合肥工業大學

Hefei University of Technology

数据结构课程设计：BP 神经网络

Data Structure Course Design: BP Neural Network

课　　程：数据结构

学生姓名：

学　　号：

专业班级：

课设题目：BP 神经网络

指导教师：

完成时间：2025 年 5 月

目 录

1 课程设计题目

本次的数据结构我选择的题目如下:

利用 C++ 语言实现 BP 神经网络，并利用 BP 神经网络解决螨虫分类问题:

蠓虫分类问题: 对两种蠓虫（A 与 B）进行鉴别，依据的资料是触角和翅膀的长度，已知了 9 支 Af 和 6 支 Apf 的数据如下: A: (1.24,1.27)，(1.36,1.74)，(1.38,1.64) ，(1.38,1.82) ，(1.38,1.90) ，(1.40,1.70) ，(1.48,1.82) ，(1.54,1.82) ，(1.56,2.08). B: (1.14,1.82)，(1.18,1.96)，(1.20,1.86)，(1.26,2.00)，(1.28,2.00)，(1.30,1.96). 要求:

- (1) 阐述 BP 神经网络的结构构成及数学原理;
- (2) 利用 C++ 实现 BP 神经网络;
- (3) 利用 BP 神经网络实现螨虫分类

BP 神经网络 (Backpropagation Neural Network, BPNN) 是一种按照误差逆向传播算法训练的多层前馈神经网络，是应用最广泛的神经网络模型之一。其核心思想是通过反向传播 (Backpropagation) 算法来调整网络中神经元之间的连接权重，以最小化网络输出与期望输出之间的误差。

本次课设我选择 BP 神经网络作为本次课程设计的题目，主要有以下几点考虑:

- 首先，我对人工智能领域抱有浓厚的兴趣，并且已经加入了学校的相关实验室进行学习和探索，希望通过这个项目进一步加深对神经网络的理解。
- 其次，BP 神经网络与我的专业 (智能科学与技术) 紧密相关，是该领域内一个基础且重要的模型。在深度学习领域，BP 神经网络扮演着至关重要的基础角色。后面我们学习的更为复杂和强大的深度学习架构，例如卷积神经网络（CNN)、循环神经网络（RNN) 以及 Transformer 等，其训练过程中的核心机制——梯度下降和误差反向传播，都源于或借鉴了 BP 神经网络的基本原理。因此，深刻理解 BP 神经网络的结构、工作方式及其训练算法，是学习和掌握现代深度学习技术的关键一步。
- 最后，在选题时，我对比了几个备选方案。BP 神经网络的实现相对而言更为经典和直观，不像卷积神经网络（CNN）或生成对抗网络（GAN）那样在结构和代码实现上更为复杂（特别是 GAN，数学原理上采用了博弈论的概念，涉及到生成器和判别器之间的对抗训练，这使得其实现和调试都极其困难，代码量和难度甚至远高于前一个题目

CNN)。因此，我认为这是一个难度适中，既能锻炼能力又不至于难以完成的课题，具有一定的挑战性，同时也保证了项目的可完成性。

出于兴趣，在实现了除题目要求的螨虫分类功能外，我还使用 BPNN 实现了基于 MNIST 数据集的手写数字识别功能，并且为这两个功能使用 Qt 集成了图形化界面，并且使用了 Qt Quick 和 QML 使得项目更为美观简洁，使得整个实验过程更加直观：

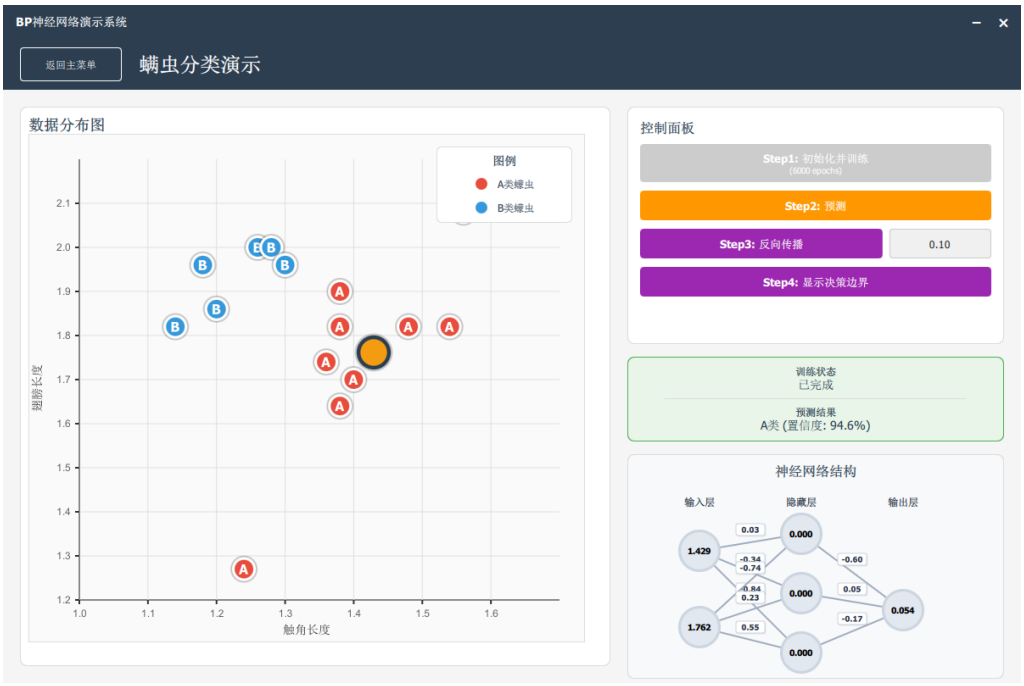

图 1.1　螨虫分类功能界面

图 1.2　MNIST 手写数字识别功能界面

2 BP 神经网络的结构构成及数学原理推导

2.1 BP 神经网络及其结构

BP 神经网络（Backpropagation Neural Network）是一种通过误差反向传播算法进行训练的多层前馈神经网络。它是当前应用最为广泛的神经网络模型之一，尤其在模式识别、函数逼近、数据挖掘等领域取得了显著成就。

在现实生活中，我们常常会碰到很多难以通过明确地编程来解决的问题。比如说，如何判断一张图片是猫还是狗，猫和狗具有很多特征不同，而这些特征对应的图像组合千变万化，难以用简单的规则来描述。如果我们想要直接编写一个程序来解决这种问题，我们会发现很难实现。在这种情况下，机器学习 (深度学习) 则提供了一种新的思路。它通过学习大量的样本数据，自动提取特征并建立模型，从而实现对未知数据的预测和分类。

从机器学习（特别是深度学习）的根本目标来看，许多问题可以归结为寻找一个最优的函数 f^*，这个函数能够将输入数据映射到期望的输出。比如说，我现在需要对猫和狗的图片进行分类，或者者预测某个房屋的价格，这些任务都可以看作是寻找一个函数 f^*，使得对于每个输入数据 x，我们能够得到一个准确的输出 y。在高等数学中，我们学过泰勒展开、傅里叶变换、拉格朗日插值等方法，这些方法依赖特定点的导数、频率、或者多项式系数来构建函数的近似形式，而我们现在所描述的神经网络，则是通过组合许多简单的非线性函数来逼近这个复杂的函数 f^*。

一个典型的 BP 神经网络结构主要包括以下几个部分：

- **输入层 (Input Layer)**：负责接收外部输入的数据。输入层神经元的数量通常与输入数据的特征维度相对应。例如，在螨虫分类问题中，输入是触角和翅膀的长度，所以输入层有两个神经元。
- **隐藏层 (Hidden Layer(s))**：位于输入层和输出层之间，可以有一层或多层。隐藏层是神经网络进行特征提取和抽象的关键部分。神经元的数量和层数是超参数，需要根据问题的复杂度和数据量进行设计。隐藏层通过非线性激活函数引入非线性变换能力，使得网络能够学习复杂的模式。
- **输出层 (Output Layer)**：负责输出网络的预测结果。输出层神经元的数量和激活函数的选择取决于具体的任务。例如，在二分类问题中，输出层通常有一个神经元（使用 Sigmoid 激活函数输出概率）；在多分类问题中，输出层神经元数量等于类别数（通常

使用 Softmax 激活函数输出各类别的概率)。

- **权重 (Weights)**：表示神经元之间的连接强度。每个连接都有一个关联的权重值。在训练过程中，神经网络通过调整这些权重来最小化预测误差。
- **偏置 (Biases)**：每个神经元（通常除了输入层）还会有一个偏置项。偏置项允许激活函数在输入为零时也能被激活，增加了模型的灵活性。

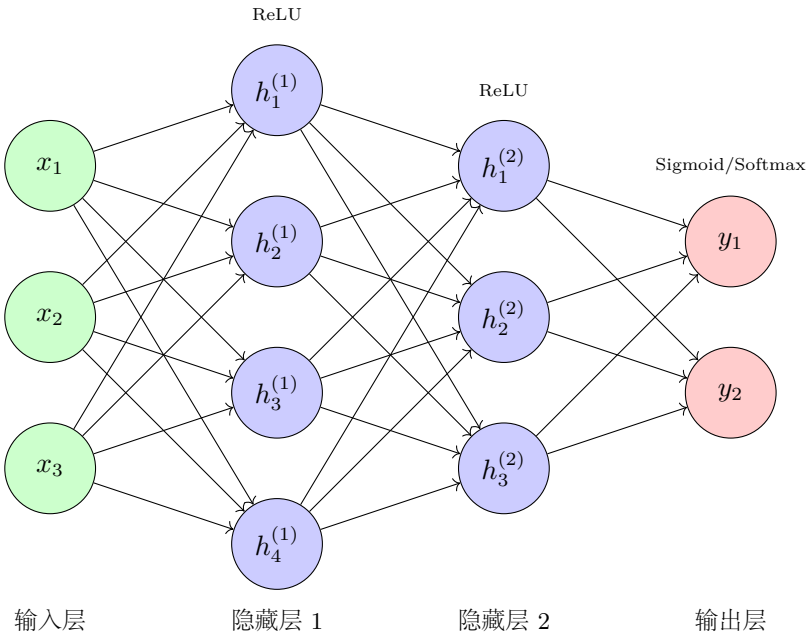

图 2.1　BP 神经网络典型结构示意图

神经元之间通过带权重的连接进行信息传递。在前向传播过程中，每个神经元接收来自前一层神经元的加权输入，加上偏置后，通过激活函数处理，得到该神经元的输出，并传递给下一层。最后，输出层的神经元将所有前面层的输出进行处理，生成最终的预测结果。如果网络本身的参数（权重和偏置）设置得当，网络就能有效地预测出我们想要的结果。

2.2 激活函数

激活函数（Activation Function）是神经网络中至关重要的组成部分。如果缺少激活函数，神经网络的每一层都只是在进行线性变换（矩阵乘法和加法)。无论网络有多少层，其最终的输出都将是输入数据的线性组合。这样的网络，即使层数再深，其表达能力也等同于一个单层的线性模型，无法学习和表示复杂数据中的非线性关系，也就无法解决现实世界中绝大多数的复杂问题。

比如说对于简单的 XOR 问题，当 x=(0,0)、(0,1)、(1,0)、(1,1) 时，输出 y 分别为 0、1、

1、0。我们可以发现，这个问题无法用一个线性函数来解决，或者说无法找出一个线性分割平面将这四个点分开。

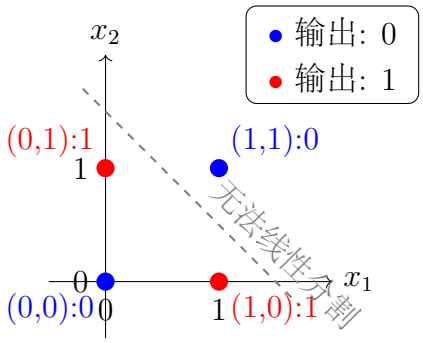

图 2.2　XOR 问题的线性不可分性示例

这时，我们可以尝试引入激活函数，使得网络具有非线性关系。激活函数的作用是对神经元的输入进行非线性变换，使得网络能够学习和表示复杂的非线性关系。通过在每个神经元上应用激活函数，网络可以将输入数据映射到更高维度的特征空间，从而实现对复杂模式的学习：

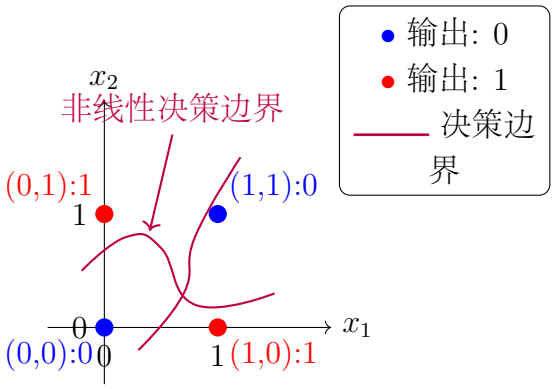

图 2.3　XOR 问题的非线性决策边界

这样的非线性分割是单层线性模型无法实现的，这也充分说明了激活函数在神经网络中的重要作用。

以下介绍几种常见的激活函数：

Sigmoid 函数：*Sigmoid* 函数是一个在生物学中常见的 S 型函数，也称为 S 型生长曲线。在信息科学中，由于其单增以及反函数单增等性质，Sigmoid 函数常被用作神经网络的阈值函数，将变量映射到 0,1 之间，公式如下：

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

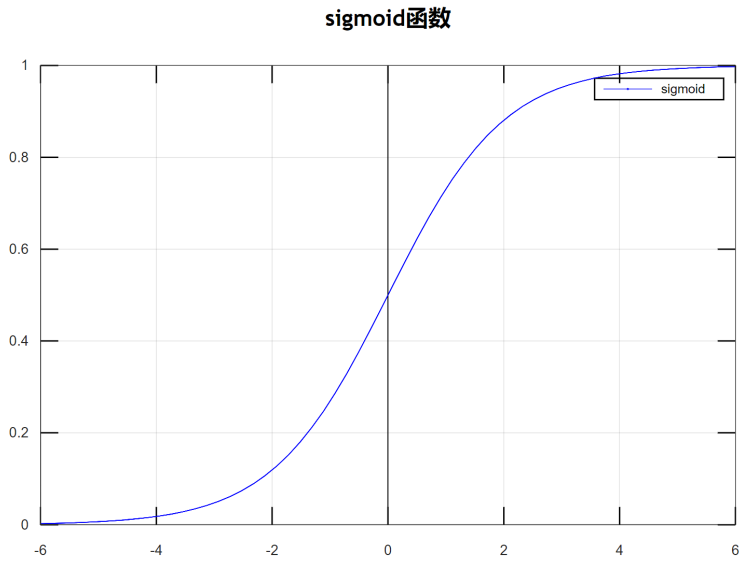

图 2.4　Sigmoid 函数图像

Sigmoid 函数的优点是输出范围在 0 到 1 之间，适合用于二分类问题的输出层。然而，它也有一些缺点，如梯度消失问题（当输入值过大或过小时，梯度接近于 0，导致学习速度变慢）和非零中心化（输出不是以 0 为中心，可能导致梯度下降时更新方向不一致)。

ReLU 函数：*ReLU*（Rectified Linear Unit）函数是目前最常用的激活函数之一。它的定义非常简单，对于输入值小于 0 时输出 0，大于等于 0 时输出该值本身，公式如下：

$$f(x) = \max(0, x)$$

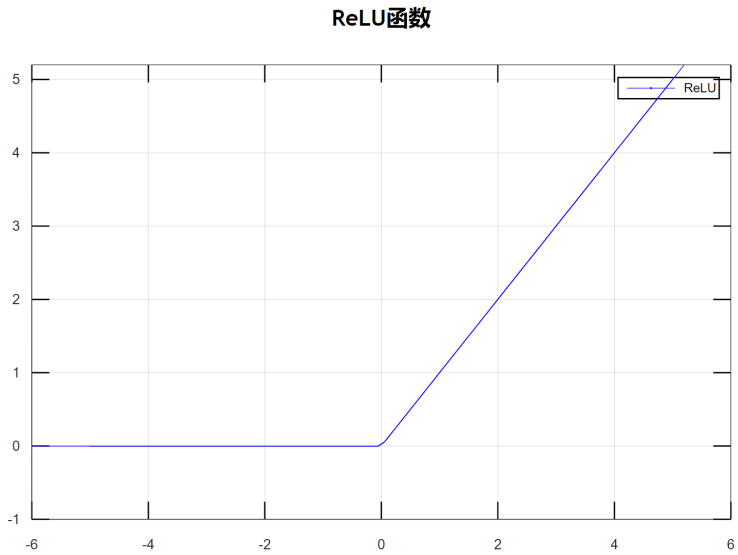

图 2.5　ReLU 函数图像

ReLU 函数的优点是计算简单，收敛速度快，并且在正区间具有线性特性，能够有效缓

解梯度消失问题。然而，它也有一个缺点，即当输入值小于 0 时，梯度为 0，这可能导致某些神经元在训练过程中“死亡”，即永远不会被激活。

$Tanh$ 函数：$Tanh$ 函数是双曲正切函数，其输出范围在-1 到 1 之间。它的公式如下：

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

$Tanh$ 函数的图像呈现出 S 型曲线，类似于 $Sigmoid$ 函数，但其输出是以 0 为中心的，这有助于缓解梯度消失问题。

$Softmax$ 函数：$Softmax$ 函数通常用于多分类问题的输出层。它将输入向量转换为一个概率分布，使得所有输出值的和为 1。其公式如下：

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$$

在实际应用中，选择合适的激活函数对于神经网络的性能至关重要。通常情况下，隐藏层使用 $ReLU$ 等非线性激活函数，而输出层根据具体任务选择 $Sigmoid$（二分类）或 $Softmax$（多分类）等激活函数。

