1. Introduction

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import seaborn as sns
%matplotlib inline


from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
import itertools

from keras.utils.np_utils import to_categorical # convert to one-hot-encodin
g
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPool2D
from keras.optimizers import RMSprop, Adam, SGD
from keras.preprocessing.image import ImageDataGenerator
from keras.callbacks import ReduceLROnPlateau
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPool2D
from keras.layers.normalization import BatchNormalization
from keras.preprocessing.image import ImageDataGenerator
from keras.callbacks import ReduceLROnPlateau
from sklearn.model_selection import train_test_split

sns.set(style='white', context='notebook', palette='deep')
```

2. Data preparation

2.1 Load data

```
# >>>>>填写<<<< 利用pandas 的load_csv 函数，读取我们的train 和 test 数
据集合 变量已经给出 >>>>>填写<<<< ######
train = pd.read_csv("subset_train.csv")
test = pd.read_csv("Small_test.csv")
#####train validation test(完全独立的，与训练过程无关的)
# >>>>>填写<<<< 利用pandas 的header 选择，将label 列传递给Y_train
 >>>>>填写<<<<
```

```python
Y_train = train["label"]

Y_test = test['label']
# 因为train.csv 中，第一列label 在上述代码已经传递给Y_label，这里对于x_train 我们不需要训练集的第一列 #####
X_train = train.drop(labels = ["label"],axis = 1)
X_test = test.drop(labels = ["label"],axis = 1)
# 释放内存

X_train
```

Out[372]:

	pixel10	pixel11	pixel12	pixel13	pixel14	pixel15	pixel16	pixel17	pixel18	pixel19	...	pixel774	pixel775	pixel776	pixel777	pixel778	pixel779	pixel780	pixel781	pixel782	pixel783
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...

	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel774	pixel775	pixel776	pixel777	pixel778	pixel779	pixel780	pixel781	pixel782	pixel783
4195	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
4196	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
4197	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
4198	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
4199	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

4200 rows × 784 columns

We have similar counts for the 10 digits.

2.2 Check for null and missing values

```
# 检查训练数据是否有空值
X_train.isnull().any().describe()
```

Out[373]:

```
count       784
unique        1
```

```
top       False
freq        784
dtype: object
```

```
# >>>>填写<<<< 检查训练数据是否有空值 >>>>填写<<<< ###
X_test.isnull().any().describe()
```

Out[374]:

```
count       784
unique        1
top       False
freq        784
dtype: object
```

I check for corrupted images (missing values inside).

There is no missing values in the train and test dataset. So we can safely go ahead.

2.3 Normalization

We perform a grayscale normalization to reduce the effect of illumination's differences.

Moreover the CNN converg faster on [0..1] data than on [0..255]. 标准化，将灰度值 0-255 映射到 0 - 1 区间

```
# Normalize the data
X_train = X_train / 255.0
###### >>>填写<<< 标准化测试集合 #######
X_test = X_test / 255.0
X_train.shape
```

Out[375]:

```
(4200, 784)
```

2.3 Reshape

```
# >>>>填写<<<<< 利用 reshape 函数， 将X_train 变换成 (height = 28px, width = 28px , canal = 1)>>>>填写<<<<< ######
X_train = X_train.values.reshape(-1,28,28)
X_test = X_test.values.reshape(-1,28,28)

X_train.shape
```

Out[376]:

```
(4200, 28, 28)
```

Train and test images (28px x 28px) has been stock into pandas.Dataframe as 1D vectors of 784 values. We reshape all data to 28x28x1 3D matrices.

Keras requires an extra dimension in the end which correspond to channels. MNIST images are gray scaled so it use only one channel. For RGB images, there is 3 channels, we would have reshaped 784px vectors to 28x28x3 3D matrices.

2.5 Label encoding

```
# 利用0 1 编码 将0-9 数字标签编码成10 维向量 (ex : 9 -> [0,0,0,0,0,0,0,0,0,
1])
##
Y_train = to_categorical(Y_train, num_classes = 10)
Y_test = to_categorical(Y_test, num_classes = 10)
## one-hot encoding
```

Labels are 10 digits numbers from 0 to 9. We need to encode these lables to one hot vectors (ex : 2 -> [0,0,1,0,0,0,0,0,0,0]).

2.6 Split training and valdiation set

```
# Set the random seed
random_seed = 2
# 将训练集合按照9:1 分成训练集合 和验证集合 validation 10 折交叉验证 1
0-fold validation  ####
X_train, X_val, Y_train, Y_val = train_test_split(X_train, Y_train, test_size = 0.
1, random_state=random_seed)
```

We can get a better sense for one of these examples by visualising the image and looking at the label.

