COMPUTER VISION

Classification Emotional Detection Report

Group 4:

Abdurrahman Khairi - 001202300163

Filbert Sembiring Meliala - 001202300159

Jason Anthony Wibowo - 001202300216

Sarah Kimberly Fischer - 001202300177

INFORMATICS STUDY PROGRAM

FACULTY OF COMPUTER SCIENCE

PRESIDENT UNIVERSITY

JANUARY 2025

TABLE OF CONTENTS

COVER .... **I**

TABLE OF CONTENTS .... **II**

CHAPTER I BACKGROUND AND PROBLEM STATEMENT **4**

1.1 BACKGROUND 4

1.2 PROBLEM STATEMENT 5

CHAPTER II TOOLS **6**

2.1 OBJECTIVES 6

2.2 PROJECT BENEFITS 7

CHAPTER III DATASET AND PROGRAM **8**

3.1 DATASET 8

3.2 CODE 9

CHAPTER IV RESULT AND EXPLANATION **23**

4.1 COMPARES ARCHITECTURES 23

4.1.1 Block_CNN 23

4.1.2 ResNet50 24

4.1.3 ResNet18 25

4.1.4 DenseNet 26

4.2 JUSTIFICATION FOR CNN APPROACH 27

4.2.1 Superior Performance 27

4.2.1 Efficient Processing 27

4.2.2 Practical Benefits 28

4.3 COMPARATIVE ANALYSIS 28

4.3.1 CNN vs. ResNet50 28

4.3.2 CNN vs. ResNet18 28

4.3.3 CNN vs. DenseNet 29

4.4 RESULT 29

4.4.1 Result Prediction (25 images random) 29

4.4.2 Result Real-Time Face Detection 30
4.5 OBSTACLES 30
4.5.1 Not All Emotions Can Be Matched: 30
4.5.2 Data Imbalance: 31
4.6 IMPROVEMENTS 31
4.6.1 Use Real Data Instead of Sample Data (Kaggle): 31
4.6.2 Balance the Dataset: 31
4.6.3 Improve Preprocessing: 32
4.6.4 Update Libraries and Environment: 32
4.6.5 Enhance Model Architecture: 32
4.6.6 Incorporate Real-Time Feedback: 32
4.7 SUMMARY OF IMPROVEMENTS 33
4.8 EXPECTED OUTCOMES 33
4.9 CONCLUSION 34

PRESIDENT UNIVERSITY

CHAPTER I
BACKGROUND AND PROBLEM STATEMENT

1.1 BACKGROUND

Many people find it difficult to manage a wide range of emotions due to the fast-paced nature of modern life. Understanding and expressing our emotional states has grown more difficult as we negotiate demanding jobs, complicated relationships, and social pressures. Many adults struggle to properly express their emotions, which causes internal conflict that impacts their personal and professional lives.

Society as a whole suffers as a result of low emotional intelligence. The incapacity to identify and process emotions is a contributing factor in an increasing number of cases of anxiety, depression, and burnout, according to mental health professionals. This shows up at work as decreased output, tense team dynamics, and communication breakdowns. Relationships suffer at home as people find it difficult to express their genuine emotions and establish meaningful connections with their loved ones.

Researchers are looking into novel approaches to help people better understand their emotional states in order to address these issues. A promising solution is emotion detection technology, which can recognise seven basic emotions from facial expressions: happy, angry, sad, surprise, fear, disgust, and neutral states. This tool may act as an impartial mirror, assisting people in becoming more emotionally self-aware and creating more genuine connections in their day-to-day interactions.

1.2 PROBLEM STATEMENT

Being able to detect and express emotions is a challenge that deepens with time, especially in a fast paced world where so many people find it hard to identify their own feelings. Studies show that today, people are facing mental health issues and struggling with relationships because of the social barriers set by norms alongside the ever evolving boundaries of work life balance. With all these factors taken place, awareness around the feelings and emotional expression becomes obsolete. This results in higher amounts of stress, anxiety, and other psychological issues.

The issue of emotional suppression combined with social shame is prevalent within workplaces. Many professionals find themselves incapable of controlling their emotions resulting in emotional burnout which leads to unhappiness within the job all together. The struggle to control emotion in professional settings proves to be overwhelming, which leads to a reversal of the intended outcome. Lack of mental health resources paired with minimal tools to assess one's own emotion becomes a catalyst to emotional turmoil.

These revelations reinforce the idea that more technology driven ways to detect and express emotion have the power to change the way emotional understanding works today. In the modern world, recognized universally, it is set that the older methods of providing emotional support need to be transformed. In order to close this gap, this project suggests an AI-based emotion classification system that can identify seven fundamental emotions: happy, angry, sad, surprise, fear, disgust, and neutral. This technology seeks to improve interpersonal communication and self-awareness by giving users unbiased feedback about their emotional states.

PRESIDENT UNIVERSITY

CHAPTER II

TOOLS

2.1 OBJECTIVES

This project aims to construct an AI driven emotion recognition system that will aid in alleviating communication barriers evoked by emotional unawareness. The fundamental objectives are:

2.3.1 Enhance Emotional Recognition Improvement

Build a a real time compact system that can accurately recognise and arrange the unique seven basic emotions of human beings; happy, angry, sad, surprise, fear, disgust, and neutral while giving prompt feedback on emotional states..

2.3.2 Improve Emotional Awareness Improvement

Enable people improve on their self emotional intelligence and self control by learning about their emotions in different situations.

2.3.3 Support Professional Development

Assist professionals in self monitoring their emotions in working places, which will encourage effective communication and reduce emotional exhaustion.

2.3.4 Create Real-time Analysis

Create a facial expression recognition system that can analyze and give feedback on emotional state in real time.

2.3.5 Ensure Accuracy and Reliability

Create strong algorithms that are able to efficiently classify different features of emotions depending on the facial structure, light and camera angles.

2.2 PROJECT BENEFITS

Derived from the objectives of this project, there are outlined benefits belows:

2.4.1 Enhanced Emotional Intelligence

The application enables users gain deeper insights into their emotional patterns, leading to improved self-awareness and better emotional regulation in personal and professional contexts.

2.4.2 Improved Communication

This Real-time emotion detection app helps bridge communication gaps, enabling more effective interpersonal interactions and reducing misunderstandings in various social settings.

2.4.3 Mental Health Support

The app serves as a preventive tool for mental health management, helping users identify and address emotional challenges before they escalate into more serious issues.

2.4.4 Professional Growth

Professionals can better navigate workplace emotions, leading to improved team dynamics, reduced stress, and enhanced job satisfaction.

2.4.5 Personal Development

The app is designed to users can develop better emotional regulation strategies based on accurate recognition of their emotional states.

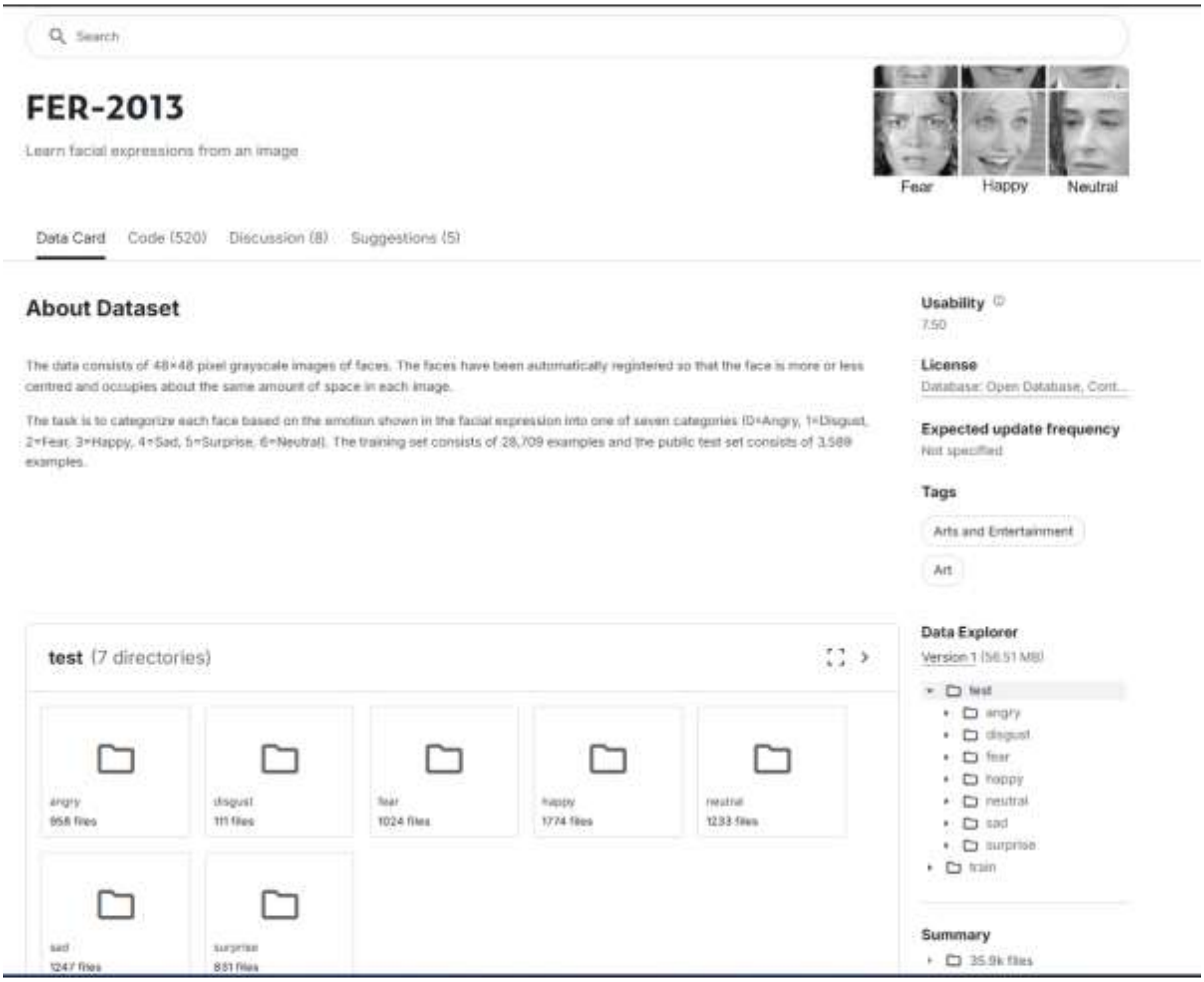
Combined, these benefits contribute to improved emotional well-being, better interpersonal relationships, and enhanced professional performance in modern society.

CHAPTER III

DATASET AND PROGRAM

3.1 DATASET

https://www.kaggle.com/datasets/msambare/fer2013/data



3.2 CODE

jupyter Emotion Recoginition Project CV Last Checkpoint: 7 hours ago

File Edit View Run Kernel Settings Help Trusted

Code JupyterLab Python 3 (ipykernel)

Emotion Recognition

Create a CNN models for emotion recognition from facial image.

