Fast Food logo recognition using Python, OpenCV & Tensorflow

Elliot Bensabat & Maxwell Donkoh

Northeastern University – CS 4100
bensabat.e@husky.neu.edu, donkoh.m@husky.neu.edu

Abstract

The problem we tried to solve is to allow a user to interact with a fast-food nearby by taking a photo of the company's logo. We approached this by training a classification model using a Convolutional Neural Network with a dataset of multiple logos and their associated restaurant labels, for 5 different companies: Dunkin Donuts, McDonald's, Starbucks, Au Bon Pain, 5 Guys. Together with location data, this classifier predicts which particular restaurant a person wants to engage with, using the Google API. This prediction will then be used to open up the Yelp page of that particular restaurant, using the Yelp API. We trained our model on 5k+ images per class (5 classes) and ended up with a ~92% accuracy on our test test.

Introduction

As college students, sometimes we just want to quickly figure out the menu of a specific restaurant or food spot we are not necessarily aware off, without having to stand in a long line or actually go in to speak with a waiter/waitress. We realized however that most of these places have Yelp Profile pages. The only thing separating these profile page reviews from students like us was quick knowledge of these places. We felt we could use a visual representation of common fast food places around us, as part of a test app to determine if students will actually utilize an app that shows a menu upon taking a picture. To do that, we decided to use deep learning to train a model that can predict a fast food location and open up its Yelp page on the fly, after being trained with a specific group of fast food places. More specifically, we decided to use a convolutional neural network because we can use that to effectively input 3D points to get a 3-D output, that efficiently represents the pixel points in the picture. We used google's tensorflow because it has an active community and we can have a layered model easily built from just a few lines of code.

Project Description

Building the dataset

The first challenge that we encountered doing that project was about getting the right dataset to be able to train our model and get accurate results. Deep Neural Networks are very popular because of their accuracy capabilities, but require much more data and computing power that most algorithms.

Focusing on 5 classes, we knew we were going to need thousands of training samples to get results we would be happy with. These training samples needed to be images where each company's logo would be visible. Our first thought was to use Google's Search API but we quickly encountered our first road block. Google allows users to request 100 queries a day per user for free. Let's say that we would need 5k images per class, well that would take 25 days per class (we are 2 users) and this was definitely impossible to do.

After few days of research on other APIs, publicly available datasets and web scraping, we realized that we would have to create our own dataset using OpenCV. OpenCV provides a function called opencv_createsamples that allows the creation of 2k samples from a single image. Logos are very 'static' images, in the sense that for a specific brand, its logo is not going to change much from image to image. A contrary example would be cars, where cars are going to have very different shapes depending on the brands, the style (SUV, sports car, truck, etc.) and the specific shape of the car.

How opencv_createsamples work is that OpenCV provides a collection of 2k 'negative' images. Negative images are random images which don't contain the image part that you are trying to classify. If you were trying to detect cars versus not cars, having a dataset of cars ready, you could think of negative images as the one that you would be labeled as 0 (not cars) to train your algorithm. In the createsamples case, these negative images are used as background to the original image. The source image is going to be put on every background, is going to be rotated on the 3 axis randomly within the boundary specified by -max?angle and colors are going to be distorted if necessary.

The workflow to create the dataset was the following:
-Have one folder per class
-Have the original images of each class in its folder
-Resize every image in every folder to the desired size
-Create the samples with opencv_createsamples for each image in each folder

Our resized target size was 100x100 pixels and we used the Nearest Neighbor algorithm to resize the images through the library resizeimage. For every image in every folder, the following code was applied:

```
im1 = Image.open(image_path)
im1 = im1.convert('RGB')
#use nearest neighbour to resize
im2=im1.resize((RESIZED_WIDTH,
RESIZED_HEIGHT), Image.NEAREST)
ext = ".jpg"
im2.save(image_path)
```

Once every image was at the right size, we had another script to apply the opencv_createsamples command to every image for every company. We decided to create 1500 images for every original image which would give us a dataset of 1500*5 = 7500 images per class. This was done through the following line of code (-bg accesses a .txt file which redirects to the backgrounds, also called negative images & -info specifies where the new samples are saved):

```
os.system('opencv_createsamples -img ' + image_path + ' -
bg new_bg.txt -info ' + comp + '/info.lst -maxxangle 0.5 -
maxyangle 0.5 -maxzangle 0.5 -num 1500')
```

That resizing and creating samples done, the dataset is ready. We had a collection of 7500 images per class, so 37500 images total, that would be split up later in 70% training set - 20% validation set - 10% test set.

