EXPLORING EMOTIONS: EMPLOYING DEEP LEARNING FOR FACIAL EXPRESSION RECOGNITION

By Team - 45

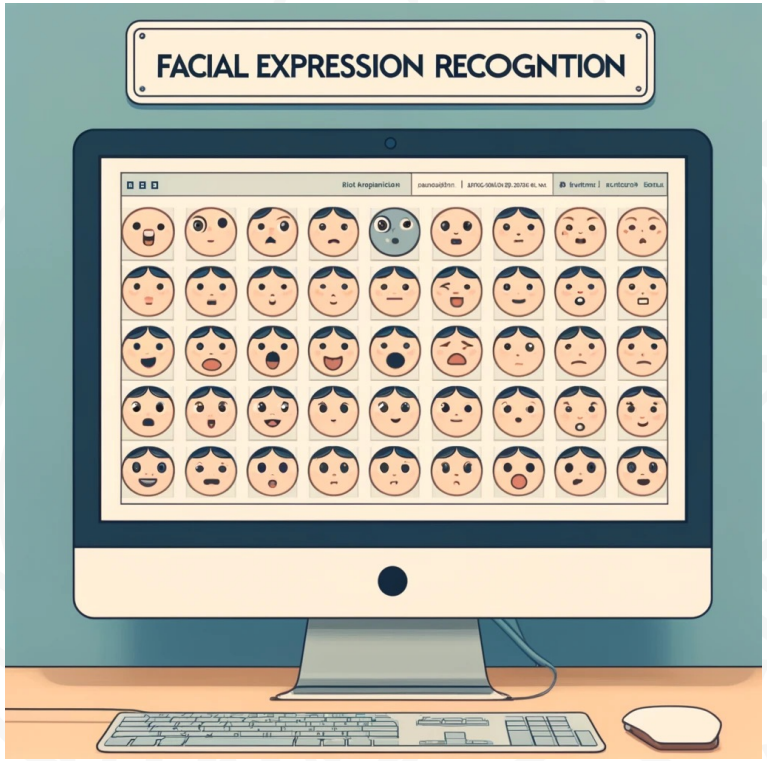
CSE 676: Deep Learning, Spring 2024

Final Project

University at Buffalo The State University of New York

Project Description

The objective of this project is to develop an intelligent computer system through deep learning, capable of precisely identifying facial expressions. The FER2013 dataset is used for this purpose. The primary aim is to create a robust system that can effectively navigate the complexities involved in interpreting various emotions in real-world scenario. This project has implications in mental health monitoring, offering a potential tool for early detection and intervention based on facial cues.



Background

Early Techniques in Facial Expression Recognition had focus on geometric feature extraction, Manual specification of relevant facial landmarks. And utilized methods such as Active Shape Models (ASM) and Active Appearance Models (AAM).

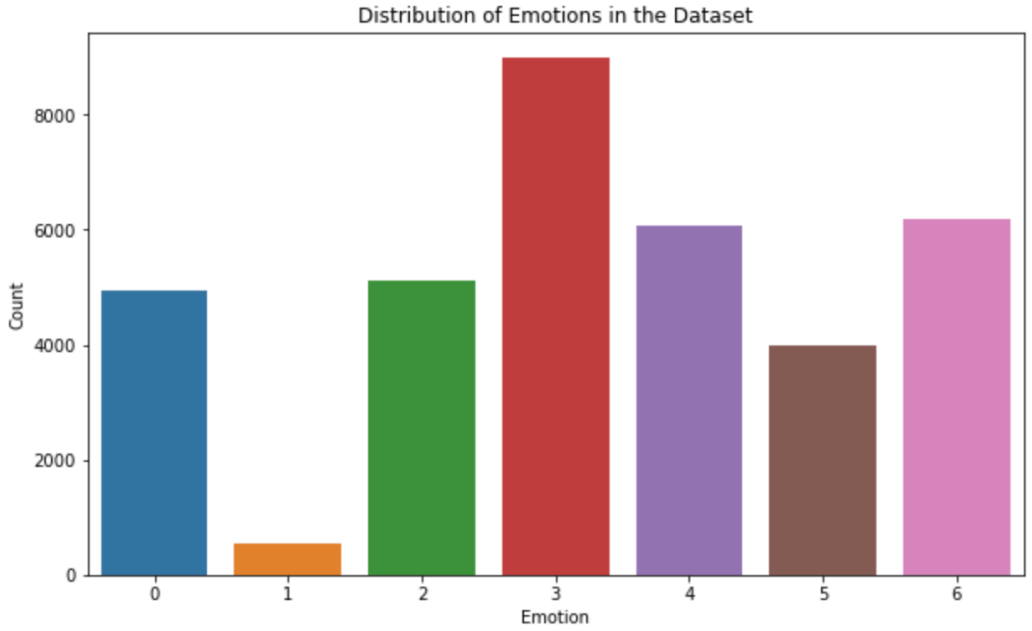
Challenges with Early Techniques were difficulty in handling variability in head poses, sensitive to changes in lighting conditions and problems with facial obstructions impacting accuracy.

