22AIE304 - DEEP LEARNING
CASE STUDY - PHASE 2 REPORT

Name	Roll number
Anuvind M P	AM.EN.U4AIE22010
R S Harish Kumar	AM.EN.U4AIE22042

AttendEase: An Intelligent Attendance System using Face Detection

I. Abstract

The proposed attendance system, *AttendEase*, leverages deep learning to automate attendance marking from group photos, addressing challenges posed by limited initial data. With only five to ten images per individual, the project employs advanced image augmentation techniques—such as rotation, scaling, flipping, and brightness adjustment—to synthetically expand the dataset. These enhanced datasets enable robust face detection and identity classification.

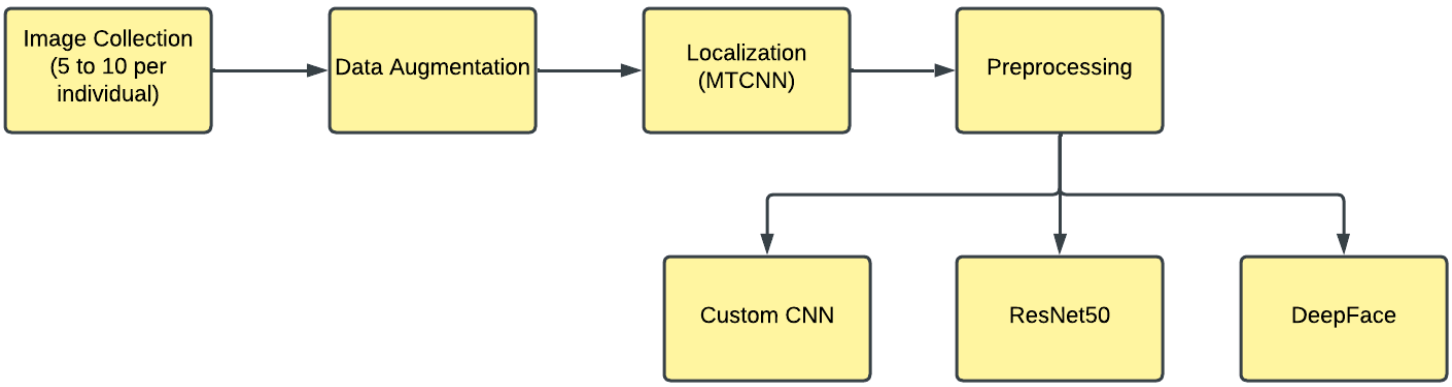
The project implements and evaluates three models: a custom Convolutional Neural Network (CNN), ResNet50, and DeepFace by Facebook. The analysis considers critical model parameters, including learning rates, optimizers, activation functions, and epoch counts, offering insights into their impact on accuracy and computational efficiency.

A key focus of the project is the comparative study of these models, analyzing their performance across various parameters such as learning rates, optimizers, epoch counts, and computational efficiency. This study aims to identify the optimal configurations and strengths of each model in terms of accuracy, generalization,

and processing overhead. By systematically examining these aspects, the project offers valuable insights into the trade-offs and capabilities of different architectures for real-world attendance tracking applications.

This innovative approach demonstrates scalability, accuracy, and adaptability, providing a practical solution for real-world applications requiring efficient attendance tracking through facial recognition. Deliverables include the custom dataset, trained models, and a comparative analysis of methodologies to validate the approach.

Pipeline :



II. Dataset Details

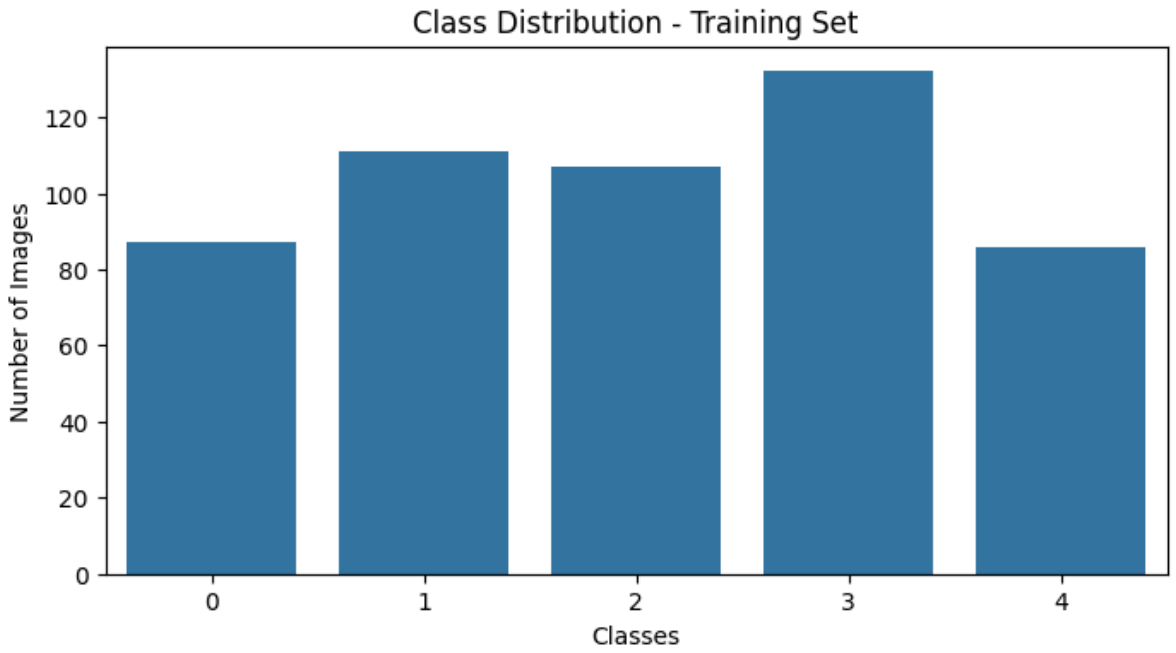
The dataset used in this project is custom-designed, tailored specifically for the development and evaluation of the attendance system. It was created by capturing images of five individuals, namely *Anuvind, Ashwin, Girish, Hari*, and *Harish*. The dataset is augmented using techniques such as rotation, scaling, flipping, and brightness adjustment to increase its diversity and improve model generalization.

