Project 3 - Facial Expression Recognition

Group 2: Leesann Sutherland, Maryan Mahamed, Stanislav Taov

03/01/2020

Introduction

Neural Networks have been at the foreground of Computer Science over the last decade, with applications varying from hand writing recognition, audio and video interpretation and facial recognition. With theory based on the biological function of the brain, however, building a neural network is no simple task. Much like teaching a baby through repeated exposure, training an artificial neural network requires large amounts of training data, extensive computing power, is time consuming and can be very expensive.

While there are different types of neural networks, we focus our attention on Convolution Neural Networks (CNN), which are primarily used in computer vision and image classification tasks. CNN's are composed of a series of layers designed to extract and filter relevant features from an input image, passing them to the next layers for further processing via a series of neurons. Similarly to how the biological visual system work, the input image is broken into overlapping regions known as receptive fields, from which these features are extracted. Neurons respond to stimuli from these regions and transfer data to the next layer to extract more complex features, until it output a classification.

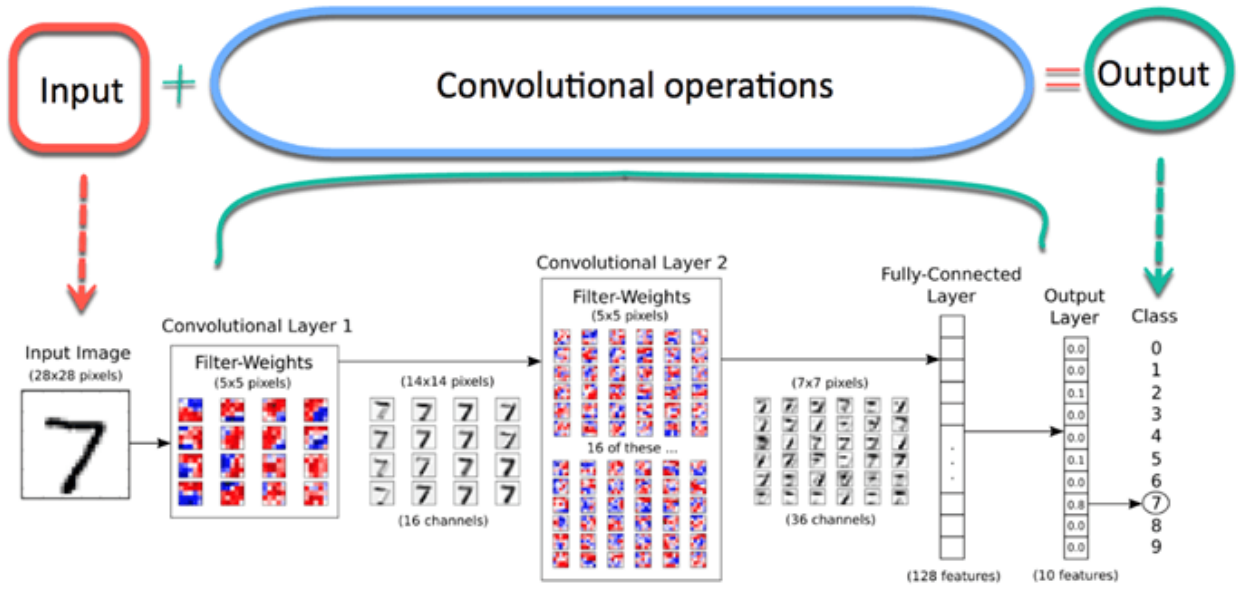

Figure 1: CNN Architecture

Since CNN's can be complex and time consuming to build and train from scratch, this process can be bypassed through the use of transfer learning. Transfer learning employs the knowledge of a pre-trained model in order to extract features from a new set of data it has never seen before. This can be likened

to the situation where a person who speaks French can apply that base knowledge to learn Spanish, which has similar linguistic roots. Keras neural networks package offers a series of pre-trained models that can be applied for transfer learning, but also allows users to build neural frameworks from scratch, in conjunction with TensorFlow.

