深度学习方法与实践第四次作业

姓名：杨玉雷
学号：18023040

1.基础作业

将 LENET 封装为 class，并用此封装好的 lenet 对 minist 进行分类。
有关 lenet 定义请参考卷积网络课件最后 2 页；class 封装的内容，请参考 class 封装课件

1. lenet 结构如附件描述。注意：

（1）lenet 输入为 32x32，而 minist 为 28x28，故需要先对数据进行填充，例如：

```
import numpy as np

# Pad images with 0s
X_train      = np.pad(X_train, ((0,0),(2,2),(2,2),(0,0)), 'constant')
X_validation = np.pad(X_validation, ((0,0),(2,2),(2,2),(0,0)),
'constant')
X_test       = np.pad(X_test, ((0,0),(2,2),(2,2),(0,0)), 'constant')
print("Updated Image Shape: {}".format(X_train[0].shape))
from sklearn.utils import shuffle
X_train, y_train = shuffle(X_train, y_train)
```

（2）lenet 输出 10 位的 one-hot 形式的输出 logits，故 minist 的标签读取需采用 one-hot 的形式。
采用 softmax 交叉熵作为损失函数。用 softmax 进行分类。

2. 在 init 函数中传入初始化变量所需的 mu， sigma 参数，以及其他所需定制化参数。
例如：

```
def __init__(self,mu):
    self.mu=mu
```

设计需要的输入输出接，例如，如果想把对外数据的交互也封装在 class 里

self.raw_input_image = tf.placeholder(tf.float32, [None, 784]) 或者需要的进一步变换，例如
self.input_x = tf.reshape(self.raw_input_image, [-1, 28, 28, 1])

或者把外部交互的事情交给外部去做，class 只是想实现一个纯净的 net 计算通路：
self.input_x=input (input 是你外部给的输入引用)

3. 对 lenet 中常见的 conv 层，fc 层，pooling 层定义统一的定制化功能层 graph 绘图函数. 为层次化组织网络，给每个层定义一个不同的名字空间，例如：

```
def conv(w_shape, scope_name, .......):
    with tf.variable_scope(scope_name) as scope:
        xxxx.....
```

4. 绘制整个网络计算图的函数，_build_graph(). 这里要求调_build_graph()的过程放在_init_函数里，这样外部每调用并生成一个 class 的实例，实际上就自动绘制了一次 lenet。

_build_graph()绘制整个 lenet 的时候，调用之前你定义的各个功能层，并逐层搭建出整个网络。期望网络对外的输出 tensor 引用都用 self 记录，例如：

```
def __init__(self, config):
   self.config = config
   self._build_graph()
    .....
def _build_graph(self, network_name='Lenet'):
    self._setup_placeholders_graph()
    self._build_network_graph(network_name)
    self._compute_loss_graph()
    self._compute_acc_graph()
```

5. 在外部调用该模块并通过实例化实现对 lenet 的绘制，例如：
from lenet import Lenet （lenet.py 里定义的 class Lenet）

lenet_part = Lenet()

这样调用一下已经完成了 lenet 的绘制了，你需要引用的 lenet 中间的 tensor 都保存在 lenet_part 里。

例如：

```
sess.run(train_op,feed_dict={lenet.raw_input_image:
batch[0],lenet.raw_input_label: batch[1]})
```

要求：用 class 封装好的 lenet 对 minist 进行分类，训练和模型定义分开成两个文件 train.py，lenet.py，打印训练和测试截图，测试分类准确率 ACC。