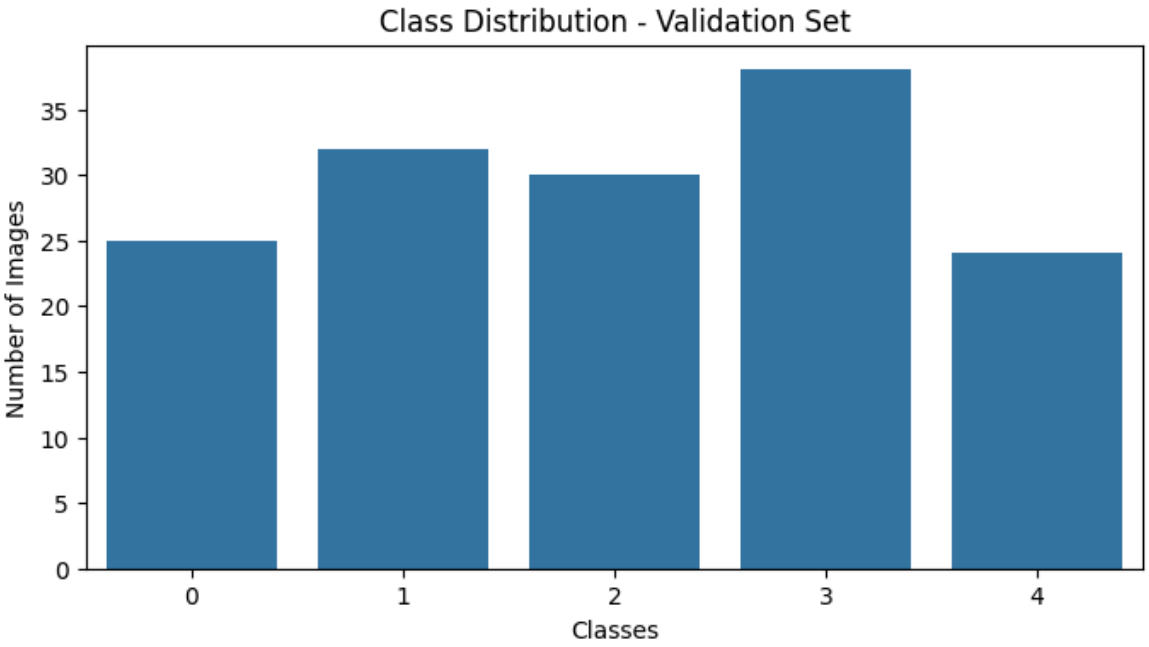
- Source: Custom-collected dataset.

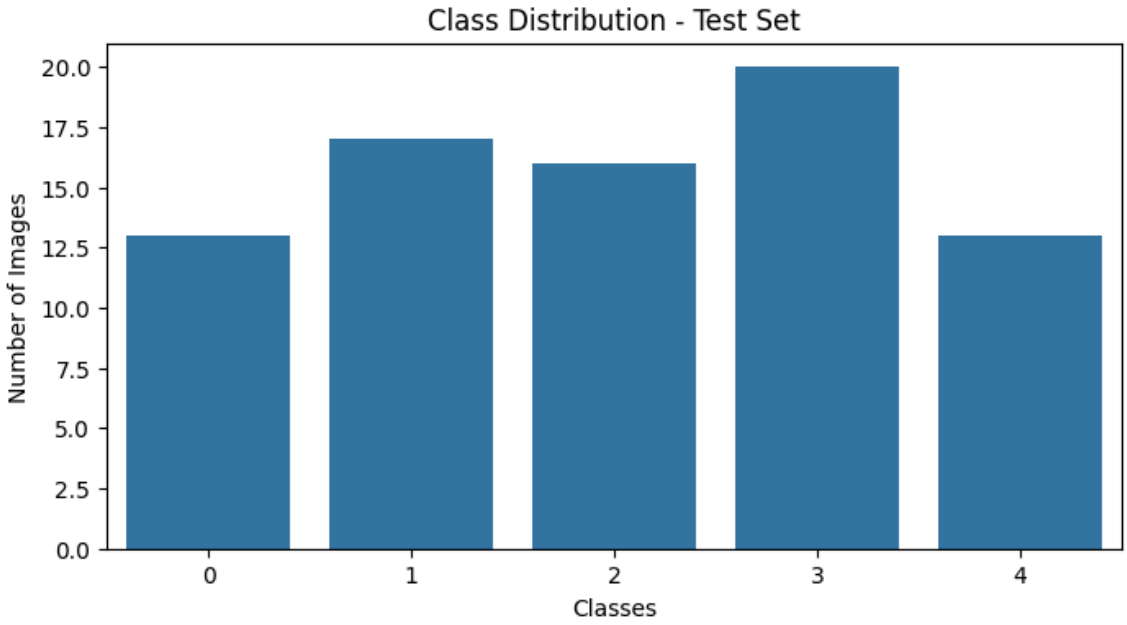
- Size: A total of 751 images, distributed across the training, validation, and testing stages.
- Description:
  Each image is labeled according to the identity of the individual, with a class-to-index mapping defined as:
  - 'anuvind': 0
  - 'ashwin': 1
  - 'girish': 2
  - 'hari': 3
  - 'harish': 4

These labeled and augmented images were further utilized to synthesize group photos. This synthetic dataset enabled the training and evaluation of models for accurate face detection and identity classification, even with minimal original data. The careful design and augmentation process ensure that the dataset mimics real-world variability in poses, lighting conditions, and perspectives, making it well-suited for robust attendance tracking.

