aiml554-deep-learning-assignment-2

January 30, 2025

1 Assignment 2

Problem Statement: During the COVID-19 pandemic, identifying whether individuals are wearing face masks has become a critical task. In this assignment, your objective is to develop a Convolutional Neural Network (CNN) model to classify face images as either "masked" or "unmasked."

Note: To simplify the task, the dataset provided primarily includes images with a single face and minimal background interference. However, in real-world scenarios, challenges such as multiple faces, varied backgrounds, and different types of masks (e.g., patterned masks, skin-tone masks) may arise.

Dataset Description: The dataset includes face images categorized into "masked" and "unmasked" folders. These images are further divided into training, validation, and testing sets, as shown below:

dataset/ + train_validate/ + unmasked/ (840 images) + masked/ (840 images) + test/ + unmasked/ (160 images) + masked/ (160 images) Download Link: Mask Detection Dataset

Tasks You may use Python libraries to solve the tasks outlined below:

Prepare the Dataset Load the dataset into appropriate data structures, ensuring images are resized to 64x64x3 to be fed as input to the CNN. [1 mark]

Build the CNN Model Using TensorFlow and Keras, create a CNN model with the following indicative architecture:

Convolution Layer $\rightarrow$ Activation Function (ReLU) $\rightarrow$ Pooling Layer (Convolution Layer $\rightarrow$ Activation Function) $\times$ 2 $\rightarrow$ Pooling Layer Fully Connected Layer $\rightarrow$ Activation Function Softmax Classifier Use a pool size of 2x2, filter size of 3x3, and any other standard parameters as needed. [2 marks]

Train the Model Train the model for 70 epochs (E=70). Log and plot the following metrics for each epoch:

Training Loss Training Accuracy Validation Loss Validation Accuracy Save these metrics and present them as a graph after training is complete. [4 marks]

Evaluate the Model Test the trained CNN on the testing dataset and print the classification metrics, including precision, recall, and F1-score. [2 marks]

Model Improvement Modify the default CNN model to improve its performance. For example, you may change hyperparameters, add layers, or use techniques like data augmentation. Compare the performance of the original ("default") and modified ("improved") models by plotting precision and recall side-by-side in a bar chart. [2 marks]

Visualize Predictions Display 5 sample images from the test set predicted as "masked" and 5 predicted as "unmasked." Include the predicted labels for each image. [1 mark]

1.1 Task 1:- Prepare the Dataset

Load the dataset into appropriate data structures, ensuring images are resized to 64x64x3 to be fed as input to the CNN. [1 mark]

[30]:

```
# Importing required packages
import numpy as np
from sklearn.metrics import classification_report
from sklearn.metrics import classification_report
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import layers, models
import matplotlib.pyplot as plt
import os
```

[31]:

```
# Constants
BASE_DIR = os.getcwd()  # Get the current working directory
TRAIN_VAL_DIR = os.path.join(BASE_DIR, "train_validate-20250129T184506Z-001",␣
↪"train_validate")
TEST_DIR = os.path.join(BASE_DIR, "test-20250129T184445Z-001", "test")

IMG_SIZE = (64, 64)
BATCH_SIZE = 32

def create_image_generator(rescale=1.0, validation_split=None, augment=False):
    """
    Create an ImageDataGenerator with optional data augmentation and validation␣
↪split.
    """
    if augment:
        return ImageDataGenerator(
            rescale=rescale,
            validation_split=validation_split,
            rotation_range=20,
            width_shift_range=0.2,
            height_shift_range=0.2,
            horizontal_flip=True,
            zoom_range=0.2
        )
    return ImageDataGenerator(rescale=rescale,␣
↪validation_split=validation_split)

def create_data_flow(directory, generator, target_size, batch_size,␣
↪subset=None, shuffle=True):
```

```
    """
    Create a data flow object using a given ImageDataGenerator.
    """
    return generator.flow_from_directory(
        directory=directory,
        target_size=target_size,
        batch_size=batch_size,
        class_mode='categorical',
        subset=subset,
        shuffle=shuffle
    )

# Create data generators
train_val_generator = create_image_generator(rescale=1./255, validation_split=0.
 ↪2, augment=True)
test_generator = create_image_generator(rescale=1./255)

# Prepare data flows
train_data = create_data_flow(TRAIN_VAL_DIR, train_val_generator, IMG_SIZE,␣
 ↪BATCH_SIZE, subset="training")
val_data = create_data_flow(TRAIN_VAL_DIR, train_val_generator, IMG_SIZE,␣
 ↪BATCH_SIZE, subset="validation")
test_data = create_data_flow(TEST_DIR, test_generator, IMG_SIZE, BATCH_SIZE,␣
 ↪shuffle=False)

# Print class labels mapping
print("Class Labels Mapping:", train_data.class_indices)
```

```
Found 1343 images belonging to 2 classes.
Found 335 images belonging to 2 classes.
Found 320 images belonging to 2 classes.
Class Labels Mapping: {'masked': 0, 'unmasked': 1}
```

1.2 Task 2:- Build the CNN Model

Using TensorFlow and Keras, create a CNN model with the following indicative architecture:

–> Convolution Layer → Activation Function (ReLU) → Pooling Layer

–> (Convolution Layer → Activation Function) × 2 → Pooling Layer

–> Fully Connected Layer → Activation Function

–> Softmax Classifier

Use a pool size of 2x2, filter size of 3x3, and any other standard parameters as needed. [2 marks]

[32]:
```
# Importing required packages
from tensorflow.keras.models import Sequential
```

```python
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense,␣
↪Dropout

def build_CNN_model():
    model = Sequential([
        Conv2D(32, (3, 3), activation='relu', input_shape=(64, 64, 3)),
        MaxPooling2D(pool_size=(2, 2)),

        Conv2D(64, (3, 3), activation='relu'),
        Conv2D(64, (3, 3), activation='relu'),
        MaxPooling2D(pool_size=(2, 2)),

        Flatten(),
        Dense(128, activation='relu'),
        Dropout(0.5),  # Regularization
        Dense(2, activation='softmax')  # Output layer
    ])

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

# Build the model
CNN_model = build_CNN_model()
CNN_model.summary()
```

```
C:\Users\ASUS\miniconda3\Lib\site-
packages\keras\src\layers\convolutional\base_conv.py:107: UserWarning: Do not
pass an `input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in the model
instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential_7"

