

BD30 Report

北京航空航天大学 郑金阳

Contents

1	项目背景	1
1.1	Background	1
1.2	课程概述	1
2	项目回顾	2
2.1	Git与Github	2
2.2	Docker与容器化管理	3
2.3	Flask 与 RESTful API	4
2.4	Cassandra	6
3	项目实施	6
3.1	训练MNIST模型	6
3.2	测试图片准备	7
3.3	TensorFlow的本地预测	8
3.4	Docker部署	9
3.5	Flask与cassandra通信	9
3.6	Volume挂载以及项目整体效果	10
4	项目总结	13

1 项目背景

1.1 Background

大数据(big data)是指无法在一定时间范围内用常规软件工具进行捕捉、管理和处理的数据集合,是需要新处理模式才能具有更强的决策力、洞察发现力和流程优化能力的海量、高增长率和多样化的信息资产。在维克托·迈尔-舍恩伯格及肯尼斯·库克耶编写的《大数据时代》中大数据指不用随机分析法(抽样调查)这样捷径,而采用所有数据进行分析处理。大数据的5V特点(IBM提出): Volume(大量)、Velocity(高速)、Variety(多样)、Value(低价值密度)、Veracity(真实性)。

现在的社会是一个高速发展的社会,科技发达,信息流通,人们之间的交流越来越密切,生活也越来越方便,大数据就是这个高科技时代的产物。阿里巴巴创办人马云来台演讲中就提到,未来的时代将不是IT时代,而是DT的时代,DT就是Data Technology数据科技,显示大数据对于阿里巴巴集团来说举足轻重。有人把数据比喻为蕴藏能量的煤矿。煤炭按照性质有焦煤、无烟煤、肥煤、贫煤等分类,而露天煤矿、深山煤矿的挖掘成本又不一样。与此类似,大数据并不在“大”,而在于“有用”。价值含量、挖掘成本比数量更为重要。对于很多行业而言,如何利用这些大规模数据是赢得竞争的关键。

大数据的价值体现在以下几个方面:

- 对大量消费者提供产品或服务的企业可以利用大数据进行精准营销
- 面临互联网压力之下必须转型的传统企业需要与时俱进充分利用大数据的价值
- 做小而美模式的中小微企业可以利用大数据做服务转型

不过,“大数据”在经济发展中的巨大意义并不代表其能取代一切对于社会问题的理性思考,科学发展的逻辑不能被湮没在海量数据中。著名经济学家路德维希·冯·米塞斯曾提醒过:“就今日言,有很多人忙碌于资料之无益累积,以致对问题之说明与解决,丧失了其对特殊的经济意义的了解。”这确实是需要警惕的。

1.2 课程概述

本课程中,我们首先学习了一些机器学习的基础知识,如Tensorflow,版本控制软件Git,之后又学习了比较通用的容器管理软件Docker。在学习容器的过程中,我们接触了Python网站编写程序Flask和NoSQL型数据库Cassandra。在课程的末尾,我们学习了用于分布式计算的Hadoop,它是市面上使用最多的大数据分布式文件存储系统和分布式处理系统。还有Spark,它是专为大规模数据处理而设计的快速通用的计算引擎。Spark是Hadoop MapReduce的通用并行框架,拥有Hadoop MapReduce所具有的优点;但不同于MapReduce的是——Job中间输出结果可以保存在内存中,从而不再需要读写HDFS,因此Spark能更好地适用于数据挖掘与机器学习等需要迭代的MapReduce的算法。并展望了整个大数据领域研究的知识和所使用的软件,让我感到学习科研的路途遥远。

