第 5 章

分类与卷积神经网络

分类就是将某个事物判定为属于预先设定的有限个集合中的某一个的过程。在日常生活中经常分类，比如从远处观察判断某人的性别。分类是机器学习中应用最为广泛的任务。分类问题包括二分类问题和多分类问题。分类任务中样本的类别是预先设定的。分类是监督学习。

本章分别讨论决策函数分类模型、概率分类模型和神经网络分类模型中的常用模型。决策函数分类模型中，讨论决策树与随机森林模型。概率分类模型中，讨论朴素贝叶斯分类模型。神经网络分类模型中，讨论全连接层神经网络与卷积神经网络及其在分类中的应用。

本章基于 MindSpore 和 TensorFlow 2 深度学习框架，对误差反向传播学习算法、激活函数、损失函数和优化方法等多层神经网络的基础知识以及卷积层、池化层、批标准化层等卷积神经网络基本组成单元进行讨论。

5.1 分类算法基础

本节讨论一般性的分类任务以及分类算法的评价指标等基础知识。

5.1.1 分类任务

分类任务的目标是给未标记的测试样本进行标记。与聚类不同的是，分类任务的训练样本已经划分为若干个子集了，每个子集称为“类”，用类别标签来区分。与回归不同的是，分类任务的标签数量是有限的。

设样本集 $\boldsymbol{S}=\{\boldsymbol{s}_1,\boldsymbol{s}_2,\cdots,\boldsymbol{s}_m\}$ 包含 m 个样本，样本 $\boldsymbol{s}_i=(\boldsymbol{x}_i,\boldsymbol{y}_i)$ 包括一个实例 $\boldsymbol{x}_i$ 和一个标签 y_i，实例由 n 维特征向量表示，即 $\boldsymbol{x}_i=(x_i^{(1)},x_i^{(2)},\cdots,x_i^{(n)})$。分类任务可分为学习过程和判别（预测）过程，如图 5-1 所示。

在学习过程，分类任务将样本集中的知识提炼出来，形成模型。完成分类任务的模型有决策函数模型、概率模型和神经网络模型等。

决策函数分类模型建立了从实例特征向量到类别标签的映射 $Y=f(X)$，其中，X 是定义域，它是所有实例特征向量的集合；Y 是值域，它是所有类别标签的集合。

概率分类模型建立了条件概率分布函数 $\hat{P}(Y|X)$，它反映了从实例特征向量到类别标签的概率映射。

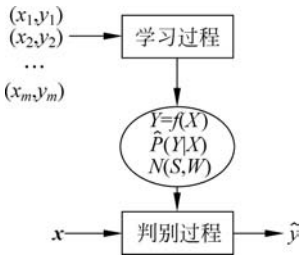


图 5-1　分类任务的模型

神经网络分类模型建立了能正确反映实例特征向量与类别标签关系的神经网络 $N(S,W)$。

记测试样本为 $\boldsymbol{x}=(x^{(1)},x^{(2)},\cdots,x^{(n)})$。在判别过程中，决策函数分类模型依据决策函数 $Y=f(X)$ 给予测试样本 $\boldsymbol{x}$ 一个类标签 $\hat{y}$；概率分类模型依据条件概率 $\hat{P}(Y|X)$ 计算在给定 $\boldsymbol{x}$ 时取每一个类标签 $\hat{y}$ 的条件概率值，取最大值对应的 $\hat{y}$ 作为输出；神经网络分类模型将 $\boldsymbol{x}$ 馈入已经训练好的网络 $N(S,W)$，从输出得到类标签 $\hat{y}$。

如果值域只有两个值，则该模型是二分类的；如果多于两个值，则该模型是多分类的。

5.1.2　分类模型的评价指标

本节主要讨论二分类模型的评价指标，它们中的大部分可以容易地扩展到多分类任务。

1. 准确率

准确率（Accuracy）是指在分类中，用模型对测试集进行分类，分类正确的样本数占总数的比例：

$$\text{accuracy}=\frac{n_{\text{correct}}}{n_{\text{total}}} \tag{5-1}$$

sklearny 扩展库中提供了一个专门对模型进行评估的包 metrics，该包可以满足一般的模型评估需求。包中提供了准确率计算函数，函数原型为：sklearn. metrics. accuracy_score(y_true,y_pred,normalize=True,sample_weight=None)。其中，normalize 默认值为 True，返回正确分类的比例；如果设为 False，则返回正确分类的样本数。

2. 混淆矩阵

混淆矩阵（Confusion Matrix）是对分类的结果进行详细描述的矩阵，对于二分类则是一个 2×2 的矩阵，对于 n 分类则是 $n\times n$ 的矩阵。二分类的混淆矩阵，如表 5-1 所示，

第一行是真实类别为“正(Positive)”的样本数,第二行则是真实类别为“负(Negative)”的样本数,第一列是预测值为“正”的样本数,第二列则是预测值为“负”的样本数。

表 5-1 二分类的混淆矩阵

	预测为“正”的样本数	预测为“负”的样本数
标签为“正”的样本数	True Positive(TP)	False Negative(FN)
标签为“负”的样本数	False Positive(FP)	True Negative(TN)

表 5-1 中 TP 表示真正,即被算法分类正确的正样本;FN 表示假正,即被算法分类错误的正样本;FP 表示假负,即被算法分类错误的负样本;TN 表示真负,即被算法分类正确的负样本。

sklearn. metrics 中计算混淆矩阵的函数为 confusion_matrix。

可以由混淆矩阵计算出准确率 accuracy:

$$\text{accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{FN} + \text{TN}} \tag{5-2}$$

3. 平均准确率

准确率指标虽然简单、易懂,但它没有对不同类别进行区分。不同类别下分类错误的代价可能不同,例如在重大病患诊断中,漏诊(False Negative)可能要比误诊(False Positive)给治疗带来更为严重的后果,此时准确率就不足以反映预测的效果。如果样本类别分布不平衡(即有的类别下的样本过多,有的类别下的样本个数过少),准确率也难以反映真实预测效果。如在类别样本数量差别极端不平衡时,只需要将全部实例预测为多的那类,就可以取得很高的准确率。

平均准确率(Average Per-class Accuracy)的全称为:按类平均准确率,即计算每个类别的准确率,然后再计算它们的平均值。

平均准确率也可以通过混淆矩阵来计算:

$$\text{average\_accuracy} = \frac{\left(\frac{\text{TP}}{\text{TP} + \text{FN}} + \frac{\text{TN}}{\text{FP} + \text{TN}}\right)}{2} \tag{5-3}$$

在样本类别分布不平衡的评价问题上,有一个称为 AUC(Area Under the Curve)的评价指标得到了广泛应用,有需要的读者可参考原版书。

4. 精确率-召回率

精确率-召回率(Precision-Recall)包含两个评价指标,一般同时使用。精确率是指分类器分类正确的正(负)样本的个数占该分类器所有分类为正(负)样本个数的比例。召回率是指分类器分类正确的正(负)样本个数占所有的正(负)样本个数的比例。

精确率是从预测的角度来看的,即预测为正(负)的样本中,预测成功的比例。召回率是从样本的角度来看的,即实际标签为正(负)的样本中,被成功预测的比例。准确率也是从样本的角度来看的,即所有样本中,正确预测的比例。与召回率不同,准确率是不分类别的。

在混淆矩阵中，预测为正的样本的精确率为：

$$\text{precision}_{\text{Positive}} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{5-4}$$

预测为负的样本的精确率为：

$$\text{precision}_{\text{Negative}} = \frac{\text{TN}}{\text{TN} + \text{FN}} \tag{5-5}$$

真实正样本的召回率为：

$$\text{recall}_{\text{Positive}} = \frac{\text{TP}}{\text{TP} + \text{FN}} = \text{TPR} \tag{5-6}$$

真实负样本的召回率为：

$$\text{recall}_{\text{Negative}} = \frac{\text{TN}}{\text{TN} + \text{FP}} = \text{TNR} \tag{5-7}$$

5. F_1-score

精确率与召回率实际上是一对矛盾的值，有时候单独采用一个值难以全面衡量算法，F_1-score 试图将两者结合起来作为一个指标来衡量算法。F_1-score 为精确率与召回率的调和平均值，即：

$$F_1 = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \tag{5-8}$$

还可以给精确率和召回率加权重系数来区别两者的重要性，将 F_1-score 扩展为 F_β-score：

$$F_\beta = (1 + \beta^2) \frac{\text{precision} \times \text{recall}}{(\beta^2 \times \text{precision}) + \text{recall}} \tag{5-9}$$

其中，β 表示召回率比精确率的重要程度，除了 1 之外，常取 2 或 0.5，分别表示召回率的重要程度是精确率的 2 倍或一半。

sklearn.metrics 包中提供了计算 F_1-score 和 F_β-score 的函数，可在需要时调用。

5.2　决策树与随机森林


视频讲解

决策树（Decision Tree）是常用的分类方法，以它为基础的随机森林（Random Forests，RF）在大多数应用情景中都表现较好。

5.2.1　决策树基本思想

决策树的基本思想很容易理解，在生活中人们经常应用决策树的思想来做决定，某相亲决策过程如图 5-2 所示。

分类的建模过程与上面做决定的过程相反，由于事先不知道人们的决策思路，需要通过人们已经做出的大量决定来“揣摩”出其决策思路，也就是通过大量数据来归纳道理，如通过如表 5-2 所示的相亲数据来分析某人的相亲决策条件。

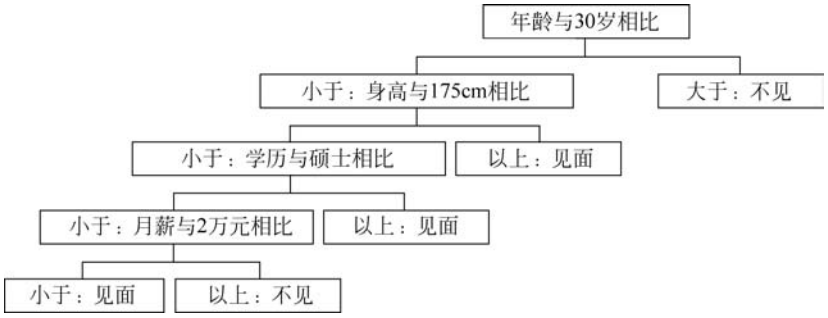


图 5-2　某相亲决策过程

表 5-2　某人相亲数据

编　号	年龄/岁	身高/cm	学　历	月薪/元	是否相亲
1	35	176	本科	20000	否
2	28	178	硕士	10000	是
3	26	172	本科	25000	否
4	29	173	博士	20000	是
5	28	174	本科	15000	是

当影响决策的因素较少时，人们可以直观地从表 5-2 所示的数据(即训练样本)中推测出如图 5-2 所示的相亲决策思路，从而了解此人的想法，更有目标地给他推荐相亲对象。

当样本和特征数量较多时，且训练样本可能出现冲突，人就难以胜任建立模型的任务。此时，一般要按一定算法由计算机来自动完成归纳，从而建立起可用来预测的模型，并用该模型来预测测试样本，从而筛选相亲对象。

决策树模型是一种对测试样本进行分类的树形结构，该结构由节点(Node)和有向边(Directed Edge)组成，节点分为内部节点(Internal Node)和叶节点(Leaf Node)两类。内部节点表示对测试样本的一个特征进行测试，内部节点下面的分支表示该特征测试的输出。如果只对特征的一个具体值进行测试，那么将只有正(大于或等于)或负(小于)2 个输出，可以生成二叉树。本书中，二叉树的左子树默认表示测试为负的输出，右子树默认表示测试为正的输出。如果对特征的多个具体值进行测试，那么将产生多个输出，可以生成多叉树。叶节点表示样本的一个分类，如果样本只有两个分类类别，那么该模型是二分类模型，否则是多分类模型。

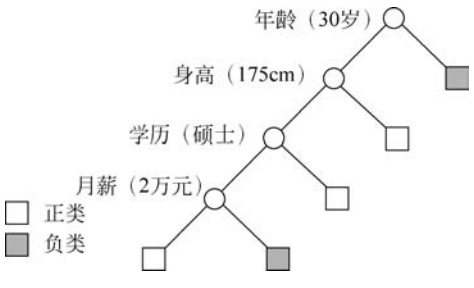


图 5-3　决策树示例 1

用圆点表示内部节点，用方块表示叶节点，可将图 5-2 所示的决策过程表示为决策树模型，如图 5-3 所示。在该决策树模型中，每个内部节点的输出只有两个分支，因此它是二叉树模型，同时，叶节点只有正、负两类，分别表示相亲和不相亲两种情况，因此它是二分类模型。图中分别用空心和实心的方块表

示相亲和不相亲两类结果。

图 5-3 中，最高的内部节点（根节点）表示对年龄特征是否大于 30 岁进行测试，左子树表示年龄小于 30 岁的输出，右子树表示年龄大于或等于 30 岁的输出。值得注意的是，一个特征可以在树的多个不同分支出现，如果在身高超过 175cm 后，还要考察月薪是否超过 8000 元条件时，则决策过程可以表示为如图 5-4 所示的模型。

对于表 5-2 所示的相亲数据，还可以归纳成图 5-5 所示的二叉决策树。

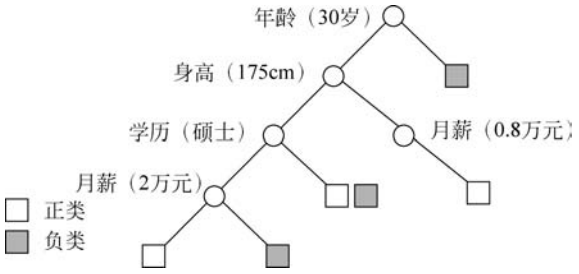


图 5-4　决策树示例 2

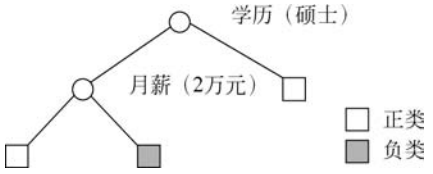


图 5-5　决策树示例 3

就表 5-2 中的训练数据而言，图 5-3 和图 5-5 所示的二叉决策树能起到完全相同的区分效果。但是，图 5-5 所示的二叉决策树只用了两个特征及相应的决策值就达到了相同的效果，在进行预测的时候，显然要简单、高效得多。该例子说明，在生成决策树时，选择合适的特征及其决策值是非常重要的。

使用决策树进行决策的过程是从根节点开始，依次测试样本相应的特征，并按照其值选择输出分支，直到到达叶子节点，然后将叶子节点存放的类别作为决策结果。如对年龄为 27 岁、身高为 176cm、学历为本科、月薪为 25000 元的对象，依据图 5-3 所示的模型，先测试根节点年龄特征，小于 30 岁，沿左子树继续测试，身高大于 175cm，走右子树，到达叶节点，得出相亲的决策结论。

5.2.2　决策树建立与应用

决策树算法一般采用递归方式建树。

建立二叉决策树的流程如图 5-6 所示。

流程中，找分裂点是算法的关键，选择哪一个特征及其决策值来划分训练集对生成的树结构影响很大。对决策树的研究基本上集中于该问题，该问题习惯上称为样本集分裂，依其解决方法可将决策树算法分为 ID3、C4.5、CART 等算法。这些算法对样本集进行分裂的方法都是依据某个指标对所有潜在分裂点进行试分裂，找出最符合指标要求的那个点作为实际分裂点。依据的指标分为信息增益（Information Gain）、增益率（Gain Ratio）和基尼指数（Gini Index）等，它们都以信息论为理论基础，它们的目标都是建立如图 5-5 所示的层次尽可能少的决策树。决策树的层次少，说明对新样本的测试次数就少，所做的测试越有效。有关信息增益、增益率和基尼指数等测试指标，感兴趣的读者可参考原版书。

