作业 1 实验报告

2022244056-李泽康

1 网络结构设计

以下详细描述本文所设计的全连接神经网络的计算过程、参数定义、网络结构等信息。约定矩阵用正体粗体大写字母表示，向量用斜体粗体小写字母表示，标量用其他字母表示。

1.1 单层神经元

1.1.1 前向计算

首先观察单层神经网络中某一神经元的计算。图 1-1 为共K层的神经网络中第k层中某一神经元的结构。

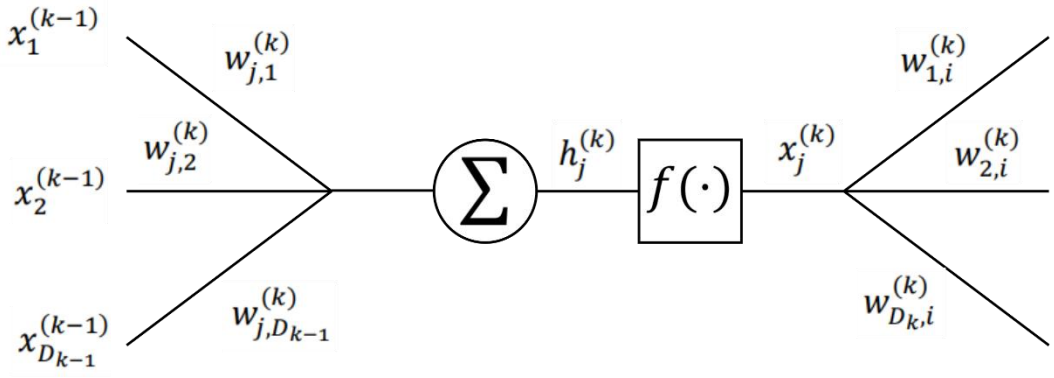

图 1-1　第k层某一神经元结构

令第k层神经元一个样本输入$\boldsymbol{x}^{(k-1)} \in \mathbb{R}^{1\times D_{k-1}}$，输出$\boldsymbol{x}^{(k)} \in \mathbb{R}^{1\times D_k}$，第$k$层神经元的计算如式（1-1）和式（1-2）所示。

$$\begin{aligned}\boldsymbol{h}^{(k)} &= \boldsymbol{x}^{(k-1)}\mathbf{W}^{(k)} + \boldsymbol{b}^{(k)} \\ &= \begin{bmatrix} x_1^{(k-1)} \\ x_2^{(k-1)} \\ \vdots \\ x_{D_{k-1}}^{(k-1)} \end{bmatrix}^T \begin{bmatrix} w_{1,1}^{(k)} & w_{1,2}^{(k)} & \cdots & w_{1,D_{k-1}}^{(k)} \\ w_{2,1}^{(k)} & w_{2,2}^{(k)} & \cdots & w_{2,D_{k-1}}^{(k)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{D_{k-1},1}^{(k)} & w_{D_{k-1},2}^{(k)} & \cdots & w_{D_{k-1},D_k}^{(k)} \end{bmatrix} + \begin{bmatrix} b_1^{(k)} \\ b_2^{(k)} \\ \vdots \\ b_{D_k}^{(k)} \end{bmatrix}^T \end{aligned} \tag{1-1}$$

$$\boldsymbol{x}^{(k)} = f\left(\boldsymbol{h}^{(k)}\right) = \begin{bmatrix} f(h_1^{(k)}) \\ f(h_2^{(k)}) \\ \vdots \\ f(h_{D_k}^{(k)}) \end{bmatrix}^T \tag{1-2}$$

其中$\mathbf{W}^{(k)} \in \mathbb{R}^{D_{k-1}\times D_k}$为第$k$层的权重矩阵，$\boldsymbol{b}^{(k)} \in \mathbb{R}^{1\times D_k}$为第$k$层的偏置向量，

$f(\cdot)$为第k层的激活函数，$\boldsymbol{h}^{(k)}$为经过激活函数前的隐变量。

在批量训练方式下，有N个样本输入$\mathbf{X}^{(k-1)} \in \mathbb{R}^{N\times D_{k-1}}$，输出$\mathbf{X}^{(k)} \in \mathbb{R}^{N\times D_k}$，则式（1-1）和式（1-2）的矩阵如式（1-3）和式（1-4）所示。

$$\mathbf{H}^{(k)} = \mathbf{X}^{(k-1)}\mathbf{W}^{(k)} + \mathbf{B}^{(k)} \tag{1-3}$$

$$\mathbf{X}^{(k)} = f\left(\mathbf{H}^{(k)}\right) \tag{1-4}$$

其中$\mathbf{W}^{(k)} \in \mathbb{R}^{D_{k-1}\times D_k}$不变。$\mathbf{B}^{(k)} \in \mathbb{R}^{N\times D_k}$中每一行的值不变，但 Broadcast 为矩阵形式。单层神经元核心代码如图 1-2 和图 1-3 所示。

```
class Linear(object):
    def __init__(self, in_dim, out_dim, optimizer):
        super(Linear, self).__init__()
        self.optimizer = optimizer
        self.weights = np.zeros((in_dim, out_dim))  # 权重
        self.bias = np.zeros((out_dim,))  # 偏置
```

图 1-2　单层神经元参数定义核心代码

```
def __call__(self, x):
    self.input = x  # 输入
    self.output = np.dot(self.input, self.weights) + self.bias  # y = wx+b
    return self.output
```

图 1-3　单层神经元前向计算核心代码

1.1.2 梯度下降

设神经网络对输入样本$\boldsymbol{x} \in \mathbb{R}^{D_0\times D}$计算最终输出$\boldsymbol{y} \in \mathbb{R}^{D_k\times D}$（此处$\boldsymbol{x}$和$\boldsymbol{y}$用列向量表示，为了推导形式上的方便，则相应的权重矩阵为转置形式即$\mathbf{W}^{(k)} \in \mathbb{R}^{D_k\times D_{k-1}}$），然后计算样本目标值与输出值的目标函数$E$。这里为了分析先将$E$取为 MSE，如式（1-5）所示。

$$E = \frac{1}{2}\sum_{i=1}^{D_K}(\bar{y}_i - y_i)^2 = \frac{1}{2}(\bar{\boldsymbol{y}} - \boldsymbol{y})^T(\bar{\boldsymbol{y}} - \boldsymbol{y}) \tag{1-5}$$

则使用梯度下降法对神经网络中的每一个权重进行更新的方式如式（1-6）所示。

$$w_{ji}^{(k)}(s+1) = w_{ji}^{(k)}(s) - \eta\frac{\partial E}{\partial w_{ji}^{(k)}} \tag{1-6}$$

其中η为学习率，s是训练更新次数，需要计算E对每一个权重$w_{ji}^{(k)}$的偏导数。

对每一个提交给神经网络的样本用式（1-1）对全体权值进行一次更新，直到所有样本的误差值都小于一个预设的阈值，此时可以认为训练结束。训练的关键问题是如何计算每一个$\frac{\partial E}{\partial w_{ji}^{(k)}}$。

1.1.3 反向传播

根据上述定义的第k层神经元的计算式，首先对第k层第j个神经元定义一个值为$\delta_j^{(k)}$，如式（1-7）所示。

$$-\delta_j^{(k)} = \frac{\partial E}{\partial h_j^{(k)}} \tag{1-7}$$

将$\delta_j^{(k)}$定义为E对第k层第j个神经元的隐变量的偏导数的相反数。根据求导链式法则，有式（1-8）：

$$\frac{\partial E}{\partial w_{ji}^{(k)}} = \frac{\partial E}{\partial h_j^{(k)}} \frac{\partial h_j^{(k)}}{\partial w_{ji}^{(k)}} \tag{1-8}$$

将$\frac{\partial h_j^{(k)}}{\partial w_{ji}^{(k)}}$展开，如式（1-9）所示。

$$\frac{\partial h_j^{(k)}}{\partial w_{ij}^{(k)}} = \frac{\partial}{\partial w_{ji}^{(k)}} \left(\sum_{s=1}^{n_{k-1}} w_{js}^{(k)} x_s^{(k-1)} \right) = x_i^{(k-1)} \tag{1-9}$$