2 项目回顾

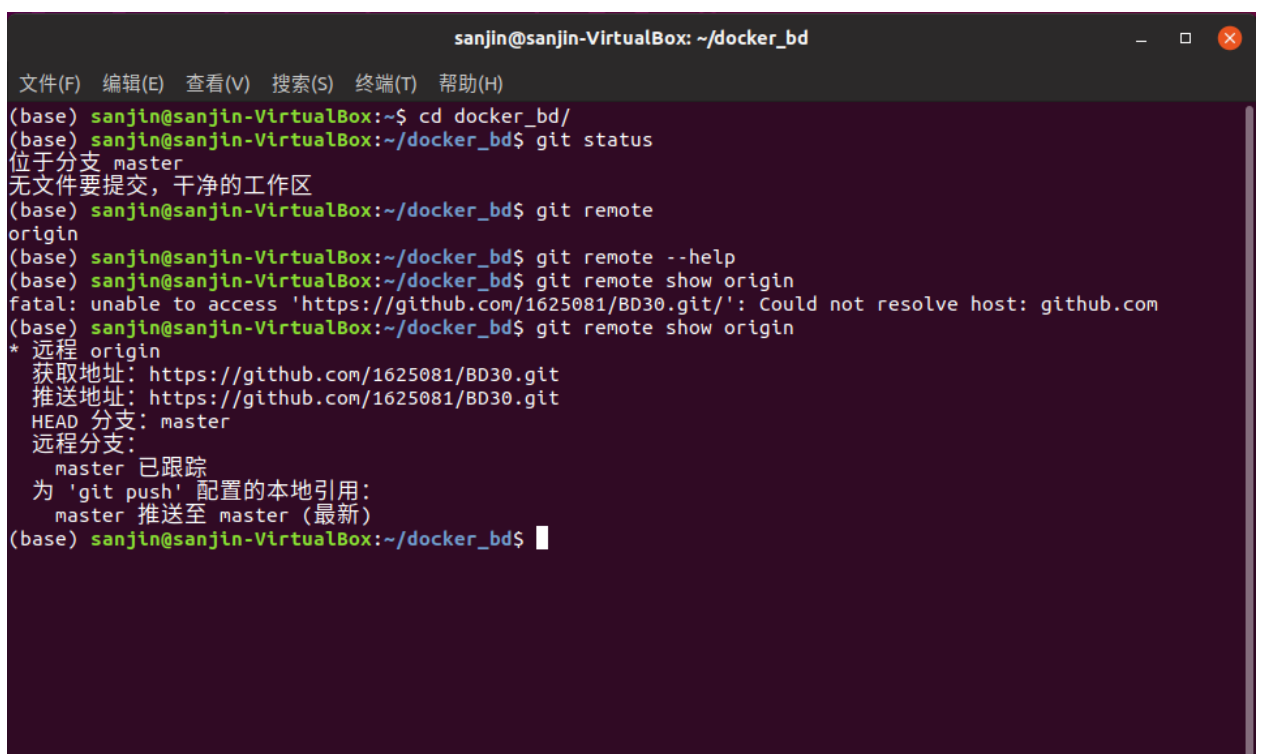
2.1 Git与Github

Git是目前世界上最先进的分布式版本控制系统（没有之一）。它是一个软件，能自动记录每次文件的改动，还可以协作编辑，如果想查看某次改动，只需要在软件里瞄一眼就可以。

版本	文件名	用户	说明	日期
1	service.doc	张三	删除了软件服务条款5	7/12 10:38
2	service.doc	张三	增加了License人数限制	7/12 18:09
3	service.doc	李四	财务部门调整了合同金额	7/13 9:51
4	service.doc	张三	延长了免费升级周期	7/14 15:17

Fig 1. 版本控制效果如下

我是用的环境是VirtualBox虚拟机Ubuntu18.10版本的系统，本项目中我事先在项目文件夹里初始化了Git，并添加了我的远程origin链接。



```
sanjin@sanjin-VirtualBox: ~/docker_bd
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
(base) sanjin@sanjin-VirtualBox:~$ cd docker_bd/
(base) sanjin@sanjin-VirtualBox:~/docker_bd$ git status
位于分支 master
无文件要提交，干净的工作区
(base) sanjin@sanjin-VirtualBox:~/docker_bd$ git remote
origin
(base) sanjin@sanjin-VirtualBox:~/docker_bd$ git remote --help
(base) sanjin@sanjin-VirtualBox:~/docker_bd$ git remote show origin
fatal: unable to access 'https://github.com/1625081/BD30.git/': Could not resolve host: github.com
(base) sanjin@sanjin-VirtualBox:~/docker_bd$ git remote show origin
* 远程 origin
  获取地址: https://github.com/1625081/BD30.git
  推送地址: https://github.com/1625081/BD30.git
  HEAD 分支: master
  远程分支:
    master 已跟踪
  为 'git push' 配置的本地引用:
    master 推送至 master (最新)
(base) sanjin@sanjin-VirtualBox:~/docker_bd$
```

Fig 2. 本项目中的Git

2.2 Docker与容器化管理

在本项目中，我第一次接触到了Docker 它是一个开源的应用容器引擎，让开发者可以打包他们的应用以及依赖包到一个可移植的镜像中，然后发布到任何流行的 Linux或Windows 机器上，也可以实现虚拟化。容器是完全使用沙箱机制，相互之间不会有任何接口。

