远程科研项目报告

姓名：何冠岚

项目：在线手写体文字识别

在线地址：http://mnist.gtbl2012.cn/

仓库地址：https://github.com/gtbl2012/mnist_web_with_cassandra

1 目录

2 项目摘要

Mnist 是现在非常常见的一种 TensorFlow 实例，在本次项目中，我们采用 Softmax 算法和 Deep 算法两种模式对 Mnist 进行实现，使用 TensorFlow 官方提供的标准 Mnist 数据集进行训练、验证与预测，实现基本的手写数字字体识别功能。同时，项目利用常见的 Flask Web 框架，结合前端的 Bootstrap 框架，提供了完善的用户上传识别接口，同时支持用户浏览器可视化上传，以及基于 Curl 的 CLI 上传方式。并且，本项目采用常见的 NoSQL 数据库 Apache Cassandra 进行数据储存，以实现识别历史记录的功能，由于该数据库的特性，在性能和可扩容性上都有一定的优势。

3 项目原理与细节实现

3.1 MNIST 部分

MNIST 是一个基础的计算机视觉数据集，它包含各种手写数字图片：

它也包含每一张图片对应的标签，告诉我们这个是数字几。比如，上面这四张图片的标签分别是 5，0，4，1。

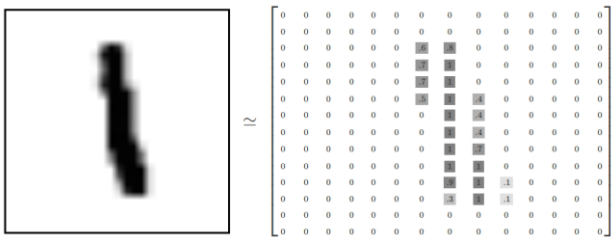
Mnist 主要由两大部分组成，分别是训练部分和预测部分，训练部分分为训练和校验两个模块，预测部分分为图像预处理和预测两个部分。

(1) 数据集与算法简介

1. **图片数据**

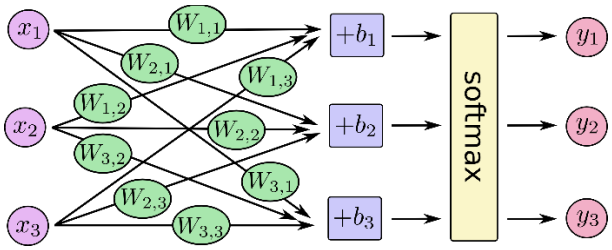
   TensorFlow 提供的 Mnist 数据集有 60000 条训练数据以及 10000 条测试数据，但是为了演示项目的效率起见，我们仅使用其中 1000 条数据进行了预训练并将结果储存在 models 文件夹中（使用 Docker 时也会基于这个预训练的数据）

