第3章

深度学习基础模型简介

3.1 反向传播算法

3.1.1 反向传播算法简介

反向传播(Back Propagation,BP)算法是一种神经网络优化算法。介绍算法前先简单解释神经网络定义。

神经网络的设计灵感来源于生物学上的神经网络。典型的神经网络如图 3-1 所示,每个节点就是一个神经元,神经元与神经元之间的连线表示信息传递的方向。Layer 1 表示输入层,Layer 2、Layer 3 表示隐藏层,Layer 4 表示输出层。通过神经网络,对输入数据进行某种变换,从而获得期望的输出。

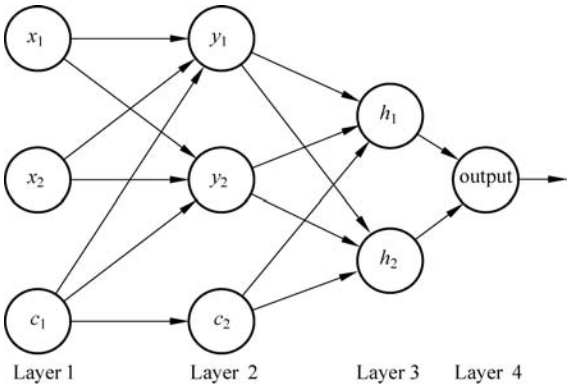

图 3-1 典型神经网络

总的来说,神经网络就是一种映射,将原数据映射成期望获得的数据。BP 算法就是其中的一种映射。下面通过一个具体的例子来演示 BP 算法的过程。

BP 算法示例-网络层，如图 3-2 所示，第一层有两个神经元 x_1、x_2，一个截距项 c_1；第二层有两个神经元 y_1、y_2，一个截距项 c_2；第三层是输出，有两个神经元 h_1、h_2；每条线上的数值表示神经元之间连接的权重，具体数值如图 3-2 所示。激活函数 σ 选用 Sigmoid 函数。Sigmoid 函数及其对 x 的导数如式(3-1)所示。

$$S(x)=\frac{1}{1+e^{-x}}$$
$$S'(x)=S(x)\times(1-S(x)) \tag{3-1}$$

式中，输入 $x_1=0.05$，$x_2=0.1$，目标：输出 h_1、h_2 尽可能接近[0.03，0.05]。下面将具体介绍其实现过程。

1. 前向传播

(1) 输入层→隐藏层。

为了方便理解，可以把神经元再进一步细化(以 y_1 为例)。神经元细化图，如图 3-3 所示。

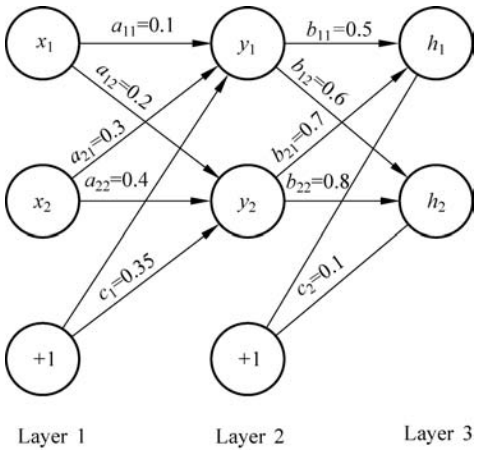

图 3-2 BP 算法示例-网络层

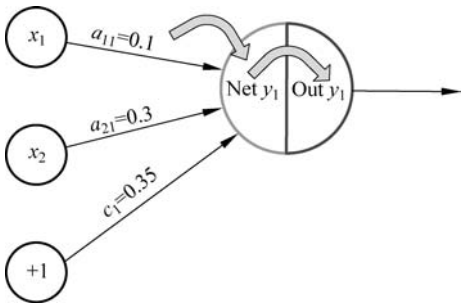

图 3-3 神经元细化图

计算神经元 y_1 的输入加权和：

$$\text{net}_{y_1}=x_1\times a_{11}+x_2\times a_{21}+1\times c_1$$

带入数值可得：

$$\text{net}_{y_1}=0.05\times 0.1+0.1\times 0.3+1\times 0.35=0.385$$

神经元 y_1 的输出为：

$$\text{out}_{y_1}=\frac{1}{1+e^{-\text{net}_{y_1}}}=0.595$$

同理可得：

$$\text{out}_{y_2}=0.599$$

(2) 隐藏层→输出层。

这一步的方法与“输入层→隐藏层”相似，计算神经元 h_1、h_2 的值。

$$\text{net}_{h_1} = \text{out}_{y_1} \times b_{11} + \text{out}_{y_2} \times b_{21} + c_2$$

$$\text{out}_{h_1} = \frac{1}{1 + \text{e}^{-\text{net}_{h_1}}}$$

计算可得：

$$\text{net}_{h_1} = 0.817, \quad \text{out}_{h_1} = 0.694$$

$$\text{net}_{h_2} = 0.936, \quad \text{out}_{h_2} = 0.718$$

至此，已经完成了前向传播的过程，此时输出为[0.694，0.718]，与期望的输出[0.03，0.05]相差较大。接下来，通过反向传播，更新每条边上的权值，重新计算输出。

2. 反向传播

(1) 计算总误差。

为了简化例子，方便理解，本例中采用均方误差作为总误差函数，均方误差的计算如式(3-2)所示。

$$E(y, \hat{y}) = J_{\text{MSE}}(y, \hat{y}) = \sum_{i=1}^{N} \frac{1}{N} (\text{target}_{\text{out}_{h_i}} - \text{out}_{h_i})^2 \tag{3-2}$$

式中，$\text{target}_{\text{out}_{h_i}}$ 表示第 i 个真实值，在本例中指目标输出值；out_{h_i} 表示预测值，在本例中指每次迭代时的输出值；N 取值为 2。

依据式(3-2)计算现在的误差：

$$E = \frac{1}{2} [(\text{target}_{\text{out}_{h_1}} - \text{out}_{h_1})^2 + (\text{target}_{\text{out}_{h_2}} - \text{out}_{h_2})^2]$$

$$= \frac{1}{2} [(0.03 - 0.694)^2 + (0.05 - 0.718)^2] = 0.444$$

(2) 权值更新(输出层→隐藏层)。

因为每个权值对误差都产生了影响，为了了解每个权值对误差产生了多少影响，可以用整体误差对特定的权值求偏导来实现这一目的。从输出层到隐藏层，共有 5 个参数需要更新，分别为 b_{11}，b_{12}，b_{21}，b_{22}，c_2。以 b_{22} 为例，通过链式法则进行计算，如式(3-3)所示。

$$\frac{\partial E}{\partial b_{22}} = \frac{\partial E}{\partial \text{out}_{h_2}} \times \frac{\partial \text{out}_{h_2}}{\partial \text{net}_{h_2}} \times \frac{\partial \text{net}_{h_2}}{\partial b_{22}} \tag{3-3}$$

如图 3-3 所示，前向传播和反向传播其实就是对这一过程的形象描述，指明了应该如何去处理这一偏导数。结合神经元细化图(反向传播)，如图 3-4 所示，可以更好地理解。箭头的方向就表明了求偏导的方向。

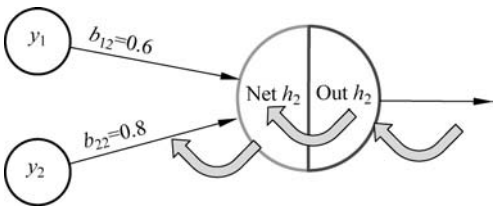

图 3-4　神经元细化图(反向传播)

这里逐项求解(σ 表示激活函数)：

① $\dfrac{\partial E}{\partial \text{out}_{h_2}}$。

$$\frac{\partial E}{\partial \text{out}_{h_2}}=2\times\frac{1}{2}\times(\text{target}_{\text{out}_{h_2}}-\text{out}_{h_2})\times(-1)$$

② $\dfrac{\partial \text{out}_{h_2}}{\partial \text{net}_{h_2}}$。

$$\frac{\partial \text{out}_{h_2}}{\partial \text{net}_{h_2}}=\frac{\partial}{\partial \text{net}_{h_2}}\left(\frac{1}{1+\text{e}^{-\text{net}_{h_2}}}\right)=\frac{1}{1+\text{e}^{-\text{net}_{h_2}}}\times\left(1-\frac{1}{1+\text{e}^{-\text{net}_{h_2}}}\right)$$

$$=\sigma(\text{net}_{h_2})(1-\sigma(\text{net}_{h_2}))$$

③ $\dfrac{\partial \text{net}_{h_2}}{\partial b_{22}}$。

$$\frac{\partial \text{net}_{h_2}}{\partial b_{22}}=\frac{\partial}{\partial b_{22}}(\text{out}_{y_1}\times b_{12}+\text{out}_{y_2}\times b_{22}+c_2)=\text{out}_{y_2}$$

综上所述，

$$\frac{\partial E}{\partial b_{22}}=(-1)\times(\text{target}_{\text{out}_{h_2}}-\text{out}_{h_2})\times\text{out}_{h_2}(1-\text{out}_{h_2})\times\text{out}_{y_2}$$

带入具体数值可得：

$$b_{22}^{\text{new}}=b_{22}-\rho\times\frac{\partial E}{\partial b_{22}}=0.760$$

其中，ρ 表示学习率，本例设定为 0.5。同理可得 $b_{11}^{\text{new}}=0.458$，$b_{12}^{\text{new}}=0.560$，$b_{21}^{\text{new}}=0.658$。

同样，对于偏置项，求解方法类似，但由于偏置项对于每个神经元的损失都有贡献，所以应为对每个神经元求偏导后再求和，如式(3-4)所示。

$$\frac{\partial E}{\partial c_2}=\sum_i\frac{\partial E}{\partial \text{out}_{h_i}}\times\frac{\partial \text{out}_{h_i}}{\partial \text{net}_{h_i}}\times\frac{\partial \text{net}_{h_i}}{\partial c_2}\tag{3-4}$$

由于最后一项在本例中求导后值为 1，一般情况下都为 1，故可简化为式(3-5)。

$$\frac{\partial E}{\partial c_2}=\sum_i\frac{\partial E}{\partial \text{out}_{h_i}}\times\frac{\partial \text{out}_{h_i}}{\partial \text{net}_{h_i}}\tag{3-5}$$

求偏导方法与上文相似，不再赘述。代入具体数值可以求得新的偏置项：

$$c_2^{\text{new}}=c_2-\rho\times\frac{\partial E}{\partial c_2}=-0.038$$

(3) 权值更新(隐藏层→输入层)。

方法与“输出层→隐藏层”类似，但是有一点区别。如图 3-4 所示，可以发现神经元 h_1 向后就直接输出了，没有再输入到下一个神经元，而神经元 y_1 的输出值要输入到神经元 h_1、h_2，导致神经元 y_1 会接收来自 h_1、h_2 两个神经元传递的误差，因此 h_1、h_2 均要计算。结合神经元细化图(反向传播-隐藏层→隐藏层)，如图 3-5 所示，可以更直观地理解。

同样，可以通过链式法则求出误差对权值的影响。从隐藏层到输入层，共有 5 个参数

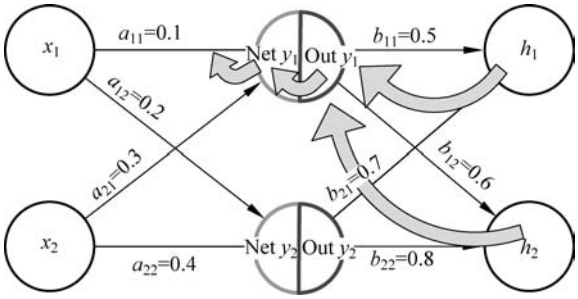

图 3-5　神经元细化图(反向传播-隐藏层→隐藏层)

需要更新,分别为 a_{11},a_{12},a_{21},a_{22},c_1。以 a_{11} 为例:

$$\frac{\partial E}{\partial a_{11}}=\frac{\partial E}{\partial \text{out}_{y_1}}\frac{\partial \text{out}_{y_1}}{\partial \text{net}_{y_1}}\frac{\partial \text{net}_{y_1}}{\partial a_{11}} \tag{3-6}$$

① $\dfrac{\partial E}{\partial \text{out}_{y_1}}$。

$$\frac{\partial E}{\partial \text{out}_{y_1}}=\frac{\partial E}{\partial \text{out}_{h_1}}\times\frac{\partial \text{out}_{h_1}}{\partial \text{net}_{h_1}}\times\frac{\partial \text{net}_{h_1}}{\partial \text{out}_{y_1}}+\frac{\partial E}{\partial \text{out}_{h_2}}\times\frac{\partial \text{out}_{h_2}}{\partial \text{net}_{h_2}}\times\frac{\partial \text{net}_{h_2}}{\partial \text{out}_{y_2}}$$

上式中几项偏微分均已在输出层→隐藏层的权值更新中有相应的计算公式,因此:

$$\begin{aligned}\frac{\partial E}{\partial \text{out}_{y_1}}&=(\text{target}_{\text{out}_{h_1}}-\text{out}_{h_1})\times(-1)\times\sigma(\text{net}_{h_1})(1-\sigma(\text{net}_{h_1}))\times b_{11}+\\&\quad(\text{target}_{\text{out}_{h_2}}-\text{out}_{h_2})\times(-1)\times\sigma(\text{net}_{h_2})(1-\sigma(\text{net}_{h_2}))\times b_{12}\\&=0.644\times0.212\times0.5+0.668\times0.202\times0.6\\&=0.149\end{aligned}$$

② $\dfrac{\partial \text{out}_{y_1}}{\partial \text{net}_{y_1}}$。

$$\begin{aligned}\frac{\partial \text{out}_{y_1}}{\partial \text{net}_{y_1}}&=\frac{\partial}{\partial \text{net}_{y_1}}\left(\frac{1}{1+\text{e}^{-\text{net}_{y_1}}}\right)=\sigma(\text{net}_{y_1})(1-\sigma(\text{net}_{y_1}))\\&=0.595\times(1-0.595)=0.241\end{aligned}$$

③ $\dfrac{\partial \text{net}_{y_1}}{\partial a_{11}}$。

$$\frac{\partial \text{net}_{y_1}}{\partial a_{11}}=\frac{\partial}{\partial a_{11}}(x_1\times a_{11}+x_2\times a_{12}+c_1)=x_1=0.05$$

综上所述,

$$\frac{\partial E}{\partial a_{11}}=0.149\times0.241\times0.05=0.002$$

代入具体数值可得,$a_{11}^{\text{new}}=a_{11}-\rho\times\dfrac{\partial E}{\partial a_{11}}=0.1-0.5\times0.002=0.099$。