2.3 符号约定

在数学公式推导中，为了简化表达和提高可读性，我们需要对一些符号进行约定：

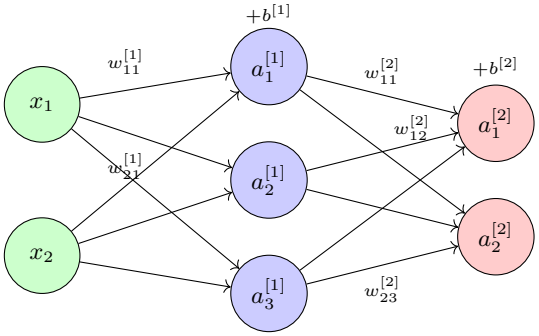

图 2.6　神经网络权重符号示意图

$w_{jk}^{[l]}$ 表示从网络第 $(l-1)^{th}$ 层第 k^{th} 个神经元指向第 l^{th} 层第 j^{th} 个神经元的连接权重，同时也是第 l 层权重矩阵第 j 行第 k 列的元素。例如，上图中 $w_{21}^{[1]}$ ，第 0 层第 1 个神经元指向第 1 层第 2 个神经元的权重，也就是第 1 层权重矩阵第 2 行第 1 列的元素。同理，使用 $b_j^{[l]}$ 表示第 l^{th} 层第 j^{th} 个神经元的偏置，同时也是第 l 层偏置向量的第 j 个元素。使用

$z_j^{[l]}$ 表示第 l^{th} 层第 j^{th} 个神经元的线性结果，使用 $a_j^{[l]}$ 来表示第 l^{th} 层第 j^{th} 个神经元的激活函数输出。其中，激活函数使用符号 σ 表示，第 l^{th} 层中第 j^{th} 个神经元的激活为:

$$a_j^{[l]} = \sigma(z_j^{[l]}) = \sigma\left(\sum_k w_{jk}^{[l]} a_k^{[l-1]} + b_j^{[l]}\right)$$

$w^{[l]}$ 表示第 l 层的权重矩阵，$b^{[l]}$ 表示第 l 层的偏置向量，$a^{[l]}$ 表示第 l 层的神经元向量。

2.4 前向传播算法

前向传播算法（Forward Propagation Algorithm）是神经网络从输入数据计算输出结果的过程。它描述了信息如何从输入层开始，逐层向前传递，经过隐藏层，最终到达输出层，并产生预测值的过程。

在前向传播过程中，每一层的神经元都会接收来自前一层神经元的输出（或者是原始输入数据，对于第一层隐藏层而言)。对于第 l 层的第 j 个神经元，其计算过程如下:

1. **计算加权和**：首先，将前一层（第 $l-1$ 层）所有神经元的激活值 $a_k^{[l-1]}$ 与相应的连接权重 $w_{jk}^{[l]}$ 相乘，然后将这些乘积加起来，并加上当前神经元的偏置项 $b_j^{[l]}$。这个结果称为净输入或线性组合，用 $z_j^{[l]}$ 表示:

$$z_j^{[l]} = \sum_k (w_{jk}^{[l]} a_k^{[l-1]}) + b_j^{[l]}$$

其中，k 遍历第 $l-1$ 层的所有神经元。我们还可以使用矩阵形式来表示这个计算过程:

$$z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$$

2. **应用激活函数**：然后，将计算得到的线性结果 $z_j^{[l]}$ 输入到该神经元的激活函数 σ 中，得到该神经元的输出（或称为激活值）$a_j^{[l]}$:

$$a_j^{[l]} = \sigma(z_j^{[l]})$$

这里也可以使用矩阵形式来表示:

$$a^{[l]} = \sigma(z^{[l]})$$

这个过程从输入层开始（通常将输入数据视为第 0 层的激活值，即 $a^{[0]} = x$)，逐层向前计算，直到输出层。

整个前向传播过程可以概括为:

1. 将输入样本 x 提供给输入层，即 $a^{[0]} = x$。
2. 对于网络中的每一层 l (从第一层隐藏层到输出层):
   - 计算该层的线性组合 $z^{[l]} = w^{[l]}a^{[l-1]} + b^{[l]}$。
   - 计算该层的激活输出 $a^{[l]} = \sigma(z^{[l]})$。
3. 最后一层（输出层）的激活输出 $a^{[L]}$ (假设网络有 L 层) 即为神经网络对输入样本 x 的预测结果 $\hat{y}$。

2.5 损失函数与梯度下降

在神经网络的训练过程中，我们需要一个衡量标准来评估当前网络的预测结果与真实目标之间的差距。这个标准就是损失函数（Loss Function），有时也称为代价函数（Cost Function）或目标函数（Objective Function)。损失函数输出一个标量值，该值越大，表示网络预测的误差越大，模型性能越差；反之，该值越小，表示模型性能越好。训练神经网络的目标就是通过调整网络的权重和偏置，来最小化这个损失函数的值。

对于一个训练样本 (x, y)，其中 x 是输入，y 是真实的标签，神经网络的输出为 $\hat{y} = f(x; W, b)$，其中 W 和 b 分别代表网络的权重和偏置。损失函数 $L(\hat{y}, y)$ 用来量化预测值 $\hat{y}$ 和真实值 y 之间的差异。

这里介绍两个常用的损失函数，也是我们本次课程设计会使用到的两个损失函数:

- **均方误差 (Mean Squared Error, MSE)**：常用于回归问题。对于单个样本，其定义为:

  $$L(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$$

  对于包含 N 个样本的整个训练集，均方误差通常定义为所有样本损失的平均值:

  $$J(W, b) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}(\hat{y}^{(i)} - y^{(i)})^2$$

  这里的 $\frac{1}{2}$ 是为了后续求导方便而添加的常数因子。

- **交叉熵损失 (Cross-Entropy Loss)**：常用于分类问题。对于二分类问题，如果真实标签 $y \in \{0, 1\}$，预测概率 $\hat{y} \in [0, 1]$，则交叉熵损失为:

  $$L(\hat{y}, y) = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})]$$

对于多分类问题(例如 K 个类别), 如果真实标签是 one-hot 编码向量 $y = [y_1, y_2, ..., y_K]$, 预测概率向量为 $\hat{y} = [\hat{y}_1, \hat{y}_2, ..., \hat{y}_K]$, 则交叉熵损失为:

$$L(\hat{y}, y) = -\sum_{k=1}^{K} y_k \log(\hat{y}_k)$$

确定了损失函数后, 接下来的目标就是找到一组最优的权重 W 和偏置 b, 使得损失函数 $J(W, b)$ 的值最小。梯度下降（Gradient Descent）是最常用的优化算法之一。

梯度下降法的核心思想是: 沿着损失函数梯度的反方向调整参数, 可以使得损失函数的值下降最快。梯度是一个向量, 指向函数值增加最快的方向。因此, 梯度的反方向就是函数值减小最快的方向。

假设我们当前的参数是 θ (代表所有的权重 W 和偏置 b), 损失函数为 $J(\theta)$。梯度下降的更新规则如下:

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \nabla_\theta J(\theta)$$

其中:

- θ_{new} 是更新后的参数。
- θ_{old} 是当前的参数。
- η (eta) 是学习率（Learning Rate), 一个正的标量, 控制每次更新的步长。学习率的选择非常重要: 如果太小, 收敛速度会很慢; 如果太大, 可能会导致在最小值附近震荡甚至发散。
- $\nabla_\theta J(\theta)$ 是损失函数 $J(\theta)$ 关于参数 θ 的梯度。它表示损失函数在当前参数点 θ 处的变化率最大的方向。对于神经网络中的每个权重 $w_{jk}^{[l]}$ 和偏置 $b_j^{[l]}$, 我们需要计算它们各自的偏导数:

$$\frac{\partial J}{\partial w_{jk}^{[l]}} \quad \text{和} \quad \frac{\partial J}{\partial b_j^{[l]}}$$

梯度下降是一个迭代的过程。在每次迭代中, 我们首先计算损失函数关于当前参数的梯度, 然后根据上述更新规则更新参数。这个过程会一直重复, 直到损失函数的值收敛到某个可接受的最小值, 或者达到预设的迭代次数。

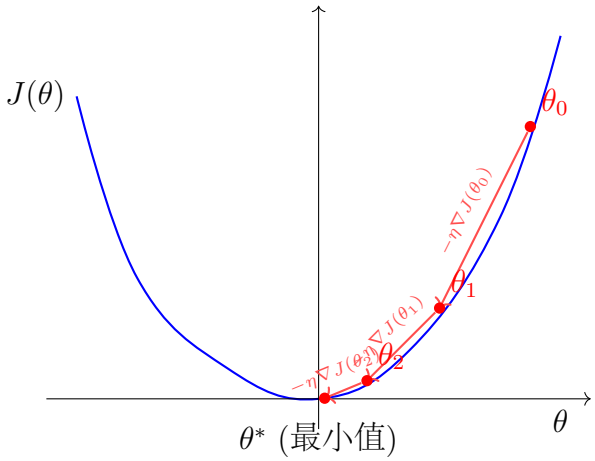

图 2.7　梯度下降示意图

神经网络实际上也是一个复杂的函数，损失函数 J 是关于所有权重和偏置的函数。通过梯度下降法，我们可以找到一组最优的参数，使得损失函数达到最小值，从而使得神经网络在训练集上表现良好。

2.6 反向传播算法

BP 神经网络的核心是反向传播算法（Backpropagation Algorithm），在这里，我们先不给出反向传播算法的定义和公式，而是从要解决的问题入手，逐步推导出反向传播算法的公式。

首先，我们的目标是找到一组最优的权重和偏置，使得损失函数 $J(W,b)$ 达到最小值。

优化目标:

设神经网络的所有参数为 $\theta = \{W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, ..., W^{[L]}, b^{[L]}\}$，其中 L 是网络的总层数。我们的优化目标是:

$$\theta^* = \arg\min_{\theta} J(\theta)$$

为了使用梯度下降法，我们需要计算损失函数关于每个参数的偏导数:

$$\frac{\partial J}{\partial W^{[l]}} \quad 和 \quad \frac{\partial J}{\partial b^{[l]}} \quad 对于所有层 \quad l = 1, 2, ..., L$$

更具体地，对于第 l 层的每个权重 $w_{jk}^{[l]}$ 和偏置 $b_j^{[l]}$，我们需要计算:

$$\frac{\partial J}{\partial w_{jk}^{[l]}} \quad 和 \quad \frac{\partial J}{\partial b_j^{[l]}}$$

然而，如果我们尝试直接计算这些偏导数，会发现这是一个非常复杂的过程。以一个简单的三层网络为例，损失函数 J 的计算涉及多层的复合函数运算：

$$J = J(a^{[3]}, y) \quad \text{其中} \quad a^{[3]} = \sigma(z^{[3]}) = \sigma(W^{[3]}a^{[2]} + b^{[3]})$$

而 $a^{[2]}$ 又依赖于第二层的参数：

$$a^{[2]} = \sigma(z^{[2]}) = \sigma(W^{[2]}a^{[1]} + b^{[2]})$$

这样一层层嵌套下去，直接求偏导会导致表达式极其复杂，并且计算效率极低。更重要的是，这种直接计算方法会产生大量重复计算，因为不同参数的偏导数之间存在公共的子表达式。

反向传播的基本思想:

反向传播算法利用链式法则，通过从输出层开始，逐层向前（反向）计算偏导数，避免了重复计算，大大提高了效率。

让我们从一个具体的例子开始，考虑一个三层网络（一个隐藏层），并逐步推导梯度计算的过程：

对于输出层（第 3 层），我们首先计算损失函数关于输出层激活值的偏导数。以均方误差为例：

$$J = \frac{1}{2}(a^{[3]} - y)^2$$

则有：

$$\frac{\partial J}{\partial a^{[3]}} = a^{[3]} - y$$

接下来，利用链式法则计算关于 $z^{[3]}$ 的偏导数：

$$\frac{\partial J}{\partial z^{[3]}} = \frac{\partial J}{\partial a^{[3]}} \cdot \frac{\partial a^{[3]}}{\partial z^{[3]}} = (a^{[3]} - y) \cdot \sigma'(z^{[3]})$$

最后，计算关于权重和偏置的偏导数：

$$\frac{\partial J}{\partial W^{[3]}} = \frac{\partial J}{\partial z^{[3]}} \cdot \frac{\partial z^{[3]}}{\partial W^{[3]}} = \frac{\partial J}{\partial z^{[3]}} \cdot (a^{[2]})^T$$

$$\frac{\partial J}{\partial b^{[3]}} = \frac{\partial J}{\partial z^{[3]}}$$

对于第 2 层（隐藏层），情况变得稍微复杂，因为第 2 层的激活值 $a^{[2]}$ 会影响第 3 层的输入 $z^{[3]}$，从而间接影响损失函数。

首先计算关于 $a^{[2]}$ 的偏导数：

$$\frac{\partial J}{\partial a^{[2]}} = \frac{\partial J}{\partial z^{[3]}} \cdot \frac{\partial z^{[3]}}{\partial a^{[2]}} = \frac{\partial J}{\partial z^{[3]}} \cdot (W^{[3]})^T$$

然后计算关于 $z^{[2]}$ 的偏导数：

$$\frac{\partial J}{\partial z^{[2]}} = \frac{\partial J}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} = \frac{\partial J}{\partial a^{[2]}} \cdot \sigma'(z^{[2]})$$

最后计算关于权重和偏置的偏导数：

$$\frac{\partial J}{\partial W^{[2]}} = \frac{\partial J}{\partial z^{[2]}} \cdot (a^{[1]})^T$$

$$\frac{\partial J}{\partial b^{[2]}} = \frac{\partial J}{\partial z^{[2]}}$$

对于第 1 层，我们继续使用相同的模式：

$$\frac{\partial J}{\partial a^{[1]}} = \frac{\partial J}{\partial z^{[2]}} \cdot (W^{[2]})^T$$

$$\frac{\partial J}{\partial z^{[1]}} = \frac{\partial J}{\partial a^{[1]}} \cdot \sigma'(z^{[1]})$$

$$\frac{\partial J}{\partial W^{[1]}} = \frac{\partial J}{\partial z^{[1]}} \cdot (a^{[0]})^T = \frac{\partial J}{\partial z^{[1]}} \cdot x^T$$

$$\frac{\partial J}{\partial b^{[1]}} = \frac{\partial J}{\partial z^{[1]}}$$

反向传播的递推关系：

通过观察上述计算过程，我们可以发现一个清晰的递推模式。定义误差项 $\delta^{[l]}$：

$$\delta^{[l]} = \frac{\partial J}{\partial z^{[l]}}$$

则反向传播的递推关系可以表示为：

1. 输出层误差：

$$\delta^{[L]} = \frac{\partial J}{\partial a^{[L]}} \cdot \sigma'(z^{[L]})$$

2. 隐藏层误差递推：

$$\delta^{[l]} = ((W^{[l+1]})^T \delta^{[l+1]}) \cdot \sigma'(z^{[l]})$$

3. 权重梯度：

$$\frac{\partial J}{\partial W^{[l]}} = \delta^{[l]} (a^{[l-1]})^T$$

4. 偏置梯度：

$$\frac{\partial J}{\partial b^{[l]}} = \delta^{[l]}$$

这就是著名的反向传播算法的核心公式。算法的名称" 反向传播" 正是因为误差信息从输出层开始，逐层向前（反向）传播，每一层都利用后一层的误差信息来计算自己的梯度。

基于以上公式，我们便可以从最后一层开始，逐层向前计算每一层的误差和梯度，每次都会使用前一层的结果来计算当前层的梯度，极大地减少了重复计算的开销，最终得到所有参数的梯度。

反向传播算法步骤:

完整的反向传播算法可以总结为以下步骤:

