bngttrxju

February 11, 2025

1 INTRODUCTION TO THE PROBLEM AND DATASET

1.1 PROBLEM STATEMENT

A leading fresh produce supply chain company is revolutionizing the agricultural sector by integrating `technology` and `automation`. It directly sources vegetables from `farmers` and delivers them to businesses within hours, ensuring `quality` and `efficiency`. To optimize operations, an `AI-driven` classification system is needed for accurate and automated sorting.

The challenge lies in differentiating between `onions`, `potatoes`, `tomatoes`, and `market scenes` under varying `lighting`, `angles`, and `backgrounds`. A `multiclass classifier` must be developed to correctly categorize these images while identifying unrelated ones as `noise`, reducing manual effort and improving accuracy.

1.2 OBJECTIVE OF THE PROJECT

This project aims to develop a `deep learning-based` image classifier that accurately identifies `onions`, `potatoes`, and `tomatoes` while filtering out `market scenes`. The model should handle variations in `lighting`, `angles`, and `backgrounds` to ensure `reliable performance`.

By automating `produce identification`, the system will minimize `manual sorting`, enhance `inventory management`, and optimize `distribution`. This will help reduce `errors`, decrease `waste`, and improve overall `supply chain efficiency`.

1.3 IMPORT ALL THE REQUIRED LIBRARIES

[9]:
```
# Standard library imports
import os
import glob
import zipfile
import pickle
import random

# Scientific computing and data handling
import numpy as np
import pandas as pd

# Deep learning and neural networks
```

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, regularizers,Sequential,applications

# Data visualization
import matplotlib.pyplot as plt
import seaborn as sns

# Machine learning metrics
import sklearn.metrics as metrics
from sklearn.metrics import classification_report, confusion_matrix,␣
 ↪accuracy_score, precision_score, recall_score

# mlflow logging
import mlflow
import mlflow.tensorflow

# File downloading
import gdown

# Image processing
import cv2
```

1.4 IMPORT DATASET

1.4.1 Download the zip file

[97]:
```
import gdown

# File ID from your URL
file_id = "1clZX-lV_MLxKHSyeyTheX5OCQtNCUcqT"
url = f"https://drive.google.com/uc?id={file_id}"

# Output file name
output = "downloaded_file.zip"

# Download the file
gdown.download(url, output, quiet=False)

print(f"File downloaded as {output}")
```

```
Downloading…
From (original): https://drive.google.com/uc?id=1clZX-
lV_MLxKHSyeyTheX5OCQtNCUcqT
From (redirected): https://drive.google.com/uc?id=1clZX-
lV_MLxKHSyeyTheX5OCQtNCUcqT&confirm=t&uuid=2f71366b-b3ce-4fa1-b2f1-3cb275fc79ed
To: c:\Users\saina\Desktop\DS_ML_AI\Scaler\Module_20_Computer_Vision\Case_studie
```

```
s\Ninja_cart\Multiclass_Vegetable_Image_Classification_Using_Deep_Learning\noteb
ook\downloaded_file.zip
100%|      | 275M/275M [00:54<00:00, 5.02MB/s]

File downloaded as downloaded_file.zip
```

1.4.2 Unzip the files

[98]:
```
# File to be unzipped
zip_file = "downloaded_file.zip"  # Update this if the file has a specific␣
 ↪extension like 'downloaded_file.zip'

# Output directory
output_dir = "unzipped_files"

# Check if the file exists and is a ZIP file
if zipfile.is_zipfile(zip_file):
    # Create the output directory if it doesn't exist
    os.makedirs(output_dir, exist_ok=True)

    # Unzipping the file
    with zipfile.ZipFile(zip_file, 'r') as zip_ref:
        zip_ref.extractall(output_dir)
    print(f"Files extracted to: {output_dir}")
else:
    print(f"{zip_file} is not a valid ZIP file.")
```