   数据集中的内容均为已经转化为数字数组的 28*28 大小的图片，示例如下：



2. Softmax 算法

softmax 回归模型可以用下面的图解释，对于输入的 xs 加权求和，再分别加上一个偏置量，最后再输入到 softmax 函数中：

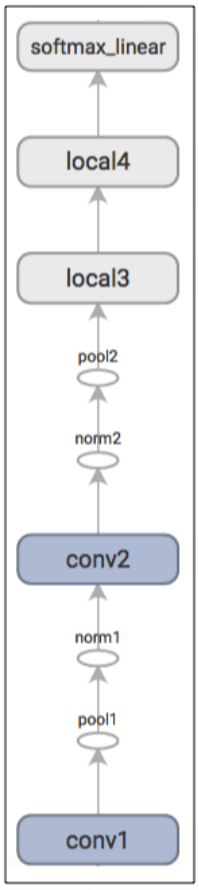


更进一步，可以写成更加紧凑的方式：

$$y = \text{softmax}(Wx + b)$$

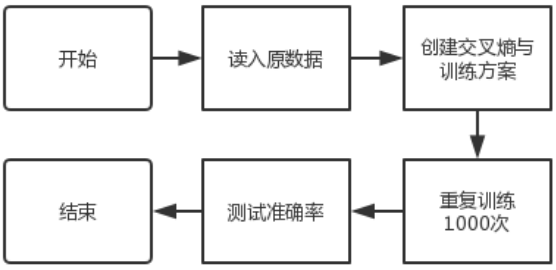
3. DeepNN 算法

DeepNN 由卷积层和非线性层(nonlinearities)交替多次排列后构成。这些层最终通过全连通层对接到 softmax 分类器上。具体图例如下：



(2) 训练部分

此处以 Softmax 算法为例，讲述训练部分的实现，两个算法的大方向流程基本一致，只是使用模型不同。



模型训练的主要流程如上图所示，核心功能代码如下：

① 创建交叉熵与训练方案

```
# Create model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
y = tf.matmul(x, W) + b

y_ = tf.placeholder(tf.float32, [None, 10])

cross_entropy = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y))
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
```

② 训练

```
sess = tf.InteractiveSession()
tf.global_variables_initializer().run()
print("[*] Training mnist softmax")
# Train
for _ in range(1000):
  batch_xs, batch_ys = mnist.train.next_batch(100)
  sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
  if (_ + 1) % 100 == 0:
    print((_ + 1), "Trained")
```

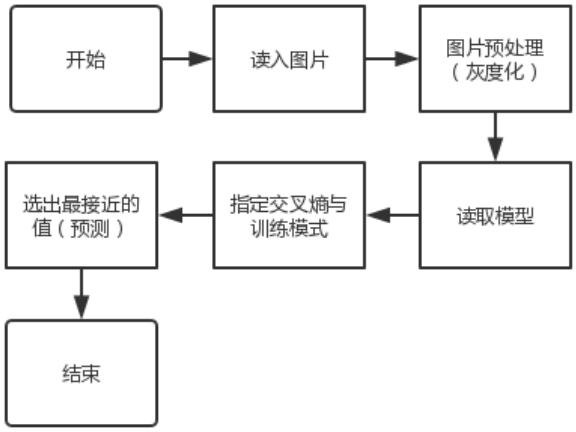
③ 验证准确性

```
# Test trained model
  correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
```

```
  accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
  print("Total Accuracy: ", sess.run(accuracy, feed_dict={x:
mnist.test.images, y_: mnist.test.labels}))
```

(3) 预测部分

此处以 DeepNN 算法为例，讲述预测部分的实现，两个算法的大方向流程基本一致，只是使用模型不同。



模型预测的主要流程如上图所示，核心功能代码如下：

① 读取与预处理图片

```
def imageprepare(filepath):
    im = Image.open(filepath)
    im = im.convert('L')
    tv = list(im.getdata())
    return tv
```

② 定义交叉熵

```
# Create the model
    x = tf.placeholder(tf.float32, [None, 784])

    # Define loss and optimizer
    y_ = tf.placeholder(tf.float32, [None, 10])

    # Build the graph for the deep net
    y_conv, keep_prob = deepnn(x)

    with tf.name_scope('loss'):
        cross_entropy = tf.nn.softmax_cross_entropy_with_logits(labels=y_,
                                                                logits=y_conv)
```

```
    cross_entropy = tf.reduce_mean(cross_entropy)


    with tf.name_scope('accuracy'):
        correct_prediction = tf.equal(tf.argmax(y_conv, 1), tf.argmax(y_, 1))
        correct_prediction = tf.cast(correct_prediction, tf.float32)
```

③ 读取保存的模型数据

```
    saver = tf.train.Saver()

    sess.run(tf.global_variables_initializer())
    saver.restore(sess, "models/deep_model.ckpt") #使用模型，参数和之前的代码保持一致
```

④ 预测

```
prediction=tf.argmax(y_conv,1)
predint=prediction.eval(feed_dict={x: [result], keep_prob: 0.5}, session=sess)
```

3.2 FLASK 部分

(1) 简介

本项目使用了 Flask 构建简单的 HTTP 接口与前端交互式界面。

(2) 接口定义

接口地址	接口类型	请求数据	响应数据
/	GET	无	主页页面 (index.html)

/api/softmax	POST	Image: 图片数据 (binary)	JSON 数据
			{
			“code”: “状态码”,
			“predict_result”: 预测结果,
			“url”: 图片地址
			}

/api/deep	POST	Image: 图片数据 (binary)	JSON 数据
			{
			“code”: “状态码”,
			“predict_result”: 预测结果,
			“url”: 图片地址
			}

