**Facial Emotion Recognition**

ABSTRACT

In the realm of artificial intelligence, the ability to interpret human emotions plays a pivotal role in creating more natural and engaging interactions between machines and humans. This article embarks on an exploration of this intersection of computer vision and affective computing. The paper presents a comprehensive approach to facial emotion recognition, leveraging the power of Python in data analysis and machine learning tasks. The methodology hinges on the principles of Image Classification, Convolutional Neural Networks (CNN), and Deep Learning - the cutting-edge technologies that have revolutionized the field of computer vision. Through the lens of these advanced techniques, the article delves into the intricacies of detecting and classifying a spectrum of human emotions, captured and expressed through facial expressions. The objective is to build a model that can accurately and efficiently recognize these emotions, thereby contributing to the development of more empathetic and responsive AI systems.

INTRODUCTION

Facial Emotion Recognition (FER) plays a crucial role in human-computer interaction, psychological studies, and various other applications. The advent of Convolutional Neural Networks (CNN) and Deep Learning has significantly enhanced the capabilities of FER systems. In the realm of human communication, facial expressions serve as a critical source of nonverbal cues that complement and enhance the understanding of verbal communication. Research shows that 60-80% of communication is nonverbal in nature. This nonverbal dimension includes elements such as facial expressions, eye contact, vocal intonations, hand gestures, and physical distancing.

Among these, analyzing facial expression has become an important area of research. Facial emotion recognition (FER), a key application of this research, has found wide application in the field of human-computer interaction (HCI). Its applications span various fields including autonomous driving, education, healthcare, psychotherapy, surveillance, and computer vision-based psychological analysis.

CNN, a class of deep learning models, is particularly effective in processing grid-like data, such as images. It uses convolutional layers with sliding windows to capture local features and pooling layers to reduce dimensionality, thereby enabling the model to learn complex patterns in the data. This makes CNN an ideal choice for FER, where the task is to identify subtle emotional cues from facial images.

Deep Learning, a subset of machine learning, uses artificial neural networks with multiple layers (deep structures) to model high-level abstractions in data. This allows the model to learn intricate structures from large datasets, making it highly effective for tasks like FER. Deep Learning models can automatically learn representative features from raw pixels of facial images, eliminating the need for manual feature extraction, which is a significant advantage.

In FER, CNN and Deep Learning are often used together. The CNN layers are used to extract robust and discriminative features from facial images, and the deep learning model is used to classify these features into different emotions. This combination has proven to be highly effective, leading to state-of-the-art performance on several FER benchmarks.

## Dataset Information

The objective of the project is to detect facial expression using facial image dataset. Convolutional Neural Network is used to classify the images. The output class consists of 7 different types namely angry, disgust, fear, happy, neutral, sad, surprise.

## Import Modules

**import** **pandas** **as** **pd**

**import** **numpy** **as** **np**

**import** **os**

**import** **matplotlib.pyplot** **as** **plt**

**import** **seaborn** **as** **sns**

**import** **warnings**

**import** **random**

**from** **tqdm.notebook** **import** tqdm

warnings.filterwarnings('ignore')