Exploratory Data Analysis - Custom Dataset



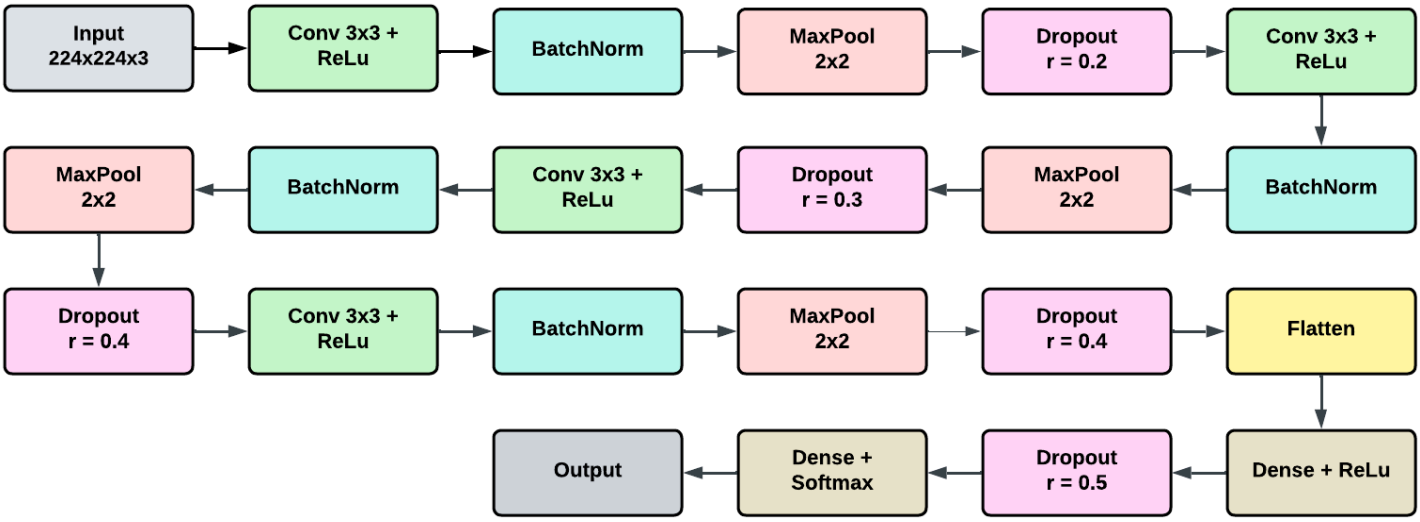

Class Distribution - Validation Set
Number of Images
0
5
10
15
20
25
30
35
0
1
2
3
4
Classes


Class Distribution - Test Set
Number of Images
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
0
1
2
3
4
Classes

III. Deep Learning Architectures

a. Custom CNN

The proposed CNN model is a custom deep convolutional neural network designed for efficient multi-class classification with a focus on scalability and robust generalization. It consists of four convolutional blocks, each incorporating Conv2D layers with ReLU activation, Batch Normalization for stabilized learning, and MaxPooling2D for spatial dimension reduction. Regularization is achieved through progressively increasing dropout rates to mitigate overfitting. The network culminates in fully connected layers with dense neurons to learn high-level representations, followed by a softmax-activated output layer for class prediction. With its compact yet effective design, this model is well-suited for tasks like face detection and recognition, offering a balance between computational efficiency and accuracy.



Model Description

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 222, 222, 32)	896
batch_normalization (BatchNormalization)	(None, 222, 222, 32)	128
max_pooling2d (MaxPooling2D)	(None, 111, 111, 32)	0
dropout (Dropout)	(None, 111, 111, 32)	0
conv2d_1 (Conv2D)	(None, 109, 109, 64)	18,496
batch_normalization_1 (BatchNormalization)	(None, 109, 109, 64)	256
max_pooling2d_1 (MaxPooling2D)	(None, 54, 54, 64)	0
dropout_1 (Dropout)	(None, 54, 54, 64)	0
conv2d_2 (Conv2D)	(None, 52, 52, 128)	73,856
batch_normalization_2 (BatchNormalization)	(None, 52, 52, 128)	512
max_pooling2d_2 (MaxPooling2D)	(None, 26, 26, 128)	0
dropout_2 (Dropout)	(None, 26, 26, 128)	0
conv2d_3 (Conv2D)	(None, 24, 24, 256)	295,168
batch_normalization_3 (BatchNormalization)	(None, 24, 24, 256)	1,024
max_pooling2d_3 (MaxPooling2D)	(None, 12, 12, 256)	0
dropout_3 (Dropout)	(None, 12, 12, 256)	0
flatten (Flatten)	(None, 36864)	0
dense (Dense)	(None, 512)	18,874,880
dropout_4 (Dropout)	(None, 512)	0
dense_1 (Dense)	(None, 256)	131,328
dropout_5 (Dropout)	(None, 256)	0
dense_2 (Dense)	(None, 5)	1,285

Total params: 19,397,829 (74.00 MB)
Trainable params: 19,396,869 (73.99 MB)
Non-trainable params: 960 (3.75 KB)

b. ResNet50

ResNet50 is a deep convolutional neural network architecture consisting of 50 layers, specifically designed to address the vanishing gradient problem in deep networks through the use of residual connections (or skip connections). These connections allow the network to learn identity mappings, ensuring that deeper layers do not degrade performance. ResNet50 employs a combination of convolutional, pooling, and fully connected layers, making it capable of extracting rich, hierarchical features from input images. Its

modular structure and pre-trained weights on large datasets make it a versatile choice for transfer learning in a variety of applications, including facial recognition.

