華東理工大學

模式识别大作业

题　　目　基于 MINST 数据集的手写数字识别

学　　院　信息科学与工程

专　　业　控制科学与工程

组　　员　孙琦钰

指导教师　赵海涛

完成日期：　2019 年 12 月 10 日

模式识别作业报告——基于 MINST 数据集的手写数字识别

组员：孙琦钰

通过半学期的模式识别课程，尤其是考试复习，我掌握了一些经典模式识别方法的数学推导。这对我来说是一个挑战，同时受益匪浅。赵老师的耐心讲解、逐步推导、经验分享以及讲解时重难点的强调，让我在全局上对于模式识别这门课程有了初步的理解。由于我的研究方向与机器视觉相关并且受我的技术水平的局限，我选择了 Kaggle 中的 MINST 数据集手写数字识别的题目作为本次的大作业。

一、MINST 数据集简介

MNIST（Modified National Institute of Standards and Technology）数据集是模式识别领域中非常经典的一个数据集。最早于 1998 年由 Yan Lecun 在论文“Gradient-based learning applied to document recognition”中提出。MNIST 数据集来自美国国家标准与技术研究所（National Institute of Standards and Technology，NIST）。训练集和测试集均由 250 位不同人手写的数字构成，其中 50%是高中学生，50%来自人口普查局的工作人员。MNIST 共包含 70000 张手写数字图片，由 60000 个训练样本和 10000 个测试样本组成。原始数据集可在 MNIST 官网（http://yann.lecun.com/exdb/mnist/ ）上下载。

MNIST 数据集中包含了阿拉伯数字 0-9 共 10 类手写数字图片。每张图片均做了尺寸归一化，由 28×28 个像素点构成，每个像素点用一个灰度值表示。像素值大小介于 0-255 之间，且数字都会出现在图片的正中间。数据集中的图片如下图所示：

图 1 MNIST 数据集图片示例

MNIST 数据集相当于机器视觉领域的“Hello World！”。自 1999 年发布以来，这个经典的手写图像数据集一直被当做分类算法的基准数据集。随着新机器学习算法的不断出现，MNIST 仍然备受研究人员的青睐。MNIST 官方的数据集包含

四个部分：Training set images（包含 60000 个样本），Training set labels（包含 60000 个样本标签），Test set images（包含 10000 个样本），Test set labels（包含 10000 个样本标签），即分别为训练集图片及其对应的真值标签，测试集图片及其对应的真值标签。如下表所示：

表 1 MNIST 数据集内容

	文件名称	大小	内容
训练集	train-images-idx3-ubyte.gz	9681KB	55000 张训练集，5000 张验证集
训练集标签真值	train-labels-idx1-ubyte.gz	29KB	训练集图片对应的标签
测试集	t10k-images-idx3-ubyte.gz	1611KB	10000 张测试集
测试集标签真值	t10k-labels-idx1-ubyte.gz	5KB	测试集图片对应的标签

二、解题思路

2.1 问题分析

基于 MINST 数据集的手写数字识别任务的目标是通过某种分类方法，使得系统具有正确识别手写数字的能力。即通过利用含有真实标签的训练集，运用某种方法，使得系统具有正确分类不含标签的测试集的能力。有两个解决问题的思路：一是搭建简单的神经网络，二是使用 SVM 或者 K 近邻算法等分类算法。本报告采用第一个思路，即通过构建简单的神经网络实现分类功能。

与 MNIST 官网上给出的数据集不同，Kaggle 给出的训练集包含 42000 个样本。对应的，测试集中包含 28000 个样本。样本以 csv 的形式给出，每行以像素值的形式记录了图片的信息。对于训练集，每行包含 749 个值，即包含图片分类结果和 748（28×28）个像素值；对于测试集，每行包括 748 个值，即与训练集相比，测试集中不包含图片的分类结果。

考虑到本文数据集由图片组成，因此使用卷积神经网络提取特征，最后使用 softmax 函数进行十分类。使用训练集训练出神经网络模型，然后用测试集进行测试，得到预测结果。本报告内使用基于 Keras 框架的卷积神经网络解决这个十分类问题，使用 GPU 加速训练。

