CMP 304
Machine Learning
Unit 2 Report

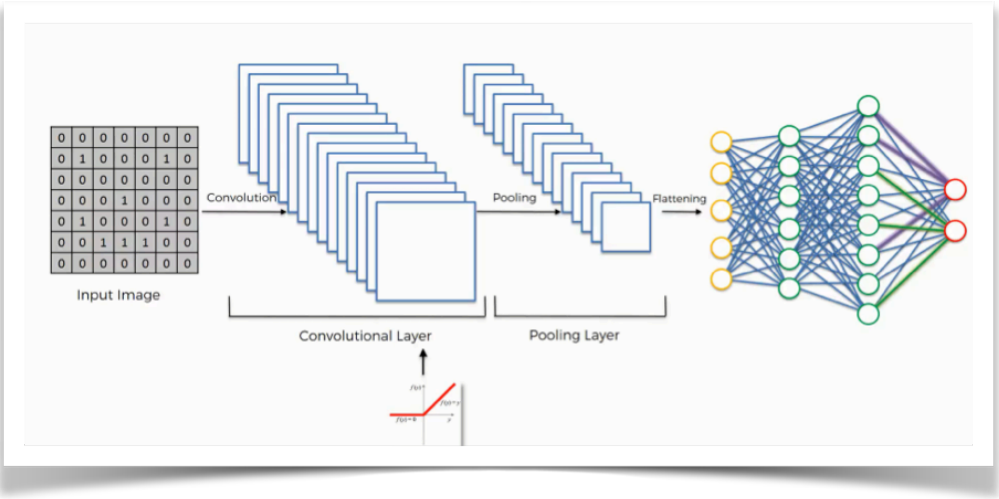
Cameron-Stewart Smart
1901578

Contents

Introduction

The main goal of this project will be to research, design, develop, among other things, a machine learning algorithm that attempts to recognise and categorise the facial expression of a given image as input. Facial expression recognition has a huge amount of use cases that span across many fields. The most obvious being checking the satisfaction of a subject which could be applied to market research or even surveillance systems to check the happiness of employees. Facial expression recognition could also be expanded to recognise individual and collect data about their emotions. Facial recognition should be close guarded by laws to protect individuals as it could have a huge privacy impact by storing vast amounts of data on individuals without their knowledge. For example 'Sesame', a social credit scheme run a Chinese company which is used by the Chinese government and involves collecting big data from an unknown large amount of sources and calculating a credit score based both financial and social decisions has started linking facial recognition data from government surveillance to their system in order to take users real life activity in account when calculating their credit score (Campbell, 2019).

During research for this project the method used by others developing facial expression recognition AI's was investigated using *Google School* and *Papers With Code*. The outcome of this research was that most decided to firstly generate testing and training (validation) batches in order to use them later on to train the models. Most often a CNN (convolutional neural network) model is then created to be used to find specific desired points of interest on the images. Next the model architecture is created and the model is trained. The model is then applied to the images in order to make predictions (Singhal, 2021). The CNN model was the most apparent in research and the model this project will be implementing. It work by using a convolutional layer, a pooling layer, a normalisation layer and a classifier (Kang, 2018). CNN's are a subset of deep learning similar to basic neural networks that allows working with images and videos. They do so by taking an images raw pixel data, training a model and then extracting features automatically for image classification (Badole, 2021).



The above image shows the general structure of the CNN that will be applied to the solution (Badole, 2021). The brief structure of the CNN in this solution will be as follows and will be expanded on later in this report (Tatan, 2019).

Brief CNN Structure

Convolution

A convolution will use a window to sweep through the data, the image, to calculate its input and filter dot product pixel values which will later allow it to emphasise relevant features. This process will create a feature map that emphasises the important features needed to emotional recognition.

Max Pooling

The next stage of the CNN is max pooling, this involves replacing the outputted data from the convolution stage with a max summary to reduce the size of the data and processing time.

