华东师范大学数据学院人工智能导论实验报告

课程名称：人工智能	**年级：2017**
指导教师：罗轶凤	**姓名：熊双宇**

任务一：基于RNN实现文本分类任务

一、目的与使用环境

1. 数据使用搜狐新闻数据(SogouCS, 网址：http://www.sogou.com/labs/resource/cs.php)。
2. 任务重点在于搭建并训练RNN网络来提取特征，最后通过一个全连接层实现分类目标。
3. 使用环境：
   - Tensorflow1.13.1
   - Anaconda4.6.14

二、内容与设计思想

利用LSTM模型

1. **数据预处理**
   - 读取数据：

     ```
     data=pd.read_csv("sohu.csv")
     ```

   - 检查样本数的分布

     ```
     dic1=dict(zip(*np.unique(data['label'],return_counts=True)))
     print(dic1)

      # 输出
         {'business': 1051, 'news': 2989, 'pic': 312, 'sports': 1200, 'yule': 185}
     ```

   - 删除样本数少的标签

     ```
     n=data.shape[0]
      for i in range(n):
         s=data["label"][i]
         if s == 'cul'or s == 'mil' or s == 'caipiao':
             data.drop([i],axis=0,inplace=True)
     ```

2. **jieba分词**

- 先将series类型转化为列表

```
data_lines=data['text'].tolist()
data_labels=data['label'].tolist()
```

- 用函数pseg.cut(txt)分词

```
for line in data_lines:
    num += 1
    if num%100 == 0:
        print(num)
    words = pseg.cut(line)
    line0 = []
    # 除去标点符号（除去词性为'x'的词）
    for w in words:
        if 'x' != w.flag:
            line0.append(w.word)
    data_words.append(' '.join(line0))
```

- 将结果存储在文件中

```
f1 = open('data.txt','w')
f1.write('\n'.join(data_words))
```

3. **向量化数据**
    - keras.Tokenizer 将新闻文档处理成 单词索引序列
    - keras.pad_sequences 实现用 0 填充长度不足 100 的新闻（在前端填充）
    - preprocessing.LabelEncoder() 将标签映射为数字

```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.utils import to_categorical
import numpy as np
from sklearn import preprocessing

def preprocess_text(all_texts,all_labels):
    '''
    input: 所有text，所有labels
    output: 返回word_index字典，data序列和编码后的labels
    '''
    tokenizer = Tokenizer() #转化为索引序列
    tokenizer.fit_on_texts(all_texts)
    sequences = tokenizer.texts_to_sequences(all_texts)
    word_index = tokenizer.word_index
    print('Found %s unique tokens.' % len(word_index))
    data = pad_sequences(sequences, maxlen=MAX_SEQUENCE_LENGTH)

    labels=preprocessing.LabelEncoder().fit_transform(all_labels)
    print('Shape of data tensor:', data.shape)
    print('Shape of label tensor:', labels.shape)
    # 返回word_index字典，data序列和编码后的labels
    return word_index, data, labels

data_word_index, data_pd_sequence, data_labels_encode=preprocess_text(data_lines,
data_labels)
```

4. **处理样本分布不均：Oversampling**
   - 使用ADASYN

```
from matplotlib import pyplot as plt
from imblearn.over_sampling import ADASYN
from collections import Counter
def makeOverSamplesADASYN(X,y):
    '''
    input: origin data
    output: data after resampling
    '''
    print('Original dataset shape %s' % Counter(y))
    sm = ADASYN()
    X_res, y_res = sm.fit_sample(X, y)
    print('Resampled dataset shape %s' % Counter(y_res))
    print('ADASYN {}'.format(Counter(y_res)))
    return(X_res,y_res)
```

5. **将映射后的数字处理成 one-hot 向量**
   - keras.to_categorical

```
data_labels_ot=to_categorical(np.asarray(data_labels_res))
```

6. **划分训练集、验证集和测试集**
   - 设置划分的比例

```
VALIDATION_SPLIT=0.3
TEST_SPLIT=0.1
```

   - 划分集合

```
p1 = int(len(data_data_res)*(1-VALIDATION_SPLIT-TEST_SPLIT))
p2 = int(len(data_data_res)*(1-TEST_SPLIT))
train_data,validation_data,
test_data=data_data_res[:p1],data_data_res[p1:p2],data_data_res[p2:]
train_labels_ot, validation_labels_ot,
test_labels_ot=data_labels_ot[:p1],data_labels_ot[p1:p2],data_labels_ot[p2:]
```

7. **搭建模型并保存为'lstm.h5'**
   - Embedding层:将文本处理为向量：一个新闻doc被处理为一个100*200的二维向量[100为新闻固定长度，200为该单词在空间中的词向量]
   - Convolutional层、Pooling层：缩小向量长度

   - Flattening层：将2维空间压缩到1维
   - 两层Dense层：将向量长度收缩到8上，对应新闻分类的8个类

   ```
   from keras.layers import Dense, Input, Flatten, Dropout
   from keras.layers import LSTM, Embedding
   from keras.models import Sequential

   model = Sequential()
   model.add(Embedding(len(data_word_index) + 1, EMBEDDING_DIM,
   input_length=MAX_SEQUENCE_LENGTH))
   model.add(LSTM(200, dropout=0.2, recurrent_dropout=0.2))
   model.add(Dropout(0.2))
   model.add(Dense(data_labels_ot.shape[1], activation='softmax'))
   model.summary()


   model.compile(loss='categorical_crossentropy',
                 optimizer='rmsprop',
                 metrics=['acc'])
   print (model.metrics_names)
   model.fit(train_data, train_labels_ot, validation_data=(validation_data,
   validation_labels_ot), epochs=2, batch_size=10)
   model.save('lstm.h5')
   ```