2.2 数据预处理

首先，为了验证数据导入是否正确，我可视化了训练集中前 20 个训练样本。图 2 画出了这 20 个样本图形。

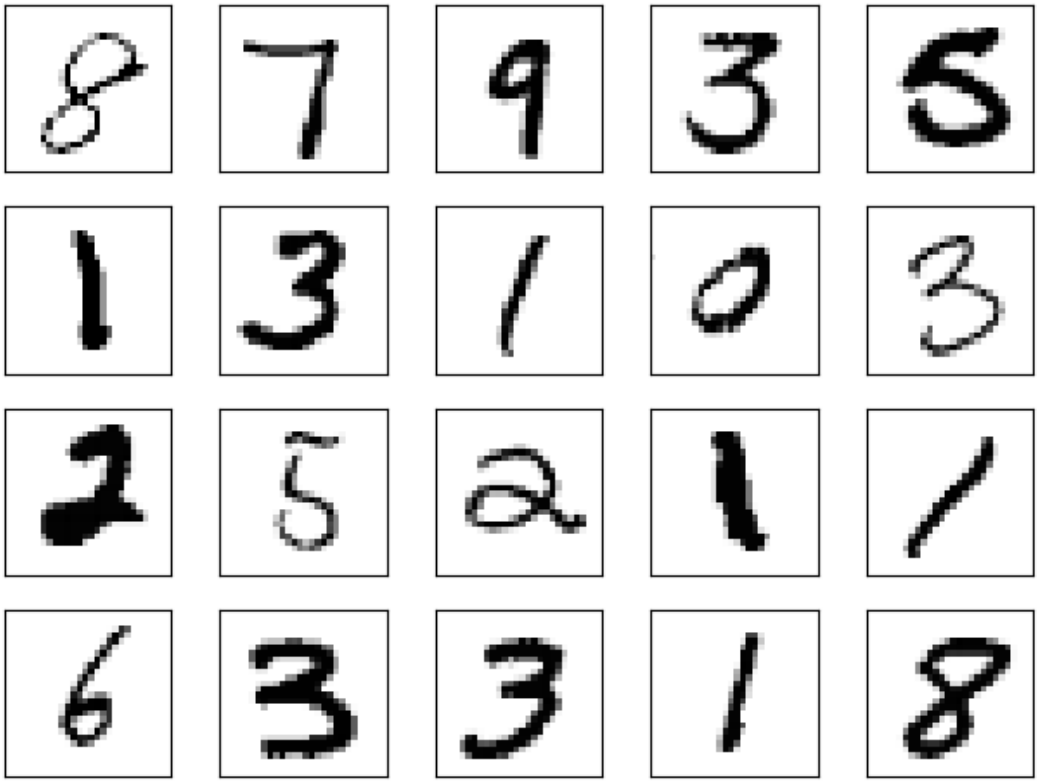

图 2 训练样本可视化

本部分对数据进行了预处理，为网络提供了训练数据，详细代码代码见/mnist/data_pepare.py。

- 数据导入

```
train = pd.read_csv("../input/train.csv")
test = pd.read_csv("../input/test.csv")
```

- 提取手写图片及真值标签

```
Y_train = train["label"]
X_train = train.drop(labels = ["label"],axis = 1)
X_test = test
```

其中，Y_train 为训练集的真实值标签，X_train 为训练集数据，X_test 为测试集数据。

- 数据归一化

```
X_train = X_train / 255.0
X_test = X_test / 255.0
Y_train = to_categorical(Y_train, num_classes = 10)
```

此处的数据归一化的预处理，参考 TensorFlow 对 MNIST 数据集处理的思路。TensorFlow 采用封装的函数读取 MNIST，读进来的图像像素值为

0-1 之间，标签是由热编码组成的大小为 1*10 的行向量。这样的归一化使得特征的提取不受像素值的影响。

- 重排训练集

```
X_train = X_train.values.reshape(-1,28,28,1)
X_test = X_test.values.reshape(-1,28,28,1)
```

每张训练图片以 1×784 的形式存储为 csv 格式，由于本方法中使用卷积神经网络作为手写数字的分类器，因此将其重排为 28×28 的图片方便输入卷积神经网络。

- 划分训练集和验证集

```
random_seed = 2
X_train, X_val, Y_train, Y_val = train_test_split(X_train, Y_train, test_size = 0.1, random_state=random_seed)
```

