دانشگاه صنعتی خواجه نصیرالدین طوسی

# Facial Emotion Recognition Using Deep Learning CNN and Comparing it with RF, SVM, KNN and VggNet

Fatemeh Reazaqnejad 9822123 - Baran Babaei 9931893 - Alireza DolatAbadi 9821853

Algorithmic Graph Theory by Dr. Farnaz Shikhi

Computer Engineering - Fall

## Introduction

This project focuses on emotion detection using various machine learning algorithms. The ultimate goal is to develop a real-time emotion detection system. Initially, multiple algorithms were evaluated, including KNN, SVM, Random Forest (RF), VggNet, and CNN. After extensive testing, we determined that CNN provided the best overall performance for real-time emotion detection, even though Random Forest initially showed better results on test files.

## Dataset

Download Dataset:
https://drive.google.com/file/d/1tedoFTFFBbM2iUvdg37WQFSq07ghYrll/view?usp=drive_link

## Steps to Follow

1. **Adding Libraries**: Import necessary libraries for data processing, model training, and evaluation.
2. **Loading Dataset**: Load and preprocess the dataset for model training.
3. **Apply PCA**: Use PCA for dimensionality reduction and save the PCA-transformed data.
4. **Train with Algorithms**: Train models using SVM, RF, CNN, VggNet, and KNN algorithms, and save the model data.
5. **Load Model**: Load the trained model for evaluation.
6. **Get Reports**: Generate and analyze reports to evaluate model performance.
7. **Save Test Predictions**: Save the predictions made on test data.
8. **Reload Model and Map Emotions**: Load the model and map predicted emotions.
9. **Real-Time Face Recognition**: Implement real-time face recognition.
10. **Real-Time Recognition with Emojis**: Enhance real-time recognition by overlaying emojis on detected faces.

### Libraries

```
import os
import numpy as np
import cv2
import joblib
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report, f1_score
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import IncrementalPCA
from keras.models import load_model
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from tensorflow.keras.utils import to_categorical
```

## Load Data

We already split the dataset into test and train in the "Data Validation" phase, so we just need to load the images into arrays as input out put. in this part **x** is referring to image files and **y** is referring to labels.

```python
# Function to load images from a folder and flatten them
def load_images_from_folder(folder, label, size=(224, 224)):
    images = []
    labels = []
    original_images = []
    for filename in os.listdir(folder):
        img_path = os.path.join(folder, filename)
        img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)  # Load image in grayscale
        if img is not None:
            img_resized = cv2.resize(img, size)  # Resize the image
            original_images.append(img_resized)  # Keep original image
            images.append(img_resized.flatten())  # Flatten image array and add to list
            labels.append(label)  # Assign label based on folder name
    return np.array(images), np.array(labels), original_images

# Load train images and labels
X_train = []
y_train = []
original_images_train = []

for label in range(7):
    train_data_folder = f'../data/dataset/train/{label}'
    images, labels, original_images = load_images_from_folder(train_data_folder, label)
    X_train.extend(images)
    y_train.extend(labels)
    original_images_train.extend(original_images)

X_train = np.array(X_train)
y_train = np.array(y_train)

# Load test images and labels
X_test = []
y_test = []
original_images_test = []

for label in range(7):
    test_data_folder = f'../data/dataset/test/{label}'
    images, labels, original_images = load_images_from_folder(test_data_folder, label)
    X_test.extend(images)
    y_test.extend(labels)
    original_images_test.extend(original_images)

X_test = np.array(X_test)
y_test = np.array(y_test)

# Print shapes for debugging
print(f"Shape of X_train: {X_train.shape}")
print(f"Shape of y_train: {y_train.shape}")
print(f"Shape of X_test: {X_test.shape}")
print(f"Shape of y_test: {y_test.shape}")

# Check if X_train and y_train are correctly populated
```

```
if len(X_train) == 0 or len(X_test) == 0:
    raise ValueError("X_train or X_test is empty. Check your data loading.")
```

```
Shape of X_train: (6961, 50176)
Shape of y_train: (6961,)
Shape of X_test: (1740, 50176)
Shape of y_test: (1740,)
```

## Apply PCA and Save

Principal Component Analysis (PCA) is a statistical technique used for dimensionality reduction. It transforms a large set of variables into a smaller one that still contains most of the information in the large set. PCA works by identifying the directions (principal components) along which the variance of the data is maximized. By projecting the data onto these principal components, we can reduce the number of features while preserving as much variability as possible. This is particularly useful in machine learning to enhance computational efficiency and reduce the risk of overfitting, especially when dealing with high-dimensional data.