同理可得,$a_{12}^{\text{new}}=0.199$,$a_{21}^{\text{new}}=0.298$,$a_{22}^{\text{new}}=0.398$。

偏置项的求法与输出层→隐含层方法一致，这里不再赘述，但应注意的是，c_1 的更新与 y_1、y_2、h_1、h_2 均有关系，代入数值可得：

$$c_1^{\text{new}} = 0.307$$

至此，所有参数均已更新完毕，BP 算法示例-网络层，如图 3-6 所示，利用更新完毕之后的参数可以计算得到新的输出为[0.667，0.693]（原来的输出为[0.694，0.718]，目标输出为[0.03，0.05]），新的总误差为 0.44356（原来的总误差为 0.444）。通过新的权值计算，可以发现输出值与目标值逐渐接近，总误差逐渐减小，随着迭代次数的增加，输出值会与目标值高度相近。

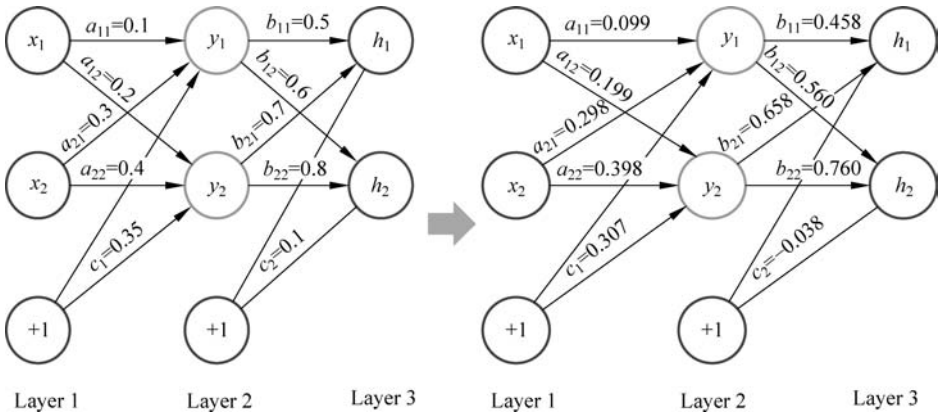

图 3-6　BP 算法示例-网络层

3.1.2　NumPy 实现反向传播算法

本节基于 NumPy 编程实现简单的神经网络分类任务，过程中手动实现反向传播算法，以加深对反向传播算法的理解。本节数据集采用 sklearn.make_moons()数据集，并借助 Scikit-Learn 包进行数据预处理。数据集可视化，如图 3-7 所示，数据集是二维的。实现代码如下。

```
from sklearn.datasets import make_moons
X, y = make_moons(n_samples = 2000, noise = 0.4, random_state = None)
```

选取 60%为训练集，40%为测试集，并进行预处理，实现代码如下。

```
from sklearn.model_selection import train_test_split
trainX, testX, trainY, testY = train_test_split(X, y, test_size = 0.4, random_state = 32)

from sklearn.preprocessing import StandardScaler
standard = StandardScaler()
trainX = standard.fit_transform(trainX)
testX = standard.transform(testX)
```

本节重新搭建一个两层的神经网络。神经网络结构如图 3-8 所示，其中，$\boldsymbol{X}$ 是一个 $p \times q$ 的矩阵。选取交叉熵损失函数作为该神经网络的损失函数(式(3-7))，其中，$\hat{y}_i$ 表示预测值，y_i 表示真实值。学习率为 0.05，采用梯度下降法进行参数更新，实现代码如下。

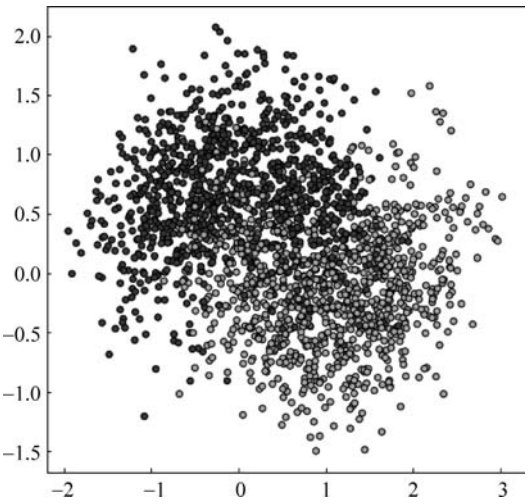

图 3-7　数据集可视化

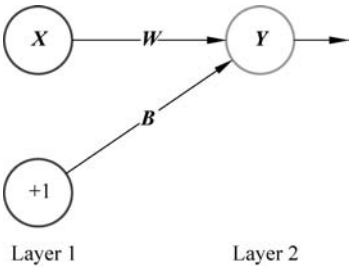

图 3-8　神经网络结构

$$E(y,\hat{y}) = -\frac{1}{N}\sum_{i=1}^{N}(y_i \ln \hat{y_i} + (1-y_i)\ln(1-\hat{y_i})) \tag{3-7}$$

```
def loss_func(trueY, predY):
    loss = - np.mean(trueY * np.log(predY) + (1 - trueY) * np.log(1 - predY))
    return loss
```

接着定义构建的神经网络。对于权重矩阵 $\boldsymbol{W}$ 及偏置矩阵 $\boldsymbol{B}$，采用随机初始化的方法。$\boldsymbol{W}$ 和 $\boldsymbol{B}$ 的纬度较高时，不易手工初始化。若 $\boldsymbol{W}$ 和 $\boldsymbol{B}$ 初始化为 0 或者同一个值，会导致在梯度下降的更新过程中梯度保持相等，权值相同，导致不同的隐藏单元都以相同的函数或函数值作为输入，可以通过参数的随机初始化克服这种问题。同时要注意参数初始化应合理，否则会出现梯度消失或梯度爆炸，具体实现代码如下。

```
def _init_(q):
    #q:W 的维度(也就是有 q 个神经元)
    #将 W 与 B 随机初始化
    np.random.seed(3)
    W = np.random.normal(size = (q,)) * 0.05   #随机初始化 W
    B = np.zeros(1, )                          #初始化 B 为全 0 矩阵
    return W, B
```

初始化完权重矩阵与偏置矩阵后，再定义一个激活函数 Sigmoid，就可以完成前向传播的部分了。通过 scipy.special.expit()实现激活函数 Sigmoid，代码如下。

```
from scipy.special import expit
def sigmoid(X):
    #X: np.ndarray, 待激活值
    #sigmoid(x) = 1/(1+exp(-x))
    return expit(X)
```

至此，可以实现前向传播部分，代码如下。

```
def forward(X, W, B):
    #X: np.ndarray, 输入数据, 维度 = (p,q)
    #W:np.ndarray, 权重矩阵, 维度 = (q,)
    #B: np.ndarray, 偏置项, 维度 = (1,)
    #计算 Z = X*W + B
    #predY = sigmoid(Z)
    Z = np.dot(X, W) + B  #计算 Z
    predY = sigmoid(Z)    #通过激活函数获取输出值
    return predY
```

完成前向传播后，开始编写反向传播部分，通过计算可得：

$$\frac{\partial E}{\partial W}=\frac{\partial E}{\partial \text{out}_y}\times\frac{\partial \text{out}_y}{\partial \text{net}_y}\times\frac{\partial \text{net}_y}{\partial W}=\frac{1}{N}[X^{\mathrm{T}}(\hat{y}-y)]$$

$$\frac{\partial E}{\partial B}=\frac{\partial E}{\partial \text{out}_y}\times\frac{\partial \text{out}_y}{\partial \text{net}_y}\times\frac{\partial \text{net}_y}{\partial B}=\frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i-y_i)$$

实现代码如下。

```
def backword(W, B, trueY, predY, X, learning_rate):
    #W: np.ndarray, 权重矩阵,维度 = (q,)
    #B: np.ndarray, 偏置项,维度 = (1,)
    #trueY: np.ndarray, 真值,维度 = (p, )
    #predY: np.ndarray, 预测值,维度 = (p, )
    #X: np.ndarray, 输入数据,维度 = (p,q)
    #learning_rate: float,学习率

    #1. 计算梯度
    #dW: np.ndarray, 损失函数对 W 的偏导, 维度 = (q,)
    dW = np.dot(X.T, predY - trueY) / len(trueY)
    #dB: float, 损失函数对 B 的偏导
    dB = np.sum(predY - trueY) / len(trueY)

    #2. 参数更新
    W -= learning_rate * dW
    B -= learning_rate * dB
```

接下来定义训练函数，代码如下。

```
def train(trainX, trainY, testX, testY, W, B, epochs, flag):
    #trainX: np.ndarray, 训练集,维度 = (p,q)
    #trainY: np.ndarray, 训练集标签,维度 = (p, )
```

```
    #testX: np.ndarray, 测试集,维度 = (m,q)
    #testY: np.ndarray, 测试集标签,维度 = (m, )
    #W:np.ndarray, 权重矩阵,维度 = (q,)
    #B: np.ndarray, 偏置项,维度 = (1,)
    #epochs: 迭代次数
    #flag: flag == True 打印损失值, flag == False 不打印损失值

    train_loss_list = []                                    #存储在训练集上的损失值
    test_loss_list = []                                     #存储在测试集上的损失值
    for i in range(epochs):
        #训练集
        pred_train_Y = forward(trainX, W, B)                #前向传播
        train_loss = loss_func(trainY, pred_train_Y)        #计算损失值

        #测试集
        pred_test_Y = forward(testX, W, B)                  #前向传播
        test_loss = loss_func(testY, pred_test_Y)           #计算损失值

        if flag == True:                                    #打印第 i 轮训练集、测试集上的损失值
            print('the traing loss of %s epoch : %s' % (i + 1, train_loss))
            print('the test loss of %s epoch : %s' % (i + 1, test_loss))
            print('==========================')

        train_loss_list.append(train_loss)                  #存储该轮训练集上的损失函数
        test_loss_list.append(test_loss)                    #存储该轮测试集上的损失函数

        #反向传播
        backword(W, B, trainY, pred_train_Y, trainX, learning_rate)
    return train_loss_list, test_loss_list
```

最后调用 train()函数,并绘制(训练集、测试集)损失曲线。(训练集、测试集)损失曲线如图 3-9 所示。

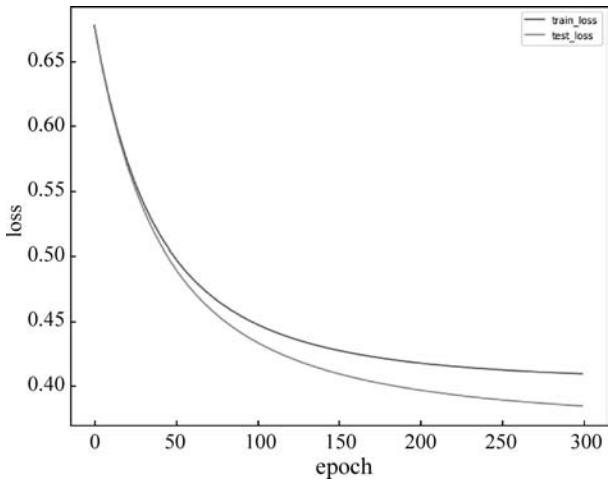

图 3-9　(训练集、测试集)损失曲线

定义预测函数，在测试集上通过以下代码预测，并将预测结果可视化，代码如下。

```
def predict(X, W, B):
    #X: np.ndarray, 训练集, 维度 = (n, m)
    #W: np.ndarray, 参数, 维度 = (m, 1)
    #B: np.ndarray, 参数 b, 维度 = (1, )

    y_pred = forward(X, W, B)
    n = len(y_pred)
    prediction = np.zeros((n, 1))
    for i in range(n):
        if y_pred[i] > 0.5:
            prediction[i, 0] = 1
        else:
            prediction[i, 0] = 0
    return prediction
pred = predict(testX, W, B)
#将测试集可视化
plt.figure(figsize = (8, 8))
plt.scatter(testX[:, 0], testX[:, 1], c = pred, cmap = ListedColormap(['#B22222', '#87CEFA']),
edgecolors = 'k')
```

测试集预测结果如图 3-10 所示。可以发现，模型将数据集分为较为明显的两类。

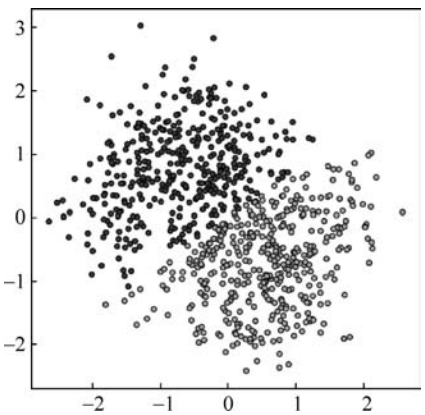

图 3-10　测试集预测结果

3.2　循环神经网络

3.2.1　循环神经网络简介

对于普通神经网络，例如多层感知机，其前一个输入和后一个输入完全没有联系，导致在处理时间序列问题时，难以精准刻画时间序列中的时间关系，例如，难以识别语言内容、预测商品价格的未来趋势等。人类在语言表达时，每个字都与前面表达的内容有逻辑

顺序，使得在理解语音内容的时候，不可能孤立地通过单个字来理解，而是需要在记住前面关键信息的基础上，理解后面的表达。同样地，预测商品的未来价格趋势也需要在充分理解和记忆商品历史价格的基础上，预测商品未来的价格。交通专业领域同样涉及很多时间序列问题，如短时客流预测问题，即利用以往的数据预测未来短时内的客流量，也是一种时间序列问题。

为了更好地处理时间序列问题，学者提出了循环神经网络结构(Recurrent Neural Network，RNN)，最基本的循环神经网络由输入层、一个隐藏层和一个输出层组成。

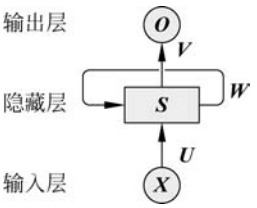

图 3-11　RNN 结构

RNN 结构如图 3-11 所示，如果去掉有 $\boldsymbol{W}$ 的带箭头的连接线，即为普通的全连接神经网络。其中，$\boldsymbol{X}$ 是一个向量，代表输入层的值；$\boldsymbol{S}$ 是一个向量，表示隐藏层的值，其不仅取决于当前的输入 $\boldsymbol{X}$，还取决于上一时刻隐藏层的值；$\boldsymbol{O}$ 也是一个向量，它表示输出层的值；$\boldsymbol{U}$ 是输入层到隐藏层的权重矩阵；$\boldsymbol{V}$ 是隐藏层到输出层的权重矩阵；$\boldsymbol{W}$ 是隐藏层上一时刻的值作为当前时刻输入的权重矩阵。

