**实验一 手写数字识别**

1. 实验目的
2. 掌握卷积神经网络基本原理；
3. 掌握Tensorflow的基本用法以及构建卷积网络的基本操作；
4. 了解Tensorflow在GPU上的使用方法。
5. 实验要求
6. 搭建Tensorflow环境；
7. 构建一个规范的卷积神经网络组织结构；
8. 在MNIST手写数字数据集上进行训练和评估。
9. 实验原理
10. TensorFlow基本用法：

使用 TensorFlow, 必须了解TensorFlow:

* 使用图(graph) 来表示计算任务。
* 在被称之为会话 (Session) 的上下文 (context) 中执行图。
* 使用 tensor 表示数据。
* 通过 变量 (Variable) 维护状态。
* 使用 feed 和 fetch 可以为任意的操作(arbitrary operation) 赋值或者从其中获取数据。

TensorFlow 是一个编程系统, 使用图来表示计算任务。图中的节点被称之为 op (operation 的缩写)。一个 op 获得 0 个或多个 Tensor, 执行计算, 产生 0 个或多个 Tensor。每个 Tensor 是一个类型化的多维数组。例如, 你可以将一小组图像集表示为一个四维浮点数数组, 这四个维度分别是 [batch, height, width, channels]。

一个 TensorFlow 图描述了计算的过程。为了进行计算, 图必须在“会话”里被启动。 “会话“将图的 op 分发到诸如 CPU 或 GPU 之类的设备上, 同时提供执行op的方法。 这些方法执行后, 将产生的tensor返回。在 Python 语言中, 返回的 tensor 是 numpy ndarray 对象; 在 C 和 C++ 语言中, 返回的 tensor 是 tensorflow::Tensor 实例。

1. 卷积神经网络：

典型的卷积神经网络由卷积层、池化层、激活函数层交替组合构成，因此可将其视为一种层次模型，形象地体现了深度学习中“深度”之所在。

* 卷积操作

卷积运算是卷积神经网络的核心操作，给定二维的图像I作为输入，二维卷积核K， 卷积运算可表示为：

 (1)

给定5×5输入矩阵、3×3卷积核，相应的卷积操作如图1所示。



图1 卷积运算

在使用TensorFlow等深度学习框架时，卷积层会有padding参数，常用的有两种选择，一个是“valid”，一个是“same”。前者是不进行填充，后者则是进行数据填充并保证输出与输入具有相同尺寸。

构建卷积或池化神经网络时，卷积步长也是一个很重要的基本参数。它控制了每个操作在特征图上的执行间隔。

* 池化操作

池化操作使用某位置相邻输出的总体统计特征作为该位置的输出，常用最大池化（max-pooling）和均值池化（average-pooling）。池化层不包含需要训练学习的参数，仅需指定池化操作的核大小、操作步长以及池化类型。池化操作示意如图2所示。



图2 池化操作

* 激活函数层

卷积操作可视为对输入数值进行线性计算发挥线性映射的作用。激活函数的引入，则增强了深度网络的非线性表达能力，从而提高了模型的学习能力。常用的激活函数有sigmoid、tanh和ReLU函数。

1. 实验所用工具及数据集
2. 工具

Anaconda、TensorFlow

（Tensorflow安装教程参考：Tensorflow官网、Tensorflow中文社区、<https://github.com/tensorflow/tensorflow>）

1. 数据集

MNIST手写数字数据集

（下载地址及相关介绍：http://yann.lecun.com/exdb/mnist/）

1. 实验步骤与方法
2. 安装实验环境，包括Anaconda、TensorFlow（建议安装GPU版本），若使用GPU版本还需要安装cuda、cudnn；
3. 下载MNIST手写数字数据集；
4. 加载MNIST数据；