与建立二叉树时以某特征的某个值作为分裂点不同，建立多叉决策树的分裂点是某

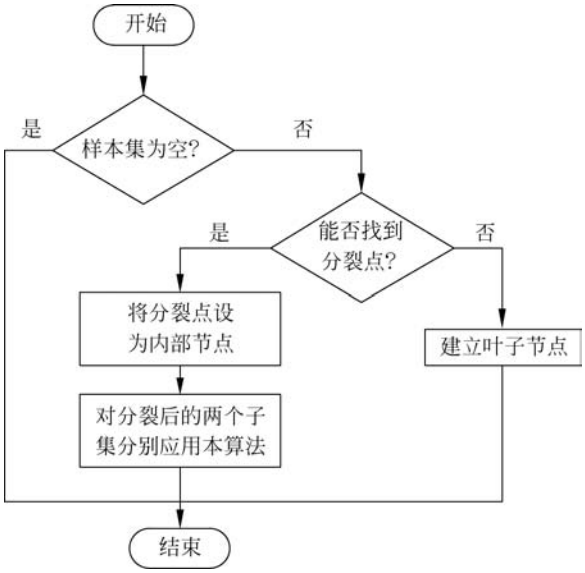


图 5-6　建立二叉决策树流程

一个特征。在试分裂时,它对样本集按某特征的每个取值都分裂一个子集,然后计算指标值。最后选择最符合指标要求的特征作为分裂点。

sklearn 的决策树类在 tree 模块中,DecisionTreeClassifier 类和方法原型见代码 5-1。

代码 5-1　sklearn 中的决策树算法

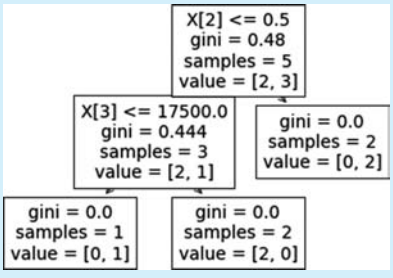
```
1. class sklearn.tree.DecisionTreeClassifier(criterion = 'gini', splitter = 'best', max_
   depth = None, min_samples_split = 2, min_samples_leaf = 1, min_weight_fraction_leaf =
   0.0, max_features = None, random_state = None, max_leaf_nodes = None, min_impurity_
   decrease = 0.0, min_impurity_split = None, class_weight = None, presort = False)
2.
3. apply(self, X[, check_input])
4. decision_path(self, X[, check_input])
5. fit(self, X, y[, sample_weight, …])
6. get_depth(self)
7. get_n_leaves(self)
8. get_params(self[, deep])
9. predict(self, X[, check_input])
10. predict_log_proba(self, X)
11. predict_proba(self, X[, check_input])
12. score(self, X, y[, sample_weight])
13. set_params(self, \*\*params)
```

其中,criterion 参数指定是采用基尼指数或信息增益作为样本集分裂的指标,fit 方法用来建树。predict_broba 用来产生概率值的预测输出,它是计算叶节点中不同种类样本的比例值作为输出。predict_log_proba 方法用来产生对数概率值的预测输出。

用它来示例表 5-2 所示的相亲决策模型见代码 5-2。

代码 5-2　决策树示例(决策树示例.ipynb)

```
1. from sklearn import tree
2. # 训练样本集
3. blind_date_X = [ [35, 176, 0, 20000],
4.                  [28, 178, 1, 10000],
5.                  [26, 172, 0, 25000],
6.                  [29, 173, 2, 20000],
7.                  [28, 174, 0, 15000] ]
8. blind_date_y = [ 0, 1, 0, 1, 1 ]
9. # 测试样本集
10. test_sample = [ [24, 178, 2, 17000],
11.                 [27, 176, 0, 25000],
12.                 [27, 176, 0, 10000] ]
13. clf = tree.DecisionTreeClassifier()              # 实例化
14. clf = clf.fit(blind_date_X, blind_date_y)        # 建树
15. clf.predict(test_sample)                         # 预测
16. >>> array([1, 0, 1])
17. tree.plot_tree(clf)                              # 画出树结构
18. >>> [Text(200.88000000000002, 181.2, 'X[2] <= 0.5\ngini = 0.48\nsamples = 5\nvalue =
    [2, 3]'),
19. Text(133.92000000000002, 108.72, 'X[3] <= 17500.0\ngini = 0.444\nsamples = 3\nvalue =
    [2, 1]'),
20. Text(66.96000000000001, 36.23999999999998, 'gini = 0.0\nsamples = 1\nvalue = [0, 1]'),
21. Text(200.88000000000002, 36.23999999999998, 'gini = 0.0\nsamples = 2\nvalue = [2, 0]'),
22. Text(267.84000000000003, 108.72, 'gini = 0.0\nsamples = 2\nvalue = [0, 2]')]
```




```
23.
24. print(clf.feature_importances_)                  # 给出特征的重要度
25. >>> [0.         0.         0.44444444 0.55555556]
```

第 17 行画出树结构,可见与图 5-5 所示决策树结构一样。

第 24 行打印出 feature_importances_属性值,它给出了特征的重要度。从第 25 行输出可知年龄和身高特征并不重要,因此,在树结构中,并没有用到这两个特征。这说明决策树算法能够立足现有的训练集发现最起作用的特征。这个功能也可以用来降维,将这两个重要度为 0 的特征去掉,并不会影响模型的建立和预测。

决策树算法容易出现过拟合现象。如图 5-7 所示的二维平面上的样本集中,圆点和十字点分别表示不同的两类样本。在左下角出现了一个与周围圆点不同的十字点(图中圆圈所示),一般认为该点为噪声点。如果不加处理,生成的决策树将会将该点单独延伸

出一个分枝来,从而产生过拟合现象。对此类过拟合的一般处理方法是剪枝(Pruning),它是将延伸出来的分枝剪掉,避免受到噪声的影响。有关过拟合和剪枝进一步的讨论,可参考原版书。

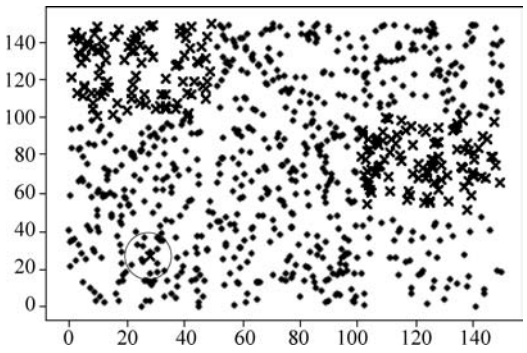


图 5-7 混入噪声的示例样本(见彩插)

决策树模型还可以用于回归问题。树模型解决回归问题的基本思想是将样本空间切分为多个子空间,在每个子空间中单独建立回归模型,因此,基于树的回归模型属于局部回归模型。与局部加权线性回归模型和 K 近邻法不同的是,基于树的回归模型会先生成固定的模型,不需要在每次预测时都计算每个训练样本的权值,因此效率相对较高。

sklearn 中的树回归算法在 tree 模块中的 DecisionTreeRegressor 类中实现。

5.2.3 随机森林

随机森林算法的基本思想是从样本集中有放回地重复随机抽样生成新的样本集合,然后无放回地随机选择若干特征生成一棵决策树,若干棵决策树组成随机森林,在预测分类时,将测试样本交由每个决策树判断,并根据每棵树的结果投票决定最终分类。

随机森林算法具有准确率高、能够处理高维数据和大数据集、能够评估各特征的重要性等优势,在工程实践和各类机器学习竞赛中被广泛地应用。

sklearn 中的随机森林分类算法类在 ensemble 模块中,类和方法原型见代码 5-3。

代码 5-3 sklearn 中的随机森林算法

```
1. class sklearn.ensemble.RandomForestClassifier(n_estimators = 'warn', criterion = 'gini',
   max_depth = None, min_samples_split = 2, min_samples_leaf = 1, min_weight_fraction_leaf =
   0.0, max_features = 'auto', max_leaf_nodes = None, min_impurity_decrease = 0.0, min_
   impurity_split = None, bootstrap = True, oob_score = False, n_jobs = None, random_state =
   None, verbose = 0, warm_start = False, class_weight = None)
2.
3. apply(self, X)
4. decision_path(self, X)
5. fit(self, X, y[, sample_weight])
6. get_params(self[, deep])
7. predict(self, X)
8. predict_log_proba(self, X)
```

```
 9. predict_proba(self, X)
10. score(self, X, y[, sample_weight])
11. set_params(self, \*\*params)
```

其中，n_estimators 是森林中树的棵数，max_features 是用来分裂时的最大特征数。

随机森林同样可用于回归任务，相应的类为 sklearn. ensemble. RandomForestRegressor。

像随机森林这样由多个分类器来集体决策的方法称为集成学习方法。集成学习(Ensemble Learning)是一种有效的机器学习方法，也是各类竞赛中的常用工具，在工业界得到了广泛的应用。目前，集成学习有三种主要方法，分别为装袋方法、提升方法和投票方法，有关它们的详细讨论，可参考原版书。

5.3　朴素贝叶斯分类

朴素贝叶斯(Naïve Bayes)分类是基于贝叶斯定理与特征条件独立假定的分类方法。

贝叶斯公式可由条件概率的定义直接得到。设试验 E 的样本空间为 S，A 为 E 的事件，$B_1,B_2,\cdots,B_n$ 为 S 的一个划分，且 $P(A)>0,P(B_i)>0(i=1,2,\cdots,n)$，则贝叶斯公式为：

$$P(B_i \mid A)=\frac{P(B_iA)}{P(A)}=\frac{P(A \mid B_i)P(B_i)}{\sum_{j=1}^{n}P(A \mid B_j)P(B_j)},\quad i=1,2,\cdots,n \tag{5-10}$$

其中，$P(B_i)$称为先验概率，即分类 B_i 发生的概率，它和条件概率 $P(A|B_i)$可从样本集中估计得到。通过贝叶斯公式就可以找到使后验概率 $P(B_i|A)$最大的 B_i。即 A 事件发生时，最有可能的分类 B_i。

在机器学习领域，A 可以看成一个样本，而 $B_1,B_2,\cdots,B_n$ 可以看成样本的所有可能的分类，或者是样本的所有可能的标签。贝叶斯分类，就是通过贝叶斯公式计算概率，将样本 A 分到可能性最大的类中，或者说是给样本 A 分一个可能性最大的标签。

设样本集为 $\boldsymbol{S}=\{\boldsymbol{s}_1,\boldsymbol{s}_2,\cdots,\boldsymbol{s}_m\}$，每个样本 $\boldsymbol{s}_i=(\boldsymbol{x}_i,y_i)$包括一个实例 $\boldsymbol{x}_i$ 和一个标签 y_i。标签 y_i 有 k 种取值$\{y_i^{(1)},y_i^{(2)},\cdots,y_i^{(k)}\}$。

朴素贝叶斯法首先基于特征条件独立假定，从样本集中学习到先验概率和条件概率，然后基于它们，对给定的测试样本 $\boldsymbol{x}$，利用贝叶斯公式求出使后验概率最大的预测值 y。y 可看作 $\boldsymbol{x}$ 所属分类的编号。

特征条件独立假定，是指假定样本的各个特征是相互独立的，互不关联。这个假定显然是不符合实际的，但它可以在大数据量、大特征量的情况下极大简化计算，使得贝叶斯算法实际可行。从实际应用情况来看，朴素贝叶斯分类也取得了不错的效果。

有关朴素贝叶斯法原理的深入讨论可参考原版书。

在应用朴素贝叶斯法进行分类时，根据条件概率 $P(A|B_i)$的不同假定分布，可以分为不同的分类器。

1. 多项式朴素贝叶斯分类器

多项式朴素贝叶斯(Multinomial Naïve Bayes)分类器假设条件概率 $P(A|B_i)$ 服从多项式分布。多次抛硬币试验中,出现指定次数正面(或反面)的概率是二项分布。将二项分布中的两种状态推广到多种状态,就得到了多项式分布。

多项式分布适用于离散取值的分类场合。

在 sklearn. naive_bayes 中的 MultinomialNB 实现了多项式分类器,其原型见代码 5-4。

代码 5-4　sklearn 中的多项式朴素贝叶斯分类器

```
1. class sklearn.naive_bayes.MultinomialNB( * , alpha = 1.0, fit_prior = True, class_prior =
   None)
2. fit(X, y, sample_weight = None)
3. predict(X)
4. predict_proba(X)
```

其中,alpha 称为平滑值,它用来避免在估计条件概率时出现值为 0 的情况,它的取值大于 0。当 alpha 等于 1 时,称为 Laplace(拉普拉斯)平滑。

当假定特征取值符合 0-1 分布时,多项式分类器退化为伯努利朴素贝叶斯(Bernoulli Naïve Bayes)分类器。即伯努利朴素分类器中,特征只能取两个值(条件概率 $P(A|B_i)$ 服从二项分布),它在某些场合下比多项式分类器效果要好一些。使用伯努利分类器之前,需要先将非二值的特征转化为二值的特征。

sklearn. naive_bayes 中的 BernoulliNB 实现了伯努利朴素贝叶斯分类器。

2. 高斯朴素贝叶斯分类器

当特征值是连续变量的时候,可采用高斯朴素贝叶斯(Gaussian Naïve Bayes)分类器。高斯朴素贝叶斯分类器假设条件概率 $P(A|B_i)$ 服从参数未知的高斯分布。

在 sklearn. naive_bayes 中的 GaussianNB 实现了高斯分类器。

用朴素贝叶斯分类器来对表 5-2 所示的相亲数据进行建模并预测测试样本的示例见代码 5-5。

代码 5-5　朴素贝叶斯分类器示例(贝叶斯分类器示例. ipynb)

```
1. # 训练样本集
2. blind_date_X = [ [35, 176, 0, 20000],
3.                  [28, 178, 1, 10000],
4.                  [26, 172, 0, 25000],
5.                  [29, 173, 2, 20000],
6.                  [28, 174, 0, 15000] ]
7. blind_date_y = [ 0, 1, 0, 1, 1 ]
8. # 测试样本集
9. test_sample = [ [24, 178, 2, 17000],
10.                [27, 176, 0, 25000],
11.                [27, 176, 0, 10000] ]
```

```
12.
13. # 多项式朴素贝叶斯分类器
14. from sklearn.naive_bayes import MultinomialNB
15. clf = MultinomialNB()
16. clf.fit(blind_date_X, blind_date_y)
17. print(clf.predict(test_sample))
18. >>> [1 0 1]
19.
20. # 高斯朴素贝叶斯分类器
21. from sklearn.naive_bayes import GaussianNB
22. clf = GaussianNB()
23. clf.fit(blind_date_X, blind_date_y)
24. print(clf.predict(test_sample))
25. >>> [1 0 1]
26. print(clf.class_prior_)                    # 标签的先验概率
27. >>> [0.4 0.6]
28. print(clf.class_count_)                   # 每个标签的样本数量
29. >>> [2. 3.]
30. print(clf.theta_)                         # 高斯模型的期望值
31. >>> [[3.05000000e+01 1.74000000e+02 0.00000000e+00 2.25000000e+04]
32.      [2.83333333e+01 1.75000000e+02 1.00000000e+00 1.50000000e+04]]
33. print(clf.sigma_)                         # 高斯模型的方差
34. >>> [[2.02760000e+01 4.02600000e+00 2.60000000e-02 6.25000003e+06]
35.      [2.48222222e-01 4.69266667e+00 6.92666667e-01 1.66666667e+07]]
```

从示例的输出来验证高斯朴素贝叶斯分类器。

第 26 行输出的是每类标签的先验概率。

第 30 行和第 33 行输出的是高斯分布的均值和方差。因为训练样本有 4 个特征和 2 个标签，每个特征与每个标签生成一个高斯分布，因此共有 8 个高斯分布。验算第一个高斯分布的均值(第 31 行的第一个值 3.05000000e+01)，它是标签值为 0 时的年龄特征两个取值 35 和 26 的均值。

而对于多项式朴素贝叶斯分类器，它对每一个特征生成一个线性分类器(线性分类器可看作将式(4-2)所示的线性回归用于分类，它用“直线”将空间划分两个部分，不同部分的样本点分别属于不同的两个类)，读者可以修改代码查看多项式朴素贝叶斯分类器的 coef_属性，它代表 4 个线性函数的斜率。

朴素贝叶斯法实现简单，学习与预测的效率都很高，甚至在某些特征相关性较高的情况下都有不错的表现，是一种常用的方法。

5.4　神经网络与分类任务

本节讨论多层神经网络的一些基础问题，并示例全连接层神经网络在分类任务中的应用。


视频讲解

5.4.1 误差反向传播学习算法

用神经网络来完成机器学习任务,先要设计好网络结构 S,然后用训练样本去学习网络中的连接系数和阈值系数(即网络参数 W),最后对测试样本进行预测。

在第 4 章讨论了多层神经网络在回归问题中的初步应用,在本节讨论多层神经网络的参数学习问题。

在研究早期,没有适合多层神经网络的有效的参数学习方法是长期困扰该领域研究者的关键问题,以至于让人们对人工神经网络的前途产生了怀疑,导致该领域的研究进入了低谷期。直到 1986 年,以 Rumelhart 和 McCelland 为首的小组发表了误差反向传播(Error Back Propagation,BP)算法[10],该问题才得以解决,多层神经网络从此得到快速发展。

采用 BP 算法来学习的、无反馈的、同层节点无连接的、多层结构的前馈神经网络称为 BP 神经网络。BP 学习算法属于监督学习算法。BP 神经网络可用于解决分类问题和回归问题,是应用最多的神经网络。

