Emotion Recognition Using Facial Images

Group 17

Contributors: Faruk Beygo, Bora Haliloğlu, Kadir İhsan Sayıcı, İkranur Yılmaz, Ege Şirvan

Course Coordinator: Ayşegül Dündar Boral
Course: CS 464 : Introduction to Machine Learning
Semester: Fall 2024

Bilkent University
Department of Mathematics
2024

Contents

1 Introduction

The aim of this project is to create a model capable of detecting emotions from facial expressions with high accuracy. Facial expressions are crucial for understanding human emotions and predicting subsequent actions. By analyzing these expressions in detail, we can gain insights into how individuals feel and respond in different scenarios.

The dataset used in this project consists of images labeled with six distinct emotions: happy, angry, sad, neutral, surprise, and ahegao. The dataset has the following distribution: 3740 happy expressions, 1313 angry expressions, 3934 sad expressions, 4027 neutral expressions, 1234 surprise expressions, and 1205 ahegao expressions.

We started by preparing the images, converting them to grayscale and using data augmentation to expand the dataset. To make the classes more balanced, we adjusted the dataset for better representation. We also used PCA to simplify the data and make processing faster. Different algorithms were tested to see how they performed in terms of speed and memory usage.

The dataset is divided into training, validation, and testing sets, with an initial split of 80% for training, 10% for validation, and 10% for testing. This split ensures a balanced distribution and robust model evaluation. The results and findings of this study are documented in a detailed report.

2 Dataset [Kap24]

The dataset used in this study was made specifically for recognizing emotions. It includes facial images labeled with six different emotions: Happy, Angry, Sad, Neutral, Surprise, and Ahegao. These categories cover a wide range of expressions, making the dataset a great tool for studying and training models in emotion recognition.

2.1 Dataset Analysis

Initially, our dataset consisted of two distinct components: the images used for training the model and the metadata stored in a CSV file, which contained the corresponding labels for each image. To begin the analysis, we visualized the distribution of the labels in a bar chart, as shown below. This visualization revealed a significant imbalance in the distribution of classes. To address this issue, we decided to normalize the dataset, ensuring a more balanced representation of each class.

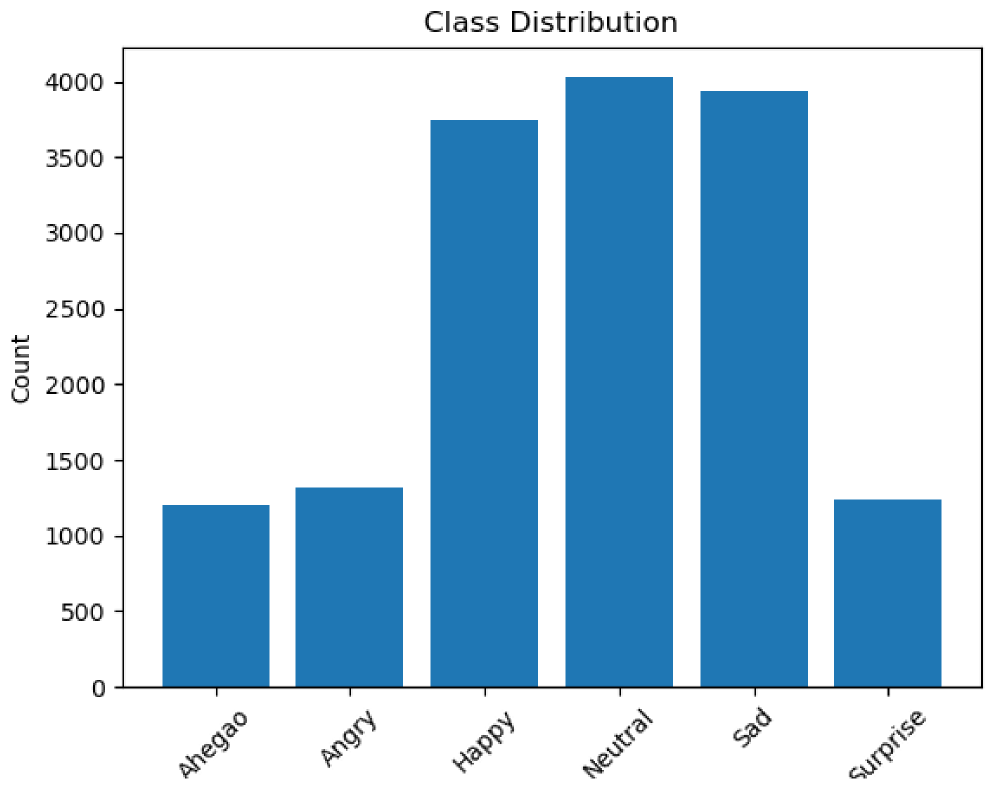


Figure 1: Initial Class Distribution

To address the imbalance in the dataset, we applied data augmentation using effective techniques such as rotation (±10 degrees), translation (up to 10%), and horizontal flipping. These methods helped generate diverse representations of the existing data. For the target classes (Ahegao, Angry, and Surprise), we created 7,000 samples per class, while for the other classes, we generated 4,000 samples each. This augmentation process was essential to increase the amount of training data and ensure better model performance. Additionally, the metadata was updated to reflect the newly augmented samples. As a result, the class distribution became more balanced, as illustrated below.

