The Implementation of Deploying the Mnist Application in the Container and Using Cassandra to Store the Data.

Zhifan Gao

Content:

[1. Introduction 3](#_Toc9163651)

[2. Training Model 4](#_Toc9163652)

[3. app.py 5](#_Toc9163653)

[1. Model Loading 5](#_Toc9163654)

[2. Using Flask to Upload ,Download and Use Pictures 6](#_Toc9163655)

[3. The Use of Cassandra 10](#_Toc9163656)

[4. Creation and Operation of Containers 13](#_Toc9163657)

[5. Improvements 16](#_Toc9163658)

[6. Reference 17](#_Toc9163659)

# Introduction

Handwriting recognition has always been a classic topic in deep learning. In order to facilitate people's use, we hope to write an app that allows people to upload pictures on the Web page, and then get predicted numbers of their handwriting. Additionally, the information of upload time, filename and the prediction number can also be recorded in the database at the same time, since the information is conducive to further model checking and data analysis. To achieve this goal, the knowledge of python is required, and at the same time, some simple operations of Cassandra databases and Docker containers can also be applied.

# Training Model

At the beginning, the data of Mnist dataset should be used to train a model. A simple regression model can be chosen (in order to achieve higher accuracy, different kinds of complex models can be chosen to do this step. Since the model is not an important part of the goal, a simpler model is considered to use here).

So, the first step, set the variable 'x' as the input image pixel vector data, 'w' as the weight, 'b' as the deviation, 'y' as the output.

**def** regression(x):  
 W = tf.Variable(tf.zeros([784, 10]), name=**"W"**)  
 b = tf.Variable(tf.zeros([10]), name=**"b"**)  
 y = tf.nn.softmax(tf.matmul(x, W) + b)  
  
 **return** y, [W, b]

Then, in this process of training the model, the gradient descent method can be used to constantly adjusting 'w' and 'b' to minimize the cross-entropy. The function of argmax() can return the subscript of the maximum value, which is the predicted answer, and accuracy of the model is also calculated in the code.

*#train*y\_ = tf.placeholder(**"float"**, [**None**, 10])  
cross\_entropy = -tf.reduce\_sum(y\_ \* tf.log(y))  
train\_step = tf.train.GradientDescentOptimizer(0.01).minimize(cross\_entropy)  
correct\_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y\_, 1))  
accuracy = tf.reduce\_mean(tf.cast(correct\_prediction,tf.float32))

Finally, the trained model can be saved as a .ckpt file, so if the model should be called in the further steps, the model does not have to be trained over and over again.

# app.py

Totally there are three parts in this file. The first part is to load the model and define the functions. Next, Flask can be used to achieve the goal of upload, download and use the picture. Finally, the communication between the containers should be possible in the code.

## Model Loading

First, importing the variables from the model.py file in the ‘mnist’ folder, and load the model which is in the 'ckpt' folder stored in the previous step.

**from** mnist **import** model  
  
x = tf.placeholder(**"float"**, [**None**, 784])  
sess = tf.Session()  
  
**with** tf.variable\_scope(**"regression"**):  
 y1, variables = model.regression(x)  
saver = tf.train.Saver(variables)  
saver.restore(sess, **"mnist/ckpt/regression.ckpt"**)

Then define the function of regression and prediction, which can be used in the further steps.

**def** regression(input):  
 **return** sess.run(y1, feed\_dict={x: input}).flatten().tolist()  
  
**def** predict(image\_path):  
 img = Image.open(image\_path).convert(**'L'**)  
 flatten\_img = np.reshape(img, 784)  
 input = np.array([1 - flatten\_img])  
 output = regression(input)  
 **return** np.argmax(output)

## Using Flask to Upload ,Download and Use Pictures

In order to achieve the goal that users can upload pictures and do the prediction of handwritten numbers, a simple flask structure can be written in this file.

The upload folder address and allowed file extensions should be set at the beginning. The file size and some other requirements can also be set according to different requirements.

ALLOWED\_EXTENSIONS = set([**'png'**, **'jpg'**, **'jpeg'**, **'gif'**])  
app = Flask(\_\_name\_\_)  
app.config[**'UPLOAD\_FOLDER'**] = os.getcwd() + **'/upload/'**

Next, a function can be created to determine whether the users have uploaded the image file that follows the previously requested to file extensions.

**def** allowed\_file(filename):  
 **return '.' in** filename **and** \  
 filename.rsplit(**'.'**, 1)[1] **in** ALLOWED\_EXTENSIONS

The first decorator, which can upload files is defined by this code.