In this project we use the Keras package for R in conjunction with TensorFlow to build and train a model to recognize facial expressions. Our data, obtained from Kaggle, contains 7 classes of facial expressions: Angry, Disgust, Fear, Happy, Neutral, Sad and Surprise, already separated into training and validation sets. Due to limitations in computing power and time, we built our model to strictly recognize Happy or Sad expressions. Our initial attempt sought to use transfer learning to compare the ability of 3 pre-tained models, InceptionV3, resNet50 and VGG16, to learn how to recognize these expressions, but were unsuccessful. Ultimately, we built a model for scratch, which we describe below, and created a Shiny app where a user can.....(input an image to see whether model accurately identfies a happy or sad expression.)

Data Preparation

Our dataset had been previous split into training and test sets, and to reduce the amount of computational power and time needed to train our neural network, we limited our expression classes to "Happy" and "Sad". Our training set was composed of 7164 Happy images and 4938 Sad images, which our validation set was composed of 1825 Happy images and 1139 Sad images, all of size 48 x 48 pixels and greyscale. To begin, these two expression classes were assigned a variable called "face_categories", with the length of this vector indicating the number of output classes.

```
# List of categories
#  face_categories <- c("happy", "sad")

# Number of output classes
#  output_n <- length(face_categories)
```

Our pre-processing parameters were then defined. Our standardized input image size was set to 244 X 244 pixels and since our images were greyscale we set the channel number to 1. The directory from which the training and validation images would be called was then defined, and their corresponding image data generators were set to normalize each pixel from a values between 0 and 255 to values between 0 and 1. The images were then loaded from the assigned path, applying all the defined pre-processing parameters that would allow the data to then be passed into the convolutional layers.

```
#  img_width <- 244      #Add explanation for increasing size above
#  img_height <- 244
#  target_size <- c(img_width, img_height)
#  channels <- 1

# Path to image folders
#   train_image_files_path <- choose.dir()
#   valid_image_files_path <- choose.dir()

# Normalizing the image matrix
#   train_data_gen = image_data_generator(
#     rescale = 1/255)

#   valid_data_gen <- image_data_generator(
#     rescale = 1/255)
```

```
# Loading training images
#   train_images <- flow_images_from_directory(train_image_files_path,
#                                              train_data_gen,
#                                              target_size = target_size,
#                                              class_mode = "categorical",
#                                              classes = face_categories,
#                                              color_mode = "grayscale",
#                                              seed = 42)

# Loading test images
#   valid_images <- flow_images_from_directory(valid_image_files_path,
#                                              valid_data_gen,
#                                              target_size = target_size,
#                                              class_mode = "categorical",
#                                              classes = face_categories,
#                                              color_mode = "grayscale",
#                                              seed = 42)
```

Model Creation

With the test and training images pre-processed, we were then ready to set our training and validation parameters. Passing a large dataset into the training algorithms at once is inefficient, therefore we broke the total set into smaller batches. To optimize the model, the data needs to pass through the algorithm several time, as determined by the number of epochs. The number of epochs selected help to reduce the possibility of over- or underfitting the model, where too few epochs leads to an underfit, or too many can lead to an overfit. For our data, we selected a batch size of 32 to be interated over 10 epochs.

To build the model from scratch we used a sequential Keras model, allowing us to define a series of linearly stacked layers. These layers are defined and described below.

First Convolutional Layer

The first convolutional layer passes the pre-processed input images through 32 3 x 3 pixel filters, with a default stride of 1. This process inevitably reduces the width and height of the input, thus by setting padding = "same", we are ensuring that the output maintains the input size, by padding the outer edges of the feature map with zeros. The output feature map of this layer is then passed to the Relu activation function to convert all negative valued pixels into zeros.

Second Convolutional Layer

The second convolutional layer uses the out feature map of the first layer and its input, passing it through 16 more 3 X 3 pixel filters to extract deeper features, maintaining the same stride pattern and padding. Rather than applying the the regular Relu activation function to convert negative pixels to zeros, we allow allow for small negative values and then normalize.