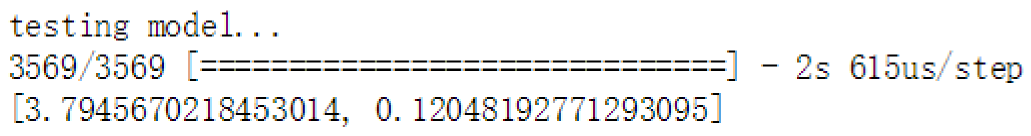
三、测试结果

-
  ```
  print ("testing model...")
  print (model.evaluate(test_data, test_labels_ot))
  ```

  输出：

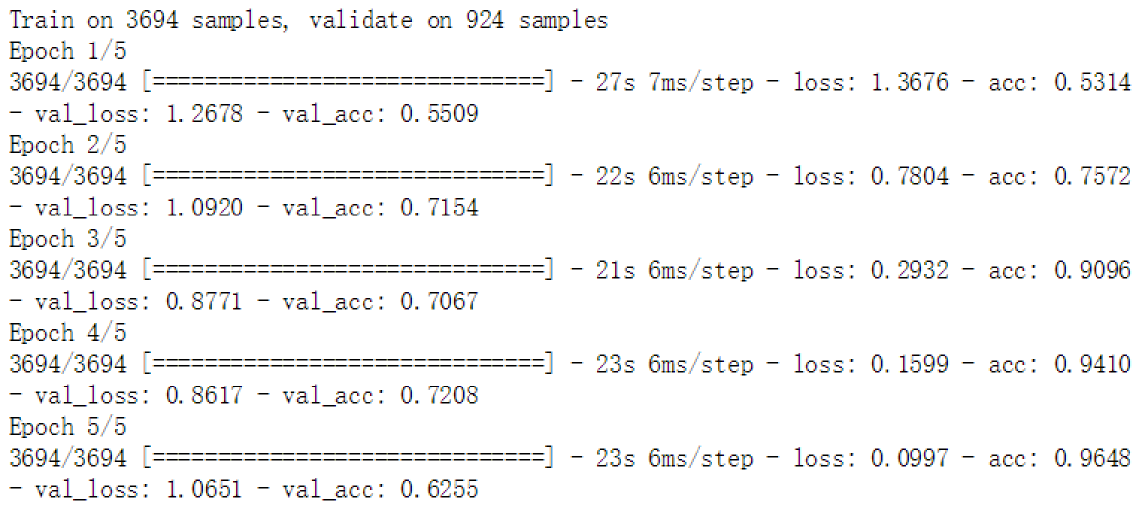
  - 使用过采样：
    - 过采样结果：
      ```
      Original dataset shape Counter({2: 2989, 4: 1200, 0: 1051, 3: 312, 5: 185, 1: 36})
      Resampled dataset shape Counter({3: 3078, 5: 2998, 1: 2990, 2: 2989, 4: 2989, 0: 2797})
      ```
    - 模型训练：epochs=5，batch_size=100
      ```
      ['loss', 'acc']
      Train on 11418 samples, validate on 2854 samples
      Epoch 1/5
      11418/11418 [==============================] - 73s 6ms/step - loss: 1.4349 - acc: 0.390
      9 - val_loss: 2.0697 - val_acc: 0.3357
      Epoch 2/5
      11418/11418 [==============================] - 75s 7ms/step - loss: 0.7705 - acc: 0.728
      6 - val_loss: 1.6366 - val_acc: 0.4636
      Epoch 3/5
      11418/11418 [==============================] - 75s 7ms/step - loss: 0.2964 - acc: 0.907
      8 - val_loss: 1.9499 - val_acc: 0.4639
      Epoch 4/5
      11418/11418 [==============================] - 76s 7ms/step - loss: 0.1424 - acc: 0.957
      6 - val_loss: 2.6150 - val_acc: 0.3238
      Epoch 5/5
      11418/11418 [==============================] - 73s 6ms/step - loss: 0.0856 - acc: 0.972
      1 - val_loss: 1.6010 - val_acc: 0.5333
      ```

        - 测验结果



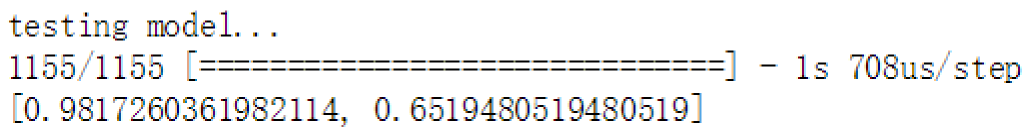
```
testing model...
3569/3569 [==============================] - 2s 615us/step
[3.7945670218453014, 0.1204819277129309 5]
```