Fully Connected Layer

After the outputted feature maps have been through the max pooling stage they will then have the fully connected layer applied to them. This will flatten the inputted maps that will then apply a probability for each emotion that will all add up to one and will generate a most likely emotion.

Methodology and Data

Finding a Data Set

Dataset: https://www.kaggle.com/datasets/sudarshanvaidya/random-images-for-face-emotion-recognition

The initial stage of development for this solution was to find an appropriate dataset. As set out in the introduction the rough outline of how a CNN would be developed was established already. With that in mind an appropriate data set was found using Google researches dataset search tool, the search term used was ‘Emotional Images’. I decided to use the second returned result which provided 8 sets of images for 8 emotions: anger, contempt, disgust, fear, happiness, neutrality, sadness and surprise. There are 5,559 images; so an average of around 700 per emotion which gives me a huge amount to train the model with and each image is already formatted in grayscale and the same size of 224 x 224 pixels in PNG format which reduces the amount of preprocessing required (Vaidya, 2021).

Setting Up Project Files and Imports

For this development the stock Python launcher was used so this was downloaded and set up. The initial project file called ‘Emotional Recognition’ was created and the dataset was downloaded locally.

The following libraries also had to be imported on to my machine using the MacOS terminal command below:

```
python3 -m pip install 'Libary Name'
```

Libraries imported:

- Seaborn for data visualisation
- Numpy to allow for multidimensional arrays
- Tensorflow for machine learning and artificial intelligences
- Matplotlib for plotting graphs

- Livelossplot for plotting graph of loss vs accuracy
- Utils
- OS
- IPython
- Pydot
- Graphviz
- OpenCV

Linking Dataset to Project

To link the dataset to the project dataset was downloaded and moved into the project folder under the test and train folders, inside each of these were folders named after each emotion full of images of that emotion. Next two image data generator classes were created using TensorFlow Kera, one for training and one for testing. These were then passed into the training and validation generators. These generators apply any preprocessing required to the images. All images are converted to grayscale, modified to be uniform size of 48 x 48 and shuffled as shown in Figure 1.

```
dataGeneratorTrain = ImageDataGenerator(horizontal_flip=True) # Flips images horizontally
trainingGenerator = dataGeneratorTrain.flow_from_directory("train/", # Image directory for training
                                                    target_size=(img_size,img_size),
                                                    color_mode="grayscale",
                                                    batch_size=batch_size,
                                                    class_mode='categorical',
                                                    shuffle=True)
```

Figure 1: Image generator for training.

Building the CNN Model

Most approaches to this problem that use a CNN take the approach of applying 4 convolution layers, flattening and adding two fully connected layers to a neural network model so that approach was applied to this project. This

meant firstly creating a sequential neural network model using TensorFlow Keras as shown in Figure 2.

```
model = Sequential() # Setting up the model
```

Figure 2: Setting up sequential model.

The first layer applied is the first convolutional layer. This layer includes 5 stages as shown in Figure 3.

```
# 1st Convolution
model.add(Conv2D(64,(3,3), padding='same', input_shape=(48, 48,1)))
# .add(Conv2D()) is used to specify that a convolutional layer is to be added to model
model.add(BatchNormalization())
# Applies a transformation that normalises model that keeps the mean close to 0 and the standard deviation close to 1
model.add(Activation('relu'))
# Applies rectified linear unit transformation to model to perform non linear transformation
model.add(MaxPooling2D(pool_size=(2, 2)))
# Takes only the maximum data from the previous step with pool 2 x 2
model.add(Dropout(0.25))
# Removes model excess and stops it overfitting. 0.25 mean 1/4 units are dropped
```

Figure 3: First convolution layer.