本节先用一个简单的示例来讨论 BP 算法,然后再推广到一般情况。

1. 误差反向传播学习示例

逻辑代数中的异或运算是非线性的,它不能由单个神经元来模拟。下面用模拟异或运算的神经网络为例来说明 BP 学习过程。

设模拟异或运算的训练样本集如表 5-3 所示。表中,$x^{(1)}$ 和 $x^{(2)}$ 是异或运算的两个输入。$l^{(1)}$ 表示异或运算的真值输出,即当异或运算为真时值为 1,否则为 0。$l^{(2)}$ 是异或运算的假值输出,即当异或运算为真时值为 0,否则为 1。

表 5-3 模拟异或运算的训练样本集

	$x^{(1)}$	$x^{(2)}$	$l^{(1)}$	$l^{(2)}$
1	0	0	0	1
2	0	1	1	0
3	1	0	1	0
4	1	1	0	1

用如图 5-8 所示网络结构的三层全连接神经网络来模拟异或运算。接下来用表 5-3 所示的训练样本来学习该神经网络的参数。

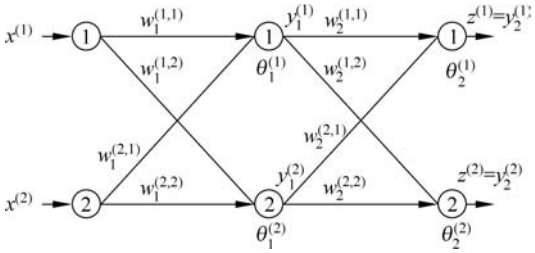


图 5-8 模拟异或运算的三层感知机

图 5-8 所示的神经网络中，最左边为输入层，有两个节点，从上至下编号为节点 1 和节点 2。输入层的输入向量为 $\boldsymbol{x}=(x^{(1)},x^{(2)})$，用带括号的上标表示输入节点序号。

为了统一标识，将输出层也看作隐层，即三层神经网络里有两个隐层。第 1 隐层共有 2 个节点，也从上至下编号，分别用 $y_1^{(1)}$ 和 $y_1^{(2)}$ 表示它们的输出，即用下标来表示隐层序号，用带括号的上标来表示层内节点序号。第 2 隐层，即输出层，也有 2 个节点，它的输出分别用 $z^{(1)}=y_2^{(1)}$ 和 $z^{(2)}=y_2^{(2)}$ 表示。

从输入层第 1 节点到第 1 隐层的第 1 节点的连接系数记为 $w_1^{(1,1)}$，用下标表示到第 1 隐层节点的连接系数，上标括号内表示是从前一层的 1 号节点到本层的 1 号节点。

用 $\theta_1^{(1)}$ 表示第 1 隐层的第 1 节点的阈值系数。类似可得其他系数的表示方法如图 5-8 所示。

为方便起见，还可以用矩阵和向量来表示各参数。如从输入层到第 1 隐层的连接系数可以用一个 2×2 的矩阵 $\boldsymbol{W}_1$ 来表示：$\boldsymbol{W}_1=\begin{bmatrix} w_1^{(1,1)} & w_1^{(1,2)} \\ w_1^{(2,1)} & w_1^{(2,2)} \end{bmatrix}$，其中，行表示前一层的节点，列表示本层的节点，如第 1 行第 2 列的元素 $w_1^{(1,2)}$ 表示是从输入层的第 1 个节点到第 1 隐层的第 2 个节点的连接系数。

同样，第 1 隐层的阈值可表示为向量：$\boldsymbol{\theta}_1=[\theta_1^{(1)} \quad \theta_1^{(2)}]$。从第 1 隐层到第 2 隐层（输出层）的连接系数可表示为向量：$\boldsymbol{W}_2=\begin{bmatrix} w_2^{(1,1)} & w_2^{(1,2)} \\ w_2^{(2,1)} & w_2^{(2,2)} \end{bmatrix}$，第 2 隐层的阈值可表示为向量：$\boldsymbol{\theta}_2=[\theta_2^{(1)} \quad \theta_2^{(2)}]$。

为了方便求导，隐层和输出层的激励函数采用如图 2-7 中虚线所示的 Sigmoid 函数，它的定义如式(2-4)所示。Sigmoid 函数的导数为：

$$g'(z)=\frac{-1}{(1+\mathrm{e}^{-z})^2}\mathrm{e}^{-z}(-1)=\frac{1}{1+\mathrm{e}^{-z}}\cdot\frac{\mathrm{e}^{-z}}{1+\mathrm{e}^{-z}}$$
$$=g(z)(1-g(z)) \tag{5-11}$$

BP 学习算法可分为前向传播预测与反向传播学习两个过程。要学习的各参数值一般先作随机初始化。取训练样本输入网络，逐层前向计算输出，在输出层得到预测值，此为前向传播预测过程。根据预测值与实际值的误差再从输出层开始逐层反向调节各层的参数，此为反向传播学习过程。经过多样本的多次前向传播预测和反向传播学习，最终学习得到网络各参数的值。

1）前向传播预测过程

前向传播预测的过程是一个逐层计算的过程。设网络各参数初值为：$\boldsymbol{W}_1=\begin{bmatrix} 0.1 & 0.2 \\ 0.2 & 0.3 \end{bmatrix}$，$\boldsymbol{\theta}_1=[0.3 \quad 0.3]$，$\boldsymbol{W}_2=\begin{bmatrix} 0.4 & 0.5 \\ 0.4 & 0.5 \end{bmatrix}$，$\boldsymbol{\theta}_2=[0.6 \quad 0.6]$。

取第一个训练样本(0,0)，由式(2-2)和式(2-3)可得第 1 隐层的输出：

$$\begin{cases} y_1^{(1)} = g(w_1^{(1,1)}x^{(1)} + w_1^{(2,1)}x^{(2)} + \theta_1^{(1)}) \\ \quad = \dfrac{1}{1+\mathrm{e}^{-(w_1^{(1,1)}x^{(1)}+w_1^{(2,1)}x^{(2)}+\theta_1^{(1)})}} = \dfrac{1}{1+\mathrm{e}^{-0.3}} = 0.574 \\ y_1^{(2)} = g(w_1^{(1,2)}x^{(1)} + w_1^{(2,2)}x^{(2)} + \theta_1^{(2)}) \\ \quad = \dfrac{1}{1+\mathrm{e}^{-(w_1^{(1,2)}x^{(1)}+w_1^{(2,2)}x^{(2)}+\theta_1^{(2)})}} = 0.574 \end{cases} \tag{5-12}$$

同样计算第 2 隐层,也就是输出层的输出:

$$\begin{cases} z^{(1)} = y_2^{(1)} = g(w_2^{(1,1)}y_1^{(1)} + w_2^{(2,1)}y_1^{(2)} + \theta_2^{(1)}) \\ \quad = \dfrac{1}{1+\mathrm{e}^{-(w_2^{(1,1)}y_1^{(1)}+w_2^{(2,1)}y_1^{(2)}+\theta_2^{(1)})}} \\ \quad = \dfrac{1}{1+\mathrm{e}^{-(0.4\times 0.574+0.4\times 0.574+0.6)}} = 0.743 \\ z^{(2)} = y_2^{(2)} = g(w_2^{(1,2)}y_1^{(1)} + w_2^{(2,2)}y_1^{(2)} + \theta_2^{(2)}) \\ \quad = \dfrac{1}{1+\mathrm{e}^{-(w_2^{(1,2)}y_1^{(1)}+w_2^{(2,2)}y_1^{(2)}+\theta_2^{(2)})}} = 0.764 \end{cases} \tag{5-13}$$

2) 反向传播学习过程

用 $l^{(1)}$ 和 $l^{(2)}$ 表示标签值,采用各标签值的均方误差 MSE 作为总误差,并将总误差依次展开至输入层:

$$\begin{aligned} E &= \frac{1}{2}\sum_{i=1}^{2}(z^{(i)} - l^{(i)})^2 = \frac{1}{2}\sum_{i=1}^{2}(g(w_2^{(1,i)}y_1^{(1)} + w_2^{(2,i)}y_1^{(2)} + \theta_2^{(i)}) - l^{(i)})^2 \\ &= \frac{1}{2}\sum_{i=1}^{2}(g(w_2^{(1,i)}g(w_1^{(1,1)}x^{(1)} + w_1^{(2,1)}x^{(2)} + \theta_1^{(1)}) + \\ &\quad w_2^{(2,i)}g(w_1^{(1,2)}x^{(1)} + w_1^{(2,2)}x^{(2)} + \theta_1^{(2)}) + \theta_2^{(i)}) - l^{(i)})^2 \end{aligned} \tag{5-14}$$

可见,总误差 E 是各层参数变量的函数,因此学习的目的就是通过调整各参数变量的值,使 E 最小。可采用梯度下降法来迭代更新所有参数的值:先求出总误差对各参数变量的偏导数,即梯度,再沿梯度负方向前进一定步长。

第一个训练样本的标签值为(0,1),计算总误差为:

$$E = \frac{1}{2}\sum_{i=1}^{2}(z^{(i)} - l^{(i)})^2 = 0.304 \tag{5-15}$$

输出层节点的参数更新,以节点 1 的 $w_2^{(1,1)}$ 和 $\theta_2^{(1)}$ 为例详细讨论。先求偏导 $\dfrac{\partial E}{\partial w_2^{(1,1)}}$,根据链式求导法则和式(5-13)、式(5-15)可知:

$$\begin{aligned} \frac{\partial E}{\partial w_2^{(1,1)}} &= \frac{\partial E}{\partial y_2^{(1)}} \cdot \frac{\partial y_2^{(1)}}{\partial w_2^{(1,1)}} = \frac{\partial\left[\dfrac{1}{2}\displaystyle\sum_{i=1}^{2}(y_2^{(i)} - l^{(i)})^2\right]}{\partial y_2^{(1)}} \cdot \frac{\partial y_2^{(1)}}{\partial w_2^{(1,1)}} \\ &= (y_2^{(1)} - l^{(1)}) \cdot \frac{\partial y_2^{(1)}}{\partial w_2^{(1,1)}} \end{aligned} \tag{5-16}$$

其中，$y_2^{(1)}-l^{(1)}$ 是输出层节点 1 的误差，记为 E_2^1，即 $E_2^1=y_2^{(1)}-l^{(1)}=0.743$。因此 $\dfrac{\partial E}{\partial w_2^{(1,1)}}$ 可视为该节点的误差乘以该节点输出对待更新参数变量的偏导：

$$\frac{\partial E}{\partial w_2^{(1,1)}}=E_2^1\cdot\frac{\partial y_2^{(1)}}{\partial w_2^{(1,1)}} \tag{5-17}$$

其中，误差 E_2^1 用来求偏导并更新参数，称之为校对误差。

设梯度下降法中的步长 α 为 0.5，由式(4-11)可知 $w_2^{(1,1)}$ 更新为：

$$w_2^{(1,1)}\leftarrow w_2^{(1,1)}-\alpha E_2^1\cdot\frac{\partial y_2^{(1)}}{\partial w_2^{(1,1)}} \tag{5-18}$$

其中，偏导数 $\dfrac{\partial y_2^{(1)}}{\partial w_2^{(1,1)}}$ 的计算为：

$$\begin{aligned}\frac{\partial y_2^{(1)}}{\partial w_2^{(1,1)}}&=\frac{\partial g(w_2^{(1,1)}y_1^{(1)}+w_2^{(2,1)}y_1^{(2)}+\theta_2^{(1)})}{\partial w_2^{(1,1)}}\\&=\frac{\partial g(w_2^{(1,1)}y_1^{(1)}+w_2^{(2,1)}y_1^{(2)}+\theta_2^{(1)})}{\partial(w_2^{(1,1)}y_1^{(1)}+w_2^{(2,1)}y_1^{(2)}+\theta_2^{(1)})}\cdot\frac{\partial(w_2^{(1,1)}y_1^{(1)}+w_2^{(2,1)}y_1^{(2)}+\theta_2^{(1)})}{\partial w_2^{(1,1)}}\\&=g(w_2^{(1,1)}y_1^{(1)}+w_2^{(2,1)}y_1^{(2)}+\theta_2^{(1)})\cdot(1-g(w_2^{(1,1)}y_1^{(1)}+w_2^{(2,1)}y_1^{(2)}+\theta_2^{(1)}))\cdot y_1^{(1)}\\&=y_2^{(1)}\cdot(1-y_2^{(1)})\cdot y_1^{(1)}\end{aligned} \tag{5-19}$$

式(5-19)中，用到了 Sigmoid 函数的导数，见式(5-11)。

因此：

$$\begin{aligned}&w_2^{(1,1)}\leftarrow w_2^{(1,1)}-\alpha E_2^1\cdot\frac{\partial y_2^{(1)}}{\partial w_2^{(1,1)}}\\&=w_2^{(1,1)}-\alpha E_2^1\cdot y_2^{(1)}\cdot(1-y_2^{(1)})\cdot y_1^{(1)}\\&=0.4-0.5\times0.743\times0.743\times(1-0.743)\times0.574\\&=0.359\end{aligned} \tag{5-20}$$

$\dfrac{\partial E}{\partial w_2^{(1,1)}}$ 的求导路径如图 5-9 中粗实线所示。

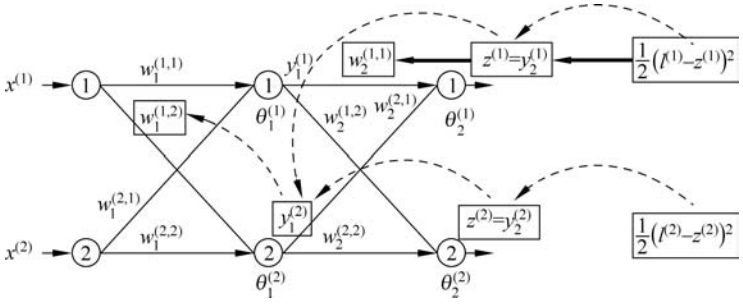


图 5-9　BP 算法中求导路径示例

同样,可得 $w_2^{(1,2)}$、$w_2^{(2,1)}$ 和 $w_2^{(2,2)}$ 的更新值分别为:0.512、0.359 和 0.512。

对于 $\theta_2^{(1)}$ 的更新,先求总误差对它的偏导数:

$$\begin{aligned}\frac{\partial E}{\partial \theta_2^{(1)}}&=\frac{\partial E}{\partial y_2^{(1)}}\cdot\frac{\partial y_2^{(1)}}{\partial \theta_2^{(1)}}=E_2^1\cdot\frac{\partial y_2^{(1)}}{\partial \theta_2^{(1)}}\\&=E_2^1\cdot y_2^{(1)}\cdot(1-y_2^{(1)})\cdot\frac{\partial(w_2^{(1,1)}y_1^{(1)}+w_2^{(2,1)}y_1^{(2)}+\theta_2^{(1)})}{\partial\theta_2^{(1)}}\\&=E_2^1\cdot y_2^{(1)}\cdot(1-y_2^{(1)})\end{aligned}\tag{5-21}$$

因此$\frac{\partial E}{\partial \theta_2^{(1)}}$可视为该节点的校对误差乘以该节点输出对待更新阈值变量的偏导。$\theta_2^{(1)}$ 的更新为:

$$\theta_2^{(1)}\leftarrow\theta_2^{(1)}-\alpha\frac{\partial E}{\partial\theta_2^{(1)}}=0.529\tag{5-22}$$

同样可得 $\theta_2^{(2)}$ 的更新为:0.621。

第 1 隐层的参数更新,以节点 2 的 $w_1^{(1,2)}$ 和 $\theta_1^{(2)}$ 为例详细讨论。对 $w_1^{(1,2)}$ 的求导有两条路径,如图 5-9 中粗虚线所示。

$$\begin{aligned}\frac{\partial E}{\partial w_1^{(1,2)}}&=\frac{\partial E}{\partial y_2^{(1)}}\cdot\frac{\partial y_2^{(1)}}{\partial w_1^{(1,2)}}+\frac{\partial E}{\partial y_2^{(2)}}\cdot\frac{\partial y_2^{(2)}}{\partial w_1^{(1,2)}}\\&=\frac{\partial E}{\partial y_2^{(1)}}\cdot\frac{\partial y_2^{(1)}}{\partial y_1^{(2)}}\cdot\frac{\partial y_1^{(2)}}{\partial w_1^{(1,2)}}+\frac{\partial E}{\partial y_2^{(2)}}\cdot\frac{\partial y_2^{(2)}}{\partial y_1^{(2)}}\cdot\frac{\partial y_1^{(2)}}{\partial w_1^{(1,2)}}\\&=\left(\frac{\partial E}{\partial y_2^{(1)}}\cdot\frac{\partial y_2^{(1)}}{\partial y_1^{(2)}}+\frac{\partial E}{\partial y_2^{(2)}}\cdot\frac{\partial y_2^{(2)}}{\partial y_1^{(2)}}\right)\cdot\frac{\partial y_1^{(2)}}{\partial w_1^{(1,2)}}\\&=\left(E_2^1\cdot\frac{\partial y_2^{(1)}}{\partial y_1^{(2)}}+E_2^2\cdot\frac{\partial y_2^{(2)}}{\partial y_1^{(2)}}\right)\cdot\frac{\partial y_1^{(2)}}{\partial w_1^{(1,2)}}\end{aligned}\tag{5-23}$$

