**报告**

# 1 简介

手写数字识别是深度学习的经典应用之一，特别是对Mnist数据集的训练识别，是深度学习训练常选的基础数据集。为此，我们基于经典的AlexNet编写卷积神经网络，对Mnist数据集进行训练，生成手写数字识别模型，并基于Flask框架提供识别服务。通过设计简单的网页端界面，允许用户选择图片并上传到此识别服务，然后识别并返回给前端用户。此外，本工程还基于Cassandra提供上传记录的保存功能，记录上传时间、文件名和预测结果等信息，有助于进一步的模型检查和数据分析。

为此，我们基于Docker容器进行封装，可方便的进行编译打包，实现识别服务的快速部署和启动。

# 2 训练

首先，下载Mnist数据集对模型进行训练。通过设置download参数为True，可以自动指定目录并下载Mnist数据集。

root\_path = './data'

train\_set = mnist.MNIST(root\_path, train=True, transform=train\_transform, download=True)  
test\_set = mnist.MNIST(root\_path, train=False, transform=test\_transform, download=True)

然后，选择经典的的AlexNet模型（注：为了获得更高的精度，可以选择不同类型的复杂模型来完成这一步骤）。

# 定义Net  
class Net(nn.Module):  
 '''  
 自定义AlexNet的神经网络  
 '''  
 def \_\_init\_\_(self, num\_classes=10):  
 super(Net, self).\_\_init\_\_()  
 self.features = nn.Sequential(  
 nn.Conv2d(1, 64, kernel\_size=5, stride=1, padding=2),  
 nn.ReLU(inplace=True),  
 nn.MaxPool2d(kernel\_size=3, stride=1),  
 nn.Conv2d(64, 192, kernel\_size=3, padding=2),  
 nn.ReLU(inplace=True),  
 nn.MaxPool2d(kernel\_size=3, stride=2),  
 nn.Conv2d(192, 384, kernel\_size=3, padding=1),  
 nn.ReLU(inplace=True),  
 nn.Conv2d(384, 256, kernel\_size=3, padding=1),  
 nn.ReLU(inplace=True),  
 nn.Conv2d(256, 256, kernel\_size=3, padding=1),  
 nn.ReLU(inplace=True),  
 nn.MaxPool2d(kernel\_size=3, stride=2),  
 )  
 self.classifier = nn.Sequential(  
 nn.Dropout(),  
 nn.Linear(256 \* 6 \* 6, 4096),  
 nn.ReLU(inplace=True),  
 nn.Dropout(),  
 nn.Linear(4096, 4096),  
 nn.ReLU(inplace=True),  
 nn.Linear(4096, num\_classes),  
 )  
  
 def forward(self, x):  
 x = self.features(x)  
 x = x.view(x.size(0), 256 \* 6 \* 6)  
 x = self.classifier(x)  
 return x

最后，利用数据加载器DataLoader处理数据，调用模型执行训练，并保存模型参数到文件。

def train(self):  
 print("train:")  
 self.model.train()  
 train\_loss = 0  
 train\_correct = 0  
 total = 0  
  
 for batch\_num, (data, target) in enumerate(self.train\_loader):  
 batch\_size\_start = time.time()  
  
 data, target = data.to(self.device), target.to(self.device)  
 self.optimizer.zero\_grad()  
 output = self.model(data)  
 loss = self.criterion(output, target)  
 loss.backward()  
 self.optimizer.step()  
 train\_loss += loss.item()  
 prediction = torch.max(output, 1)  
 total += target.size(0)  
  
 train\_correct += np.sum(prediction[1].cpu().numpy() == target.cpu().numpy())  
 print('%s Train, Epoch: %d/%d, BatchNum: %d/%d, Loss: %.4f | Acc: %.3f%%, (TrainCorrect:%d | Total:%d), CostTime :%.4f s'  
 % (time.strftime("%Y-%m-%d %H:%M:%S", time.localtime()),  
 self.epoch, self.epochs,  
 batch\_num, len(self.train\_loader), train\_loss / (batch\_num + 1),  
 100. \* train\_correct / total, train\_correct, total, time.time() - batch\_size\_start))  
  
 return train\_loss, train\_correct / total

# 3 封装

App.py将调用模型并结合Flask提供Http服务，这个文件一共有三部分。第1部分是加载模型并定义函数。第2部分，利用Flask实现上传、下载和调用图片进行识别，返回结果。第3部分，通过容器之间的通信实现数据记录的保存。