The first stage of the convolutional is a Conv2D layer which is a 2D convolution layer (e.g. spatial convolution over images). This layer creates a convolution kernel that is convolved with the layer input to produce a tensor of outputs that can have other layers applied to it. Batch normalisation is then added to the layer, this applies a transformation that centres and rescales the layer to reduce internal covariate shift (Wikipedia, 2022). The third stage of layer one applies a Relu activation function. Rectified Linear Unit activation is a simple function that returns 0 if any negative values are given and returns positive values as they are, so this function essentially filters out negative values. The fourth stage applies pooling to the data. This means a filter of 2x2 passes over the data and only returns the max value for each area. Finally a dropout filter is applied. Since 0.25 is passed into this filter every 1/4 inputs are dropped. This first convolution layer forms the base structure for the other 3 convolution layers that were next applied.

The next layer, convolution layer two is shown in Figure 4. This layer is almost identical to the previous but the stride is increased. In layer one we are using a stride of 3x3, which means when we move to the next area affected we are moving 3 in width and height and working on the next 128x128 area but with layer two this is changed to 5x5 which is done to reduce the resolution of the outputted layer.

```
# 2nd Convolution - Kernal size is doubled and strides goes from (3,3) to (5,5)
model.add(Conv2D(128,(5,5), padding='same'))
# .add(Conv2D()) is used to specify that a convolutional layer is to be added to model. Input shape was specified earlier.
model.add(BatchNormalization())
# Applies a transformation that normalises model that keeps the mean close to 0 and the standard deviation close to 1
model.add(Activation('relu'))
# Applies rectified linear unit transformation to model to perform non linear transformation
model.add(MaxPooling2D(pool_size=(2, 2)))
# Takes only the maximum data from the previous step with pool 2 x 2
model.add(Dropout(0.25))
# Removes model excess and stops it overfitting. 0.25 mean 1/4 units are dropped
```

Figure 4: Second convolution layer.

The third layer is almost identical to the first layer too as shown in Figure 5. As is also done in the second layer the kernel size provided within Conv2D is increased as shown in Figure 5. The kernel size is the size of the area being cross-correlated. In layer one this is 64, in layer two this is 128, then 512 for layers three and four. This means a larger area is being used to calculate the next layer of the neural network (Zhang, 2018).

```
# 3rd Convolution - Kernal size is quadrupled and strides goes from (5,5) to (3,3)
model.add(Conv2D(512,(3,3), padding='same'))
# .add(Conv2D()) is used to specify that a convolutional layer is to be added to model. Input shape was specified earlier
model.add(BatchNormalization())
# Applies a transformation that normalises model that keeps the mean close to 0 and the standard deviation close to 1
model.add(Activation('relu'))
# Applies rectified linear unit transformation to model to perform non linear transformation
model.add(MaxPooling2D(pool_size=(2, 2)))
# Takes only the maximum data from the previous step with pool 2 x 2
model.add(Dropout(0.25))
# Removes model excess and stops it overfitting. 0.25 mean 1/4 units are dropped
```

Figure 5: Third convolutional layer.

The fourth convolutional layer is identical to the third. After all four convolutional layers are applied the model is flattened as shown in Figure 6. Flattening is applied to merge all the previous layers into a one dimensional array instead of having them stacked on top of each other in a two dimensional array.

```
model.add(Flatten()) # Converts data into 1 dimensional array creating long feature vector
```

Figure 6: Flattening layers.

After the layers have been flattened the first fully connected layer is applied as shown in Figure 7. This stage begins by adding a dense layer. A dense layer is a deeply connected neural network layer which means each neuron receives input from every neuron in the previous layer and performs matrix-vector multiplication to apply a transformation on the matrix. The size of the output matrix is given as the input, so 256 in the first fully connected layer. In the

same way as the previous layers, batch normalisation, Relu activation and dropout layers are applied to the data before moving on to the second fully connected layer.

```
# 1st Fully connected layer
model.add(Dense(256))
# Adding dense layer which is a regular deeply connected neural network layer
model.add(BatchNormalization())
# Applies a transformation that normalises model that keeps the mean close to 0 and the standard deviation close to 1
model.add(Activation('relu'))
# Applies rectified linear unit transformation to model to perform non linear transformation
model.add(Dropout(0.25))
# Removes model excess and stops it overfitting
```

Figure 7: First fully connected layer.