一个完整的Docker有以下几个部分组成，它是有别于VMware 等传统虚拟机的轻量级虚拟化方案。

- Docker Client 客户端
- Docker Image 镜像
- Docker Container 容器
- Docker Daemon 守护进程

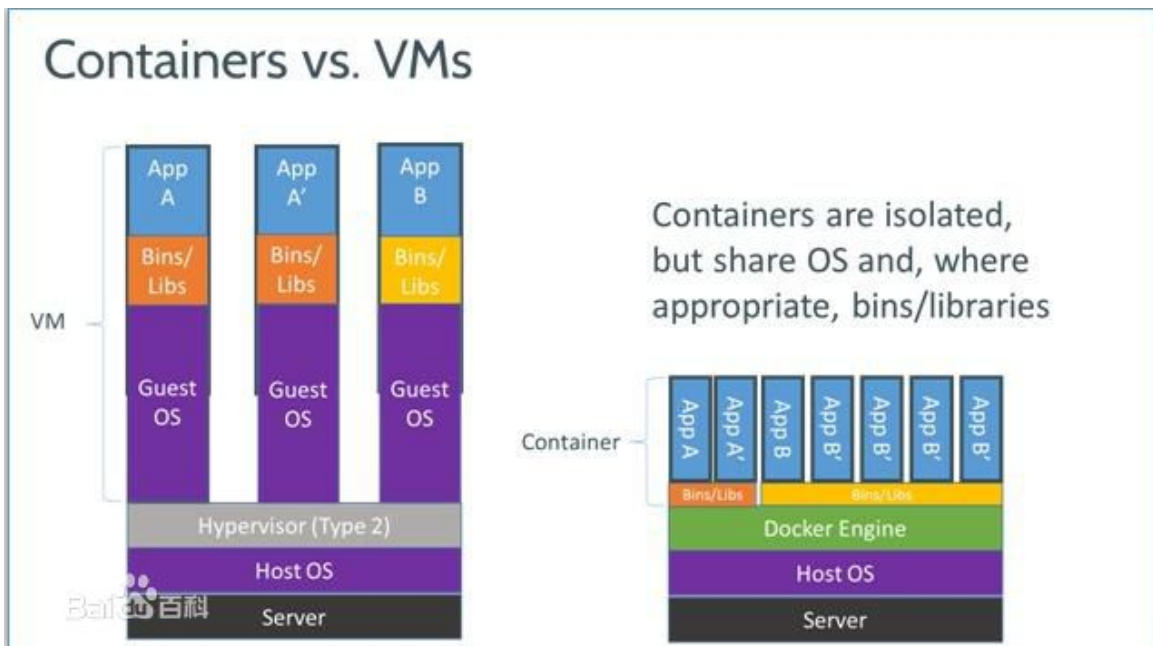


Fig 3. Docker与虚拟机对比

在学习的过程中，我也通过docker下载/创建了很多镜像，并尝试运行它们。

```
(base) sanjin@sanjin-VirtualBox:~/docker_bd$ sudo docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
bd30                 latest             7ecb31050291       2 days ago         987MB
<none>              <none>             08392f32b46c       2 days ago         246MB
<none>              <none>             42a340c27a85       2 days ago         246MB
python              2.7-slim           f462855313cd       4 weeks ago        137MB
python              3.7-slim           b5a7c089ece3       4 weeks ago        179MB
cassandra            latest             ce5334a7b86c       4 weeks ago        324MB
continuumio/miniconda3 latest             4a51de2367be       7 weeks ago        443MB
hello-world          latest             fce289e99eb9       9 months ago       1.84kB
(base) sanjin@sanjin-VirtualBox:~/docker_bd$ sudo docker container start 0e1b89d2997c
0e1b89d2997c
(base) sanjin@sanjin-VirtualBox:~/docker_bd$ sudo docker container ls
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS              NAMES
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS              NAMES
5698e7eb418f        hello-world        "/hello"           5 minutes ago       Exited (0) 5 minutes ago              elated_gagarin
2eff505719bf        bd30               "python app.py"    2 days ago          Exited (0) 2 days ago              BigDataWeb1
ec755bc46f0f        fe2e6abe6de7      "/bin/sh -c 'pip ins..." 2 days ago          Exited (2) 2 days ago              awesome_wiles
faa38980bae0        0b1ff460f7f3      "python app.py"    2 days ago          Exited (1) 2 days ago              BigDataWeb
965d168a84e9        e794b2b2c635      "/bin/bash"        2 days ago          Exited (0) 2 days ago              condescending_pare
80aa98c30a35        continuumio/miniconda3 "/bin/bash"        3 days ago          Exited (0) 3 days ago              vigilant_mendel
29d9d84fed15        8c662dbf759e      "/bin/sh -c 'conda e..." 5 days ago          Exited (1) 5 days ago              unruffled_basst
df1c02bb78ca        cassandra          "docker-entrypoint.s..." 2 weeks ago         Exited (143) 2 days ago              zjy-cassandra3
0e1b89d2997c        hello-world        "/hello"           3 weeks ago         Exited (0) 17 seconds ago              rlanboyant_nobel
(base) sanjin@sanjin-VirtualBox:~/docker_bd$
```

Fig 4. Docker镜像

运行-it指令，可以进入container进行查看：

```
(base) sanjin@sanjin-VirtualBox:~/docker_bd$ sudo docker run -it bd30 /bin/bash
root@031d03f91868:/app#
root@031d03f91868:/app#
root@031d03f91868:/app#
root@031d03f91868:/app#
root@031d03f91868:/app#
root@031d03f91868:/app#
root@031d03f91868:/app#
root@031d03f91868:/app#
```

Fig 5. Inner Docker container

Docker 与虚拟机最大的不同在于其存储镜像的方式。Docker 中的每一个镜像都是由一系列只读的层组成的，Dockerfile 中的每一个命令都会已在有的只读层上创建一个新的层：

当镜像被 docker run 命令创建时就会在镜像的最上层添加一个可写的层，也就是容器层，所有对于运行时容器的修改其实都是对这个容器读写层的修改。容器和镜像的区别就在于，所有的镜像都是只读的，而每一个容器其实等于镜像加上一个可读写的层，也就是同一个镜像可以对应多个容器。