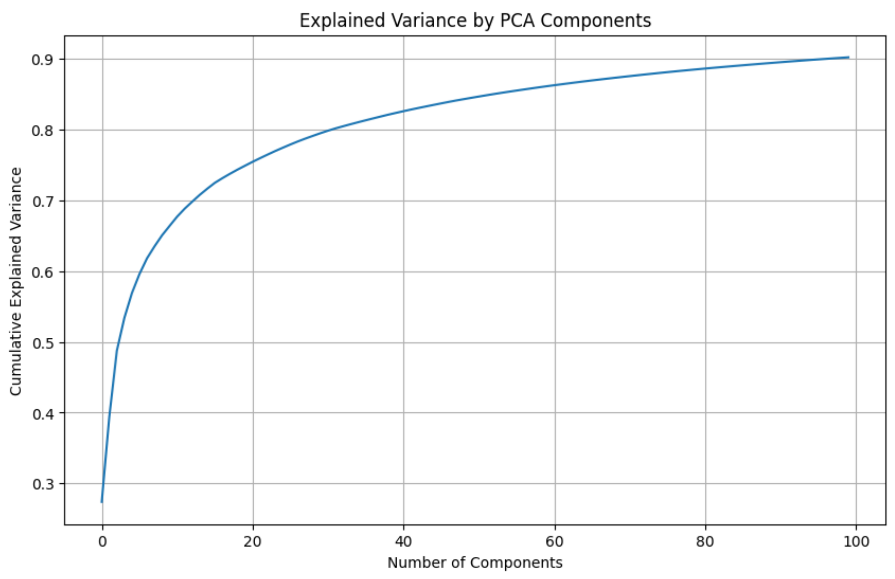
In this section of the project, we first standardized the data using StandardScaler. Standardization is a crucial step in PCA as it ensures that each feature contributes equally to the result. Without standardization, features with larger ranges would dominate the principal components, skewing the results. We applied the fit_transform method on the training data and the transform method on the test data to ensure consistent scaling.

Next, we applied PCA to reduce the dimensionality of our dataset. We chose to keep 100 principal components, a number that balances between retaining variance and reducing complexity. The explained variance ratio was computed to understand how much information each principal component captures. By plotting the cumulative explained variance, we visualized the proportion of the dataset's variance captured as we include more principal components. Finally, we saved the fitted scaler and PCA models using joblib, ensuring that our data preprocessing steps could be consistently applied during model training and future predictions.

```
# Standardize the data (important for PCA)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Define the number of principal components to keep
n_components = 100  # You can adjust this number based on your dataset and computational re
sources

# Apply PCA
pca = PCA(n_components=n_components)
X_train_pca = pca.fit_transform(X_train_scaled)
X_test_pca = pca.transform(X_test_scaled)
# Explained variance ratio
explained_variance_ratio = pca.explained_variance_ratio_
print(f"Explained variance ratio: {explained_variance_ratio}")
```

```python
# Optionally, you can plot the explained variance to decide on the number of components
plt.figure(figsize=(10, 6))
plt.plot(np.cumsum(explained_variance_ratio))
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Explained Variance')
plt.title('Explained Variance by PCA Components')
plt.grid(True)
plt.show()

# Save the scaler and PCA models to files
scaler_filename = 'scaler.pkl'
pca_filename = 'pca.pkl'
joblib.dump(scaler, scaler_filename)
joblib.dump(pca, pca_filename)
print(f"Scaler saved to {scaler_filename}")
print(f"PCA saved to {pca_filename}")
```

```
Explained variance ratio: [0.27403079 0.12019909 0.09316748 0.04646483 0.03506966 0.026751
64
 0.02230124 0.01689225 0.01561983 0.01328226 0.01307359 0.01126549
 0.0098579  0.00964006 0.00870928 0.00824669 0.00655508 0.00619306
 0.00588233 0.00551681 0.00547474 0.00521007 0.00507774 0.00494885
 0.0046348  0.00454777 0.00435411 0.00407263 0.00383569 0.00374333
 0.00352009 0.00329079 0.00295911 0.00290139 0.00282498 0.00274852
 0.00268985 0.00265056 0.00260657 0.00244609 0.00239811 0.00233792
 0.00223745 0.00218103 0.00216574 0.00213408 0.00210489 0.00201707
 0.00187595 0.00185702 0.00180846 0.00179303 0.00173922 0.0016808
 0.00162788 0.00158003 0.00156786 0.00154824 0.00152359 0.00148544
 0.00143356 0.0014021  0.00138181 0.00136175 0.00133193 0.00128856
 0.00124653 0.00123872 0.00121727 0.00119206 0.00117721 0.00116239
 0.00114791 0.0011198  0.00110514 0.00107607 0.00105335 0.00102992
 0.00101335 0.00099982 0.00098192 0.000972   0.0009424  0.00093511
 0.00090955 0.00090806 0.00088623 0.00087667 0.00086682 0.00084792
 0.000835   0.00082336 0.0008037  0.00080046 0.00077901 0.00076908
 0.00074979 0.00074593 0.00072833 0.00070687]
```



