Optimized Real-Time Emotion Detection and Classification by Deep Neural Networks

Gavin GF Rice
gavinr@sfu.ca

Abstract

Many systems in the modern world rely on image recognition and classification models to operate many critical services. The most common form of image recognition neural network architecture is a Convolutional Neural Network or CNN. What is a CNN and how does it work? CNNs take the input data, typically a matrix which is passed through a series of convolutional layers and then it applies a filter over the image. This filter works across the entire matrix and it outputs a transformed matrix that is determined by the weights and bias parameters. This act is called the convolution step and it is followed by a batch normalization where the data of each batch is normalized such that its data points have a mean of 0 and a standard deviation of 1. The network then has a pooling layer in order to extract the most important features of each feature map. After removing certain data points to reduce overfitting, and repeating this process, the model will need to be fully connected by a dense layer. Once the data is propagated through these final layers where it repeats the normalization and dropout steps again until arriving at a final output layer. An output is returned after the final fully connected layer is computed and where softmax is used to receive the end result or classification. Thousands of applications for CNNs exist and can be used to classify just about anything that can be captured by an image. From classifying the digits 1-10 to recognizing different street signs, CNN and its applications are diverse and can be used in almost any case. Many of these use cases are in regards to distinct classification problems like determining model and make of a given vehicle. Fewer models tackle the issues of interpreting and classifying organism exclusive matters such as emotion and sentiment. This use case is interesting as even humans struggle occasionally to determine the underlying human emotion. How can a machine be taught to understand and interpolate emotion? If the heavy philosophical implications that such a question imposes are excluded, can the same approach that other classification models be used for such a task? These questions are the primary concern of this paper. Specifically, it will demonstrate that, yes, such a technique is effective for identifying emotions and this can be applied to problems in real-time for a variety of applications.

Experimental parameters and conditions

For the CNN, a Python notebook is used in order to extract the model and use it repeatedly elsewhere in another environment. For the initial CNN model development environment the Kaggle kernel, which runs the Python Notebooks in a cloud environment so that there is no need to install dependencies on a local machine, was used. This particular notebook VM is configured with a Linux operating system and 12GB of RAM available as well as having access to a Nvidia Tesla P100 2 Cores, 16 GB GPU for runtime acceleration. To build the real time face and emotion detector, Google Colab is used as the cloud development environment and its VM uses the Linux kernel with 12GB of RAM and a Nvidia K80 2 Core, 16GB GPU. Cloud based virtual environments are used in favor of a local environment in order to increase testing/training performance as well as to make the entire experiment far more repeatable. This is in addition to cloud VMs being way easier to configure as the experiment and requirements evolve. The webcam that will be used for the real-time face and emotion detection is from a Macbook Pro 2020 with an Apple M1 chip and it possesses a 720p HD camera. Links to all of the code is provided at the end of the paper in the code links section.

CNN Model: Network Architecture

The model is broken up into 2 different parts, the convolutional layers and the fully connected layers. It begins by creating the convolutional layers and then instantiates the model with Keras's `Sequential` class which is the most common for multi-class classification models. For the first convolutional layer the `Conv2D` layer class is applied. It creates a convolutional kernel or window that is of size 3x3 which scans across the input matrix and performs some operation on it and it outputs a transformed tensor of slightly smaller dimensions.

$$G[m,n] = (f * h)[m,n] = \sum_j \sum_k h[j,k] f[m-j, n-k]$$

Fig. 1. The transformed matrix $G[m,n]$ with m columns and n rows is derived with this equation where the summation is the convolutional product. Here h represents the kernel of size j by k and f is the input image of size m x n.

The first argument is the filter number which describes the amount of filters that convolutional layers will learn from. As the layers depth gets deeper, the amount of filters and kernel size is increased to capture different patterns such as eyes, mouths and face edges. The larger the filter and kernel size means it can capture more and more complex combinations of patterns that would not be seen otherwise. Same padding is used in order to deal with potential information loss due to the convolution. This allows each stride of the convolution to have the 'same' amount of information in each iteration.