%**matplotlib** inline

**import** **tensorflow** **as** **tf**

**from** **tensorflow.keras.utils** **import** to\_categorical

**from** **keras.preprocessing.image** **import** load\_img

**from** **keras.models** **import** Sequential

**from** **keras.layers** **import** Dense, Conv2D, Dropout, Flatten, MaxPooling2D

* **pandas**: This is a powerful Python library for data manipulation and analysis. It provides data structures and functions needed to manipulate structured data, including functions for reading and writing data, handling missing data, filtering data, cleaning messy data, and more. It’s particularly useful for working with numerical tables and time series data.
* **numpy**: This is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. It’s used in various fields like machine learning, scientific computing, etc.
* **matplotlib**: This is a plotting library for Python. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like Tkinter, wxPython, Qt, or GTK. It’s also a popular choice for creating static, animated, and interactive visualizations in Python.
* **seaborn**: This is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics. It’s built on top of matplotlib and closely integrated with pandas data structures.
* **os**: This is a module in Python that provides functions for interacting with the operating system. It comes under Python’s standard utility modules. All functions in os module raise OSError in the case of invalid or inaccessible file names and paths, or other arguments that have the correct type, but are not accepted by the operating system.
* **tqdm**: This is a fast, extensible progress bar for Python and CLI. It can wrap around any iterable or be used as a decorator, to provide a visual indication of progress, time estimation, and more.
* **warnings**: This is a module in Python used to warn the programmer about changes in language or library features in anticipation of backwards incompatible changes coming with future versions of Python, or to warn about issues in the code.
* **load\_img**: This is a function in the keras.preprocessing.image module, used to load an image into PIL format.
* **tensorflow**: This is a free and open-source software library for machine learning and artificial intelligence. It can be used across a range of tasks but has a particular focus on training and inference of deep neural networks.
* **Dense**: This is a layer type in neural networks. It’s the regular deeply connected neural network layer. It’s most common and frequently used layer. Dense layer does the below operation on the input and return the output.
* **Dropout**: This is a regularization technique for reducing overfitting in neural networks. It works by temporarily dropping out, or ignoring, some proportion of the neurons during training, which helps to make the model more robust and less prone to overfitting on the training data.
* **Activation**: This is a layer in a neural network that applies an activation function. The activation function is used to determine the output of neural network like yes or no. It maps the resulting values in between 0 to 1 or -1 to 1 etc. (depending upon the function).
* **Flatten**: This is a function that converts a 2D array (a matrix) into a 1D array. It’s often used when you need to transform a multi-dimensional input into a format that a machine learning model can interpret and learn from.
* **Conv2D**: This is a 2D convolution layer, creating a convolution kernel that is convolved with the layer input to produce a tensor of outputs. It’s most commonly used to analyze visual imagery.
* **MaxPooling2D**: This is a function that performs max pooling operation for spatial data. The objective is to down-sample an input representation (image, hidden-layer output matrix, etc.), reducing its dimensionality and allowing for assumptions to be made about features contained in the sub-regions binned.

## Load the Dataset

**Now we load the test data and the train data for processing**

TRAIN\_DIR = '../input/facial-expression-dataset/train/train/'

TEST\_DIR = '../input/facial-expression-dataset/test/test/'

**Now we define the directory for the images and with the corresponding labels**

**def** load\_dataset(directory):

image\_paths = []

labels = []

**for** label **in** os.listdir(directory):

**for** filename **in** os.listdir(directory+label):

image\_path = os.path.join(directory, label, filename)

image\_paths.append(image\_path)

labels.append(label)

print(label, "Completed")

**return** image\_paths, labels

**TRAIN\_DIR** and **TEST\_DIR**: These are the directories where the training and testing data are stored. The training data is used to train the model, and the testing data is used to evaluate its performance.

1. **load\_dataset(directory)**: This is a function that loads the dataset from a given directory. It initializes two empty lists, image\_paths and labels.
2. The function then iterates over each label (or subdirectory) in the given directory. For each label, it iterates over each file in the label’s directory.
3. For each file, it constructs the full path to the file by joining the directory, label, and filename. This path is then appended to the image\_paths list. The label is also appended to the labels list.
4. After all files for a label have been processed, it prints a message indicating that the label has been completed.
5. Once all labels and files have been processed, the function returns the image\_paths and labels lists. These lists can then be used to load the actual image data and corresponding labels for use in a machine learning model.

In summary, this code is used to gather all the image paths and their corresponding labels from a structured directory. The directory is expected to have one subdirectory for each label, and each subdirectory should contain the image files for that label. The function returns a list of image paths and a list of their corresponding labels.

surprise Completed

fear Completed

angry Completed

neutral Completed

sad Completed

disgust Completed

happy Completed



Facial Expression Train Dataset

* Display of image paths with labels of the train dataset
* Data was shuffled and the index was removed

**Test**

test = pd.DataFrame()

test['image'], test['label'] = load\_dataset(TEST\_DIR)

test.head()

surprise Completed

fear Completed

angry Completed

neutral Completed

sad Completed

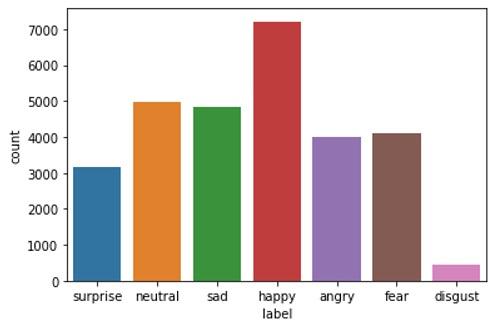
disgust Completed

happy Completed

* First line creates an empty DataFrame named test. A DataFrame is a two-dimensional labeled data structure with columns of potentially different types.
* Next line calls a function named load\_dataset with TEST\_DIR as an argument. TEST\_DIR is likely a directory path where the test dataset is stored. The load\_dataset function is expected to return two values which are then stored in the ‘image’ and ‘label’ columns of the test DataFrame.
* Last line displays the first 5 rows of the test DataFrame. This is often used for a quick preview of the data.