    - 未使用过采样
        - 模型训练：



```
Train on 3694 samples, validate on 924 samples
Epoch 1/5
3694/3694 [==============================] - 27s 7ms/step - loss: 1.3676 - acc: 0.5314
- val_loss: 1.2678 - val_acc: 0.5509
Epoch 2/5
3694/3694 [==============================] - 22s 6ms/step - loss: 0.7804 - acc: 0.7572
- val_loss: 1.0920 - val_acc: 0.7154
Epoch 3/5
3694/3694 [==============================] - 21s 6ms/step - loss: 0.2932 - acc: 0.9096
- val_loss: 0.8771 - val_acc: 0.7067
Epoch 4/5
3694/3694 [==============================] - 23s 6ms/step - loss: 0.1599 - acc: 0.9410
- val_loss: 0.8617 - val_acc: 0.7208
Epoch 5/5
3694/3694 [==============================] - 23s 6ms/step - loss: 0.0997 - acc: 0.9648
- val_loss: 1.0651 - val_acc: 0.6255
```

        - 测验结果：



```
testing model...
1155/1155 [==============================] - 1s 708us/step
[0.9817260361982114, 0.6519480519480519]
```

任务二：基于CIFAR-10数据集使用CNN完成图像分类任务

一、数据集

1. 数据集：（https://www.cs.toronto.edu/~kriz/cifar.html）

二、使用环境

- Tensorflow1.13.1
- Anaconda4.6.14

三、内容与设计思想

1. **下载数据集**
    - 设置下载后文件夹名称

    ```
    cifar10_dataset_folder_path = 'cifar-10-batches-py'
    ```

    - 定义类继承tqdm，并且定义一个成员函数

```
class DownloadProgress(tqdm):
    last_block = 0
    #reporthook definition
    def hook(self, block_num=1, block_size=1, total_size=None):
        '''回调函数
        @a:已经下载的数据块
        @b:远程文件的大小
        @c:数据块的大小
        '''
        self.total = total_size
        self.update((block_num - self.last_block) * block_size)
        self.last_block = block_num
```

- 若不存在数据集cifar-10-python.tar.gz则利用urlretrieve()下载数据集

```
if not isfile('cifar-10-python.tar.gz'):
    with DownloadProgress(unit='B', unit_scale=True, miniters=1, desc='CIFAR-10
Dataset') as pbar:
        urlretrieve(
            'https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz',
            'cifar-10-python.tar.gz',
            pbar.hook)
```

2. **创建字符串标签**

   -
     ```
     def load_label_names():
         return ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',
     'ship', 'truck']
     ```

3. **展示数据**

   - batch_id: id for batch(1-5)
   2. sample_id: id for a image and label pair in the batch

4. **数据预处理**

   - tensorflow expect input with the argument, data_format. "NHWC": [batch, height, width, channels], "NCHW": [batch, channels, height, width].

     reshape

     1. 将行向量 (3072) 划分为 3 个片段. 每个片段对应一个channel.
     - 结果为 (3 x 1024) 维度的tensor [3个channels]
     2. 划分每一个tensor为 32. 32 意味着图像的宽度（像素）
     - 结果为 (3 x 32 x 32)(num_channel, width, height)

     transpose

     1. 将(num_channel, width, height)转化为(width, height, num_channel)：
     - 调用transpose(2,0,1)

```
def load_cfar10_batch(cifar10_dataset_folder_path, batch_id):
    with open(cifar10_dataset_folder_path + '/data_batch_' + str(batch_id),
mode='rb') as file:
        # note the encoding type is 'latin1'
        batch = pickle.load(file, encoding='latin1')

    features = batch['data'].reshape((len(batch['data']), 3, 32, 32)).transpose(0,
2, 3, 1)
    labels = batch['labels']

    return features, labels
```

- **标准化Normalize**
    1. Min-Max Normalization
        - this simply makes all x values to range between 0 and 1.
        - y = (x-min) / (max-min)
    2. why normalization should be performed
        - somewhat related to activation function.
            - sigmoid activation function takes an input value and outputs a new value ranging from 0 to 1.
                - if input value is large, the output value easily reaches 1;
                - if input small,output easily reaches 0
            - ReLU activation function takes an input value and outputs a new value ranging from 0 to infinity.
                - if input value is large, the output value increases linearly;
                - if input small, output easily reaches 0
    3. All the image data originally ranges from 0 to 255.
        - when it is passed into sigmoid function, the output is almost always 1
        - when it is passed into ReLu function, the output could be very huge.
        - When backpropagation process is performed to optimize the networks, this could lead to an exploding gradient which leads to an aweful learning steps. In order to avoid this issue, ideally, it is better let all the values be around 0 and 1.

```
def normalize(x):
    """
        argument
            - x: input image data in numpy array [32, 32, 3]
        return
            - normalized x
    """
    min_val = np.min(x)
    max_val = np.max(x)
    x = (x-min_val) / (max_val-min_val)
    return x
```

- **one-hot编码**

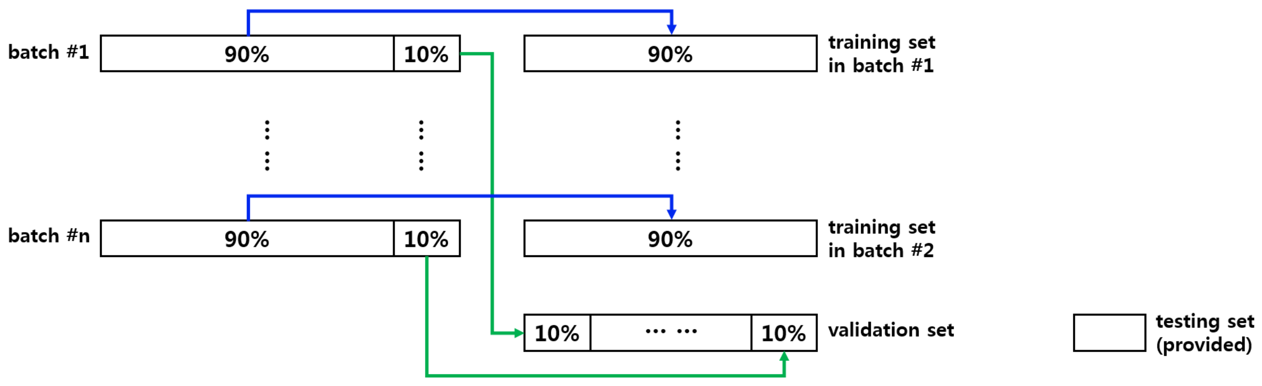
CIFAR-10 提供10 个不同的类别，因此我们需要一个(10,)的向量，每个分量表示预测的类别，当分量为1是表示该图像对应的下标所表示的类别