2.3 Flask 与 RESTful API

Flask是一个轻量级的可定制框架，使用Python语言编写，较其他同类型框架更为灵活、轻便、安全且容易上手。它可以很好地结合MVC模式进行开发，开发人员分工合作，小型团队在短时间内就可以完成功能丰富的中小型网站或Web服务的实现。另外，Flask还有很强的定制性，用户可以根据自己的需求来添加相应的功能，在保持核心功能简单的同时实现功能的丰富与扩展，其强大的插件库可以让用户实现个性化的网站定制，开发出功能强大的网站。

Flask是目前十分流行的web框架，采用Python编程语言来实现相关功能。它被称为微框架(microframework)， “微”并不是意味着把整个Web应用放入到一个Python文件，微框架中的“微”是指Flask旨在保持代码简洁且易于扩展，Flask框架的主要特征是核心构成比较

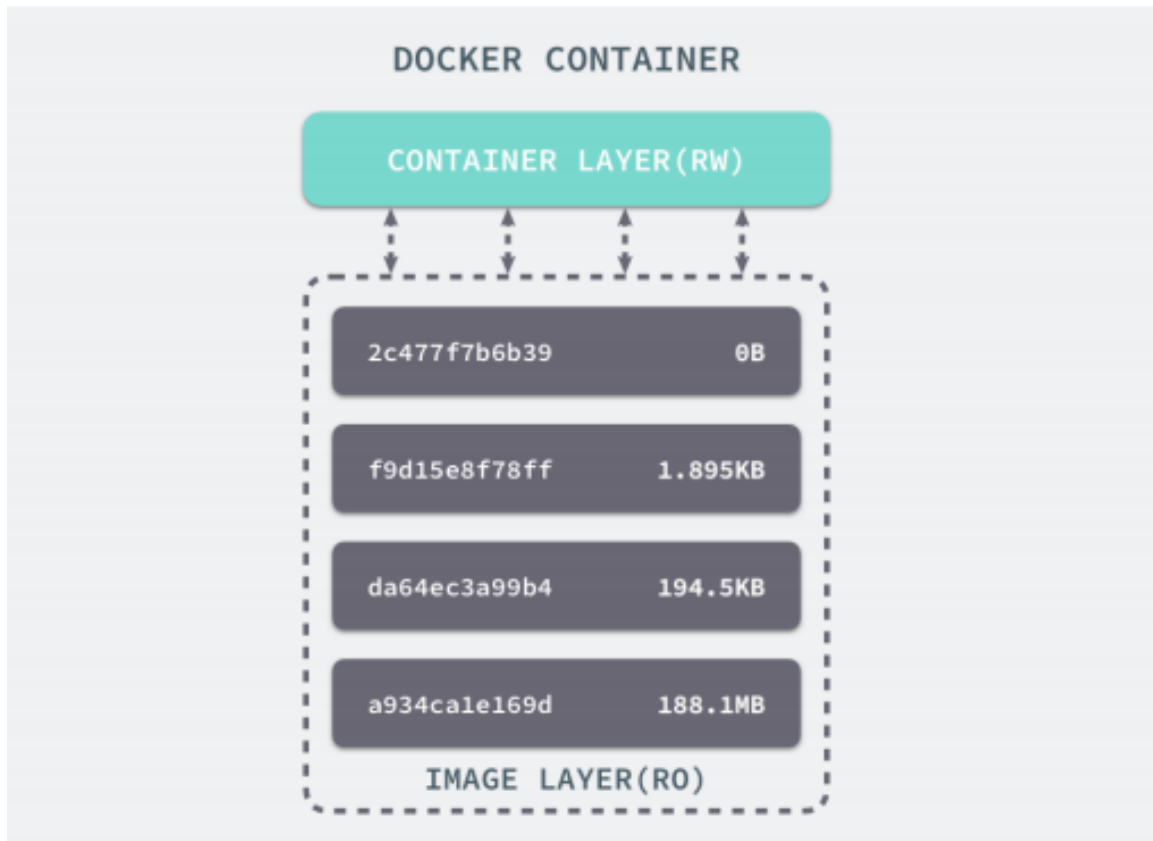


Fig 6. Inner Docker container

简单，但具有很强的扩展性和兼容性，程序员可以使用Python语言快速实现一个网站或Web服务。

HTTP Methods

HTTP (the protocol web applications are speaking) knows different methods for accessing URLs. By default, a route only answers to GET requests, but that can be changed by providing the `methods` argument to the `route()` decorator. Here are some examples:

```
from flask import request

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        do_the_login()
    else:
        show_the_login_form()
```

If `GET` is present, `HEAD` will be added automatically for you. You don't have to deal with that. It will also make sure that `HEAD` requests are handled as the [HTTP RFC](#) (the document describing the HTTP protocol) demands, so you can completely ignore that part of the HTTP specification. Likewise, as of Flask 0.6, `OPTIONS` is implemented for you automatically as well.