由于测试集没有提供真值，因此需要将原本的训练集划分为训练集和验证集，方便测试方法分类的能力。

2.3 基于 Keras 框架的神经网络搭建

- 神经网络搭建，构建代码如下：

```
model = Sequential()

model.add(Conv2D(filters = 32, kernel_size = (5,5),padding = 'Same',
activation ='relu', input_shape = (28,28,1)))
model.add(Conv2D(filters = 32, kernel_size = (5,5),padding = 'Same',
activation ='relu'))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(Dropout(0.25))

model.add(Conv2D(filters = 64, kernel_size = (3,3),padding = 'Same',
activation ='relu'))
model.add(Conv2D(filters = 64, kernel_size = (3,3),padding = 'Same',
activation ='relu'))
model.add(MaxPool2D(pool_size=(2,2), strides=(2,2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(256, activation = "relu"))
```

```
model.add(Dropout(0.5))
model.add(Dense(10, activation = "softmax"))
```

本文的神经网络基于Keras框架，通过卷积和池化操作提取输入样本的特征，使用 Dropout 避免过拟合的现象。详细的网络结果如表 2 所示。

表 2 神经网络结构

层数	操作	卷积核大小	卷积核个数	激活函数
0	输入图像			
1	卷积	5×5	32	Relu
2	卷积	5×5	32	Relu
3	池化	2×2		
Dropout=0.25				
4	卷积	3×3	64	Relu
5	卷积	3×3	64	Relu
6	池化	2×2		
Dropout=0.25				
7	全连接 256			Relu
Dropout=0.25				
8	全连接 10			Softmax

2.4 网络训练参数设置

- 硬件系统：
  本模型在一块GTX 2080 Ti 的GPU上训练，运行一个epoch用时大概7-8秒。
- 优化器

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

- 设置学习率

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

- 构建训练模型

```
model.compile(optimizer = optimizer , loss = "categorical_crossentropy",
metrics=["accuracy"])
```

  其中损失使用交叉熵函数，即通过计算预测值和真实值之间的交叉熵作为网络训练的损失函数。

- 训练次数

```
epochs = 50
```

```
batch_size = 86
```

2.5 数据增强

由于给出的 MNIST 训练集较小，因此本作业中还使用了数据增强的方法用来防止过拟合。我们知道，随着神经网络的加深，需要学习的参数也会随之增加，这样就会更容易导致过拟合，当数据集较小的时候，过多的参数会拟合数据集的所有特点，而非数据之间的共性。

过拟合的情况即为神经网络可以高度拟合训练数据的分布情况，但是对于测试数据来说准确率很低，缺乏泛化能力。因此在这种情况下，为了防止过拟合现象，数据增强应运而生。当然除了数据增强，还有正则项/dropout 等方式可以防止过拟合。

常见的数据增强方法有：

1）随机旋转：是对输入图像随机旋转 0-360 度；

2）随机裁剪：对输入图像随机切割掉一部分；

3）色彩抖动：在颜色空间如 RGB 中，每个通道随机抖动一定的程度，但不常用。

4）高斯噪声：在图像中随机加入少量的噪声。该方法对防止过拟合比较有效，这会让神经网络不能拟合输入图像的所有特征。

5）水平翻转

6）竖直翻转

本实验中数据增强的代码为：

```
datagen = 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)
datagen.fit(X_train)

history = model.fit_generator(datagen.flow(X_train,Y_train,
```

```
batch_size=batch_size), epochs = epochs, validation_data = (X_val,Y_val),
verbose = 2, steps_per_epoch=X_train.shape[0] // batch_size,
callbacks=[learning_rate_reduction])
```

2.6 程序调试

通过使用 GPU 训练神经网络，使得训练速度大大提升。为了确保神经网络收敛，在初次训练时，共训练了 50 个 epoch，共花费 6min，平均每个 epoch 耗时 7-8 秒钟。训练过程中训练集和验证集上的训练损失和精度曲线如图 3 所示，具体训练损失和精度的数值如图 4 所示。随着迭代网络逐渐收敛，在验证集上的精度达到 99%以上。且由图 3 可知，训练到 15 个 epoch 已经收敛，精确度和损失都达到较为满意的值，为了避免过拟合，因此最终确定迭代次数为 15 个 epoch。最终上交的预测值使用训练 15 个 epoch 的模型。