```
def one_hot_encode(x):
    """
        argument
            - x: a list of labels
        return
            - one hot encoding matrix (number of labels, number of class)
    """
    encoded = np.zeros((len(x), 10))

    for idx, val in enumerate(x):
        encoded[idx][val] = 1

    return encoded
```

5. **划分训练集和检验集，读取测试集**



6. **模型搭建**

   - 创建tf.Tensor

```
import tensorflow as tf
# Remove previous weights, bias, inputs, etc..
tf.reset_default_graph()

# Inputs
x = tf.placeholder(tf.float32, shape=(None, 32, 32, 3), name='input_x')
y =  tf.placeholder(tf.float32, shape=(None, 10), name='output_y')
keep_prob = tf.placeholder(tf.float32, name ='keep_prob')
```

   - 设置超参数

```
epochs = 10 #迭代次数
batch_size = 128
keep_probability = 0.7
learning_rate = 0.001
```

   - 模型结构

    1. convolution with 64 different filters in size of (3*3)
    2. max pooling by 2
        - ReLU activation function
        - batch normalization
    3. convolution with 128 different filters in size of (3*3)
    4. max-pooling by 2
        - ReLU activation function
        - batch normalization
    5. convolution with 256 different filters in size of (5*5)
    6. max-pooling by 2
        - ReLU activation function
        - batch normalization
    7. convolution with 513different filters in size of (5*5)
    8. max-pooling by 2
        - ReLU activation function
        - batch normalization
    9. flatten
    10. fully connected with 128 units
        - dropout
        - batch normalization
    11. fully connected with 256 units
        - dropout
        - batch normalization
    12. fully connected with 512 units
        - dropout
        - batch normalization
    13. fully connected with 1024 units
        - dropout
        - batch normalization
    14. fully connected with 10units

7. **定义损失函数和优化函数**
    - **cost**:
        - cost=tf.reduce_mean returns => The reduced **Tensor**
    - **optimizer**:
        - optimizer=tf.train.AdamOptimizer returns => An **Operation** that applies the specified gradients.
    - **accuracy**:
        - accuracy=tf.reduce_mean returns => The reduced **Tensor**
8. **训练模型**
    - 定义模型存储路径

```
save_model_path = './image_classification'
```

- 将每个batch划分为batch_size个batches并且返回

```
def batch_features_labels(features, labels, batch_size):
    """
    划分特征和标签至batches
    """
    for start in range(0, len(features), batch_size):
        end = min(start + batch_size, len(features))
        yield features[start:end], labels[start:end]

def load_preprocess_training_batch(batch_id, batch_size):
    """
    加载预处理的training data并且返回batches，返回的大小是batch_size或者更小
    """
    filename = 'preprocess_batch_' + str(batch_id) + '.p'
    features, labels = pickle.load(open(filename, mode='rb'))

    # Return the training data in batches of size <batch_size> or less
    return batch_features_labels(features, labels, batch_size)
```

  训练模型

```
 for batch_i in range(1, n_batches + 1):
            for batch_features, batch_labels in
load_preprocess_training_batch(batch_i, batch_size):
                train_neural_network(sess, optimizer, keep_probability,
batch_features, batch_labels)

            print('Epoch {:>2}, CIFAR-10 Batch {}:  '.format(epoch + 1, batch_i),
end='')
            print_stats(sess, batch_features, batch_labels, cost, accuracy)
```

  存储模型

```
 Save Model
    saver = tf.train.Saver()
    save_path = saver.save(sess, save_model_path)
```

9. **测试模型**

- 定义hyper-parameters

```
batch_size = 64
n_samples = 10
top_n_predictions = 5
```

- 加载测试集

```
test_features, test_labels = pickle.load(open('preprocess_test.p', mode='rb'))
```

- 加载模型

```
save_model_path = './image_classification'
loader = tf.train.import_meta_graph(save_model_path + '.meta')
```

- 获取tensor

```
loaded_x = loaded_graph.get_tensor_by_name('input_x:0')
loaded_y = loaded_graph.get_tensor_by_name('output_y:0')
loaded_keep_prob = loaded_graph.get_tensor_by_name('keep_prob:0')
loaded_logits = loaded_graph.get_tensor_by_name('logits:0')
loaded_acc = loaded_graph.get_tensor_by_name('accuracy:0')
```

- 获取精度