Listing 1: Augmentation of the Dataset

```
classes_to_augment = ["Ahegao", "Surprise", "Angry"]

data_augmenter = ImageDataGenerator(
    rotation_range=10,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True
)

augmented_images = []
augmented_labels = []

for emotion in emotion_map:
    cls = emotion_map[emotion] # Get the class index
    print(f"Augmenting data for class: {emotion} (Class ID: {cls})")

    class_images = images[labels == cls]
    class_images = np.expand_dims(class_images, axis=-1) # Add channel
        dimension

    class_save_dir = os.path.join(preprocessed_save_dir, emotion)
    os.makedirs(class_save_dir, exist_ok=True) # Create subdirectory
        for each emotion

    class_image_generator = data_augmenter.flow(
        class_images,
        batch_size=1,
        save_to_dir=class_save_dir, # Save augmented images here
        save_prefix=f"{emotion}_aug", # Prefix for file names
        save_format="jpeg" # Save as JPEG format
    )

    initial_files = set(os.listdir(class_save_dir))

    num_to_generate = 4000
    if emotion in classes_to_augment:
        num_to_generate = 7000 # Define how many augmented samples to
            generate per class
    for _ in range(num_to_generate):
        augmented_img = next(class_image_generator)[0] # Get the
            augmented image
        augmented_images.append(augmented_img.astype(np.float32))
        augmented_labels.append(cls)

        current_files = set(os.listdir(class_save_dir))
        new_files = current_files - initial_files # Get the new file(s)

        if new_files:
            aug_img_name = new_files.pop() # There should only be one
                new file

            aug_img_path = os.path.join(class_save_dir, aug_img_name)

            # Update metadata for augmented images
            metadata.append({"file_path": aug_img_path, "label": cls})

            # Update initial files to include the new file
            initial_files.add(aug_img_name)
```

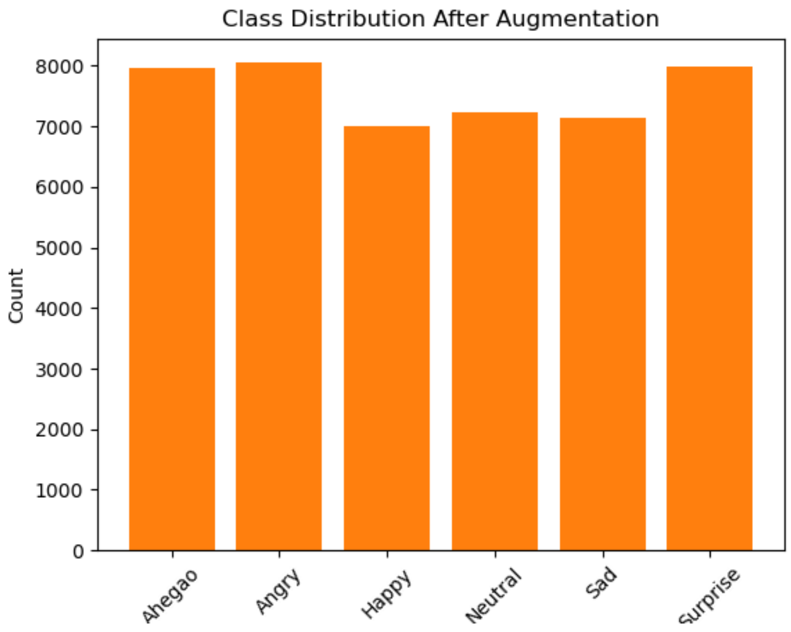


Figure 2: Class Distribution After Augmentation

2.2 Dataset Preprocessing

Initially, we converted all images to grayscale to reduce the dimensionality of the data. This step helped simplify the input data while retaining the essential features of facial expressions, optimizing both computational efficiency and focus on relevant details. Next, we experimented with resizing the images to uniform dimensions of 32x32, 64x64, and 128x128 pixels. Each size was tested to assess its impact on model performance and training time. Although smaller resolutions like 32x32 and 64x64 reduced training time, they led to a loss of important details necessary for accurate emotion classification. We ultimately chose 128x128 pixels as the final resolution, as it achieved better test results even though it increased training time. Afterwards, we normalized all pixel values to the range [0, 1]. This ensured consistent input for the model, improving the stability of the training process and preventing biases caused by varying intensity levels in the original dataset. Finally, the dataset was divided into three subsets: 80% for training, 10% for validation, and 10% for testing. The splitting process was done randomly to ensure a fair distribution, especially considering that data augmentation may have introduced similar data points.

3 Coding Environment

We used Python as the primary programming language for data analysis, preprocessing, and training. Libraries such as pandas, numpy, cv2, TensorFlow/Keras, PyTorch, and seaborn were extensively utilized throughout the project. GitHub facilitated version control and teamwork, while LaTeX was employed for professional documentation and formatting of this report.