The final layer is the second fully connected layer. This stage applies another dense layer but has an output size of 512 x 512 instead of the first fully connected layers output size of 256 x 256 as shown in Figure 8.

```
# 2nd Fully connected layer
model.add(Dense(512))
# Adding dense layer which is a regular deeply connected neural network layer
model.add(BatchNormalization())
# Applies a transformation that normalises model that keeps the mean close to 0 and the standard deviation close to 1
model.add(Activation('relu'))
# Applies rectified linear unit transformation to model to perform non linear transformation
model.add(Dropout(0.25))
# Removes model excess and stops it overfitting. 0.25 mean 1/4 units are dropped
model.add(Dense(7, activation='softmax'))
# Converts vector of values to a probability distribution that all sum up to equal one
```

Figure 8: Second full connected layer.

The final stage o developing the model was to compile it. An optimiser was required for tis so an Adam optimiser that keeps the learning rate constant was then added as an input for completion of the model shown in Figure 9.

```
opt = Adam(lr=0.0005)
# Stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments
model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
# Configures and creates the model so it can be used
```

Figure 9: Model compilation

Visualising the Architecture of the Model

In order to visualise the CNN model that has now been created a few Keras functions were called. These outputted a neatly formatted image of the CNN

that is saved to the source folder. The code that performs this is shown in Figure 10 and the outputted image in Figure 11.

```
model.summary()
# Outputs summary of the model
plot_model(model, to_file='model.png', show_shapes=True, show_layer_names=True)
# Prints out a png of model structure
Image('model.png',width=400, height=200)
# Displays image
```

