科研营总结报告

项目: 计算机大数据方向
学校：中国石油大学（北京）
专业：机械设计指导及其自动化
报告人：常冉
实习时间：2018.9.24-2018.10.21

目录

一.研究背景

随着大数据时代的到来， 如何借助计算机进行大数据的处理、分析、挖掘和可视化成为大数据时代要解决的首要任务。这个过程中需要借助于许多大数据科学方面的知识，深度学习，虚拟化技术，容器技术，*NoSQL*技术等具体的工程研究知识，并且需要运用通用的开发语言和云计算数据处理平台完成大数据的处理、分析、挖掘和可视化。

*Docker*作为一个开源的应用容器引擎，让开发者们可以将应用打包封装到一个容器中，然后部署到各种规模的环境中。*Docker*作为一种新兴的虚拟化方式，跟传统的虚拟化方式相比，可以更高效地利用系统资源、更快速的启动时间、一致的运行环境、持续交付和部署、更轻松的迁移以及更轻松的维护和扩展，这些优良的特性可以为大数据的研究提供更快的开发测试部署和更一致的环境，从而大大提升了效率。本次科研过程，我们用*Docker*将*Cassandra*和训练好的*Tensorflow*封装进两个容器，并进行容器间的联系，在*Web*前端，用户提交手写体数字图片，*Tensorflow*将图片识别出来后返回给用户，而每次提交的图片字节信息和时间戳等信息都记录到*Cassandra*中储存。

*Cassandra*是一套开源分布式*NoSQL*数据库系统，用于储存简单的格式数据集，本次科研中用于记录用户每次提交的信息以及识别结果等内容。

*Tensorflow*是一个开源软件库，可以用于各种感知和语言理解任务的机器学习。*MNIST*是一个包含各种手写数字图片的数据集，需要进行手写识别时，我们先用*MNIST*数据集训练*Tensorflow*，记录下训练结果，再将结果用于新的手写体鉴定。

二.研究过程

*1.*基础知识学习（*python*，*Github*， *Linux*）

这次的项目代码是由*python*语言编写完成的，*python*作为比较简洁、可读性较高的语言在大数据的研究中得到了广泛的应用。我的项目作业选择了*pycharm*开发环境。

*Github*是一个面向开源及私有软件项目的托管平台，相当于一个代码仓库，在项目过程中用到的许多代码，许多下载安装包都可以从*Github*上直接拉取下来，十分方便。

*Linux*的常用指令也是我们需要掌握的，可以在*Mac*的*Terminal*下实践，我们以此来快速安装各种程序和安装包，也可以查看*Docker*的镜像、容器信息等等，同时项目中用户提交图片也是用*curl-XPOST*命令提交。

以上三个方面是进行本次科研的基本功和基础知识，有了这些知识我们才能高效地完成项目。

2.项目研究过程

本次项目整体构建思路是构建一个手写数字识别系统，用户通过curl命令提交图片，系统返回对应的数字并且将本次提交的具体内容储存在数据库中。

为了完成这个项目，进行了分析和研究之后，我将它分成以下几个问题和模块构建：

- 如何通过cURL进行用户和系统之间的交互？“通过构建HTTP服务”
- 系统如何返回对应的数字？“通过预训练的Tensorflow模型，用模型进行预测”
- 系统将本次请求的内容存储到数据库中？“通过HTTP服务的后端与数据库建立联系，创建条目”

通过这三个问题，我们将整体的系统大致分为三个模块——HTTP服务模块、Tensorflow模型预测模块以及数据库存储模块，接下来我们依照问题的导向对这三个模块进行逐一的讲解。

2.1、用户如何通过curl和系统进行交互?

2.1.1 cURL介绍

首先，cURL是一个利用URL语法在命令行下工作的文件传输工具，它支持FTP、HTTP、HTTPS、Telnet等协议。在这里为了方便起见，我们尝试（使用课上讲授的技术）构建一个简单的HTTP服务器，用于接受cURL传递的HTTP POST请求。这里POST代表HTTP协议中的一种请求方法（常见的有GET、DELETE等），它的基本含义是向服务器上传一个文件或内容。所以，我们可以通过把图片文件附在HTTP的POST请求体（request body）中，达到将其上传到我们的HTTP服务器的目的。

cURL在我们的macOS系统上已经搭载，只需要在terminal里面执行简单的命令就可以访问我们HTTP服务器的端口，达到上传图片的目的。（可以插图片也可以不插）

