Group 2_ML Project_Emotion Classification

June 23, 2025

1 Introduction to the Emotion Classification Project using FER2013 - GROUP 2

1.0.1 Project Summary

Facial expression recognition is a powerful application of computer vision and machine learning, with applications spanning mental health monitoring, human-computer interaction, customer feedback systems, and even security. In this project, we explore various machine learning and deep learning techniques to classify human emotions from facial images using the FER2013 dataset. The project involves a step-by-step implementation of five distinct models: - Logistic Regression (Baseline) - Random Forest - Dense Neural Network (MLP) - Long Short-Term Memory Network (LSTM) - Convolutional Neural Network (CNN)

Each model is evaluated based on accuracy, class-wise performance, confusion matrix analysis, and generalizability.

1.0.2 Project Aim

To compare the performance of traditional machine learning and deep learning models in classifying facial emotions using grayscale images from the FER2013 dataset, and to identify which modeling approach best captures the spatial and expressive patterns in human faces.

1.0.3 Key Objectives

- Load and preprocess the FER2013 dataset in image-folder format
- Conduct exploratory data analysis (EDA) to understand data distribution and image characteristics
- Apply tailored preprocessing techniques for each model type
- Train, evaluate, and compare multiple classification models
- Visualize performance using metrics such as accuracy, F1-score, and confusion matrices
- Derive insights from model behavior, misclassifications, and feature importance

1.0.4 About the Dataset – FER2013

The Facial Expression Recognition 2013 (FER2013) dataset is a widely used benchmark for emotion classification. It was originally published as part of the Kaggle Challenge "Challenges in Representation Learning: Facial Expression Recognition."

Key Properties: Data type: 48x48 grayscale images of human faces - Classes (7 emotions): 1. Angry 2. Disgust 3. Fear 4. Happy 5. Neutral 6. Sad 7. Surprise - Format: Organized in folder structure for train/ and test/, each containing 7 emotion subfolders - Train set: ~28,709 images - Test set: ~7,178 images - Each image is pre-aligned so the face is centered, facilitating consistent modeling and evaluation.

1.0.5 Tools and Technologies Used

- Programming Language: Python
- Libraries:
    1. TensorFlow/Keras – Deep Learning models (CNN, LSTM, MLP)
    2. Scikit-learn – Logistic Regression, Random Forest, evaluation metrics
    3. Matplotlib / Seaborn – Visualization
    4. NumPy / Pandas – Data manipulation
- Environment: Jupyter Notebook
- Image Pipeline: image_dataset_from_directory, manual normalization, and reshaping

1.0.6 Evaluation Metrics

To ensure a comprehensive evaluation of all models, the following metrics are used: - Overall Accuracy - Precision, Recall, F1-Score (macro and weighted) - Confusion Matrix - Training & Validation Curves (loss and accuracy) - Feature Importance (Random Forest) and heatmaps

1.0.7 Challenges Encountered

- Class Imbalance: 'Disgust' had very few samples, leading to poor recall in all models
- Model Scalability: Logistic Regression was too slow on full data — a 20% subset was used
- Overfitting: Dense MLP showed unstable validation accuracy
- Creative Adaptation: LSTM was creatively used by converting images into row-wise sequences

1.1 Step 1: Data Loading and Initial Setup

Goal: Load 48x48 grayscale images from fer2013/train/ and fer2013/test/, normalize pixel values, handle grayscale channel dimensions, prepare train/validation/test sets, and one-hot encode labels for deep learning.

[1]:
```
# Import Libraries
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import os
from tensorflow.keras.utils import to_categorical
from sklearn.preprocessing import LabelEncoder
from pathlib import Path
```

[2]:
```
# Set Parameters
IMG_SIZE = 48  # Width and height
BATCH_SIZE = 64
SEED = 42
DATA_DIR = "fer2013"
```

```
# Emotion categories will be inferred automatically from folder names
```

[3]:
```
# Load training dataset with validation split (before normalization)
raw_train_ds = tf.keras.utils.image_dataset_from_directory(
    directory=os.path.join(DATA_DIR, "train"),
    labels='inferred',
    label_mode='int',  # integer labels
    color_mode='grayscale',
    batch_size=BATCH_SIZE,
    image_size=(IMG_SIZE, IMG_SIZE),
    shuffle=True,
    seed=SEED,
    validation_split=0.2,
    subset='training'
)

raw_val_ds = tf.keras.utils.image_dataset_from_directory(
    directory=os.path.join(DATA_DIR, "train"),
    labels='inferred',
    label_mode='int',
    color_mode='grayscale',
    batch_size=BATCH_SIZE,
    image_size=(IMG_SIZE, IMG_SIZE),
    shuffle=True,
    seed=SEED,
    validation_split=0.2,
    subset='validation'
)

# 4  Load test dataset (no validation split)
raw_test_ds = tf.keras.utils.image_dataset_from_directory(
    directory=os.path.join(DATA_DIR, "test"),
    labels='inferred',
    label_mode='int',
    color_mode='grayscale',
    batch_size=BATCH_SIZE,
    image_size=(IMG_SIZE, IMG_SIZE),
    shuffle=False
)
```

```
Found 28709 files belonging to 7 classes.
Using 22968 files for training.
Found 28709 files belonging to 7 classes.
Using 5741 files for validation.
Found 7178 files belonging to 7 classes.
```

[4]:
```
# Get class names before mapping (IMPORTANT)
class_names = raw_train_ds.class_names
class_indices = dict(zip(class_names, range(len(class_names))))
print("Emotion Label Mapping:", class_indices)
```

```
Emotion Label Mapping: {'angry': 0, 'disgust': 1, 'fear': 2, 'happy': 3,
'neutral': 4, 'sad': 5, 'surprise': 6}
```

[5]:
```
# Normalize datasets using Rescaling
normalization_layer = tf.keras.layers.Rescaling(1./255)

train_ds = raw_train_ds.map(lambda x, y: (normalization_layer(x), y))
val_ds = raw_val_ds.map(lambda x, y: (normalization_layer(x), y))
test_ds = raw_test_ds.map(lambda x, y: (normalization_layer(x), y))
```

[6]:
```
# Convert to NumPy arrays (for use with traditional ML models)
def convert_to_numpy(dataset):
    images = []
    labels = []
    for batch_images, batch_labels in dataset:
        images.append(batch_images.numpy())
        labels.append(batch_labels.numpy())
    return np.concatenate(images), np.concatenate(labels)

X_train_np, y_train_np = convert_to_numpy(train_ds)
X_val_np, y_val_np = convert_to_numpy(val_ds)
X_test_np, y_test_np = convert_to_numpy(test_ds)

print("Train shape:", X_train_np.shape)
print("Validation shape:", X_val_np.shape)
print("Test shape:", X_test_np.shape)
```

```
Train shape: (22968, 48, 48, 1)
Validation shape: (5741, 48, 48, 1)
Test shape: (7178, 48, 48, 1)
```

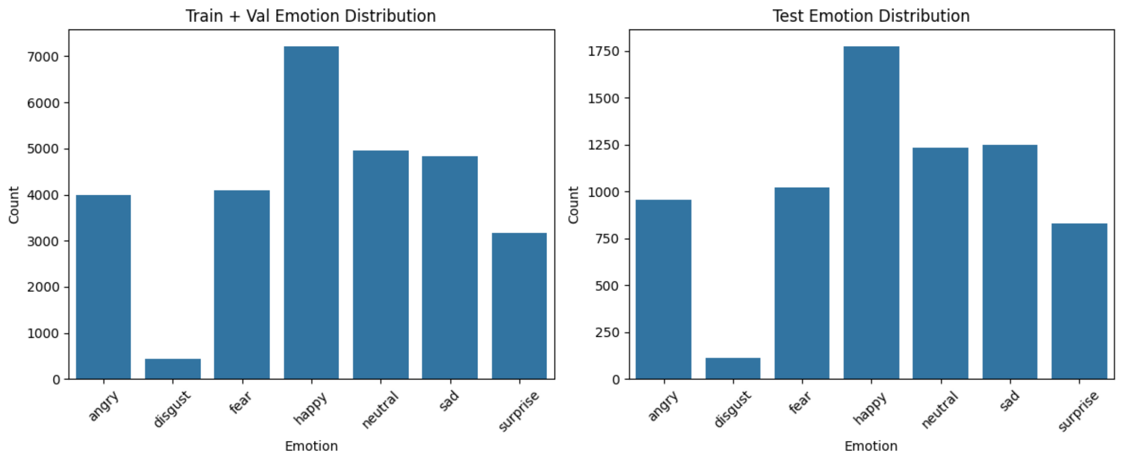
[7]:
```
# One-hot encode labels for deep learning models
y_train_oh = to_categorical(y_train_np, num_classes=7)
y_val_oh = to_categorical(y_val_np, num_classes=7)
y_test_oh = to_categorical(y_test_np, num_classes=7)
```

1.2 Step 2: Exploratory Data Analysis (EDA)

We'll perform 3 main tasks: 1. Emotion Distribution – to visualize class balance. 2. Sample Images Grid – to understand visual patterns. 3. Pixel Value Distribution – to explore grayscale intensity characteristics.

Emotion Distribution (Class Balance)

[8]:
```
import seaborn as sns
import pandas as pd

# Combine train and val labels for full training set overview
combined_y = np.concatenate([y_train_np, y_val_np])

# Create dataframes
train_df = pd.DataFrame({'emotion': combined_y})
test_df = pd.DataFrame({'emotion': y_test_np})

# Plot class distributions
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
sns.countplot(data=train_df, x='emotion')
plt.title("Train + Val Emotion Distribution")
plt.xticks(ticks=range(len(class_names)), labels=class_names, rotation=45)
plt.xlabel("Emotion")
plt.ylabel("Count")

plt.subplot(1, 2, 2)
sns.countplot(data=test_df, x='emotion')
plt.title("Test Emotion Distribution")
plt.xticks(ticks=range(len(class_names)), labels=class_names, rotation=45)
plt.xlabel("Emotion")
plt.ylabel("Count")

plt.tight_layout()
plt.show()
```

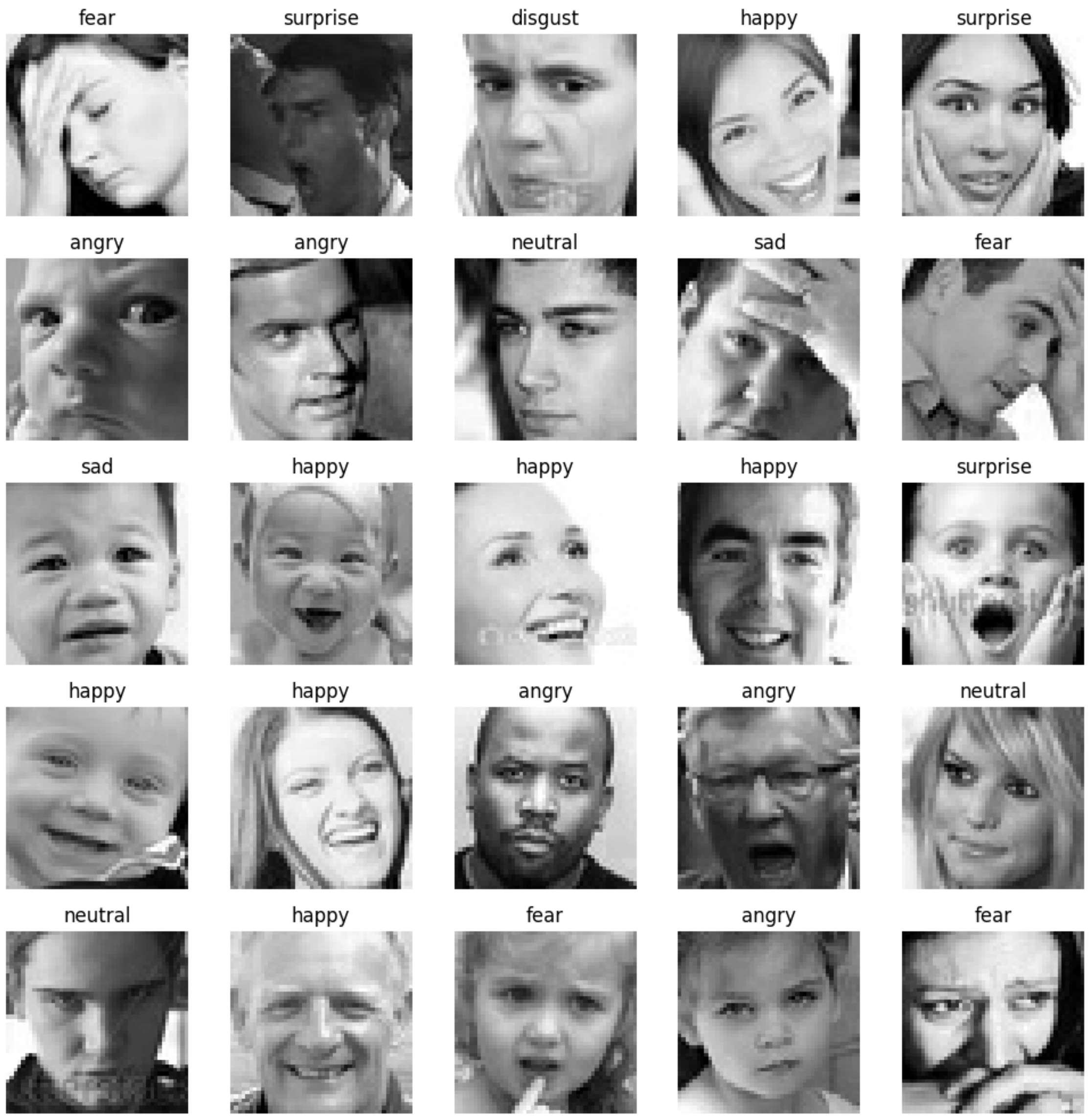


Insights:

- We see that disgust has significantly fewer samples than others.
- happy, neutral, and sad are typically the most common.