```
 Layer (type)                          Output Shape                          ␣
↪Param #

 conv2d_20 (Conv2D)                    (None, 62, 62, 32)                    ␣
↪896

 max_pooling2d_16 (MaxPooling2D)       (None, 31, 31, 32)                    ␣
↪  0
```

```
conv2d_21 (Conv2D)                    (None, 29, 29, 64)                 ␣
↪18,496

conv2d_22 (Conv2D)                    (None, 27, 27, 64)                 ␣
↪36,928

max_pooling2d_17 (MaxPooling2D)       (None, 13, 13, 64)                    ␣
↪  0

flatten_7 (Flatten)                   (None, 10816)                         ␣
↪  0

dense_14 (Dense)                      (None, 128)                     ␣
↪1,384,576

dropout_6 (Dropout)                   (None, 128)                           ␣
↪  0

dense_15 (Dense)                      (None, 2)                             ␣
↪258
```

```
Total params: 1,441,154 (5.50 MB)

Trainable params: 1,441,154 (5.50 MB)

Non-trainable params: 0 (0.00 B)
```

1.3 Task 3:- Train the Model

Train the model for 70 epochs (E=70). Log and plot the following metrics for each epoch:

–> Training Loss

–> Training Accuracy

–> Validation Loss

–> Validation Accuracy

Save these metrics and present them as a graph after training is complete. [4 marks]

[33]:
```
# Train the model (use the 'model' variable from earlier)
history = CNN_model.fit(train_data, epochs=70,
                    validation_data=val_data)

# Plot training history
```

```python
plt.figure(figsize=(12, 5))

# Plot accuracy
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Training vs Validation Accuracy')

# Plot loss
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.title('Training vs Validation Loss')

# Show the plots
plt.show()
```

```
C:\Users\ASUS\miniconda3\Lib\site-
packages\keras\src\trainers\data_adapters\py_dataset_adapter.py:121:
UserWarning: Your `PyDataset` class should call `super().__init__(**kwargs)` in
its constructor. `**kwargs` can include `workers`, `use_multiprocessing`,
`max_queue_size`. Do not pass these arguments to `fit()`, as they will be
ignored.
  self._warn_if_super_not_called()
```

```
Epoch 1/70
42/42                 8s 149ms/step -
accuracy: 0.6962 - loss: 0.6026 - val_accuracy: 0.9045 - val_loss: 0.2392
Epoch 2/70
42/42                 6s 138ms/step -
accuracy: 0.8796 - loss: 0.3170 - val_accuracy: 0.8985 - val_loss: 0.2754
Epoch 3/70
42/42                 6s 133ms/step -
accuracy: 0.8930 - loss: 0.2903 - val_accuracy: 0.8925 - val_loss: 0.2368
Epoch 4/70
42/42                 6s 143ms/step -
accuracy: 0.8919 - loss: 0.2366 - val_accuracy: 0.9284 - val_loss: 0.2021
Epoch 5/70
42/42                 6s 143ms/step -
accuracy: 0.9223 - loss: 0.2316 - val_accuracy: 0.9403 - val_loss: 0.1709
Epoch 6/70
42/42                 6s 134ms/step -
```

```
accuracy: 0.9169 - loss: 0.2419 - val_accuracy: 0.9373 - val_loss: 0.1715
Epoch 7/70
42/42            5s 127ms/step -
accuracy: 0.9165 - loss: 0.2265 - val_accuracy: 0.8985 - val_loss: 0.2260
Epoch 8/70
42/42            6s 135ms/step -
accuracy: 0.9107 - loss: 0.2517 - val_accuracy: 0.9642 - val_loss: 0.1479
Epoch 9/70
42/42            6s 144ms/step -
accuracy: 0.9190 - loss: 0.2139 - val_accuracy: 0.9433 - val_loss: 0.1461
Epoch 10/70
42/42            6s 132ms/step -
accuracy: 0.9413 - loss: 0.2021 - val_accuracy: 0.9582 - val_loss: 0.1342
Epoch 11/70
42/42            6s 136ms/step -
accuracy: 0.9459 - loss: 0.1343 - val_accuracy: 0.9433 - val_loss: 0.1689
Epoch 12/70
42/42            6s 133ms/step -
accuracy: 0.9250 - loss: 0.1880 - val_accuracy: 0.9403 - val_loss: 0.1847
Epoch 13/70
42/42            6s 134ms/step -
accuracy: 0.9182 - loss: 0.2031 - val_accuracy: 0.9403 - val_loss: 0.1774
Epoch 14/70
42/42            6s 143ms/step -
accuracy: 0.9505 - loss: 0.1365 - val_accuracy: 0.9403 - val_loss: 0.1543
Epoch 15/70
42/42            6s 139ms/step -
accuracy: 0.9357 - loss: 0.1655 - val_accuracy: 0.9313 - val_loss: 0.1925
Epoch 16/70
42/42            6s 136ms/step -
accuracy: 0.9394 - loss: 0.1743 - val_accuracy: 0.9403 - val_loss: 0.1748
Epoch 17/70
42/42            6s 133ms/step -
accuracy: 0.9198 - loss: 0.2012 - val_accuracy: 0.9284 - val_loss: 0.1576
Epoch 18/70
42/42            6s 136ms/step -
accuracy: 0.9348 - loss: 0.1760 - val_accuracy: 0.9552 - val_loss: 0.1604
Epoch 19/70
42/42            6s 144ms/step -
accuracy: 0.9437 - loss: 0.1527 - val_accuracy: 0.9433 - val_loss: 0.1543
Epoch 20/70
42/42            6s 133ms/step -
accuracy: 0.9298 - loss: 0.1877 - val_accuracy: 0.9493 - val_loss: 0.1614
Epoch 21/70
42/42            6s 142ms/step -
accuracy: 0.9484 - loss: 0.1749 - val_accuracy: 0.9522 - val_loss: 0.1338
Epoch 22/70
42/42            6s 135ms/step -
```