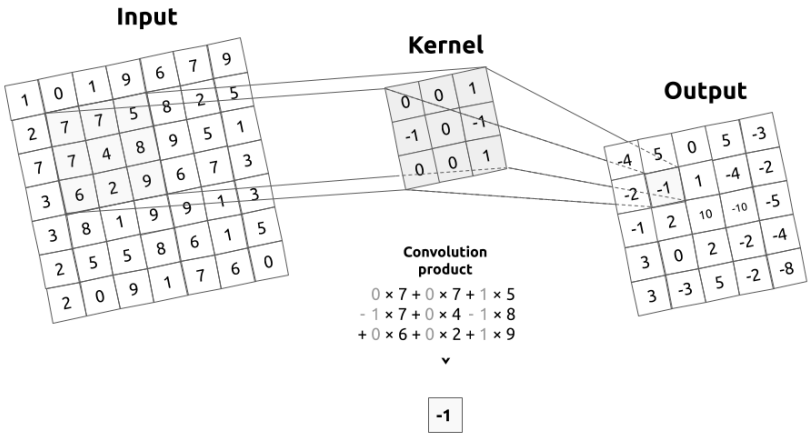

Fig. 2. The 3x3 filter slides across each entry in every row/column where the convolutional product is computed and returned to the corresponding entry index in the output tensor.

Following the convolutional step is the batch normalization step provided by Keras's `BatchNormalization` layer which seeks to increase performance and avoid overfitting. It is important that the data is normalized such that all its data points have a mean of 0 and a standard deviation of 1. In batch normalization, it is normalizing the layers of the neural network as opposed to the actual raw input data, which helps boost training speed and increases learning rates over later epochs. For this layer to actually be able to have data that gets neurons to be 'activated' or passed on to the next layer, an activation function like the Rectified Linear Unit activation function. ReLU is very common in CNN models as it helps prevent exponential growth for the computations in the network. As the CNN scales in size and parameters, the computational cost of adding additional ReLUs increases at a linear rate hence the name. ReLU is defined as $f(x) = max(0, x)$. To extract the important features from the feature map, for this, max pooling is utilized. `MaxPool2D` works similar to `Conv2D`, but instead of a

convolutional product, it takes the maximal element that is covered by the pooling filter. Its output is a feature map that contains the most prominent features of the original map. For all of the pooling layers, the model used a pool size and stride size of 2x2.

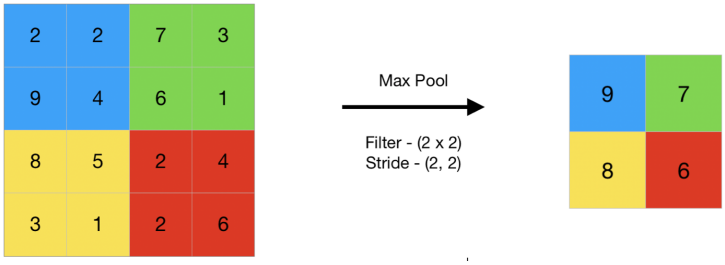

Fig. 3. Each quadrant (2x2 matrix) is pooled and its maximal element is returned to the pooled feature map.

This process however can cause something called overfitting, where a model learns about details and noise in the training data that negatively affects the output of the model. To reduce the threat of overfitting, it is common practice to perform a dropout using the Dropout layer which randomly selects neurons that are ignored or 'dropped' by making their inputs 0. For this model, 0.25 or 25% of the inputs of each layer are 'dropped'. Once all of the convolutional layers are completed, the fully-connected layers are built. Before the fully connected layer is derived, a 'flatten' or collapse operation occurs and the input tensors are flattened into a one-dimensional array via the `Flatten` layer. The `Dense` layer starts off the fully connected layer by having each neuron from the previous layer. A dense layer is important for performing matrix-vector multiplication during backpropagation of data. This dense layer also has the same steps performed on it minus the max pooling (due to lack of convolutions).

After creating one more fully connected layer that has 512 output units, the mode is completed with one final dense layer which has 7 outputs corresponding to the 7 total classes of emotions being targeted. In this final layer, by applying the softmax activation function to return a probability distribution between the 7 different emotion classes of the problem.

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, \ldots, K \text{ and } \mathbf{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K$$

Fig. 4. The softmax activation function is represented with the lowercase sigma. It takes an input vector z_i, z_j being the output vector for K classes.