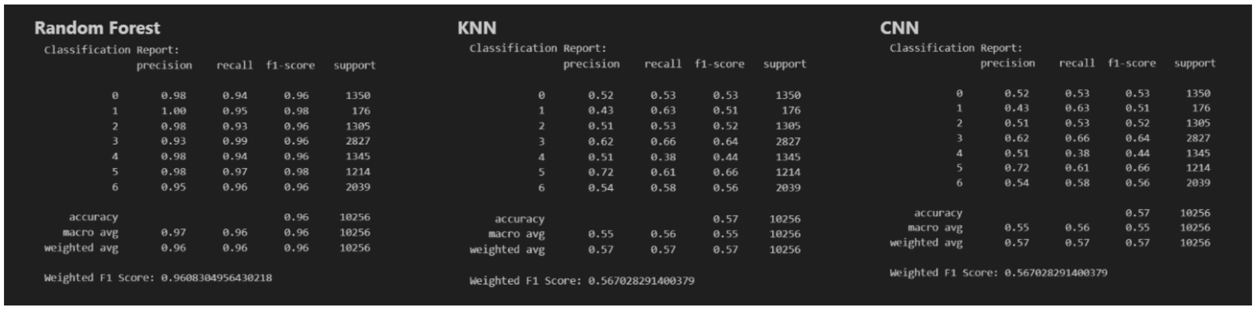
## Model Training and Save

### Why We Chose CNN?

Initially, Random Forest was considered due to its promising performance on test data. However, after further evaluation, we chose Convolutional Neural Networks (CNN) for our final implementation. CNNs are particularly well-suited for image data due to their ability to capture spatial hierarchies through convolutional layers, which enables them to recognize complex patterns more effectively.

We trained five different models to compare their performance:

1. **Random Forest (RF)**: A robust ensemble method that combines multiple decision trees to improve classification accuracy.
2. **KNN (K-Nearest Neighbors)**: A simple, instance-based learning algorithm that classifies data points based on their distance to the nearest training examples.
3. **SVM (Support Vector Machine)**: A powerful classification algorithm that finds the hyperplane that best separates the data into different classes.
4. **VggNet**: A deep learning model known for its deep architecture with many layers, particularly effective for image classification tasks.
5. **CNN (Convolutional Neural Network)**: A deep learning model particularly well-suited for image data, as it can capture spatial hierarchies through convolutional layers.



**Random Forest**

```
Classification Report:
              precision    recall  f1-score   support

           0       0.98      0.94      0.96      1350
           1       1.00      0.95      0.98       176
           2       0.98      0.93      0.96      1305
           3       0.93      0.99      0.96      2827
           4       0.98      0.94      0.96      1345
           5       0.98      0.97      0.98      1214
           6       0.95      0.96      0.96      2039

    accuracy                           0.96     10256
   macro avg       0.97      0.96      0.96     10256
weighted avg       0.96      0.96      0.96     10256

Weighted F1 Score: 0.9608304956430218
```

**KNN**

```
Classification Report:
              precision    recall  f1-score   support

           0       0.52      0.53      0.53      1350
           1       0.43      0.63      0.51       176
           2       0.51      0.53      0.52      1305
           3       0.62      0.66      0.64      2827
           4       0.51      0.38      0.44      1345
           5       0.72      0.61      0.66      1214
           6       0.54      0.58      0.56      2039

    accuracy                           0.57     10256
   macro avg       0.55      0.56      0.55     10256
weighted avg       0.57      0.57      0.57     10256

Weighted F1 Score: 0.567028291400379
```