```
accuracy: 0.9332 - loss: 0.1602 - val_accuracy: 0.9403 - val_loss: 0.1529
Epoch 23/70
42/42            6s 142ms/step -
accuracy: 0.9348 - loss: 0.1471 - val_accuracy: 0.9313 - val_loss: 0.1927
Epoch 24/70
42/42            6s 136ms/step -
accuracy: 0.9420 - loss: 0.1471 - val_accuracy: 0.9313 - val_loss: 0.1852
Epoch 25/70
42/42            6s 140ms/step -
accuracy: 0.9574 - loss: 0.1300 - val_accuracy: 0.9463 - val_loss: 0.1547
Epoch 26/70
42/42            6s 144ms/step -
accuracy: 0.9494 - loss: 0.1442 - val_accuracy: 0.9433 - val_loss: 0.1600
Epoch 27/70
42/42            6s 145ms/step -
accuracy: 0.9392 - loss: 0.1459 - val_accuracy: 0.9373 - val_loss: 0.1798
Epoch 28/70
42/42            6s 135ms/step -
accuracy: 0.9474 - loss: 0.1461 - val_accuracy: 0.9373 - val_loss: 0.1728
Epoch 29/70
42/42            6s 147ms/step -
accuracy: 0.9544 - loss: 0.1236 - val_accuracy: 0.9463 - val_loss: 0.1531
Epoch 30/70
42/42            6s 138ms/step -
accuracy: 0.9509 - loss: 0.1434 - val_accuracy: 0.9463 - val_loss: 0.1507
Epoch 31/70
42/42            6s 142ms/step -
accuracy: 0.9388 - loss: 0.1525 - val_accuracy: 0.9552 - val_loss: 0.1130
Epoch 32/70
42/42            6s 145ms/step -
accuracy: 0.9682 - loss: 0.0909 - val_accuracy: 0.9463 - val_loss: 0.1665
Epoch 33/70
42/42            6s 137ms/step -
accuracy: 0.9640 - loss: 0.1067 - val_accuracy: 0.9433 - val_loss: 0.1311
Epoch 34/70
42/42            6s 137ms/step -
accuracy: 0.9483 - loss: 0.1524 - val_accuracy: 0.9403 - val_loss: 0.1913
Epoch 35/70
42/42            6s 141ms/step -
accuracy: 0.9571 - loss: 0.1209 - val_accuracy: 0.9493 - val_loss: 0.1598
Epoch 36/70
42/42            6s 142ms/step -
accuracy: 0.9582 - loss: 0.1308 - val_accuracy: 0.9463 - val_loss: 0.1869
Epoch 37/70
42/42            6s 132ms/step -
accuracy: 0.9326 - loss: 0.1617 - val_accuracy: 0.9493 - val_loss: 0.1560
Epoch 38/70
42/42            6s 139ms/step -
```

```
accuracy: 0.9576 - loss: 0.1283 - val_accuracy: 0.9433 - val_loss: 0.1771
Epoch 39/70
42/42            6s 134ms/step -
accuracy: 0.9525 - loss: 0.1174 - val_accuracy: 0.9343 - val_loss: 0.2031
Epoch 40/70
42/42            6s 139ms/step -
accuracy: 0.9481 - loss: 0.1521 - val_accuracy: 0.9582 - val_loss: 0.1308
Epoch 41/70
42/42            6s 149ms/step -
accuracy: 0.9515 - loss: 0.1102 - val_accuracy: 0.9493 - val_loss: 0.1407
Epoch 42/70
42/42            6s 138ms/step -
accuracy: 0.9503 - loss: 0.1216 - val_accuracy: 0.9522 - val_loss: 0.1432
Epoch 43/70
42/42            6s 135ms/step -
accuracy: 0.9489 - loss: 0.1376 - val_accuracy: 0.9373 - val_loss: 0.1846
Epoch 44/70
42/42            6s 143ms/step -
accuracy: 0.9598 - loss: 0.1276 - val_accuracy: 0.9612 - val_loss: 0.1241
Epoch 45/70
42/42            6s 135ms/step -
accuracy: 0.9661 - loss: 0.1148 - val_accuracy: 0.9403 - val_loss: 0.1670
Epoch 46/70
42/42            6s 137ms/step -
accuracy: 0.9489 - loss: 0.1374 - val_accuracy: 0.9015 - val_loss: 0.3628
Epoch 47/70
42/42            6s 143ms/step -
accuracy: 0.9431 - loss: 0.1406 - val_accuracy: 0.9373 - val_loss: 0.1828
Epoch 48/70
42/42            6s 153ms/step -
accuracy: 0.9584 - loss: 0.1250 - val_accuracy: 0.9463 - val_loss: 0.1613
Epoch 49/70
42/42            6s 134ms/step -
accuracy: 0.9648 - loss: 0.1081 - val_accuracy: 0.9552 - val_loss: 0.1421
Epoch 50/70
42/42            6s 132ms/step -
accuracy: 0.9707 - loss: 0.0862 - val_accuracy: 0.9493 - val_loss: 0.1433
Epoch 51/70
42/42            6s 140ms/step -
accuracy: 0.9567 - loss: 0.0976 - val_accuracy: 0.9582 - val_loss: 0.1454
Epoch 52/70
42/42            6s 137ms/step -
accuracy: 0.9602 - loss: 0.1056 - val_accuracy: 0.9552 - val_loss: 0.1831
Epoch 53/70
42/42            6s 143ms/step -
accuracy: 0.9511 - loss: 0.1415 - val_accuracy: 0.9582 - val_loss: 0.1815
Epoch 54/70
42/42            6s 138ms/step -
```

