Final Project Report

-----Big Data Program

Yuting Xue
2019.01.18

Content

1. Introduction

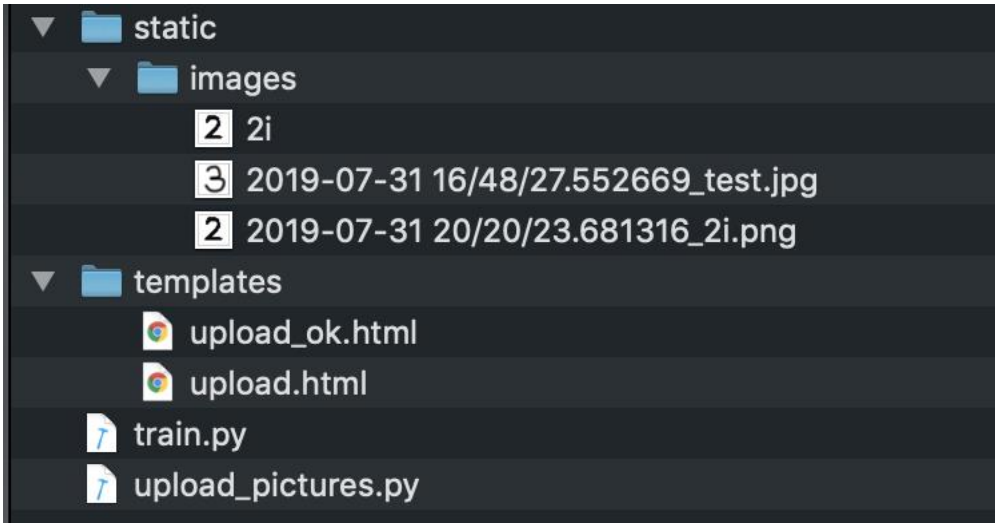
In this project, we want to build an application that recognizes a user-uploaded image of a handwritten digit using technologies like Docker, Flask, and Cassandra. While the user submits a picture through the website built by Flask, the Flask Router first to the image file it contains and make sure it has the right file type. The Router then saves the image and requests the prediction from the MNIST Tensorflow model stored on the server. After the result be returned, the Router forwards the result to the user and submits all four data to the Cassandra Database Container through the Docker Network Bridge.

The Cassandra Database records the user's IP address, the service request time, the predicted result, and the path of uploaded image. The MNIST model it is currently using is trained by 20000 steps using CNN algorithm and provides the prediction service with an accuracy of 0.97.

2. Research Process

2.1 Build the Website by Flask

The first thing we want to do is to build a website that can take in users' requests, which are to remotely upload pictures to the server, and to post the picture to the front-end webpage. The approach I used is as the follows:



The "static/images" directory is used to store the uploaded images.
The two html files under "templates" directory are used to define the display webpage.
The "upload_pictures.py" file is the engineering code to actually make the webpage work.
The code of "upload_pictures.py" is as follows:

```
# coding:utf-8
import os
import flask
import io
import datetime
import time
from flask import Flask, request, jsonify, render_template
from werkzeug.utils import secure_filename

import tensorflow as tf
import numpy as np
from PIL import Image
import train
import predict
import connect
```

```
ALLOWED_EXTENSIONS = set(['png', 'jpg', 'JPG', 'PNG', 'bmp'])

CKPT_DIR = 'model'


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


app = Flask(__name__)
app.config['UPLOAD_FOLDER'] = 'static/images/'


def parse(name, time):
    temp = []
    temp.append(str(time))
    temp.append('_')
    temp.append(name)
    name = ''.join(temp)
    return name


@app.route('/upload', methods=['POST', 'GET'])
def upload():
    req_time = datetime.datetime.now()
    if request.method == 'POST':
        f = request.files['file']
        upload_filename = secure_filename((f).filename)

        if not (f and allowed_file(f.filename)):
            return jsonify({"error": 1001, "msg": "Types of files only limited to png, PNG, jpg, JPG, bmp"})

        save_filename = parse(upload_filename, req_time)
        save_filepath = os.path.join(
            app.root_path, app.config['UPLOAD_FOLDER'], save_filename)
        f.save(save_filepath)
        mnist_result = (predict.predict(save_filepath))
```

```
        user_input = request.form.get("name")

        basepath = os.path.dirname(__file__)

        f.save(save_filepath)

        mnist_result = str(predict(save_filepath))
        connect.insertData(request.remote_addr, req_time,
                           save_filepath, mnist_result)

        result1 = "%s%s%s%s%s%s%s%s%s" % ("Upload File Name: ", upload_filename, "\n",
                                          "Upload Time: ", req_time, "\n",
                                          "Prediction: ", mnist_result, "\n")

        return render_template('upload_ok.html', userinput=user_input, val1=time.time(), result=result1)

    return render_template('upload.html')


# Go to '127.0.0.1:8987/upload' on the server computer; if on other computers,
# just replace '127.0.0,1' with ip address of the server computer.
if __name__ == '__main__':
    if not os.path.exists(app.config['UPLOAD_FOLDER']):
        os.makedirs(app.config['UPLOAD_FOLDER'])
    # app.debug = True
    # app.run(debug=True, use_reloader=False, host='0.0.0.0')
    app.run(debug=True, use_reloader=False, port=8987, host='0.0.0.0')
```