Figure 10: Model summary and image plot / output

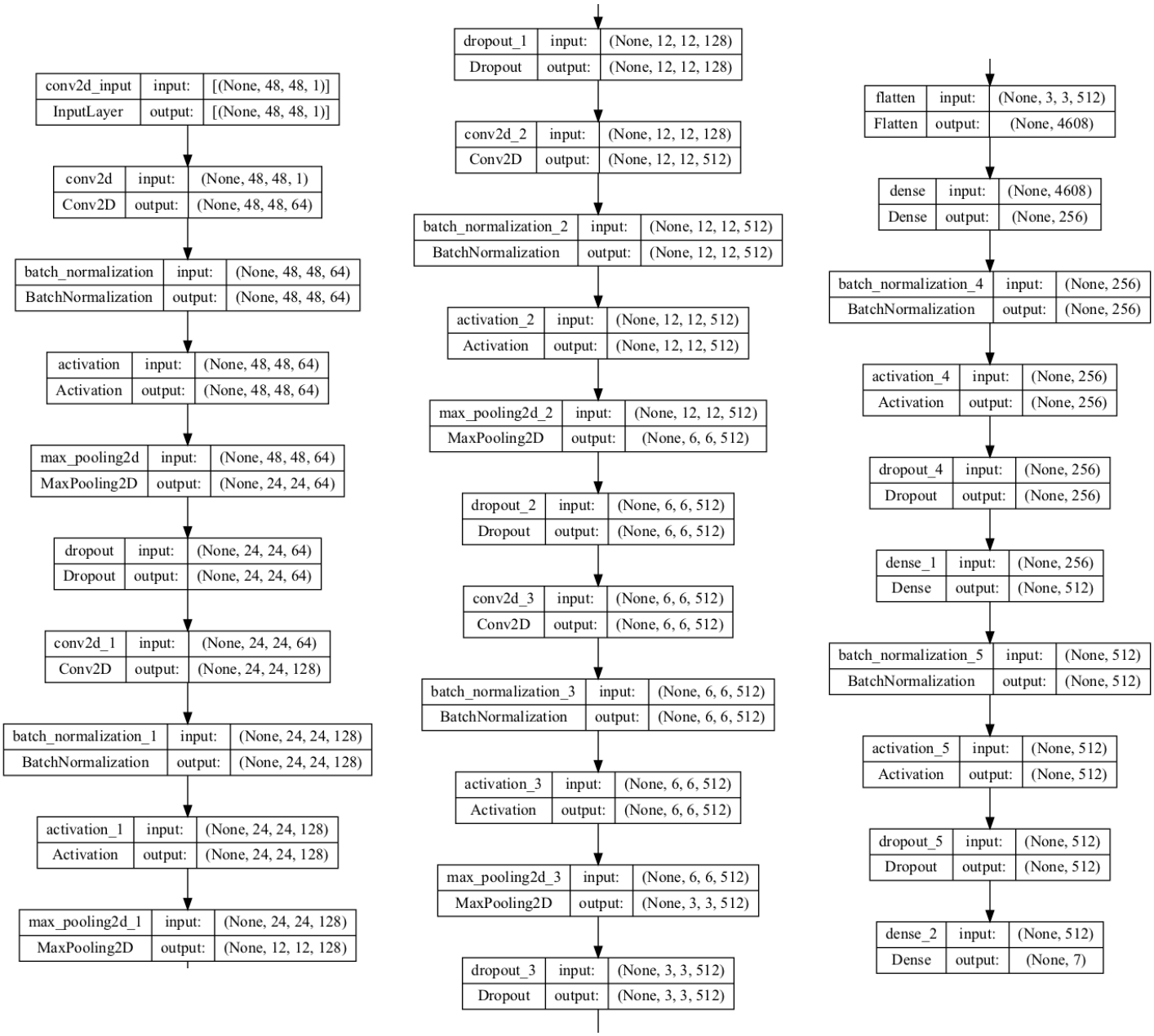


Figure 11: Model summary image split into three. Top to bottom for left, middle and right image.

Training the Model

The next important stage of development was to train the model. This is done by looping over the data in epochs (a full travel forwards and backwards of the training data). Each epoch is broken down in to manageable chunks. The size of these chunks are calculated in Figure 12 by dividing the amount of images loaded in to the training generator by a user set batch size. By default this is set to 16 images per batch. The same is done for the validation generator.

```
stepsPerEpoch = trainingGenerator.n//trainingGenerator.batch_size
validation_steps = validationGenerator.n//validationGenerator.batch_size
```

Figure 12: Calculating amount of steps per epoch.

Next the reduction and checkpoints were defined. The reduction object define when the learning rate should be decreased and the checkpoint object defines where the calculated model weights should be saved to as shown in Figure 13. The callback object is also defined to be fed into the model later. This calls the reduction and checkpoint objects. The callback object also calls a function to plot the graph of losses vs accuracy when a checkpoint is reached so that progress can be tracked.

```
reduction = ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=2, min_lr=0.00001, mode='auto')
checkpoint = ModelCheckpoint("model_weights.h5", monitor='val_accuracy', save_weights_only=True, mode='max', verbose=1)
callbacks = [PlotLossesCallback(), checkpoint, reduction]
```

Figure 13: Defining reduction, checkpoint and callback objects.

To carry out model training the Keras fit function is used. This function trains a model using the model, the training generator, the epochs, the steps per epoch, the validation generator, the validation steps and the callback object as inputs as shown in Figure 14.

```
history = model.fit( # Starts training a validation of model
    x=trainingGenerator, # Training dataset declared earlier as 'train/'
    stepsPerEpoch=stepsPerEpoch, # Calculated earlier by dividing number of images by batch size
    epochs=epochs, # Number of full passes back and forth
    validation_data = validationGenerator, ## Validaton dataset declared earlier as 'test/'
    validation_steps = validation_steps, # Calculated earlier by dividing numberof images by batch size
    callbacks=callbacks
)
```

Figure 14: The fit function uses to train the model to the training data.