```
# Some examples #x-train 里面第一个sample 的 0:最大 0:最大 0 [:,:,0]
#g = plt.imshow(X_train[0][:,:,0],cmap='gray') #plt 为什么把灰度可以生
```

3. CNN

3.1 Define the model

Type *Markdown* and LaTeX: $\alpha 2$α2

```
batch_size = 100
num_classes = 10
epochs = 50
from keras.layers import SimpleRNN
model = Sequential()
model.add(SimpleRNN(128, input_shape = input_shape, return_sequences=Tr
ue))
model.add(Dropout(0.3))
model.add(SimpleRNN(128))
model.add(Dropout(0.2))

model.add(Dense(10,activation='softmax'))
X_train.shape
```

Out[388]:

```
(3780, 28, 28)
```

```
### 运行 model.summary（）回答下列问题 第二天课上一起讨论 ####
model.summary()x
```

```
Model: "sequential_42"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
simple_rnn_43 (SimpleRNN)    (None, 28, 128)           20096     
_________________________________________________________________
dropout_55 (Dropout)         (None, 28, 128)           0         
_________________________________________________________________
simple_rnn_44 (SimpleRNN)    (None, 128)               32896     
_________________________________________________________________
dropout_56 (Dropout)         (None, 128)               0         
_________________________________________________________________
dense_33 (Dense)             (None, 10)                1290      
=================================================================
Total params: 54,282
Trainable params: 54,282
Non-trainable params: 0
_________________________________________________________________
```

```
#优化器   尝试使用不同的优化器 至少以下三种 在 DL 一个调节的点
## 中文参考 https://keras.io/zh/optimizers/

## SGD(lr=0.01, momentum=0.0, decay=0.0, nesterov=False)
## RMSprop(lr=0.001, rho=0.9, epsilon=None, decay=0.0)
## Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None, decay=0.0, amsgrad=False)

optimizer = RMSprop(lr=0.001, rho=0.9, epsilon=0.0000001, decay=0.0)
```

```
### 将模型compile 编译
### 调节loss 参数，即loss function
### mean_squared_error
### categorical_crossentropy/为什么不用binary_crossentropy
###
### mean_absolute_error
model.compile(optimizer = optimizer , loss = "categorical_crossentropy", metrics=["accuracy"])

### training 过程中的 自动调节函数
### Reduce LR On Plateau = 减少学习率，当某一个参数达到一个平台期 自动的 把上面优化器中的 lr 减小

learning_rate_reduction = ReduceLROnPlateau(monitor='val_accuracy',
                                            patience=3,
                                            verbose=1,
                                            factor=0.5,
                                            min_lr=0.00001)
history = model.fit(X_train,Y_train, batch_size=batch_size,
                    epochs = epochs, validation_data = (X_val, Y_val))
```

