第5章 MNIST数据集及神经网络

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 课程名称 | 人工智能基础与应用 | | 章名 | | | MNIST数据集及神经网络 | |
| 教学内容 | MNIST数据集及神经网络2 | | | | 课时 | 2 | |
| 项目性质 | □演示性 □验证性 □设计性 √综合性 | | | | | | |
| 授课班级 |  | 授课日期 | |  | | 授课地点 |  |
| 教学目标 | 掌握经典卷积神经网络  掌握循环神经网络  了解优化器及优化方法 | | | | | | |
| 教学内容 | 1. 经典卷积神经网络 2. 循环神经网络 3. 优化器及优化方法 | | | | | | |
| 教学重点 | 经典卷积神经网络、循环神经网络 | | | | | | |
| 教学难点 | 经典卷积神经网络、循环神经网络 | | | | | | |
| 教学准备 | 装有Python的计算机  教学课件PPT  教材：《人工智能基础与应用（微课版）》 | | | | | | |
| 作业设计 |  | | | | | | |

教学过程

|  |  |
| --- | --- |
| **教学环节** | **教学内容与过程**  **（教学内容、教学方法、组织形式、教学手段）** |
| **课前组织** | 做好上课前的各项准备工作（打开计算机、打开课件、打开软件、打开授课计划、教案等），吸引学生注意力。 |
| **课程说明** | 【课前说明】  回顾MNIST数据集的基本概念和神经元常用函数、深度神经网络等知识点。  【目的】  使学生从了解本节课的学习目标、学习重点、考评方式等方面明确课程学习的要求和目标。 |
| **课程内容描述** | 5.4 经典卷积神经网络介绍  卷积神经网络（Convolutional Neural Networks，CNN）是一类深度神经网络，最常用于分析视觉图像。卷积神经网络是深度神经网络的正则化版本。在卷积神经网络中，至少在一个层中使用卷积运算代替矩阵乘法运算。卷积是一种特殊的线性操作。常规的神经网络在处理稍微大一些的图像时，效果并不尽如人意，如输入的图像是28像素×28像素×1，在第一个隐藏层中，每一个单独的全连接神经元有28×28=784个权重，但是如果输入的是100像素×100像素×3的图像，那么一个神经元就有30000个权重，而且在一个神经网络中不会只有一个神经元。全连接神经网络的“完全连接”使它们容易过度拟合数据，正则可以向损失函数添加某种形式的权重值测量，使得权重更加低阶，阻止过拟合。  一个简单的卷积神经网络是由各种层按一定顺序排列的。卷积神经网络主要由卷积层（Convolutional Layers）、池化层（Pooling Layers）、全连接层（Fully Connected Layers，FC Layers）构成。将这些层按一定的顺序排列，就可以搭建一个卷积神经网络。  5.4.1 LeNet**-**5模型及其实现  LeNet-5模型是由杨立昆（Yann LeCun）教授于1998年在论文*Gradient-Based Learning Applied to Document Recognition*中提出的，是一种用于手写体字符识别的非常高效的卷积神经网络。网络实现过程如图所示。  5-7  网络实现过程  LeNet-5是一个7层的网络，当然，其名称中的5可以理解为整个网络中含可训练参数的层数。在MNIST数据集上，LeNet-5模型可以达到99.2%的正确率。  卷积层是一个将卷积运算和加法运算组合在一起的隐藏层，在图像识别里提到的卷积是二维卷积，即离散二维滤波器（也称作卷积核）与二维图像做卷积操作，简单地讲是二维滤波器滑动到二维图像上的所有位置，并在每个位置上与该像素点及其领域像素点做内积。卷积运算过程如图所示。  5-8  卷积运算过程  在上图中，实现了3×3的卷积核对5×5的输入数据的卷积运算。输入数据中，3×3的灰色区域为卷积核，卷积核的移动步长为1，即每次卷积核向右或者向下移动一格，在输入数据左上角的3×3区域中，计算过程为1×1+1×0+1×1+0×0+1×1+1×0+0×1+0×0+1×1=4，当卷积核移动时，在输出的3×3区域中就会填充一个空格。  1．LeNet-5实现的过程  （1） 输入层。输入图像的原始像素，大小为32×32×1，即长、宽为32像素，1通道（Channels），即32×32的灰度照片。  （2）卷积层（第一层）。与输入层相连，接收32×32×1的矩阵数据，卷积核（Convolutions）大小为5×5×6，大小为5，深度为6，填充方式为“VALID”，卷积步长为1，所以根据公式（其中，*w*为输入尺寸；*k*为卷积核尺寸；*p*为填充数量，填充方式为“VALID”时不填充，填充方式为“SAME”时使用全0填充；*s*为卷积步长），可得这一层的输出尺寸为，深度为6，本层的输出矩阵大小为28×28×6。  （3）池化层（第二层）。