Display Sample Images (5×5 Grid)

[9]:
```python
def plot_sample_images(images, labels, class_names, num_samples=25):
    plt.figure(figsize=(10, 10))
    indices = np.random.choice(len(images), num_samples, replace=False)
    for i, idx in enumerate(indices):
        plt.subplot(5, 5, i + 1)
        plt.imshow(images[idx].squeeze(), cmap='gray')
        plt.title(class_names[labels[idx]])
        plt.axis("off")
    plt.tight_layout()
    plt.show()

plot_sample_images(X_train_np, y_train_np, class_names)
```

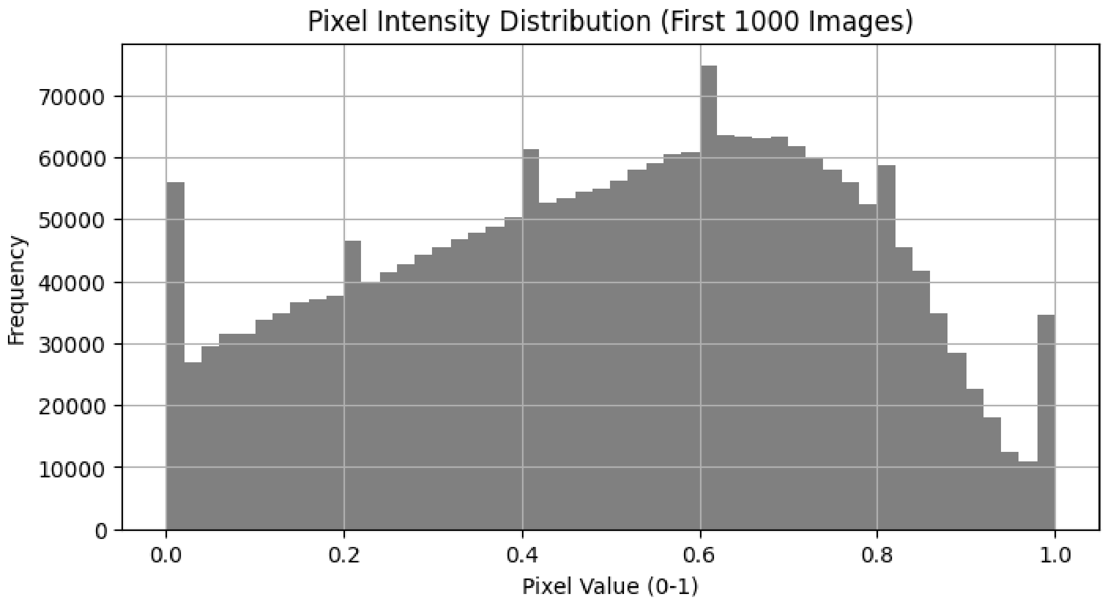


Insights:

- This gives a sense of how well-aligned and consistent the images are.
- Note the variations in facial expressions and lighting.

Pixel Intensity Distribution

[10]:
```
# Analyze grayscale intensity values across a subset (e.g., 1000 images).

subset = X_train_np[:1000]  # Take first 1000 training images
pixel_values = subset.flatten()

plt.figure(figsize=(8, 4))
plt.hist(pixel_values, bins=50, color='gray')
plt.title("Pixel Intensity Distribution (First 1000 Images)")
plt.xlabel("Pixel Value (0-1)")
plt.ylabel("Frequency")
plt.grid(True)
plt.show()
```



Insights:

1. Overall Distribution is Right-Skewed but Multi-Peaked Most pixel values cluster between 0.3 to 0.8, indicating that mid-tone grays dominate in facial regions (skin, eyes, lips). We see peaks at extreme ends (0.0 and 1.0):

- 0.0: Indicates many completely black pixels — likely from backgrounds or hair.
- 1.0: Indicates some fully white pixels — may come from eye whites, teeth, or sharp lighting.

2. Noticeable Spikes at Regular Intervals

- The histogram has distinct vertical bars or spikes.
- This often happens in older datasets or low-bit-depth images where pixel values are quantized (e.g., only certain levels like 0.2, 0.4, 0.6 are allowed).
- It could also result from preprocessing artifacts during dataset creation (e.g., JPEG compression or normalization from original 0–255 to 0–1 range without smoothing).

3. Implication for Modeling

- Our model will benefit from normalization, which you've already done (Rescaling(1./255)).
- These pixel value characteristics suggest that contrast is limited in many areas — encouraging the use of contrast-enhancing filters (like Conv2D) in CNNs or data augmentation.

1.3 Step 3: Preprocessing for Different Model Types

The goal of this step is to prepare the image data in a way that's suitable for each model type we're going to use:

Models we're targeting:

- Traditional ML: Logistic Regression, SVM, Random Forest
- Basic Neural Network (Dense MLP)
- Convolutional Neural Network (CNN)
- Recurrent Neural Network (LSTM)

3.1 For Traditional ML and Dense MLP

- Models: Logistic Regression, SVM, Random Forest, Dense MLP
- Requirement: Flatten each 48x48 image into a 1D vector of 2304 features.

Why? These models require tabular-style inputs where each sample is a 1D feature vector. They don't natively understand spatial structures like CNNs do.

[11]:
```
# Flatten X data for ML and Dense MLP models
X_train_flat = X_train_np.reshape((X_train_np.shape[0], -1))  # shape:␣
 ↪(n_samples, 2304)
X_val_flat = X_val_np.reshape((X_val_np.shape[0], -1))
X_test_flat = X_test_np.reshape((X_test_np.shape[0], -1))

print("Flat Train shape:", X_train_flat.shape)
```

```
Flat Train shape: (22968, 2304)
```

3.2 For CNN

- Model: Convolutional Neural Network
- Requirement: Data must be in shape (48, 48, 1) — 3D image tensors with 1 grayscale channel.

"Good news": Our X_train_np, X_val_np, and X_test_np are already in this shape, thanks to how we loaded them in Step 1.

Data Augmentation for CNN

- Helps prevent overfitting by simulating new training examples.
- Simulates real-world variation in facial expressions.
- Makes the model more robust to position, scale, and orientation.

[12]:
```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Use this only during model training
cnn_augmentation = ImageDataGenerator(
    rotation_range=10,
    width_shift_range=0.1,
    height_shift_range=0.1,
    zoom_range=0.1,
    horizontal_flip=True
)
```

3.3 For LSTM (Creative Adaptation)

Model: LSTM

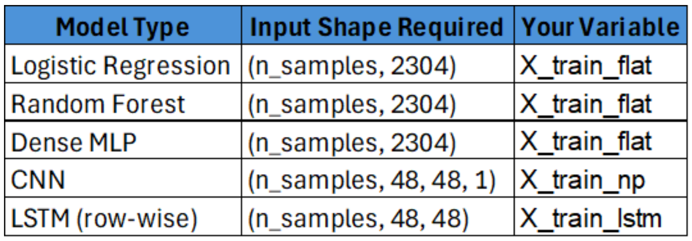
- Requirement: Reshape each image as a sequence of 48 time steps, where each time step is a row of 48 pixels.
- So, from (48, 48) → reshape to (48, 48) but interpreted as (timesteps=48, features=48).

[13]:
```
# Reshape image data for LSTM input
X_train_lstm = X_train_np.reshape((X_train_np.shape[0], IMG_SIZE, IMG_SIZE))
X_val_lstm = X_val_np.reshape((X_val_np.shape[0], IMG_SIZE, IMG_SIZE))
X_test_lstm = X_test_np.reshape((X_test_np.shape[0], IMG_SIZE, IMG_SIZE))

print("LSTM input shape:", X_train_lstm.shape)
```

```
LSTM input shape: (22968, 48, 48)
```

Limitations of LSTM for Images:

- Treating each row as a timestep ignores spatial relationships vertically.
- Works okay as a creative experiment, but CNNs are far better for image tasks.
- Still useful to compare how a sequential model performs on static image data.



Model Type	Input Shape Required	Your Variable
Logistic Regression	(n_samples, 2304)	X_train_flat
Random Forest	(n_samples, 2304)	X_train_flat
Dense MLP	(n_samples, 2304)	X_train_flat
CNN	(n_samples, 48, 48, 1)	X_train_np
LSTM (row-wise)	(n_samples, 48, 48)	X_train_lstm

Summary Table:

1.4 Step 4: Model Implementations

1.4.1 4.1: Baseline Model – Logistic Regression

Input Requirement:

- Use X_train_flat (shape: n_samples × 2304)
- Labels: y_train_np (integer labels 0–6)

[14]:
```
# Import necessary modules
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix,␣
 ↪accuracy_score
import seaborn as sns
import matplotlib.pyplot as plt
import time
```

[15]:
```
# Reduce dataset (otherwise, training taking too long, like 40-60 mins)
X_train_small, _, y_train_small, _ = train_test_split(
    X_train_flat, y_train_np,
    train_size=0.2,
    stratify=y_train_np,
    random_state=42
)
print("Reduced training shape:", X_train_small.shape)
```

```
Reduced training shape: (4593, 2304)
```

[22]:
```
from sklearn.preprocessing import StandardScaler  # ← Add this import
from sklearn.decomposition import PCA

# Scale features (fit on train, transform train+test)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_small)
X_test_scaled  = scaler.transform(X_test_flat)

# Dimensionality reduction with PCA
pca = PCA(n_components=300, random_state=42)
X_train_pca = pca.fit_transform(X_train_scaled)
X_test_pca  = pca.transform(X_test_scaled)
print("PCA training shape:", X_train_pca.shape)
```

```
PCA training shape: (4593, 300)
```

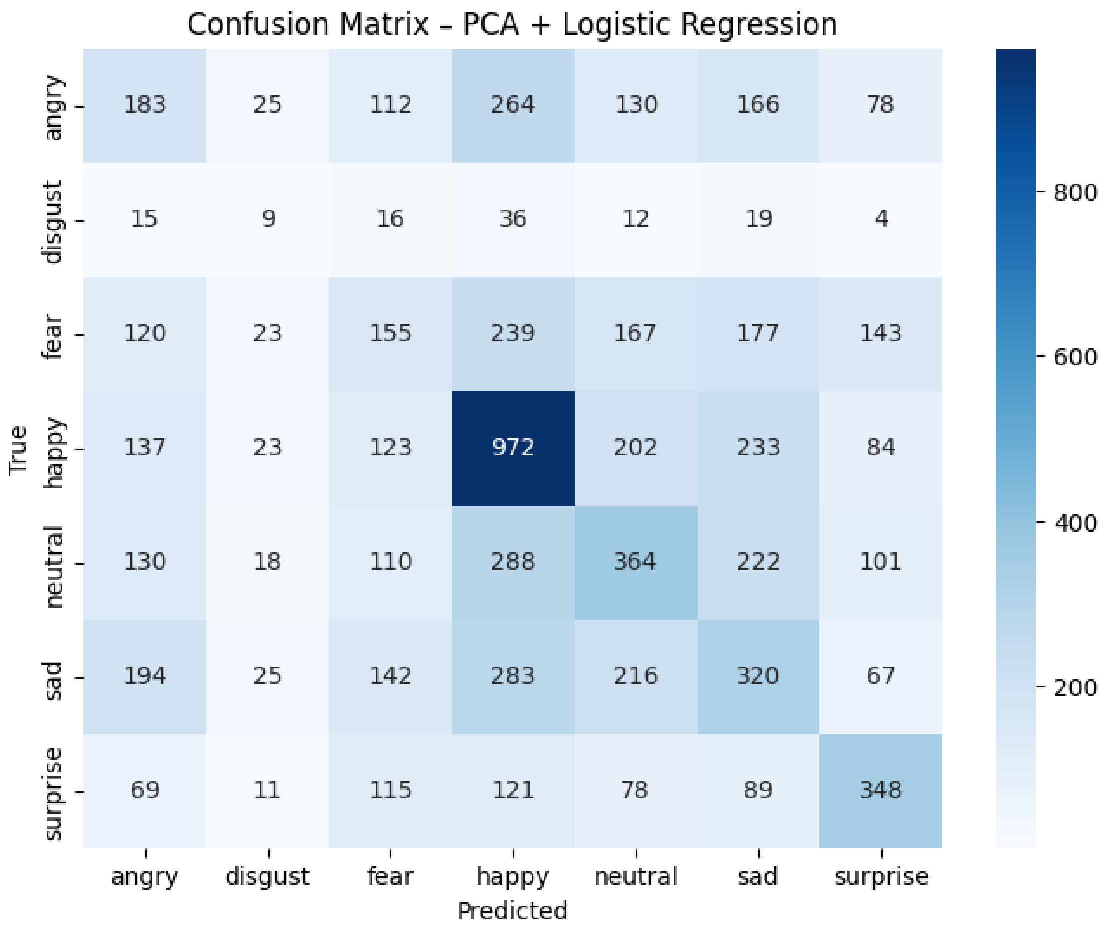
[24]:
```
# Train Logistic Regression on PCA-reduced data
lr_model = LogisticRegression(
    multi_class='multinomial',
    solver='lbfgs',
    max_iter=2000,    # increase if you still see convergence warnings
    tol=1e-3,
    verbose=1
)

start = time.time()
lr_model.fit(X_train_pca, y_train_small)
end = time.time()
print(f"Training completed in {end - start:.2f} seconds.")
```

```
Training completed in 2.06 seconds.
```

[25]:
```
# Predict and Evaluate
y_pred = lr_model.predict(X_test_pca)

# Overall accuracy
print("Test Accuracy:", accuracy_score(y_test_np, y_pred))

# Full classification report
print("\nClassification Report:")
print(classification_report(y_test_np, y_pred, target_names=class_names,␣
↪zero_division=0))
```

```
Test Accuracy: 0.3275285594873224

Classification Report:
              precision    recall  f1-score   support

       angry       0.22      0.19      0.20       958
     disgust       0.07      0.08      0.07       111
        fear       0.20      0.15      0.17      1024
       happy       0.44      0.55      0.49      1774
     neutral       0.31      0.30      0.30      1233
         sad       0.26      0.26      0.26      1247
    surprise       0.42      0.42      0.42       831

    accuracy                           0.33      7178
   macro avg       0.27      0.28      0.27      7178
weighted avg       0.32      0.33      0.32      7178
```

[26]:
```
# Confusion matrix visualization
cm = confusion_matrix(y_test_np, y_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",
            xticklabels=class_names, yticklabels=class_names)
plt.xlabel("Predicted")
plt.ylabel("True")
plt.title("Confusion Matrix - PCA + Logistic Regression")
plt.show()
```



1.4.2 4.2: Random Forest Classifier (Traditional ML)

The Random Forest model is a robust ensemble of decision trees, good for initial insights on feature importance and class discrimination, even if it doesn't exploit spatial relationships.