The code in “upload.html” is:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>Flask Picture Upload</title>
</head>
```

```
<body>
    <h1>Local Picture Upload</h1>
    <form action="" enctype='multipart/form-data' method='POST'>
        <input type="file" name="file" style="margin-top:20px;" />
        <br>
        <i>Please type in the number you expect to get:</i>
        <input type="text" class="txt_input" name="name" value="" style="margin-top:10px;" />
        <input type="submit" value="Upload" class="button-new" style="margin-top:15px;" />
    </form>
</body>

</html>
```

The code in “upload_ok.html” is:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>Flask Picture Upload</title>
</head>

<body>
    <h1>Local Picture Upload</h1>
    <form action="" enctype='multipart/form-data' method='POST'>
        <input type="file" name="file" style="margin-top:20px;" />
        <br>
        <i>Please type in the number you expect to get:</i>
        <input type="text" class="txt_input" name="name" value="" style="margin-top:10px;" />
        <input type="submit" value="Upload" class="button-new" style="margin-top:15px;" />
    </form>
</body>

</html>
```

Then we can run the “upload_pictures.py” directly. As we already defined port 8987, we will have to go to “127.0.0.1:8987/upload”, and we see the webpage as the following:

Local Picture Upload

Choose File No file chosen

Please type in the number you expect to get: Upload

Click “Choose File” button to choose the picture to be analyzed and click “Upload” to submit the picture. Then the website will automatically return the uploaded file name, upload time, and the predicted number.

2.2 Train the Model

In this part, we use the Convolutional Neural Networks (CNNs/ConvNets) to train the model. CNNs are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other. And they still have a loss function (e.g. SVM/Softmax) on the last (fully-connected) layer and all the tips/tricks we developed for learning regular Neural Networks still apply.

CNN architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the amount of parameters in the network.

Here are the codes used to train and restore the MNIST model:

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function


import argparse
```

```
import sys
import tempfile

from tensorflow.examples.tutorials.mnist import input_data

import tensorflow as tf

FLAGS = None


def deepnn(x):
    """deepnn builds the graph for a deep net for classifying digits.
    Args:
      x: an input tensor with the dimensions (N_examples, 784), where 784 is the
      number of pixels in a standard MNIST image.
    Returns:
      A tuple (y, keep_prob). y is a tensor of shape (N_examples, 10), with values
      equal to the logits of classifying the digit into one of 10 classes (the
      digits 0-9). keep_prob is a scalar placeholder for the probability of
      dropout.
    """
    # Reshape to use within a convolutional neural net.
    # Last dimension is for "features" - there is only one here, since images are
    # grayscale -- it would be 3 for an RGB image, 4 for RGBA, etc.
    with tf.name_scope('reshape'):
        x_image = tf.reshape(x, [-1, 28, 28, 1])

    # First convolutional layer - maps one grayscale image to 32 feature maps.
    with tf.name_scope('conv1'):
        W_conv1 = weight_variable([5, 5, 1, 32])
        b_conv1 = bias_variable([32])
        h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)

    # Pooling layer - downsamples by 2X.
    with tf.name_scope('pool1'):
        h_pool1 = max_pool_2x2(h_conv1)
```

```
  # Second convolutional layer -- maps 32 feature maps to 64.
  with tf.name_scope('conv2'):
    W_conv2 = weight_variable([5, 5, 32, 64])
    b_conv2 = bias_variable([64])
    h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)

  # Second pooling layer.
  with tf.name_scope('pool2'):
    h_pool2 = max_pool_2x2(h_conv2)

  # Fully connected layer 1 -- after 2 round of downsampling, our 28x28 image
  # is down to 7x7x64 feature maps -- maps this to 1024 features.
  with tf.name_scope('fc1'):
    W_fc1 = weight_variable([7 * 7 * 64, 1024])
    b_fc1 = bias_variable([1024])

    h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
    h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

  # Dropout - controls the complexity of the model, prevents co-adaptation of
  # features.
  with tf.name_scope('dropout'):
    keep_prob = tf.placeholder(tf.float32)
    h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