2.1.2 HTTP服务器构建

我们的HTTP服务器构建选择在docker中，使用python的一个web框架——Flask来创建。Flask是使用python来方便构建web服务器的一个开源框架，它的template（前端渲染模板）使用的是Jinja框架，后端则使用（Werkzeug WSGI toolkit）。使用Flask，我们就能够在很短的时间内，通过短短几行代码创建一个标准的RESTful API（目前最流行的API 设计规范，用于Web 数据接口的设计），用来接收cURL的信息。

Flask的安装非常简单，我们可以直接使用pip来完成：

<code> pip install flask </code>

2.1.3 HTTP RESTful API设计

我们准备用Flask创建两个接口（RESTful API）用来与前端互动，一个是GET /train接口，用来直接通过请求来训练模型并将模型保存到后端服务器中；另一个就是要求的POST / 接口，用来接收用户的cURL请求，并完成一系列的操作：

1. 获取用户上传的图片
2. 把图片转变成MNIST模型可以处理的样式
3. 通过预训练的模型进行预测结果，并将结果格式化准备返回
4. 将本次处理的具体信息加入到数据库中
5. 返回本次处理结果，等待下一次上传

我们这里提到的接口是相对路径，相对于服务器的构建路径而言，对于我们的服务器来说，Flask的默认监听地址设置在0.0.0.0: 5000（5000为HTTP端口，用于区分不同服务），那么具体的URL就是http://0.0.0.0:5000/train，其余类似。

```
# coding: utf-8
from flask import Flask, request

SUCCESS_STRING = '''
<!doctype html>
<title>Test Result</title>
<h1>Your result for picture {} is </h1>
<p> {} </p>
'''
TRAIN_STRING = '''
<!doctype html>
<title>Train Complete</title>
<h1>Train Complete</h1>
'''
GET_STRING = '''
<!doctype html>
<title>Upload new File</title>
<h1>Upload new File</h1>
<form method=post enctype=multipart/form-data>
    <p><input type=file name=file>
    <input type=submit value=Upload>
</form>
    '''

app = Flask(__name__)


@app.route('/', methods=['GET', 'POST'])
def upload_file():
    if request.method == 'POST':
        # Get picture first
        # Then send it to out tf model, getting its predict ans
        # Save this request to Cassandra
        # Finally, return out formatted ans
        return SUCCESS_STRING.format('filename', 'ans')
    return GET_STRING


@app.route('/train', methods=['GET'])
def train():
    # Train tensorflow model
    # Finally return 'Train Success'
    return TRAIN_STRING


if __name__ == '__main__':
    app.run('0.0.0.0:5000')
```

具体的Flask代码非常简单，根据官网的教程，我们首先创建一个Flask app，然后通过这个app的route方法作为具体函数的decorator就可以进行路由：

实现的代码如上所示，在具体的实现中，我们没有返回简单的字符串，而是返回了一串HTML代码，当用户通过GET方法访问我们的URL的时候，给他返回一个上传图片的表格（form），从Flask的官方文档中我们发现，需要指定HTTP的enctype为multipart/form-data，这样Flask才能正确处理上传上来的文件。在POST方法结束的时候，我们返回格式化好的HTML字符串，向用户提示最后的预测结果和他们上传的文件名，方便用户识别。同样的，在/train接口也添加对应的HTML字符串。

2.2 通过Tensorflow预训练模型预测答案

在这一章中，有四个重要的问题需要我们解决。

其一，用户上传后的图片经由Flask框架处理，我们需要从Flask的框架中拿到对应的图片数据。如果用户没有上传正确的图片，我们向用户重新指引回GET方法的上传页面。

其二，我们需要通过URL预先训练我们的模型，将参数保存。

其三，我们需要将用户输入的图片变成我们的模型可以接收的格式（X*X的numpy布尔数组，0代表黑色，1代表白色），这样模型才能正常使用。

最后，我们需要将之前预训练好的模型参数载入到新的模型中，如果我们不载入参数，整个模型的效果会变得非常差，就无法满足我们系统的预期要求。

2.2.1从Flask框架中获取用户的上传图片

这一步在Flask的官方文档中有详细的介绍，Flask通过Werkzeug的Storage类来管理具体上传的文件并且进行了封装，我们只需要简单的几步代码就可以完成文件的获取，至于对文件类型和文件名的安全过滤，在文档中也有很详细的说明，我这里直接使用官方文档中的代码进行处理：

```
@app.route('/', methods=['GET', 'POST'])
def upload_file():
    if request.method == 'POST':
        # check if the post request has the file part
        if 'file' not in request.files:
            flash('No file part')
            return redirect(request.url)
        img = request.files['file']
        # if user does not select file, browser also
        # submit a empty part without filename
        if img.filename == '':
            flash('No selected file')
            return redirect(request.url)
        if img and allowed_file(img.filename):
            # save image to local
            img.read()
            filename = secure_filename(img.filename)
            file_path = os.path.join(app.config['UPLOAD_FOLDER'], filename)
            img.save(file_path)
            return SUCCESS_STRING.format(filename, ans)
    return GET_STRING
```