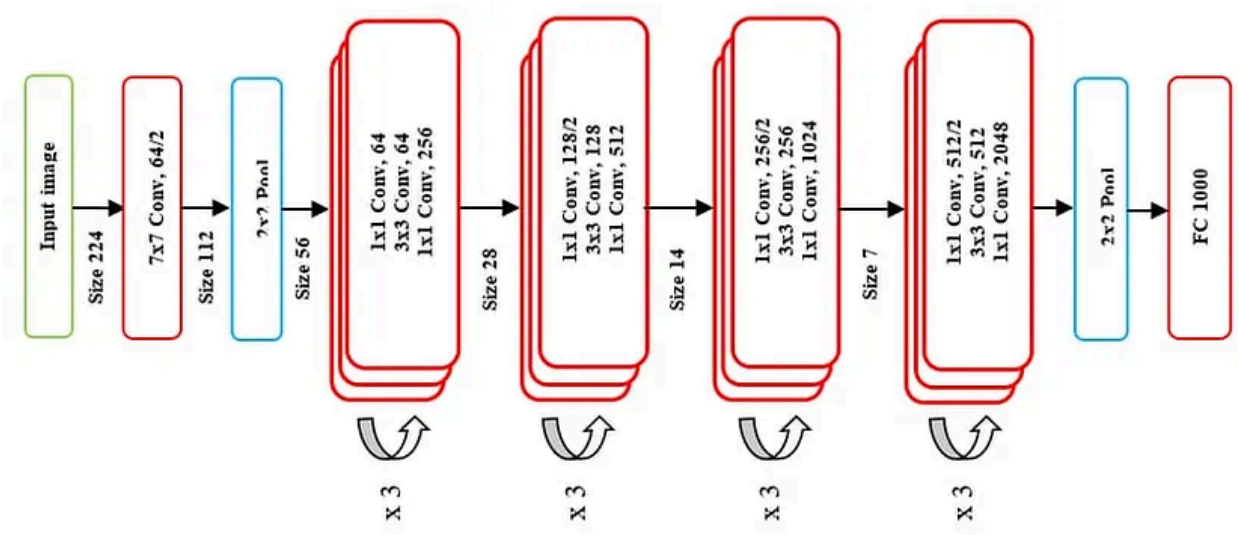


Image reference : https://towardsdatascience.com/the-annotated-resnet-50-a6c536034758

Why ResNet50?

ResNet50 is chosen for its proven track record in achieving high accuracy across various computer vision tasks. The residual learning mechanism enables the network to maintain performance even at great depths, ensuring precise feature extraction. Moreover, its widespread adoption and availability of pre-trained weights on datasets such as ImageNet make it an ideal starting point for fine-tuning on domain-specific tasks like facial recognition.

c. DeepFace by facebook

DeepFace consists of a 9-layer deep neural network designed specifically for facial recognition tasks. The architecture includes convolutional layers for extracting facial features, max-pooling layers to reduce spatial dimensions, and fully connected layers for generating compact and discriminative embeddings. These embeddings map input images into a feature space that captures unique facial characteristics. The network concludes with a

classification layer or a similarity measure, depending on the application (e.g., face verification or identification).

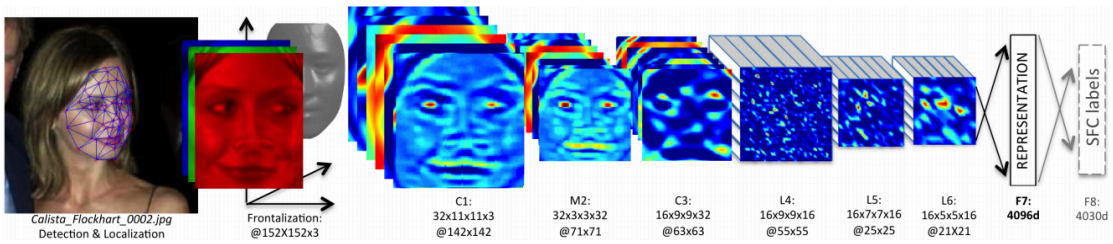


Image reference : [ResearchGate]

Why DeepFace?

DeepFace is chosen because it is a purpose-built model for facial recognition, validated on large-scale datasets with diverse conditions. Its architecture is highly optimized for this specific task, offering pre-trained embeddings that significantly reduce training time while maintaining state-of-the-art performance. Additionally, its simplicity and adaptability make it an excellent choice to integrate with our dataset and project pipeline.

IV. Code Till Date

1. Augmentation

```
'''
This augmentation pipeline applies a series of random transformations,
including
horizontal flips, rotations, brightness/contrast adjustments, shifts,
scaling, and Gaussian blur,
to increase dataset diversity and improve model generalization.
'''

augmentation_pipeline = A.Compose([
    A.HorizontalFlip(p=0.5),
    A.Rotate(limit=20, p=0.5),
    A.RandomBrightnessContrast(p=0.2),
    A.ShiftScaleRotate(shift_limit=0.05, scale_limit=0.05,
rotate_limit=15, p=0.5),
    A.GaussianBlur(blur_limit=3, p=0.2),
])
```

```
"""
This function performs image augmentation for each class in the input
directory.
It applies the specified augmentation pipeline to every image, creating a
given number
of augmented images per original image. The augmented images are then
saved to the
output directory, preserving the class structure to ensure consistency for
training.
"""

def augment_images(input_dir, output_dir, augment_count=10):
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    for class_name in os.listdir(input_dir):
        class_path = os.path.join(input_dir, class_name)
        output_class_path = os.path.join(output_dir, class_name)
        os.makedirs(output_class_path, exist_ok=True)

        for image_name in os.listdir(class_path):
            image_path = os.path.join(class_path, image_name)
            image = np.array(Image.open(image_path))

            for i in range(augment_count):
                augmented = augmentation_pipeline(image=image)
                aug_image = Image.fromarray(augmented["image"])
                aug_image.save(os.path.join(output_class_path,
f"{os.path.splitext(image_name)[0]}_aug{i}.jpg"))
```