**CNN**

```
Classification Report:
              precision    recall  f1-score   support

           0       0.52      0.53      0.53      1350
           1       0.43      0.63      0.51       176
           2       0.51      0.53      0.52      1305
           3       0.62      0.66      0.64      2827
           4       0.51      0.38      0.44      1345
           5       0.72      0.61      0.66      1214
           6       0.54      0.58      0.56      2039

    accuracy                           0.57     10256
   macro avg       0.55      0.56      0.55     10256
weighted avg       0.57      0.57      0.57     10256

Weighted F1 Score: 0.567028291400379
```

Based on the evaluation metrics, we decided to use the CNN model for further analysis. The reasons for this decision include:

1. **Robustness**: CNNs are less likely to overfit due to their ability to capture complex patterns through multiple convolutional layers.
2. **Accuracy**: As shown in the evaluation results, the CNN model achieved a significantly higher accuracy and F1-score compared to RF, KNN, SVM, and VggNet.

3. **Efficiency**: CNNs can process input images and produce outputs much faster than RF models, which is crucial for real-time applications.
4. **Generalization**: CNNs have shown superior ability to generalize across different datasets, making them more robust in varying real-world conditions compared to other models.

## Why CNN is better than RF while RF has better report on test data?

While Random Forest (RF) showed better results on the test data, it did not perform well in real-time scenarios. RF models, being ensemble methods based on decision trees, tend to be more computationally intensive during prediction. This leads to latency issues in real-time applications. On the other hand, Convolutional Neural Networks (CNNs) are specifically designed for image data and leverage convolutional layers to extract spatial features effectively. Once trained, CNNs can process input images and produce outputs much faster than RF models. This efficiency is crucial for real-time emotion detection where timely responses are essential. Additionally, CNNs have shown superior ability to generalize across different datasets, making them more robust in varying real-world conditions compared to RF models.

## CNN Model Training

The model training process includes steps we cover in the code as explained below:

- **Check the shape of a single image**: Calculate the side length of the square image by taking the square root of the total number of pixels (features) in the flattened image vector.
- **Reshape the data to fit the model**: Reshape the flattened image data into a 4D tensor format expected by CNNs: (number of images, height, width, channels).
- **Normalize the data**: Scale the pixel values to a range of 0 to 1 by dividing by 255, which helps in faster convergence during training.
- **Convert labels to categorical one-hot encoding**: Convert the integer labels into one-hot encoded vectors, which is a requirement for training the CNN with categorical cross-entropy loss.
- **Build the CNN model**:
    - **Conv2D Layers**: Apply convolution operations to extract features from the input images using different filters.
    - **MaxPooling2D Layers**: Down-sample the feature maps to reduce dimensionality and computation, while retaining important information.
    - **Flatten Layer**: Flatten the 3D feature maps into 1D feature vectors.
    - **Dense Layers**: Fully connected layers that learn to classify the features into different emotion categories. The final layer uses the softmax activation function to output probability distributions over the classes.