结合式（1-7）有式（1-10）：

$$\frac{\partial E}{\partial w_{ji}^{(k)}} = -\delta_j^{(k)} x_i^{(k-1)} \tag{1-10}$$

可见有了$\delta_j^{(k)}$就能计算E对任一权重$w_{ji}^{(k)}$的偏导数。接下来的问题就是如何计算$\delta_j^{(k)}$。采用一种类似数学归纳法的方法。首先计算第K层（输出层）第j个神经元的$\delta_j^{(K)}$。如式（1-11）所示。

$$\begin{aligned} -\delta_j^{(K)} &= \frac{\partial E}{\partial h_j^{(K)}} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial h_j^{(K)}} = \frac{\partial E}{\partial y_j} f'\left(h_j^{(K)}\right) \\ &= \frac{\partial}{\partial y_j} \left(\frac{1}{2} \sum_{i=1}^{D_K} (\bar{y}_i - y_i)^2 \right) f'\left(h_j^{(K)}\right) = -\left(\bar{y}_j - y_j\right) f'\left(h_j^{(K)}\right) \end{aligned} \tag{1-11}$$

$f'(\cdot)$表示激活函数$f(\cdot)$的导函数。式（1-11）表示了推导过程，其结论是：对于输出层 (第K 层)第j个神经元来说，有式（1-12）：

$$\delta_j^{(K)} = \left(\bar{y}_j - y_j\right) f'\left(h_j^{(K)}\right) \tag{1-12}$$

$\delta_j^{(K)}$等于目标值$\bar{y}_j$与输出y_j之差乘上$f(\cdot)$在$h_j^{(K)}$的导数。现在推导某个隐藏层，即第k层第j个神经元的$\delta_j^{(k)}(k<K)$。将第$k+1$层的全体$h_j^{(k+1)}$值视作一个向量$\mathbf{h}^{(k+1)}$。连续使用链式法则，有式（1-13）：

$$-\delta_j^{(k)}=\frac{\partial E}{\partial h_j^{(k)}}=h'g'f'=\frac{\partial E}{\partial h^{(k+1)}}\frac{\partial h^{(k+1)}}{\partial x_j^{(k)}}\frac{\partial x_j^{(k)}}{\partial h_j^{(k)}} \tag{1-13}$$

等号右侧第一项是一个$R^{D_{k+1}}\to R$函数的导数。它是$1\times D_{k+1}$元向量。它的第i个元素如式（1-14）所示。

$$\frac{\partial E}{\partial h_i^{(k+1)}}=-\delta_i^{(k+1)} \tag{1-14}$$

第二项是一个$R\to R^{D_{k+1}}$函数的导数。它是$D_{k+1}\times 1$元向量。它的第i个元素如式（1-15）所示。

$$\frac{\partial h_i^{(k+1)}}{\partial x_j^{(k)}}=\frac{\partial}{x_j^{(k)}}\left(\sum_{s=1}^{D_k}w_{is}^{(k+1)}x_s^{(k)}\right)=w_{ij}^{(k+1)} \tag{1-15}$$

最后一项是激活函数$f(\cdot)$在$h_j^{(k)}$的偏导数。结合式（1-13）、（1-14）、（1-15）得到式（1-16）：

$$\begin{aligned}-\delta_j^{(k)}&=\frac{\partial E}{\partial h_j^{(k)}}\\&=\left(-\delta_1^{(k+1)},-\delta_2^{(k+1)},\cdots,-\delta_{D_{k+1}}^{(k+1)}\right)\begin{pmatrix}w_{1j}^{(k+1)}\\w_{2j}^{(k+1)}\\\vdots\\w_{n_{k+1}j}^{(k+1)}\end{pmatrix}f'\left(h_j^{(k)}\right)\\&=-\left(\sum_{s=1}^{D_{k+1}}\delta_s^{(k+1)}w_{sj}^{(k+1)}\right)f'\left(h_j^{(k)}\right)\end{aligned} \tag{1-16}$$

上式（1-16）是推导过程，最终得到$\delta_j^{(k)}$的表达式（1-17）：

$$\delta_j^{(k)}=\left(\sum_{s=1}^{D_{k+1}}\delta_s^{(k+1)}w_{sj}^{(k+1)}\right)f'\left(h_j^{(k)}\right) \tag{1-17}$$

综合上述推导过程，可得反向传播算法如下：

- **反向传播阶段：**

$$\begin{cases} \delta_j^{(K)} = (\bar{y}_j - y_j) f'\left(h_j^{(K)}\right) \\ \delta_j^{(k)} = \left(\sum_{s=1}^{D_{k+1}} \delta_s^{(k+1)} w_{sj}^{(k+1)}\right) f'\left(h_j^{(k)}\right), k < K \end{cases} \tag{1-18}$$

- **权值更新阶段:**

$$w_{ji}^{(k)}(s+1) = w_{ji}^{(k)}(s) - \eta \frac{\partial E}{\partial w_{ji}^{(k)}} = w_{ji}^{(k)}(s) + \eta \delta_j^{(k)} x_i^{(k-1)} \tag{1-19}$$

使用矩阵形式表示的反向传播算法如式（1-20）所示。

$$\begin{aligned} \Delta^{(k)} &= \left(\delta_1^{(k)}, \delta_2^{(k)}, \cdots, \delta_{n_k}^{(k)}\right) = \left(-\frac{\partial E}{\partial v_1^{(k)}}, -\frac{\partial E}{\partial v_2^{(k)}}, \cdots, -\frac{\partial E}{\partial v_{n_k}^{(k)}}\right) \\ &= \Delta^{(k+1)} \begin{pmatrix} w_{11}^{(k+1)} & w_{12}^{(k+1)} & \cdots & w_{1n_k}^{(k+1)} \\ w_{21}^{(k+1)} & w_{22}^{(k+1)} & \cdots & w_{2n_k}^{(k+1)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_{k+1}1}^{(k+1)} & w_{n_{k+1}2}^{(k+1)} & \cdots & w_{n_{k+1}n_k}^{(k+1)} \end{pmatrix} \\ &\cdot \begin{pmatrix} f'\left(v_1^{(k)}\right) & 0 & \cdots & 0 \\ 0 & f'\left(v_2^{(k)}\right) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & f'\left(v_{n_k}^{(k)}\right) \end{pmatrix} \\ &= \Delta^{(k+1)} \mathbf{W}^{(k+1)} \mathbf{F}^{(k)} \end{aligned} \tag{1-20}$$

如式（1-1）所示，第k层$(k < K)$全体$\delta_j^{(k)}$值组成的向量$\Delta^{(k)}$可由本层的激活函数导数对角阵$\mathbf{F}^{(k)}$、第$k+1$层的$\Delta^{(k+1)}$和权值矩阵$\mathbf{W}^{(k+1)}$计算得到。输出层(第 K 层)的$\Delta^{(K)}$如式（1-21）计算:

$$\begin{aligned} \Delta^{(K)} &= \left(\bar{y}_1 - y_1, \bar{y}_2 - y_2, \cdots, \bar{y}_{D_K} - y_{D_K}\right) \\ &= (\bar{\mathbf{y}} - \mathbf{y})^T \mathbf{F}^{(K)} \end{aligned} \tag{1-21}$$

可以看到隐藏层$\Delta^{(k)}$的计算利用了下一层的$\Delta^{(k+1)}$。一个训练样本$\boldsymbol{x}$"正向"通过网络计算输出$\boldsymbol{y}$。之后“反向"逐层计算$\Delta^{(k)}$更新权值，并将$\Delta^{(k)}$向前一层传播。所谓“反向"传播就是$\Delta^{(k)}$的传播。以上推导没有包括神经元的偏置。把偏置看成一个连接到常量 1 的连接上的权值即可。

根据上述分析，单层神经元的反向传播核心代码、梯度更新核心代码（结合不同的优化器一起使用）分别如图 1-4、图 1-5 所示。

```python
def backward(self, grad):
    N = self.input.shape[0]
    data_grad = np.dot(grad, self.weights.T)  # 当前层的梯度, dy/dx = W^T
    self.weights_grad = np.dot(self.input.T, grad) / N  # 当前层权重的梯度
    self.bias_grad = np.sum(grad, axis=0) / N  # 当前层偏置的梯度
    return data_grad
```