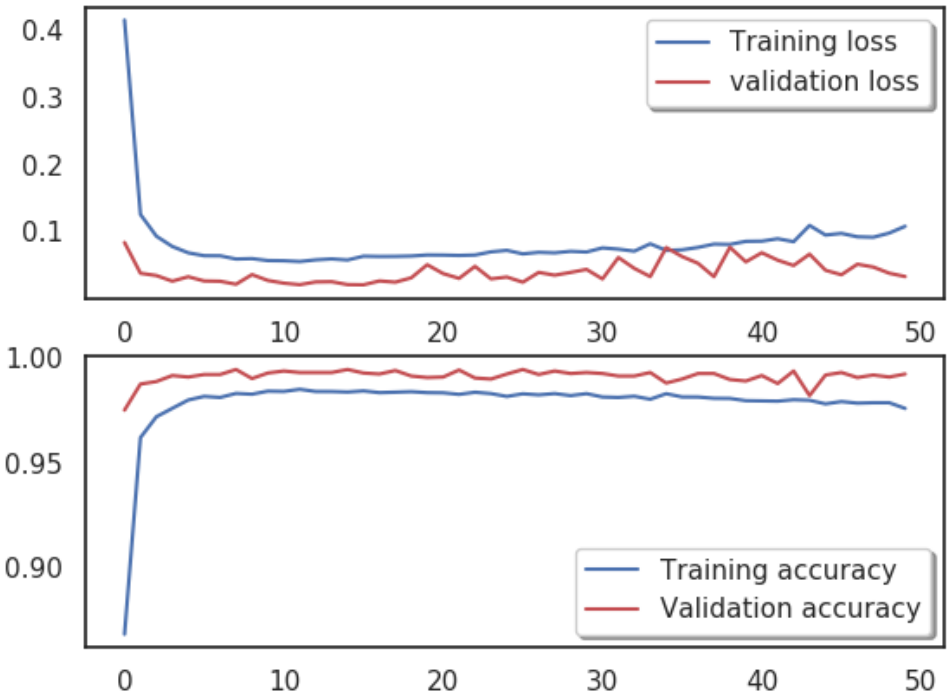

图 3 训练 50 个 epoch 的训练损失和精度曲线

```
Epoch 1/50: - 10s - loss: 0.4160 - accuracy: 0.8685 - val_loss: 0.0840 - val_accuracy: 0.9750
Epoch 2/50: - 8s - loss: 0.1258 - accuracy: 0.9619 - val_loss: 0.0381 - val_accuracy: 0.9874
Epoch 3/50: - 8s - loss: 0.0935 - accuracy: 0.9718 - val_loss: 0.0347 - val_accuracy: 0.9886
Epoch 4/50: - 8s - loss: 0.0781 - accuracy: 0.9758 - val_loss: 0.0263 - val_accuracy: 0.9914
Epoch 5/50: - 7s - loss: 0.0686 - accuracy: 0.9798 - val_loss: 0.0329 - val_accuracy: 0.9907
Epoch 6/50: - 8s - loss: 0.0643 - accuracy: 0.9815 - val_loss: 0.0266 - val_accuracy: 0.9919
Epoch 7/50: - 7s - loss: 0.0642 - accuracy: 0.9810 - val_loss: 0.0262 - val_accuracy: 0.9919
Epoch 8/50: - 7s - loss: 0.0593 - accuracy: 0.9829 - val_loss: 0.0217 - val_accuracy: 0.9943
Epoch 9/50: - 7s - loss: 0.0599 - accuracy: 0.9826 - val_loss: 0.0361 - val_accuracy: 0.9900
Epoch 10/50: - 7s - loss: 0.0572 - accuracy: 0.9840 - val_loss: 0.0274 - val_accuracy: 0.9926
Epoch 11/50: - 7s - loss: 0.0567 - accuracy: 0.9839 - val_loss: 0.0234 - val_accuracy: 0.9936
Epoch 12/50: - 7s - loss: 0.0556 - accuracy: 0.9849 - val_loss: 0.0211 - val_accuracy: 0.9929
Epoch 13/50: - 8s - loss: 0.0581 - accuracy: 0.9839 - val_loss: 0.0252 - val_accuracy: 0.9929
Epoch 14/50: - 7s - loss: 0.0595 - accuracy: 0.9838 - val_loss: 0.0255 - val_accuracy: 0.9929
Epoch 15/50: - 8s - loss: 0.0579 - accuracy: 0.9836 - val_loss: 0.0212 - val_accuracy: 0.9943
Epoch 16/50: - 7s - loss: 0.0639 - accuracy: 0.9841 - val_loss: 0.0209 - val_accuracy: 0.9926
Epoch 17/50: - 7s - loss: 0.0632 - accuracy: 0.9833 - val_loss: 0.0266 - val_accuracy: 0.9921
Epoch 18/50: - 7s - loss: 0.0633 - accuracy: 0.9835 - val_loss: 0.0248 - val_accuracy: 0.9938
Epoch 19/50: - 7s - loss: 0.0638 - accuracy: 0.9837 - val_loss: 0.0313 - val_accuracy: 0.9912
Epoch 20/50: - 7s - loss: 0.0654 - accuracy: 0.9833 - val_loss: 0.0509 - val_accuracy: 0.9905
Epoch 21/50: - 7s - loss: 0.0653 - accuracy: 0.9832 - val_loss: 0.0376 - val_accuracy: 0.9907
Epoch 22/50: - 7s - loss: 0.0647 - accuracy: 0.9825 - val_loss: 0.0305 - val_accuracy: 0.9940
Epoch 23/50: - 8s - loss: 0.0655 - accuracy: 0.9835 - val_loss: 0.0484 - val_accuracy: 0.9902
Epoch 24/50: - 7s - loss: 0.0701 - accuracy: 0.9828 - val_loss: 0.0298 - val_accuracy: 0.9898
Epoch 25/50: - 8s - loss: 0.0723 - accuracy: 0.9815 - val_loss: 0.0325 - val_accuracy: 0.9921
