p，真实目标为 y。那么，定义一个值 $loss$，计算公式如式 (1.6)。

$$loss = (y_p - y)^2 \tag{1.6}$$

这个值称之为损失（loss），我们的目标就是使对所有训练数据的损失和尽可能的小。下面的问题就是求：如何优化参数，能够让损失函数的值最小。

这个问题就被转化为一个优化问题。一个常用方法就是高等数学中的求导，但是这里的问题由于参数不止一个，求导后计算导数等于 0 的运算量很大，所以一般来说解决这个优化问题使用的是梯度下降算法。梯度下降算法每次计算参数在当前的梯度，然后让参数向着梯度的反方向前进一段距离，不断重复，直到梯度接近零时截止。一般这个时候，所有的参数恰好达到使损失函数达到一个最低值的状态。

在神经网络模型中，由于结构复杂，每次计算梯度的代价很大。因此还需要使用反向传播算法。反向传播算法是利用了神经网络的结构进行的计算。不一次计算所有参数的梯度，而是从后往前。首先计算输出层的梯度，然后是第二个参数矩阵的梯度，接着是中间层的梯度，再然后是第一个参数矩阵的梯度，最

后是输入层的梯度。计算结束以后，所要的两个参数矩阵的梯度就都有了。

两层神经网络在多个地方的应用说明了其效用与价值。10 年前困扰神经网络界的异或问题被轻松解决。神经网络在这个时候，已经可以发力于语音识别，图像识别，自动驾驶等多个领域。

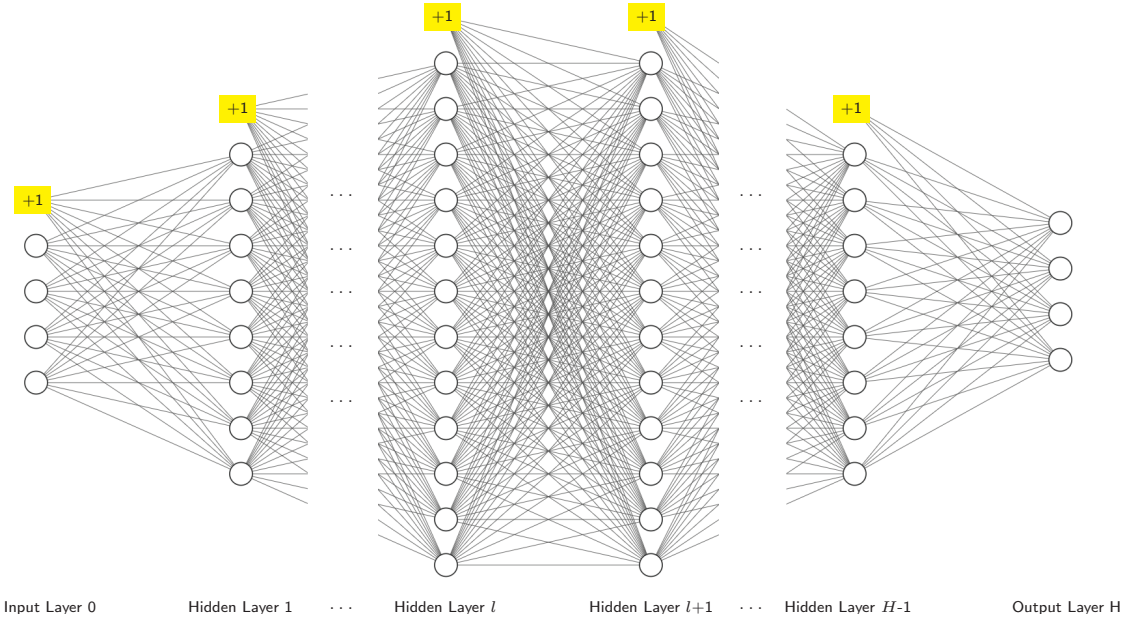

图 1.4: 全连接神经网络结构

在被人摒弃的 10 年中，有几个学者仍然在坚持研究。这其中的棋手就是加拿大多伦多大学的 Geoffery Hinton 教授。2006 年，Hinton 在《Science》和相关期刊上发表了论文，首次提出了“深度信念网络”的概念 [11]。与传统的训练方式不同，“深度信念网络”有一个“预训练”（pre-training）的过程，这可以方便的让神经网络中的权值找到一个接近最优解的值，之后再使用“微调”(fine-tuning) 技术来对整个网络进行优化训练。这两个技术的运用大幅度减少了训练多层神经网络的时间。他给多层神经网络相关的学习方法赋予了一个新名词–“深度学习”。在这之后，关于深度神经网络的研究与应用不断涌现。

在两层神经网络的输出层后面，继续添加层次。原来的输出层变成中间层，新加的层次成为新的输出层。所以可以得到图 (1.4)。依照这样的方式不断添加，我们可以得到更多层的多层神经网络。公式推导跟两层神经网络类似，使用矩阵运算仅仅是加一个公式。

与两层层神经网络不同。多层神经网络中的层数增加了很多。增加更多的层次有更深入的表示特征，以及更强的函数模拟能力。更深入的表示特征可以这样理解，随着网络的层数增加，每一层对于前一层次的抽象表示更深入。在神经网络中，每一层神经元学习到的是前一层神经元值的更抽象的表示。例如第一个隐藏层学习到的是“边缘”的特征，第二个隐藏层学习到的是由“边缘”组成的“形状”的特征，第三个隐藏层学习到的是由“形状”组成的“图案”的特征，最后的隐藏层学习到的是由“图案”组成的“目标”的特征。通过抽取更抽象的特征来对事物进行区分，从而获得更好的区分与分类能力。

第二章

神经网络理论基础

2.1 神经网络激活函数

激活函数是神经网络的一个重要组成部分。如果不用激活函数，在这种情况下，网络的每一层的输入都是上一层的线性输出，因此，无论该神经网络有多少层，最终的输出都是输入的线性组合，与没有隐藏层的效果相当，这种情况就是最原始的感知机。因此，需要引入非线性函数作为激活函数，这样深层神经网络才有意义，输出不再是输入的线性组合，就可以逼近任意函数 [9]。本节给出五种比较常用的激活函数及其导函数。

2.1.1 sigmoid [5]

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{2.1}$$

$$\sigma'(z) = \sigma(z)(1-\sigma(z)) \tag{2.2}$$

2.1.2 tanh

$$\sigma(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}} = 1 - \frac{2e^{-z}}{e^{z}+e^{-z}} \tag{2.3}$$

$$\sigma'(z) = 1-\sigma(z)^2 \tag{2.4}$$

2.1.3 relu [14]

$$\sigma(z) = \begin{cases} 0, & \text{if } z < 0 \\ z, & \text{otherwise.} \end{cases} \tag{2.5}$$

$$\sigma'(z) = \begin{cases} 0, & \text{if } z < 0 \\ 1, & \text{otherwise.} \end{cases} \tag{2.6}$$

2.1.4 leaky relu

$$\sigma(z) = \begin{cases} z, & \text{if } z > 0 \\ az, & \text{otherwise.} \end{cases} \tag{2.7}$$

其中 a 为斜率，又称倾斜因子。

$$\sigma'(z) = \begin{cases} 1, & \text{if } z > 0 \\ a, & \text{otherwise.} \end{cases} \tag{2.8}$$

2.1.5 softmax [18]

第 i 个类目的概率，$p_i = a_i^{(N+1)} = \sigma(s_i^{(N+1)}) = \dfrac{e^{s_i^{(N+1)}}}{\sum_{j=1}^{K} e^{s_j^{(N+1)}}}$ (2.9)

$$\delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

$$\frac{\partial p_i}{\partial s_j} = p_i(\delta_{ij} - p_j) \tag{2.10}$$

证明: 公式 (2.10)

对 softmax 函数求导：

$$\frac{\partial p_i}{\partial s_j} = \frac{\partial \frac{e^{s_i}}{\sum_{k=1}^{N} e^{s_k}}}{\partial s_j}$$

$$f(x) = \frac{g(x)}{h(x)}, \qquad f'(x) = \frac{g'(x)h(x) - h'(x)g(x)}{h^2(x)}$$

$$g(x) = e^{s_i}, \qquad h(x) = \sum_{k=1}^{N} e^{s_k}$$

当 $i = j$ ：

$$\frac{\partial \frac{e^{s_i}}{\sum_{k=1}^{N} e^{s_k}}}{\partial s_j} = \frac{e^{s_i} \sum_{k=1}^{N} e^{s_k} - e^{s_j} e^{s_i}}{\left(\sum_{k=1}^{N} e^{s_k}\right)^2} = \frac{e^{s_i}\left(\sum_{k=1}^{N} e^{s_k} - e^{s_j}\right)}{\left(\sum_{k=1}^{N} e^{s_k}\right)^2} = p_i(1 - p_j)$$

当 $i \neq j$ ：

$$\frac{\partial \frac{e^{s_i}}{\sum_{k=1}^{N}}}{\partial s_j} = \frac{0 - e^{s_j} e^{s_i}}{\left(\sum_{k=1}^{N} e^{s_k}\right)^2} = -\frac{e^{s_i}}{\sum_{k=1}^{N} e^{s_k}} \frac{e^{s^j}}{\sum_{k=1}^{N} e^{s_k}} = -p_i p_j$$

因此：

$$\delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

$$\frac{\partial p_i}{\partial s_j} = p_i(\delta_{ij} - p_j)$$

□

2.2 神经网络损失函数

损失函数有助于优化神经网络的参数。训练神经网络的目标是通过优化神经网络的参数来最大程度地减少神经网络的损失。通过神经网络将目标值与预测值进行匹配，再经过损失函数就可以计算出损失。然后，我们使用梯度下降法来优化网络权重，以使损失最小化 [9]。此节给出两种常用的损失函数及其导函数。

2.2.1 Mean Square Error

$$Loss(\boldsymbol{z}, \boldsymbol{y}) = \frac{1}{2} \sum_{i=1}^{K} (z_i - y_i)^2 \tag{2.11}$$

$$\frac{\partial Loss}{\partial \boldsymbol{z}} = (\boldsymbol{z} - \boldsymbol{y}) \tag{2.12}$$

2.2.2 Cross Entropy Loss [18]

$$Loss(z,y) = -\sum_{i=1}^{K} y_i \log z_i = -y_k \log z_k \tag{2.13}$$

$$\frac{\partial Loss}{\partial \boldsymbol{z}} = -\frac{\boldsymbol{y}}{\boldsymbol{z}} \tag{2.14}$$

一般如果多分类问题输出层使用 CrossEntropy 损失，并用 Softmax 激活。可以极大的削弱梯度消失现象。

$$\frac{\partial Loss(a^{(N+1)},y)}{\partial s_j^{(N+1)}} = a_j^{(N+1)} - y_j \tag{2.15}$$

这里给出证明。

证明: 使用 CrossEntropy 损失，并用 Softmax 激活。可以极大的削弱梯度消失现象。

$$\frac{\partial Loss(a^{(N+1)},y)}{\partial s_j^{(N+1)}} = a_j^{(N+1)} - y_j$$

$$\begin{aligned}
\frac{\partial Loss(a^{(N+1)},y)}{\partial s_j^{(N+1)}} &= \sum_{i=1}^{K} \frac{\partial Loss(a^{(N+1)},y)}{\partial a_i} \frac{\partial a_i}{\partial s_j} \\
&= -\sum_{i=1}^{K} \frac{y_i}{a_i} \frac{\partial a_i}{\partial s_j} \\
&= \big(-\frac{y_i}{a_i}\frac{\partial a_i}{\partial s_j}\big)_{i=j} - \sum_{i=1,i\neq j}^{K} \frac{y_i}{a_i}\frac{\partial a_i}{\partial s_j} \\
&= -\frac{y_j}{a_j} a_j(1-a_j) - \sum_{i=1,i\neq j}^{K} \frac{y_i}{a_i} \cdot -a_i a_j \\
&= -y_j + y_j a_j + \sum_{i=1,i\neq j}^{K} y_i a_j \\
&= -y_j + a_j \sum_{i=1}^{K} y_i \\
&= a_j^{(N+1)} - y_j
\end{aligned}$$

□

2.3 神经网络初始化方法

2.3.1 全零初始化

将神经网络所有参数全部初始化为零。

在线性回归，logistics 回归的时候，基本上都是把参数初始化为 0，模型也能够很好的工作。但在神经网络中，把权值初始化为零是不可以的。如果这样那么每一层的神经元学到的东西都是一样的，而且在反向传播的时候，每一层内的神经元也是相同的，因为梯度相同。造成的结果就是神经网络学不到有效的知识。

2.3.2 随机初始化

将神经网络所有参数初始化为某种数据分布 [4]。

- 高斯分布初始化

  参数服从固定均值和方差的高斯分布进行随机初始化，也可以考虑输入和输出神经元的数量，分别记作叫 n_{in} 和 n_{out}:

$$\boldsymbol{W} \sim N(0,\ \sqrt{\frac{2}{n_{in}+n_{out}}}) \tag{2.16}$$

- 均匀分布初始化

  参数服从 $U[-\alpha,\ \alpha]$ 的均匀分布进行随机初始化，也可以考虑输入神经元的数量 n_{in}:

$$\boldsymbol{W} \sim U(-\sqrt{\frac{1}{n_{in}}},\ \sqrt{\frac{1}{n_{in}}}) \tag{2.17}$$

2.3.3 Xavier 初始化

参数服从 $U[-\alpha,\ \alpha]$ 的均匀分布进行随机初始化，目的是让输入和输出神经元的方差尽量一致 [8]。其中，α 通过推导证明得出:

$$\boldsymbol{W} \sim U(-\sqrt{\frac{2}{n_{in}+n_{out}}},\ \sqrt{\frac{2}{n_{in}+n_{out}}}) \tag{2.18}$$

证明:

$$\alpha = \sqrt{\frac{2}{n_{in}+n_{out}}}$$

n 个成分构成的输入向量 $\boldsymbol{x}$，经过一个随机权值矩阵为 $\boldsymbol{W}$ 的线性神经元，得到输出

$$\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x} = \boldsymbol{w_1}x_1 + \boldsymbol{w_2}x_2 + \ldots + \boldsymbol{w_n}x_n$$

已知 x_i 是独立同分布的，且均值方差已知，此时求输出 $\boldsymbol{y}$ 的方差。推导如下，由独立变量积的方差计算公式可知，

$$\mathrm{Var}\left(W_iX_i\right) = \left[E\left(X_i\right)\right]^2 \mathrm{Var}\left(W_i\right) + \left[E\left(W_i\right)\right]^2 \mathrm{Var}\left(X_i\right) + \mathrm{Var}\left(X_i\right)\mathrm{Var}\left(W_i\right)$$

又已对输入向量取均值，输入和权值矩阵均值均为 0，则：

$$\mathrm{Var}\left(W_iX_i\right) = \mathrm{Var}\left(X_i\right)\mathrm{Var}\left(W_i\right)$$

所以进一步有：

$$\mathrm{Var}(\boldsymbol{y}) = \mathrm{Var}\left(\sum_i w_i x_i\right) = \sum_i \mathrm{Var}\left(w_i x_i\right) = \sum_i \mathrm{Var}\left(x_i\right)\mathrm{Var}\left(w_i\right) = n\,\mathrm{Var}\left(x_i\right)\mathrm{Var}\left(w_i\right)$$

