卷积神经网络（CNN）实现手写数字识别

一、 目的描述

准备 MNIST 数据。安装运行 TensorFlow，最终实现用 TensorFlow 利用卷积神经网络（CNN）训练测试手写的数字识别模型。

二、 方法原理

2.1 MNIST 数据集

MNIST 数据集的训练集 (train set) 由来自 250 个不同人手写的数字构成，其中 50%是高中学生，50%来自人口普查局的工作人员. 测试集(test set) 也是同样比例的手写数字数据，MNIST 手写体图片就可以看做是一个 28*28 的二维向量。

2.2 卷积神经网络（CNN）

2.2.1 神经网络到卷积神经网络

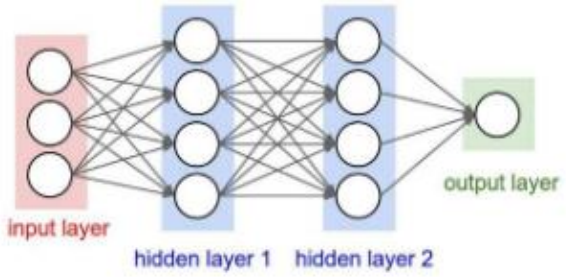

图 1 神经网络图

上图为经典神经网络结构，这是一个包含四个层次的神经网络。红色的是输入层，绿色的是输出层，蓝色的的是中间层（也叫隐藏层），为层级网络。CNN 属于一种特殊的多层神经网络，依旧是层级网络，只是层的功能和形式做了变化，可以说是传统神经网络的一个改进。

CNN 是一种前馈神经网络，它的人工神经元可以响应一部分覆盖范围内的周围单元，对于大型图像处理有出色表现。图像处理中，往往会将图像看成是一个或多个的二维向量，与传统神经网络采用全连接的方式不同，CNN 通过**局部连接**和**权值共享**的方法来避免参数过多。

2.2.2 卷积神经网络的层级结构：

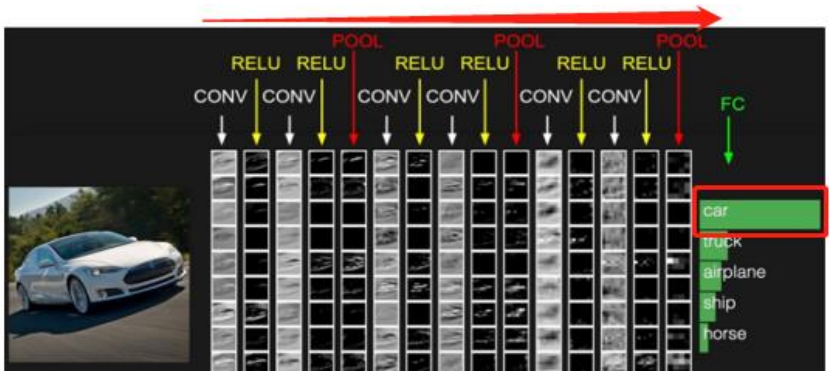

图 2 卷积神经网络过程

如上图所示，对于输入的图像数据，经过多次的卷积计算层（CONV）和 ReLU 激励层、池化层 （Pool），进行识别。下面具体介绍层次。

- 数据输入层/ Input layer

对原始图像数据进行预处理，其中包括：去均值；归一化；PCA 化

- 卷积计算层/ CONV layer

是卷积神经网络最重要的一个层次，也是“卷积神经网络”的名字来源。在卷积层，有两个关键操作：局部关联。卷积运算的目的是提取输入的不同特征，第一层卷积层可能只能提取一些低级的特征如边缘、线条和角等层级，更多层的网络能从低级特征中迭代提取更复杂的特征。

1) 卷积计算方法：（见图 1）

每个神经元看做滤波器；窗口滑动，通过输入图像矩阵与滤波器矩阵求内积，并将内积运算的结果与偏置值 b 相加来计算。

a) 深度/depth：待卷积矩阵的个数

b) 步长/stride：窗口滑动步长

c) 填充值/padding：在窗口滑动过程中，不能正好将矩阵遍历完，填充矩阵使之能正好滑动结束，见图 2。SAME 为输出矩阵大小与输入矩阵大小一致。

2) 权值共享机制

在卷积层中每个神经元连接连接数据窗的权重是固定的，每个神经元只关注一个特性。神经元就是图像处理中的滤波器，每个滤波器都会有关注的一个图像特征，比如垂直边缘，水平边缘，颜色，纹理等，这些所有神经元加起来就好比就是整张图像的特征提取器集合。相比传统神经网络，所要估算的权重数量大大减少。