Epoch 26/50: - 8s - loss: 0.0670 - accuracy: 0.9827 - val_loss: 0.0248 - val_accuracy: 0.9943
Epoch 27/50: - 8s - loss: 0.0693 - accuracy: 0.9822 - val_loss: 0.0394 - val_accuracy: 0.9919
Epoch 28/50: - 7s - loss: 0.0686 - accuracy: 0.9828 - val_loss: 0.0352 - val_accuracy: 0.9936
Epoch 29/50: - 8s - loss: 0.0709 - accuracy: 0.9819 - val_loss: 0.0394 - val_accuracy: 0.9924
Epoch 30/50: - 8s - loss: 0.0698 - accuracy: 0.9828 - val_loss: 0.0439 - val_accuracy: 0.9929
Epoch 31/50: - 8s - loss: 0.0759 - accuracy: 0.9812 - val_loss: 0.0296 - val_accuracy: 0.9924
Epoch 32/50: - 7s - loss: 0.0740 - accuracy: 0.9810 - val_loss: 0.0616 - val_accuracy: 0.9912
Epoch 33/50: - 8s - loss: 0.0711 - accuracy: 0.9816 - val_loss: 0.0454 - val_accuracy: 0.9912
Epoch 34/50: - 8s - loss: 0.0820 - accuracy: 0.9802 - val_loss: 0.0331 - val_accuracy: 0.9929
Epoch 35/50: - 8s - loss: 0.0724 - accuracy: 0.9828 - val_loss: 0.0765 - val_accuracy: 0.9879
Epoch 36/50: - 8s - loss: 0.0731 - accuracy: 0.9813 - val_loss: 0.0631 - val_accuracy: 0.9898
Epoch 37/50: - 7s - loss: 0.0767 - accuracy: 0.9812 - val_loss: 0.0529 - val_accuracy: 0.9924
Epoch 38/50: - 8s - loss: 0.0815 - accuracy: 0.9806 - val_loss: 0.0330 - val_accuracy: 0.9924
Epoch 39/50: - 8s - loss: 0.0812 - accuracy: 0.9805 - val_loss: 0.0769 - val_accuracy: 0.9895
Epoch 40/50: - 8s - loss: 0.0857 - accuracy: 0.9795 - val_loss: 0.0551 - val_accuracy: 0.9888
Epoch 41/50: - 8s - loss: 0.0861 - accuracy: 0.9794 - val_loss: 0.0687 - val_accuracy: 0.9914
Epoch 42/50: - 8s - loss: 0.0898 - accuracy: 0.9792 - val_loss: 0.0576 - val_accuracy: 0.9876
Epoch 43/50: - 7s - loss: 0.0852 - accuracy: 0.9799 - val_loss: 0.0497 - val_accuracy: 0.9936
Epoch 44/50: - 8s - loss: 0.1095 - accuracy: 0.9797 - val_loss: 0.0666 - val_accuracy: 0.9819
Epoch 45/50: - 8s - loss: 0.0950 - accuracy: 0.9780 - val_loss: 0.0426 - val_accuracy: 0.9917
Epoch 46/50: - 8s - loss: 0.0975 - accuracy: 0.9791 - val_loss: 0.0359 - val_accuracy: 0.9929
Epoch 47/50: - 8s - loss: 0.0925 - accuracy: 0.9783 - val_loss: 0.0516 - val_accuracy: 0.9905
Epoch 48/50: - 8s - loss: 0.0918 - accuracy: 0.9785 - val_loss: 0.0474 - val_accuracy: 0.9917
Epoch 49/50: - 7s - loss: 0.0981 - accuracy: 0.9785 - val_loss: 0.0382 - val_accuracy: 0.9907
Epoch 50/50: - 8s - loss: 0.1082 - accuracy: 0.9758 - val_loss: 0.0332 - val_accuracy: 0.9921
```