1. **前向传播**：计算网络的输出和所有中间层的激活值。
2. **计算输出层误差**：$\delta^{[L]} = \frac{\partial J}{\partial a^{[L]}} \cdot \sigma'(z^{[L]})$
3. **反向传播误差**：对于 $l = L-1, L-2, ..., 1$，计算 $\delta^{[l]} = ((W^{[l+1]})^T \delta^{[l+1]}) \cdot \sigma'(z^{[l]})$
4. **计算梯度**：对于所有层 l，计算 $\frac{\partial J}{\partial W^{[l]}} = \delta^{[l]} (a^{[l-1]})^T$ 和 $\frac{\partial J}{\partial b^{[l]}} = \delta^{[l]}$
5. **更新参数**：使用梯度下降更新所有权重和偏置。

以上，我们完成了反向传播公式的推导，这是 BP 神经网络的核心算法。在了解了 BP 神经网络的基本原理和反向传播算法后，我们可以开始使用 C++ 实现一个简单的 BP 神经网络。

3 BP 神经网络的 C++ 实现

3.1 系统架构设计

在实现 BP 神经网络之前，我们首先需要设计一个合理的系统架构。整个系统采用面向对象的设计模式，将不同的功能模块封装到独立的类中，以提高代码的可维护性和扩展性。

系统的核心架构包含以下几个主要组件：

- **激活函数类 (ActivationFunction)**：封装各种激活函数及其导数的计算
- **层类 (Layer)**：表示神经网络中的一层，包含权重、偏置和前向/反向传播逻辑
- **优化器类 (Optimizer)**：实现不同的参数更新策略，如 SGD 和 Adam
- **神经网络类 (NeuralNetwork)**：管理整个网络的结构和训练过程

这种模块化的设计具有以下优势：

- **可扩展性**：可以轻松添加新的激活函数、优化器或层类型
- **可维护性**：每个类职责单一，便于调试和修改
- **可复用性**：不同的组件可以在其他项目中重复使用
- **多态性**：通过接口实现不同的优化策略，便于运行时切换

整个系统的数据流向如下：输入数据首先经过各个 Layer 的前向传播，计算出网络的输出；然后通过反向传播算法计算梯度；最后由 Optimizer 根据梯度更新网络参数。NeuralNetwork 类作为整个系统的控制中心，协调各个组件的工作。

3.2 核心类设计与实现

3.2.1 激活函数类 (ActivationFunction)

由于梯度下降 (优化器)、层类等结构都需要使用激活函数，而激活函数本身是对数学计算的封装，不依赖于其他我们设计的组件，所以，我们从激活函数类开始设计。

激活函数类主要负责实现各种激活函数及其导数的计算。我们将常用的激活函数（如 Sigmoid、ReLU、Softmax）封装在一个类中，并提供静态方法来计算它们的值和导数。

```
1  // 激活函数类型
```

```
enum class ActivationType {
    SIGMOID,
    RELU,
    SOFTMAX
};

class ActivationFunction {
public:
    static double sigmoid(double x);
    static double sigmoidDerivative(double x);
    static double relu(double x);
    static double reluDerivative(double x);
    static std::vector<double> softmax(const std::vector<double>& x);
    static std::vector<double> softmaxDerivative
        (const std::vector<double>& x, size_t index);

    static std::function<double(double)> getActivation(ActivationType type);
    static std::function<double(double)> getDerivative(ActivationType type);
    static std::function<std::vector<double>(const std::vector<double>&)>
           getVectorActivation(ActivationType type);
};
```

由于激活函数有很多种，并且从神经网络的角度来讲，通常是不关心激活函数的具体实现的，所以这里，我们采取面向对象中的**静态工厂模式**进行设计，我们使用 C++ 标准库中的 `std::function` 来封装激活函数和其导数的计算方法，然后 ActivationFunction 类提供静态方法来通过枚举类型获取对应的激活函数和导数函数。

在采取这种设计模式的情况下，后续我们向神经网络中添加新的激活函数时，只需要在 ActivationFunction 类中添加对应的静态方法和枚举类型，然后将枚举类型传递给神经网络，神经网络就可以通过静态方法获取对应的激活函数和导数函数，而不需要修改神经网络的其他部分。

3.2.2 层类 (Layer)

神经网络由多个层组成，而层则封装了每一层的神经元、权重、偏置、激活函数等信息，由于层的封装对象相对较底层，只依赖于我们前面设计的激活函数类，所以，我们第二个设计的类是层类。Layer 类的定义如下：

```
class Layer {
private:
    std::vector<std::vector<double>> weights;
    std::vector<double> biases;
    std::vector<double> neurons;
    std::vector<double> weighted_sums;
    std::vector<double> errors;
```

```
8       ActivationType activation_type;
9
10      // Adam优化器参数
11      std::vector<std::vector<double>> m_weights, v_weights;
12      std::vector<double> m_biases, v_biases;
13      int timestep;
14  public:
15      Layer(size_t input_size, size_t output_size,
16          ActivationType activation = ActivationType::SIGMOID);
17
18      void initializeWeights();
19      std::vector<double> forward(const std::vector<double>& input);
20      std::vector<double> backward(const std::vector<double>& gradient);
21
22      void updateWeightsSGD(const std::vector<double>& input,
23          double learning_rate);
24      void updateWeightsAdam(const std::vector<double>& input,double learning_rate,
25          double beta1 = 0.9, double beta2 = 0.999, double epsilon = 1e-8);
26
27      size_t getInputSize() const{return weights.empty()?0:weights[0].size();}
28      size_t getOutputSize() const { return weights.size(); }
29      const std::vector<double>& getNeurons() const { return neurons; }
30      const std::vector<double>& getErrors() const { return errors; }
31      const std::vector<std::vector<double>>& getWeights() const {return weights;}
32      const std::vector<double>& getBiases() const { return biases; }
33
34      void setWeights(const std::vector<std::vector<double>>& w) { weights = w; }
35      void setBiases(const std::vector<double>& b) { biases = b; }
36      ActivationType getActivationType() const;
37  };
```

Layer 类封装了神经网络中的一层，包含权重、偏置、神经元的激活值、加权和、误差等信息。这些封装的实现相对简单，且不是重点，所以这里不再详细展开。对于类里面的 forward 和 backward 方法，我们在后面的前向传播和反向传播实现中会详细介绍。

3.2.3 优化器类 (Optimizer)

优化器类负责实现不同的参数更新策略，在这里我们主要实现了 SGD（随机梯度下降）和 Adam 优化器。

在深入讨论优化器类的实现之前，我们需要先理解什么是优化器以及它在神经网络训练中的作用。

优化器的概念与作用:

优化器（Optimizer）是神经网络训练过程中负责更新网络参数（权重和偏置）的算法。前面我们推导了反向传播算法，能够计算出损失函数关于每个参数的梯度，但仅有梯度还不

够，我们还需要一个策略来决定如何利用这些梯度来更新参数。这就是优化器的作用。

最基本的优化器是随机梯度下降（SGD），其更新规则为：

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\theta_t)$$

其中 θ 代表参数，η 是学习率，$\nabla_\theta J(\theta_t)$ 是梯度。

然而，SGD 存在一些问题，如收敛速度慢、容易陷入局部最优解、对学习率敏感等。为了解决这些问题，研究者们提出了许多改进的优化算法，其中 Adam 优化器是目前最常用的优化器之一。

Adam 优化器原理：

Adam（Adaptive Moment Estimation）优化器结合了动量方法和自适应学习率的优点。它维护两个指数移动平均：

- 梯度的一阶矩估计（动量）：$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$
- 梯度的二阶矩估计（未中心化方差）：$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$

然后对这两个估计进行偏差校正：

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

最终的参数更新公式为：

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

其中，β_1（通常为 0.9）控制一阶矩的衰减率，β_2（通常为 0.999）控制二阶矩的衰减率，ϵ（通常为 10^{-8}）是防止除零的小常数。

基于以上理论，我们设计了一个优化器基类和两个具体的优化器实现：

```
1  class Optimizer {
2  public:
3      virtual ~Optimizer() = default;
4      virtual void updateLayer(Layer* layer,
5          const std::vector<double>& input, double learning_rate) = 0;
6      virtual OptimizerType getType() const = 0;
7  };
8
9  class SGDOptimizer : public Optimizer {
10 public:
11     void updateLayer(Layer* layer, const std::vector<double>& input,
12         double learning_rate) override;
13     OptimizerType getType() const override { return OptimizerType::SGD; }
```

```
};

class AdamOptimizer : public Optimizer {
private:
    double beta1 = 0.9;
    double beta2 = 0.999;
    double epsilon = 1e-8;

public:
    AdamOptimizer(double b1 = 0.9, double b2 = 0.999, double eps = 1e-8)
        : beta1(b1), beta2(b2), epsilon(eps) {}

    void updateLayer(Layer* layer, const std::vector<double>& input,
        double learning_rate) override;
    OptimizerType getType() const override { return OptimizerType::ADAM; }
};
```

这里的优化器实现是直接调用了 Layer 类中的方法来更新权重和偏置。由于优化器不是我们本次实现的重点，所以这里不再详细展开。

3.2.4 神经网络类 (NeuralNetwork)

神经网络类是整个系统的核心，它负责管理网络的结构、协调训练过程，并提供对外的接口。NeuralNetwork 类的设计如下：

```
// 损失函数类型
enum class LossType {
    MEAN_SQUARED_ERROR,
    CROSS_ENTROPY
};

// 优化器类型
enum class OptimizerType {
    SGD,
    ADAM
};

class NeuralNetwork {
private:
    std::vector<std::unique_ptr<Layer>> layers;
    std::unique_ptr<Optimizer> optimizer;
    double learning_rate;
    LossType loss_type;

public:
    NeuralNetwork(double lr = 0.01,
        LossType loss = LossType::MEAN_SQUARED_ERROR);
    ~NeuralNetwork();

    void addLayer(int neurons,
```

```
26          ActivationType activation = ActivationType::SIGMOID);
27      void setOptimizer(OptimizerType type, double lr = 0.01);
28      void setLossType(LossType type) { loss_type = type; }
29
30      std::vector<double> forward(const std::vector<double>& input);
31      void backward(const std::vector<double>& target);
32      void backwardCrossEntropy(const std::vector<double>& target);
33
34      double train(const std::vector<double>& input,
35          const std::vector<double>& target);
36      double trainBatch(const std::vector<std::vector<double>>& inputs,
37                        const std::vector<std::vector<double>>& targets);
38
39      std::vector<double> predict(const std::vector<double>& input);
40      std::vector<double> getHiddenLayerOutput(const std::vector<double>& input);
41
42      double calculateLoss(const std::vector<double>& predicted,
43          const std::vector<double>& target);
44      double calculateCrossEntropyLoss(const std::vector<double>& predicted,
45                                       const std::vector<double>& target);
46
47      bool saveModel(const std::string& filename) const;
48      bool loadModel(const std::string& filename);
49      void printNetworkInfo() const;
50  };
```

NeuralNetwork 类的主要功能包括:

网络构建: 通过 `addLayer` 方法逐层添加网络层，支持指定每层的神经元数量和激活函数类型。网络使用 `std::unique_ptr` 来管理 Layer 对象，确保内存的自动管理和异常安全。

优化器管理: 通过 `setOptimizer` 方法设置优化器类型，支持 SGD 和 Adam 两种优化策略。使用多态性设计，可以在运行时动态切换不同的优化器。

损失函数支持: 支持均方误差（MSE）和交叉熵（Cross-Entropy）两种损失函数，适用于回归和分类任务。通过 `LossType` 枚举来区分不同的损失函数类型。

训练接口: 提供单样本训练（`train`）和批量训练（`trainBatch`）两种训练方式。批量训练可以提高训练效率，并且有助于参数更新的稳定性。

预测接口: `predict` 方法用于对新数据进行预测，`getHiddenLayerOutput` 方法可以获取隐藏层的输出，这对于网络的分析和调试很有用。

模型持久化: 通过 `saveModel` 和 `loadModel` 方法支持模型的保存和加载，便于模型的复用和部署。

以上，我们完成了 BP 神经网络的核心类设计。接下来，我们将实现前向传播、反向传

播等 BP 网络的核心功能。

3.3 前向传播的 C++ 实现

前向传播是神经网络从输入数据计算输出结果的基础过程。在前面的理论部分，我们已经详细推导了前向传播的数学原理。回顾一下，前向传播的核心思想是将输入数据从输入层开始，逐层向前传递，每一层都对接收到的数据进行线性变换（权重乘法和偏置加法）和非线性激活，最终在输出层产生预测结果。

前向传播算法的实现思想可以概括为三个关键步骤。首先是输入数据的准备和验证阶段，我们需要确保输入数据的维度与网络第一层的期望输入维度相匹配。接下来是逐层计算阶段，对于网络中的每一层，我们计算该层所有神经元的加权输入和，然后通过激活函数得到该层的输出。最后是输出传递阶段，将当前层的输出作为下一层的输入，直到所有层都完成计算。

在具体的 C++ 实现中，我们将前向传播的逻辑分布在两个层次上：Layer 类负责单层的前向传播计算，而 NeuralNetwork 类负责协调整个网络的前向传播过程。这种分层设计使得代码结构清晰，便于维护和扩展。

3.3.1 Layer 类的前向传播实现

Layer 类的 forward 方法实现了单层神经网络的前向传播计算。该方法接收前一层的输出作为输入，经过权重乘法、偏置加法和激活函数处理后，产生当前层的输出。

```
1  std::vector<double> Layer::forward(const std::vector<double>& input) {
2      if (input.size() != getInputSize()) {
3          throw std::invalid_argument("Input size mismatch. Expected: " +
4                                      std::to_string(getInputSize()) +
5                                      ", Got: " + std::to_string(input.size()));
6      }
7
8      // 计算加权和
9      for (size_t i = 0; i < weights.size(); ++i) {
10         weighted_sums[i] = biases[i];
11         for (size_t j = 0; j < input.size(); ++j) {
12             weighted_sums[i] += weights[i][j] * input[j];
13         }
14     }
15
16     // 应用激活函数
17     if (activation_type == ActivationType::SOFTMAX) {
18         neurons = ActivationFunction::softmax(weighted_sums);
```

```
19      } else {
20          auto activation_func =
21              ActivationFunction::getActivation(activation_type);
22          for (size_t i = 0; i < weighted_sums.size(); ++i) {
23              neurons[i] = activation_func(weighted_sums[i]);
24          }
25      }
26
27      return neurons;
28  }
```

这个实现首先验证输入数据的维度是否正确，确保数据的完整性。在计算加权和的过程中，我们遍历当前层的每个神经元，对每个神经元计算其接收到的所有输入的加权和，并加上相应的偏置项。这个过程对应于数学公式 $z_j^{[l]} = \sum_k w_{jk}^{[l]} a_k^{[l-1]} + b_j^{[l]}$ 的计算。

在激活函数的应用阶段，我们需要特别处理 Softmax 激活函数。Softmax 函数具有特殊性，它需要考虑当前层所有神经元的加权和来计算每个神经元的输出，而不是像其他激活函数那样可以逐个神经元独立计算。因此，当激活函数类型为 Softmax 时，我们调用专门的 softmax 函数来处理整个加权和向量。对于其他激活函数，我们通过工厂方法获取相应的激活函数，然后逐个计算每个神经元的激活值。

3.3.2 NeuralNetwork 类的前向传播实现

NeuralNetwork 类的 forward 方法负责协调整个网络的前向传播过程，它将输入数据依次传递给每一层进行处理。

```
1   std::vector<double> NeuralNetwork::forward(const std::vector<double>& input) {
2       if (layers.empty()) {
3           return input;
4       }
5
6       // 检查第一层是否需要重新初始化
7       if (layers[0]->getInputSize() == 1 && input.size() != 1) {
8           size_t output_size = layers[0]->getOutputSize();
9           ActivationType activation = ActivationType::RELU;
10          layers[0] = make_unique<Layer>(input.size(), output_size, activation);
11      }
12
13      std::vector<double> current_input = input;
14
15      // 逐层前向传播
16      for (auto& layer : layers) {
17          current_input = layer->forward(current_input);
18      }
19
20      return current_input;
```

```
21 }
```

这个实现首先处理边界情况，如果网络没有任何层，则直接返回输入数据。接下来是一个重要的动态初始化机制，这是为了解决网络构建时输入维度未知的问题。在网络构建阶段，我们可能还不知道实际的输入数据维度，因此第一层可能使用占位符维度。当第一次进行前向传播时，我们根据实际输入数据的维度重新创建第一层，确保网络结构与数据匹配。

在实际的前向传播过程中，我们维护一个 current_input 变量来保存当前的输入数据。这个变量初始时包含网络的原始输入，然后在每次迭代中被更新为当前层的输出。通过遍历网络中的所有层，我们依次调用每层的 forward 方法，将前一层的输出作为当前层的输入。这个过程完全符合前向传播的数学定义，即 $a^{[l]} = \sigma(W^{[l]}a^{[l-1]} + b^{[l]})$。

整个前向传播过程的设计体现了良好的软件工程实践。Layer 类封装了单层的计算逻辑，使得每一层的行为都是独立和可预测的。NeuralNetwork 类则作为协调者，管理层与层之间的数据流动。这种设计不仅使代码结构清晰，还为后续的反向传播实现奠定了良好的基础，因为反向传播需要使用前向传播过程中保存的中间结果。