因此为使得，输出 y 与输入 x 具有相同的均值和方差，权值矩阵的方差则要求：

$$\mathrm{Var}\left(w_i\right) = \frac{1}{n} = \frac{1}{n_{\text{in}}}$$

这里的 n 指的是输入样本的维数。

在此基础上，还需考虑反向传播时的情形，反向传播是正向传播的逆过程，此时的输入是前向传播的输出，则有：

$$\mathrm{Var}\left(w_i\right) = \frac{1}{n} = \frac{1}{n_{\text{out}}}$$

综合以下两点要求，则可得到满足以上两点要求的权值矩阵的方差为：

$$\mathrm{Var}\left(W_i\right) = \frac{2}{n_{\text{in}} + n_{\text{out}}}$$

因此，

$$\alpha = \sqrt{\frac{2}{n_{in} + n_{out}}}$$

□

2.3.4 凯明初始化

思想：正向传播时，状态值的方差保持不变；反向传播时，关于激活值的梯度的方差保持不变 [10]。

- 适用于 ReLU 的初始化方法：

$$\boldsymbol{W} \sim N(0, \sqrt{\frac{2}{n_{in}}}) \tag{2.19}$$

- 适用于 Leaky ReLU 的初始化方法：

$$\boldsymbol{W} \sim N(0, \sqrt{\frac{2}{(1+\alpha^2)n_{in}}}) \tag{2.20}$$

第三章

神经网络的正反向传播

神经网络的正反向传播是神经网络训练的重要组成部分，以图 (3.1) 为例进行数学建模，最终扩展得到模型的矩阵运算形式。

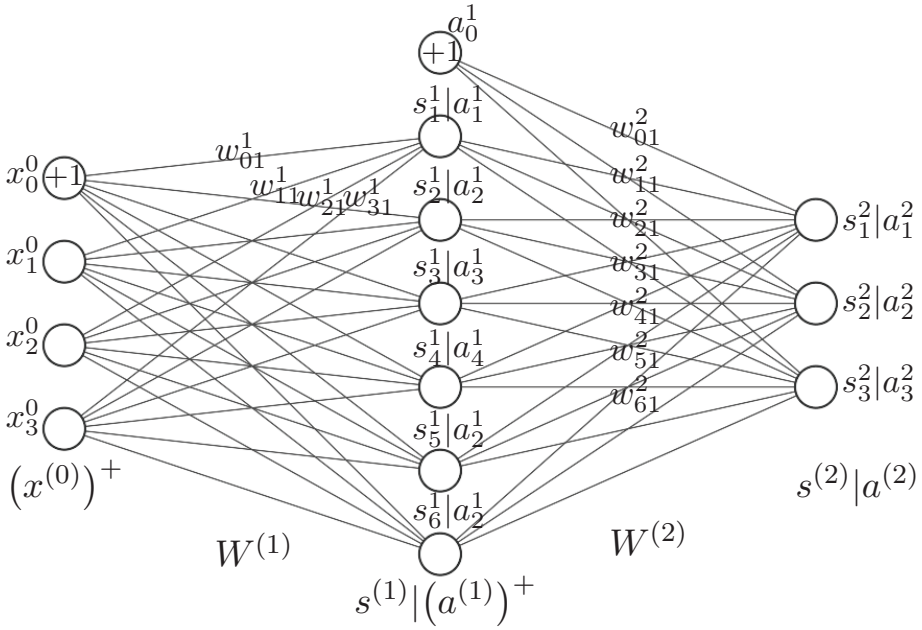

图 3.1: 推导示例神经网络

推导过程中用到大量的变量在此进行解释说明。推导过程中用到自定义的矩阵的一元运算，在此补充。

定义 3.1 $\boldsymbol{x}^+$ 运算符：为向量 $\boldsymbol{x}$ 添加 $Bias$ 列，具体在第 0 列添加全 1。

例 3.1
$$\boldsymbol{x} = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \qquad \boldsymbol{x}^+ = \begin{pmatrix} 1 & a & b & c \\ 1 & d & e & f \\ 1 & g & h & i \end{pmatrix}$$

定义 3.2 $\boldsymbol{x}^-$ 运算符：为向量 $\boldsymbol{x}$ 删去 $Bias$ 列，具体删除第 0 列。

表 3.1: 神经网络参数相关符号

符号	含义	备注
N_t	训练样本维度	$N_t > 0$
N_T	训练样本数量	$N_T > 0$
$N^{(i)}$	第 i 层神经元个数	$i \in \{0, 1, 2, \cdots, H+1\}$ $i = H+1$ 表示输出层 不带 Bias
H	隐藏层数	包含 H 个隐藏层
K	分类类别数	$K > 1$
S_l	损失函数	$S_l \in \{$Mean Square Error，Cross Entropy Loss$\}$

表 3.2: 神经网络推导相关符号

符号	含义	备注
$\boldsymbol{x}$	输入行向量	一个训练样本特征为一个行向量
$\boldsymbol{y}$	标签向量	一个训练样本标签为一个行向量 (ONEHOT)
$\boldsymbol{s}$	求和值行向量	每层对应一个行向量
$\boldsymbol{a}$	激活值行向量	每层对应一个行向量
σ	激活函数	$\sigma \in \{sigmoid, tanh, relu, leakyrelu, softmax\}$
$\boldsymbol{\cdot}_j^{(i)}$	i 代表网络层次 j 代表分量	$0 \leq i \leq H+1$

符号	含义	备注
$\boldsymbol{X}$	输入训练样本	维度 $N_T \times N_t$ 相当激活值矩阵 $A^{(0)}$
$\boldsymbol{X}^+$	输入训练样本加偏置列	维度 $N_T \times (N_t+1)$ 相当激活值矩阵 $\left(A^{(0)}\right)^+$
$\boldsymbol{S}^{(i)}$	第 i 层求和值矩阵	维度 $N_T \times N^{(i)}$ $i = 1, \cdots, H, H+1$
$\boldsymbol{A}^{(i)}$	第 i 层激活值矩阵	维度 $N_T \times N^{(i)}$ $i = 0, 1, \cdots, H, H+1$
$\left(\boldsymbol{A}^{(i)}\right)^+$	第 i 层激活值矩阵加偏置列	维度 $N_T \times (N^{(i)}+1)$ $i = 0, 1, \cdots, H$
$\boldsymbol{W}^{(i)}$	第 i 层权值矩阵	维度 $N^{(i-1)} \times N^{(i)}, i = 1, \cdots, H, H+1.$
$\boldsymbol{W}^{(H+1)}$	输出层权值矩阵	维度 $N^{(N)} \times K$ 标签有 K 个类目
$\boldsymbol{W}_b^{(i)}$	第 i 层权值偏置矩阵	维度 $(N^{(i-1)}+1) \times N^{(i)}, i = 1, \cdots, H, H+1.$
$\boldsymbol{W}_b^{(H+1)}$	输出层权值偏置矩阵	维度 $(N^{(N)}+1) \times K$ 标签有 K 个类目
$\boldsymbol{S}^{(H+1)}$	输出层求和值矩阵	维度 $N_T \times K$
$\boldsymbol{A}^{(H+1)}$	输出层激活值矩阵	维度 $N_T \times K$
$\boldsymbol{Y}$	训练数据标签	维度 $N_T \times K$ (ONE-HOT)
$\boldsymbol{\Delta}^{(i)}$	反向传播中间变量矩阵	维度 $N_T \times N^{(i)}$ $i = 1, \cdots, H, H+1$
$\boldsymbol{\Delta}^{(H+1)}$	输出层反向传播中间变量矩阵	维度 $N_T \times K$

例 3.2 $\boldsymbol{x}=\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \quad \boldsymbol{x}^{-}=\begin{pmatrix} b & c \\ e & f \\ h & i \end{pmatrix}$

3.1 神经网络正向传播

- 示例推导：

$$\begin{aligned} &\left(\boldsymbol{x}^{(0)}\right)^{+}\boldsymbol{W}^{(1)}=\boldsymbol{s}^{(1)}, \quad \sigma(\boldsymbol{s}^{(1)})=\boldsymbol{a}^{(1)} \\ &\left(\boldsymbol{a}^{(1)}\right)^{+}\boldsymbol{W}^{(2)}=\boldsymbol{s}^{(2)}, \quad \sigma(\boldsymbol{s}^{(2)})=\boldsymbol{a}^{(2)} \\ &Loss \quad L=\frac{1}{2}\sum_{k=1}^{3}(\boldsymbol{a}_k^{(2)}-\boldsymbol{y}_k)^2 \end{aligned}$$

- 矩阵扩展推导形式：

$$\begin{aligned} \left(\boldsymbol{A}^{(0)}\right)^{+}\boldsymbol{W}_b^{(1)}=\boldsymbol{S}^{(1)} \xrightarrow{\sigma+b} \left(\boldsymbol{A}^{(1)}\right)^{+} &\Rightarrow \cdots \Rightarrow \cdots \\ &\Rightarrow \left(\boldsymbol{A}^{(H-1)}\right)^{+}\boldsymbol{W}_b^{(H)}=\boldsymbol{S}^{(H)} \xrightarrow{\sigma+b} \left(\boldsymbol{A}^{(H)}\right)^{+} \\ &\Rightarrow \left(\boldsymbol{A}^{(H)}\right)^{+}\boldsymbol{W}_b^{(H+1)}=\boldsymbol{S}^{(H+1)} \xrightarrow{\sigma} \boldsymbol{A}^{(H+1)} \\ &\Rightarrow \ \mathcal{L}(\boldsymbol{A}^{(H+1)},\boldsymbol{Y}) \end{aligned}$$

3.2 神经网络反向传播 [6]

- 示例推导：

(1) 输出层权值梯度：

$$\begin{aligned} Loss, \quad L&=\frac{1}{2}\sum_{i=1}^{K}(\boldsymbol{a}_i^{(2)}-\boldsymbol{y}_i)^2 \\ \frac{\partial L}{\partial \boldsymbol{W}_{ik}^{(2)}}&=\frac{\partial}{\partial \boldsymbol{W}_{ik}^{(2)}}\frac{1}{2}\sum_{i=1}^{K}(\boldsymbol{a}_i^{(2)}-\boldsymbol{y}_i)^2 \\ &=\left(\sigma(\boldsymbol{S}_k^{(2)})-\boldsymbol{y}_k\right)\sigma(\boldsymbol{S}_k^{(2)})\left(1-\sigma(\boldsymbol{S}_k^{(2)})\right)\frac{\partial \boldsymbol{S}_k^{(2)}}{\partial \boldsymbol{W}_{ik}^{(2)}} \\ &=\left(\sigma(\boldsymbol{S}_k^{(2)})-\boldsymbol{y}_k\right)\sigma(\boldsymbol{S}_k^{(2)})\left(1-\sigma(\boldsymbol{S}_k^{(2)})\right)\left(\boldsymbol{a}_i^{(1)}\right)^{+} \\ 令, \quad \boldsymbol{\delta}_k^{(2)}&=\left(\sigma(\boldsymbol{S}_k^{(2)})-\boldsymbol{y}_k\right)\sigma(\boldsymbol{S}_k^{(2)})\left(1-\sigma(\boldsymbol{S}_k^{(2)})\right) \\ &=\left(\boldsymbol{a}_k^{(2)}-\boldsymbol{y}_k\right)\boldsymbol{a}_k^{(2)}\left(1-\boldsymbol{a}_k^{(2)}\right) \\ 则, \quad \frac{\partial L}{\partial \boldsymbol{W}_{ik}^{(2)}}&=\boldsymbol{\delta}_k^{(2)}\left(\boldsymbol{a}_i^{(1)}\right)^{+} \end{aligned}$$

(2) 隐藏层权值梯度：

$$
\begin{aligned}
\frac{\partial L}{\partial \boldsymbol{W}_{ij}^{(1)}} &= \frac{\partial}{\partial \boldsymbol{W}_{ij}^{(1)}} \frac{1}{2} \sum_{k \in K} (\boldsymbol{a}_k^{(2)} - \boldsymbol{y}_k)^2 \\
&= \sum_{k \in K} (\boldsymbol{a}_k^{(2)} - \boldsymbol{y}_k) \frac{\partial \sigma(\boldsymbol{s}_k^{(2)})}{\partial \boldsymbol{W}_{ij}^{(1)}} \\
&= \sum_{k \in K} (\boldsymbol{a}_k^{(2)} - \boldsymbol{y}_k) \sigma(\boldsymbol{s}_k^{(2)}) \big(1 - \sigma(\boldsymbol{s}_k^{(2)}) \frac{\partial \boldsymbol{s}_k^{(2)}}{\partial \boldsymbol{W}_{ij}^{(1)}} \\
&= \sum_{k \in K} (\boldsymbol{a}_k^{(2)} - \boldsymbol{y}_k) \sigma(\boldsymbol{s}_k^{(2)}) \big(1 - \sigma(\boldsymbol{s}_k^{(2)}) \frac{\partial \boldsymbol{s}_k^{(2)}}{\partial \boldsymbol{a}_j^{(1)}} \frac{\partial \boldsymbol{a}_j^{(1)}}{\partial \boldsymbol{W}_{ij}^{(1)}} \\
&= \sum_{k \in K} (\boldsymbol{a}_k^{(2)} - \boldsymbol{y}_k) \sigma(\boldsymbol{s}_k^{(2)}) \big(1 - \sigma(\boldsymbol{s}_k^{(2)}) \boldsymbol{W}_{jk}^{(2)} \frac{\partial \boldsymbol{a}_j^{(1)}}{\partial \boldsymbol{W}_{ij}^{(1)}} \\
&= \frac{\partial \boldsymbol{a}_j^{(1)}}{\partial \boldsymbol{W}_{ij}^{(1)}} \sum_{k \in K} (\boldsymbol{a}_k^{(2)} - \boldsymbol{y}_k) \sigma(\boldsymbol{s}_k^{(2)}) \big(1 - \sigma(\boldsymbol{s}_k^{(2)}) \boldsymbol{W}_{jk}^{(2)} \\
&= \boldsymbol{a}_j^{(1)} \big(1 - \boldsymbol{a}_j^{(1)}\big) \frac{\partial \boldsymbol{s}_j^{(1)}}{\boldsymbol{W}_{ij}^{(1)}} \sum_{k \in K} \boldsymbol{\delta}_k^{(2)} \boldsymbol{W}_{jk}^{(2)} \\
&= \boldsymbol{a}_j^{(1)} \big(1 - \boldsymbol{a}_j^{(1)}\big) \big(x_i^{(0)}\big)^+ \sum_{k \in K} \boldsymbol{\delta}_k^{(2)} \boldsymbol{W}_{jk}^{(2)}
\end{aligned}
$$

$$
\text{令，} \quad \boldsymbol{\delta}_j^{(1)} = \boldsymbol{a}_j^{(1)} (1 - \boldsymbol{a}_j^{(1)}) \sum_{k \in K} \boldsymbol{\delta}_k^{(2)} \boldsymbol{W}_{jk}^{(2)}
$$

$$
\text{则，} \frac{\partial L}{\partial \boldsymbol{W}_{ij}^{(1)}} = \boldsymbol{\delta}_j^{(1)} \big(\boldsymbol{x}_i^{(0)}\big)^+
$$

- 矩阵扩展推导形式：

1. 输出层权值求导：

$$
\nabla_{\boldsymbol{W}_b^{(H+1)}} \mathcal{L}(\boldsymbol{A}^{(H+1)}, \boldsymbol{Y}) = \frac{1}{H_T} (\boldsymbol{A}^{(H)^+})^T \boldsymbol{\Delta}^{(H+1)} \quad \text{（矩阵化）}
$$

$$
\boldsymbol{\Delta}^{(H+1)} = \frac{\partial \mathcal{L}(\boldsymbol{A}^{(H+1)}, \boldsymbol{Y})}{\partial \boldsymbol{A}^{(H+1)}} \circ \sigma'(\boldsymbol{S}^{(H+1)})
$$