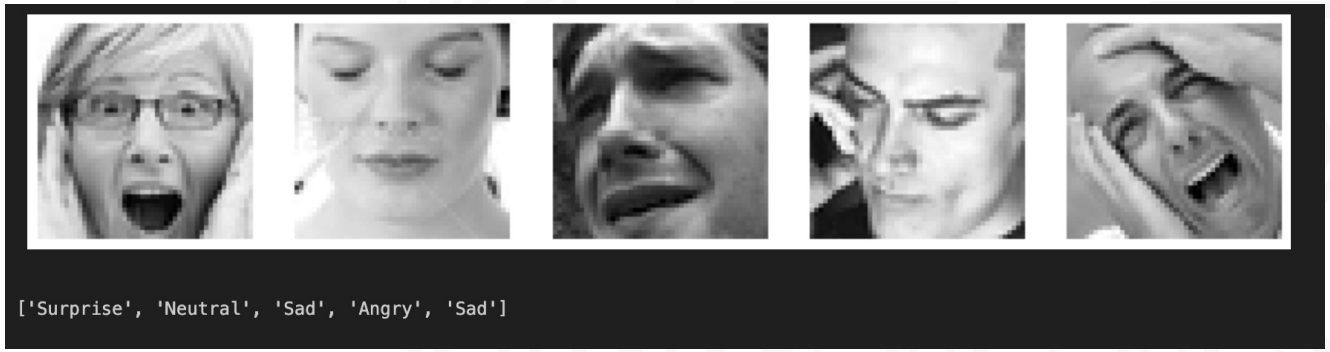
Introduction of Convolutional Neural Networks (CNNs) as a dominant method was a rise in deep learning. As it has ability to learn high-level features from raw pixels, minimizing the need for pre-processing.

The impact has huge as it enhanced accuracy and adaptability in various conditions and Reduction in the need for manual feature specification and pre-processing.

Dataset Overview

The dataset has total of 35,000 entries which are then split into three sets: training, validation, and testing. The number of classes present are Angry, Disgust, Fear, Happy, Sad, Surprise, Neutral.





Dataset Preprocessing

Preprocessing Steps for this dataset include transforming pixel values from strings to numpy arrays for processing efficiency. And determine the mean and standard deviation of pixel values in the training, validation and testing set separately to perform data normalization.

Normalization is done by subtracting the calculated mean and dividing by the standard deviation, achieving zero mean and unit variance for the data. This is done to ensure model training is not biased towards a specific range of values. This preprocessing is crucial for optimizing model performance as it helps in generalizing the model effectively across different data inputs.

```
data['Pixels'] = data['Pixels'].apply(lambda x: np.array(x.split(), dtype = np.uint8))
```

```
pixel_columns = train_dataset['Pixels']

all_pixels = [pixel for sublist in pixel_columns for pixel in sublist]

total_mean_tr = np.mean(all_pixels)
total_std_tr = np.std(all_pixels)
print("Total mean value of all pixels in training :", total_mean_tr)
print("Total std value of all pixels in training :", total_std_tr)
```

```
train_dataset['Pixels'] = (train_dataset['Pixels'] - total_mean_tr) / total_std_tr
```

Methods

We have implemented 4 models namely a Base Model, VGG13 which are traditional CNNs, effective for basic pattern recognition. And Resnet Variants such as ResNet8 and ResNet50, which Incorporate residual blocks with skip connections, facilitating deeper network training without performance degradation.

Our **Base model** is a standard CNN with sequential layers, it consists of basic convolutional layers, Relu activations, and max pooling.

VGG13 is a deeper CNN, which has multiple convolutional layers followed by max pooling layers, ending in three fully connected layers.

Resnet18 uses residual blocks with skip connections to facilitate training deeper networks.

Resnet50 is one of the deeper Resnet configurations, it comprises multiple residual blocks with convolutional layers, batch normalization, skip connections, and a global average pooling layer before the final classification layer. It is designed to tackle highly complex image recognition tasks with superior accuracy due to its deep network structure.