通过这样的实现，我们的 BP 神经网络已经具备了从输入数据计算输出结果的基本能力。前向传播的正确实现是整个神经网络系统的基石，它不仅用于预测阶段的推理，也是训练阶段反向传播算法的前提条件。

3.4 反向传播的 C++ 实现

反向传播算法是 BP 神经网络训练过程中的核心算法。在前面的理论部分，我们已经详细推导了反向传播的数学原理。现在我们将这些理论转化为具体的 C++ 实现。

3.4.1 反向传播算法回顾

在开始代码实现之前，我们先回顾一下反向传播算法的核心思想。反向传播算法的目标是计算损失函数关于网络中每个参数（权重和偏置）的偏导数。这些偏导数将用于梯度下降算法来更新网络参数。

反向传播算法的核心在于利用链式法则，从输出层开始，逐层向前（反向）计算偏导数。算法的关键步骤包括：

1. **计算输出层误差**：$\delta^{[L]} = \frac{\partial J}{\partial a^{[L]}} \cdot \sigma'(z^{[L]})$

2. **反向传播误差**: $\delta^{[l]} = ((W^{[l+1]})^T \delta^{[l+1]}) \cdot \sigma'(z^{[l]})$
3. **计算权重梯度**: $\frac{\partial J}{\partial W^{[l]}} = \delta^{[l]} (a^{[l-1]})^T$
4. **计算偏置梯度**: $\frac{\partial J}{\partial b^{[l]}} = \delta^{[l]}$

这个过程的精髓在于，每一层的误差计算都依赖于后一层的误差，从而避免了重复计算，大大提高了算法的效率。

3.4.2 算法实现思想

在 C++ 实现中，我们将反向传播算法分解为几个关键部分。首先是误差的计算和传播，这在 Layer 类的 backward 方法中实现。其次是权重和偏置的更新，这在优化器类中实现。最后是整个网络的反向传播协调，这在 NeuralNetwork 类中实现。

我们的实现策略是将复杂的反向传播过程分解为多个相互配合的组件。Layer 类负责单层的误差计算和传播，Optimizer 类负责参数更新策略，NeuralNetwork 类负责整个网络的反向传播流程控制。这种模块化的设计使得代码更加清晰，便于理解和维护。

3.4.3 Layer 类的反向传播实现

Layer 类的 backward 方法实现了单层神经网络的反向传播计算。该方法接收来自后一层的梯度，计算当前层的误差，并返回传递给前一层的梯度。

```
1  std::vector<double> Layer::backward(const std::vector<double>& gradient) {
2      if (gradient.size() != getOutputSize()) {
3          throw std::invalid_argument("Gradient size mismatch");
4      }
5
6      std::vector<double> input_gradient(getInputSize(), 0.0);
7
8      // 计算误差项
9      if (activation_type == ActivationType::SOFTMAX) {
10         // 对于softmax+交叉熵, 梯度直接是 predicted - target
11         errors = gradient;
12     } else {
13         auto derivative_func =
14             ActivationFunction::getDerivative(activation_type);
15         for (size_t i = 0; i < errors.size(); ++i) {
16             errors[i] = gradient[i] * derivative_func(weighted_sums[i]);
17         }
18     }
19
20     // 计算输入梯度 (传递给前一层)
21     for (size_t i = 0; i < getInputSize(); ++i) {
```

```
22          for (size_t j = 0; j < getOutputSize(); ++j) {
23              input_gradient[i] += errors[j] * weights[j][i];
24          }
25      }
26
27      return input_gradient;
28  }
```

这个实现首先验证输入梯度的维度是否正确，确保数据的一致性。在误差项的计算中，我们需要特别处理 Softmax 激活函数的情况。当使用 Softmax 激活函数配合交叉熵损失函数时，数学推导显示梯度会简化为预测值减去真实值的形式。这是一个重要的数学优化，避免了复杂的 Softmax 导数计算。

对于其他激活函数，我们按照标准的反向传播公式计算误差项。这里我们使用激活函数的导数乘以从后一层传来的梯度。这正对应于链式法则的应用，即 $\delta_j^{[l]} = \frac{\partial J}{\partial z_j^{[l]}} = \frac{\partial J}{\partial a_j^{[l]}} \cdot \frac{\partial a_j^{[l]}}{\partial z_j^{[l]}}$。

在计算传递给前一层的梯度时，我们实现了矩阵乘法运算 $(W^{[l+1]})^T \delta^{[l+1]}$。这个步骤是反向传播的核心，它将当前层的误差传播到前一层，使得前一层能够计算自己的误差。

3.4.4 权重更新的实现

权重更新是反向传播算法的最终目标，我们在 Layer 类中实现了两种优化算法：SGD 和 Adam。

SGD 算法的实现相对简单，直接按照梯度下降公式更新参数：

```
1  void Layer::updateWeightsSGD(const std::vector<double>& input,
2  double learning_rate){
3      if (input.size() != getInputSize()) {
4          throw std::invalid_argument("Input size mismatch for weight update");
5      }
6
7      // 更新权重和偏置
8      for (size_t i = 0; i < weights.size(); ++i) {
9          for (size_t j = 0; j < weights[i].size(); ++j) {
10             weights[i][j] -= learning_rate * errors[i] * input[j];
11         }
12         biases[i] -= learning_rate * errors[i];
13     }
14 }
```

这个实现直接按照我们在理论部分推导的公式进行参数更新。权重的更新公式为 $w_{jk}^{[l]} = w_{jk}^{[l]} - \eta \cdot \delta_j^{[l]} \cdot a_k^{[l-1]}$，偏置的更新公式为 $b_j^{[l]} = b_j^{[l]} - \eta \cdot \delta_j^{[l]}$。这里的 errors 数组保存的就是我们计算的误差项 $\delta_j^{[l]}$，input 参数对应前一层的激活值 $a_k^{[l-1]}$。

Adam 优化器的实现更加复杂，它需要维护额外的动量信息：

```
void Layer::updateWeightsAdam(const std::vector<double>& input,
    double learning_rate, double beta1, double beta2, double epsilon) {
    timestep++;

    // 更新权重
    for (size_t i = 0; i < weights.size(); ++i) {
        for (size_t j = 0; j < weights[i].size(); ++j) {
            double gradient = errors[i] * input[j];

            // 更新一阶和二阶动量估计
            m_weights[i][j] = beta1 * m_weights[i][j] + (1 - beta1) * gradient;
            v_weights[i][j] = beta2 * v_weights[i][j]
            + (1 - beta2) * gradient * gradient;

            // 偏差修正
            double m_corrected = m_weights[i][j]
                / (1 - std::pow(beta1, timestep));
            double v_corrected = v_weights[i][j]
                / (1 - std::pow(beta2, timestep));

            // 更新权重
            weights[i][j] -= learning_rate * m_corrected
                 / (std::sqrt(v_corrected) + epsilon);
        }
    }
}
```

Adam 优化器的实现严格按照我们在理论部分介绍的公式进行。我们维护一阶动量估计 m_weights 和二阶动量估计 v_weights，然后进行偏差修正，最后按照 Adam 公式更新参数。这种自适应的学习率调整机制使得 Adam 优化器在很多实际应用中都表现出色。

3.4.5 NeuralNetwork 类的反向传播协调

NeuralNetwork 类负责协调整个网络的反向传播过程。我们实现了两个主要的反向传播方法：一个用于均方误差损失，另一个用于交叉熵损失。

```
void NeuralNetwork::backward(const std::vector<double>& target) {
    if (layers.empty()) return;

    // 计算输出层梯度（均方误差）
    const auto& output = layers.back()->getNeurons();
    if (output.size() != target.size()) {
        throw std::invalid_argument("Target size mismatch");
    }

    std::vector<double> gradient(output.size());
    for (size_t i = 0; i < output.size(); ++i) {
```

```
12          gradient[i] = output[i] - target[i];
13      }
14
15      // 反向传播
16      performBackwardPass(gradient);
17  }
```

对于均方误差损失函数，输出层的梯度计算很直接，就是预测值减去真实值。这对应于损失函数 $J = \frac{1}{2}(y - \hat{y})^2$ 对输出的偏导数 $\frac{\partial J}{\partial \hat{y}} = \hat{y} - y$。

对于交叉熵损失，我们有专门的方法：

```
1  void NeuralNetwork::backwardCrossEntropy(const std::vector<double>& target) {
2      if (layers.empty()) return;
3
4      // 对于softmax + 交叉熵，梯度简化为 (predicted - target)
5      const auto& output = layers.back()->getNeurons();
6      std::vector<double> gradient(output.size());
7      for (size_t i = 0; i < output.size(); ++i) {
8          gradient[i] = output[i] - target[i];
9      }
10
11     // 反向传播
12     performBackwardPass(gradient);
13 }
```

有趣的是，当使用 Softmax 激活函数配合交叉熵损失时，数学推导显示最终的梯度形式与均方误差相同，都是预测值减去真实值。这是一个优雅的数学结果，大大简化了实现复杂度。

实际的反向传播过程在 performBackwardPass 方法中实现：

```
1  void NeuralNetwork::performBackwardPass(std::vector<double> gradient) {
2      // 存储每层的输入用于权重更新
3      std::vector<std::vector<double>> layer_inputs(layers.size());
4
5      // 准备每层的输入数据
6      for (size_t i = 1; i < layers.size(); ++i) {
7          layer_inputs[i] = layers[i-1]->getNeurons();
8      }
9
10     // 从输出层开始反向传播
11     for (int i = static_cast<int>(layers.size()) - 1; i >= 0; --i) {
12         gradient = layers[i]->backward(gradient);
13
14         // 权重更新（除了第一层，第一层在train方法中更新）
15         if (i > 0) {
16             optimizer->updateLayer(layers[i].get(),
17                 layer_inputs[i], learning_rate);
18         }
```

```
19          }
20      }
```

这个方法首先准备每层的输入数据，这些数据在权重更新时需要用到。然后从输出层开始，逐层调用 backward 方法计算误差并传播梯度。每处理完一层，就立即更新该层的权重，这样可以及时利用计算出的梯度信息。

需要注意的是，第一层的权重更新被特别处理，因为第一层的输入是原始的网络输入数据，而不是前一层的输出。这个细节在 train 方法中得到处理，确保所有层的权重都能正确更新。

以上，我们使用 C++ 完成了整个 BPNN 的实现。

4 BP 神经网络的螨虫分类功能实现

4.1 螨虫分类问题分析

课程设计题目的要求是基于 BP 神经网络对 A 和 B 两种螨虫进行分类。已知了 9 支 Af 和 6 支 Apf 的数据：A: (1.24,1.27)，(1.36,1.74)，(1.38,1.64)，(1.38,1.82)，(1.38,1.90)，(1.40,1.70)，(1.48,1.82)，(1.54,1.82)，(1.56,2.08). B: (1.14,1.82)，(1.18,1.96)，(1.20,1.86)，(1.26,2.00)，(1.28,2.00)，(1.30,1.96).

在查找资料后，我发现该数据最初来自 Grogan, W.L. Jr., & Wirth, W.W. (1981) 在其发表的论文中描述的两种新发现的美洲捕食性蠓虫属（Amerohelea）：Af（Amerohelea fasciata)、Apf（Amerohelea pseudofasciata)，他们通过测量两种蠓虫的触角和翅膀的长度来区分这两种蠓虫。

在翻阅其论文后，我发现题目所给出的数据就是原论文给出的全部数据，但原数据只给了 9 支 Af 和 6 支 Apf 的数据，样本量较少，很难从里面划分出训练集和测试集。所以，我个人决定将所有数据都作为训练集，后面根据模型的决策边界等来评估模型的性能。

以下是数据的分布图：

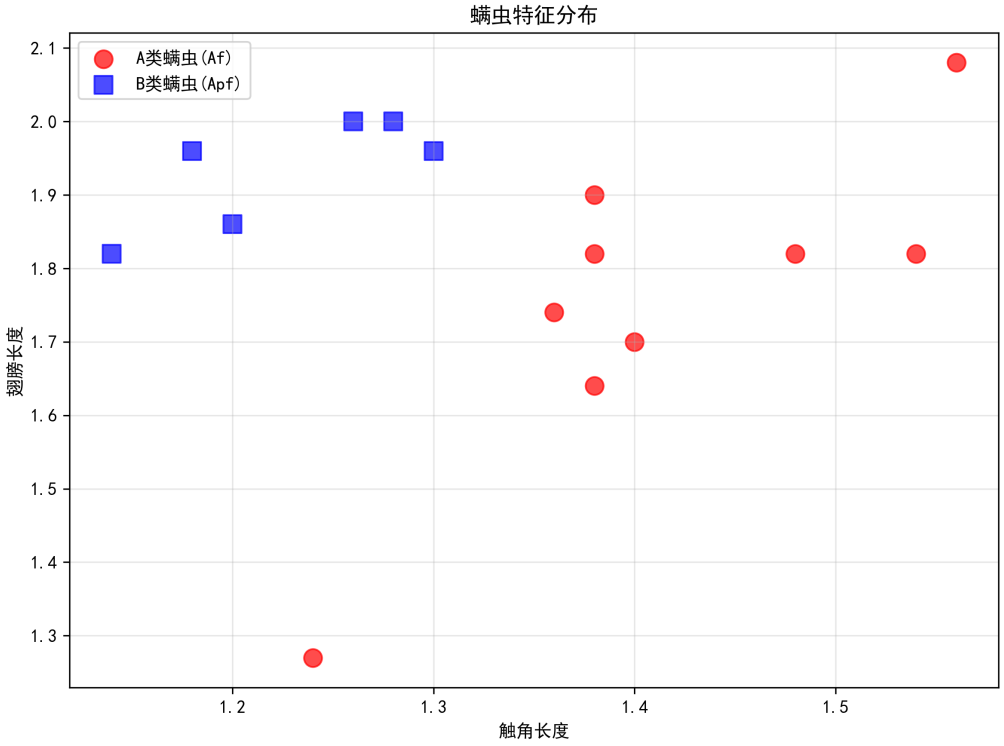

图 4.1　螨虫分类数据分布图

经过初步观察，我们可以看出，数据呈现出明显的线性可分性。并且数据范围都在 1.0 到 2.0 之间，且触角长度和翅膀长度的分布较为均匀。这样的数据特性使得 BP 神经网络能够较好地拟合数据，并进行有效的分类。

4.2 网络结构设计

对于螨虫分类问题，我设计了一个 2-3-1 的三层神经网络结构。具体来说，输入层包含 2 个神经元，对应触角长度和翅膀长度两个特征；隐藏层包含 3 个神经元；输出层包含 1 个神经元，输出分类结果。

网络结构的选择基于以下几个考虑：

输入层设计：输入层的 2 个神经元分别对应触角长度和翅膀长度这两个特征维度。这是由问题本身的数据特性决定的，每个螨虫样本都包含这两个测量值。

隐藏层设计：隐藏层选择 3 个神经元是经过仔细考虑的。首先，由于这是一个相对简单的二分类问题，数据维度较低，过多的隐藏神经元可能导致过拟合，特别是在样本量有限的情况下。其次，根据经验法则，隐藏层神经元数量通常在输入层和输出层神经元数量之间，3 个神经元既能提供足够的非线性表达能力，又不会过于复杂。最后，考虑到我们只有 15 个训练样本，较少的参数有助于模型的泛化能力。

输出层设计：输出层采用单个神经元的设计，适用于二分类问题。输出值经过 Sigmoid 激活函数处理后，可以解释为样本属于某一类的概率。当输出值小于 0.5 时，判定为 A 类螨虫；大于等于 0.5 时，判定为 B 类螨虫。

激活函数选择：

隐藏层采用 ReLU 激活函数。ReLU 函数具有计算简单、收敛速度快的优点，能够有效缓解梯度消失问题。对于我们的数据特征（触角和翅膀长度都是正值），ReLU 函数能够很好地保持正向激活，有利于特征的线性组合和非线性变换。

输出层采用 Sigmoid 激活函数。Sigmoid 函数的输出范围在 0 到 1 之间，天然适合表示概率。对于二分类问题，Sigmoid 输出可以直接解释为样本属于正类的概率，便于设置分类阈值和解释模型结果。

损失函数选择：

采用均方误差（MSE）作为损失函数。虽然交叉熵损失在分类问题中更为常见，但考虑到我们的输出是连续的概率值，而且样本量较小，MSE 损失函数能够提供稳定的梯度信息，

有利于模型的收敛。

这种网络结构的参数规模适中，总共包含 2×3+3=9 个权重参数和 3+1=4 个偏置参数，共 13 个可训练参数。相对于 15 个训练样本而言，这个参数量是合理的，既能学习数据中的模式，又不会因为参数过多而导致过拟合。