Once the model architecture is defined, everything is combined together using the `model.compile` method. Firstly an optimizer must be selected, the optimizer selected, which in this case is `Adam`, has the goal to update the model in response to the outputs of the loss function. `Adam` seeks to minimize the loss function as much as possible through gradient descent. `Adam` has a parameter for learning rate which will determine the step length that the optimizer takes during gradient descent. The larger the rate is, the faster the convergence is to the local minimum, the smaller it is, increases the iteration number but also increases the accuracy and closeness to the true minimum. Opting for the middle ground, a moderate learning rate of 0.0001 is applied. Categorical cross entropy is a loss function that is well suited to multi-class classification problems and it works in conjunction with softmax activation functions. Its function is to quantify the differences between different probability distributions such as the experimental distribution (what the model would output) and the true target labels.

$$\text{Loss} = -\sum_{i=1}^{\substack{\text{output}\\ \text{size}}} y_i \cdot \log \hat{y}_i$$

Fig. 5. Loss is computed using this formula, $\hat{y}_i$ is the i-th scalar value stochastic vector of the output model (experimental) and y_i is the corresponding target value. Cross-entropy is calculated for each entry of the probability distribution for every class label.

The final parameter for compilation is the particular metric that is quantified by the algorithm, this generally is the accuracy or the rate at which the correct label is applied to the correct image.

CNN Model: Data setup

For the development environment for the model, the Kaggle environment is used for training and testing. Firstly tools like `matplotlib.pyplot` and `seaborn` are used for data visualization and graphing. For data scraping and manipulation `numpy` is used along with `pandas`. The vast majority of the modeling and image preprocessing needs to be done through `keras` and its image processing tools/functions. Finally the model building blocks are imported, notably, the `Sequential` and `Model` libraries as well as some layer constructors, `Dense`, `BatchNormalization`, `Activation`,`Dropout`, `Conv2D` and `MaxPooling2D`.

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import os

# Importing Deep Learning Libraries

from keras.preprocessing.image import load_img, img_to_array
from keras.preprocessing.image import ImageDataGenerator
from keras.layers import Dense,Input,Dropout,GlobalAveragePool
from keras.models import Model,Sequential
from tensorflow.keras.optimizers import Adam,SGD,RMSprop
```

Fig. 6. Import cell from the code to include all of the libraries, packages and classes required for training and testing the model.

After the initialization of the necessary libraries and functions, the data from the face-expression-recognition-dataset can be imported. Within the dataset, there are 2 folders containing 2 different sets of images, one is `train` (for training) and the other is `validation` (for testing). Inside each set are 7 separate folders, one for each emotion label. In the train set for this particular dataset there are 7 distinct classes an image can possess, in order they are: ‘angry’ (3993 images), ‘disgust’ (436 images), ‘fear’ (4103 images), ‘happy’ (7164 images), ‘neutral’(4982 images), ‘sad’ (4938 images) and ‘surprise’ (3205 images). The images are organized numerically and are all greyscale and of size 48x48.

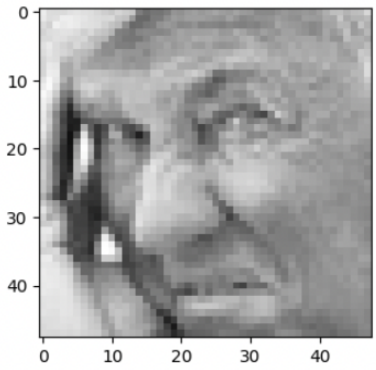

Fig. 7. An example of how the image appears in the dataset using `plt.imshow(img).` This particular example is that of the 'disgust' class.

Now the training and validation data is added for the model. Via the ImageDataGenerator function, the training and test sets are initialized. Using the flow_from_directory function, the data is able to be resized, reshaped, segmented and logged as grayscale. This function will essentially convert the data into something the model can interpret, that being a numpy array object of shape (48,48,1). The first two elements in this tuple represent the dimensions of the numpy matrix object where the pixel data of each image is located. The final element in the object is the size of the object contained in the 48x48 array, in this case it is the normalized pixel grayscale value channel. If the images had RGB values associated with them, this value would be 3. Next step is to normalize the data in the preprocessing stage in order to make the various features have similar value ranges in order for the gradient descent step to converge in fewer iterations. Images are arranged automatically into 'batches' which are propagated through the network each epoch, in this case `batch_size` is set to 128 images.