/api/history	GET	无	JSON 数据
			{
			“code”: “状态码”,
			“data” : [
			{
			“url”: 图片地址,
			“result”: 预测结果,
			“createtime”: 创建时间
			}
			]
			}

(3) 文件上传与 Mnist 对接

程序中，前端采用用户选择图片后提交 multipart/form-data 的方式向后端提交文件，后端使用 request.files 来进行接收，因此，后端接口不仅支持前端网页图片上传，也支持使用 curl 的方式直接提交图片给后端进行判断。

两种 Mnist 算法预测接口分别为 lib 下的 mnist_softmax.py 与 mnist_deep.py 文件，通过 import 的方式提供给 Flask 主文件，封装为接口 run_predict(filepath)。

Flask 接收到用户上传的文件后，将其储存为[sample_时间戳.原后缀名]的形式，并且将储存后的绝对路径作为 filepath 提供给 mnist 接口，mnist 接口判断后将判断结果返回给 Flask，Flask 将记录储存到数据库后，将文件相对路径、预测结果以 JSON 的格式返回给用户。

(4) 前端实现

前端采用 Boostrap 框架以及 ECMAScript 6 原生 JS 进行实现，提供算法选择，图片上传，结果显示，历史记录显示，历史记录图片下载功能，全部功能采用 ajax 异步请求，以保证良好的用户体验，具体界面效果如下：

(5) 环境变量控制

由于最终项目需要封装进入 Docker，部署者无法直接修改代码，而我们需要部署者提供所使用的 Cassandra 数据库所在的地址，因此我们采用环境变量传参的形式动态设置 Cassandra 数据库的所在地址，Python 端采用 os.environ 读取环境变量，Docker 在 docker run 的时候采用—env 参数进行设置传入。

3.3 CASSANDRA 部分

(1) Cassandra Docker 的拉取与部署

Docker 化的 Cassandra 部署非常简单，只需要执行下面这条命令即可把数据库部署在本地 Docker 的 Container 中，并且开放外部端口以供 Flask 连接与访问。

```
docker run -d --name "cassandra_cluster" -e
CASSANDRA_BROADCAST_ADDRESS="192.168.199.245" -e
CASSANDRA_CLUSTER_NAME="Cassandra-Cluser-v2" -e
CASSANDRA_SEEDS="192.168.199.245" -v /Users/gtbl2012/cassandra-docker-
data:/var/lib/cassandra -p 7000:7000 -p 7001:7001 -p 7199:7199 -p 9042:9042 -
p 9160:9160 cassandra:latest
```

(2) 表结构

本次项目数据库仅用户提供查询历史记录，因此表结构比较简单:

Table: MnistHistory			
Device	Createtime	Result	Url

(3) CQL 特性与编写

由于 Cassandra 允许多端部署，并且是 NoSQL 架构，因此提供的功能与我们所熟悉的 SQL 有所区别。

本次项目中所遇到的区别有：①CQL 本身没有提供 AUTO INCREASEMENT 的字段，因此无法实现 MySQL 中我们常用的自增 ID 列（后面的特性可以发现不应该设置该列），因此，在本次项目中，我们采用创建时间戳（毫秒级）作为主键。②由于 Cassandra 的特性，如果需要对筛选的内容进行排序，必须有 WHERE 语句指向第一主键，同时，被排序的必须为第二及之后的主键。

由于项目中我希望数据库能筛选出最新的 10 条记录进行返回，我加入了一个名为 device 的第一主键，指定储存的设备。在该操作之后，能正常对数据进行排序与筛选。

3.4 DOCKER 部分

(1) Dockerfile 的设计与编写

在实际测试中我发现，如果在 docker 中使用 conda 会导致 docker 的体积过于庞大，不利于镜像的推送和拉取，因此我尝试在 Docker 中仅使用 Python3.6 作为运行环境，发现方案确实可行。