把此结构按照时间线展开，可得到展开的 RNN 结构。按时间线展开的 RNN 结构如图 3-12 所示。

网络在 t 时刻接收到输入 $\boldsymbol{X}_t$ 之后，隐藏层的值是 $\boldsymbol{S}_t$，输出值是 $\boldsymbol{O}_t$，$\boldsymbol{S}_t$ 的值不仅取决于 $\boldsymbol{X}_t$，还取决于 $\boldsymbol{S}_{t-1}$。因此，在 RNN 结构下，当前时刻的信息在下一时刻也会被输入到网络中，网络中的信息形成时间相关性，解决了处理时间序列的问题。神经网络模型“学”到的东西隐含在权值 $\boldsymbol{W}$ 中。基础的神经网络只在层与层之间建立全连接，而 RNN 与它们最大的不同之处在于同一层内的神经元在不同时刻也建立了全连接，即 $\boldsymbol{W}$ 与时间有关。

根据不同的输入输出模式，可以将 RNN 分为以下四种结构：one to one、one to many、many to one 和 many to many。其中，最典型的结构属于 one to one 结构，这种结构为给定一个输入值来预测一个输出值。one to one 结构如图 3-13 所示。

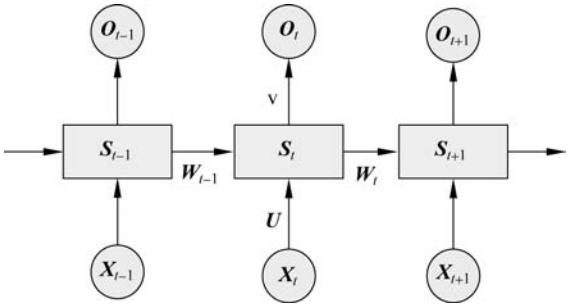

图 3-12　按时间线展开的 RNN 结构

图 3-13　one to one 结构

one to many 结构和 many to one 结构如图 3-14 所示，当输入值为一个输出值为多个的时候，比如在网络中输入一个关键字，通过网络输出一首以这个关键字为主题的诗歌，就是一个 one to many 场景。当输入值为多个，输出值为一个时，比如输入一段语音判断这段语音的情感分类，再比如输入以往的股票信息判断未来股票价格是涨还是跌，诸如此类的分类等问题就是 many to one 场景。

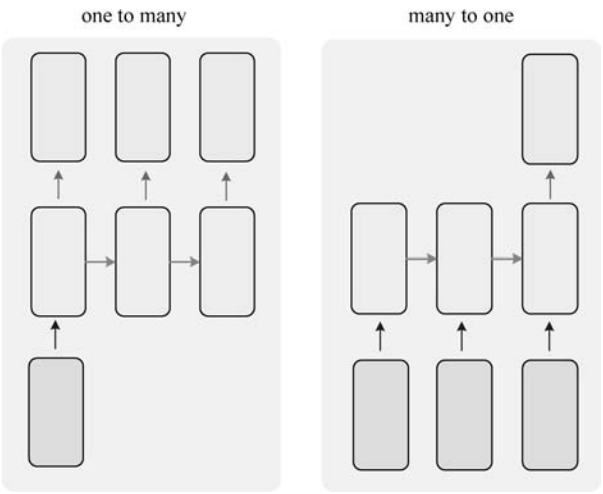

图 3-14　one to many 结构和 many to one 结构

many to many 结构如图 3-15 所示。其中，第一种 many to many 结构适用于机器翻译、自动问答等场景，比如输入一句英文，输出一句中文，输入与输出的都是序列。第二种 many to many 结构适用于视频每一帧的分类和命名实体标记等领域，比如输入一段视频，将其每一帧进行分类。

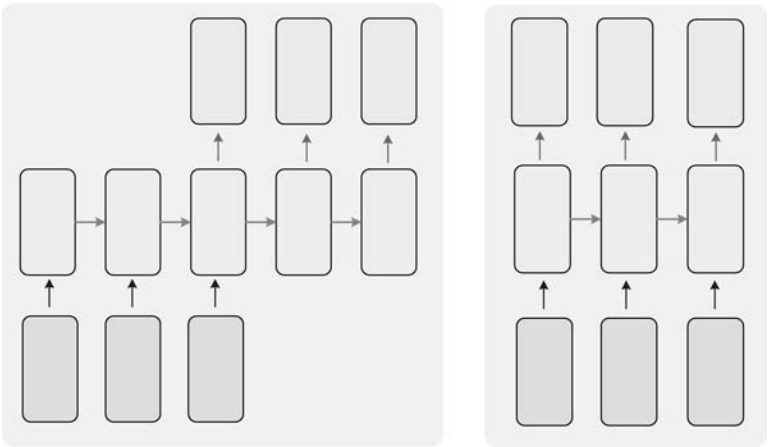

图 3-15　many to many 结构

3.2.2　LSTM 简介

LSTM(Long Short-Term Memory)模型属于循环神经网络 RNN 的一种，由 Hochreiter 等人于 1997 年提出，至今已有 20 多年的发展历史。传统 RNN 取得巨大成功的原因在于其能够将历史信息进行记忆存储并通过前馈神经网络(例如多层感知机)向前传递，当经历长期的信息传递之后，由于梯度消失和梯度爆炸等原因，会存在有用信息丢失的问题，导致无法记忆长期信息。不同于传统 RNN，LSTM 由于其独特的记忆单元，能在一定程度上解决长期依赖问题，因此在自然语言处理(NLP)、模式识别、短时交通预测等领域的表现均超越了传统 RNN，取得了巨大的成功。

RNN 和 LSTM 的最大区别在于分布在隐藏层的神经元结构，传统 RNN 的神经元结构简单，例如仅包含一个激活函数层。LSTM 的记忆单元(Block)更加复杂，LSTM 模型中增加了状态 c，称为单元状态(Cell State)，用来保存长期的状态，而 LSTM 的关键就是怎样控制长期状态 c，LSTM 使用三个控制开关，第一个开关负责控制如何继续保存长期状态 c，第二个开关负责控制把即时状态输入到长期状态 c，第三个开关负责控制是否把长期状态 c 作为当前的 LSTM 的输入。长期状态 c 的控制如图 3-16 所示，其中三个开关分别是使用三个门来控制的，这种去除和增加单元状态中信息的门是一种让信息选择性通过的方法。

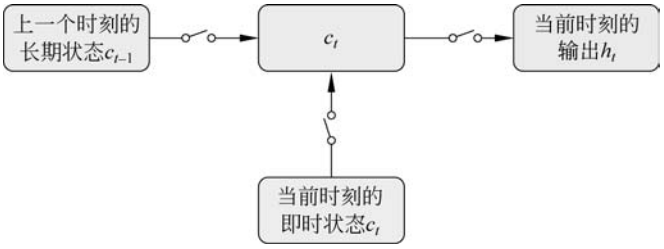

图 3-16　长期状态 c 的控制

LSTM 用两个门来控制单元状态 c 的内容，一是遗忘门(Forget Gate)，遗忘门决定了上一时刻的单元状态 c_{t-1} 有多少保留到当前时刻 c_t；二是输入门(Input Gate)，输入门决定了当前时刻网络的输入 x_t 有多少保存到单元状态 c_t。LSTM 用输出门(Output Gate)来控制单元状态 c_t 有多少输出到 LSTM 的当前输出值 h_t。

下面对这三个门逐一介绍，在以下示意图中，黄色矩形是学习得到的神经网络层，粉色圆形表示运算操作，箭头表示向量的传输过程。

遗忘门 f_t，如图 3-17 所示是其具体结构。式(3-8)中，W_f 是遗忘门的权重矩阵，$[h_{t-1}, x_t]$表示把两个向量连接成一个更长的向量，b_f 是遗忘门的偏置项，σ 是 Sigmoid 函数。

$$f_t=\sigma(W_f\cdot[h_{t-1},x_t]+b_f) \tag{3-8}$$

输入门，如图 3-18 所示是其具体结构。Sigmoid 函数称为输入门，决定将要更新什么值，Tanh 层创建一个新的候选值向量，$\widetilde{C}_t$ 会被加入到状态中。

$$i_t=\sigma(W_i\cdot[h_{t-1},x_t]+b_i) \tag{3-9}$$

$$\widetilde{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_C) \tag{3-10}$$

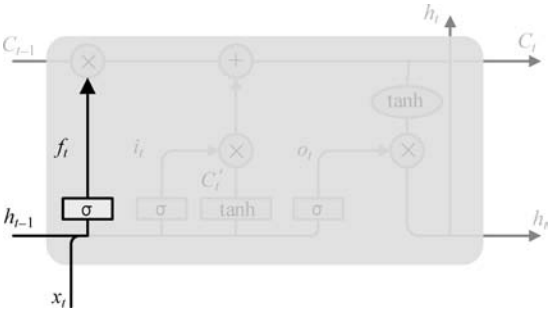

图 3-17 遗忘门

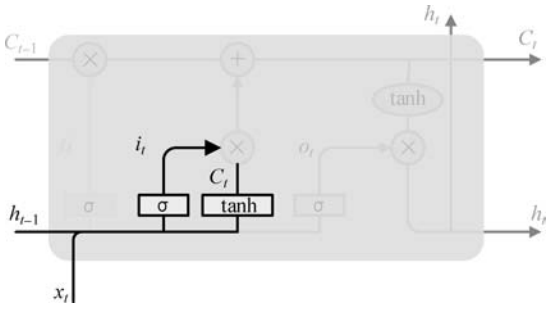

图 3-18 输入门

更新单元状态，如图 3-19 所示是其具体结构。在遗忘门的控制下，网络可以保存很久之前的信息，在输入门的控制下，无用信息无法进入到网络当中。

$$C_t = f_t \times C_{t-1} + i_t \times \widetilde{C}_t \tag{3-11}$$

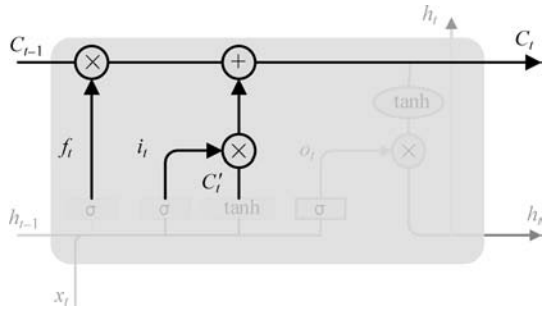

图 3-19 更新单元状态

输出门具体结构如图 3-20 所示。输出门控制了长期记忆对当前输出的影响，由输出门和单元状态共同确定。

$$O_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{3-12}$$

$$h_t = O_t \times \tanh C_t \tag{3-13}$$

LSTM 的重复模块如图 3-21 所示，是 LSTM 依时间展开的重复模块，类似于图 3-12。总而言之，LSTM 的核心是单元的状态，单元状态的传递类似于传送带，直接在整个时间链上运行，中间值有少量的线性交互，以便保存相关信息。

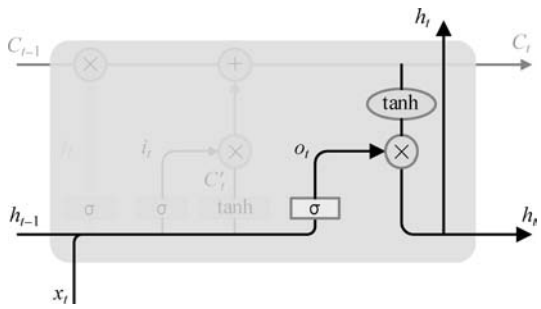

图 3-20　输出门具体结构

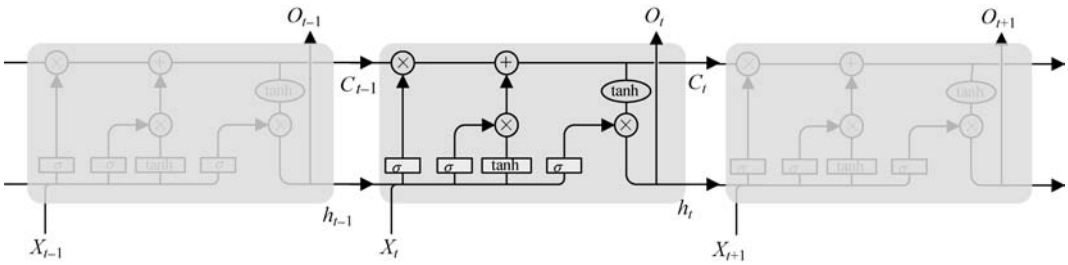

图 3-21　LSTM 的重复模块

3.2.3　PyTorch 实现 LSTM 时间序列预测

本节内容将以 2016 年北京地铁西直门站时间粒度为 15min 的进站客流数据为例，利用 PyTorch 搭建 LSTM 网络，实现对进站客流数据的预测。旨在帮助读者进一步了解 LSTM 网络并掌握利用 PyTorch 实现基于 LSTM 网络的时间序列预测。具体实现步骤如下。

首先导入需要的包，其中，NumPy 包、Pandas 包和 Matplotlib 包在 Python 基础知识简介部分的 1.5 节、1.6 节、1.9 节学习过了，这里直接使用，代码如下。

```
import torch
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
# % matplotlib inline
from torch import nn
from torch.autograd import Variable
```

使用 Pandas 包里的 read_csv()函数读取西直门站 3 天的 CSV 格式的客流数据，代码如下。

```
data_csv = pd.read_csv('#此处填数据文件的存放路径', usecols=[1])
```

使用 Matplotlib 包里的 pyplot.plot()函数来可视化输入的数据，代码如下。

```
plt.plot(data_csv)
```

数据集的可视化如图 3-22 所示。

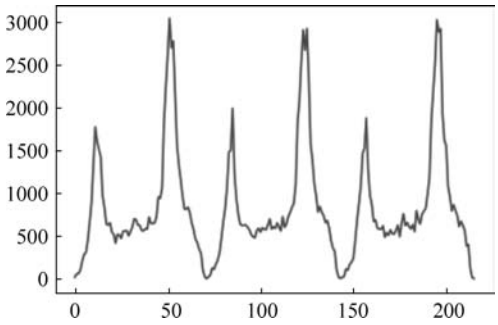

图 3-22 数据集的可视化

从输出的客流图可以看出，三天时间从早到晚地铁站的客流变化趋势规律性较强，并且能够明显地观察到一天中的客流早晚高峰情况。接着把处理后的数据输入到 LSTM 模型里面进行训练，希望通过 LSTM 模型来预测客流。