```
batch_size  = 128

datagen_train  = ImageDataGenerator()
datagen_val = ImageDataGenerator()

train_set = datagen_train.flow_from_directory(folder_path+"train",
                                              target_size = (picture_size,picture_size),
                                              color_mode = "grayscale",
                                              batch_size=batch_size,
                                              class_mode='categorical',
                                              shuffle=True)


test_set = datagen_val.flow_from_directory(folder_path+"validation",
                                              target_size = (picture_size,picture_size),
                                              color_mode = "grayscale",
                                              batch_size=batch_size,
                                              class_mode='categorical',
                                              shuffle=False)
```

Fig. 8. Running this function, it found 28821 images belonging to 7 classes in `train_set`, alongside it also found 7066 images belonging to 7 classes in the `test_set`.

Model code structure and hyperparameters

During the course of research, there were a total of 3 proposed model designs. The first was a model (Model A) with only 2 CNN layers with far smaller neuron count of 64 then 256 along with a lesser dropout rate of 0.20. The second model (Model B) utilized the same number of layers with the same number of layers as in the proposed method (Model C) but used average pooling for the pooling layer and also had a larger learning rate of 0.001 with the `SGD` (Stochastic Gradient Descent) optimizer. Below is the precise implementation for the neural network with the most optimal accuracy and loss:

```
#from tensorflow.keras.optimizers import Adam,SGD,RMSprop


no_of_classes = 7

model = Sequential()
#1st CNN layer
model.add(Conv2D(64,(3,3),padding = 'same',input_shape = (48,48,1)))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size = (2,2)))
model.add(Dropout(0.25))

#2nd CNN layer
model.add(Conv2D(128,(5,5),padding = 'same'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size = (2,2)))
model.add(Dropout (0.25))

#3rd CNN layer
model.add(Conv2D(512,(3,3),padding = 'same'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size = (2,2)))
model.add(Dropout (0.25))

#4th CNN layer
model.add(Conv2D(512,(3,3), padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
```

```
#Fully connected 1st layer
model.add(Dense(256))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dropout(0.25))
# Fully connected layer 2nd layer
model.add(Dense(512))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dropout(0.25))
model.add(Dense(no_of_classes, activation='softmax'))

opt = Adam(lr = 0.0001)
model.compile(optimizer=opt,loss='categorical_crossentropy', metrics=['accuracy'])
model.summary()
```

Fig. 9. The implementation of the neural network.

Training

Now that the model is compiled, the training may begin. Before training, a number of helper methods should be employed to further optimize the model during the training period. Using `ModelCheckpoint` to monitor for a specific condition to be met, once this trigger is activated, the model will automatically save the current weight and bias values to the `model.h5` file that provides the best metric readings. The metric to be monitored is validation accuracy which defines how accurate the model prediction is on the validation or `test_set` which is to be maximized. In order to prevent overfitting of the model, it is sometimes beneficial to cease training if the model accuracy is not increasing. Keras's `EarlyStopping` method monitors the accuracy and validation loss. Validation loss indicates how efficiently the model performs against new data such as the training set. It is calculated by finding the errors of each predicted label versus its true label. Prior to training, if the current learning rate does not notably decrease the validation loss, it reduces the learning rate in an attempt to increase accuracy with the smaller steps (learning rate = gradient descent step).

```
checkpoint = ModelCheckpoint("./model.h5", monitor='val_acc', verbose=1, save_best_only=True, mode='max')

early_stopping = EarlyStopping(monitor='val_loss',
                               min_delta=0,
                               patience=3,
                               verbose=1,
                               restore_best_weights=True
                               )

reduce_learningrate = ReduceLROnPlateau(monitor='val_loss',
                                        factor=0.2,
                                        patience=3,
                                        verbose=1,
                                        min_delta=0.0001)

callbacks_list = [early_stopping,checkpoint,reduce_learningrate]

epochs = 48

model.compile(loss='categorical_crossentropy',
              optimizer = Adam(lr=0.001),
              metrics=['accuracy'])
```

```
history = model.fit_generator(generator=train_set,
                              steps_per_epoch=train_set.n//train_set.batch_size,
                              epochs=epochs,
                              validation_data = test_set,
                              validation_steps = test_set.n//test_set.batch_size,
                              callbacks=callbacks_list
                              )
```

Fig. 10. Implementation scheme for training.

These parameters are considered callback functions which will be applied to the model during training. Now the training parameters are set by defining the `epochs` parameter to 48 as well as including the callback functions and specifying the training and validation sets. In order to access the training history data a variable `history` is set using the `model.fit_generator` function. This also begins the training process. Whilst training, the model terminated after 14 epochs with an accuracy of 75.13%. Testing loss logged in at 0.6859 after running for 8 minutes in the Kaggle kernel. The target accuracy was above 75% so this result is fairly reasonable for the needs of this paper..

