Name Prediction based on a person's Image using a Convolutional Neural Network

Moein Roghani

April 20 2023

Pages: 11

1 Introduction

In this project, our goal is to create a model that can predict a person's name based on their image using a Convolutional Neural Network (CNN). And further in this report we will thoroughly go through the followings:

- Data preprocessing: We will describe the process of cleaning and organizing the dataset, converting images to a PNG format, and resizing them to a same size.
- Model Implementation: We will go over the structure of the CNN, including the different layers and their functions.
- Training process: We will explain our training model.

- Evaluation: We will discuss how the model's performance is measured and the results achieved.
- Prediction: We will show how the trained model can be used to predict a person's name from their image.

2 Data Preprocessing

The first step in our project is to preprocess the dataset. The dataset is organized into multiple folders, with each folder representing a name, and the images inside each folder corresponding to the data points for that name. We have gathered in total around 89,700 data of peoples images, however the labeling and the data is completely different in each data set and some of them have lastnames as well, so I needed to arrange all into a single clean data set. I did multiple bash scripting to handle this problem. So basically our training set is images that are each inside their labeled classification folder. So we have bunch of folders that are names (our labels) and inside each folder e have the data points representing that class. And finally converted each image to PNG file and processed them.

lfw: The data set contains more than 13,000 images of faces (5749 people) collected from the web. Each face has been labeled with the name of the person pictured: http://vis-www.cs.umass.edu/lfw/

Names100: Which contains 80,000 unconstrained human face images, including 100 popular names and 800 images per name. The dataset can used to study the relation between a person's first name and his/her facial appearance: https://exhibits.stanford.edu/data/catalog/tp945cq9122

After going through all of the available resources I eventually decided to use two available datasets mentioned above. After the first analysis, we found that certain names, like "Michael," had around 162 corresponding images, causing the data to become unbalanced and leading to biased predictions. To resolve this issue, we gathered more data points for different names, so increasing the collection of names. We then ran a bash script for each corresponding label, ensuring that each label had a minimum of 3 and a maximum of 10 images.

However, getting images of people that were labeled with their names turned out to be incredibly difficult due to legal and privacy issues. Even after going through the datasets and processing the images, we still didn't have enough images. To overcome this, I came up with a better solution. I created a script that used the Google API to search for the top 200 names across all search engines and download 10 images for each label. This helped us to get the necessary images for our project.

So we did the following:

2.1 Most Popular Names

The first step in the data collection process was to compile a list of the 200 most popular names for men and women. This list will be used to query images of people with these names.

2.2 Creating Folders

For each name in the list, a folder was created to store the images corresponding to that name.

2.3 Querying Images

A Python script to search for images of people with each name using the Google Custom Search JSON API. For each name, the script queried the API to retrieve the top 10 image results. To access the API, an API key and a cx ID were required. These credentials were obtained by creating a project on the Google Cloud Platform and enabling the Custom Search JSON API.

2.4 Downloading and Saving Images

Once we got the image results, the script downloaded each image and saved it in the corresponding folder for the name.

2.5 Function for Processing the Images

1. Converting images to PNG format: We converted all the images in the dataset to the PNG format for consistency.

2. Resizing images: As images in the dataset have different dimensions, we resized them to a uniform size of 224x224 pixels.

3. Converting images to arrays: We converted each image to a NumPy array representing the RGB values of each pixel. Each array has a shape of 224x224x3, corresponding to the height, width, and color channels (RGB) of the image. This will represent the images and their RGB matrix.