```
test_batch_acc_total = 0
test_batch_count = 0

for train_feature_batch, train_label_batch in batch_features_labels(test_features,
test_labels, batch_size):
            test_batch_acc_total += sess.run(
                loaded_acc,
                feed_dict={loaded_x: train_feature_batch, loaded_y:
train_label_batch, loaded_keep_prob: 1.0})
            test_batch_count += 1
print('Testing Accuracy: {}\n'.format(test_batch_acc_total/test_batch_count))
```

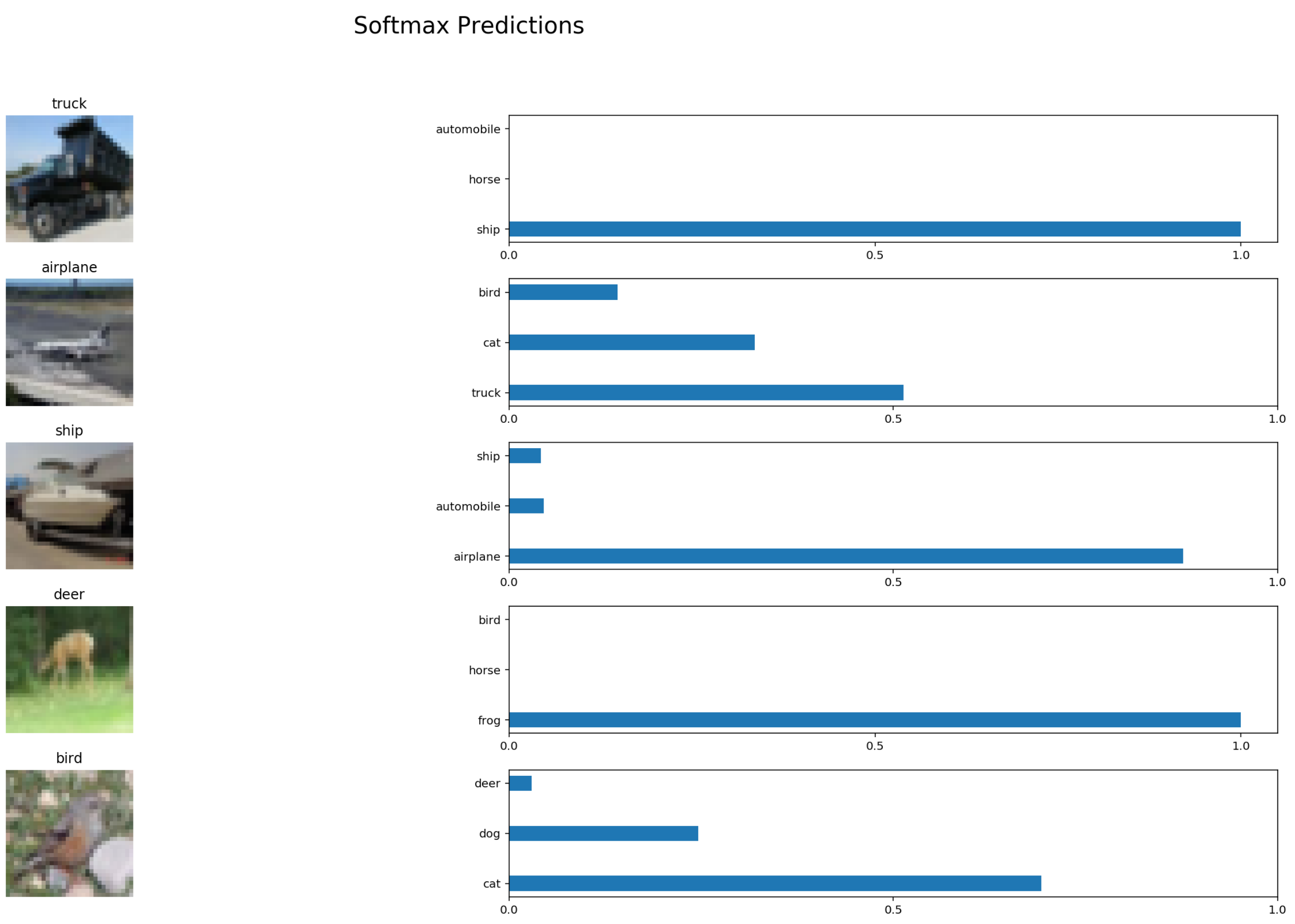
10. **结果**

- 迭代结果