当输出层使用 Softmax+CrossEntropy 时:

$$
\boldsymbol{\Delta}^{(H+1)} = \boldsymbol{A}^{(H+1)} - \boldsymbol{Y}
$$

2. 隐藏层权值求导：

$$
\nabla_{\boldsymbol{W}_b^{(H)}} \mathcal{L}(\boldsymbol{A}^{(H+1)}, \boldsymbol{Y}) = \frac{1}{H_T} \big(\boldsymbol{A}^{(H-1)^+}\big)^T \boldsymbol{\Delta}^{(H)}
$$

$$
\text{令：} \boldsymbol{\Delta}^{(H)} = \big(\boldsymbol{\Delta}^{(H+1)} (\boldsymbol{W}^{(H+1)})^T\big) \circ \sigma'^{(H)}(\boldsymbol{S}^{(H)})
$$

扩展： 对于第 i 个隐藏层的权值矩阵的梯度为 ($i = 1, 2, \cdots, H.$)：

$$
\nabla_{\boldsymbol{W}_b^{(i)}} \mathcal{L}(\boldsymbol{A}^{(H+1)}, \boldsymbol{Y}) = \frac{1}{H_T} \big(\boldsymbol{A}^{(i-1)^+}\big)^T \boldsymbol{\Delta}^{(i)}
$$

$$
\text{令：} \boldsymbol{\Delta}^{(i)} = \boldsymbol{\Delta}^{(i+1)} (\boldsymbol{W}^{(i+1)})^T \circ \sigma'^{(i)}(\boldsymbol{S}^{(i)})
$$

3.3 神经网络张量流

探究神经网络运算过程中的张量变化能更清楚的了解神经网络的作用规律，表 (3.3) 展示了神经网络的张量流。

表 3.3: 神经网络张量流

前向传播	维度	反向传播	维度
$\boldsymbol{X}$	$N_T \times (N_t + 1)$	LOSS	Scalar
$\boldsymbol{S}^{(1)}$	$N_T \times N^{(1)}$	$\boldsymbol{\Delta}^{(H+1)}$	$N_T \times K$
$(\boldsymbol{A}^{(1)})^+$	$N_T \times (N^{(1)} + 1)$	$\nabla_{\boldsymbol{W}^{(H+1)}}\mathcal{L}$	$(N^{(H)} + 1) \times K$
$\boldsymbol{S}^{(2)}$	$N_T \times N^{(2)}$	$\boldsymbol{\Delta}^{(H)}$	$N_T \times (N^{(H)} + 1)$
$(\boldsymbol{A}^{(2)})^+$	$N_T \times (N^{(2)} + 1)$	$\nabla_{\boldsymbol{W}^{(N)}}\mathcal{L}$	$(N^{(H-1)} + 1) \times N^{(H)}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\boldsymbol{S}^{(H)}$	$N_T \times N^{(H)}$	$\boldsymbol{\Delta}^{(i)}$	$N_T \times (N^{(i)} + 1)$
$(\boldsymbol{A}^{(H)})^+$	$N_T \times (N^{(H)} + 1)$	$\nabla_{\boldsymbol{W}^{(i)}}\mathcal{L}$	$(N^{(i-1)} + 1) \times N^{(i)}$
$\boldsymbol{S}^{(H+1)}$	$N_T \times K$	$\boldsymbol{\Delta}^{(1)}$	$N_T \times (N^{(1)} + 1)$
$\boldsymbol{A}^{(H+1)}$	$N_T \times K$	$\nabla_{\boldsymbol{W}^{(1)}}\mathcal{L}$	$(N_t + 1) \times N^{(1)}$
计算求得损失		更新权值	

第四章

神经网络优化方法

在当神经网络一次正反向传播后，得到权值梯度，接下来需要进行优化去更新梯度。优化是指寻找可以让函数最小化或最大化的参数的过程。其中梯度下降是优化算法的核心，Adam 优化算法为其变种。

4.1 梯度下降优化方法

4.1.1 批量梯度下降优化方法

批量梯度下降法 (Batch Gradient Descent，BGD) 是最原始的形式，它是指在每一次迭代时使用所有样本来进行梯度的更新 [17]。具体步骤是：

1. 对目标函数求关于每个权值的偏导：

$$\frac{\partial J\left(\boldsymbol{\theta}_0, \boldsymbol{\theta}_1\right)}{\partial \boldsymbol{\theta}_j}=\frac{1}{m} \sum_{i=1}^{m}\left(h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right)-\boldsymbol{y}^{(i)}\right) \boldsymbol{x}_j^{(i)}$$

其中 $i=1,2, \cdots, m$ 表示样本数，$j=0,1$ 表示特征数，这里我们使用了偏置项 $x_0^{(i)}=1$ 。

2. 每次迭代对参数进行更新：

$$\boldsymbol{\theta}_j:=\boldsymbol{\theta}_j-\alpha \frac{1}{m} \sum_{i=1}^{m}\left(h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right)-\boldsymbol{y}^{(i)}\right) \boldsymbol{x}_j^{(i)}$$

梯度下降优化方法伪码 [24]

repeat

  for $j=0,1$

    $\boldsymbol{\theta}_j:=\boldsymbol{\theta}_j-\alpha \frac{1}{m} \sum_{i=1}^{m}\left(h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right)-\boldsymbol{y}^{(i)}\right) \boldsymbol{x}_j^{(i)}$

1. 优点：

- 一次迭代是对所有样本进行计算，此时利用矩阵进行操作，实现了并行。
- 由全数据集确定的方向能够更好地代表样本总体，从而更准确地朝向极值所在的方向。当目标函数为凸函数时，BGD 一定能够得到全局最优。

2. 缺点：

- 当样本数目 m 很大时，每迭代一步都需要对所有样本计算，训练过程很慢。

4.1.2 随机梯度下降优化方法

随机梯度下降法 (Stochastic Gradient Descent，SGD) 不同于批量梯度下降，随机梯度下降是每次迭代使用一个样本来对参数进行更新 [1]。使得训练速度加快。具体步骤是：

1. 对一个样本的目标函数求关于每个权值的偏导：

$$\frac{\partial J^{(i)}\left(\boldsymbol{\theta}_0, \boldsymbol{\theta}_1\right)}{\partial \boldsymbol{\theta}_j}=\left(h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right)-\boldsymbol{y}^{(i)}\right) \boldsymbol{x}_j^{(i)}$$

其中 $i=1,2, \cdots, m$ 表示样本数，$j=0,1$ 表示特征数，这里我们使用了偏置项 $x_0^{(i)}=1$ 。

2. 参数更新：

$$\boldsymbol{\theta}_j:=\boldsymbol{\theta}_j-\alpha\left(h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right)-\boldsymbol{y}^{(i)}\right) \boldsymbol{x}_j^{(i)}$$

随机梯度下降优化方法伪码

repeat
  for $i=1, \cdots, m$
    for $j=0,1$
      $\boldsymbol{\theta}_j:=\boldsymbol{\theta}_j-\alpha\left(h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right)-\boldsymbol{y}^{(i)}\right) \boldsymbol{x}_j^{(i)}$

1. 优点：

- 由于不是在全部训练数据上的损失函数，而是在每轮迭代中，随机优化某一条训练数据上的损失函数，这样每一轮参数的更新速度大大加快。

2. 缺点：

- 准确度下降。由于即使在目标函数为强凸函数的情况下，SGD 仍旧无法做到线性收敛。
- 可能会收敛到局部最优，由于单个样本并不能代表全体样本的趋势。
- 不易于并行实现。

4.1.3 小批量梯度下降优化方法

小批量梯度下降优化方法 ((Mini-Batch Gradient Descent，MBGD)) 是对 BGD 以及 SGD 的一个折中办法。其思想是：每次迭代使用 batch_size 个样本来对参数进行更新。

这里我们假设 batch_size=10 ，样本数 m=1000 ，$j=0,1$ 表示特征数。

小批量梯度下降优化方法伪码

repeat

for i $=1,11,21,31,\ldots,991$

for $j=0,1$

$$\boldsymbol{\theta}_j := \boldsymbol{\theta}_j - \alpha \frac{1}{10} \sum_{k=i}^{(i+9)} \left(h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(k)}\right) - \boldsymbol{y}^{(k)}\right) \boldsymbol{x}_j^{(k)}$$

1. 优点：
   - 通过矩阵运算，每次在一个 batch 上优化神经网络参数并不会比单个数据慢太多。
   - 每次使用一个 batch 可以大大减小收敛所需要的迭代次数，同时可以使收敛到的结果更加接近梯度下降的效果。(比如上例中的 30W，设置 batch_size=100 时，需要迭代 3000 次，远小于 SGD 的 30W 次)
   - 可实现并行化。
2. 缺点：
   - batch_size 的不当选择可能会带来一些问题。

利用小批量优化或者随机优化算法相比批量优化算法能快速收敛，减少训练时间。

4.2 Adam 优化方法

Adam 算法使用了动量变量 $\boldsymbol{v}_t$ [20] 和 RMSProp 算法中小批量随机梯度按元素平方的指数加权移动平均变量 $\boldsymbol{s}_t$ [13]，并在时间步 0 将它们中每个元素初始化为 0 [12]。给定超参数 $0<\beta_1<1$（默认设为 0.9），时间步 t 的动量变量 $\boldsymbol{v}_t$ 即小批量随机梯度 $\boldsymbol{g}_t$ 的指数加权移动平均：

$$\boldsymbol{v}_t \leftarrow \beta_1 \boldsymbol{v}_{t-1} + (1-\beta_1)\boldsymbol{g}_t \tag{4.1}$$

和 RMSProp 算法中一样，给定超参数 $0 \leq \beta_2 < 1$（默认设为设为 0.999），将小批量随机梯度按元素平方后的项 $\boldsymbol{g}_t \odot \boldsymbol{g}_t$ 做指数加权移动平均得到 $\boldsymbol{s}_t$ ：

$$\boldsymbol{s}_t \leftarrow \beta_2 \boldsymbol{s}_{t-1} + (1-\beta_2)\boldsymbol{g}_t \odot \boldsymbol{g}_t \tag{4.2}$$

由于我们将 $\boldsymbol{v}_0$ 和 $\boldsymbol{s}_0$ 中的元素都初始化为 0，在时间步 t 我们得到 $\boldsymbol{v}_t = (1-\beta_1)\sum_{i=1}^{t}\beta_1^{t-i}\boldsymbol{g}_i$ 。将过去各时间步小批量随机梯度的权值相加，得到 $(1-\beta_1)\sum_{i=1}^{t}\beta_1^{t-i} = 1-\beta_1^t$ 。需要注意的是，当 t 较小时，过去

各时间步小批量随机梯度权值之和会较小。例如，当 $\beta_1 = 0.9$ 时，$\boldsymbol{v}_1 = 0.1\boldsymbol{g}_1$ 。为了消除这样的影响，对于任意时间步 t ，我们可以将 $\boldsymbol{v}_t$ 再除以 $1-\beta_1^t$ ，从而使过去各时间步小批量随机梯度权值之和为 1。这也叫作偏差修正。在 Adam 算法中，我们对变量 $\boldsymbol{v}_t$ 和 $\boldsymbol{s}_t$ 均作偏差修正：

$$\hat{\boldsymbol{v}}_t \leftarrow \frac{\boldsymbol{v}_t}{1-\beta_1^t} \tag{4.3}$$

$$\hat{\boldsymbol{s}}_t \leftarrow \frac{\boldsymbol{s}_t}{1-\beta_2^t} \tag{4.4}$$

接下来，Adam 算法使用以上偏差修正后的变量 $\hat{\boldsymbol{v}}_t$ 和 $\hat{\boldsymbol{s}}_t$ ，将模型参数中每个元素的学习率通过按元素运算重新调整：

$$\boldsymbol{g}_t^{'} \leftarrow \frac{\eta \hat{\boldsymbol{v}}_t}{\sqrt{\hat{\boldsymbol{s}}_t} + \epsilon} \tag{4.5}$$

其中 η 是学习率，ϵ 是为了维持数值稳定性而添加的常数，如 10^{-8} 。和 AdaGrad 算法、RMSProp 算法以及 AdaDelta 算法一样，目标函数自变量中每个元素都分别拥有自己的学习率。最后，使用 $\boldsymbol{g}_t^{'}$ 迭代自变量：

$$x_t \leftarrow x_{t-1} - \boldsymbol{g}_t^{'} \tag{4.6}$$

Adam 权值更新核心伪码

$$\begin{aligned}
&\boldsymbol{v}_t = \beta_1 \boldsymbol{v}_{t-1} + (1-\beta_1)\boldsymbol{g}_t \\
&\boldsymbol{s}_t = \beta_2 \boldsymbol{s}_{t-1} + (1-\beta_2)\boldsymbol{g}_t \odot \boldsymbol{g}_t \\
&\hat{\boldsymbol{v}}_t = \frac{\boldsymbol{v}_t}{1-\beta_1^t} \\
&\hat{\boldsymbol{s}}_t = \frac{\boldsymbol{s}_t}{1-\beta_2^t} \\
&\boldsymbol{g}_t' = \frac{\eta \hat{\boldsymbol{v}}_t}{\sqrt{\hat{\boldsymbol{s}}_t + \epsilon}} \\
&\boldsymbol{x}_t = \boldsymbol{x}_{t-1} - \boldsymbol{g}_t'
\end{aligned}$$

1. 优点：
   - 直截了当地实现。
   - 高效的计算。
   - 所需内存少。
   - 梯度对角缩放的不变性。
   - 适合解决含大规模数据和参数的优化问题。
   - 适用于非稳态（non-stationary）目标。
   - 适用于解决包含很高噪声或稀疏梯度的问题。
   - 超参数可以很直观地解释，并且基本上只需极少量的调参。
2. 缺点：
   - 特殊状况下可能不收敛。
   - 可能错过全局最优解。

PART II 第二部分

框架文档

第五章

框架结构

全连接神经网络模型主要以层次建模图 (5.1)，在以矩阵数据结构的基础上，为神经网络隐藏层次（求和层以及激活层），输出层以及损失层进行模型建设，之后再对神经网络的初始化方式，前向传播，反向传播，优化算法等进行建模。

1. 神经网络层次模型

   (a) 隐藏层

      i. 求和层

      ii. 激活层

   (b) 输出层

   (c) 损失层

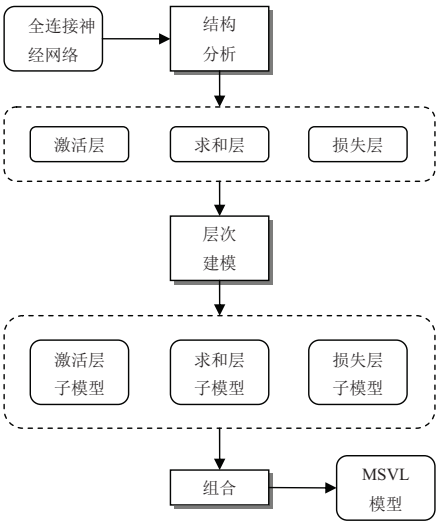

图 5.1: 全连接神经网络建模方案

第六章

编码说明

6.1 对象结构体说明

6.1.1 矩阵结构体

代码 6.1: 矩阵结构体

```
struct Mat {
int row,col and      // 矩阵的行数和列数
float **element      // 二维矩阵元素值
};
```