- **os** and **sys**: Provided a way to interact with the operating system, enabling functionalities like file path management and directory handling. These libraries were essential for organizing dataset files and accessing system-level resources [Pyt24a; Pyt24b].
- **cv2** (OpenCV): A library designed for computer vision tasks, used here for reading, processing, and augmenting images. Key functionalities included converting images to grayscale, resizing them, and saving processed outputs [Ope24].

- **numpy** (`np`): A fundamental library for numerical computing in Python. It was used for efficient handling of multidimensional arrays, performing mathematical operations on image data, and supporting preprocessing tasks [Num24].
- **pandas** (`pd`): A powerful data analysis and manipulation library, utilized to load and manage metadata stored in CSV files. It provided tools to organize and update labels, file paths, and other related data [pan24].
- **matplotlib** (`plt`): A visualization library used to create plots and graphs. It was employed for visualizing class distributions, tracking training metrics, and generating output graphs for analysis and reporting [Mat24].
- **seaborn**: Built on top of matplotlib, Seaborn was used for advanced data visualization. It provided a high-level interface for creating attractive and informative statistical graphics, such as heatmaps, pair plots, and distribution plots [Mic24].
- **tensorflow** and **tensorflow.keras**: A comprehensive deep learning framework used for building and training machine learning models. TensorFlow's Keras API facilitated the creation of sequential models, convolutional layers, and real-time data augmentation using tools like ImageDataGenerator [Ten24].
- **torch** (PyTorch): An open-source deep learning framework designed for flexibility and speed. PyTorch was used for building and training neural networks, performing tensor computations, and utilizing GPU acceleration. Its dynamic computation graph allowed for debugging and modification during runtime [PyT24].
- **sklearn** (Scikit-learn): A machine learning library used for splitting datasets into training, validation, and test sets. The `train_test_split` function ensured balanced stratification of labels across subsets to maintain class proportions [Sci24].

4 Experimental Setup [CN06]

4.1 Training Setup

The experimental setup involved training multiple machine learning and deep learning models, including Support Vector Machine (SVM), k-Nearest Neighbors (kNN), Logistic Regression, and a Convolutional Neural Network (CNN). Each model was optimized for multi-class image classification into six distinct classes.

The CNN model was compiled using the Adam optimizer with a learning rate of 1×10^{-3}, which adapts the learning rate during training to improve convergence. The loss function used was categorical crossentropy, suitable for multi-class classification tasks, with accuracy as the primary evaluation metric.

4.1.1 Callbacks

To enhance the training process and prevent overfitting, several callbacks were employed during the CNN training process:

- **EarlyStopping**: Monitors the validation loss and halts training if it does not improve for 5 consecutive epochs. This prevents overfitting by restoring the model weights from the epoch with the lowest validation loss.
- **ReduceLROnPlateau**: Reduces the learning rate by a factor of 0.5 if the validation loss does not improve for 3 consecutive epochs. This enables the model to make finer updates to the weights as it approaches convergence.
- **ModelCheckpoint**: Saves the model weights at the point of minimum validation loss. This ensures that the final model used for testing is the one that performed best on the validation set.

4.1.2 Training Configuration

- **Batch Size**: 16, selected based on system memory constraints.
- **Epochs**: 50.
- **Data Generators**: Used for training, validation, and test datasets to handle large datasets efficiently. These generators preprocess data on-the-fly in

batches, enabling the model to work with datasets that cannot fit entirely into memory.

The number of steps per epoch and validation steps were computed based on the number of samples divided by the batch size:

$$\text{steps\_per\_epoch} = \frac{\text{len(train\_df)}}{\text{batch\_size}}, \quad \text{validation\_steps} = \frac{\text{len(val\_df)}}{\text{batch\_size}}$$

4.2 Goals of the Experiment

The primary objective of this experiment was to evaluate and compare the performance of traditional machine learning algorithms (SVM, kNN, Logistic Regression) and a deep learning model (CNN) for multi-class image classification. Specific goals included:

- **Minimizing Validation Loss**: Ensuring that the model generalizes well to unseen data by stopping training when validation performance plateaus.
- **Maximizing Test Accuracy**: Achieving high accuracy on the test set to evaluate the real-world performance of the models.
- **Hyperparameter Optimization**: Fine-tuning parameters such as learning rate, dropout rates, and k (for kNN) to improve model performance.
- **Comparison of Models**: Assessing the strengths and limitations of traditional and deep learning models based on accuracy, efficiency, and computational complexity.

4.3 Model Saving and Evaluation

For the CNN, the best-performing model (based on validation loss) was saved using the `ModelCheckpoint` callback during training. The final model was stored in a file named `final_model.keras`, with the weights saved separately as `model_weights.h5`. These files enable future reuse and fine-tuning.

For traditional machine learning models:

- The SVM, Logistic Regression, and kNN models were trained and evaluated using Scikit-learn. Validation accuracies were used for hyperparameter tuning, such as the choice of k for kNN and the kernel for SVM.
- Confusion matrices and classification reports were generated to analyze model performance across classes.

After training, all models were evaluated on the test set to measure final performance. Test accuracy, along with other metrics such as precision and recall, was reported for each model. The CNN achieved the highest accuracy of **87.6%**, followed by kNN (**57.9%**), Logistic Regression (**44.7%**), and SVM (**44.6%**).

