神经网络与 MNIST 手写识别简介

GitHub Copilot

2024 年 5 月 9 日

摘要

本文简要介绍了神经网络的基本原理和结构，以及如何训练神经网络。然后，我们将探讨神经网络在 MNIST 手写识别中的应用，并讨论如何使用 C++ 实现神经网络，包括模型的训练和评估。

目录

1 神经网络简介

1.1 神经网络的基本原理

神经网络是一种模拟人脑神经元工作方式的算法模型。它由大量的神经元（也称为节点）组成，这些神经元按照一定的结构组织在一起。每个神经元都有一组权重和一个偏置值，这些权重和偏置值可以在训练过程中进行调整。

每个神经元接收到来自其他神经元的输入，然后将这些输入与其权重相乘，加上偏置值，然后通过一个激活函数（如 sigmoid 函数或 ReLU 函数）进行处理，最后将结果输出给其他神经元。这个过程可以被看作是一种函数逼近，神经网络通过调整权重和偏置值，可以逼近任何复杂的函数。

神经网络的训练通常使用一种称为梯度下降的优化算法。在每次训练迭代中，都会计算损失函数（即网络预测结果和真实结果之间的差异）关于权重和偏置值的梯度，然后按照梯度的反方向更新权重和偏置值，以减小损失函数的值。

神经网络可以用来学习和识别模式，广泛应用于机器学习和人工智能领域，如图像识别、语音识别、自然语言处理等 [1]。

1.2 神经网络的结构

神经网络的基本结构是由多层神经元（也称为节点）组成的网络。这些层通常包括输入层、一个或多个隐藏层和输出层。

输入层是网络的最初层，它接收原始数据作为输入。每个神经元对应于一个输入特征，例如，在图像识别任务中，输入层的神经元可能对应于图像的像素值。

隐藏层位于输入层和输出层之间，它们的任务是从输入数据中提取有用的信息。每个隐藏层都由多个神经元组成，每个神经元都与上一层的所有神经元相连，并将其输出传递给下一层的所有神经元。隐藏层的神经元通常使用非线性激活函数，如 ReLU 或 sigmoid，这使得神经网络能够学习和表示非线性关系。

输出层是网络的最后一层，它产生网络的最终输出。在分类任务中，输出层的神经元通常对应于各个类别，每个神经元的输出表示输入属于该类别的概率。

神经网络的这种层次结构使得它能够学习和识别复杂的模式。通过增加隐藏层的数量和每层的神经元数量，神经网络可以表示更复杂的函数 [2].

1.3 神经网络的训练

神经网络的训练是一个迭代过程，包括前向传播和反向传播两个步骤。

在前向传播阶段，网络从输入层开始，数据通过每一层的神经元，每个神经元都会根据其权重（w）和偏置值（b）计算出一个输出值（a）。这个过程可以用下面的公式表示：

$$a = f(w \cdot x + b) \tag{1}$$

其中，x 是输入值，f 是激活函数。这个过程一直持续到输出层，最后输出层的神经元会生成网络的预测结果。

接下来，我们需要计算损失函数（L），它是网络预测结果和真实结果（y）之间的差异的度量。常用的损失函数包括均方误差（用于回归任务）和交叉熵（用于分类任务），可以用下面的公式表示：

$$L = \frac{1}{2}(y - a)^2 \quad \text{(均方误差)} \tag{2}$$

$$L = -y \log(a) - (1 - y) \log(1 - a) \quad \text{(交叉熵)} \tag{3}$$

在反向传播阶段，我们计算损失函数关于网络权重和偏置值的梯度，然后按照梯度的反方向更新权重和偏置值。这个过程是通过链式法则实现的，可以用下面的公式表示：

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial a} \frac{\partial a}{\partial w} \tag{4}$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial a} \frac{\partial a}{\partial b} \tag{5}$$

然后，我们按照梯度的反方向更新权重和偏置值：

$$w = w - \eta \frac{\partial L}{\partial w} \tag{6}$$

$$b = b - \eta \frac{\partial L}{\partial b} \tag{7}$$

其中，η 是学习率。

这个前向传播和反向传播的过程会反复进行，每次迭代都会更新网络的权重和偏置值，以减小损失函数的值。训练过程会持续进行，直到网络的预测结果达到满意的精度，或者达到预设的最大迭代次数 [4].

