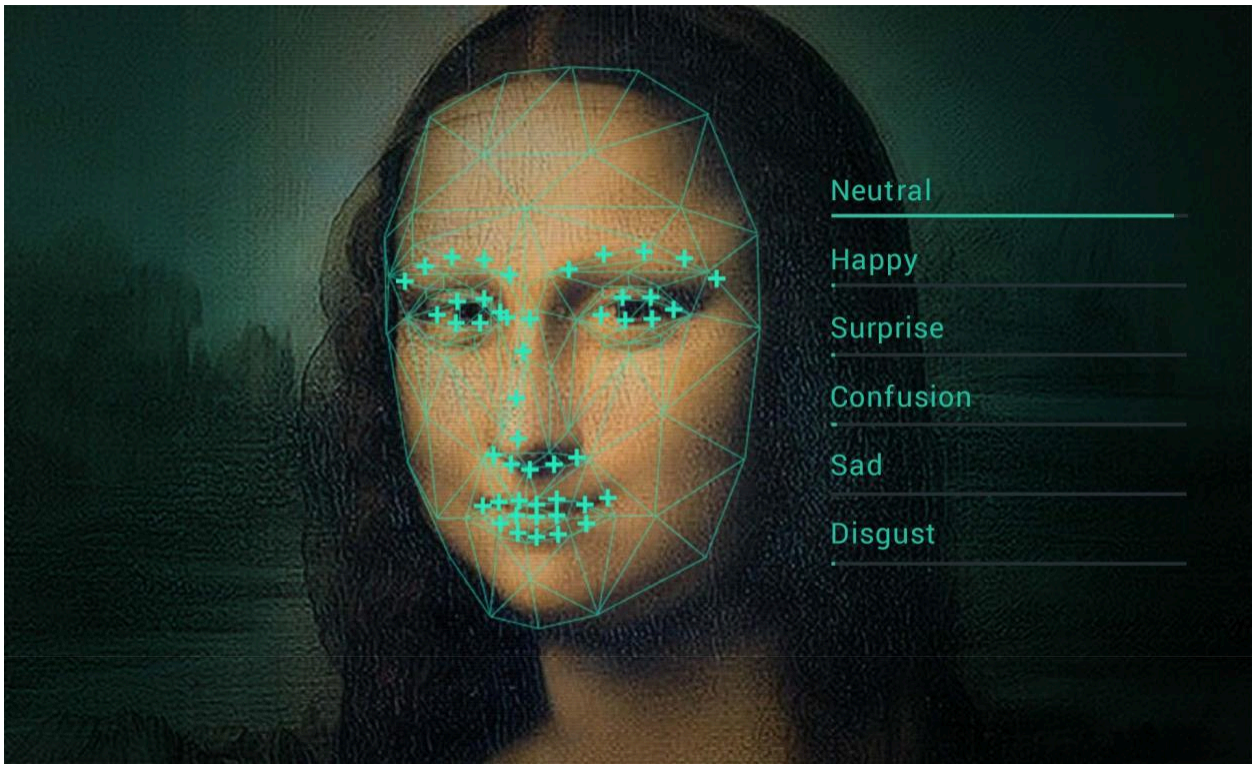


Emotion Detection Project

Phase 2

Ahmadreza Enayati

Mohammad Shah Abadi

Erfan Parifard

Table of contents

What is Emotion detection?

Emotion detection, a pivotal facet of affective computing, stands at the intersection of technology and human psychology, offering profound insights into understanding and interpreting human emotions. In the modern digital landscape, where human-computer interaction plays an increasingly integral role, the ability to discern and respond to human emotions has emerged as a critical area of research and application.

The project at hand delves into the realm of emotion detection, endeavoring to harness the power of machine learning and computer vision to discern and classify human emotions from visual cues. Through meticulous analysis and preprocessing of image data, the project aims to equip machines with the capability to recognize subtle facial expressions and infer underlying emotional states.

The significance of such endeavors reverberates across diverse domains, ranging from healthcare and education to marketing and human-computer interaction. In healthcare, emotion detection holds promise for aiding in the diagnosis and monitoring of mental health disorders. In education, it can facilitate personalized learning experiences tailored to individual emotional responses. Moreover, in marketing and advertising, it offers the potential to gauge consumer sentiment and tailor strategies accordingly, fostering more effective communication and engagement.