You have no idea what an HTTP method is? Worry not, here is a quick introduction to HTTP methods and why they matter:

The HTTP method (also often called "the verb") tells the server what the client wants to *do* with the requested page. The following methods are very common:

Fig 7. Flask教程

REST，即 Representational State Transfer 的缩写。它是一种互联网应用程序的 API 设计理念：URL 定位资源，用 HTTP 动词（GET,POST,DELETE,PUT）描述操作。

2.4 Cassandra

Cassandra是一套开源分布式NoSQL数据库系统。它最初由Facebook开发，用于储存收件箱等简单格式数据，集GoogleBigTable的数据模型与Amazon Dynamo的完全分布式的架构于一身Facebook于2008将Cassandra开源，此后，由于Cassandra良好的可扩展性，被Digg、Twitter等知名Web 2.0网站所采纳，成为了一种流行的分布式结构化数据存储方案。

Cassandra的主要特点就是它不是一个数据库，而是由一堆数据库节点共同构成的一个分布式网络服务，对Cassandra的一个写操作，会被复制到其他节点上去，对Cassandra的读操作，也会被路由到某个节点上面去读取。对于一个Cassandra集群来说，扩展性能是比较简单的事情，只管在群集里面添加节点就可以了。

与传统的关系型数据库不同，比起一致性，非关系型数据库更追求写入性能。CAP理论指出在一个分布式系统中，你只能强化其中两个方面：

- **Consistent:** 一致性，每次读取都是最新的数据
- **Available:** 可用性，客户端总是可以读写数据
- **Partition Tolerant:** 分区耐受性，数据库分散到多台机器，即使某台机器故障，也可以提供服务

Apache Cassandra 是一个高度可扩展的高性能分布式数据库，用于处理大量商用服务器上的大量数据，提供高可用性和耐受性，牺牲了一致性。

3 项目实现

3.1 训练MNIST模型

由于网上现存的MNIST代码有些过时，我就在实现项目的时候使用新的Python Estimator复现了原版代码：

Estimator的好处在于它可以自动保存模型，方便更换模型，使CNN的编写更加简单，随时切换模式，而不需要自己编写多余的代码，是避免Tensorflow更新过快使得代码过期的好方法。

```
train_input_fn = tf.estimator.inputs.numpy_input_fn(  
    x={"x": train_data},  
    y=train_labels,  
    batch_size=100,  
    num_epochs=None,  
    shuffle=True)  
mnist_classifier.train(  
    input_fn=train_input_fn,  
    steps=20000,  
    hooks=[logging_hook])
```

```

if mode == tf.estimator.ModeKeys.PREDICT:
    return tf.estimator.EstimatorSpec(mode=mode, predictions=predictions)

# Calculate Loss (for both TRAIN and EVAL modes)
loss = tf.losses.sparse_softmax_cross_entropy(labels=labels, logits=logits)
# Configure the Training Op (for TRAIN mode)
if mode == tf.estimator.ModeKeys.TRAIN:
    optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)
    train_op = optimizer.minimize(
        loss=loss,
        global_step=tf.train.get_global_step())
    return tf.estimator.EstimatorSpec(mode=mode, loss=loss, train_op=train_op)

# Add evaluation metrics (for EVAL mode)
eval_metric_ops = {
    "accuracy": tf.metrics.accuracy(
        labels=labels, predictions=predictions["classes"])}
return tf.estimator.EstimatorSpec(
    mode=mode, loss=loss, eval_metric_ops=eval_metric_ops)

```

Fig 8. Tensorflow Estimator

```

0.00000005 0.00023798 0.00023002 0.0002250 ]
[0.99979275 0.00000013 0.00001771 0.00000217 0.00000147 0.00005282
0.00009168 0.00000301 0.00002821 0.00001012]
[0.00001127 0.00042348 0.99888283 0.00052748 0.00000007 0.0000171
0.00002074 0.00000067 0.00011648 0.00000002]
[0.00039706 0.9836754 0.00193052 0.0032079 0.00107624 0.00103785
0.00282537 0.00055583 0.0049566 0.0003371 ]
[0.00000007 0.00040929 0.00002326 0.00006185 0.9914944 0.00006307
0.00053075 0.00000916 0.00580843 0.00159961]
[0.03024984 0.00034163 0.9335804 0.01831247 0.00008888 0.00277733
0.00146758 0.01211213 0.0009622 0.00010746]
[0.00282754 0.00000269 0.00002291 0.00000302 0.00000392 0.00297633
0.00095542 0.00000457 0.9931591 0.0000445 ]
[0.00011219 0.0022229 0.01001741 0.01132679 0.00086636 0.00335099
0.00108326 0.00003755 0.97072756 0.00025505]
[0.0012931 0.9875743 0.0016676 0.00053677 0.00097822 0.00004478
0.00031388 0.001046 0.00643482 0.00011071]
[0.00012088 0.00012232 0.00139961 0.17896153 0.00000143 0.81181306
0.00003418 0.00382451 0.00070594 0.00301646]
[0.0000232 0.9951698 0.00053252 0.00048797 0.0000774 0.0000068
0.0003489 0.00003198 0.00323875 0.00000257]
[0.9770343 0.0000002 0.00082748 0.00004838 0.0000035 0.01494976
0.0000131 0.00216929 0.00012862 0.00482544]
[0.00000111 0.00000001 0.00001286 0.00004628 0.00004402 0.00009041
0.00000002 0.00000063 0.9996629 0.00014158]
[0.00000055 0.00000176 0.0002258 0.00017003 0.00000007 0.00000102
0.
0.99837923 0.00000354 0.00121792]
[0.00003115 0.00148919 0.00179636 0.00105731 0.0424789 0.00024735
0.00000737 0.00373265 0.01124733 0.9379124 ]
[0.00003399 0.00000289 0.00015838 0.00000777 0.99150085 0.00000617
0.00010628 0.00010768 0.00001031 0.00806563] (8.906 sec)
INFO:tensorflow:loss = 0.070913315, step = 20101 (19.024 sec)
I1012 20:45:24.151432 140083805747008 basic_session_run_hooks.py:260] loss = 0.070913315, step = 20101 (19.024 sec)