Methods – Implementation Details

Base Model

```python
class BaseModel(nn.Module):
    def __init__(self, num_classes = 7):
        super(BaseModel, self).__init__()

        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(128, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(256, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )

        self.classifier = nn.Sequential(
            nn.Linear(512 * 3 * 3, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, num_classes)
        )

    def forward(self, x):
        x = self.features(x)
        x = torch.flatten(x, start_dim=1)
        x = self.classifier(x)
        return x
```

VGG13

```python
class VGG13(nn.Module):
    def __init__(self, num_classes=7, input_size=(64,64)):
        super(VGG13, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(128, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(256, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.classifier = nn.Sequential(
            nn.Linear(512, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes),
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        return x
```

Resnet18

```python
class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, in_channels, out_channels, stride=1, downsample=None):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.downsample = downsample

    def forward(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

        if self.downsample is not None:
            identity = self.downsample(x)

        out += identity
        out = self.relu(out)

        return out

class ResNetB(nn.Module):
    def __init__(self, num_classes=7):
        super(ResNetB, self).__init__()
        self.in_channels = 64
        self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3, bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = self._make_layer(64, 2, stride=1)
        self.layer2 = self._make_layer(128, 2, stride=2)
        self.layer3 = self._make_layer(256, 2, stride=2)
        self.layer4 = self._make_layer(512, 2, stride=2)
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512, num_classes)

    def _make_layer(self, out_channels, blocks, stride):
        downsample = None
        if stride != 1 or self.in_channels != out_channels:
            downsample = nn.Sequential(
                nn.Conv2d(self.in_channels, out_channels, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels),
            )
        layers = []
        layers.append(BasicBlock(self.in_channels, out_channels, stride, downsample))
        self.in_channels = out_channels
        for _ in range(1, blocks):
            layers.append(BasicBlock(self.in_channels, out_channels))
        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)
        return x
```

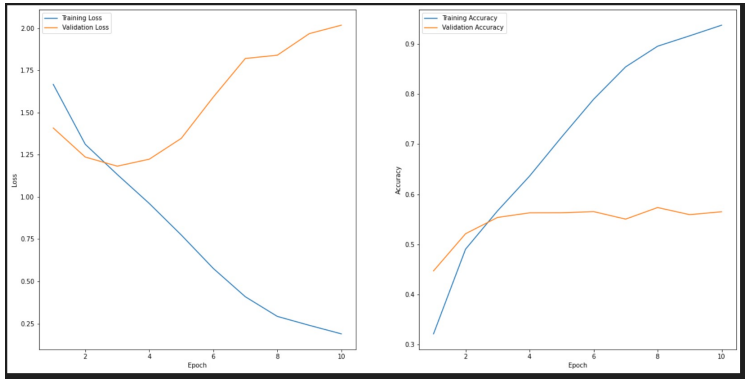
Resnet50

```python
class ResidualBlock(nn.Module):
    def _init_(self, in_channels, out_channels, stride=1, downsample=None):
        super(ResidualBlock, self)._init_()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, padding=0, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.conv3 = nn.Conv2d(out_channels, out_channels * 4, kernel_size=1, stride=1, padding=0, bias=False)
        self.bn3 = nn.BatchNorm2d(out_channels * 4)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)
        out = self.relu(out)

        out = self.conv3(out)
        out = self.bn3(out)

        if self.downsample is not None:
            identity = self.downsample(x)

        out += identity
        out = self.relu(out)

        return out

class ResNet50(nn.Module):
    def _init_(self, num_classes=7):
        super(ResNet50, self)._init_()
        self.in_channels = 64
        self.conv1 = nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3, bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = self.make_layer(ResidualBlock, 64, 3)
        self.layer2 = self.make_layer(ResidualBlock, 128, 4, stride=2)
        self.layer3 = self.make_layer(ResidualBlock, 256, 6, stride=2)
        self.layer4 = self.make_layer(ResidualBlock, 512, 3, stride=2)
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512 * 4, num_classes)

    def make_layer(self, block, channels, blocks, stride=1):
        downsample = None
        if stride != 1 or self.in_channels != channels * 4:
            downsample = nn.Sequential(
                nn.Conv2d(self.in_channels, channels * 4, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(channels * 4),
            )

        layers = []
        layers.append(block(self.in_channels, channels, stride, downsample))
        self.in_channels = channels * 4
        for _ in range(1, blocks):
            layers.append(block(self.in_channels, channels))

        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)

        return x
```