```
Train on 3780 samples, validate on 420 samples
Epoch 1/50
3780/3780 [==============================] - 1s 384us/step
 - loss: 1.1583 - accuracy: 0.6069 - val_loss: 0.7660 - val
_accuracy: 0.7500
Epoch 2/50
3780/3780 [==============================] - 1s 235us/step
 - loss: 0.6869 - accuracy: 0.7833 - val_loss: 0.6638 - val
_accuracy: 0.7976
Epoch 3/50
3780/3780 [==============================] - 1s 255us/step
 - loss: 0.5320 - accuracy: 0.8304 - val_loss: 0.5354 - val
_accuracy: 0.8548
Epoch 4/50
3780/3780 [==============================] - 1s 267us/step
 - loss: 0.4496 - accuracy: 0.8590 - val_loss: 0.6949 - val
_accuracy: 0.7643
Epoch 5/50
```

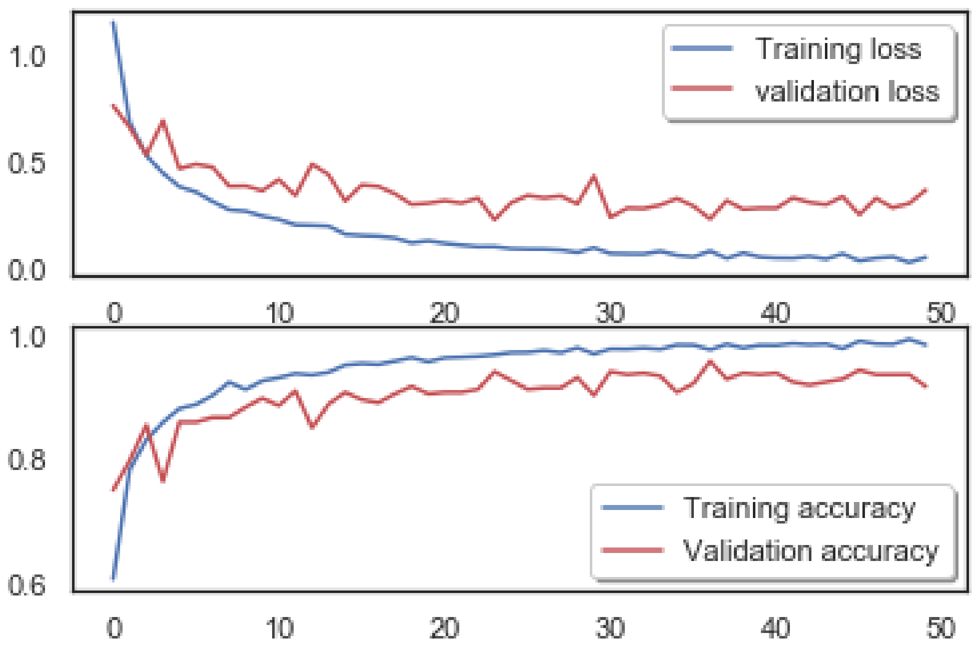
```
3780/3780 [==============================] - 1s 246us/step
 - loss: 0.3852 - accuracy: 0.8810 - val_loss: 0.4706 - val
_accuracy: 0.8595
Epoch 6/50
3780/3780 [==============================] - 1s 251us/step
 - loss: 0.3602 - accuracy: 0.8876 - val_loss: 0.4911 - val
_accuracy: 0.8595
Epoch 7/50
3780/3780 [==============================] - 1s 247us/step
 - loss: 0.3153 - accuracy: 0.9021 - val_loss: 0.4772 - val
_accuracy: 0.8667
Epoch 8/50
3780/3780 [==============================] - 1s 225us/step
 - loss: 0.2765 - accuracy: 0.9233 - val_loss: 0.3877 - val
_accuracy: 0.8667
Epoch 9/50
3780/3780 [==============================] - 1s 219us/step
 - loss: 0.2696 - accuracy: 0.9116 - val_loss: 0.3889 - val
_accuracy: 0.8833
Epoch 10/50
3780/3780 [==============================] - 1s 215us/step
 - loss: 0.2467 - accuracy: 0.9257 - val_loss: 0.3663 - val
_accuracy: 0.8976
Epoch 11/50
3780/3780 [==============================] - 1s 210us/step
 - loss: 0.2311 - accuracy: 0.9307 - val_loss: 0.4190 - val
_accuracy: 0.8857
Epoch 12/50
3780/3780 [==============================] - 1s 215us/step
 - loss: 0.2053 - accuracy: 0.9373 - val_loss: 0.3434 - val
_accuracy: 0.9095
Epoch 13/50
3780/3780 [==============================] - 1s 217us/step
 - loss: 0.2034 - accuracy: 0.9349 - val_loss: 0.4919 - val
_accuracy: 0.8500
Epoch 14/50
3780/3780 [==============================] - 1s 234us/step
 - loss: 0.1997 - accuracy: 0.9394 - val_loss: 0.4415 - val
_accuracy: 0.8881
Epoch 15/50
3780/3780 [==============================] - 1s 232us/step
 - loss: 0.1615 - accuracy: 0.9511 - val_loss: 0.3174 - val
_accuracy: 0.9071
Epoch 16/50
```