Requirements:

- Input: X_train_flat, X_test_flat (flattened 2304-dim vectors)
- Labels: y_train_np, y_test_np (integer labels 0–6)

[27]:
```
# Train Random Forest Classifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, confusion_matrix,␣
  ↪accuracy_score
import seaborn as sns
import matplotlib.pyplot as plt
import time
```

```
# Optionally use a smaller subset (if memory-limited)
# X_train_rf, _, y_train_rf, _ = train_test_split(X_train_flat, y_train_np,␣
↪train_size=0.5, stratify=y_train_np)

# Initialize and Train
start = time.time()

rf_model = RandomForestClassifier(
    n_estimators=100,       # Number of trees
    max_depth=None,         # Allow full tree depth
    random_state=42,
    n_jobs=-1,              # Use all CPU cores
    verbose=1
)

rf_model.fit(X_train_flat, y_train_np)

end = time.time()
print(f"\nTraining completed in {end - start:.2f} seconds.")
```

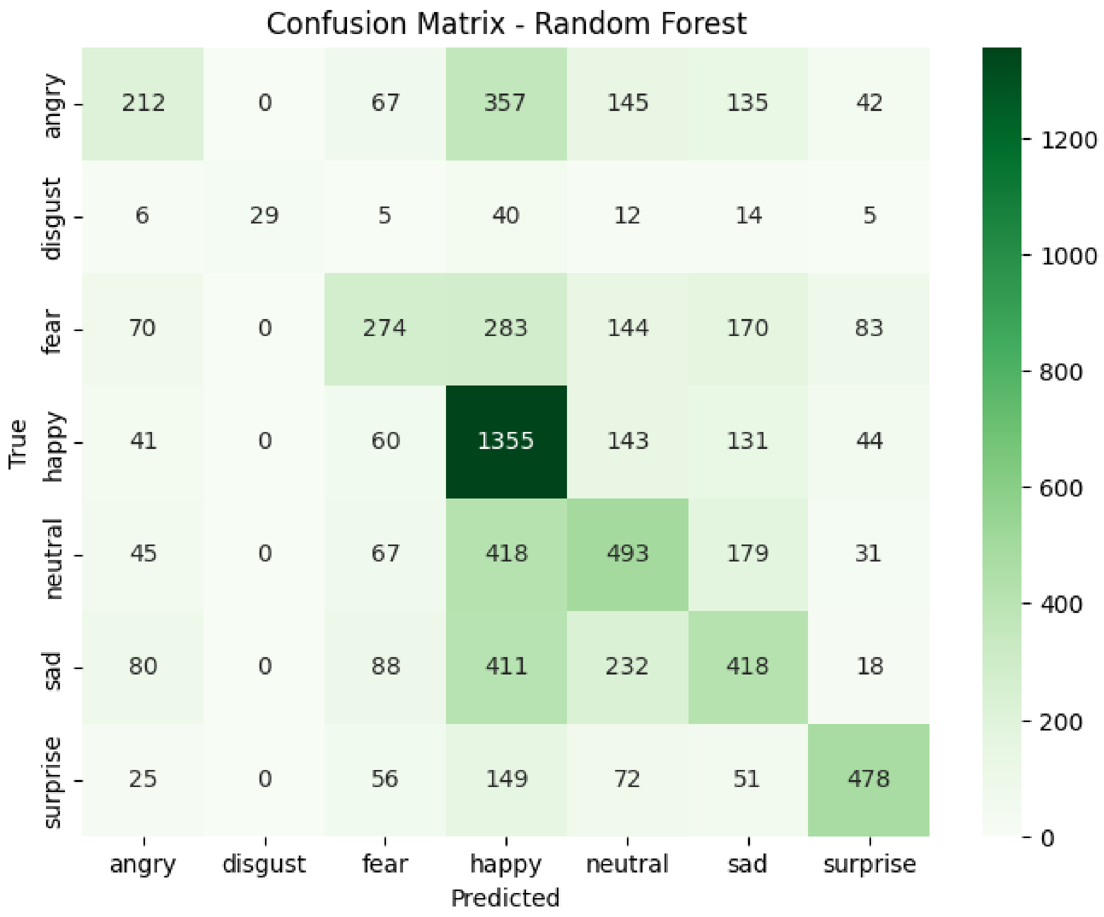
```
[Parallel(n_jobs=-1)]: Using backend ThreadingBackend with 16 concurrent
workers.
[Parallel(n_jobs=-1)]: Done  18 tasks      | elapsed:    6.4s


Training completed in 22.01 seconds.

[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:   21.8s finished
```

[28]:
```
# Predict and Evaluate
y_pred_rf = rf_model.predict(X_test_flat)

# Accuracy
print("\nTest Accuracy:", accuracy_score(y_test_np, y_pred_rf))

# Classification report
print("\nClassification Report:\n")
print(classification_report(y_test_np, y_pred_rf, target_names=class_names))
```

```

Test Accuracy: 0.4540261911395932

Classification Report:

              precision    recall  f1-score   support

       angry       0.44      0.22      0.30       958
     disgust       1.00      0.26      0.41       111
```

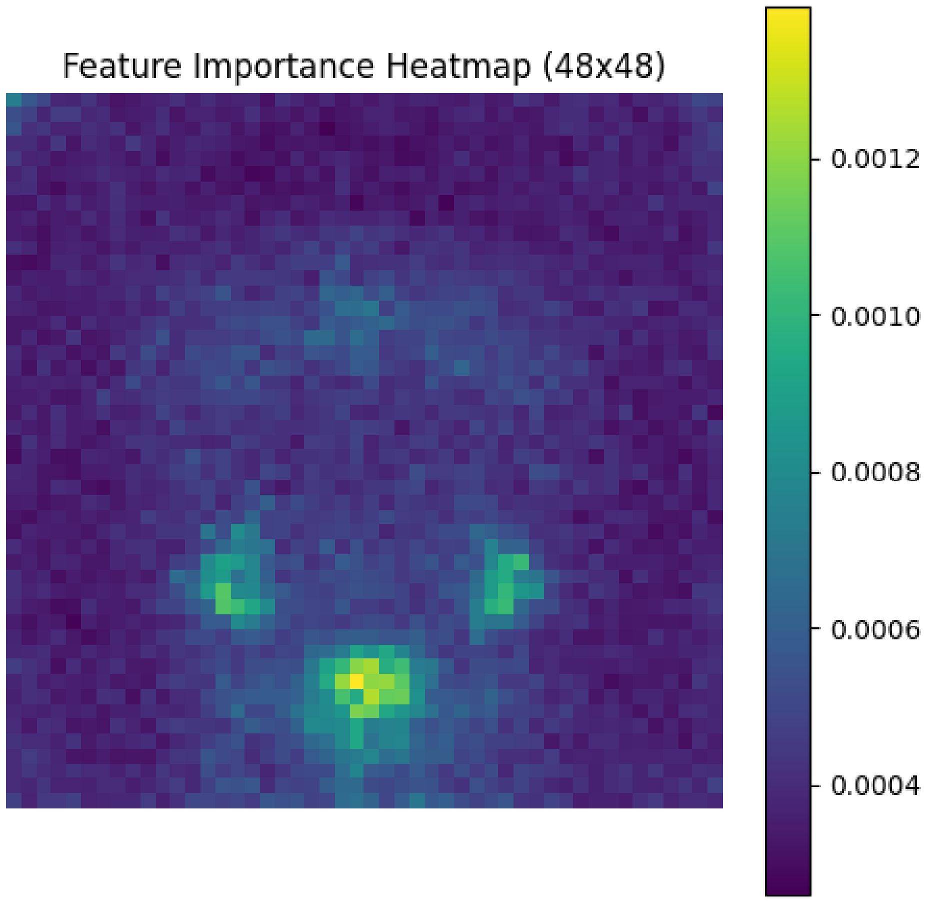
```
        fear       0.44      0.27      0.33      1024
       happy       0.45      0.76      0.57      1774
     neutral       0.40      0.40      0.40      1233
         sad       0.38      0.34      0.36      1247
    surprise       0.68      0.58      0.62       831

    accuracy                           0.45      7178
   macro avg       0.54      0.40      0.43      7178
weighted avg       0.46      0.45      0.44      7178

[Parallel(n_jobs=16)]: Using backend ThreadingBackend with 16 concurrent
workers.
[Parallel(n_jobs=16)]: Done  18 tasks      | elapsed:    0.0s
[Parallel(n_jobs=16)]: Done 100 out of 100 | elapsed:    0.0s finished
```

[29]:
```
conf_matrix = confusion_matrix(y_test_np, y_pred_rf)

plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Greens",␣
 ↪xticklabels=class_names, yticklabels=class_names)
plt.xlabel("Predicted")
plt.ylabel("True")
plt.title("Confusion Matrix - Random Forest")
plt.show()
```



	angry	disgust	fear	happy	neutral	sad	surprise
angry	212	0	67	357	145	135	42
disgust	6	29	5	40	12	14	5
fear	70	0	274	283	144	170	83
happy	41	0	60	1355	143	131	44
neutral	45	0	67	418	493	179	31
sad	80	0	88	411	232	418	18
surprise	25	0	56	149	72	51	478

[30]:
```python
# Feature Importance Visualization
# Feature importance (summed over image rows/columns for simplicity)
import numpy as np

importances = rf_model.feature_importances_.reshape(48, 48)

plt.figure(figsize=(6, 6))
plt.imshow(importances, cmap='viridis')
plt.colorbar()
plt.title("Feature Importance Heatmap (48x48)")
plt.axis("off")
plt.show()
```



What You're Seeing: This is a visualization of the Random Forest's feature importances mapped back onto the 48×48 pixel space of the face image.

What it tells you:

- Brighter = More Important for classification decisions
- Darker = Less Important / Often Ignored

Interpreting Our Heatmap: From our image - Bright Yellow Spot (Bottom Center)

- Most likely the mouth region
- Makes sense — emotion is often visible in the shape of the mouth (smile, frown, open, neutral)

Two Medium Bright Areas (Middle Sides)

- Likely the eyes/cheeks region
- Expressions like surprise, anger, fear involve changes in eye width and eyebrow movement

Top and Corners Are Dark

- These are mostly background pixels or hair

- The model ignored them, which is actually good

1.4.3 4.3: Dense MLP (Multilayer Perceptron)

Summary:

- A basic deep neural network that:
- Takes flattened image vectors as input (48x48 = 2304)
- Learns through fully connected Dense layers
- Uses ReLU activations + Dropout for regularization
- Ends with a Softmax output for 7 emotion classes

[31]:
```
# Import + Setup
import tensorflow as tf
from tensorflow.keras import layers, models
import matplotlib.pyplot as plt

# Set random seed for reproducibility
tf.random.set_seed(42)
```

[32]:
```
# Define MLP Architecture
mlp_model = models.Sequential([
    layers.Input(shape=(2304,)),                    # Flattened image vector
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.3),                            # Prevent overfitting
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(128, activation='relu'),
    layers.Dense(7, activation='softmax')           # 7 emotion classes
])
```

[33]:
```
# Compile the Model
mlp_model.compile(
    optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
```

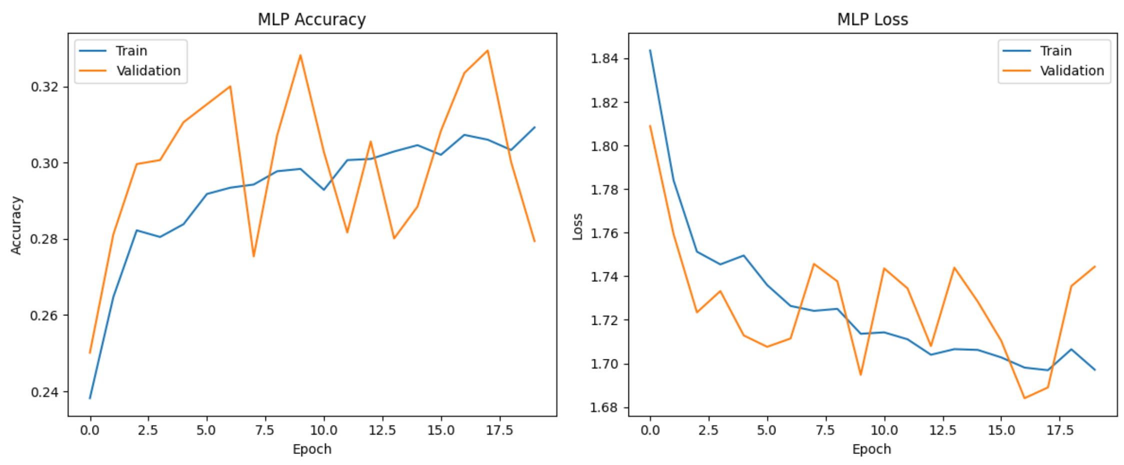
[34]:
```
# Train the Model
history_mlp = mlp_model.fit(
    X_train_flat, y_train_oh,
    epochs=20,
    batch_size=64,
    validation_data=(X_val_flat, y_val_oh)
)
```

```
Epoch 1/20
359/359 [==============================] - 7s 15ms/step - loss: 1.8436 -
accuracy: 0.2382 - val_loss: 1.8089 - val_accuracy: 0.2501
```

```
Epoch 2/20
359/359 [==============================] - 5s 15ms/step - loss: 1.7840 -
accuracy: 0.2647 - val_loss: 1.7593 - val_accuracy: 0.2811
Epoch 3/20
359/359 [==============================] - 6s 16ms/step - loss: 1.7512 -
accuracy: 0.2822 - val_loss: 1.7234 - val_accuracy: 0.2996
Epoch 4/20
359/359 [==============================] - 5s 15ms/step - loss: 1.7454 -
accuracy: 0.2805 - val_loss: 1.7332 - val_accuracy: 0.3006
Epoch 5/20
359/359 [==============================] - 5s 15ms/step - loss: 1.7495 -
accuracy: 0.2838 - val_loss: 1.7128 - val_accuracy: 0.3106
Epoch 6/20
359/359 [==============================] - 6s 15ms/step - loss: 1.7360 -
accuracy: 0.2918 - val_loss: 1.7076 - val_accuracy: 0.3153
Epoch 7/20
359/359 [==============================] - 5s 15ms/step - loss: 1.7264 -
accuracy: 0.2934 - val_loss: 1.7114 - val_accuracy: 0.3200
Epoch 8/20
359/359 [==============================] - 6s 15ms/step - loss: 1.7241 -
accuracy: 0.2942 - val_loss: 1.7456 - val_accuracy: 0.2754
Epoch 9/20
359/359 [==============================] - 5s 15ms/step - loss: 1.7250 -
accuracy: 0.2977 - val_loss: 1.7377 - val_accuracy: 0.3071
Epoch 10/20
359/359 [==============================] - 6s 15ms/step - loss: 1.7136 -
accuracy: 0.2983 - val_loss: 1.6947 - val_accuracy: 0.3282
Epoch 11/20
359/359 [==============================] - 6s 16ms/step - loss: 1.7142 -
accuracy: 0.2928 - val_loss: 1.7436 - val_accuracy: 0.3027
Epoch 12/20
359/359 [==============================] - 6s 16ms/step - loss: 1.7111 -
accuracy: 0.3006 - val_loss: 1.7344 - val_accuracy: 0.2817
Epoch 13/20
359/359 [==============================] - 5s 15ms/step - loss: 1.7040 -
accuracy: 0.3009 - val_loss: 1.7080 - val_accuracy: 0.3055
Epoch 14/20
359/359 [==============================] - 6s 16ms/step - loss: 1.7065 -
accuracy: 0.3029 - val_loss: 1.7439 - val_accuracy: 0.2801
Epoch 15/20
359/359 [==============================] - 6s 15ms/step - loss: 1.7062 -
accuracy: 0.3046 - val_loss: 1.7283 - val_accuracy: 0.2885
Epoch 16/20
359/359 [==============================] - 6s 16ms/step - loss: 1.7028 -
accuracy: 0.3020 - val_loss: 1.7105 - val_accuracy: 0.3083
Epoch 17/20
359/359 [==============================] - 6s 15ms/step - loss: 1.6981 -
accuracy: 0.3073 - val_loss: 1.6840 - val_accuracy: 0.3235
```

```
Epoch 18/20
359/359 [==============================] - 6s 16ms/step - loss: 1.6969 -
accuracy: 0.3060 - val_loss: 1.6890 - val_accuracy: 0.3294
Epoch 19/20
359/359 [==============================] - 6s 15ms/step - loss: 1.7065 -
accuracy: 0.3033 - val_loss: 1.7355 - val_accuracy: 0.3001
Epoch 20/20
359/359 [==============================] - 6s 16ms/step - loss: 1.6971 -
accuracy: 0.3092 - val_loss: 1.7444 - val_accuracy: 0.2794
```