## **Exploratory Data Analysis**

sns.countplot(train['label'])



Visualization of Label Count for the Dataset

* Display of the no. of samples in the dataset
* Happy has more data samples compared to other classes
* The rest of the classes has a uniform distribution
* Disgust has less samples for training

**from** **PIL** **import** Image

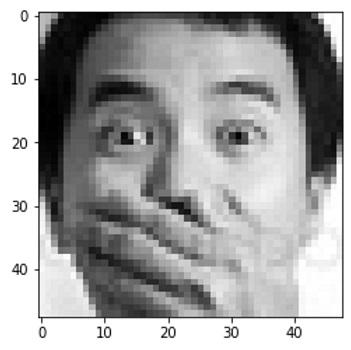
img = Image.open(train['image'][0])

plt.imshow(img, cmap='gray');

The Image module from the PIL library, which stands for Python Imaging Library. This library provides extensive file format support, an efficient internal representation, and powerful image processing capabilities.

The Image.open() function is used to open, manipulate, and save many different image file formats. Here, it’s opening the image file located at the path specified by train['image'][0]. The opened image is stored in the img variable.

This line uses the imshow function from the matplotlib.pyplot library (which is usually imported as plt) to display the image data. The cmap='gray' argument sets the color map of the image to grayscale.



Sample Image

* The images are in grayscale, so color map is declared as gray
* This is the first sample from the data, which is surprise
* The resolution of the image is 48 x 48

**Now we will display a grid of images to see various images at once**

plt.figure(figsize=(20,20))

files = train.iloc[0:25]

**for** index, file, label **in** files.itertuples():

plt.subplot(5, 5, index+1)

img = load\_img(file)

img = np.array(img)

plt.imshow(img)

plt.title(label)

plt.axis('off')

1. line creates a new figure for plotting with a specified size of 20x20
2. line selects the first 25 rows from the train DataFrame and stores them in the files variable.
3. line starts a loop that iterates over the tuples produced by the itertuples() method of the DataFrame files. Each tuple contains the index of the row and the values of each column in the row. In this case, the columns are assumed to be ‘file’ and ‘label’.
4. line creates a subplot in the figure. The figure is divided into a 5x5 grid (as specified by the first two arguments), and the subplot is placed at the position specified by index+1 (the third argument).
5. line calls a function named load\_img with file as an argument. file is likely a file path where an image is stored. The load\_img function is expected to return an image which is then stored in the img variable.
6. line converts the image into a NumPy array. This is often done to facilitate image processing tasks.
7. line displays the image data in the current subplot.
8. line sets the title of the current subplot to the value of label.
9. line turns off the axis lines and labels for the current subplot.

Image size was reduced to minimize the blurriness of the images.

## Feature Extraction

**Now we define the feature extraction method**

**def** extract\_features(images):

features = []

**for** image **in** tqdm(images):

img = load\_img(image, grayscale=**True**)

img = np.array(img)

features.append(img)

features = np.array(features)

features = features.reshape(len(features), 48, 48, 1)

**return** features

1. line defines a function named extract\_features that takes an argument images, which is expected to be a list of image file paths.
2. line initializes an empty list named features to store the features extracted from each image.
3. line starts a loop that iterates over each image file path in the images list. The tqdm function is a progress bar and it wraps around any iterable.
4. line calls a function named load\_img with image (the current image file path) and grayscale=True as arguments. The load\_img function is expected to return an image which is then stored in the img variable. The grayscale=True argument indicates that the image should be loaded in grayscale.
5. line converts the image into a NumPy array. This is often done to facilitate image processing tasks.
6. line appends the NumPy array representation of the image to the features list.

After all images have been processed, [7] line converts the features list into a NumPy array.

Its reshapes the features array into a 4D array with dimensions (number of images, image height, image width, number of channels). Here, it’s assumed that each image is 48x48 pixels and has 1 channel (since the images were loaded in grayscale).

Finally, the function returns the features array.

**Now we extract the features from both train and test**

train\_features = extract\_features(train['image'])

test\_features = extract\_features(test['image'])

These lines call the extract\_features function (which was defined earlier) on the ‘image’ column of the train and test DataFrames. The function is expected to return a 4D array of images for each DataFrame. The returned arrays are stored in train\_features and test\_features.