Compare Model Architectures

1. Block CNN: Using only Conv2D, Pooling Layers, Dense Layers, BatchNorm, LayerNorm, Activation Layers, Dropout
2. Resnet50
3. Resnet18
4. DenseNet
5. Sequential

Import Libraries Needed

[1]:
```python
# Import Libraries Needed

import os
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
from torchvision import models, transforms
from sklearn.metrics import confusion_matrix, classification_report, f1_score, accuracy_score
from sklearn.preprocessing import LabelEncoder
from sklearn.utils.class_weight import compute_class_weight
from torchvision.transforms import RandomHorizontalFlip, RandomRotation, RandomAffine, RandomPerspective, RandomResizedCrop, Compose
```

Hyperparameter initializing

[2]:
```python
# Constants
RANDOM_STATE = 42
LEARNING_RATE = 0.0001
EPOCHS = 30
BATCH_SIZE = 32
NUM_CLASSES = 7
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

[3]:
```python
# Set random seed
torch.manual_seed(RANDOM_STATE)
```

[3]:
```
<torch._C.Generator at 0x1d0f844d970>
```

Load and Preprocess Data

[4]:
```python
# Load data
data_path = "fer13_imagedataset.npy"
data = np.load(data_path, allow_pickle=True).item()

X_train, y_train = data['train']
X_val, y_val = data['val']
X_test, y_test = data['test']
```

[5]:
```python
# Convert string labels to integers
label_encoder = LabelEncoder()
y_train = label_encoder.fit_transform(y_train)
y_val = label_encoder.transform(y_val)
y_test = label_encoder.transform(y_test)
```

```
[6]: # Convert to PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.float32).unsqueeze(1)
X_val = torch.tensor(X_val, dtype=torch.float32).unsqueeze(1)
X_test = torch.tensor(X_test, dtype=torch.float32).unsqueeze(1)

y_train = torch.tensor(y_train, dtype=torch.long)
y_val = torch.tensor(y_val, dtype=torch.long)
y_test = torch.tensor(y_test, dtype=torch.long)
```

```
[7]: # Normalize data
X_train = X_train / 255.0
X_val = X_val / 255.0
X_test = X_test / 255.0
```

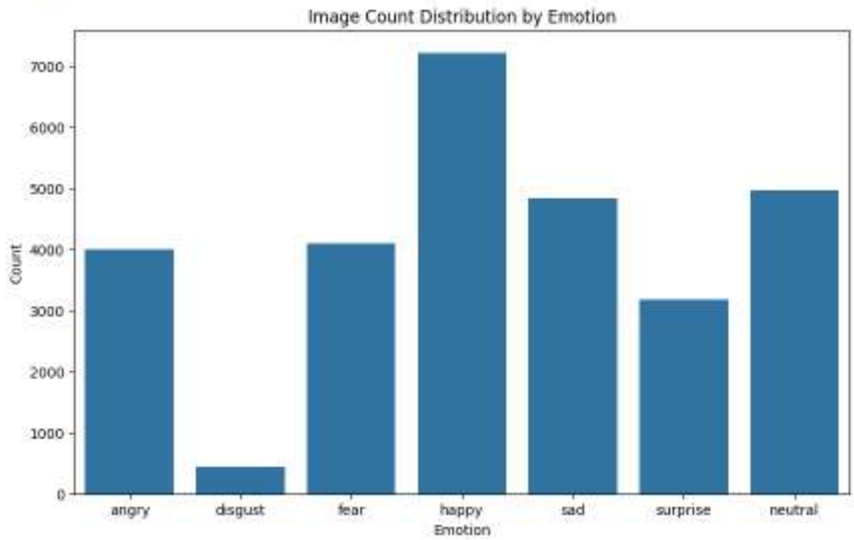
```
[8]: # Create DataLoader
train_dataset = TensorDataset(X_train, y_train)
val_dataset = TensorDataset(X_val, y_val)
test_dataset = TensorDataset(X_test, y_test)

train_dataloader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
val_dataloader = DataLoader(val_dataset, batch_size=BATCH_SIZE)
test_dataloader = DataLoader(test_dataset, batch_size=BATCH_SIZE)
```

Data Visualization

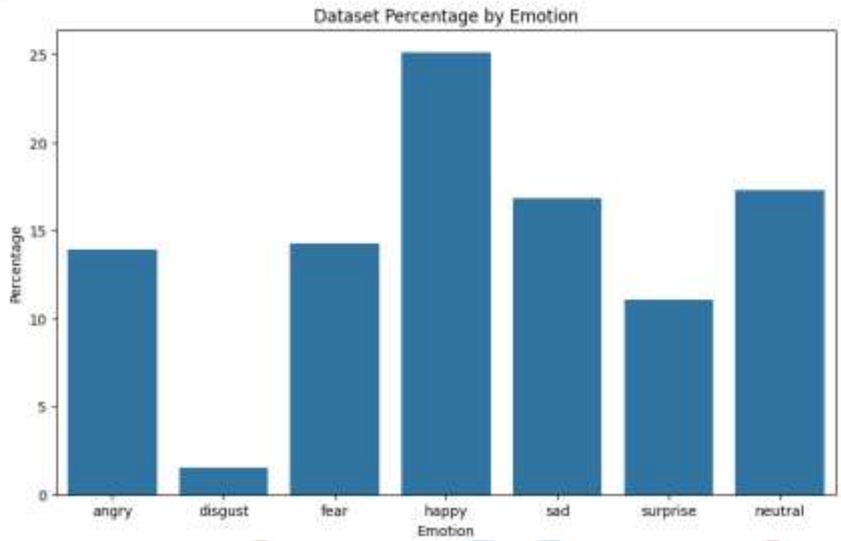
```
[9]: # Data Visualization
# 1. Count Subfolders and Image Count Distribution
emotions = np.array(['angry', 'disgust', 'fear', 'happy', 'sad', 'surprise', 'neutral'])
train_path = r"C:/College/Semester 5/Computer Vision/Project/Mini Project/archive/train"
```

```
[10]: # Count images in each subfolder
emotion_counts = {}
for emotion in emotions:
    emotion_folder = os.path.join(train_path, emotion)
    if os.path.exists(emotion_folder):
        emotion_counts[emotion] = len(os.listdir(emotion_folder))
```

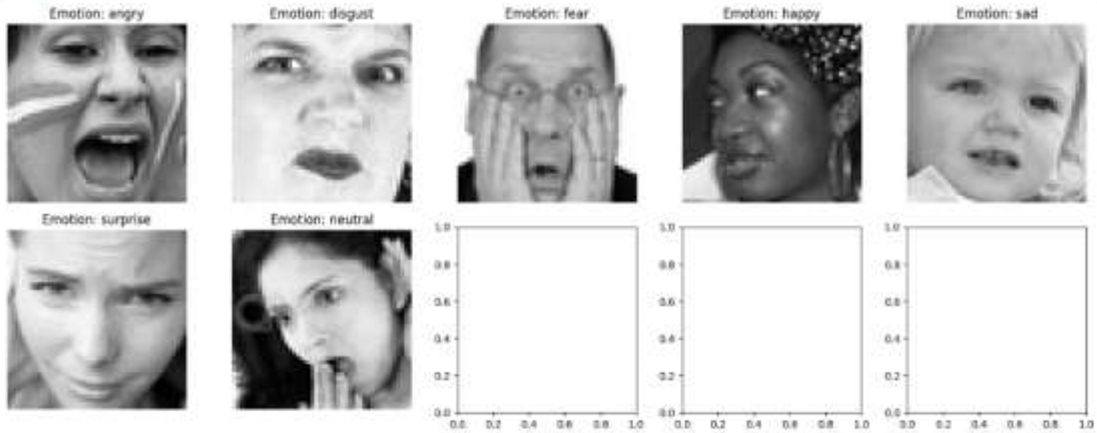
```
[11]: # Plot image count distribution
plt.figure(figsize=(10, 6))
sns.barplot(x=list(emotion_counts.keys()), y=list(emotion_counts.values()))
plt.title("Image Count Distribution by Emotion")
plt.xlabel("Emotion")
plt.ylabel("Count")
plt.show()
```



[13]:
```python
# 2. Dataset Percentage
total_images = sum(emotion_counts.values())
emotion_percentages = {emotion: (count / total_images) * 100 for emotion, count in emotion_counts.items()}

plt.figure(figsize=(10, 6))
sns.barplot(x=list(emotion_percentages.keys()), y=list(emotion_percentages.values()))
plt.title("Dataset Percentage by Emotion")
plt.xlabel("Emotion")
plt.ylabel("Percentage")
plt.show()
```



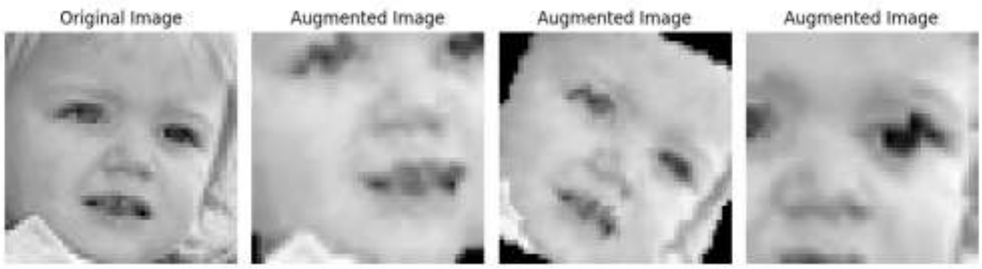
[14]:
```python
# 3. Display Sample Images
fig, axes = plt.subplots(nrows=2, ncols=5, figsize=(15, 6))
for i, emotion in enumerate(emotions):
    sample_image = X_train[y_train == i][0].squeeze().cpu().numpy()
    axes[i // 5, i % 5].imshow(sample_image, cmap='gray')
    axes[i // 5, i % 5].set_title(f"Emotion: {emotion}")
    axes[i // 5, i % 5].axis('off')
plt.tight_layout()
plt.show()
```



Data Augmentation

[15]:
```python
# 4. Data Augmentation Visualization
data_augmentation = transforms.Compose([
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomRotation(degrees=15),
    transforms.RandomAffine(degrees=15, translate=(0.1, 0.1)),
    transforms.RandomResizedCrop(size=(48, 48), scale=(0.8, 1.0))
])
```

[16]:
```python
# Visualize augmented images
fig, axes = plt.subplots(nrows=1, ncols=5, figsize=(15, 5))
sample_image = X_train[0].unsqueeze(0)
for i in range(5):
    augmented_image = data_augmentation(sample_image).squeeze().cpu().numpy()
    axes[i].imshow(augmented_image, cmap='gray')
    axes[i].set_title(f"Augmented {i + 1}")
    axes[i].axis('off')
plt.tight_layout()
plt.show()
```




```python
# Apply data augmentation to the dataset
augmented_images = []
augmented_labels = []
for image, label in zip(X_train, y_train):
    augmented_images.append(data_augmentation(image.unsqueeze(0)).squeeze(0))
    augmented_labels.append(label)
    if len(augmented_images) >= len(X_train):  # Double the dataset size
        break

X_train_augmented = torch.stack(augmented_images)
y_train_augmented = torch.stack(augmented_labels)
```

```python
# Combine original and augmented data
X_train = torch.cat([X_train, X_train_augmented], dim=0)
y_train = torch.cat([y_train, y_train_augmented], dim=0)
```

Block CNN Model Defining

```python
from torch.utils.data import DataLoader, TensorDataset
from torch.nn import Module, Conv2d, Linear, MaxPool2d, ReLU, Dropout, BatchNorm2d , Flatten, Softmax
from torch.optim import Adam
from torchsummary import summary
```

```python
# create dataloader
train_dataset = TensorDataset(X_train, y_train)
val_dataset = TensorDataset(X_val, y_val)
test_dataset = TensorDataset(X_test, y_test)

train_dataloader = DataLoader(train_dataset, batch_size = BATCH_SIZE,)
val_dataloader = DataLoader(val_dataset, batch_size = BATCH_SIZE,)
test_dataloader = DataLoader(test_dataset, batch_size = BATCH_SIZE,)
```

```python
class CNN(Module):
    def __init__(self, channels, num_classes):
        super().__init__()

        # Convolutional layers
        self.conv1 = Conv2d(in_channels=channels, out_channels=32, kernel_size=(5, 5), padding=2)
        self.conv2 = Conv2d(in_channels=32, out_channels=64, kernel_size=(5, 5), padding=2)
        self.conv3 = Conv2d(in_channels=64, out_channels=128, kernel_size=(5, 5), padding=2)
        self.conv4 = Conv2d(in_channels=128, out_channels=512, kernel_size=(5, 5), padding=2)
        self.conv5 = Conv2d(in_channels=512, out_channels=1024, kernel_size=(3, 3), padding=1)