其中,$E_2^2=(y_2^{(2)}-l^{(2)})$是输出层节点 2 的校对误差。可将 $E_2^1\cdot\frac{\partial y_2^{(1)}}{\partial y_1^{(2)}}+E_2^2\cdot\frac{\partial y_2^{(2)}}{\partial y_1^{(2)}}$视为校对误差 E_2^1 和 E_2^2 沿求导路径反向传播到第 1 隐层节点 2 的校对误差,如图 5-10 所示,将该校对误差记为 E_1^2:

$$E_1^2=E_2^1\cdot\frac{\partial y_2^{(1)}}{\partial y_1^{(2)}}+E_2^2\cdot\frac{\partial y_2^{(2)}}{\partial y_1^{(2)}}\tag{5-24}$$

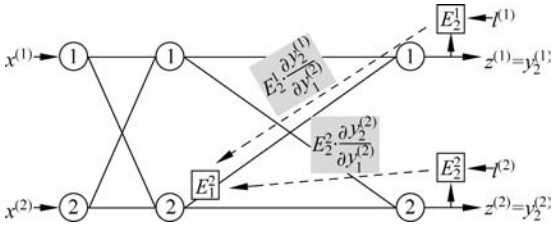


图 5-10 BP 算法中校对误差反向传播示例

式(5-23)可写为：

$$\frac{\partial E}{\partial w_1^{(1,2)}}=E_1^2\cdot\frac{\partial y_1^{(2)}}{\partial w_1^{(1,2)}} \tag{5-25}$$

因此，$\frac{\partial E}{\partial w_1^{(1,2)}}$可视为该节点的校对误差乘以该节点输出值对待更新参数变量的偏导数。式(5-25)与式(5-17)具有相同的形式。据此，反向传播学习过程中的求梯度可以看成是先计算出每个节点的反向传播校对误差，再乘以一个本地偏导数。

式(5-25)的两项因子计算如下：

$$\begin{cases}E_1^2=E_2^1\cdot\dfrac{\partial y_2^{(1)}}{\partial y_1^{(2)}}+E_2^2\cdot\dfrac{\partial y_2^{(2)}}{\partial y_1^{(2)}}=E_2^1\cdot y_2^{(1)}(1-y_2^{(1)})w_2^{(2,1)}+E_2^2\cdot y_2^{(2)}(1-y_2^{(2)})w_2^{(2,2)}\\ \dfrac{\partial y_1^{(2)}}{\partial w_1^{(1,2)}}=y_1^{(2)}(1-y_1^{(2)})\dfrac{\partial(w_1^{(1,2)}x^{(1)}+w_1^{(2,2)}x^{(2)}+\theta_1^{(2)})}{\partial w_1^{(1,2)}}=y_1^{(2)}(1-y_1^{(2)})x^{(1)}=0\end{cases} \tag{5-26}$$

因此，$\frac{\partial E}{\partial w_1^{(1,2)}}=0$。

$w_1^{(1,2)}$ 的更新为：

$$w_1^{(1,2)}\leftarrow w_1^{(1,2)}-\alpha\frac{\partial E}{\partial w_1^{(1,2)}}=w_1^{(1,2)}=0.2 \tag{5-27}$$

同样可计算第 1 隐层的其他三个连接系数也保持不变。

可知 $\theta_1^{(2)}$ 更新为：

$$\theta_1^{(2)}\leftarrow\theta_1^{(2)}-\alpha\frac{\partial E}{\partial\theta_1^{(2)}}=\theta_1^{(2)}-\alpha E_1^2\cdot y_1^{(2)}(1-y_1^{(2)})=0.296 \tag{5-28}$$

同样可得 $\theta_1^{(1)}$ 更新为：0.296。

以上给出了输入第一个训练样本后，网络的前向预测和反向学习过程。可将样本依次输入网络进行训练。一般要将样本多次输入网络进行多轮训练。

示例的实现见代码 5-6。共运行了 2000 轮(第 44 行)，每一轮对每一个样本进行一次前向传播预测和一次后向传播学习，并计算所有四个样本的平均总误差(第 64 行和第 86 行)。

代码 5-6　模拟异或运算三层感知机的误差反向传播学习(误差反向传播算法示例.ipynb)

```
1. import numpy as np
2.
3. # 样本示例
4. XX = np.array([[0.0,0.0],
5.                [0.0,1.0],
6.                [1.0,0.0],
7.                [1.0,1.0]])
8. # 样本标签
9. L = np.array([[0.0,1.0],
```

```
            [1.0,0.0],
            [1.0,0.0],
            [0.0,1.0]])

a = 0.5                          # 步长
W1 = np.array([[0.1, 0.2],       # 第 1 隐层的连接系数
               [0.2, 0.3]])
theta1 = np.array([0.3, 0.3])    # 第 1 隐层的阈值
W2 = np.array([[0.4, 0.5],       # 第 2 隐层的连接系数
               [0.4, 0.5]])
theta2 = np.array([0.6, 0.6])    # 第 2 隐层的阈值
Y1 = np.array([0,0, 0.0])        # 第 1 隐层的输出
Y2 = np.array([0,0, 0.0])        # 第 2 隐层的输出
E2 = np.array([0,0, 0.0])        # 第 2 隐层的误差
E1 = np.array([0,0, 0.0])        # 第 1 隐层的误差

def sigmoid(x):
    return 1/(1 + np.exp( - x))

# 计算第 1 隐层节点 1 的输出
def y_1_1(W1, theta1, X):
    return sigmoid(W1[0,0] * X[0] + W1[1,0] * X[1] + theta1[0])

# 计算第 1 隐层节点 2 的输出
def y_1_2(W1, theta1, X):
    return sigmoid(W1[0,1] * X[0] + W1[1,1] * X[1] + theta1[1])

# 计算第 2 隐层节点 1 的输出
def y_2_1(W2, theta2, Y1):
    return sigmoid(W2[0,0] * Y1[0] + W2[1,0] * Y1[1] + theta2[0])

# 计算第 2 隐层节点 2 的输出
def y_2_2(W2, theta2, Y1):
    return sigmoid(W2[0,1] * Y1[0] + W2[1,1] * Y1[1] + theta2[1])

for j in range(2000):            # 训练轮数
    print("\n\n 轮: ", j)
    E = 0.0
    for i in range(4):
        print("样本: ", i)
        print("实例: ", XX[i])
        print("标签", L[i])
        # 前向传播预测
        # 计算第 1 隐层的输出
        Y1[0] = y_1_1(W1, theta1, XX[i])
        Y1[1] = y_1_2(W1, theta1, XX[i])
        #print("第 1 隐层的输出:", Y1)

```

```
        # 计算第 2 隐层的输出
        Y2[0] = y_2_1(W2, theta2, Y1)
        Y2[1] = y_2_2(W2, theta2, Y1)
        print("第 2 隐层的输出:", Y2)

        # 后向传播误差
        # 计算第 2 隐层的校对误差
        E2[0] = Y2[0] - L[i][0]
        E2[1] = Y2[1] - L[i][1]
        E += 0.5 * (E2[0] * E2[0] + E2[1] * E2[1])
        #print("总误差", E)
        #print("第 2 隐层的校对误差", E2)

        # 计算第 1 隐层的校对误差
        E1[0] = E2[0] * Y2[0] * (1 - Y2[0]) * W2[0,0] + E2[1] * Y2[1] * (1 - Y2[1]) * W2[0,1]
        E1[1] = E2[0] * Y2[0] * (1 - Y2[0]) * W2[1,0] + E2[1] * Y2[1] * (1 - Y2[1]) * W2[1,1]
        #print("第 1 隐层的校对误差", E1)

        # 更新系数
        # 更新第 2 隐层的系数
        W2[0,0] = W2[0,0] - a * E2[0] * Y2[0] * (1 - Y2[0]) * Y1[0]
        W2[1,0] = W2[1,0] - a * E2[0] * Y2[0] * (1 - Y2[0]) * Y1[1]
        theta2[0] = theta2[0] - a * E2[0] * Y2[0] * (1 - Y2[0])
        W2[0,1] = W2[0,1] - a * E2[1] * Y2[1] * (1 - Y2[1]) * Y1[0]
        W2[1,1] = W2[1,1] - a * E2[1] * Y2[1] * (1 - Y2[1]) * Y1[1]
        theta2[1] = theta2[1] - a * E2[1] * Y2[1] * (1 - Y2[1])
        #print("第 2 隐层的连接系数", W2)
        #print("第 2 隐层的阈值系数", theta2)

        # 更新第 1 隐层的系数
        W1[0,0] = W1[0,0] - a * E1[0] * Y1[0] * (1 - Y1[0]) * XX[i][0]
        W1[1,0] = W1[1,0] - a * E1[0] * Y1[0] * (1 - Y1[0]) * XX[i][1]
        theta1[0] = theta1[0] - a * E1[0] * Y1[0] * (1 - Y1[0])
        W1[0,1] = W1[0,1] - a * E1[1] * Y1[1] * (1 - Y1[1]) * XX[i][0]
        W1[1,1] = W1[1,1] - a * E1[1] * Y1[1] * (1 - Y1[1]) * XX[i][1]
        theta1[1] = theta1[1] - a * E1[1] * Y1[1] * (1 - Y1[1])
        #print("第 1 隐层的连接系数", W1)
        #print("第 1 隐层的阈值系数", theta1)
    print("平均总误差" + str(E/4.0))
>>> …
轮: 1999
样本: 0
实例: [0. 0.]
标签 [0. 1.]
第 2 隐层的输出: [0.07158904 0.92822515 0.         ]
样本: 1
实例: [0. 1.]
标签 [1. 0.]
```

```
106. 第 2 隐层的输出: [0.9138734 0.08633152 0.         ]
107. 样本: 2
108. 实例: [1. 0.]
109. 标签 [1. 0.]
110. 第 2 隐层的输出: [0.91375259 0.08644981 0.         ]
111. 样本: 3
112. 实例: [1. 1.]
113. 标签 [0. 1.]
114. 第 2 隐层的输出: [0.11774177 0.88200493 0.         ]
115. 平均总误差 0.008480711186161102
```

第 29 行到第 43 行的代码分别是前向传播预测中式(5-12)和式(5-13)的实现。后面反向传播学习过程的代码也分别是按层计算校对误差并更新参数的计算式的实现。

经过 2000 轮训练,每轮平均总误差由 0.32 降为 0.008,能够准确地模拟异或运算,最后一轮的四个输出与相应标签值对比为:

```
[0.07158904,0.92822515]→[0.,1.],
[0.9138734,0.08633152]→[1.,0.],
[0.91375259,0.08644981]→[1.,0.],
[ 0.11774177,0.88200493]→[0.,1.].
```

可见,预测输出很接近实际标签值。关于这些输出与标签值的比较,将在 5.4.2 节有关损失函数的内容中进一步讨论。

下面用深度学习框架来模拟异或运算,因为截至本书完稿时 MindSpore 还不支持在 CPU 平台上运行 SGD 算子,该示例只用 TensorFlow 2 框架来实现,见代码 5-7。

代码 5-7　深度学习框架模拟异或运算(TensorFlow 2 模拟异或运算示例.ipynb)

```
1. import tensorflow as tf
2. import numpy as np
3.
4. # 样本实例
5. XX = np.array([[0.0,0.0],
6.              [0.0,1.0],
7.              [1.0,0.0],
8.              [1.0,1.0]])
9. # 样本标签
10. L = np.array([[0.0,1.0],
11.              [1.0,0.0],
12.              [1.0,0.0],
13.              [0.0,1.0]])
14.
15. tf_model = tf.keras.Sequential([
16.       tf.keras.layers.Dense(4, activation = 'sigmoid', input_shape = (2,), kernel_
    initializer = 'random_uniform', bias_initializer = 'zeros'),
17.      tf.keras.layers.Dense(2, activation = 'sigmoid', kernel_initializer = 'random_
    uniform', bias_initializer = 'zeros')
```

```
18. ])
19.
20. tf_model.compile(optimiaer = tf.keras.optimizers.SGD(), loss = tf.keras.losses.mean_
    squared_error, metrics = ['accuracy'])
21.
22. tf_model.summary()
23. tf_model.fit(XX, L, batch_size = 4, epochs = 2000, verbose = 1)
24. tf_model.evaluate(XX, L)
25. >>> …
26. …
27. Epoch 2000/2000
28. 4/4 [ ============================== ] - 0s 1ms/sample - loss: 0.1588 -
    accuracy: 1.0000
29. 4/1 [ ============================== ] - 0s 61ms/sample - loss: 0.1587 -
    accuracy: 1.0000
30. [0.1586894541978836, 1.0]
31.
32. tf_model.predict(XX)
33. >>> array([[0.3823219 , 0.6143209 ],
34.           [0.60479236, 0.39570323],
35.           [0.6001088 , 0.40094683],
36.           [0.41395947, 0.58794016]], dtype = float32)
```

当采用如图 5-10 所示的(2,2,2)全连接层神经网络时,训练 2000 轮时,误差约为 0.19,四个标签对应的输出为:

```
[0.43767142,0.56202793]→[0.,1.],
[0.5493321,0.45261452]→[1.,0.],
[0.575727,0.42299467]→[1.,0.],
[0.43716326,0.5625658 ]→[ 0.,1.].
```

如果增加隐层的数量,将有效提高模拟效果,比如在第 16 行,将隐层节点数量增加到 4 个,则如第 30 行输出,误差降到约 0.16,对应标签输出如第 33 行到第 36 行所示。读者可以尝试继续增加隐层数量、层数,或者增加训练轮数,比较模拟效果的差异。

2. 误差反向传播学习算法

将 5.4.1 节中的示例推导过程推广到一般情况。

设 BP 神经网络共有 $M+1$ 层,包括输入层和 M 个隐层(第 M 个隐层为输出层)。网络输入分量个数为 U,输出分量个数为 V。其节点编号方法与图 5-8 所示的示例相同。

设神经元采用的激励函数为 $f(x)$。

设训练样本为($\boldsymbol{x}$,$\boldsymbol{l}$),实例向量 $\boldsymbol{x}=(x^{(1)},x^{(2)},\cdots,x^{(U)})$,标签向量 $\boldsymbol{l}=(l^{(1)},l^{(2)},\cdots,l^{(V)})$。

1) 前向传播预测

设第 1 隐层共有 n_1 个节点,它们的输出记为 $\boldsymbol{y}_1=[y_1^{(1)},y_1^{(2)},\cdots,y_1^{(n_1)}]$,它们的

阈值系数记为$\boldsymbol{\theta}_1=[\theta_1^{(1)},\theta_1^{(2)},\cdots,\theta_1^{(n_1)}]$，从输入层到该隐层的连接系数记为$\boldsymbol{W}_1=\begin{bmatrix} w_1^{(1,1)} & \cdots & w_1^{(1,n_1)} \\ \vdots & \ddots & \vdots \\ w_1^{(U,1)} & \cdots & w_1^{(U,n_1)} \end{bmatrix}$。可得：

$$\boldsymbol{y}_1=f(\boldsymbol{x}\boldsymbol{W}_1+\boldsymbol{\theta}_1) \tag{5-29}$$

设第2隐层共有n_2个节点，它们的输出记为$\boldsymbol{y}_2=[y_2^{(1)},y_2^{(2)},\cdots,y_2^{(n_2)}]$，它们的阈值系数记为$\boldsymbol{\theta}_2=[\theta_2^{(1)},\theta_2^{(2)},\cdots,\theta_2^{(n_2)}]$，从第1隐层到该隐层的连接系数记为$\boldsymbol{W}_2=\begin{bmatrix} w_1^{(1,1)} & \cdots & w_1^{(1,n_2)} \\ \vdots & \ddots & \vdots \\ w_1^{(n_1,1)} & \cdots & w_1^{(n_1,n_2)} \end{bmatrix}$。可得：

$$\boldsymbol{y}_2=f(\boldsymbol{y}_1\boldsymbol{W}_2+\boldsymbol{\theta}_2) \tag{5-30}$$

依次可前向计算各层输出，直到输出层。输出为$\boldsymbol{z}=(z^{(1)},z^{(2)},\cdots,z^{(V)})$。

需要注意的是，所有连接系数和阈值系数在算法运行前都需要指定一个初始值，可采用赋予随机数的方式。

2）反向传播学习

设损失函数采用均方误差。输出层的校对误差记为$\boldsymbol{E}_M$：

$$\boldsymbol{E}_M=(E_M^1,E_M^2,\cdots,E_M^V)=\boldsymbol{z}-\boldsymbol{l} \tag{5-31}$$