注：

- 二维矩阵结构体是本框架最基本的数据结构，一切操作都源于此。

6.1.2 神经网络单层结构体

代码 6.2: 神经网络单层结构体

```
struct FCLayer {
Mat ActiMat and                 // 不带有偏置列的激活值矩阵
Mat ActiMatPlus and             // 带有偏置列的激活值矩阵
Mat SumMat and                  // 求和值矩阵
Mat WeightMat and               // 不带有偏置的权值矩阵
Mat WeightBiasMat and           // 带有偏置的权值矩阵
Mat DeltaMat and                // 反向传播的中间临时矩阵
Mat NablaWbMat and              // 带有偏置的权值梯度矩阵
```

```
Mat ActiFunDerivationMat and     // 激活函数导数矩阵
int NeuronNum and                // 神经元个数
int AcitFuncNum                  // 激活函数
};
```

注：

- 激活函数整形映射规则见表6.1。

表 6.1: 激活函数映射规则

整型变量值	激活函数
0	no activation
1	sigmoid
2	tanh
3	relu
4	leak relu
5	softmax

6.1.3 神经网络结构体

代码 6.3: 神经网络结构体

```
struct FCNN {
int CurrentSampleNum and         // 当前神经网络处理的样本数量
int SampleDimensionNum and       // 样本的特征数
int HiddenLayerNum and           // 隐藏层个数
int WeightInitWayNum and         // 权值初始化方式
FCLayer *Layer and               // 神经网络所有层
Mat OnehotMat and                // onehot 编码的标签
int ClassificationNum and        // 神经网络的分类个数
int LossFuncNum                  // 损失函数
};
```

注：

- 权值初始化方式映射规则见表6.2以及损失函数映射规则见表6.3。

6.1.4 数据集结构体

代码 6.4: 数据集结构体

```
struct DataSet {
Mat CompleteFeatureDataSet and      // 全体特征数据
Mat CompleteLabelDataSet and        // 全体数据的标签
Mat CompleteTrainFeature and        // 训练特征数据
Mat CompleteTrainLabel and          // 训练数据标签（非独热编码）
Mat *BatchTrainFeature and          // 批量训练特征数据（三维）
Mat *BatchTrainLabel and            // 批量训练数据标签（非独热编码）
Mat *BatchTrainLabelOneHot and      // 批量训练数据标签（独热编码）
Mat TestFeature and                 // 测试特征数据
Mat TestLabel and                   // 测试数据标签（非独热编码）
Mat TestLabelOneHot and             // 测试数据标签（独热编码）
int CompleteSampleNum and           // 全体数据集个数
int TrainSampleNum and              // 训练数据个数
int TestSampleNum and               // 测试数据个数
int SampleDimensionNum and          // 数据样本特征数
int ClassificationNum and           // 分类个数
int BatchSize and                   // 批量数据块大小
int BatchNum and                    // 批量数据块数
int remainder                       // 最后一批数据块大小（看能否整除）
};
```

注：

- 余数机制是为了保证神经网络的小批量优化训练数据完整性。

表 6.2: 权值初始化方式映射规则

整型变量值	初始化方式
0	全零初始化
1	随机初始化
2	xavier 初始化
3	凯明初始化

表 6.3: 损失函数映射规则

整型变量值	损失函数
0	均方差损失
1	交叉熵损失

6.1.5 用户自定义参数结构体

代码 6.5: 用户自定义结构体

```
struct Custom {
int CompleteSampleNum and        // 全体数据集个数
int TrainSampleNum and           // 训练数据个数
int TestSampleNum and            // 测试数据个数
int SampleDimensionNum and       // 数据样本特征数
int HiddenLayerNum and           // 神经网络隐藏层个数
int WeightInitWayNum and         // 神经网络初始化方式
float *XValArray and             // 全体特征数据数组
float *YValArray and             // 全体标签数据数组
int *NeuronNumArray and          // 神经网络各层神经元个数
int *ActiFuncNumArray and        // 神经网络各层激活函数
int ClassificationNum and        // 神经网络分类数
int LossFuncNum and              // 损失函数
int BatchSize                    // 批量优化算法块大小
};
```

6.1.6 Adam 优化参数结构体

代码 6.6: Adam 优化参数结构体

```
typedef struct {
    float beta1;
    float beta2;
    float eta;
    float epsilon;
    int time;
    Mat *v;
    Mat *hat_v;
    Mat *s;
    Mat *hat_s;
    Mat *hat_g;
}AdamPara;
```

6.2 基础函数说明

6.2.1 辅助函数

表 6.4: 辅助函数

函数名	输入类型		输出类型	功能
absolute	float		float	浮点数绝对值
min	float	float	float	两浮点数较小的一个
equal	float	float	int	判定两浮点数是否相等
F2S	float	char *	char *	实现浮点数向字符串的转换

由于 MSVL 编译器对浮点型数据标准输出会出现错误，而能正确标准输出字符串。所以其中转换函数 F2S 是解决 MSVL 编译器对浮点型数据的标准输出问题的一种方案。

6.2.2 二维矩阵操作函数

注：

- 函数 12 MatMul2 矩阵乘法是对函数 11 MatMul 矩阵乘法关于稀疏矩阵的优化 [7]。
- 函数 13 MatProduct 矩阵哈曼积是同维矩阵对应元素相乘，区别于矩阵乘法。
- 函数 20, 21, 22 MatSquare, MatSqrt, MatExp 均是对矩阵中的每一个元素进行一元运算。
- 函数 23 MatVectorSub 矩阵向量减得运算规则是：

$$\boldsymbol{D}_{m\times n},\ \boldsymbol{A}_{m\times n},\ \boldsymbol{B}_{m\times 1}:\qquad \boldsymbol{D}_{ij}=\boldsymbol{A}_{ij}-\boldsymbol{B}_{i1}$$

- 函数 26 MatPlusCol 矩阵列扩展规则是目标矩阵的最左列扩展一列并全部赋值为 1; 函数 27 MatPlusRow 类似。

函数 12 MatMul2 矩阵乘法关于稀疏矩阵的优化，相比函数 11 MatMul 此举可将大大减少神经网络的训练时间。代码详见例 (6.1)。

例 6.1 矩阵乘法代码：

代码 6.7: 函数 11 MatMul

```
1 function MatMul ( Mat *src1,Mat *src2,Mat *dst,Mat* RValue )
2  {
```

表 6.5: 二维矩阵操作函数

编号	函数名	输入类型			输出类型	功能
1	MatCreate	Mat *	int	int	Mat *	创建固定行数列数的矩阵空间
2	MatDelete	Mat *			NULL	删除目标矩阵的空间
3	MatSetVal	Mat *	float *		Mat *	将数组元素赋值给矩阵
4	MatShape	const Mat *			NULL	标准输出目标矩阵的行数列数
5	MatDump	const Mat *			NULL	标椎输出目标矩阵
6	MatZeros	Mat *			Mat *	目标矩阵赋值为全 0 阵
7	MatOnes	Mat *			Mat *	目标矩阵赋值为全 1 阵
8	MatEye	Mat *			Mat *	目标矩阵赋值为对角 1 阵
9	MatAdd	Mat *	Mat *	Mat *	Mat *	前两矩阵相加赋值给末矩阵
10	MatSub	Mat *	Mat *	Mat *	Mat *	前两矩阵相减赋值给末矩阵
11	MatMul	Mat *	Mat *	Mat *	Mat *	前两矩阵相乘赋值给末矩阵
12	MatMul2	Mat *	Mat *	Mat *	Mat *	矩阵相乘赋值给末矩阵（稀疏优化）
13	MatProduct	Mat *	Mat *	Mat *	Mat *	前两矩阵哈曼积赋值给末矩阵
14	MatDiv	Mat *	Mat *	Mat *	Mat *	前两矩阵相除赋值给末矩阵
15	MatNumMul	int	Mat *	Mat *	Mat *	数乘矩阵赋值给末矩阵
16	MatNumAdd	Mat *	Mat *	Mat *	Mat *	数加矩阵赋值给末矩阵
17	MatTrans	Mat *		Mat *	Mat *	矩阵转置赋值给末矩阵
18	MatRowSum	Mat *		Mat *	Mat *	矩阵每一行相加赋值给末矩阵
19	MatRowMax	Mat *		Mat *	Mat *	矩阵每一行最大值赋值给末矩阵
20	MatSquare	Mat *		Mat *	Mat *	矩阵平方赋值给末矩阵
21	MatSqrt	Mat *		Mat *	Mat *	矩阵开根号赋值给末矩阵
22	MatExp	Mat *		Mat *	Mat *	矩阵自然底数幂赋值给末矩阵
23	MatVectorSub	Mat *	Mat *	Mat *	Mat *	矩阵行减向量赋值给末矩阵
24	MatVectorDiv	Mat *	Mat *	Mat *	Mat *	矩阵行除向量赋值给末矩阵
25	MatCopy	Mat *		Mat *	Mat *	矩阵拷贝赋值给末矩阵
26	MatPlusCol	Mat *			Mat *	矩阵一列扩展赋值给末矩阵
27	MatPlusRow	Mat *			Mat *	矩阵一行扩展赋值给末矩阵

```
3       frame(MatMul_row,MatMul_col,MatMul_i,MatMul_temp,return) and (
4       int return<==0 and skip;
5       int MatMul_row,MatMul_col and skip;
6       int MatMul_i and skip;
7       float MatMul_temp and skip;
8       MatMul_row:=0;
```

```
9
10      while( (MatMul_row<dst->row) )
11      {
12          MatMul_col:=0;
13
14          while( (MatMul_col<dst->col) )
15          {
16              MatMul_temp:=0.0;
17              MatMul_i:=0;
18
19              while( (MatMul_i<src1->col) )
20              {
21                  MatMul_temp:=MatMul_temp+(src1->element[MatMul_row])[
    MatMul_i]*(src2->element[MatMul_i])[MatMul_col];
22                  MatMul_i:=MatMul_i+1
23              };
24              (dst->element[MatMul_row])[MatMul_col]:=MatMul_temp;
25              MatMul_col:=MatMul_col+1
26          };
27          MatMul_row:=MatMul_row+1
28      };
29      return<==1 and RValue:=dst;
30      skip
31      )
32      };
```

代码 6.8: 函数 12 MatMul2

```
1  function MatMul2 ( Mat *src1,Mat *src2,Mat *dst,Mat* RValue )
2   {
3       frame(MatMul2_row,MatMul2_col,MatMul2_i,MatMul2_1_temp$_1,return) and (
4       int return<==0 and skip;
5       int MatMul2_row,MatMul2_col and skip;
6       int MatMul2_i and skip;
7       MatZeros(dst,RValue);
8       MatMul2_row:=0;
9
10      while( (MatMul2_row<src1->row) )
```

```
11      {
12          MatMul2_col:=0;
13
14          while( (MatMul2_col<src1->col) )
15          {
16              int MatMul2_1_temp$_1 and skip;
17              MatMul2_1_temp$_1:=equal((src1->element[MatMul2_row])[
    MatMul2_col],0,RValue);
18              if(MatMul2_1_temp$_1=0) then                    // 判定0因子
19              {
20                  MatMul2_i:=0;
21                  while( (MatMul2_i<src2->col) )
22                  {
23                      (dst->element[MatMul2_row])[MatMul2_i]:=(dst->element[
    MatMul2_row])[MatMul2_i]+(src1->element[MatMul2_row])[MatMul2_col]*(src2
    ->element[MatMul2_col])[MatMul2_i];           // 改写二维数组指针为两个一
    维
24                      MatMul2_i:=MatMul2_i+1
25                  }
26              }
27              else
28              {
29                   skip
30              };
31              MatMul2_col:=MatMul2_col+1
32          };
33          MatMul2_row:=MatMul2_row+1
34      };
35      return<==1 and RValue:=dst;
36      skip
37      )
38      };
```

函数 12 MatMul2 相比函数 11 MatMul 对稀疏矩阵优化的核心在于判定零因子，每判定一个零因子将会减少第二个矩阵的列次乘操作，稀疏矩阵的零因子多，因此能提高矩阵乘法的效率。 □

MSVL 编译器不支持二维数组指针，将其改写为两个一维。

6.2.3 激活函数相关函数

表 6.6: 激活函数相关函数

编号	函数名	输入类型	输出类型	功能
1	sigmoid (公式2.1)	float	float	sigmoid 激活
2	tanh (公式2.3)	float	float	tanh 激活
3	relu (公式2.5)	float	float	relu 激活
4	leakyRelu (公式2.7)	float float	float	leakyRelu 激活
5	MatSoftmax (公式2.9)	Mat *	Mat *	Softmax 激活
6	MatNoneActi	Mat *	Mat *	不激活
7	MatSigmoid	Mat *	Mat *	sigmoid 激活矩阵
8	MatTanh	Mat *	Mat *	tanh 激活矩阵
9	MatRelu	Mat *	Mat *	relu 激活矩阵
10	MatLeakyRelu	Mat *	Mat *	leakyRelu 激活矩阵
11	MatDerivationSoftmax (公式2.10)	Mat *	Mat *	Softmax 矩阵求导函数
12	MatDerivationNoneActi	Mat *	Mat *	不激活矩阵求导函数
13	MatDerivationSigmoid (公式2.2)	Mat *	Mat *	sigmoid 矩阵求导函数
14	MatDerivationTanh (公式2.4)	Mat *	Mat *	tanh 矩阵求导函数
15	MatDerivationRelu (公式2.6)	Mat *	Mat *	relu 矩阵求导函数
16	MatDerivationLeakyRelu(公式2.8)	Mat *	Mat *	leakyrelu 矩阵求导函数

注：

- 函数 2 tanh 是利用 C 语言 math 库中，未重写。
- 函数 6 MatNoneActi 未激活矩阵函数是为了统一规范编写激活函数。

MSVL 编译器可以外部调用 C 语言的函数库里的函数，将相关函数声明到 MC8.0 项目资源文件 external.c 里适当位置上即可。

例 6.2 softmax 及其导函数代码：

代码 6.9: 函数 5 MatSoftmax

```
function MatSoftmax ( Mat *src,Mat *dst,Mat* RValue )
 {
     frame(MatSoftmax_tempV,return) and (
     int return<==0 and skip;
     Mat MatSoftmax_tempV and skip;
```

```
6       MatCreate(&MatSoftmax_tempV,src->row,1,RValue);     //存临时的最大行值以
    及求和行值向量
7       MatRowMax(src,&MatSoftmax_tempV,RValue);            //求最大行值向量
8       MatVectorSub(src,&MatSoftmax_tempV,dst,RValue);     //矩阵向量相减
9       MatExp(dst,dst,RValue);                             //矩阵求对数
10      MatRowSum(dst,&MatSoftmax_tempV,RValue);            //求行求和向量
11      MatVectorDiv(dst,&MatSoftmax_tempV,dst,RValue);     //矩阵向量相除
12      MatDelete(&MatSoftmax_tempV);
13      return<==1 and RValue:=dst;
14      skip
15      )
16      };
```