4. Normalizing pixel values: To speed uo the model training, we normalized the pixel values in the image arrays by dividing them by 255. And this

will scales the pixel values to the range [0, 1].

```
def preprocess_image(image, size = (224, 224)):

    #resize the image to a specific dimenssion and convert it to RGB
        channel
    image = Image.open(image).convert('RGB')
    image = image.resize(size)

    #convert image to array of pixel values (RGB value of every pixel)
    #it is a matrix of size 224*224, where each item is an array of size
        3 (RGB)
    image_array = np.array(image) / 255.0

    return image_array
```

3 Model Implementation

For this problem, we are using a Convolutional Neural Network (CNN) to predict a person's name from their image. Which as we know it is very suitable for image classification. Since we are using the CNN for our optimizing algorithm and we have to defind our layers, and we are using ReLu Activation which is basically the Max cost function, and Softmax Activation for our last layers. (ReLU activation is used to use non-linearity into the model and learn more complex features)

```
def tensorflow_model_optimized(input_shape, num_classes):
    model = Sequential([
        #Convolutional Layer
        Conv2D(32, (3, 3), activation='relu', input_shape=input_shape),
        BatchNormalization(),
        #decrease the number of parameters
        MaxPooling2D((2, 2)),
        Conv2D(64, (3, 3), activation='relu'),
        BatchNormalization(),
        #decrease the number of parameters
        MaxPooling2D((2, 2)),
        Conv2D(128, (3, 3), activation='relu'),
        BatchNormalization(),
        #decrease the number of parameters
        MaxPooling2D((2, 2)),
        Flatten(),
        #first fully connected layers
        Dense(128, activation='relu'),
        #for regularization
        Dropout(0.5),
        #fully connected layers (Number of neurons should equal to the
            number of classes)
        Dense(num_classes, activation='softmax')
    ])

    model.compile(optimizer='adam', loss='categorical_crossentropy',
        metrics=['accuracy'])
    return model
```

1. **Conv2D (32 filters, 3x3 kernel size, ReLU activation)**: This is the first convolutional layer, which applies 32 filters with a kernel size of 3x3 to the input image.

2. **MaxPooling2D (2x2)**: This layer performs max-pooling with a pool size of 2x2, which reduces the spatial dimensions of the feature maps and helps to decrease the number of parameters in the model. This helps the efficiency and overfitting problem.

3. **Conv2D (64 filters, 3x3 kernel size, ReLU activation)**: The second convolutional layer uses 64 filters and a 3x3 kernel size. We increased number of filters so that the model can learn more complex features from the input data of the last layer.

4. **MaxPooling2D (2x2)**: Another max-pooling layer with a pool size of 2x2 is used same as before.

5. **Conv2D (128 filters, 3x3 kernel size, ReLU activation)**: The third convolutional layer applies 128 filters with a 3x3 kernel size, for even more complex features as mentioned before.

6. **MaxPooling2D (2x2)**: The final max-pooling layer further reduces the spatial dimensions and decreases the number of parameters in the model.

7. **Flatten**: This layer flattens the output of the previous layer into a one-dimensional array. This will allow us to use this as an input for the fully connected layers an the next step.

8. **Dense (128 units, ReLU activation)**: The first fully connected layer consists of 128 units.

9. **Dropout (0.5)**: A dropout layer with a rate of 0.5 is included to overcome overfitting and have generalization. This layer randomly sets 50 percent

of its input units to 0, so that the model will have to learn more strong features.

10. **Dense (num_classes units, Softmax activation)**: The final fully connected layer that has exactly the amount of classes (names) units. The Softmax activation function is used at the end, so we can convert the output of this layer into a differential function to have the probabilities, and represent the likelihood of the input image belonging to each class.

4 Training Process

We used the Keras library to implement our CNN model and the training process. And our model is sequential model from keras:
https://www.tensorflow.org/guide/keras/sequential_model
To train our CNN model, we used the following parameters:

- **Optimizer**: We used the Adam optimizer, which is a popular choice for training deep learning models due to its adaptive learning rate and efficient computation.
- **Loss function**: We use a loss function (categorical crossentropy) to measure how well our model is doing. During training, which measures the dissimilarity between the true class probabilities and the predicted class probabilities. We want to minimize this loss so that the model can make better predictions.
- **Batch size**: We used a batch size of 32. This means that the model is updated using the gradients computed from 32 images at a time.

```
def data_batch(data, batch_size=32, num_classes=None):
    while True:
```

```
np.random.shuffle(data)
for i in range(0, len(data), batch_size):
    batch_data = data[i:i+batch_size]
    X_batch = np.array([preprocess_image(img_path) for
        img_path, _ in batch_data], dtype=np.float32)
    y_batch = np.array([label for _, label in batch_data])
    y_batch = tf.keras.utils.to_categorical(y_batch,
        num_classes=num_classes)
    yield X_batch, y_batch
```

- **Epochs**: We trained the model for 20 epochs, meaning that the model goes through the entire training dataset 20 times, and updates its weights at each pass.