To complete the training the models and the model weights are outputted to JSON to be called on later or reused s shown in Figure 15.

```
model_json = model.to_json() # Model is converted to JSON so it can used to make predictons
model.save_weights('model_weights.h5') # Weights for model are saved to different file

with open("model.json", "w") as JsonFile:
    JsonFile.write(model_json)
```

Figure 15: Writing JSON files.

Loading the Model Weight and Making Predictions

At this stage the goal is now to use the generated and trained model to perform predictions. This will be done on two data sources, videos created of emotions being displayed and live webcam feed. For this task OpenCV2 is used. Firstly a class has to be created that can be used by the the OpenCV2library to apply the model created earlier to the solution as shown in Figure 16. This model has the functions '__init__' (must be called this to work with OpenCV) and 'predictEmotion'. The initialise function takes the JSON files for the model and the JSON file for the weights. It then uses the OpenCV functions to load the weights and the model into itself. The predictEmotion function is then set up to take in an image and a prediction using the model and return a prediction.

```
class FacialExpressions(object):
    emotions = ["Anger", "Disgust",
                "Fear", "Happiness",
                "Neutrality", "Sadness",
                "Surprise"]
    def __init__(self, modelJsonFile, modelWeightsFile):
        # load model from JSON file
        with open(modelJsonFile, "r") as JsonFile:
            loadedModelJson = JsonFile.read()
            self.loadedModel = model_from_json(loadedModelJson)
        # load weights into the new model
        self.loadedModel.load_weights(modelWeightsFile)
        self.loadedModel.make_predict_function()
    # Predict emotions from a given image
    def predictEmotion(self, img):
        self.predictions = self.loadedModel.predict(img)
        return FacialExpressions.emotions[np.argmax(self.predictions)]
```

Figure 16: FacialExpressions class.

Before the model can be used to make predictions a cascade classifier must be fed into OpenCV. This classifier is used to defined the points on a face. For this the Haar Cascade Classifiers library of XML files (Pisarevsky, 2013) was loaded in to the project folder and called into OpenCV. An object of class FacialExpressions is then created and named model as shown in Figure 16.

```
face = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
model = FacialExpressions("model.json", "model_weights.h5")
```

Figure 16: Loading in classier and creating model object.

Getting the Frames of the Video for Evaluation

To test out the model an image must be retrieved. In this case we are using the frames coming in from the webcam or a given video file. As shown in Figure 17 the object self.video is set to OpenCVs video capture function. If the input parameters to this is set to 0 the webcam will be used, otherwise a file can be named as done in Figure 17. A function is also created inside the object to release the video feed. The function used to get frames is also declare. This function reads in the video as a frame. The frame is then preprocessed before it can have the predictor applied. The frame is adapted to be grayscale using OpenCV again. OpenCV then crops out only detected faces using the detectMultiScale function on the grayscale image. This image is then resized to be the same size as the images used for training, 48 x 48 so that the model works best. The predictEmotion function from earlier is finally applied and the OpenCV function to print text over the face with the predicted emotion is called. Finally a square is drawn around the face detected and the frame is returned to be displayed.

```
class VideoCamera(object):
    def __init__(self):
        self.video = cv2.VideoCapture('sample2.mp4') # Loads in video capture frame, change to file to predict on existing video
    def __del__(self):
        self.video.release() # Deletes loaded in video capture frame
    def getFrame(self): # Creates predictions
        _, frame = self.video.read()
        grayFrame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY) # Converts image frame to grayscale
        faces = face.detectMultiScale(grayFrame, 1.3, 5) # Detects face
        for (x, y, w, h) in faces:
            frameContent = grayFrame[y:y+h, x:x+w] # Make frame grayscale to match training images
            resized = cv2.resize(frameContent, (48, 48)) # Resize frame of face to match training images that model is used to
            prediction = model.predictEmotion(resized[np.newaxis, :, :, np.newaxis]) # Line that actually gets predicted emotion
            cv2.putText(frame, prediction, (x, y), font, 1, (255, 255, 0), 2) # Overlays prediction on video view
            cv2.rectangle(frame,(x,y),(x+w,y+h),(255,0,0),2) # Overlays rectangle around face
        return frame
```