```
3780/3780 [==============================] - 1s 226us/step
 - loss: 0.1560 - accuracy: 0.9534 - val_loss: 0.3938 - val
_accuracy: 0.8952
Epoch 17/50
3780/3780 [==============================] - 1s 226us/step
 - loss: 0.1529 - accuracy: 0.9521 - val_loss: 0.3866 - val
_accuracy: 0.8905
Epoch 18/50
3780/3780 [==============================] - 1s 239us/step
 - loss: 0.1438 - accuracy: 0.9574 - val_loss: 0.3515 - val
_accuracy: 0.9048
Epoch 19/50
3780/3780 [==============================] - 1s 225us/step
 - loss: 0.1214 - accuracy: 0.9630 - val_loss: 0.3015 - val
_accuracy: 0.9167
Epoch 20/50
3780/3780 [==============================] - 1s 231us/step
 - loss: 0.1312 - accuracy: 0.9563 - val_loss: 0.3073 - val
_accuracy: 0.9048
Epoch 21/50
3780/3780 [==============================] - 1s 220us/step
 - loss: 0.1191 - accuracy: 0.9627 - val_loss: 0.3197 - val
_accuracy: 0.9071
Epoch 22/50
3780/3780 [==============================] - 1s 230us/step
 - loss: 0.1105 - accuracy: 0.9638 - val_loss: 0.3078 - val
_accuracy: 0.9071
Epoch 23/50
3780/3780 [==============================] - 1s 230us/step
 - loss: 0.1033 - accuracy: 0.9651 - val_loss: 0.3312 - val
_accuracy: 0.9119
Epoch 24/50
3780/3780 [==============================] - 1s 229us/step
 - loss: 0.1038 - accuracy: 0.9675 - val_loss: 0.2290 - val
_accuracy: 0.9405
Epoch 25/50
3780/3780 [==============================] - 1s 223us/step
 - loss: 0.0941 - accuracy: 0.9712 - val_loss: 0.3092 - val
_accuracy: 0.9262
Epoch 26/50
3780/3780 [==============================] - 1s 222us/step
 - loss: 0.0922 - accuracy: 0.9714 - val_loss: 0.3441 - val
_accuracy: 0.9119
Epoch 27/50
```