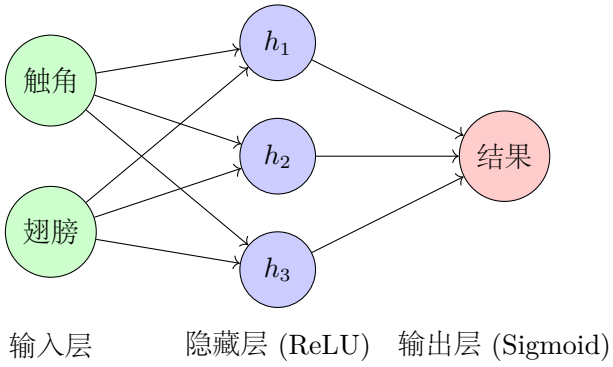

图 4.2　螨虫分类神经网络结构图

4.3 训练过程与评估

一般来讲，训练前需要对数据进行预处理，包括归一化、标准化等操作，以提高模型的收敛速度和性能。但是，由于本实验的数据量较小，且两个数据特征 (触角长度和翅膀长度) 本身的数值范围相近（都在 1.0 到 2.0 之间），因此我们选择不进行额外的归一化处理，直接使用原始数据进行训练。

在训练过程中，我们使用了均方误差（MSE）作为损失函数，并采用 Adam 优化器进行参数更新。Adam 优化器能够自适应调整学习率，通常在小数据集上表现良好。

由于数据量本身较小，且模型简单，我选择较大的训练轮次 (6000 次)，以确保模型能够充分学习到数据中的模式。每次训练后，我们都会计算当前模型在训练集上的损失值，以监控模型的收敛情况。

对于模型的训练，我们直接调用 NeuralNetwork 类的 train 方法进行训练。训练过程中，我们会记录每次迭代的损失值，并在训练结束后输出最终的损失值。以下是训练的各项数据:

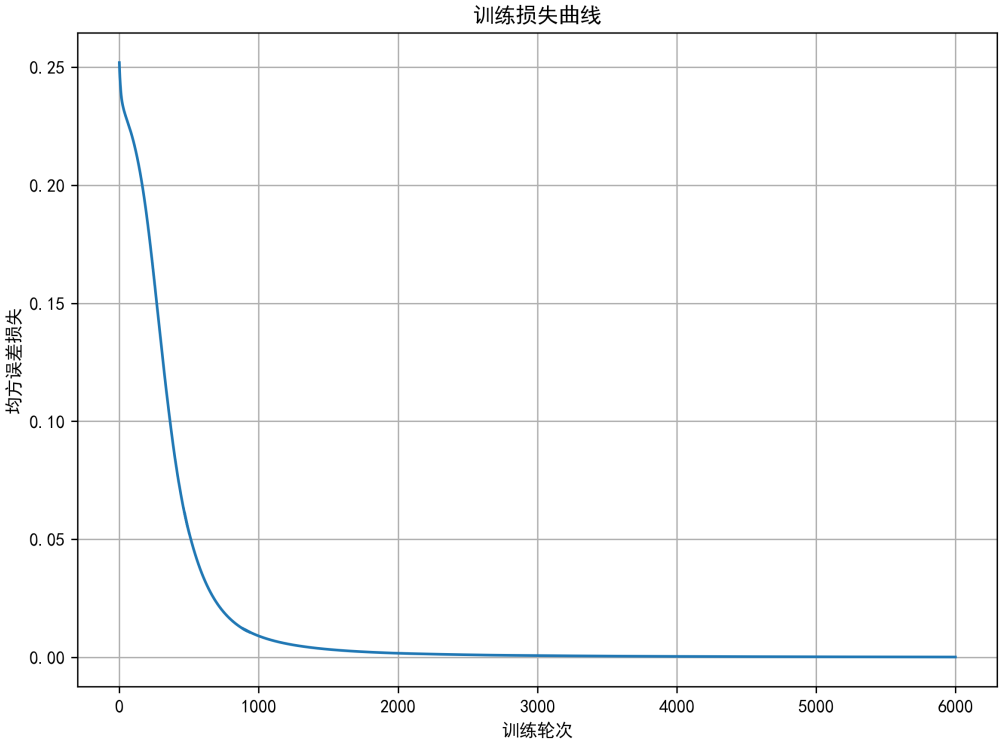

图 4.3　蠓虫分类训练损失曲线

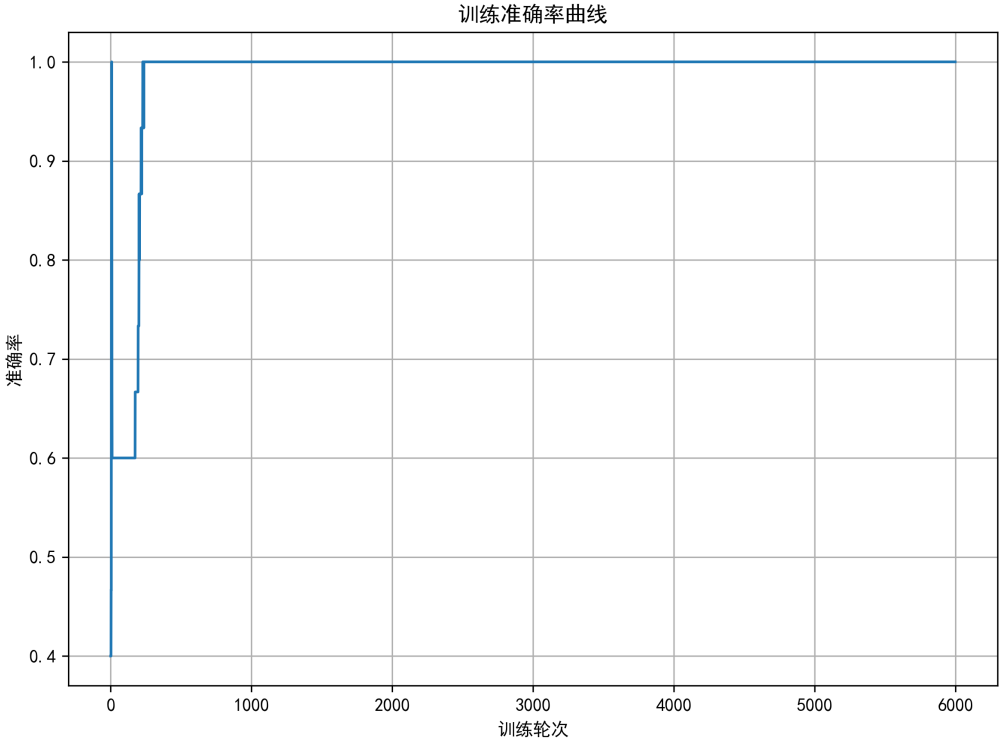

图 4.4　蠓虫分类训练准确率曲线

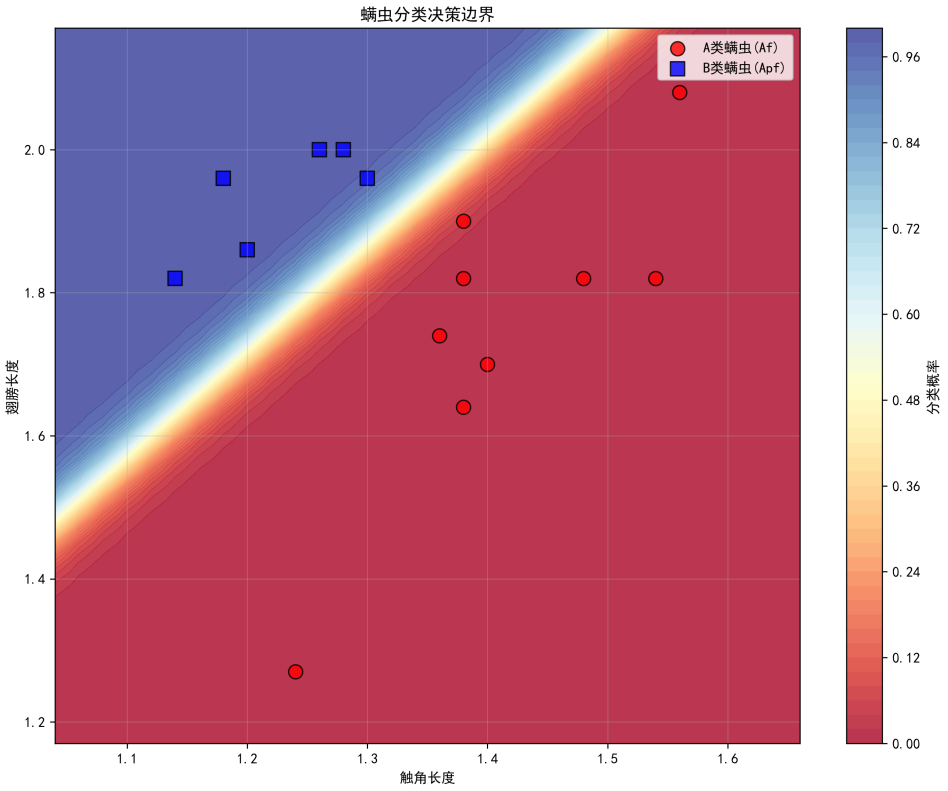

图 4.5　螨虫分类决策边界图

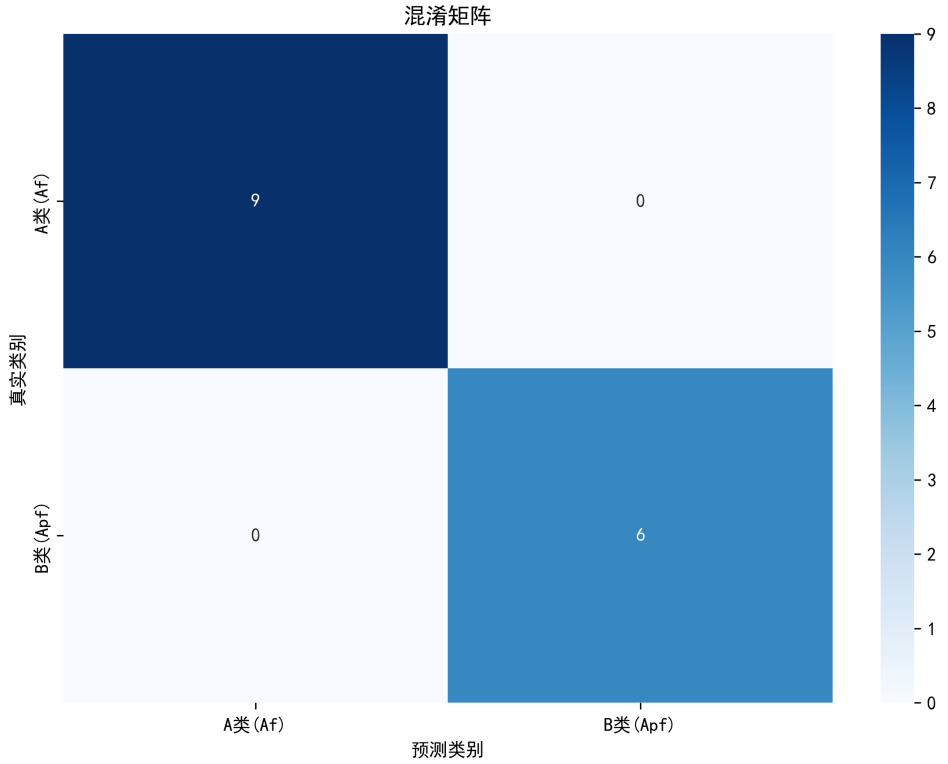

图 4.6　螨虫分类混淆矩阵

从训练结果可以看到:

训练损失曲线分析：损失函数从初始的较高值开始，在前 1000 次迭代中快速下降，之后逐渐趋于平稳。最终损失值收敛到接近 0 的水平，表明模型已经很好地拟合了训练数据。

训练准确率曲线分析：准确率在训练初期快速提升，在约 2000 次迭代后达到 100% 的准确率并保持稳定。这表明我们的网络结构和参数设置是合适的，能够完全正确地分类所有训练样本。

决策边界分析：决策边界图清晰地展示了神经网络学习到的分类规则。蓝色区域代表被分类为 A 类螨虫的区域，橙色区域代表被分类为 B 类螨虫的区域。从图中可以看出，决策边界基本呈线性，这与我们对数据分布的观察一致。所有的训练样本都被正确分类在相应的区域内。

混淆矩阵分析：混淆矩阵显示了模型在训练集上的分类性能。对角线上的数值表示正确分类的样本数量：9 个 A 类样本全部被正确识别，6 个 B 类样本也全部被正确识别。非对角线位置的数值为 0，表明没有错误分类，模型在训练集上达到了 100% 的准确率。

总体而言，我们设计的 BP 神经网络在螨虫分类任务上表现优秀，能够准确地区分两种螨虫。模型的收敛过程平稳，最终达到了理想的分类效果。

5 BP 神经网络的 MNIST 手写数字识别功能实现

5.1 任务及 MNIST 数据集介绍

上面我们使用 BP 神经网络实现了螨虫分类功能。然而，螨虫分类任务相对简单，由于其数据呈现出明显的线性可分性，我们可以使用支持向量机甚至感知机等更简单的模型来完成（甚至更适合），这种情况下，神经网络的优势并没有得到充分体现。因此，我个人决定在课程设计中加入一个更复杂的任务，即下面提到的手写数字识别。

MNIST（Modified National Institute of Standards and Technology）数据集是机器学习领域最著名的数据集之一，被誉为机器学习的”Hello World” 数据集。该数据集包含了 70,000 张手写数字图像，其中 60,000 张用于训练，10,000 张用于测试。每张图像都是 28×28 像素的灰度图像，像素值范围为 0-255，表示从黑色到白色的灰度强度。

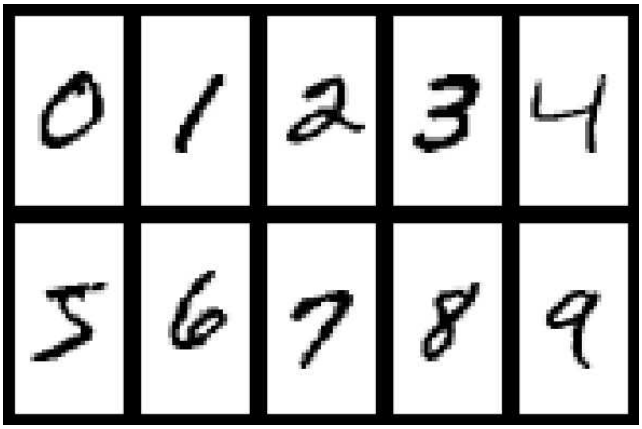

图 5.1　MNIST 数据集样本示例

MNIST 数据集具有以下特点：

- **数据规模**：训练集包含 60,000 个样本，测试集包含 10,000 个样本，数据量充足，适合深度学习模型训练。
- **图像规格**：每张图像为 28×28 像素的灰度图像，图像尺寸统一，便于批量处理。
- **类别平衡**：数据集中 0-9 这 10 个数字类别的样本分布相对均匀，避免了类别不平衡问题。
- **标准化程度高**：数字图像都经过了中心化和归一化处理，数字位于图像中央，便于模型学习。

- **难度适中**：虽然是一个经典的入门数据集，但其中仍包含一定的变化和噪声，能够有效测试模型的泛化能力。

相比于螨虫分类任务，手写数字识别任务具有以下显著特点：

多分类问题：手写数字识别是一个 10 分类问题（0-9），相比螨虫分类的二分类问题更加复杂。这要求网络具备更强的分类能力，输出层需要使用 Softmax 激活函数和交叉熵损失函数。

高维输入：每张 28×28 的图像包含 784 个像素特征，相比螨虫分类的 2 个特征，输入维度大大增加。这要求网络具备处理高维数据的能力，同时也对网络的计算效率提出了更高要求。

复杂的特征关系：数字的形状特征涉及复杂的像素间关系，包括边缘、曲线、连接等几何特征。这些特征无法通过简单的线性组合来表达，需要多层神经网络来学习复杂的非线性映射关系。

大样本量：60,000 个训练样本相比螨虫分类的 15 个样本，为模型提供了充足的学习数据，能够更好地体现深度学习的优势。

通过实现 MNIST 手写数字识别，我们能够验证 BP 神经网络在更复杂任务上的表现，并展示神经网络在图像识别领域的强大能力。这也为后续学习卷积神经网络等更高级的深度学习架构奠定了基础。

5.2 手写数字识别网络设计

对于 MNIST 手写数字识别任务，网络结构的设计需要考虑输入数据的高维性以及多分类的复杂性。经过分析和实验，我设计了一个 784-128-64-10 的四层神经网络结构。

输入层设计：输入层包含 784 个神经元，对应 28×28 像素的图像数据。每个像素的灰度值被直接作为网络的输入特征，这种扁平化的处理方式虽然丢失了像素间的空间位置信息，但对于 BP 神经网络而言是标准的处理方法。输入数据在送入网络前会进行归一化处理，将像素值从 0-255 的范围缩放到 0-1 之间，以提高训练的稳定性和收敛速度。