```
accuracy: 0.9639 - loss: 0.0917 - val_accuracy: 0.9463 - val_loss: 0.1787
Epoch 55/70
42/42            6s 135ms/step -
accuracy: 0.9706 - loss: 0.0986 - val_accuracy: 0.9612 - val_loss: 0.1479
Epoch 56/70
42/42            6s 137ms/step -
accuracy: 0.9625 - loss: 0.1077 - val_accuracy: 0.9582 - val_loss: 0.1673
Epoch 57/70
42/42            6s 136ms/step -
accuracy: 0.9613 - loss: 0.1070 - val_accuracy: 0.9493 - val_loss: 0.1618
Epoch 58/70
42/42            6s 134ms/step -
accuracy: 0.9567 - loss: 0.1240 - val_accuracy: 0.9373 - val_loss: 0.2073
Epoch 59/70
42/42            6s 139ms/step -
accuracy: 0.9546 - loss: 0.1052 - val_accuracy: 0.9642 - val_loss: 0.1902
Epoch 60/70
42/42            6s 133ms/step -
accuracy: 0.9761 - loss: 0.0633 - val_accuracy: 0.9493 - val_loss: 0.1613
Epoch 61/70
42/42            6s 135ms/step -
accuracy: 0.9719 - loss: 0.0733 - val_accuracy: 0.9493 - val_loss: 0.2314
Epoch 62/70
42/42            6s 136ms/step -
accuracy: 0.9692 - loss: 0.0701 - val_accuracy: 0.9403 - val_loss: 0.1588
Epoch 63/70
42/42            6s 141ms/step -
accuracy: 0.9569 - loss: 0.1377 - val_accuracy: 0.9373 - val_loss: 0.2093
Epoch 64/70
42/42            6s 144ms/step -
accuracy: 0.9411 - loss: 0.1784 - val_accuracy: 0.9493 - val_loss: 0.1721
Epoch 65/70
42/42            6s 142ms/step -
accuracy: 0.9644 - loss: 0.1041 - val_accuracy: 0.9373 - val_loss: 0.2453
Epoch 66/70
42/42            6s 134ms/step -
accuracy: 0.9692 - loss: 0.0976 - val_accuracy: 0.9403 - val_loss: 0.2306
Epoch 67/70
42/42            6s 140ms/step -
accuracy: 0.9652 - loss: 0.1054 - val_accuracy: 0.9493 - val_loss: 0.1612
Epoch 68/70
42/42            6s 140ms/step -
accuracy: 0.9686 - loss: 0.0901 - val_accuracy: 0.9701 - val_loss: 0.1681
Epoch 69/70
42/42            6s 138ms/step -
accuracy: 0.9636 - loss: 0.1015 - val_accuracy: 0.9493 - val_loss: 0.1700
Epoch 70/70
42/42            6s 146ms/step -
```

```
accuracy: 0.9567 - loss: 0.1143 - val_accuracy: 0.9433 - val_loss: 0.1871
```

1.4 Task 4:- Evaluate the Model

Test the trained CNN on the testing dataset and print the classification metrics, including precision, recall, and F1-score. [2 marks]

[35]:
```python
from sklearn.metrics import classification_report

# Assuming 'test_data' contains the test images and 'test_labels' contains the corresponding labels
# You might need to adjust the data according to your actual variable names.

# Step 1: Make predictions on the test set
test_predictions = CNN_model.predict(test_data)

# Step 2: Convert predicted probabilities to class labels (0 or 1 for binary classification)
predicted_classes = test_predictions.argmax(axis=1)  # For binary or multi-class classification

# Step 3: Get the true class labels
true_classes = test_data.classes # Assuming test_labels are one-hot encoded, if not, just use test_labels directly.

# Step 4: Generate classification report
report = classification_report(true_classes, predicted_classes, target_names=['masked', 'unmasked'])

# Print the classification metrics
```

```
print("Classification Report:\n",report)
```

```
10/10            1s 106ms/step
Classification Report:
               precision    recall  f1-score   support

      masked       0.86      0.91      0.89       160
    unmasked       0.91      0.86      0.88       160

    accuracy                           0.88       320
   macro avg       0.89      0.88      0.88       320
weighted avg       0.89      0.88      0.88       320
```

1.5 Task 5:-Model Improvement

Modify the default CNN model to improve its performance. For example, you may change hyper-parameters, add layers, or use techniques like data augmentation. Compare the performance of the original ("default") and modified ("improved") models by plotting precision and recall side-by-side in a bar chart. [2 marks]

1.5.1 To improve the performance of a Convolutional Neural Network (CNN), we can apply several techniques like adjusting hyperparameters, adding layers, or using data augmentation. Here's how you can approach this task step by step:

Step 1. Default CNN Model The default CNN model can be a simple architecture. For example, consider the following basic CNN structure:

[49]:
```
# Importing required packages
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense,␣
 ↪Dropout

def build_CNN_model():
    model = Sequential([
        Conv2D(32, (3, 3), activation='relu', input_shape=(64, 64, 3)),
        MaxPooling2D(pool_size=(2, 2)),

        Conv2D(64, (3, 3), activation='relu'),
        Conv2D(64, (3, 3), activation='relu'),
        MaxPooling2D(pool_size=(2, 2)),

        Flatten(),
        Dense(128, activation='relu'),
        Dropout(0.5),  # Regularization
        Dense(2, activation='softmax')  # Output layer
    ])
```

```
    model.compile(optimizer='adam',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    return model

# Build the model
CNN_model = build_CNN_model()
```

```
C:\Users\ASUS\miniconda3\Lib\site-
packages\keras\src\layers\convolutional\base_conv.py:107: UserWarning: Do not
pass an `input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in the model
instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Step 2. Improved CNN Model For improving performance, we can:

–> Increase the depth of the network by adding more convolutional layers.

–> Use Batch Normalization to stabilize and speed up training.

–> Use data augmentation to artificially increase the dataset size and variability.