  # Map the 1024 features to 10 classes, one for each digit
  with tf.name_scope('fc2'):
    W_fc2 = weight_variable([1024, 10])
    b_fc2 = bias_variable([10])

    y_conv = tf.matmul(h_fc1_drop, W_fc2) + b_fc2
  return y_conv, keep_prob


def conv2d(x, W):
  """conv2d returns a 2d convolution layer with full stride."""
  return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')
```

```
def max_pool_2x2(x):
    """max_pool_2x2 downsamples a feature map by 2X."""
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
                          strides=[1, 2, 2, 1], padding='SAME')


def weight_variable(shape):
    """weight_variable generates a weight variable of a given shape."""
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)


def bias_variable(shape):
    """bias_variable generates a bias variable of a given shape."""
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)


def main(_):
    # Import data
    mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot=True)

    # 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('adam_optimizer'):
      train_step = tf.train.AdamOptimizer(1e-4).minimize(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)
  accuracy = tf.reduce_mean(correct_prediction)

  graph_location = tempfile.mkdtemp()
  print('Saving graph to: %s' % graph_location)
  train_writer = tf.summary.FileWriter(graph_location)
  train_writer.add_graph(tf.get_default_graph())

  saver = tf.train.Saver()  # Define the saver

  with tf.Session() as sess:
      sess.run(tf.global_variables_initializer())
      for i in range(20000):
          batch = mnist.train.next_batch(50)
          max_accu = 0
          if i % 100 == 0:
              train_accuracy = accuracy.eval(feed_dict={
                  x: batch[0], y_: batch[1], keep_prob: 1.0})
              if train_accuracy > max_accu:
                  max_acc = train_accuracy
                  saver.save(sess, 'model/model.ckpt', global_step=i+1)
              print('step %d, training accuracy %g' % (i, train_accuracy))
          train_step.run(
              feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})
      # # model saving location
      # saver.save(sess, '../mnist-app/model/')

      print('test accuracy %g' % accuracy.eval(feed_dict={
          x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))


if __name__ == '__main__':
```

```
    parser = argparse.ArgumentParser()
    parser.add_argument('--data_dir', type=str,
                        default='/tmp/tensorflow/mnist/input_data',
                        help='Directory for storing input data')
    FLAGS, unparsed = parser.parse_known_args()
    tf.app.run(main=main, argv=[sys.argv[0]] + unparsed)
```

2.3 Restore the Model to predict

After saving the models trained using CNN algorithms, we want to restore the most recent one that was stored and apply it to the pictures of hand-written digits that users posted. To achieve this goal, we reopen a session and retain the model:

```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
import cv2
import tensorflow as tf

def resize(input):
    image = Image.open(input)
    image = image.resize((28, 28), Image.ANTIALIAS)
    image = image.convert('L')
    tv = list(image.getdata())
    tva = [(255-x)*1.0/255.0 for x in tv]
```

```
    # gray = cv2.cvtColor(resized, cv2.COLOR_BGR2GRAY)
    # ret, thresh = gray.threshold(gray, 127, 255, cv2.THRESH_BINARY)
    return tva


FLAGS = None
def deepnn(x):
    """deepnn builds the graph for a deep net for classifying digits.
    Args:
      x: an input tensor with the dimensions (N_examples, 784), where 784 is the
      number of pixels in a standard MNIST image.
    Returns:
      A tuple (y, keep_prob). y is a tensor of shape (N_examples, 10), with values
      equal to the logits of classifying the digit into one of 10 classes (the
      digits 0-9). keep_prob is a scalar placeholder for the probability of
      dropout.
    """
    # Reshape to use within a convolutional neural net.
    # Last dimension is for "features" - there is only one here, since images are
    # grayscale -- it would be 3 for an RGB image, 4 for RGBA, etc.
    with tf.name_scope('reshape'):
        x_image = tf.reshape(x, [-1, 28, 28, 1])

    # First convolutional layer - maps one grayscale image to 32 feature maps.
    with tf.name_scope('conv1'):
        W_conv1 = weight_variable([5, 5, 1, 32])
        b_conv1 = bias_variable([32])
        h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)

    # Pooling layer - downsamples by 2X.
    with tf.name_scope('pool1'):
        h_pool1 = max_pool_2x2(h_conv1)

    # Second convolutional layer -- maps 32 feature maps to 64.
    with tf.name_scope('conv2'):
        W_conv2 = weight_variable([5, 5, 32, 64])
        b_conv2 = bias_variable([64])
```

```
        h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)