By embarking on this journey, we not only advance the frontiers of technology but also deepen our understanding of human emotions and the nuanced ways in which they manifest. As we navigate through the intricacies of emotion detection, we embark on a quest to imbue machines with a semblance of emotional intelligence, paving the way for more empathetic and responsive human-computer interactions.

Libraries utilized

In the pursuit of developing an efficient and robust emotion detection system, a suite of libraries and tools were employed to facilitate various aspects of the project. Leveraging the capabilities of these libraries streamlined the implementation process and enhanced the functionality of the codebase.

Brief explanation of libraries

Here's a concise introduction to the libraries used in the project:

1. **OpenCV (Open Source Computer Vision Library):**
    - ➢ OpenCV is a popular open-source library primarily focused on computer vision and image processing tasks.
    - ➢ It provides a wide range of functionalities for tasks such as image manipulation, feature detection, object recognition, and more.
    - ➢ OpenCV is widely used in various applications, including robotics, augmented reality, facial recognition, and surveillance systems.
2. **NumPy (Numerical Python):**
    - ➢ NumPy is a fundamental library for numerical computing in Python.
    - ➢ It offers support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays efficiently.

➢ NumPy is extensively used in scientific computing, data analysis, machine learning, and other domains for its performance and versatility.

3. **Pandas:**

➢ Pandas is a powerful library for data manipulation and analysis in Python

➢ It provides data structures such as DataFrame and Series, which offer convenient and efficient methods for handling structured data.

➢ Pandas simplifies tasks like data cleaning, transformation, aggregation, and visualization, making it a popular choice for data-centric projects and data science workflows.

4. **TensorFlow**:

TensorFlow is a powerful open-source machine learning library developed by Google Brain. It provides a comprehensive set of tools and resources for building and training deep learning models. TensorFlow is designed to be flexible and scalable, making it suitable for a wide range of applications, including image classification, natural language processing, and speech recognition.

Key Features of TensorFlow:

- **Flexibility:** TensorFlow allows for the creation of complex neural network architectures, enabling the development of sophisticated machine learning models.
- **Scalability:** TensorFlow can be used to train models on large datasets, making it suitable for real-world applications.
- **Efficiency:** TensorFlow optimizes the execution of machine learning models, ensuring efficient training and inference.
- **Accessibility:** TensorFlow provides a user-friendly interface and extensive documentation, making it accessible to both beginners and experienced machine learning practitioners.

TensorFlow has gained popularity due to its versatility and ability to handle complex machine learning tasks. It is widely used in various fields, including computer vision, natural language processing, and robotics.

Usage in code

1. **OpenCV (Open Source Computer Vision Library):**

   - ➢ OpenCV played a pivotal role in the pre-processing pipeline by providing essential functionalities for image manipulation and computer vision tasks.
   - ➢ Specifically, the Haar Cascade classifier implemented in OpenCV enabled the detection of facial features within images, a crucial step in isolating and extracting facial regions.

2. **NumPy (Numerical Python):**

   - ➢ NumPy emerged as a fundamental library for numerical computing and array manipulation in Python.
   - ➢ Its efficient array operations facilitated the manipulation and transformation of image pixel data, enabling seamless integration with OpenCV and other components of the project.

3. **Pandas:**

   - ➢ Pandas, a powerful data manipulation library, was employed for handling and processing tabular data structures, particularly CSV files.
   - ➢ It facilitated the organization and management of pre-processed image data, enabling efficient aggregation and preparation of training and testing datasets.

By harnessing the capabilities of these libraries, the project benefitted from a robust foundation for image processing, data manipulation, and file management. This amalgamation of powerful tools not only expedited the development process but also contributed to the overall efficiency and effectiveness of the emotion detection system.

4. **TensorFlow:**

- Import TensorFlow as tf.
- Load the pre-trained TensorFlow model from 'emotion_detection_model.h5'.
- Define a function called that takes an image as an input.
    - Preprocess the image by resizing it to (224, 224) and applying VGG16 preprocessing.
    - Make a prediction using the pre-trained model.
    - Return the emotion with the highest probability.
- Load an image using .
- Predict the emotion using the function.
- Print the predicted emotion.

Pre processing TensorFlow phase flow

In the pre-processing phase of the emotion detection project, several steps are undertaken to ensure the quality and homogeneity of the dataset. This phase is crucial as the quality of the data significantly impacts the implementation and performance of the project. The following steps are typically involved:

1. **Data Quality Check:**
    - ➢ The collected dataset is examined to ensure the homogeneity of input information.
    - ➢ Quality assurance measures are implemented to identify and address any inconsistencies or anomalies in the dataset.