Pooling Layer

To reduce the dimensions and computational complexity of the resultant feature map from the previous layer a pooling layer is added. Here we use the 2 X 2 max pooling matrix, which scans over 2 strides. This halves the size of the feature map by selecting only the maximum weight value from each 2 x 2 jump. In order to prevent overfitting the dropout method is employed, whereby randomly dropping some neurons with each batch that passes through the layer, feature weights are allowed to average out. Here we use a drop rate of 0.25.

```
# Initialise model
#    model <- keras_model_sequential()

# Add layers
#    model %>%
#      layer_conv_2d(filter = 32, kernel_size = c(3,3), padding = "same",
#                    input_shape = c(img_width, img_height, channels)) %>%
#      layer_activation("relu") %>%

# Second hidden layer
#      layer_conv_2d(filter = 16, kernel_size = c(3,3), padding = "same") %>%
#      layer_activation_leaky_relu(0.5) %>%
#      layer_batch_normalization() %>%

# Use max pooling
#      layer_max_pooling_2d(pool_size = c(2,2)) %>%
#      layer_dropout(0.25) %>%

# Flatten max filtered output into feature vector and feed into dense layer
#      layer_flatten() %>%
#      layer_dense(200) %>%
#      layer_activation("relu") %>%
#      layer_dropout(0.5) %>%

# Outputs from dense layer are projected onto output layer
#      layer_dense(output_n) %>%
#      layer_activation("softmax")

# Compile
#   model %>% compile(
#               loss = "categorical_crossentropy",
#               optimizer = "SGD",
#               metrics = "accuracy"
#             )
```

Dense and Outer Layers

The Dense (or fully connected) layer of the model enable the model to output a classification for the input image. In order to pass the pooling output to the dense layer, however, the multi-dimensional tensors of the pooling output must be flatten into a 1-dimensional tensor vector. The values of this vector represent the probability of a particular feature belonging to a particular classification. The output of the dense layer is then subjected to Relu activation and dropout before it is projected to the outermost layer, using the softmax activation function, to be classified as happy or sad.

Compiling the Model

Before passing our training set through the model, we needed to compile the layers built in the previous steps. Compiling the model allows us to define how best to improve the model each time a batch is passed through the network. For our model we use the stochastic gradient descent (SGD) optimizer to readjust the weights between passes, and the cross-entropy loss function to measure the model's performance. If our model is good, we expect a smaller log loss with each pass. We also tell it to report on accuracy.

Model Testing and Validation

Our next step was to train and validate our model, which were combined into one operation using the fit_generator function, shown below. In this function we indicated the number of training/validation images to use per batch (steps_per_epoch), which after trial and error ended up being the total number of images divided into 32 groups. We set up the fit-generator to save the best version of the model after each epoch to be used in the deployment of our Shiny app later, as well as a log of training and validation metrics for visualization, which can be seen below.

```
# Fit

# hist <- model %>% fit_generator(
#               #Training data
#               train_images,

#               #Epochs
#               steps_per_epoch = as.integer(train_samples / batch_size),
#               epochs = epochs,

#               #Validation data
#               validation_data = valid_images,
#               validation_steps = as.integer(valid_samples / batch_size),

#               #Print progress
#               verbose = 1,
#               callbacks = list(
#                  #Save best model after every epoch
#                  callback_model_checkpoint("/Users/stantaov/Documents/keras/face.h5",
#                                            save_best_only = TRUE),
#                  #Only needed for visualising with TensorBoard
#                  callback_tensorboard(log_dir = "/Users/stantaov/Documents/keras/logs")
#               ))
```