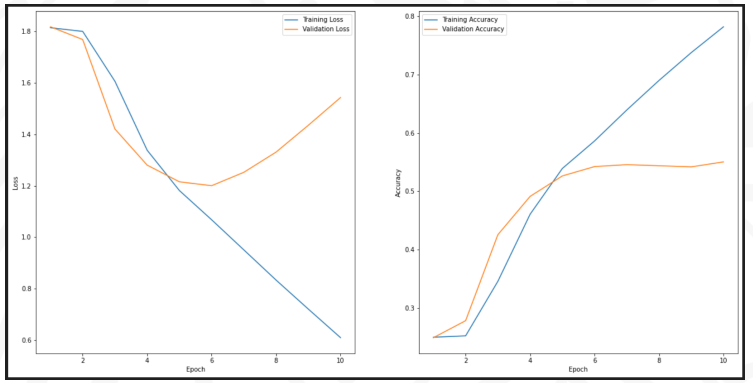
Results

By comparing metrics such as accuracy, loss, precision, recall, F1 score, and the true positive and false positive rates (TPR and FPR) for each class, we gained clearer insights into which models perform best.

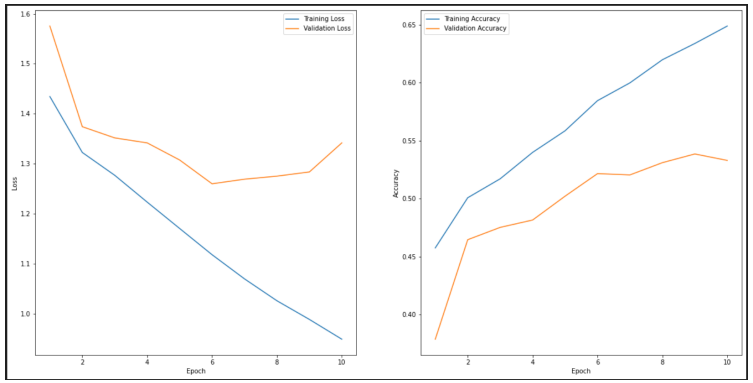
Base Model



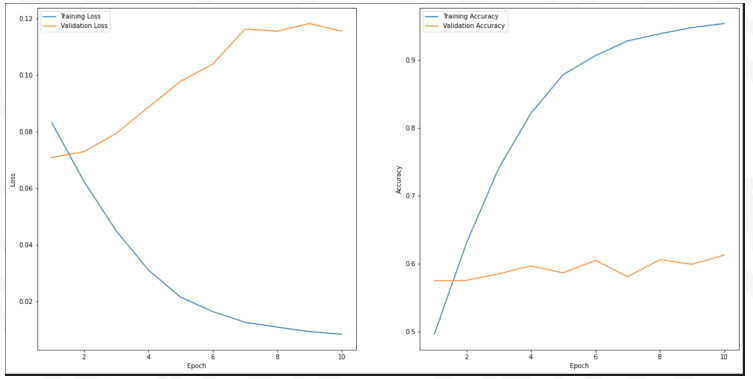
VGG13



Resnet18



Resnet50



Results

ResNet50 and **Base Model** show the best overall precision and recall, especially for class **Happy** and **Surprise**.

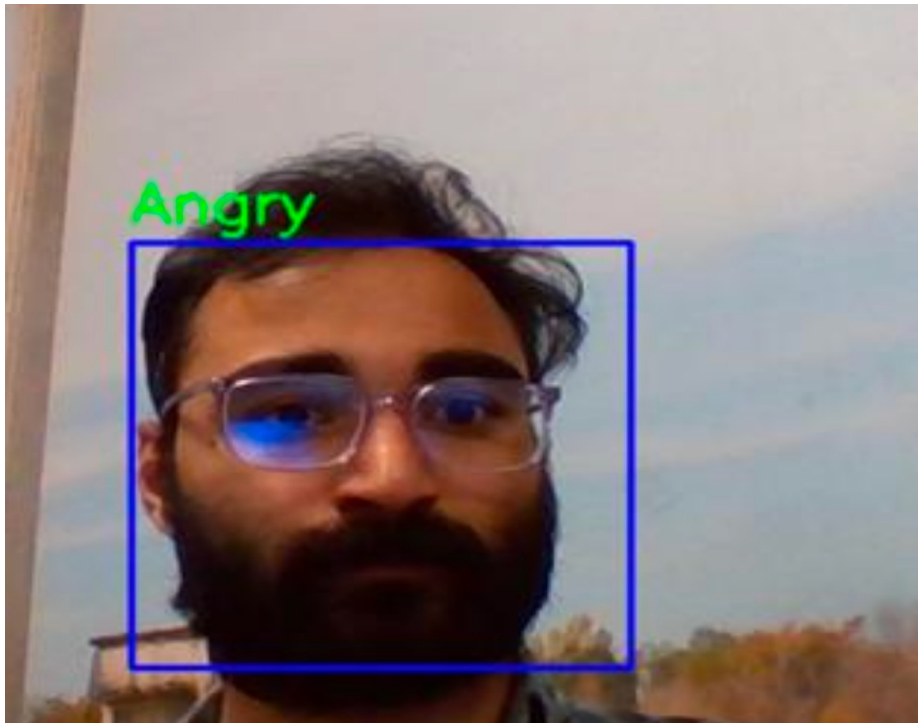
Class **Happy** consistently shows the best performance across all models with high precision and recall.