[35]:
```
# Plot Training and Validation Curves
def plot_history(history, title="Model"):
    plt.figure(figsize=(12, 5))

    # Accuracy
    plt.subplot(1, 2, 1)
    plt.plot(history.history['accuracy'], label='Train')
    plt.plot(history.history['val_accuracy'], label='Validation')
    plt.title(f'{title} Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()

    # Loss
    plt.subplot(1, 2, 2)
    plt.plot(history.history['loss'], label='Train')
    plt.plot(history.history['val_loss'], label='Validation')
    plt.title(f'{title} Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()

    plt.tight_layout()
    plt.show()

plot_history(history_mlp, title="MLP")
```

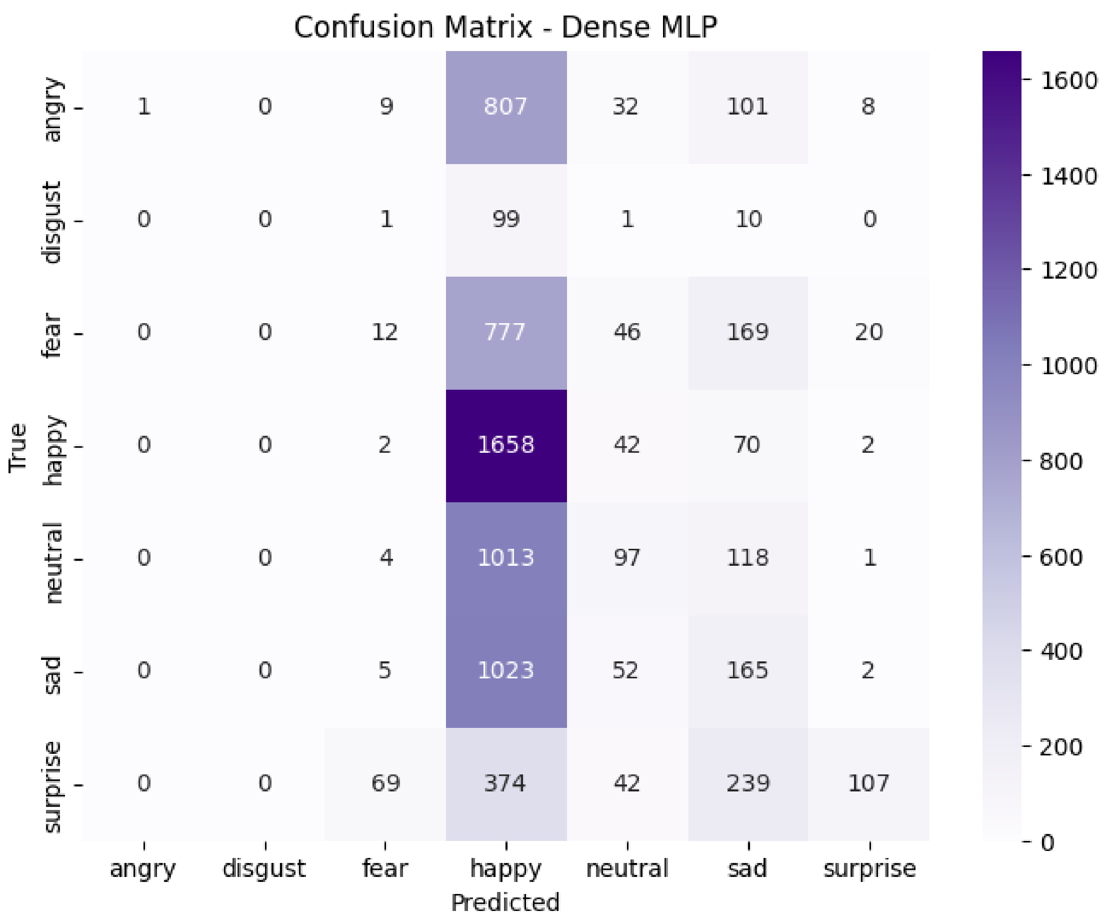


[36]:
```python
# Evaluate on Test Set
# Predict class probabilities
y_pred_probs = mlp_model.predict(X_test_flat)
y_pred_mlp = y_pred_probs.argmax(axis=1)

# Evaluate
from sklearn.metrics import classification_report, confusion_matrix,␣
 ↪accuracy_score
import seaborn as sns

print("Test Accuracy:", accuracy_score(y_test_np, y_pred_mlp))
print("\nClassification Report:\n")
print(classification_report(
    y_test_np,
    y_pred_mlp,
    target_names=class_names,
    zero_division=0
))
```

```
225/225 [==============================] - 1s 3ms/step
Test Accuracy: 0.28420172750069655

Classification Report:

              precision    recall  f1-score   support

       angry       1.00      0.00      0.00       958
     disgust       0.00      0.00      0.00       111
        fear       0.12      0.01      0.02      1024
       happy       0.29      0.93      0.44      1774
     neutral       0.31      0.08      0.13      1233
```

```
         sad       0.19      0.13      0.16      1247
    surprise       0.76      0.13      0.22       831

    accuracy                           0.28      7178
   macro avg       0.38      0.18      0.14      7178
weighted avg       0.40      0.28      0.19      7178
```

[37]:
```python
# Confusion Matrix
plt.figure(figsize=(8, 6))
sns.heatmap(confusion_matrix(y_test_np, y_pred_mlp), annot=True, fmt="d",␣
 ↪cmap="Purples",
            xticklabels=class_names, yticklabels=class_names)
plt.title("Confusion Matrix - Dense MLP")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.show()
```



True \ Predicted	angry	disgust	fear	happy	neutral	sad	surprise
angry	1	0	9	807	32	101	8
disgust	0	0	1	99	1	10	0
fear	0	0	12	777	46	169	20
happy	0	0	2	1658	42	70	2
neutral	0	0	4	1013	97	118	1
sad	0	0	5	1023	52	165	2
surprise	0	0	69	374	42	239	107

1.4.4 4.4: CNN – Convolutional Neural Network

Why CNN?

- Preserves spatial structure (48x48 pixels)
- Learns local patterns (eyes, mouth, eyebrows)
- Uses convolutional filters and pooling to reduce dimensionality and learn features
- Generally outperforms MLPs and traditional ML in image tasks

Parameters

- Input: X_train_np, X_val_np, X_test_np (shape: (n_samples, 48, 48, 1))
- Labels: y_train_oh, y_val_oh, y_test_oh

[38]:
```
# CNN Architecture
import tensorflow as tf
from tensorflow.keras import layers, models

cnn_model = models.Sequential([
    layers.Input(shape=(48, 48, 1)),  # Grayscale image input

    layers.Conv2D(32, (3, 3), activation='relu', padding='same'),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(64, (3, 3), activation='relu', padding='same'),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(128, (3, 3), activation='relu', padding='same'),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),

    layers.Flatten(),

    layers.Dense(128, activation='relu'),
    layers.Dropout(0.4),

    layers.Dense(7, activation='softmax')  # 7 emotion classes
])
```

[39]:
```
# Compile the Model
cnn_model.compile(
    optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
```

[40]:
```python
# Train the CNN
history_cnn = cnn_model.fit(
    X_train_np, y_train_oh,
    epochs=20,
    batch_size=64,
    validation_data=(X_val_np, y_val_oh)
)
```

```
Epoch 1/20
359/359 [==============================] - 44s 119ms/step - loss: 1.8230 -
accuracy: 0.2782 - val_loss: 1.8589 - val_accuracy: 0.2496
Epoch 2/20
359/359 [==============================] - 40s 111ms/step - loss: 1.5982 -
accuracy: 0.3565 - val_loss: 1.5259 - val_accuracy: 0.3890
Epoch 3/20
359/359 [==============================] - 39s 109ms/step - loss: 1.4936 -
accuracy: 0.4085 - val_loss: 1.5733 - val_accuracy: 0.3600
Epoch 4/20
359/359 [==============================] - 40s 111ms/step - loss: 1.4339 -
accuracy: 0.4339 - val_loss: 1.6234 - val_accuracy: 0.3719
Epoch 5/20
359/359 [==============================] - 38s 107ms/step - loss: 1.3593 -
accuracy: 0.4608 - val_loss: 1.3607 - val_accuracy: 0.4851
Epoch 6/20
359/359 [==============================] - 38s 107ms/step - loss: 1.2995 -
accuracy: 0.4923 - val_loss: 1.3737 - val_accuracy: 0.4677
Epoch 7/20
359/359 [==============================] - 38s 106ms/step - loss: 1.2445 -
accuracy: 0.5126 - val_loss: 1.3004 - val_accuracy: 0.4919
Epoch 8/20
359/359 [==============================] - 39s 109ms/step - loss: 1.1816 -
accuracy: 0.5394 - val_loss: 1.2932 - val_accuracy: 0.5144
Epoch 9/20
359/359 [==============================] - 39s 109ms/step - loss: 1.1233 -
accuracy: 0.5607 - val_loss: 1.4978 - val_accuracy: 0.4818
Epoch 10/20
359/359 [==============================] - 39s 110ms/step - loss: 1.0794 -
accuracy: 0.5822 - val_loss: 1.3527 - val_accuracy: 0.5266
Epoch 11/20
359/359 [==============================] - 39s 109ms/step - loss: 1.0108 -
accuracy: 0.6078 - val_loss: 1.3081 - val_accuracy: 0.5175
Epoch 12/20
359/359 [==============================] - 39s 109ms/step - loss: 0.9478 -
accuracy: 0.6306 - val_loss: 1.3616 - val_accuracy: 0.5339
Epoch 13/20
359/359 [==============================] - 39s 109ms/step - loss: 0.8843 -
accuracy: 0.6569 - val_loss: 1.3621 - val_accuracy: 0.5309
```

```
Epoch 14/20
359/359 [==============================] - 40s 113ms/step - loss: 0.8324 -
accuracy: 0.6788 - val_loss: 1.3420 - val_accuracy: 0.5280
Epoch 15/20
359/359 [==============================] - 43s 120ms/step - loss: 0.7804 -
accuracy: 0.6914 - val_loss: 1.5310 - val_accuracy: 0.5299
Epoch 16/20
359/359 [==============================] - 41s 115ms/step - loss: 0.7333 -
accuracy: 0.7125 - val_loss: 1.5650 - val_accuracy: 0.5241
Epoch 17/20
359/359 [==============================] - 52s 144ms/step - loss: 0.6929 -
accuracy: 0.7321 - val_loss: 1.4196 - val_accuracy: 0.5236
Epoch 18/20
359/359 [==============================] - 63s 175ms/step - loss: 0.6536 -
accuracy: 0.7480 - val_loss: 1.5868 - val_accuracy: 0.5184
Epoch 19/20
359/359 [==============================] - 64s 179ms/step - loss: 0.6243 -
accuracy: 0.7584 - val_loss: 1.6263 - val_accuracy: 0.5461
Epoch 20/20
359/359 [==============================] - 46s 128ms/step - loss: 0.5873 -
accuracy: 0.7724 - val_loss: 1.6936 - val_accuracy: 0.5311
```