代码 6.10: 函数 11 MatDerivationSoftmax

```
1  function MatDerivationSoftmax ( Mat *src,Mat *dst,Mat* RValue )
2   {
3       frame(MatDerivationSoftmax_row,MatDerivationSoftmax_col,
    MatDerivationSoftmax_i,return) and (
4       int return<==0 and skip;
5       int MatDerivationSoftmax_row,MatDerivationSoftmax_col,
    MatDerivationSoftmax_i and skip;
6       MatSoftmax(src,src,RValue);                 // 先用 softmax 激活，在根
    据公式判断选择。
7       MatZeros(dst,RValue);
8       MatDerivationSoftmax_row:=0;
9
10      while( (MatDerivationSoftmax_row<src->row) )
11      {
12          MatDerivationSoftmax_col:=0;
13
14          while( (MatDerivationSoftmax_col<src->col) )
15          {
16              MatDerivationSoftmax_i:=0;
17
18              while( (MatDerivationSoftmax_i<src->col) )
19              {
20                  if(MatDerivationSoftmax_i=MatDerivationSoftmax_col) then
```

```
        // 选择一
21                  {
22                      (dst->element[MatDerivationSoftmax_row])[
    MatDerivationSoftmax_col]:=(dst->element[MatDerivationSoftmax_row])[
    MatDerivationSoftmax_col]+(src->element[MatDerivationSoftmax_row])[
    MatDerivationSoftmax_i]*(1-(src->element[MatDerivationSoftmax_row])[
    MatDerivationSoftmax_col])
23                  }
24                  else
        // 选择二
25                  {
26                      (dst->element[MatDerivationSoftmax_row])[
    MatDerivationSoftmax_col]:=(dst->element[MatDerivationSoftmax_row])[
    MatDerivationSoftmax_col]+-(src->element[MatDerivationSoftmax_row])[
    MatDerivationSoftmax_i]*(src->element[MatDerivationSoftmax_row])[
    MatDerivationSoftmax_col]
27                  };
28                  MatDerivationSoftmax_i:=MatDerivationSoftmax_i+1
29              };
30              MatDerivationSoftmax_col:=MatDerivationSoftmax_col+1
31          };
32          MatDerivationSoftmax_row:=MatDerivationSoftmax_row+1
33      };
34      return<==1 and RValue:=dst;
35      skip
36      )
37      };
```

□

6.2.4 损失函数相关函数

注：

- 函数 1 OneHot 将参数一（必须是从零开始连续地标签数字向量），按照第二个参数分类数 K，进行 ONEHOT 编码，形成目标矩阵 [21]。

例 6.3 交叉熵及其导函数代码：

表 6.7: 损失函数相关函数

编号	函数名	输入类型			输出类型	功能
1	OneHot	Mat *	int	Mat *	Mat *	进行 ONEHOT 编码
2	MSE(公式2.11)	Mat *	Mat *		float	得到两矩阵的均方误差
3	CrossEntropy(公式2.13)	Mat *	Mat *		float	得到两矩阵的交叉熵损失
4	MSEDerivative(公式2.12)	Mat *	Mat *	Mat	Mat *	得到均方误差对权值的导函数
5	CrossEntropyDerivative(公式2.14)	Mat *	Mat *	Mat	Mat *	得到交叉熵损失对权值的导函数

代码 6.11: 函数 3 CrossEntropy

```
function CrossEntropy ( Mat *src,Mat *dst,float RValue )
 {
     frame(CrossEntropy_row,CrossEntropy_col,CrossEntropy_loss,return) and (
     int return<==0 and skip;
     int CrossEntropy_row,CrossEntropy_col and skip;
     float CrossEntropy_loss<==0.0 and skip;
     CrossEntropy_row:=0;

     while( (CrossEntropy_row<src->row) )
     {
         CrossEntropy_col:=0;

         while( (CrossEntropy_col<src->col) )
         {
             CrossEntropy_loss:=CrossEntropy_loss+(float)(-1*(dst->element[
    CrossEntropy_row])[CrossEntropy_col]*log((src->element[CrossEntropy_row
    ])[CrossEntropy_col]));            //  依照公式求总  之后再平均
             CrossEntropy_col:=CrossEntropy_col+1

         };
         CrossEntropy_row:=CrossEntropy_row+1

     };
     CrossEntropy_loss:=CrossEntropy_loss/ (src->row);          // 平均
     return<==1 and RValue:=CrossEntropy_loss;
     skip
     )
```

```
26      };
```

代码 6.12: 函数 5 CrossEntropyDerivative

```
1  function CrossEntropyDerivative ( Mat *ActiMat,Mat *DerivativeActiMat,Mat
     One_hotMat,Mat* RValue )
2  {
3      frame(CrossEntropyDerivative_row,CrossEntropyDerivative_col,
     CrossEntropyDerivative_1_2_temp$_1,return) and (
4      int return<==0 and skip;
5      int CrossEntropyDerivative_row,CrossEntropyDerivative_col and skip;
6      CrossEntropyDerivative_row:=0;
7
8      while( (CrossEntropyDerivative_row<ActiMat->row) )
9      {
10         CrossEntropyDerivative_col:=0;
11
12         while( (CrossEntropyDerivative_col<ActiMat->col) )
13         {
14             int CrossEntropyDerivative_1_2_temp$_1 and skip;
15             CrossEntropyDerivative_1_2_temp$_1:=equal((ActiMat->element[
     CrossEntropyDerivative_row])[CrossEntropyDerivative_col],0.0,RValue);
16             if(CrossEntropyDerivative_1_2_temp$_1=1) then      // 当除数为
     零时  为防止除零报错 直接给被除数乘无限大
17             {
18                 (DerivativeActiMat->element[CrossEntropyDerivative_row])[
     CrossEntropyDerivative_col]:=-(One_hotMat.element[
     CrossEntropyDerivative_row])[CrossEntropyDerivative_col]*10000000000
19
20             }
21             else       // 正常除
22             {
23                 (DerivativeActiMat->element[CrossEntropyDerivative_row])[
     CrossEntropyDerivative_col]:=-(One_hotMat.element[
     CrossEntropyDerivative_row])[CrossEntropyDerivative_col]/ (ActiMat->
     element[CrossEntropyDerivative_row])[CrossEntropyDerivative_col]
24             };
25             CrossEntropyDerivative_col:=CrossEntropyDerivative_col+1
```

```
26
27          };
28          CrossEntropyDerivative_row:=CrossEntropyDerivative_row+1
29
30      };
31      return<==1 and RValue:=DerivativeActiMat;
32      skip
33      )
34      };
```

□

6.2.5 权值初始化函数

表 6.8: 激活函数相关函数

编号	函数名	输入类型		输出类型	功能
1	gaussrand_NORMAL	None		float	随机生成标椎正态分布数据
2	gaussrand	float	float	float	随机生成指定正态分布数据
3	MatInitZero	Mat *		Mat *	全零初始化
4	MatInitRandomNormalization	Mat *		Mat *	随机初始化
5	MatInitXavier(公式2.18)	Mat *		Mat *	Xavier 初始化
6	MatInitHe(公式2.19)	Mat *		Mat *	凯明初始化

注：

- 所有的权值初始化均不包括 Bias，Bias 初始化成零。
- 函数 1 gaussrand_NORMAL 生成正态分布数据采用的是 Box-Muller 方法。
- 函数 4 MatInitRandomNormalization 随机初始生成是 $\mu = 0,\ \sigma = 0.1$ 正态分布数据。

例 6.4 Xavier *初始化以及凯明初始化代码*：

代码 6.13: 函数 5 MatInitXavier

```
1  function MatInitXavier ( Mat *src,Mat* RValue )
2   {
3       frame(MatInitXavier_temp$_1,MatInitXavier_row,MatInitXavier_col,return)
      and (
4       int return<==0 and skip;
5       int MatInitXavier_temp$_1 and skip;
```

```
    MatInitXavier_temp$_1:=time(NULL);
    srand((unsigned int)MatInitXavier_temp$_1) and skip;
    int MatInitXavier_row,MatInitXavier_col and skip;
    MatInitXavier_row:=0;

    while( (MatInitXavier_row<src->row) )
    {
        MatInitXavier_col:=0;

        while( (MatInitXavier_col<src->col) )
        {
            (src->element[MatInitXavier_row])[MatInitXavier_col]:=gaussrand
(0.0,0.1,RValue)*(float)sqrt(1.0/ src->row);        // 依照公式  每项生
成
            MatInitXavier_col:=MatInitXavier_col+1

        };
        MatInitXavier_row:=MatInitXavier_row+1
    };
    return<==1 and RValue:=src;
    skip
    )
    };
```

代码 6.14: 函数 6 MatInitHe

```
function MatInitHe ( Mat *src,Mat* RValue )
{
    frame(MatInitHe_row,MatInitHe_col,return) and (
    int return<==0 and skip;
    srand(19950826) and skip;
    int MatInitHe_row,MatInitHe_col and skip;
    MatInitHe_row:=0;

    while( (MatInitHe_row<src->row) )
    {
        MatInitHe_col:=0;

```

```
13          while( (MatInitHe_col<src->col) )
14          {
15              (src->element[MatInitHe_row])[MatInitHe_col]:=gaussrand
    (0.0,0.9,RValue)*(float)sqrt(2.0/ src->row);          // 依照公式  每项生
    成
16              MatInitHe_col:=MatInitHe_col+1
17
18          };
19          MatInitHe_row:=MatInitHe_row+1
20      };
21      return<==1 and RValue:=src;
22      skip
23      )
24      };
```

□

6.3 开发流程及其函数说明

具体开发流程：

1. 用户自定义参数录入
2. 数据集构建
3. 神经网络初始化
4. 前向传播构建
5. 反向传播构建
6. 优化算法构建
7. 神经网络训练构建
8. 神经网络测试构建

6.3.1 用户自定义参数录入

目的是将相关参数及总的数据赋值给用户自定义结构体中（其中注意对数组数据的赋值，传引用即可），再将各个参数分发给各个结构体。

表 6.9: 用户自定义参数录入相关函数

编号	函数名	输入类型			输出类型	功能
1	InitCustom	Custom *			int	初始化 Custom 参数
2	DumpFloatArray	float*	int		None	标椎输出浮点型数组
3	DumpIntArray	int*	int		None	标椎输出整型数组
4	DumpCustom	Custom			int	标椎输出 Custom
5	LoadParaFromCustom	Custom	DataSet *	FCNN *	None	传参到具体结构体中

注：

- 用户自定义参数的录入，包括整体数据以数组的形式录入。
- 四个数组形式的自定义参数单独录入，不再录入函数中录入。

例 6.5 基本参数录入函数代码：

代码 6.15: 函数 5 LoadParaFromCustom

```
1  function LoadParaFromCustom ( Custom userDefine,DataSet *dataSet,FCNN *fcnn
       )
2   {
3       // Custom ==>> DataSet
4       dataSet->BatchSize:=userDefine.BatchSize;
5       dataSet->CompleteSampleNum:=userDefine.CompleteSampleNum;
6       dataSet->SampleDimensionNum:=userDefine.SampleDimensionNum;
7       dataSet->TrainSampleNum:=userDefine.TrainSampleNum;
8       dataSet->TestSampleNum:=userDefine.TestSampleNum;
9       dataSet->ClassificationNum:=userDefine.ClassificationNum;
10
11      // Custom ==>> FCNN
12      fcnn->CurrentSampleNum:=userDefine.BatchSize;                // 代表当前
      神经网络前向传播时的数据样本的个数
13      fcnn->SampleDimensionNum:=userDefine.SampleDimensionNum;
14      fcnn->HiddenLayerNum:=userDefine.HiddenLayerNum;
15      fcnn->WeightInitWayNum:=userDefine.WeightInitWayNum;
```

```
16      fcnn->ClassificationNum:=userDefine.ClassificationNum;
17      fcnn->LossFuncNum:=userDefine.LossFuncNum
18  };
```

函数 5 LoadParaFromCustom 不包括数组形式的数据录入，以及为创建空间的相关参数的录入。 □

数组数据的录入，可直接在主函数下传引用即可。

6.3.2 数据集构建

根据用户输入的数据相关参数将训练测试数据集构建完成。构建路线：

1. 根据训练集数据量以及 Batch_Size 大小计算并导入相关参数。
2. 根据整体数据集的各个参数，开辟所需的全部空间。
3. 录入数据集。
   (a) 将整体数据集数组根据训练测试集数目划分成两个数组，包括训练特征数组和训练标签数组，测试特征数组和测试标签数组。
   (b) 再将训练特征数组和训练标签数组划分成若干个块大小的块训练特征数组和块训练标签数组。
   (c) 最后将数据喂入到相应的矩阵结构体当中。

表 6.10: 数据集构建相关函数

编号	函数名	输入类型		输出类型	功能
1	InitDataSet	DataSet *		int	初始化 DataSet 参数
2	CalculateAndLoadDataSetPara	DataSet *		None	计算相关参数并录入
3	CreateDataSetSpace	DataSet *		None	创建数据集所需空间
4	DataLoading	Custom	DataSet *	int	录入矩阵形式数据集
5	DatasetConstruction	Custom	DataSet *	None	数据及构建主函数

注：

- 总体数据集分开成为训练集和测试集，暂不分出验证集。

例 6.6 数据及构建函数代码：

代码 6.16: 函数 5 DatasetConstruction

```
1  function DatasetConstruction ( Custom userDefine,DataSet *dataSet )
```

```
{
    CalculateAndLoadDataSetPara(dataSet);        // 步骤1
    CreateDataSetSpace(dataSet);                 // 步骤2
    DataLoading(userDefine,dataSet,RValue)       // 步骤3
};
```

□

6.3.3 神经网络初始化

神经网络初始化

1. 开辟神经网络所需空间并导入每层相关参数（注意创建顺序不要变化，1.c 与 1.d 可以互换）
   (a) 创建神经网络层空间
      i. FCLayer
   (b) 导入每层参数
      i. AcitFuncNum
      ii. NeuronNum
   (c) 创建神经网络运算空间
      i. ActiMat
      ii. ActiMatPlus
      iii. SumMat
      iv. ActiFunDerivationMat
      v. DeltaMat
      vi. OnehotMat（FCNN）
   (d) 创建神经网络参数空间(参数空间的大小不随输入神经网络样本大小而改变)
      i. WeightMat
      ii. WeightBiasMat
      iii. NablaWbMat
2. 进行权值初始化
   (a) WeightMat
   (b) WeightBiasMat

表 6.11: 创建空间及录入相关函数

编号	函数名	输入类型		输出类型	功能
1	InitFCNN	FCNN *		int	初始化 FCNN 参数
2	InitFCLayer	FCNN *		int	初始化 FCNNLayer 参数
3	SpaceCreateFCLayer	FCNN *		FCLayer *	创建 FCLayer 结构体所需空间
4	SpaceCreateActi	FCNN *		None	创建激活值矩阵所需空间
5	SpaceDeleteActi	FCNN *		None	删除激活值矩阵空间
6	SpaceCreateActiPlus	FCNN *		None	创建激活值 Plus 矩阵所需空间
7	SpaceDeleteActiPlus	FCNN *		None	删除激活值 Plus 矩阵空间
8	SpaceCreateSum	FCNN *		None	创建求和值矩阵所需空间
9	SpaceDeleteSum	FCNN *		None	删除求和值矩阵空间
10	SpaceCreateActiFunDerivation	FCNN *		None	创建激活导函数矩阵所需空间
11	SpaceDeleteActiFunDerivation	FCNN *		None	删除激活导函数矩阵空间
12	SpaceCreateDelta	FCNN *		None	创建反向中间变量矩阵所需空间
13	SpaceDeleteDelta	FCNN *		None	删除反向中间变量矩阵空间
14	SpaceCreateFCNNOneHotMat	FCNN *		None	创建 ONEHOT 矩阵所需空间
15	SpaceDeleteFCNNOneHotMat	FCNN *		None	删除 ONEHOT 矩阵空间
16	SpaceCreateWeight	FCNN *		None	创建权值矩阵所需空间
17	SpaceCreateWeightBias	FCNN *		None	创建权值偏置矩阵所需空间
18	SpaceCreateNablaWeightBias	FCNN *		None	创建权值偏置梯度矩阵所需空间
19	CreateNNOperationSpace	FCNN *		None	创建神经网络运算所需空间
20	DeleteNNOperationSpace	FCNN *		None	删除神经网络运算空间
21	CreateNNParaSpace	FCNN *		None	创建神经网络参数所需空间
22	DoadinPara2FCLayer	FCNN *	Custom	None	录入每层相关参数
23	CreateNNSpaceAndLoadinPara2FCLayer	FCNN *	Custom	int	总创建空间及录入函数