```
3780/3780 [==============================] - 1s 221us/step
 - loss: 0.0918 - accuracy: 0.9746 - val_loss: 0.3327 - val
_accuracy: 0.9143
Epoch 28/50
3780/3780 [==============================] - 1s 225us/step
 - loss: 0.0878 - accuracy: 0.9709 - val_loss: 0.3418 - val
_accuracy: 0.9143
Epoch 29/50
3780/3780 [==============================] - 1s 224us/step
 - loss: 0.0766 - accuracy: 0.9788 - val_loss: 0.3041 - val
_accuracy: 0.9310
Epoch 30/50
3780/3780 [==============================] - 1s 232us/step
 - loss: 0.0980 - accuracy: 0.9690 - val_loss: 0.4351 - val
_accuracy: 0.9024
Epoch 31/50
3780/3780 [==============================] - 1s 232us/step
 - loss: 0.0710 - accuracy: 0.9770 - val_loss: 0.2412 - val
_accuracy: 0.9405
Epoch 32/50
3780/3780 [==============================] - 1s 230us/step
 - loss: 0.0694 - accuracy: 0.9767 - val_loss: 0.2845 - val
_accuracy: 0.9357
Epoch 33/50
3780/3780 [==============================] - 1s 226us/step
 - loss: 0.0686 - accuracy: 0.9786 - val_loss: 0.2824 - val
_accuracy: 0.9381
Epoch 34/50
3780/3780 [==============================] - 1s 223us/step
 - loss: 0.0821 - accuracy: 0.9765 - val_loss: 0.2990 - val
_accuracy: 0.9333
Epoch 35/50
3780/3780 [==============================] - 1s 238us/step
 - loss: 0.0629 - accuracy: 0.9836 - val_loss: 0.3296 - val
_accuracy: 0.9071
Epoch 36/50
3780/3780 [==============================] - 1s 234us/step
 - loss: 0.0553 - accuracy: 0.9833 - val_loss: 0.2938 - val
_accuracy: 0.9214
Epoch 37/50
3780/3780 [==============================] - 1s 228us/step
 - loss: 0.0829 - accuracy: 0.9757 - val_loss: 0.2332 - val
_accuracy: 0.9571
Epoch 38/50
```

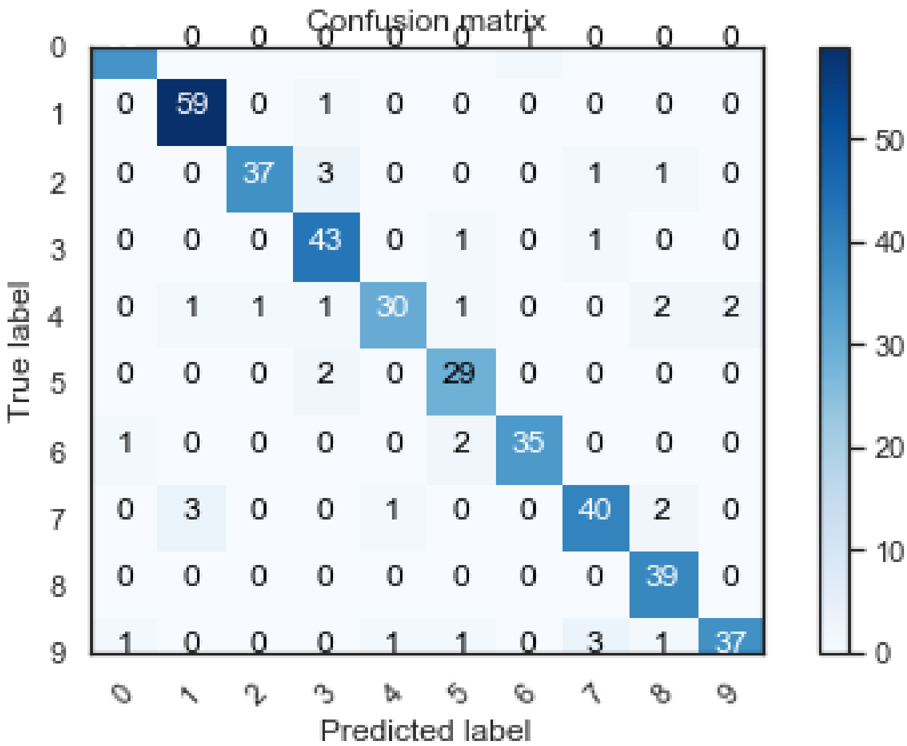
```
3780/3780 [==============================] - 1s 223us/step
 - loss: 0.0485 - accuracy: 0.9847 - val_loss: 0.3193 - val
_accuracy: 0.9286
Epoch 39/50
3780/3780 [==============================] - 1s 243us/step
 - loss: 0.0736 - accuracy: 0.9786 - val_loss: 0.2791 - val
_accuracy: 0.9381
Epoch 40/50
3780/3780 [==============================] - 1s 225us/step
 - loss: 0.0551 - accuracy: 0.9831 - val_loss: 0.2839 - val
_accuracy: 0.9357
Epoch 41/50
3780/3780 [==============================] - 1s 228us/step
 - loss: 0.0502 - accuracy: 0.9828 - val_loss: 0.2838 - val
_accuracy: 0.9381
Epoch 42/50
3780/3780 [==============================] - 1s 226us/step
 - loss: 0.0495 - accuracy: 0.9857 - val_loss: 0.3326 - val
_accuracy: 0.9238
Epoch 43/50
3780/3780 [==============================] - 1s 236us/step
 - loss: 0.0579 - accuracy: 0.9839 - val_loss: 0.3113 - val
_accuracy: 0.9190
Epoch 44/50
3780/3780 [==============================] - 1s 231us/step
 - loss: 0.0453 - accuracy: 0.9849 - val_loss: 0.3005 - val
_accuracy: 0.9238
Epoch 45/50
3780/3780 [==============================] - 1s 231us/step
 - loss: 0.0700 - accuracy: 0.9775 - val_loss: 0.3391 - val
_accuracy: 0.9286
Epoch 46/50
3780/3780 [==============================] - 1s 230us/step
 - loss: 0.0370 - accuracy: 0.9897 - val_loss: 0.2543 - val
_accuracy: 0.9429
Epoch 47/50
3780/3780 [==============================] - 1s 232us/step
 - loss: 0.0496 - accuracy: 0.9852 - val_loss: 0.3305 - val
_accuracy: 0.9357
Epoch 48/50
3780/3780 [==============================] - 1s 232us/step
 - loss: 0.0559 - accuracy: 0.9844 - val_loss: 0.2865 - val
_accuracy: 0.9357
Epoch 49/50
```