前接第一个卷积层，接收28×28×6的矩阵数据，卷积核大小为2×2，即长和宽都是2，步长为2，填充方式为“SAME”。在原始的LeNet-5中，采用的是平均值池化，本层的输出矩阵大小为14×14×6。  （4）卷积层（第三层）。前接第一个池化层，接收14×14×6的矩阵数据，卷积核大小为5×5，深度为16，填充方式为“VALID”，步长为1。本层的输出矩阵大小为10×10×16。  （5）池化层（第四层）。前接第二个卷积层，接收10×10×16的矩阵数据，卷积核大小为2×2，即长和宽都是2，步长为2，填充方式为“SAME”。本层的输出矩阵大小为5×5×16。  （6）全连接层（第五层）。前接第二个池化层，接收5×5×16的矩阵数据，在LeNet-5原始论文中称它为卷积层，但是由于卷积核的大小也是5×5，所以其和全连接层没有区别，本层的输出节点是120个。  （7）全连接层（第六层）。前接第一个全连接层，接收120个节点输入，本层的输出节点是84个。  （8）全连接层，输出层（第七层）。前接第二个全连接层，接收84个节点输入，本层的输出节点是10个。  2．欠拟合  欠拟合与过拟合一直是机器学习训练过程中的难题，在训练模型的过程中往往要对二者进行权衡，使得模型在训练数据集和测试数据集上也有很好的表现。  欠拟合（Under Fitting）是由于特征维度过少，模型过于简单，使神经网络没办法完全满足数据集的特征提取要求，体现在训练以及预测时表现不佳、成功率低。欠拟合解决方法介绍如下。  （1）将模型复杂化。可以将原算法复杂化，例如，在神经网络中增加隐藏层和隐藏单元，在回归模型中增加更多的高次项。可以更换算法，使用更加复杂的算法代替现有算法，例如，使用神经网络代替线性回归。  （2）增加更多特征，使输入数据的特征更明显。  （3）调整超参数。包括神经网络中的学习率、学习衰减率、神经网络隐藏层数、隐藏层的单元数等，以及其他算法中的正则化参数等。  （4）减弱正则化约束或者去掉正则化约束。  3．过拟合  过拟合是指模型在训练集上表现很好，但在验证和测试阶段效果比较差，即模型的泛化能力很差。过拟合的解决方法如下。  （1）增加训练数据量。发生过拟合最常见的原因就是数据量太少或者模型太复杂，增加数据量可以缓解该问题，如在图像识别时，增加训练数据集的图像数量可以降低过拟合的风险。如果数据获取比较困难，可以将现有数据集上的图像进行旋转、拉伸等操作，从而实现数据集扩展。  （2）减少数据特征，去掉数据中非共性的特征。  （3）调整超参数。  （4）使用正则化约束或者增强正则化约束。  （5）降低模型的复杂度。  （6）使用Dropout。Dropout只适用于神经网络，按照一定的比例失活隐藏层的神经元，使得神经网络更简单。  （7）Early Stopping，即提前结束训练。在训练模型的过程中，如果训练误差一直在降低，但是验证误差却不再降低甚至上升，这时候便可以结束模型训练。  4．卷积神经网络的TensorFlow实现  通过TensorFlow框架实现一个类似于LeNet-5的神经网络，来解决MNIST数据集上的数字识别问题。  本网络与原LeNet-5的区别是：卷积核的个数不同；激活函数不同，此处用的是ReLU；输出层选择softmax()函数。二者的整体过程是一致的。  在TensorFlow目录下新建目录MNIST\_data，将MNIST数据集的4个压缩包放在MNIST\_data目录下，虽然程序会自己下载该数据集，但是有时会出现下载不下来的情况。  【例5-1】 在TensorFlow目录下新建文件，命名为LeNet-5.py，利用TensorFlow解决类似于LeNet-5在MNIST数据集上进行数字识别的问题，在PyCharm中编写以下代码。  # -×- coding: utf-8 -×-  # 载入MINIST数据需要的库  from tensorflow.examples.tutorials.mnist import input\_data  # 保存模型需要的库  from tensorflow.python.framework.graph\_util import convert\_variables\_to\_constants  from tensorflow.python.framework import graph\_util  # 导入其他库  import tensorflow as tf  import time  import os  os.environ['TF\_CPP\_MIN\_LOG\_LEVEL'] = '2'  # 获取MINIST数据  mnist = input\_data.read\_data\_sets("./MNIST\_data", one\_hot=True)  # 占位符  # x是特征值，也就是像素  # 使用一个28×28=784列的数据来表示一个图像的构成  # 每一个点都是这个图像的一个特征  # 因为每一个点都会对图像的外观和表达的含义有影响，只是影响的大小不同而已  