2. Localization

```
"""
Initializes the MTCNN (Multi-Task Cascaded Convolutional Networks) model
for face detection
keep_all=True : Ensures that the model detects and processes multiple
faces in a single image
```

```
device : Automatically selects 'cuda' for GPU acceleration if available,
or defaults to 'cpu'
This configuration optimizes performance and enables efficient face
detection on various hardware setups
"""
mtcnn = MTCNN(keep_all=True, device="cuda" if torch.cuda.is_available()
else "cpu")
```

```
"""
This function performs face localization using the MTCNN model for all
images in a given input directory
- The input directory should have subdirectories for each class containing
images
- For each image, the function detects faces, crops the detected regions,
and saves them into a corresponding
  subdirectory in the output directory
- Key Steps:
  1. Create the output directory structure to match the input directory
  2. Load each image, convert it to RGB format, and pass it to the MTCNN
model for face detection
  3. If faces are detected, crop each face based on the bounding boxes
provided by MTCNN
  4. Save the cropped faces as new image files with unique filenames in
the corresponding output subdirectory
- If any error occurs during processing, it logs the error and moves to
the next image
This ensures the creation of a well-organized dataset of cropped face
images for further analysis or training
"""

def localize_faces_mtcnn(input_dir, output_dir):
    os.makedirs(output_dir, exist_ok=True)

    for class_name in os.listdir(input_dir):
        class_path = os.path.join(input_dir, class_name)
        if not os.path.isdir(class_path):
            continue

        # Create output subdirectory for the class
```

```
        output_class_path = os.path.join(output_dir, class_name)
        os.makedirs(output_class_path, exist_ok=True)

        for img_name in os.listdir(class_path):
            img_path = os.path.join(class_path, img_name)
            try:
                # Open the image
                image = Image.open(img_path).convert("RGB")

                # Detect faces
                boxes, _ = mtcnn.detect(image)

                if boxes is not None:
                    for i, box in enumerate(boxes):
                        # Crop face
                        cropped_face = image.crop(box)

                        # Save the cropped face
                        output_path = os.path.join(output_class_path,
f"{os.path.splitext(img_name)[0]}_face{i}.jpg")
                        cropped_face.save(output_path)
                        print(f"Saved cropped face to: {output_path}")
            except Exception as e:
                print(f"Error processing {img_path}: {e}")
```

3. Preprocessing

```
transform = transforms.Compose([
    transforms.Resize((224, 224)),  # Resize to 224x224
    transforms.ToTensor(),          # Convert to tensor
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
0.225]) #normalizing the pixel values
])
```

4. ResNet50 training

```
"""
This code performs a grid search over different hyperparameters (learning
rate, batch size, and optimizer) to train a ResNet50 model
```

```
- For each combination of hyperparameters, the model is reinitialized, and
the data loaders are created with the specified batch size
- The optimizer is set based on the chosen optimizer type (Adam, RMSprop,
or Adagrad) and learning rate
- The model is trained over multiple epochs, where after each epoch,
training and validation accuracies are calculated and stored
- If a checkpoint exists for the current configuration, the training
resumes from that checkpoint
- The best validation accuracy is tracked, and if the current model
performs better than the previous one, it is saved as the best model
- A checkpoint is saved at the end of each epoch, containing the model and
optimizer state, the current epoch, and the best accuracy so far
- After training is complete, the best validation accuracy for each
hyperparameter combination is stored and marked as completed
- A configuration is considered completed if it has already been run, and
such configurations are skipped to avoid redundant computations
"""


for lr in learning_rates:
    for batch_size in batch_sizes:
        for opt in optimizers:
            # Check if this configuration is already completed
            if {'lr': lr, 'batch_size': batch_size, 'optimizer': opt} in
completed_configs:
                print(f"Skipping already completed configuration: lr={lr},
batch_size={batch_size}, optimizer={opt}")
                continue

            # Reinitialize the model, optimizer, and dataloaders for each
combination
            resnet = models.resnet50(pretrained=True)
            num_features = resnet.fc.in_features
            num_classes = len(train_data.classes)  # Ensure train_data is
defined elsewhere
            resnet.fc = nn.Linear(num_features, num_classes)
            device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")
            resnet = resnet.to(device)
```

```
            # Define dataloaders with the current batch size
            train_loader = DataLoader(train_data, batch_size=batch_size,
shuffle=True)
            valid_loader = DataLoader(valid_data, batch_size=batch_size,
shuffle=False)

            # Set optimizer
            if opt == "adam":
                optimizer = optim.Adam(resnet.parameters(), lr=lr)
            elif opt == "rmsprop":
                optimizer = optim.RMSprop(resnet.parameters(), lr=lr)
            elif opt == "adagrad":
                optimizer = optim.Adagrad(resnet.parameters(), lr=lr)

            # Loss function
            criterion = nn.CrossEntropyLoss()

            # Store metrics for visualization
            key = f"lr={lr}_batch={batch_size}_opt={opt}"
            train_accuracies[key] = []
            valid_accuracies[key] = []

            # Check for an existing checkpoint
            checkpoint_path = os.path.join(drive_checkpoint_dir,
f"checkpoint_resnet50_lr{lr}_batch{batch_size}_opt{opt}.pth")
            start_epoch = 0
            if os.path.exists(checkpoint_path):
                print(f"Resuming training from checkpoint:
{checkpoint_path}")
                checkpoint = torch.load(checkpoint_path)
                resnet.load_state_dict(checkpoint['model_state_dict'])

optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
                start_epoch = checkpoint['epoch'] + 1
                best_accuracy = checkpoint['best_accuracy']
                best_params = checkpoint['best_params']

            # Training Loop
            print(f"Training with learning rate: {lr}, batch size:
{batch_size}, optimizer: {opt}")
```

```
            for epoch in range(start_epoch, epochs):
                print(f"Epoch {epoch + 1}/{epochs}")
                resnet.train()
                running_loss = 0.0
                correct = 0
                total = 0

                for images, labels in train_loader:
                    images, labels = images.to(device), labels.to(device)
                    optimizer.zero_grad()
                    outputs = resnet(images)
                    loss = criterion(outputs, labels)
                    loss.backward()
                    optimizer.step()

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

                epoch_accuracy = 100 * correct / total
                train_accuracies[key].append(epoch_accuracy)

                # Validate after each epoch
                resnet.eval()
                valid_loss = 0.0
                correct = 0
                total = 0

                with torch.no_grad():
                    for images, labels in valid_loader:
                        images, labels = images.to(device),
labels.to(device)
                        outputs = resnet(images)
                        loss = criterion(outputs, labels)
                        valid_loss += loss.item()
                        _, predicted = torch.max(outputs, 1)
                        total += labels.size(0)
                        correct += (predicted == labels).sum().item()
```

```
                valid_accuracy = 100 * correct / total
                valid_accuracies[key].append(valid_accuracy)

                # Keep track of the best performing hyperparameters
                if valid_accuracy > best_accuracy:
                    best_accuracy = valid_accuracy
                    best_params = {'learning_rate': lr, 'batch_size':
batch_size, 'optimizer': opt}
                    # Save the best model checkpoint
                    torch.save(resnet.state_dict(),
os.path.join(drive_checkpoint_dir, "best_resnet50.pth"))

                print(f"Train Accuracy: {epoch_accuracy:.2f}%, Validation
Accuracy: {valid_accuracy:.2f}%")

                # Save a checkpoint after each epoch
                torch.save({
                    'model_state_dict': resnet.state_dict(),
                    'optimizer_state_dict': optimizer.state_dict(),
                    'epoch': epoch,
                    'best_accuracy': best_accuracy,
                    'best_params': best_params
                }, checkpoint_path)

            # Store the best validation accuracy for this hyperparameter
set
            hyperparam_results[key] = max(valid_accuracies[key])

            # Mark this configuration as completed
            completed_configs.append({'lr': lr, 'batch_size': batch_size,
'optimizer': opt})
            with open(completed_configs_path, 'w') as f:
                json.dump(completed_configs, f)
```