[37]:
```
from tensorflow.keras.layers import BatchNormalization

def build_improved_model():
    model = Sequential([
        Conv2D(32, (3, 3), activation='relu', input_shape=(64, 64, 3)),
        BatchNormalization(),
        MaxPooling2D(pool_size=(2, 2)),

        Conv2D(64, (3, 3), activation='relu'),
        BatchNormalization(),
        Conv2D(64, (3, 3), activation='relu'),
        BatchNormalization(),
        MaxPooling2D(pool_size=(2, 2)),

        Flatten(),
        Dense(256, activation='relu'),
        Dropout(0.5),
        Dense(2, activation='softmax')
    ])

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

# Train improved model
```

```python
improved_model = build_improved_model()
history_improved = improved_model.fit(train_data, epochs=70,␣
 ↪validation_data=val_data)
```

```
C:\Users\ASUS\miniconda3\Lib\site-
packages\keras\src\layers\convolutional\base_conv.py:107: UserWarning: Do not
pass an `input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in the model
instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
Epoch 1/70
42/42          15s 258ms/step -
accuracy: 0.7930 - loss: 1.6362 - val_accuracy: 0.6776 - val_loss: 1.8728
Epoch 2/70
42/42          7s 176ms/step -
accuracy: 0.9086 - loss: 0.7176 - val_accuracy: 0.5015 - val_loss: 6.6504
Epoch 3/70
42/42          7s 167ms/step -
accuracy: 0.8996 - loss: 0.5179 - val_accuracy: 0.5701 - val_loss: 2.2219
Epoch 4/70
42/42          7s 172ms/step -
accuracy: 0.9085 - loss: 0.3700 - val_accuracy: 0.6269 - val_loss: 2.4328
Epoch 5/70
42/42          7s 173ms/step -
accuracy: 0.9007 - loss: 0.3946 - val_accuracy: 0.5045 - val_loss: 7.0093
Epoch 6/70
42/42          7s 170ms/step -
accuracy: 0.9159 - loss: 0.3087 - val_accuracy: 0.5015 - val_loss: 15.2860
Epoch 7/70
42/42          7s 171ms/step -
accuracy: 0.9256 - loss: 0.2439 - val_accuracy: 0.5463 - val_loss: 6.8610
Epoch 8/70
42/42          7s 176ms/step -
accuracy: 0.9133 - loss: 0.2523 - val_accuracy: 0.5254 - val_loss: 8.8404
Epoch 9/70
42/42          7s 174ms/step -
accuracy: 0.9367 - loss: 0.1766 - val_accuracy: 0.5940 - val_loss: 7.7771
Epoch 10/70
42/42          7s 172ms/step -
accuracy: 0.9206 - loss: 0.2363 - val_accuracy: 0.6239 - val_loss: 1.5684
Epoch 11/70
42/42          7s 163ms/step -
accuracy: 0.9467 - loss: 0.1739 - val_accuracy: 0.8299 - val_loss: 0.5902
Epoch 12/70
42/42          7s 173ms/step -
accuracy: 0.9433 - loss: 0.1962 - val_accuracy: 0.9164 - val_loss: 0.3768
Epoch 13/70
```

```
42/42            7s 169ms/step -
accuracy: 0.9303 - loss: 0.2088 - val_accuracy: 0.8925 - val_loss: 0.3445
Epoch 14/70
42/42            7s 173ms/step -
accuracy: 0.9482 - loss: 0.1535 - val_accuracy: 0.7642 - val_loss: 1.4761
Epoch 15/70
42/42            7s 165ms/step -
accuracy: 0.9445 - loss: 0.1736 - val_accuracy: 0.9373 - val_loss: 0.3660
Epoch 16/70
42/42            7s 171ms/step -
accuracy: 0.9476 - loss: 0.1496 - val_accuracy: 0.9493 - val_loss: 0.2196
Epoch 17/70
42/42            7s 175ms/step -
accuracy: 0.9430 - loss: 0.1660 - val_accuracy: 0.9164 - val_loss: 0.3798
Epoch 18/70
42/42            7s 170ms/step -
accuracy: 0.9600 - loss: 0.1346 - val_accuracy: 0.9015 - val_loss: 0.5008
Epoch 19/70
42/42            7s 170ms/step -
accuracy: 0.9578 - loss: 0.1346 - val_accuracy: 0.9254 - val_loss: 0.2506
Epoch 20/70
42/42            7s 172ms/step -
accuracy: 0.9592 - loss: 0.1218 - val_accuracy: 0.9104 - val_loss: 0.2145
Epoch 21/70
42/42            7s 173ms/step -
accuracy: 0.9479 - loss: 0.1448 - val_accuracy: 0.5672 - val_loss: 3.2680
Epoch 22/70
42/42            7s 169ms/step -
accuracy: 0.9539 - loss: 0.1652 - val_accuracy: 0.9552 - val_loss: 0.1435
Epoch 23/70
42/42            7s 176ms/step -
accuracy: 0.9352 - loss: 0.1777 - val_accuracy: 0.9463 - val_loss: 0.1673
Epoch 24/70
42/42            7s 176ms/step -
accuracy: 0.9619 - loss: 0.0918 - val_accuracy: 0.9194 - val_loss: 0.2380
Epoch 25/70
42/42            7s 173ms/step -
accuracy: 0.9583 - loss: 0.1212 - val_accuracy: 0.9463 - val_loss: 0.1499
Epoch 26/70
42/42            7s 175ms/step -
accuracy: 0.9569 - loss: 0.1259 - val_accuracy: 0.9672 - val_loss: 0.1339
Epoch 27/70
42/42            7s 172ms/step -
accuracy: 0.9642 - loss: 0.0945 - val_accuracy: 0.4985 - val_loss: 4.9681
Epoch 28/70
42/42            7s 168ms/step -
accuracy: 0.9588 - loss: 0.0988 - val_accuracy: 0.8507 - val_loss: 0.3312
Epoch 29/70
```

```
42/42             7s 173ms/step -
accuracy: 0.9448 - loss: 0.1496 - val_accuracy: 0.8896 - val_loss: 0.2950
Epoch 30/70
42/42             7s 167ms/step -
accuracy: 0.9623 - loss: 0.1155 - val_accuracy: 0.9433 - val_loss: 0.1239
Epoch 31/70
42/42             7s 174ms/step -
accuracy: 0.9629 - loss: 0.1346 - val_accuracy: 0.9642 - val_loss: 0.1406
Epoch 32/70
42/42             7s 171ms/step -
accuracy: 0.9640 - loss: 0.1092 - val_accuracy: 0.5463 - val_loss: 5.8765
Epoch 33/70
42/42             7s 169ms/step -
accuracy: 0.9593 - loss: 0.1337 - val_accuracy: 0.9104 - val_loss: 0.2290
Epoch 34/70
42/42             7s 174ms/step -
accuracy: 0.9625 - loss: 0.1301 - val_accuracy: 0.5224 - val_loss: 2.1568
Epoch 35/70
42/42             8s 178ms/step -
accuracy: 0.9594 - loss: 0.1160 - val_accuracy: 0.9194 - val_loss: 0.2835
Epoch 36/70
42/42             7s 171ms/step -
accuracy: 0.9641 - loss: 0.1145 - val_accuracy: 0.6776 - val_loss: 0.6513
Epoch 37/70
42/42             7s 176ms/step -
accuracy: 0.9503 - loss: 0.1275 - val_accuracy: 0.9433 - val_loss: 0.1990
Epoch 38/70
42/42             7s 171ms/step -
accuracy: 0.9589 - loss: 0.1461 - val_accuracy: 0.9284 - val_loss: 0.2097
Epoch 39/70
42/42             7s 168ms/step -
accuracy: 0.9731 - loss: 0.0965 - val_accuracy: 0.9582 - val_loss: 0.2367
Epoch 40/70
42/42             7s 166ms/step -
accuracy: 0.9686 - loss: 0.1027 - val_accuracy: 0.8836 - val_loss: 0.4355
Epoch 41/70
42/42             7s 174ms/step -
accuracy: 0.9658 - loss: 0.0938 - val_accuracy: 0.7731 - val_loss: 0.6783
Epoch 42/70
42/42             7s 172ms/step -
accuracy: 0.9723 - loss: 0.1115 - val_accuracy: 0.7164 - val_loss: 0.9034
Epoch 43/70
42/42             7s 172ms/step -
accuracy: 0.9663 - loss: 0.0874 - val_accuracy: 0.9134 - val_loss: 0.4280
Epoch 44/70
42/42             7s 171ms/step -
accuracy: 0.9593 - loss: 0.1281 - val_accuracy: 0.9075 - val_loss: 0.3434
Epoch 45/70
```

```
42/42            7s 175ms/step -
accuracy: 0.9700 - loss: 0.0925 - val_accuracy: 0.9403 - val_loss: 0.1738
Epoch 46/70
42/42            7s 164ms/step -
accuracy: 0.9570 - loss: 0.1078 - val_accuracy: 0.9433 - val_loss: 0.2369
Epoch 47/70
42/42            7s 173ms/step -
accuracy: 0.9684 - loss: 0.0817 - val_accuracy: 0.9373 - val_loss: 0.1735
Epoch 48/70
42/42            7s 170ms/step -
accuracy: 0.9770 - loss: 0.0772 - val_accuracy: 0.8925 - val_loss: 0.3053
Epoch 49/70
42/42            7s 170ms/step -
accuracy: 0.9716 - loss: 0.0976 - val_accuracy: 0.9343 - val_loss: 0.1641
Epoch 50/70
42/42            7s 159ms/step -
accuracy: 0.9629 - loss: 0.1226 - val_accuracy: 0.9403 - val_loss: 0.2458
Epoch 51/70
42/42            7s 170ms/step -
accuracy: 0.9759 - loss: 0.0555 - val_accuracy: 0.9493 - val_loss: 0.1597
Epoch 52/70
42/42            7s 175ms/step -
accuracy: 0.9784 - loss: 0.0780 - val_accuracy: 0.6567 - val_loss: 1.3010
Epoch 53/70
42/42            7s 176ms/step -
accuracy: 0.9604 - loss: 0.1093 - val_accuracy: 0.6955 - val_loss: 0.7502
Epoch 54/70
42/42            7s 173ms/step -
accuracy: 0.9703 - loss: 0.0865 - val_accuracy: 0.9433 - val_loss: 0.1960
Epoch 55/70
42/42            7s 167ms/step -
accuracy: 0.9746 - loss: 0.0712 - val_accuracy: 0.9194 - val_loss: 0.2048
Epoch 56/70
42/42            7s 167ms/step -
accuracy: 0.9718 - loss: 0.0714 - val_accuracy: 0.9164 - val_loss: 0.2141
Epoch 57/70
42/42            7s 164ms/step -
accuracy: 0.9603 - loss: 0.1139 - val_accuracy: 0.9672 - val_loss: 0.1608
Epoch 58/70
42/42            7s 171ms/step -
accuracy: 0.9743 - loss: 0.0849 - val_accuracy: 0.9552 - val_loss: 0.1103
Epoch 59/70
42/42            7s 174ms/step -
accuracy: 0.9739 - loss: 0.0802 - val_accuracy: 0.9642 - val_loss: 0.1212
Epoch 60/70
42/42            7s 171ms/step -
accuracy: 0.9760 - loss: 0.0657 - val_accuracy: 0.9701 - val_loss: 0.1149
Epoch 61/70
```

```
42/42               7s 170ms/step -
accuracy: 0.9747 - loss: 0.0718 - val_accuracy: 0.9463 - val_loss: 0.2777
Epoch 62/70
42/42               7s 174ms/step -
accuracy: 0.9718 - loss: 0.1442 - val_accuracy: 0.6746 - val_loss: 1.7598
Epoch 63/70
42/42               7s 176ms/step -
accuracy: 0.9663 - loss: 0.1186 - val_accuracy: 0.9403 - val_loss: 0.3955
Epoch 64/70
42/42               7s 171ms/step -
accuracy: 0.9600 - loss: 0.1251 - val_accuracy: 0.6299 - val_loss: 9.6778
Epoch 65/70
42/42               7s 166ms/step -
accuracy: 0.9528 - loss: 0.2260 - val_accuracy: 0.4955 - val_loss: 7.1845
Epoch 66/70
42/42               7s 166ms/step -
accuracy: 0.9715 - loss: 0.0835 - val_accuracy: 0.5433 - val_loss: 2.8430
Epoch 67/70
42/42               7s 172ms/step -
accuracy: 0.9710 - loss: 0.1102 - val_accuracy: 0.7612 - val_loss: 0.7189
Epoch 68/70
42/42               7s 168ms/step -
accuracy: 0.9692 - loss: 0.1120 - val_accuracy: 0.9075 - val_loss: 0.3456
Epoch 69/70
42/42               8s 184ms/step -
accuracy: 0.9739 - loss: 0.0779 - val_accuracy: 0.8985 - val_loss: 0.4546
Epoch 70/70
42/42               7s 178ms/step -
accuracy: 0.9727 - loss: 0.1020 - val_accuracy: 0.9642 - val_loss: 0.1205
```