第$M-1$层的校对误差记为$\boldsymbol{E}_{M-1}$：

$$\boldsymbol{E}_{M-1}=\boldsymbol{E}_M\times\begin{bmatrix} \dfrac{\partial y_M^{(1)}}{\partial y_{M-1}^{(1)}} & \cdots & \dfrac{\partial y_M^{(1)}}{\partial y_{M-1}^{(n_{M-1})}} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial y_M^{(V)}}{\partial y_{M-1}^{(1)}} & \cdots & \dfrac{\partial y_M^{(V)}}{\partial y_{M-1}^{(n_{M-1})}} \end{bmatrix} \tag{5-32}$$

其中，右侧的矩阵是第M层输出对第$M-1$层输出的偏导数排列的矩阵（即第M层输出对第$M-1$层输出的雅可比矩阵）；n_{M-1}是第$M-1$层的节点数。

依次可反向计算各层的校对误差，直到第1隐层。

接下来，根据校对误差更新连接系数和阈值系数。对第i隐层的第j节点的第k个连接系数$w_i^{(k,j)}$：

$$w_i^{(k,j)}\leftarrow w_i^{(k,j)}-\alpha\cdot E_i^j\cdot\frac{\partial y_i^{(j)}}{\partial w_i^{(k,j)}} \tag{5-33}$$

其中，$\dfrac{\partial y_i^{(j)}}{\partial w_i^{(k,j)}}$的计算为：

$$\begin{aligned}\frac{\partial y_i^{(j)}}{\partial w_i^{(k,j)}}&=\frac{\partial y_i^{(j)}}{\partial(\boldsymbol{y}_{i-1}\times\boldsymbol{W}_{i|j}+\theta_i^{(j)})}\cdot\frac{\partial(\boldsymbol{y}_{i-1}\times\boldsymbol{W}_{i|j}+\theta_i^{(j)})}{\partial w_i^{(k,j)}}\\&=f'(x)\big|_{x=y_{i-1}\times\boldsymbol{W}_{i|j}+\theta_i^{(j)}}\cdot y_{i-1}^{(k)}\end{aligned} \tag{5-34}$$

其中，$\boldsymbol{y}_{i-1}\times\boldsymbol{W}_{i|j}+\theta_i^{(j)}$ 为该节点输入的线性组合部分；$\boldsymbol{W}_{i|j}$ 表示 $\boldsymbol{W}_i$ 的第 j 列。式(5-34)中，如果出现 $y_0^{(k)}$，则它表示 $x^{(k)}$，即原始输入。

对该节点的阈值系数 $\theta_i^{(j)}$：

$$\theta_i^{(j)}\leftarrow\theta_i^{(j)}-\alpha\cdot E_i^j\cdot\frac{\partial y_i^{(j)}}{\partial\theta_i^{(j)}}=\theta_i^{(j)}-\alpha\cdot E_i^j\cdot f'(x)\mid_{x=\boldsymbol{y}_{i-1}\times\boldsymbol{W}_{i|j}+\theta_i^{(j)}} \tag{5-35}$$

以上给出了单个训练样本的 BP 算法计算过程。当采用批梯度下降法时，对一批训练样本计算出导数后，取平均数作为下降的梯度。

一般的深度学习框架都内置实现了 BP 算法，除了进行特别的研究外，一般不需要用户实现或修改 BP 算法。

5.4.2　神经网络常用激活函数、损失函数和优化方法

前文的示例，主要采用的激活函数、损失函数和优化方法分别为：Sigmoid、MSE 和 SGD。本节用示例来讨论其他常用的激活函数、损失函数和优化方法，比较各函数和方法的效果。

先给出一个经典的分类任务示例：手写体数字识别。该示例采用全连接层神经网络，因为截至本书完稿时 MindSpore 框架对算子在 CPU 平台上运行的支持还不够多，仍然采用在 TensorFlow 2 深度学习框架下实现。

MNIST 数据集[①]是一个手写体的数字图片集，它包含有训练集和测试集，由 250 个人手写的数字构成。训练集包含 60000 个样本，测试集包含 10000 个样本。每个样本包括一幅图片和一个标签。每幅图片由 28×28 个像素点构成，每个像素点用 1 个灰度值表示。标签是与图片对应的 0～9 的数字。训练集的前 10 幅图片如图 5-11 所示。

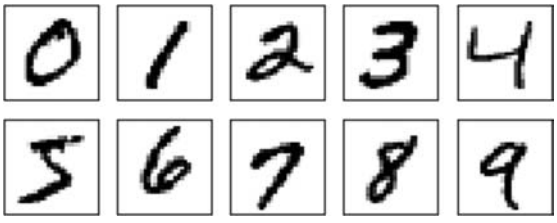


图 5-11　MNIST 图片示例

MNIST 数据集相对简单，适合作为学习神经网络的入门示例。手写体数字识别的任务是构建神经网络，并用训练集让神经网络进行有监督地学习，用验证集来验证它的分类效果。

构建多层全连接神经网络来进行分类任务，示例代码见代码 5-8。

代码 5-8　手写体数字识别多层全连接神经网络示例(MNIST 多层全连接神经网络应用示例.ipynb)

```
1. import numpy as np
2. import tensorflow.keras as ka
```

① http://yann.lecun.com/exdb/mnist/

```
import datetime

np.random.seed(0)

(X_train, y_train), (X_val, y_val) = ka.datasets.mnist.load_data("E:\datasets\MNIST_Data\mnist.npz")                    # 加载数据集,并分成训练集和验证集

num_pixels = X_train.shape[1] * X_train.shape[2]   # 每幅图片的像素数为 784

# 将二维的数组拉成一维的向量
X_train = X_train.reshape(X_train.shape[0], num_pixels).astype('float32')
X_val = X_val.reshape(X_val.shape[0], num_pixels).astype('float32')

# 归一化
X_train = X_train / 255
X_val = X_val / 255

y_train = ka.utils.to_categorical(y_train)   # 转化为独热编码
y_val = ka.utils.to_categorical(y_val)
num_classes = y_val.shape[1]                 # 10

# 多层全连接神经网络模型
model = ka.Sequential([
    ka.layers.Dense(num_pixels, input_shape=(num_pixels,), kernel_initializer='normal', activation='sigmoid'),
    ka.layers.Dense(784, kernel_initializer='normal', activation='sigmoid'),
    ka.layers.Dense(num_classes, kernel_initializer='normal', activation='sigmoid')
])
model.summary()

model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
#model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

startdate = datetime.datetime.now()          # 获取当前时间
model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=20, batch_size=200, verbose=2)
enddate = datetime.datetime.now()

print("训练用时: " + str(enddate - startdate))
>>>
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              #       Param
=================================================================
dense (Dense)                (None, 784)               615440
_________________________________________________________________
```

```
46. dense_1 (Dense)              (None, 784)               615440
47. _________________________________________________________________
48. dense_2 (Dense)              (None, 10)                7850
49. =================================================================
50. Total params: 1,238,730
51. Trainable params: 1,238,730
52. Non-trainable params: 0
53. _________________________________________________________________
54. Train on 60000 samples, validate on 10000 samples
55. Epoch 1/20
56. 60000/60000 - 23s - loss: 0.1025 - accuracy: 0.1292 - val_loss: 0.0903 - val_
    accuracy: 0.1221
57. Epoch 2/20
58. 60000/60000 - 14s - loss: 0.0901 - accuracy: 0.1226 - val_loss: 0.0899 - val_
    accuracy: 0.1230
59. Epoch 3/20
60. ……
```

第 7 行加载数据集。通过 keras.datasets.mnist.load_data()可能无法成功从官网下载，可以用下载工具提前下载或者从其他源下载。本例中已经提前下载 mnist.npz 文件，并存放在 E:\datasets\MNIST_Data\目录下。

第 12 行将二维的图像数据拉成一维，使数据适合多层神经网络的输入要求。第 16～17 行将样本特征进行归一化，灰度的取值范围是 0～255，因此除以 255 就实现了归一化。

第 19 行采用了独热(One-Hot)编码。独热编码常用来处理没有次序的分类特征。

分类特征是在一个集合里没有次序的有限个值，如人的性别、班级编号等。对分类特征常见的编码方式是整数，如男女性别分别表示为 1、0，一班、二班、三班等分别表示为 1、2、3 等。但是，整数编码天然存在次序，而原来的分类特征是没有次序的。如果算法不考虑它们的差别，则会带来意想不到的后果。比如，班级分别用 1、2、3、4 等来编码时，如果机器学习算法忽略了次序问题，就会认为一班和二班之间的距离是 1，而一班和三班之间的距离是 2。

为了防止此类错误的出现，常采用独热编码。假如分类特征有 n 个类别，独热编码则使用 n 位来对它们进行编码。例如，假设有四个班，则一班到四班分别编码为 0001，0010，0100，1000，每个编码只有一位有效。如此，任意两个班之间的 L_p 距离都相等，如 L_1 距离都为 2，L_2 距离都为 $\sqrt{2}$，L_3 距离都为$\sqrt[3]{2}$，…，L_∞距离都为 1。

在用于分类的神经网络中常对输出的类标签采用独热编码。如本示例中，输出的标签类别数为 10，如果不采用独热编码，那么神经网络的输出层为 1 个节点，输出值则可能出现 0 到 9 以外的数。如果采用独热编码，则输出层的节点为 10 个，每次只有一个节点输出 1，其他全为 0。实际上，如表 5-3 所示的模拟异或运算的训练样本的标签就采用了独热编码。

第 24 行到 28 行构建了一个四层神经网络，它有三个隐层(全连接层)，激活函数都采用 Sigmoid 函数。损失函数采用均方误差 MSE，优化算法采用梯度下降法，评测指标采用准确率。

训练 20 轮,对测试样本仅能达到 0.1921 的识别率。

为了使读者更加深入地理解全连接层神经网络,结合该示例来解读两个有关模型的问题。

第 29 行用 summary()方法输出了模型的参数情况,如第 39 行至 51 行所示。可以看到三个全连接层的参数个数分别是 615440、615440 和 7850。因为图片是由 28×28 个像素点构成,因此输入层的节点个数为 784(第 9 代码),第一隐层为 784 个节点,因此作为全连接层,其连接系数个数为 784×784=614656,再加上 784 个节点的阈值系数,所以第一隐层共有 615440 个要学习的参数。

第 7 行在加载数据时,分成了训练集和验证集。在第 35 行模型训练时,在每轮训练结束时用验证集来验证模型效果。将 verbose 参数设置为 2,可以显示详细的训练过程,如第 56 行所示,分别列出每轮训练结束后的训练样本损失值、训练样本准确率、验证样本损失值和验证样本准确率。它们在训练迭代过程中的变化,可以揭示出某些训练情况,如训练样本损失值下降而验证样本损失值上升,则可能已经开始过拟合,如两者持续不变或微小变化,则说明训练遇到瓶颈,可能需要采取减少梯度下降法中的学习率(步长)等措施。

下面用该示例来讨论神经网络中常用的激活函数、损失函数和优化方法。

1. 激活函数

常用的激活函数还有 ReLU 函数、Softplus 函数、tanh 函数和 Softmax 函数等。

ReLU 函数的定义为:

$$f(x)=\max(0,x) \tag{5-36}$$

Softplus 函数的定义为:

$$f(x)=\ln(1+e^{x}) \tag{5-37}$$

ReLU 函数和 Softplus 函数求导简单、收敛快,在神经网络中得到了广泛应用。它们的图像如图 5-12 中实线和虚线所示,Softplus 函数可以看作是“软化”了的 ReLU 函数。

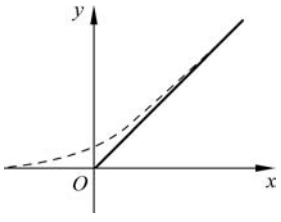


图 5-12 ReLU 函数与 Softplus 函数

tanh 函数的图像类似于 Sigmoid 函数,作用也类似于 Sigmoid 函数。它的定义为:

$$\tanh(x)=\frac{\sinh(x)}{\cosh(x)}=\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \tag{5-38}$$

实际上:

$$\tanh(x)=2\mathrm{Sigmoid}(2x)-1 \tag{5-39}$$

假设有一组实数 $y_1,y_2,\cdots,y_K$(可看作多分类的结果),Softmax 函数将它们转化为一组对应的概率值:

$$p_k=\frac{e^{y_k}}{\sum_{i=1}^{K}e^{y_i}},k=1,2,\cdots,K \tag{5-40}$$

易知 $\sum p_k=1$。

Softmax 函数通过指数运算放大 $y_1,y_2,\cdots,y_K$ 之间的差别,使小的值趋近 0,而使最大值趋近 1,因此它的作用类似于取最大值 max 函数,但又不那么生硬,所以叫

Softmax。假如有一组数 1、2、5、3，容易计算出它们的 Softmax 函数值分别约为 0.01、0.04、0.83、0.11，将它们的原数值和 Softmax 函数值、max 函数值等比例画出，如图 5-13 所示。

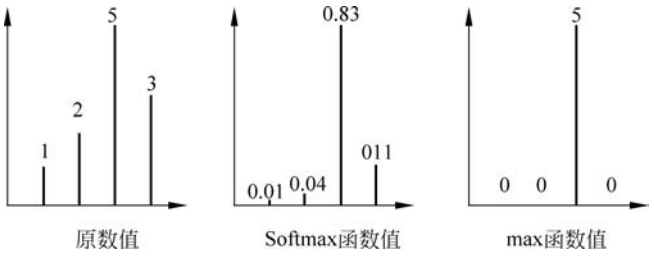


图 5-13　Softmax 函数作用示例

Softmax 函数在神经网络中主要用来作输出值的归一化，常用于分类任务的神经网络的输出层的激活函数中。

修改代码 5-8 第 24 行到第 27 行代码，使模型分别采用不同激活函数组合进行比较，其他参数不变，仍为 MSE 损失函数、SGD 优化方法，并训练 20 轮，运行结果如表 5-4 所示。

表 5-4　MNIST 分类中不同激活函数组合时的效果比较

序　号	隐层 1	隐层 2	输出层	测试样本准确率
1	Softmax	Softmax	Softmax	0.1135
2	ReLU	ReLU	ReLU	0.9202
3	Softplus	Softplus	Softplus	0.8136
4	tanh	tanh	tanh	0.9030
5	Sigmoid	Sigmoid	Softmax	0.2195
6	ReLU	ReLU	Softmax	0.8617

可见，采用不同的激活函数，其效果有很大的差异。

采用什么样的激活函数，要根据理论研究、工程经验和试验综合分析。如在 4.5.3 节的过拟合示例中，如果采用 Softplus 激活函数，训练轮数仍为 5000，网络结构仍然是四层(1,5,5,1)结构，分别对样本特征进行归一化处理和不归一化处理时拟合多项式的结果如图 5-14 所示。

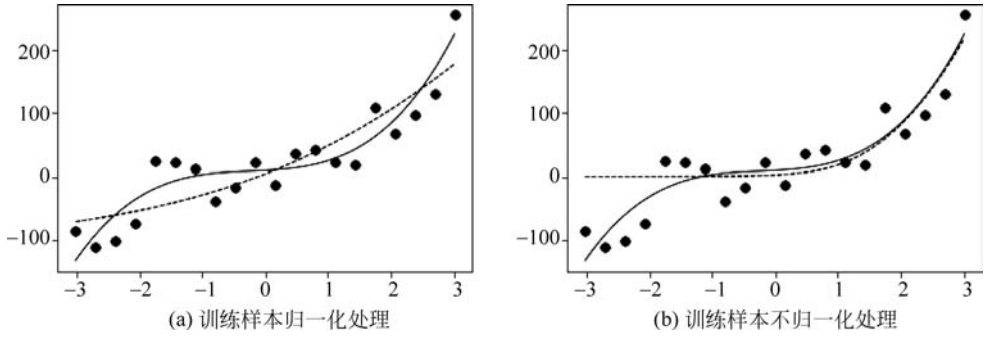


图 5-14　采用 Softplus 激活函数拟合多项式的结果

这是因为 Softplus 函数将负数趋近 0(见图 5-12),因此在不归一化处理时,网络对目标函数的负数部分处理能力很低。

2. 损失函数

前文采用的平方和形式的损失函数 MSE 是基于欧氏距离的损失函数。神经网络中常用的损失函数还有 KL 散度损失函数(Kullback-Leibler Divergence)、交叉熵(Crossentropy)损失函数等。

交叉熵可以用来衡量两个分布之间的差距,下面以示例入手讨论。

代码 5-6 模拟了异或运算三层感知机的误差反向传播学习过程,最后给出了预测输出与标签值的对比,重新列出如下:

```
(a) [ 0.07158904 0.92822515 ] → [ 0. 1.]
(b) [ 0.9138734 0.08633152 ] → [ 1. 0.]
(c) [ 0.91375259 0.08644981 ] → [ 1. 0.]
(d) [ 0.11774177 0.88200493 ] → [ 0. 1.]
```

对于(a)和(d)两项输出,标签值都是[0. 1.],直观来看(a)的预测应该更准一些。如何形式化地度量它们与标签值的差距呢?