5 Evaluation

To evaluate our model's performance, we can use different methods such as F1 score. After training our model for 20 epochs, we achieved a model that was best optimized to the input data with an accuracy of less than 25 percent, as it may look very unsatisfying at the first glance, in further analysis of our topic we would understand that for our model the accuracy will not have the meaning as it has in other problems, since a person's name cannot be labeled as a Right or Wrong label. Hence in this concept we would be dealing with a completely different evaluation.

6 Prediction

It is important to mentioned that we won't be able to upload the dataset since it is about 4GB, including the training set and the test set. The test set is a

disjoint of 500 images that are not included in the training set.

Once our CNN model has been trained, we can use it to predict a person's name from their image. We first preprocess the input image converting it to a 224x224 pixel array and normalizing the pixel values.

Then we then pass this preprocessed image through our trained model, which outputs a probability distribution over the possible classes. The class with the highest probability is chosen as the predicted name for the input image.

```
def name_pridiction(model, image_path, class_labels, img_size=(224,
    224)):

    #preprocess the image
    image_array = preprocess_image(image_path, img_size)
    image_batch = np.expand_dims(image_array, axis=0)
    predictions = model.predict(image_batch)

    #class value prediction in dic
    predicted_class_index = np.argmax(predictions[0])
    for label, index in class_labels_dic.items():
        if index == predicted_class_index:
            return label
```

```
Epoch 1/20
164/164 [==============================] - 305s 2s/step - loss: 7.1925
Epoch 2/20
164/164 [==============================] - 304s 2s/step - loss: 6.8690
Epoch 3/20
164/164 [==============================] - 305s 2s/step - loss: 6.8461
Epoch 4/20
164/164 [==============================] - 307s 2s/step - loss: 6.8301
Epoch 5/20
164/164 [==============================] - 316s 2s/step - loss: 6.8406
Epoch 6/20
164/164 [==============================] - 304s 2s/step - loss: 6.8130
Epoch 7/20
164/164 [==============================] - 288s 2s/step - loss: 6.8017
Epoch 8/20
164/164 [==============================] - 292s 2s/step - loss: 6.7976
Epoch 9/20
164/164 [==============================] - 285s 2s/step - loss: 6.8085
Epoch 10/20
164/164 [==============================] - 283s 2s/step - loss: 6.7765
Epoch 11/20
164/164 [==============================] - 284s 2s/step - loss: 6.7803
Epoch 12/20
164/164 [==============================] - 287s 2s/step - loss: 6.7639
Epoch 13/20
164/164 [==============================] - 294s 2s/step - loss: 6.7600
Epoch 14/20
164/164 [==============================] - 298s 2s/step - loss: 6.7534
Epoch 15/20
164/164 [==============================] - 296s 2s/step - loss: 6.7750
Epoch 16/20
164/164 [==============================] - 296s 2s/step - loss: 6.7466
Epoch 17/20
164/164 [==============================] - 297s 2s/step - loss: 6.7430
Epoch 18/20
164/164 [==============================] - 316s 2s/step - loss: 6.7389
Epoch 19/20
164/164 [==============================] - 304s 2s/step - loss: 6.7414
Epoch 20/20
164/164 [==============================] - 296s 2s/step - loss: 6.7356
```

Figure 1: CNN Optimization model history with batch_size of 32 and epochs 20

```
print("image\n---------------------- \n")
image = "IMG_1828profile.png"
prediction = name_pridiction(model, image, class_labels_dic)
print("The predicted class label for the image is: %s\n" %prediction)
```

```
image
----------------------

1/1 [==============================] - 0s 224ms/step
The predicted class label for the image is: Tyler
```

Figure 2: A prediction based on an image, which hasn't been processed