5. Custom CNN

```
# Create the CNN model
model = Sequential()

# Input Layer and First Convolutional Block
```

```
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(224, 224,
3)))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.2))

# Second Convolutional Block
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.3))

# Third Convolutional Block
model.add(Conv2D(128, (3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.4))

# Fourth Convolutional Block
model.add(Conv2D(256, (3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.4))

# Fully Connected Layers
model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))

# Output Layer
model.add(Dense(5, activation='softmax'))

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

# Display the model summary
```

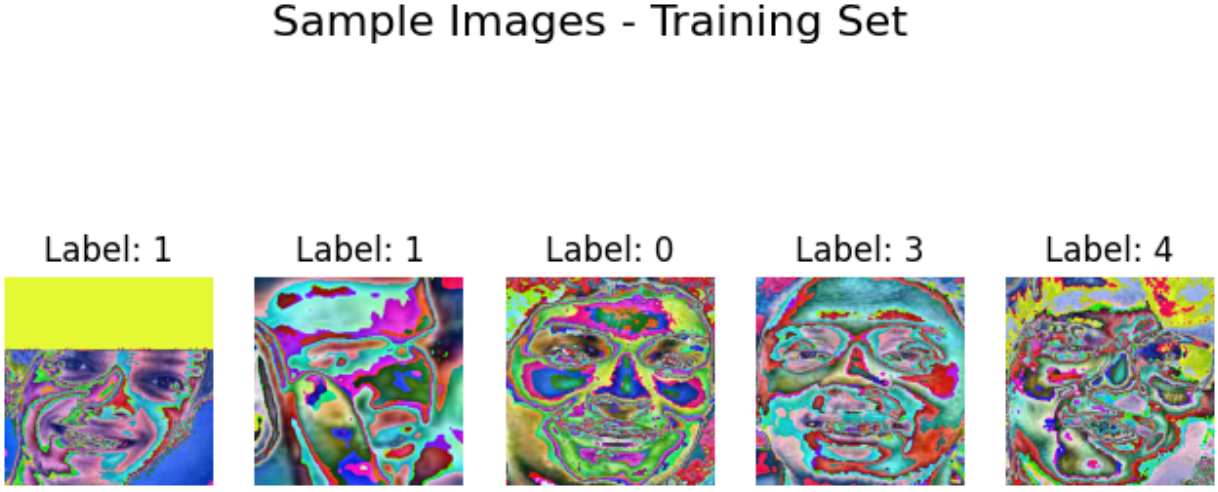
```
model.summary()
```

V. Localization and Preprocessing

Localization using MTCNN:
To detect and localize faces within images, the MTCNN (Multi-task Cascaded Convolutional Networks) algorithm is used. MTCNN is a popular face detection method that works by first generating candidate facial regions, then refining them to accurately localize faces. It outputs bounding boxes around each face, which are then used to crop the images for further processing. This step ensures that the model only focuses on the relevant facial features, improving both the accuracy and efficiency of facial recognition.

Preprocessing:
The images undergo preprocessing to ensure they are properly formatted for model input. The preprocessing pipeline includes:

1. Resize: Each image is resized to 224x224 pixels, ensuring consistency in input size for the models, particularly for architectures like ResNet50, which expect fixed input dimensions.
2. ToTensor: Converts the image into a tensor format, enabling it to be fed into the deep learning model for training.
3. Normalization: The pixel values are normalized with a mean of [0.485, 0.456, 0.406] and standard deviation of [0.229, 0.224, 0.225], which are the values used for pre-trained networks like ResNet50. This step ensures the input images are within a similar distribution to the data the pre-trained models were trained on, facilitating better convergence during training.



