

Springboard Data Science Capstone Project II

BEHOLDER

Attractiveness in Human Faces

Akiro Semtei

July, 2020

Table of Contents

1. Introduction	3
2. Data Acquisition and Cleaning	4
3. Data Manipulation	5
3.1 Merging	5
3.2 TFRecord Dataset	6
3.3 Google Cloud Storage	8
4. Experiment	9
5. Results	12
5.1 Evaluation	12
5.2 Discussion	13

1. Introduction

Appraisal of facial attractiveness seems natural to humans, but an actual definition of facial beauty remains elusive. Lately, human attractiveness prediction projects, which intends to achieve an automatic, human-consistent, facial attraction evaluation through a computational model, are becoming increasingly popular in pattern recognition and machine learning groups. It has potential use in facial makeup, image recapture, esthetic surgery or face embellishment.

From the computational point of view, attractiveness prediction remains a challenging problem. Numerous data-driven models were implemented in this experiment to tackle this issue. One line of work follows the classic pattern recognition process which uses the combination of hand-crafted features and shallow predictors.

Another line of research is driven by the renaissance of neural networks, in particular the state-of-the-art deep learning convolutional neural networks. The deep learning model's hierarchical structure allows for the creation of an end-to-end framework that automatically learns both the representation and the predictor of facial attractiveness from the data simultaneously.

This paper's principal contributions can be summarized as follows:

- 1) Dataset: We propose a new large-scale benchmark dataset BEHOLDER that has a total of 7,816 front facing faces at neutral angles either smiling or not and the accompanying attractiveness attributes.
- 2) Model Results: We statistically evaluate the samples and attributes of the BEHOLDER dataset and offer a fitted model with 0.66 loss that can accurately identify human beauty and can be used to start other projects built upon the results of this model. For example a generative adversarial network.

2. Data Acquisition and Cleaning

The data set was acquired from open and private sources. The first set was retrieved from the SCUT-FBP5500 dataset. Dubbed 'SCUT-5500' or 'SCUT' includes face attractiveness ratings of 5,500 young men and women of Asian and Caucasian ethnicity. including 2000 Asian females, 2000 Asian males, 750 Caucasian females and 750 Caucasian males. All the images are labelled with beauty scores ranging from [1, 5] by a total of 60 volunteers.

The second dataset is the London Face Research Lab dataset that's publicly available. All individuals gave signed consent for their images to be "used in lab-based and web-based studies in their original or altered forms and to illustrate research (e.g., in scientific journals, news media or presentations)." Images were taken in London, UK, in April 2012.

Attractiveness ratings, on a [1-7] scale from "much less attractiveness than average" to "much more attractive than average for 2513 people (ages 17-90).

The third set is the 10k Adult Faces Database. This database contains 10,168 natural face photographs and several measures for 2,222 of the faces, including attractiveness amongst others. The face photographs are JPEGs with 72 pixels/in resolution and 256-pixel height-width. The attribute data are stored in Excel files.

Reference: Bainbridge, W.A., Isola, P., & Oliva, A. (2013). The intrinsic memorability of face images. *Journal of Experimental Psychology: General*. *Journal of Experimental Psychology: General*, 142(4), 1323-1334

SCUT

The first dataset contains images labelled with beauty scores ranging from [1, 5] by totally 60 volunteers, and 86 facial landmarks are also located to the significant facial components of

each images. The data is accompanied by a .txt file that has all the file names and the attractiveness attributes with a [space] as a separator. We proceed to load the data into csv format and this set is ready for the next step.

LONDON Face Research Lab

The second dataset has images of 102 adult faces 1350x1350 pixels in full colour. Self-reported age, gender and ethnicity are included in the file `london_faces_info.csv`. Attractiveness ratings are included in the file `london_faces_ratings.csv`. This dataset is naturally ready for the next step.

10K Faces

The final dataset is This database contains 10,168 natural face photographs and several measures for 2,222 of the faces, including memorability scores, computer vision and psychology attributes, and landmark point annotations. The face photographs are JPEGs with 72 pixels/in resolution and 256-pixel height. The attribute data are stored in excel files that are easily converted to csv.

Finally, we have 3 sets of images with 3 csv files linking the image names to its attributes. We will continue work on the combined set.

3. Data Manipulation

3.1 Merging

In order to run our experiment efficiently we want to join the SCUT, LONDON, and 10K databases into a single data frame, `train_label_df`, that has all the randomly sorted image names in column 1 and the attributes normalised to a scale of [1-9].