注：

- 神经网络运算参数空间在小批量优化算法当中，有余数时，会进行运算参数空间的重建。
- 在此导入 FCLayer 相关参数，而不再一开始导入是因为创造空间的原因。

例 6.7 总创建空间及录入函数代码：

代码 6.17: 函数 23 CreateNNSpaceAndLoadinPara2FCLayer

```
1  function CreateNNSpaceAndLoadinPara2FCLayer ( FCNN *fcnn,Custom userDefine
     ,int RValue )
2  {
3      frame(return) and (
```

```
4       int return<==0 and skip;
5       fcnn->Layer:=SpaceCreateFCLayer(fcnn,RValue);  // 步骤1.a
6       DoadinPara2FCLayer(fcnn,userDefine);           // 步骤1.b
7       CreateNNOperationSpace(fcnn);                  // 步骤1.c
8       CreateNNParaSpace(fcnn);                       // 步骤1.d
9       return<==1 and RValue:=0;
10      skip
11      )
12      };
```

创建 FCLayer 结构体所需空间函数代码：

代码 6.18: 函数 3 SpaceCreateFCLayer

```
1  function SpaceCreateFCLayer ( FCNN *fcnn,FCLayer* RValue )
2   {
3       frame(return) and (
4       int return<==0 and skip;
5       fcnn->Layer:=(FCLayer *)malloc((fcnn->HiddenLayerNum+2)*sizeof(FCLayer)
      );  // 步骤1.a.i
6       return<==1 and RValue:=fcnn->Layer;
7       skip
8       )
9       };
```

录入每层相关参数函数代码：

代码 6.19: 函数 22 DoadinPara2FCLayer

```
1   function DoadinPara2FCLayer ( FCNN *fcnn,Custom userDefine )
2   {
3       frame(DoadinPara2FCLayer_i) and (
4       int DoadinPara2FCLayer_i<==0 and skip;
5
6       while( (DoadinPara2FCLayer_i<fcnn->HiddenLayerNum+2) )
7       {
8           fcnn->Layer[DoadinPara2FCLayer_i].AcitFuncNum:=userDefine.
      ActiFuncNumArray[DoadinPara2FCLayer_i];  // 步骤1.b.i
9           fcnn->Layer[DoadinPara2FCLayer_i].NeuronNum:=userDefine.
      NeuronNumArray[DoadinPara2FCLayer_i];      // 步骤1.b.ii
10          DoadinPara2FCLayer_i:=DoadinPara2FCLayer_i+1
```

```
11        }
12        )
13        };
```

创建神经网络运算所需空间函数代码：

代码 6.20: 函数 19 CreateNNOperationSpace

```
1   function CreateNNOperationSpace ( FCNN *fcnn )
2   {
3       SpaceCreateActi(fcnn);                      // 步骤1.c.i
4       SpaceCreateActiPlus(fcnn);                  // 步骤1.c.ii
5       SpaceCreateSum(fcnn);                       // 步骤1.c.iii
6       SpaceCreateActiFunDerivation(fcnn);         // 步骤1.c.iv
7       SpaceCreateDelta(fcnn);                     // 步骤1.c.v
8       SpaceCreateFCNNOneHotMat(fcnn)              // 步骤1.c.vi
9   };
```

创建神经网络参数所需空间函数代码：

代码 6.21: 函数 21 CreateNNParaSpace

```
1   function CreateNNParaSpace ( FCNN *fcnn )
2   {
3       SpaceCreateWeight(fcnn);            // 步骤1.d.i
4       SpaceCreateWeightBias(fcnn);        // 步骤1.d.ii
5       SpaceCreateNablaWeightBias(fcnn)    // 步骤1.d.iii
6
7   };
```

□

表 6.12: 权值初始化相关函数

编号	函数名	输入类型		输出类型	功能
24	WeightInit_ChooseWay	Mat *	int	None	权值初始化方式选择函数
25	NNWeightinit	FCNN *		int	初始化权值

注：

- 权值初始化方式选择函数 24 第二个 int 型参数可根据表 (6.2) 选择具体的初始化方式。
- 权值初始化虽分为 2.a、2.b，但是只初始化了一次，2.b 是 2.a 的行扩展复制。

例 6.8 *初始化权值函数代码：*

代码 6.22: 函数 25 NNWeightinit

```
function NNWeightinit ( FCNN *fcnn,int RValue )
{
    frame(NNWeightinit_i,return) and (
    int return<==0 and skip;
    int NNWeightinit_i<==1 and skip;

    while( (NNWeightinit_i<fcnn->HiddenLayerNum+2) )
    {
        WeightInit_ChooseWay(&fcnn->Layer[NNWeightinit_i].WeightMat,fcnn->
  WeightInitWayNum);                    // 步骤2.a
        MatPlusRow(&fcnn->Layer[NNWeightinit_i].WeightMat,&fcnn->Layer[
  NNWeightinit_i].WeightBiasMat);      // 步骤2.b 仅仅是行扩展复制 并没有
  重新初始化 也不能重新初始化
        NNWeightinit_i:=NNWeightinit_i+1
    };
    return<==1 and RValue:=0;
    skip
    )
    };
```

□

6.3.4 前向传播构建

神经网络向前传播将小批量输入样本导入神经网络输入层激活值矩阵，并将对应的标签数据进行独热编码，然后进行矩阵运算损失处理得到标量损失值。其中要注意小批量输入样本的数量不一定和已经开辟的神经网络有关矩阵空间的数量相一致，要做检查处理。

1. 小批量余数处理
2. 输入特征数据拷贝到激活值矩阵
3. 输入标签独热编码数据拷贝到神经网络独热编码矩阵中
4. 前向传播矩阵相乘激活扩展激活值矩阵
5. 预测值矩阵与真实值矩阵求损失返回

表 6.13: 前向传播相关函数

编号	函数名	输入类型			输出类型	功能
1	MatActivate	Mat *	Mat *	int	None	带方式选择激活函数
2	LossFunction	Mat *	Mat *	int	None	带方式选择损失函数
3	NNforward	Mat	Mat	FCNN *	float	神经网络前向传播函数

注：

- 函数 1 激活方式的选择可参照表6.1。
- 函数 2 损失函数的选择可参照表6.3。

例 6.9 神经网络前向传播函数代码：

代码 6.23: 函数 3 NNforward

```
function NNforward ( Mat featureMat,Mat labelMatOneHot,FCNN *fcnn,float
    RValue )
 {
     frame(NNforward_i,NNforward_loss,return) and (
     int return<==0 and skip;
     if(featureMat.row!=fcnn->CurrentSampleNum) then                    // 步骤1
     {
         fcnn->CurrentSampleNum:=featureMat.row;
         DeleteNNOperationSpace(fcnn);
         CreateNNOperationSpace(fcnn)
     }
     else
     {
          skip
     };
     MatCopy(&featureMat,&fcnn->Layer[0].ActiMat);                      // 步骤2
     MatPlusCol(&fcnn->Layer[0].ActiMat,&fcnn->Layer[0].ActiMatPlus);
    // 步骤3
     MatCopy(&labelMatOneHot,&fcnn->OnehotMat);
     int NNforward_i<==0 and skip;

     while( (NNforward_i<fcnn->HiddenLayerNum+1) )                      // 步骤4
     {
```

```
22          MatMul(&fcnn->Layer[NNforward_i].ActiMatPlus,&fcnn->Layer[
    NNforward_i+1].WeightBiasMat,&fcnn->Layer[NNforward_i+1].SumMat,RValue);
23          MatActivate(&fcnn->Layer[NNforward_i+1].SumMat,&fcnn->Layer[
    NNforward_i+1].ActiMat,fcnn->Layer[NNforward_i+1].AcitFuncNum,RValue);
24          if(NNforward_i!=fcnn->HiddenLayerNum) then
25          {
26              MatPlusCol(&fcnn->Layer[NNforward_i+1].ActiMat,&fcnn->Layer[
    NNforward_i+1].ActiMatPlus)
27          }
28          else
29          {
30               skip
31          };
32          NNforward_i:=NNforward_i+1
33      };
34      float NNforward_loss<==-1.0 and skip;
35      NNforward_loss:=LossFunction(&fcnn->Layer[fcnn->HiddenLayerNum+1].
    ActiMat,&fcnn->OnehotMat,fcnn->LossFuncNum,RValue);       // 步骤5
36      return<==1 and RValue:=NNforward_loss;
37      skip
38      )
39      };
```

□

6.3.5 反向传播构建

神经网络先经过一次正向传播得到损失之后，再求其损失对所有权值偏置的导数得到梯度，用到了反向传播算法。

1. 输出层反向传播求得输出层的中间变量矩阵以及权值导数
   (a) 输出层使用 Softmax 激活以及使用 CrossEntropy 损失 参见公式 (2.15)
   (b) 不使用上种选择 链式法则正常求导
2. 隐藏层反向传播求得各隐藏层的中间变量矩阵以及权值导数

注：

- 函数 1 激活方式的选择可参照表6.1。

表 6.14: 反向传播相关函数

编号	函数名	输入类型			输出类型	功能
1	ActiFunDerivation	Mat *	Mat *	int	Mat *	带方式选择激活导函数
2	LossFunDerivation	Mat *	Mat *	int	Mat *	带方式选择损失导函数
3	NNOuputLayerBackward	FCNN *			Mat *	输出层反向传播函数
4	NNBackward	FCNN *			Mat *	神经网络反向传播函数

- 函数 2 损失函数的选择可参照表6.3。

例 6.10 神经网络反向传播函数代码：

代码 6.24: 函数 4 NNBackward

```
function NNBackward ( FCNN *fcnn,Mat* RValue )
 {
    frame(NNBackward_i,NNBackward_tempTransW,NNBackward_ActiFuncMat,
   NNBackward_tempMulMat,NNBackward_tempProdMat,NNBackward_tempTransActi,
   return) and (
    int return<==0 and skip;
    NNOuputLayerBackward(fcnn,RValue);          // 步骤1
    int NNBackward_i<==fcnn->HiddenLayerNum and skip;

    while( (NNBackward_i>0) )       // 步骤2
    {
        Mat NNBackward_tempTransW and skip;
        Mat NNBackward_ActiFuncMat and skip;
        Mat NNBackward_tempMulMat and skip;
        Mat NNBackward_tempProdMat and skip;
        Mat NNBackward_tempTransActi and skip;
        MatCreate(&NNBackward_tempTransW,fcnn->Layer[NNBackward_i+1].
   WeightMat.col,fcnn->Layer[NNBackward_i+1].WeightMat.row,RValue);
        MatCreate(&NNBackward_ActiFuncMat,fcnn->Layer[NNBackward_i].SumMat.
   row,fcnn->Layer[NNBackward_i].SumMat.col,RValue);
        MatCreate(&NNBackward_tempMulMat,fcnn->Layer[NNBackward_i+1].
   DeltaMat.row,fcnn->Layer[NNBackward_i+1].WeightMat.row,RValue);
        MatCreate(&NNBackward_tempProdMat,fcnn->Layer[NNBackward_i].SumMat.
   row,fcnn->Layer[NNBackward_i].SumMat.col,RValue);
        MatCreate(&NNBackward_tempTransActi,fcnn->Layer[NNBackward_i-1].
   ActiMatPlus.col,fcnn->Layer[NNBackward_i-1].ActiMatPlus.row,RValue);
```

```
20          MatTrans(&fcnn->Layer[NNBackward_i+1].WeightMat,&
    NNBackward_tempTransW,RValue);
21          ActiFunDerivation(fcnn->Layer[NNBackward_i].SumMat,&
    NNBackward_ActiFuncMat,fcnn->Layer[NNBackward_i].AcitFuncNum,RValue);
22          MatMul(&fcnn->Layer[NNBackward_i+1].DeltaMat,&NNBackward_tempTransW
    ,&NNBackward_tempMulMat,RValue);
23          MatProduct(&NNBackward_tempMulMat,&NNBackward_ActiFuncMat,&fcnn->
    Layer[NNBackward_i].DeltaMat,RValue);
24          MatTrans(&fcnn->Layer[NNBackward_i-1].ActiMatPlus,&
    NNBackward_tempTransActi,RValue);
25          MatMul(&NNBackward_tempTransActi,&fcnn->Layer[NNBackward_i].
    DeltaMat,&fcnn->Layer[NNBackward_i].NablaWbMat,RValue);
26          MatNumMul(1.0/ fcnn->CurrentSampleNum,&fcnn->Layer[NNBackward_i].
    NablaWbMat,&fcnn->Layer[NNBackward_i].NablaWbMat,RValue);
27          MatDelete(&NNBackward_tempTransW);
28          MatDelete(&NNBackward_ActiFuncMat);
29          MatDelete(&NNBackward_tempMulMat);
30          MatDelete(&NNBackward_tempProdMat);
31          MatDelete(&NNBackward_tempTransActi);
32          NNBackward_i:=NNBackward_i-1
33
34      };
35      return<==1 and RValue:=NULL;
36      skip
37      )
38      };
```

神经网络输出层反向传播函数代码：

代码 6.25: 函数 3 NNOuputLayerBackward

```
1  function NNOuputLayerBackward ( FCNN *fcnn,Mat* RValue )
2   {
3       frame(NNOuputLayerBackward_2_tempMat,NNOuputLayerBackward_ActiPlusTrans
    ,return) and (
4       int return<==0 and skip;
5       if(fcnn->Layer[fcnn->HiddenLayerNum+1].AcitFuncNum=5 AND fcnn->
    LossFuncNum=1) then      // 步骤1.a
6       {
```

```
7           MatSub(&fcnn->Layer[fcnn->HiddenLayerNum+1].ActiMat,&fcnn->
    OnehotMat,&fcnn->Layer[fcnn->HiddenLayerNum+1].DeltaMat,RValue)
8       }
9       else        // 步骤1.b
10      {
11          Mat NNOuputLayerBackward_2_tempMat and skip;
12          MatCreate(&NNOuputLayerBackward_2_tempMat,fcnn->CurrentSampleNum,
    fcnn->Layer[fcnn->HiddenLayerNum+1].NeuronNum,RValue);
13          LossFunDerivation(&fcnn->Layer[fcnn->HiddenLayerNum+1].ActiMat,&
    NNOuputLayerBackward_2_tempMat,fcnn->OnehotMat,fcnn->LossFuncNum,RValue)
    ;
14          ActiFunDerivation(fcnn->Layer[fcnn->HiddenLayerNum+1].SumMat,&fcnn
    ->Layer[fcnn->HiddenLayerNum+1].ActiFunDerivationMat,fcnn->Layer[fcnn->
    HiddenLayerNum+1].AcitFuncNum,RValue);
15          MatProduct(&fcnn->Layer[fcnn->HiddenLayerNum+1].ActiMat,&fcnn->
    Layer[fcnn->HiddenLayerNum+1].ActiFunDerivationMat,&fcnn->Layer[fcnn->
    HiddenLayerNum+1].DeltaMat,RValue);
16          MatDelete(&NNOuputLayerBackward_2_tempMat)
17      };
18      Mat NNOuputLayerBackward_ActiPlusTrans and skip;
19      MatCreate(&NNOuputLayerBackward_ActiPlusTrans,fcnn->Layer[fcnn->
    HiddenLayerNum].NeuronNum+1,fcnn->CurrentSampleNum,RValue);
20      MatTrans(&fcnn->Layer[fcnn->HiddenLayerNum].ActiMatPlus,&
    NNOuputLayerBackward_ActiPlusTrans,RValue);
21      MatMul(&NNOuputLayerBackward_ActiPlusTrans,&fcnn->Layer[fcnn->
    HiddenLayerNum+1].DeltaMat,&fcnn->Layer[fcnn->HiddenLayerNum+1].
    NablaWbMat,RValue);
22      MatNumMul(1.0/ fcnn->CurrentSampleNum,&fcnn->Layer[fcnn->HiddenLayerNum
    +1].NablaWbMat,&fcnn->Layer[fcnn->HiddenLayerNum+1].NablaWbMat,RValue);
23      MatDelete(&NNOuputLayerBackward_ActiPlusTrans);
24      return<==1 and RValue:=NULL;
25      skip
26      )
27      };
```