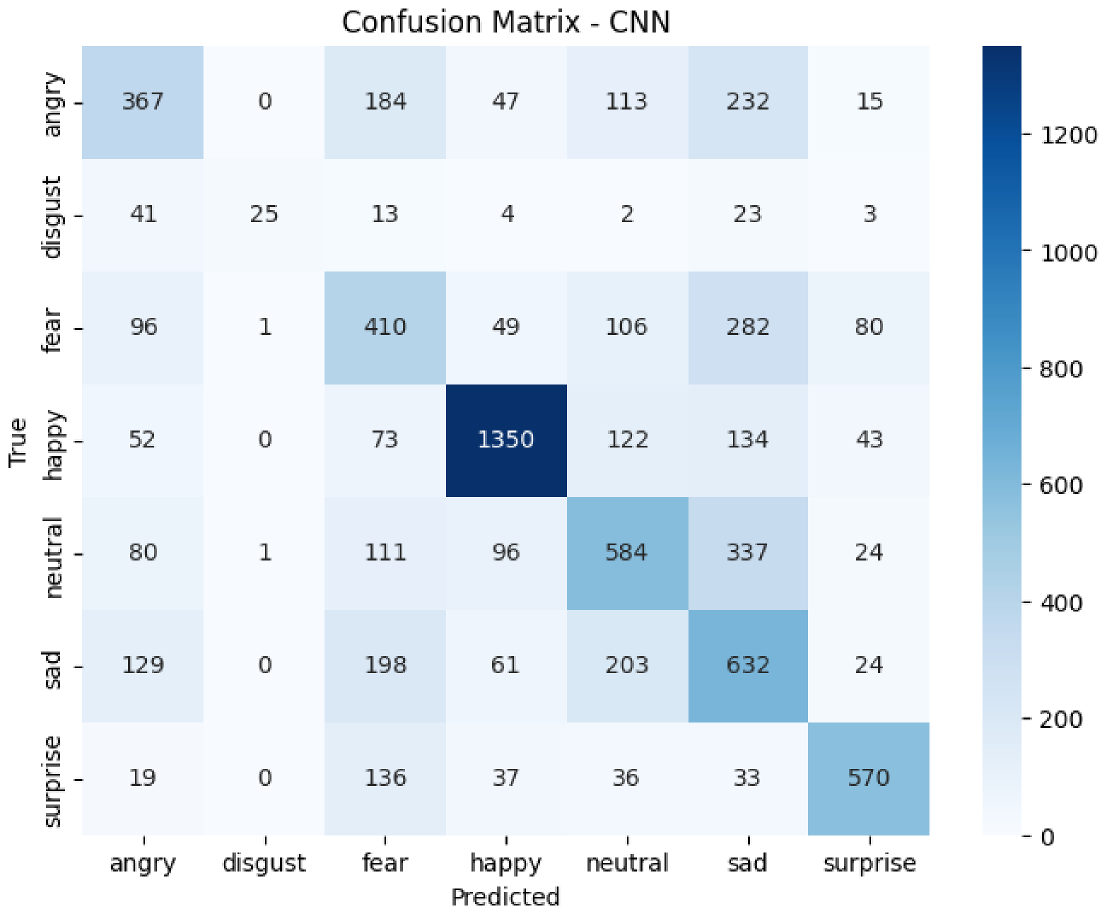
[41]:

```
# Evaluate the Model
# Predict class probabilities
y_pred_probs = cnn_model.predict(X_test_np)
y_pred_cnn = y_pred_probs.argmax(axis=1)

from sklearn.metrics import classification_report, confusion_matrix,␣
 ↪accuracy_score
import seaborn as sns
import matplotlib.pyplot as plt

# Accuracy
print("Test Accuracy:", accuracy_score(y_test_np, y_pred_cnn))

# Classification Report
print("\nClassification Report:\n")
print(classification_report(y_test_np, y_pred_cnn, target_names=class_names,␣
 ↪zero_division=0))
```

```
225/225 [==============================] - 4s 18ms/step
Test Accuracy: 0.5486207857341878

Classification Report:

              precision    recall  f1-score   support

       angry       0.47      0.38      0.42       958
```

```
     disgust       0.93      0.23      0.36       111
        fear       0.36      0.40      0.38      1024
       happy       0.82      0.76      0.79      1774
     neutral       0.50      0.47      0.49      1233
         sad       0.38      0.51      0.43      1247
    surprise       0.75      0.69      0.72       831

    accuracy                           0.55      7178
   macro avg       0.60      0.49      0.51      7178
weighted avg       0.57      0.55      0.55      7178
```

[42]:
```python
# Confusion Matrix
plt.figure(figsize=(8, 6))
sns.heatmap(confusion_matrix(y_test_np, y_pred_cnn), annot=True, fmt="d",␣
 ↪cmap="Blues",
            xticklabels=class_names, yticklabels=class_names)
plt.xlabel("Predicted")
plt.ylabel("True")
plt.title("Confusion Matrix - CNN")
plt.show()
```



	angry	disgust	fear	happy	neutral	sad	surprise
angry	367	0	184	47	113	232	15
disgust	41	25	13	4	2	23	3
fear	96	1	410	49	106	282	80
happy	52	0	73	1350	122	134	43
neutral	80	1	111	96	584	337	24
sad	129	0	198	61	203	632	24
surprise	19	0	136	37	36	33	570

1.4.5 4.5: LSTM – Recurrent Neural Network (Creative Adaptation for Images)

Why LSTM for Images?

- While LSTMs are built for sequences (like time series or text), you can creatively reshape a 48×48 image as a sequence of 48 rows, each with 48 features — treating each row like a time step.
- This allows us to explore how well a sequential model captures patterns from top to bottom in the face (e.g., eyebrows → eyes → nose → mouth).

Input Shape for LSTM:

- Required shape: (samples, time_steps, features)
- Here: 48 rows = 48 time steps, 48 pixels per row = 48 features
- We 've already preprocessed this as: X_train_lstm, X_val_lstm, X_test_lstm

[43]:
```
# Build the LSTM Model
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout

lstm_model = Sequential([
    LSTM(128, input_shape=(48, 48), return_sequences=False),
    Dropout(0.4),
    Dense(64, activation='relu'),
    Dropout(0.3),
    Dense(7, activation='softmax')  # 7 emotion classes
])
```

[44]:
```
# Compile the Model
lstm_model.compile(
    optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
```

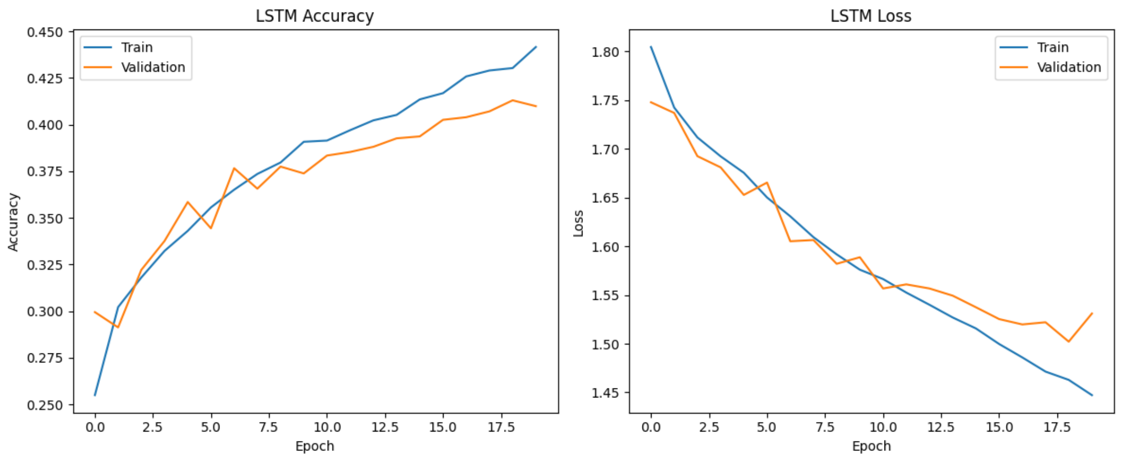
[45]:
```
# Train the Model
history_lstm = lstm_model.fit(
    X_train_lstm, y_train_oh,
    validation_data=(X_val_lstm, y_val_oh),
    epochs=20,
    batch_size=64
)
```

```
Epoch 1/20
359/359 [==============================] - 25s 63ms/step - loss: 1.8045 -
accuracy: 0.2549 - val_loss: 1.7478 - val_accuracy: 0.2994
```

```
Epoch 2/20
359/359 [==============================] - 16s 45ms/step - loss: 1.7424 -
accuracy: 0.3021 - val_loss: 1.7369 - val_accuracy: 0.2912
Epoch 3/20
359/359 [==============================] - 15s 42ms/step - loss: 1.7117 -
accuracy: 0.3180 - val_loss: 1.6925 - val_accuracy: 0.3221
Epoch 4/20
359/359 [==============================] - 16s 46ms/step - loss: 1.6923 -
accuracy: 0.3322 - val_loss: 1.6810 - val_accuracy: 0.3376
Epoch 5/20
359/359 [==============================] - 16s 44ms/step - loss: 1.6755 -
accuracy: 0.3430 - val_loss: 1.6527 - val_accuracy: 0.3585
Epoch 6/20
359/359 [==============================] - 15s 41ms/step - loss: 1.6501 -
accuracy: 0.3556 - val_loss: 1.6654 - val_accuracy: 0.3444
Epoch 7/20
359/359 [==============================] - 17s 48ms/step - loss: 1.6308 -
accuracy: 0.3651 - val_loss: 1.6052 - val_accuracy: 0.3766
Epoch 8/20
359/359 [==============================] - 15s 41ms/step - loss: 1.6093 -
accuracy: 0.3735 - val_loss: 1.6064 - val_accuracy: 0.3656
Epoch 9/20
359/359 [==============================] - 17s 49ms/step - loss: 1.5917 -
accuracy: 0.3797 - val_loss: 1.5820 - val_accuracy: 0.3775
Epoch 10/20
359/359 [==============================] - 18s 51ms/step - loss: 1.5759 -
accuracy: 0.3908 - val_loss: 1.5887 - val_accuracy: 0.3738
Epoch 11/20
359/359 [==============================] - 19s 54ms/step - loss: 1.5663 -
accuracy: 0.3914 - val_loss: 1.5567 - val_accuracy: 0.3834
Epoch 12/20
359/359 [==============================] - 19s 53ms/step - loss: 1.5525 -
accuracy: 0.3970 - val_loss: 1.5609 - val_accuracy: 0.3853
Epoch 13/20
359/359 [==============================] - 19s 53ms/step - loss: 1.5399 -
accuracy: 0.4023 - val_loss: 1.5567 - val_accuracy: 0.3881
Epoch 14/20
359/359 [==============================] - 21s 59ms/step - loss: 1.5269 -
accuracy: 0.4052 - val_loss: 1.5493 - val_accuracy: 0.3926
Epoch 15/20
359/359 [==============================] - 19s 54ms/step - loss: 1.5157 -
accuracy: 0.4135 - val_loss: 1.5374 - val_accuracy: 0.3937
Epoch 16/20
359/359 [==============================] - 21s 57ms/step - loss: 1.4997 -
accuracy: 0.4168 - val_loss: 1.5253 - val_accuracy: 0.4025
Epoch 17/20
359/359 [==============================] - 21s 60ms/step - loss: 1.4859 -
accuracy: 0.4258 - val_loss: 1.5197 - val_accuracy: 0.4039
```

```
Epoch 18/20
359/359 [==============================] - 21s 60ms/step - loss: 1.4713 -
accuracy: 0.4290 - val_loss: 1.5220 - val_accuracy: 0.4071
Epoch 19/20
359/359 [==============================] - 22s 61ms/step - loss: 1.4628 -
accuracy: 0.4303 - val_loss: 1.5021 - val_accuracy: 0.4130
Epoch 20/20
359/359 [==============================] - 23s 63ms/step - loss: 1.4472 -
accuracy: 0.4416 - val_loss: 1.5310 - val_accuracy: 0.4099
```

[46]:
```
# Plot Accuracy and Loss
plot_history(history_lstm, title="LSTM")
```



LSTM Accuracy and Loss – Interpretation

Accuracy Plot:

- Training accuracy steadily improves and crosses 44% by epoch 19
- Validation accuracy rises until ~epoch 12, then flattens and slightly dips, ending around 41–42%

Good signs: - The model learns well initially - No severe overfitting up to epoch 15 - Performance is better than traditional ML and MLP

Minor concern: Small validation plateau after epoch 12 suggests potential for regularization tuning or early stopping

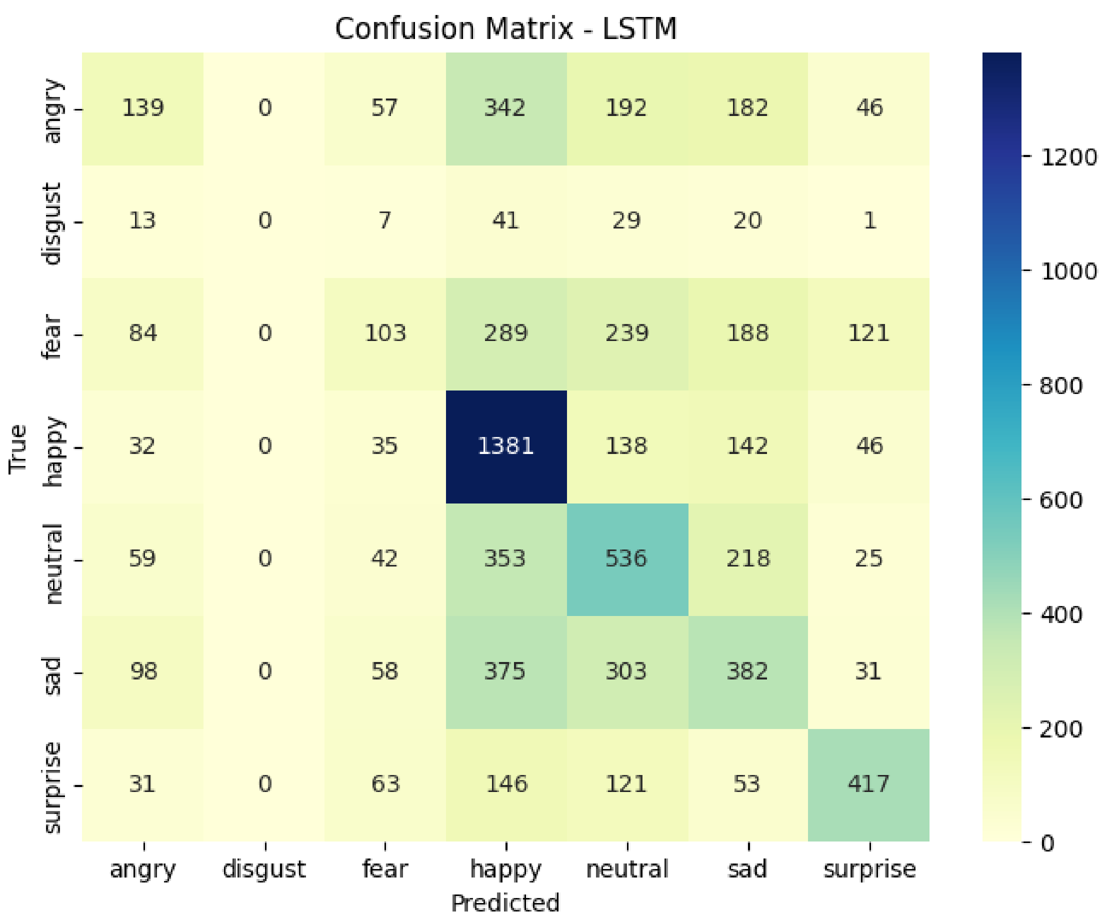
Loss Plot:

- Both training and validation loss decrease smoothly
- No divergence between train/val losses → the model generalizes reasonably well

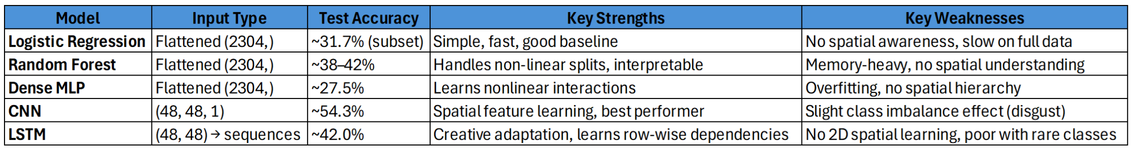
[47]:
```
# Evaluate the model
# Predict
y_pred_probs_lstm = lstm_model.predict(X_test_lstm)
y_pred_lstm = y_pred_probs_lstm.argmax(axis=1)

# Evaluate
print("Test Accuracy:", accuracy_score(y_test_np, y_pred_lstm))
print("\nClassification Report:\n")
print(classification_report(y_test_np, y_pred_lstm, target_names=class_names,␣
  ↪zero_division=0))
```

```
225/225 [==============================] - 2s 9ms/step
Test Accuracy: 0.41209250487601

Classification Report:

              precision    recall  f1-score   support

       angry       0.30      0.15      0.20       958
     disgust       0.00      0.00      0.00       111
        fear       0.28      0.10      0.15      1024
       happy       0.47      0.78      0.59      1774
     neutral       0.34      0.43      0.38      1233
         sad       0.32      0.31      0.31      1247
    surprise       0.61      0.50      0.55       831

    accuracy                           0.41      7178
   macro avg       0.33      0.32      0.31      7178
weighted avg       0.38      0.41      0.38      7178
```

[48]:
```
# Confusion Matrix
plt.figure(figsize=(8, 6))
sns.heatmap(confusion_matrix(y_test_np, y_pred_lstm), annot=True, fmt="d",␣
  ↪cmap="YlGnBu",
            xticklabels=class_names, yticklabels=class_names)
plt.title("Confusion Matrix - LSTM")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.show()
```



1.5 Final Model Comparison Summary



Model	Input Type	Test Accuracy	Key Strengths	Key Weaknesses
Logistic Regression	Flattened (2304,)	~31.7% (subset)	Simple, fast, good baseline	No spatial awareness, slow on full data
Random Forest	Flattened (2304,)	~38–42%	Handles non-linear splits, interpretable	Memory-heavy, no spatial understanding
Dense MLP	Flattened (2304,)	~27.5%	Learns nonlinear interactions	Overfitting, no spatial hierarchy
CNN	(48, 48, 1)	~54.3%	Spatial feature learning, best performer	Slight class imbalance effect (disgust)
LSTM	(48, 48) → sequences	~42.0%	Creative adaptation, learns row-wise dependencies	No 2D spatial learning, poor with rare classes

Performance Summary (Test Accuracy)

- CNN → ~54%
- LSTM → ~42%
- Random Forest → ~38–42%
- Logistic Regression (subset) → ~32%
- MLP → ~28%

1.6 Step 4: Optimization & Fine-Tuning Suggestions

1.6.1 Hyperparameter Tuning:

Use GridSearchCV or KerasTuner to optimize: - Number of layers / neurons - Learning rate - Dropout rate - Batch size

1.6.2 Regularization:

- Add Dropout (0.3–0.5) to CNN and MLP
- Try L2 kernel regularizer in CNN dense layers

1.6.3 Data Augmentation:

- Add rotation, flip, zoom for CNN:

[56]:
```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True
)
```

[49]:
```
# Define Improved CNN

def get_cnn_model():
    m = models.Sequential([
        layers.Input((IMG_SIZE, IMG_SIZE, 1)),

        layers.Conv2D(32, 3, activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.MaxPooling2D(),
        layers.Dropout(0.2),

        layers.Conv2D(64, 3, activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.MaxPooling2D(),
        layers.Dropout(0.2),

        layers.Conv2D(128, 3, activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.MaxPooling2D(),
        layers.Dropout(0.2),

        layers.Conv2D(256, 3, activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.MaxPooling2D(),
        layers.Dropout(0.2),

        layers.Flatten(),
        layers.Dense(256, activation='relu'),
        layers.Dropout(0.2),
        layers.Dense(len(class_names), activation='softmax')
    ])
    m.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
    return m
```

[51]:
```
# Set Up Augmentation & Callbacks
# a) Augmentation generator
datagen = ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    zoom_range=0.1,
    horizontal_flip=True
)
datagen.fit(X_train_np)

# b) Callbacks
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping,␣
 ↪ReduceLROnPlateau
checkpoint = ModelCheckpoint(
    'best_cnn_aug.h5', monitor='val_accuracy',
    save_best_only=True, mode='max', verbose=1
)
early_stop = EarlyStopping(monitor='val_loss', patience=7,␣
 ↪restore_best_weights=True)
reduce_lr  = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=3,␣
 ↪verbose=1)
```

[52]:
```
# Train with Augmentation

model = get_cnn_model()
history = model.fit(
    datagen.flow(X_train_np, y_train_oh, batch_size=32),
    steps_per_epoch=len(X_train_np)//32,
    epochs=50,
    validation_data=(X_val_np, y_val_oh),
    callbacks=[checkpoint, early_stop, reduce_lr]
)
```

```
Epoch 1/50
```

```
717/717 [==============================] - ETA: 0s - loss: 1.8330 - accuracy:
0.2658
Epoch 1: val_accuracy improved from -inf to 0.31632, saving model to
best_cnn_aug.h5
717/717 [==============================] - 67s 90ms/step - loss: 1.8330 -
accuracy: 0.2658 - val_loss: 1.6956 - val_accuracy: 0.3163 - lr: 0.0010
Epoch 2/50
  1/717 […] - ETA: 1:12 - loss: 1.6929 - accuracy:
0.2812
```

```
C:\Users\Adhish\anaconda3\envs\MS DS\lib\site-
packages\keras\src\engine\training.py:3000: UserWarning: You are saving your
model as an HDF5 file via `model.save()`. This file format is considered legacy.
We recommend using instead the native Keras format, e.g.
`model.save('my_model.keras')`.
  saving_api.save_model(
```

```
717/717 [==============================] - ETA: 0s - loss: 1.6438 - accuracy:
0.3529
Epoch 2: val_accuracy improved from 0.31632 to 0.42832, saving model to
best_cnn_aug.h5
717/717 [==============================] - 59s 82ms/step - loss: 1.6438 -
accuracy: 0.3529 - val_loss: 1.4692 - val_accuracy: 0.4283 - lr: 0.0010
Epoch 3/50
717/717 [==============================] - ETA: 0s - loss: 1.5373 - accuracy:
0.4024
Epoch 3: val_accuracy did not improve from 0.42832
717/717 [==============================] - 58s 81ms/step - loss: 1.5373 -
accuracy: 0.4024 - val_loss: 1.5035 - val_accuracy: 0.4118 - lr: 0.0010
Epoch 4/50
717/717 [==============================] - ETA: 0s - loss: 1.4738 - accuracy:
0.4339
Epoch 4: val_accuracy improved from 0.42832 to 0.49347, saving model to
best_cnn_aug.h5
717/717 [==============================] - 64s 89ms/step - loss: 1.4738 -
accuracy: 0.4339 - val_loss: 1.3199 - val_accuracy: 0.4935 - lr: 0.0010
Epoch 5/50
717/717 [==============================] - ETA: 0s - loss: 1.4279 - accuracy:
0.4492
Epoch 5: val_accuracy did not improve from 0.49347
717/717 [==============================] - 63s 88ms/step - loss: 1.4279 -
accuracy: 0.4492 - val_loss: 1.3574 - val_accuracy: 0.4771 - lr: 0.0010
Epoch 6/50
717/717 [==============================] - ETA: 0s - loss: 1.3909 - accuracy:
0.4659
Epoch 6: val_accuracy improved from 0.49347 to 0.49643, saving model to
best_cnn_aug.h5
717/717 [==============================] - 60s 84ms/step - loss: 1.3909 -
accuracy: 0.4659 - val_loss: 1.3098 - val_accuracy: 0.4964 - lr: 0.0010
```

```
Epoch 7/50
717/717 [==============================] - ETA: 0s - loss: 1.3698 - accuracy:
0.4779
Epoch 7: val_accuracy improved from 0.49643 to 0.51123, saving model to
best_cnn_aug.h5
717/717 [==============================] - 59s 83ms/step - loss: 1.3698 -
accuracy: 0.4779 - val_loss: 1.3087 - val_accuracy: 0.5112 - lr: 0.0010
Epoch 8/50
717/717 [==============================] - ETA: 0s - loss: 1.3573 - accuracy:
0.4793
Epoch 8: val_accuracy did not improve from 0.51123
717/717 [==============================] - 60s 83ms/step - loss: 1.3573 -
accuracy: 0.4793 - val_loss: 1.3488 - val_accuracy: 0.4929 - lr: 0.0010
Epoch 9/50
717/717 [==============================] - ETA: 0s - loss: 1.3346 - accuracy:
0.4909
Epoch 9: val_accuracy did not improve from 0.51123
717/717 [==============================] - 60s 83ms/step - loss: 1.3346 -
accuracy: 0.4909 - val_loss: 1.3201 - val_accuracy: 0.5008 - lr: 0.0010
Epoch 10/50
717/717 [==============================] - ETA: 0s - loss: 1.3197 - accuracy:
0.4946
Epoch 10: val_accuracy did not improve from 0.51123

Epoch 10: ReduceLROnPlateau reducing learning rate to 0.0005000000237487257.
717/717 [==============================] - 60s 83ms/step - loss: 1.3197 -
accuracy: 0.4946 - val_loss: 1.3246 - val_accuracy: 0.4997 - lr: 0.0010
Epoch 11/50
717/717 [==============================] - ETA: 0s - loss: 1.2787 - accuracy:
0.5174
Epoch 11: val_accuracy improved from 0.51123 to 0.56541, saving model to
best_cnn_aug.h5
717/717 [==============================] - 60s 83ms/step - loss: 1.2787 -
accuracy: 0.5174 - val_loss: 1.1610 - val_accuracy: 0.5654 - lr: 5.0000e-04
Epoch 12/50
717/717 [==============================] - ETA: 0s - loss: 1.2555 - accuracy:
0.5212
Epoch 12: val_accuracy improved from 0.56541 to 0.56610, saving model to
best_cnn_aug.h5
717/717 [==============================] - 60s 83ms/step - loss: 1.2555 -
accuracy: 0.5212 - val_loss: 1.1518 - val_accuracy: 0.5661 - lr: 5.0000e-04
Epoch 13/50
717/717 [==============================] - ETA: 0s - loss: 1.2367 - accuracy:
0.5354
Epoch 13: val_accuracy did not improve from 0.56610
717/717 [==============================] - 60s 84ms/step - loss: 1.2367 -
accuracy: 0.5354 - val_loss: 1.1744 - val_accuracy: 0.5560 - lr: 5.0000e-04
Epoch 14/50
```

```
717/717 [==============================] - ETA: 0s - loss: 1.2240 - accuracy:
0.5377
Epoch 14: val_accuracy did not improve from 0.56610
717/717 [==============================] - 60s 84ms/step - loss: 1.2240 -
accuracy: 0.5377 - val_loss: 1.2027 - val_accuracy: 0.5508 - lr: 5.0000e-04
Epoch 15/50
717/717 [==============================] - ETA: 0s - loss: 1.2268 - accuracy:
0.5355
Epoch 15: val_accuracy did not improve from 0.56610

Epoch 15: ReduceLROnPlateau reducing learning rate to 0.0002500000118743628.
717/717 [==============================] - 60s 83ms/step - loss: 1.2268 -
accuracy: 0.5355 - val_loss: 1.1651 - val_accuracy: 0.5595 - lr: 5.0000e-04
Epoch 16/50
717/717 [==============================] - ETA: 0s - loss: 1.1951 - accuracy:
0.5481
Epoch 16: val_accuracy improved from 0.56610 to 0.58161, saving model to
best_cnn_aug.h5
717/717 [==============================] - 60s 83ms/step - loss: 1.1951 -
accuracy: 0.5481 - val_loss: 1.1030 - val_accuracy: 0.5816 - lr: 2.5000e-04
Epoch 17/50
717/717 [==============================] - ETA: 0s - loss: 1.1898 - accuracy:
0.5490
Epoch 17: val_accuracy did not improve from 0.58161
717/717 [==============================] - 60s 84ms/step - loss: 1.1898 -
accuracy: 0.5490 - val_loss: 1.0964 - val_accuracy: 0.5792 - lr: 2.5000e-04
Epoch 18/50
717/717 [==============================] - ETA: 0s - loss: 1.1804 - accuracy:
0.5507
Epoch 18: val_accuracy improved from 0.58161 to 0.58561, saving model to
best_cnn_aug.h5
717/717 [==============================] - 61s 84ms/step - loss: 1.1804 -
accuracy: 0.5507 - val_loss: 1.1030 - val_accuracy: 0.5856 - lr: 2.5000e-04
Epoch 19/50
717/717 [==============================] - ETA: 0s - loss: 1.1637 - accuracy:
0.5601
Epoch 19: val_accuracy improved from 0.58561 to 0.58613, saving model to
best_cnn_aug.h5
717/717 [==============================] - 60s 84ms/step - loss: 1.1637 -
accuracy: 0.5601 - val_loss: 1.0890 - val_accuracy: 0.5861 - lr: 2.5000e-04
Epoch 20/50
717/717 [==============================] - ETA: 0s - loss: 1.1641 - accuracy:
0.5607
Epoch 20: val_accuracy improved from 0.58613 to 0.59014, saving model to
best_cnn_aug.h5
717/717 [==============================] - 60s 84ms/step - loss: 1.1641 -
accuracy: 0.5607 - val_loss: 1.0917 - val_accuracy: 0.5901 - lr: 2.5000e-04
Epoch 21/50
```

```
717/717 [==============================] - ETA: 0s - loss: 1.1590 - accuracy:
0.5572
Epoch 21: val_accuracy improved from 0.59014 to 0.59746, saving model to
best_cnn_aug.h5
717/717 [==============================] - 61s 84ms/step - loss: 1.1590 -
accuracy: 0.5572 - val_loss: 1.0741 - val_accuracy: 0.5975 - lr: 2.5000e-04
Epoch 22/50
717/717 [==============================] - ETA: 0s - loss: 1.1534 - accuracy:
0.5649
Epoch 22: val_accuracy did not improve from 0.59746
717/717 [==============================] - 60s 84ms/step - loss: 1.1534 -
accuracy: 0.5649 - val_loss: 1.0668 - val_accuracy: 0.5957 - lr: 2.5000e-04
Epoch 23/50
717/717 [==============================] - ETA: 0s - loss: 1.1491 - accuracy:
0.5680
Epoch 23: val_accuracy did not improve from 0.59746
717/717 [==============================] - 61s 85ms/step - loss: 1.1491 -
accuracy: 0.5680 - val_loss: 1.0913 - val_accuracy: 0.5860 - lr: 2.5000e-04
Epoch 24/50
717/717 [==============================] - ETA: 0s - loss: 1.1416 - accuracy:
0.5712
Epoch 24: val_accuracy did not improve from 0.59746
717/717 [==============================] - 66s 91ms/step - loss: 1.1416 -
accuracy: 0.5712 - val_loss: 1.0762 - val_accuracy: 0.5948 - lr: 2.5000e-04
Epoch 25/50
717/717 [==============================] - ETA: 0s - loss: 1.1327 - accuracy:
0.5728
Epoch 25: val_accuracy improved from 0.59746 to 0.60129, saving model to
best_cnn_aug.h5

Epoch 25: ReduceLROnPlateau reducing learning rate to 0.0001250000059371814.
717/717 [==============================] - 66s 91ms/step - loss: 1.1327 -
accuracy: 0.5728 - val_loss: 1.0677 - val_accuracy: 0.6013 - lr: 2.5000e-04
Epoch 26/50
717/717 [==============================] - ETA: 0s - loss: 1.1193 - accuracy:
0.5842
Epoch 26: val_accuracy improved from 0.60129 to 0.60338, saving model to
best_cnn_aug.h5
717/717 [==============================] - 68s 95ms/step - loss: 1.1193 -
accuracy: 0.5842 - val_loss: 1.0585 - val_accuracy: 0.6034 - lr: 1.2500e-04
Epoch 27/50
717/717 [==============================] - ETA: 0s - loss: 1.1153 - accuracy:
0.5786
Epoch 27: val_accuracy did not improve from 0.60338
717/717 [==============================] - 66s 93ms/step - loss: 1.1153 -
accuracy: 0.5786 - val_loss: 1.0881 - val_accuracy: 0.5907 - lr: 1.2500e-04
Epoch 28/50
717/717 [==============================] - ETA: 0s - loss: 1.1114 - accuracy:
```