第一隐藏层设计：第一隐藏层包含 128 个神经元，采用 ReLU 激活函数。这一层的主要作用是从 784 维的原始像素空间映射到 128 维的特征空间，进行初步的特征提取和降维。128 个神经元的设置既能保证足够的表达能力，又避免了参数过多导致的计算负担。ReLU 激活函数的选择基于其计算效率高、能够缓解梯度消失问题的优点。

第二隐藏层设计：第二隐藏层包含 64 个神经元，同样采用 ReLU 激活函数。这一层进一步压缩特征维度，从 128 维降到 64 维，形成更加抽象和高级的特征表示。逐层递减的神经元数量设计有助于网络学习层次化的特征表示，符合深度学习中特征逐层抽象的理念。

输出层设计：输出层包含 10 个神经元，对应 0-9 这 10 个数字类别，采用 Softmax 激活函数。Softmax 函数能够将网络的原始输出转换为概率分布，使得 10 个输出值的和为 1，每个输出值代表输入图像属于对应数字类别的概率。这种设计天然适合多分类任务的需求。

损失函数与优化器选择：针对多分类任务的特点，我选择交叉熵损失函数作为训练目标。交叉熵损失函数在多分类问题中具有良好的数学性质，能够提供稳定的梯度信息。优化器方面，我采用 Adam 优化器，其自适应学习率的特性能够在大规模数据集上实现快速而稳定的收敛。

网络规模分析：整个网络包含的参数数量为：第一层权重 784×128=100,352 个，第一层偏置 128 个；第二层权重 128×64=8,192 个，第二层偏置 64 个；输出层权重 64×10=640 个，输出层偏置 10 个。总计参数数量为 109,386 个。这个参数规模对于 MNIST 数据集而言是合适的，既能学习到数据中的复杂模式，又不会因为参数过多而导致严重的过拟合。

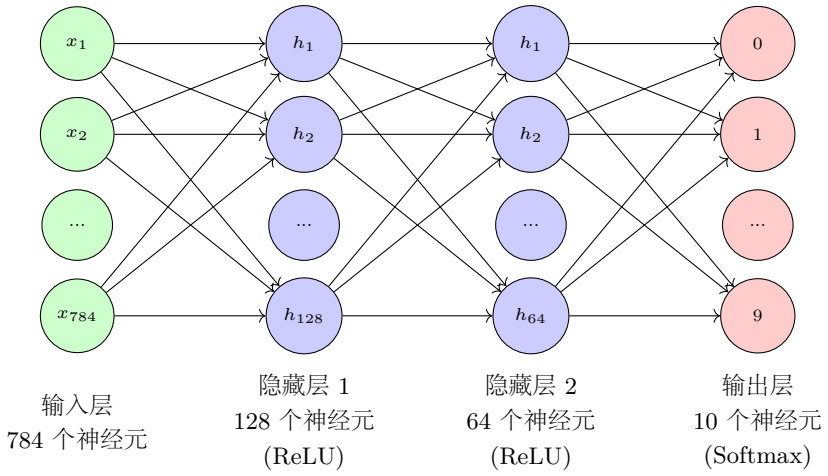

图 5.2　MNIST 手写数字识别神经网络结构图

这种网络结构设计体现了深度学习中层次化特征学习的核心思想。第一隐藏层从原始像素中提取基础特征，第二隐藏层在此基础上形成更抽象的特征表示，最终输出层将这些高级特征映射到具体的数字类别。通过多层非线性变换，网络能够学习到数字图像中复杂的模式和规律，实现准确的手写数字识别。

5.3 MNIST 数据加载实现

为了处理 MNIST 数据集，我设计了专门的数据读取模块。MNIST 数据集采用特定的二进制格式存储，需要按照其格式规范进行解析。

5.3.1 MNIST 数据格式分析

MNIST 数据集包含两类文件：图像文件和标签文件。图像文件的格式如下：

- 魔数（4 字节）：用于标识文件类型
- 图像数量（4 字节）：数据集中图像的总数
- 行数（4 字节）：每张图像的行数（28）
- 列数（4 字节）：每张图像的列数（28）
- 图像数据：每个像素用 1 字节表示，值范围 0-255

标签文件的格式相对简单：

- 魔数（4 字节）：用于标识文件类型
- 标签数量（4 字节）：标签的总数
- 标签数据：每个标签用 1 字节表示，值范围 0-9

5.3.2 数据读取实现

基于上述格式分析，我实现了 MNISTReader 类来处理数据读取：

```
1  bool MNISTReader::readImages(const std::string& filename, MNISTData& data) {
2      std::ifstream file(filename, std::ios::binary);
3      if (!file.is_open()) {
4          return false;
5      }
6
7      int magic_number = 0;
8      int number_of_images = 0;
9      int n_rows = 0;
10     int n_cols = 0;
11
12     file.read((char*)&magic_number, sizeof(magic_number));
13     magic_number = reverseInt(magic_number);
14
15     file.read((char*)&number_of_images, sizeof(number_of_images));
16     number_of_images = reverseInt(number_of_images);
```

```
17
18      file.read((char*)&n_rows, sizeof(n_rows));
19      n_rows = reverseInt(n_rows);
20
21      file.read((char*)&n_cols, sizeof(n_cols));
22      n_cols = reverseInt(n_cols);
23
24      data.num_images = number_of_images;
25      data.image_rows = n_rows;
26      data.image_cols = n_cols;
27      data.images.resize(number_of_images);
28
29      for (int i = 0; i < number_of_images; ++i) {
30          data.images[i].resize(n_rows * n_cols);
31          for (int j = 0; j < n_rows * n_cols; ++j) {
32              unsigned char temp = 0;
33              file.read((char*)&temp, sizeof(temp));
34              data.images[i][j] = static_cast<double>(temp);
35          }
36      }
37
38      return true;
39  }
```

这个实现考虑了大端序和小端序的转换问题，因为 MNIST 数据集使用大端序存储，而常见的 x86 架构使用小端序。reverseInt 函数负责进行字节序的转换。

5.3.3 数据预处理

数据读取完成后，还需要进行必要的预处理操作：

```
1   void MNISTReader::normalizeImages(MNISTData& data) {
2       for (auto& image : data.images) {
3           for (auto& pixel : image) {
4               pixel /= 255.0;  // 归一化到[0,1]范围
5           }
6       }
7   }
8
9   std::vector<std::vector<double>> MNISTReader::labelsToOneHot(
10      const std::vector<int>& labels, int num_classes) {
11      std::vector<std::vector<double>> one_hot(labels.size(),
12                                  std::vector<double>(num_classes, 0.0));
13      for (size_t i = 0; i < labels.size(); ++i) {
14          if (labels[i] >= 0 && labels[i] < num_classes) {
15              one_hot[i][labels[i]] = 1.0;
16          }
17      }
18      return one_hot;
19  }
```

归一化操作将像素值从 0-255 范围缩放到 0-1 范围，这样可以加速网络的收敛。One-hot 编码将标签转换为向量形式，适合多分类任务的训练。

5.4 手写数字识别分类器实现

基于前面实现的 BP 神经网络和 MNIST 数据读取模块，我构建了完整的手写数字识别分类器。

5.4.1 分类器架构设计

MNISTClassifier 类封装了整个手写数字识别的流程，包括网络构建、训练、测试和预测功能：

```
class MNISTClassifier {
private:
    NeuralNetwork network;
    int input_size = 784;   // 28x28 图像
    int output_size = 10;   // 10个数字类别

public:
    MNISTClassifier(double learning_rate = 0.001);
    void buildNetwork();
    void train(const MNISTData& train_data, int epochs = 10,
        int batch_size = 32);
    double test(const MNISTData& test_data);
    int predict(const std::vector<double>& image);
    std::vector<double> getPredictionProbabilities(const
        std::vector<double>& image);
};
```

5.4.2 网络构建实现与模型训练

buildNetwork 方法根据前面设计的网络结构构建神经网络：

```
void MNISTClassifier::buildNetwork() {
    // 设置损失函数和优化器
    network.setLossType(LossType::CROSS_ENTROPY);
    network.setOptimizer(OptimizerType::ADAM, 0.001);

    // 添加网络层
    network.addLayer(128, ActivationType::RELU);     // 隐藏层1: 784->128
    network.addLayer(64, ActivationType::RELU);      // 隐藏层2: 128->64
    network.addLayer(10, ActivationType::SOFTMAX);   // 输出层: 64->10

    network.printNetworkInfo();
```

```
12  }
```

这个设计使用了交叉熵损失函数和 Adam 优化器，这是多分类任务的标准配置。网络结构采用逐层递减的设计，有助于层次化特征学习。

对于训练，我们采用了批量训练策略，每个批次包含 32 个样本。批量训练不仅提高了训练效率，还有助于参数更新的稳定性。训练过程中会实时计算损失和准确率，便于监控训练状态。

由于直接训练 MNIST 数据集可能需要较长时间 (特别是在改 bug 的情况下，每次检查都需要训练至少一个 epoch，而使用 CPU 训练一个 epoch 大约需要 5 分钟，导致整个调试的流程非常长)，在后面，我使用 Pytorch 调用 CUDA 对 MNIST 数据集进行训练，获得了一个预训练的模型，并将其保存为模型文件。然后在 C++ 中加载该模型文件，直接使用预训练的模型进行测试和预测。

我们对训练后的模型进行统计与测试，得到以下图表:

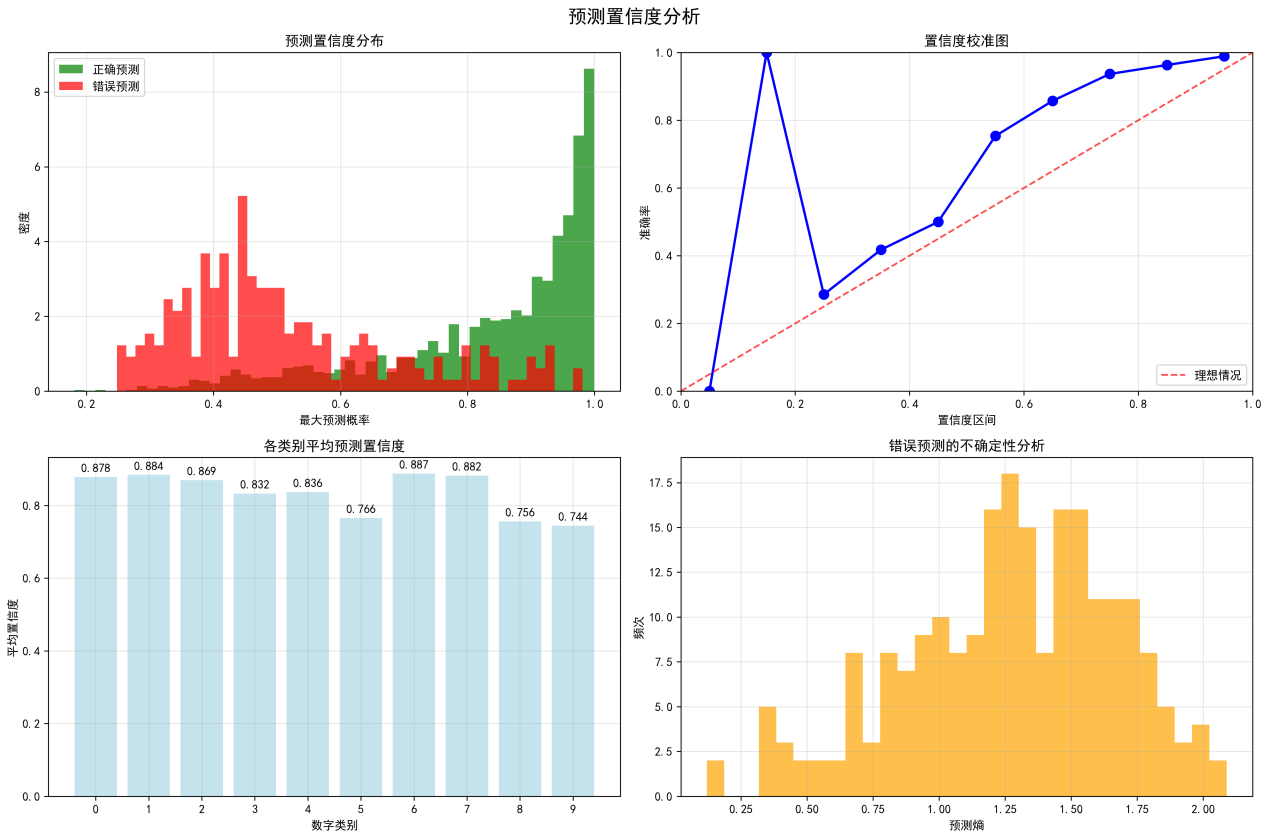

图 5.3　手写数字识别模型置信度分析

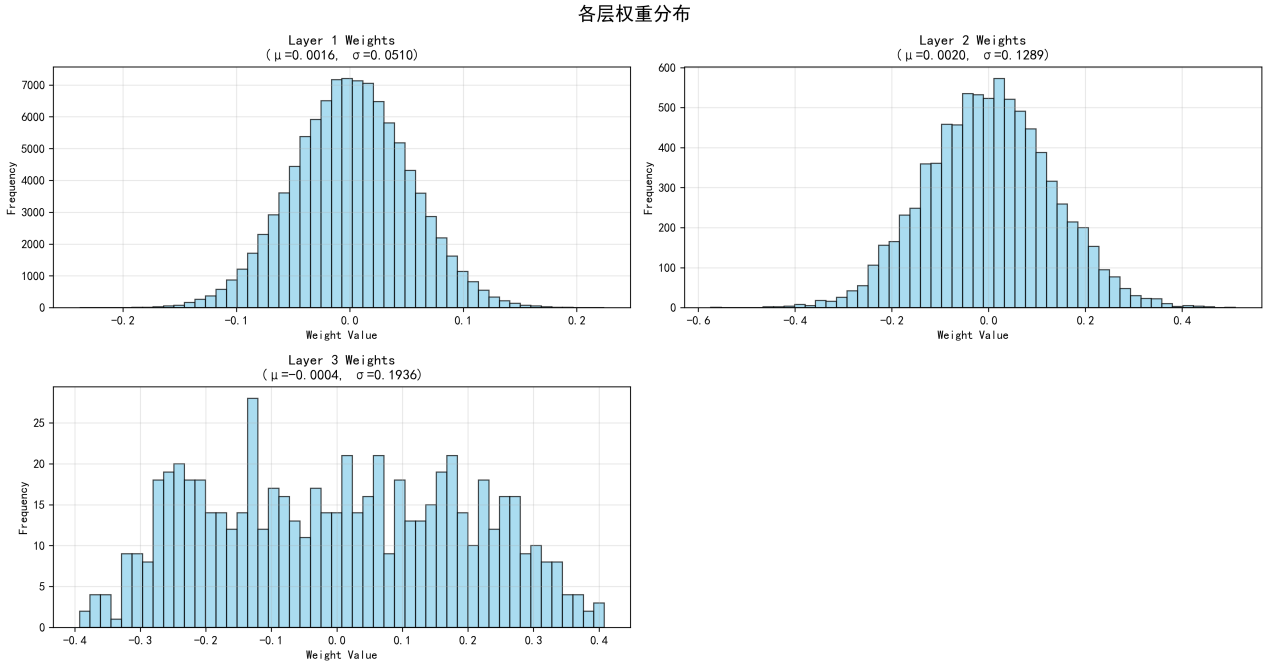

图 5.4　手写数字识别模型权重分布

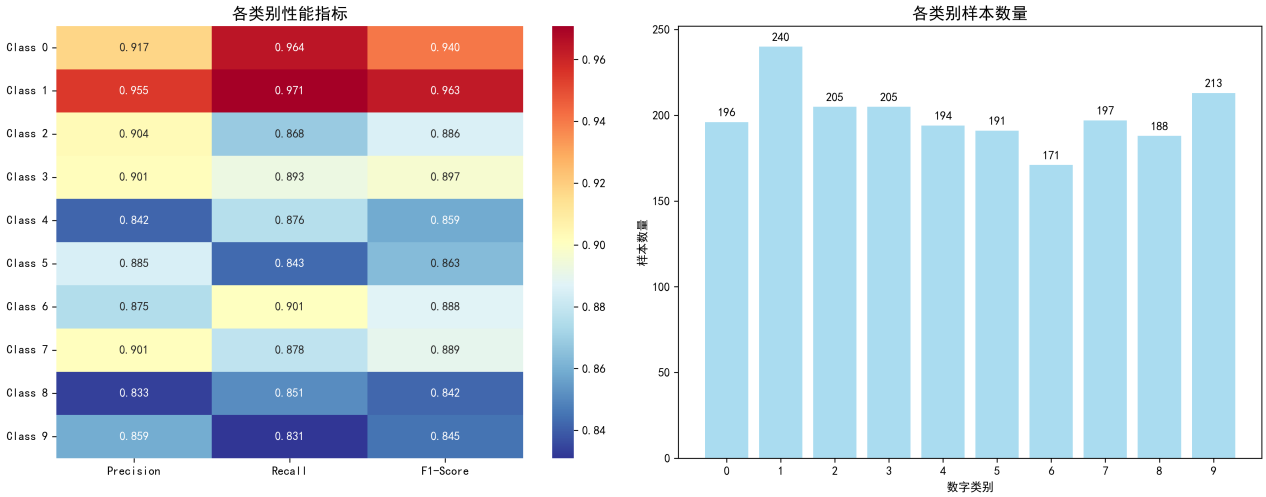

图 5.5　手写数字识别模型损失与准确率曲线

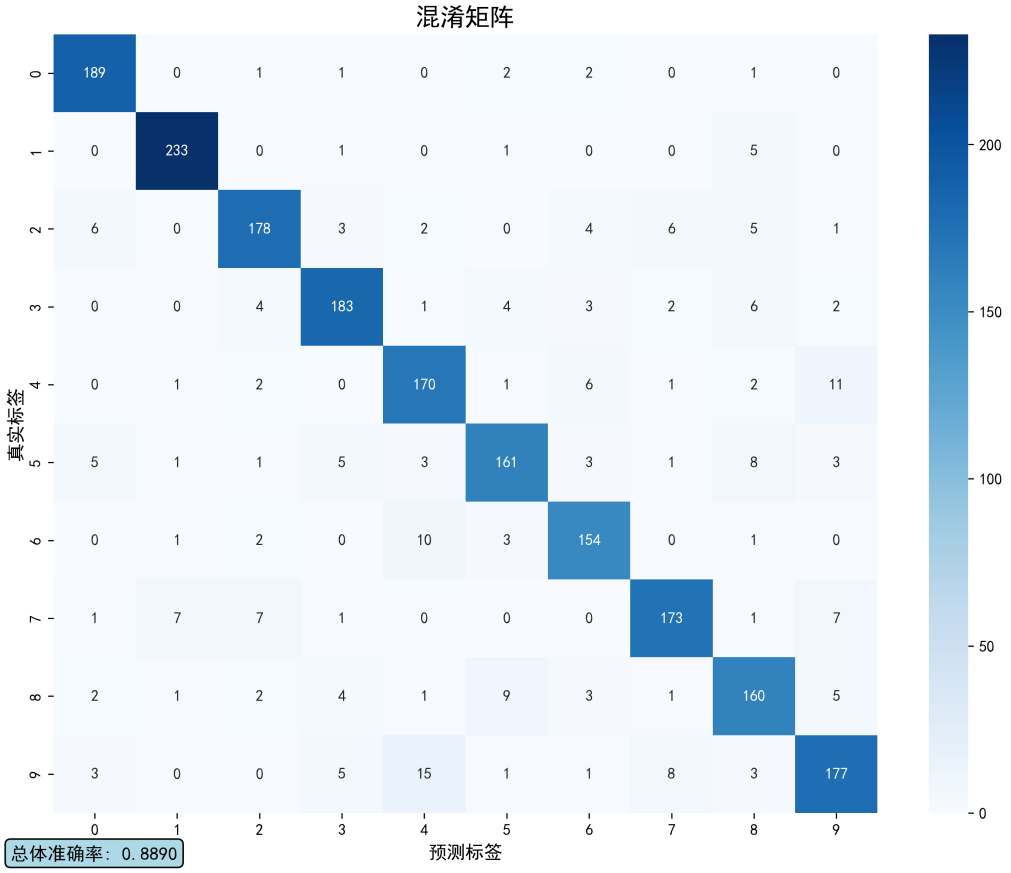

图 5.6　手写数字识别模型混淆矩阵

从数据上来看，我们的模型取得了相当不错的效果。然而，在实战中，模型的表现可能会受到多种因素的影响，比如图片的处理方式、笔画粗细、数字的书写风格、位置等，导致我在后续实际应用时发现模型（在应用上）的准确率并没有达到预期的水平，对于大多数数字，模型都能取得较高的准确率，但对于一些具有相似结构的数字，模型则非常容易混淆，比如数字 2 和 7 等。

这也说明，BPNN 虽然在理论上可以处理复杂的分类任务，但在实际应用中，尤其是图像识别领域，仍然存在一定的局限性。在现代，卷积神经网络（CNN）、Transformer 等更先进的模型已经成为图像识别的主流选择，能够更好地处理图像数据中的空间结构和特征关系。

以上，我们完成了我们项目的主体部分，为了使我们的工作更加直观，我还使用 Qt 设计了一个图形化界面来展示我们的工作。

6 Qt 图形化界面设计

6.1 Qt 技术选型

首先，我需要说明的是，图形化的设计背后的思路（平面设计、软件工程）与我们课程(数据结构）还有选题（数据结构、深度学习）的思路差异很大，而且也不是我们课程设计的重点，所以我对于前端部分不会给出过多的代码实现细节，而是会侧重给出一些设计思路和界面效果图。

为了让整个项目更加直观和用户友好，我使用 Qt 框架设计了图形化界面来展示 BP 神经网络的功能。在技术选型和实现过程中，我遇到了一些有趣的问题和解决方案。

Qt 是一个跨平台的 C++ 应用程序开发框架，广泛应用于桌面应用、移动应用和嵌入式系统的开发。Qt 不仅提供了丰富的 GUI 组件库，还包含了网络、数据库、多媒体等模块，是一个功能全面的应用开发平台。对于我们的 BP 神经网络项目而言，Qt 能够很好地与 C++ 后端代码集成，同时提供现代化的用户界面。

初期遇到的技术挑战:

在项目开始阶段，我最初考虑使用 Qt 的传统 Widgets 系统来构建界面。然而，在实际开发过程中，我发现了一些显著的局限性:

首先是原生组件支持的限制。Qt Widgets 虽然提供了基本的 UI 组件，但对于现代化的界面设计需求而言显得较为传统。想要实现圆角按钮、渐变背景、阴影效果等现代 UI 元素需要大量的自定义绘制代码，开发效率较低。

其次是动画效果实现的复杂性。在我的设计构想中，希望为用户界面添加一些平滑的过渡动画，比如页面切换时的滑动效果、按钮悬停时的缩放效果、训练过程中的进度动画等。然而，使用传统的 Widgets 系统实现这些动画效果需要编写大量的动画控制代码，不仅开发复杂度高，而且最终效果往往不够理想。

最后是整体美观性的考虑。现代用户对应用程序的视觉体验要求越来越高，期望看到简洁、美观、具有现代感的界面设计。传统的 Widgets 系统虽然功能完备，但在视觉效果方面相对传统，难以实现类似移动应用或现代 Web 应用那样的视觉效果。

技术方案的转变:

在遇到上述问题后，我开始思考是否有更好的解决方案。回想起我在 Web 前端开发中

的经验，现代 Web 技术栈（如 HTML5、CSS3、JavaScript）在实现美观界面和流畅动画方面具有天然的优势。CSS3 的动画属性、过渡效果、弹性布局等特性能够轻松实现各种视觉效果，而且代码简洁、易于维护。

这让我产生了一个想法：能否在 Qt 开发中借鉴 Web 前端的开发模式，使用类似的声明式语法来构建界面，同时享受 Web 技术在视觉效果方面的优势？

经过调研，我发现 Qt 提供了 Qt Quick 技术栈，它采用 QML（Qt Meta-Object Language）作为声明式的界面描述语言。QML 的语法类似于 JSON，同时融合了 JavaScript 的动态特性和 CSS 的样式概念。更重要的是，Qt Quick 原生支持各种动画效果、现代化的 UI 组件，能够轻松实现我所期望的视觉效果。

最终技术选择:

综合考虑开发效率、视觉效果和维护性，我最终决定采用 Qt Quick + QML 的技术方案来实现图形化界面。这种技术组合具有以下优势：

- **声明式语法**：QML 采用声明式语法，界面结构清晰，易于理解和维护，类似于 HTML 的标记语言特性。
- **丰富的动画支持**：Qt Quick 内置了强大的动画系统，可以轻松实现各种过渡效果、缓动动画和状态切换动画。
- **现代化组件**：提供了大量现代化的 UI 组件，支持 Material Design 和其他现代设计风格。
- **灵活的布局系统**：支持弹性布局、锚点布局等现代布局方式，能够适应不同屏幕尺寸。
- **JavaScript 集成**：支持 JavaScript 脚本，可以在 QML 中处理复杂的界面逻辑。
- **C++ 后端集成**：能够无缝地与 C++ 后端代码集成，通过信号槽机制实现前后端通信。

通过这种技术方案，我能够在保持 C++ 后端强大计算能力的同时，实现现代化、美观且具有良好用户体验的前端界面，为整个 BP 神经网络项目提供了完整的用户交互体验。

6.2 主界面设计

基于 Qt Quick + QML 的技术方案，我设计了一个现代化的主界面，整体风格采用简约的设计理念，使用渐变色背景和圆角卡片式布局，力求在保持功能性的同时提供良好的视觉体验。

首先，在风格上，我采用了现代化的扁平设计风格，避免了过多的装饰元素，注重内容的呈现和用户体验的流畅性。整体色彩搭配以蓝紫色渐变为主色调，辅以白色卡片，营造出更符合现代计算机设计的简约感。

主界面的设计围绕几个核心元素展开：

自定义标题栏设计：考虑到界面风格的统一性，我去掉了系统默认的标题栏，设计了自定义的标题栏。

主体内容区域：主体区域采用垂直居中的布局方式，包括合肥工业大学校徽、课设标题、学生信息和功能按钮等。

渐变背景设计：主界面采用从蓝色到浅紫的径向渐变背景，使整个界面看起来更有层次感和现代感。

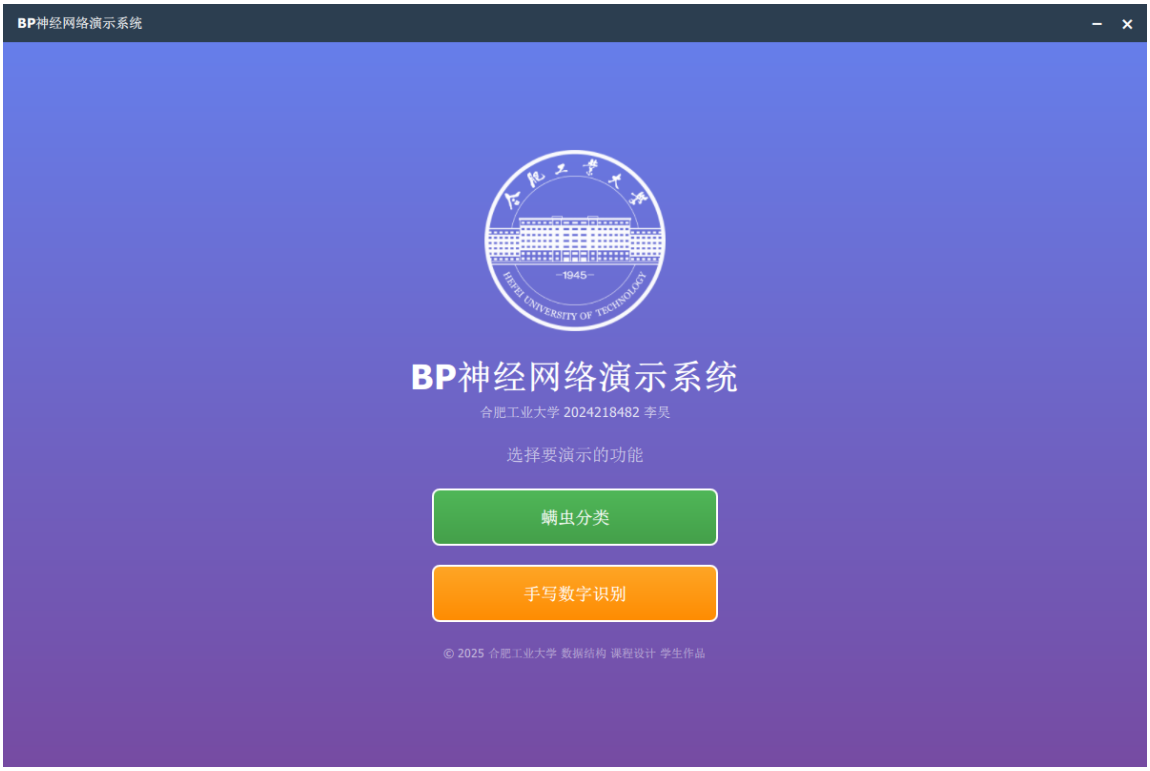

图 6.1　主界面设计效果图

6.3 螨虫分类界面实现 & 神经网络动画效果设计

螨虫分类界面是整个项目中最具视觉效果的部分，我设计了一个功能丰富且直观的界面，能够实时展示神经网络的训练过程和预测结果。整个界面主要包含以下几个核心组件：

数据可视化坐标轴设计：

界面左侧是一个标准的二维坐标系，横轴表示触角长度，纵轴表示翅膀长度。坐标轴上预先标注了题目提供的 15 个初始数据点：9 个红色圆点代表 A 类蠓虫数据，6 个蓝色圆点代表 B 类蠓虫数据。坐标轴的范围设置为 1.0 到 2.2，能够完全覆盖数据分布范围，同时为用户交互预留了充足的空间。

坐标轴支持多种交互操作：

- **左键点击**：在坐标轴上添加橙色的测试点，用于预测功能
- **右键点击**：弹出选择对话框，允许用户选择 A 类或 B 类，然后在点击位置添加对应的训练数据
- **决策边界显示**：通过” 显示决策边界” 按钮可以在坐标轴上绘制紫色的决策边界曲线

神经网络可视化设计：

界面右侧展示了一个 2-3-1 结构的神经网络示意图，包含输入层（2 个神经元)、隐藏层(3 个神经元）和输出层（1 个神经元)。每个神经元用圆形表示，神经元之间的连接用直线表示。整个神经网络图不仅是静态的结构展示，更重要的是能够动态展示网络的计算过程。

功能按钮设计：

界面底部设计了四个主要的功能按钮：

训练按钮：点击后启动神经网络的训练过程。系统会首先随机初始化网络权重，然后使用题目提供的 15 个数据点进行多轮训练。训练过程中会显示当前的损失值和准确率，让用户了解网络的学习进度。训练完成后，神经网络就具备了基本的分类能力。

预测按钮：用于对用户在坐标轴上添加的橙色测试点进行分类预测。点击后会触发前向传播动画，最终在界面上显示预测结果（A 类或 B 类）以及预测的置信度。

反向传播按钮：当用户通过右键添加新的训练数据后，可以点击此按钮让网络根据新数据进行学习。系统会计算新数据的损失，并通过反向传播算法更新网络参数，同时展示反向传播动画效果。

显示决策边界按钮：点击后会在坐标轴上绘制当前神经网络学习到的决策边界。决策边界以紫色曲线的形式显示，直观地展示了网络如何划分不同类别的区域。用户可以通过观察决策边界来理解网络的分类逻辑。

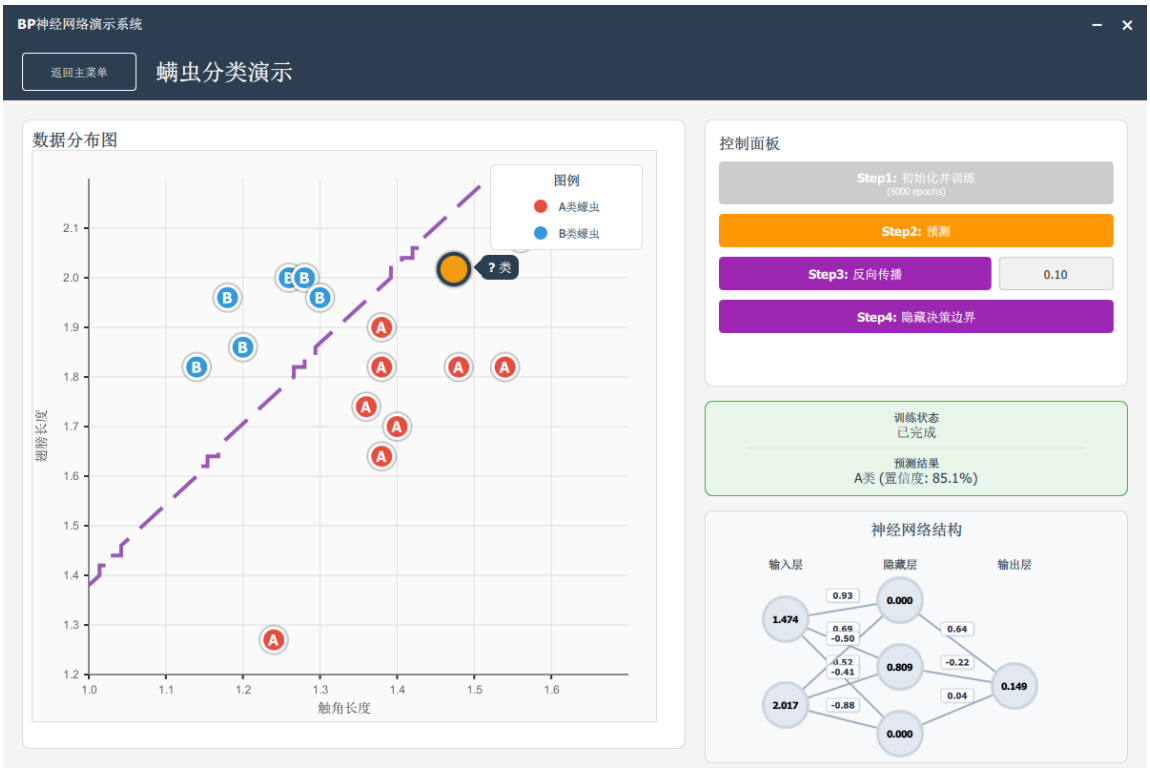

图 6.2　蠕虫分类界面设计效果图，包括测试点预测和决策边界显示

为了让用户更直观地理解神经网络的工作原理，我为神经网络设计了丰富的动画效果：

前向传播动画效果：当用户点击” 预测” 按钮时，神经网络会展示前向传播的动画过程。动画从输入层开始，每个被激活的神经元会呈现以下视觉效果：

- 神经元圆圈会逐渐变大，表示正在进行计算
- 神经元内部填充为蓝色，突出显示当前活跃状态
- 神经元边框变为蓝色，代表前向传播过程
- 连接线会呈现流动的蓝色效果，模拟信号传递过程