```
Training...
Epoch  1, CIFAR-10 Batch 1:  Loss:     2.2066 Validation Accuracy: 0.207000
Epoch  1, CIFAR-10 Batch 2:  Loss:     1.9811 Validation Accuracy: 0.178400
Epoch  1, CIFAR-10 Batch 3:  Loss:     1.7738 Validation Accuracy: 0.222600
Epoch  1, CIFAR-10 Batch 4:  Loss:     1.7428 Validation Accuracy: 0.273800
Epoch  1, CIFAR-10 Batch 5:  Loss:     1.5097 Validation Accuracy: 0.283600
Epoch  2, CIFAR-10 Batch 1:  Loss:     1.6550 Validation Accuracy: 0.389400
Epoch  2, CIFAR-10 Batch 2:  Loss:     1.4150 Validation Accuracy: 0.422000
Epoch  2, CIFAR-10 Batch 3:  Loss:     1.1440 Validation Accuracy: 0.479600
Epoch  2, CIFAR-10 Batch 4:  Loss:     1.2207 Validation Accuracy: 0.539800
Epoch  2, CIFAR-10 Batch 5:  Loss:     0.9834 Validation Accuracy: 0.569200
Epoch  3, CIFAR-10 Batch 1:  Loss:     0.8408 Validation Accuracy: 0.593200
Epoch  3, CIFAR-10 Batch 2:  Loss:     0.7868 Validation Accuracy: 0.597800
Epoch  3, CIFAR-10 Batch 3:  Loss:     0.6302 Validation Accuracy: 0.622000
Epoch  3, CIFAR-10 Batch 4:  Loss:     0.7629 Validation Accuracy: 0.625600
Epoch  3, CIFAR-10 Batch 5:  Loss:     0.6366 Validation Accuracy: 0.661800
Epoch  4, CIFAR-10 Batch 1:  Loss:     0.5317 Validation Accuracy: 0.672400
Epoch  4, CIFAR-10 Batch 2:  Loss:     0.4004 Validation Accuracy: 0.679600
Epoch  4, CIFAR-10 Batch 3:  Loss:     0.3144 Validation Accuracy: 0.672200
Epoch  4, CIFAR-10 Batch 4:  Loss:     0.4148 Validation Accuracy: 0.716400
Epoch  4, CIFAR-10 Batch 5:  Loss:     0.2478 Validation Accuracy: 0.718600
Epoch  5, CIFAR-10 Batch 1:  Loss:     0.2822 Validation Accuracy: 0.700000
Epoch  5, CIFAR-10 Batch 2:  Loss:     0.1639 Validation Accuracy: 0.721800
Epoch  5, CIFAR-10 Batch 3:  Loss:     0.1673 Validation Accuracy: 0.702000
Epoch  5, CIFAR-10 Batch 4:  Loss:     0.2547 Validation Accuracy: 0.708200
Epoch  5, CIFAR-10 Batch 5:  Loss:     0.1661 Validation Accuracy: 0.680400
Epoch  6, CIFAR-10 Batch 1:  Loss:     0.1799 Validation Accuracy: 0.702000
Epoch  6, CIFAR-10 Batch 2:  Loss:     0.0867 Validation Accuracy: 0.722000
Epoch  6, CIFAR-10 Batch 3:  Loss:     0.0765 Validation Accuracy: 0.728400
Epoch  6, CIFAR-10 Batch 4:  Loss:     0.1362 Validation Accuracy: 0.706400
Epoch  6, CIFAR-10 Batch 5:  Loss:     0.0491 Validation Accuracy: 0.742600
Epoch  7, CIFAR-10 Batch 1:  Loss:     0.1421 Validation Accuracy: 0.720800
Epoch  7, CIFAR-10 Batch 2:  Loss:     0.0508 Validation Accuracy: 0.739200
Epoch  7, CIFAR-10 Batch 3:  Loss:     0.0405 Validation Accuracy: 0.742800
Epoch  7, CIFAR-10 Batch 4:  Loss:     0.0733 Validation Accuracy: 0.732000
Epoch  7, CIFAR-10 Batch 5:  Loss:     0.0303 Validation Accuracy: 0.735200
Epoch  8, CIFAR-10 Batch 1:  Loss:     0.0927 Validation Accuracy: 0.742400
Epoch  8, CIFAR-10 Batch 2:  Loss:     0.0657 Validation Accuracy: 0.715200
Epoch  8, CIFAR-10 Batch 3:  Loss:     0.0397 Validation Accuracy: 0.719400
Epoch  8, CIFAR-10 Batch 4:  Loss:     0.0928 Validation Accuracy: 0.722200
Epoch  8, CIFAR-10 Batch 5:  Loss:     0.0113 Validation Accuracy: 0.743800
Epoch  9, CIFAR-10 Batch 1:  Loss:     0.0463 Validation Accuracy: 0.737000
Epoch  9, CIFAR-10 Batch 2:  Loss:     0.0132 Validation Accuracy: 0.723400
Epoch  9, CIFAR-10 Batch 3:  Loss:     0.0539 Validation Accuracy: 0.734200
Epoch  9, CIFAR-10 Batch 4:  Loss:     0.0324 Validation Accuracy: 0.743800
Epoch  9, CIFAR-10 Batch 5:  Loss:     0.0107 Validation Accuracy: 0.732000
Epoch 10, CIFAR-10 Batch 1:  Loss:     0.0279 Validation Accuracy: 0.745600
Epoch 10, CIFAR-10 Batch 2:  Loss:     0.0117 Validation Accuracy: 0.743000
Epoch 10, CIFAR-10 Batch 3:  Loss:     0.0052 Validation Accuracy: 0.731800
Epoch 10, CIFAR-10 Batch 4:  Loss:     0.0055 Validation Accuracy: 0.745200
Epoch 10, CIFAR-10 Batch 5:  Loss:     0.0104 Validation Accuracy: 0.738000
```

四、测试结果

```
INFO:tensorflow:Restoring parameters from ./image_classification
Testing Accuracy: 0.7254179936305732
```



任务三：基于MNIST数据集使用GAN实现手写图像生成的任务

一、数据集与使用环境

1. 数据集：（http://yann.lecun.com/exdb/mnist/）
2. 使用环境：
    - Tensorflow1.13.1
    - Anaconda4.6.14

二、内容与设计思想

1. **generator**

- take random noise as input and generate a fake image which look like the real training data set.
- input dimension
    - the number of rows is the number of the observations
    - the columns is the random noise dimensions
- output dimension
    - the dimension of the image
- relu activation

2. **discriminator**

- sometimes takes the real image(training), and sometimes takes the fake image(generator) as input
- tries to discriminate whether the image is real or fake
- input dimension
    - image dimension
- output dimension
    - one dimension: 1 or 0(whether it is real or fake)
- sigmoid activation

3. **optimal**

- generator should be good in creating a new image which looks alike the training data set
- discriminator should discriminate better

4. **parameters**
    - image dimension(discriminator dimension): 28*28=784
    - inside the generator there is a layer with dimension: 256
    - inside the discriminator there is a layer with dimension: 256
5. **matrix multiply**

    None: batch_size

    - generator:
        - [None, noise_dim] * [noise_dim, gen_dim] =[None, gen_dim]
        - [None, gen_dim] * [gen_dim, img_dim] = [None, image_dim]
    - discriminator:
        - [None, image_dim] * [image_dim, disc_dim] = [None, disc_dim]
        - [None, disc_dim] * [disc_dim, 1] = [None, 1]

三、实现过程

1. **下载数据**