□

6.3.6 优化算法构建

神经网络优化算法主要以梯度下降为核心，实现了小批量梯度下降优化算法，以及效果以及收敛速度更好的 Adam 优化算法。

表 6.15: 优化算法相关函数

编号	函数名	输入类型		输出类型	功能
1	MBGD	FCNN *	float	None	小批量梯度下降优化函数
2	SpaceCreateAdamPara	FCNN *	AdamPara *	None	创建 Adam 优化器参数空间
3	initAdam	FCNN *	AdamPara *	None	初始化 Adam 优化器参数
4	Adam	FCNN *	AdamPara *	None	Adam 优化算法

例 6.11 Adam 优化算法函数代码：

代码 6.26: 函数 4 Adam

```
function Adam ( FCNN *fcnn,AdamPara *adamPara )
{
    frame(Adam_i,Adam_temp$_1,Adam_temp$_2) and (
    int Adam_i<==1 and skip;

    while( (Adam_i<=fcnn->HiddenLayerNum+1) )
    {
        // 公式(4.1)
        MatNumMul(adamPara->beta1,&adamPara->v[Adam_i],&adamPara->v[Adam_i],RValue);
        MatNumMul(1-adamPara->beta1,&fcnn->Layer[Adam_i].NablaWbMat,&adamPara->hat_g[Adam_i],RValue);
        MatAdd(&adamPara->v[Adam_i],&adamPara->hat_g[Adam_i],&adamPara->v[Adam_i],RValue);

        // 公式(4.2)
        MatNumMul(adamPara->beta2,&adamPara->s[Adam_i],&adamPara->s[Adam_i],RValue);
        MatSquare(&adamPara->hat_g[Adam_i],&adamPara->hat_g[Adam_i],RValue);
        MatNumMul(1-adamPara->beta2,&adamPara->hat_g[Adam_i],&adamPara->hat_g[Adam_i],RValue);
```

```
17          MatAdd(&adamPara->s[Adam_i],&adamPara->hat_g[Adam_i],&adamPara->s[
    Adam_i],RValue);
18
19          // 公式(4.3)
20          int Adam_temp$_1 and skip;
21          Adam_temp$_1:=pow(adamPara->beta1,adamPara->time);
22          MatNumMul((1/ (1-Adam_temp$_1)),&adamPara->v[Adam_i],&adamPara->
    hat_v[Adam_i],RValue);
23
24          // 公式(4.4)
25          int Adam_temp$_2 and skip;
26          Adam_temp$_2:=pow(adamPara->beta2,adamPara->time);
27          MatNumMul((1/ (1-Adam_temp$_2)),&adamPara->s[Adam_i],&adamPara->
    hat_s[Adam_i],RValue);
28
29          // 公式(4.5)
30          MatNumMul(adamPara->eta,&adamPara->hat_v[Adam_i],&adamPara->hat_v[
    Adam_i],RValue);
31          MatSqrt(&adamPara->hat_s[Adam_i],&adamPara->hat_s[Adam_i],RValue);
32          MatNumAdd(adamPara->epsilon,&adamPara->hat_s[Adam_i],&adamPara->
    hat_s[Adam_i],RValue);
33          MatDiv(&adamPara->hat_v[Adam_i],&adamPara->hat_s[Adam_i],&adamPara
    ->hat_g[Adam_i],RValue);
34
35          // 公式(4.6)
36          MatSub(&fcnn->Layer[Adam_i].WeightBiasMat,&adamPara->hat_g[Adam_i
    ],&fcnn->Layer[Adam_i].WeightBiasMat,RValue);
37          Adam_i:=Adam_i+1
38
39        };
40        adamPara->time:=adamPara->time+1
41        )
42        };
```

6.3.7 神经网络训练构建

神经网络训练包含多次神经网络的单次训练，一次神经网络的单次训练包括：

1. 前向传播
2. 反向传播
3. 优化算法更新权值

在小批量优化算法当中，一个数据块的单次训练称为 iteration，一次训练集的所有数据块的一次训练称为 epoch。

神经网络训练伪码

for $epoch = 1, 2, \cdots, N\{$
    for $iteration = 1, 2, \cdots, M\{$
        神经网络前向传播求损失
        神经网络反向传播求梯度
        神经网络优化算法更新权值
    }
}

6.3.8 神经网络测试构建

神经网络测试构建是测试集神经网络一次前向传播得到损失，同时将输出层激活矩阵与相对应的标签值矩阵相比较最终求得准确率。

表 6.16: 神经网络测试构建相关函数

编号	函数名	输入类型		输出类型	功能
1	judge_max	float *	int	int	求得数组最大值索引
2	testAcc	FCNN	DataSet	float	求得准确率

例 6.12 求准确率函数代码：

代码 6.27: 函数 2 testAcc

```
1 function testAcc ( FCNN fcnn,DataSet dataset,float RValue )
2  {
```

```
3       frame(testAcc_MatCompare,testAcc_buf,testAcc_i,testAcc_correctNum,
    testAcc_1_temp$_1,return) and (
4       int return<==0 and skip;
5       Mat testAcc_MatCompare and skip;
6       char testAcc_buf[20] and skip;
7       int testAcc_i<==0 and skip;
8       MatCreate(&testAcc_MatCompare,dataset.TestLabel.row,1,RValue);
9       MatZeros(&testAcc_MatCompare,RValue);
10      testAcc_i:=0;
11
12      while( (testAcc_i<dataset.TestLabel.row) )
13      {
14          (testAcc_MatCompare.element[testAcc_i])[0]:=(float)judge_max(fcnn.
    Layer[fcnn.HiddenLayerNum+1].ActiMat.element[testAcc_i],fcnn.Layer[fcnn.
    HiddenLayerNum+1].ActiMat.col,RValue);        // 求得每个数据概率最大的索
    引
15          testAcc_i:=testAcc_i+1
16      };
17      int testAcc_correctNum<==0 and skip;
18      testAcc_i:=0;
19
20      while( (testAcc_i<dataset.TestLabel.row) )
21      {
22          int testAcc_1_temp$_1 and skip;
23          testAcc_1_temp$_1:=equal((testAcc_MatCompare.element[testAcc_i])
    [0],(dataset.TestLabel.element[testAcc_i])[0],RValue);      // 与原始标签
    数据作比较
24          if(testAcc_1_temp$_1) then
25          {
26              testAcc_correctNum:=testAcc_correctNum+1
27          }
28          else
29          {
30               skip
31          };
32          testAcc_i:=testAcc_i+1
33      };
```

```
34      return<==1 and RValue:=(float)testAcc_correctNum/ (float)dataset.
    TestLabel.row;          // 平均
35      skip
36      )
37      };
```

□

6.4 项目目录说明

- FCNNF_MC8.0_VS2015_WIN10 项目根文件夹
  - MC8.0 MSVL 编译项目文件夹
    - MC8.0.sln MSVL 编译项目 VS 工程文件
    - MSVLLIB MSVL 编译项目库文件夹
    - MC8.0 MSVL 编译项目主工程文件夹
      - _MSVInput.m MSVL 代码输入文件
      - external.c 外部调用声明文件
  - FCNNF C 语言项目文件夹
    - FCNNF.sln C 语言项目 VS 工程文件
    - FCNNF C 语言项目主文件夹
      - main.cpp C 语言项目主文件
  - DataSet 数据集文件夹
    - MinstHandWriting 手写字体识别数据文件夹

6.5 矩阵运算的优化

6.5.1 原始矩阵乘操作

6.5.2 寄存器优化矩阵乘操作

6.5.3 稀疏矩阵优化矩阵乘操作

6.5.4 多级缓存优化矩阵乘操作

PART III 第三部分

应用示例

第七章

Minst 手写字体识别

7.1 数据集介绍

7.1.1 数据集来源

MNIST 手写数字识别数据集来自美国国家标准与技术研究所 (National Institute of Standards and Technology,NIST)。这个数据集由 250 个不同人手写的数字构成, 其中 50% 来自高中生, 50% 来自美国人口普查局的工作人员。整个数据集由三部分构成，训练集有 60000 条数据，测试集有 10000 条数据, 共 70000 条数据。获取地址：`http://yann.lecun.com/exdb/mnist/`。

7.1.2 文件说明

目录说明

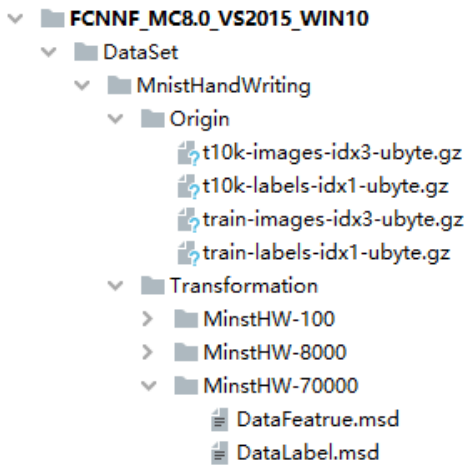

图 7.1: 数据目录树

- FCNNF_MC8.0_VS2015_WIN10 项目根文件夹
  - DataSet 数据集文件夹
    - MinstHandWriting 手写字体识别数据文件夹
      - Origin 原始数据文件夹
        * train-images-idx3-ubyte.gz 60000 训练数据特征文件
        * train-labels-idx1-ubyte.gz 60000 训练数据标签文件
        * t10k-images-idx3-ubyte.gz 10000 测试数据特征文件
        * t10k-labels-idx1-ubyte.gz 10000 测试数据标签文件
      - Transformation 改造后数据文件夹
        - MinstHW-100 100 条数据对文件夹
          * DataFeatrue.msd 数据特征文件
          * DataLabel.msd 数据标签文件
        - MinstHW-8000 8000 条数据对文件夹
          * DataFeatrue.msd 数据特征文件
          * DataLabel.msd 数据标签文件
        - MinstHW-70000 70000 条数据对文件夹
          * DataFeatrue.msd 数据特征文件
          * DataLabel.msd 数据标签文件

数据文件说明

原始数据集中有四个文件，将训练测试数据集分开，而在改造后的数据集中将其融合打乱，但是标签还是一一对应的。原始数据集存储的是二进制文件，而改造后的数据文件是形式是msd，可用文本编辑器打开。改造后的完整数据集是有 70000 条数据，此外还有利于调试的 8000 条以及 100 条数据可供选择。

代码 7.1: .\DataSet\MinstHandWriting\Transformation\MinstHW-70000\DataFeatrue.msd

```
...
0.0
0.0
0.011764706
0.07058824
0.07058824
0.07058824
0.49411765
0.53333336
0.6862745
```

```
0.101960786
0.6509804
...
```

代码 7.2: .\DataSet\MinstHandWriting\Transformation\MinstHW-70000\DataLabel.msd

```
5
0
4
1
9
2
1
3
1
4
...
```

7.1.3 数据可视化

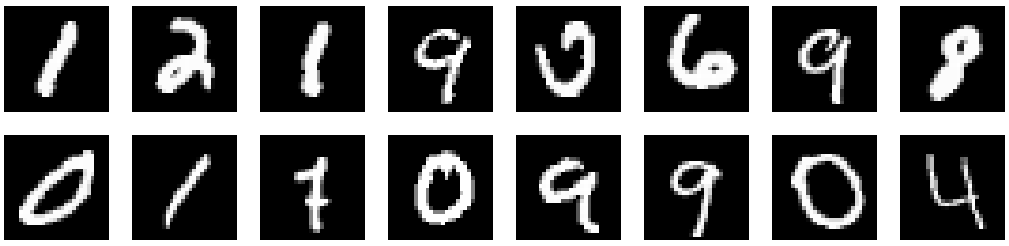

图 7.2: 数据可视化

图 (7.2) 中有许多手写的阿拉伯数字 (0-9)，我们要做的事情就是把这些手写的数字图像作为样本输入，让机器学习生成的模型能够自动地去判别当前这幅图像写的数字。

7.2 代码样例解析

7.2.1 数据录入

代码 7.3: 数据录入函数代码

```
struct _iobuf {
     char *_ptr and
```

```
3       int _cnt and
4       char *_base and
5       int _flag and
6       int _file and
7       int _charbuf and
8       int _bufsiz and
9       char *_tmpfname
10  };      // 文件结构体（必须）
11  float Xval[54880000]<==0} and skip;
12  float Yval[70000]<==0} and skip;
13  function MinstHWDataLoading (  )  // 将数据从文件中录入到一维数组中
14  {
15      frame(MinstHWDataLoading_buf,MinstHWDataLoading_fp,
    MinstHWDataLoading_fp2,MinstHWDataLoading_len$,MinstHWDataLoading_i,
    MinstHWDataLoading_j,MinstHWDataLoading_flag) and (
16      char MinstHWDataLoading_buf[16] and skip;
17      FILE *MinstHWDataLoading_fp<==NULL and skip;
18      FILE *MinstHWDataLoading_fp2<==NULL and skip;
19      int MinstHWDataLoading_len$ and skip;
20      int MinstHWDataLoading_i<==0 and skip;
21      int MinstHWDataLoading_j<==0 and skip;
22      char *MinstHWDataLoading_flag and skip;
23      MinstHWDataLoading_fp:=fopen("../../DataSet/MinstHandWriting/
    Transformation/MinstHW-70000/DataFeatrue.msd","r");
24      MinstHWDataLoading_flag:=fgets(MinstHWDataLoading_buf,16,
    MinstHWDataLoading_fp);
25      while( (MinstHWDataLoading_flag!=NULL) )
26      {
27          MinstHWDataLoading_len$:=strlen(MinstHWDataLoading_buf);
28          MinstHWDataLoading_buf[MinstHWDataLoading_len$-1]:='\0';
29          Xval[MinstHWDataLoading_i]:=(float)atof(MinstHWDataLoading_buf);
30          MinstHWDataLoading_i:=MinstHWDataLoading_i+1;
31          MinstHWDataLoading_flag:=fgets(MinstHWDataLoading_buf,16,
    MinstHWDataLoading_fp)
32      };
33      fclose(MinstHWDataLoading_fp) and skip;
34      MinstHWDataLoading_fp2:=fopen("../../DataSet/MinstHandWriting/
```

```
    Transformation/MinstHW-70000/DataLabel.msd","r");
35    MinstHWDataLoading_flag:=fgets(MinstHWDataLoading_buf,16,
    MinstHWDataLoading_fp2);
36    while( (MinstHWDataLoading_flag!=NULL) )
37    {
38        MinstHWDataLoading_len$:=strlen(MinstHWDataLoading_buf);
39        MinstHWDataLoading_buf[MinstHWDataLoading_len$-1]:='\0';
40        Yval[MinstHWDataLoading_j]:=(float)atof(MinstHWDataLoading_buf);
41        MinstHWDataLoading_j:=MinstHWDataLoading_j+1;
42        MinstHWDataLoading_flag:=fgets(MinstHWDataLoading_buf,16,
    MinstHWDataLoading_fp2)
43    };
44    fclose(MinstHWDataLoading_fp2) and skip
45    )
46    };
```