x = tf.placeholder("float", shape=[None, 784], name="Mul") # 输入28×28=784  # y\_是真实数据[0,0,0,0,1,0,0,0,0]，为4  y\_ = tf.placeholder("float", shape=[None, 10], name="y\_") # 输出  # 变量 784×10的矩阵  # W表示每一个特征值（像素点）影响结果的权重  # 这个值很重要，因为深度学习的过程就是发现特征  # 经过一系列训练，得出每一个特征值对结果影响的权重  # 训练就是为了得到这个最佳权重值  W = tf.Variable(tf.zeros([784, 10]), name='x')  b = tf.Variable(tf.zeros([10]), 'y\_')  # 权重  def weight\_variable(shape):  # 生成的值服从具有指定平均值和标准偏差的正态分布  # 如果生成的值大于平均值的两个标准偏差的值，则丢弃重新选择  initial = tf.truncated\_normal(shape, stddev=0.1) # 标准差为0.1  return tf.Variable(initial)  # 偏差  def bias\_variable(shape):  initial = tf.constant(0.1, shape=shape)  return tf.Variable(initial)  # 卷积  def conv2d(x, W):  # 参数x指需要做卷积的输入图像，要求它是一个Tensor  # 具有[batch, in\_height, in\_width, in\_channels]这样的shape  # 具体含义是[训练时一个batch的图像数量, 图像高度, 图像宽度, 图像通道数]  # 注意这是一个4维的Tensor，batch和in\_channels在卷积层中通常设为1  # 参数W相当于CNN中的卷积核，要求它是一个Tensor  # 具有[filter\_height, filter\_width, in\_channels, out\_channels]这样的shape  # 具体含义是[卷积核的高度，卷积核的宽度，图像通道数，卷积核个数]  # 注意，第三维in\_channels就是参数x的第四维  return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='VALID')  # 参数strides：卷积时在图像每一维的步长，这是一个一维的向量，长度为4  # 参数padding：string类型的量，只能是“VALID”，不补零  # 最大池化  def max\_pool\_2x2(x):  # x：input  # ksize：filter，滤波器大小为2×2  # strides：步长，2×2，表示filter窗口每次水平移动两格，每次垂直移动两格  # padding：填充方式，补零  return tf.nn.max\_pool(x, ksize=[1, 2, 2, 1],  strides=[1, 2, 2, 1], padding='SAME')  # 第一层卷积  # 权重+偏置+激活+池化  # patch为5×5；in\_size为1，即图像的厚度，如果是彩色的，则为3；32个卷积核（滤波器）  W\_conv1 = weight\_variable([5, 5, 1, 32])  b\_conv1 = bias\_variable([32])  # 对数据进行重新排列，形成图像  x\_image = tf.reshape(x, [-1, 28, 28, 1])  # print("x",x)  # print("x\_image",x\_image)  # ReLU操作，输出大小为28×28×32  h\_conv1 = tf.nn.relu(conv2d(x\_image, W\_conv1) + b\_conv1)  # Pooling操作，输出大小为14×14×32  h\_pool1 = max\_pool\_2x2(h\_conv1)  # 第二层卷积  # 权重+偏置+激活+池化  # patch为5×5；in\_size为32，即图像的厚度；out\_size是64，即输出的大小  W\_conv2 = weight\_variable([5, 5, 32, 64])  b\_conv2 = bias\_variable([64])  # ReLU操作，输出大小为14×14×64  h\_conv2 = tf.nn.relu(conv2d(h\_pool1, W\_conv2) + b\_conv2)  # Pooling操作，输出大小为7×7×64  h\_pool2 = max\_pool\_2x2(h\_conv2)  # 全连接一  W\_fc1 = weight\_variable([7 × 7 × 64, 1024])  b\_fc1 = bias\_variable([1024])  # 全连接二  W\_fc2 = weight\_variable([1024, 10])  b\_fc2 = bias\_variable([10])  # 输入数据变换  # 变换为m×n，列n为7×7×64  h\_pool2\_flat = tf.reshape(h\_pool2, [-1, 7 × 7 × 64])  # 进行全连接操作  # tf.nn.relu()函数可将大于0的数保持不变，将小于0的数置为0  h\_fc1 = tf.nn.relu(tf.matmul(h\_pool2\_flat, W\_fc1) + b\_fc1)  # Dropout可防止过拟合，它一般用在全连接层，训练用，测试不用  # Dropout就是在不同的训练过程中随机扔掉一部分神经元  # Dropout可以让某个神经元的激活值以一定的概率p停止工作  # 参数keep\_prob：设置神经元被选中的概率，在初始化时keep\_prob是一个占位符  # TensorFlow在运行时设置keep\_prob具体的值，如keep\_prob: 0.5  keep\_prob = tf.placeholder("float", name='rob')  h\_fc1\_drop = tf.nn.dropout(h\_fc1, keep\_prob)  # 用于训练的softmax()函数将所有数据归一化到0～1之间，大的数据特征更明显  y\_conv = tf.nn.softmax(tf.matmul(h\_fc1\_drop, W\_fc2) + b\_fc2, name='res')  # 训练完成后，进行测试用的softmax()函数  y\_conv2 = tf.nn.softmax(tf.matmul(h\_fc1, W\_fc2) + b\_fc2, name="final\_result")  # 交叉熵的计算，返回包含了损失值/误差的Tensor  # 熵是衡量事物混乱程度的一个值  cross\_entropy = -tf.reduce\_sum(y\_ × tf.log(y\_conv))  # 优化器，负责最小化交叉熵  train\_step = tf.train.AdamOptimizer(1e-4).minimize(cross\_entropy)  # tf.argmax()：取出该数组最大值的下角标  correct\_prediction = tf.equal(tf.argmax(y\_conv, 1), tf.argmax(y\_, 1))  # 计算准确率  accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, "float"))  # 创建会话  with tf.Session() as sess:  time\_begin = time.time()  # 初始化所有变量  sess.run(tf.global\_variables\_initializer())  # print(sess.run(W\_conv1))  # 保存输入/输出，可以在之后用  tf.add\_to\_collection('res', y\_conv)  tf.add\_to\_collection('output', y\_conv2)  tf.add\_to\_collection('x', x)  # 训练开始  for i in range(10000):  # 取出MNIST数据集中的50个数据  batch = mnist.train.next\_batch(50)  # run()可以看作输入相关值到函数中的占位符，然后计算出结果  # 这里将batch[0]给x，将batch[1]给y\_  # 执行训练过程并传入真实数据  train\_step.run(feed\_dict={x: batch[0], y\_: batch[1], keep\_prob: 0.5})  if i % 100 == 0:  train\_accuracy = accuracy.eval(feed\_dict={x: batch[0],\  y\_: batch[1],  keep\_prob: 1.0})  print("step %d, training accuracy %g" % (i, train\_accuracy))  time\_elapsed = time.time() - time\_begin  print("训练所用时间：%d秒" % time\_elapsed)  # 用saver 保存模型  saver = tf.train.Saver()  saver.save(sess, "model\_data/model")  执行该程序，如果计算机安装的是CPU版本的TensorFlow，建议将训练步数调少，否则会训练非常久。