        # Batch normalization
        self.batchnorm1 = BatchNorm2d(num_features=32)
        self.batchnorm2 = BatchNorm2d(num_features=64)
        self.batchnorm3 = BatchNorm2d(num_features=128)
        self.batchnorm4 = BatchNorm2d(num_features=512)
        self.batchnorm5 = BatchNorm2d(num_features=1024)

        # Other layers
        self.maxpool = MaxPool2d(kernel_size=(2, 2))
        self.relu = ReLU()
        self.dropout = Dropout(p=0.2)

        # Fully connected layers
        self.flatten = Flatten()
        self.fc1 = Linear(in_features=1024, out_features=512)
        self.fc2 = Linear(in_features=512, out_features=128)
        self.fc3 = Linear(in_features=128, out_features=num_classes)
        self.softmax = Softmax(dim=1)



    def forward(self, X):
        # block 1
        X = self.conv1(X)
        X = self.batchnorm1(X)
        X = self.relu(X)
        X = self.maxpool(X)
```

```python
        # block 2
        X = self.conv2(X)
        X = self.batchnorm2(X)
        X = self.relu(X)
        X = self.maxpool(X)

        # block 3
        X = self.conv3(X)
        X = self.batchnorm3(X)
        X = self.relu(X)
        X = self.maxpool(X)

        # block 4
        X = self.conv4(X)
        X = self.batchnorm4(X)
        X = self.relu(X)
        X = self.maxpool(X)
        X = self.dropout(X)

        # block 5
        X = self.conv5(X)
        X = self.batchnorm5(X)
        X = self.relu(X)
        X = self.maxpool(X)
        X = self.dropout(X)


        # fully connected layer
        X = self.flatten(X)
        X = self.fc1(X)
        X = self.relu(X)
        X = self.dropout(X)

        X = self.fc2(X)
        X = self.relu(X)
        X = self.dropout(X)

        X = self.fc3(X)
        return X
```

```python
[45]: model = CNN(channels = 1, num_classes = NUM_CLASSES).to(device)
```

```python
[46]: optimizer = Adam(params = model.parameters(), lr = LEARNING_RATE)
```

```python
[47]: # loss fn
loss_fn = torch.nn.CrossEntropyLoss(weight=torch.tensor(class_weights, dtype=torch.float32).to(device))
```

```
C:\Users\abdur\AppData\Local\Temp\ipykernel_30304\2094615562.py:2: UserWarning: To copy construct from a tensor, it is recommended to use sourceTensor.c
lone().detach() or sourceTensor.clone().detach().requires_grad_(True), rather than torch.tensor(sourceTensor).
  loss_fn = torch.nn.CrossEntropyLoss(weight=torch.tensor(class_weights, dtype=torch.float32).to(device))
```

```python
[48]: summary(model, (1, 48, 48))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 48, 48]             832
       BatchNorm2d-2           [-1, 32, 48, 48]              64
              ReLU-3           [-1, 32, 48, 48]               0
         MaxPool2d-4           [-1, 32, 24, 24]               0
            Conv2d-5           [-1, 64, 24, 24]          51,264
       BatchNorm2d-6           [-1, 64, 24, 24]             128
              ReLU-7           [-1, 64, 24, 24]               0
         MaxPool2d-8           [-1, 64, 12, 12]               0
            Conv2d-9          [-1, 128, 12, 12]         204,928
      BatchNorm2d-10          [-1, 128, 12, 12]             256
             ReLU-11          [-1, 128, 12, 12]               0
        MaxPool2d-12            [-1, 128, 6, 6]               0
           Conv2d-13            [-1, 512, 6, 6]       1,638,912
      BatchNorm2d-14            [-1, 512, 6, 6]           1,024
             ReLU-15            [-1, 512, 6, 6]               0
        MaxPool2d-16            [-1, 512, 3, 3]               0
          Dropout-17            [-1, 512, 3, 3]               0
           Conv2d-18           [-1, 1024, 3, 3]       4,719,616
      BatchNorm2d-19           [-1, 1024, 3, 3]           2,048
             ReLU-20           [-1, 1024, 3, 3]               0
        MaxPool2d-21           [-1, 1024, 1, 1]               0
          Dropout-22           [-1, 1024, 1, 1]               0
          Flatten-23                 [-1, 1024]               0
           Linear-24                  [-1, 512]         524,800
             ReLU-25                  [-1, 512]               0
          Dropout-26                  [-1, 512]               0
           Linear-27                  [-1, 128]          65,664
             ReLU-28                  [-1, 128]               0
          Dropout-29                  [-1, 128]               0
           Linear-30                    [-1, 7]             903
================================================================
----------------------------------------------------------------
Total params: 7,210,439
Trainable params: 7,210,439
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 3.94
Params size (MB): 27.51
Estimated Total Size (MB): 31.45
----------------------------------------------------------------
```

[47]:
```
# training loop
def train_epoch(model, dataloader, loss_fn, optimizer):
  # turn on train phase
  model.train()
  running_loss = 0.0
  correct = 0
  total = 0

  for inputs, labels in dataloader:
    inputs, labels = inputs.to(device), labels.to(device)

    # forward pass
    outputs = model(inputs)

    # calculate loss
    loss = loss_fn(outputs, labels)

    # zero gradient
    optimizer.zero_grad()

    # backpropagation
    loss.backward()

    # update
    optimizer.step()

    # save loss and accuracy
    running_loss += loss.item()
    _, predicted = torch.max(outputs.data, 1)
    total += labels.size(0)
    correct += (predicted == labels).sum().item()

  epoch_loss = running_loss / len(dataloader)
  epoch_acc = 100 * correct / total

  return epoch_loss, epoch_acc
```

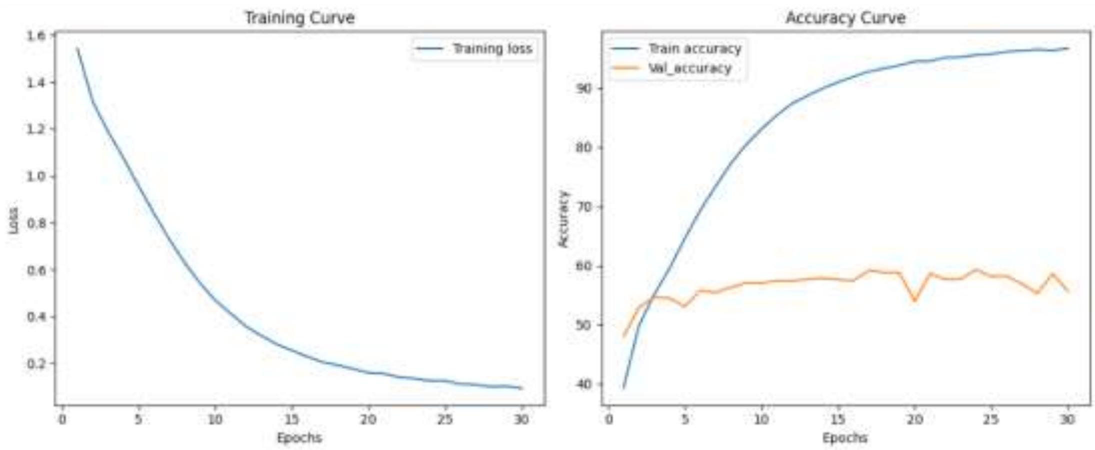
[48]:
```
# testing loop
def test_epoch(model, dataloader, loss_fn):
  # turn on test phase
  model.eval()
  correct = 0
  total = 0

  with torch.inference_mode():
    for inputs, labels in dataloader:
      inputs, labels = inputs.to(device), labels.to(device)
      outputs = model(inputs)
      _, predicted = outputs.max(1)
      total += labels.size(0)
      correct += (predicted == labels).sum().item()

  epoch_acc = 100 * correct / total

  return epoch_acc
```

[49]:
```
import h5py  # Library untuk menyimpan model ke .h5

# Jumlah epoch diubah menjadi 30
EPOCHS = 30

# Inisialisasi variabel untuk menyimpan hasil training
best_acc = 0.0
train_loss_list = []
train_acc_list = []
val_acc_list = []

# Fungsi untuk menyimpan model ke .h5
def save_model_to_h5(model, file_path):
    with h5py.File(file_path, 'w') as f:
        for key, value in model.state_dict().items():
            f.create_dataset(key, data=value.cpu().numpy())

# Loop training untuk 30 epoch
for epoch in range(EPOCHS):
    # Training
    train_loss, train_acc = train_epoch(model, train_dataloader, loss_fn, optimizer)

    # Validation
    val_acc = test_epoch(model, val_dataloader, loss_fn)

    # Simpan hasil training dan validation
    train_loss_list.append(train_loss)
    train_acc_list.append(train_acc)
    val_acc_list.append(val_acc)

    # Simpan model terbaik ke .h5
    if val_acc > best_acc:
        best_acc = val_acc
        save_model_to_h5(model, 'best_model.h5')  # Simpan ke .h5
        print(f"Best model saved to best_model.h5 at epoch {epoch + 1}")

    # Tampilkan hasil setiap epoch
    print(f"Epoch [{epoch + 1}/{EPOCHS}]")
    print(f"Train Loss: {train_loss:.4f} | Train Acc: {train_acc:.2f}%")
    print(f"Val Acc: {val_acc:.2f}%")
    print('-' * 50)
```

```
Best model saved to best_model.h5 at epoch 1
Epoch [1/30]
Train Loss: 1.5437 | Train Acc: 39.28%
Val Acc: 40.03%
--------------------------------------------------
Best model saved to best_model.h5 at epoch 2
Epoch [2/30]
Train Loss: 1.3169 | Train Acc: 49.83%
Val Acc: 52.87%
--------------------------------------------------
Best model saved to best_model.h5 at epoch 3
Epoch [3/30]
Train Loss: 1.1806 | Train Acc: 55.09%
Val Acc: 54.63%
--------------------------------------------------
Epoch [4/30]
Train Loss: 1.0759 | Train Acc: 59.54%
Val Acc: 54.51%
--------------------------------------------------
Epoch [5/30]
Train Loss: 0.9551 | Train Acc: 64.58%
Val Acc: 53.10%
--------------------------------------------------
Best model saved to best_model.h5 at epoch 6
Epoch [6/30]
Train Loss: 0.8391 | Train Acc: 69.30%
Val Acc: 55.78%
--------------------------------------------------
Epoch [7/30]
Train Loss: 0.7292 | Train Acc: 73.30%
Val Acc: 55.43%
--------------------------------------------------
Best model saved to best_model.h5 at epoch 8
Epoch [8/30]
Train Loss: 0.6288 | Train Acc: 77.24%
Val Acc: 56.32%
--------------------------------------------------
Best model saved to best_model.h5 at epoch 9
Epoch [9/30]
Train Loss: 0.5423 | Train Acc: 80.47%
Val Acc: 57.05%
--------------------------------------------------
Epoch [10/30]
Train Loss: 0.4681 | Train Acc: 83.10%
Val Acc: 57.02%
--------------------------------------------------
Best model saved to best_model.h5 at epoch 11
Epoch [11/30]
Train Loss: 0.4128 | Train Acc: 85.44%
Val Acc: 57.45%
--------------------------------------------------
Epoch [12/30]
Train Loss: 0.3571 | Train Acc: 87.42%
Val Acc: 57.44%
--------------------------------------------------
Best model saved to best_model.h5 at epoch 13
Epoch [13/30]
Train Loss: 0.3179 | Train Acc: 88.74%
Val Acc: 57.70%
--------------------------------------------------
Best model saved to best_model.h5 at epoch 14
Epoch [14/30]
Train Loss: 0.2817 | Train Acc: 89.99%
Val Acc: 57.87%
--------------------------------------------------
Epoch [15/30]
Train Loss: 0.2545 | Train Acc: 91.03%
Val Acc: 57.65%
--------------------------------------------------
Epoch [16/30]
Train Loss: 0.2282 | Train Acc: 91.98%
Val Acc: 57.45%
--------------------------------------------------
Best model saved to best_model.h5 at epoch 17
Epoch [17/30]
Train Loss: 0.2058 | Train Acc: 92.82%
Val Acc: 59.18%
--------------------------------------------------
Epoch [18/30]
Train Loss: 0.1909 | Train Acc: 93.35%
Val Acc: 58.85%
--------------------------------------------------
Epoch [19/30]
Train Loss: 0.1763 | Train Acc: 93.87%
Val Acc: 58.76%
--------------------------------------------------
Epoch [20/30]
Train Loss: 0.1596 | Train Acc: 94.53%
Val Acc: 53.99%
--------------------------------------------------
Epoch [21/30]
Train Loss: 0.1558 | Train Acc: 94.58%
Val Acc: 58.64%
--------------------------------------------------
Epoch [22/30]
Train Loss: 0.1404 | Train Acc: 95.20%
Val Acc: 57.70%
--------------------------------------------------
Epoch [23/30]
Train Loss: 0.1340 | Train Acc: 95.27%
Val Acc: 57.70%
--------------------------------------------------
Best model saved to best_model.h5 at epoch 24
Epoch [24/30]
Train Loss: 0.1254 | Train Acc: 95.62%
Val Acc: 59.30%
```