首先开始数据预处理，去掉无效数据，并且将数据归一化到[0,1]，数据的归一化在深度学习中可以提升模型的收敛速度和精度。

使用 dropna()函数去掉数据中的空值所在的行和列，使用 astype()函数变化数组类型，并且手动将数据集中的数据值大小固定到[0,1]，实现代码如下。

```
data_csv = data_csv.dropna()
dataset = data_csv.values
dataset = dataset.astype('float32')
max_value = np.max(dataset)
min_value = np.min(dataset)
scalar = max_value - min_value
dataset = list(map(lambda x: x / scalar, dataset))
```

创建训练和测试 LSTM 模型的数据集，明确目标是通过前面几个时间粒度的客流量来预测当前时间粒度的客流量，将前两个时间粒度的客流数据作为输入，对应代码中的 step=2，把当前时间粒度的客流数据作为输出，划分数据集为训练集和测试集，通过测试集得到的效果来评估模型的预测性能，实现代码如下。

```
def create_dataset(dataset, step=2):
    dataA, dataB = [], []
    for i in range(len(dataset) - step):
        a = dataset[i:(i + step)]
        dataA.append(a)
        dataB.append(dataset[i + step])
    return np.array(dataA), np.array(dataB)
```

定义好输入和输出，实现代码如下。

```
data_A, data_B = create_dataset(dataset)
```

划分训练集和测试集，70％的数据作为训练集，30％的数据作为测试集，实现代码如下。

```
train_size = int(len(data_A) * 0.7)
test_size = len(data_A) - train_size
train_A = data_A[:train_size]
train_B = data_B[:train_size]
test_A = data_A[train_size:]
test_B = data_B[train_size:]
```

改变数据维度，对一个样本而言，序列只有一个，所以 Batch_Size＝1，由于算法是根据前两个时间粒度预测第三个，所以 feature＝2，具体代码如下。

```
train_A = train_A.reshape(-1, 1, 2)
train_B = train_B.reshape(-1, 1, 1)
test_A = test_A.reshape(-1, 1, 2)

train1 = torch.from_numpy(train_A)
train2 = torch.from_numpy(train_B)
test1 = torch.from_numpy(test_A)
```

定义模型并将输出值回归到流量预测的最终结果，模型的第一部分是一个两层的 RNN，具体代码如下。

```
class lstm_reg(nn.Module):
    def __init__(self, input_size, hidden_size, output_size=1, num_layers=2):
        super(lstm_reg, self).__init__()

        self.rnn = nn.LSTM(input_size, hidden_size, num_layers)
        self.reg = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x, _ = self.rnn(x)
        s, b, h = x.shape
        x = x.view(s * b, h)
        x = self.reg(x)
        x = x.view(s, b, -1)
        return x
```

输入维度是 2，隐藏层维度为 4，其中隐藏层维度可以任意指定，使用均方损失函数，代码如下。

```
net = lstm_reg(2, 4)
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(net.parameters(), lr=1e-2)
```

开始训练模型，训练 2000 个 Epoch，每 200 次输出一次训练结果，即 Loss 值，代码如下。

```
for e in range(2000):
    var1 = Variable(train1)
    var2 = Variable(train2)
    out = net(var1)
    loss = criterion(out, var2)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    if (e + 1) % 200 == 0:
        print('Epoch: {}, Loss: {:.5f}'.format(e + 1, loss.item()))
```

模型的训练结果如图 3-23 所示。

通过训练过程中输出的 Loss 值可以看到，损失值在逐渐下降，模型训练效果比较可观。训练完成后，转换为测试模式，开始预测客流并且输出预测结果，代码如下。

```
net = net.eval()
data_A = data_A.reshape(-1, 1, 2)
data_A = torch.from_numpy(data_A)
var_data = Variable(data_A)
pred_test = net(var_data)
pred_test = pred_test.view(-1).data.numpy()
```

将实际结果和预测结果用 Matplotlib 包画图输出，其中真实数据用蓝色表示，预测的结果用橙色表示，代码如下。

```
plt.plot(dataset, label = 'real')
plt.plot(pred_test,label = 'prediction')
plt.legend(loc = 'best')
```

模型的预测效果如图 3-24 所示。可以看到训练后的 LSTM 模型预测的客流数据可以比较准确地拟合真实客流数据，说明 LSTM 模型的时间序列预测能力是比较可观的。

```
Epoch: 200, Loss: 0.00306
Epoch: 400, Loss: 0.00261
Epoch: 600, Loss: 0.00229
Epoch: 800, Loss: 0.00215
Epoch: 1000, Loss: 0.00172
Epoch: 1200, Loss: 0.00103
Epoch: 1400, Loss: 0.00094
Epoch: 1600, Loss: 0.00075
Epoch: 1800, Loss: 0.00069
Epoch: 2000, Loss: 0.00064
```

图 3-23　模型的训练结果

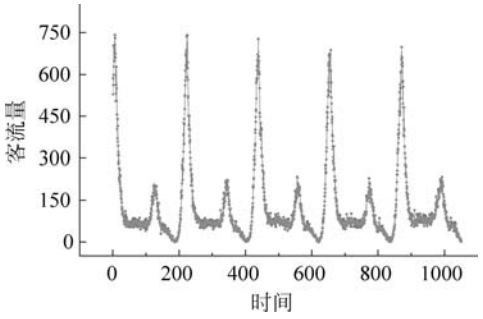

图 3-24　模型的预测效果

3.3　卷积神经网络

3.3.1　卷积神经网络简介

使用全连接前馈神经网络处理图像时，往往存在以下三个明显的缺陷。

（1）参数过多。

如果输入图像大小为100×100×3，在全连接前馈网络中，第一个隐藏层的每个神经元到输入层都有30 000个互相独立的连接，每个连接都对应一个权重参数。随着隐藏层神经元数量的增多，参数的规模也会急剧增加，导致整个神经网络的成本很高，训练效率非常低，且容易出现过拟合。

（2）难以捕捉局部特征。

自然图像中的物体都具有局部不变性特征，如尺度缩放、平移、旋转等操作不影响其语义信息。而全连接前馈网络很难提取这些局部不变性特征，一般需要进行数据增强来提高其性能。

（3）导致信息丢失。

由于全连接神经网络在处理图像信息时，首先需要将图像展开为向量，因此部分空间信息容易丢失，导致图像识别的准确率不高。

针对全连接前馈神经网络处理图像时存在的缺陷，学者受到生物学上的感受野机制的启发，提出了卷积神经网络（Convolutional Neural Network，CNN），较好地化解了以上的三个缺陷。

首先简单了解一下感受野（Receptive Field）机制。感受野机制主要是指听觉、视觉等神经系统中一些神经元的特性，即神经元只接收其所支配的刺激区域内的信号。基于该机制提出的卷积神经网络就是通过建立卷积层、池化层以及全连接层实现对图像的精确处理。卷积层负责提取图像中的局部特征，接着利用池化层大幅降低参数数量（降维）从而提高训练效率，最后通过全连接层进行线性转换，输出结果。由于卷积神经网络在结构上具有局部连接、权值共享以及池化三个特性，使得网络具有一定程度上的平移、缩放和旋转不变性，可以保留图片的空间特性，因此在图像处理方面具有很大优势。

接下来将简单介绍卷积神经网络各个层的基本原理，为了让读者更好地理解，忽略部分技术细节。

（1）卷积层——提取特征。

对图像（不同的数据窗口数据）和滤波矩阵（一组固定的权重）做矩阵内积（对应元素相乘再求和）的操作就是所谓的“卷积”，也是卷积神经网络的来源。具体的卷积计算如图3-25所示，用一个卷积核（相当于一个滤波器filter）扫描整张图片。

在具体应用中，往往会有多种卷积核，每种卷积核代表一种图像特征，如颜色深浅、轮廓等。如果某个图像块与该卷积核内积得到的数值大，则认为非常接近该图像特征。总之，与人类的感受野机制相似，卷积层通过卷积核的过滤可以提取图像中的局部特征。

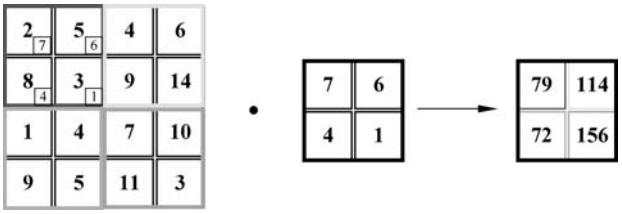

图 3-25 卷积计算

(2) 池化层(下采样)——数据降维,避免过拟合。

池化层简单来说就是下采样,用于压缩数据和参数的量,降低位数,减小过拟合的现象,通常来说就是取区域最大或者区域平均。池化计算如图 3-26 所示。

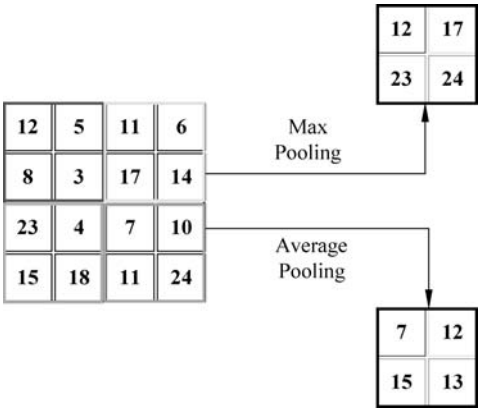

图 3-26 池化计算

此处简单地对池化层的作用进行描述。

① 特征不变性。

通过池化操作,依然可以保留图像的重要特征,对图像进行池化只是去掉一些无关紧要的信息,留下的信息则是具有尺度不变性的特征,可以充分表达图像特征。

② 数据降维。

即使经过卷积,图像的数据依然很大,包含很多重复的以及没有太大作用的信息,通过池化可以剔除这些冗余的信息,对数据起到降维的作用。大大降低数据维度,防止过拟合。

(3) 全连接层——输出结果。

这个部分是整个卷积神经网络的最后一步,经过卷积和池化处理后的数据输入到全连接层,根据回归或者分类问题的需要,选择不同激活函数获得最终想要的结果。也就是跟传统的神经网络神经元的连接方式是一样的,即所有神经元都有权重连接。

以上就是卷积神经网络的基本结构,实际上,CNN 并非只是上面提到的 3 层结构,而是多层结构,需要通过多层卷积、池化实现对图像的处理。下面将针对不同维数对卷积神经网络展开进一步的讲解。

3.3.2　一维和二维卷积神经网络

近年来，由于计算机视觉的飞速发展，卷积神经网络得到了广泛的应用。目前来说，二维卷积神经网络的使用范围是最广的，受到了许多计算机爱好者的追捧与研究。当提及卷积神经网络(CNN)时，通常是指用于图像分类的二维卷积，上文中对于卷积神经网络的介绍正是基于二维卷积。除了二维卷积神经网络，还有用于预测时间序列的一维卷积神经网络，以及面向视频处理领域(检测动作及人物行为)的三维卷积神经网络。此处仅对一维和二维神经网络做一个简单介绍。

初学者在接触卷积神经网络(CNN)时，可能会直观地理解为一维卷积就是处理一维数据，而二维卷积就是处理二维数据，这是错误的。事实上，一维和二维滤波器并不是指真正的一维和二维，这只是描述的惯例。无论是一维还是二维，CNN 都具有相同的特征并采用相同的方法，关键区别在于输入数据的维度以及特征检测器(卷积核)如何在数据上滑动，一维和二维卷积神经网络的区别如图 3-27 所示。

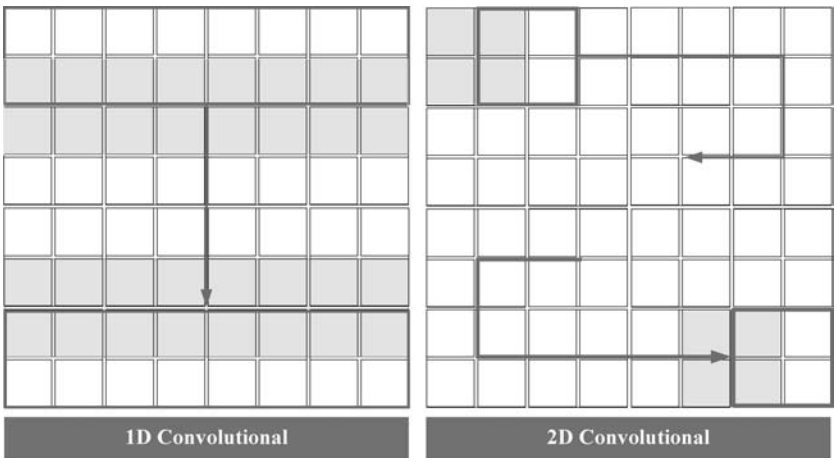

图 3-27　一维和二维卷积神经网络的区别

一维卷积神经网络，其卷积核在数据上只能沿一维(水平)方向滑动，通常输入的是一个向量，输出的也是一个向量。如果要从整体数据集的较短片段中获取重要的特征，且该特征与空间位置不相关时(比如所有特征都是在同一个位置产生的数据)，一维卷积神经网络将非常有效。那么哪种类型的数据仅需要卷积核在一个维度上滑动并具有空间特性呢？比如一维的时间序列数据。一维卷积神经网络非常适用于分析类似于传感器记录的时间序列数据，也适用于在固定长度的时间段内分析多种信号数据(例如音频数据)等，这些数据沿时间序列生成，卷积核仅需沿着时间序列方向进行滑动即可，具体一维卷积计算如图 3-28 所示。

有了对一维卷积神经网络的了解，相信读者将更容易理解二维卷积神经网络。与一维卷积相比，二维卷积神经网络的卷积核在数据上沿二维方向(水平和竖直方向)滑动，二维卷积计算如图 3-29 所示。由于二维卷积神经网络可以使用其卷积核从数据中提取空间特征，例如，检测图像的边缘、颜色分布等，使得二维卷积神经网络在图像分类和包含空间属性的其他类似数据的处理中功能非常强大，目前来说也是使用范围最广的卷积神经网络。

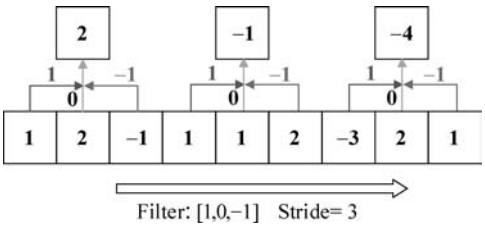

图 3-28 一维卷积计算

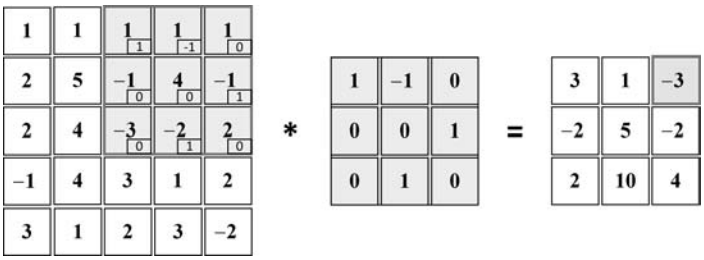

图 3-29 二维卷积计算

以下两节将对两种卷积神经网络的实现进行详细讲解，手把手教读者利用 PyTorch 实现卷积神经网络的搭建。

3.3.3 PyTorch 实现一维卷积神经网络时间序列预测

本节内容将以 2016 年北京地铁西直门站 15min 的进站客流数据为例，利用 PyTorch 搭建一维卷积神经网络，实现对进站客流数据的预测。旨在帮助读者进一步了解卷积神经网络并掌握利用 PyTorch 实现基于一维卷积神经网络的时间序列预测。以下为具体实现步骤。