VI. Results and Analysis Till Date

a. ResNet50 Model Performance

The ResNet50 model has been fully implemented with multiple configurations of hyperparameters and optimizers. Key hyperparameters such as learning rates, and batch sizes were tuned systematically, and optimizers including Adam, Adagrad, and RMSprop were tested to find the best combination for model performance.

Challenges Faced:

- Runtime Disconnections: Training ResNet50 often exceeded the runtime limits of the colab environment, leading to interruptions.
- Solution: To address this, model checkpoints and training progress were saved periodically using a `.json` file. This allowed training to resume seamlessly without losing progress.
- A summary of the highest validation accuracies achieved for each optimizer with different learning rates:

Learning Rate	Batch Size	Optimizer	Train Accuracy	Validation Accuracy
0.01	32	Adam	36.90%	50.34%
0.01	32	RMSprop	49.90%	48.32%
0.01	32	Adagrad	47.61%	38.26%
0.001	32	Adam	91.40%	90.60%
0.001	32	RMSprop	62.14%	58.39%
0.001	32	Adagrad	99.62%	93.96%
0.0001	32	Adam	99.24%	95.30%
0.0001	32	RMSprop	98.47%	96.64%
0.0001	32	Adagrad	100%	93.96%

Key Observations:

Best Results:

- *RMSprop* with a learning rate of 0.0001 achieved the highest validation accuracy of 96.64%.
- *Adam* with a learning rate of 0.0001 also performed exceptionally well, achieving a validation accuracy of 95.30%.

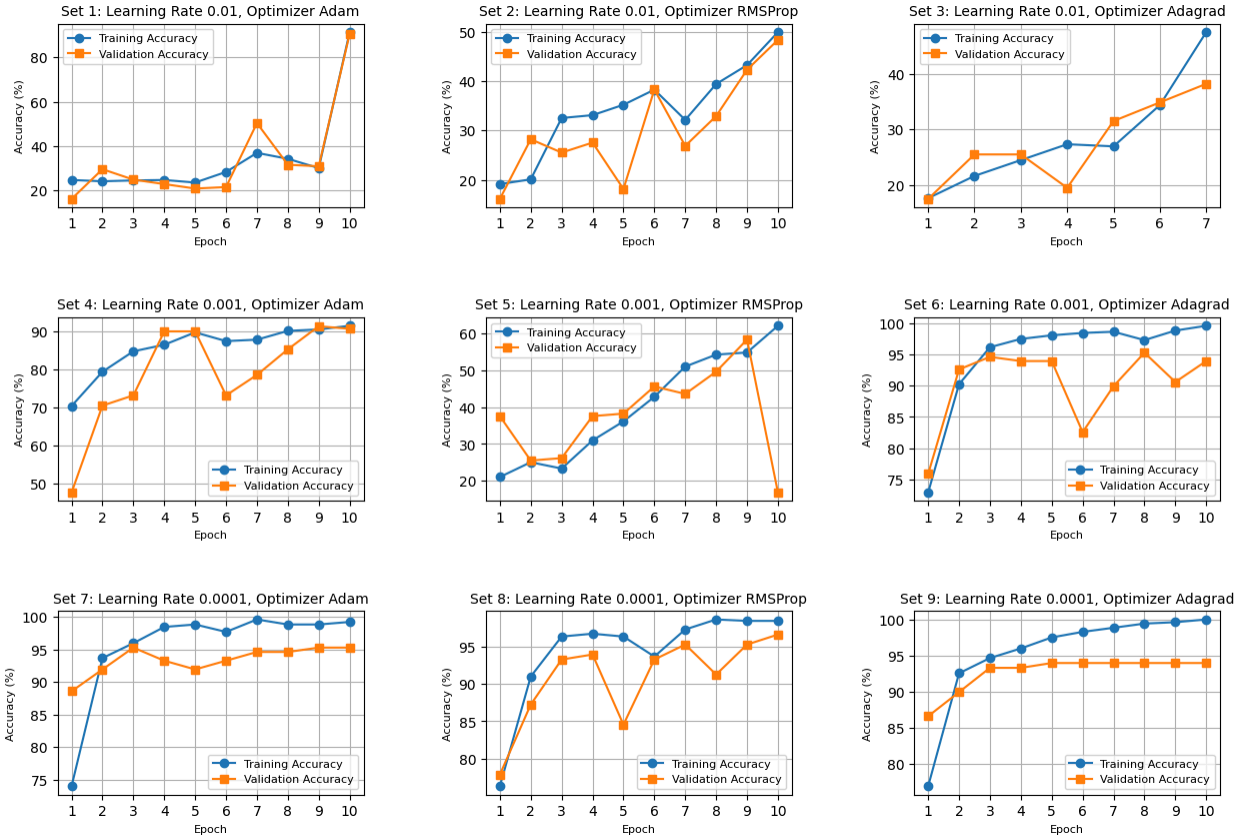
Overfitting Trends:

- High training accuracies were often accompanied by slightly lower validation accuracies, indicating mild overfitting, particularly with *Adagrad*.