图 1-4　单层神经元反向传播核心代码

```python
def step(self):
    lr_weights = 0
    lr_bias = 0
    if self.optimizer.__class__.__name__ == 'SGD':
        lr_weights = self.optimizer(self.weights_grad)
        lr_bias = self.optimizer(self.bias_grad)
    elif self.optimizer.__class__.__name__ == 'Adam':
        lr_weights, self.w_m_t, self.w_v_t = self.optimizer(self.weights_grad, self.w_m_t, self.w_v_t)
        lr_bias, self.b_m_t, self.b_v_t = self.optimizer(self.bias_grad, self.b_m_t, self.b_v_t)
    self.weights += lr_weights
    self.bias += lr_bias
```

图 1-5　单层神经元梯度更新核心代码

一个经典的优化器即随机梯度下降优化器（SGD）如图 1-6 所示。

```python
class SGD(object):
    def __init__(self, lr=0.0001):
        super(SGD, self).__init__()
        self.lr = lr

    def __call__(self, grad):
        return -self.lr * grad
```

图 1-6　SGD 优化器代码

1.2 激活函数

激活函数使用 ReLU 函数，如图 1-7 和式（1-22）所示。

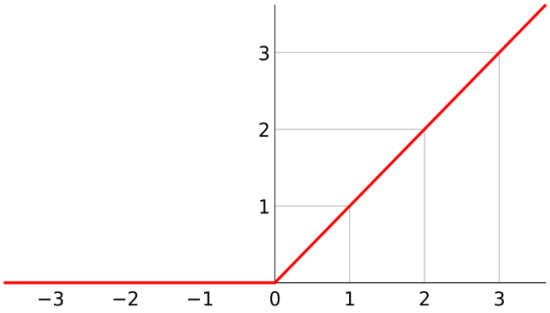

图 1-7　ReLU 函数

$$\mathrm{ReLU}(x) = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases} \tag{1-22}$$

ReLU 函数（实际使用类进行实现）完整代码如图 1-8 所示。

```python
class ReLu(object):
    def __init__(self):
        super(ReLu, self).__init__()

    def __call__(self, x):
        self.x = x  # (N x dim)
        self.output = np.maximum(0, x)
        return self.output  # (N x dim)

    def backward(self, grad):
        """
        f(x) = ReLu(x)
        f'(x) = 1 if x > 0
        """
        grad[self.x <= 0] = 0  # ReLu函数的梯度
        return grad

    def step(self):
        """没有参数需要更新，所以pass"""
        pass
```

图 1-8　ReLU 函数完整代码

1.3 输出层

用于进行回归任务的神经网络直接将最后一层神经元的输出标量值进行损失函数计算，而用于分类任务的神经网络在最后一层神经元的输出向量值后要接一个分类层，一般使用 Softmax 函数。

令网络在 Softmax 函数之前的输出为$\boldsymbol{z}^K \in \mathbb{R}^{1\times D_K}$，Softmax 函数如式（1-23）所示。

$$\mathrm{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{J} e^{z_j}} \tag{1-23}$$

Softmax 函数实际上是在 CrossEntropyLoss 代码中实现的，目的是为了计算方便，如图 1-9 所示。

```python
class CrossEntropyLoss(object):
    def __init__(self):
        super(CrossEntropyLoss, self).__init__()

    def __call__(self, y_predict, y_true):
        self.y_predict = y_predict
        y_exp = np.exp(self.y_predict)
        self.y_softmax = y_exp / np.sum(y_exp, axis=-1, keepdims=True)
```

图 1-9　CrossEntropyLoss 中 softmax 函数的前向计算核心代码

1.4 损失函数

回归任务使用 MSE（mean square error），分类任务使用 CrossEntropyLoss。回归任务中，令网络最后一层的输出为$\boldsymbol{z}^K \in \mathbb{R}^{1\times D_K}$,真实标签为$\hat{\boldsymbol{z}}^K \in \mathbb{R}^{1\times D_K}$，则真实标签与输出值的 MSE 的计算如式（1-24）所示。

$$Loss_{MSE} = \frac{1}{D_K}\sum_{i=1}^{D_K}(z_i - \hat{z}_i)^2 = \frac{1}{D_K}(\boldsymbol{z} - \hat{\boldsymbol{z}})(\boldsymbol{z} - \hat{\boldsymbol{z}})^T \tag{1-24}$$

由前文推导知，$Loss_{MSE}$的梯度计算如式（1-25）所示。

$$\frac{\partial E}{\partial z_j} = \frac{2}{D_K}\sum_{i=1}^{D_K}(\hat{z_i} - z_i)^2 \tag{1-25}$$

MSE 的完整代码如图 1-10 所示。

```python
class MSELoss(object):
    def __init__(self):
        super(MSELoss, self).__init__()

    def __call__(self, y_predict, y_true):
        self.y_predict = y_predict
        self.y_true = y_true
        loss = np.mean(np.mean(np.square(self.y_predict - self.y_true), axis=-1))  # 损失函数值
        return loss

    def backward(self):
        grad = 2 * (self.y_predict - self.y_true) / (self.y_predict.shape[0] * self.y_predict.shape[1])  # 损失函数关于网络输出的梯度
        return grad

    def step(self):
        """没有参数需要更新，所以pass"""
        pass
```

图 1-10　MSE 完整代码

分类任务中，记经过 Softmax 函数处理的概率值向量为$\boldsymbol{y} \in \mathbb{R}^{1\times D_K} = \text{Softmax}(\boldsymbol{z}^K)$，经过 one-hot 编码的真实标签为$\hat{\boldsymbol{y}} \in \mathbb{R}^{1\times D_K}$，$\boldsymbol{z}^K \in \mathbb{R}^{1\times D_K}$为神经网络最后一层输出值，且$\sum_{i=1} y_i = \sum_{i=1} \hat{y}_i = 1$。则 CrossEntropyLoss 的计算如式（1-26）所示。

$$Loss_{CE} = -\sum_{i=1}^{D_K}\hat{y}_i \ln y_i \tag{1-26}$$

前文提到，Softmax 函数整合在 CrossEntropyLoss 中，因此先计算 Softmax 函数的梯度，如式（1-27 所示）。

$$\begin{aligned}\frac{\partial \mathrm{S}(z_i)}{\partial z_j} &= \frac{\partial\left(\frac{e^{z_i}}{\sum_k e^{z_k}}\right)}{\partial z_j} \\ &= \frac{\partial e^{z_i}}{\partial z_j}\cdot\frac{1}{\sum_k e^{z_k}} - e^{z_i}\cdot\frac{\partial\sum_k e^{z_k}}{\partial z_j}\cdot\frac{1}{(\sum_k e^{z_k})^2} \\ &= \frac{\partial e^{z_i}}{\partial z_j}\cdot\frac{1}{\sum_k e^{z_k}} - e^{z_i}\cdot\frac{\partial e^{z_j}}{\partial z_j}\cdot\frac{1}{(\sum_k e^{z_k})^2} \\ &= \frac{\partial e^{z_i}}{\partial z_j}\cdot\frac{1}{\sum_k e^{z_k}} - e^{z_i}\cdot e^{z_j}\cdot\frac{1}{(\sum_k e^{z_k})^2} \\ &= \frac{\partial e^{z_i}}{\partial z_j}\cdot\frac{1}{\sum_k e^{z_k}} - S(z_i)\cdot S(z_j)\end{aligned} \tag{1-27}$$

上式分情况讨论：

- 当$i=j$时，$\frac{\partial e^{z_i}}{\partial z_j}=e^{z_i}$，所以上式等于：$S(z_i)-S(z_i)^2=S(z_i)\big(1-S(z_i)\big)$
- 当$i\neq j$时，$\frac{\partial e^{z_i}}{\partial z_j}=0$，所以上式等于：$-S(z_i)\cdot S(z_j)$