```
Epoch 00014: ReduceLROnPlateau reducing learning rate to 0.00020000000949949026.
Epoch 00014: early stopping
Testing loss: 0.6859492659568787
 Testing accuracy 0.7513271570205688
```

Fig. 11. Console output that gives the best model and testing accuracy.

These numbers are good but how does the model actually learn and improve overtime? To answer this It's best to visualize the historical data from the training data. Using the `matplotlib.pyplot` package, to produce two graphs shown in figure 13.

Model	Accuracy	Loss Value
A	71.08%	0.6582
B	69.55%	0.8071
C	75.13%	0.6859

Fig. 12. Table representing the corresponding averaged accuracy and loss values across all of the different models proposed which is computed at the end of training.

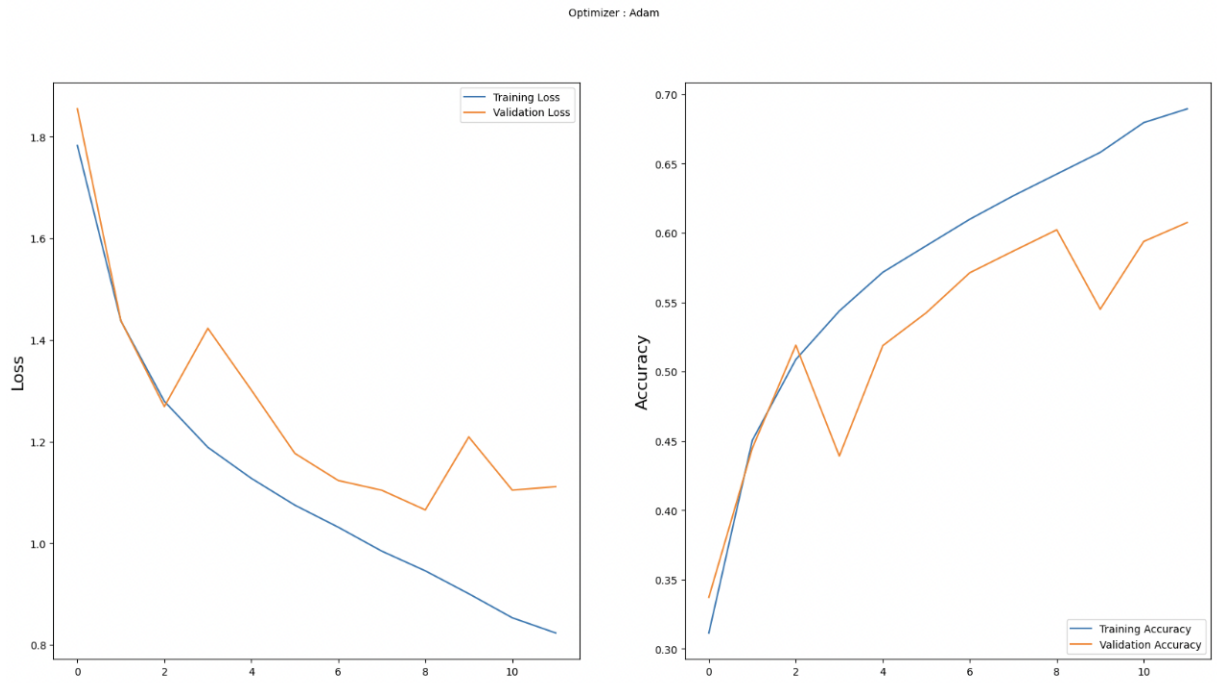

Fig. 13 - Left: The graph on the left plots the training (blue curve) and validation (orange curve) loss on the vertical axis with respect to the epoch number on the horizontal axis. Right: shows the training (blue curve) and validation (orange curve) accuracy with respect to the epoch number.

What can be observed from these graphs is that as expected, when the model is tested against images from the training set, the loss improves quite quickly as it has trained with those images before. Validation set consists of new images that the network would not have seen yet so it gives a good idea of how the model would perform on custom inputs. Due to lack of experience on the validation set, the model struggles and after about 9 epochs begins to stagnate and show little improvement in terms of loss and accuracy.

Now that the model training is complete, the model is saved using `model.save('./model.h5')` which saves the model as a Hierarchical Data Format file or .h5 file. This file type contains multidimensional arrays that are used to process scientific data by manipulating tensors through matrix and multivariate calculus operations. This file is used in order to deploy the model in another file where the webcam is used for real-time face and emotion detection.

A brief overview of the real-time webcam capture and classification software