用 p_i 表示第 i 个输出的标签值,即真实值;用 q_i 表示第 i 个输出值,即预测值。将 p_i 与 q_i 之间的对数差在 p_i 上的期望值称为相对熵:

$$D_{KL}(\|p\|q)=E_{p_i}(\ln p_i-\ln q_i)=\sum_{i=1}^{n}p_i(\ln p_i-\ln q_i)=\sum_{i=1}^{n}p_i\ln\frac{p_i}{q_i} \tag{5-41}$$

计算(a)和(d)两项输出的相对熵:

$$\begin{cases}D_a=0\times\ln\dfrac{0}{0.07158904}+1\times\ln\dfrac{1}{0.92822515}=0.07447962\\ D_d=0\times\ln\dfrac{0}{0.11774177}+1\times\ln\dfrac{1}{0.88200493}=0.12555622\end{cases} \tag{5-42}$$

其中,$0\times\ln 0$ 计为 0。

与直接观察的结论相同。可见,相对熵越大的输出与标签值差距越大。如果 p_i 与 q_i 相同,那么 $D_{KL}(p\|q)=0$。

值得注意的是,相对熵不具有对称性。相对熵又称为 KL 散度。

将相对熵的定义式(5-41)进一步展开:

$$D_{KL}(p\|q)=\sum_{i=1}^{n}p_i(\ln p_i-\ln q_i)=\sum_{i=1}^{n}p_i\ln p_i+\left[-\sum_{i=1}^{n}p_i\ln q_i\right] \tag{5-43}$$

前一项的值只与真实值 p_i 有关,因此一般用后一项作为两个分布之间差异的度量,称为交叉熵:

$$H(p,q)=-\sum_{i=1}^{n}p_i\ln q_i \tag{5-44}$$

如果只有正负两个分类(标签记为+1 和-1),记标签为正类的概率为 y,记预测为正类的概率为 p,那么式(5-44)为:

$$H(y,p)=-[y\ln p+(1-y)\ln(1-p)] \tag{5-45}$$

交叉熵损失函数在梯度下降法中可以改善 MSE 学习速率降低的问题，得到了广泛的应用。

采用 SGD 优化方法，三层分别采用 ReLU、ReLU 和 Softmax 激活函数，训练 20 轮，采用不同的损失函数进行比较，代码 5-8 所示的示例的运行结果如表 5-5 所示。

表 5-5　MNIST 分类中采用不同损失函数时的效果比较

序　号	损 失 函 数	测试样本准确率
1	KLD	0.9523
2	categorical_crossentropy	0.9540
3	MSE	0.8617

3. 多层神经网络常用优化算法

下面讨论常用于多层神经网络中的优化算法，它们都是梯度下降法的改进方法，主要从增加动量和调整优化步长两方面着手。

1）步长优化算法

在 4.2.1 节简要讨论了步长对梯度下降的影响及调整大小的策略。为了克服固定步长的弊端，MindSpore 深度学习框架和 TensorFlow 2 深度学习框架都提供了动态调整步长的方法。

代码 5-9　MindSpore 和 TensorFlow 2 中的 SGD 原型

```
# MindSpore 框架下
class mindspore.nn.SGD(params, learning_rate = 0.1, momentum = 0.0, dampening = 0.0,
    weight_decay = 0.0, nesterov = False, loss_scale = 1.0)

# TensorFlow 框架下
tf.keras.optimizers.SGD(
    learning_rate = 0.01, momentum = 0.0, nesterov = False, name = 'SGD', **kwargs
)
```

MindSpore 和 TensorFlow 2 中的 SGD 原型见代码 5-9。两者原型中的 learning_rate 超参数（即梯度下降法中的步长，也称为学习率）默认初始值都是固定的 0.1，可以设置为动态的步长。设置动态步长可以使用框架预定义的方法，也可以使用用户自行定义的方法。

MindSpore 提供了函数和类两种预定义的动态调整步长方法，两种方法的具体功能相近，它们分别按余弦函数、指数函数、与时间成反比、多项式函数等方式衰减步长。用官网上的指数函数衰减例子[①]来说明，见代码 5-10。

① https://www.mindspore.cn/doc/api_python/zh-CN/r1.1/mindspore/nn/mindspore.nn.exponential_decay_lr.html#mindspore.nn.exponential_decay_lr

代码 5-10　mindspore. nn. exponential_decay_lr 应用示例

```
learning_rate = 0.1
decay_rate = 0.9
total_step = 6
step_per_epoch = 2
decay_epoch = 1
output = exponential_decay_lr(learning_rate, decay_rate, total_step, step_per_epoch,
decay_epoch)
print(output)
>>> [0.1, 0.1, 0.09000000000000001, 0.09000000000000001, 0.08100000000000002,
0.08100000000000002]
```

设当前为第 i 步，其步长的计算方法为：

$$\text{decayed\_learning\_rate}[i] = \text{learning\_rate} \times \text{decay\_rate}^{\frac{\text{current\_epoch}}{\text{decay\_epoch}}} \tag{5-46}$$

其中，$\text{current\_epoch} = \text{floor}\left(\dfrac{i}{\text{step\_per\_epoch}}\right)$，floor 为向下取整运算。

在示例中，当 $i=0$ 时，$\text{current}_{\text{epoch}} = \text{floor}\left(\dfrac{0}{2}\right) = 0$，即当前为第 0 轮，可知 decayed_learning_rate[0]=0.1。读者可自行验算其他输出值。

在 TensorFlow 2 框架中也提供了类似的动态调整步长方法，它们都在 tensorflow. keras. optimezers. schedules 模块内。读者可在需要时查阅资料，不再赘述。

这些动态调整步长的方法，实际上并没有结合优化的具体进展来设定步长，仍然可以看成是一组预先设定的步长，只不过它们的大小按一定的方式逐步衰减了。

因此，人们又研究出结合优化具体进展的自适应步长调整方法。

Adagrad(Adaptive Gradient)算法记录下所有历史梯度的平方和，并用它的平方根来除以步长，这样就使得当前的实际步长越来越小。

MindSpore 中实现该算法的类为 mindspore. nn. Adagrad。TensorFlow 2 中实现该算法的类是 tf. keras. optimizers. Adagrad。

2) 动量优化算法

在经典力学中，动量(Momentum)表示物体的质量和其质心速度的乘积，体现为物体在其运动方向上保持运动的趋势。在梯度下降法中，如果使梯度下降的过程具有一定的“动量”，具有保持原方向运动的一定的“惯性”，则有可能在下降的过程中“冲过”小的“洼地”，避免陷入极小值点，如图 5-15 所示。其中，在第 3 个点处，其梯度负方向如虚线实箭头所示，而在动量的影响下，仍然保持向左的“惯性”，从而“冲出”了局部极小点。

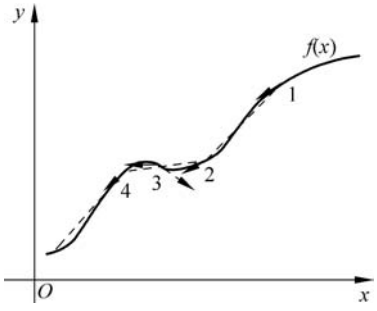


图 5-15　加入动量的梯度下降过程示意图

加入动量优化，梯度下降法还可以克服前进路线振荡的问题，从而加快收敛速度。

在 SGD 算法中，通过配置 Momentum 参数(见

代码 5-9 中相应的参数),就可以使梯度下降法利用这种“惯性”。Momentum 参数设置的是“惯性”的大小。

加入动量的梯度下降的迭代关系式还有一种改进方法,称为 NAG(Nesterov Accelerated Gradient)。该方法中,计算梯度的点发生了变化,它可以理解为先按“惯性”前进一小步,再计算梯度。这种方法在每一步都往前多走了一小步,有时可以加快收敛速度。设置 SGD 的 nesterov(见代码 5-9 中相应的参数)为 True,即可使用该算法。

在 MindSpore 中专门实现了该算法:mindspore. nn. Momentum,实际上,在第 4 章的模拟线性回归示例中已经应用过(代码 4-14 的第 32 行)该算子。

3) 结合动量和步长优化的算法

结合动量和步长进行优化的算法有 RMSProp(Root Mean Square Prop)算法和 Adam(Adaptive Moment Estimation)算法等。

RMSProp 算法通过对 Adagrad 算法逐步增加控制历史信息与当前梯度的比例系数、增加动量因子和中心化操作形成了三个版本。在 MindSpore 中,实现该算法的类是 mindspore. nn. RMSProp,在 TensorFlow 2 中实现该算法的是 tensorflow. keras. optimizers. RMSprop。

Adam 算法是一种结合了 AdaGrad 算法和 RMSProp 算法优点的算法。Adam 算法综合效果较好,应用广泛。

在 MindSpore 中,实现 Adam 算法的是 mindspore. nn. Adam。在 TensorFlow 2 中实现该算法的是 tf. keras. optimizers. Adam。

下面仍然示例它们的效果,如果需要深入研究原理,可参考原版书。

代码 5-8 所示的示例,如果采用 Adam 算法,还是训练 20 轮,能够达到 0. 9812 的识别率。读者可自行试验一下。

神经网络三隐层分别采用 ReLU、ReLU 和 Softmax 激活函数组合,采用交叉熵损失函数,训练 20 轮,采用不同的优化方法,代码 5-8 所示的示例的运行结果如表 5-6 所示。

表 5-6　MNIST 分类中采用不同优化方法时的效果比较

序　　号	优化方法	测试样本准确率
1	SGD	0. 9540
2	AdaGrad	0. 9735
3	rmsprop	0. 9824
4	Adam	0. 9823

不同的优化算法有不同的特点,读者可通过更多的练习来摸索它们的应用方法和特点。

5.4.3　局部收敛与梯度消散

本节简要讨论多层神经网络的两个问题。

1. 局部收敛

BP 神经网络不一定收敛,也就是说,网络的训练不一定成功。误差的平方是非凸函

数,BP 神经网络是否收敛或者能否收敛到全局最优,与初始值有关。读者可以将代码 5-6 中的参数全部置初值为 0.1 再运行,看能否收敛。

全局优化与凸函数的问题,以及机器学习算法尽量避免局部最优的方法,前文已经进行了简要讨论,有需要的读者也可参考原版书。

2. 梯度消散和梯度爆炸

在校对误差反向传播的过程中,见式(5-32),如果偏导数较小(如图 2-7 中大于 c 的区域,称为处于非线性激活函数的饱和区),在多次连乘之后,校对误差会趋近 0,导致梯度也趋近 0,前面层的参数无法得到有效更新,这种情况称为梯度消散。梯度消散会使得增加再多的层也无法提高效果,甚至反而会降低效果。

相反,如果偏导数较大,则梯度会在反向传播的过程中呈指数级增长,导致溢出,无法计算,网络不稳定,这种情况称为梯度爆炸。

梯度消散和梯度爆炸只在层次较多的网络中出现,常用的解决方法包括尽量使用合适的激活函数(如 ReLU 函数,它在正数部分导数为 1);预训练;合适的网络模型(有些网络模型具有预防梯度消散和梯度爆炸能力);梯度截断,等等。

5.5 卷积神经网络

卷积神经网络(Convolutional Neural Network,CNN)在提出之初被成功应用于手写字符图像识别[11],2012 年的 AlexNet 网络[12]在图像分类任务中取得成功,此后,卷积神经网络发展迅速,现在已经被广泛应用于图形、图像、语音识别等领域。

图片的像素数往往非常大,如果用多层全连接网络来处理,则参数数量将大到难以有效训练的地步。受猫脑研究的启发,卷积神经网络在多层全连接网络的基础上进行了改进,它在不减少层数的前提下有效地提升了训练速度。卷积神经网络在多个研究领域都取得了成功,特别是在与图形有关的分类任务中。


视频讲解

5.5.1 卷积神经网络示例

本节用示例来展示卷积神经网络在图像识别方面的优势,并将在随后的几节中逐一剖析其中的关键点。

代码 5-8 所示的是用多层全连接神经网络来完成手写体数字识别示例。通过采用交叉熵损失函数和 Adam 优化算法,以及修改网络结构、增加训练轮数等措施,发现最高能达到 0.983 左右的识别率。

先示例在 TensorFlow 2 框架下的实现,再对比示例 MindSpore 框架下的实现。

在 TensorFlow 2 框架下,用较简单的卷积神经网络只需要 2 轮训练就可以轻松达到 0.986 的识别率,见代码 5-11。

代码 5-11 TensorFlow 2 框架下 MNIST 示例(MINST 卷积神经网络示例.ipynb)

```
1. import numpy as np
2. import tensorflow.keras as ka
```

```
import datetime

np.random.seed(0)

(X_train, y_train), (X_val, y_val) = ka.datasets.mnist.load_data("E:\datasets\MNIST_Data\mnist.npz")

# 将数组转换成卷积层需要的格式
X_train = X_train.reshape(X_train.shape[0],28, 28, 1).astype('float32')
X_val = X_val.reshape(X_val.shape[0], 28, 28, 1).astype('float32')

X_train = X_train / 255
X_val = X_val / 255

y_train = ka.utils.to_categorical(y_train)          # 转化为独热编码
y_val = ka.utils.to_categorical(y_val)
num_classes = y_val.shape[1]                        # 10

# CNN 模型
model = ka.Sequential([
    ka.layers.Conv2D(filters = 32, kernel_size = (5, 5), input_shape = (28, 28, 1), activation = 'relu'),
    ka.layers.MaxPooling2D(pool_size = (2, 2)),
    ka.layers.Dropout(0.2),
    ka.layers.BatchNormalization(),
    ka.layers.Flatten(),
    ka.layers.Dense(128, activation = 'relu'),
    ka.layers.Dense(num_classes, activation = 'softmax')
])
model.summary()

model.compile(loss = 'categorical_crossentropy', optimizer = 'adam', metrics = ['accuracy'])

startdate = datetime.datetime.now()                 # 获取当前时间
model.fit(X_train, y_train, validation_data = (X_val, y_val), epochs = 2, batch_size = 200, verbose = 2)
enddate = datetime.datetime.now()
print("训练用时: " + str(enddate - startdate))
```

第 21 行到 29 行是构建卷积神经网络的代码，第 22 行添加的是卷积层，第 23 行添加的是池化层。

卷积层和池化层是卷积神经网络的核心组成，它们和全连接层一起可以组合成很多层次的网络。卷积神经网络还可以按需添加用来抑制过拟合的 Dropout 层（第 24 行）、拉平多维数据的 Flatten 层（第 25 行）、加快收敛和抑制梯度消散的批标准化 BatchNormalization 层（第 26 行）等。