Convolutional Neural Network

Convolutional Neural Network (CNN or ConvNet) is a class of deep, feed-forward artificial neural networks optimized to be applied to images. A CNN takes an input layer, apply multiple hidden layers and return an output layer. The hidden layers are usually convolutional layers, pooling layers, fully connected layers and normalization layers.

Convolutional layers reduce the dimensionality of the network and reduces the number of weights the Network has to learn. In our case where each image is 100x100 pixels, a fully connected layer would need 100x100 = 10,000 weights for each neuron in the following layer. This is definitely not scalable and parameters like filter and stride allows the network to capture all the important elements of an image with much less weights. The first thing we usually determine for a convolutional layer is the filter size also called patch size. We used a patch size of 5 which mean that we went from a dimension of (100,100,1), with 1 because the image is in black & white (vs 3 for colors RGB), so 10k weights for each neuron to (5,5,1) so 25 weights per neuron. In addition to the filter, we use what's called depth. Depth determines the number of filters we would like to use in the input. Then, another important parameter is the stride which defines by how much pixels we're going to move the filter along the image. We decided to use a stride of 2. We usually follow a convolutional layer with an activation layer using the Sigmoid function, Tahn or ReLu. We decided to pick ReLu which has 2 main advantages: sparsity and reduced likelihood of vanishing gradient. ReLy is defined as:
$f(z) = \max(0,z)$

The architecture of our model is the following:
-convolutional
-ReLu
-convolutional
-ReLu
-hidden
-dropout
-output

In Tensorflow, the architecture of the models is defined as the following:

```
# Variables.
layer1_weights =
tf.Variable(tf.truncated_normal([patch_size, patch_size,
num_channels, depth], stddev=0.1))
layer1_biases = tf.Variable(tf.zeros([depth]))

layer2_weights =
tf.Variable(tf.truncated_normal([patch_size, patch_size,
depth, depth], stddev=0.1))
layer2_biases = tf.Variable(tf.constant(1.0, shape=[depth]))

layer3_weights =
tf.Variable(tf.truncated_normal([image_size // 4 *
image_size // 4 * depth, num_hidden], stddev=0.1))
layer3_biases = tf.Variable(tf.constant(1.0,
shape=[num_hidden]))

layer4_weights =
tf.Variable(tf.truncated_normal([num_hidden,
num_labels], stddev=0.1))
layer4_biases = tf.Variable(tf.constant(1.0,
shape=[num_labels]))

# Model.
def model(data, keep_prob):
    conv = tf.nn.conv2d(data, layer1_weights, [1, 2, 2, 1],
padding='SAME')
```

```
    hidden = tf.nn.relu(conv + layer1_biases)

    conv = tf.nn.conv2d(hidden, layer2_weights, [1, 2, 2,
1], padding='SAME')
    hidden = tf.nn.relu(conv + layer2_biases)

    shape = hidden.get_shape().as_list()
    reshape = tf.reshape(hidden, [shape[0], shape[1] *
shape[2] * shape[3]])
    hidden = tf.nn.relu(tf.matmul(reshape,
layer3_weights) + layer3_biases)

    drop_out = tf.nn.dropout(hidden, keep_prob)
    # output layer with linear activation
    out_layer = tf.matmul(drop_out, layer4_weights) +
layer4_biases
    return out_layer
```

Loss function & Optimizer

To evaluate our model at every step in order to modify the weights, we need a loss function which is going to define how well our model is doing & we need an optimizer which will be updating the weights based on that error function.

For the error function of our model we used the mean of the softmax cross entropy between logits, which is the label predicted based on the probability and the labels we are trying to predict. As Tensorflow defines it, softmax cross entropy 'Measures the probability error in discrete classification tasks in which the classes are mutually exclusive'. This loss function is defined in Tensorflow as the following:

```
logits = model(tf_train_dataset, keep_prob)
loss =
tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(l
abels=tf_train_labels, logits=logits))
```

In terms of optimizer we used an the Adam optimizer algorithm with a learning rate of 1e-4. Adam Optimizer is an extension of Stochastic Gradient Descent. Adam combines the advantages of 2 extensions of Stochastic Gradient Descent which are AdaGrad and RMSProp. Specifically, the algorithm calculates an exponential moving average of the gradient and the squared gradient, and the parameters beta1 and beta2 control the decay rates of these moving averages. This optimizer is defined in Tensorflow as the following:

```
optimizer = tf.train.AdamOptimizer(1e-4).minimize(loss)
```