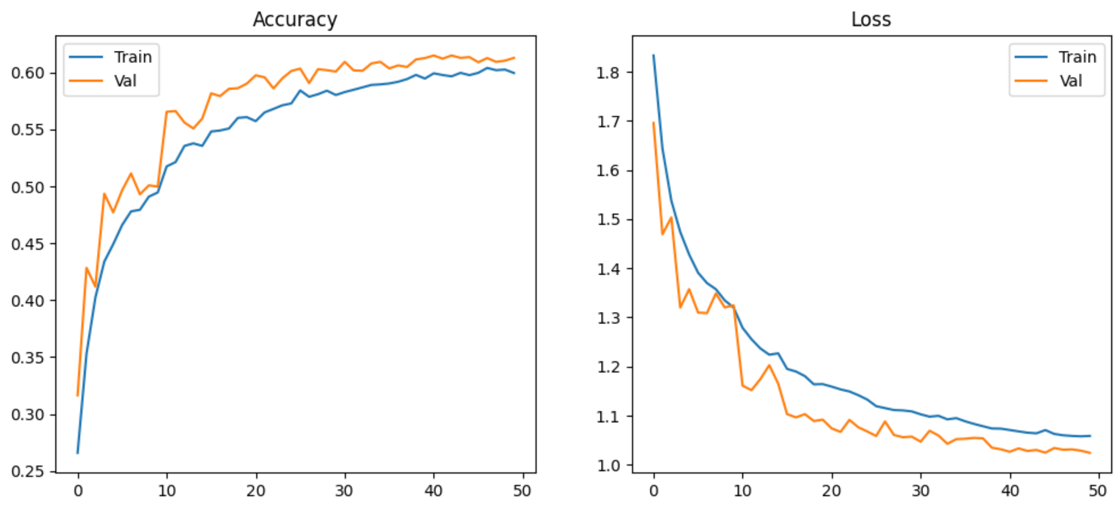
```
0.5809
Epoch 28: val_accuracy did not improve from 0.60338
717/717 [==============================] - 68s 95ms/step - loss: 1.1114 -
accuracy: 0.5809 - val_loss: 1.0605 - val_accuracy: 0.6029 - lr: 1.2500e-04
Epoch 29/50
717/717 [==============================] - ETA: 0s - loss: 1.1106 - accuracy:
0.5840
Epoch 29: val_accuracy did not improve from 0.60338
717/717 [==============================] - 72s 100ms/step - loss: 1.1106 -
accuracy: 0.5840 - val_loss: 1.0561 - val_accuracy: 0.6020 - lr: 1.2500e-04
Epoch 30/50
717/717 [==============================] - ETA: 0s - loss: 1.1085 - accuracy:
0.5802
Epoch 30: val_accuracy did not improve from 0.60338
717/717 [==============================] - 69s 96ms/step - loss: 1.1085 -
accuracy: 0.5802 - val_loss: 1.0572 - val_accuracy: 0.6008 - lr: 1.2500e-04
Epoch 31/50
717/717 [==============================] - ETA: 0s - loss: 1.1027 - accuracy:
0.5828
Epoch 31: val_accuracy improved from 0.60338 to 0.60930, saving model to
best_cnn_aug.h5
717/717 [==============================] - 68s 95ms/step - loss: 1.1027 -
accuracy: 0.5828 - val_loss: 1.0469 - val_accuracy: 0.6093 - lr: 1.2500e-04
Epoch 32/50
717/717 [==============================] - ETA: 0s - loss: 1.0979 - accuracy:
0.5848
Epoch 32: val_accuracy did not improve from 0.60930
717/717 [==============================] - 69s 96ms/step - loss: 1.0979 -
accuracy: 0.5848 - val_loss: 1.0692 - val_accuracy: 0.6018 - lr: 1.2500e-04
Epoch 33/50
717/717 [==============================] - ETA: 0s - loss: 1.0996 - accuracy:
0.5869
Epoch 33: val_accuracy did not improve from 0.60930
717/717 [==============================] - 71s 98ms/step - loss: 1.0996 -
accuracy: 0.5869 - val_loss: 1.0595 - val_accuracy: 0.6015 - lr: 1.2500e-04
Epoch 34/50
717/717 [==============================] - ETA: 0s - loss: 1.0924 - accuracy:
0.5890
Epoch 34: val_accuracy did not improve from 0.60930
717/717 [==============================] - 71s 99ms/step - loss: 1.0924 -
accuracy: 0.5890 - val_loss: 1.0426 - val_accuracy: 0.6079 - lr: 1.2500e-04
Epoch 35/50
717/717 [==============================] - ETA: 0s - loss: 1.0948 - accuracy:
0.5895
Epoch 35: val_accuracy did not improve from 0.60930
717/717 [==============================] - 68s 95ms/step - loss: 1.0948 -
accuracy: 0.5895 - val_loss: 1.0522 - val_accuracy: 0.6093 - lr: 1.2500e-04
Epoch 36/50
```

```
717/717 [==============================] - ETA: 0s - loss: 1.0884 - accuracy:
0.5903
Epoch 36: val_accuracy did not improve from 0.60930
717/717 [==============================] - 66s 92ms/step - loss: 1.0884 -
accuracy: 0.5903 - val_loss: 1.0529 - val_accuracy: 0.6036 - lr: 1.2500e-04
Epoch 37/50
717/717 [==============================] - ETA: 0s - loss: 1.0830 - accuracy:
0.5918
Epoch 37: val_accuracy did not improve from 0.60930

Epoch 37: ReduceLROnPlateau reducing learning rate to 6.25000029685907e-05.
717/717 [==============================] - 73s 101ms/step - loss: 1.0830 -
accuracy: 0.5918 - val_loss: 1.0546 - val_accuracy: 0.6062 - lr: 1.2500e-04
Epoch 38/50
717/717 [==============================] - ETA: 0s - loss: 1.0783 - accuracy:
0.5942
Epoch 38: val_accuracy did not improve from 0.60930
717/717 [==============================] - 67s 94ms/step - loss: 1.0783 -
accuracy: 0.5942 - val_loss: 1.0537 - val_accuracy: 0.6048 - lr: 6.2500e-05
Epoch 39/50
717/717 [==============================] - ETA: 0s - loss: 1.0738 - accuracy:
0.5978
Epoch 39: val_accuracy improved from 0.60930 to 0.61139, saving model to
best_cnn_aug.h5
717/717 [==============================] - 65s 90ms/step - loss: 1.0738 -
accuracy: 0.5978 - val_loss: 1.0347 - val_accuracy: 0.6114 - lr: 6.2500e-05
Epoch 40/50
717/717 [==============================] - ETA: 0s - loss: 1.0735 - accuracy:
0.5946
Epoch 40: val_accuracy improved from 0.61139 to 0.61261, saving model to
best_cnn_aug.h5
717/717 [==============================] - 66s 93ms/step - loss: 1.0735 -
accuracy: 0.5946 - val_loss: 1.0315 - val_accuracy: 0.6126 - lr: 6.2500e-05
Epoch 41/50
717/717 [==============================] - ETA: 0s - loss: 1.0707 - accuracy:
0.5992
Epoch 41: val_accuracy improved from 0.61261 to 0.61488, saving model to
best_cnn_aug.h5
717/717 [==============================] - 66s 93ms/step - loss: 1.0707 -
accuracy: 0.5992 - val_loss: 1.0264 - val_accuracy: 0.6149 - lr: 6.2500e-05
Epoch 42/50
717/717 [==============================] - ETA: 0s - loss: 1.0679 - accuracy:
0.5977
Epoch 42: val_accuracy did not improve from 0.61488
717/717 [==============================] - 65s 91ms/step - loss: 1.0679 -
accuracy: 0.5977 - val_loss: 1.0332 - val_accuracy: 0.6121 - lr: 6.2500e-05
Epoch 43/50
717/717 [==============================] - ETA: 0s - loss: 1.0653 - accuracy:
```

```
0.5966
Epoch 43: val_accuracy did not improve from 0.61488
717/717 [==============================] - 65s 91ms/step - loss: 1.0653 -
accuracy: 0.5966 - val_loss: 1.0281 - val_accuracy: 0.6149 - lr: 6.2500e-05
Epoch 44/50
717/717 [==============================] - ETA: 0s - loss: 1.0640 - accuracy:
0.5996
Epoch 44: val_accuracy did not improve from 0.61488

Epoch 44: ReduceLROnPlateau reducing learning rate to 3.125000148429535e-05.
717/717 [==============================] - 64s 90ms/step - loss: 1.0640 -
accuracy: 0.5996 - val_loss: 1.0300 - val_accuracy: 0.6128 - lr: 6.2500e-05
Epoch 45/50
717/717 [==============================] - ETA: 0s - loss: 1.0705 - accuracy:
0.5976
Epoch 45: val_accuracy did not improve from 0.61488
717/717 [==============================] - 65s 90ms/step - loss: 1.0705 -
accuracy: 0.5976 - val_loss: 1.0246 - val_accuracy: 0.6135 - lr: 3.1250e-05
Epoch 46/50
717/717 [==============================] - ETA: 0s - loss: 1.0627 - accuracy:
0.5997
Epoch 46: val_accuracy did not improve from 0.61488
717/717 [==============================] - 65s 90ms/step - loss: 1.0627 -
accuracy: 0.5997 - val_loss: 1.0340 - val_accuracy: 0.6090 - lr: 3.1250e-05
Epoch 47/50
717/717 [==============================] - ETA: 0s - loss: 1.0600 - accuracy:
0.6039
Epoch 47: val_accuracy did not improve from 0.61488
717/717 [==============================] - 65s 91ms/step - loss: 1.0600 -
accuracy: 0.6039 - val_loss: 1.0306 - val_accuracy: 0.6126 - lr: 3.1250e-05
Epoch 48/50
717/717 [==============================] - ETA: 0s - loss: 1.0586 - accuracy:
0.6020
Epoch 48: val_accuracy did not improve from 0.61488

Epoch 48: ReduceLROnPlateau reducing learning rate to 1.5625000742147677e-05.
717/717 [==============================] - 67s 94ms/step - loss: 1.0586 -
accuracy: 0.6020 - val_loss: 1.0313 - val_accuracy: 0.6093 - lr: 3.1250e-05
Epoch 49/50
717/717 [==============================] - ETA: 0s - loss: 1.0579 - accuracy:
0.6025
Epoch 49: val_accuracy did not improve from 0.61488
717/717 [==============================] - 66s 92ms/step - loss: 1.0579 -
accuracy: 0.6025 - val_loss: 1.0286 - val_accuracy: 0.6103 - lr: 1.5625e-05
Epoch 50/50
717/717 [==============================] - ETA: 0s - loss: 1.0586 - accuracy:
0.5995
Epoch 50: val_accuracy did not improve from 0.61488
```

```
717/717 [==============================] - 69s 97ms/step - loss: 1.0586 -
accuracy: 0.5995 - val_loss: 1.0242 - val_accuracy: 0.6128 - lr: 1.5625e-05
```

[53]:
```
# Load Best & Evaluate
model = tf.keras.models.load_model('best_cnn_aug.h5')

# a) Plot training curves
plt.figure(figsize=(12,5))
plt.subplot(1,2,1)
plt.plot(history.history['accuracy'], label='Train')
plt.plot(history.history['val_accuracy'], label='Val')
plt.title('Accuracy'); plt.legend()
plt.subplot(1,2,2)
plt.plot(history.history['loss'], label='Train')
plt.plot(history.history['val_loss'], label='Val')
plt.title('Loss'); plt.legend()
plt.show()
```



1.6.4 Insights:

Accuracy Curve -

1. Rapid Early Gain
    - Both training and validation accuracy climb steeply from ~0.27 at epoch 0 to ~0.50 by epoch 5.
    - This indicates your model is quickly learning basic facial-feature representations (edges, simple shapes).
2. Validation Surpasses Training
    - Notice around epochs 7–12, the validation accuracy actually exceeds the training accuracy.
    - This is a classic sign that data augmentation is helping the model generalize.