图 4 训练损失和精度

绘制迭代损失的代码如下：

```
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)
plt.show()
```

2.7 模型评估

图 5 为训练 15 个 epoch 的训练损失和精度曲线，可知，此时训练收敛且达到较高的预测精度。

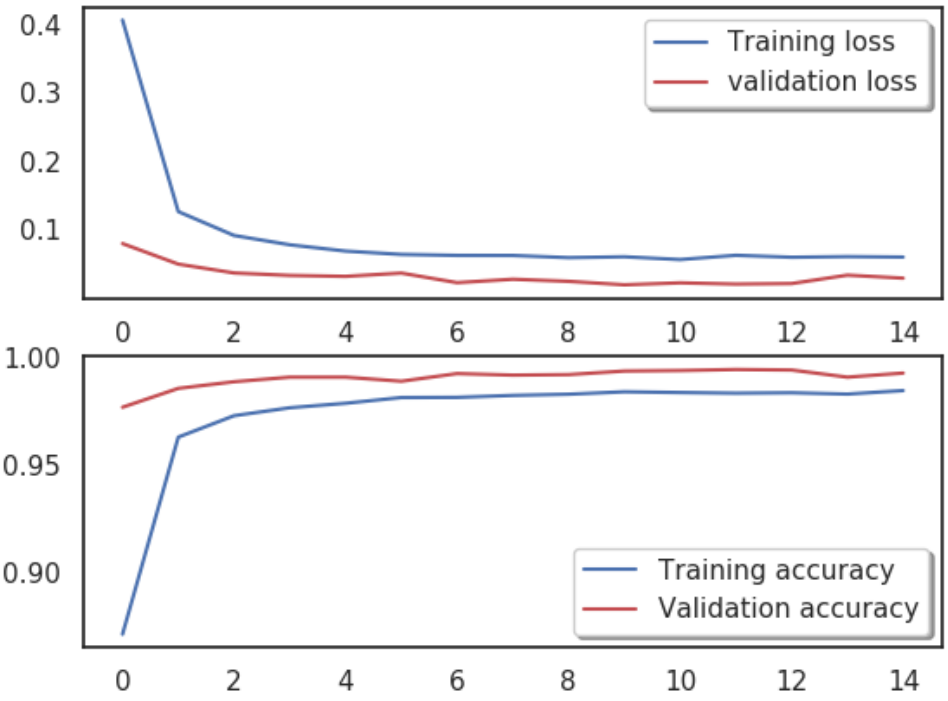

图 5 训练 15 个 epoch 的训练损失和精度曲线

混淆矩阵是机器学习中总结分类模型预测结果的情形分析表，以矩阵形式将数据集中的记录按照真实的类别与分类模型预测的类别判断两个标准进行汇总。其中矩阵的行表示真实值，矩阵的列表示预测值。此处的混淆矩阵也是使用迭代 15 个 epoch 得到的结果，混淆矩阵如图 6 所示，