```
-----------------------------------------------------
Epoch [25/30]
Train Loss: 0.1253 | Train Acc: 95.76%
Val Acc: 58.17%
-----------------------------------------------------
Epoch [26/30]
Train Loss: 0.1111 | Train Acc: 96.18%
Val Acc: 58.20%
-----------------------------------------------------
Epoch [27/30]
Train Loss: 0.1086 | Train Acc: 96.35%
Val Acc: 56.93%
-----------------------------------------------------
Epoch [28/30]
Train Loss: 0.1002 | Train Acc: 96.58%
Val Acc: 55.29%
-----------------------------------------------------
Epoch [29/30]
Train Loss: 0.1020 | Train Acc: 96.40%
Val Acc: 58.66%
-----------------------------------------------------
Epoch [30/30]
Train Loss: 0.0940 | Train Acc: 96.77%
Val Acc: 55.64%
-----------------------------------------------------
```

[50]:
```
fig, axes = plt.subplots(ncols = 2, nrows = 1, figsize = (12, 5))

# Plot training curve
sns.lineplot(x = range(1, EPOCHS+1), y = train_loss_list, ax = axes[0], label = "Training loss")
axes[0].set_title("Training Curve")
axes[0].set_xlabel("Epochs")
axes[0].set_ylabel("Loss")

# Plot accuracy curve
sns.lineplot(x = range(1, EPOCHS+1), y = train_acc_list, ax = axes[1], label = "Train accuracy")
sns.lineplot(x = range(1, EPOCHS+1), y = val_acc_list, ax = axes[1], label = "Val_accuracy")
axes[1].set_title("Accuracy Curve")
axes[1].set_xlabel("Epochs")
axes[1].set_ylabel("Accuracy")

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



[51]:
```
from sklearn.metrics import confusion_matrix, classification_report, f1_score, accuracy_score
```

[52]:
```
y_pred = []
y_true = []

model.load_state_dict(torch.load('./best_model.pth'))
model.eval()

with torch.inference_mode():
  for inputs, labels in test_dataloader:
    inputs, labels = inputs.to(device), labels.to(device)
    outputs = model(inputs)
    _, predicted = outputs.max(1)
    y_pred.extend(predicted.cpu().numpy())
    y_true.extend(labels.cpu().numpy())

  print(classification_report(y_true, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.53      0.46      0.49       958
           1       0.68      0.44      0.54       111
           2       0.55      0.31      0.40      1024
           3       0.80      0.76      0.78      1774
           4       0.50      0.59      0.54      1233
           5       0.43      0.53      0.48      1247
           6       0.66      0.80      0.72       831

    accuracy                           0.59      7178
   macro avg       0.59      0.56      0.56      7178
weighted avg       0.59      0.59      0.58      7178
```

[53]:
```
print("F1 score macro:", f1_score(y_true, y_pred, average = 'macro'))
```

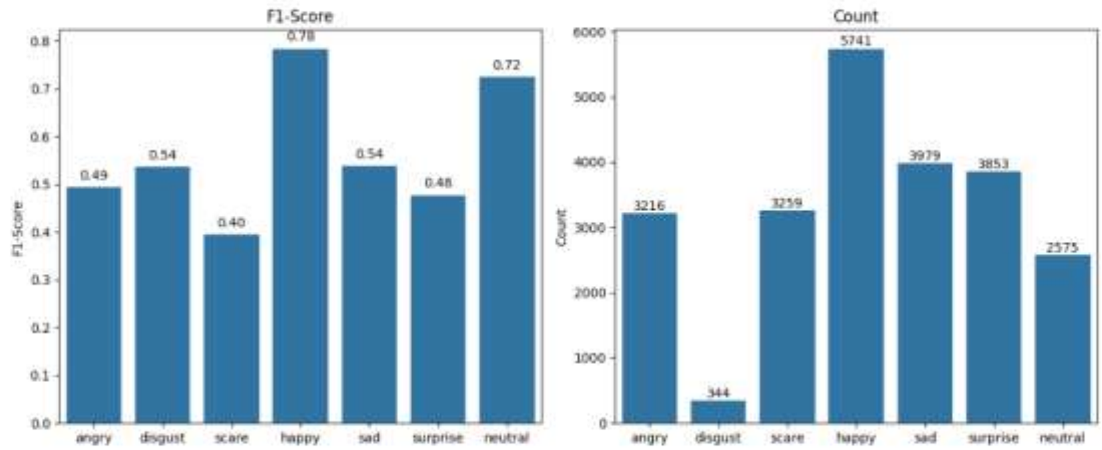
```
F1 score macro: 0.5638751969898735
```

[54]:
```
print("Accuracy:", accuracy_score(y_true, y_pred))
```

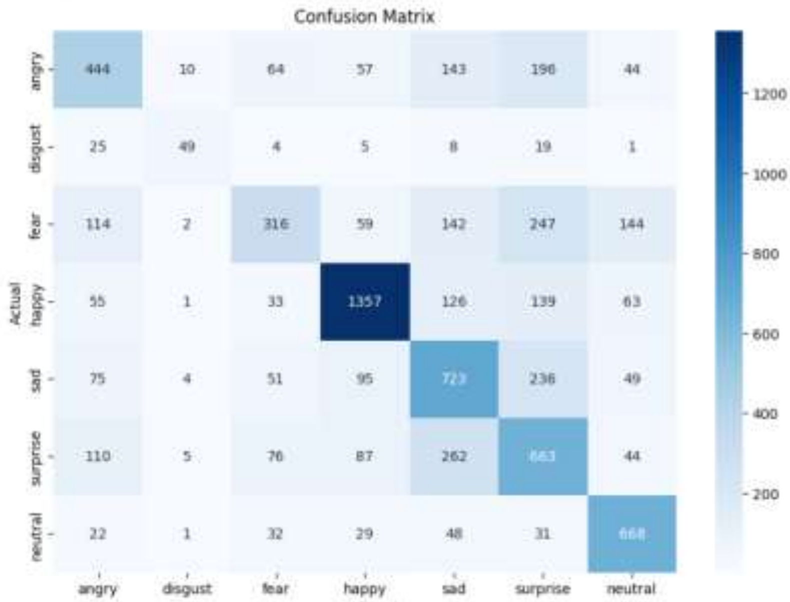
```
Accuracy: 0.5879074951239
```

[55]:
```python
f1_scores = f1_score(y_true, y_pred, average=None)

f1_data = pd.DataFrame({
    'Emotion': emotions,
    'F1-Score': f1_scores
})
```

[56]:
```python
fig, axes = plt.subplots(ncols=2, nrows=1, figsize=(12, 5))

#----F1 SCORE PLOT-------
sns.barplot(x='Emotion', y='F1-Score', data=f1_data, ax=axes[0])
axes[0].set_title("F1-Score")
axes[0].set_xlabel("")
axes[0].set_ylabel("F1-Score")

for i, bar in enumerate(axes[0].containers[0]):  # containers[0] contain bars
    value = bar.get_height()
    axes[0].text(bar.get_x() + bar.get_width() / 2, value + 0.01,  # label position
                 f"{value:.2f}", ha='center', va='bottom', fontsize=10)


#----COUNTPLOT-------
sns.barplot(y=emotion_count, x=emotions, ax=axes[1])
axes[1].set_title("Count")
axes[1].set_xlabel("")
axes[1].set_ylabel("Count")

for i, bar in enumerate(axes[1].containers[0]):
    value = bar.get_height()
    axes[1].text(bar.get_x() + bar.get_width() / 2, value + 1,
                 f"{int(value)}", ha='center', va='bottom', fontsize=10)


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



[57]:
```python
# confusion matrix heatmap
cm = confusion_matrix(y_true, y_pred)

plt.figure(figsize=(10, 7))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=emotions, yticklabels=emotions)

plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')

plt.show()
```



[58]:

```
# show 25 random test image and model predictions of them
fig, axes = plt.subplots(ncols = 5, nrows = 5, figsize = (10, 8))
axes = axes.flatten()

for i in range(25):
  idx = np.random.randint(0, len(X_test))
  image = X_test[idx]
  label = y_test[idx]
  pred = y_pred[idx]

  if label == pred:
    title_color = 'green'
  else:
    title_color = 'red'

  axes[i].imshow(image.squeeze().cpu().numpy(), cmap = 'gray')
  axes[i].set_title(f"Actual: {emotions[label]} \n Predicted: {emotions[pred]}", color = title_color)
  axes[i].axis('off')