**Now we normalize the images**

x\_train = train\_features/255.0

x\_test = test\_features/255.0

These lines normalize the image data in train\_features and test\_features. Each pixel value in an image is divided by 255.0, converting the pixel values from the range 0-255 to the range 0-1. This is done because pixel values in images are usually integers in the range 0-255, but neural networks work best with input data that is on a smaller scale, typically between 0 and 1 or -1 and 1 (this is a form of data normalization).

The normalized image data is stored in x\_train and x\_test. These will be the inputs to the neural network.

**Label Encoding**: The LabelEncoder class in sklearn is a utility class to help normalize labels such that they contain only values between 0 and n\_classes-1. This can be useful for certain types of machine learning algorithms that expect numerical input rather than categorical (string) input.

**from** **sklearn.preprocessing** **import** LabelEncoder

le = LabelEncoder()

le.fit(train['label'])

y\_train = le.transform(train['label'])

y\_test = le.transform(test['label'])

In the above code:

* An instance of LabelEncoder is created and is fit to the labels in the training data (train['label']).
* The fit method finds all unique class labels.
* The transform method then replaces the original class labels with the encoded labels (integers). This is done for both the training and test datasets. Converting the data type of 'label' to integer for easier processing

**One-Hot Encoding**: After label encoding, the labels are one-hot encoded. One-hot encoding is a process of converting categorical data variables so they can be provided to machine learning algorithms to improve predictions. With one-hot, we convert each categorical value into a new categorical value and assign a binary value of 1 or 0. Each integer value is represented as a binary vector. All the values are zero, and the index is marked with a 1.

y\_train = to\_categorical(y\_train, num\_classes=7)

y\_test = to\_categorical(y\_test, num\_classes=7)

* The to\_categorical function is used to convert class vectors (y\_train and y\_test) to binary class matrices, which are needed for categorical\_crossentropy.
* num\_classes defines the total number of unique classes in the ‘label’ column. In this case, it’s 7.

y\_train[0]

*# config*

input\_shape = (48, 48, 1)

output\_class = 7

* **input\_shape = (48, 48, 1)** - Converts the input image into 48 x 48 resolution in grayscale
* **output\_class** = 7 - Total no. of classes

**Model Creation**

model = Sequential()

*# convolutional layers*

model.add(Conv2D(128, kernel\_size=(3,3), activation='relu', input\_shape=input\_shape))

model.add(MaxPooling2D(pool\_size=(2,2)))

model.add(Dropout(0.4))

model.add(Conv2D(256, kernel\_size=(3,3), activation='relu'))

model.add(MaxPooling2D(pool\_size=(2,2)))

model.add(Dropout(0.4))

model.add(Conv2D(512, kernel\_size=(3,3), activation='relu'))

model.add(MaxPooling2D(pool\_size=(2,2)))

model.add(Dropout(0.4))

model.add(Conv2D(512, kernel\_size=(3,3), activation='relu'))

model.add(MaxPooling2D(pool\_size=(2,2)))

model.add(Dropout(0.4))

model.add(Flatten())

*# fully connected layers*

model.add(Dense(512, activation='relu'))

model.add(Dropout(0.4))

model.add(Dense(256, activation='relu'))

model.add(Dropout(0.3))

*# output layer*

model.add(Dense(output\_class, activation='softmax'))

model.compile(optimizer='adam', loss='categorical\_crossentropy', metrics='accuracy')

**Sequential Model**: The Sequential model is a linear stack of layers. You can create a Sequential model and define all of the layers in the constructor, or you can add them one by one using the add() method.

**Convolutional Layers**: The Conv2D layer creates a convolution kernel that is convolved with the layer input to produce a tensor of outputs. MaxPooling2D is a way to reduce the number of parameters in our model by sliding a 2x2 pooling filter across the previous layer and taking the max of the 4 values in the 2x2 filter. Dropout is a regularization method, where a proportion of nodes in the layer are randomly ignored by setting their weights to zero for each training sample. This drops randomly a proportion of the network and forces the network to learn features in a distributed way.

**Flatten Layer**: This layer flattens the input. It does not affect the batch size. It’s used without parameters.

**Dense Layers**: Dense layer is the regular deeply connected neural network layer. It’s most common and frequently used layer. Dense layer does the below operation on the input and return the output.