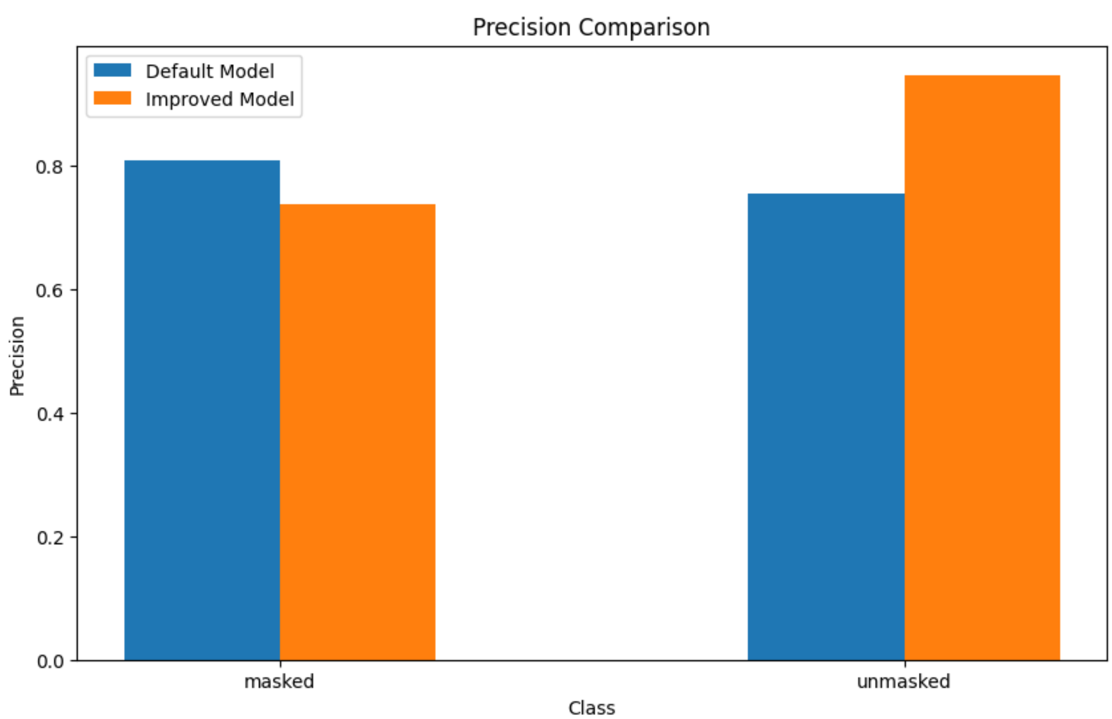
Step 3:- Model Comparison (Precision and Recall)