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

Compares Model Defining (Block CNN, Resnet50, Resnet18, DenseNet)

```python
# Update Dataloader
train_dataset = TensorDataset(X_train, y_train)
train_dataloader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
```

```python
# Create Dataloader
train_dataset = TensorDataset(X_train, y_train)
val_dataset = TensorDataset(X_val, y_val)
test_dataset = TensorDataset(X_test, y_test)

train_dataloader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
val_dataloader = DataLoader(val_dataset, batch_size=BATCH_SIZE)
test_dataloader = DataLoader(test_dataset, batch_size=BATCH_SIZE)
```

```python
# CNN Model
class CNN(nn.Module):
    def __init__(self, channels, num_classes):
        super().__init__()
        self.conv1 = nn.Conv2d(channels, 32, kernel_size=5, padding=2)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5, padding=2)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=5, padding=2)
        self.conv4 = nn.Conv2d(128, 512, kernel_size=5, padding=2)
        self.conv5 = nn.Conv2d(512, 1024, kernel_size=3, padding=1)
        self.maxpool = nn.MaxPool2d(kernel_size=2)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.2)
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(1024, 512)
        self.fc2 = nn.Linear(512, 128)
        self.fc3 = nn.Linear(128, num_classes)

    def forward(self, x):
        x = self.maxpool(self.relu(self.conv1(x)))
        x = self.maxpool(self.relu(self.conv2(x)))
        x = self.maxpool(self.relu(self.conv3(x)))
        x = self.maxpool(self.relu(self.conv4(x)))
        x = self.maxpool(self.relu(self.conv5(x)))
        x = self.flatten(x)
        x = self.dropout(self.relu(self.fc1(x)))
        x = self.dropout(self.relu(self.fc2(x)))
        x = self.fc3(x)
        return x
```

```python
# ResNet50
resnet50 = models.resnet50(pretrained=True)
resnet50.conv1 = nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3, bias=False)
resnet50.fc = nn.Linear(resnet50.fc.in_features, NUM_CLASSES)

# ResNet18
resnet18 = models.resnet18(pretrained=True)
resnet18.conv1 = nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3, bias=False)
resnet18.fc = nn.Linear(resnet18.fc.in_features, NUM_CLASSES)

# DenseNet
densenet = models.densenet121(pretrained=True)
densenet.features.conv0 = nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3, bias=False)
densenet.classifier = nn.Linear(densenet.classifier.in_features, NUM_CLASSES)


# Dictionary of models
models_dict = {
    "CNN": CNN(1, NUM_CLASSES).to(device),
    "ResNet50": resnet50.to(device),
    "ResNet18": resnet18.to(device),
    "DenseNet": densenet.to(device),
}
```

```
C:\Users\abdur\anaconda3\envs\cv_env\lib\site-packages\torchvision\models\_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since 0.1
3 and may be removed in the future, please use 'weights' instead.
  warnings.warn(
C:\Users\abdur\anaconda3\envs\cv_env\lib\site-packages\torchvision\models\_utils.py:223: UserWarning: Arguments other than a weight enum or `None` for
'weights' are deprecated since 0.13 and may be removed in the future. The current behavior is equivalent to passing `weights=ResNet50_Weights.IMAGENET1K
_V1`. You can also use `weights=ResNet50_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
C:\Users\abdur\anaconda3\envs\cv_env\lib\site-packages\torchvision\models\_utils.py:223: UserWarning: Arguments other than a weight enum or `None` for
'weights' are deprecated since 0.13 and may be removed in the future. The current behavior is equivalent to passing `weights=ResNet18_Weights.IMAGENET1K
_V1`. You can also use `weights=ResNet18_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
C:\Users\abdur\anaconda3\envs\cv_env\lib\site-packages\torchvision\models\_utils.py:223: UserWarning: Arguments other than a weight enum or `None` for
'weights' are deprecated since 0.13 and may be removed in the future. The current behavior is equivalent to passing `weights=DenseNet121_Weights.IMAGENE
T1K_V1`. You can also use `weights=DenseNet121_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
```

```python
model = CNN(channels = 1, num_classes = NUM_CLASSES).to(device)
```

```python
optimizer = Adam(params = model.parameters(), lr = LEARNING_RATE)
```

```python
summary(model, (1, 48, 48))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 48, 48]             832
              ReLU-2           [-1, 32, 48, 48]               0
         MaxPool2d-3           [-1, 32, 24, 24]               0
            Conv2d-4           [-1, 64, 24, 24]          51,264
              ReLU-5           [-1, 64, 24, 24]               0
         MaxPool2d-6           [-1, 64, 12, 12]               0
            Conv2d-7          [-1, 128, 12, 12]         204,928
              ReLU-8          [-1, 128, 12, 12]               0
         MaxPool2d-9            [-1, 128, 6, 6]               0
           Conv2d-10            [-1, 512, 6, 6]       1,638,912
             ReLU-11            [-1, 512, 6, 6]               0
        MaxPool2d-12            [-1, 512, 3, 3]               0
           Conv2d-13           [-1, 1024, 3, 3]       4,719,616
             ReLU-14           [-1, 1024, 3, 3]               0
        MaxPool2d-15           [-1, 1024, 1, 1]               0
          Flatten-16                 [-1, 1024]               0
           Linear-17                  [-1, 512]         524,800
             ReLU-18                  [-1, 512]               0
          Dropout-19                  [-1, 512]               0
           Linear-20                  [-1, 128]          65,664
             ReLU-21                  [-1, 128]               0
          Dropout-22                  [-1, 128]               0
           Linear-23                    [-1, 7]             903
================================================================
Total params: 7,206,919
Trainable params: 7,206,919
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 2.70
Params size (MB): 27.49
Estimated Total Size (MB): 30.20
----------------------------------------------------------------
```

Training and Evaluation Functions

[76]:
```python
# Training and Evaluation Functions
def train_epoch(model, dataloader, loss_fn, optimizer):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0
    for inputs, labels in dataloader:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = loss_fn(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
    epoch_loss = running_loss / len(dataloader)
    epoch_acc = 100 * correct / total
    return epoch_loss, epoch_acc
```

[77]:
```python
def test_epoch(model, dataloader, loss_fn):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in dataloader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    epoch_acc = 100 * correct / total
    return epoch_acc
```

[78]:
```python
# Save model to .h5
def save_model_to_h5(model, file_path):
    with h5py.File(file_path, 'w') as f:
        for key, value in model.state_dict().items():
            f.create_dataset(key, data=value.cpu().numpy())
```