训练完成后的准确率如图所示。    训练完成后的准确率  生成的模型在TensorFlow目录下的model\_data目录中，模型文件如图所示。    模型文件  LeNet-5模型的结构比较清晰，即输入层→卷积层（池化层）→全连接层→输出层，但是有些网络中是没有池化层的。除了LeNet-5模型外，2012年的AlexNet模型、2014年的VGGNet模型都是这样的结构，卷积层的卷积核大小一般不会超过5，且大多是奇数，卷积核的深度一般都是递增的，池化层的核大小一般是2或者3。  5.4.2 AlexNet介绍  AlexNet模型是2012年大规模视觉识别挑战赛中的冠军模型，AlexNet将LeNet的思想发扬光大，把CNN的基本原理应用到了更深、更宽的网络结构中。  另外，AlexNet模型首次在CNN中采取了ReLU激活函数、Dropout防止过拟合、GPU加速训练等技术。  5.4.3 VGGNet介绍  牛津大学计算机视觉组（Visual Geometry Group，VGG）是牛津大学在2014年大规模视觉识别挑战赛中提出的模型，该模型相对于以往模型，进一步加宽和加深了网络结构，它的核心是5组卷积操作，每两组之间做Max-Pooling空间降维。同一组内采用多次连续的3×3卷积，卷积核的数目由较浅组的64增多到最深组的512，同一组内的卷积核数目是一样的。卷积之后接两层全连接层，之后是分类层。每组内的卷积层数不同，有11、13、16、19层这几种模型。16层的VGGNet网络结构如图所示。  5-11  16层的VGGNet网络结构  5.4.4 Inception模型及其实现  Inception模型的出现改变了CNN的发展趋势。在Inception模型出现之前，CNN经历了LeNet-5、AlexNet、VGGNet等模型，但是随着网络的发展，为了追求更好的性能，网络越来越深，参数越来越多，且全连接层的参数量占据了很大的比重，如AlexNet模型参数有6000万个左右，最后的全连接层就有3800万个左右的参数，占了总参数数量的一半多。第一次提出Inception模型的GoogLeNet网络的最后并没有全连接层，而是采用了全局平均值池化，所以参数的数量大大减少。  1．Inception模型介绍  Inception模型的思想和之前的卷积思想不同，LeNet-5模型是将不同的卷积层通过串联连接起来，但是Inception模型是通过串联+并联的方式将卷积层连接起来的。Inception模型是对输入图像并行地执行多个卷积运算或池化操作，并将所有输出结果拼接为一个非常深的特征图，且不同大小卷积核的卷积运算可以得到图像中的不同信息，处理获取到的图像中的不同信息可以得到更好的图像特征。  Inception模型的主要成员包括Inception v1、Inception v2、Inception v3、Inception v4和Inception-ResNet。  下图给出了Inception模块的一个单元结构示意图。  5-12  Inception模块的一个单元结构示意图  对于输入矩阵，分别经过1×1卷积核、3×3卷积核、5×5卷积核以及3×3的最大值池化后，由于采用的padding方式为“SAME”，也就是使用全0填充且步长为1，所以输出矩阵的长宽与输入矩阵相同，然后将4个输出矩阵纵向拼接在一起，以得到一个更深的矩阵。  由于一个Inception模块要进行多次运算，所以需要耗费大量的计算资源。为了实现降维，降低运算成本，在3×3、5×5的卷积运算前，在最大值池化的运算后，加入1×1的卷积核。  下图所示为降维的Inception模块的一个单元结构示意图。  5-13  降维的Inception模块的一个单元结构示意图  例如，输入矩阵大小为50×50×64，经过128个5×5、padding方式为“SAME”、步长为1的卷积后，根据公式，输出矩阵大小是50×50×128，运算的参数量为5×5×64×128= 204800。如果在之前加入32个1×1的卷积，再通过128个5×5的卷积，那么运算的参数量为1×1×64×32+5×5×32×128=104448，参数量大概是前一种方式的一半，而且这个差距还会随着深度的加深变大。  2．GoogLeNet实现的过程  这里以Inception模块构建的GoogLeNet（Inception v1）为例，描述GoogLeNet的实现过程。  （1）输入层  原始输入图像大小为224×224×3，为三通道RGB图像。  （2）卷积层  接收224×224×3的矩阵数据，与64个大小为7×7的卷积核（步长为2，padding为“SAME”）做运算，输出为112×112×64，经过3×3的最大值池化（步长为2），输出矩阵为56×56×64。  （3）卷积层  接收56×56×64的矩阵数据，与192个大小为3×3的卷积核（步长为1，padding为“SAME”）做运算，输出为56×56×192，经过3×3的最大值池化（步长为2），输出矩阵为28×28×192。  （4）Inception 3a层  接收28×28×192的矩阵数据，共4个分支，采用不同尺度的卷积核运算，4个分支步长都为1。  第一个分支：与64个大小为1×1的卷积核做运算，输出矩阵为28×28×64。  第二个分支：与96个大小为1×1的卷积核做运算，输出为28×28×96，再与128个大小为3×3的卷积核（padding为“SAME”）做运算，输出矩阵为28×28×128。  第三个分支：与16个大小为1×1的卷积核做运算，输出为28×28×16，再与32个大小为5×5的卷积核（padding为“SAME”）做运算，输出矩阵为28×28×32。  第四个分支：3×3的最大值池化（padding为“SAME”），输出为28×28×192，32个大小为1×1的卷积核，输出矩阵为28×28×32。  将4个分支进行纵向拼接，层数为64+128+32+32=256，所以输出矩阵为28×28×256。  （5）之后的层  以后的层数都以此类推，GoogLeNet模型有9个堆叠的Inception模块，共有22层（如果包括池化层，则是27层）。由于GoogLeNet模型层数比较多，会造成梯度消失的问题，所以，为了阻止梯度消失，在网络中加入了两个辅助分类器。辅助分类器对网络中的两个Inception模块进行了softmax操作，得到的这两个Inception模块的损失，称为辅助损失。总的损失等于最后得到的损失与辅助损失之和，辅助损失仅在训练时使用，在预测过程中不使用。  除了GoogLeNet实现的Inception v1外，还有其他Inception版本，每一个Inception版本的发布都带来了准确率的提升。  5.5 循环神经网络  循环神经网络（Recurrent Neural Network，RNN）是一类用于处理序列数据的神经网络。  5.5.1 循环神经网络简介  时间序列数据是序列数据中最常见的一种。时间序列数据（Time Series Data）是在不同时间上收集到的数据，用于描述现象随时间变化的情况。这类数据反映了某一事物、现象等随时间的变化状态或程度。  在全连接的神经网络以及卷积神经网络中有输入层、隐藏层、输出层，层与层之间通过学习到的权重进行连接，在同一层中，节点与节点之间是不连接的。如果现在需要解决一个问题，如“打雷了，可能要\_\_了”，在横线上填一个词，那么这个词很大概率是“下雨”，但是如果孤立地理解这句话中每个词的意思，则神经网络并不能知道要填入什么，所以需要将整句话连接成整个序列进行理解。循环神经网络可以找到当前序列的输出与之前序列的关系，也就是说，循环神经网络会记录之前的信息。在隐藏层中，每层内的节点都是有连接的，隐藏层的输入不仅包括输入层的输出，还包括上一时刻隐藏层的输入。  1．RNN基本结构  先了解一个节点前后的运算，一个节点的输入为*x*，经过运算得到输出*y*，节点网络如图所示。  5-14  节点网络  如果输入的数据*x*是有序列的，如翻译问题，则每一个输入的*x*表示一个汉字，在一个完整的句子中就有很多*x*。  在RNN中，引入了隐状态*h*（hidden state），隐状态可以对序列数据进行特征提取，对输入进行运算，其公式为：，其中，***U***是从输入层到隐藏层的权重矩阵。  可以得到输入经过隐状态后的输出值，如图所示。  5-15  引入隐状态  经过隐状态后，采用softmax()函数通过得到输出*y*，一个完整的RNN结构可以表示成图所示的形式。  5-16  一个完整的RNN结构  从RNN基本结构可以看出，RNN的输入和输出是等长的。上图中有4个序列，但实际上序列数是不定的，假设有不定个序列，可以组成RNN不展开表达样式，如图所示，其中*A*为某特殊序列。  5-17  RNN不展开表达样式  2．RNN结构分析  RNN常用的结构有3种，分别是Vector-to-Sequence结构、Sequence-to-Vector结构、Encoder-Decoder结构。  （1）Vector-to-Sequence结构  假设一个问题的输入是单独的值，输出是一个序列，则可以建立Vector-to-Sequence结构模型，将输入放到某一个序列进行计算（如图所示）。  5-18  将输入放到某一个序列进行计算  也可以将输入放到全部序列中进行计算（如图所示）。  5-19  将输入放到全部序列中进行计算  （2）Sequence-to-Vector结构  假设一个问题的输入是一个序列，输出是一个单独的值，一般会在最后一个序列上进行输出变换，可以建立Sequence-to-Vector结构，如图所示。  5-20  Sequence-to-Vector结构  （3）Encoder-Decoder结构  有时也会将Encoder-Decoder结构称为Sequence-to-Sequence结构，该结构的具体过程就是编码及解码。  首先将输入的数据编码成一个上下文向量***c***，这个过程称为Encoder，得到***c***后，用另一个RNN网络进行解码，这个过程称为Decoder。  ***c***作为新的RNN的*h*0时，结构如图所示。  