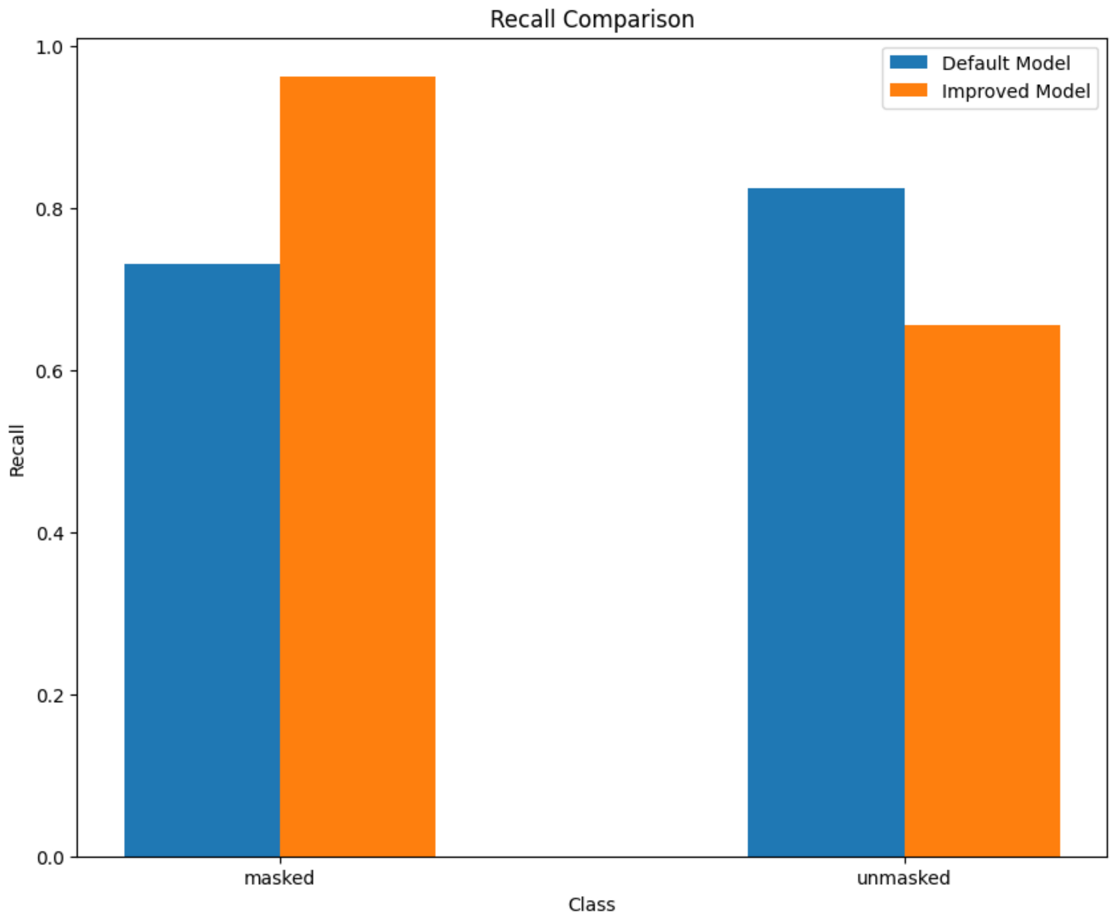
[56]:
```python
from sklearn.metrics import classification_report
import matplotlib.pyplot as plt

# Step 1: Predict using the default model
test_predictions_default = default_model.predict(test_data)
predicted_classes_default = test_predictions_default.argmax(axis=1)

# Step 2: Predict using the improved model
test_predictions_improved = improved_model.predict(test_data)
predicted_classes_improved = test_predictions_improved.argmax(axis=1)

# Step 3: Get the true class labels
true_classes = test_data.classes

# Step 4: Get the classification reports for both models
```

```
report_default = classification_report(true_classes, predicted_classes_default,␣
↪output_dict=True)
report_improved = classification_report(true_classes,␣
↪predicted_classes_improved, output_dict=True)

# Extract precision and recall for the default model
precision_default = [
    report_default[str(i)]['precision'] for i in range(len(train_data.
↪class_indices))
]
recall_default = [
    report_default[str(i)]['recall'] for i in range(len(train_data.
↪class_indices))
]

# Extract precision and recall for the improved model
precision_improved = [
    report_improved[str(i)]['precision'] for i in range(len(train_data.
↪class_indices))
]
recall_improved = [
    report_improved[str(i)]['recall'] for i in range(len(train_data.
↪class_indices))
]

# Step 6: Plot the comparison
# Class names for labels
class_names = list(train_data.class_indices.keys())

# Set up the bar width and positions
x = np.arange(len(class_names))  # [0, 1] for two classes
width = 0.25  # Bar width

plt.figure(figsize=(10, 6))

# Plot Precision
plt.bar(x - width / 2, precision_default, width, label='Default Model')
plt.bar(x + width / 2, precision_improved, width, label='Improved Model')

plt.xlabel('Class')
plt.ylabel('Precision')
plt.title('Precision Comparison')
plt.xticks(x, class_names)  # Use class names as labels
plt.legend()

plt.show()
```

```
# Plot Recall
plt.figure(figsize=(10, 8))

plt.bar(x - width / 2, recall_default, width, label='Default Model')
plt.bar(x + width / 2, recall_improved, width, label='Improved Model')

plt.xlabel('Class')
plt.ylabel('Recall')
plt.title('Recall Comparison')
plt.xticks(x, class_names)  # Use class names as labels
plt.legend()

plt.show()
```

```
10/10              1s 51ms/step
10/10              1s 65ms/step
```