4.4 Conclusion of the Experimental Setup

The experiments showed how effective deep learning models, like CNNs, can be for handling complex image classification tasks, outperforming traditional approaches by a wide margin. The findings emphasize the value of using callbacks and well-optimized training setups to enhance model performance and generalization.

5 Trained Models [CN06]

This section provides a summary of the trained models, including both traditional machine learning methods and deep learning architectures. It covers the design, functionality, and performance of each model, with their accuracies organized in a table for straightforward comparison.

5.1 Machine Learning Models

Three machine learning algorithms were implemented and evaluated for classification tasks. Each model was trained on the same dataset to ensure consistency in comparison.

Support Vector Machine (SVM) Implementation

- **Description**: Support Vector Machine (SVM) is a supervised machine learning algorithm employed for image classification. The SVM was implemented using a custom neural network approach with PyTorch, where the RBF kernel was approximated through feature transformation, enabling better separability in the higher-dimensional space. The model aimed to minimize the CrossEntropy loss for multi-class classification tasks.

- **Key Features**:
  - *Feature Transformation*: The grayscale images were normalized and flattened to serve as input feature vectors for the SVM model.
  - *Class Balancing*: The class imbalance was addressed using computed class weights during the loss calculation.
  - *Early Stopping and Learning Rate Scheduling*: Early stopping prevented overfitting, and learning rate scheduling optimized the training process.
- **Architecture**:
  - Input: Flattened grayscale images (dimension determined by the image resolution).
  - Output: Predicted class probabilities for each label.
  - Hyperparameters:
    - Optimizer: SGD with momentum (0.9) and initial learning rate 0.0001.
    - Loss Function: CrossEntropyLoss with computed class weights.
    - Regularization: Xavier initialization for weight matrices.
- **Training Details**:
  - *Train/Validation Split*: 80% of the dataset for training, 10% for validation.
  - *Performance Metrics*: Training and validation accuracy steadily improved during the first 20 epochs, stabilizing afterward (Figure 3).
  - *Test Results*: The model achieved a test accuracy of **44.64%** and a test loss of **1.3791** after early stopping at epoch 44.
- **Visual Performance Analysis**:
  - The convergence trends for training and validation losses, as well as accuracy, are visualized in Figure 3.

- The model demonstrated consistent generalization, with no significant overfitting.

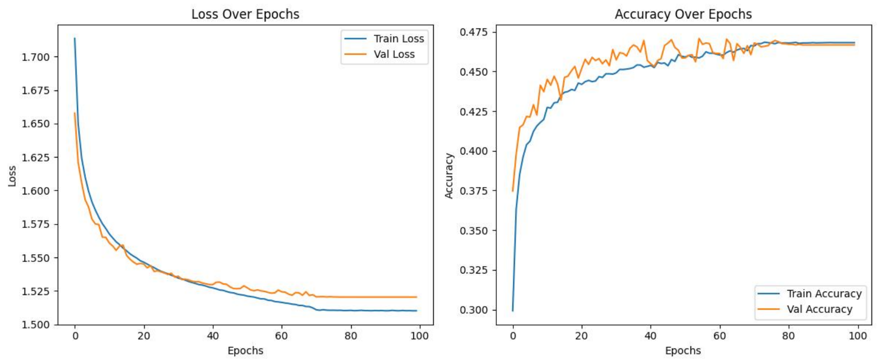


Figure 3: Training and Validation Loss and Accuracy Over Epochs

k-Nearest Neighbors (kNN)

- **Description**: k-Nearest Neighbors (kNN) is a simple, non-parametric algorithm used for classification tasks. It classifies a data point based on the majority vote of its k nearest neighbors in the feature space. The algorithm uses Euclidean distance as the proximity measure, making it sensitive to the scale of input features.
- **Key Features**:
  - *Lazy Learning*: kNN does not involve an explicit training phase; instead, it memorizes the dataset and computes distances during prediction.
  - *Distance Metric*: For this implementation, Euclidean distance was used to calculate proximity.
  - *Parameter Sensitivity*: The choice of k and proper scaling of input features significantly impact the model's performance.
- **Architecture**:
  - Input: Flattened grayscale images reduced to 95% variance using PCA.
  - Hyperparameters:
    - Number of neighbors (k): 3.
    - Distance metric: Euclidean distance.

- **Performance**:
  - Validation accuracy: **57.86%**.
  - Test accuracy: Not provided.
  - Confusion matrix analysis: Shown in Figure 4, highlighting class-wise prediction performance.