因此，$Loss_{CE}$又可以表达成式（1-28）：

$$Loss_{CE}=-\sum_{i=1}^{D_K}\hat{y}_i\ln\big(\mathrm{S}(z_i)\big) \tag{1-28}$$

开始计算$Loss_{CE}$的梯度。不失一般性，假设当前样本的类别为i，那么只有$\hat{y}_i$等于 1，其他$\hat{y}$都等于 0，所以式（1-28）可以简化为式（1-29）：

$$Loss_{CE}=-\ln\big(\mathrm{S}(z_i)\big) \tag{1-29}$$

注意上述需要对所有的$z_j, j=1,2,\dots,n$求导:

$$\frac{\partial\, Loss_{CE}}{\partial z_j}=-\frac{1}{S(z_i)}\cdot\frac{\partial \mathrm{S}(z_i)}{\partial z_j} \tag{1-30}$$

这个公式的右侧第二项就是 Softmax 函数的导数，上面已经算过了，所以可以直接分情况讨论:

当$i=j$时：

$$\begin{aligned}\frac{\partial\, Loss_{CE}}{\partial z_j} &= -\frac{1}{\mathrm{S}(z_i)}\cdot\frac{\partial \mathrm{S}(z_i)}{\partial z_j} \\ &= -\frac{1}{\mathrm{S}(z_i)}\cdot \mathrm{S}(z_i)\big(1-\mathrm{S}(z_i)\big) \\ &= \mathrm{S}(z_i)-1 \\ &= \mathrm{S}(z_j)-1\end{aligned} \tag{1-30}$$

当$i\neq j$，时：

$$\begin{aligned}\frac{\partial \text{ Loss}}{\partial z_j} &= -\frac{1}{\mathrm{S}(z_i)} \cdot \frac{\partial \mathrm{S}(z_i)}{\partial z_j} \\ &= -\frac{1}{\mathrm{S}(z_i)} \cdot \left(-\mathrm{S}(z_i) \cdot \mathrm{S}(z_j)\right) \\ &= \mathrm{S}(z_j) \\ &= \mathrm{S}(z_j) - 0\end{aligned} \tag{1-31}$$

最后之所以写成$\mathrm{S}(z_j) - 0$，是为了将公式写为向量化表达式时看起来更整齐。以向量化的方法表示标签$\boldsymbol{y}$，那么导数就不用再分情况写了，可以向量化地写为式（1-32）：

$$dz = S(z) - y \tag{1-32}$$

CrossEntropyLoss 完整代码如图 1-11 所示。

```python
class CrossEntropyLoss(object):
    def __init__(self):
        super(CrossEntropyLoss, self).__init__()

    def __call__(self, y_predict, y_true):
        self.y_predict = y_predict
        y_exp = np.exp(self.y_predict)
        self.y_softmax = y_exp / np.sum(y_exp, axis=-1, keepdims=True)
        # 避免数值溢出的等价式
        theta = self.y_predict - np.max(self.y_predict, axis=-1, keepdims=True)
        self.y_predict_logsoftmax = theta - np.log(np.sum(np.exp(theta), axis=-1, keepdims=True))
        self.y_true = y_true  # 必须是one-hot编码!
        loss = -np.sum(y_true * self.y_predict_logsoftmax) / self.y_true.shape[0]  # 损失函数值
        return loss

    def backward(self):
        grad = self.y_softmax - self.y_true / self.y_predict.shape[0]  # 损失函数关于网络输出的梯度
        return grad

    def step(self):
        """没有参数需要更新，所以pass"""
        pass
```

图 1-11　CrossEntropyLoss 完整代码

1.5 网络详细结构

用于拟合三角函数$y = AsinBx + CcosDx$的全连接神经网络详细结构如表 1-1 所示。

表 1-1　全连接神经网络详细结构(拟合三角函数)表

Layer name	Function	(Input dimension, Output dimension)
Linear1	$y = wx + b$	(1, 16)
ReLU1	ReLU(x)	(16, 16)
Linear2	$y = wx + b$	(16, 16)
ReLU2	ReLU(x)	(16, 16)

Linear3	$y = wx + b$	(16, 16)
ReLU3	$\mathrm{ReLU}(x)$	(16, 16)
Linear4	$y = wx + b$	(16, 16)
ReLU4	$\mathrm{ReLU}(x)$	(16, 16)
Linear5	$y = wx + b$	(16, 1)

用于拟合三角函数的全连接神经网络完整代码如图 1-12 所示。其中的backward(·)函数依据链式法则从最后一层反向计算每一层的梯度，而step(·)函数在所有梯度计算完毕后，动态更新每一层的参数。

```python
class ZeroNet(object):
    def __init__(self, optimizer=None):
        super(ZeroNet, self).__init__()
        self.network = [
            Linear(1, 16, optimizer=optimizer),
            ReLu(),
            Linear(16, 16, optimizer=optimizer),
            ReLu(),
            Linear(16, 16, optimizer=optimizer),
            ReLu(),
            Linear(16, 16, optimizer=optimizer),
            ReLu(),
            Linear(16, 1, optimizer=optimizer),
            # 分类用的softmax输出头靠CrossEntropyLoss完成
        ]

    def __call__(self, x):
        for layer in self.network:
            x = layer(x)
        return x

    def backward(self, grad):
        last_grad = grad.copy()
        for layer in self.network[::-1]:  # 倒序翻转
            last_grad = layer.backward(last_grad)
        return last_grad

    def step(self):
        for layer in self.network:
            layer.step()
```

图 1-12　全连接神经网络(拟合三角函数)完整代码

2 实验结果分析

2.1 拟合三角函数实验

本实验拟合的三角函数表示式为$y = 24\sin(3.2x) + 42\cos(2.3x)$。

2.1.1 数据采样

输入数据样本x从均匀分布中采样 1100 个点，采样范围为$[-2\pi, 2\pi]$。均匀分布表达式为：

$$\mathrm{U}(x) = \begin{cases} \dfrac{1}{b-a}, & a < x < b \\ 0, & \text{else} \end{cases}$$

因此x~U$(-2\pi, 2\pi)$。真实输出值$\hat{y} = 24\sin(3.2x) + 42\cos(2.3x)$。根据采样得到的数据样本对$(x, y)$绘制的三角函数曲线如图 2-1 所示。