**加载图像：**

with gzip.open(filename) as bytestream:

bytestream.read(16) #每个像素存储在文件中的大小为16bits

buf = bytestream.read(IMAGE\_SIZE \* IMAGE\_SIZE \* num\_images \* NUM\_CHANNELS)

data = numpy.frombuffer(buf, dtype=numpy.uint8).astype(numpy.float32)

#像素值[0, 255]被调整到[-0.5, 0.5]

data = (data - (PIXEL\_DEPTH / 2.0)) / PIXEL\_DEPTH

#调整为4维张量[image index, y, x, channels]

data = data.reshape(num\_images, IMAGE\_SIZE, IMAGE\_SIZE, NUM\_CHANNELS)

**加载标签：**

with gzip.open(filename) as bytestream:

bytestream.read(8) #每个标签存储在文件中的大小为8bits

buf = bytestream.read(1 \* num\_images)

labels = numpy.frombuffer(buf, dtype=numpy.uint8).astype(numpy.int64)

4）构建模型计算图；

**创建输入占位符：**

#这是训练样本和标签被送到图表的地方。

#这些占位符节点将在每个节点输入一批训练数据

#训练步骤使用{feed\_dict}参数进行下面的Run（）调用。

train\_data\_node = tf.placeholder( data\_type(),

shape=(BATCH\_SIZE, IMAGE\_SIZE, IMAGE\_SIZE, NUM\_CHANNELS))

train\_labels\_node = tf.placeholder(tf.int64, shape=(BATCH\_SIZE,))

eval\_data = tf.placeholder(data\_type(),

shape=(EVAL\_BATCH\_SIZE, IMAGE\_SIZE, IMAGE\_SIZE, NUM\_CHANNELS))

**初始化变量：**

# 下面的变量包含所有可训练的权重。当我们调用时将分配它们时，它们被传递一个初始值： # {tf.global\_variables\_initializer().run()}

conv1\_weights = tf.Variable(

tf.truncated\_normal([5, 5, NUM\_CHANNELS, 32], # 5x5 filter, depth 32.

stddev=0.1,

seed=SEED, dtype=data\_type()))

conv1\_biases = tf.Variable(tf.zeros([32], dtype=data\_type()))

conv2\_weights = tf.Variable(tf.truncated\_normal(

[5, 5, 32, 64], stddev=0.1,

seed=SEED, dtype=data\_type()))

conv2\_biases = tf.Variable(tf.constant(0.1, shape=[64], dtype=data\_type()))

fc1\_weights = tf.Variable( # fully connected, depth 512.

tf.truncated\_normal([IMAGE\_SIZE // 4 \* IMAGE\_SIZE // 4 \* 64, 512],

stddev=0.1,

seed=SEED,

dtype=data\_type()))

fc1\_biases = tf.Variable(tf.constant(0.1, shape=[512], dtype=data\_type()))

fc2\_weights = tf.Variable(tf.truncated\_normal([512, NUM\_LABELS],

stddev=0.1,

seed=SEED,

dtype=data\_type()))

fc2\_biases = tf.Variable(tf.constant(

0.1, shape=[NUM\_LABELS], dtype=data\_type()))

**CNN模型构建：**

# 2D卷积，带有“SAME”填充（即输出要素图与输入的大小相同）。

# 请注意，{strides}是一个4D数组，其形状与数据布局匹配：[image index，y，x，depth]。

conv = tf.nn.conv2d(data,

conv1\_weights,

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

padding='SAME')

# 偏置和ReLU非线性激活。

relu = tf.nn.relu(tf.nn.bias\_add(conv, conv1\_biases))

# 最大池化。

# 内核大小规范{ksize}也遵循数据布局。 这里我们有一个2的池化窗口和2的步幅。

pool = tf.nn.max\_pool(relu,

ksize=[1, 2, 2, 1],

strides=[1, 2, 2, 1],

padding='SAME')

conv = tf.nn.conv2d(pool,

conv2\_weights,

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

padding='SAME')

relu = tf.nn.relu(tf.nn.bias\_add(conv, conv2\_biases))

pool = tf.nn.max\_pool(relu,

ksize=[1, 2, 2, 1],

strides=[1, 2, 2, 1],

padding='SAME')