TensorBoard Output

```
Found 12102 images belonging to 2 classes.
Found 2964 images belonging to 2 classes.
Epoch 1/10
379/379 [==============================] - 1071s 3s/step - loss: 0.7570 - acc: 0.6484 - val_loss: 0.6579 - val_acc: 0.6586
Epoch 2/10
379/379 [==============================] - 1208s 3s/step - loss: 0.5398 - acc: 0.7221 - val_loss: 0.5488 - val_acc: 0.7105
Epoch 3/10
379/379 [==============================] - 1137s 3s/step - loss: 0.4845 - acc: 0.7645 - val_loss: 0.4853 - val_acc: 0.7611
Epoch 4/10
379/379 [==============================] - 1126s 3s/step - loss: 0.4302 - acc: 0.7976 - val_loss: 0.5889 - val_acc: 0.7291
Epoch 5/10
379/379 [==============================] - 1116s 3s/step - loss: 0.3717 - acc: 0.8321 - val_loss: 0.5077 - val_acc: 0.7547
Epoch 6/10
379/379 [==============================] - 1106s 3s/step - loss: 0.3038 - acc: 0.8686 - val_loss: 0.5218 - val_acc: 0.7750
Epoch 7/10
379/379 [==============================] - 1112s 3s/step - loss: 0.2372 - acc: 0.9038 - val_loss: 0.5076 - val_acc: 0.7827
Epoch 8/10
379/379 [==============================] - 1102s 3s/step - loss: 0.1887 - acc: 0.9271 - val_loss: 0.6683 - val_acc: 0.7540
Epoch 9/10
379/379 [==============================] - 1124s 3s/step - loss: 0.1524 - acc: 0.9423 - val_loss: 0.6798 - val_acc: 0.7736
Epoch 10/10
379/379 [==============================] - 1133s 3s/step - loss: 0.1192 - acc: 0.9566 - val_loss: 0.7050 - val_acc: 0.7652
```

Figure 2: Loss and Accuracy Report

Figure 2 above shows the Loss and Accuracy reported for each epoch for both the training set and validation set. As expected, with each training epoch the network became better at accurately recognizing whether a facial expression was happy or sad. That is, we see the loss values decrease from 0.7570 in the first epoch to 0.1192 by the tenth, while the accuracy increased from 64.84% to 95.66%. The accuracy for the validation set peaked at 78.27%, stabilizing within the upper 70's range. These results can be visualized below in Figure 3.