Model Training

For the training, we decided to use batches of 32 images and train during 300k steps. Knowing that we have 0.7*7500*5 = 26,250 training samples, we need 26,250/32 = 820 steps to go through all the samples once, which is defined as one epoch. Thus, 300k steps corresponds to 365 epochs. The training phase was made in Tensorflow using the following code:

```
num_steps = 300000
dropout_rate = 0.8

with tf.Session(graph=graph) as session:
  tf.global_variables_initializer().run()
  print('Initialized')

  for step in range(num_steps):
    offset = (step * batch_size) % (train_labels.shape[0] -
batch_size)
    batch_data = train_dataset[offset:(offset + batch_size), :,
:, :]
    batch_labels = train_labels[offset:(offset + batch_size),
:]
    feed_dict = {tf_train_dataset : batch_data,
tf_train_labels : batch_labels, keep_prob : dropout_rate}
    _, l, predictions = session.run(
      [optimizer, loss, train_prediction], feed_dict=feed_dict)
    if (step % 3000 == 0):
      print('Minibatch loss at step %d: %f' % (step, l))
      acc_minibatch = accuracy(predictions, batch_labels)
      acc_validation = accuracy(valid_prediction.eval(),
valid_labels)
      print('Minibatch accuracy: %.1f%%' %
acc_minibatch)
      print('Validation accuracy: %.1f%%' %
acc_validation)
  saver.save(session, './logo-model-new.ckpt')
  print('Test accuracy: %.1f%%' %
accuracy(test_prediction.eval(), test_labels))
```

Predictions

For the predictions, we need to be able to retrieve the learnt weights from the training phase. This is done through a tf.train.Saver() which saves the session during training with saver.save(session, path). This saver is restored in the prediction phase with saver.restore(session, path).

To be able to predict the class of an inputted image, we need to resize that image, reshape it to fit the input layer and feed it to the model to get the probability of that image being in each class. The highest probability is

the one that will be assigned the predicted class. This is done through the following code:

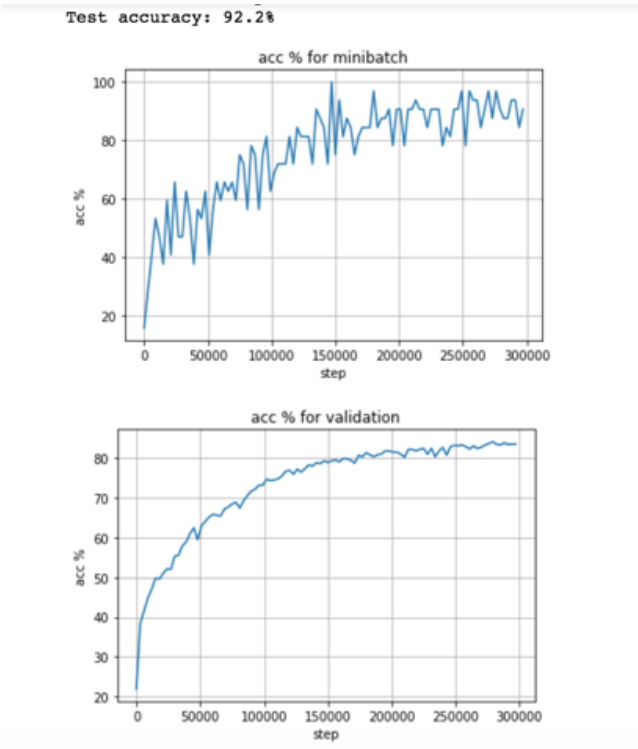
```
RESIZED_WIDTH = 100
RESIZED_HEIGHT = 100
pixel_depth = 255.0  # Number of levels per pixel.

def get_proba(path):
    im1 = Image.open(path)
    im1 = im1.convert('L')
    #use nearest neighbour to resize
    im2 = im1.resize((RESIZED_WIDTH,
RESIZED_HEIGHT), Image.NEAREST)
    ext = ".jpg"
    im2.save(path)
    im2 = (imageio.imread(path).astype(float) -
                pixel_depth / 2) / pixel_depth
    x = im2.reshape(
        (-1, image_size, image_size,
num_channels)).astype(np.float32)
    with tf.Session(graph=graph) as session:
        saver.restore(session, "./logo-model.ckpt")
        tf_predict_input = tf.constant(x)
        predict_input = tf.nn.softmax(model(tf_predict_input,
1.0))
        return(predict_input.eval())

logos = ['Dunkin Donuts', 'McDonald\'s', 'Starbucks', '5
Guys', 'Au Bon Pain']

def get_pred(path):
    p = get_proba(path)
    while np.isnan(p).all():
        p = get_proba(path)
    i = np.argmax(p, 1)[0]
    print(logos[i])
```

Returning the fast food page based on user's location

We used Google maps geolocation API to turn a latitude/longitude pair into a specific pinpoint on the map. To accomodate for certain inconsistencies including lat/longs for high high building areas, we expand the pinpoint to an incremental 0.1 mile radius. As that happens, we use the prediction result together with a lat/long pair to start off a job that queries Yelp's REST API. If we get a non-empty result list, we validate the name string in the first result with the prediction string and then if there is a substring that is similar up to about 90%, we return that result. The user can then interact with the local spot through reservations, reading reviews, among others.