Figure 17: VideoCamera class to load in image for prediction

Running above code

The final step in the code is the camera function shown in Figure 18. This function runs an infinite loop that until broken uses the getFrame function from Figure 17 to make predictions on the image and then output this using OpenCVs image show function.

```
def gen(camera): # If camera is being used to test that generate camera function
    while True:
        frame = camera.getFrame()
        cv2.imshow('Emotion Recognition',frame)
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break
    cv2.destroyAllWindows()

gen(VideoCamera())
```

Figure 18: gen(Camera) function.

Results

To begin testing the solution first a test of 1 epoch over the data was run. With this an accuracy of 31.32% with a loss of 1.7884 is achieved as shown from Figure 19.

```
448/448 [==============================] - ETA: 0s - loss: 1.7884 - accuracy: 0.3132
```

Figure 19: Output from program after all epochs have been completed.

To better test the solution the program was set to run 18 epochs over the data. As shown in Figure 20 you can see as each epoch passes the accuracy of the model increases and the loss decreases. As shown the begins to reach a plateau. From the graph you could estimated that the plateau would be almost reached at around 25 epochs. Each epoch was split into 448 segments takes around 4.75 minutes running on an i7-9750H with 16GB of RAM so its quite a time intensive process.

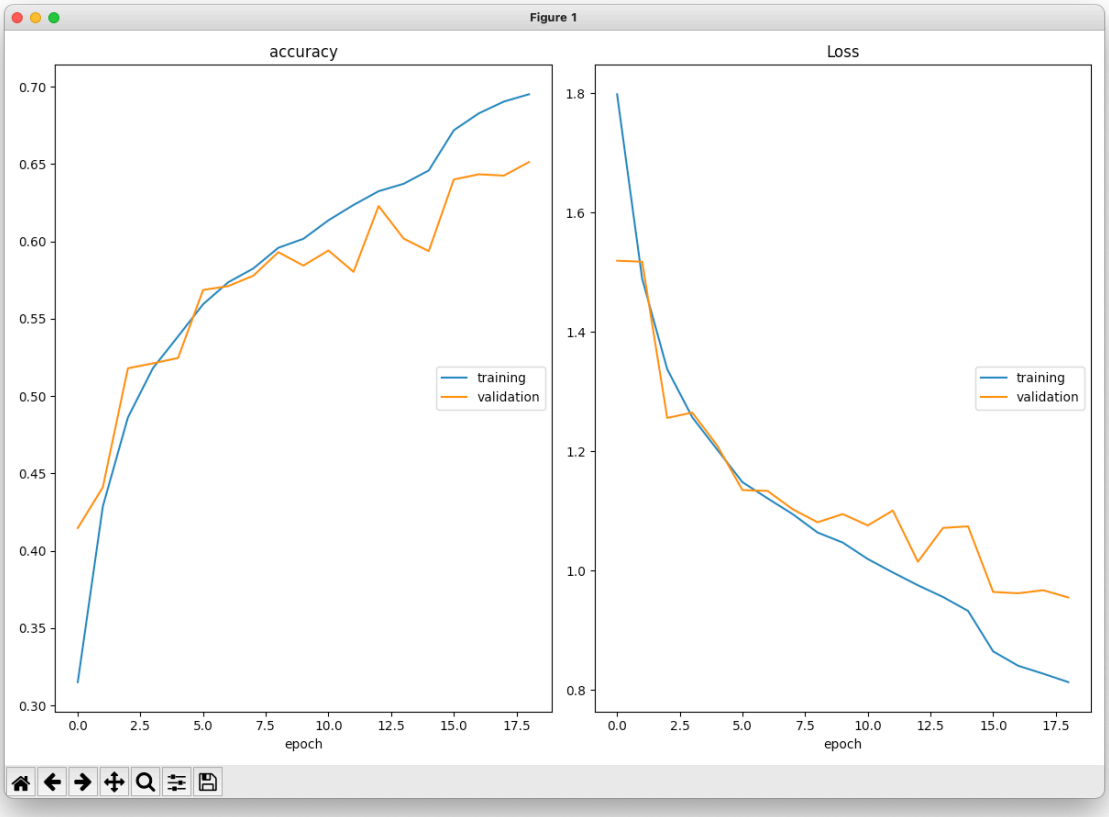


Figure 20: Accuracy vs Epochs and Loss vs Epochs.

Discussion and Conclusion

The main goal of the project was to create a program that could correctly identify the emotions of a person in a given video, webcam feed or photo. This was done by creating a convolutional neural network machine learning model that contained 4 convolution layers and then two fully connected layers. This model is trained and validated using the provided dataset. The model is saved in JSON then recalled and applied to the frames of the input. The OpenCV library is used to apply this and then display the results.

In future this solution could be approved by using a larger selection of training data with more epochs over the data. This would most likely increase the accuracy that the model plateaus at.

