实验二 mnist 手写体识别

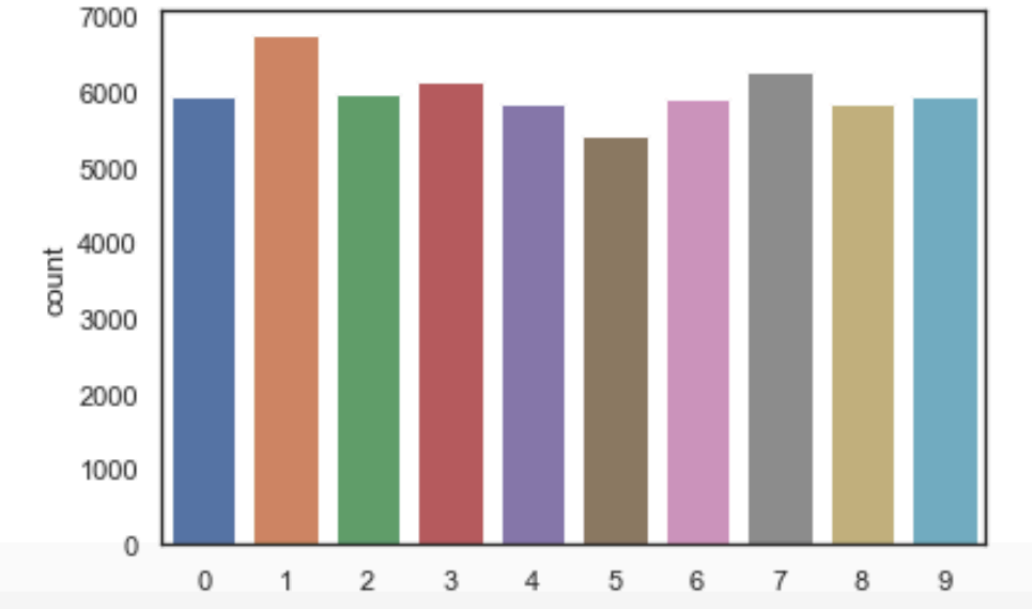
数据加载和数据预处理

数据加载

直接从包内导入数据集

```
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

数据的类别分布情况如下图所示，可以看到十种类别每种的数目较平均



数据处理和分析

空值判断

```
In [7]: np.isnan(x_train).any()
Out[7]: False

In [9]: np.isnan(x_test).any()
Out[9]: False
```

可以看到不包含空值，所以不需要对于空值的预处理

标准化

标准化避免尺度差异造成的影响。CNN模型在标准化后一般运行效果更好。这里对于 MNIST 数据集，我们希望每个值都在 0.0 和 1.0 之间。 由于所有值最初都在 0.0-255.0 范围内，因此除以 255.0。

```
x_train = x_train / 255.0
x_test = x_test / 255.0
```

cnn输入预处理

x_train 数据集中的值是 28x28 图像，需要指定输入形状，将图像转化为三个维度$28*28*1$。

```
x_train = x_train.reshape(-1,28,28,1)
x_test = x_test.reshape(-1,28,28,1)
```

one-hot 编码

将对应的数字标签转化成one-hot向量，比如将3 转为 [0,0,0,1,0,0,0,0,0,0]

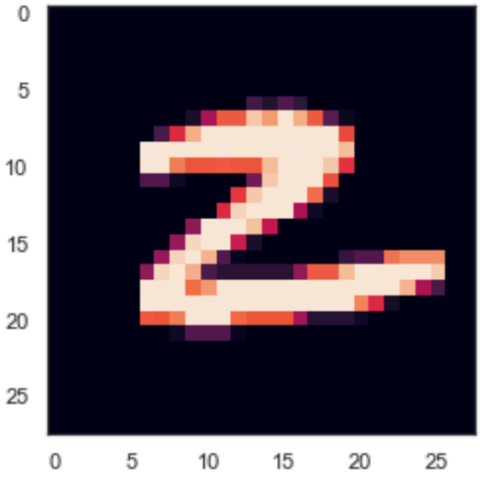
```
In [ ]: y_train = to_categorical(y_train, num_classes = 10)
```