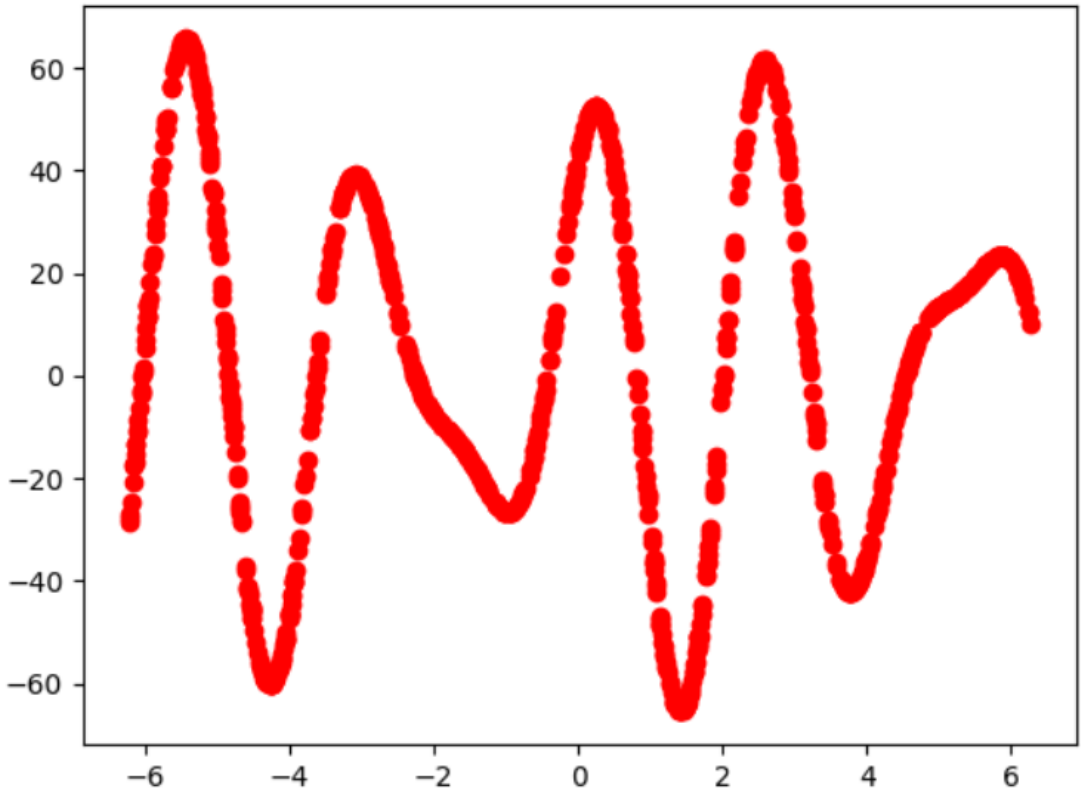

图 2-1　三角函数曲线图

对采样得到的数据进行随机划分，其中 1000 个点作为训练数据用于神经网络的训练，剩余的 100 个点作为测试数据用于训练完成后评估神经网络的预测准确性。将采样的所有数据保存为.csv 格式。数据生成的核心代码如图 2-2 所示。

```
np.random.seed(42)
Pi = math.pi
A = 24
B = 3.2
C = 42
D = 2.3

# train data
x_train = np.random.uniform(-2 * Pi, 2 * Pi, (1000, 1))
y_train = [A * math.sin(B * i) + C * math.cos(D * i) for i in x_train]
y_train = np.array(y_train).reshape(1000, 1)
# 保存数据
pd.DataFrame(x_train).to_csv('x_train.csv')
pd.DataFrame(y_train).to_csv('y_train.csv')
np.savetxt('x_train.csv', x_train, delimiter=",")
np.savetxt('y_train.csv', y_train, delimiter=",")
# test data
x_test = np.random.uniform(-2 * Pi, 2 * Pi, (100, 1))
y_test = [A * math.sin(B * i) + C * math.cos(D * i) for i in x_test]
y_test = np.array(y_test).reshape(100, 1)
# 保存数据
pd.DataFrame(x_test).to_csv('x_test.csv')
pd.DataFrame(y_test).to_csv('y_test.csv')
np.savetxt('x_test.csv', x_test, delimiter=",")
np.savetxt('y_test.csv', y_test, delimiter=",")
```

图 2-2　数据生成核心代码

2.1.2 训练过程

首先从数据文件中读取数据，读取数据的代码如图 2-3 所示。

```
def load_data(x_filename, y_filename):
    x_data = np.loadtxt(x_filename)
    x_data = x_data.reshape(x_data.shape[0], 1)
    y_data = np.loadtxt(y_filename)
    y_data = y_data.reshape(y_data.shape[0], 1)
    return x_data, y_data
```

图 2-3　读取数据代码

使用小批次训练的方式，设置 batch size=100，每次训练迭代时向模型输入 100 个数据。对数据进行批次切分及转换处理的具体代码见附件。当所有训练数据都已经输入到模型后定义为 1 个 epoch，设置 epochs=2000。优化器使用 SGD 优化器。

因为拟合三角函数为典型的预测任务，因此损失函数选用前文提到的 MSE。核心的训练过程如图 2-4 所示。讲训练数据依批次输入到模型中，计算预测值，然后计算损失值，并且计算每一层的梯度，最后更新每一层的当前参数值。

```python
def train(model, epochs, train_dataloader, x_test, y_test, criterion):
    for epoch in range(epochs):
        tmp_loss = 0
        for data in train_dataloader():
            x_train, y_train = data

            y_pred = model(x_train)
            loss = criterion(y_pred, y_train)
            tmp_loss += loss.item()
            loss_grad = criterion.backward()
            print('epoch: {}, loss: {}'.format(epoch, loss.item()))
            # 梯度回传
            model.backward(loss_grad)
            # 梯度更新
            model.step()
        tmp_loss /= 10
        train_loss.append(tmp_loss)
        # testing
        test(x_test, y_test)
```

图 2-4　训练过程核心代码

测试过程如图 2-5 所示。

```python
def test(x_test, y_test):
    y_pred = model(x_test)
    loss = criterion(y_pred, y_test)
    test_loss.append(loss.item())
    print('testing loss: {}'.format(loss.item()))
```

图 2-5　测试过程代码

在对模型进行训练完成后，使用测试数据对模型的最终性能进行评估，指标为 MSE，MSE 值越小，则模型的预测精度越高。完整的数据加载、模型训练、测试全过程如图 2-6 所示。

```python
if __name__ == '__main__':
    # train data
    x_train, y_train = load_data('x_train.csv', 'y_train.csv')
    train_dataloader = NumpyDataLoader(x_train, y_train, batch_size=batch_size, shuffle=True)
    # test data
    x_test, y_test = load_data('x_test.csv', 'y_test.csv')
    # 模型初始化
    criterion = MSELoss()
    optimizer = SGD(lr=learning_rate)
    model = ZeroNet(optimizer=optimizer)
    # training
    start = time.time()
    train(model, epochs, train_dataloader, x_test, y_test, criterion)
    end = time.time()
    print('training finish! cost: {} s'.format(end - start))
    # testing
    y_pred = model(x_test)
    loss = criterion(y_pred, y_test)
    print('testing loss: {}'.format(loss.item()))
```

图 2-6　完整的数据加载、模型训练、测试全过程代码

超参数设置如表 2-1 所示。

表 2-1 超参数设置表

参数名称	数值大小	参数解释
batch size	100	每一批次输入到模型中的数据样本数量
epochs	2000	所有训练数据都流过模型一次为一个 epoch
learning rate	0.002	模型的学习率

2.1.3 训练结果与优化

训练过程中在训练数据上的训练损失 train_loss 和在测试数据上的测试损失 test_loss 变化折线如图 2-7 所示。

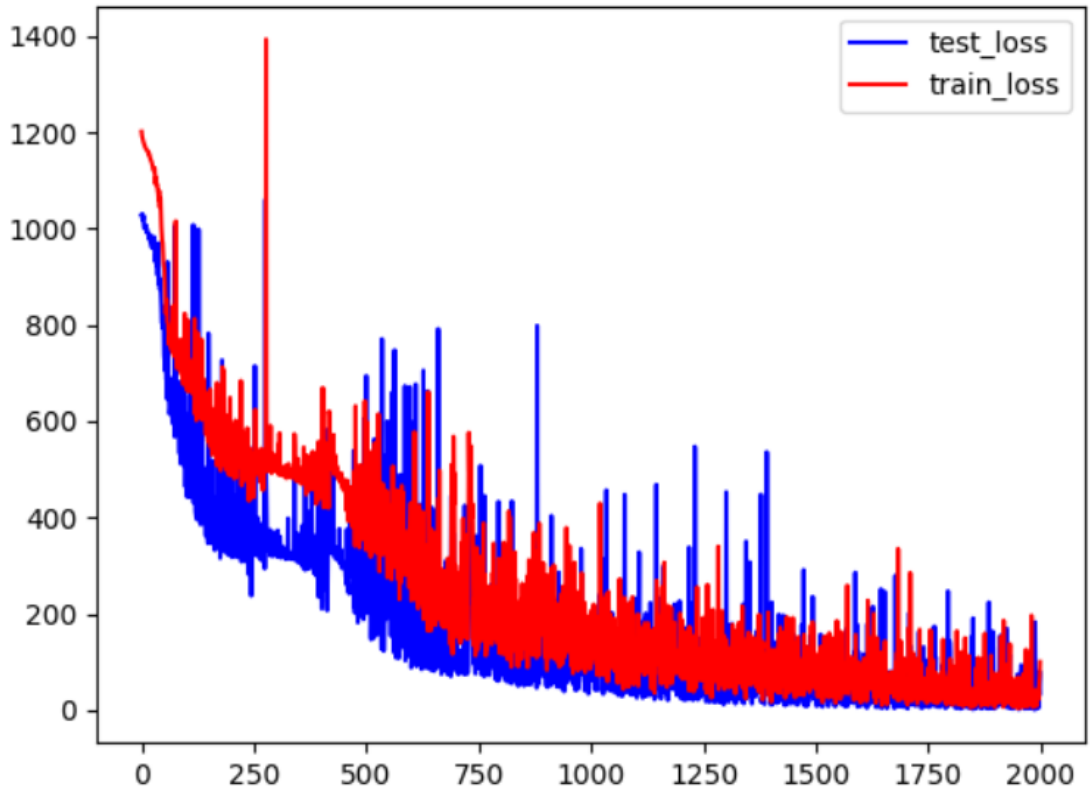

图 2-7　训练过程中 loss 变化折线图

训练完成的所耗时间和结束时的 train_loss 和 test_loss 如表 2-2 所示。

表 2-2　训练结束相关结果表

名称	数值
train_loss	23.52810382025281
test_loss	34.4841798582198
cost_time(s)	12.414246082305908