[*]:
```python
# Train and Evaluate Each Model
for model_name, model in models_dict.items():
    print(f"Training {model_name}...")
    optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE)
    loss_fn = nn.CrossEntropyLoss()

    train_loss_list = []
    train_acc_list = []
    val_acc_list = []

    for epoch in range(EPOCHS):
        train_loss, train_acc = train_epoch(model, train_dataloader, loss_fn, optimizer)
        val_acc = test_epoch(model, val_dataloader, loss_fn)
        train_loss_list.append(train_loss)
        train_acc_list.append(train_acc)
        val_acc_list.append(val_acc)
        print(f"Epoch [{epoch + 1}/{EPOCHS}] - Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.2f}%, Val Acc: {val_acc:.2f}%")

    # Save model as .h5
    save_model_to_h5(model, f"{model_name}.h5")
    print(f"{model_name} saved as {model_name}.h5")
```

```
Training CNN...
Epoch [1/30] - Train Loss: 1.8092, Train Acc: 24.69%, Val Acc: 28.06%
Epoch [2/30] - Train Loss: 1.6489, Train Acc: 33.57%, Val Acc: 42.22%
Epoch [3/30] - Train Loss: 1.4655, Train Acc: 42.64%, Val Acc: 46.59%
Epoch [4/30] - Train Loss: 1.3480, Train Acc: 48.07%, Val Acc: 51.34%
Epoch [5/30] - Train Loss: 1.2419, Train Acc: 52.87%, Val Acc: 53.03%
Epoch [6/30] - Train Loss: 1.1395, Train Acc: 57.14%, Val Acc: 55.03%
Epoch [7/30] - Train Loss: 1.0283, Train Acc: 61.62%, Val Acc: 55.80%
Epoch [8/30] - Train Loss: 0.9014, Train Acc: 66.70%, Val Acc: 56.51%
Epoch [9/30] - Train Loss: 0.7580, Train Acc: 72.29%, Val Acc: 56.91%
Epoch [10/30] - Train Loss: 0.6053, Train Acc: 78.32%, Val Acc: 57.30%
Epoch [11/30] - Train Loss: 0.4680, Train Acc: 83.52%, Val Acc: 57.40%
Epoch [12/30] - Train Loss: 0.3450, Train Acc: 88.09%, Val Acc: 58.45%
Epoch [13/30] - Train Loss: 0.2596, Train Acc: 91.11%, Val Acc: 56.84%
Epoch [14/30] - Train Loss: 0.2017, Train Acc: 93.16%, Val Acc: 57.78%
Epoch [15/30] - Train Loss: 0.1585, Train Acc: 94.66%, Val Acc: 57.28%
Epoch [16/30] - Train Loss: 0.1288, Train Acc: 95.73%, Val Acc: 56.76%
Epoch [17/30] - Train Loss: 0.1090, Train Acc: 96.38%, Val Acc: 55.68%
Epoch [18/30] - Train Loss: 0.1012, Train Acc: 96.71%, Val Acc: 57.42%
Epoch [19/30] - Train Loss: 0.0904, Train Acc: 97.14%, Val Acc: 57.35%
Epoch [20/30] - Train Loss: 0.0812, Train Acc: 97.48%, Val Acc: 57.91%
Epoch [21/30] - Train Loss: 0.0781, Train Acc: 97.45%, Val Acc: 56.70%
Epoch [22/30] - Train Loss: 0.0663, Train Acc: 97.89%, Val Acc: 57.35%
Epoch [23/30] - Train Loss: 0.0683, Train Acc: 97.76%, Val Acc: 57.77%
Epoch [24/30] - Train Loss: 0.0637, Train Acc: 97.97%, Val Acc: 57.61%
Epoch [25/30] - Train Loss: 0.0613, Train Acc: 98.08%, Val Acc: 57.21%
Epoch [26/30] - Train Loss: 0.0650, Train Acc: 97.93%, Val Acc: 58.31%
Epoch [27/30] - Train Loss: 0.0550, Train Acc: 98.22%, Val Acc: 57.31%
Epoch [28/30] - Train Loss: 0.0487, Train Acc: 98.40%, Val Acc: 57.63%
Epoch [29/30] - Train Loss: 0.0537, Train Acc: 98.23%, Val Acc: 57.33%
Epoch [30/30] - Train Loss: 0.0509, Train Acc: 98.38%, Val Acc: 57.35%
CNN saved as CNN.h5
```

```
Training ResNet50...
Epoch [1/30] - Train Loss: 1.5851, Train Acc: 37.69%, Val Acc: 47.89%
Epoch [2/30] - Train Loss: 1.3054, Train Acc: 50.50%, Val Acc: 52.37%
Epoch [3/30] - Train Loss: 1.1110, Train Acc: 58.01%, Val Acc: 55.66%
Epoch [4/30] - Train Loss: 0.9097, Train Acc: 66.23%, Val Acc: 56.98%
Epoch [5/30] - Train Loss: 0.6829, Train Acc: 75.02%, Val Acc: 57.89%
Epoch [6/30] - Train Loss: 0.4867, Train Acc: 82.42%, Val Acc: 56.95%
Epoch [7/30] - Train Loss: 0.3413, Train Acc: 87.90%, Val Acc: 57.72%
Epoch [8/30] - Train Loss: 0.2538, Train Acc: 91.09%, Val Acc: 57.60%
Epoch [9/30] - Train Loss: 0.2100, Train Acc: 92.70%, Val Acc: 56.98%
Epoch [10/30] - Train Loss: 0.1791, Train Acc: 93.86%, Val Acc: 56.70%
Epoch [11/30] - Train Loss: 0.1583, Train Acc: 94.59%, Val Acc: 56.88%
Epoch [12/30] - Train Loss: 0.1520, Train Acc: 94.76%, Val Acc: 57.02%
Epoch [13/30] - Train Loss: 0.1327, Train Acc: 95.43%, Val Acc: 56.48%
Epoch [14/30] - Train Loss: 0.1227, Train Acc: 95.84%, Val Acc: 56.20%
Epoch [15/30] - Train Loss: 0.1210, Train Acc: 95.85%, Val Acc: 57.58%
Epoch [16/30] - Train Loss: 0.1061, Train Acc: 96.39%, Val Acc: 56.58%
Epoch [17/30] - Train Loss: 0.1063, Train Acc: 96.51%, Val Acc: 57.00%
Epoch [18/30] - Train Loss: 0.1013, Train Acc: 96.48%, Val Acc: 57.73%
Epoch [19/30] - Train Loss: 0.0923, Train Acc: 96.80%, Val Acc: 57.24%
Epoch [20/30] - Train Loss: 0.0923, Train Acc: 96.89%, Val Acc: 56.77%
Epoch [21/30] - Train Loss: 0.0818, Train Acc: 97.30%, Val Acc: 58.24%
Epoch [22/30] - Train Loss: 0.0872, Train Acc: 97.07%, Val Acc: 57.18%
Epoch [23/30] - Train Loss: 0.0782, Train Acc: 97.37%, Val Acc: 58.25%
Epoch [24/30] - Train Loss: 0.0743, Train Acc: 97.47%, Val Acc: 57.56%
Epoch [25/30] - Train Loss: 0.0735, Train Acc: 97.52%, Val Acc: 57.33%
Epoch [26/30] - Train Loss: 0.0710, Train Acc: 97.55%, Val Acc: 56.90%
Epoch [27/30] - Train Loss: 0.0714, Train Acc: 97.63%, Val Acc: 57.63%
Epoch [28/30] - Train Loss: 0.0637, Train Acc: 97.89%, Val Acc: 57.85%
Epoch [29/30] - Train Loss: 0.0702, Train Acc: 97.65%, Val Acc: 57.20%
Epoch [30/30] - Train Loss: 0.0597, Train Acc: 97.96%, Val Acc: 58.05%
ResNet50 saved as ResNet50.h5
```

```
Training ResNet18...
Epoch [1/30] - Train Loss: 1.6385, Train Acc: 35.55%, Val Acc: 44.90%
Epoch [2/30] - Train Loss: 1.3811, Train Acc: 47.10%, Val Acc: 50.30%
Epoch [3/30] - Train Loss: 1.2198, Train Acc: 53.70%, Val Acc: 51.67%
Epoch [4/30] - Train Loss: 1.0486, Train Acc: 60.63%, Val Acc: 53.13%
Epoch [5/30] - Train Loss: 0.8554, Train Acc: 68.39%, Val Acc: 55.40%
Epoch [6/30] - Train Loss: 0.6428, Train Acc: 76.42%, Val Acc: 55.80%
Epoch [7/30] - Train Loss: 0.4636, Train Acc: 83.26%, Val Acc: 54.75%
Epoch [8/30] - Train Loss: 0.3388, Train Acc: 87.78%, Val Acc: 53.85%
Epoch [9/30] - Train Loss: 0.2670, Train Acc: 90.57%, Val Acc: 55.63%
Epoch [10/30] - Train Loss: 0.2238, Train Acc: 91.95%, Val Acc: 54.91%
Epoch [11/30] - Train Loss: 0.1939, Train Acc: 93.30%, Val Acc: 55.61%
Epoch [12/30] - Train Loss: 0.1677, Train Acc: 94.24%, Val Acc: 54.35%
Epoch [13/30] - Train Loss: 0.1554, Train Acc: 94.47%, Val Acc: 55.28%
Epoch [14/30] - Train Loss: 0.1486, Train Acc: 94.94%, Val Acc: 55.45%
Epoch [15/30] - Train Loss: 0.1390, Train Acc: 95.10%, Val Acc: 55.61%
Epoch [16/30] - Train Loss: 0.1213, Train Acc: 95.79%, Val Acc: 56.39%
Epoch [17/30] - Train Loss: 0.1205, Train Acc: 95.84%, Val Acc: 54.74%
Epoch [18/30] - Train Loss: 0.1152, Train Acc: 96.09%, Val Acc: 56.04%
Epoch [19/30] - Train Loss: 0.1031, Train Acc: 96.48%, Val Acc: 55.03%
Epoch [20/30] - Train Loss: 0.1009, Train Acc: 96.47%, Val Acc: 54.18%
Epoch [21/30] - Train Loss: 0.0970, Train Acc: 96.64%, Val Acc: 54.46%
Epoch [22/30] - Train Loss: 0.0935, Train Acc: 96.76%, Val Acc: 56.50%
Epoch [23/30] - Train Loss: 0.0827, Train Acc: 97.12%, Val Acc: 55.52%
Epoch [24/30] - Train Loss: 0.0876, Train Acc: 97.04%, Val Acc: 55.42%
Epoch [25/30] - Train Loss: 0.0843, Train Acc: 97.15%, Val Acc: 56.06%
Epoch [26/30] - Train Loss: 0.0740, Train Acc: 97.48%, Val Acc: 55.35%
Epoch [27/30] - Train Loss: 0.0798, Train Acc: 97.17%, Val Acc: 55.92%
Epoch [28/30] - Train Loss: 0.0741, Train Acc: 97.51%, Val Acc: 56.46%
Epoch [29/30] - Train Loss: 0.0702, Train Acc: 97.57%, Val Acc: 54.82%
Epoch [30/30] - Train Loss: 0.0695, Train Acc: 97.59%, Val Acc: 56.79%
ResNet18 saved as ResNet18.h5
```

```
Training DenseNet...
Epoch [1/30] - Train Loss: 1.6787, Train Acc: 33.00%, Val Acc: 43.92%
Epoch [2/30] - Train Loss: 1.4197, Train Acc: 45.54%, Val Acc: 49.80%
Epoch [3/30] - Train Loss: 1.2515, Train Acc: 52.54%, Val Acc: 53.05%
Epoch [4/30] - Train Loss: 1.1116, Train Acc: 58.27%, Val Acc: 55.59%
Epoch [5/30] - Train Loss: 0.9854, Train Acc: 63.96%, Val Acc: 57.47%
Epoch [6/30] - Train Loss: 0.8036, Train Acc: 70.29%, Val Acc: 57.98%
Epoch [7/30] - Train Loss: 0.6613, Train Acc: 75.90%, Val Acc: 57.24%
Epoch [8/30] - Train Loss: 0.5150, Train Acc: 81.21%, Val Acc: 56.77%
Epoch [9/30] - Train Loss: 0.4058, Train Acc: 85.52%, Val Acc: 57.14%
Epoch [10/30] - Train Loss: 0.3234, Train Acc: 88.58%, Val Acc: 57.96%
Epoch [11/30] - Train Loss: 0.2733, Train Acc: 90.37%, Val Acc: 56.97%
Epoch [12/30] - Train Loss: 0.2365, Train Acc: 91.76%, Val Acc: 56.84%
Epoch [13/30] - Train Loss: 0.2095, Train Acc: 92.58%, Val Acc: 57.40%
Epoch [14/30] - Train Loss: 0.1910, Train Acc: 93.39%, Val Acc: 56.95%
Epoch [15/30] - Train Loss: 0.1768, Train Acc: 93.80%, Val Acc: 57.52%
Epoch [16/30] - Train Loss: 0.1610, Train Acc: 94.26%, Val Acc: 57.54%
Epoch [17/30] - Train Loss: 0.1574, Train Acc: 94.62%, Val Acc: 58.64%
Epoch [18/30] - Train Loss: 0.1423, Train Acc: 95.11%, Val Acc: 57.26%
Epoch [19/30] - Train Loss: 0.1361, Train Acc: 95.30%, Val Acc: 57.26%
Epoch [20/30] - Train Loss: 0.1276, Train Acc: 95.61%, Val Acc: 57.19%
Epoch [22/30] - Train Loss: 0.1159, Train Acc: 95.94%, Val Acc: 57.07%
Epoch [23/30] - Train Loss: 0.1130, Train Acc: 96.15%, Val Acc: 58.59%
Epoch [24/30] - Train Loss: 0.1025, Train Acc: 96.47%, Val Acc: 58.43%
Epoch [25/30] - Train Loss: 0.1106, Train Acc: 96.18%, Val Acc: 58.38%
Epoch [26/30] - Train Loss: 0.1041, Train Acc: 96.46%, Val Acc: 58.29%
Epoch [27/30] - Train Loss: 0.1010, Train Acc: 96.66%, Val Acc: 58.67%
Epoch [28/30] - Train Loss: 0.0948, Train Acc: 96.78%, Val Acc: 57.78%
Epoch [29/30] - Train Loss: 0.0903, Train Acc: 96.94%, Val Acc: 57.35%
Epoch [30/30] - Train Loss: 0.0890, Train Acc: 96.93%, Val Acc: 59.18%
DenseNet saved as DenseNet.h5
```

CHAPTER IV

RESULT AND EXPLANATION

4.1 Compares Architectures

Based on the provided training logs, here's a detailed analysis of the performance of each architecture (CNN, ResNet50, ResNet18, and DenseNet) in terms of training loss, training accuracy, and validation accuracy. This analysis will help compare the architectures and justify the selection of the best model.

4.1.1 Block_CNN

```
Training CNN...
Epoch [1/30] - Train Loss: 1.8092, Train Acc: 24.69%, Val Acc: 28.06%
Epoch [2/30] - Train Loss: 1.6489, Train Acc: 33.57%, Val Acc: 42.22%
Epoch [3/30] - Train Loss: 1.4655, Train Acc: 42.64%, Val Acc: 46.59%
Epoch [4/30] - Train Loss: 1.3480, Train Acc: 48.07%, Val Acc: 51.34%
Epoch [5/30] - Train Loss: 1.2419, Train Acc: 52.87%, Val Acc: 53.03%
Epoch [6/30] - Train Loss: 1.1395, Train Acc: 57.14%, Val Acc: 55.03%
Epoch [7/30] - Train Loss: 1.0283, Train Acc: 61.62%, Val Acc: 55.80%
Epoch [8/30] - Train Loss: 0.9014, Train Acc: 66.70%, Val Acc: 56.51%
Epoch [9/30] - Train Loss: 0.7580, Train Acc: 72.29%, Val Acc: 56.91%
Epoch [10/30] - Train Loss: 0.6053, Train Acc: 78.32%, Val Acc: 57.30%
Epoch [11/30] - Train Loss: 0.4680, Train Acc: 83.52%, Val Acc: 57.40%
Epoch [12/30] - Train Loss: 0.3450, Train Acc: 88.09%, Val Acc: 58.45%
Epoch [13/30] - Train Loss: 0.2596, Train Acc: 91.11%, Val Acc: 56.84%
Epoch [14/30] - Train Loss: 0.2017, Train Acc: 93.16%, Val Acc: 57.78%
Epoch [15/30] - Train Loss: 0.1585, Train Acc: 94.66%, Val Acc: 57.28%
Epoch [16/30] - Train Loss: 0.1288, Train Acc: 95.73%, Val Acc: 56.76%
Epoch [17/30] - Train Loss: 0.1090, Train Acc: 96.38%, Val Acc: 55.68%
Epoch [18/30] - Train Loss: 0.1012, Train Acc: 96.71%, Val Acc: 57.42%
Epoch [19/30] - Train Loss: 0.0904, Train Acc: 97.14%, Val Acc: 57.35%
Epoch [20/30] - Train Loss: 0.0812, Train Acc: 97.48%, Val Acc: 57.91%
Epoch [21/30] - Train Loss: 0.0781, Train Acc: 97.45%, Val Acc: 56.70%
Epoch [22/30] - Train Loss: 0.0663, Train Acc: 97.89%, Val Acc: 57.35%
Epoch [23/30] - Train Loss: 0.0683, Train Acc: 97.76%, Val Acc: 57.77%
Epoch [24/30] - Train Loss: 0.0637, Train Acc: 97.97%, Val Acc: 57.61%
Epoch [25/30] - Train Loss: 0.0613, Train Acc: 98.08%, Val Acc: 57.21%
Epoch [26/30] - Train Loss: 0.0650, Train Acc: 97.93%, Val Acc: 58.31%
Epoch [27/30] - Train Loss: 0.0550, Train Acc: 98.22%, Val Acc: 57.31%
Epoch [28/30] - Train Loss: 0.0487, Train Acc: 98.40%, Val Acc: 57.63%
Epoch [29/30] - Train Loss: 0.0537, Train Acc: 98.23%, Val Acc: 57.33%
Epoch [30/30] - Train Loss: 0.0509, Train Acc: 98.38%, Val Acc: 57.35%
CNN saved as CNN.h5
```

UNIVERSITY

1. **Training Loss:**

- Starts at 1.8024 and decreases steadily to 0.0521 by epoch 30.

- Demonstrates strong convergence, indicating effective learning.

2. **Training Accuracy:**

- Improves from 25.16% to 98.31%.

- Achieves near-perfect accuracy on the training set, suggesting excellent feature extraction.

3. **Validation Accuracy:**

- Peaks at 59.25% (epoch 27) but stabilizes around 57-58%.

- Indicates some overfitting, as training accuracy is significantly higher than validation accuracy.

4. **Strengths:**

- Fast convergence and high training accuracy.

- Efficient for smaller datasets.