```
3780/3780 [==============================] - 1s 232us/step
 - loss: 0.0294 - accuracy: 0.9926 - val_loss: 0.3063 - val
_accuracy: 0.9357
Epoch 50/50
3780/3780 [==============================] - 1s 236us/step
 - loss: 0.0535 - accuracy: 0.9836 - val_loss: 0.3695 - val
_accuracy: 0.9167
# 生成学习曲线 和损失函数 随着 epoch 的变化曲线
# 模型的学习效果怎么样？ 能找到适合的 epoch 吗？
# 简单的评价标准应该用什么？
# 尝试改变模型参数 生成不同的学习曲线 比较
# 提示 从 epoch> 优化器> 损失函数> 学习率> dropout 有无 依次调试

fig, ax = plt.subplots(2,1)

ax[0].plot(history.history['loss'], color='b', label="Training loss")
ax[0].plot(history.history['val_loss'], color='r', label="validation loss",axes =ax
[0])
legend = ax[0].legend(loc='best', shadow=True)

ax[1].plot(history.history['accuracy'], color='b', label="Training accuracy")
ax[1].plot(history.history['val_accuracy'], color='r',label="Validation accuracy")
legend = ax[1].legend(loc='best', shadow=True)
```



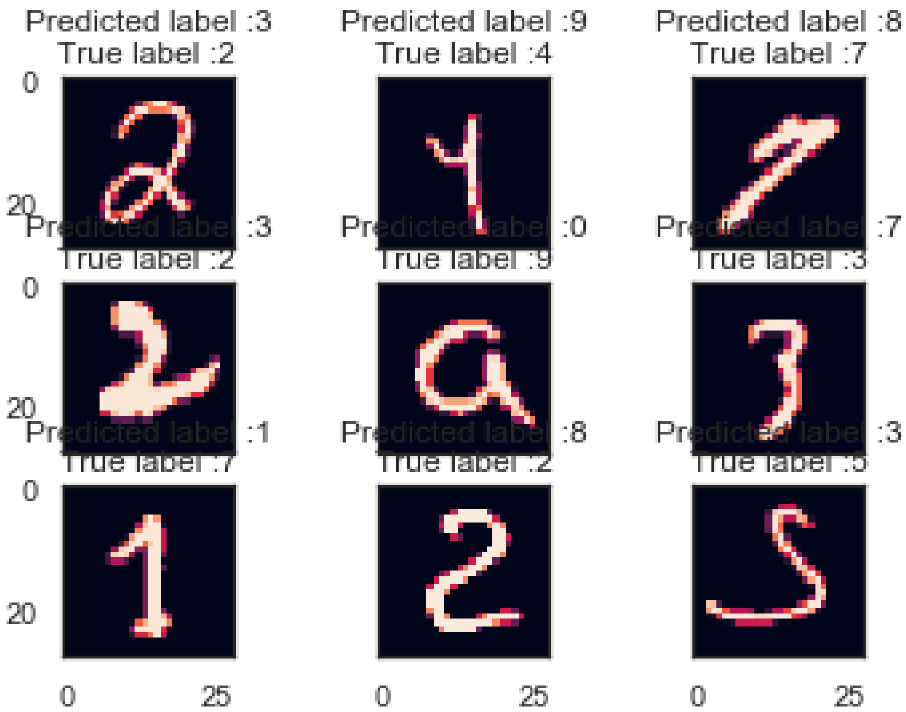

```
# 生成 10 标签混淆矩阵

def plot_confusion_matrix(cm, classes,
                          normalize=False,
```

```
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, cm[i, j],
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