可以看到，最终的 train_loss 和 test_loss 的数值都比较大，且 test_loss 明显比 train_loss 大，说明模型的训练效果并不是很好，且呈现出在训练集上过拟合的特点，过多的学习了训练集的特征，而在测试集上的泛化性能较差。从 loss 变化折线图可以看到，loss 的下降波动大，不稳定，且收敛速度慢，使得模型不能快速稳定的训练。

观察模型在测试集上的预测输出结果，如图 2-8 所示。

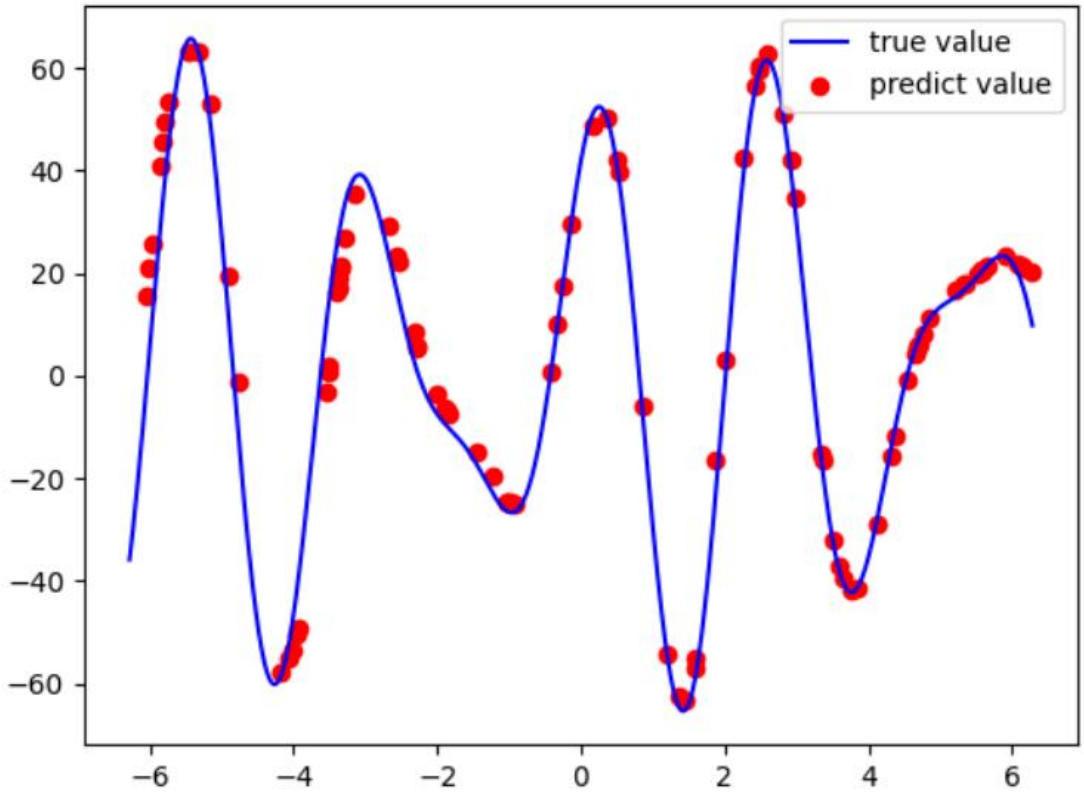

图 2-8　真实值与模型预测值曲线对比图

从图 2-8 可看出，模型预测的输出值与真实值之间虽然已经比较贴近，但仍存在肉眼可见的误差。需要优化。

为了使模型在训练过程中参数优化过程能更稳定，即使得梯度下降的方向更优，将原来的 SGD 优化器更改为 Adam 优化器。Adam 优化器的更新过程如下：

$$m_{dw} = \beta_1 m_{dw} + (1-\beta_1)dw$$
$$m_{db} = \beta_1 m_{db} + (1-\beta_1)db$$
$$v_{dw} = \beta_2 v_{dw} + (1-\beta_2)dw^2$$
$$v_{db} = \beta_2 v_{db} + (1-\beta_2)db^2$$
$$m'_{dw} = \frac{m_{dw}}{1-\beta_1^t}$$
$$m'_{db} = \frac{m_{db}}{1-\beta_1^t}$$
$$v'_{dw} = \frac{v_{dw}}{1-\beta_2^t}$$

$$v'_{db} = \frac{v_{db}}{1-\beta_2^t}$$

$$w := w - \eta \frac{m'_{dw}}{\sqrt{v'_{dw}} + \epsilon}$$

$$b := b - \eta \frac{m'_{db}}{\sqrt{v'_{db}} + \epsilon}$$

η为前文定义的学习率。其它超参数设置$\beta_1 = 0.9$，$\beta_2 = 0.999$。$\epsilon = 1 \times 10^{-8}$用于防止除零操作。Adam 优化器完整代码如图 2-9 所示。

```
class Adam(object):
    def __init__(self, lr=0.0001, eps=1e-8, beta1=0.9, beta2=0.999):
        super(Adam, self).__init__()
        self.lr = lr
        self.eps = eps
        self.beta1 = beta1
        self.beta2 = beta2

    def __call__(self, grad, m_t, v_t):
        m_t = self.beta1 * m_t + (1 - self.beta1) * grad
        v_t = self.beta2 * v_t + (1 - self.beta2) * (grad ** 2)

        m_hat = m_t / (1 - self.beta1)
        v_hat = v_t / (1 - self.beta2)

        return -self.lr * m_hat / (np.sqrt(v_hat) + self.eps), m_t, v_t
```

图 2-9　Adam 优化器完整代码

同时，为了使得模型训练的初始阶段不坍缩，使用何恺明提出的模型参数初始化方法即 kaiming_uniform，在训练开始前对神经网络的权重参数进行初始化。

kaiming_uniform 的思想是使用均匀分布对模型参数进行初始化，即：

$$w \sim \mathrm{U}(-\mathrm{bound}_w, \mathrm{bound}_w)$$

$$b \sim \mathrm{U}(-\mathrm{bound}_b, \mathrm{bound}_b)$$

其中，bound_w的计算如下：

$$\mathrm{bound}_w = \mathrm{gain} \times \sqrt{\frac{3}{\mathrm{fan\_mode}}}$$

bound_b的计算如下：

$$\mathrm{bound}_b = \sqrt{\frac{1}{\mathrm{fan\_mode}}}$$

对于 ReLU 激活函数，gain $= \sqrt{2}$，fan_mode这里设置为每一层神经元的权重矩阵的输出维度。

kaiming_uniform 的完整代码如图 2-10 所示。

```python
def kaiming_uniform(weights, bias=None, gain=math.sqrt(2)):
    """
    version: for ReLu
    梯度降不下去，weights和bias用kaiming初始化
    """
    fan_in = weights.shape[1]
    fan_out = weights.shape[0]
    std = gain / math.sqrt(fan_in)
    bound = math.sqrt(3.0) * std
    weights = np.random.uniform(-bound, bound, weights.shape)

    if bias is not None:
        bound = 1 / math.sqrt(fan_in) if fan_in > 0 else 0
        bias = np.random.uniform(-bound, bound, bias.shape)
        return weights, bias
    return weights
```

图 2-10　kaiming_uniform 完整代码

完成上述优化后，其他设置保持不变，重新对改进后的模型进行训练。改进后训练过程中在训练数据上的训练损失 train_loss 和在测试数据上的测试损失 test_loss 变化折线如图 2-11 所示。