（1）数据集的导入和处理。

此次一维卷积实验仅仅是对北京西直门地铁站进站客流进行预测，而原始数据集内包含北京所有地铁站点的进站客流数据，因此在导入数据集后需要选定西直门地铁站的进站客流数据进行处理。此处调用了 Python 中的 Pandas 包，通过 Pandas 包导入数据集并进行索引处理，具体代码如下。

```
import torch
import torch.nn as nn
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler

#导入北京地铁站点 15min 进站客流
df = pd.read_csv('./15min_in.csv', index_col = 0, encoding = "gbk", parse_dates = True)
len(df)
df.head()  #观察数据集，这是一个单变量时间序列
```

```
y = df['西直门'].values.astype(float)          #数据类型改为浮点型
plt.figure(figsize=(12, 4))
plt.grid(True)                                 #网格化
plt.plot(y)
plt.show()
```

导入西直门地铁站进站客流数据后，需要划分训练集和测试集。共有1799个时间段的客流数据，以数据集的后300个作为测试集。为了获得更好的训练效果，将客流数据进行归一化处理，归一化到[－1,1]区间，具体代码如下。

```
#划分测试集和训练集,最后300个作为测试集
test_size = 300
train_iter = y[:-test_size]
test_iter = y[-test_size:]
#归一化至[-1,1]区间,为了获得更好的训练效果
scaler = MinMaxScaler(feature_range=(-1, 1))
train_norm = scaler.fit_transform(train_iter.reshape(-1, 1))

#创建时间序列训练集
train_set = torch.FloatTensor(train_norm).view(-1)
```

(2) 时间窗口的设定。

为了更好地了解网络预测的准确性，笔者设定时间窗口来选取数据进行预测，时间窗口的大小为72，即从原时间序列中抽取出训练样本，用第1～72个数据作为 x 输入，预测第73个值作为 y 输出。此处定义了input_data()函数进行训练样本抽取，并且返回由输入数据和输出标签构成的列表，代码如下。

```
#定义时间窗口
Time_window_size = 72

#从原时间序列中抽取出训练样本,用第1～72个值作为X输入,预测第73个值作为y输出,这
#是一个用于训练的数据点,时间窗口向后滑动以此类推
def input_data(seq, ws):
    out = []
    L = len(seq)
    for i in range(L-ws):
        window = seq[i:i + ws]
        label = seq[i + ws:i + ws + 1]
        out.append((window, label))
    return out


train_data = input_data(train_set, Time_window_size)
len(train_data)  #等于1799(原始数据集长度)-300(测试集长度)-72(时间窗口)
```

最新版本的 NumPy 中提供了一个 sliding_window_view()函数，该函数通过输入时间序列以及时间窗大小，可自动实现训练样本的提取，使用方式代码如下。

```
from numpy.lib.stride_tricks import sliding_window_view
output = sliding_window_view(seq, ws)
```

（3）一维卷积神经网络的搭建。

目前常用来搭建神经网络的工具包包括 Keras、TensorFlow 和 PyTorch 等，它们对函数的实现过程进行了封装，并提供了完整的网络框架，使得搭建神经网络非常方便。本次实验选用了 PyTorch 进行一维卷积神经网络的搭建，对其他框架有兴趣的读者可以去网上查询相关资料，这里就不再一一赘述。

为提高预测的精确度，该一维卷积神经网络由两层卷积层、一层最大池化层以及两层全连接层堆叠而成，使用 ReLU 作为激活函数，在池化作用后还使用 nn.Dropout()函数避免训练出现过拟合现象。另外，本次实验采用 GPU 运算，提高计算效率，具体实现代码如下。

```
class CNNnetwork(nn.Module):
    def __init__(self):
        super(CNNnetwork, self).__init__()
        self.conv1 = nn.Conv1d(in_channels=1, out_channels=64, kernel_size=2)
        #输出(72-2+0)/1+1 = 71 shape(64,71,None)
        self.conv2 = nn.Conv1d(in_channels=64, out_channels=32,
                               kernel_size=2)  #输出 (71 -2 +0)/1 +1 = 70 shape
                                               #(32,70,None)
        self.pool = nn.MaxPool1d(kernel_size=2, stride=2) #输出 (70 -2 +0)/2 +1 =
                                                          #35 shape (32, 35, None)
        self.fc1 = nn.Linear(32 * 35, 640)
        self.fc2 = nn.Linear(640, 1)
        self.drop = nn.Dropout(0.5)

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu(x)
        x = self.conv2(x)
        x = self.relu(x)
        x = self.pool(x)
        x = self.drop(x)
        x = x.view(-1)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x


device = torch.device("cuda")
net = CNNnetwork().to(device)
```

接着就是定义损失函数和优化器。本实验选择 MSELoss 作为训练的损失函数，选择 Adam 作为训练的优化器，学习率 lr=0.0005，代码如下。

```
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(net.parameters(), lr = 0.0005)
```

(4) 模型训练。

以上就是卷积神经网络的搭建过程，接下来需要对网络进行训练。首先定义迭代次数 epochs=20，同时将网络调整为训练模式。接着利用 for 循环遍历训练所用的样本数据，需要注意的是，在每次更新参数前需要进行梯度归零和初始化。由于输入的数据形状不符合网络输入的格式，还需要对样本数据的形状进行调整，调整为 conv1 的 input_size：(batch_size, in_channels, series_length)，具体模型训练实现的代码如下。

```
#开始训练模型
epochs = 20
net.train()

for epoch in range(epochs):

    for seq, y_train in train_data:
        #每次更新参数前需要进行梯度归零和初始化
        optimizer.zero_grad()
        y_train = y_train.to(device)
        #对样本进行 reshape,调整为 conv1d 的 input size(batch_size, channel, series_length)
        seq = seq.reshape(1, 1, -1).to(device)
        y_pred = net(seq)
        loss = criterion(y_pred, y_train)
        loss.backward()
        optimizer.step()

    print(f'Epoch: {epoch + 1:2} Loss:{loss.item():10.8f}')
```

(5) 数据预测。

模型训练完成以后，选取序列最后的 72 个数据开始预测。首先将网络模式设为 eval 模式，由于需要预测数据集的后 300 个数据，因此需要遍历 300 次，循环的每一步表示时间窗口沿时间序列向后滑动一格，这样每一次最近时刻的真实值都会加入数据集作输入去预测输出新的客流数据，最新加入到时间窗口的值为真实值，而非预测值，可以一定程度上避免误差累积。同时因为是使用训练好的模型进行预测，因此不需要再对模型的权重和偏差进行反向传播和优化。另外在预测完成以后，为了体现预测的效果，将预测值进行逆归一化操作还原为真实的客流值，便于与实际客流进行比较，具体代码如下。

```
future = 300

#选取序列最后 72 个值开始预测
```

```
preds = train_norm[ - Time_window_size:].tolist()

#设置成 eval 模式
net.eval()

for i in range(future):
    seq = torch.FloatTensor(preds[ - Time_window_size:])
    with torch.no_grad():
        seq = seq.reshape(1, 1, - 1).to(device)
        preds.append(net(seq).item())
#逆归一化还原真实值
true_predictions = scaler.inverse_transform(np.array(preds[Time_window_size:]).reshape
( - 1, 1))
```

(6) 预测数据对比。

为了体现预测精度,对预测结果进行可视化,利用 matplotlib.pyplot()函数绘制预测结果和真实值的曲线图,代码如下。比较二者数据的差异,得出网络训练的效果。最终客流进站数据预测曲线图如图 3-30 所示。

```
#对比真实值和预测值
plt.figure(figsize = (12, 4))
plt.grid(True)
plt.plot(y)
x = np.arange(1500, 1800)
plt.plot(x, true_predictions)
plt.show()
```

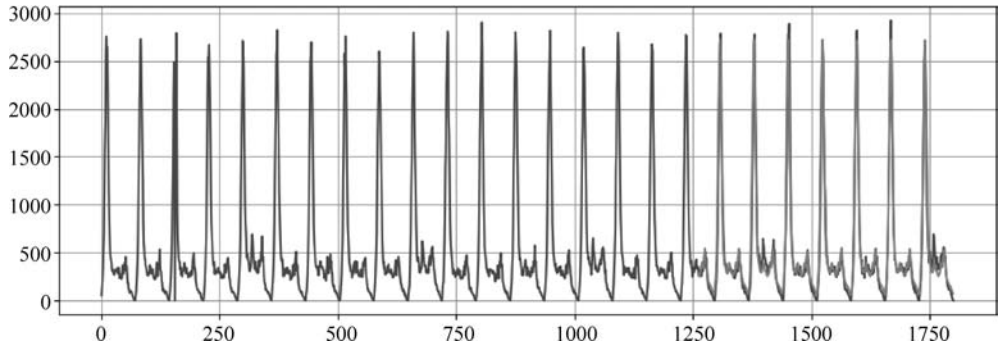

图 3-30 客流进站数据预测曲线图

以上就是利用 PyTorch 实现基于一维卷积神经网络的时间序列预测的全过程。由于 PyTorch 已经将所有的函数过程进行封装,因此网络搭建时可以直接调用,使用起来非常方便。另外,读者可以对模型的参数进行调整,比较预测的精确程度,同时还可以改变输入输出的维度,对其他时间序列数据集进行预测,加深对一维卷积神经网络的理解。

3.3.4　PyTorch 实现二维卷积神经网络手写数字识别

通过 3.3.3 节的介绍，相信读者对使用 PyTorch 搭建一维卷积神经网络有了初步的了解。在实际应用中，二维卷积神经网络的使用更加频繁，尤其在图像处理、图像识别等方面，它有着非常广泛的应用，因此读者有必要掌握二维卷积神经网络的构成以及搭建。接下来将使用 PyTorch 搭建二维卷积神经网络，用来识别 MNIST 手写数字数据集，对手写数字进行分类。读者可以参照以下步骤自行搭建二维卷积神经网络，了解如何使用 PyTorch 实现基于二维卷积神经网络的手写数字识别。以下为具体实现步骤。

(1) 数据集的导入和处理。

本次实验使用的是 MNIST 数据集，该数据集是由美国国家标准与技术研究院收集整理的大型手写数据库，可以直接下载，常用来训练网络，测试网络的准确率。数据集包含 60 000 个训练集和 10 000 个测试集，分为图片和标签，图片是 28×28 的像素矩阵，标签为 0～9 共 10 个数字。

首先要导入数据集，数据样本的格式为[data, label]，第一个存放数据，第二个存放标签。此处使用 torchvision.datasets()函数导入数据集。其中包括设置数据集存放地址 root，对数据格式进行调整 transform(需要对数据进行归一化处理)。由于需要从网络上下载数据集，所以 download=True，数据集的下载时间通常比较慢，需要耐心等待。下载完成后，要利用 DataLoader()函数对训练集和测试集数据分别进行封装。封装的 batch_size=64，shuffle=True 表示将数据进行打乱，num_workers=0 表示不需要多线程工作，具体实现代码如下。

```
import torch
import torch.nn as nn
from matplotlib import pyplot as plt
from PIL import Image
import torchvision
import torchvision.transforms as transforms
import numpy as np
import torch.utils.data as Data
import torch.optim as optim


#首先对数据进行归一化处理，由于 MINIST 是一维的灰度图数据，所以 mean 和 std 只有一维
#Normalized_image = (image - mean) / std
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize(mean =
[0.5], std = [0.5])])
train_dataset = torchvision.datasets.MNIST(root = "./Datasets/MNIST", train = True,
transform = transform, download = False)
test_dataset = torchvision.datasets.MNIST(root = "./Datasets/MNIST", train = False,
transform = transform, download = False)

batch_size = 64
```

```
train_loader = Data.DataLoader(train_dataset, batch_size = batch_size, shuffle = True, num
_workers = 0)
test_loader = Data.DataLoader(test_dataset, batch_size = batch_size, shuffle = True, num_
workers = 0)
```

(2) 网络搭建。

在数据集导入并处理完成后,就开始搭建二维卷积神经网络模型。首先对一些参数进行设定。num_classes=10 表示共有十种类别的数据图像,学习率 lr=0.001,迭代次数 epochs=20,运行设备 device 选择 GPU(cuda)进行网络的测试优化。接着利用 PyTorch 搭建二维卷积神经网络,还是以继承 nn.Module 的方式创建 ConvModule 类。该网络由两层卷积层、两层最大池化层以及三层全连接层堆叠而成,激活函数选择 ReLU 函数。二维卷积 Conv2d()函数参数表如表 3-1 所示。

表 3-1 Conv2d 函数参数表

Conv2d 参数	含 义
in_channels(int)	输入信号的通道数目
out_channels(int)	输出信号的通道数目(由卷积核个数决定)
kerner_size(int or tuple)	卷积核的尺寸
stride(int or tuple)	卷积步长
padding(int or tuple)	输入的每一边补充 0 的层数

在搭建卷积神经网络时,很多公式、函数等都是 PyTorch 打包封装好的,需要时直接调用就可以,非常容易上手。不过需要特别注意的是,函数数据输入与输出维度的确定,必须要确保数据维度在网络各个层次的变化与函数输入输出一致,这样模型才能正常运行,否则将会报错。关于维度的变化,建议读者亲自推导,这样将会加深对于卷积神经网络数据输入与输出维度的理解。另外,网络损失函数选择了交叉熵损失函数,优化器选择了 Adam 优化器,具体实现代码如下。

```
num_classes = 10
lr = 0.001
epochs = 20
device = torch.device("cuda:0")


#pytorch 封装卷积层
class ConvModule(nn.Module):
    def __init__(self):
        super(ConvModule, self).__init__()
        #定义两层卷积层:
        self.conv2d = nn.Sequential(
            #第一层 input_size = (1,28,28)
            nn.Conv2d(in_channels = 1, out_channels = 32, kernel_size = 3, stride = 1,
padding = 1),
            nn.MaxPool2d(2, 2),
```

```
            nn.ReLU(inplace = True),   #inplace 表示是否进行覆盖计算
            #第二层
            nn.Conv2d(in_channels = 32, out_channels = 64, kernel_size = 3, stride = 1,
padding = 1),
            nn.MaxPool2d(2, 2),
            nn.ReLU(inplace = True),
        )
        #输出层,将通道数变为分类数
        self.relu = nn.ReLU()
        self.fc1 = nn.Linear(64 * 7 * 7, 1024)
        self.fc2 = nn.Linear(1024, 512)
        self.fc3 = nn.Linear(512, num_classes)

    def forward(self, x):
        out = self.conv2d(x)
        #将数据平整成一维
        out = out.view(-1, 64 * 7 * 7)
        out = self.fc1(out)
        out = self.relu(out)
        out = self.fc2(out)
        out = self.relu(out)
        out = self.fc3(out)
        return out


net = ConvModule().to(device)

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(net.parameters(), lr = lr)
```