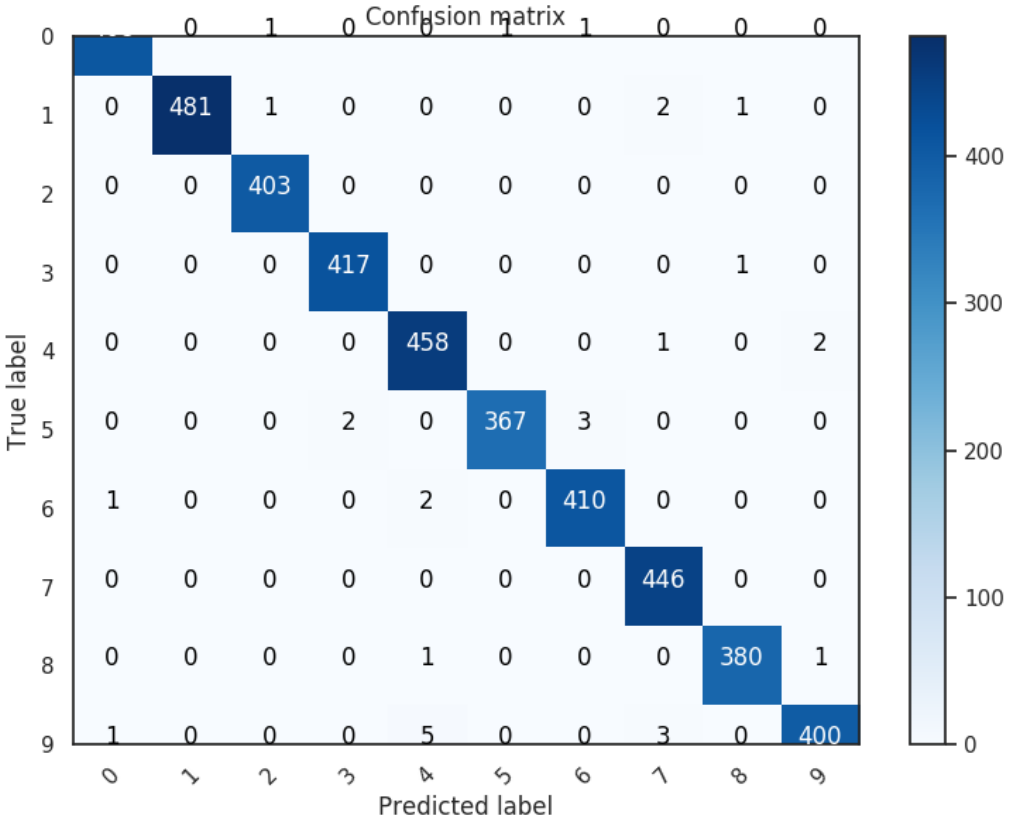

图 6 混淆矩阵

由图 6 可知，该模型在验证集上获得了较好的分类效果。

混淆矩阵绘制的代码如下：

```
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    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')
        plt.show()
    Y_pred = model.predict(X_val)
    Y_pred_classes = np.argmax(Y_pred,axis = 1)
    Y_true = np.argmax(Y_val,axis = 1)
    confusion_mtx = confusion_matrix(Y_true, Y_pred_classes)
    plot_confusion_matrix(confusion_mtx, classes = range(10))
```

为了分析网络预测错误的原因，图 7 画出了一些预测错的图片以便于分析。可以发现，这些预测错误的图片，即使是人眼看，也不一定能给出正确的预测。这是由于手写数字的独特性造成的，由于每个人的书写习惯不一样，造成不同数字的书写方式不一样，有时候甚至写出来的数字让人难以区分。此外，由于本实验采用数据增强的方法，使得一些图片经过增强的操作后，有一些变形，使得难以辨识出正确对的结果。总的来说，本方法在基于 MNIST 数据集的手写数字识别中取得了较好的效果。