在这里，我们通过img.read方法将图片载入，再通过img.save方法将文件保存到对应的目录中，这里有两个地方需要注意，一个是allowed_file函数，另一个是app.config['UPLOAD_FOLDER']，它们都是官方文档自带的，用于过滤图片类型和指定保存图片的路径（为当前目录中的uploads文件夹），具体实现如下：

```
app.config['UPLOAD_FOLDER'] = "./uploads"
ALLOWED_EXTENSIONS = {'png', 'jpg', 'jpeg', 'gif'}

def allowed_file(filename):
    return '.' in filename and \
           filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS
```

2.2.2 通过URL预训练模型

对于Tensorflow的模型，我们没有使用学习资料里推荐的低阶API，而是使用了Tensorflow官网推荐的高阶API：Keras，通过这一API和其官方文档，我们可以方便的对MNIST数据集进行建模、训练以及储存其训练好的模型。官方推荐的训练和保存代码如下，我们将其加入到我们的代码中：

首先是模型的创建过程，它保证每次调用都能创建一个符合要求的新的模型，方便我们训练和保存加载参数，要注意的是这里第一层的input_shape被官方写成了784，这是对一个MNIST数据中28*28的矩阵进行压平后的大小，所以我们的输入尺寸也必须预处理到同样的大小。

```
def create_model():
    model = tf.keras.models.Sequential([
        tf.keras.layers.Dense(512, activation=tf.nn.relu, input_shape=(784,)),
        tf.keras.layers.Dropout(0.2),
        tf.keras.layers.Dense(10, activation=tf.nn.softmax)
    ])

    model.compile(optimizer=tf.keras.optimizers.Adam(),
                  loss=tf.keras.losses.sparse_categorical_crossentropy,
                  metrics=['accuracy'])

    return model
```

然后我们简单地把教程中的代码加入到已经设计好的/train接口中，这里文档使用了reshape方法对训练集和测试集的图片进行了处理，然后除以255.0将其转换为只包含01的矩阵。

```
@app.route('/train', methods=['GET'])
def train():
    # Create checkpoint callback
    cp_callback = tf.keras.callbacks.ModelCheckpoint(checkpoint_path,
                                                     save_weights_only=True,
                                                     verbose=1)
    mnist = tf.keras.datasets.mnist

    (x_train, y_train), (x_test, y_test) = mnist.load_data()
    x_train, x_test = x_train.reshape(-1, 28 * 28) / 255.0, x_test.reshape(-1, 28 * 28) / 255.0

    model = create_model()

    model.fit(x_train, y_train, epochs=5, callbacks=[cp_callback])
    return TRAIN_STRING
```

2.2.3 处理之前保存好的图片

用户直接上传给我们的图片是不能直接使用的，因为MNIST的模型只能够识别28*28的01矩阵，所以我们需要先做一些处理，在做这方面的时候我遇到了一些困难，不知道怎样处理才能够把它变成需要的格式，最后在StackOverflow这个网站上我找到了可以使用的答案：它使用了两个python包分别叫PIL和skimage，调用里面的函数就可以简单地将图片变成MNIST需要的格式，在最后一步这里我加入了跟训练里面一样的代码，将其调整为784的大小：

```
from PIL import Image
import numpy as np
from skimage.color import rgb2gray
from skimage.filters import threshold_sauvola

def load_image(file):
    im = Image.open(file)
    resized_im = im.resize((28, 28))

    pix = np.array(resized_im)
    img = rgb2gray(pix)
    window_size = 25
    thresh_sauvola = threshold_sauvola(img, window_size=window_size)
    binary_sauvola = img > thresh_sauvola
    return binary_sauvola.reshape(-1, 28 * 28)
```

2.2.4 载入保存好的参数并进行预测

最后在使用的时候非常简单，我们从新创建一个模型，并将保存好的参数加载给它，用它进行预测就可以了，在最后一句的地方，由于predict最后返回的是一个答案的向量，里面每个值代表取这个数字的概率，我们使用numpy的argmax函数取概率最大的坐标，就是预测的答案：

```
@app.route('/', methods=['GET', 'POST'])
def upload_file():
    if request.method == 'POST':
        # check if the post request has the file part
        if 'file' not in request.files:
            flash('No file part')
            return redirect(request.url)
        img = request.files['file']
        # if user does not select file, browser also
        # submit a empty part without filename
        if img.filename == '':
            flash('No selected file')
            return redirect(request.url)
        if img and allowed_file(img.filename):
            # save image to local
            img.read()
            filename = secure_filename(img.filename)
            file_path = os.path.join(app.config['UPLOAD_FOLDER'], filename)
            img.save(file_path)

            # get ans generated by tf
            model = create_model()
            model.load_weights(checkpoint_path)

            im = load_image(img)
            ans = np.argmax(model.predict(im))

            return SUCCESS_STRING.format(filename, ans)
    return GET_STRING
```