```

Fig 9. Tensorflow 训练过程

3.2 测试图片准备

这里我们就直接选取MNIST数据集中TEST部分的数据进行提取，这些图片本身应该就是28*28*1的单通道灰阶图片，与网络的结构一致，这样就满足了测试的要求，首先我们先直观地看一下这个测试图片是数字几。

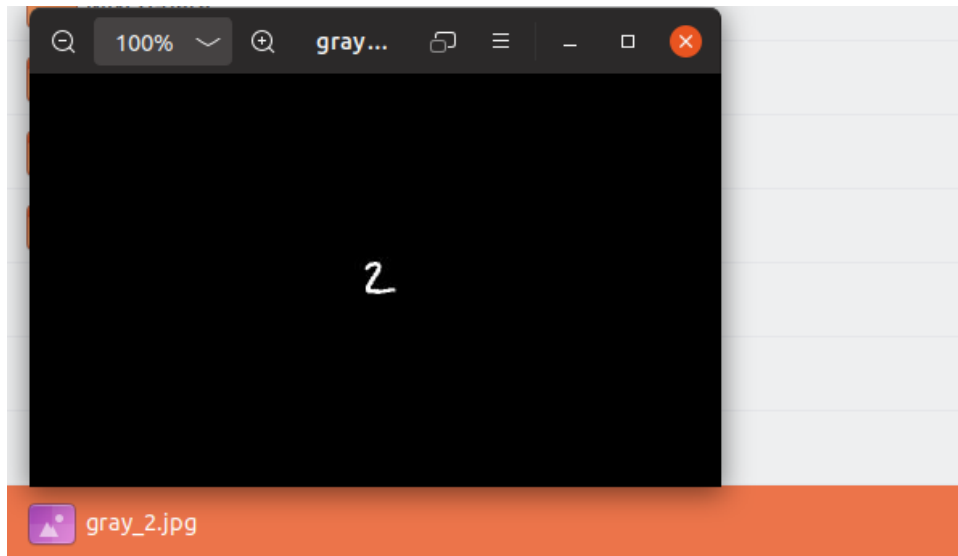


Fig 10. MNIST测试数据

很明显这是数字2，接下来我们会完成整个项目看看是否会做出正确的预测。

3.3 TensorFlow的本地预测

```
# Preparisations for tensorflow
# Imports
import numpy as np
import tensorflow as tf
from mnist_cnn import cnn_model_fn
#tf.logging.set_verbosity(tf.logging.INFO)
mnist_classifier = tf.estimator.Estimator(
    model_fn=cnn_model_fn,
    model_dir="/data/mnist_model")

def predict(file_path):
    image_raw = tf.gfile.FastGFile(file_path, 'rb').read()
    img = tf.image.decode_jpeg(image_raw)
    with tf.Session() as sess:
        img_ = img.eval()
        print(img_.shape)
        img_ = img_.astype(np.float32)
        pred_input_fn = tf.compat.v1.estimator.inputs.numpy_input_fn(
            x={"x":img_},
            num_epochs=1,
            shuffle=False)
        pred_results = mnist_classifier.predict(input_fn=pred_input_fn)
        for item in pred_results:
            pred = item['classes']
        return pred
```

可以看到，由于使用了Estimator，整个读取模型的步骤非常的简单，首先读取保存在本地的用户上传的图片，之后经过网络cnn_model_fn得到输出pred，即分类结果。

3.4 Docker部署

Dockerfile:

```
FROM python:3.7-slim
# Set the working directory to /app
WORKDIR /app
# Copy the current directory contents into the container at /app
COPY . /app
# Install any needed packages specified in requirements.txt
RUN pip install -i https://pypi.tuna.tsinghua.edu.cn/simple -r requirements.txt
# Run app.py when the container launches
CMD ["python", "app.py"]
```

requirements:

```
flask
wtforms
flask-wtf
werkzeug
cassandra-driver
numpy
tensorflow==1.14
```

3.5 Flask与cassandra通信

Flask的任务是负责把用户前台上传的图片post到后台并保存。这里我做了一些处理，来验证用户上传的图片是否真的符合要求：

```
class MyForm(Form):
    pic = FileField(u'Upload', validators=[FileRequired(), FileAllowed(['jpg', 'png', 'gif'])])

@app.route("/mnist", methods=['POST'])
def mnist():
    form = MyForm(CombinedMultiDict([request.form, request.files]))
    if form.validate():
        pic = request.files.get('pic')
        file_path = os.path.join(UPLOAD_PATH, pic.filename)
        pic.save(file_path)
        pred = predict(file_path)
        datetime = time.strftime("%Y-%m-%d %H:%M:%S")
        insertPic(pic.filename, pred, datetime)
        return ("%s%s%s%s%s%s%s%s%s" % ("Filename: ", pic.filename, "\n",
                                         "Pred: ", str(pred), "\n",
                                         "UpdateTime: ", datetime, "\n"))
    else:
        return 'Failed'
```