```
In [6]: x_train.shape, x_test.shape, y_train.shape, y_test.shape
Out[6]: ((60000, 28, 28, 1), (10000, 28, 28, 1), (60000, 10), (10000,))
```

```
In [7]: y_test1=to_categorical(y_test, num_classes = 10)
```

```
In [8]: y_test1.shape
Out[8]: (10000, 10)
```

查看训练集中的数字图像



CNN模型训练过程

模型结构

使用keras sequential模型。直接通过model.add添加对应的层。

CNN结构如下：

[卷积层->批标准化->卷积层->批标准化->池化层->dropout]*2->

卷积层->批标准化->dropout->扁平化->全连接层->dropout->全连接层->输出

卷积层 Convolution2D,卷积核个数选为64个，前三层卷积层窗口大小为(5,5),后两次窗口大小设为(3,3),激活函数选择relu。（relu函数的优点包括：可以使网络训练更快；增加网络的非线性；防止梯度消失。）

批标准化 BatchNormalization,该层可以防止梯度爆炸或弥散、可以提高训练时模型对于不同超参（学习率、初始化）的鲁棒性、可以让大部分的激活函数能够远离其饱和区域。

池化层 MaxPooling2D,池化窗口的大小(2,2),步长(2,2)

全连接层 dense,第一个全连接层：输出维度为256，激活函数为relu；第二个全连接层：输出维度为10，激活函数为softmax.softmax作用：将神经网络的输出变为概率分布。数据之和为1；负数变为正数。

Dropout dropout rate 一般不宜过高，这里设为0.25,经过参数比较这样可以较好地加大模型鲁棒性,并且不会出现overfitting

```
model = Sequential()
model.add(Conv2D(filters = 64, kernel_size = (5,5),activation ='relu',
input_shape = (28,28,1)))
model.add(BatchNormalization())
model.add(Conv2D(filters = 64, kernel_size = (5,5),activation ='relu'))
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(2,2)))
model.add(Dropout(0.25))
```

优化器比较

RMSprop

```
optimizer = RMSprop(lr=0.001, rho=0.9, epsilon=1e-08, decay=0.0)
```

最后的优化效果验证准确率可达99.65%

Adam

```
optimizer1=Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-08, decay=0.0,
amsgrad=False)
```

最后的优化效果验证准确率可达99.63%

RMSprop, Adadelta, Adam 在很多情况下的效果是相似的。

Adam 就是在 RMSprop 的基础上加了 bias-correction 和 momentum。

模型优化

数据增强

进行数据增强，从而防止过拟合的情况发生。

进行的操作包括将一些训练图像随机旋转 10 度；将一些训练图像随机放大 10%；将图像水平随机移动 10% 的宽度；将图像垂直随机移动 10% 的高度；不进行上下翻转或者左右翻转，避免6和9分类错误。

利用实现keras.preprocessing.image中的ImageDataGenerator

```
ImageDataGenerator(featurewise_center=False,samplewise_center=False,
 featurewise_std_normalization=False,
        samplewise_std_normalization=False,  zca_whitening=False,
        rotation_range=10,  zoom_range = 0.1, width_shift_range=0.1,
 height_shift_range=0.1,  horizontal_flip=False,  vertical_flip=False)
```

学习率优化

使用回调函数ReduceLROnPlateau在训练中优化学习率。定义学习率之后，经过一定epoch迭代之后，模型效果不再提升，该学习率可能已经不再适应该模型。需要在训练过程中缩小学习率，进而提升模型。

监控val_acc,缩放学习率的因子设为0.5，学习率最小值设为0.00001。当3个epoch过去而模型性能不提升时，学习率减少的动作会被触发。

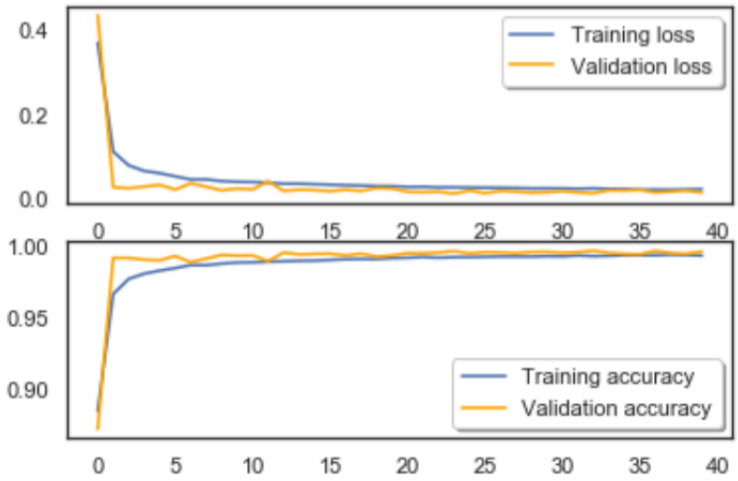
```
ReduceLROnPlateau(monitor='val_acc', patience=3, verbose=1,
factor=0.5,min_lr=0.00001)
```

模型评价

训练和验证准确率

Adam训练和验证准确率

RMSprop训练和验证准确率



可以看到两个优化器到后期训练和验证准确率都稳定在0.99以上，RMSprop前十轮的准确率没有adam高,后期两种优化器准确率差不多

RMSprop前十轮