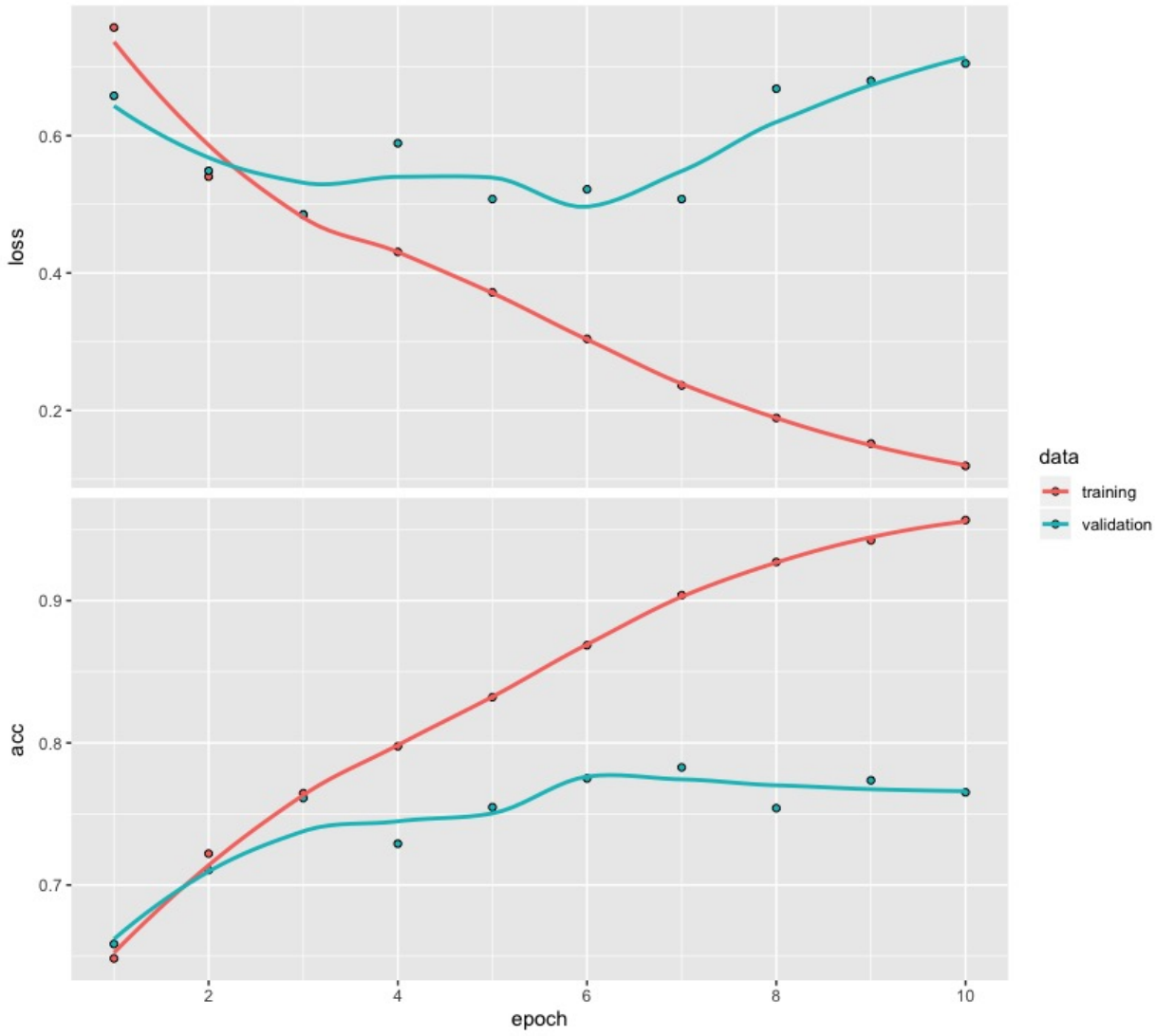

Figure 3: Loss and Accuracy per Epoch

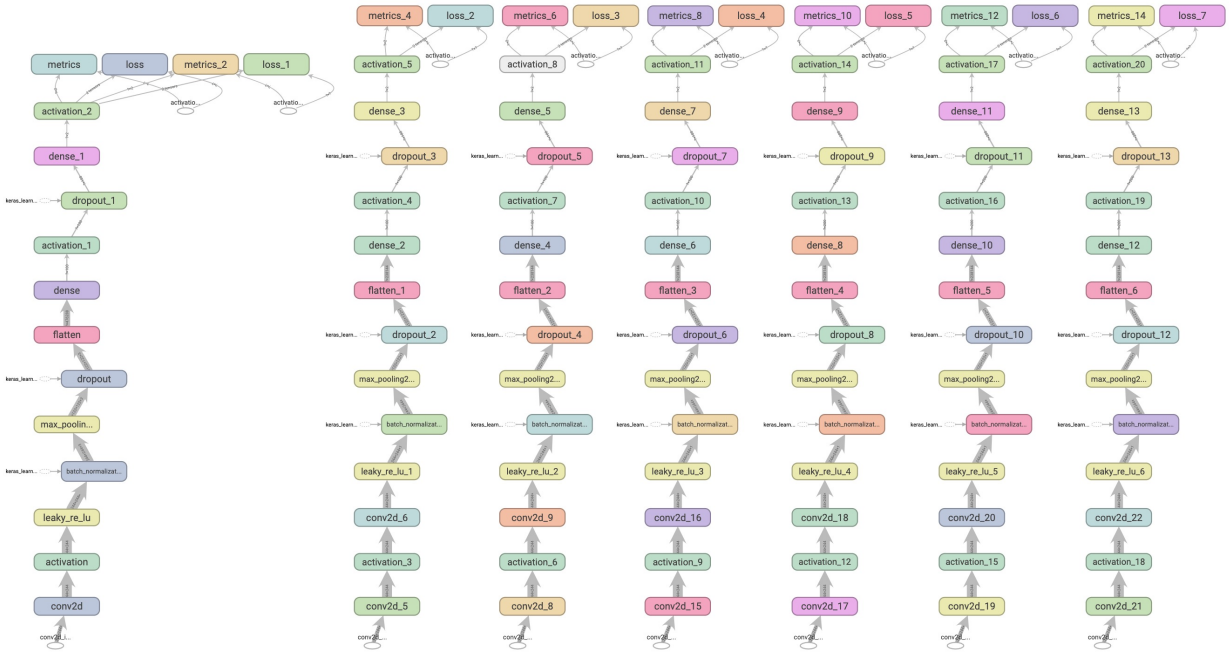

Figure 4: Model Structure

Figure 4 shows the structure of our neural network as outputed in the TensorBoard.

Limitations

For this project our initial intention was to apply Transfer Learning to our data set, using pre-trained models VGG16, ResNet50 and InceptionV3, which had been used in similar facial recognition tasks before, and compare their levels accuracy. During the implementation, however, we encountered a number of limitations due to the size of the data, time needed to train, and package installation problems. None of the 3 models would run properly, often causing R to crash, leading us to limit out input data from 7 facial expression classes to 2. This unfortunately did not work, and we unltimately opted to build the model from scratch using just 2 image classes.

While we successfully built a CNN from scratch, it was not without its own problems. Finding the right parameters took trial and error, and numerous hours of attempted training only for it to crash in the final epoch. Once the best suited parameters were found, we discovered that while training presented good results, it was unable to accurately classify new input. In order to improve this these results, the entire set of training and validation images were used. This increased the training time to approximately 5 hours, but produced the results presented above.

Shiny App Implementation

We created an app that would allow users to gauge whether their photos show expressions which tended toward "Happy" or "Sad". The app, called "Facial Expression Recognition Test" can be found at https://www.dropbox.com/s/djtss9dl4goqxo9/Facial%20Expression%20Recognition.zip?dl=0. The model file was too large to upload to shiny.io or GitHub, so we opted to provide access via Dropbox. The app's functionality is described below.

The app allows users to upload a greyscale image file in either jpeg or png format. This is done using the "Browse" button to search images within the users desktop libraries. Once an appropriate image is selected and uploaded, the model is activated and begins to processes the image to determine facial expression.

There are three aspects to the app's output:

1. The app will display the user's uploaded image in the main panel, allowing the user to view the image alongside the model's output.
2. The app will display a percentage that represents the probability that the model's prediction matches the actual expression in the image. The model outputs two prediction percentages for "Happy" or "Sad", with a benchmark of 50%.
3. In conjunction with the percentage, the app will output a statement telling the user whether the uploaded image tends to a "Happy" or "Sad" expression. The statement output depends on the predictions made by the model.