Graphical Visualizations :

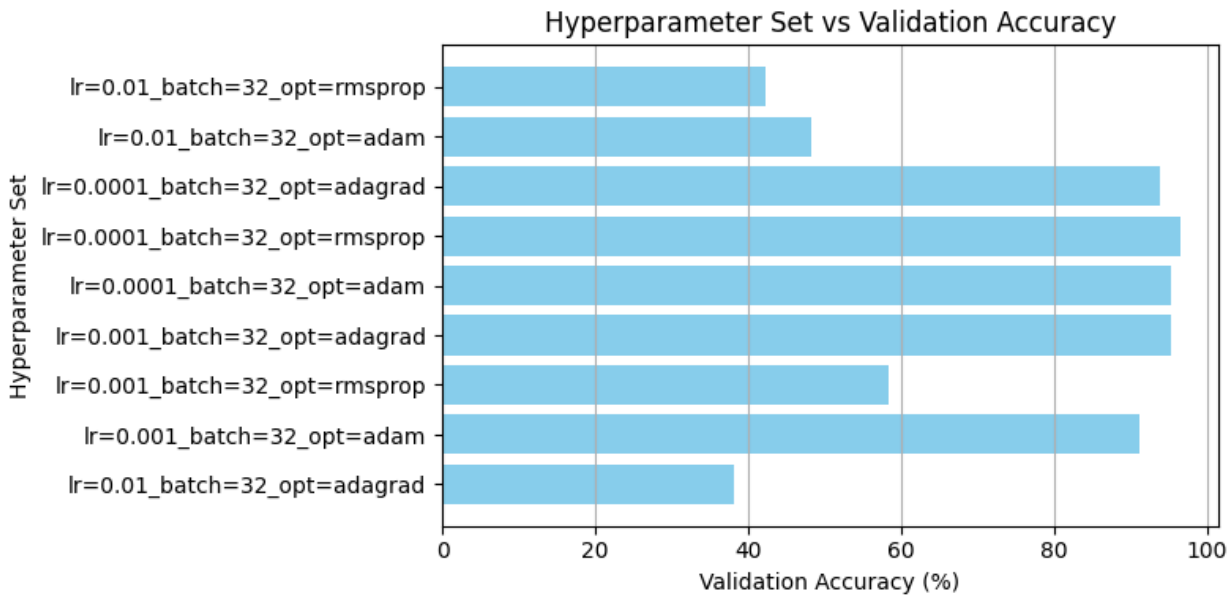
1. **Epoch-wise Accuracy Trends**

Accuracy trends for training and validation across epochs for different learning rates and optimizers (Adam, RMSProp, Adagrad).



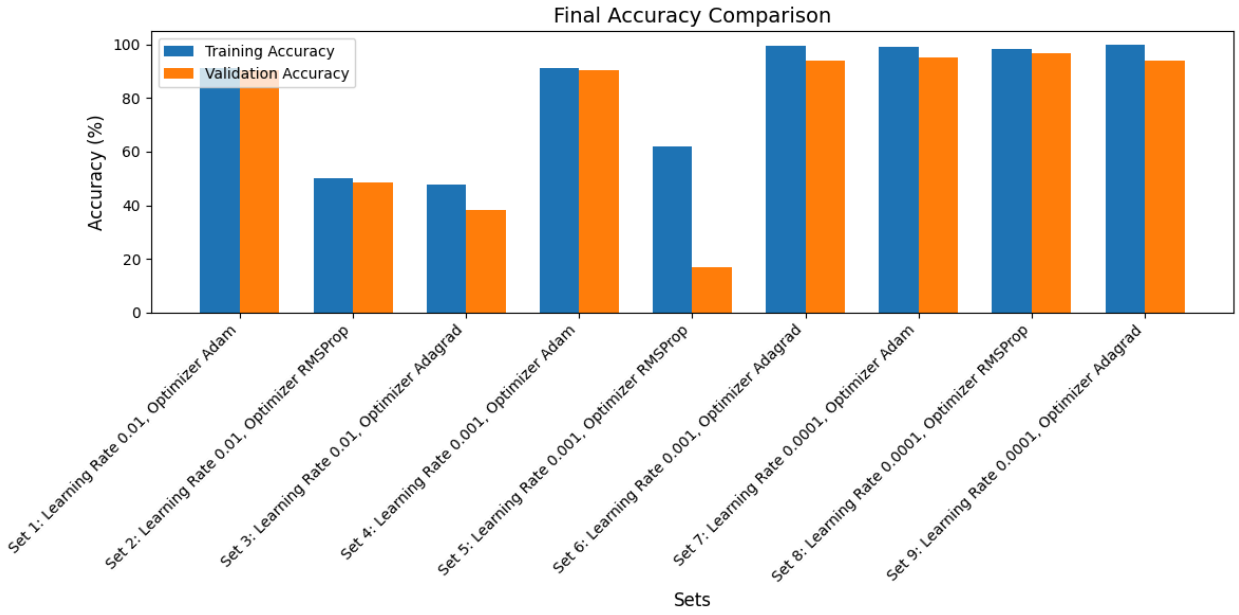
2. Hyperparameter Set vs Accuracy:

Compares the final validation accuracy achieved by each hyperparameter set (combination of learning rate, optimizer, and batch size).



3. Final Accuracy Comparison

Direct comparison of the final training and validation accuracies for each hyperparameter set, showing the relationship between model generalization and overfitting.



Insights:

1. **Learning Rate 0.01**:
    - The performance is unstable across optimizers, with the highest validation accuracy remaining low (below 50%). This indicates that the learning rate is too high, causing the model to struggle with convergence and leading to fluctuating or poor results.
2. **Learning Rate 0.001**:
    - Validation accuracy improves significantly, achieving excellent results with optimizers like Adam (90.60%) and Adagrad (95.30%). This suggests that 0.001 provides an optimal balance between convergence speed and stability.
3. **Learning Rate 0.0001**:
    - Further reduction in learning rate leads to consistent and high validation accuracies, with peaks like 96.64% (RMSProp) and 95.30% (Adam). This demonstrates that while 0.0001 allows precise weight adjustments, it may slightly reduce the speed of convergence compared to 0.001.

VII. Pending Modules

1. **Custom CNN Training:**

- Time Required: < 1 day
- Execute the training process using the finalized architecture and selected hyperparameters.

2. **DeepFace Training:**

- Time Required: < 1 day

3. **Performance Comparison:**

- Time Required: 2 to 3 hours

- Analyzing and comparing metrics from both models.

Total Estimated completion time : 2 days