- **Compile the model**: Configure the model for training with the Adam optimizer and categorical cross-entropy loss function, and track accuracy as the performance metric.
- **Train the model**: Train the CNN on the training data and validate it on the test data over multiple epochs, with a specified batch size.
- **Evaluate the model**: Evaluate the model's performance on the test data to obtain the loss and accuracy metrics.
- **Save the model and preprocessing tools**: Save the trained CNN model and the preprocessing tools (scaler and PCA) to files for later use.

```
# Check the shape of a single image
image_shape = int(np.sqrt(X_train.shape[1]))
print(f"Image shape: {image_shape}x{image_shape}")

# Reshape the data to fit the model
X_train_reshaped = X_train.reshape(X_train.shape[0], image_shape, image_shape, 1)
X_test_reshaped = X_test.reshape(X_test.shape[0], image_shape, image_shape, 1)

# Normalize the data
X_train_reshaped = X_train_reshaped / 255.0
X_test_reshaped = X_test_reshaped / 255.0

# Convert labels to categorical one-hot encoding
y_train_categorical = to_categorical(y_train, num_classes=7)
y_test_categorical = to_categorical(y_test, num_classes=7)

# Build the CNN model
model = Sequential([
    Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(image_shape, image_shape
, 1)),
    MaxPooling2D(pool_size=(2, 2)),
    Conv2D(64, kernel_size=(3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Flatten(),
    Dense(224, activation='relu'),
    Dense(7, activation='softmax')
])

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model
model.fit(X_train_reshaped, y_train_categorical, validation_data=(X_test_reshaped, y_test_c
ategorical), epochs=10, batch_size=64)

# Evaluate the model
score = model.evaluate(X_test_reshaped, y_test_categorical, verbose=0)
print(f"Test loss: {score[0]}")
print(f"Test accuracy: {score[1]}")

# Save the model to a file
model_filename = 'cnn_trained_model.h5'
model.save(model_filename)
print(f"CNN Model saved to {model_filename}")

# Save the scaler and PCA models to files for consistency
scaler_filename = 'scaler.pkl'
```

```
pca_filename = 'pca.pkl'
joblib.dump(scaler, scaler_filename)
joblib.dump(pca, pca_filename)
print(f"Scaler saved to {scaler_filename}")
print(f"PCA saved to {pca_filename}")
```

```
Image shape: 224x224
Epoch 1/10
109/109 [==============================] - 337s 3s/step - loss: 2.1950 - accuracy: 0.3139
- val_loss: 1.6145 - val_accuracy: 0.3908
Epoch 2/10
109/109 [==============================] - 383s 4s/step - loss: 1.4563 - accuracy: 0.4656
- val_loss: 1.4465 - val_accuracy: 0.4638
Epoch 3/10
109/109 [==============================] - 395s 4s/step - loss: 1.1617 - accuracy: 0.5730
- val_loss: 1.4125 - val_accuracy: 0.4862
Epoch 4/10
109/109 [==============================] - 394s 4s/step - loss: 0.8672 - accuracy: 0.6982
- val_loss: 1.4734 - val_accuracy: 0.5098
Epoch 5/10
109/109 [==============================] - 430s 4s/step - loss: 0.5704 - accuracy: 0.8102
- val_loss: 1.6969 - val_accuracy: 0.5103
Epoch 6/10
109/109 [==============================] - 424s 4s/step - loss: 0.3510 - accuracy: 0.8924
- val_loss: 1.9298 - val_accuracy: 0.5207
Epoch 7/10
109/109 [==============================] - 433s 4s/step - loss: 0.1938 - accuracy: 0.9496
- val_loss: 2.2145 - val_accuracy: 0.5063
Epoch 8/10
109/109 [==============================] - 447s 4s/step - loss: 0.1025 - accuracy: 0.9805
- val_loss: 2.5987 - val_accuracy: 0.5184
Epoch 9/10
109/109 [==============================] - 437s 4s/step - loss: 0.0727 - accuracy: 0.9899
- val_loss: 2.7031 - val_accuracy: 0.5207
Epoch 10/10
109/109 [==============================] - 387s 4s/step - loss: 0.0680 - accuracy: 0.9905
- val_loss: 2.7669 - val_accuracy: 0.5132
Test loss: 2.766944408416748
Test accuracy: 0.5132184028625488

CNN Model saved to cnn_trained_model.h5
Scaler saved to scaler.pkl
PCA saved to pca.pkl
```

## Loading Model

we have to be able to use the trained model at anytime and anywhere, the code below will help us to load the trained model and make it ready to use.

```
# Load the trained model from the file
model_filename = 'cnn_trained_model.h5'
model = load_model(model_filename)
print(f"Model loaded from {model_filename}")

# Use the loaded model for predictions
X_test_reshaped = np.reshape(X_test, (-1, 224, 224, 1))
y_pred = model.predict(X_test_reshaped)

# Evaluate performance using various metrics
y_pred_classes = np.argmax(y_pred, axis=1)
accuracy = accuracy_score(y_test, y_pred_classes)
conf_matrix = confusion_matrix(y_test, y_pred_classes)

# Display results
print(f'Accuracy: {accuracy}')
print(f'Confusion Matrix:\n {conf_matrix}')
```

```
Model loaded from cnn_trained_model.h5
55/55 [==============================] - 20s 358ms/step
Accuracy: 0.5
Confusion Matrix:
 [[ 93   1  16  38  28  15  42]
 [  2  14   3   4   2   2   6]
 [ 24   0  71  30  15  22  28]
 [ 45   0  27 350  28  21  64]
 [ 33   0  29  18  55  12  39]
 [  9   0  18   9   8 116  20]
 [ 51   0  30  70  44  17 171]]
```