模型搭建完成以后,要定义训练函数和测试函数。两个函数的构成相差不大,均是使用 For 循环对 data_loader 的每个 Batch 进行遍历,然后运行网络,记录每次迭代的损失,最后返回整个过程的平均损失和模型准确率。需要注意的是,在训练函数中,要把网络指定为训练模式,每次更新参数前需要对梯度进行归零和初始化;在测试函数中则需要把网络指定为 eval 模式,测试时使用的参数是经过训练优化得到的,所以无须对权重和偏置求导,即卷积神经网络在 with torch.no_grad()的环境下运行,函数代码如下。

```
#训练函数
def train_epoch(net, data_loader, device):
    net.train()                          #指定当前为训练模式
    train_batch_num = len(data_loader)   #记录共有多少个 epoch
    total_loss = 0                       #记录 LOSS
    correct = 0                          #记录共有多少个样本被正确分类
    sample_num = 0                       #记录样本数

    #遍历每个 batch 进行训练
```

```
    for batch_id, (inputs, labels) in enumerate(data_loader):
        #将每个图片放入指定的 device 中
        inputs = inputs.to(device).float()
        #将图片标签放入指定的 device 中
        labels = labels.to(device).long()
        #梯度清零
        optimizer.zero_grad()
        #计算结果
        output = net(inputs)
        #计算损失
        loss = criterion(output, labels.squeeze())
        #进行反向传播
        loss.backward()
        optimizer.step()
        #累加 loss
        total_loss += loss.item()
        #找出每个样本值的最大 idx,即代表预测此图片属于哪个类别
        prediction = torch.argmax(output, 1)
        #统计预测正确的类别数量
        correct += (prediction == labels).sum().item()
        #累加当前样本总数
        sample_num += len(prediction)

    #计算平均 loss 和准确率
    loss = total_loss / train_batch_num
    acc = correct / sample_num
    return loss, acc


def test_epoch(net, data_loader, device):
    net.eval()  #指定当前模式为测试模式
    test_batch_num = len(data_loader)
    total_loss = 0
    correct = 0
    sample_num = 0
    #指定不进行梯度变化:
    with torch.no_grad():
        for batch_idx, (data, target) in enumerate(data_loader):
            data = data.to(device).float()
            target = target.to(device).long()
            output = net(data)
            loss = criterion(output, target)
            total_loss += loss.item()
            prediction = torch.argmax(output, 1)
            correct += (prediction == target).sum().item()
            sample_num += len(prediction)
    loss = total_loss / test_batch_num
    acc = correct / sample_num
    return loss, acc
```

（3）模型训练。

定义好训练和测试函数以后，就可以进行网络模型的训练了，首先分别创建 train_loss、train_acc、test_loss、test_acc 四个列表，用于存储每一次迭代的 Loss 以及 Acc，便于后面可视化展示。紧接着使用 For 循环进行训练迭代，每次迭代完都输出模型的损失以及准确率，具体代码如下。

```
#存储每一个 epoch 的 loss 与 acc 的变化，便于后面的可视化
train_loss_list = []
train_acc_list = []
test_loss_list = []
test_acc_list = []

#进行训练
for epoch in range(epochs):
    train_loss, train_acc = train_epoch(net, data_loader=train_loader, device=device)
    test_loss, test_acc = test_epoch(net, data_loader=test_loader, device=device)

    train_loss_list.append(train_loss)
    train_acc_list.append(train_acc)
    test_loss_list.append(test_loss)
    test_acc_list.append(test_acc)
    print('epoch %d, train_loss %.6f, train_acc %.6f' % (epoch + 1, train_loss, train_acc))
    print('test_loss %.6f , test_acc %.6f' % (test_loss, test_acc))
```

（4）网络损失、准确率的可视化。

在收到每次迭代返回的损失以及准确率以后，使用 Matplotlib 包画出训练和测试时的损失曲线及准确率曲线，并且将曲线图像分别存储为“Loss.jpg”“Acc.jpg”的 jpg 格式图片，具体代码如下。

```
x = np.linspace(0, len(train_loss_list), len(train_loss_list))
plt.plot(x, train_loss_list, label="train_loss", linewidth=1.5)
plt.plot(x, test_loss_list, label="test_loss", linewidth=1.5)
plt.xlabel("epoch")
plt.ylabel("loss")
plt.legend()
plt.show()
plt.savefig('Loss.jpg')
plt.clf()

x = np.linspace(0, len(train_acc_list), len(train_acc_list))
plt.plot(x, train_acc_list, label="train_acc", linewidth=1.5)
plt.plot(x, test_acc_list, label="test_acc", linewidth=1.5)
plt.xlabel("epoch")
plt.ylabel("acc")
plt.legend()
plt.show()
plt.savefig("Acc.jpg")
```

(5) 模型评估。

网络经过 20 次迭代以后,训练集和测试集的损失由原来的 0.31 和 0.18 降到了 0.04 和 0.10,准确率都达到 98%,表明网络模型的训练效果非常可观,分类效果准确。运行得到的模型损失曲线图以及模型准确率曲线图,分别如图 3-31 和图 3-32 所示。

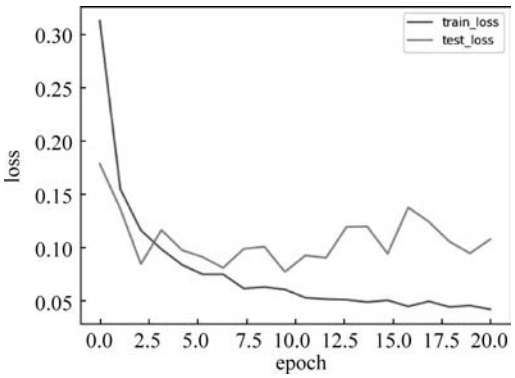

图 3-31 模型损失曲线图

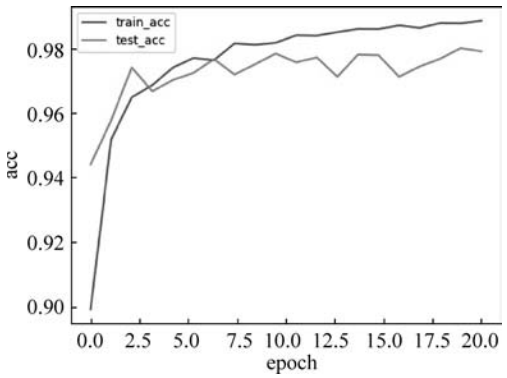

图 3-32 模型准确率曲线图

3.4 图卷积神经网络

3.4.1 图卷积神经网络简介

图卷积神经网络(Graph Convolutional Network,GCN)是近些年逐渐流行的一种神经网络,发展到现在已经有无数改进的版本,在图网络领域的地位如同卷积操作在图像处理里的地位一样重要。图卷积神经网络与传统的网络模型 LSTM 和 CNN 所处理的数据类型有所不同。LSTM 和 CNN 只能用于网络结构的数据,而图卷积神经网络能够处理具有广义拓扑图结构的数据,并深入发掘其特征和规律。

在具体介绍图卷积神经网络之前,先介绍一些图的基本知识。

(1) 图(Graph)。

定义一张图 $G=(V,E)$,V 中元素为图的顶点,E 中元素为图的边。图中边为无序时为无向图,有序时为有向图。

(2) 邻居(Neighborhood)。

顶点 V_i 的邻居 N_i:$\{V_i \in V | V_i V_j \in E\}$。在无向图中,如果顶点 V_i 是顶点 V_j 的邻居,那么顶点 V_j 也是顶点 V_i 的邻居。

(3) 度矩阵(Degree)。

度矩阵是对角阵,对角上的元素为各个顶点的度。顶点 V_i 的度表示和该顶点相关联的边的数量。

无向图中顶点 V_i 的度 $d(V_i)=N_i$。有向图中,顶点 V_i 的度分为顶点 V_i 的出度和入度,即从顶点 V_i 出去的有向边的数量和进入顶点 V_i 的有向边的数量。

$$\Delta(G)=\begin{pmatrix} d(V_1) & \cdots & 0 \\ \vdots & & \vdots \\ 0 & \cdots & d(V_n) \end{pmatrix}$$

(4) 邻接矩阵(Adjacency)。

邻接矩阵表示顶点间关系，是 n 阶方阵(n 为顶点数量)。

邻接矩阵分为有向图邻接矩阵和无向图邻接矩阵。无向图邻接矩阵是对称矩阵，而有向图的邻接矩阵不一定对称。

$$[A(G)]_{ij}=\begin{cases}1, & V_iV_j \in E \\ 0, & \text{其他}\end{cases}$$

现实中更多重要的数据集都是用图的形式存储的，例如，知识图谱、社交网络、通信网络、蛋白质分子结构等。这些图网络的形式并不像图像一样是排列整齐的矩阵，而是具有空间拓扑图结构的不规则数据。图 3-33 所示为图论中所定义的拓扑图。

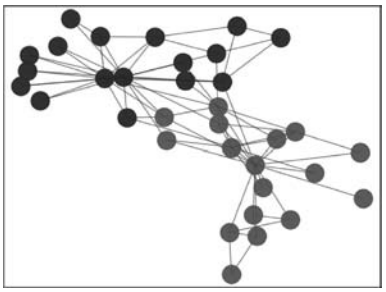

图 3-33　社交网络拓扑图

对于具有拓扑结构的图数据，可以按照定义用于网格化结构数据的卷积的思想来定义。拓扑图上的卷积操作，如图 3-34 所示，将每个节点的邻居节点的特征传播到该节点，再进行加权，就可以得到该点的聚合特征值。类似于 CNN，图卷积也共享权重，不过不同于 CNN 中每个 kernel 的权重都是规则的矩阵，按照对应位置分配，图卷积中的权重通常是一个集合。在对一个节点计算聚合特征值时，按一定规律将参与聚合的所有点分配为多个不同的子集，同一个子集内的节点采用相同的权重，从而实现权重共享。例如，对于图 3-34，可以规定和红色点距离为 1 的点为 1 邻域子集，距离为 2 的点为 2 邻域子集。当然，也可以采用更加复杂的策略，如按照距离图重心的远近来分配权重。权重的分配策略有时也称为 label 策略，对邻接节点分配 label，label 相同节点的共享一个权重。

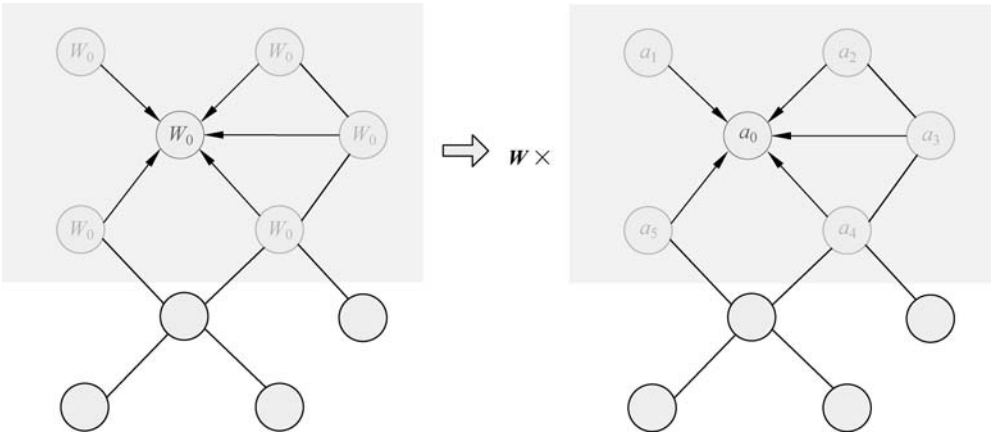

图 3-34　拓扑图上的卷积操作

对于拓扑图结构的数据，图卷积神经网络能很好地抽取节点与节点直接的连接关系。因此，GCN 近几年得到了快速发展，从谱图卷积滤波器，到切比雪夫多项式滤波器，再到一阶近似滤波器，GCN 的表现能力得到了极大的提升。

目前，图上的卷积定义基本上可以分为两类，一个是基于谱的图卷积，它们通过傅里叶变换将节点映射到频域空间，通过在频域空间上做乘积来实现时域上的卷积，最后再将做完乘积的特征映射回时域空间。而另一种是基于空间域的图卷积，与传统的 CNN 很像，只不过在图结构上更难定义节点的邻居以及与邻居之间的关系。

定义一张图 $G=(V,E,\boldsymbol{A})$，V 是图的节点集合，E 是图的边集合，$\boldsymbol{A}\in R^{n\times n}$ 代表该网络的邻接矩阵，则 GCN 卷积操作定义如公式(3-14)和公式(3-15)所示(Kipf 等人提出的 GCN 版本)。

$$\boldsymbol{H}^{l+1}=f(\boldsymbol{H}^{l},\boldsymbol{A})=\sigma(\hat{\boldsymbol{D}}^{-\frac{1}{2}}\hat{\boldsymbol{A}}\hat{\boldsymbol{D}}^{-\frac{1}{2}}\boldsymbol{H}^{l}\boldsymbol{W}^{l}+\boldsymbol{b}^{l}) \tag{3-14}$$

$$\boldsymbol{H}^{l'}=\hat{\boldsymbol{D}}^{-\frac{1}{2}}\hat{\boldsymbol{A}}\hat{\boldsymbol{D}}^{-\frac{1}{2}}\boldsymbol{H}^{l} \tag{3-15}$$

其中，$\hat{\boldsymbol{A}}=\boldsymbol{A}+\boldsymbol{I}$，$\boldsymbol{A}\in R^{n\times n}$ 为邻接矩阵，$\boldsymbol{I}\in R^{n\times n}$ 为单位矩阵，$\hat{\boldsymbol{D}}$ 是 $\hat{\boldsymbol{A}}$ 的对角节点度矩阵，$\boldsymbol{W}$ 为第 l 层的参数矩阵，$\boldsymbol{b}$ 为第 l 层的偏置向量，$\boldsymbol{H}\in R^{n\times t}$ 为特征矩阵，其中，n 为节点数目，t 为每个节点的特征数目，$\boldsymbol{H}'\in R^{n\times t}$ 为含有拓扑信息的特征矩阵，$\sigma(\cdot)$ 为激活函数。