Experiments

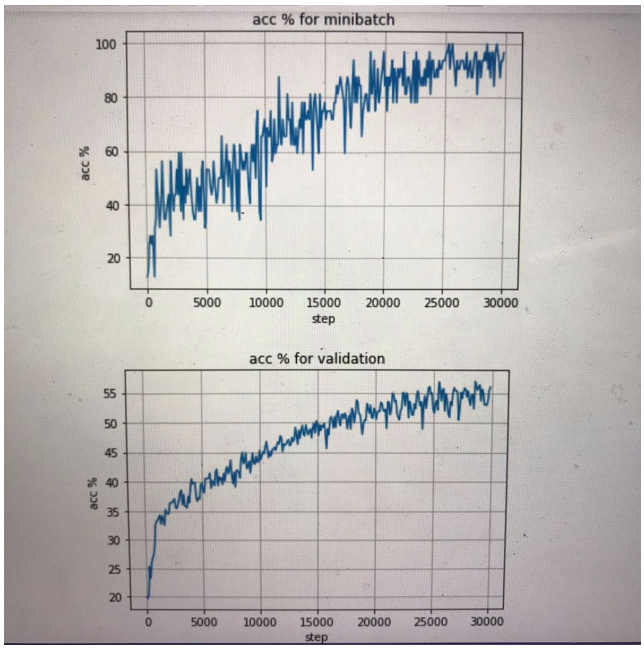
The training phase of 300k steps was run locally and took ~8 hours to run. We finished with a accuracy on our test set of 92.2%.



But before getting to successful results, we ran into multiple problems. The first one was to find the right architecture for the Neural Network: How many Convolutional Layer do we need? What activation function to use? How big should be the fully connected layers?
We researched multiple papers about CNN, including papers from Yann LeCun, inventor of the CNN, and followed multiple tutorials to have a better idea of what kind of architecture we would need.

The architecture however, is not enough. We had to run multiple experiments with the parameters from filter size, to stride, to padding, to batch size, etc.

We finally got to an 'okay' model which got to 95% accuracy on batches but only 50% on validation set and that would stop learning after about 30k steps.



We were clearly overfitting. The best way to control that overfitting was to add a dropout layer before the output layer that takes an output rate and drops neurons based on that rate. More technically, at each training stage, individual nodes are either dropped out of the net with probability *1-p* or kept with probability *p*, so that a reduced network is left; incoming and outgoing edges to a dropped-out node are also removed. This allowed the model to go from our original ~50% to our final ~90% accuracy.

Conclusion

During this project, we obviously learnt a lot about Neural Network applications and more specifically Convolutional Neural Network, its applications and its benefits. But we also used tools that we've never used in depth before such as OpenCV and TensorFlow.

OpenCV is a great library for computer vision. We only used it to create our dataset but we could also have used it to train our model. However, OpenCV doesn't offer much flexibility on how to train the model and the training part is a black box.

Therefore we decided to move on with Tensorflow which offer a lot of flexibility, maybe too much. Tensorflow wasn't the easiest to use and no wonder why Keras is so popular. We would use Tensorflow for research but for known applications of Neural Network, we definitely thought that Tensorflow was overcomplicated.

The 92% accuracy that we got to was very satisfying. Next steps could be to add more classes (fast foods) and see if we can keep a similar accuracy. We would also like to compare the results based on black & white vs color and based on different image sizes.

References

http://ufldl.stanford.edu/tutorial/supervised/ConvolutionalNeuralNetwork/

http://deeplearning.net/tutorial/lenet.html
https://www.tensorflow.org/tutorials/deep_cnn

https://keras.io/layers/convolutional/
https://www.tandfonline.com/doi/pdf/10.1080/01431160600746456

http://www.jars1974.net/pdf/12_Chapter11.pdf
https://www.technologyreview.com/s/513696/deep-learning/

http://neuralnetworksanddeeplearning.com/chap6.html
http://cs231n.github.io/convolutional-networks/

http://yann.lecun.com/exdb/publis/index.html

https://en.wikipedia.org/wiki/Convolutional_neural_network

https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5

https://stackoverflow.com

http://cs231n.github.io/convolutional-networks/