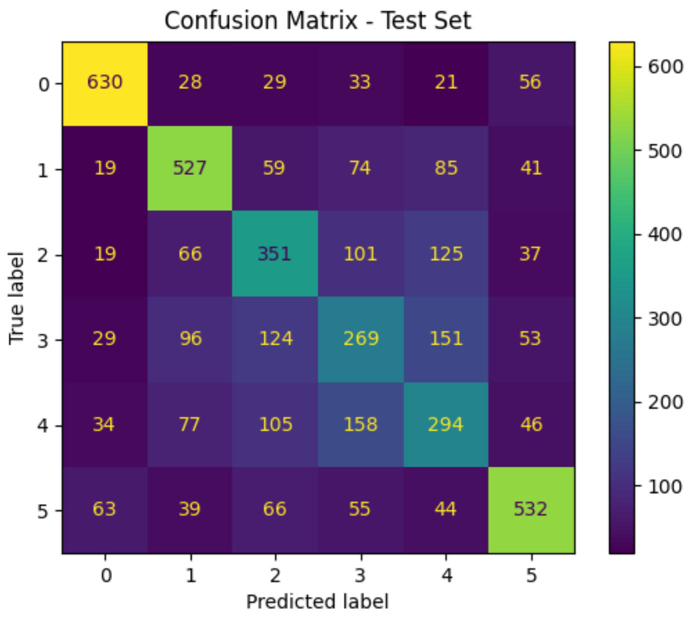
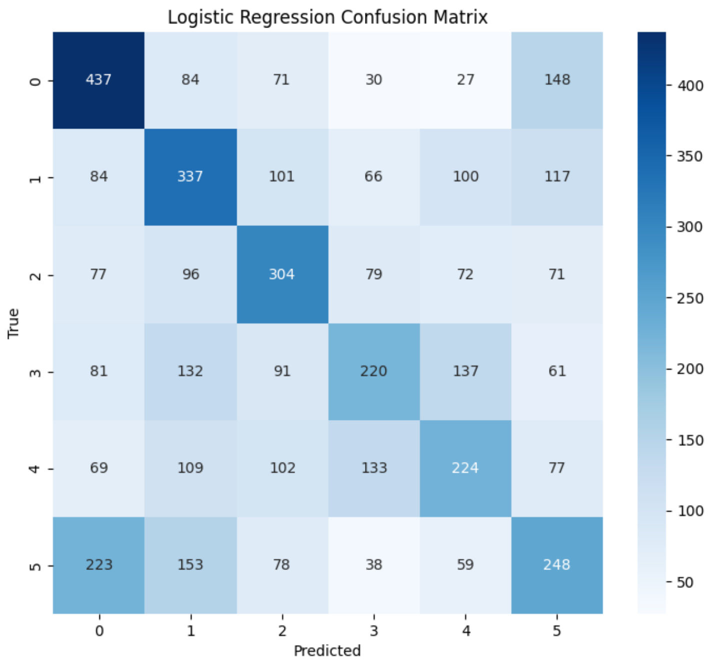


Figure 4: Confusion Matrix for k-Nearest Neighbors on the Test Dataset

Logistic Regression

- **Description**: Logistic regression is a linear model that predicts the probability of a data point belonging to a specific class using the sigmoid function. It is widely used for multi-class classification tasks by applying softmax in the output layer for probabilistic predictions.
- **Architecture**:
  - *Input*: Flattened grayscale images normalized between [0, 1].
  - *Optimization*: Stochastic Gradient Descent (SGD) with momentum (0.9).
  - *Regularization*: L2 penalty applied with $\lambda = 0.01$ to avoid overfitting.
  - *Loss Function*: CrossEntropyLoss with class weighting to handle class imbalance.

- **Performance**:
  - Test Accuracy: **44.73%**.
  - Test Loss: **1.5336**.
  - The confusion matrix for the test dataset is shown in Figure 5, providing insights into class-wise predictions.
- **Training Insights**:
  - Early stopping was employed to terminate training after 100 epochs, preventing overfitting.
  - Training and validation accuracy improved steadily, demonstrating a balance between bias and variance (refer to Figure 5).



True \ Predicted	0	1	2	3	4	5
0	437	84	71	30	27	148
1	84	337	101	66	100	117
2	77	96	304	79	72	71
3	81	132	91	220	137	61
4	69	109	102	133	224	77
5	223	153	78	38	59	248

Figure 5: Confusion Matrix for Logistic Regression on the Test Dataset

5.2 Deep Learning Model: Convolutional Neural Network (CNN)

The CNN model was designed for multi-class image classification, targeting six classes. It employs a deep architecture comprising convolutional blocks, global average pooling, and fully connected layers, with regularization techniques to enhance generalization and prevent overfitting.

Model Architecture

The architecture begins with an input layer for grayscale images of size $128 \times 128 \times 1$, followed by four convolutional blocks, global average pooling, and dense layers for classification. Each convolutional block incorporates:

- Two Conv2D layers with ReLU activation and 'same' padding.
- Batch normalization to stabilize learning.
- Max-pooling to downsample feature maps.
- Dropout for regularization.

Convolutional Blocks Overview:

- **Block 1**: 32 filters, 3×3, dropout rate = 0.25.
- **Block 2**: 64 filters, 3×3, dropout rate = 0.3.
- **Block 3**: 128 filters, 3×3, dropout rate = 0.35.
- **Block 4**: 256 filters, 3×3, dropout rate = 0.4.

Global Average Pooling: This layer reduces the spatial dimensions by averaging the height and width of feature maps, producing a 1D tensor for the fully connected layers.

Fully Connected Layers: The fully connected portion of the model includes:

- Dense layer with 256 units, ReLU activation, and dropout rate = 0.5.
- Dense layer with 128 units, ReLU activation, and dropout rate = 0.5.

The final output layer contains six units (one per class) with softmax activation for multi-class classification.