Cassandra的任务是将文件名，预测结果和上传时间保存在Cassandra数据库内。这里我们选定一下KEYSPACE，时区选择东八区，这里我们选择容器与容器之间的通信方式，就是将zjy-cassandra3这个地址绑定在另一个容器cassandra上。然后在docker上独立运行两个container。

```
KEYSPACE = "bigdataspace"
os.environ['TZ']='AEST-8AEDT-11,M10.5.0,M3.5.0'
time.tzset()
# cassandra
def insertPic(f,pred,time):
    cluster = Cluster(contact_points=['zjy-cassandra3'],port=9042)
    session = cluster.connect()
    log.info("Using keyspace...")
    try:
        log.info("setting keyspace...")
        session.set_keyspace(KEYSPACE)
        log.info("inserting to table...")
        session.execute("INSERT INTO MNIST (filename,pred,time) VALUES ('"+f+"','"+str(pred)+"','"+time+"')")
    except Exception as e:
        log.error("Unable to insert pic")
        log.error(e)
```

3.6 Volume挂载以及项目整体效果

```
sudo docker run -p 4000:80 -v /home/sanjin/docker_bd:/data
                    --name BigDataWeb
                    --network zjy-net
                    --link zjy-cassandra3:zjy-cassandra3
                    bd30
```

其中-v代表volume挂载，-name表示容器别名，-network表示与cassandra同属的网络，-link表示将两个容器之间链接以完成容器之间通信。这里值的注意的是，由于我顺便开启了Flask的表单认证机制，需要提交CSRF token，所以以下curl代码不会运行成功，接下来我展示的是正常用户访问网站的流程。

```
curl -F "file=@gray_2.jpg" localhost:4000/mnist
Failed.
```