**from** flask **import** send\_from\_directory

@app.route(**'/upload/<filename>'**)  
**def** uploaded\_file(filename):  
 **return** send\_from\_directory(app.config[**'UPLOAD\_FOLDER'**],  
 filename)

In order to get pictures and make predictions, another decorator should be defined.

The obtained file firstly should be compressed to the required pixel size for recognition.

file = request.files[**'file'**]  
 size = (28, 28)  
 im = Image.open(file)  
 im.thumbnail(size)

If the uploaded picture meets the requirements, it can be saved. And then, the path of the picture needs to be obtained, and the model trained before can predict the handwriting number that the image shows. Also, the function can get the time node of the uploading the picture. These are also information that should be stored in the database.

**if** file **and** allowed\_file(file.filename):  
 filename = secure\_filename(file.filename)  
 im.save(os.path.join(app.config[**'UPLOAD\_FOLDER'**], filename))  
 file\_url = url\_for(**'uploaded\_file'**, filename=filename  
 test\_result = predict(**'.'** + file\_url)  
 current=time.strftime(**"%Y/%m/%d %H:%M:%S"**)  
   
A .html file is required to facilitate the submission of pictures. Then the predicted results can be returned to the web page. Create a template folder and add the 'base. HTML' file to the folder.

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**>图片上传</**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**>结果</**h4**>  
 <**p class="text-info "**>  
 {{ result }}  
 </**p**>  
 </**center**>  
 </**div**>  
 </**div**>  
</**center**>  
{% endif %}  
  
</**body**>  
</**html**>

Here is the complete Implementation of code：

**import** os

**from** werkzeug.utils **import** secure\_filename  
**from** flask **import** Flask, request, url\_for，render\_template  
**import** time

@app.route(**'/'**, methods=[**'GET'**, **'POST'**])  
**def** upload\_file():  
 **if** request.method == **'POST'**:  
 file = request.files[**'file'**]  
 size = (28, 28)  
 im = Image.open(file)  
 im.thumbnail(size)  
 **if** file **and** allowed\_file(file.filename):  
 filename = secure\_filename(file.filename)  
 im.save(os.path.join(app.config[**'UPLOAD\_FOLDER'**], filename))  
 file\_url = url\_for(**'uploaded\_file'**, filename=filename)  
 print(file\_url)  
 test\_result = predict(**'.'** + file\_url)  
 current=time.strftime(**"%Y/%m/%d %H:%M:%S"**)  
 print(**"Test result is {}"**.format(test\_result))  
 **return** render\_template(**'base.html'**, result=test\_result)  
 **return** render\_template(**'base.html'**)

**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 app.run(debug=**True**)

## The Use of Cassandra

Cassandra is chosen to store the information in this process. First, pulling the image of Cassandra down from docker, and then create a new Cassandra image through docker. This can be achieved by executing the following code in the command line.

A subnetwork should be built first.

docker network create some-network

Name a new name to Cassandra image, build it on a subnet, and map it to port 9042 of the host.

docker run --name gzf-cassandra --network some-network -p 9042:9042 -d cassandra:latest

Once it's built, docker can be run by the following command.

docker run -it --network some-network --rm cassandra cqlsh gzf-cassandra

The following can show that the image is built and run successfully.



Next, it is necessary to create a table in the database to store information, such as the time of uploading, file name and predicted the number.

In order to make it easier for people to eliminate errors when errors occur, the following code can generate logs.

log = logging.getLogger()  
log.setLevel(**'INFO'**)  
handler = logging.StreamHandler()  
handler.setFormatter(logging.Formatter(**"%(asctime)s[%(levelname)s] %(name)s: %(message)s"**))  
log.addHandler(handler)

Then a keyspace needs to be set to create a table.  
KEYSPACE = **"mykeyspace"**

Another important step is to create a function to connect to the Cassandra image just created. Previously, the Cassandra image was built and mapped to port 9042 of the host to connect to Cassandra for data transmission.  
**def** createKeySpace():  
 cluster = Cluster(contact\_points=[**'127.0.0.1'**],port=9042)  
 session = cluster.connect()

Next, we create three columns for the created tables, which will record the information of uploading time, file name and the predicted number.

Complete Implementation of Code：

**import** logging  
**from** cassandra.cluster **import** Cluster

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 mytable (  
 mykey text,  
 col1 text,  
 col2 text,  
 PRIMARY KEY (mykey,col1)  
 )  
 """**)  
 **except** Exception **as** e:  
 log.error(**"Unable to create keyspace"**)  
 log.error(e)  
  
createKeySpace();

Then, define a function for data inserting.