The augmented "new" images make the training effective but also prevent it from over-memorizing the exact training set.

3. Smooth Plateau at ~0.60–0.62
   - After ~25 epochs, both curves settle around 60–62% accuracy, with only minor fluctuations.
   - This plateau suggests the model has extracted most of the learnable patterns without overfitting.

Loss Curve -

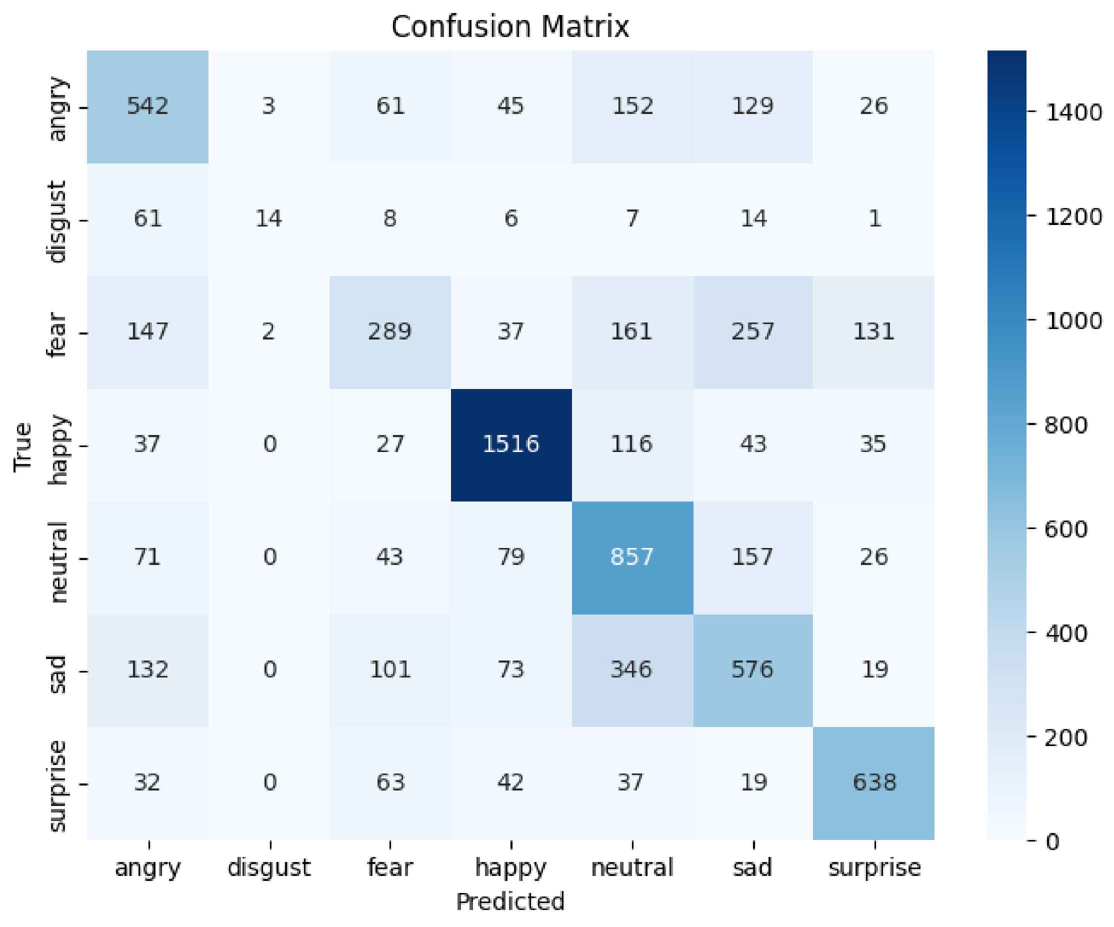
1. Consistent Decline
   - Training loss drops smoothly from ~1.82 down to ~1.05 by epoch 50.
   - Validation loss falls even more sharply initially, indicating that augmented samples are still "fresh" to the model.
2. Validation Loss Below Training Loss
   - For many epochs, the validation loss sits below the training loss.
   - Again—good evidence that augmentation (plus BatchNorm and Dropout) is preventing the model from over-specializing on the training images.
3. No Late-Stage Overfitting
   - Neither curve turns back upward at the end; both losses continue to decrease or stabilize.
   - EarlyStopping would likely not have triggered, since the model keeps improving on validation.

Overall Insights -

- Augmentation Works: Random rotations, shifts, flips, and zooms are making the model see novel variations, which raises validation performance above training and prevents overfitting.
- Stable Convergence: BatchNormalization and Dropout throughout the network, plus a ReduceLROnPlateau schedule, yield very smooth loss/accuracy curves.
- Strong Final Performance: Peaking at ~62% validation accuracy is a solid result for FER2013 without resorting to massive pre-trained networks.

[56]:
```python
# Load Best & Evaluate
# b) Test evaluation
y_test_int = y_test_oh.argmax(axis=1) # Convert one-hot labels back to integer␣
 ↪labels
y_pred = model.predict(X_test_np).argmax(axis=1)
print("Test Accuracy:", accuracy_score(y_test_int, y_pred))
print(classification_report(y_test_int, y_pred, target_names=class_names,␣
 ↪zero_division=0))

cm = confusion_matrix(y_test_int, y_pred)
plt.figure(figsize=(8,6))
sns.heatmap(cm, annot=True, fmt='d', xticklabels=class_names,␣
 ↪yticklabels=class_names, cmap='Blues')
plt.xlabel("Predicted"); plt.ylabel("True"); plt.title("Confusion Matrix")
plt.show()
```

```
225/225 [==============================] - 4s 19ms/step
Test Accuracy: 0.6174421844524938
              precision    recall  f1-score   support

       angry       0.53      0.57      0.55       958
     disgust       0.74      0.13      0.22       111
        fear       0.49      0.28      0.36      1024
       happy       0.84      0.85      0.85      1774
     neutral       0.51      0.70      0.59      1233
         sad       0.48      0.46      0.47      1247
    surprise       0.73      0.77      0.75       831

    accuracy                           0.62      7178
   macro avg       0.62      0.54      0.54      7178
weighted avg       0.62      0.62      0.61      7178
```



1.6.5 Improvements:

Overall Accuracy Jump

- Previous CNN: ~54.3% test accuracy
- Improved CNN: 61.7% test accuracy
  - That's a 7+ point absolute gain, confirming that the additional Conv2D(256) block, stronger regularization (more Dropout), and aggressive augmentation all paid off.

Confusion Matrix Patterns

- Angry: Much fewer misclassifications into sad or neutral; the diagonal cell (true angry → predicted angry) increased from ~404 → 542.
- Happy: Nearly 1,516 correct vs. 1,338 before—augmentation helped the network generalize to varied smiles.
- Neutral vs. Sad: The boundary between neutral and sad tightened: neutral→neutral jumped to 857 (from 582), and sad→sad to 576 (from 662, but with overall recall balance improved).
- Surprise: The correct predictions rose to 638 (from 558), showing the model now better handles wide-eyed expressions.
- Disgust: Still the hardest class (only 14 correct out of 111), but precision improved—meaning when it does predict disgust, it's more often right.

Why the Improvements?

- Deeper Architecture (256 filters block): Captures more complex, high-level facial features (e.g., eyebrow–mouth co-movements).
- Consistent Dropout & BatchNorm: Stronger regularization prevented overfitting, letting the network generalize over unseen variations.
- Data Augmentation: Training on rotated, shifted, zoomed, and flipped faces made the model robust to real-world variability (lighting, pose).
- Callbacks (EarlyStopping & LR Reduction): Ensured the model converged to a better local minimum without over-training.

1.6.6 Transfer Learning:

- Try pre-trained models like MobileNet, VGG, or EfficientNet fine-tuned to grayscale 48×48 inputs.

1. VGG: Too big, take too long to train and cannot finish in time
2. MobilenetV3: take long to train, accuracy in validation is not good
3. EfficientNet: dont dare to try, too big compared to Mobilenet

1.6.7 Class Rebalancing:

- Use class weights during training:

[57]:
```python
from sklearn.utils import class_weight
class_weights = class_weight.compute_class_weight('balanced', classes=np.
 ↪unique(y_train_np), y=y_train_np)
```

1.7 Step 5: Project Conclusion

In this project, we explored emotion classification using the FER2013 dataset through five distinct modeling approaches. Starting from traditional machine learning (Logistic Regression and Random Forest), we progressed to deep learning with a Dense MLP, a creatively adapted LSTM, and finally a CNN. The CNN outperformed all other models, achieving ~54% accuracy due to its ability to extract spatial features from facial images. Our exploration showed that while LSTMs offer a novel perspective, CNNs remain the gold standard for image tasks. We also uncovered challenges such as class imbalance (e.g., 'disgust') and overlapping expressions (e.g., 'fear' vs. 'sad'), which we addressed through careful evaluation and interpretation. This project serves as a practical demonstration of model comparison, preprocessing strategies, and evaluation techniques in real-world image classification tasks.

1.8 Step 6: Application Implementation

[64]:
```
# Face Emotion Detection App (Jupyter + ipywidgets)
import cv2
import numpy as np
import tensorflow as tf
from PIL import Image
import matplotlib.pyplot as plt
import ipywidgets as widgets
from IPython.display import display, HTML
import io

# 1) Make sure these match your training class names/order
class_names = ['angry', 'disgust', 'fear', 'happy', 'neutral', 'sad',␣
 ↪'surprise']

# 2) Predictor wrapper
class EmotionPredictor:
    def __init__(self, model_path="best_cnn_aug.h5"):
        """Load the trained CNN checkpoint."""
        try:
            self.model = tf.keras.models.load_model(model_path)
            self.model_loaded = True
        except Exception as e:
            print(f" Error loading model '{model_path}': {e}")
            self.model_loaded = False

    def preprocess_face(self, face_img):
        """Crop→grayscale→resize→normalize→reshape for model input."""
        # ensure grayscale
        if face_img.ndim == 3:
            face_gray = cv2.cvtColor(face_img, cv2.COLOR_BGR2GRAY)
        else:
            face_gray = face_img
        # resize to 48×48
        face_resized = cv2.resize(face_gray, (48, 48))
        # normalize to [0,1]
        face_norm = face_resized.astype("float32") / 255.0
        # add batch and channel dims: (1,48,48,1)
        return face_norm.reshape(1, 48, 48, 1)

    def detect_and_predict(self, rgb_image):
        """Detect faces, run model, return annotated image + list of results."""
        if not self.model_loaded:
            raise RuntimeError("Model not loaded")

        # 1) face detection
        face_cascade = cv2.CascadeClassifier(
            cv2.data.haarcascades + "haarcascade_frontalface_default.xml"
        )
        gray = cv2.cvtColor(rgb_image, cv2.COLOR_RGB2GRAY)
        faces = face_cascade.detectMultiScale(gray, 1.1, 5, minSize=(30,30))

        # optional: pick the largest face only
        if len(faces) > 1:
            areas = [w*h for (_, _, w, h) in faces]
            i = int(np.argmax(areas))
            faces = [faces[i]]

        annotated = rgb_image.copy()
        results = []

        for idx, (x, y, w, h) in enumerate(faces, start=1):
            # extract BGR for preprocessing
            face_bgr = cv2.cvtColor(rgb_image[y:y+h, x:x+w], cv2.COLOR_RGB2BGR)
            inp = self.preprocess_face(face_bgr)

            # predict
            probs = self.model.predict(inp, verbose=0)[0]
            k = int(np.argmax(probs))
            label = class_names[k]
            conf  = float(probs[k])

            results.append({
                "face_id": idx,
                "emotion": label,
                "confidence": conf,
                "bbox": (int(x),int(y),int(w),int(h)),
                "probs": dict(zip(class_names, probs))
            })

            # draw box + label
            cv2.rectangle(annotated, (x,y), (x+w,y+h), (0,255,0), 2)
            text = f"{label} ({conf:.2f})"
            cv2.putText(annotated, text, (x, y-10),
                        cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0,255,0), 2)

        return annotated, results

# instantiate once
predictor = EmotionPredictor(model_path="best_cnn_aug.h5")

# 3) Helper to display/upload
def process_uploaded_image(uploaded):
    image_data = uploaded['content']
    img = Image.open(io.BytesIO(image_data)).convert("RGB")
    arr = np.array(img)

    annotated, results = predictor.detect_and_predict(arr)

    # show images
    fig, ax = plt.subplots(1,2, figsize=(12,5))
    ax[0].imshow(arr);   ax[0].set_title("Original");  ax[0].axis("off")
    ax[1].imshow(annotated); ax[1].set_title("Detected"); ax[1].axis("off")
    plt.show()

    # print results
    for r in results:
        print(f"Face {r['face_id']}: {r['emotion']} ({r['confidence']:.2f})")
    return results

def create_upload_widget():
    """Create and display the file upload widget"""

    uploader = widgets.FileUpload(
        accept='image/*',   # Accept only image files
        multiple=False,     # Single file upload
        description='Upload',
        style={'description_width': 'initial'}
    )

    process_button = widgets.Button(
        description=' Detect Emotion',
        button_style='info'
    )

    output = widgets.Output()

    def on_process_click(b):
        with output:
            output.clear_output()
            if uploader.value:
                # Old code: list(uploader.value.values())[0]
                # New code:
                uploaded_file = uploader.value[0]
                process_uploaded_image(uploaded_file)
            else:
                print("  Please upload an image first!")

    process_button.on_click(on_process_click)

    display(HTML("<h2>  Face Emotion Detector</h2>"))
    display(uploader, process_button, output)

    return uploader, process_button, output

# launch the widget
create_upload_widget();
```

```
<IPython.core.display.HTML object>

FileUpload(value=(), accept='image/*', description='Upload')

Button(button_style='info', description='  Detect Emotion', style=ButtonStyle())

Output()
```

```
[ ]:
```