2. **Face Detection and Cropping:**
    - ➢ The dataset is processed to include only images containing people's faces.
    - ➢ Techniques such as Haar Cascade classifers are commonly used to detect faces within images.
    - ➢ Detected faces are cropped to focus only on the facial region, removing any irrelevant background or body parts.
    - ➢ Next, we apply padding to ensure all images are uniform in size.
3. **Output Directory Creation:**
    - ➢ Result directories are created to organize the pre-processed data.
    - ➢ Separate directories are allocated for each emotion label, along with a directory for images where faces were not detected.

4. **Data Writing:**
    - ➢ Cropped face images are written to the appropriate emotion-labeled directories, along with their corresponding pixel data stored in CSV fles.
    - ➢ Images where faces were not detected are also stored in a designated directory for further analysis.
5. **Dataset Preparation:**
    - ➢ The pre-processed data is aggregated into separate training and testing datasets, facilitating the subsequent model training and evaluation processes.

6. **Model Creation**:

- Define the model architecture.
- Initialize the model weights.
- Compile the model with the appropriate loss function, optimizer, and metrics.

7. Model Training and Evaluation

- Define the training and validation datasets.
- Specify the batch size.
- Set the number of epochs for training.
- Train the model using the defined datasets, batch size, and epochs.
- Evaluate the trained model using the validation dataset.
- Calculate the accuracy and loss metrics.
- Adjust the model parameters and repeat the training and evaluation process if necessary.
- Save the trained model for future use.

Detailed code explanation

Overall, this code serves as the main driver for executing the pre-processing pipeline, which is crucial for preparing the dataset for subsequent stages of the emotion detection project.

In this piece of code we do the primary expectation of this phase. We just iterate through all the dataset and call `face_detection_with_padding` function on each one.

```
from helper import write_to_labeled_folder, write_to_removed_folder

face_not_detected_counter = 0
face_detected_counter = 0
resized_faces = []
emotions = []
for index, row in data.iterrows():
    emotion = row['emotion']
    pixels = [int(p) for p in row['pixels'].split()]
    image = np.array(pixels, dtype=np.uint8).reshape(48, 48)

    face = face_detection_with_padding(image)
    if face is not None:
        face_detected_counter += 1
        write_to_labeled_folder(emotion, face, image)
        resized_faces.append(face)
        emotions.append(emotion)
    else:
        write_to_removed_folder(emotion, image)
        face_not_detected_counter += 1

    if index % 500 == 0:
        print(index, ' images processed')

print(f"Faces detected: {face_detected_counter}")
print(f"Faces not detected: {face_not_detected_counter}")

emotions_pd = pd.Series(emotions)
print(f"\nEmotions occurrence: \n{emotions_pd.value_counts()}")
```

If the face has been detected it will be appended to a .csv file in the relevant directory or else it will be moved into a folder dedicated to removed data (also a csv file).

And here's the result of the code:

```
Faces detected: 24922
Faces not detected: 10965

Emotions occurrence:
3    6661
6    4761
4    3394
2    3326
0    3272
5    3126
1     382
```

As we see in the image above code detects 24922 images and shows the occurrence of each emotion in the detected faces with their labels.Code below shows the implementation of `face_detection_with_padding` function:

```
def face_detection_with_padding(image, target_size=(48, 48), padding_color=(0, 0, 0)):
    faces = face_cascade.detectMultiScale(image, scaleFactor=1.01, minNeighbors=5, minSize=(30, 30))

    if len(faces) == 0:
        return None

    x, y, w, h = faces[0]
    cropped_face = image[y:y + h, x:x + w]

    top_pad = (target_size[0] - cropped_face.shape[0]) // 2
    bottom_pad = target_size[0] - cropped_face.shape[0] - top_pad
    left_pad = (target_size[1] - cropped_face.shape[1]) // 2
    right_pad = target_size[1] - cropped_face.shape[1] - left_pad

    padded_image = cv2.copyMakeBorder(cropped_face, top_pad, bottom_pad, left_pad, right_pad,
cv2.BORDER_CONSTANT, value=padding_color)

    return padded_image
```

This function is useful for ensuring that the cropped face maintains its aspect ratio and is resized to a predefined size while preserving the original spatial relationship of facial features.