```
mnist=input_data.read_data_sets("MNIST_Data/",one_hot=True)
```

2. **设置超参数**

```
batch_size=128
learning_rate=2e-4
display_step=20
num_steps=80000

img_dim=784 #not 28*28,but is 784(a straight line vector)
disc_dim=256
noise_dim=100
gen_dim=256
```

3. **声明tensor**

```
gen_inp = tf.placeholder(tf.float32,shape=[None,noise_dim])#the number row is the number
of observations(batch size)
disc_inp=tf.placeholder(tf.float32,shape=[None,img_dim])
```

4. **初始化weights**

```
def weight_init(shape):
    return tf.random_normal(shape=shape,stddev=1. / tf.sqrt(shape[0] / 2.))
```

```
w={
    "w1":tf.Variable(weight_init([noise_dim,gen_dim])),
    "w2":tf.Variable(weight_init([gen_dim,img_dim])),
    "w3":tf.Variable(weight_init([img_dim,disc_dim])),
    "w4":tf.Variable(weight_init([disc_dim,1]))
}
b={
    "b1":tf.Variable(tf.zeros([gen_dim])),
    "b2":tf.Variable(tf.zeros([img_dim])),
    "b3":tf.Variable(tf.zeros([disc_dim])),
    "b4":tf.Variable(tf.zeros([1]))
}
```

5. **建立模型**

   - generator
     - reLU
     - sigmoid

   ```
   def gen_fun(x):
       h1=tf.matmul(x,w["w1"])
       h1=tf.add(h1,b["b1"])
       h1=tf.nn.relu(h1)

       h1=tf.matmul(h1,w["w2"])
       h1=tf.add(h1,b["b2"])
       h1=tf.nn.sigmoid(h1)

       return h1
   ```

   - discriminator
     - reLU
     - sigmoid

```
def disc_fun(x):
    h2=tf.matmul(x,w["w3"])
    h2=tf.add(h2,b["b3"])
    h2=tf.nn.relu(h2)

    h2=tf.matmul(h2,w["w4"])
    h2=tf.add(h2,b["b4"])
    h2=tf.nn.sigmoid(h2)

    return h2
```

6. **损失函数、优化函数和模型评估**
    - 定义generator和discriminator的参数变量

    ```
    vars_gen=[w["w1"],w["w2"],b["b1"],b["b2"]]
    vars_disc=[w["w3"],w["w4"],b["b3"],b["b4"]]
    ```

    - 定义输出变量

    ```
    gen_out=gen_fun(gen_inp)

    disc_real_out=disc_fun(disc_inp)
    disc_fake_out=disc_fun(gen_out)
    ```

    - cost function
        - if if disc_fake_out is 1, the generator generate a almost a same image , in this case, the tf.log will return 0, meaning the cost is 0
    - else if the disc_fake_out tends towards to zero, the tf.log will be very high

    ```
    cost_gen=-tf.reduce_mean(tf.log(disc_fake_out))
    cost_disc=-tf.reduce_mean(tf.log(disc_real_out)+tf.log(1.-disc_fake_out))
    ```

    - optimizer:

    ```
    optim_gen=tf.train.AdamOptimizer(learning_rate=learning_rate)
    optmi_disc=tf.train.AdamOptimizer(learning_rate=learning_rate)
    ```

    - minimize cost function

    ```
    training_gen=optim_gen.minimize(cost_gen, var_list=vars_gen)
    training_disc=optim_gen.minimize(cost_disc, var_list=vars_disc)
    ```

7. **训练模型**
    - 初始化变量

```
init =tf.global_variables_initializer()
```

- 生成noise给generator

```
noise_temp=np.random.uniform(-1.,1.,size=[batch_size,noise_dim])
```

- 运行优化器

```
feed_dict={disc_inp:batch_x, gen_inp: noise_temp}
_,_,gl,dl=sess.run([training_gen, training_disc, cost_gen, cost_disc],
                          feed_dict=feed_dict)
```

- 输出结果