```
Epoch 1/40
 - 173s - loss: 0.3680 - accuracy: 0.8847 - val_loss: 0.4336 - val_accuracy: 0.8721
Epoch 2/40
 - 175s - loss: 0.1120 - accuracy: 0.9664 - val_loss: 0.0279 - val_accuracy: 0.9916
Epoch 3/40
 - 171s - loss: 0.0795 - accuracy: 0.9769 - val_loss: 0.0255 - val_accuracy: 0.9916
Epoch 4/40
 - 176s - loss: 0.0662 - accuracy: 0.9806 - val_loss: 0.0295 - val_accuracy: 0.9904
Epoch 5/40
 - 179s - loss: 0.0610 - accuracy: 0.9827 - val_loss: 0.0334 - val_accuracy: 0.9900
Epoch 6/40
 - 175s - loss: 0.0537 - accuracy: 0.9845 - val_loss: 0.0224 - val_accuracy: 0.9929
Epoch 7/40
 - 175s - loss: 0.0463 - accuracy: 0.9866 - val_loss: 0.0370 - val_accuracy: 0.9886
Epoch 8/40
 - 173s - loss: 0.0464 - accuracy: 0.9865 - val_loss: 0.0293 - val_accuracy: 0.9911
Epoch 9/40
 - 171s - loss: 0.0424 - accuracy: 0.9876 - val_loss: 0.0207 - val_accuracy: 0.9936
Epoch 10/40
 - 169s - loss: 0.0407 - accuracy: 0.9884 - val_loss: 0.0247 - val_accuracy: 0.9932
```

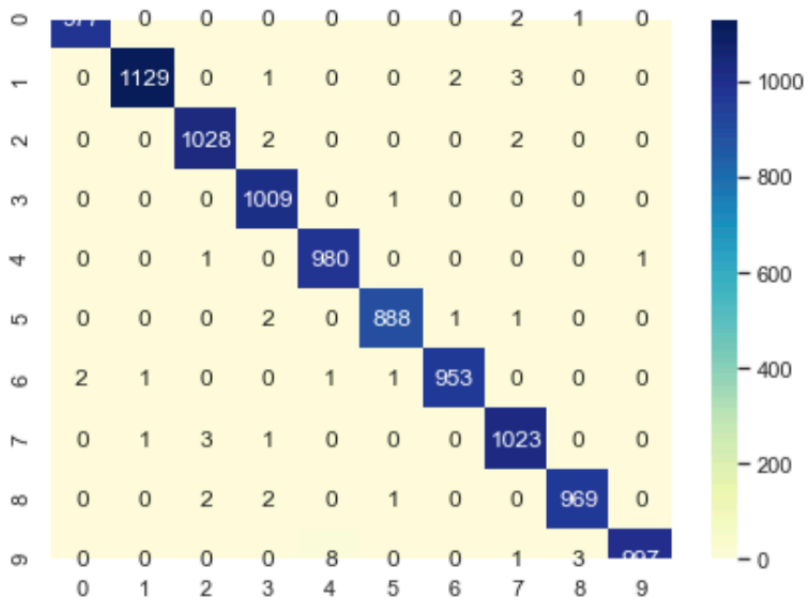
Adam前十轮