To build the real-time image capturing tool, Google colab can be used as Kaggle does not support webcam integration. Once all of the necessary libraries are imported into the notebook, importing the model comes next along with importing another pre-trained classifier called the Haar Cascade classifier. This pre-trained model works by having a kernel traverse over every pixel in an image and it searches for specific features that comprise faces. Haar Cascade kernels typically traverse from the top left of an image to the bottom right. Once the Haar Cascade classifier searches the entire image, it will return rectangular coordinates of all of the features that it detects.

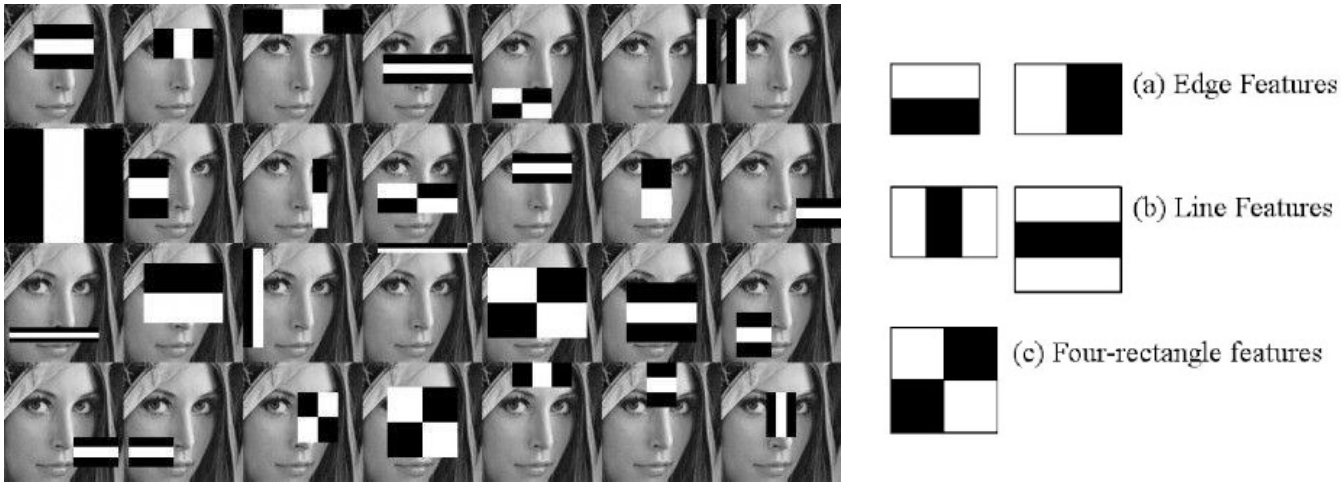

Fig. 14. Left: An example of a typical kernel traversal over an image. Right: A diagram explaining the features being detected by each particular kernel scheme.

For this application, the model from `haarcascade_frontalface_default.xml` which is designed to detect the front of human faces is imported and deployed. It achieves this by using different filters/kernels to detect common facial features such as eyebrows, lips and eyes which it uses to calculate 'Haar' features. These models were trained on thousands of positive images that contain the desired object that needs to be detected, and negative images which do not contain the desired object. Optimization is done in order to reject negative images as fast as possible and reduce the false negative detection rate through a process called AdaBoost training.

Once the models are imported onto the notebook implementing the webcam capture feature can commence. Firstly required is a function that opens the webcam through Google Colab. This is accomplished by writing a python function `video_stream` which injects some JavaScript code into the DOM of the web browser which will open the webcam. After the video stream is started, it's necessary to convert the frames of the webcam video into data that the model can parse. Another Python function, `video_frame`, is written which converts the video stream data into a JavaScript object that can be interpreted and converted to an image through the `js_to_image` function. Lastly, to render the bounding rectangles for the GUI, writing another python function `bbox_to_bytes` so that it can be displayed using `cv2` shall suffice. Bringing everything together by writing a while loop that opens the webcam stream, grabs the frames as images then runs the face detection model as well as the CNN model. Within the `while true` loop, is a nested loop which iterates over the coordinates of the faces and draws the bounding rectangles on top of them while also performing a prediction of the subject's emotion which it prints to the webcam window.