5-21  ***c***作为新的RNN的***h***0  ***c***作为新的输入时，结构如图所示。  5-22  ***c***作为新的输入  3．长短期记忆网络结构  RNN可以做到在时间序列上记忆，但是对于时间序列上较远的点，记起来比较困难，因为两个节点距离较远时，会涉及多次的雅可比矩阵相乘，导致梯度消失或者梯度膨胀，长短期记忆网络（Long Short-Term Memory，LSTM）结构可以很好地解决这个问题。  在标准的RNN结构中，会重复一些简单的结构，如tanh层，简化后的RNN标准模型如图所示。  5-23  简化后的RNN标准模型  虽然LSTM与RNN的大体结构相同，但是LSTM在重复模块中拥有一个不同的结构，LSTM结构如图所示。  5-24  LSTM结构  重复模块中的每一条线都传输一个向量，重复模块中的方框代表学习到的神经网络层，重复模块中的圆圈代表pointwise的操作，如相加和相乘等操作。  LSTM靠门结构有选择性地处理信息，每一个门包含一个sigmoid神经网络层和一个pointwise乘法操作。LSTM共拥有3个门，分别是遗忘门、输入门和输出门，用来保护和控制状态。遗忘门将状态中的信息选择性地遗忘，输入门将新的信息选择性地记录下来，输出门确定输出什么值。  5.5.2 循环神经网络实现  使用TensorFlow，LSTM能够很容易地实现。  【例5-2】 在TensorFlow目录下新建文件，命名为LSTM.py，利用TensorFlow解决类似于LSTM在MNIST数据集上进行数字识别的问题，在PyCharm中编写以下代码。  import tensorflow as tf  from tensorflow.contrib import rnn  import os  os.environ['TF\_CPP\_MIN\_LOG\_LEVEL'] = '2'  # 导入MNIST数据集  from tensorflow.examples.tutorials.mnist import input\_data  mnist = input\_data.read\_data\_sets("./MNIST\_data/", one\_hot=True)  # 设置全局变量  learning\_rate = 0.001  training\_steps = 10000  batch\_size = 128  display\_step = 200  num\_input = 28 # 输入向量的维度  timesteps = 28 # 循环层长度  num\_hidden = 128 # 隐藏层的特征数  num\_classes = 10 # 0～9  # tf Graph 输入  X = tf.placeholder("float", [None, timesteps, num\_input])  Y = tf.placeholder("float", [None, num\_classes])  # 定义权重和偏置  weights = {  'out': tf.Variable(tf.random\_normal([num\_hidden, num\_classes]))  }  biases = {  'out': tf.Variable(tf.random\_normal([num\_classes]))  }  def RNN(x, weights, biases):  x = tf.unstack(x, timesteps, 1)  # 初始的biases=1，不希望遗忘任何信息  lstm\_cell = rnn.BasicLSTMCell(num\_hidden, forget\_bias=1.0)  outputs, states = rnn.static\_rnn(lstm\_cell, x, dtype=tf.float32)  # 选择最后一个output与输出的全连接weights相乘，再加上biases  return tf.matmul(outputs[-1], weights['out']) + biases['out']  logits = RNN(X, weights, biases)  prediction = tf.nn.softmax(logits)  # 定义损失和优化  loss\_op = tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits(logits=logits, labels=Y))  optimizer = tf.train.GradientDescentOptimizer(learning\_rate=learning\_rate)  train\_op = optimizer.minimize(loss\_op)  correct\_pred = tf.equal(tf.argmax(prediction, 1), tf.argmax(Y, 1))  