3.4.2 NumPy 实现图卷积神经网络

本节将利用 NumPy 实现图卷积神经网络中的前向传播部分，以帮助读者加深对图卷积中的“卷积”操作的理解，本节使用的 GCN 层的传播规则如式(3-15)所示。笔者强烈建议读者实现以下代码部分，实现完成后必定对公式(3-14)和公式(3-15)有更清楚的理解。

首先构建一个简单的有向图，如图 3-35 所示。

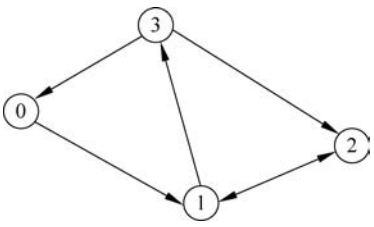

图 3-35 有向图

使用 NumPy 编写图 3-35 的邻接矩阵 $\boldsymbol{A}$，代码如下。

```
import numpy as np
from math import sqrt
A = np.matrix([
    [0, 1, 0, 0],
    [0, 0, 1, 1],
    [0, 1, 0, 0],
    [1, 0, 1, 0]],
    dtype = float)
```

基于每个节点的索引为其生成两个整数特征，生成特征矩阵为 $\boldsymbol{X}$，代码如下。

```
X = np.matrix([
    [i, -i]
    for i in range(A.shape[0])], dtype = float)
```

为每个节点添加一个自环，这可以通过在应用传播规则之前将邻接矩阵 $\boldsymbol{A}$ 与单位矩阵 $\boldsymbol{I}$ 相加来实现。生成的包含自己特征的邻接矩阵为 $\boldsymbol{A}$_hat，此步骤对应公式为 $\hat{\boldsymbol{A}}=\boldsymbol{A}+\boldsymbol{I}$，代码如下。

```
I = np.matrix(np.eye(A.shape[0]))    #np.eye()返回的是一个二维的数组(N,M),对角线的地
                                     #方为 1,其余的地方为 0
A_hat = A + I
```

生成节点度矩阵 $\boldsymbol{D}$_hat，将度矩阵处理为 $\hat{\boldsymbol{D}}^{-\frac{1}{2}}$ 形式，然后生成带有拓扑信息的特征矩阵，此步骤对应公式为式(3-15)，代码如下。

```
D_hat = np.array(np.sum(A_hat, axis = 0))[0]
D_hat_sqrt = [sqrt(x) for x in D_hat]
D_hat_sqrt = np.array(np.diag(D_hat_sqrt))
D_hat_sqrtm_inv = np.linalg.inv(D_hat_sqrt)   #开方后求逆即为矩阵的 -1/2 次方
D_A_final = np.dot(D_hat_sqrtm_inv, A_hat)
D_A_final = np.dot(D_A_final, D_hat_sqrtm_inv)
```

添加权重 W 与偏置 b，代码如下。

```
W = np.matrix([
    [1, -1, 1, -1],
    [-1, 1, -1, 1],
    [1, -1, 1, -1],
    [-1, 1, -1, 1]
])
b = np.matrix([
    [1, 0, 1, 0],
    [0, 1, 0, 1],
    [1, 0, 1, 0],
    [0, 1, 0, 1]
])
```

添加激活函数，选择保持特征矩阵的维度，并应用 ReLU 激活函数。ReLU 函数的公式是 $f(x)=\max(0,x)$，代码如下。

```
def relu(x):
    return (abs(x) + x) / 2
```

最后，应用传播规则生成下一层的特征矩阵，此步骤对应公式为式(3-14)，代码如下。

```
output = relu(D_A_final * W + b)
```

3.4.3 PyTorch 实现图卷积神经网络时间序列预测

(1) 数据准备。

本节使用的数据集为深圳市 2015 年 1 月 1 日至 1 月 31 日的出租车轨迹数据集。实验数据主要包括两部分,第一部分是 156×156 邻接矩阵,描述道路之间的空间关系。每行代表一条道路,矩阵中的值代表道路之间的连通性。第二部分是特征矩阵,它描述了每条道路上的速度随时间的变化。每一行代表一条路,每一列是不同时间段道路上的交通速度。每 15min 汇总一次每条路上的交通速度,数据维度为 2976×156,使用过去 10 个时间步的数据预测未来 1 个时间步的数据。选取 80%的数据作为训练集,20%的数据作为测试集,又将训练集中后 10%的数据作为验证集,对交通速度进行实时预测。其中,自定义函数的参数含义如表 3-2 所示。

表 3-2 自定义函数的参数含义

参 数	取 值	参数含义
time_interval	15	时间粒度
time_lag	10	使用的历史时间步
tg_in_one_day	96	一天内有多少个时间步
forecast_day_number	5	预测的天数
is_train	默认 True	是否获取训练集
is_val	默认 False	是否获取验证集
val_rate	0.1	验证集所占比例
pre_len	1	预测未来时间步

构建一个将数据集划分为训练集、验证集和测试集的函数,代码如下。

```
import torch
from torch.utils.data import Dataset
import numpy as np

"""
Parameter:
time_interval, time_lag, tg_in_one_day, forecast_day_number, is_train = True, is_val =
False, val_rate = 0.1, pre_len
"""


class Traffic_speed(Dataset):
    def __init__(self, time_interval, time_lag, tg_in_one_day, forecast_day_number, speed_
data, pre_len, is_train = True, is_val = False, val_rate = 0.1):
        super().__init__()
        #此部分的作用是将数据集划分为训练集、验证集、测试集.
        #完成后 X 的维度为 num * 156 * 10,10 代表 10 个时间步,Y 的维度为 num * 156 * 1
```

```
        #X 为临近同一时段的 10 个时间步
        #Y 为 156 条主干道未来 1 个时间步
        self.time_interval = time_interval
        self.time_lag = time_lag
        self.tg_in_one_day = tg_in_one_day
        self.forecast_day_number = forecast_day_number
        self.tg_in_one_week = self.tg_in_one_day * self.forecast_day_number
        self.speed_data = np.loadtxt(speed_data, delimiter=",").T  #对数据进行转置
        self.max_speed = np.max(self.speed_data)
        self.min_speed = np.min(self.speed_data)
        self.is_train = is_train
        self.is_val = is_val
        self.val_rate = val_rate
        self.pre_len = pre_len

        #Normalization
        self.speed_data_norm = np.zeros((self.speed_data.shape[0], self.speed_data.shape
[1]))
        for i in range(len(self.speed_data)):
            for j in range(len(self.speed_data[0])):
                self.speed_data_norm[i, j] = round((self.speed_data[i, j] - self.min_
speed) / (self.max_speed - self.min_speed), 5)
        if self.is_train:
            self.start_index = self.tg_in_one_week + time_lag
            self.end_index = len(self.speed_data[0]) - self.tg_in_one_day * self.forecast_
day_number - self.pre_len
        else:
            self.start_index = len(self.speed_data[0]) - self.tg_in_one_day * self.
forecast_day_number
            self.end_index = len(self.speed_data[0]) - self.pre_len

        self.X = [[] for index in range(self.start_index, self.end_index)]
        self.Y = []
        self.Y_original = []
        #print(self.start_index, self.end_index)
        for index in range(self.start_index, self.end_index):
            temp = self.speed_data_norm[:, index - self.time_lag: index]  #邻近几个时间
                                                                          #段的速度
            temp = temp.tolist()
            self.X[index - self.start_index] = temp
            self.Y.append(self.speed_data_norm[:, index:index + self.pre_len])
        self.X, self.Y = torch.from_numpy(np.array(self.X)), torch.from_numpy(np.array
(self.Y))  #(num, 156, time_lag)

        #if val is not zero
        if self.val_rate * len(self.X) != 0:
            val_len = int(self.val_rate * len(self.X))
            train_len = len(self.X) - val_len
```

```
            if self.is_val:
                self.X = self.X[-val_len:]
                self.Y = self.Y[-val_len:]
            else:
                self.X = self.X[:train_len]
                self.Y = self.Y[:train_len]
        print("X.shape", self.X.shape, "Y.shape", self.Y.shape)

        if not self.is_train:
            for index in range(self.start_index, self.end_index):
                self.Y_original.append(self.speed_data[:, index:index + self.pre_len])
    #the predicted speed before normalization
            self.Y_original = torch.from_numpy(np.array(self.Y_original))

    def get_max_min_speed(self):
        return self.max_speed, self.min_speed

    def __getitem__(self, item):
        if self.is_train:
            return self.X[item], self.Y[item]
        else:
            return self.X[item], self.Y[item], self.Y_original[item]

    def __len__(self):
        return len(self.X)
```

在 PyTorch 中，DataLoader 是进行数据载入的部件。必须将数据载入后，再进行深度学习模型的训练。本节使用自己构建的 DataLoader 加载数据，代码如下。

```
def get_speed_dataloader(time_interval=15, time_lag=5, tg_in_one_day=72, forecast_day
_number=5, pre_len=1, batch_size=32):
    #train speed data loader
    print("train speed")
    speed_train = Traffic_speed(time_interval=time_interval, time_lag=time_lag, tg_in_
one_day=tg_in_one_day, forecast_day_number=forecast_day_number,
                            pre_len=pre_len, speed_data=speed_data, is_train=True, is_
val=False, val_rate=0.1)
    max_speed, min_speed = speed_train.get_max_min_speed()
    speed_data_loader_train = DataLoader(speed_train, batch_size=batch_size, shuffle=
False)

    #validation speed data loader
    print("val speed")
    speed_val = Traffic_speed(time_interval=time_interval, time_lag=time_lag, tg_in_
one_day=tg_in_one_day, forecast_day_number=forecast_day_number,
                            pre_len=pre_len, speed_data=speed_data, is_train=True, is_
val=True, val_rate=0.1)
```

```
    speed_data_loader_val = DataLoader(speed_val, batch_size = batch_size, shuffle =
False)

    #test speed data loader
    print("test speed")
    speed_test = Traffic_speed(time_interval = time_interval, time_lag = time_lag, tg_in_
one_day = tg_in_one_day, forecast_day_number = forecast_day_number,
                           pre_len = pre_len, speed_data = speed_data, is_train = False, is_
val = False, val_rate = 0)
    speed_data_loader_test = DataLoader(speed_test, batch_size = batch_size, shuffle =
False)

    return speed_data_loader_train, speed_data_loader_val, speed_data_loader_test, max_
speed, min_speed
```

(2) 模型构建。

根据式(3-14)构建 GCN 层,首先计算式中的固定值 $\hat{\boldsymbol{D}}^{-\frac{1}{2}}\hat{\boldsymbol{A}}\hat{\boldsymbol{D}}^{-\frac{1}{2}}$,其计算过程的代码如下。

```
import tensorflow as tf
import scipy.sparse as sp
import numpy as np
from math import sqrt

class GetLaplacian:
    def __init__(self,adjacency):
        self.adjacency = adjacency

    def get_normalized_adj(self, station_num):
        I = np.matrix(np.eye(station_num))
        A_hat = self.adjacency + I
        D_hat = np.array(np.sum(A_hat, axis = 0))[0]
        D_hat_sqrt = [sqrt(x) for x in D_hat]
        D_hat_sqrt = np.array(np.diag(D_hat_sqrt))
        D_hat_sqrtm_inv = np.linalg.inv(D_hat_sqrt)  #开方后求逆即为矩阵的 -1/2 次方
        #D_A_final = D_hat ** -1/2 * A_hat * D_hat ** -1/2
        D_A_final = np.dot(D_hat_sqrtm_inv, A_hat)
        D_A_final = np.dot(D_A_final, D_hat_sqrtm_inv)
        #print(D_A_final.shape)
        return np.array(D_A_final, dtype = "float32")

    #Method2 to calculate laplacian
    def normalized_adj(self):
        adj = sp.coo_matrix(self.adjacency)
        rowsum = np.array(adj.sum(1))
        d_inv_sqrt = np.power(rowsum, -0.5).flatten()
        d_inv_sqrt[np.isinf(d_inv_sqrt)] = 0.
```

```
        d_mat_inv_sqrt = sp.diags(d_inv_sqrt)
        normalized_adj = adj.dot(d_mat_inv_sqrt).transpose().dot(d_mat_inv_sqrt).tocoo()
        normalized_adj = normalized_adj.astype(np.float32)
        return normalized_adj


    def sparse_to_tuple(self, mx):
        mx = mx.tocoo()
        coords = np.vstack((mx.row, mx.col)).transpose()
        L = tf.SparseTensor(coords, mx.data, mx.shape)
        return tf.sparse.reorder(L)


    def calculate_laplacian(self):
        adj = self.normalized_adj(np.array(self.adjacency) + sp.eye(np.array(self.
adjacency).shape[0]))
        adj = sp.csr_matrix(adj)
        adj = adj.astype(np.float32)
        return self.sparse_to_tuple(adj)
```

GCN 层的输入为特征矩阵 $\boldsymbol{H}^l$ 以及邻接矩阵 $\hat{\boldsymbol{D}}^{-\frac{1}{2}}\hat{\boldsymbol{A}}\hat{\boldsymbol{D}}^{-\frac{1}{2}}$。在本节中，特征矩阵即为每条主干道的道路速度矩阵，邻接矩阵为主干道之间的连通关系矩阵。初始化 GCN 层时，需要确定每个节点的输入特征个数 in_features 以及输出特征个数 out_features，GCN 层的代码如下。

```
import math
import torch
from torch.nn.parameter import Parameter
from torch.nn.modules.module import Module

class GraphConvolution(Module):
    """
    Simple GCN layer, similar to https://arxiv.org/abs/1609.02907
    """

    def __init__(self, in_features, out_features, bias=True):
        super(GraphConvolution, self).__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.weight = Parameter(torch.FloatTensor(in_features, out_features).type
(torch.float32))
        if bias:
            self.bias = Parameter(torch.FloatTensor(out_features).type(torch.float32))
        else:
            self.register_parameter('bias', None)
        self.reset_parameters()
```

```
    def reset_parameters(self):
        stdv = 1. / math.sqrt(self.weight.size(1))
        self.weight.data.uniform_(-stdv, stdv)
        if self.bias is not None:
            self.bias.data.uniform_(-stdv, stdv)

    def forward(self, x, adj):
        support = torch.matmul(x, self.weight.type(torch.float32))
        output = torch.bmm(adj.unsqueeze(0).expand(support.size(0), *adj.size()), support)
        if self.bias is not None:
            return output + self.bias.type(torch.float32)
        else:
            return output

    def __repr__(self):
        return self.__class__.__name__ + '(' + str(self.in_features) + ' -> ' + str
(self.out_features) + ')'
```