## 3.1 Model

选择模型路径，加载模型到内存。

def load\_model(self):  
 model\_out\_path = "model.pth"  
 self.device = torch.device('cpu')  
 self.model = torch.load(model\_out\_path, map\_location='cpu')

设置模型调用函数，接收图片进行识别。

def run\_single(self, img):  
 img = self.load\_data(img)  
 output = self.model(img)  
 prediction = torch.max(output, 1)  
 class\_ind = np.int32(prediction[1].cpu().numpy())  
 res = class\_info[class\_ind[0]]  
 return res

## 3.2 Flask

为了达到用户上传图片和手写数字识别的目的，编写一个简单的图像上传、下载、识别并返回的函数。为此，为了保证文件接收的安全性，这里设置上载文件夹地址和允许的文件扩展名。此外，文件大小和其他一些要求也可以根据不同的要求设置。

ALLOWED\_EXTENSIONS = set(['png', 'bmp', 'jpg', 'jpeg', 'gif'])  
  
app = Flask(\_\_name\_\_)  
app.config['UPLOAD\_FOLDER'] = os.getcwd() + '/upload/'  
if os.path.exists(app.config['UPLOAD\_FOLDER']) is False:  
 os.makedirs(app.config['UPLOAD\_FOLDER'])  
  
def allowed\_file(filename):  
 return '.' in filename and \  
 filename.rsplit('.', 1)[1] in ALLOWED\_EXTENSIONS

如果上传的图片符合要求，则程序会自动保存并获取保存路径。然后，读取图片并调用已训练过的数字识别模型来识别手写数字图像。同时，该函数还记录了上传图片的时间节点、识别结果、图片名称，并将其保存到了数据库。

@app.route('/', methods=['GET', 'POST'])  
def upload\_file():  
 if request.method == 'POST':  
 file = request.files['file']  
 im = Image.open(file)  
 if file and allowed\_file(file.filename):  
 filename = secure\_filename(file.filename)  
 fi = os.path.join(app.config['UPLOAD\_FOLDER'], filename)  
 im.save(fi)  
 test\_result = solver.run\_single(fi)  
 current = time.strftime("%Y/%m/%d %H:%M:%S")  
 insert\_into\_cassandra(current, filename, test\_result)  
 print("Test result is {}".format(test\_result))  
 return render\_template('base.html', result=test\_result)  
 return render\_template('base.html')

为了便于图片的提交，我们设计一个.html文件，提供图片文件的上传功能。程序接收图片并识别后，会将识别结果返回到网页。为此，创建一个模板文件夹并添加base.html。

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Upload File</title>

<style>

.card {

box-shadow: 0 4px 8px 0 rgba(0,0,0,0.2);

width: 40%;

border-radius: 5px;

}

.container {

padding: 2px 16px;