2.基础作业实验过程和关键代码

根据实验要求，实验过程如下：

（1）在 lenet.py 文件中封装 Lenet 网络结构，定义 Lenet 类，其中在__init__方法中调用画图函数_build_graph()，使得在创建实例的时候就开始画图，具体定义如下：

```
class Lenet:
#定义构造函数
    def __init__(self,learning_rate,sigma):
        self.learning_rate=learning_rate
        self.sigma=sigma
        # 当在创建的时候运行画图
        self._build_graph()

#涉及网络的所有画图 build graph 过程,常用一个 build graph 封起来
    def _build_graph(self, network_name='Lenet'):
        self._setup_placeholders_graph()
        self._build_network_graph(network_name)
        self._compute_loss_graph()
        self._create_train_op_graph()
        self._compute_acc_graph()

#开始构建输入占位图
    def _setup_placeholders_graph(self):
            self.x=tf.placeholder(tf.float32,(None,32,32,1),  name='input_x')
            self.y= tf.placeholder(tf.int32, (None))  # None 在模型中的占位

#构建网络结构图
    def _build_network_graph(self, scope_name):
        with tf.variable_scope(scope_name):
            #第一个卷积层
            # Input = 32x32x1. Output = 28x28x6.
            self.conv1_relu=self._cnn_layer('layer_1_conv','conv1_w',
'conv1_b', self.x, (5,5,1,6),[1,1,1,1],[6])

            #第一个池化层
            #Input = 28x28x6. Output = 14x14x6.
            self.pool1=self._pooling_layer('layer_2_pooling',
self.conv1_relu, [1, 2, 2, 1], [1, 2, 2, 1])

            #第二个卷积层
            #Output = 10x10x16.
```

```
            self.conv2_relu=self._cnn_layer('layer_3_conv','conv2_w',
'conv2_b', self.pool1, (5, 5,6,16), [1, 1, 1, 1],[16])

            #第二个池化层
            #Input = 10x10x16. Output = 5x5x16.
            self.pool2=self._pooling_layer('layer_4_pooling',
self.conv2_relu, [1, 2, 2, 1], [1, 2, 2, 1])

            #Tensor to vector:输入维度由 Nx5x5x16 压平后变为 Nx400
            self.pool2=flatten(self.pool2)

            #第一个全连接层
            #Input = 400. Output = 120.
            self.fc1_relu=self._fully_connected_layer('layer_5_fc1','fc1_w'
,'fc1_b',self.pool2,(400,120),[120])

            #第二个全连接层
            #Input = 120. Output = 84.
            self.fc2_relu=self._fully_connected_layer('layer_6_fc2','fc2_w'
,'fc2_b',self.fc1_relu,(120,84),[84])

            #第三个全连接层
            #Input = 84. Output = 10.
            self.fc3_relu=self._fully_connected_layer('layer_7_fc3','fc3_w'
,'fc3_b',self.fc2_relu,(84,10),[10])
            self.digits=self.fc3_relu
            return self.digits

#构建求梯度计算训练图
    def _create_train_op_graph(self):
            self.train_op=tf.train.AdamOptimizer(self.learning_rate).minimiz
e(self.loss)

#构建求损失函数的计算图
    def _compute_loss_graph(self):
            cross_entropy=tf.nn.softmax_cross_entropy_with_logits(labels=sel
f.y,logits=self.digits)
            self.loss=tf.reduce_mean(cross_entropy)

#构建准确率计算图
    def _compute_acc_graph(self):
        self.prediction=tf.equal(tf.argmax(self.digits,1), tf.argmax(self.y,
 1))
        self.accuracy=tf.reduce_mean(tf.cast(self.prediction, tf.float32))
```

```
#卷积层函数定义
    def _cnn_layer(self,scope_name,W_name,b_name,x,filter_shape,conv_strides,b_shape,padding_tag='VALID'):
        with tf.variable_scope(scope_name) as scope:
            conv_weights=tf.get_variable(name=W_name,shape=filter_shape,initializer=tf.truncated_normal_initializer(stddev=self.sigma))
            conv_biases=tf.get_variable(name=b_name,shape=b_shape,initializer=tf.constant_initializer(0.0))
            conv=tf.nn.conv2d(x,conv_weights,strides=conv_strides, padding=padding_tag)
            return tf.nn.relu(tf.nn.bias_add(conv, conv_biases))

#池化层函数定义
 def _pooling_layer(self, scope_name, relu, pool_ksize,pool_strides, padding_tag='VALID'):
        with tf.variable_scope(scope_name) as scope:
            return tf.nn.max_pool(relu,ksize=pool_ksize, strides =pool_strides, padding=padding_tag)

#全连接层函数定义
    def _fully_connected_layer(self,scope_name,W_name, b_name,x,W_shape,b_shape):
        with tf.variable_scope(scope_name) as scope:
            fc_weights= tf.get_variable(W_name,W_shape,initializer=tf.truncated_normal_initializer(stddev=self.sigma))
            fc_biases= tf.get_variable(b_name,b_shape,initializer=tf.constant_initializer(0.0))
            return tf.nn.relu(tf.matmul(x, fc_weights) + fc_biases)
```