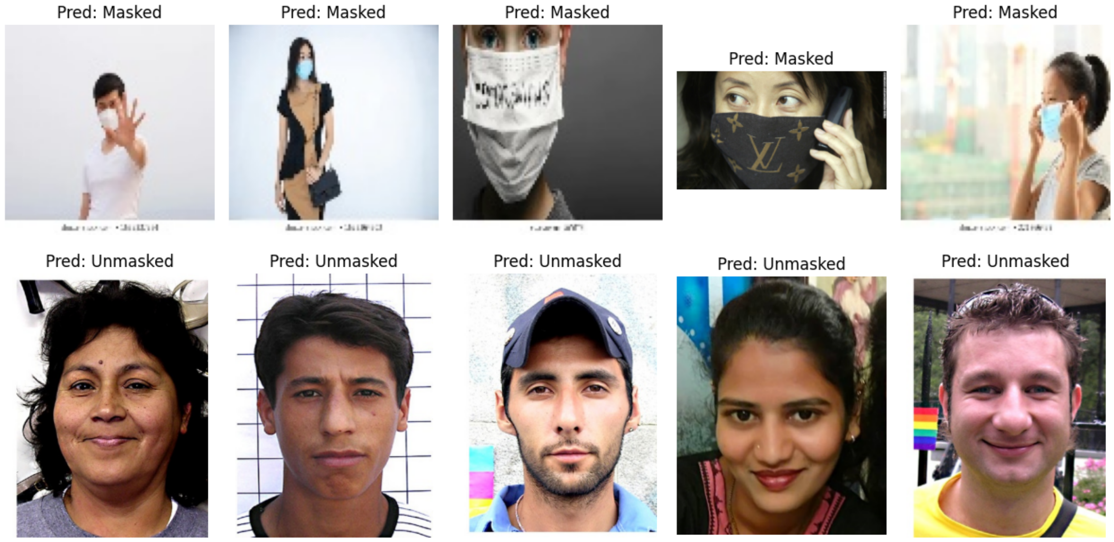
1.6 Task 6:- Visualize Predictions

Display 5 sample images from the test set predicted as "masked" and 5 predicted as "unmasked." Include the predicted labels for each image. [1 mark]

[47]:
```
# Importing required package
import random

# Predict labels for the test dataset
y_pred_probs = improved_model.predict(test_data)
y_pred_classes = np.argmax(y_pred_probs, axis=1)  # Convert probabilities to␣
↪class labels

# True labels
y_true_classes = test_data.classes
class_labels = {0: "Masked", 1: "Unmasked"}

# Get indices of masked and unmasked predictions
```

```python
masked_indices = np.where(y_pred_classes == 0)[0]  # Indices where the model
 ↪predicted "Masked"
unmasked_indices = np.where(y_pred_classes == 1)[0]  # Indices where the model
 ↪predicted "Unmasked"

# Randomly select 5 masked and 5 unmasked samples
masked_samples = random.sample(list(masked_indices), 5)
unmasked_samples = random.sample(list(unmasked_indices), 5)

# Combine selected indices
selected_indices = masked_samples + unmasked_samples

# Plot the selected images with their predicted labels
plt.figure(figsize=(12, 6))

for i, idx in enumerate(selected_indices):
    img_path = test_data.filepaths[idx]  # Get image path
    img = plt.imread(img_path)  # Read image

    # Get the predicted label
    predicted_label = class_labels[y_pred_classes[idx]]

    # Plot image
    plt.subplot(2, 5, i + 1)
    plt.imshow(img)
    plt.title(f"Pred: {predicted_label}")
    plt.axis("off")

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

```
10/10                 1s 57ms/step
```


Pred: Masked
Pred: Masked
Pred: Masked
Pred: Masked
Pred: Masked
Pred: Unmasked
Pred: Unmasked
Pred: Unmasked
Pred: Unmasked
Pred: Unmasked