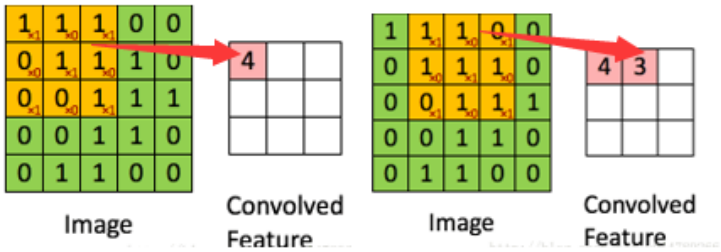

图 3 卷积过程图解，步长为 1

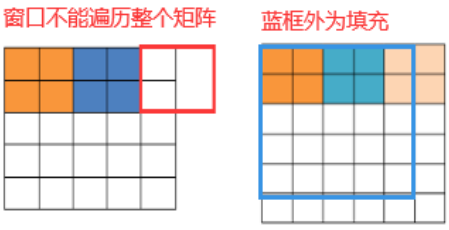

图 4 填充图解 步长为 2

- ReLU 激励层 / ReLU layer

把卷积层输出结果做非线性映射。 CNN 采用的激励函数一般为 ReLU(修正线性单元)，它的特点是收敛快，求梯度简单，但较脆弱，图像如下。

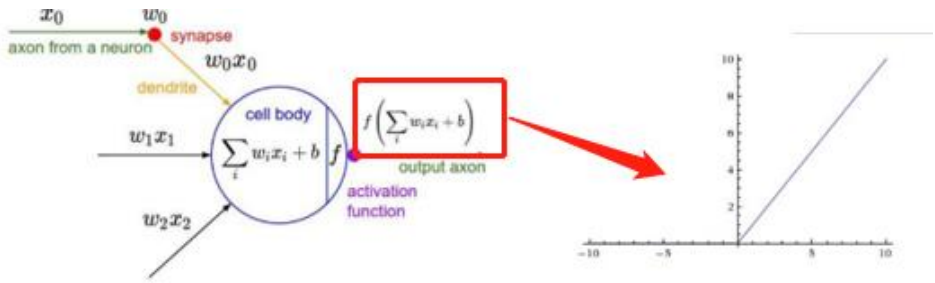

图 5 激励层图解

- 池化层 / Pooling layer

池化层夹在连续的卷积层（卷积计算层和激励层）中间， 用于压缩数据和参数的量，减小过拟合，压缩图像。用的方法有 Max pooling 和 average pooling，常用的是 Max pooling。主要作用有：

1) 特征不变性，也就是特征的尺度不变性，图像的 resize，去掉的信息只是一些无关紧要的信息，留下最能表达图像的特征的信息，则是具有尺度不变性的特征。
2) 特征降维，把冗余信息去除，把最重要的特征抽取出来。
3) 在一定程度上防止过拟合，更方便优化。

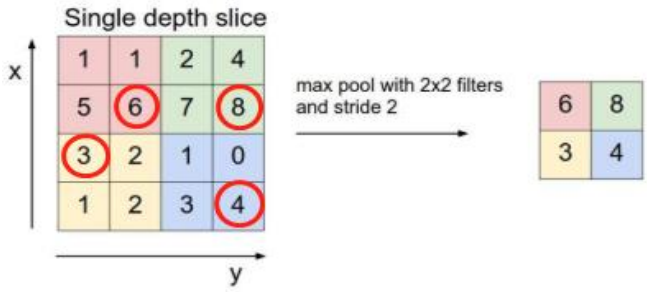

图 6 max pooling 图示：选出每个模块最大值

- 全连接层 / FC layer

两层之间所有神经元都有权重连接，通常全连接层在卷积神经网络尾部。也就是跟传统的神经网络神经元的连接方式是一样的。

三、 操作过程

3.1 准备工作

1) 配置 python 环境，下载解释器和 pycharm
2) 在 pycharm 中安装 TensorFlow
3) 准备 MNIST 数据集，下载网址：http://yann.lecun.com/exdb/mnist/，做成 input_data.py 文件，文件可以从官网上下载。

图 7 MNIST 数据集 input_data.py 文件

3.2 代码分析

代码流程：

图 8 代码流程

1) 读取数据

```
import input_data
import tensorflow as tf #用tf来表示tensorflow

#读取数据
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
sess=tf.InteractiveSession()
```

2) 构建 CNN 网络

- 定义卷积函数和池化函数。

```
#自定义卷积函数
def conv2d(x,w):return tf.nn.conv2d(x,w,strides=[1,1,1,1],padding='SAME')
#自定义池化函数
def max_pool_2x2(x):return tf.nn.max_pool(x,ksize=[1,2,2,1],strides=[1,2,2,1],padding='SAME')
```

```
strides=[1,1,1,1]
```