3.2 TFRecord Dataset

In order to run the experiment on cloud TPUs we need to format the data in such a way that the stream of information is quick enough to keep the TPUs working at all times. The initial step is the creation of the 2 tensors what we are going to experiment on, the input and output of the neural network. The first tensor, or input tensor, contains the bitwise information of all the pictures of our combined dataset. The second tensor, or output tensor, contains the float32 values for the aggregated attractiveness scores for each picture.

We start by creating a dictionary in which the keys are the image filenames and the values are the attractiveness attribute.

```
dic = {}  
for i in range(len(train_label_df)):  
    dic[train_label_df['id'][i]] = train_label_df['score'][i]
```

Figure 1 - Dataset Dictionary

We then utilise 3 helper functions that allow us to decode all the information into bytecode -for the images' pixel information; int32 -for the height/width/depth of the image; and float32 for the target values.

```
def _bytes_feature(value):  
    """Returns a bytes_list from a string / byte."""  
    if isinstance(value, type(tf.constant(0))):  
        value = value.numpy() # BytesList won't unpack a string from an EagerTensor.  
    return tf.train.Feature(bytes_list=tf.train.BytesList(value=[value]))  
  
def _float_feature(value):  
    """Returns a float_list from a float / double."""  
    return tf.train.Feature(float_list=tf.train.FloatList(value=[value]))  
  
def _int64_feature(value):  
    """Returns an int64_list from a bool / enum / int / uint."""  
    return tf.train.Feature(int64_list=tf.train.Int64List(value=[value]))
```

Figure 2 - Helper Functions

Then we build a function that will decode a single image and use the helper functions to write all the information into a `tf.example`, which is a single record in a TFRecord dataset.

```
def image_example(image_string, label):
    image_shape = tf.image.decode_jpeg(image_string).shape

    feature = {
        'height': _int64_feature(image_shape[0]),
        'width': _int64_feature(image_shape[1]),
        'depth': _int64_feature(image_shape[2]),
        'id': _float_feature(label),
        'image_raw': _bytes_feature(image_string),
    }

    return tf.train.Example(features=tf.train.Features(feature=feature))
```

Figure 3 - TF example builder

The final step is to pass the information by a writer or TFRecord generator, we do that via the following code shown in Figure 4

```
with tf.io.TFRecordWriter(record_file) as writer:
    for filename, label in dic.items():
        image_string = open(GCS_PATTERN + filename, 'rb').read()
        tf_example = image_example(image_string, label)
        writer.write(tf_example.SerializeToString())
```

Figure 4 - TFRecord dataset generator

We can see the last example written into the dataset via simply printing the last `tf_example` of the iteration. This will give us an idea on the internal structure of a TFRecord file.

```
tf_example
features {
  feature {
    key: "depth"
    value {
      int64_list {
        value: 3
      }
    }
  }
  feature {
    key: "height"
    value {
      int64_list {
        value: 350
      }
    }
  }
  feature {
    key: "id"
    value {
      float_list {
        value: 5.0399999961853027
      }
    }
  }
  feature {
    key: "image_raw"
    value {
      bytes_list {
        value: "\377\330\377\340\000\020JFIF\000\001\001\000\000\001\000
\001\000\000\377\333\000C\000\002\001\001\001\001\001\002\001\001\001\002
\002\002\002\002\004\003\002\002\002\002\005\004\004\003\004\006\005\006"
```

Figure 5 - TF.example

We see that the data is stored in a JSON-like nested nature with the last feature, image_raw, being a long list of raw bytecode that contains the image information.

3.3 Google Cloud Storage

Google Cloud TPUs do not work over local reads of TFRecord files, although there is an intention to add the feature in the future. The dataset needs to be uploaded to a google bucket inside the google cloud platform.



Name	Size	Type	Storage class	Last modified	Public access ?
 behold.tfrecords	232.46 MB	application/octet-stream	Standard	6/30/20, 11:49:52 PM UTC+1	Not public
 tensorboard/	—	Folder	—	—	Not public

Figure 6 - Google Cloud Bucket

We can see the beholder.tfrecords sitting in the bucket in Figure 6. We also can see another folder for the tensorboard files that we will create while running the experiment.

4. Experiment

We are going to apply a transfer learning technique to build upon one of the best performing CNN architectures at the time of writing. NASNet-Large with the famous 'imagenet' weights will be our foundation.