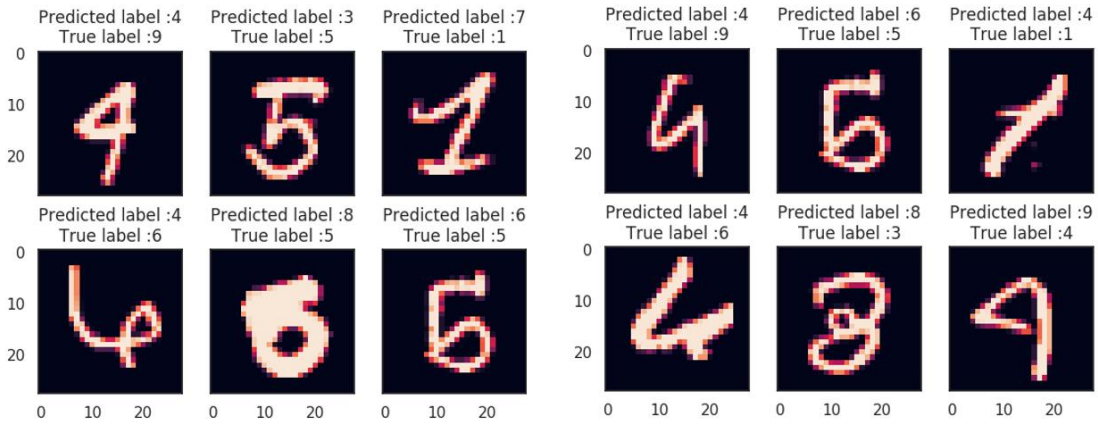

图 7 预测错误样本可视化

可视化预测错误图片的代码如下所示：

```
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):
    n = 0
    nrows = 2
    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
    plt.show()

Y_pred_errors_prob = np.max(Y_pred_errors,axis = 1)
true_prob_errors = np.diagonal(np.take(Y_pred_errors, Y_true_errors, axis=1))
delta_pred_true_errors = Y_pred_errors_prob - true_prob_errors
sorted_dela_errors = np.argsort(delta_pred_true_errors)
most_important_errors = sorted_dela_errors[-6:]
display_errors(most_important_errors, X_val_errors, Y_pred_classes_errors,
Y_true_errors)
```

2.8 预测值生成和提交

在训练好神经网络模型后，可以利用网络分类的能力，预测测试集上图片的类型。生成文件名为 mnist_test 的 csv 文件，代码如下：

```
results = model.predict(X_test)
results = np.argmax(results,axis = 1)
results = pd.Series(results,name="Label")
submission = pd.concat([pd.Series(range(1,28001),name = "ImageId"),results],axis =
1)
submission.to_csv("mnist_test.csv",index=False)
```

在 Kaggle 上提交，结果如图 8 所示。预测精度为 0.99385。

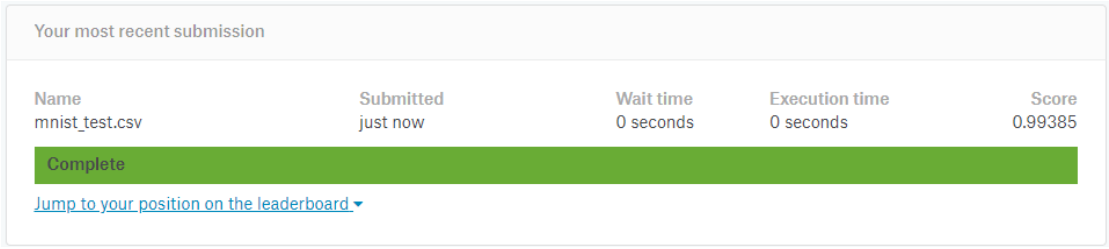

图 8 提交结果

三、小组分工

程序设计及编写：孙琦钰
程序调试：孙琦钰
实验报告：孙琦钰

四、作业总结与展望

通过本次大作业，我学习了基本神经网络的搭建方法及其在机器视觉领域的应用，熟悉了 MNIST 数据集。通过一些可视化方法，可以更好的分析网络预测失败的原因。虽然我能力有限且所选题目简单，但是从选题到程序测试，整个过程让我学到很多，提升了实践能力。最后感谢赵老师在本学期对我的帮助，最要感谢的是老师在数学公式推导上对我对的帮助，让我能够有机会在数学机理上理解和应用模式识别。

在接下来的学习中，我将充分总结这次实践的经验，为之后的研究方向打好基础，继续努力，希望获得更大的进步。

附：文件说明

本次作业的内容打包在 Digit Recognizer 上传。
其中：

- figure 文件夹中包含本次作业中的一些图片；
- input 文件夹中为初始的训练和测试的 csv 文件；
- mnist 文件夹中为一些程序。包含 data_prepare.py 和 mnist.py，data_prepare.py 为数据预处理，data_prepare.py 为神经网络搭建及一些画图的程序；
- loss.txt 为训练 50 个 epoch 的损失和精度；
- mnist_test.csv 为最终上传的预测结果。