# 将特征图变换为2D矩阵，以将其提供给完全连接的图层。

pool\_shape = pool.get\_shape().as\_list()

reshape = tf.reshape(

pool,

[pool\_shape[0], pool\_shape[1] \* pool\_shape[2] \* pool\_shape[3]])

# 全连接层。

hidden = tf.nn.relu(tf.matmul(reshape, fc1\_weights) + fc1\_biases)

# 在训练时，添加dropout层。

if train:

hidden = tf.nn.dropout(hidden, 0.5, seed=SEED)

logits = tf.matmul(hidden, fc2\_weights) + fc2\_biases

**训练与评估：**

# 损失计算（+L2正则化）

loss = tf.reduce\_mean(tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(

labels=train\_labels\_node, logits=logits))

# L2正则化损失

regularizers = (tf.nn.l2\_loss(fc1\_weights) + tf.nn.l2\_loss(fc1\_biases) +

tf.nn.l2\_loss(fc2\_weights) + tf.nn.l2\_loss(fc2\_biases))

# 总损失=样本损失+L2正则化损失

loss += 5e-4 \* regularizers

# 优化器（用于参数更新）

# 设置一个每批增加一次的变量，并控制学习率衰减。

batch = tf.Variable(0, dtype=data\_type())

# 每个时期衰减一次，使用从0.01开始的指数衰减。

learning\_rate = tf.train.exponential\_decay(

0.01, # Base learning rate.

batch \* BATCH\_SIZE, # Current index into the dataset.

train\_size, # Decay step.

0.95, # Decay rate.

staircase=True)

# 用momentum优化器优化

optimizer = tf.train.MomentumOptimizer(learning\_rate, 0.9).minimize(loss, global\_step=batch)

5）创建会话，训练和评估模型。

# 创建会话训练、评估模型

start\_time = time.time()

with tf.Session() as sess:

# 初始化可训练变量

tf.global\_variables\_initializer().run()

for step in xrange(int(num\_epochs \* train\_size) // BATCH\_SIZE):

# Compute the offset of the current minibatch in the data.

# Note that we could use better randomization across epochs.

offset = (step \* BATCH\_SIZE) % (train\_size - BATCH\_SIZE)

batch\_data = train\_data[offset:(offset + BATCH\_SIZE), ...]

batch\_labels = train\_labels[offset:(offset + BATCH\_SIZE)]

# 将数据送入计算图

feed\_dict = {train\_data\_node: batch\_data,

train\_labels\_node: batch\_labels}

# 运行优化器进行参数更新

sess.run(optimizer, feed\_dict=feed\_dict)

# 打印信息

if step % EVAL\_FREQUENCY == 0:

# 运行计算图得到评估信息

l, lr, predictions = sess.run([loss, learning\_rate, train\_prediction],

feed\_dict=feed\_dict)

elapsed\_time = time.time() - start\_time

start\_time = time.time()

print('Step %d (epoch %.2f), %.1f ms' %

(step, float(step) \* BATCH\_SIZE / train\_size,

1000 \* elapsed\_time / EVAL\_FREQUENCY))

print('Minibatch loss: %.3f, learning rate: %.6f' % (l, lr))

print('Minibatch error: %.1f%%' % error\_rate(predictions, batch\_labels))

print('Validation error: %.1f%%' % error\_rate(

eval\_in\_batches(validation\_data, sess), validation\_labels))

sys.stdout.flush()

# 输出测试结果

test\_error = error\_rate(eval\_in\_batches(test\_data, sess), test\_labels)

print('Test error: %.1f%%' % test\_error)

if FLAGS.self\_test:

print('test\_error', test\_error)

assert test\_error == 0.0, 'expected 0.0 test\_error, got %.2f' % (

test\_error,)