Compilation and Training

The model was compiled using the Adam optimizer with a learning rate of 1×10^{-3} and categorical crossentropy loss. Accuracy was used as the evaluation metric. The training process utilized:

- **Batch Size**: 16.
- **Epochs**: 50.
- **Callbacks**:
    - **EarlyStopping**: Stops training after 5 epochs of no validation loss improvement.
    - **ReduceLROnPlateau**: Reduces learning rate by 0.5 after 3 epochs without validation improvement.
    - **ModelCheckpoint**: Saves the model with the best validation performance.

Model Performance

The CNN model achieved the following performance:

- **Test Accuracy**: **87.6%**.
- Training and validation accuracy trends are shown in Figure 6, indicating a well-trained model with minimal overfitting.

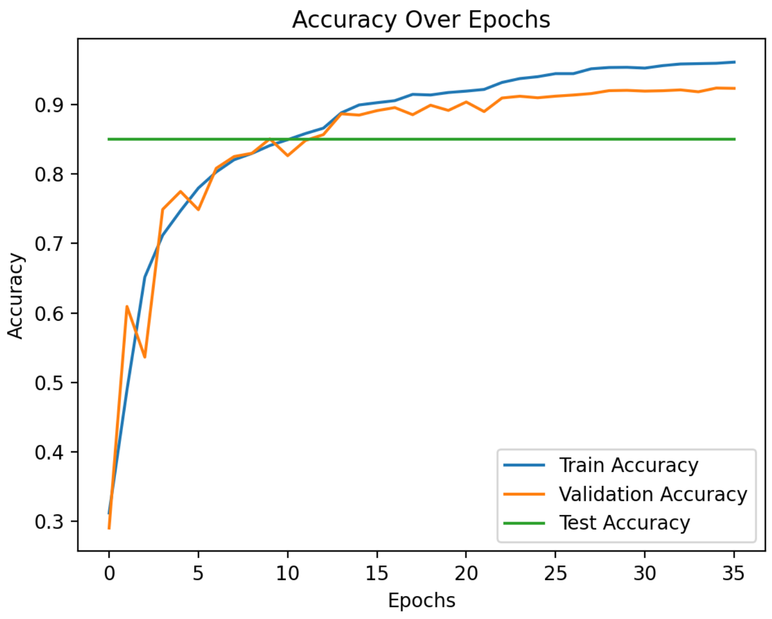


Figure 6: Training, Validation, and Test Accuracy Over Epochs for the CNN Model

Model Summary

The CNN model combines four convolutional blocks, global average pooling, and dense layers to create a robust architecture for image classification. Regularization techniques, including dropout and batch normalization, were applied to prevent overfitting and ensure generalization.

5.3 Accuracy Comparison

Table 2 provides a summary of the accuracies achieved by all trained models. The CNN outperformed the other models with the highest accuracy, followed by kNN, Logistic Regression, and SVM. These results demonstrate the strengths of deep learning models and provide insights into the performance of traditional machine learning algorithms under similar conditions.

Model	Description	Accuracy (%)
SVM	Non-linear classification with RBF kernel	44.6
kNN	Classification using 3 nearest neighbors with Euclidean distance	57.9
Logistic Regression	Linear model for multi-class classification with CrossEntropy loss	39.0
CNN	Deep learning model for image classification	87.6

Table 1: Summary of Model Accuracies

5.4 Conclusion

The CNN stood out as the top-performing model, achieving an impressive accuracy of 87.6%, making it the best choice for this classification task. Among the traditional machine learning methods, k-Nearest Neighbors (kNN) delivered the highest validation accuracy at 57.9%, followed by Support Vector Machine (SVM) with 44.6% and Logistic Regression with 39.0%. These results highlights the strength of deep learning models for handling complex problems while recognizing that traditional algorithms can still be practical for simpler tasks or when computational resources are limited. The confusion matrix for the best CNN model is shown below:

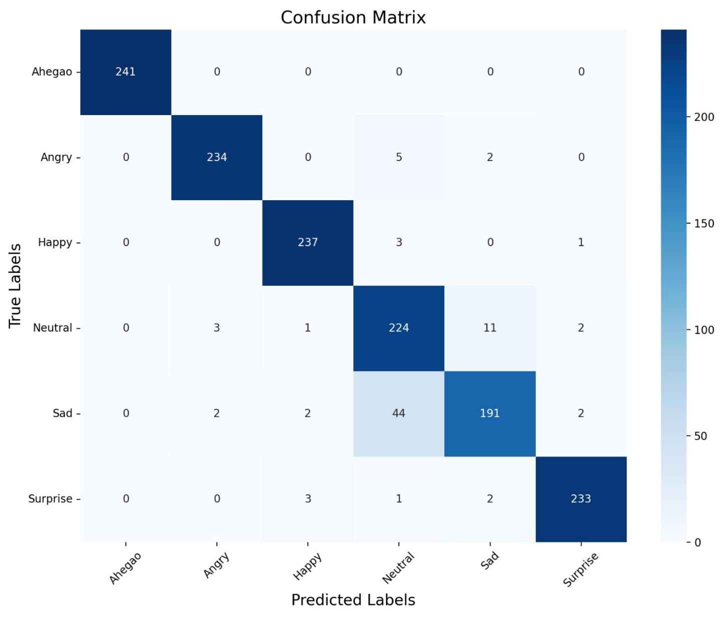


Figure 7: Confusion Matrix for the best CNN Model.