1.4 MNIST 数据集介绍

MNIST 数据集（Modified National Institute of Standards and Technology database）是一个广泛用于训练和测试机器学习模型的手写数字识别数据集。它由美国国家标准与技术研究院（NIST）的员工和美国的高中生手写的数字构成。

数据集包含 60000 个训练样本和 10000 个测试样本。每个样本都是一个 28x28 像素的灰度图像，代表了 0 到 9 的一个数字。每个像素的值在 0 到 255 之间，表示灰度级别。0 代表白色，255 代表黑色，其他值表示不同的灰度。这可以用以下公式表示：

$$I_{ij} = \frac{I_{ij}}{255} \tag{8}$$

其中，I_{ij} 是图像中第 i 行第 j 列的像素值。

除了图像数据外，MNIST 数据集还提供了每个图像对应的标签，即图像代表的数字。这些标签用于监督学习，帮助机器学习模型理解图像和数字之间的关系。标签是一个介于 0 到 9 的整数，可以用以下公式表示：

$$t = \text{one\_hot}(t) \tag{9}$$

其中，t 是图像的真实数字，$\text{one\_hot}(t)$ 是一个 10 维的向量，只有第 t 个元素为 1，其他元素为 0。

MNIST 数据集的一个重要特点是，它的图像数据已经进行了归一化和中心化处理。所有的图像都被缩放和平移，使得数字位于图像的中心，且有相同的尺度。这大大简化了后续的数据预处理工作。

此外，MNIST 数据集由于其规模适中、难度适中，被广泛用作机器学习的基准测试数据集。通过在 MNIST 数据集上的性能，可以直观地比较不同机器学习模型的优劣 [3]。

1.5 神经网络在 MNIST 手写识别中的应用

神经网络是一种非常适合处理 MNIST 手写识别任务的模型。首先，我们可以将每个图像展平成一个 784 维的向量，然后将这个向量作为神经网络的输入。这可以用以下公式表示：

$$x = \text{flatten}(I) \tag{10}$$

其中，I 是一个 28x28 的图像，x 是一个 784 维的向量。

网络的输出层有 10 个神经元，每个神经元对应一个数字类别。这意味着网络的输出是一个 10 维的向量，每个元素代表对应数字的预测概率。这可以用以下公式表示：

$$y = \text{softmax}(Wx + b) \tag{11}$$

其中，W 是权重矩阵，b 是偏置向量，y 是输出向量。

在训练过程中，我们使用交叉熵作为损失函数，通过反向传播和梯度下降的方法来更新网络的权重和偏置值。交叉熵损失函数可以用以下公式表示：

$$L = -\sum_{i=1}^{10} t_i \log(y_i) \tag{12}$$

其中，t 是真实标签的 one-hot 编码，y 是网络的输出。

在每个训练迭代中，我们都会计算网络的预测结果和真实结果之间的差异，然后根据这个差异来调整网络的参数。这可以用以下公式表示：

$$W = W - \eta \frac{\partial L}{\partial W} \tag{13}$$

$$b = b - \eta \frac{\partial L}{\partial b} \tag{14}$$

其中，η 是学习率。

通过这种方式，神经网络可以学习到如何从图像像素值预测出对应的数字。在测试阶段，我们可以使用训练好的神经网络来预测新的手写数字图像，从而实现手写数字的自动识别 [3].

2 神经网络的实现

2.1 使用 C++ 和 TinyDNN 实现神经网络

TinyDNN 是一个使用 C++ 编写的深度学习框架，它的设计目标是简单、易用和高效。TinyDNN 不需要任何依赖项或安装过程，只需要包含头文件即可。我们可以从 GitHub 上下载 TinyDNN 的源代码，地址为：https://github.com/tiny-dnn/tiny-dnn。

在开始之前，我们需要确保我们的 C++ 编程环境已经准备好。我们还需要将 MNIST 数据集的四个 ubyte 文件放入项目目录的 data 文件夹下。

下载 TinyDNN 的源代码后，我们会得到一个名为"tiny_dnn" 的文件夹，这个文件夹包含了 TinyDNN 的所有源代码。我们需要将这个文件夹放入我们的项目目录中。在我们的代码中，我们可以通过包含"tiny_dnn/tiny_dnn.h" 头文件来使用 TinyDNN。

例如，如果我们的项目目录结构如下：

```
/MNIST
    /data
        t10k-images-idx3-ubyte
        t10k-labels-idx1-ubyte
        train-images-idx3-ubyte
        train-labels-idx1-ubyte
    /tiny_dnn
        tiny_dnn.h
        ...
    main.cpp
```

那么，在"main.cpp" 中，我们可以通过以下方式包含 TinyDNN：

```
#include "tiny_dnn/tiny_dnn.h"
```

这样，我们就可以在我们的代码中使用 TinyDNN 了。

2.2 模型训练

使用 TinyDNN 进行模型训练的基本步骤如下：

1. 定义网络结构：我们可以使用 TinyDNN 提供的 API 定义我们的神经网络结构，包括层数、每层的神经元数量、激活函数等。例如，以下代码创建了一个包含三层全连接层的神经网络：

```
network<sequential> net;
net << fully_connected_layer<sigmoid>(784, 50)
    << fully_connected_layer<sigmoid>(50, 50)
    << fully_connected_layer<sigmoid>(50, 10);
```