在 MindSpore 框架中，对照实现该示例的代码见代码 5-12。

代码 5-12　MindSpore 框架下 MNIST 示例(MINST 卷积神经网络示例.ipynb)

```
import os
import mindspore.dataset as ds
import mindspore.nn as nn
from mindspore import Model
from mindspore.common.initializer import Normal
from mindspore.train.callback import LossMonitor
import mindspore.dataset.vision.c_transforms as CV
import mindspore.dataset.transforms.c_transforms as C
from mindspore.nn.metrics import Accuracy
from mindspore import dtype as mstype
from mindspore.nn import SoftmaxCrossEntropyWithLogits

def create_dataset(data_path, batch_size=32, repeat_size=1, num_parallel_workers=1):
    # 从 mnist 文件产生数据集

    mnist_ds = ds.MnistDataset(data_path)

    rescale = 1.0 / 255.0                    # 归一化比例
    shift = 0.0
    rescale_nml = 1 / 0.3081
    shift_nml = -1 * 0.1307 / 0.3081

    # map 算子
    rescale_nml_op = CV.Rescale(rescale_nml, shift_nml)
    rescale_op = CV.Rescale(rescale, shift)
    hwc2chw_op = CV.HWC2CHW() # (height, width, channel) -> (channel, height, width)
    type_cast_op = C.TypeCast(mstype.int32)

    mnist_ds = mnist_ds.map(operations=type_cast_op, input_columns="label", num_
parallel_workers=num_parallel_workers)
    mnist_ds = mnist_ds.map(operations=rescale_op, input_columns="image", num_
parallel_workers=num_parallel_workers)
    mnist_ds = mnist_ds.map(operations=rescale_nml_op, input_columns="image", num_
parallel_workers=num_parallel_workers)
    mnist_ds = mnist_ds.map(operations=hwc2chw_op, input_columns="image", num_
parallel_workers=num_parallel_workers)

    buffer_size = 10000
    mnist_ds = mnist_ds.shuffle(buffer_size=buffer_size)
    mnist_ds = mnist_ds.batch(batch_size, drop_remainder=True)
    mnist_ds = mnist_ds.repeat(repeat_size)

    return mnist_ds


class CNNNet(nn.Cell):
    def __init__(self, num_class=10, num_channel=1):
```

```
44.         super(CNNNet, self).__init__()
45.         self.conv = nn.Conv2d(num_channel, 32, 5, pad_mode = 'valid', has_bias = True)
46.         self.fc1 = nn.Dense(32 * 12 * 12, 128, weight_init = Normal(0.02))
47.         self.fc2 = nn.Dense(128, num_class, weight_init = Normal(0.02))
48.         self.relu = nn.ReLU()
49.         self.max_pool2d = nn.MaxPool2d(kernel_size = 2, stride = 2)
50.         self.flatten = nn.Flatten()
51.         self.dropout = nn.Dropout(keep_prob = 0.8)
52.         self.bn = nn.BatchNorm2d(num_features = 32)
53.         self.softmax = nn.softmax()
54.     def construct(self, x):
55.         x = self.relu(self.conv(x))
56.         x = self.max_pool2d(x)
57.         x = self.dropout(x)
58.         x = self.bn(x)
59.         x = self.flatten(x)
60.         x = self.relu(self.fc1(x))
61.         x = self.softmax(self.fc2(x))
62.         return x
63.
64. lr = 0.01
65. momentum = 0.9
66. dataset_size = 1
67. mnist_path = "E:\datasets\MNIST_Data"
68. net_loss = SoftmaxCrossEntropyWithLogits(sparse = True, reduction = 'mean')
69. train_epoch = 2
70. net = CNNNet()
71. net_opt = nn.Momentum(net.trainable_params(), lr, momentum)
72. ms_model = Model(net, net_loss, net_opt, metrics = {"Accuracy": Accuracy()})
73. ds_train = create_dataset(os.path.join(mnist_path, "train"), 200, dataset_size)
74. startdate = datetime.datetime.now()       # 获取当前时间
75. ms_model.train(train_epoch, ds_train, callbacks = [LossMonitor()], dataset_sink_mode
    = False)
76. enddate = datetime.datetime.now()
77. print("训练用时: " + str(enddate - startdate))
78. >>> epoch: 1 step: 1, loss is 2.325413
79. epoch: 1 step: 2, loss is 2.2675755
80. …
81. epoch: 2 step: 299, loss is 0.099331215
82. epoch: 2 step: 300, loss is 0.023031829
83. 训练用时: 0:03:47.365005
84.
85. ds_eval = create_dataset(os.path.join(mnist_path, "test"))
86. acc = ms_model.eval(ds_eval, dataset_sink_mode = False)
87. print(format(acc))
88. >>> {'Accuracy': 0.9801682692307693}
```

第 13 行到第 39 行数据处理函数，它完成对训练集和验证集馈入模型前的准备工作。

第 42 行到第 62 行建立 CNN 模型,该模型结构与代码 5-11 所示的 TensorFlow 2 框架下的结构相近,都包括卷积层、池化层、Dropout 层、Flatten 层和 BatchNormalization 层等。

由于截至本书完稿时 MindSpore 还不支持在 CPU 平台上运行除 Momentum 之外的优化算法,因此无法采用其他优化算法。该示例本次运行最终在验证集上的准确率为 0.98。


视频讲解

5.5.2 卷积层

代码 5-11 中第 22 行和代码 5-12 中第 45 行的二维卷积层 Conv2d 的输入是 input_shape=(28,28,1),这与前文讨论的所有机器学习模型的输入都不同。前文模型的输入是一维向量,该一维向量要么是经特征工程提取出来的特征,要么是被拉成一维的图像数据(见代码 5-8 所示的多层全连接神经网络手写体数字识别示例)。而这里卷积层的输入是图片数据组成的多维数据。

在 3.6 节介绍过有关图像的知识。在 MNIST 图片中,只有一种颜色,通常称灰色亮度。MNIST 图片的维度是(28,28,1),前面两维存储 28×28 个像素点的坐标位置,后面 1 维表示像素点的灰色亮度值,因此它是 28×28 的单通道数据。

在数学领域,卷积是一种积分变换。卷积在很多领域都得到了广泛的应用,如在统计学中它可用来做统计数据的加权滑动平均,在电子信号处理中通过将线性系统的输入与系统函数进行卷积得到系统输出……。在深度学习中,它用来做数据的卷积运算,在图像处理领域取得了非常好的效果。

在单通道数据上的卷积运算示例如图 5-16 所示。单通道数据上的卷积运算包括待处理张量 I、卷积核 K 和输出张量 S 三个组成部分,它们的大小分别为 4×4、3×3 和 2×2。

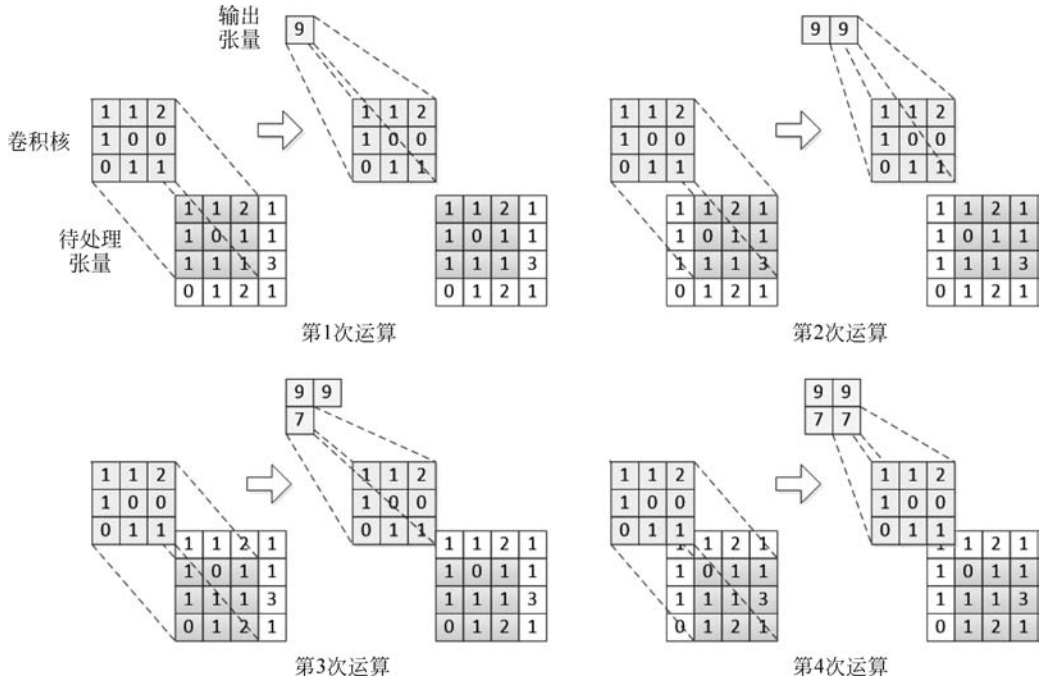


图 5-16 卷积运算示例(见彩插)

共进行了 4 次运算。第 1 次运算先用卷积核的左上角去对准待处理张量的左上角，位置为 $I(0,0)$，如图中深色部分。然后，将卷积核与对准部分的相应位置的值相乘再求和(可看作矩阵的点积运算)：$1\times1+1\times1+2\times2+1\times1+0\times0+0\times1+0\times1+1\times1+1\times1=9$。所以，第 1 次运算的输出为 9，记为 $S(0,0)=9$。

第 2 次运算，将卷积核向右移动一步，卷积核的左上角对准待处理张量的位置为 $I(0,1)$，再进行相应位置值的相乘求和，得到输出为 $S(0,1)=9$。

第 3 次运算，因为卷积核已经到达最右边，因此下移一行，从最左边 $I(1,0)$开始对准，然后再进行相应位置值的相乘求和，得到输出为 $S(1,0)=7$。

第 4 次运算，将卷积核向右移动一步，到达 $I(1,1)$，再与对准部分的相应位置的值相乘求和，得到输出为 $S(1,1)=7$。

卷积核已经到达待处理张量的最右侧和最下侧，卷积运算结束。每次输出的结果也按移动位置排列，得到输出张量 $\boldsymbol{S}=\begin{bmatrix}9 & 9\\ 7 & 7\end{bmatrix}$。

记待处理的张量为 I，卷积核为 K，每一次卷积运算可表述为：

$$S(i,j)=(I*K)(i,j)=\sum_{m=1}^{M}\sum_{n=1}^{N}I(i+m-1,j+n-1)K(m,n) \tag{5-47}$$

其中，$I*K$ 表示卷积运算，M 和 N 分别表示卷积核的长度和宽度。i,j 是待处理张量 $\boldsymbol{I}$ 的坐标位置，也是卷积核左上角对齐的位置。

按式(5-47)从上到下，从左到右依次卷积运算，可得输出张量 $\boldsymbol{S}$。记待处理张量 $\boldsymbol{I}$ 的长度和宽度为 P 和 Q，则输出张量 S 的长度 P' 和 Q' 宽度分别为：

$$\begin{cases}P'=P-M+1\\ Q'=Q-N+1\end{cases} \tag{5-48}$$

在 MindSpore 框架中，在设置有关层的输入参数时，需要计算该值(将在后文详细讨论)。

代码 5-11 所示的示例，输入为 28×28，卷积核为 5×5，因此输出为 24×24。

在实际应用中，与神经元模型一样，卷积运算往往还要加 1 个阈值 θ，即：

$$S(i,j)=(I*K)(i,j)=\sum_{m=1}^{M}\sum_{n=1}^{N}I(i+m,j+n)K(m,n)+\theta \tag{5-49}$$

其中，卷积核 K 和阈值 θ 是要学习的参数。

如果数据是多通道的，则卷积核也分为多层，每一层对应一个通道，各层参数不同。每层卷积核的操作与单通道上的卷积操作相同，最终输出是每层输出的和再加上阈值，如图 5-17 所示。因此，无论输入的张量有多少个通道，经过一个卷积核后的输出都是单层的。

从卷积运算的过程可见，卷积层的输出只与部分输入有关。虽然要扫描整个输入层，但卷积核的参数是一样的，这称为参数共享(Parameter Sharing)。参数共享显著减少了需要学习的参数的数量。

在卷积运算中，一般会设置多个卷积核。代码 5-11 所示的示例中设置了 32 个卷积核(TensorFlow 2 中称为过滤器 filters)，每个卷积核输出一层，因此该卷积层的输出是

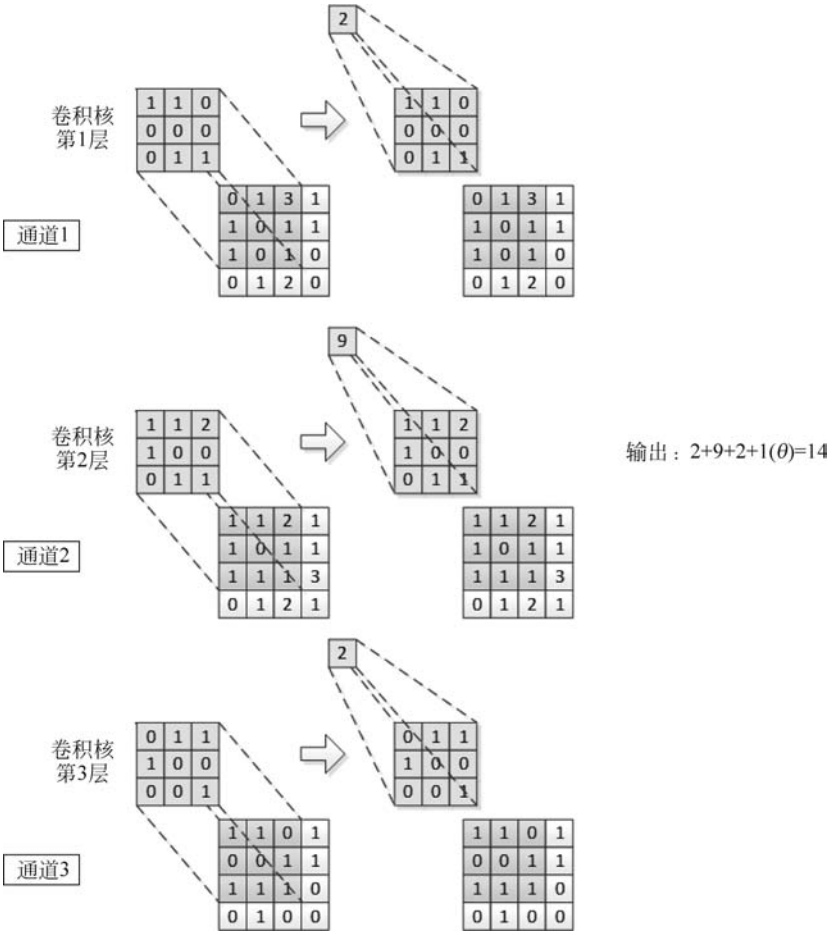


图 5-17　多通道卷积运算示例

32 层的，也就是说将 28×28×1 的数据变成了 24×24×32 的。在画神经网络结构图时，一般用图 5-18 中的长方体来表示上述卷积运算，用水平方向长度表示卷积核的数量。

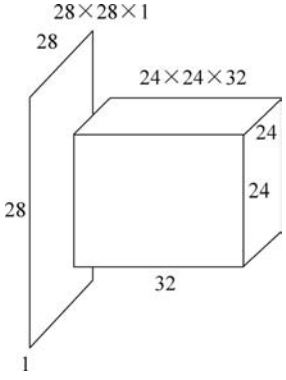


图 5-18　卷积层图示

再来算一下代码 5-11 示例中该卷积层的训练参数量。因为输入是单通道的，因此每个卷积核只有一层，它的参数为 5×5+1=26 个，共 32 个卷积核，因此训练参数为 26×32=832 个。

如果待处理的张量规模很大，可以将卷积核由依次移动改为跳跃移动，即一次移动两个或多个数据单元，这称为加大步长(Strides)。加大步长可以减少计算量、加快训练速度。

为了提取到边缘的特征，可以在待处理张量的边缘填充 0 再进行卷积运算，称为零填充(Zero-Padding)，如图 5-19 所示。填充也可以根据就近的值进行填充。

边缘填充的另一个用途是在张量与卷积核不匹配时，通

过填充使之匹配，从而卷积核能扫描到所有数据。

如采用图 5-19 所示的填充，在步长为 1 时，输出张量的长度和宽度都要加 2。

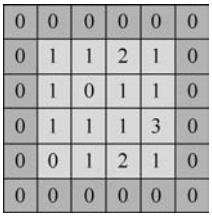


图 5-19　边缘填充示例

来观察一下代码 5-11 中第 22 行的二维卷积层的详细情况。该卷积层的输入为(28,28,1)的张量，为一幅 MNIST 图片。它设置了 32 个卷积核，每个卷积核大小为(5,5)，不进行边缘填充(默认设置)，采用 ReLU 激活函数。

代码 5-12 的第 45 行和第 55 行在 MindSpore 下完成了同样的工作。MindSpore 和 TensorFlow 2 下的 Conv2d 算子的定义原型见代码 5-13，读者可以对比一下它们的参数。

代码 5-13　MindSpore 和 TensorFlow 2 中 Conv2d 算子的原型

```
# MindSpore
class mindspore.nn.Conv2d(in_channels, out_channels, kernel_size, stride = 1, pad_mode = 'same', padding = 0, dilation = 1, group = 1, has_bias = False, weight_init = 'normal', bias_init = 'zeros', data_format = 'NCHW')

# TensorFlow 2
tf.keras.layers.Conv2D(
    filters, kernel_size, strides = (1, 1), padding = 'valid',
    data_format = None, dilation_rate = (1, 1), groups = 1, activation = None,
    use_bias = True, kernel_initializer = 'glorot_uniform',
    bias_initializer = 'zeros', kernel_regularizer = None,
    bias_regularizer = None, activity_regularizer = None, kernel_constraint = None,
    bias_constraint = None,  ** kwargs
)
```

需要注意的是，它们默认的输入数据的格式不一样。TensorFlow 2 的 Conv2d 的输入图像格式为(height,width,channel)，而 MindSpore 的 Conv2d 默认的输入图像格式为(channel,height,width)。它们的通道数的位置不一样，可以通过设置 data_format 参数来改变默认格式。代码 5-12 的第 26 行将原始格式转换成 MindSpore 框架默认的要求格式。