7.2.2 主函数

代码 7.4: 主函数代码

```
1  function main ( int  RValue )
2  {
3      frame(main_buf3,main_buf4,main_NueronNumArray,main_ActiFuncNumArray,
    main_buf1,main_buf2,main_userDefine,main_dataSet,main_fcnn,main_fclayer,
    main_adamPara,main_loss,main_losstest,main_trainOperationNum,main_i,
    main_j,main_temp$_1) and (
4      char main_buf3[20] and skip;
5      char main_buf4[20] and skip;
6      MinstHWDataLoading();
7      int main_NueronNumArray[4]<=={784,512,256,10} and skip;
8      int main_ActiFuncNumArray[4]<=={0,3,3,5} and skip;
9      char main_buf1[40] and skip;
10     char main_buf2[40] and skip;
11     Custom main_userDefine and skip;
12     DataSet main_dataSet and skip;
13     FCNN main_fcnn and skip;
14     FCLayer main_fclayer and skip;
```

```
15      InitCustom(&main_userDefine,RValue);
16      InitDataSet(&main_dataSet,RValue);
17      InitFCNN(&main_fcnn,RValue);
18      InitFCLayer(&main_fclayer,RValue);
19      main_userDefine.CompleteSampleNum:=70000;       // 完整数据集个数
20      main_userDefine.TrainSampleNum:=60000;      // 训练集个数
21      main_userDefine.TestSampleNum:=10000;       // 测试集个数
22      main_userDefine.SampleDimensionNum:=784;        // 单个数据的特征数
23      main_userDefine.HiddenLayerNum:=2;      // 隐藏层个数
24      main_userDefine.ClassificationNum:=10;      //  分类数
25      main_userDefine.LossFuncNum:=1;         // 损失函数 CE
26      main_userDefine.WeightInitWayNum:=3;        // 凯明初始化
27      main_userDefine.BatchSize:=200;         // batch 大小
28      main_userDefine.XValArray:=Xval;        // 总特征数据一维数组
29      main_userDefine.YValArray:=Yval;        // 总标签数据一维数组
30      main_userDefine.NeuronNumArray:=main_NueronNumArray;
31      main_userDefine.ActiFuncNumArray:=main_ActiFuncNumArray;
32      DumpCustom(main_userDefine,RValue);     // 标椎输出神经网络参数
33      LoadParaFromCustom(main_userDefine,&main_dataSet,&main_fcnn);       //
    参数录入到相关结构体
34      DatasetConstruction(main_userDefine,&main_dataSet);         // 数据集构
    建
35      CreateNNSpaceAndLoadinPara2FCLayer(&main_fcnn,main_userDefine,RValue);
          // 神经网络空间创建及神经网络层参数传入
36      NNWeightinit(&main_fcnn,RValue);        // 权值初始化
37      AdamPara main_adamPara and skip;
38      initAdam(main_fcnn,&main_adamPara);         // 初始化Adam参数
39      float main_loss<==0.0 and skip;
40      float main_losstest<==0.0 and skip;
41      int main_trainOperationNum<==40 and skip;
42      int main_i<==0 and skip;
43
44      while( (main_i<main_trainOperationNum) )        // 训练 epoch
45      {
46          int main_j<==0 and skip;
47
48          while( (main_j<main_dataSet.BatchNum) )         // 训练 iteration
```

```
49          {
50              main_loss:=NNforward(main_dataSet.BatchTrainFeature[main_j],
    main_dataSet.BatchTrainLabelOneHot[main_j],&main_fcnn,RValue);      // 前
    向传播求损失
51              NNBackward(&main_fcnn,RValue);      // 反向传播求梯度
52              Adam(&main_fcnn,&main_adamPara);
53              if((main_j+1) % 30=0) then
54              {
55                  output ("epoch ",main_i+1,"/",main_trainOperationNum,"
    iteration ",main_j+1,"/",main_dataSet.BatchNum," loss=",F2S(main_loss,
    main_buf1,RValue),"\n") and skip
56
57              }
58              else
59              {
60                   skip
61              };
62              main_j:=main_j+1
63
64          };
65          main_losstest:=NNforward(main_dataSet.TestFeature,main_dataSet.
    TestLabelOneHot,&main_fcnn,RValue);
66          float main_temp$_1 and skip;
67          main_temp$_1:=testAcc(main_fcnn,main_dataSet,RValue);
68          output ("====== epoch ",main_i+1,"/",main_trainOperationNum,"
    testloss=",F2S(main_losstest,main_buf2,RValue),"  acc=",F2S(main_temp$_1
    ,main_buf1,RValue)," ======\n") and skip;
69          main_i:=main_i+1
70
71      }
72      )
73  };
74   main(RValue)
75  )
```

7.2.3 输出

代码 7.5: 输出代码

```
$$$$$$$$$$$$$$$$$$$$$$$$$Path Number : 1
============================ Custom Dump ==============================
CompleteSampleNum:      70000
TrainSampleNum:         60000
TestSampleNum:          10000
SampleDimensionNum:     784
HiddenLayerNum:         2
WeightInitWayNum:       3
ClassificationNum:      10
LossFuncNum:            1
BatchSize:              200
XValArray:(length = 54880000)
YValArray:(length = 70000)
NeuronNumArray:
784             512             256             10
ActiFuncNumArray:(include output layer Acti-Function)
0               3               3               5
========================== Custom Dump finish ==========================
epoch 1/40 iteration 30/300 loss=0.2807
====== epoch 1/40  testloss=0.1498  acc=0.9539 ======
epoch 2/40 iteration 30/300 loss=0.1454
====== epoch 2/40  testloss=0.1161  acc=0.9652 ======
epoch 3/40 iteration 30/300 loss=0.1108
====== epoch 3/40  testloss=0.1010  acc=0.9701 ======
epoch 4/40 iteration 30/300 loss=0.0845
====== epoch 4/40  testloss=0.0929  acc=0.9718 ======
epoch 5/40 iteration 30/300 loss=0.0689
====== epoch 5/40  testloss=0.0864  acc=0.9734 ======
epoch 6/40 iteration 30/300 loss=0.0591
====== epoch 6/40  testloss=0.0824  acc=0.9743 ======
epoch 7/40 iteration 30/300 loss=0.0509
====== epoch 7/40  testloss=0.0791  acc=0.9757 ======
epoch 8/40 iteration 30/300 loss=0.0452
====== epoch 8/40  testloss=0.0770  acc=0.9763 ======
```

```
epoch 9/40 iteration 30/300 loss=0.0399
====== epoch 9/40  testloss=0.0751  acc=0.9768 ======
epoch 10/40 iteration 30/300 loss=0.0337
====== epoch 10/40  testloss=0.0737  acc=0.9774 ======
epoch 11/40 iteration 30/300 loss=0.0292
====== epoch 11/40  testloss=0.0727  acc=0.9777 ======
epoch 12/40 iteration 30/300 loss=0.0256
====== epoch 12/40  testloss=0.0719  acc=0.9782 ======
epoch 13/40 iteration 30/300 loss=0.0212
====== epoch 13/40  testloss=0.0714  acc=0.9779 ======
epoch 14/40 iteration 30/300 loss=0.0180
====== epoch 14/40  testloss=0.0708  acc=0.9784 ======
epoch 15/40 iteration 30/300 loss=0.0150
====== epoch 15/40  testloss=0.0706  acc=0.9784 ======
epoch 16/40 iteration 30/300 loss=0.0124
====== epoch 16/40  testloss=0.0706  acc=0.9784 ======
epoch 17/40 iteration 30/300 loss=0.0107
====== epoch 17/40  testloss=0.0704  acc=0.9793 ======
epoch 18/40 iteration 30/300 loss=0.0088
====== epoch 18/40  testloss=0.0706  acc=0.9791 ======
epoch 19/40 iteration 30/300 loss=0.0073
====== epoch 19/40  testloss=0.0706  acc=0.9796 ======
epoch 20/40 iteration 30/300 loss=0.0059
====== epoch 20/40  testloss=0.0713  acc=0.9796 ======
epoch 21/40 iteration 30/300 loss=0.0050
====== epoch 21/40  testloss=0.0719  acc=0.9798 ======
epoch 22/40 iteration 30/300 loss=0.0043
====== epoch 22/40  testloss=0.0723  acc=0.9798 ======
epoch 23/40 iteration 30/300 loss=0.0038
====== epoch 23/40  testloss=0.0727  acc=0.9802 ======
epoch 24/40 iteration 30/300 loss=0.0031
====== epoch 24/40  testloss=0.0734  acc=0.9805 ======
epoch 25/40 iteration 30/300 loss=0.0027
====== epoch 25/40  testloss=0.0742  acc=0.9807 ======
epoch 26/40 iteration 30/300 loss=0.0022
====== epoch 26/40  testloss=0.0748  acc=0.9807 ======
epoch 27/40 iteration 30/300 loss=0.0019
```

```
====== epoch 27/40  testloss=0.0757  acc=0.9810 ======
epoch 28/40 iteration 30/300 loss=0.0015
====== epoch 28/40  testloss=0.0763  acc=0.9812 ======
epoch 29/40 iteration 30/300 loss=0.0013
====== epoch 29/40  testloss=0.0772  acc=0.9812 ======
epoch 30/40 iteration 30/300 loss=0.0012
====== epoch 30/40  testloss=0.0781  acc=0.9810 ======
epoch 31/40 iteration 30/300 loss=0.0010
====== epoch 31/40  testloss=0.0791  acc=0.9809 ======
epoch 32/40 iteration 30/300 loss=0.0008
====== epoch 32/40  testloss=0.0801  acc=0.9812 ======
epoch 33/40 iteration 30/300 loss=0.0007
====== epoch 33/40  testloss=0.0811  acc=0.9815 ======
epoch 34/40 iteration 30/300 loss=0.0006
====== epoch 34/40  testloss=0.0821  acc=0.9814 ======
epoch 35/40 iteration 30/300 loss=0.0005
====== epoch 35/40  testloss=0.0831  acc=0.9817 ======
epoch 36/40 iteration 30/300 loss=0.0004
====== epoch 36/40  testloss=0.0843  acc=0.9817 ======
epoch 37/40 iteration 30/300 loss=0.0003
====== epoch 37/40  testloss=0.0854  acc=0.9815 ======
epoch 38/40 iteration 30/300 loss=0.0003
====== epoch 38/40  testloss=0.0866  acc=0.9815 ======
epoch 39/40 iteration 30/300 loss=0.0002
====== epoch 39/40  testloss=0.0876  acc=0.9819 ======
epoch 40/40 iteration 30/300 loss=0.0002
====== epoch 40/40  testloss=0.0888  acc=0.9819 ======

The run time is: 5687188ms  94m 1.58h
```

参考文献

[1] Amari, S. Backpropagation and stochastic gradient descent method. *Neurocomputing 5*, 3 (1993), 185–196.

[2] Andrew, A. M. *The Handbook of Brain Theory and Neural Networks*, edited by michael a. arbib, MIT press, cambridge, mass. and london, england, 1998, xv+1118 pp, ISBN 0-262-51102-9, paperback, £49.95 (cloth-bound version also available, originally published in 1995, ISBN 262-01148-4, £147.95). *Robotica 17*, 5 (1999), 571–572.

[3] Cajal, R. *Les nouvelles idées sur la structure du système nerveux chez l'homme et chez les vertébrés' ('New ideas on the fine anatomy of the nerve centres)*. French, 1894.

[4] Chen, Y., Chi, Y., Fan, J., and Ma, C. Gradient descent with random initialization: fast global convergence for nonconvex phase retrieval. *Math. Program. 176*, 1-2 (2019), 5–37.

[5] Cybenko, G. Approximation by superpositions of a sigmoidal function. *MCSS 2*, 4 (1989), 303–314.

[6] David E. Rumelhart, Geoffrey E. Hinton, R. J. W. Learning representations by back-propagating errors. *nature 323* (1986), 533⁻536.

[7] Emad, N., Hamdi-Larbi, O., and Mahjoub, Z. On sparse matrix-vector product optimization. In *AICCSA* (2005), IEEE Computer Society, p. 23.

[8] Glorot, X., and Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics* (Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010), Y. W. Teh and M. Titterington, Eds., vol. 9 of *Proceedings of Machine Learning Research*, PMLR, pp. 249–256.

[9] Goodfellow, I. J., Bengio, Y., and Courville, A. C. *Deep Learning*. Adaptive computation and machine learning. MIT Press, 2016.

[10] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *CVPR* (2016), IEEE Computer Society, pp. 770–778.

[11] Hinton, G. E., Osindero, S., and Teh, Y. W. A fast learning algorithm for deep belief nets. *Neural Computation 18*, 7 (2006), 1527–1554.

[12] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization. In *ICLR (Poster)* (2015).

[13] Kurbiel, T., and Khaleghian, S. Training of deep neural networks based on distance measures using rmsprop. *CoRR abs/1708.01911* (2017).

[14] Maas, A. L., Hannun, A. Y., and Ng, A. Y. Rectifier nonlinearities improve neural network acoustic models. In *in ICML Workshop on Deep Learning for Audio, Speech and Language Processing* (2013).

[15] McCulloch, W. S., and Pitts, W. H. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics 5* (1943), 115–133.

[16] Minsky, M. L. *Perceptrons*. 1969.

[17] Neelakanta, P. S. Csiszar's generalized error measures for gradient-descent-based optimizations in neural networks using the backpropagation algorithm. *Connect. Sci. 8*, 1 (1996), 79–114.

[18] Pang, T., Xu, K., Dong, Y., Du, C., Chen, N., and Zhu, J. Rethinking softmax cross-entropy loss for adversarial robustness. *CoRR abs/1905.10626* (2019).

[19] Prieto, A., Prieto, B., Ortigosa, E. M., Ros, E., Pelayo, F. J., Ortega, J., and Rojas, I. Neural networks: An overview of early research, current frameworks and new challenges. *Neurocomputing 214* (2016), 242–268.

[20] Qian, N. On the momentum term in gradient descent learning algorithms. *Neural Networks 12*, 1 (1999), 145–151.

[21] Rodríguez, P., Bautista, M. Á., Gonzàlez, J., and Escalera, S. Beyond one-hot encoding: Lower dimensional target embedding. *Image Vis. Comput. 75* (2018), 21–31.

[22] Rosenblatt, F. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review* (1958), 65–386.

[23] Schiffmann, W., and Geffers, H. W. Adaptive control of dynamic systems by back propagation networks. *Neural Networks 6*, 4 (1993), 517–524.

[24] Wilson, D. R., and Martinez, T. R. The general inefficiency of batch training for gradient descent learning. *Neural Networks 16*, 10 (2003), 1429–1451.