本节构建的模型是由两层 GCN 以及三层全连接网络堆叠而成，其 GCN 层的输入特征个数与输出特征个数均为 time_lag，交通速度的数据先经过 GCN 层处理，然后使用全连接层输出结果。该模型输入的形状为（batchsize×156×30），输出的形状为（batchsize×156×1），代码如下。

```
import torch
from torch import nn
import torch.nn.functional as F
from model.GCN_layers import GraphConvolution

class Model(nn.Module):
   def __init__(self, time_lag, pre_len, station_num, device):
      super().__init__()
      self.time_lag = time_lag
      self.pre_len = pre_len
      self.station_num = station_num
      self.device = device
      self.GCN1 = GraphConvolution(in_features = self.time_lag, out_features = self.time_lag).
to(self.device)
      self.GCN2 = GraphConvolution(in_features = self.time_lag, out_features = self.time
_lag).to(self.device)
      self.linear1 = nn.Linear(in_features = self.time_lag * self.station_num, out_
features = 1024).to(self.device)
      self.linear2 = nn.Linear(in_features = 1024, out_features = 512).to(self.device)
      self.linear3 = nn.Linear(in_features = 512, out_features = self.station_num * self.pre_
len).to(self.device)
   def forward(self, speed, adj):
```

```
        speed = speed.to(self.device)                    #[32,156,10]
        adj = adj.to(self.device)
        speed = self.GCN1(x=speed, adj=adj)              #(32, 156, 10)
        output = self.GCN2(x=speed, adj=adj)             # [32, 156, 10]
        output = output.reshape(output.size()[0], -1)    #(32, 156*10)
        output = F.relu(self.linear1(output))            #(32, 1024)
        output = F.relu(self.linear2(output))            #(32, 512)
        output = self.linear3(output)                    #(32, 156*pre_len)
        output = output.reshape(output.size()[0], self.station_num, self.pre_len)
                                                         #( 64, 156, pre_len)
        return output
```

（3）模型终止与评价。

模型终止部分采用 EarlyStopping 方法，该方法主要有两个作用，一是借助验证集损失，来保存截至当前的最优模型；二是当模型训练到一定标准后终止模型训练，代码如下。

```
import numpy as np
import torch

class EarlyStopping:
    """ Early stops the training if validation loss doesn't improve after a given
patience."""
    def __init__(self, patience=7, verbose=False):
        """
        Args:
            patience (int): How long to wait after last time validation loss improved.
                            Default: 7
            verbose (bool): If True, prints a message for each validation loss improvement.
                            Default: False
        """
        self.patience = patience
        self.verbose = verbose
        self.counter = 0
        self.best_score = None
        self.early_stop = False
        self.val_loss_min = np.Inf

    def __call__(self, val_loss, model_dict, model, epoch, save_path):

        score = -val_loss

        if self.best_score is None:
            self.best_score = score
            self.save_checkpoint(val_loss, model_dict, model, epoch, save_path)
        elif score < self.best_score:
            self.counter += 1
```

```
            print(
                f'EarlyStopping counter: {self.counter} out of {self.patience}', self.val
_loss_min
            )
            if self.counter >= self.patience:
                self.early_stop = True
        else:
            self.best_score = score
            self.save_checkpoint(val_loss, model_dict, model, epoch, save_path)
            self.counter = 0

    def save_checkpoint(self, val_loss, model_dict, model, epoch, save_path):
        '''Saves model when validation loss decrease.'''
        if self.verbose:
            print(
                f'Validation loss decreased ({self.val_loss_min:.8f} --> {val_loss:.
8f}). Saving model ...'
            )
        torch.save(model_dict, save_path + "/" + "model_dict_checkpoint_{}_{:.8f}.
pth".format(epoch, val_loss))
        #torch.save(model, save_path + "/" + "model_checkpoint_{}_{:.8f}.pth".format
(epoch, val_loss))
        self.val_loss_min = val_loss
```

本案例主要采用了均方根误差 RMSE，皮尔逊相关系数 R^2，平均绝对误差 MAE，加权平均绝对百分比误差 WMAPE 四个指标对模型进行评价，并构建了一个模型评价函数，函数输入为真实值和预测值，输出值为相应的评价指标，模型评价代码如下。

```
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import r2_score
from math import sqrt
import numpy as np

"""
class Metrics
func : define metrics for 2-d array
parameter
Y_true : grand truth (n, 156)
Y_pred : prediction (n, 156)
"""
class Metrics:
    def __init__(self, Y_true, Y_pred):
        self.Y_true = Y_true
        self.Y_pred = Y_pred

    def weighted_mean_absolute_percentage_error(self):
```

```
        total_sum = np.sum(self.Y_true)
        average = []
        for i in range(len(self.Y_true)):
            for j in range(len(self.Y_true[0])):
                if self.Y_true[i][j] > 0:
                    #加权 (y_true[i][j]/np.sum(y_true[i])) *
                    temp = (self.Y_true[i][j] / total_sum) * np.abs((self.Y_true[i][j] -
self.Y_pred[i][j]) / self.Y_true[i][j])
                    average.append(temp)
        return np.sum(average)

    def evaluate_performance(self):
        RMSE = sqrt(mean_squared_error(self.Y_true, self.Y_pred))
        R2 = r2_score(self.Y_true, self.Y_pred)
        MAE = mean_absolute_error(self.Y_true, self.Y_pred)
        WMAPE = self.weighted_mean_absolute_percentage_error()
        return RMSE, R2, MAE, WMAPE


class Metrics_1d:
    def __init__(self, Y_true, Y_pred):
        self.Y_true = Y_true
        self.Y_pred = Y_pred

    def weighted_mean_absolute_percentage_error(self):
        total_sum = np.sum(self.Y_true)
        average = []
        for i in range(len(self.Y_true)):
            if self.Y_true[i] > 0:
                #加权 (y_true[i][j]/np.sum(y_true[i])) *
                temp = (self.Y_true[i] / total_sum) * np.abs((self.Y_true[i] - self.Y_pred
[i]) / self.Y_true[i])
                average.append(temp)
        return np.sum(average)

    def evaluate_performance(self):
        RMSE = sqrt(mean_squared_error(self.Y_true, self.Y_pred))
        R2 = r2_score(self.Y_true, self.Y_pred)
        MAE = mean_absolute_error(self.Y_true, self.Y_pred)
        WMAPE = self.weighted_mean_absolute_percentage_error()
        return RMSE, R2, MAE, WMAPE
```

(4) 模型训练及测试。

在模型训练部分，首先对模型中的参数进行赋值，并加载所需要的道路速度数据、邻接矩阵数据、模型，并设置 EarlyStopping 的参数，本案例设置 EarlyStopping 的

patience 为 100。对于每一个 Epoch，先进行训练，再进行验证。每一次验证结束后，借助 EarlyStopping 判断是否保存当前模型以及是否终止模型训练，训练及验证的代码如下。

```
import numpy as np
import os, time, torch
from torch import nn
from torch.utils.tensorboard import SummaryWriter
from utils.utils import GetLaplacian
from model.main_model import Model
from utils.earlystopping import EarlyStopping
from data.get_dataloader import get_speed_dataloader
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

print(device)

epoch_num = 5000
lr = 0.001
time_interval = 15
time_lag = 10
tg_in_one_day = 72
forecast_day_number = 5
pre_len = 1
batch_size = 32
station_num = 156
model_type = 'ours'
TIMESTAMP = str(time.strftime("%Y_%m_%d_%H_%M_%S"))
save_dir = './save_model/' + model_type + '_' + TIMESTAMP
if not os.path.exists(save_dir):
    os.makedirs(save_dir)

speed_data_loader_train, speed_data_loader_val, speed_data_loader_test, max_speed, min_
speed = \
    get_speed_dataloader(time_interval = time_interval, time_lag = time_lag, tg_in_one_day
 = tg_in_one_day, forecast_day_number = forecast_day_number, pre_len = pre_len, batch_size
 = batch_size)

#get normalized adj
adjacency = np.loadtxt('./data/sz_adj1.csv', delimiter = ",")
adjacency = torch.tensor(GetLaplacian(adjacency).get_normalized_adj(station_num)).type
(tor'h.float32).to(device)

global_start_time = time.time()
writer = SummaryWriter()


model = Model(time_lag, pre_len, station_num, device)
print(model)
```

```
if torch.cuda.is_available():
    model.cuda()

model = model.to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=lr)
mse = torch.nn.MSELoss().to(device)

temp_time = time.time()
early_stopping = EarlyStopping(patience=100, verbose=True)
for epoch in range(0, epoch_num):
    #model train
    train_loss = 0
    model.train()
    for speed_tr in enumerate(speed_data_loader_train):
        i_batch, (train_speed_X, train_speed_Y) = speed_tr
        train_speed_X, train_speed_Y = train_speed_X.type(torch.float32).to(device),
train_speed_Y.type(torch.float32).to(device)
        target = model(train_speed_X, adjacency)
        loss = mse(input=train_speed_Y, target=target)
        train_loss += loss.item()
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    with torch.no_grad():
        #model validation
        model.eval()
        val_loss = 0
        for speed_val in enumerate(speed_data_loader_val):
            i_batch, (val_speed_X, val_speed_Y) = speed_tr
            val_speed_X, val_speed_Y = val_speed_X.type(torch.float32).to(device), val_
speed_Y.type(torch.float32).to(device)
            target = model(val_speed_X, adjacency)
            loss = mse(input=val_speed_Y, target=target)
            val_loss += loss.item()

    avg_train_loss = train_loss / len(speed_data_loader_train)
    avg_val_loss = val_loss / len(speed_data_loader_val)
    writer.add_scalar("loss_train", avg_train_loss, epoch)
    writer.add_scalar("loss_eval", avg_val_loss, epoch)
    print('epoch:', epoch, 'train Loss', avg_train_loss, 'val Loss:', avg_val_loss)

    if epoch > 0:
        #early stopping
        model_dict = model.state_dict()
```

```
            early_stopping(avg_val_loss, model_dict, model, epoch, save_dir)
            if early_stopping.early_stop:
                print("Early Stopping")
                break
        # 每 10 个 epoch 打印一次训练时间
        if epoch % 10 == 0:
            print("time for 10 epoches:", round(time.time() - temp_time, 2))
            temp_time = time.time()
    global_end_time = time.time() - global_start_time
    print("global end time:", global_end_time)

    Train_time_ALL = []
    Train_time_ALL.append(global_end_time)
    np.savetxt('result/lr_' + str(lr) + '_batch_size_' + str(batch_size) + '_Train_time_ALL.
    txt', Train_time_ALL)
    print("end")
```

在模型测试部分，首先需要利用 torch.load()函数将训练过程中保存的模型导入进来，然后利用 model.load_state_dict()函数将保存的参数字典加载到模型中，此步骤是将训练过程中训练好的参数直接赋予模型，使模型不需要再训练也能得到很好的测试结果。最后将测试结果反归一化至原始状态数据类型，对模型进行评价，并对预测结果进行画图。

(5) 模型训练过程及结果演示。

模型运行的数据集维度展示、模型打印、模型训练过程展示以及真实值和预测值对比图，分别如图 3-36～图 3-39 所示。

```
D:\software\anaconda\envs\pytorch\python.exe D:/deeplearning/神经网络文档/GCN_code/main.py
cuda:0
train speed
X.shape torch.Size([2021, 156, 10]) Y.shape torch.Size([2021, 156, 1])
val speed
X.shape torch.Size([224, 156, 10]) Y.shape torch.Size([224, 156, 1])
test speed
X.shape torch.Size([359, 156, 10]) Y.shape torch.Size([359, 156, 1])
```

图 3-36　数据集维度展示

```
Model(
  (GCN1): GraphConvolution(10 -> 10)
  (GCN2): GraphConvolution(10 -> 10)
  (linear1): Linear(in_features=1560, out_features=1024, bias=True)
  (linear2): Linear(in_features=1024, out_features=512, bias=True)
  (linear3): Linear(in_features=512, out_features=156, bias=True)
)
```

图 3-37　模型打印

```
epoch: 0 train Loss 0.007209608884295449 val Loss: 0.0014691189862787724
time for 10 epoches: 0.69
epoch: 1 train Loss 0.00546141951053869 val Loss: 0.0014812079025432467
Validation loss decreased (inf --> 0.00148121).  Saving model ...
epoch: 2 train Loss 0.00521545981609961 val Loss: 0.0015231113648042083
EarlyStopping counter: 1 out of 100 0.0014812079025432467
epoch: 3 train Loss 0.005346695625121356 val Loss: 0.00148610002361238
EarlyStopping counter: 2 out of 100 0.0014812079025432467
epoch: 4 train Loss 0.005317953653502627 val Loss: 0.001451124669983983
Validation loss decreased (0.00148121 --> 0.00145112).  Saving model ...
epoch: 5 train Loss 0.0052411240449146135 val Loss: 0.001460982020944357
EarlyStopping counter: 1 out of 100 0.001451124669983983
epoch: 6 train Loss 0.005050947449490195 val Loss: 0.0015315093332901597
EarlyStopping counter: 2 out of 100 0.001451124669983983
epoch: 7 train Loss 0.00502124875310983 val Loss: 0.0015283976681530476
EarlyStopping counter: 3 out of 100 0.001451124669983983
epoch: 8 train Loss 0.0050579675826156745 val Loss: 0.001446476555429399
Validation loss decreased (0.00145112 --> 0.00144648).  Saving model ...
epoch: 9 train Loss 0.004993057576939464 val Loss: 0.00152126036118716
EarlyStopping counter: 1 out of 100 0.001446476555429399
epoch: 10 train Loss 0.005252915017990745 val Loss: 0.0014830994186922908
```

图 3-38　模型训练过程展示

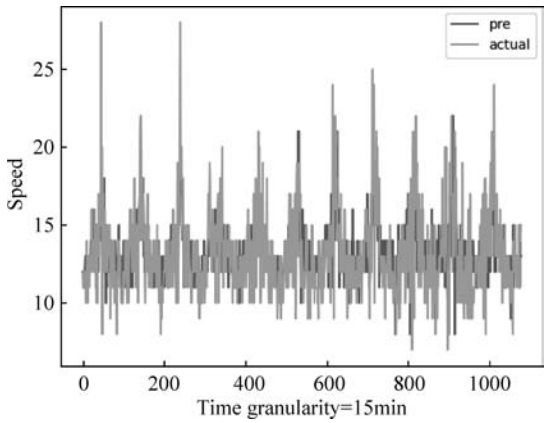

图 3-39　真实值和预测值对比图