**def** insert\_into\_cassandra(var1, var2, var3):  
 sql = **"INSERT INTO mykeyspace.mytable (mykey, col1, col2) VALUES ('%s', '%s', '%s');"""** % (var1, var2, var3)  
 session.execute(sql)

In the second decorator just set up, add the corresponding statement to insert information into the database.

insert\_into\_cassandra(current,filename, test\_result)

In this way, the goal of uploading files, prediction, and inserting data into the database can be completed when file runs locally.

The figure below is the interface that users can see when submitting images on a web page.



# Creation and Operation of Containers

Next, in order to make it more convenient for others use, the program can be packed into a container. When running locally, the Cassandra image can be connected directly to the program. But with the blocking of containers, the program may not be able to connect directly to Cassandra. So the communication problem between containers should be considered. One way to achieve communication between containers is:

To find the IP address of Cassandra image built before and let the ‘contact\_points’ be the IP address to realize the communication between containers.

Here are some modifications to the app.py files.

**def** createKeySpace():  
 cluster = Cluster(contact\_points=[**'172.18.0.2'**],port=9042)  
 session = cluster.connect()

we should define an exposed port for a closed container, the 80 ports can be used.

**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 app.run(host=**'0.0.0.0'**,port=80,debug=80)

Next, in the process of building a new image that can contain the program written before, there are some basic frameworks and statements that can be found on docker's official website. To build a container, at least three files are required：app.py, dockerfile, and requirement.txt

dockerfile:

dockerfile defines what goes on in the environment inside the container. it is a text file that contains instructions that describe how a container should be built. The corresponding code can be found on docker's official website.

# Use an official Python runtime as a parent image

FROM python:3.6.5-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 --trusted-host pypi.python.org -r requirements.txt

# Make port 80 available to the world outside this container

EXPOSE 80

# Define environment variable

ENV NAME World

# Run app.py when the container launches

CMD ["python", "app.py"]

In this case, port 80 of the container is chosen to be exposed, which will facilitate subsequent use.

requirement.txt:

All the packages the program needed should be put in the requirements. In this way, when creating the image, the packages will be downloaded automatically.

tensorflow

numpy

pillow

werkzeug

flask

cassandra-driver

app.py:

The file that should be run.

The templates which contain the html file should be put in the folder, as well as the .ckpt file which stored the predict model.

So after putting all of these folders and files in a container, the image can be created.

The following code is used to create the image. The name of tag can be decided freely, and the dot at the end of this command should be paid more attention.

docker build –tag=mnist16 .

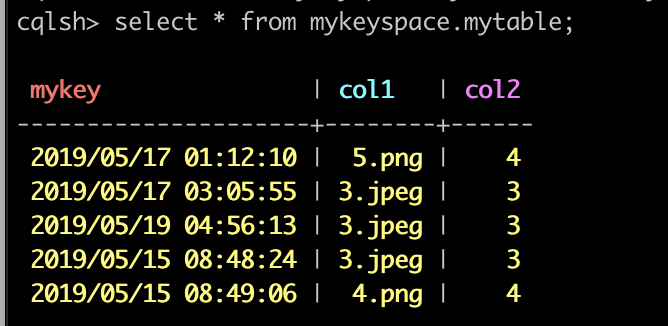
Since at the beginning, the image of Cassandra has been created on the subnetwork called some-network. In order to link with each other, both the Cassandra image and the mnist16 image should run on the same network. So ‘--network some-network’ should be in the command.

Finally, map the 80 ports exposed by the container to 4000 ports of the host.

docker run -it --network some-network -p 4000:80 mnist16

After this step, the image of mnist16 which can achieve the goal can be run smoothly in the container. Then, users are able to upload pictures on the Web and get predictions, and the information has been inserted in the table space which we create before.





# Improvements

There are many things that can be improved in this process. For instance, the algorithm of training the model can be improved. The model used can only gain an accuracy rate of 91%. However, for real handwriting recognition projects, this accuracy is not enough. In this process, some simple information can be inserted into the database. But there must be some other communication methods between containers, which should be used in different and more complex situation. And these problems are worth thinking about.

# Reference

Get Started, Part 2: Containers. Retrieved from https://docs.docker.com/get-started/part2/

Sugi, Y. (2018). tensorflow-mnist. Retrieved from https://github.com/sugyan/tensorflow-mnist

Quickstart. Retrieved from http://flask.pocoo.org/docs/0.12/quickstart/#a-minimal-application