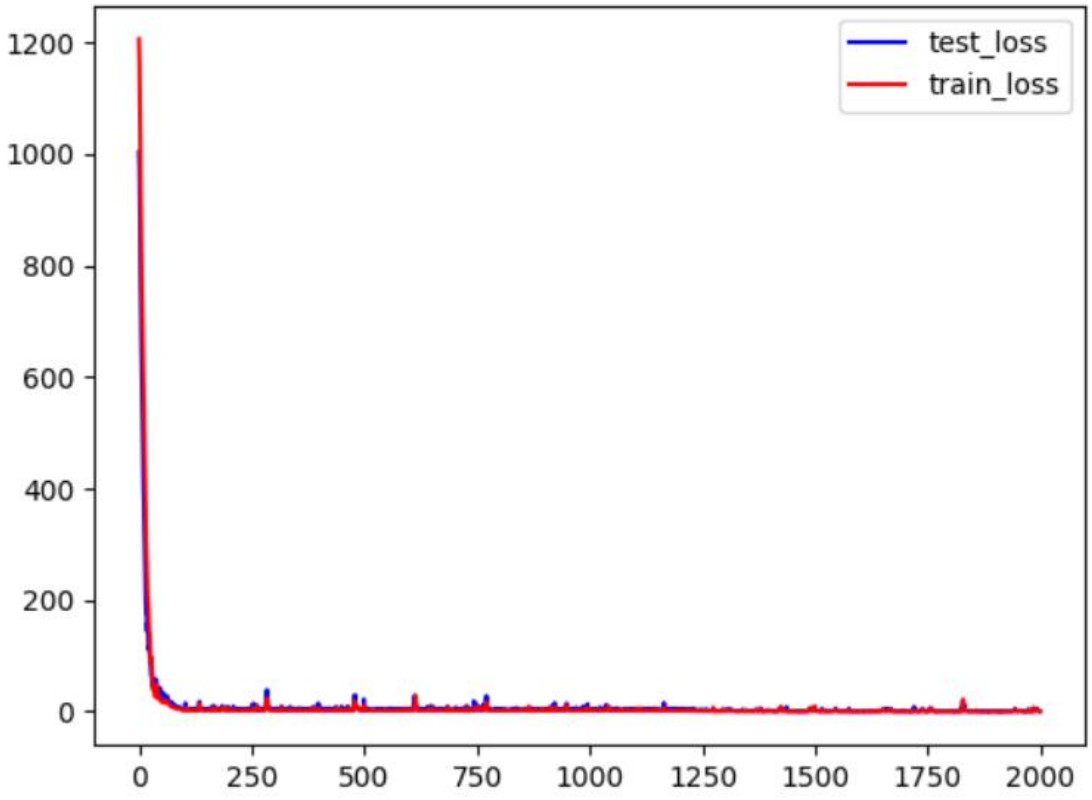

图 2-11　改进后训练过程中 loss 变化折线图

训练完成的所耗时间和结束时的 train_loss 和 test_loss 如表 2-3 所示。

表 2-3　改进后训练结束相关结果表

名称	数值
train_loss	0.24208677812596643
test_loss	0.5913421787744951
cost_time(s)	15.060735702514648 s

可以看到，改进后，最终的 train_loss 和 test_loss 的数值已经非常小，虽然 test_loss 比 train_loss 大，但差值几乎可以忽略。考虑到测试集比训练集的预测难度应该更大，改进后的模型的训练效果提升已经非常明显。从 loss 变化折线图可以看到，使用 Adam 优化器之后的 loss 的下降很稳定，且收敛速度快。能快速持久地对模型的参数进行动态优化，且由于使用了 kaiming_uniform 初始化方法，使得模型训练初始阶段避免容易陷入坍缩，陷入局部平凡解。

观察改进后的模型在测试集上的预测输出结果，如图 2-12 所示。

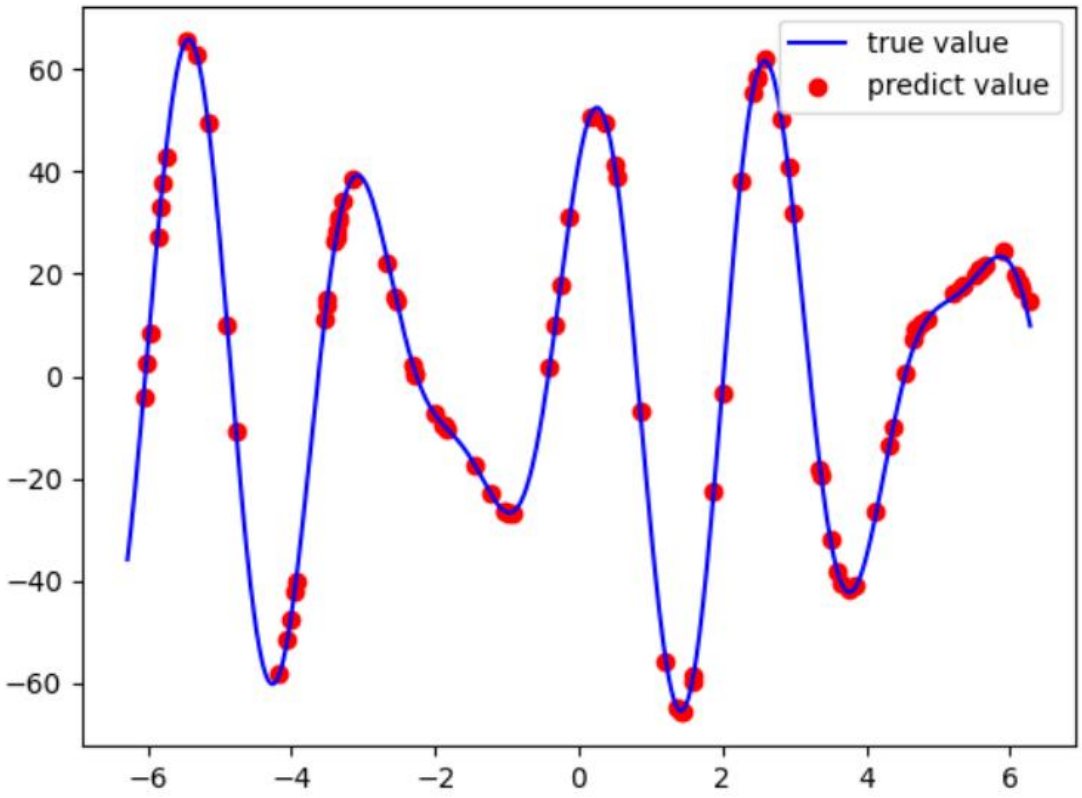

图 2-12　改进后真实值与模型预测值曲线对比图

从图 2-12 可看出，模型预测的输出值与真实值之间高度拟合，预测精度很高。

2.2 手写数字数据集 MNIST 分类实验

2.2.1 MNIST 数据集

MNIST 是一个手写体数字的图片数据集，一共统计了来自 250 个不同的人手写数字图片，其中 50%是高中生，50%来自人口普查局的工作人员。该数据集的收集目的是希望通过算法，实现对手写数字的识别。MNIST 数据集通常划分为训练集和测试集，训练集一共包含 60000 张图片和标签，而测试集一共包含 10000 张图片和标签。每张图片是一个有28×28像素点的 0～9 的灰质手写数字图片，黑底白字，图片像素值为 0～255，像素值越大则像素点越白。

从数据集库导入 MNIST 数据集的方式如图 2-13 所示。

```
from keras.datasets import mnist


def load_mnist():
    (x_train, y_train), (x_test, y_test) = mnist.load_data('D:/ziliao/pycharm/从零搭建神经网络/mnist/mnist.npz')
    return x_train, y_train, x_test, y_test
```

图 2-13　导入 MNIST 数据集的代码

x_tain 包含 6000 张训练图片，x_test 包含 10000 张测试图片，y_train 包含对应的训练标签，y_test 包含对应的测试标签。标签值为对应的真实数字数值，是标量，为了计算分类损失的方便，需要将标量标签转换成 one-hot 编码的向量形式。编码方式如图 2-14 所示。

```
def encode_onehot(labels):
    classes = set(labels)
    classes_dict = {c: np.identity(len(classes))[i, :] for i, c in enumerate(classes)}
    labels_onehot = np.array(list(map(classes_dict.get, labels)))
    return labels_onehot
```