```
Epoch 1/40
 - 213s - loss: 0.1126 - accuracy: 0.9658 - val_loss: 0.0865 - val_accuracy: 0.9756
Epoch 2/40
 - 251s - loss: 0.0786 - accuracy: 0.9767 - val_loss: 0.0299 - val_accuracy: 0.9911
Epoch 3/40
 - 181s - loss: 0.0681 - accuracy: 0.9805 - val_loss: 0.0253 - val_accuracy: 0.9925
Epoch 4/40
 - 175s - loss: 0.0577 - accuracy: 0.9822 - val_loss: 0.0220 - val_accuracy: 0.9933
Epoch 5/40
 - 176s - loss: 0.0497 - accuracy: 0.9856 - val_loss: 0.0229 - val_accuracy: 0.9931
Epoch 6/40
 - 172s - loss: 0.0478 - accuracy: 0.9862 - val_loss: 0.0241 - val_accuracy: 0.9920
Epoch 7/40
 - 172s - loss: 0.0438 - accuracy: 0.9869 - val_loss: 0.0209 - val_accuracy: 0.9941
Epoch 8/40
 - 171s - loss: 0.0424 - accuracy: 0.9871 - val_loss: 0.0196 - val_accuracy: 0.9947
Epoch 9/40
 - 168s - loss: 0.0410 - accuracy: 0.9878 - val_loss: 0.0258 - val_accuracy: 0.9916
Epoch 10/40
 - 165s - loss: 0.0392 - accuracy: 0.9881 - val_loss: 0.0159 - val_accuracy: 0.9943
```

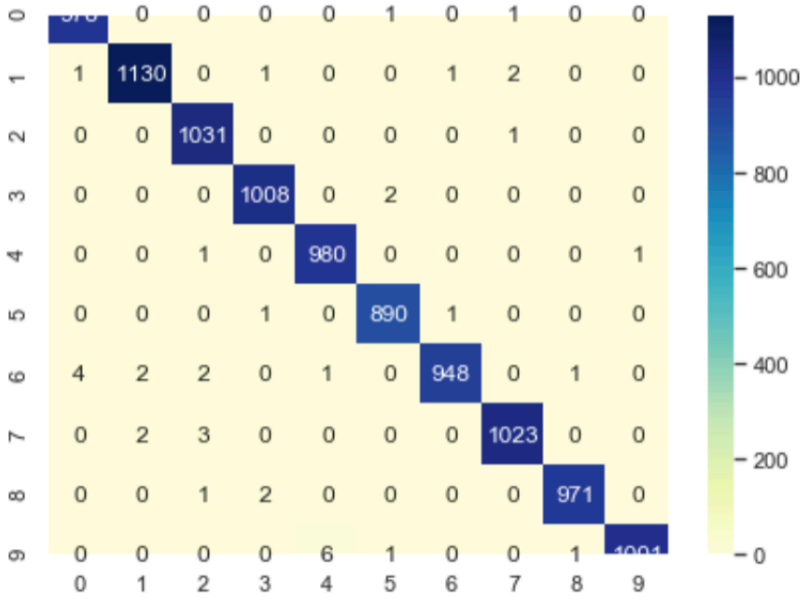
模糊矩阵

```
Y_pred = model1.predict(x_test)
Y_pred_classes = np.argmax(Y_pred,axis = 1)
Y_true = np.argmax(y_test1,axis = 1)
confusion_mtx1 = tf.math.confusion_matrix(Y_true, Y_pred_classes)
plt.figure(figsize=(7, 5))
sns.heatmap(confusion_mtx1, annot=True,cmap='YlGnBu',fmt="d")
```

如图所示，adam的混淆矩阵，可以看到绝大部分手写数字都被正确识别



RMSprop的混淆矩阵



(4,9) 在两种优化器下都较容易被混淆。在RMSprop优化器下(0,6) 之间的混淆比adam优化器下要稍高一些。