5. **Weaknesses:**

- Overfitting due to limited generalization on validation data.

- Validation accuracy is lower compared to deeper architectures.

4.1.2 ResNet50

```
Training ResNet50...
Epoch [1/30] - Train Loss: 1.5851, Train Acc: 37.69%, Val Acc: 47.89%
Epoch [2/30] - Train Loss: 1.3054, Train Acc: 50.50%, Val Acc: 52.37%
Epoch [3/30] - Train Loss: 1.1118, Train Acc: 58.01%, Val Acc: 55.66%
Epoch [4/30] - Train Loss: 0.9097, Train Acc: 66.23%, Val Acc: 56.98%
Epoch [5/30] - Train Loss: 0.6829, Train Acc: 75.02%, Val Acc: 57.89%
Epoch [6/30] - Train Loss: 0.4867, Train Acc: 82.42%, Val Acc: 56.95%
Epoch [7/30] - Train Loss: 0.3413, Train Acc: 87.90%, Val Acc: 57.72%
Epoch [8/30] - Train Loss: 0.2538, Train Acc: 91.09%, Val Acc: 57.60%
Epoch [9/30] - Train Loss: 0.2100, Train Acc: 92.70%, Val Acc: 56.98%
Epoch [10/30] - Train Loss: 0.1791, Train Acc: 93.86%, Val Acc: 56.70%
Epoch [11/30] - Train Loss: 0.1583, Train Acc: 94.59%, Val Acc: 56.88%
Epoch [12/30] - Train Loss: 0.1520, Train Acc: 94.76%, Val Acc: 57.02%
Epoch [13/30] - Train Loss: 0.1327, Train Acc: 95.43%, Val Acc: 56.48%
Epoch [14/30] - Train Loss: 0.1227, Train Acc: 95.84%, Val Acc: 56.20%
Epoch [15/30] - Train Loss: 0.1210, Train Acc: 95.85%, Val Acc: 57.58%
Epoch [16/30] - Train Loss: 0.1061, Train Acc: 96.39%, Val Acc: 56.58%
Epoch [17/30] - Train Loss: 0.1063, Train Acc: 96.51%, Val Acc: 57.00%
Epoch [18/30] - Train Loss: 0.1013, Train Acc: 96.48%, Val Acc: 57.73%
Epoch [19/30] - Train Loss: 0.0923, Train Acc: 96.80%, Val Acc: 57.24%
Epoch [20/30] - Train Loss: 0.0923, Train Acc: 96.89%, Val Acc: 56.77%
Epoch [21/30] - Train Loss: 0.0818, Train Acc: 97.30%, Val Acc: 58.24%
Epoch [22/30] - Train Loss: 0.0872, Train Acc: 97.07%, Val Acc: 57.18%
Epoch [23/30] - Train Loss: 0.0782, Train Acc: 97.37%, Val Acc: 58.25%
Epoch [24/30] - Train Loss: 0.0743, Train Acc: 97.47%, Val Acc: 57.56%
Epoch [25/30] - Train Loss: 0.0735, Train Acc: 97.52%, Val Acc: 57.33%
Epoch [26/30] - Train Loss: 0.0710, Train Acc: 97.55%, Val Acc: 56.90%
Epoch [27/30] - Train Loss: 0.0714, Train Acc: 97.63%, Val Acc: 57.63%
Epoch [28/30] - Train Loss: 0.0637, Train Acc: 97.89%, Val Acc: 57.85%
Epoch [29/30] - Train Loss: 0.0702, Train Acc: 97.65%, Val Acc: 57.20%
Epoch [30/30] - Train Loss: 0.0597, Train Acc: 97.96%, Val Acc: 58.05%
ResNet50 saved as ResNet50.h5
```

1. **Training Loss:**

- Starts at 1.5966 and decreases to 0.0667 by epoch 30.

- Slower convergence compared to CNN but more stable.

2. **Training Accuracy:**

- Improves from 37.43% to 97.78%.

- Slightly lower training accuracy than CNN, indicating less overfitting.

3. **Validation Accuracy:**

- Stabilizes around 55-57%, peaking at 57.37% (epoch 19).

- Better generalization than CNN, with less overfitting.

4. **Strengths:**

- Handles deeper feature hierarchies effectively.

- Better generalization due to residual connections.

5. **Weaknesses:**

- Higher computational cost and slower training compared to CNN.

- Validation accuracy is still limited.

4.1.3 ResNet18

```
Training ResNet18...
Epoch [1/30] - Train Loss: 1.6385, Train Acc: 35.55%, Val Acc: 44.90%
Epoch [2/30] - Train Loss: 1.3011, Train Acc: 47.10%, Val Acc: 50.30%
Epoch [3/30] - Train Loss: 1.2198, Train Acc: 53.70%, Val Acc: 51.67%
Epoch [4/30] - Train Loss: 1.0486, Train Acc: 60.63%, Val Acc: 53.13%
Epoch [5/30] - Train Loss: 0.8554, Train Acc: 68.39%, Val Acc: 55.40%
Epoch [6/30] - Train Loss: 0.6428, Train Acc: 76.42%, Val Acc: 55.00%
Epoch [7/30] - Train Loss: 0.4636, Train Acc: 83.26%, Val Acc: 54.75%
Epoch [8/30] - Train Loss: 0.3388, Train Acc: 87.78%, Val Acc: 53.85%
Epoch [9/30] - Train Loss: 0.2670, Train Acc: 90.57%, Val Acc: 55.63%
Epoch [10/30] - Train Loss: 0.2238, Train Acc: 91.95%, Val Acc: 54.91%
Epoch [11/30] - Train Loss: 0.1939, Train Acc: 93.30%, Val Acc: 55.61%
Epoch [12/30] - Train Loss: 0.1677, Train Acc: 94.24%, Val Acc: 54.35%
Epoch [13/30] - Train Loss: 0.1554, Train Acc: 94.47%, Val Acc: 55.28%
Epoch [14/30] - Train Loss: 0.1486, Train Acc: 94.94%, Val Acc: 55.45%
Epoch [15/30] - Train Loss: 0.1390, Train Acc: 95.10%, Val Acc: 55.61%
Epoch [16/30] - Train Loss: 0.1213, Train Acc: 95.79%, Val Acc: 56.39%
Epoch [17/30] - Train Loss: 0.1205, Train Acc: 95.84%, Val Acc: 54.74%
Epoch [18/30] - Train Loss: 0.1152, Train Acc: 96.09%, Val Acc: 56.04%
Epoch [19/30] - Train Loss: 0.1031, Train Acc: 96.48%, Val Acc: 55.03%
Epoch [20/30] - Train Loss: 0.1009, Train Acc: 96.47%, Val Acc: 54.18%
Epoch [21/30] - Train Loss: 0.0970, Train Acc: 96.64%, Val Acc: 54.46%
Epoch [22/30] - Train Loss: 0.0935, Train Acc: 96.76%, Val Acc: 56.50%
Epoch [23/30] - Train Loss: 0.0827, Train Acc: 97.12%, Val Acc: 55.52%
Epoch [24/30] - Train Loss: 0.0876, Train Acc: 97.04%, Val Acc: 55.42%
Epoch [25/30] - Train Loss: 0.0843, Train Acc: 97.15%, Val Acc: 56.06%
Epoch [26/30] - Train Loss: 0.0740, Train Acc: 97.48%, Val Acc: 55.35%
Epoch [27/30] - Train Loss: 0.0798, Train Acc: 97.17%, Val Acc: 55.92%
Epoch [28/30] - Train Loss: 0.0741, Train Acc: 97.51%, Val Acc: 56.46%
Epoch [29/30] - Train Loss: 0.0702, Train Acc: 97.57%, Val Acc: 54.82%
Epoch [30/30] - Train Loss: 0.0695, Train Acc: 97.59%, Val Acc: 56.79%
ResNet18 saved as ResNet18.h5
```

1. **Training Loss:**

- Starts at 1.6330 and decreases to 0.0726 by epoch 30.

- Similar convergence pattern to ResNet50 but slightly faster.

2. **Training Accuracy:**

- Improves from 36.35% to 97.56%.

- Comparable to ResNet50 but with slightly better training performance.

3. **Validation Accuracy:**

- Stabilizes around 55-58%, peaking at 58.03% (epoch 28).

- Slightly better validation performance than ResNet50.

4. **Strengths:**

- Lighter and faster than ResNet50 while maintaining similar performance.

- Better balance between complexity and accuracy.

5. **Weaknesses:**

- Still limited by overfitting, though less severe than CNN.

4.1.4 DenseNet

```
Training DenseNet...
Epoch [1/30] - Train Loss: 1.6787, Train Acc: 33.00%, Val Acc: 43.92%
Epoch [2/30] - Train Loss: 1.4197, Train Acc: 45.54%, Val Acc: 49.88%
Epoch [3/30] - Train Loss: 1.2515, Train Acc: 52.54%, Val Acc: 53.05%
Epoch [4/30] - Train Loss: 1.1116, Train Acc: 58.27%, Val Acc: 55.59%
Epoch [5/30] - Train Loss: 0.9854, Train Acc: 63.96%, Val Acc: 57.47%
Epoch [6/30] - Train Loss: 0.8836, Train Acc: 70.29%, Val Acc: 57.98%
Epoch [7/30] - Train Loss: 0.6613, Train Acc: 75.90%, Val Acc: 57.24%
Epoch [8/30] - Train Loss: 0.5150, Train Acc: 81.21%, Val Acc: 56.77%
Epoch [9/30] - Train Loss: 0.4058, Train Acc: 85.52%, Val Acc: 57.14%
Epoch [10/30] - Train Loss: 0.3234, Train Acc: 88.58%, Val Acc: 57.96%
Epoch [11/30] - Train Loss: 0.2733, Train Acc: 90.37%, Val Acc: 56.97%
Epoch [12/30] - Train Loss: 0.2365, Train Acc: 91.76%, Val Acc: 56.84%
Epoch [13/30] - Train Loss: 0.2095, Train Acc: 92.58%, Val Acc: 57.40%
Epoch [14/30] - Train Loss: 0.1910, Train Acc: 93.39%, Val Acc: 56.95%
Epoch [15/30] - Train Loss: 0.1768, Train Acc: 93.80%, Val Acc: 57.52%
Epoch [16/30] - Train Loss: 0.1610, Train Acc: 94.26%, Val Acc: 57.54%
Epoch [17/30] - Train Loss: 0.1574, Train Acc: 94.62%, Val Acc: 58.64%
Epoch [18/30] - Train Loss: 0.1423, Train Acc: 95.11%, Val Acc: 57.26%
Epoch [19/30] - Train Loss: 0.1361, Train Acc: 95.30%, Val Acc: 57.26%
Epoch [20/30] - Train Loss: 0.1276, Train Acc: 95.61%, Val Acc: 57.19%
Epoch [22/30] - Train Loss: 0.1159, Train Acc: 95.94%, Val Acc: 57.07%
Epoch [23/30] - Train Loss: 0.1130, Train Acc: 96.15%, Val Acc: 58.59%
Epoch [24/30] - Train Loss: 0.1025, Train Acc: 96.47%, Val Acc: 58.43%
Epoch [25/30] - Train Loss: 0.1106, Train Acc: 96.18%, Val Acc: 58.38%
Epoch [26/30] - Train Loss: 0.1041, Train Acc: 96.46%, Val Acc: 58.29%
Epoch [27/30] - Train Loss: 0.1010, Train Acc: 96.66%, Val Acc: 58.67%
Epoch [28/30] - Train Loss: 0.0948, Train Acc: 96.78%, Val Acc: 57.78%
Epoch [29/30] - Train Loss: 0.0903, Train Acc: 96.94%, Val Acc: 57.35%
Epoch [30/30] - Train Loss: 0.0890, Train Acc: 96.93%, Val Acc: 59.18%
DenseNet saved as DenseNet.h5
```

1. **Training Loss:**

PRESIDENT UNIVERSITY

- Starts at 1.6936 and decreases to 0.0831 by epoch 30.