Overall this solution accomplished the goal of creating a machine learning model that effectively recognises emotions on faces by using convolutional neural network and the openCV2 library. This was done effectively as shown in the results, the model can go well above 60% accuracy.

References

Campbell, Charlie (2019) ‘How China Is Using “Social Credit Scores” to Reward and Punish Its Citizens’, Time Magazine. Available at: https://time.com/collection/davos-2019/5502592/china-social-credit-score/ (Accessed: 19/05/22)

Singhal, Aaditya (2021) ‘Facial expression detection using Machine Learning in Python’, Medium. Available at: https://medium.com/analytics-vidhya/facial-expression-detection-using-machine-learning-in-python-c6a188ac765f (Accessed: 18/05/22)

Kang, Min Soo (2018) ‘A Study on the Facial Expression Recognition using Deep Learning Technique’, Korea Science. Available at: https://www.koreascience.or.kr/article/JAKO201818564286759.page (Accessed: 19/05/22)

Badole, Mayur (2021) ‘Create CNN Model and Optimize Using Keras Tuner – Deep Learning’, Analytics Vidhya. Available at https://www.analyticsvidhya.com/blog/2021/06/create-convolutional-neural-network-model-and-optimize-using-keras-tuner-deep-learning/ (Accessed: 19/05/22)

Tatan, Vincent (2019) ‘Understanding CNN (Convolutional Neural Network)’, Towards Data Science. Available at: https://towardsdatascience.com/understanding-cnn-convolutional-neural-network-69fd626ee7d4 (Accessed: 19/05/22)

Vaidya, Sudarshan (2021) ‘Natural Human Face Images for Emotion Recognition’, Kaggle. Available at: https://www.kaggle.com/datasets/sudarshanvaidya/random-images-for-face-emotion-recognition?resource=download (Accessed: 18/05/22)

Wikipedia (2022) Batch Normalisation’. Available at: https://en.wikipedia.org/wiki/Batch_normalization (Accessed: 20/05/22)

Zhang, Lerner (2018) ‘What does kernel size mean?’, StackExchange. Available at: https://stats.stackexchange.com/questions/296679/what-does-kernel-size-mean (Accessed: 20/05/22)

Pisarevsky, Vadim (2013) ‘Haar Cascade Library’, GitHub. Available at: https://github.com/opencv/opencv/blob/master/data/haarcascades/haarcascade_frontalface_default.xml (Accessed: 17/05/22)