```
video_stream()
label_html = 'Capturing...'
bbox = ''
#IT WORKS!!!!!!!!!!
while True:
    #_, frame = cap.read()
    labels = []
    js_reply = video_frame(label_html,bbox)
    if not js_reply: #if feed dies, break the loop
      break
    frame = js_to_image(js_reply["img"]) #grab the frames from the image
    gray = cv2.cvtColor(frame,cv2.COLOR_BGR2GRAY)
    faces = face_classifier.detectMultiScale(gray) #convert frames to gray in backend

    for (x,y,w,h) in faces: #detect the faces from face_classifier
        cv2.rectangle(frame,(x,y),(x+w,y+h),(0,255,255),2) #draw the rectangle over all faces detected
        roi_gray = gray[y:y+h,x:x+w] #Region of interest array
        roi_gray = cv2.resize(roi_gray,(48,48),interpolation=cv2.INTER_AREA)
        if np.sum([roi_gray])!=0:
            roi = roi_gray.astype('float')/255.0
            roi = img_to_array(roi)
            roi = np.expand_dims(roi,axis=0)

            prediction = classifier.predict(roi)[0] #get prediction on detected face
            label=emotion_labels[prediction.argmax()] #get the label
            label_position = (x,y) #find face position
            cv2.putText(frame,label,label_position,cv2.FONT_HERSHEY_SIMPLEX,1,(0,255,0),2) #print label
        else:
            cv2.putText(frame,'No Faces',(30,80),cv2.FONT_HERSHEY_SIMPLEX,1,(0,255,0),2)
    cv2_imshow(frame)
    if cv2.waitKey(1) & 0xFF == ord('q'): #press q in order to break
        break #cv2.waitKey(1) = returns 32 bit int, 0xFF shifts it by 24 bits
```

Fig. 15. Implementation of the webcam software.

Each prediction label is organized via a python list, `emotion_labels = ['Angry','Disgust', 'Fear','Happy','Neutral', 'Sad', 'Surprised']` and the output of the prediction model provides an index corresponding to one of these labels when the `argmax` function is employed.

Final product performance and results

Now it is time to begin testing the model in real-time by using the webcam from a Macbook Pro M1 2020. Upon initial inspection, the whole system appears to have an inherent bias toward certain classification. While testing the real-time, it was found that the model correctly classifies emotions such as happiness, anger and neutrality and surprise with a high degree of accuracy. However in the case of classifying fear and disgust, the model was unable to get these as outputs during testing even a single time.

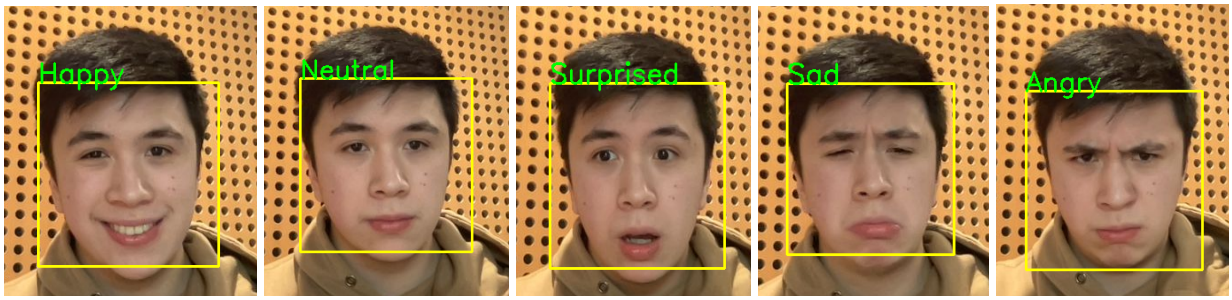

Fig. 16. Real-time outputs on a subject for the emotions from the categories happy, neutral, , surprised, sad and angry on their correct faces.

Interestingly, this outcome is somewhat expected. As seen from the distributions of the training image data, these 5 are among the most common and appear the most in the dataset. Another factor is the similarity between the classes of disgust, fear and surprise. It is not uncommon to interpret these emotions interchangeably even amongst human observers. It is also likely that within the training data, many of these emotions are present inside of the class of another emotion such as disgust and fear being occasionally equated together. Because of this feature of the data and human perception, the model is heavily biased toward the label for surprise when encountering one of these 3 emotions. As for why surprise is the most common label, this is likely due to how it is possible to consider the other 2 categories as subsets of surprise as one might experience this alongside feelings of disgust or fear and or vice versa. It is entirely possible that this neural network is experiencing similar issues that a human would while observing these particular emotions and as a result, tends to lump all 3 emotions to the surprise class in a very human-like manner. This of course is merely psychological speculation and is beyond the scope of this paper and course.

During testing, it is also found that the model can in fact, detect multiple faces and emotions simultaneously as in figure 17.

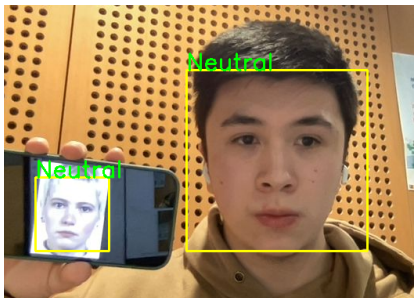

Fig. 17. The model accurately detects the faces and emotions of two subjects simultaneously.