（2）在 train.py 文件中创建 Lenet 实例对 Mnist 数据的训练集进行训练和测试，为了保持 Lenet 网络结构纯洁性，对数据的扩充处理放在 Lenet 类外，训练完成后对 Mnist 的测试集进行测试：

```
#导入训练集，测试集
def load_data():
   '''
   导入数据
   :return:训练集、测试集
   '''
   mnist = read_data_sets("MNIST_data/", reshape=False,one_hot=True)
   X_train, y_train = mnist.train.images, mnist.train.labels
   X_test, y_test = mnist.test.images, mnist.test.labels
```

```
    assert (len(X_train) == len(y_train))
    assert (len(X_test) == len(y_test))

    print()
    print("Image Shape: {}".format(X_train[0].shape))
    print()
    print("Training Set:   {} samples".format(len(X_train)))
    print("Test Set:       {} samples".format(len(X_test)))

    # 将训练集进行填充
    # 因为 mnist 数据集的图片是 28*28*1 的格式，而 lenet 只接受 32*32 的格式
    # 所以只能在这个基础上填充
    X_train = np.pad(X_train, ((0, 0), (2, 2), (2, 2), (0, 0)), 'constant')
    print("x_train_32:", X_train.shape)
    X_test = np.pad(X_test, ((0, 0), (2, 2), (2, 2), (0, 0)), 'constant')
    print("Updated Image Shape: {}".format(X_train[0].shape))

    # 打乱数据集的顺序
    X_train, y_train = shuffle(X_train, y_train)
    return X_train, y_train,X_test,y_test

#返回测试集上的准确率
def evaluate():
    num_examples = len(X_test)
    total_accuracy = 0
    sess = tf.get_default_session()  # 返回当前线程的默认会话
    for offset in range(0, num_examples, BATCH_SIZE):
        batch_x,batch_y=X_test[offset:offset+BATCH_SIZE], y_test[offset:offse
t + BATCH_SIZE]
        accuracy=sess.run(lenet5.accuracy, feed_dict={lenet5.x: batch_x, lene
t5.y: batch_y})
        total_accuracy += (accuracy * len(batch_x))
    return total_accuracy / num_examples

BATCH_SIZE = 100
MODEL_SAVE_PATH = "model/"
MODEL_NAME="lenet.ckpt"
lenet5 = Lenet(0.001,0.1) #实例化一个 Lenet
saver = tf.train.Saver()

init = tf.global_variables_initializer()
with tf.Session() as sess:
    sess.run(init)
```

```
    X_train, y_train,X_test,y_test= load_data()
    num_examples = len(X_train)
    max_iter=int(num_examples/BATCH_SIZE)#迭代次数
    print("Start Training...")
    X_train, y_train = shuffle(X_train, y_train)  # 随机排序
    for j in range(max_iter):
          start=j*BATCH_SIZE
          end=start+BATCH_SIZE
          batch_xs,batch_ys=X_train[start:end],y_train[start:end]
          sess.run([lenet5.train_op],feed_dict={lenet5.x: batch_xs, lenet5.y:
batch_ys})
          accuracy,loss=sess.run([lenet5.accuracy,lenet5.loss],
feed_dict={lenet5.x: batch_xs, lenet5.y: batch_ys})
          print("Step:",j," accuracy:",accuracy," loss:",loss)
    saver.save(sess, os.path.join(MODEL_SAVE_PATH, MODEL_NAME))
    # print("Model saved")

#测试分类准确率ACC
with tf.Session() as sess:
     saver.restore(sess,os.path.join(MODEL_SAVE_PATH, MODEL_NAME))
     test_accuracy = evaluate()
     print("Test Accuracy = {:.3f}".format(test_accuracy))
```