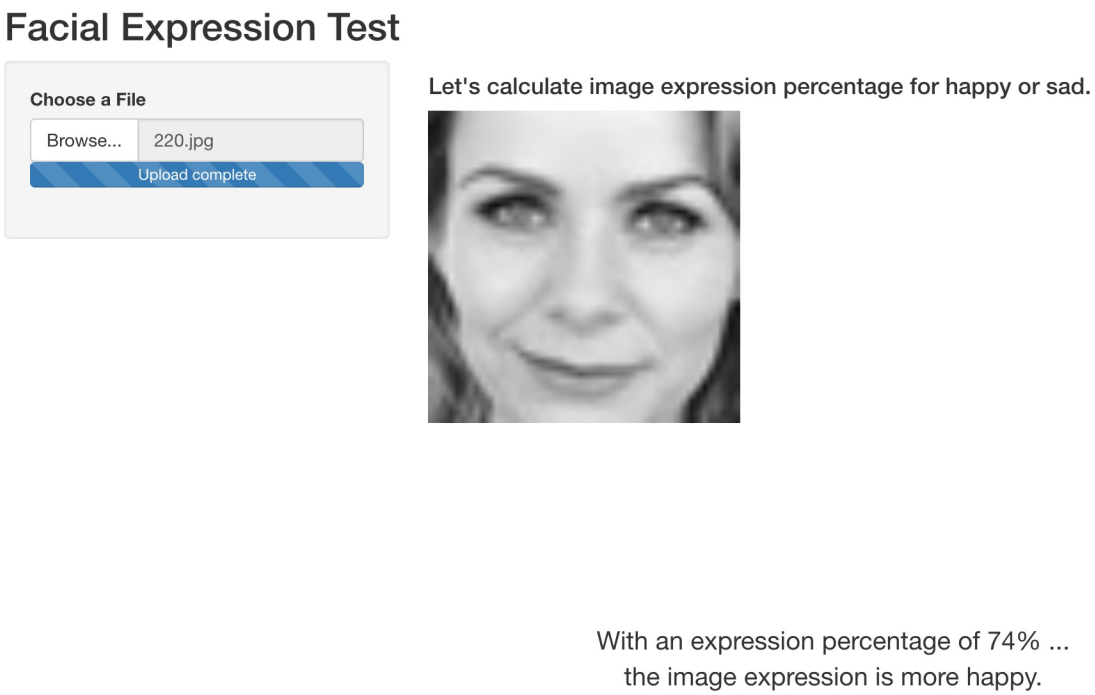

Figure 5: Shiny App Interface

Conclusion

At the beginning of this project, our initial intention was to use transfer learning to train a an existing model to recognize facial expressions. In theory (and often in practice), using a pre-trained model as a launching point for a slightly more complicated task reduces the training time as well as the amount of data needed to reach the end goal. However, with numerous pre-trained models available, limited time to find the right one, and a lack of access to adequate computing power we were unsuccessful in this endeavor, despite limiting the number of classification categories to Happy and Sad.

While we anticipate the longer training time, opting to build a convolutional neural network from scratch did come with the benefit of allowing us to obtain a much deeper grasp on the stucture and inner workings of neural networks. While the theory behind neural networks and their construction is simple enough to understand, the The most interesting insight gained lies in the complexity. As humans, the task of analysing facial expressions is so much a part of our nature that it is easy to take for granted just how computationally complex that task actually is, and why it is so difficult to accurately replicate in a machine.

In facial recognition tasks neural networks are designed to pick up on the patterns of various facial features, from the shape of the eyes, nose, mouth, etc. While this may be simple enough in the usecase of differentiating between a human and a dog, differentiating between individual human facial expressions is quite a bit more complicated. This is not only due to the fact that there is a multitude of expressions, but also because expressions range between emotional extremes, many of which share specific features. These nuances explain why with a simple neural network, trained on only two expression classes, is limited in its ability to accurately classify the expression on an image it has never seen before. A simple network like ours may perhaps accurately predict the more obvious extremes of Happy or Sad, but would certainly have difficulty classifying a smirk. Given the limitations, the success rate of predictions for our model is quite impressive. With more computing power it would be interesting to investigate how the model responds to additional levels of complexity.

Sources

1. "TensorFlow Image Classification: CNN (Convolutional Neural Network)." *Guru99*, 02/28/2020, https://www.guru99.com/convnet-tensorflow-image-classification.html.

2. Sharma, Sagar. "Epoch vs Batch Size vs Iterations." *Towards Data Science*, Medium, 09/23/2017, https://towardsdatascience.com/epoch-vs-iterations-vs-batch-size-4dfb9c7ce9c9.

3. Karpathy, Andrej. "CS231n Convolutional Neural Networks for Visual Recognition." *GitHub Pages*, GitHub Inc, 05/20/2019, http://cs231n.github.io/convolutional-networks/#conv.

4. Falbel, Daniel, et al. "R Interface to 'Keras'." *RStudio*, RStudio Inc, 02/18/2020, https://keras.rstudio.com/index.html.

5. Juvonen, Antti. "Overfitting and Dropout in Neural Networks." *CAP Data Technologies*, 01/23/2018, https://capdatatechnologies.com/theory/2018/01/23/overfitting-and-dropout-in-neural-networks.html