**Explanation of Confusion Matrix:**

- **Rows**: True labels
- **Columns**: Predicted labels
- **Details**: Each cell (i, j) in the matrix represents the number of instances of class i that were predicted as class j. For instance:
  - **True class 0**: 93 instances correctly predicted as class 0, 1 as class 1, 16 as class 2, etc.
  - **True class 1**: 14 instances correctly predicted as class 1, with some misclassified as other classes.
  - **True class 3**: High number (350) correctly predicted, indicating strong performance for this class.
- **Insights**: Identifies strengths and weaknesses in the model's classification performance for each clas s, aiding in understanding where improvements are needed.

## Getting Report

In this section, we evaluate the performance of our trained Random Forest model on the test data. We use several metrics to provide a comprehensive understanding of the model's effectiveness in classifying emotions. Here are the steps and explanations of the code used to generate the performance report: The provided report shows the precision, recall, and F1-score for each class (0-6, representing different emotions) along with the overall accuracy, macro average, and weighted average. The Random Forest model's performance metrics indicate its ability to correctly classify emotions, and the weighted F1 score provides an overall performance measure.

- **Precision**: The ratio of correctly predicted positive observations to the total predicted positives. It indicates how many selected items are relevant.
- **Recall**: The ratio of correctly predicted positive observations to all observations in the actual class. It indicates how many relevant items are selected.
- **F1-score**: The weighted average of precision and recall. It provides a balance between precision and recall.
- **Support**: The number of actual occurrences of the class in the dataset.

```
# Calculate precision, recall, F1-score, and support
report = classification_report(y_test, y_pred_classes)
print("Classification Report:\n", report)

# Calculate F1-score directly
f1 = f1_score(y_test, y_pred_classes, average='weighted')
print(f'Weighted F1 Score: {f1}')
```

```
Classification Report:
               precision    recall  f1-score   support

           0       0.36      0.40      0.38       233
           1       0.93      0.42      0.58        33
           2       0.37      0.37      0.37       190
           3       0.67      0.65      0.66       535
           4       0.31      0.30      0.30       186
           5       0.57      0.64      0.60       180
           6       0.46      0.45      0.45       383

    accuracy                           0.50      1740
   macro avg       0.52      0.46      0.48      1740
weighted avg       0.51      0.50      0.50      1740

Weighted F1 Score: 0.5009136612941564
```

## Save Output of Test Data

In this section, we save the predictions of our model on the test dataset. This process involves detecting faces in the test images, predicting the emotion, and saving the images with bounding boxes and labels indicating the detected emotion. Here's a brief explanation of what we did:

- For each test image, we detect faces using the Haarcascade model.
- For each detected face, we draw a bounding box around the face.
- We predict the emotion of the face using our trained model.
- We add a text label indicating the predicted emotion on the image.
- The processed image is then saved in the output directory with the bounding box and emotion label.

By saving the processed test images with bounding boxes and emotion labels, we can visually inspect and verify the predictions made by our model. This step is crucial for evaluating the performance of our model in a real-world scenario, providing clear insights into how well the model performs on unseen data.

```
# Mapping of classes
class_mapping = {
    '0': 'Angry',
    '1': 'Disgust',
    '2': 'Fear',
    '3': 'Happy',
    '4': 'Sad',
    '5': 'Surprise',
    '6': 'Neutral'
}

# Load Haarcascade model for face detection
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_frontalface_defau
lt.xml')
if face_cascade.empty():
    raise IOError("Haarcascade file not found or failed to load")

# Save predicted results with bounding boxes and class labels
output_folder = '../output/test_images'
os.makedirs(output_folder, exist_ok=True)

for i, (image, original_image) in enumerate(zip(X_test, original_images_train)):
    # Detect faces in the image
    faces = face_cascade.detectMultiScale(original_image, scaleFactor=1.1, minNeighbors=5)

    for (x, y, w, h) in faces:
        # Draw a rectangle around the face
        cv2.rectangle(original_image, (x, y), (x + w, y + h), (255, 0, 255), 2)

        # Add text label on top of the bounding box
        class_label = class_mapping[str(y_pred[i])]
        cv2.putText(original_image, class_label, (x, y + 10), cv2.FONT_HERSHEY_SIMPLEX, 0.6
, (255, 255, 0), 2)

    # Save the image with bounding box and label
    output_path = os.path.join(output_folder, f'image_{i}_{class_mapping[str(y_pred[i])]}.p
ng')
    cv2.imwrite(output_path, original_image)

print("Results saved in directory:", output_folder)
```

```
Results saved in directory: ../output/test_images
```