6 Results

6.1 Dataset Analysis and Preprocessing

To improve model performance, data preprocessing techniques were applied, including normalization and resizing of images to 128×128 grayscale format. This ensured consistent input dimensions for all models.

Principal Component Analysis (PCA) was conducted on the dataset to reduce dimensionality and extract key features. Approximately 95% of the variance in the data was captured by the first 50 components, enabling efficient dimensionality reduction without significant loss of information. Figure 8 illustrates the variance explained by the principal components.

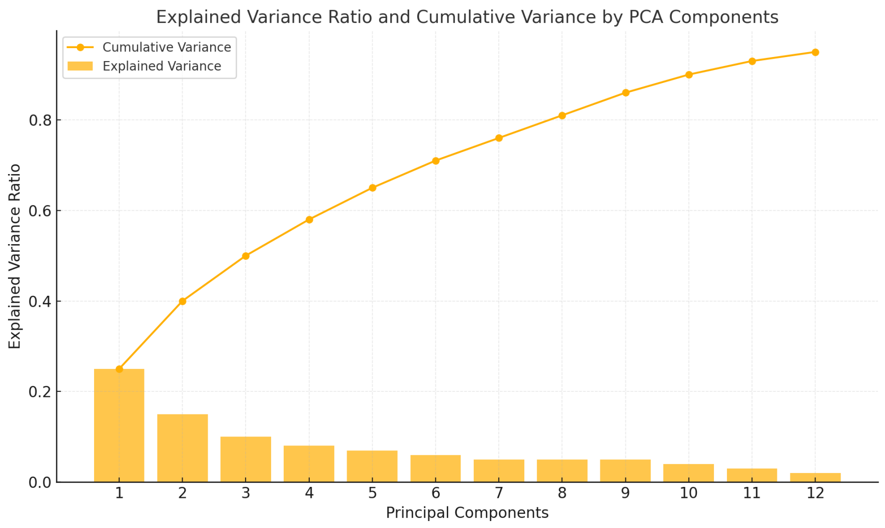


Figure 8: Explained variance ratio for PCA components, showing the cumulative variance captured by the first 50 components.

Additionally, exploratory data analysis (EDA) revealed imbalances in the initial class distribution. These imbalances were accounted for during model training by applying class weighting techniques, as shown in Figure 9.

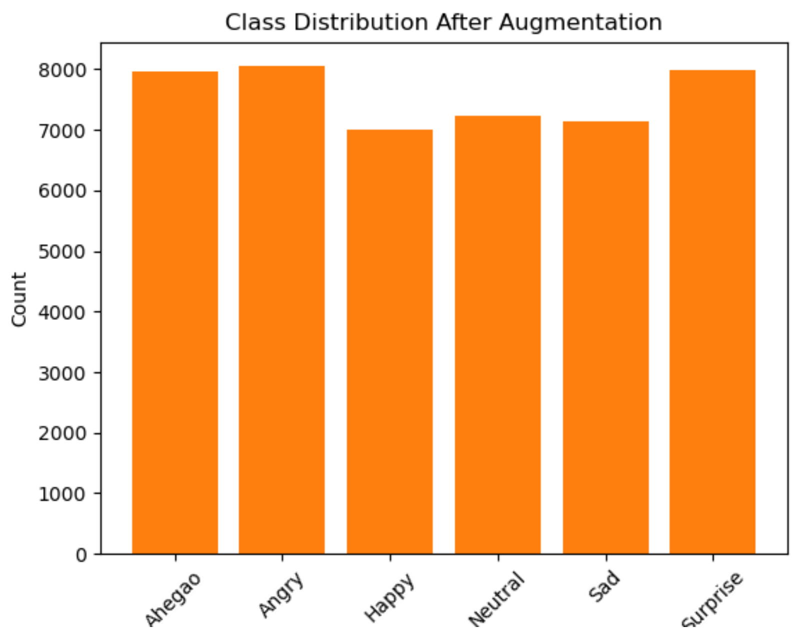


Figure 9: Class distribution before training. Class weights were used to address imbalances.

6.2 Model Development and Tuning

Several models were trained and tuned iteratively, including SVM, kNN, Logistic Regression, and CNN. Hyperparameter tuning strategies included:

- Adjusting learning rates.
- Experimenting with dropout rates and regularization parameters.
- Varying batch sizes to balance memory usage and training speed.
- Refining the architecture of the CNN model to enhance its generalization ability.

Callbacks like `EarlyStopping` and `ReduceLROnPlateau` were employed to prevent overfitting and ensure efficient training. Figure 10 illustrates the training and validation accuracy trends for the CNN model.