# Predict the values from the validation dataset
Y_pred = model.predict(X_val)
# Convert predictions classes to one hot vectors
Y_pred_classes = np.argmax(Y_pred,axis = 1)
# Convert validation observations to one hot vectors
Y_true = np.argmax(Y_val,axis = 1)
# compute the confusion matrix
confusion_mtx = confusion_matrix(Y_true, Y_pred_classes)
# plot the confusion matrix
plot_confusion_matrix(confusion_mtx, classes = range(10))
```



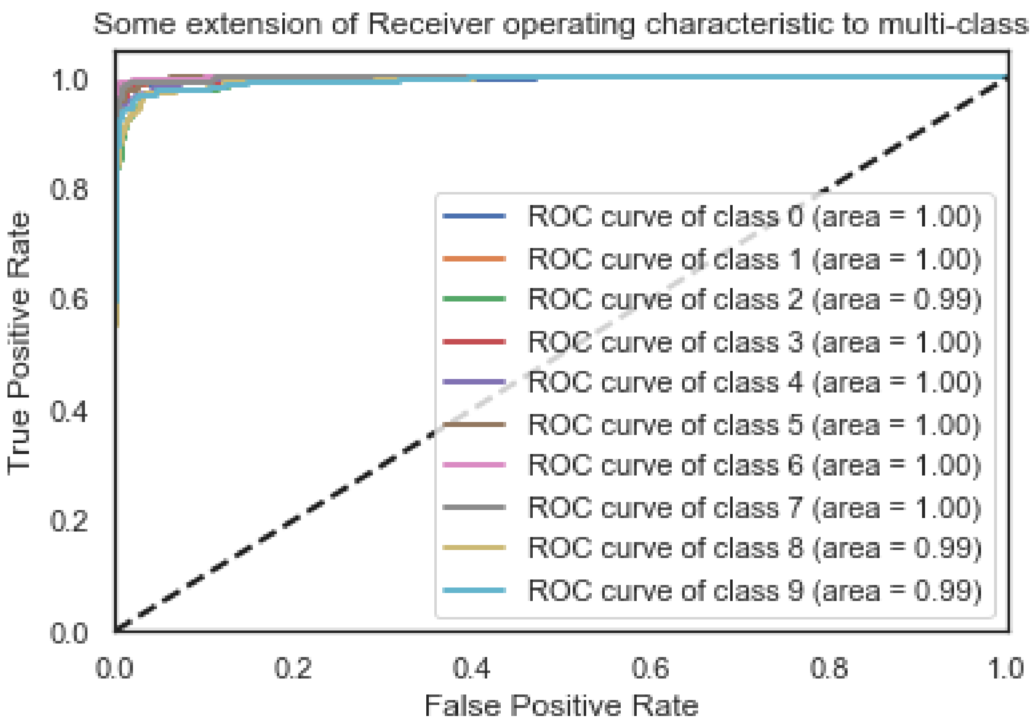

```
### 打印出认错的数字

errors = (Y_pred_classes - Y_true != 0)

Y_pred_classes_errors = Y_pred_classes[errors]
Y_pred_errors = Y_pred[errors]
Y_true_errors = Y_true[errors]
X_val_errors = X_val[errors]

def display_errors(errors_index,img_errors,pred_errors, obs_errors):
    """ This function shows 6 images with their predicted and real labels
"""
    n = 0
    nrows = 3
    ncols = 3
    fig, ax = plt.subplots(nrows,ncols,sharex=True,sharey=True)
    for row in range(nrows):
        for col in range(ncols):
            error = errors_index[n]
            ax[row,col].imshow((img_errors[error]).reshape((28,28)))
            ax[row,col].set_title("Predicted label :{}\nTrue label :{}".format
(pred_errors[error],obs_errors[error]))
            n += 1

# Probabilities of the wrong predicted numbers
```

```
Y_pred_errors_prob = np.max(Y_pred_errors,axis = 1)

# Predicted probabilities of the true values in the error set
true_prob_errors = np.diagonal(np.take(Y_pred_errors, Y_true_errors, axis=1))

# Difference between the probability of the predicted label and the true lab
el
delta_pred_true_errors = Y_pred_errors_prob - true_prob_errors