However the same issue arises where the model is unable to identify emotions belonging to the fear and disgust class. The model is heavily biased during classification yet is able to determine the correct label for 5 out of the 7 total emotions.

Conclusion and final remarks

Many psychologists claim that approximately 70-90% of all human communication is done non-verbally through facial expressions and gestures. As demonstrated, this proposed method does allow for a high degree of accuracy in regards to detecting certain facial expressions and then deriving the underlying emotion of the subject. Even when deploying the

model in real-time, the level of accuracy when the subject is in one of the emotion classes such as neutral, happy, angry, surprise or sad is found to be quite high.

The types of applications these methods could be used for range from in healthcare, where these models can be used to detect when nonverbal patients are experiencing pain, to security systems that monitor crowds for suspicious and out of place behaviors. Another use case is in mental health institutions where it could be used to monitor the emotional state of mentally disabled people. A more business oriented application lies in the use of sentiment analysis and detection systems to determine which advertisements to show consumers depending on their emotional state.

In order to further improve the model, more training data belonging to the 2 classes, fear and disgust, which were not identified at all should be added to improve their classification rates. In addition to changing the experimental environment for applying the model in real-time by having more consistent lighting, face positioning and background has a chance of positively improving the model's predictions. Lastly, it is possible that an increase in the depth of the neural network will help the model's understanding of the 2 non-detected classes as it will have more opportunities to detect the more subtle differences between those emotion categories.

Code Links

CNN model training and testing Kaggle notebook link:
https://www.kaggle.com/code/gavinrice/fork-of-emo-detect

Face and emotion detecting program Google Colaboratory notebook link:
https://colab.research.google.com/drive/1ieFdWshiCYUhP82Icq_UEegHsWU3E9Ul?usp=sharing

References

1. Deep Learning by Goodfellow et al.
2. Albawi, Saad, et al. "Understanding of a Convolutional Neural Network." IEEE Xplore, https://ieeexplore.ieee.org/document/8308186.
3. Lorente, Òscar, et al. "Image Classification with Classic and Deep Learning Techniques." *ArXiv.org*, 11 May 2021, https://arxiv.org/abs/2105.04895.
4. Yang, D., et al. "An Emotion Recognition Model Based on Facial Recognition in Virtual Learning Environment." *Procedia Computer Science*, Elsevier, 9 Jan. 2018, https://www.sciencedirect.com/science/article/pii/S1877050917327679.
5. Song, Zhenjie. "Facial Expression Emotion Recognition Model Integrating Philosophy and Machine Learning Theory." *Frontiers*, Frontiers, 1 Jan. 1AD, https://www.frontiersin.org/articles/10.3389/fpsyg.2021.759485/full.
6. Madan, Akshit. *Estimating Heuristic Mulligan Rate for Modern Humans - Youtube*. https://www.youtube.com/watch?v=rEoOhAp0v20.
7. Hassouneh, Aya, et al. "Development of a Real-Time Emotion Recognition System Using Facial Expressions and EEG Based on Machine Learning and Deep Neural Network Methods." *Informatics in Medicine Unlocked*, Elsevier, 12 June 2020, https://www.sciencedirect.com/science/article/pii/S235291482030201X.
8. Skalski, Piotr. "Gentle Dive into Math behind Convolutional Neural Networks." *Medium*, Towards Data Science, 14 Apr. 2019, https://towardsdatascience.com/gentle-dive-into-math-behind-convolutional-neural-networks-79a07dd44cf9.
9. "Categorical Crossentropy Loss Function: Peltarion Platform." *Peltarion*, https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/categorical-crossentropy.
10. Behera, Girija Shankar. "Face Detection with Haar Cascade." *Medium*, Towards Data Science, 29 Dec. 2020, https://towardsdatascience.com/face-detection-with-haar-cascade-727f68