图 2-14　标签 one-hot 编码方法

2.2.2 适应分类任务的网络结构调整

由于现在要做的是分类任务，因此损失函数要将之前的 MSE 更改为 CrossEntropyLoss。由于标签已经 one-hot 编码为了向量，因此神经网络的最后一层输出也应为相同维度的向量（即 10 维，因为有 10 个数字，也就是 10 个类别）。第一层的结构也要做调整，因为原始输入是28×28的图片像素矩阵，因此需要先将矩阵按照逐行逐列的顺序从左到右、从上到下拉直为$28 * 28$维的向量，所以第一层的输入维度也应为$28 * 28$维。同时由于像素值的范围为[0,255]，数值范围较大，容易导致模型训练不稳定，因此先将像素值归一化到[0,1]区间。调整后的用于分类的全连接神经网络详细结构如表 2-4 所示。

表 2-4　全连接神经网络详细结构(分类任务)表

Layer name	Function	(Input dimension, Output dimension)
Linear1	$y = wx + b$	(28*28, 16)
ReLU1	ReLU(x)	(16, 16)
Linear2	$y = wx + b$	(16, 16)
ReLU2	ReLU(x)	(16, 16)
Linear3	$y = wx + b$	(16, 16)

ReLU3	$\mathrm{ReLU}(x)$	(16, 16)
Linear4	$y = wx + b$	(16, 16)
ReLU4	$\mathrm{ReLU}(x)$	(16, 16)
Linear5	$y = wx + b$	(16, 10)

调整后用于分类的全连接神经网络完整代码如图 2-15 所示。

```python
class ZeroNet(object):
    def __init__(self, optimizer=None):
        super(ZeroNet, self).__init__()
        self.network = [
            Linear(28 * 28, 16, optimizer=optimizer),  # TODO 对应mnist的输入维度
            ReLu(),
            Linear(16, 16, optimizer=optimizer),
            ReLu(),
            Linear(16, 16, optimizer=optimizer),
            ReLu(),
            Linear(16, 16, optimizer=optimizer),
            ReLu(),
            Linear(16, 10, optimizer=optimizer),  # TODO 对应mnist的输出维度
            # 分类用的softmax输出头靠CrossEntropyLoss完成
        ]

    def __call__(self, x):
        for layer in self.network:
            x = layer(x)
        return x

    def backward(self, grad):
        last_grad = grad.copy()
        for layer in self.network[::-1]:  # 倒序翻转
            last_grad = layer.backward(last_grad)
        return last_grad

    def step(self):
        for layer in self.network:
            layer.step()
```

图 2-15　全连接神经网络(分类任务)完整代码

2.2.3 评估指标

因为是分类任务，所有直接使用分类精度（accuracy）作为模型训练好坏的评价指标。accuracy 的计算方式如图 2-16 所示。

```python
def accuracy(y_pred, y_true):
    y_pred = np.argmax(y_pred, axis=-1)
    y_true = np.argmax(y_true, axis=-1)
    acc = np.mean(y_pred == y_true)
    return acc
```

图 2-16　分类精度计算代码

除了上述提到的重要调整和处理方式，分类模型的数据加载、训练过程、测

试过程的方式基本上与拟合三角函数实验的方式是一致的，如图 2-17 所示。

```
if __name__ == '__main__':
    ################################
    # 训练mnist
    ################################
    # 加载mnist数据集
    x_train, y_train, x_test, y_test = load_mnist()
    y_train = encode_onehot(y_train)
    x_test.resize(x_test.shape[0], 28 * 28)
    y_test = encode_onehot(y_test)
    x_train, x_test = x_train / 255.0, x_test / 255.0  # 数值归一化
    train_dataloader = NumpyDataLoader(x_train, y_train, batch_size=batch_size, shuffle=True)
    # 模型初始化
    criterion = CrossEntropyLoss()
    optimizer = Adam(lr=learning_rate)
    model = ZeroNet(optimizer=optimizer)
    # training
    start = time.time()
    train_mnist(model, epochs, train_dataloader, x_test, y_test, criterion)
    end = time.time()
    print('training finish! cost: {} s'.format(end - start))
    # testing
    y_pred = model(x_test)
    acc = accuracy(y_pred, y_test)
    loss = criterion(y_pred, y_test)
    print('testing loss: {}, accuracy: {}'.format(loss.item(), acc))
```

图 2-17　分类模型的数据加载、模型训练、测试全过程代码

超参数设置如表 2-5 所示。

表 2-5 分类任务超参数设置表

参数名称	数值大小	参数解释
batch size	1000	每一批次输入到模型中的数据样本数量
epochs	50	所有训练数据都流过模型一次为一个 epoch
learning rate	0.009	模型的学习率

2.2.4 训练结果

分类任务训练过程中在训练数据上的训练损失 train_loss 和在测试数据上的测试损失 test_loss 变化折线如图 2-18 所示。

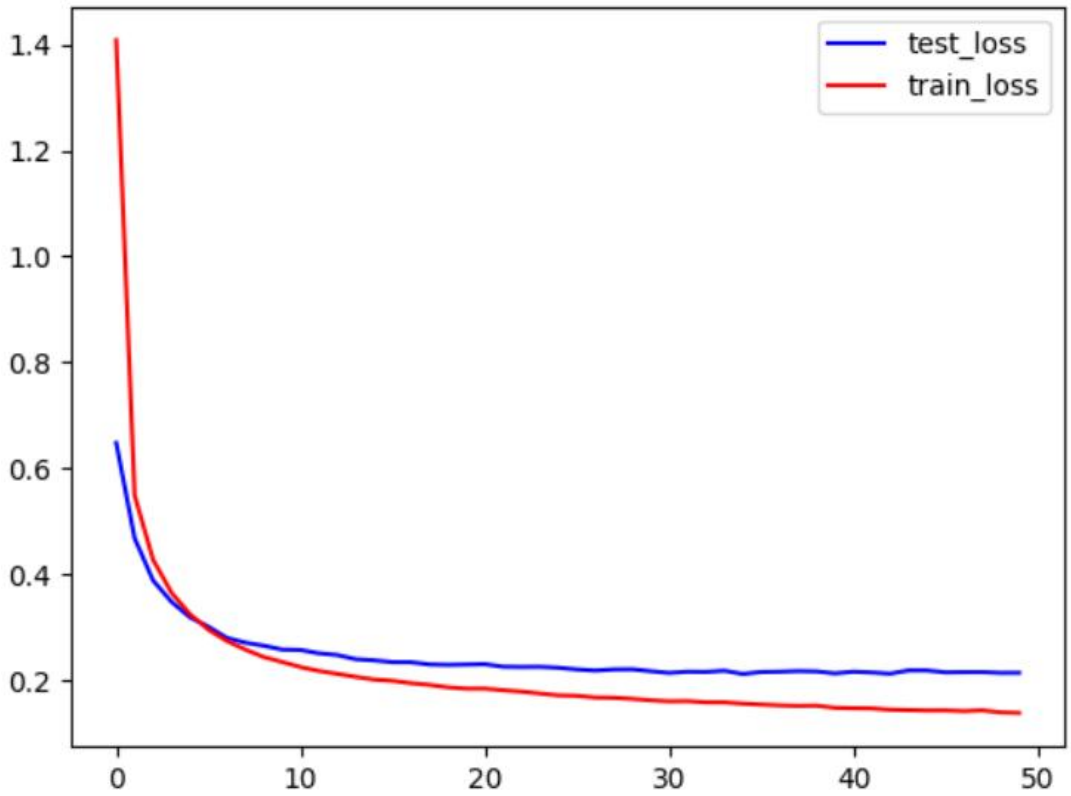

图 2-18　分类任务训练过程中 loss 变化折线图

分类任务训练完成的所耗时间和结束时的 train_loss 和 test_loss 如表 2-6 所示。

表 2-6　分类任务训练结束相关结果表

名称	数值
train_loss	0.12257579051845764
train_accuracy(%)	97.00
test_loss	0.21438339574377618
test _accuracy(%)	94.13
cost_time(s)	85.88880467414856

可以看到，分类任务训练结束最终的 train_loss 和 test_loss 的数值已经比较小。从 loss 变化折线图可以看到，使用 Adam 优化器训练的 loss 的下降很稳定，且收敛速度快。分类模型在训练集上的分类精度达到了 97.00%，已经非常高，在测试集上也能达到几乎相当的 94.13%的分类精度。

3 总结

所设计的全连接神经网络性能优异，能实现拟合三角函数和对 MINIST 数据集进行分类的任务要求。