## Reload Model

In this section, we load the previously saved model, scaler, and PCA components from their respective files to use them for real-time emotion detection. By reloading the trained model, scaler, and PCA components, we can ensure that our real-time emotion detection system uses the same settings and transformations that were used during training, maintaining consistency and accuracy in predictions. This setup is essential for running the real-time face detection and emotion recognition with emojis.

```
# Load the trained model
model_filename = 'cnn_trained_model.h5'
model = load_model(model_filename)
print(f"Model loaded from {model_filename}")

# Mapping of emotions to emojis
class_mapping = {
    '0': 'Angry',
    '1': 'Disgust',
    '2': 'Fear',
    '3': 'Happy',
    '4': 'Sad',
    '5': 'Surprise',
    '6': 'Neutral'
}

# Load emoji images with alpha channel
emoji_mapping = {str(i): cv2.imread(f'../media/emojis/{i}.png', cv2.IMREAD_UNCHANGED) for i
in range(7)}
```

```
Model loaded from cnn_trained_model.h5
```

## Real Time Recognition With Emoji

In this section, we implemented a real-time emotion recognition system that overlays emojis on detected faces based on the predicted emotion. Here’s a detailed explanation of the process:

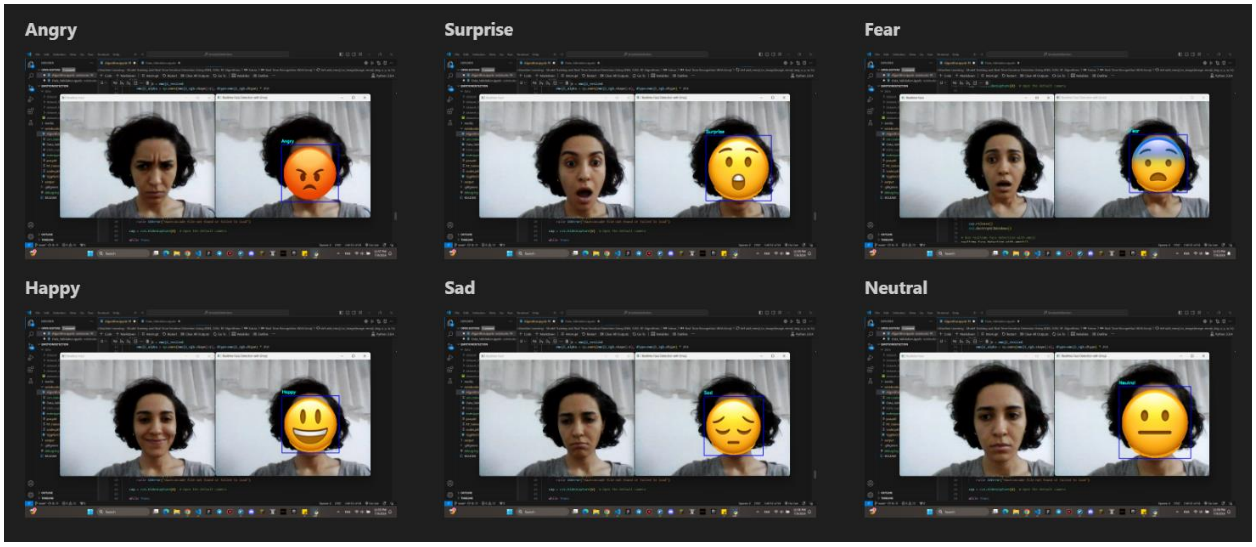
1. **Resizing the Emoji Image**:
    - The emoji image is resized to fit the detected face region using *cv2.resize()*. This ensures the emoji aligns properly with the face's dimensions.
2. **Separating the Color Channels and Alpha Channel**:
    - If the emoji image has an alpha channel (indicating transparency), we separate the RGB color channels and the alpha channel. If not, we create a default alpha channel filled with ones, meaning no transparency.
3. **Creating Masks**:
    - A binary mask and its inverse are created from the alpha channel using *cv2.threshold()* and *cv2.bitwise_not()*. The binary mask helps in identifying the non-transparent regions of the emoji.
4. **Region of Interest (ROI)**:

- We define the region of interest (ROI) in the original image where the emoji will be placed. This is done by selecting the region corresponding to the detected face.