2.3 将本次请求的操作存储在数据库中

在数据库方面，由于我们不需要通过HTTP对数据库做完全的增删改查操作，只需要增加的部分就可以了。后端python与Cassandra进行操作需要安装Cassandra的driver，我在网上找到了python-cassandra的包作为驱动，使用了一下发现操作还是过于繁琐，需要手动写入类SQL的查询语句，随后在Flask的官网发现了Flask-CQLAlchemy这个python包，它对上述的驱动进行了完善的封装，可以很方便地进行增删改查的操作，还支持ORM对象关系模型，将每张表作为一个类来操作，大大简化了代码量和操作难度，我根据它的官方例子改出了我们需要的对象关系模型如下：

```python
# coding: utf-8
import uuid
from flask import Flask, request, redirect, flash
from flask_cqlalchemy import CQLAlchemy
import os
from datetime import datetime
from werkzeug.utils import secure_filename
import tensorflow as tf
import numpy as np

app = Flask(__name__)
app.config['CASSANDRA_HOSTS'] = ['db']
app.config['CASSANDRA_KEYSPACE'] = "cqlengine"
app.config['SECRET_KEY'] = "Your_secret_string"
checkpoint_path = "model/cp.ckpt"
db = CQLAlchemy(app)


class Picture(db.Model):
    uid = db.columns.UUID(primary_key=True, default=uuid.uuid4)
    name = db.columns.Text(required=False)
    path = db.columns.Text(required=False)
    created_at = db.columns.DateTime(required=False)
    data = db.columns.Blob(required=True)
    result = db.columns.SmallInt(required=True)
```

在这里，类Picture代表Cassandra中“cqlengine”Keyspace中的表名，里面的每个行代表一个数据域，Blob为单纯的二进制，方便存储图像。

然后，我们就可以方便地同步数据库中的表格，进行插入操作了，把它加在我们第一个API的下面：

这样就完成了存储的操作，每次create就会向数据库中插入一条记录，我们连接到docker中的Cassandra，就可以看到被插入的数据：

```python
@app.route('/', methods=['GET', 'POST'])
def upload_file():
    if request.method == 'POST':
        # check if the post request has the file part
        if 'file' not in request.files:
            flash('No file part')
            return redirect(request.url)
        img = request.files['file']
        # if user does not select file, browser also
        # submit a empty part without filename
        if img.filename == '':
            flash('No selected file')
            return redirect(request.url)
        if img and allowed_file(img.filename):
            # save image to local
            img.read()
            filename = secure_filename(img.filename)
            file_path = os.path.join(app.config['UPLOAD_FOLDER'], filename)
            img.save(file_path)

            # get ans generated by tf
            model = create_model()
            model.load_weights(checkpoint_path)

            im = load_image(img)
            ans = np.argmax(model.predict(im))

            # generate timestamp and save to Cassandra
            db.sync_db()  # 建表
            data = bytearray(im)  # 把图片变成byte数组
            Picture.create(
                name=filename,
                path=file_path,
                created_at=datetime.now(),
                data=data,
                result=ans,
            )
            # return ans
            return SUCCESS_STRING.format(filename, ans)
    return GET_STRING
```

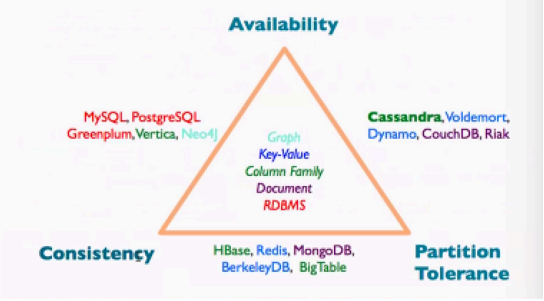
3.大数据的存储和处理

1）大数据存储

传统的数据库属于关系型列表，关系型数据库的增、删、查、改效率较低，生产环境下，使用*NoSQL*、*Cassandra*等非关系型数据库远远多于传统型关系型数据库。