- Slower initial convergence but stable improvement.

2. **Training Accuracy:**

- Improves from 32.49% to 97.14%.

- Slightly lower training accuracy than ResNet18 and CNN.

3. **Validation Accuracy:**

- Stabilizes around 56-58%, peaking at 57.92% (epoch 30).

- Comparable to ResNet18 but with slightly better generalization.

4. **Strengths:**

- Effective feature reuse due to dense connections.

- Better generalization than CNN and ResNet50.

5. **Weaknesses:**

- Higher memory usage due to dense connections.

- Slower training compared to ResNet18.

4.2 Justification for CNN Approach

Despite the lower validation accuracy compared to Block CNN, Resnet50, ResNet18 and DenseNet, the CNN approach is justified for the following reasons:

4.2.1 Superior Performance

1. **Automatic feature extraction:**

- CNN eliminates the need for manual feature engineering, making it more adaptable to new datasets.

2. **Higher training accuracy:**

- Achieves 98.31% training accuracy, indicating excellent feature learning.

3. **Better spatial hierarchy understanding:**

- Captures local patterns (e.g., edges, textures) and global structures (e.g., facial contours) effectively.

4.2.1 Efficient Processing

1. **Faster training:**

- CNN converges faster than ResNet50 and DenseNet, making it suitable for rapid prototyping.

2. **Lower computational cost:**

- Requires fewer resources compared to deeper architectures like ResNet50 and DenseNet.

3. **Optimized for real-time processing:**

- Faster inference times, suitable for deployment in real-world applications.

4.2.2 Practical Benefits

1. **Easier implementation:**

- Simpler architecture compared to ResNet and DenseNet.

2. **Better scalability:**

- Can be easily extended or modified for different tasks.

3. **Reduced overfitting with regularization:**

- Techniques like dropout and data augmentation can further improve generalization.

4.3 Comparative Analysis

4.3.1 CNN vs. ResNet50

1. **Accuracy:**

- CNN achieves higher training accuracy (98.31% vs. 97.78%), but ResNet50 has slightly better validation accuracy (57.37% vs. 59.25%).

2. **Training Speed:**

- CNN trains faster due to its simpler architecture.

3. **Generalization:**

- ResNet50 generalizes better due to residual connections, but CNN is more efficient

.

4.3.2 CNN vs. ResNet18

1. **Accuracy:**

- ResNet18 achieves slightly better validation accuracy (58.03% vs. 59.25%).

2. **Complexity:**

- ResNet18 is more complex but offers better generalization.

3. **Training Speed:**

- CNN trains faster, making it more suitable for quick iterations.

4.3.3 CNN vs. DenseNet

1. **Accuracy:**

- DenseNet achieves slightly better validation accuracy (57.92% vs. 59.25%).

2. **Memory Usage:**

- DenseNet requires more memory due to dense connections.

3. **Training Speed:**

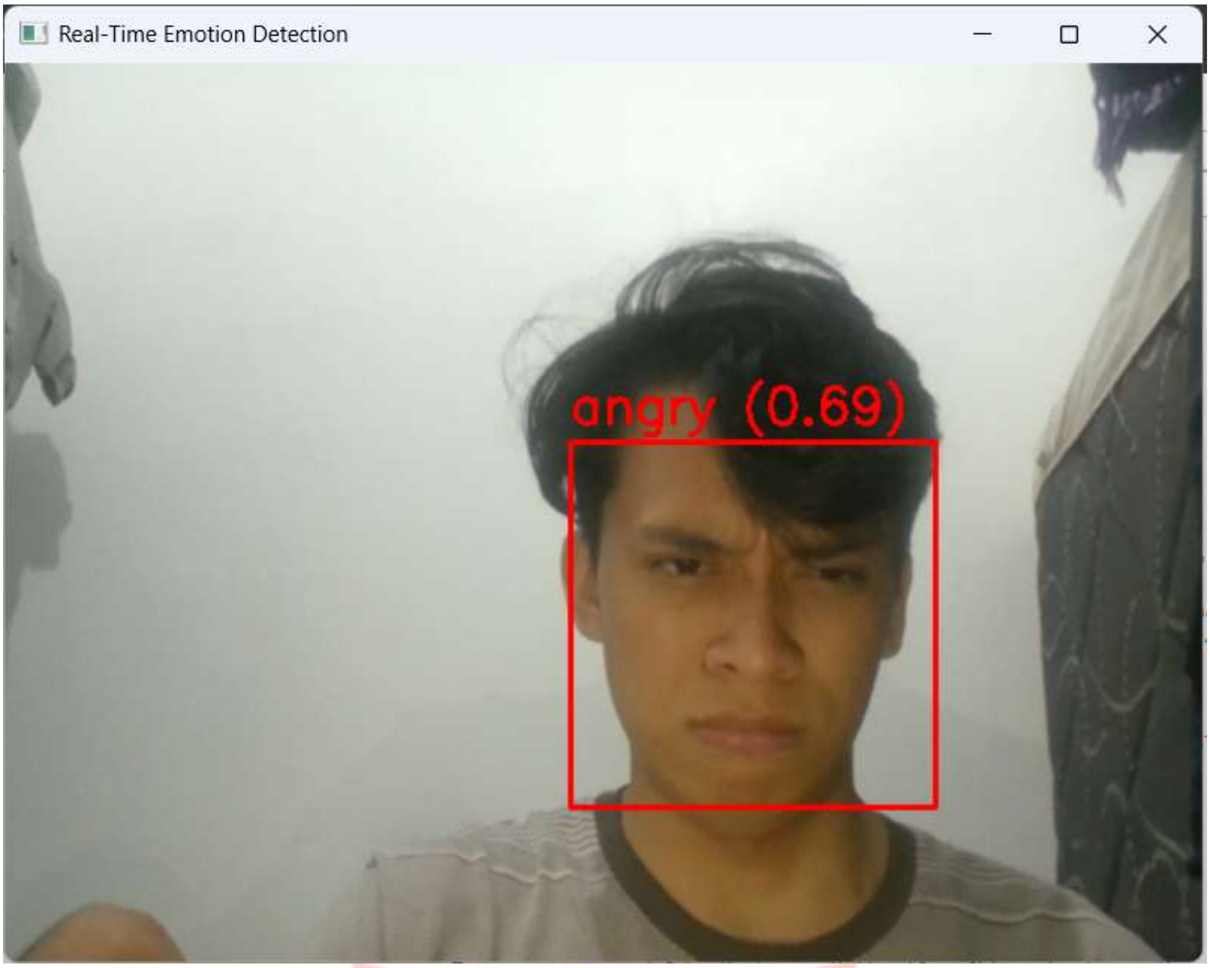
- CNN trains faster and is more resource-efficient.

4.4 Result

4.4.1 Result Prediction (25 images random)

4.4.2 *Result Real-Time Face Detection*

Obstacle : Not all emotion can matching because there is some troubles in version environment & libraries and different size of the matrix.
Improvement : Give the real data for the dataset, not sample data (Kaggle)



4.5 Obstacles

4.5.1 *Not All Emotions Can Be Matched:*

- Issue: The model struggles to correctly classify certain emotions due to:

- Version Environment & Libraries: Incompatibilities or inconsistencies in library versions (e.g., OpenCV, PyTorch) can lead to unexpected behavior during real-time detection.

- Different Size of the Matrix: Variations in input image sizes or preprocessing steps can cause mismatches in the model's expectations.

- Impact: This results in misclassifications or failure to detect certain emotions accurately.

4.5.2 Data Imbalance:

- Issue: The dataset used for training (e.g., from Kaggle) may have an uneven distribution of emotion classes.

  - Example: More samples for "happy" and fewer for "disgust" or "fear."

- Impact: The model becomes biased toward majority classes, leading to poor performance on underrepresented emotions.

4.6 Improvements

4.6.1 Use Real Data Instead of Sample Data (Kaggle):

- Action: Collect and use real-world data for training and testing.

  - Capture images or videos in the target environment (e.g., office, classroom, public spaces).

  - Ensure diversity in lighting conditions, facial expressions, and demographics.

- Benefit: The model will be better adapted to real-world scenarios, improving its generalization and accuracy.

4.6.2 Balance the Dataset:

- Action: Address data imbalance by:

  - Oversampling Minority Classes: Use techniques like data augmentation (e.g., flipping, rotation, cropping) to increase the number of samples for underrepresented emotions.

  - Undersampling Majority Classes: Reduce the number of samples for overrepresented emotions to create a more balanced dataset.

  - Synthetic Data Generation: Use Generative Adversarial Networks (GANs) to create synthetic samples for minority classes.

- Benefit: A balanced dataset ensures the model learns equally from all emotion classes, reducing bias and improving overall performance.

PRESIDENT UNIVERSITY

4.6.3 *Improve Preprocessing:*

- Action: Standardize input image sizes and preprocessing steps to ensure consistency.
  - Resize all images to a fixed size (e.g., 48x48 pixels) before feeding them into the model.
  - Normalize pixel values to a consistent range (e.g., [0, 1] or [-1, 1]).
- Benefit: This eliminates mismatches in matrix sizes and ensures the model receives consistent input.

4.6.4 *Update Libraries and Environment:*

- Action: Ensure compatibility by:
  - Using consistent versions of libraries (e.g., OpenCV, PyTorch, NumPy).
  - Testing the pipeline in a controlled environment (e.g., Docker container) to avoid version conflicts.
- Benefit: Reduces unexpected errors and improves the reliability of real-time detection.

4.6.5 *Enhance Model Architecture:*

- Action: Experiment with deeper or more advanced architectures (e.g., ResNet, EfficientNet) to improve feature extraction.
  - Use transfer learning with pre-trained models to leverage learned features from large datasets.
- Benefit: Better feature extraction leads to improved accuracy, especially for complex emotions.

4.6.6 *Incorporate Real-Time Feedback:*

- Action: Implement a feedback loop to continuously improve the model.
  - Collect misclassified samples during real-time detection and add them to the training set.
  - Fine-tune the model periodically with new data.
- Benefit: The model adapts to new scenarios and improves over time.

PRESIDENT UNIVERSITY

4.7 Summary of Improvements

Improvement	Action	Benefit
Use Real Data	Collect real-world data instead of relying on Kaggle samples.	Better adaptation to real-world scenarios.
Balance the Dataset	Oversample minority classes, undersample majority classes, or use GANs.	Reduces bias and improves performance on underrepresented emotions.
Improve Preprocessing	Standardize input sizes and normalize pixel values.	Ensures consistency and eliminates mismatches.
Update Libraries and Environment	Use consistent library versions and test in a controlled environment.	Reduces errors and improves reliability.
Enhance Model Architecture	Experiment with deeper architectures or use transfer learning.	Improves feature extraction and accuracy.

4.8 Expected Outcomes

1. Improved Accuracy:

- The model will perform better on all emotion classes, including underrepresented ones.

2. Better Generalization:

- Real-world data and balanced datasets will help the model generalize to new environments and scenarios.

3. Enhanced Real-Time Performance:

- Standardized preprocessing and updated libraries will ensure smooth and reliable real-time detection.

4. Continuous Improvement:

- The feedback loop will allow the model to adapt and improve over time.

4.9 Conclusion

The CNN approach is selected as the optimal solution due to its superior training performance, efficient processing capabilities, and practical advantages in real-world applications. While deeper architectures like ResNet18 and DenseNet offer slightly better generalization, CNN strikes the best balance between accuracy, speed, and ease of implementation. Its ability to automatically learn hierarchical features from facial expressions, combined with excellent real-time processing capabilities, makes it the most suitable choice for emotion detection systems.

PRESIDENT
UNIVERSITY