卷积步长为 1，对于二维图来说只有中间两位有效。（若步长为 2： strides= [1,2,2,1]）

```
padding='SAME'
```

padding 方法，（即输出与输入保持相同尺寸，边界处少一两个像素则自动补上）

- 设置占位符

```
#设占位符，尺寸为样本输入和输出的尺寸
x=tf.placeholder(tf.float32,[None,784])
y_=tf.placeholder(tf.float32,[None,10])
x_img=tf.reshape(x,[-1,28,28,1])
```

设置输入输出的占位符，占位符作为一个会话中喂数据的入口，因为 TensorFlow 的使用中，通过构建计算图来设计网络，而网络的运行计算则在会话中启动，这个过程我们无法直接介入，需要通过 placeholder 来对一个会话进行数据输入。占位符设置好之后，利用 tf.reshape()函数将 x 变形成为 28x28 是矩阵形式

- 设置层级

a) 卷积层和池化层

```
  #设置第一个卷积层和池化层
w_conv1=tf.Variable(tf.truncated_normal([3,3,1,32],stddev=0.1))
b_conv1=tf.Variable(tf.constant(0.1,shape=[32]))
h_conv1=tf.nn.relu(conv2d(x_img,w_conv1)+b_conv1)
h_pool1=max_pool_2x2(h_conv1)

#设置第二个卷积层和池化层
w_conv2=tf.Variable(tf.truncated_normal([3,3,32,50],stddev=0.1))
b_conv2=tf.Variable(tf.constant(0.1,shape=[50]))
h_conv2=tf.nn.relu(conv2d(h_pool1,w_conv2)+b_conv2)
h_pool2=max_pool_2x2(h_conv2)
```

第一层卷积

```
tf.truncated_normal([3,3,1,32],stddev=0.1)
```

使用 3x3x1 的卷积核，一共有 32 个卷积核，权值使用方差为 0.1 的截断正态分布（指最大值不超过方差两倍的分布）来初始化。

```
b_conv1=tf.Variable(tf.constant(0.1,shape=[32]))
```

偏置的初值设定为常值 0.1。

第二层卷积

和第一层类似，卷积核尺寸为 3x3x32（32 是通道数，因为上一层使用 32 个卷积核，所以这一层的通道数就变成了 32），这一层一共使用 50 个卷积核，其他设置与上一层相同。

池化层

```
h_pool1=max_pool_2x2(h_conv1)
```

每一层卷积完之后接上一个 2x2 的最大化池化操作。

b) 全连接层

```
#设置第一个全连接层
w_fc1=tf.Variable(tf.truncated_normal([7*7*50,1024],stddev=0.1))
b_fc1=tf.Variable(tf.constant(0.1,shape=[1024]))
h_pool2_flat=tf.reshape(h_pool2,[-1,7*7*50])
h_fc1=tf.nn.relu(tf.matmul(h_pool2_flat,w_fc1)+b_fc1)

#dropout（随机权重失活）
keep_prob=tf.placeholder(tf.float32)
h_fc1_drop=tf.nn.dropout(h_fc1,keep_prob)

#设置第二个全连接层
w_fc2=tf.Variable(tf.truncated_normal([1024,10],stddev=0.1))
b_fc2=tf.Variable(tf.constant(0.1,shape=[10]))
y_out=tf.nn.softmax(tf.matmul(h_fc1_drop,w_fc2)+b_fc2)
```

第一个全连接层

第一个全连接层有 1024 个神经元，先将卷积层得到的 2x2 输出展开成一长条，使用 Relu 激活函数得到输出，输出为 1024 维。

Dropout

在这一层使用 dropout（权值随机失活），对一些神经元突触连接进行强制的置零，这可以防止神经网络过拟合。训练时这里的 dropout 的保留比例是 0.5，即随机地保留一半权值，删除另外一半，这能保证训练集的效果，测试时为保留 1。

第二个全连接层

有 10 个神经元，分别对应 0-9 这 10 个数字，与之前的每一层不同的是，这里使用的激活函数是 Softmax，我的个人理解是 softmax 是“以指数函数作为核函数的归一化操作”，softmax 与一般归一化操作不同的是，“两极分化”现象会更明显（对同一个分布进行一般的归一化得到的分布和 softmax 得到的分布，softmax 得到的分布信息熵要更大）

3) 构建 Loss function、配置寻优器

```
#建立loss function，为交叉熵
loss=tf.reduce_mean(-tf.reduce_sum(y_*tf.log(y_out),reduction_indices=[1]))
#配置Adam优化器，学习速率为1e-4
train_step=tf.train.AdamOptimizer(1e-4).minimize(loss)

#建立正确率计算表达式
correct_prediction=tf.equal(tf.argmax(y_out,1),tf.argmax(y_,1))
accuracy=tf.reduce_mean(tf.cast(correct_prediction,tf.float32))
```