1. **Face Detection:**
    - The function starts by detecting faces in the input image using a pre-trained Haar Cascade classifer.
    - The `detectMultiScale` function identifes regions in the image that may contain faces based on certain parameters such as scale factor, minimum neighbors, and minimum size.
2. **Check for Detected Faces:**
    - If no faces are detected in the image (i.e., the length of the `faces` list is 0), the function returns `None` to indicate that no face was found.
3. **Crop the Detected Face:**
    - If a face is detected, the function extracts the coordinates and dimensions of the frst detected face (assumed to be the most prominent face).
    - It then crops the detected face from the original image based on these coordinates and dimensions.
4. **Calculate Padding:**
    - The function calculates the amount of padding required to resize the cropped face to the target size specifed by the `target_size` parameter.
    - It computes the padding values for the top, bottom, left, and right sides of the cropped face.
5. **Apply Padding:**
    - Using the calculated padding values, the function applies padding to the cropped face to resize it to the target size.
    - The `copyMakeBorder` function from OpenCV is used to add the specifed amount of padding around the cropped face.
    - The `cv2.BORDER_CONSTANT` fag indicates that the padding should be flled with a constant color, which is specifed by the `padding_color` parameter.
6. **Return Padded Image:**
    - The function returns the padded image, which now has the dimensions specifed by `target_size` and is centered within the image with padding added around it.

Chosen algorithm

Introduction to CNN algorithm

CNN can stand for various things depending on the context, but in the realm of technology and media, it commonly refers to "Convolutional Neural Network."

A Convolutional Neural Network (CNN) is a type of artificial neural network that is primarily used in analyzing visual imagery. It is a class of deep neural networks, most commonly applied to analyzing visual imagery. CNNs have proven very effective in tasks such as image recognition and classification.

The architecture of a CNN is designed to automatically and adaptively learn spatial hierarchies of features from input data, making them well-suited for tasks such as object recognition in images. They consist of multiple layers of small neuron collections that process portions of the input image, called receptive fields. These layers include convolutional layers, pooling layers, and fully connected layers, and they are typically followed by normalization layers and activation functions.

CNNs have been transformative in various fields, including computer vision, image and video recognition, medical image analysis, and autonomous vehicle technology.

Why CNN?

Overall, the ability of CNNs to learn discriminative features from visual data, coupled with their robustness and efciency, makes them well-suited for emotion detection tasks, particularly in analyzing facial expressions

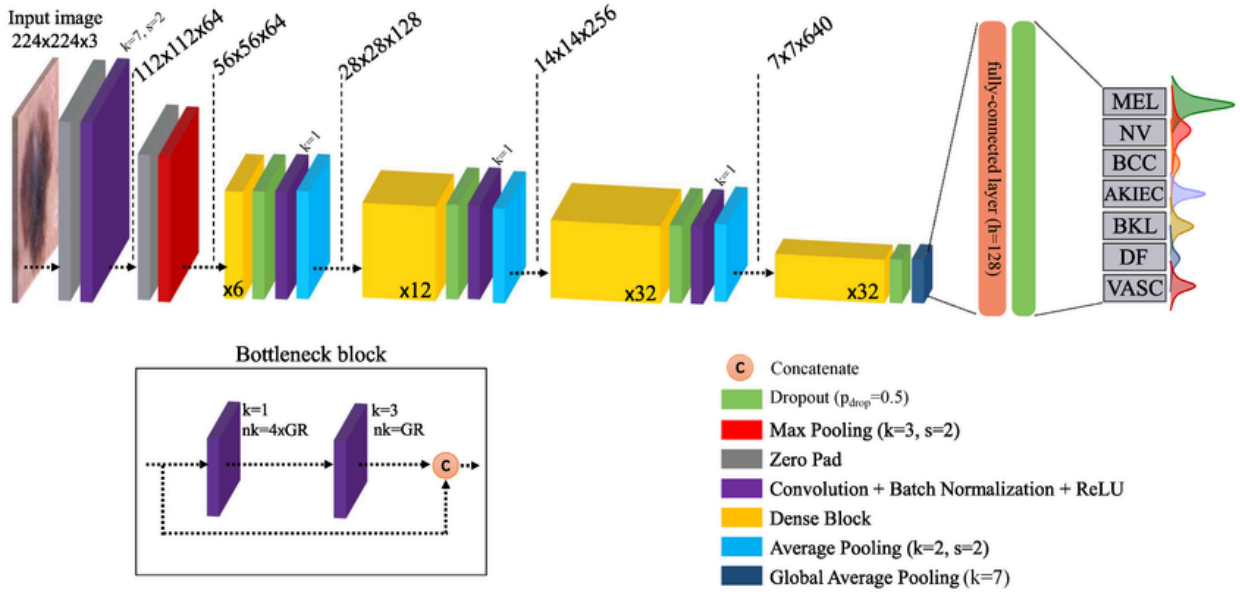
CNNs are well-suited for emotion detection for several reasons:

1. **Feature Extraction:**
    - ➢ CNNs excel at automatically learning hierarchical features from raw pixel data.
    - ➢ In emotion detection, CNNs can effectively extract discriminative features from facial images, capturing nuanced patterns and expressions indicative of different emotions.
2. **Translation Invariance:**
    - ➢ CNNs exhibit translation invariance, meaning they can detect features regardless of their position in the image.
    - ➢ This property is crucial for emotion detection, as emotions may manifest differently across various facial expressions and positions within an image.
3. **Robustness to Variations:**
    - ➢ CNNs are robust to variations in lighting, facial orientation, and other factors commonly encountered in real-world images.
    - ➢ This robustness ensures consistent performance across diverse datasets and environments, enhancing the reliability of emotion detection systems.
4. **Scalability and Generalization:**
    - ➢ CNNs can scale to handle large datasets and complex image inputs, making them suitable for real-world applications with varied data distributions.
    - ➢ Moreover, CNNs have demonstrated strong generalization capabilities, effectively learning from limited labeled data to classify unseen images accurately.

Model

Densenet 169 Transfer Learning using Keras:

- Import necessary libraries
- Define image dimensions and number of classes
- Define feature extractor using DenseNet169 pre-trained model
- Define classifier with dense layers and dropout
- Define final model by combining feature extractor and classifier
- Define model compilation with optimizer, loss function, and accuracy metric
- Create model input and output
- Compile the model
- Print model summary



Densenet 169

DenseNet169 is a convolutional neural network (CNN) architecture developed by Gao Huang, Zhuang Liu, and Kilian Q. Weinberger in 2017. It is a member of the DenseNet family of models, which are characterized by their dense connectivity between layers.

DenseNet169 is a deep neural network with 169 layers, including convolutional layers, pooling layers, and fully connected layers. The dense connectivity in DenseNet169 means that each layer is connected to every other layer in a feed-forward fashion, allowing for better information flow and gradient propagation throughout the network.

One of the key advantages of DenseNet169 is its ability to alleviate the vanishing gradient problem, which can occur in deep neural networks when gradients become too small to propagate effectively through the network. By connecting each layer to every other layer, DenseNet169 ensures that gradients can flow more easily, leading to faster and more efficient training.

DenseNet169 has been shown to achieve state-of-the-art results on various image classification tasks, including the ImageNet Large Scale Visual Recognition Challenge (ILSVRC). It has also been successfully applied to other tasks such as object detection, semantic segmentation, and medical imaging.

Here are some of the reasons why DenseNet169 is a popular choice for deep learning tasks:

- **Dense connectivity:** The dense connectivity in DenseNet169 allows for better information flow and gradient propagation, leading to faster and more efficient training.
- **Reduced vanishing gradient problem:** DenseNet169's dense connectivity helps to alleviate the vanishing gradient problem, which can occur in deep neural networks when gradients become too small to propagate effectively through the network.
- **State-of-the-art performance:** DenseNet169 has been shown to achieve state-of-the-art results on various image classification tasks, including the ImageNet Large Scale Visual Recognition Challenge (ILSVRC).
- **Versatility:** DenseNet169 has been successfully applied to a wide range of tasks beyond image classification, such as object detection, semantic segmentation, and medical imaging.

Overall, DenseNet169 is a powerful and versatile deep learning model that has been shown to achieve excellent results on a wide range of tasks. Its dense connectivity, reduced vanishing gradient problem, and state-of-the-art performance make it a popular choice for deep learning practitioners.

The provided code snippet outlines a neural network architecture used for classification tasks, particularly leveraging transfer learning. Here's an explanation of each component:

Transfer Learning

The technique of leveraging a pre-trained model (DenseNet in this case) on a new, but related task. This approach is beneficial when you have limited data or computational resources, as the pre-trained model has already learned useful features from a large dataset.