动画按照实际的计算顺序进行：首先激活输入层神经元，然后信号传递到隐藏层，隐藏层神经元依次被激活，最后信号到达输出层。整个过程持续约 2-3 秒，让用户能够清楚地观察到数据在网络中的流动轨迹。

反向传播动画效果：当用户添加新的训练数据并点击” 反向传播” 按钮时，神经网络会展示反向传播的动画过程。与前向传播相比，反向传播动画具有不同的视觉特征：

- 动画从输出层开始，向输入层方向进行
- 神经元边框变为红色，代表反向传播过程

- 连接线呈现流动的红色效果，表示梯度信息的反向传递
- 神经元会出现略微的闪烁效果，表示权重正在更新

反向传播动画同样按照实际的计算顺序进行: 从输出层计算误差开始，逐层向前传播误差信息，最后更新所有层的权重和偏置。

(a) 前向传播　　(b) 反向传播

图 6.3　前向传播和反向传播的动画示例

6.4 手写数字识别界面实现 & 手写数字图像处理

手写数字识别界面是整个项目中技术挑战最大的部分，需要实现从用户手写输入到神经网络识别的完整流程。整个界面设计围绕用户体验和技术实现的平衡展开，既要提供直观的交互方式，又要确保图像处理的准确性。

界面组件设计:

手写数字识别界面主要包含以下几个核心组件:

手写画板区域: 界面左侧是一个 280×280 像素的 Canvas 手写画板，采用深灰色背景配以白色笔迹。画板支持鼠标拖拽绘制，笔触粗细设置为 8 像素，能够模拟真实的手写体验。Canvas 组件通过 QML 的 Canvas 类型实现，支持实时绘制和路径记录。

模型状态面板: 界面右上角显示当前加载的模型状态。系统启动时会自动检测程序目录下的模型文件（mnist_model.bin），如果模型文件存在则显示” 模型已加载” 状态，否则显示” 模型未找到” 的警告信息。这种设计确保用户能够及时了解系统的准备状态。

处理后图像展示区域: 界面右侧中部展示经过预处理后的 28×28 像素图像。这个小尺寸的图像正是输入神经网络的实际数据，用户可以直观地看到系统如何将手写输入转换为标准化的网络输入格式。

识别结果面板：界面右下角展示识别结果，包括最终的预测数字（以大字体突出显示）和 0-9 各个数字的置信度条形图。置信度以百分比形式显示，帮助用户理解模型的预测确定性。

功能按钮：界面底部提供” 识别” 和” 清除” 两个主要功能按钮。识别按钮触发图像处理和神经网络推理过程，清除按钮重置画板和所有显示内容。

图 6.4　手写数字识别界面效果图，展示了手写数字 3 的识别过程

图像压缩和处理技术实现：

在实践中，我发现模型在 MNIST 数据集上的准确率非常高，但是一旦应用到手写面板上面，效果就不尽人意。经过分析后，我发现问题出在图像处理方面，一般来讲，MNIST 数据集里面的数据通常在图像正中央，且不会占满整个屏幕，并且一般比较光滑，而我直接将面板上的图像压缩得到的图像则不符合这些要求。因此，我们需要对图像进行预处理操作，我设计了一套完整的图像预处理流水线来确保输入质量：

智能边界框检测：

首先需要自动识别手写内容的有效区域，避免大量空白像素影响后续处理。算法遍历整个 280×280 图像，找到所有像素值大于检测阈值（50）的像素点，然后计算这些点的最小和最大坐标，形成最小边界框（Minimum Bounding Box）。

```
QRect ImageProcessor::findBoundingBox(const QImage& image, int threshold) {
    int minX = image.width(), maxX = 0;
    int minY = image.height(), maxY = 0;
    bool hasContent = false;

    for (int y = 0; y < image.height(); ++y) {
        for (int x = 0; x < image.width(); ++x) {
            QRgb pixel = image.pixel(x, y);
            int gray = qGray(pixel);

            if (gray > threshold) {  // 检测到内容
                minX = qMin(minX, x);
                maxX = qMax(maxX, x);
                minY = qMin(minY, y);
                maxY = qMax(maxY, y);
                hasContent = true;
            }
        }
    }

    if (!hasContent) {
        return QRect();  // 没有检测到内容
    }

    return QRect(minX, minY, maxX - minX + 1, maxY - minY + 1);
}
```

这种边界框检测算法能够自动适应不同大小和位置的手写内容，确保后续处理只关注有效的笔迹区域。

正方形化处理:

由于手写数字的形状可能是矩形（如数字 1 偏高瘦、数字 0 偏正方），直接缩放会导致纵横比失真。为了保持数字的原始比例，需要将边界框扩展为正方形区域。

算法计算边界框的宽度和高度，取较大值作为正方形的边长，然后将内容居中放置在正方形区域内。这样既保证了数字形状不会被拉伸变形，又为后续的标准化处理提供了统一的输入格式。

```
QRect ImageProcessor::makeSquare(const QRect& bbox, int padding) {
    int size = qMax(bbox.width(), bbox.height()) + 2 * padding;
    int centerX = bbox.center().x();
    int centerY = bbox.center().y();

    return QRect(centerX - size/2, centerY - size/2, size, size);
}
```

padding 参数添加了适当的边距，确保数字不会紧贴边界，这有助于后续的模糊处理和

神经网络识别。

高斯模糊处理:

手写输入往往存在锯齿状边缘和噪声，这与训练数据中平滑的 MNIST 数字存在差异。为了缩小这种差距，我使用高斯模糊算法对图像进行平滑处理。

高斯模糊通过卷积核对图像进行滤波，使边缘更加平滑自然。模糊半径设置为 1-2 像素，既能有效平滑边缘，又不会过度模糊导致数字特征丢失。

```
QImage ImageProcessor::applyGaussianBlur(const QImage& image, double radius) {
    QImage result = image;

    // 创建高斯核
    int kernelSize = 2 * (int)ceil(2 * radius) + 1;
    std::vector<std::vector<double>> kernel(kernelSize,
                                            std::vector<double>(kernelSize));

    double sigma = radius / 3.0;
    double sum = 0.0;
    int center = kernelSize / 2;

    // 计算高斯核权重
    for (int i = 0; i < kernelSize; ++i) {
        for (int j = 0; j < kernelSize; ++j) {
            double x = i - center;
            double y = j - center;
            kernel[i][j] = exp(-(x*x + y*y) / (2 * sigma * sigma));
            sum += kernel[i][j];
        }
    }

    // 归一化核
    for (int i = 0; i < kernelSize; ++i) {
        for (int j = 0; j < kernelSize; ++j) {
            kernel[i][j] /= sum;
        }
    }

    // 应用卷积
    for (int y = center; y < image.height() - center; ++y) {
        for (int x = center; x < image.width() - center; ++x) {
            double value = 0.0;
            for (int ky = 0; ky < kernelSize; ++ky) {
                for (int kx = 0; kx < kernelSize; ++kx) {
                    QRgb pixel = image.pixel(x + kx - center, y + ky - center);
                    value += qGray(pixel) * kernel[ky][kx];
                }
            }
            result.setPixel(x, y, qRgb(value, value, value));
        }
    }
```

```
43
44      return result;
45  }
```

尺寸处理和图像压缩:

经过前面的预处理后，需要将图像转换为 28×28 的 MNIST 标准格式。这个过程分为两个步骤:

第一步是将正方形化的内容缩放到 20×20 像素。选择 20×20 而不是直接缩放到 28×28 是为了在最终图像中保留边距，这与 MNIST 数据集的特征一致——数字内容通常不会占满整个 28×28 区域，而是留有适当的边距。

```
1  QImage ImageProcessor::scaleToSize(const QImage& image, int size) {
2      return image.scaled(size, size, Qt::KeepAspectRatio,
3                          Qt::SmoothTransformation);
4  }
```

第二步是将 20×20 的内容放置在 28×28 图像的中央。具体来说，将 20×20 图像放置在 (4, 4) 到 (24, 24) 的位置，周围保留 4 像素的黑色边框。这种处理方式确保了与 MNIST 训练数据的格式一致性。

```
1  QImage ImageProcessor::centerInCanvas(const QImage& content, int canvasSize) {
2      QImage canvas(canvasSize, canvasSize, QImage::Format_RGB32);
3      canvas.fill(Qt::black);
4
5      int offsetX = (canvasSize - content.width()) / 2;
6      int offsetY = (canvasSize - content.height()) / 2;
7
8      QPainter painter(&canvas);
9      painter.drawImage(offsetX, offsetY, content);
10
11     return canvas;
12 }
```

这套完整的图像处理流水线确保了用户手写输入能够被准确地转换为神经网络可以识别的标准格式。通过智能边界框检测、正方形化处理、高斯模糊、分步缩放等技术，系统能够处理各种大小、位置和风格的手写数字，大大提高了识别的准确性和鲁棒性。

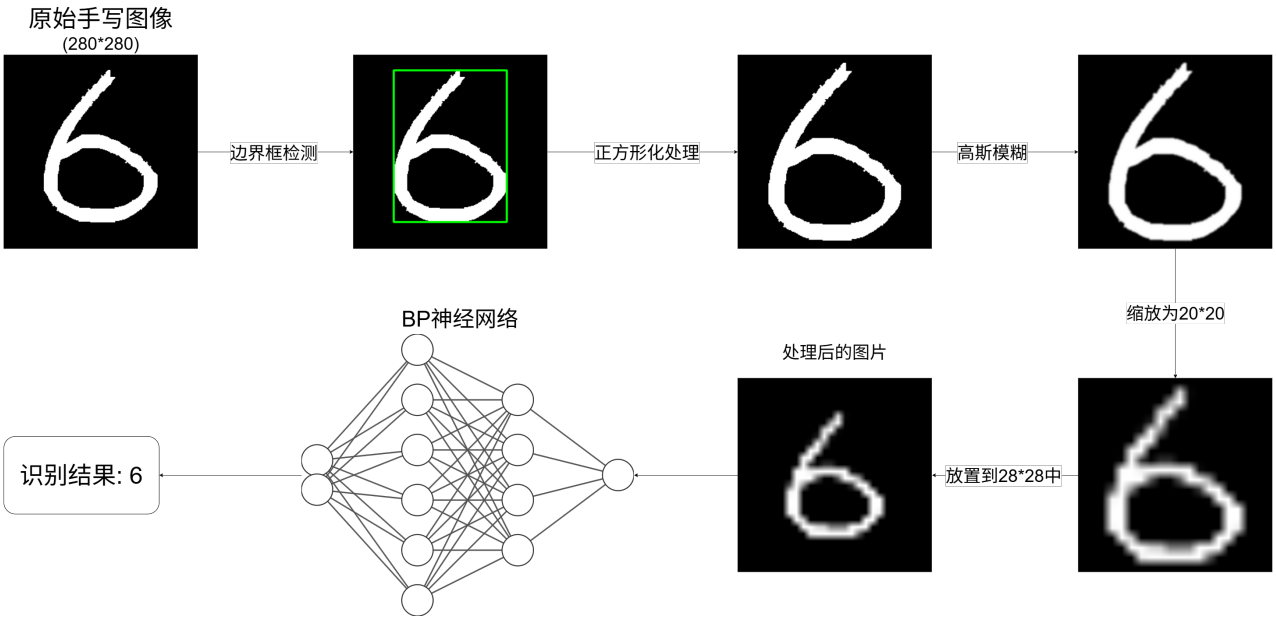

图 6.5　手写数字处理流程图

以上，我们完成了手写数字识别界面的设计和实现。用户可以通过手写输入数字，系统会自动处理图像并调用预训练的神经网络模型进行识别，最终展示预测结果和置信度。

7 总结与心得体会

本次课程设计以 BP 神经网络为核心，系统性地实现了从理论推导、C++ 代码开发到实际应用（螨虫分类与手写数字识别）的完整流程，并结合 Qt 技术开发了可视化界面。

今年的课程设计选题中出现了 BPNN、CNN、GAN 等深度学习算法，这让我感到相当意外。我个人对人工智能抱有浓厚兴趣，上学期加入了学校一个与人工智能科研相关的实验室，学习了机器学习的基础知识。但由于课业压力，我鲜有时间深入钻研底层算法推导，多是对许多机器学习和深度学习算法进行浮光掠影式的了解，留下一个基本印象。然而，在完成本次课设的过程中，我尝试完全独立地推导 BPNN 的算法。尽管花费了大量时间去理解，但我能清晰感受到自己对神经网络和人工智能的理解加深了。这种深刻的体验，是浮光掠影式的“科普”学习难以带来的。例如，在参加实验室考核时，我曾手动推导过 SVM（支持向量机）算法，即便时隔半年，其原理依旧记忆犹新；而其他仅略作了解的算法，则大多已模糊。这也让我深刻意识到，**在学习中，亲自动手推导、将所学付诸实践，才能真正内化知识**。这是我在本次课设中最大的收获。

此外，我对深度神经网络的理解也得到了进一步深化。以往，我虽听闻 ReLU、Sigmoid 等函数被用作激活函数，却不甚明了其具体作用。通过本次课设中的理论推导，我才真正意识到激活函数的核心作用在于引入非线性特性。鉴于现实世界中许多关系本质上是非线性的，这种能力的引入赋予了神经网络远超感知机线性拟合的强大表达力。同时，对反向传播算法的深入理解，也为我将来学习 CNN、RNN、Transformer 等更先进模型奠定了坚实基础。

其次，我深刻体会到**数据预处理**的重要性。在设计手写字体识别功能时，我发现模型在 MNIST 测试集上表现优异，准确率很高；然而，当直接识别自己手写的数字时，效果却不尽人人意。经过分析，症结在于图像处理：手写板输入的数字与 MNIST 数据集中的图像在特征上存在显著差异。为此，我投入精力研究并实施了一系列图像处理方法，力求使手写输入更接近 MNIST 数据特征，最终显著提升了模型的实际识别效果。这一经历不仅揭示了 BPNN 作为早期深度学习算法在应对原始输入变化时的某些局限性，更凸显了数据预处理在机器学习项目中的关键作用。

在课设实现中，理论推导出的算法往往显得简洁优美，但在将其转化为实际代码时，常会面临诸多工程细节挑战：数值稳定性、内存管理、性能优化、边界条件处理等。例如，在实现激活函数时需考虑数值溢出问题，进行矩阵运算时要注意内存对齐和缓存友好性，在批

量训练时则要合理组织数据结构以提高效率。这些工程细节在理论学习中往往被忽略，但在实际开发中却至关重要。

最后，**跨学科知识的融合**也让我受益匪浅。这个项目涉及了数学（线性代数、微积分、概率论)、计算机科学（数据结构、算法、面向对象编程)、软件工程（模块化设计、接口抽象)、前端开发（Qt、QML、界面设计）以及各类工具软件的运用（如使用 LaTeX 进行公式编辑与文档排版，利用 TikZ、drawio、matplotlib 绘制图表等）等多个领域的知识。通过这次实践，我深刻体会到现代项目的复杂性和交叉性，也更深刻地认识到培养综合运用知识解决实际问题的能力（而非仅仅停留在书本知识的学习）的重要性。

总的来说，虽然本次课设花费了我相当多的时间，但我感到过程非常充实且富有价值。这次经历不仅让我掌握了神经网络的核心算法和实现技术，更重要的是，它锻炼了我的工程实践能力和综合解决问题的素养。这些收获将为我未来在计算机领域的深入学习和研究奠定坚实的基础。