To proceed with such a large network, we require massive resources that are not available locally, we need to turn to Google Colab. A free service that gives user access to a Jupyter-like interface to write python-based notebooks, but much more important, and appreciated, is the free access to Titan class NVIDIA GPUs and TPUs. This is the sort of resources that can take on a large NN without extremely large training times and very small batches.

The notebook itself can be found at

<https://colab.research.google.com/drive/1W06JDd2IPjFYpnFJ7zxXWWOwYI09ltwv?usp=sharing>

and it offers a step-by-step guide of the code required to build the Neural Network and run it over 150 epochs. Let's go over the most important cells in this notebook.

```

resolver = tf.distribute.cluster_resolver.TPUClusterResolver(tpu='grpc://' + os.environ['COLAB_TPU_ADDR'])
tf.config.experimental_connect_to_cluster(resolver)
# This is the TPU initialization code that has to be at the beginning.
tf.tpu.experimental.initialize_tpu_system(resolver)
print("All devices: ", tf.config.list_logical_devices('TPU'))

INFO:tensorflow:Initializing the TPU system: grpc://10.99.206.74:8470
INFO:tensorflow:Initializing the TPU system: grpc://10.99.206.74:8470
INFO:tensorflow:Clearing out eager caches
INFO:tensorflow:Clearing out eager caches
INFO:tensorflow:Finished initializing TPU system.
INFO:tensorflow:Finished initializing TPU system.
All devices: [LogicalDevice(name='/job:worker/replica:0/task:0/device:TPU:5', device_type='TPU'), LogicalDevice(name='/job:worker/replica:0/task:0/device:TPU:4', device_type='TPU'), LogicalDevice(name='/job:worker/replica:0/task:0/device:TPU:3', device_type='TPU'), LogicalDevice(name='/job:worker/replica:0/task:0/device:TPU:2', device_type='TPU'), LogicalDevice(name='/job:worker/replica:0/task:0/device:TPU:1', device_type='TPU'), LogicalDevice(name='/job:worker/replica:0/task:0/device:TPU:0', device_type='TPU')]

strategy = tf.distribute.experimental.TPUStrategy(resolver)

```

Figure 7 - TPU setting up

The first cells after we import all necessary libraries are standard to enable the cloud TPUs in Google Colab. We have chosen to go with the experimental TPUStrategy solver. This is the method by which the data will be separated and streamed to the 8 available TPUs. The strategy then assembles the data and presents the results.

We proceed to mount both our Google Drive and our Google Cloud Bucket to store the results and read the dataset respectively.

```

▶ from google.colab import drive
  drive.mount('/content/drive/')

[ ] from google.colab import auth
    auth.authenticate_user()

[ ] !gcloud config set project {project_id}

[ ] Updated property [core/project].

```

Figure 8 - Google APIs

We load tensorboard and set the initial parameters.

```
[ ] %load_ext tensorboard

[ ] EPOCHS = 150
    CYCLE = 150
    BATCH_SIZE = 64
    IMAGE_SIZE = [331, 331]
    DATASET_SIZE = (2735+5081)
    start_lr = 0.00001
    min_lr = 0.00001
    max_lr = (0.00005 * 8)
    rampup_epochs = 4
    sustain_epochs = 1
    exp_decay = 0.85

    gcs_pattern = 'gs://beholder/ behold.tfrecords'

    train_size = int(0.7 * DATASET_SIZE)
    val_size = int(0.15 * DATASET_SIZE)
    test_size = int(0.15 * DATASET_SIZE)

    TRAIN_STEPS = (train_size // BATCH_SIZE)
    VAL_STEPS = (val_size // BATCH_SIZE)
```

Figure 9 - Initial Parameters

We build our model applying a MaxPooling layer to the final NASNET-Large layer without the 'top'. This means that when we pull the pretrained network we trim the aggregated layers that categorize images and instead return un-pooled and heavily convoluted tensors. The MaxPooling aggregates our results and then we add a few hidden dense layers to approach the final output layer with a single float32 neuron.

```
[ ] with strategy.scope(): # creating the model in the TPUStrategy scope means we will train the model on the TPU
    pretrained_model = NASNetLarge(
        weights='imagenet',
        include_top=False,
        input_shape=(331, 331, 3))
    model = tf.keras.Sequential([
        pretrained_model,
        GlobalMaxPooling2D(),
        Dense(1000),
        BatchNormalization(),
        Activation('relu'),
        Dropout(0.5),
        Dense(250),
        BatchNormalization(),
        Activation('relu'),
        Dropout(0.5),
        Dense(1, activation='linear')])
    model.compile(optimizer='adam', loss='mean_squared_error', metrics=['mean_absolute_error', 'mean_absolute_percent

[ ] Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/nasnet/NASNet-large-no-top.h5
343613440/343610240 [=====] - 3s 0us/step
```