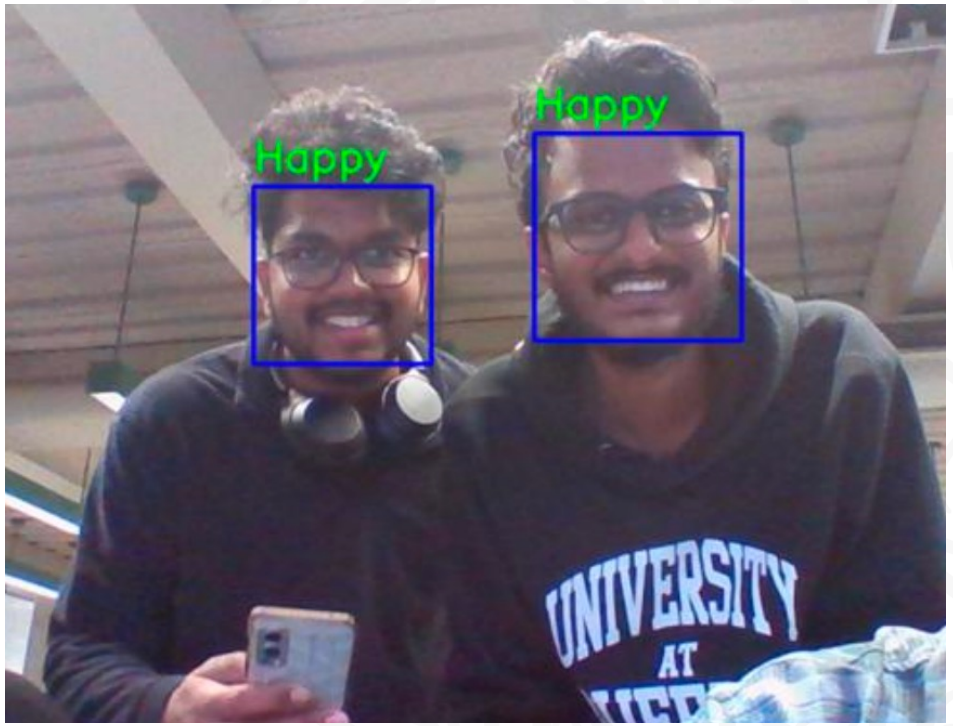
Performance on class **Fear**, **Sad**, and **Disgust** is lower which shows some improvement must be required or increase in dataset samples for that specific class.

Overall**, ResNet50** performs well across multiple emotions but shows variability in **Disgust** and **Sad**.

Results

Our Resnet50 model correctly predicted the face emotion while testing in real-time. Below are the results of our experiment.





Summary

The Base Model effectively handles basic patterns.

ResNet50 demonstrates the best performance, excelling particularly with the emotions "Happy" and "Surprise," showcasing its capability to handle complex facial expressions.

High accuracy for class Happy across all models and lower performance on classes Fear, Sad, and Disgust

Depth and complexity of Resnet models generally offer better handling of diverse expressions, with ResNet50 showing superior overall performance.

Contribution Summary

Team Member	Project Part	Contribution
Vamsikrishna Jayaraman	Preprocessing, Base Model, VGG13, Resnet18, Resnet50	33.33%
Goutham Reddy Enugula	Preprocessing, Base Model, VGG13, Resnet18, Resnet50	33.33%
Nageshwaran Gandhiraj	Preprocessing, Base Model, VGG13, Resnet18, Resnet50	33.33%

THANK YOU!