这里使用交叉熵来作为 loss，交叉熵是用来衡量两个分布的相似程度的，两个分布越接近，则交叉熵越小。使用 Adam 优化器来最小化 loss，配置学习速率为 1e-4。然后建立正确率的计算表达式（注意，仅仅是建立而已，现在还没有开始真正的计算），

tf.argmax(y_,1)，函数用来返回其中最大的值的下标

tf.equal()用来计算两个值是否相等。

tf.cast()函数用来实现数据类型转换（这里是转换为 float32）

tf.reduce_mean()用来求平均（得到正确率）。

4) 训练

```
#开始喂数据，训练
tf.global_variables_initializer().run()
for i in range(20000):
    batch=mnist.train.next_batch(50)
    if i%100==0:
        train_accuracy=accuracy.eval(feed_dict={x:batch[0],y_:batch[1],keep_prob:1})
        print ("step %d,train_accuracy= %g"%(i,train_accuracy))
    train_step.run(feed_dict={x:batch[0],y_:batch[1],keep_prob:0.5})
```

首先使用 tf.global_variables_initializer().run()初始化所有数据，从 mnist 训练数据集中一次取 50 个样本作为一组进行训练，一共进行 20000 组训练，每 100 次就输出一次该组数据上的正确率。进行训练计算的方式是：

```
train_step.run(feed_dict={x:batch[0],y_:batch[1],keep_prob:0.5})
```

通过 feed_dict 来对会话输送训练数据（以及其他一些想在计算过程中实时调整的参数，比如 dropout 比例）。可以看到，训练时 dropout 的保留比例是 0.5，测试时的保留比例是 1。

```
#使用测试集进行测试
accuracy_sum = tf.reduce_sum(tf.cast(correct_prediction, tf.float32))
good = 0
total = 0
for i in range(2):
    testSet = mnist.test.next_batch(100)
    if i ==1 : print(testSet[0].shape[0])
    good += accuracy_sum.eval(feed_dict = { x: testSet[0], y_: testSet[1], keep_prob: 1.0})
    total += testSet[0].shape[0]  # testSet[0].shape[0] 是本batch有的样本数量
print("test accuracy %g"%(good/total))
```

输入测试数据集进行测试验证。

3.3 结果分析

3.3.1 训练结果

```
step 0,train_accuracy= 0.08
step 100,train_accuracy= 0.88
step 200,train_accuracy= 0.86
step 300,train_accuracy= 0.84
step 400,train_accuracy= 0.94
step 500,train_accuracy= 0.94
step 600,train_accuracy= 0.98
step 700,train_accuracy= 0.94
step 800,train_accuracy= 0.96
step 900,train_accuracy= 0.94
step 1000,train_accuracy= 0.94
step 1100,train_accuracy= 0.92
step 1200,train_accuracy= 0.96
step 1300,train_accuracy= 1
step 1400,train_accuracy= 0.98
```

```
step 18500,train_accuracy= 1
step 18600,train_accuracy= 1
step 18700,train_accuracy= 1
step 18800,train_accuracy= 1
step 18900,train_accuracy= 1
step 19000,train_accuracy= 1
step 19100,train_accuracy= 1
step 19200,train_accuracy= 1
step 19300,train_accuracy= 1
step 19400,train_accuracy= 1
step 19500,train_accuracy= 1
step 19600,train_accuracy= 1
step 19700,train_accuracy= 1
step 19800,train_accuracy= 1
step 19900,train_accuracy= 1
```

随着训练数据的增加，训练准确率逐渐提高，在 2000 次左右逐渐稳定在 1。

3.3.2 测试结果

```
test accuracy 0.99
```

测试准确率为 0.99，较高。查阅资料后了解到，相比使用经典三层神经网络的准确率 94%左右，使用卷积神经网络效率大大提升，准确得到提升。

四、 遇到的问题与启发

在下载 TensorFlow 的过程中遇到了很大的问题，首先在 cmd 窗口直接下载安装速度过慢，经过在网上查阅资料慢是正常现象，后使用镜像高速下载，但下载之后，程序依然报错不能运行，原来是没有在 pycharm 中启用 TensorFlow，正确启用后。报错显示程序函数不存在。在查阅资料以及咨询朋友后了解随着 TensorFlow 版本的更新，很多函数和使用方法已经摒弃了，此时我安装软件已经安装了 2 天了。于是我卸载掉下载的 TensorFlow2.0，寻找方法安装了较低的版本。期间还陆陆续续出现了很多的小问题，但是在我咨询朋友和查阅资料的方法下慢慢迎刃而解，有时候人就是会在挫折和他人帮助中跌跌撞撞成长！

经过对 CNN 的理论了解和程序解读以及结果的显示，让我了解到了机器学习的魅力。