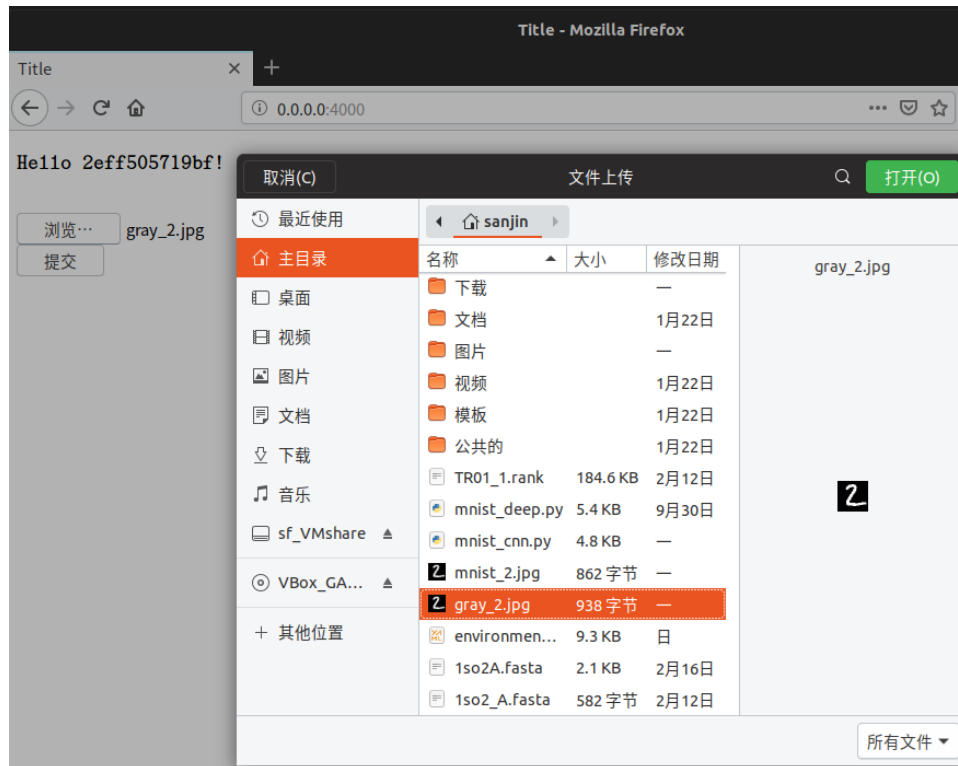


Fig 11. 网页效果

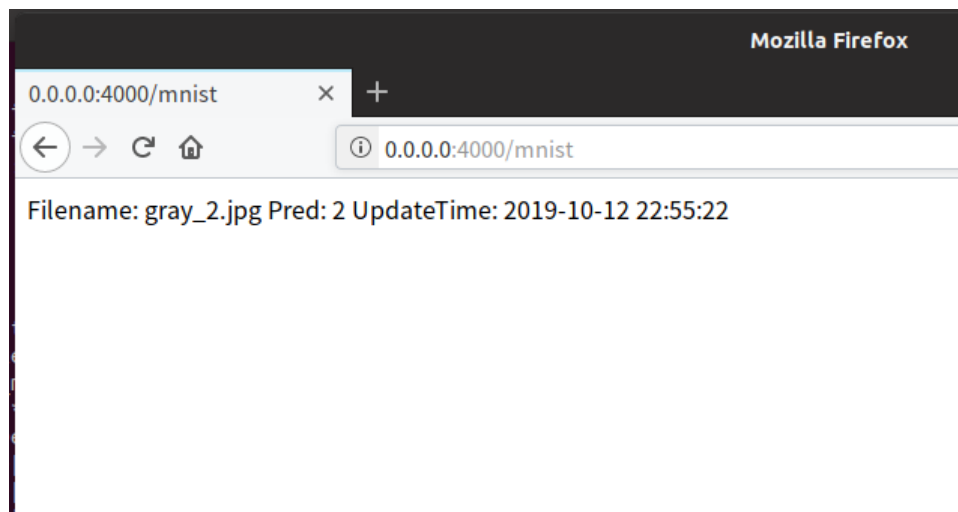


Fig 12. 提交后

```
(base) sanjin@sanjin-VirtualBox:~$ sudo docker run -it --network zjy-net --rm cassandra cqlsh zjy-cassandra3
Connected to Test Cluster at zjy-cassandra3:9042.
[cqlsh 5.0.1 | Cassandra 3.11.4 | CQL spec 3.4.4 | Native protocol v4]
Use HELP for help.
cqlsh> use bigdataspace;
cqlsh:bigdataspace> select * from MNIST;
```

filename	pred	time
test.jpg	0	2019-10-07 13:49:10.000000+0000
test0.jpg	0	2019-10-07 05:14:25.000000+0000
gray_2.jpg	2	2019-10-07 20:39:38.000000+0000
gray_2.jpg	2	2019-10-07 20:47:33.000000+0000
gray_2.jpg	2	2019-10-07 22:44:56.000000+0000
gray_2.jpg	2	2019-10-08 00:38:55.000000+0000
gray_2.jpg	2	2019-10-10 18:08:39.000000+0000
gray_2.jpg	2	2019-10-12 22:55:22.000000+0000

```
(8 rows)
cqlsh:bigdataspace>
```

Fig 13. 红笔圈出来的部分为数据库刚刚保存的图片

整个项目运行良好，对于这个测试图片也给出了正确的结果。

4 项目总结

在加入本项目之前，我对于大数据的了解只限于数学方面，即统计，聚类优化和机器学习。但在经过这一个月对产业界各类技术的学习后，我方才领略到大数据工程领域的技术细节，对其中的技术细节、丰富的生态与广阔的应用前景有了基本了解。

前期我接触了 Docker 及其相关技术。当然在学习Docker的过程中我遇到了相当大的麻烦，我本想原封不动地复制我的Conda环境，然而却发现整个Conda环境做成的Docker Image太大了，导致虚拟机无法运行，系统几度崩溃。但好在后来妥善解决了问题，在我看来使用Docker开发的最大好处就是它方便了人们的协作，就和Git一样，其次它的轻量化也很方便工业生产。我觉得这是一个十分有用的技术，这使基本没有全职开发经历的我收获很大。

而后学习的 Cassandra 和 Flask 等技术则是Docker部署的应用。这些容器完全可以独立起来作为一个小型试验服务器相互通信。原来Docker是可以被这么使用的，这的确给了我不少的启发。平时在学习不太能接触到这样具体的工业知识，不太考虑实用的时候做出的分治。就像本项目其实把Cassandra和Flask都写到一个程序里，放在一个主机上是最方便调试的，然而生产环境下出了问题则是最难调试的。所以虽然学习容器通信费了我不少时间，但事实证明这些努力是有价值的。

在项目的最后一节课上，老师给我们展示了一张大表，表上有各种各样的大数据技术，让我意识到现在的大数据产业界不可谓不繁荣了。但老师也说了一句话很扎我的心，他说就算是他也只掌握了其中的40%。我在本项目中实现Tensorflow就是一个扎心的过程。Tensorflow有1.0,1.2,1.4,1.14,1.15,2.0这么多的版本，谁能保证我学的版本1.14不会在1-2年后过时呢？Tensorflow还有GPU加速，自定义模型，自定义网络等等高深的操作，仅仅看Tensorflow，我学会的达到40%了么？世界一直在变化，我学会的还是太少太少。尽管我通过学习锻炼出了自主开发能力，但这种微薄的能力对于知识的海洋来说实在是太渺小了，我想不管世界如何变迁，我们的技术如何更新迭代，是不是会有一些不曾改变的东西，就像定理一样，学了后永远不会后悔呢？

抱着这样的想法，在面对新技术的时候，我时常犹豫要不要去学习，但我也清楚地知道，什么都不学是不行的，什么都学是不可能的。我还年轻，不想像夸父一样一直追着太阳，我希望能遇到一个科研方向，那个方向代表着真正的革新与未来，过千百年后仍能散发永恒的炽热，就像我们看待百年前的科学一样。希望我能在今后的学习中找到答案。