Dockerfile 主要职能为：拉取基本镜像，配置运行环境并安装所需要的库，建立工作目录并拷贝源代码，开放 80 端口，指定 ENRTYPOINT。

具体 Dockerfile 如下：

```
FROM ubuntu:latest

MAINTAINER admin@gtbl2012.cn

# Change source
COPY ./sources.list /etc/apt/sources.list

# Install basic dependencies
RUN apt-get update && apt-get install -y --no-install-recommends \
```

```
        python3-dev \
        python3-pip \
        python3-setuptools

# Set timezone
RUN ln -sf /usr/share/zoneinfo/Asia/Shanghai /etc/localtime

# Set locale
ENV LANG C.UTF-8 LC_ALL=C.UTF-8

# Initialize work environment
RUN pip3 install wheel
RUN pip3 install --upgrade setuptools
RUN pip3 install werkzeug==0.14.1 \
                tensorflow \
                flask \
                cassandra-driver \
                pillow \
                -i https://pypi.tuna.tsinghua.edu.cn/simple

# Initialize workspace
RUN mkdir /workspace
COPY ./ /workspace

# Train basic data set (Already built)
# RUN python3 /workspace/lib/mnist_softmax_train.py
# RUN python3 /workspace/lib/mnist_deep_train.py

# Runtime
ENTRYPOINT ["python3", "/workspace/app.py"]
WORKDIR /workspace
EXPOSE 80
```

(2) Docker 启动指令设计

Docker 启动指令需要允许用户传入自定义环境变量，端口映射。因此设计启动指令如下：

```
docker run --name "mnist_web"  -e CASSANDRA_HOST={your cassandra
host:port} -p 80:80 -d mnist-web:v1.0
```

参数解析：

--name 设置容器名

-e 设置环境变量

-p 设置端口银蛇

-d 后台运行，不连接交互控制台

(3) 推送 Docker 到 Docker Hub

Docker 提供了 docker hub 作为在线镜像储存，将生成好的镜像推送至 Docker Hub 可以为之后的拉取和部署带来极大的方便，将镜像推送到 Docker Hub 只需要如下的两条指令：

```
docker tag a446dcd95dff gtbl2012/mnist-web:v1.0
docker push gtbl2012/mnist-web:v1.0
```

本项目已经推送至 Docker Hub，可以通过如下地址查看与拉取：
https://hub.docker.com/r/gtbl2012/mnist-web

4 项目部署与测试

4.1 部署方案

项目报告中仅提供基于 Docker 的构建与部署方案，更多部署方式请参考项目仓库。

(1) 拉取源代码

```
git clone https://github.com/gtbl2012/mnist_web_with_cassandra.git
```

(2) 构建 Docker 镜像

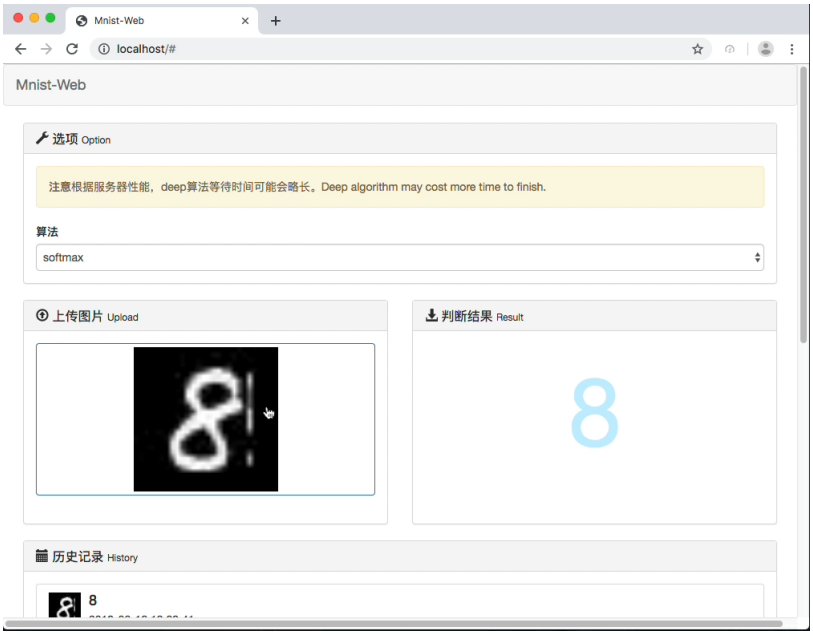
```
cd mnist_web_with_cassandra
docker build -t mnist_web:v1.0 .
```