1. **Feature Extraction using DenseNet**:
   - **DenseNet**: A Dense Convolutional Network (DenseNet) is a type of convolutional neural network (CNN) known for its dense connections between layers, enhancing the flow of information and gradients through the network. In this context, DenseNet is used as a pre-trained feature extractor.

2. **Global Average Pooling Layer**:
   This layer reduces each feature map to a single value by computing the average of all elements in the feature map. This operation reduces the spatial dimensions of the feature maps, producing a fixed-length output vector regardless of the input size, making it suitable for connecting to fully connected layers.

3. **First Dense Block**:
   - Dense Layer: A fully connected layer with 256 neurons, ReLU activation function, and L2 regularization (penalizing large weights to prevent overfitting).
   - Dropout Layer: Regularization technique where 30% of the neurons are randomly set to zero during training to prevent overfitting and improve generalization.

4. **Second Dense Block**:
   - Dense Layer: Fully connected layer with 1024 neurons, ReLU activation, and L2 regularization.
   - Dropout Layer: 50% dropout to prevent overfitting.

5. **Third Dense Block**:
   - Dense Layer: Fully connected layer with 512 neurons, ReLU activation, and L2 regularization.
   - Dropout Layer: 50% dropout.

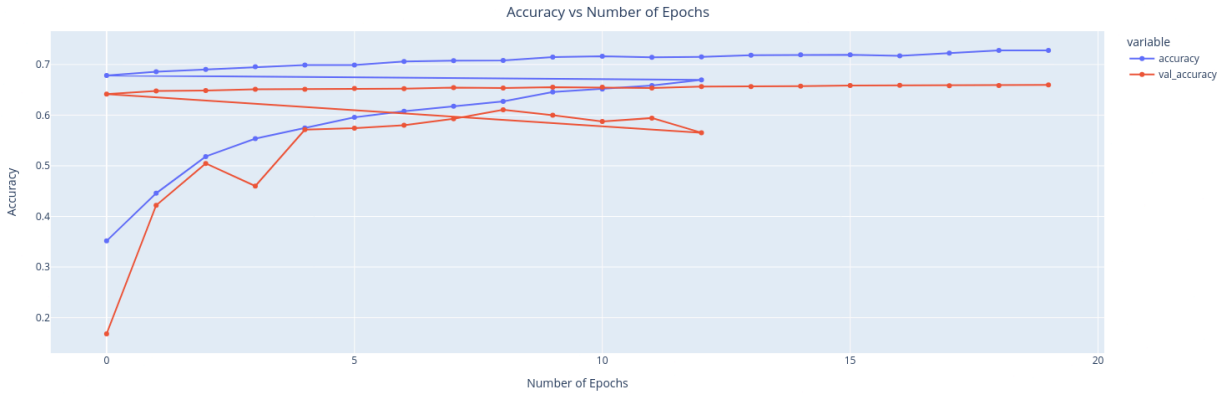
6. **Final Classification Layer**:
   - Dense Layer: The final layer with a number of neurons equal to the number of classes in the classification problem.
   - Softmax Activation: Converts the output scores into probabilities, making it suitable for multi-class classification.