```
if step % 2000 == 0 or step == 1:
            print("Step %i: Generator Loss: %f, Discriminator Loss: %f" %
(step,gl,dl))

    print("Finished!")
```

输出：

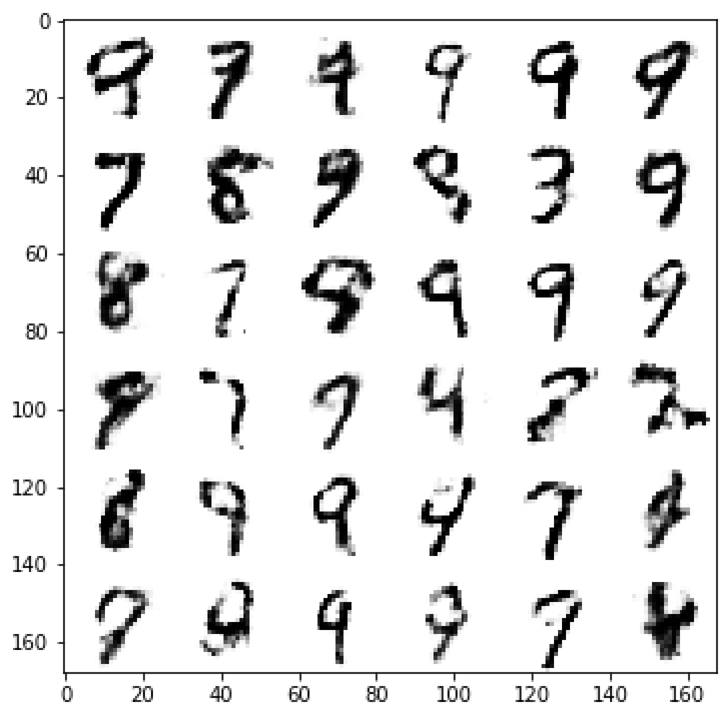
```
Step 1: Generator Loss: 1.174713, Discriminator Loss: 1.145784
Step 2000: Generator Loss: 5.084534, Discriminator Loss: 0.019912
Step 4000: Generator Loss: 4.039534, Discriminator Loss: 0.063871
Step 6000: Generator Loss: 4.140039, Discriminator Loss: 0.143743
Step 8000: Generator Loss: 4.645658, Discriminator Loss: 0.070311
Step 10000: Generator Loss: 4.126902, Discriminator Loss: 0.150643
Step 12000: Generator Loss: 4.166895, Discriminator Loss: 0.251455
Step 14000: Generator Loss: 4.333557, Discriminator Loss: 0.127097
Step 16000: Generator Loss: 4.250654, Discriminator Loss: 0.166644
Step 18000: Generator Loss: 3.869440, Discriminator Loss: 0.205827
Step 20000: Generator Loss: 3.800434, Discriminator Loss: 0.289734
Step 22000: Generator Loss: 4.157008, Discriminator Loss: 0.189885
Step 24000: Generator Loss: 3.889543, Discriminator Loss: 0.218796
Step 26000: Generator Loss: 4.074484, Discriminator Loss: 0.412890
Step 28000: Generator Loss: 3.008605, Discriminator Loss: 0.339312
Step 30000: Generator Loss: 3.648920, Discriminator Loss: 0.320491
Step 32000: Generator Loss: 3.554184, Discriminator Loss: 0.465728
Step 34000: Generator Loss: 3.443735, Discriminator Loss: 0.356761
Step 36000: Generator Loss: 2.991017, Discriminator Loss: 0.343976
Step 38000: Generator Loss: 3.041965, Discriminator Loss: 0.390022
Step 40000: Generator Loss: 2.643777, Discriminator Loss: 0.505138
Step 42000: Generator Loss: 3.184446, Discriminator Loss: 0.286290
Step 44000: Generator Loss: 3.054142, Discriminator Loss: 0.499869
Step 46000: Generator Loss: 3.214205, Discriminator Loss: 0.334696
Step 48000: Generator Loss: 2.757978, Discriminator Loss: 0.337420
Step 50000: Generator Loss: 2.776406, Discriminator Loss: 0.490548
Step 52000: Generator Loss: 2.547231, Discriminator Loss: 0.422968
Step 54000: Generator Loss: 2.723878, Discriminator Loss: 0.516140
Step 56000: Generator Loss: 3.019481, Discriminator Loss: 0.361977
Step 58000: Generator Loss: 2.613147, Discriminator Loss: 0.481575
Step 60000: Generator Loss: 3.196715, Discriminator Loss: 0.374996
Step 62000: Generator Loss: 2.741857, Discriminator Loss: 0.442101
Step 64000: Generator Loss: 2.784485, Discriminator Loss: 0.458598
Step 66000: Generator Loss: 3.133292, Discriminator Loss: 0.380699
Step 68000: Generator Loss: 3.020260, Discriminator Loss: 0.530613
Step 70000: Generator Loss: 2.489679, Discriminator Loss: 0.613974
Step 72000: Generator Loss: 2.756352, Discriminator Loss: 0.466526
Step 74000: Generator Loss: 2.722961, Discriminator Loss: 0.400713
Step 76000: Generator Loss: 2.960650, Discriminator Loss: 0.456019
Step 78000: Generator Loss: 3.226742, Discriminator Loss: 0.479321
Step 80000: Generator Loss: 2.898147, Discriminator Loss: 0.377330
Finished!
```

8. **测验模型**
    - 利用generator生成图片

```
n=6
   canvas = np.empty((28 * n, 28 * n))

   for i in range(n):
       # Noise input
       z=np.random.uniform(-1., 1., size=[n, noise_dim])
       # Generate image from noise
       g = sess.run(gen_out, feed_dict = {gen_inp:z})
       # Reverse colors for better display
       g = -1 * (g - 1)
       for j in range(n):
           # Draw the generated digits
           canvas[i * 28: (i+1)*28, j*28:(j+1)*28]=g[j].reshape([28,28])
   plt.figure(figsize=(n,n))
   plt.imshow(canvas, origin="upper", cmap="gray")
   plt.show
```

四、测试结果


0
20
40
60
80
100
120
140
160
0
20
40
60
80
100
120
140
160