# Sorted list of the delta prob errors
sorted_dela_errors = np.argsort(delta_pred_true_errors)

# Top 9 errors
most_important_errors = sorted_dela_errors[-9:]

# Show the top 9 errors
display_errors(most_important_errors, X_val_errors, Y_pred_classes_errors, Y_t
rue_errors)
```



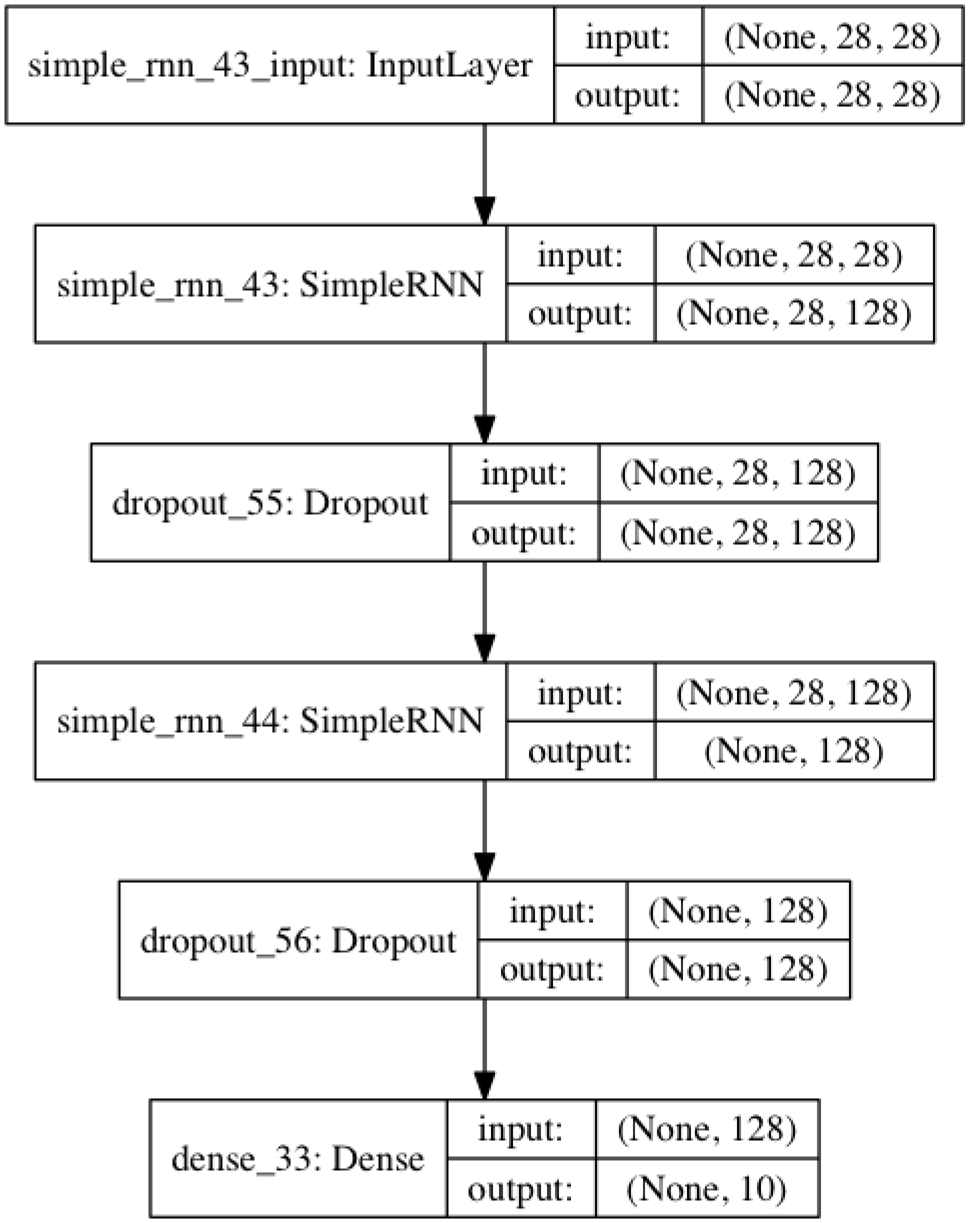

```
#optional 画出 roc
from sklearn.metrics import roc_curve, auc
fpr = dict()
tpr = dict()
roc_auc = dict()
y_score = model.predict(X_test)
# 在前天的作业中 y_test Pandas 下的 DataFrame 类型：y_test
# 让数据为 Pandas DataFrame 类型的话 调用/使用他 第 i 行第 j 列的数据:
```

```
#   y_test.iloc[i,j]

# 在今天的作业中，y_test 是 numpy 的 numarry 数据类型
# 让数据为numarray 类型的话 调用/使用他 第i 行第j 列的数据:
# y_test[i,j]
for i in range(num_classes):
     fpr[i], tpr[i], _ = roc_curve(Y_test[:,i], y_score[:,i]) #
    # AUC Area Under the Curve
     roc_auc[i] = auc(fpr[i], tpr[i])
#y_pred_keras = model.predict(X_test).ravel()
##fpr_keras, tpr_keras, thresholds_keras = roc_curve(Y_test, y_pred_keras)
#y_pred_keras

for i in range(num_classes):
    plt.plot(fpr[i], tpr[i], lw=2, label='ROC curve of class {0} (area = {1:0.2
f})'
             ''.format(i, roc_auc[i]))

plt.plot([0, 1], [0, 1], 'k--', lw=2)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Some extension of Receiver operating characteristic to multi-class')
plt.legend(loc="lower right")
plt.show()
```




```
from keras.utils import plot_model
plot_model(model, to_file='model.png', show_shapes=True)
```

Out[398]:



y_score.shape

Out[128]:

```
(210, 10)
```