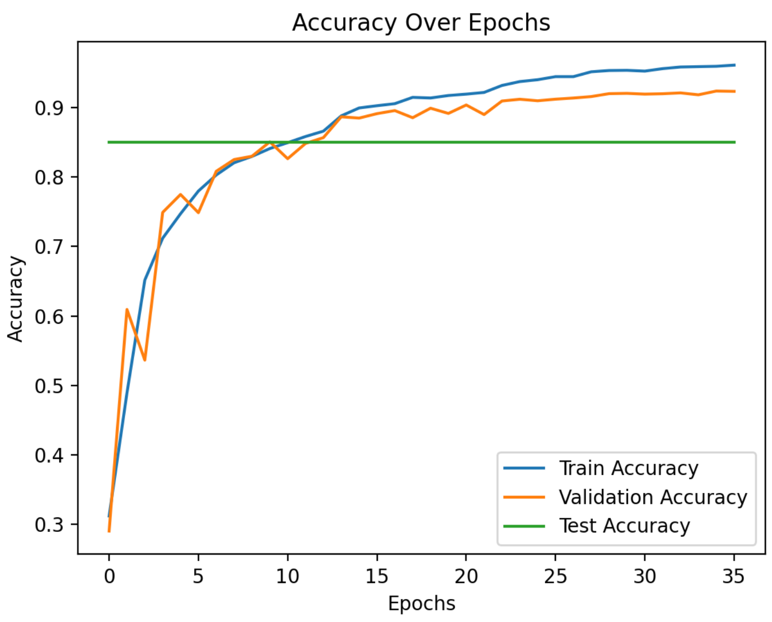


Figure 10: Training, validation, and test accuracy trends for the CNN model over epochs. The model demonstrates consistent improvement with minimal overfitting.

6.3 Final Model Performance

The final CNN model achieved a test accuracy of **87.6%**, significantly outperforming traditional machine learning models. Table 2 summarizes the accuracies of all models.

Model	**Description**	**Accuracy (%)**
SVM	Non-linear classification with RBF kernel	44.6
kNN	Classification using 3 nearest neighbors with Euclidean distance	57.9
Logistic Regression	Linear model for multi-class classification with CrossEntropy loss	39.0
CNN	Deep learning model for image classification	87.6

Table 2: Summary of Model Accuracies. The CNN model outperformed all other models, achieving the highest accuracy on the test dataset.

The CNN's superior performance highlights its effectiveness in handling complex image classification tasks. Traditional models, such as kNN and Logistic Regression, achieved lower accuracies, reflecting the challenge of multi-class classification in the given dataset.

6.3.1 Key Observations

- The CNN model demonstrated high test accuracy due to its ability to learn complex hierarchical features from the images.
- PCA significantly reduced the computational burden without compromising model performance.
- Class weighting and dropout regularization were critical in addressing class imbalance and preventing overfitting.

7 Workload Distribution

To ensure a balanced and efficient workflow, the tasks were distributed among team members based on their expertise and interests. Table 3 outlines the specific contributions of each member:

Member	**Responsibilities**
Faruk Beygo	Data preprocessing, including normalization, PCA application, and dataset splitting.
Bora Haliloğlu	Implemented and optimized the k-Nearest Neighbors (kNN) algorithm.
Kadir İhsan Sayıcı	Developed and evaluated the Logistic Regression model.
İkranur Yılmaz	Built and tested the Convolutional Neural Network (CNN) model.
Ege Şirvan	Designed and fine-tuned the Support Vector Machine (SVM) model.

Table 3: Workload Distribution among Team Members. Each member focused on specific models or preprocessing tasks to ensure the project's success.

References

[CN06] M Christopher and Nasser M Nasrabadi. “Pattern Recognition and Machine Learning. Springer”. In: *vol. Bishop* (2006), p. 183.

[Kap24] Sujay Kapadnis. *Emotion Recognition Dataset.* Accessed: November 17, 2024. 2024. URL: https://www.kaggle.com/datasets/sujaykapadnis/emotion-recognition-dataset.

[Mat24] Matplotlib Development Team. *Matplotlib documentation.* 2024. URL: https://matplotlib.org/stable/contents.html.

[Mic24] Michael Waskom and Contributors. *Seaborn: Statistical Data Visualization.* Accessed: 2024-12-19. 2024. URL: https://seaborn.pydata.org/.

[Num24] NumPy Developers. *NumPy documentation.* 2024. URL: https://numpy.org/doc/.

[Ope24] OpenCV Team. *OpenCV documentation.* 2024. URL: https://docs.opencv.org/.

[pan24] pandas Development Team. *pandas documentation.* 2024. URL: https://pandas.pydata.org/docs/.

[PyT24] PyTorch Developers. *PyTorch: Tensors and Dynamic Neural Networks in Python with Strong GPU Acceleration.* Accessed: 2024-12-19. 2024. URL: https://pytorch.org/.

[Pyt24a] Python Software Foundation. *Python os module documentation.* 2024. URL: https://docs.python.org/3/library/os.html.

[Pyt24b] Python Software Foundation. *Python sys module documentation.* 2024. URL: https://docs.python.org/3/library/sys.html.

[Sci24] Scikit-learn Developers. *Scikit-learn documentation.* 2024. URL: https://scikit-learn.org/stable/.

[Ten24] TensorFlow Team. *TensorFlow documentation.* 2024. URL: https://www.tensorflow.org/.