**Output Layer**: This is the final layer, which is responsible for outputting a probability distribution over the classes.

**Compile the Model**: After the model is constructed, configure its learning process with compile() method. It receives three arguments: an optimizer, a loss function and a list of metrics.

In this code, the optimizer is Adam, the loss function is categorical\_crossentropy which is used for multi-class classification, and the metric is accuracy which is the proportion of correct predictions with respect to the targets.

**Now we proceed to train the dataset**

*# train the model*

history = model.fit(x=x\_train, y=y\_train, batch\_size=128, epochs=100, validation\_data=(x\_test, y\_test))

* Set the no. of epochs and batch size according to the hardware specifications
* Training accuracy and validation accuracy increases each iteration
* Training loss and validation loss decreases each iteration

## **Plot the Results**

acc = history.history['accuracy']

val\_acc = history.history['val\_accuracy']

epochs = range(len(acc))

plt.plot(epochs, acc, 'b', label='Training Accuracy')

plt.plot(epochs, val\_acc, 'r', label='Validation Accuracy')

plt.title('Accuracy Graph')

plt.legend()

plt.figure()

loss = history.history['loss']

val\_loss = history.history['val\_loss']

epochs = range(len(acc))

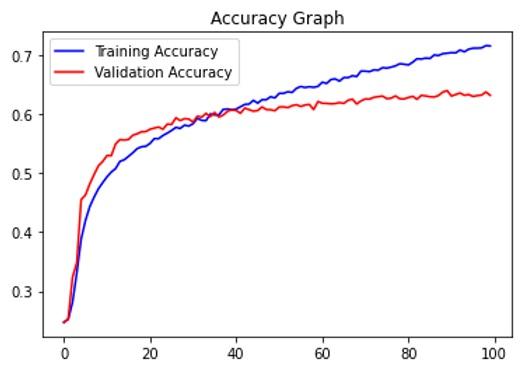
plt.plot(epochs, loss, 'b', label='Training Loss')

plt.plot(epochs, val\_loss, 'r', label='Validation Loss')

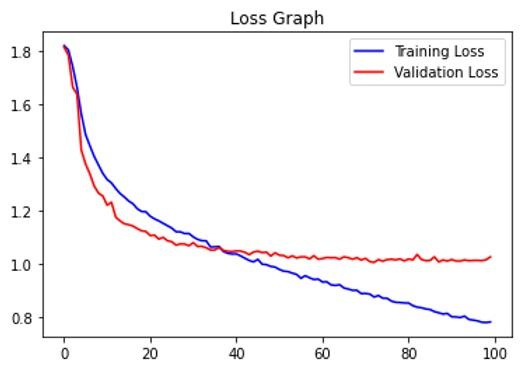
plt.title('Loss Graph')

plt.legend()

plt.show()



Training & Validation Accuracy



Training & Validation Loss

## **Test with Image Data**

image\_index = random.randint(0, len(test))

print("Original Output:", test['label'][image\_index])

pred = model.predict(x\_test[image\_index].reshape(1, 48, 48, 1))

prediction\_label = le.inverse\_transform([pred.argmax()])[0]

print("Predicted Output:", prediction\_label)

plt.imshow(x\_test[image\_index].reshape(48, 48), cmap='gray');

**Final**

The project on Facial Emotion Recognition using Image Classification, Convolutional Neural Networks (CNN), and Deep Learning is a comprehensive study of how machine learning and deep learning techniques can be utilized to understand and interpret human emotions.

The project involves the following steps:

1. **Data Preparation**: The data used in this project are images of faces, each labeled with an emotion. The labels are first converted from categorical to numerical format, and then one-hot encoded for the model to perform multi-class classification.
2. **Model Creation**: A Convolutional Neural Network (CNN), a class of deep learning models most commonly applied to analyzing visual imagery, is built using Keras. The model includes several layers - convolutional layers, max pooling layers, dropout layers for regularization, a flatten layer, and fully connected dense layers. The output layer uses the softmax activation function for multi-class classification.
3. **Model Compilation and Training**: The model is compiled with the Adam optimizer and the categorical cross entropy loss function, which is suitable for multi-class classification. The model is then trained on the prepared data.

In conclusion, this project demonstrates the power and potential of deep learning, particularly Convolutional Neural Networks, in recognizing and classifying facial emotions. It’s a testament to how far we’ve come in the field of Artificial Intelligence, where machines can now understand and interpret human emotions, opening up new possibilities for human-computer interaction.