(3) 运行 Docker 镜像

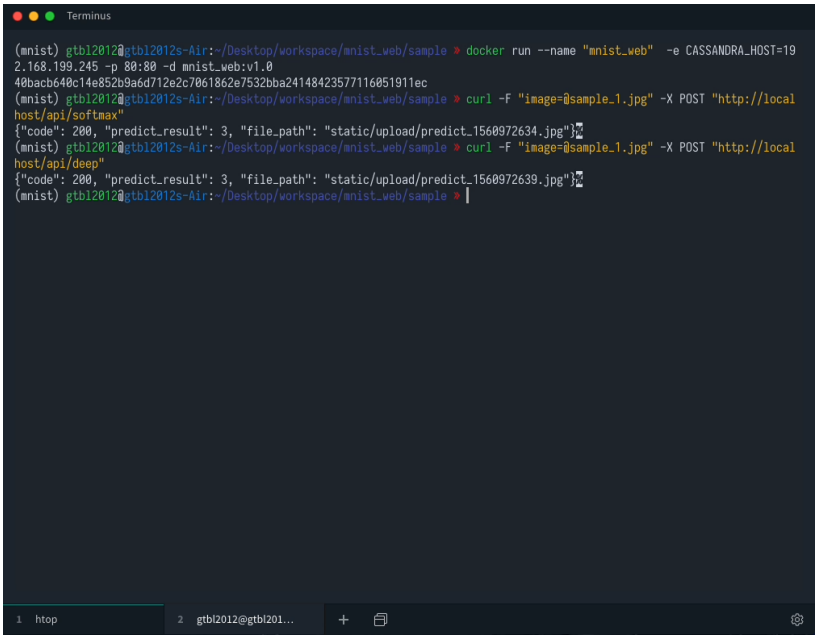
```
docker run --name "mnist_web"  -e CASSANDRA_HOST={your cassandra
host:port} -p 80:80 -d mnist_web:v1.0
```

4.2 运行效果

(1) 交互式界面



(2) 命令行交互



(3) 演示视频

请参见附录文件。

5 项目总结

在这次的项目中，我遇到了一些困难，但同时也给了我非常多的收获。在这次的项目中，我们接触并运用了 TensorFlow、Flask、Cassandra 和 Docker 四个主要内容，其中 Flask 这部分是我非常熟悉的内容，基本没有遇到什么困难，Docker 这部分内容我以前的使用仅停留在拉取和容器管理的层面上，在这次的项目中我体验到了 Dockerfile 的编写，和 Docker Build 过程中的各种坑点，比如说我之前是希望在 Docker 中使用 conda，但是遇到了两个比较严重的问题，第一个是 docker 使用了 /bin/sh 而非 bash，导致 source 指令无法使用，也就无法进入 conda env，同时，安装 Conda 会使得容器的体积变得非常庞大，虽然尝试了 miniconda 的方案，但是最后还是选择了直接使用 Python3.6 作为运行环境，并没有出现兼容性问题。在 TensorFlow 的使用中，由于这次在这方面的内容不算非常深入，并且之前在课程中多多少少都有对这方面有点接触，所以并没有遇到什么难题，就是模型保存和重新读取的时候会有变量不兼容的问题发生。而 Cassandra 的使用对于我来说是一次非常有意义的体验，在这之前，我对 SQL 类数据库比较熟悉，有较长的使用经验，但是对 NoSQL 数据库基本没有什么了解，Cassandra 是一个非常易上手，但又具有典型的 NoSQL 特征和多端可部署的特性，对于我对 NoSQL 类数据库的了解有着莫大的帮助。

同时，在远程科研的过程中，老师还带我们了解了 MapReduce 和 Spark 相关的知识，虽然在项目中没有体现，但是也让我受益匪浅。同时，在这个过程中，我也了解到了一些关于 Openface 人脸识别相关的知识，也尝试使用来做一些实践，虽然最后没有作为提交的项目，但也获得了不少知识，并且给 Openface 开源库提交了一个小问题。

经过这次的项目，虽然最终项目完成得比较顺利，但是我发现我还有许多有待提升的地方，比如说对 TensorFlow 方面的一些理解，还有 Docker 的高级应用和优化方案，都等着我去了解。