2. 加载数据：我们可以使用 TinyDNN 提供的函数加载 MNIST 数据集。例如，以下代码加载了 MNIST 的训练集和测试集：

```
std::vector<label_t> train_labels, test_labels
    ;
std::vector<vec_t> train_images, test_images;

parse_mnist_labels("data/train-labels-idx1-
    ubyte", &train_labels);
parse_mnist_images("data/train-images-idx3-
    ubyte", &train_images, -1.0, 1.0, 2, 2);
parse_mnist_labels("data/t10k-labels-idx1-
    ubyte", &test_labels);
parse_mnist_images("data/t10k-images-idx3-
    ubyte", &test_images, -1.0, 1.0, 2, 2);
```

3. 训练模型：我们可以使用 TinyDNN 提供的函数进行模型训练。我们可以设置训练的轮数（epochs）、批次大小（batch size）、学习率等参数。例如，以下代码使用 Adagrad 优化器和交叉熵损失函数进行训练：

```
int epochs = 20;
int batch_size = 10;
double learning_rate = 0.1;

adagrad optimizer;
net.train<cross_entropy>(optimizer,
    train_images, train_labels, batch_size,
    epochs);
```

2.3 模型评估

训练完成后，我们可以使用 TinyDNN 提供的函数对模型进行评估，包括计算模型在测试集上的准确率等。例如，以下代码计算了模型在测试集上的准确率：

```
result res = net.test(test_images, test_labels
    );
std::cout << "accuracy: " << res.num_success /
     static_cast<double>(res.num_total) << std
    ::endl;
```

在这段代码中，‘net.test‘函数计算了模型在测试集上的预测结果，并返回了一个‘result‘对象。‘result‘对象包含了预测的成功数（‘num_success‘）和总数（‘num_total‘）。然后，我们可以计算准确率，即成功数除以总数。

我们还可以使用其他的评估指标，例如混淆矩阵、精确率、召回率等，具体的评估指标应根据我们的任务和需求来选择。

2.4 分析 TinyDNN 的源代码

为了更深入地理解 TinyDNN 的工作原理，我们可以分析 TinyDNN 的源代码。TinyDNN 的源代码结构清晰，注释详细，是学习深度学习实现的好资源。我们可以从 GitHub 上下载 TinyDNN 的源代码，然后使用我们的 IDE 或文本编辑器打开和阅读。

例如，我们可以看一下 TinyDNN 中实现全连接层的源代码。全连接层是神经网络中最基本的一种层，每个神经元都与前一层的所有神经元相连。在 TinyDNN 中，全连接层的实现在‘fully_connected_layer.h‘文件中。

```
class fully_connected_layer : public layer
    {
    // ...
    fully_connected_layer(
        serial_size_t in_dim,
        serial_size_t out_dim,
        bool has_bias = true,
        backend_t backend_type = core::
            default_engine()
```

```
        ) : layer(std_input_order(has_bias), {
            vector_type::data}),
            params_(in_dim, out_dim, has_bias)
                ,
            kernel_fwd_(backend_type),
            kernel_back_(backend_type) {
            // ...
        }
        // ...
};
```

在训练模型时，我们通常使用梯度下降法来优化模型的参数。梯度下降法的基本公式是：

$$\theta = \theta - \alpha \nabla J(\theta)$$

其中，θ 是模型的参数，α 是学习率，$\nabla J(\theta)$ 是损失函数 J 关于参数 θ 的梯度。在每一步训练中，我们计算损失函数的梯度，然后用梯度乘以学习率更新模型的参数。这个过程在 TinyDNN 中的 ‘optimizer‘类中实现。

3 结论

神经网络是一种强大的机器学习模型，它能够学习和识别复杂的模式，广泛应用于图像识别、语音识别、自然语言处理等领域。通过调整神经元的权重和偏置值，神经网络可以逼近任何复杂的函数。神经网络的训练通常使用梯度下降算法，通过计算损失函数关于权重和偏置值的梯度，然后按照梯度的反方向更新权重和偏置值，以减小损失函数的值。

在 MNIST 手写识别任务中，神经网络表现出了优秀的性能。通过将图像的像素值作为输入，神经网络可以学习到如何从这些像素值预测出对应的数字。这种方法不仅准确率高，而且可以很好地处理新的、未见过的手写数字图像。

然而，神经网络也有其局限性。例如，神经网络的训练通常需要大量的数据和计算资源，而且神经网络的内部工作机制往往难以解释。此外，神经网络的性能在很大程度上取决于其结构（如层数、每层的神经元数量等）和

超参数（如学习率、正则化参数等）的选择，而这些选择往往需要大量的实验和经验。

尽管如此，随着计算能力的提高和数据量的增加，神经网络仍将在未来的机器学习和人工智能领域发挥重要的作用。

参考文献

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[2] Simon Haykin. *Neural Networks and Learning Machines*. Pearson Education, 2009.

[3] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[4] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.