*Cassandra*是一套开源分布式*NoSQL*数据库系统，用于储存收件箱等简单格式数据，*Cassandra*拥有良好的可扩展性和性能，被各大知名网站所采用，成了一种流行的分布式结构化数据存储方案。*Cassandra*最大的特点是可伸缩性，它可以用横向拓展（将两台机器变成四台机器，而非将两台机器进行升级）来提升性能。

大数据存储有三个原则，即*CAP*原则，一是一致性原则（*Consistency*），可用性（*Availability*）以及区域容错性（*Partition Tolerance*），一致性原则是指用户在访问数据库时，不论从哪个数据中心访问结果都是一样的，可用性原则是指数据库可以相应大量的请求读写，分区容错性是指系统在部分失灵时仍然可用。数据库的设计过程中，无法同时满足这三个原则，但是区域容错性（*P*）是必须要满足的，而一致性和可用性应该根据不同的场景作出取舍。



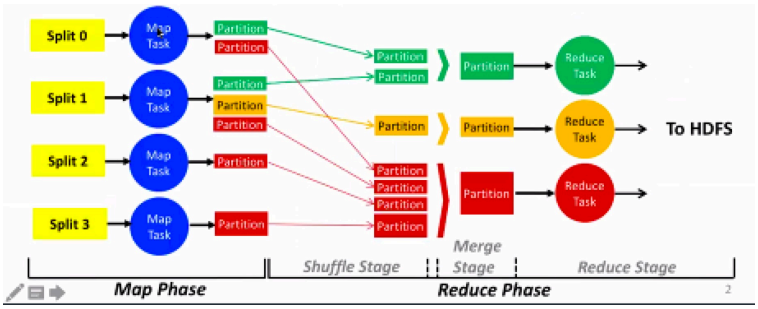
2）大数据处理

这次项目我们主要了解了*Hadoop*、*Mapreduce*等常用的大数据处理系统。

*Hadoop*是一个开源的框架，可编写和运行分布式应用处理大规模数据，为离线和大规模数据分析而设计。*Hadoop*是*Mapreduce*的开源工具。

*MapReduce*是用于大规模数据集并行运算的软件架构，当前的软件实现是指定一个*Map*（映射）函数，用来把一组键值对映射成一组新的键值对，指定并发的*Reduce*（归纳）函数，用来保证所有映射的键值对中的每一个共享相同的键组。

Mapreduce的具体实现如下图。首先有很多要处理的数据需要过滤和分析，Hadoop分布式文件系统由多块磁盘组成，每块的大小可以自己定义，把原始数据切成许多较小的split，分配到各个块，所有数据都进入maptest（数量和split相关），会产生输出partition，这个环节叫mapface。reduce task是把相同颜色的partition中的数据传输到一块，shuffle洗牌环节之后merge到一块，把一大块数据进入到reduce task里面，产生输出到HDFS里。



4.大数据整体框架和体系

关于大数据的整体框架与体系，成为云原生（*clouds native landscape*）。

首先是最底层，基础设施层，是由大量的硬件网络机器组成，比如构建很多机房和服务器，例如亚马逊的*web service*是龙头老大，除此之外还有公有云等等。

其次是供给层，切成小程序，例如网吧统一管理的工具、装软件等等，做切割的软件例如*ansible*，将脚本在所有的机器上运行一遍，用*ansible*传到每个机器上。

再次是运行层，在这一层多台机器的磁盘会合起来，就像在访问本地一样。本次我们学习的*Docker*就是这个用处。

再次是编排和管理层，这层中是各种规则的设定，例如*cpu*中使用率超过多少就要扩充一倍的机器等等，这一层中进行服务的发现和协同，并进行负载的均衡。

最后是最上层资源供给层，重点介绍了*API*管理技术，它用来做网页发布的重要内容。

三.项目感想与收获

这次项目总体来说我完成的有些吃力，因为这是我第一次接触关于大数据的项目，对于不论是软件的使用，还是背景知识的了解都十分有限。但是第一次接触大数据，我觉得它是非常有趣并且有意义的

老师为我们讲解了许多关于大数据的知识。课程总体来看分成四个部分：*1.*讲解关于*python*、*GitHub*、*Linux*等基础知识，让我们在应用起来不会受局限；*2.docker*容器技术，包括调用别人的技术和发送自己的服务，这也是这次项目的核心内容；*3.*大数据存储*Cassandra*技术；*4.*以*Mapreduce*为原型讲解了大数据可视化工具，呈现一张云原生图（*cloud native landscape*），对大数据生态呈现全貌。

总体来说，这次项目为我留下了非常深刻的印象，我深刻认识到大数据的知识绝非我之前所想的那么简单，它也需要许多许多互联网的知识，我也认识到自己还有很长的路要走。