代码 5-14 对 MindSpore 和 TensorFlow 2 框架中 Conv2d 算子以本小节的例子进行了验证，通过分别设置卷积核的数量(即通道数)和边缘填充方式，来观察输出张量的 shape。如第 11 行的输出，是在 MindSpore 框架下，对图 5-16 所示的卷积运算在不进行边缘填充时的验证输出。MindSpore 框架中 Conv2d 算子的输入和输出的四维张量的含义分别为(批大小，通道数，高，宽)。其他情况读者可自行对比验证，不再赘述。

代码 5-14　Conv2d 算子验证(Conv2d 算子验证.ipynb)

```
# MindSpore
import mindspore
import numpy as np
import mindspore.nn as nn
from mindspore import Tensor
```

```
input = Tensor(np.ones([1, 1, 4, 4]), mindspore.float32)

net = nn.Conv2d(1, 1, 3, has_bias = True, pad_mode = 'valid')     # 1 卷积核,valid 边缘填充
output = net(input).shape
print("1 卷积核,valid 边缘填充", output)
>>> 1 卷积核,valid 边缘填充 (1, 1, 2, 2)
net = nn.Conv2d(1, 5, 3, has_bias = True, pad_mode = 'valid') # 5 卷积核,valid 边缘填充
output = net(input).shape
print("5 卷积核,valid 边缘填充", output)
>>> 5 卷积核,valid 边缘填充 (1, 5, 2, 2)
net = nn.Conv2d(1, 1, 3, has_bias = True, pad_mode = 'same')  # 1 卷积核,same 边缘填充
output = net(input).shape
print("1 卷积核,same 边缘填充", output)
>>> 1 卷积核,same 边缘填充 (1, 1, 4, 4)
net = nn.Conv2d(1, 1, 3, has_bias = True, pad_mode = 'pad')   # 1 卷积核,pad 边缘填充
output = net(input).shape
print("1 卷积核,pad 边缘填充", output)
>>> 1 卷积核,pad 边缘填充 (1, 1, 2, 2)

# TensorFlow 2
import tensorflow as tf
input_shape = (1, 4, 4, 1)
x = tf.random.normal(input_shape)

network = tf.keras.layers.Conv2D(1, 3, activation = 'relu', padding = "valid", input_
    shape = input_shape[1:])
y = network(x)
print("1 卷积核,valid 边缘填充", y.shape)
>>> 1 卷积核,valid 边缘填充 (1, 2, 2, 1)
network = tf.keras.layers.Conv2D(1, 3, activation = 'relu', padding = "same", input_
    shape = input_shape[1:])
y = network(x)
print("1 卷积核,same 边缘填充", y.shape)
>>> 1 卷积核,same 边缘填充 (1, 4, 4, 1)
network = tf.keras.layers.Conv2D(5, 3, activation = 'relu', padding = "valid", input_
    shape = input_shape[1:])
y = network(x)
print("5 卷积核,valid 边缘填充", y.shape)
>>> 5 卷积核,valid 边缘填充 (1, 2, 2, 5)
```

5.5.3 池化层和 Flatten 层

池化(Pooling)层一般跟在卷积层之后,用于压缩数据和参数的数量。

池化操作也称为下采样(Sub-Sampling),具体过程与卷积层基本相同,只不过池化层的卷积核只取对应位置的最大值或平均值,分别称为最大池化或平均池化。最大池化操作如图 5-20 所示,将对应位置中的最大值输出,结果为 2。如果是平均池化,则将对应位置中的所有值求平均值,得到输出 1。池化层没有需要训练的参数。

池化层的移动方式与卷积层不同，它不重叠地移动，图 5-20 所示的池化操作，输出的张量的规模为 2×2。代码 5-11 第 23 行和代码 5-12 第 56 行池化层输出的张量为 12×12×32。

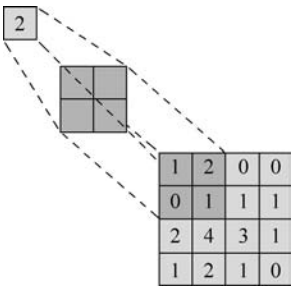


图 5-20　最大池化操作示例

代码 5-11 第 24 行和和代码 5-12 第 57 行添加的是 Dropout 层。

代码 5-11 第 25 行和和代码 5-12 第 58 行添加的是所谓的批标准化层，将在 5.5.4 节讨论。

代码 5-11 第 26 行和和代码 5-12 第 59 行添加的是 Flatten 层。Flatten 层很简单，只是将输入的多维数据拉成一维的，可以理解为将数据“压平”。

代码 5-11 第 27、28 行和和代码 5-12 第 60、61 行添加的是全连接层。代码 5-12 中的全连接层在第 46、47 行定义。

MindSpore 中的全连接层算子 Dense 需要显式设置输入参数个数，来看第 46 行定义的 Dense 算子的输入参数个数是如何计算的。在卷积层中，每个卷积核将输入的 1×28×28（按 MindSpore 默认的数据格式要求，将通道数写在前面）格式的数据转换成了 1×24×24（式(5-48)）格式的数据，因为有 32 个卷积核，因此该卷积层的最终输出数据格式为 32×24×24。再经过一个核为 2×2 的池化层，输出数据格式为 32×12×12。因此，它就是第 46 行定义 Dense 算子时的输入参数。

在画神经网络结构图时，可以用类似图 5-18 中的不同颜色的长方体来表示池化层和全连接层。除卷积层、池化层和全连接层（输入之前隐含 Flatten 层）之外的层，不改变网络结构，因此，一般只用这三层来表示神经网络的结构。画出代码 5-11 和代码 5-12 所示示例的神经网络结构如图 5-21 所示。

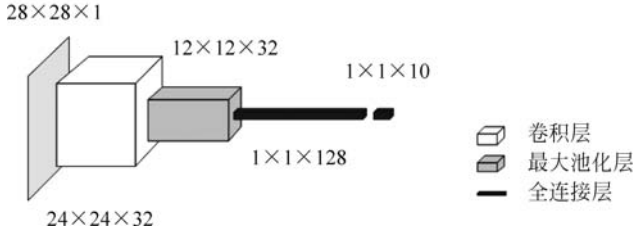


图 5-21　代码 5-11 和代码 5-12 示例的神经网络结构

5.5.4　批标准化层

批标准化（Batch Normalization）可以抑制梯度消散，加速神经网络训练。批标准化的提出者认为深度神经网络的训练之所以复杂，是因为在训练时每层的输入都随着前一层的参数的变化而变化。因此，在训练时，需要仔细调整步长和初始化参数来取得好的效果。

针对上述问题，在训练阶段，批标准化对每一层的批量输入数据 x 进行标准化操作

(见 7.1 节),使之尽量避免落入非线性激活函数的饱和区。具体来讲就是使之均值为 0,方差为 1。记每一批输入数据为$\mathcal{B}=\{\boldsymbol{x}_1,\boldsymbol{x}_2,\cdots,\boldsymbol{x}_m\}$,对其中任一 x_i 进行如下操作:

$$\begin{cases}\mu_B=\dfrac{1}{m}\displaystyle\sum_{i=1}^{m}\boldsymbol{x}_i\\ \sigma_B^2=\dfrac{1}{m}\displaystyle\sum_{i=1}^{m}(\boldsymbol{x}_i-\mu_B)^2\\ \hat{\boldsymbol{x}}_i=\dfrac{\boldsymbol{x}_i-\mu_B}{\sqrt{\sigma_B^2+\varepsilon}}\\ \boldsymbol{y}_i=\gamma_i\hat{\boldsymbol{x}}_i+\boldsymbol{\beta}_i\end{cases}\tag{5-50}$$

其中,ε 为防止分母为 0 的很小的常数。前三步分别为计算均值、计算方差、标准化,最后一步是对归一化后的结果进行缩放和平移,其中的 $\boldsymbol{\gamma}_i$ 和 $\boldsymbol{\beta}_i$ 是要学习的参数,它们都是 m 维的向量。μ_B 和 σ_B^2 是从输入数据中计算得到,是不需要学习的参数。

代码 5-11 和代码 5-12 所示的示例中,在 Dropout 层和 Flatten 层之间加入了批标准化层。对比是否加入该层的运行结果,可以发现在加入该层后,网络将更快收敛。读者可以自行验证。

5.5.5 典型卷积神经网络

在深度学习的发展过程中,出现了很多经典的卷积神经网络,它们对深度学习的学术研究和工业生产都起到了促进的作用,如 VGG、ResNet、Inception 和 DenseNet 等,很多实际使用的卷积神经都是在它们的基础上进行改进的。初学者应从试验开始,阅读论文和实现代码(MindSpore 框架中的 model_zoo[①] 和 TensorFlow 2 框架中的 keras.applications 包中包含了很多有影响力的神经网络模型的源代码)来全面了解它们。

下面简要讨论 VGG 卷积神经网络,并简要示例其应用。

VGG-16 是牛津大学的 Visual Geometry Group 在 2015 年发布的共 16 层的卷积神经网络,有约 1.38 亿个网络参数。该网络常被初学者用来学习和体验卷积神经网络。

VGG-16 模型是针对 ImageNet 挑战赛设计的,该挑战赛的数据集为 ILSVRC-2012 图像分类数据集。ILSVRC-2012 图像分类数据集的训练集有总共有 1281167 张图片,分为 1000 个类别,它的验证集有 50000 张图片样本,每个类别 50 个样本。

ILSVRC-2012 图像分类数据集是 2009 年开始创建的 ImageNet 图像数据集的一部分。基于该图像数据集举办了具有很大影响力的 ImageNet 挑战赛,很多新模型就是在该挑战赛上发布的。

VGG-16 模型的网络结构如图 5-22 所示,从左侧输入大小为 224×224×3 的彩色图片,在右侧输出该图片的分类。

输入层之后,先是 2 个大小为 3×3、卷积核数为 64、步长为 1、零填充的卷积层,此时的数据维度大小为 224×224×64,在水平方向被拉长了。

① https://gitee.com/mindspore/mindspore/tree/r1.1/model_zoo/official

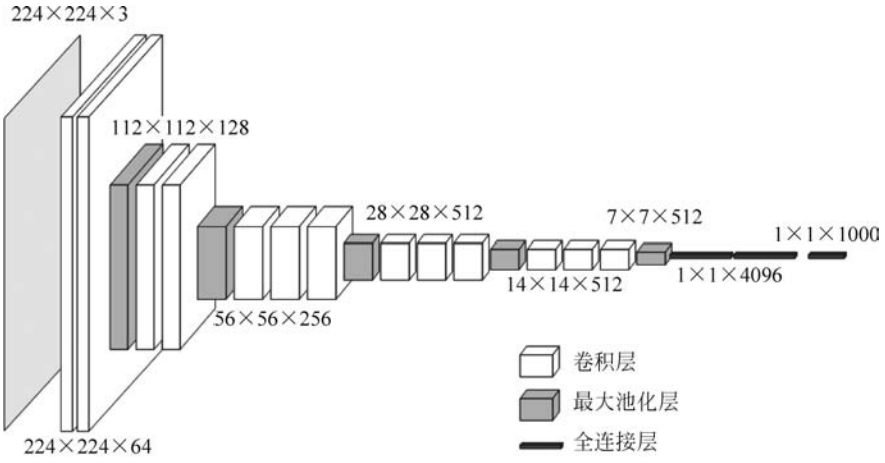


图 5-22　VGG-16 模型的网络结构

然后是 1 个大小为 2×2 的最大池化层，将数据的维度降为 112×112×64，再经过 2 个大小为 3×3、卷积核数为 128、步长为 1、零填充的卷积层，再一次在水平方向上被拉长，变为 112×112×128。

然后是 1 个大小为 2×2 的最大池化层，和 3 个大小为 3×3、卷积核数为 256、步长为 1、零填充的卷积层，数据维度变为 56×56×256。

然后是 1 个大小为 2×2 的最大池化层，和 3 个大小为 3×3、卷积核数为 512、步长为 1、零填充的卷积层，数据维度变为 28×28×512。

然后是 1 个大小为 2×2 的最大池化层，和 3 个大小为 3×3、卷积核数为 512、步长为 1、零填充的卷积层，数据维度变为 14×14×512。

然后是 1 个大小为 2×2 的最大池化层，数据维度变为 7×7×512。

然后是 1 个 Flatten 层将数据拉平。

最后是 3 个全连接层，节点数分别为 4096、4096 和 1000。

除最后一层全连接层采用 Softmax 激活函数外，所有卷积层和全连接层都采用 ReLU 激活函数。

从上面网络结构可见，经过卷积层，通道数量不断增加，而经过池化层，数据的高度和宽度不断减少。

Visual Geometry Group 后又发布了 19 层的 VGG-19 模型。

MindSpore 和 TensorFlow 2 实现了 VGG-16 模型和 VGG-19 模型[①②]，建议读者仔细阅读并分析。TensorFlow 2 还提供了用 ILSVRC-2012-CLS 图像分类数据集预先训练好的 VGG-16 和 VGG-19 模型，下面给出一个用预先训练好的模型来识别一幅图片(图 5-23)的例子。

① https://gitee.com/mindspore/mindspore/blob/master/model_zoo/official/cv/vgg16/README.md

② https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/keras/applications/vgg16.py

图 5-23　试验用小狗图片

例子代码见代码 5-15。

代码 5-15　VGG-19 预训练模型应用(vgg19_app.ipynb)

```
import tensorflow.keras.applications.vgg19 as vgg19
import tensorflow.keras.preprocessing.image as imagepre

# 加载预训练模型
model = vgg19.VGG19(weights='E:\\MLDatas\\vgg19_weights_tf_dim_ordering_tf_kernels.h5', include_top=True)                    # 加载预先下载的模型
# 加载图片并转换为合适的数据形式
image = imagepre.load_img('116.jpg', target_size=(224, 224))
imagedata = imagepre.img_to_array(image)
imagedata = imagedata.reshape((1,) + imagedata.shape)

imagedata = vgg19.preprocess_input(imagedata)
prediction = model.predict(imagedata)       # 分类预测
results = vgg19.decode_predictions(prediction, top=3)
print(results)
#[[('n02113624', 'toy_poodle', 0.6034094), ('n02113712', 'miniature_poodle', 0.34426507), ('n02113799', 'standard_poodle', 0.0124355545)]]
```

可见,图片为 toy poodle(玩具贵宾犬)的概率最大,约为 0.6。

5.6　习题

1. 下表为某二分类器预测结果的混淆矩阵,试计算准确率、平均准确率、精确率、召回率和 F_1-score。

	预测为“0”的样本数	预测为“1”的样本数
标签为“0”的样本数	1026	1101
标签为“1”的样本数	1007	911026

2. 与 MNIST 手写体数字集一样，CIFAR-10 包含了 60 000 张图片，共 10 类。训练集 50 000 张，测试集 10 000 张。但与 MNIST 不同的是，CIFAR-10 数据集中的图片是彩色的，每张图片的大小是 $32\times32\times3$，3 代表 R/G/B 三个通道，每个像素点的颜色由 R/G/B 三个值决定，R/G/B 的取值范围为 0～255。仿照 MNIST 手写体数字识别，用 MindSpore 框架或 TensorFlow 2.0 框架实现卷积神经网络对 CIFAR-10 进行分类试验。

3. 试计算代码 5-11 和代码 5-12 所示例的卷积神经网络中各层需要学习的参数数量。

4. 在 5.4.1 节的误差反向传播学习示例中，计算第 2 个训练样本(0,1)的前向传播过程。网络参数的初值与示例初值相同：$\boldsymbol{W}_1=\begin{bmatrix}0.1 & 0.2\\ 0.2 & 0.3\end{bmatrix}$，$\boldsymbol{\theta}_1=[0.3 \quad 0.3]$，$\boldsymbol{W}_2=\begin{bmatrix}0.4 & 0.5\\ 0.4 & 0.5\end{bmatrix}$，$\boldsymbol{\theta}_2=[0.6 \quad 0.6]$。

5. 接第 4 题，再计算反向传播学习过程中 $\boldsymbol{w}_1^{(1,2)}$ 的更新。

6. 在单通道数据上进行卷积运算，待处理张量 $\boldsymbol{I}$ 和卷积核 $\boldsymbol{K}$ 分别如下，请计算在卷积核移动步长为 1 的输出张量 $\boldsymbol{S}$。阈值 $\theta=0$。

待处理张量：

19	11	2	0	8
3	22	0	9	1
58	4	23	11	15
7	1	0	9	10
0	0	1	8	2

卷积核：

1	2
0	1

7. 接第 6 题，如果在边缘采用 0 填充，请计算输出张量 $\boldsymbol{S}$。