Figure 10 - Model Architecture

Finally, we run the model of our experiment for 150 epochs.

```
start_time = time.time()
history = model.fit(train_dataset, validation_data=val_dataset,
                    steps_per_epoch=TRAIN_STEPS, epochs=EPOCHS,
                    validation_steps=VAL_STEPS,
                    callbacks=[lr_callback,checkpoint,tensorboard], verbose=1)

final_accuracy = history.history["val_loss"][-5:]
print("FINAL ACCURACY MEAN-5: ", np.mean(final_accuracy))
print("TRAINING TIME: ", time.time() - start_time, " sec")

WARNING:tensorflow:Method (on_train_batch_end) is slow compared to the batch update (0.691125). Check your callback
85/85 [=====] - 304s 4s/step - loss: 0.6265 - mean_absolute_error: 0.6263 - mean_absolute_
Epoch 142/150
1/85 [.....] - ETA: 0s - loss: 0.6987 - mean_absolute_error: 0.7059 - mean_absolute_perce
WARNING:tensorflow:Method (on_train_batch_end) is slow compared to the batch update (1.344198). Check your callback
2/85 [.....] - ETA: 22s - loss: 0.7322 - mean_absolute_error: 0.6997 - mean_absolute_perc
WARNING:tensorflow:Method (on_train_batch_end) is slow compared to the batch update (0.681743). Check your callback
85/85 [=====] - 85s 1s/step - loss: 0.6324 - mean_absolute_error: 0.6258 - mean_absolute_p
Epoch 143/150
1/85 [.....] - ETA: 0s - loss: 0.6360 - mean_absolute_error: 0.6483 - mean_absolute_perce
WARNING:tensorflow:Method (on_train_batch_end) is slow compared to the batch update (1.204577). Check your callback
2/85 [.....] - ETA: 23s - loss: 0.6589 - mean_absolute_error: 0.6655 - mean_absolute_perc
WARNING:tensorflow:Method (on_train_batch_end) is slow compared to the batch update (0.613298). Check your callback
85/85 [=====] - 83s 971ms/step - loss: 0.6489 - mean_absolute_error: 0.6314 - mean_absolut
Epoch 144/150
1/85 [.....] - ETA: 0s - loss: 0.8294 - mean absolute error: 0.7362 - mean absolute perce
```

Figure 11 - Model Running

5. Results

5.1 Evaluation

We evaluate our model against a never previously seen dataset we split out from the main TFRecord and observe an MSE of 0.66 which is a strong value considering the output variable was not standardised and instead ranges from [1,9].

```
[ ] model.evaluate(test_dataset)

19/19 [=====] - 20s 1s/step - loss: 0.6661 - mean_absolute_error: 0.6508 - mean_absolute_percentage_error: 12.1480
[0.6661132574081421, 0.6508312225341797, 12.148016929626465]
```

Figure 12 - Model Evaluation

5.2 Discussion

In this experiment we implemented a new broad benchmark dataset called BEHOLDER to achieve multiparadigm analysis of facial attractiveness. The dataset has a total of over 8,000 frontal face images with varied characteristics (sex, race, ages) and a broadly generalized attractiveness score averaged over at least 15 ratings from other humans. Benchmark analysis was performed for the beauty scores, and the data reveals the efficacy of CNN networks for this type of problem. This experiment results in a model that can be run against any human face, male or female, and will return an accurate representation of its attractiveness. However, the limitations are a posed face, either smiling or resting, a simple background, and a straight neutral angle facing the camera head-on. These are limitations that can be worked around by enriching the images with random rotations, saturations, crops and other manipulations that could be a continuation of the work presented herein.

Because the BEHOLDER dataset generation is designed for multi-paradigm implementations, our work can be tailored to various models for specific functions, such as appearance-based or shape-based facial image classification / regression / ranking. The BEHOLDER dataset was tested using various combinations of features and predictors and deep learning models, where the tests show the precision of the data collection. We only presented the best results achieved across the experimentation.