（3）训练结束后打印在测试集上的准确率，由于参数初始化不同，得到的结果也不同，其中一次训练过程和测试准确率结果如下：

```
Start Training...
Step: 0  accuracy: 0.2  loss: 2.26552
Step: 1  accuracy: 0.15  loss: 2.27068
Step: 2  accuracy: 0.2  loss: 2.2536
Step: 3  accuracy: 0.3  loss: 2.24858
Step: 4  accuracy: 0.34  loss: 2.2396
Step: 5  accuracy: 0.46  loss: 2.18873
Step: 6  accuracy: 0.5  loss: 2.17834
Step: 7  accuracy: 0.52  loss: 2.15371
Step: 8  accuracy: 0.64  loss: 2.086
Step: 9  accuracy: 0.53  loss: 2.07728
Step: 10  accuracy: 0.41  loss: 2.14029
Step: 11  accuracy: 0.51  loss: 1.98592
Step: 12  accuracy: 0.43  loss: 1.94971
Step: 13  accuracy: 0.4  loss: 2.00364
Step: 14  accuracy: 0.4  loss: 2.01284
Step: 15  accuracy: 0.45  loss: 1.92407
Step: 16  accuracy: 0.47  loss: 1.96254
Step: 17  accuracy: 0.69  loss: 1.70013
Step: 18  accuracy: 0.64  loss: 1.70509
Step: 19  accuracy: 0.55  loss: 1.67897
Step: 20  accuracy: 0.54  loss: 1.67961
```

```
Step: 532  accuracy: 0.899999988079  loss: 0.772873
Step: 533  accuracy: 0.889999997616  loss: 0.778061
Step: 534  accuracy: 0.959999990463  loss: 0.612538
Step: 535  accuracy: 0.769999992847  loss: 1.05344
Step: 536  accuracy: 0.880000007153  loss: 0.783344
Step: 537  accuracy: 0.839999985695  loss: 0.886219
Step: 538  accuracy: 0.839999985695  loss: 0.902399
Step: 539  accuracy: 0.930000019073  loss: 0.682709
Step: 540  accuracy: 0.880000007153  loss: 0.754257
Step: 541  accuracy: 0.839999985695  loss: 0.853037
Step: 542  accuracy: 0.889999997616  loss: 0.750142
Step: 543  accuracy: 0.899999988079  loss: 0.772959
Step: 544  accuracy: 0.95  loss: 0.638693
Step: 545  accuracy: 0.940000009537  loss: 0.665755
Step: 546  accuracy: 0.930000019073  loss: 0.687064
Step: 547  accuracy: 0.95  loss: 0.636063
Step: 548  accuracy: 0.899999988079  loss: 0.737802
Step: 549  accuracy: 0.92000002861  loss: 0.667991
2019-05-27 20:45:25.329292: I T:\src\github\tensorflow\tensorflc
2019-05-27 20:45:25.329881: I T:\src\github\tensorflow\tensorflc
2019-05-27 20:45:25.330791: I T:\src\github\tensorflow\tensorflc
2019-05-27 20:45:25.331211: I T:\src\github\tensorflow\tensorflc
2019-05-27 20:45:25.331725: I T:\src\github\tensorflow\tensorflc
Test Accuracy = 0.898
```

2.进阶作业

Lenet 输入为改为 28x28,直接使用 minist 为 28x28 而不对数据进行填充。卷积、池化的 padding 方式依然是
padding='VALID' （就是 no padding）
在保持 fc1 的输出任然是 120 的时候修改第 1 个 fc1 的输入维度，其他层不变，使得网络依然可以进行训练和预测。

实验过程：

Mnist 为 28x28 而不对数据进行填充时，fc1 输入维度更改为 256，Lenet 定义输入数据时改变尺寸大小，即可：

```
self.fc1_relu=self._fully_connected_layer('layer_5_fc1','fc1_w','fc1_b',self.pool2,(256,120),[120])
```

```
  def _setup_placeholders_graph(self):
            self.x =tf.placeholder(tf.float32, (None, 28, 28, 1),name='input_x')
            self.y= tf.placeholder(tf.int32, (None))
```

具体可见源码中 lenet2.py 和 train2.py 部分。

实验结果如下：

```
Step: 533  accuracy: 0.909999978542  loss: 0.70954
Step: 534  accuracy: 0.899999988079  loss: 0.781165
Step: 535  accuracy: 0.880000007153  loss: 0.774042
Step: 536  accuracy: 0.930000019073  loss: 0.665858
Step: 537  accuracy: 0.839999985695  loss: 0.900459
Step: 538  accuracy: 0.810000014305  loss: 0.947212
Step: 539  accuracy: 0.909999978542  loss: 0.729936
Step: 540  accuracy: 0.820000004768  loss: 0.911797
Step: 541  accuracy: 0.95  loss: 0.601933
Step: 542  accuracy: 0.899999988079  loss: 0.724224
Step: 543  accuracy: 0.940000009537  loss: 0.631293
Step: 544  accuracy: 0.880000007153  loss: 0.764233
Step: 545  accuracy: 0.849999976158  loss: 0.803616
Step: 546  accuracy: 0.930000019073  loss: 0.635106
Step: 547  accuracy: 0.800000023842  loss: 0.961415
Step: 548  accuracy: 0.95  loss: 0.661241
Step: 549  accuracy: 0.899999988079  loss: 0.770469
2019-05-27 20:49:39.583861: I T:\src\github\tensorflow\tensorf
2019-05-27 20:49:39.584465: I T:\src\github\tensorflow\tensorf
2019-05-27 20:49:39.585060: I T:\src\github\tensorflow\tensorf
2019-05-27 20:49:39.585477: I T:\src\github\tensorflow\tensorf
2019-05-27 20:49:39.586040: I T:\src\github\tensorflow\tensorf
Test Accuracy = 0.892
```