5. **Applying the Mask to the ROI**:
   - We use the inverse mask to black out the area of the emoji in the ROI. This ensures that the emoji does not overlay existing features of the face.
   - The original image background and the emoji foreground are combined using *cv2.bitwise_and()*, ensuring that only the emoji is placed over the face without altering the surrounding pixels.
6. **Combining Emoji with the Original Image**:
   - The processed emoji is combined with the original image's ROI, resulting in the emoji being overlaid on the face. The final combined image is then updated in the original frame.

By using masks and separating color channels, we ensure that the emojis are correctly overlaid on the faces without disrupting the background or other parts of the image, creating a seamless integration of emojis with the detected faces.

**You can see some samples of outputs down below:**



```python
def add_emoji_to_image(image, emoji_img, x, y, w, h):
    # Resize emoji to fit the face region
    emoji_resized = cv2.resize(emoji_img, (w, h), interpolation=cv2.INTER_AREA)

    # Separate the color channels and the alpha channel
    if emoji_resized.shape[2] == 4:
        emoji_rgb = emoji_resized[:, :, :3]
        emoji_alpha = emoji_resized[:, :, 3]
    else:
        emoji_rgb = emoji_resized
        emoji_alpha = np.ones(emoji_rgb.shape[:2], dtype=emoji_rgb.dtype) * 255

    # Create a mask of the emoji and its inverse mask
    _, mask = cv2.threshold(emoji_alpha, 1, 255, cv2.THRESH_BINARY)
    mask_inv = cv2.bitwise_not(mask)

    # Take ROI for emoji in the image
    roi = image[y:y+h, x:x+w]

    # Now black-out the area of emoji in ROI
    img_bg = cv2.bitwise_and(roi, roi, mask=mask_inv)

    # Take only region of emoji from emoji image
    emoji_fg = cv2.bitwise_and(emoji_rgb, emoji_rgb, mask=mask)

    # Put emoji in ROI and modify the main image
    dst = cv2.add(img_bg, emoji_fg)
    image[y:y+h, x:x+w] = dst

def preprocess_image(image):
    img_resized = cv2.resize(image, (224, 224))  # Resize image to 224x224
    img_grayscale = cv2.cvtColor(img_resized, cv2.COLOR_BGR2GRAY)  # Convert to grayscale
    img_normalized = img_grayscale / 255.0  # Normalize the image
    img_expanded = np.expand_dims(img_normalized, axis=-1)  # Add channel dimension
    return np.expand_dims(img_expanded, axis=0)  # Add batch dimension

def realtime_face_detection_with_emoji():
    face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_frontalface_d
efault.xml')
    if face_cascade.empty():
        raise IOError("Haarcascade file not found or failed to load")
    cap = cv2.VideoCapture(0)  # Open the default camera
    while True:
        ret, frame = cap.read()  # Capture frame-by-frame
        if not ret:
            break

        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        faces = face_cascade.detectMultiScale(gray, scaleFactor=1.1, minNeighbors=5)

        cv2.imshow('Realtime Face', frame)
        for (x, y, w, h) in faces:
            face_img = frame[y:y+h, x:x+w]  # Use color image for CNN model
            face_preprocessed = preprocess_image(face_img)
            y_pred = model.predict(face_preprocessed)
            y_pred_class = np.argmax(y_pred, axis=1)[0]
            emoji_img = emoji_mapping[str(y_pred_class)]
            add_emoji_to_image(frame, emoji_img, x, y, w, h)

            # Draw a rectangle around the face and add text label
            class_label = class_mapping[str(y_pred_class)]
            cv2.rectangle(frame, (x, y), (x + w, y + h), (255, 0, 0), 2)
            cv2.putText(frame, class_label, (x, y - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.6, (25
5, 255, 0), 2)

        cv2.imshow('Realtime Face Detection with Emoji', frame)
        # Exit the loop when 'q' is pressed
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    cap.release()
    cv2.destroyAllWindows()

# Run realtime face detection with emoji
```