    # Second pooling layer.
    with tf.name_scope('pool2'):
        h_pool2 = max_pool_2x2(h_conv2)

    # Fully connected layer 1 -- after 2 round of downsampling, our 28x28 image
    # is down to 7x7x64 feature maps -- maps this to 1024 features.
    with tf.name_scope('fc1'):
        W_fc1 = weight_variable([7 * 7 * 64, 1024])
        b_fc1 = bias_variable([1024])

        h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
        h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

    # Dropout - controls the complexity of the model, prevents co-adaptation of
    # features.
    with tf.name_scope('dropout'):
        keep_prob = tf.placeholder(tf.float32)
        h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

    # Map the 1024 features to 10 classes, one for each digit
    with tf.name_scope('fc2'):
        W_fc2 = weight_variable([1024, 10])
        b_fc2 = bias_variable([10])

        y_conv = tf.matmul(h_fc1_drop, W_fc2) + b_fc2
    return y_conv, keep_prob


def conv2d(x, W):
    """conv2d returns a 2d convolution layer with full stride."""
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')


def max_pool_2x2(x):
    """max_pool_2x2 downsamples a feature map by 2X."""
```

```
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
                          strides=[1, 2, 2, 1], padding='SAME')


def weight_variable(shape):
    """weight_variable generates a weight variable of a given shape."""
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)


def bias_variable(shape):
    """bias_variable generates a bias variable of a given shape."""
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)


def predict(result):
    # 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('adam_optimizer'):
        train_step = tf.train.AdamOptimizer(1e-4).minimize(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)
    accuracy = tf.reduce_mean(correct_prediction)
```

```
    saver = tf.train.Saver()  # Define the saver
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        model_file = tf.train.latest_checkpoint('../mnist-app/model/')
        saver.restore(sess, model_file)
        prediction = tf.argmax(y_conv, 1)
        predint = prediction.eval(
            feed_dict={x: [result], keep_prob: 1.0}, session=sess)
        prediction1 = predint[0]

        print('recognized result:')
        print(prediction1)

    return prediction1
```

2.4 Build the Cassandra Keyspace

Here we use the codes provided to connect the docker container and then create the Cassandra keyspace inside the container. The code is as follows:

```
from cassandra import ConsistencyLevel
from cassandra.query import SimpleStatement
from cassandra.cluster import Cluster
from cassandra.policies import RoundRobinPolicy

import logging

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

KEYSPACE = "MNIST1"


def createKeySpace():
    cluster = Cluster(contact_points=[
```

```
                '127.0.0.1'], load_balancing_policy=RoundRobinPolicy(), port=9042)
    session = cluster.connect()
    log.info("Creating keyspace...")
    try:
        session.execute("""
                CREATE KEYSPACE %s
                WITH replication = { 'class': 'SimpleStrategy', 'replication_factor': '1' }
                """ % KEYSPACE)
        log.info("setting keyspace...")
        session.set_keyspace(KEYSPACE)

        log.info("creating table...")
        session.execute("""
          CREATE TABLE history (
              IP_Address text,
              access_time timestamp,
              image_path text,
              mnist_result text,
              PRIMARY KEY (IP_Address, time)
          )
          """)
    except Exception as e:
        log.error("Unable to create keyspace")
        log.error(e)


def insertData(ip_addr, access_time, image_path, mnist_result):
    cluster = Cluster(contact_points=['127.0.0.1'],
                      load_balancing_policy=None, port=9042,)
    session = cluster.connect()
    log.info("Inserting data...")
    try:
        session.execute("""
            INSERT INTO mnist.History (IP_Address, access_time, image_path, mnist_result)
            VALUES(%s, %s, %s, %s);
            """,
                        (ip_addr, access_time, image_path, mnist_result)
```

```
                )
except Exception as e:
    log.error("Unable to insert data")
    log.error(e)
```

As shown in the code, the port we are using now is 9042. Thus, we can use the following method to connect the container:

```
docker run --name fzhang-cassandra -p 9042:9042 -d cassandra:latest
```

If run the command `docker ps` after running the command above, we can see the running container, especially that the port has been successfully mapped.

After running the code, we have a Cassandra table created, with user's IP address, access time, the path of image, and the predicted number. After this program being built, if we run the "upload_pictures.py" directly, there will be upload file name, upload time, and predicted number shown on the page.

3. Conclusion

During the research process, I've learned a lot about big data technologies. When the program began, I knew nothing about how to build a website and even how to use Unix command. But now, I knew about Flask, tensorFlow, container technologies like Docker and Cassandra. Even though the knowledge I knew for now is scarce compared to the overall information in the field of big data analysis and data mining, I believe that interest is the biggest motivation, and I will continue learning to apply what I got efficiently in practice.