accuracy = tf.reduce\_mean(tf.cast(correct\_pred, tf.float32))  init = tf.global\_variables\_initializer()  with tf.Session() as sess:  sess.run(init)  for step in range(1, training\_steps+1):  # 随机抽出这一次迭代训练时用的数据  batch\_x, batch\_y = mnist.train.next\_batch(batch\_size)  # 对数据进行处理，使得其符合输入  batch\_x = batch\_x.reshape((batch\_size, timesteps, num\_input))  # 迭代  sess.run(train\_op, feed\_dict={X: batch\_x, Y: batch\_y})  if step % display\_step == 0 or step == 1:  # 计算损失  loss, acc = sess.run([loss\_op, accuracy], feed\_dict={X: batch\_x,  Y: batch\_y})  print("Step " + str(step) + ", Minibatch Loss= " + \  "{:.4f}".format(loss) + ", Training Accuracy= " + \  "{:.3f}".format(acc))  print("优化完成!")  # 计算128个测试的准确率  test\_len = 128  test\_data = mnist.test.images[:test\_len].reshape((-1, timesteps, num\_input))  test\_label = mnist.test.labels[:test\_len]  print("测试准确率:", sess.run(accuracy, feed\_dict={X: test\_data, Y: test\_ label}))  通过这段代码，读者可以很快了解LSTM实现的过程。循环神经网络还包括双向循环神经网络以及深层循环神经网络，本书对其不做过多介绍。  5.6 优化器及优化方法  机器学习训练过程的本质就是在最小化损失，而在定义了损失函数后，优化器就派上了用场。  5.6.1 优化方法  4.1.4小节将梯度下降法类比为下山，这里用一个具体的实例介绍梯度下降法是如何工作的。  假如有一个实例，它的输入为*x*，需要使用梯度下降法来优化，这个实例的损失函数为，初始值为。对于该函数，需要找到*J*的最小值。要找到*J*的最小值，首先需要确定优化方向，也就是梯度的方向，每次优化的幅度就是学习率，那么更新公式为。对于该损失函数，，初始化的时候，设置=1，学习率为0.2，那么根据公式进行计算后，5次迭代结果如表所示。  5次迭代结果   |  |  |  | | --- | --- | --- | | 迭代次数 | 参数值 | 更新后的参数值 | | 1 | 1 | 1-2×1×0.2=0.6 | | 2 | 0.6 | 0.6-2×0.6×0.2=0.36 | | 3 | 0.36 | 0.36-2×0.36×0.2=0.216 | | 4 | 0.216 | 0.216-2×0.216×0.2=0.1296 | | 5 | 0.1296 | 0.1296-2×0.1296×0.2=0.07776 |   经过5次优化后，所得的结果已经比较小了。但是梯度下降法有一些问题：首先，如果损失函数不是一个凸函数，梯度下降并不一定能给出全局最优解，可能仅仅是局部最优解；其次，对全部数据的损失函数进行梯度下降是非常耗费时间的。一般为了加快速度，会选择随机梯度下降（Stochastic Gradient Descent），随机梯度下降不是对所有训练数据集的损失函数进行训练，而是训练某一个数据的损失函数，但是这一个数据损失函数的下降并不能很好地代表整体下降，所以要采用一个办法，就是引入一个批次（batch）训练，每次取训练数据集的一部分构造损失函数进行梯度下降，进而优化损失函数。  5.6.2 学习率设置  从5.6.1小节可以了解到学习率在梯度下降时的作用，学习率决定了每次参数更新的幅度。学习率太大，最优点的位置会在极值点左右摆动，导致找不到最优解；学习率太小，找最优解的过程又太漫长，所以设置一个合适的学习率至关重要。  一般在训练过程中，学习率并不是一直不变的，而是随着训练的过程而逐渐减小的。例如，需要根据全局的学习步数，通过指数衰减实现学习率的减小。在本书中，使用卷积神经网络进行手写数字识别时使用的tf.train.AdamOptimizer()函数，就是利用梯度的一阶矩估计和二阶矩估计动态调整每个参数的学习率。 |
| **总结评价** | 本节课主要介绍了卷积神经网络和循环神经网络，并简要介绍了优化器及优化方法。 |