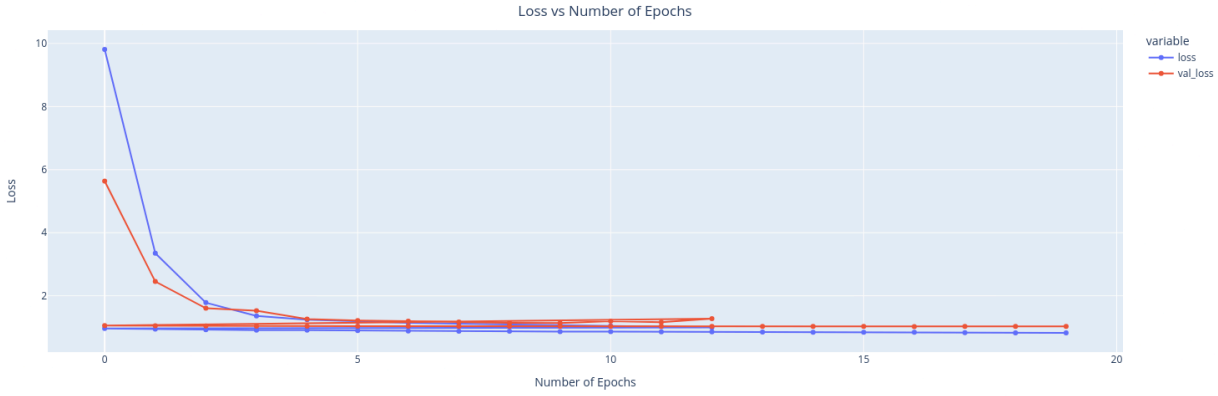
```python
def feature_extractor(inputs):
    feature_extractor = tf.keras.applications.DenseNet169(input_shape=(IMG_HEIGHT,IMG_WIDTH, 3),
                                                include_top=False,
                                                weights="imagenet")(inputs)

    return feature_extractor

def classifier(inputs):
    x = tf.keras.layers.GlobalAveragePooling2D()(inputs)
    x = tf.keras.layers.Dense(256, activation="relu", kernel_regularizer = tf.keras.regularizers.l2(0.01))(x)
    x = tf.keras.layers.Dropout(0.3)(x)
    x = tf.keras.layers.Dense(1024, activation="relu", kernel_regularizer = tf.keras.regularizers.l2(0.01))(x)
    x = tf.keras.layers.Dropout(0.5)(x)
    x = tf.keras.layers.Dense(512, activation="relu", kernel_regularizer = tf.keras.regularizers.l2(0.01))(x)
    x = tf.keras.layers.Dropout(0.5) (x)
    x = tf.keras.layers.Dense(NUM_CLASSES, activation="softmax", name="classification")(x)

    return x

def final_model(inputs):
    densenet_feature_extractor = feature_extractor(inputs)
    classification_output = classifier(densenet_feature_extractor)

    return classification_output

def define_compile_model():

    inputs = tf.keras.layers.Input(shape=(IMG_HEIGHT ,IMG_WIDTH,3))
    classification_output = final_model(inputs)
    model = tf.keras.Model(inputs=inputs, outputs = classification_output)

    model.compile(optimizer=tf.keras.optimizers.SGD(0.1),
                loss='categorical_crossentropy',
                metrics = ['accuracy'])

    return model
```

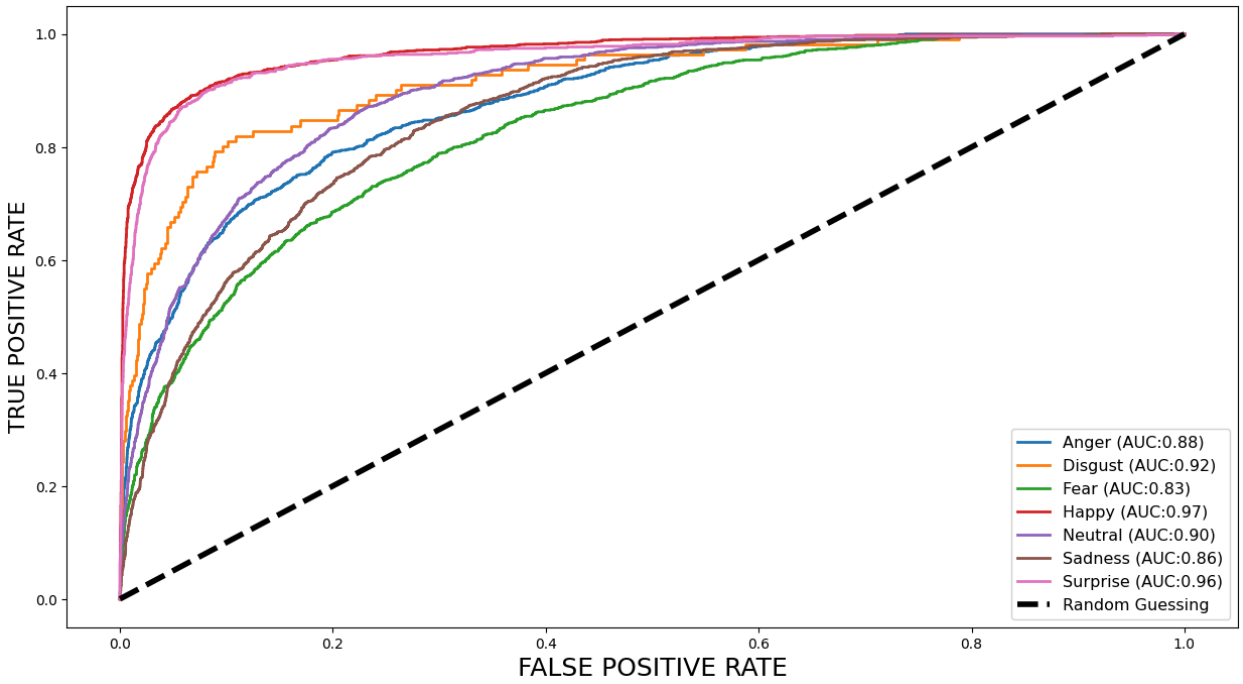
Fine Tuning:

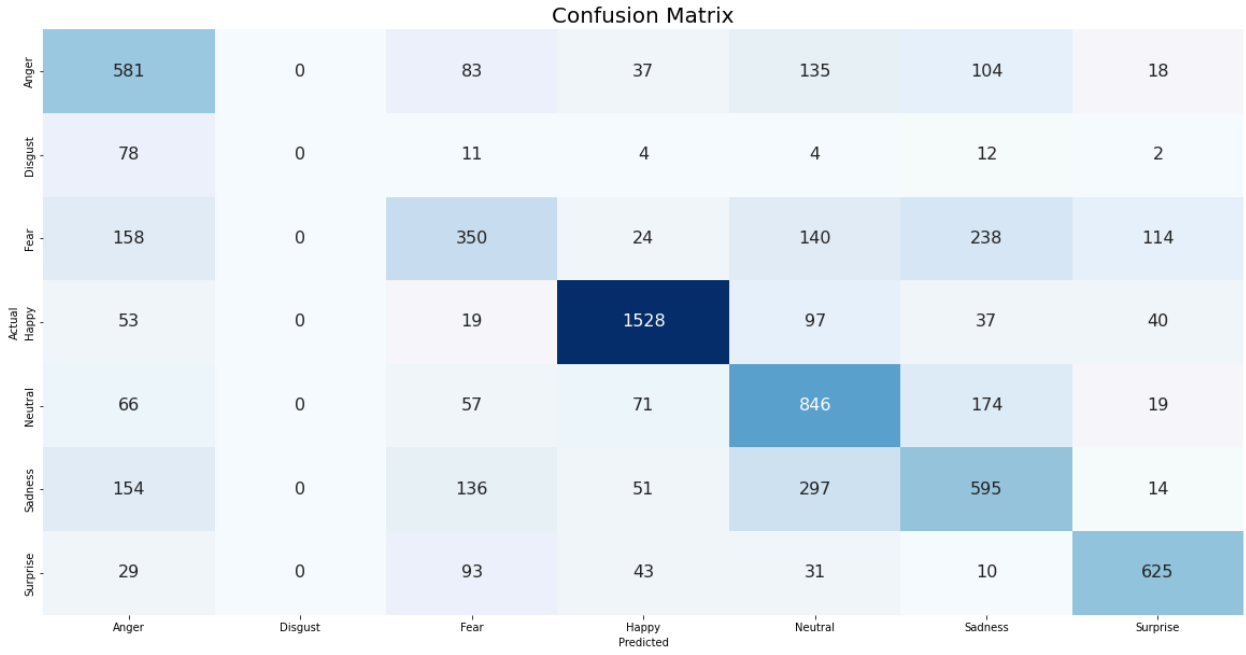
- **Un-Freezing Layers**: Sets the feature extraction layers (DenseNet) to be trainable, allowing them to be fine-tuned.
- **Recompile Model**: Compiles the model with a lower learning rate to avoid large updates that could destabilize the learning process.
- Trains the model again with the feature extraction layers unfrozen for a set number of fine-tuning epochs (`FINE_TUNING_EPOCHS`).
- Validates on the validation data.
- **Combining Histories**: Merges the training histories from both phases for a comprehensive view of the training process.

Results:






TRUE POSITIVE RATE
FALSE POSITIVE RATE
0.0
0.2
0.4
0.6
0.8
1.0
Anger (AUC:0.88)
Disgust (AUC:0.92)
Fear (AUC:0.83)
Happy (AUC:0.97)
Neutral (AUC:0.90)
Sadness (AUC:0.86)
Surprise (AUC:0.96)
Random Guessing


Confusion Matrix
Actual
Predicted
Anger
Disgust
Fear
Happy
Neutral
Sadness
Surprise
581 0 83 37 135 104 18
78 0 11 4 4 12 2
158 0 350 24 140 238 114
53 0 19 1528 97 37 40
66 0 57 71 846 174 19
154 0 136 51 297 595 14
29 0 93 43 31 10 625