}

</style>

</head>

<body>

<center>

<div class="card">

<div class="container">

<center>

<h4>数字识别</h4>

<form method=post enctype=multipart/form-data>

<input type=file name=file>

<input type=submit value=上传>

</form>

</center>

</div>

</div>

</center>

{% if result %}

<center>

<div class="card">

<div class="container">

<center><h4>识别结果</h4>

<p class="text-info ">

{{ result }}

</p>

</center>

</div>

</div>

</center>

{% endif %}

</body>

</html>

## 3.3 Cassandra

为了建立数据存储，我们选择Cassandra来存储识别过程中的记录信息。

docker run -d -p 9042:9042 cassandra

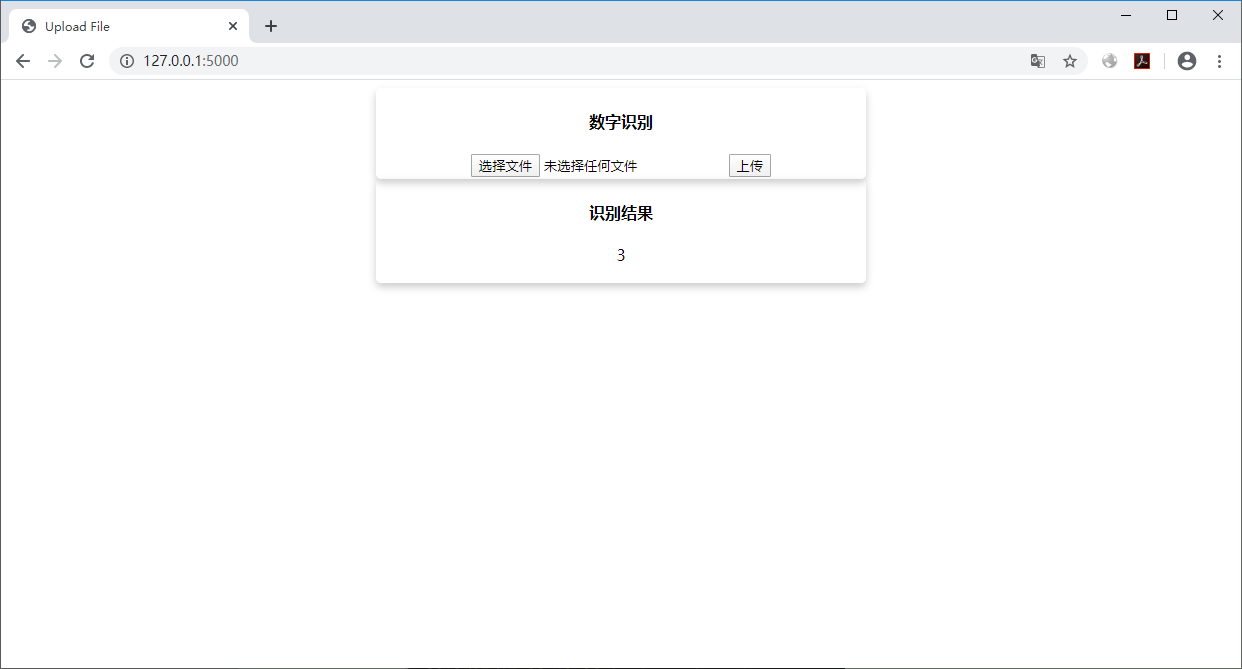
程序设置了创建数据空间、数据表的操作，并将对应的记录信息进行插入。

# create a table  
KEYSPACE = "selfkeyspace"  
  
  
def createKeySpace():  
 cluster = Cluster(contact\_points=[ip], port=9042)  
 session = cluster.connect()  
  
 log.info("Creating keyspace...")  
 try:  
 session.execute("""  
 CREATE KEYSPACE %s  
 WITH replication = { 'class': 'SimpleStrategy', 'replication\_factor': '2' }  
 """ % KEYSPACE)  
  
 log.info("setting keyspace...")  
 session.set\_keyspace(KEYSPACE)  
  
 log.info("creating table...")  
 session.execute("""  
 CREATE TABLE selftable (  
 selfkey text,  
 pic\_name text,  
 pic\_result text,  
 PRIMARY KEY (selfkey,pic\_name)  
 )  
 """)  
 except Exception as e:  
 log.error("Unable to create keyspace")  
 log.error(e)  
  
  
createKeySpace()  
  
# insert data into database  
def insert\_into\_cassandra(var1, var2, var3):  
 sql = "INSERT INTO selfkeyspace.selftable (selfkey, pic\_name, pic\_result) VALUES ('%s', '%s', '%s');""" % (  
 var1, var2, var3)  
 session.execute(sql)

# 4 执行

启动docker后，可以设置固定的端口进行访问。

docker run -p 5000:4000 simple\_app



执行启动cqlsh命令行，查询记录。

docker container ls

docker exec -it cf788e3b9040 cqlsh

cqlsh> select \* from selfkeyspace.selftable;

selfkey | pic\_name | pic\_result

---------------------+----------+------------

2020/03/21 09:35:22 | 7.jpg | 7

2020/03/21 09:14:05 | 02.png | 3

2020/03/21 09:33:51 | 02.png | 3

2020/03/21 09:35:25 | 01.png | 4

(4 rows)

cqlsh>