```
Files extracted to: unzipped_files
```

1.4.3 Dataset Description

`unzipped_files` as `ninjacart_data` folder. Ninjacart has two folders in it. 1. `train` 2. `test`

Both train and test have four sub folders 1. `indian market` - (images related to market) 2. `onion` - (images related to onions) 3. `potato` - (images related to potatoes) 4. `tomato` - (images related to tomatoes)

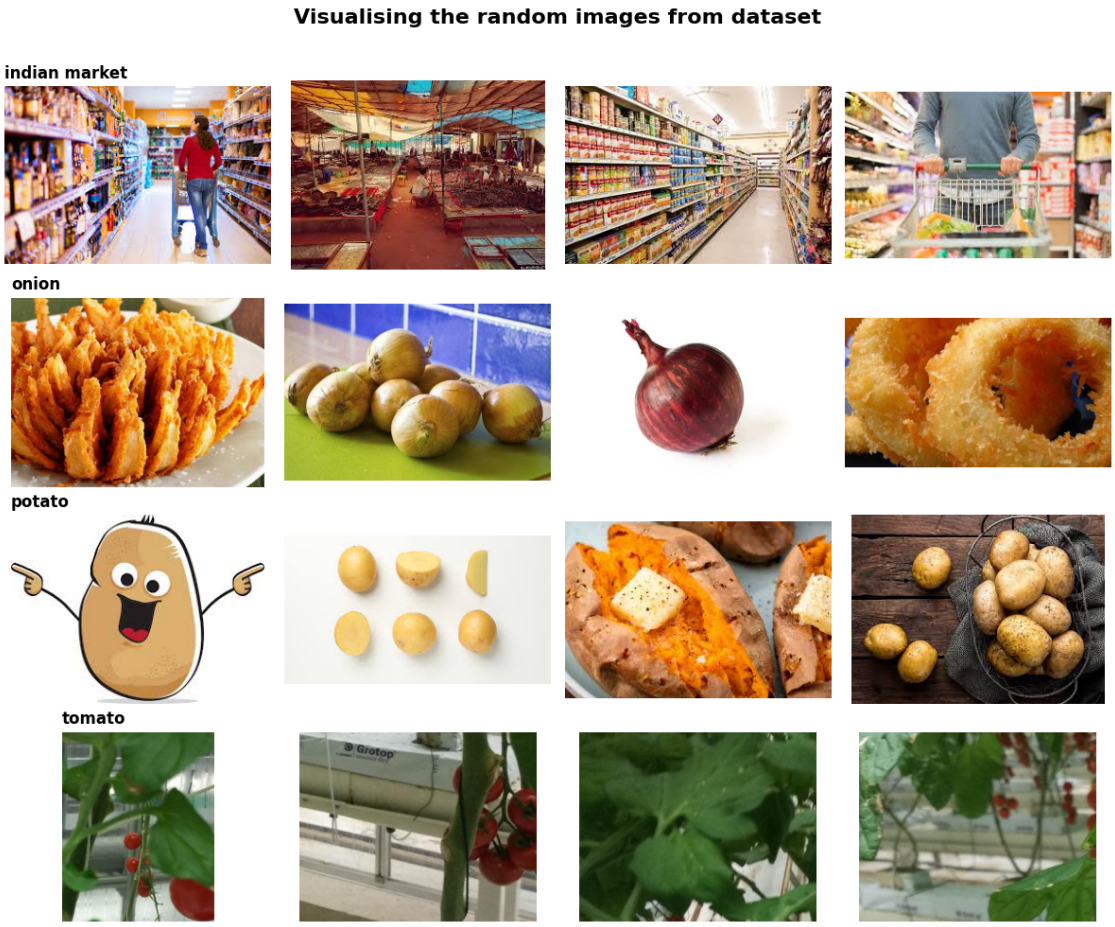
Observations done from downloaded folder: - **Problem Complexity**: Differentiating vegetables (onion, potato, tomato) from noisy "Indian market" scenes under varying lighting/angles is challenging.
- **Dataset Structure**:
- **Train/Test Split**: Train (3135 images), Test (351 images).
- **Class Imbalance**: "Potato" has the highest training samples (898), while "Indian market" (noise) has the fewest (599).
- **Noise Handling**: Market scenes act as a "noise" class, requiring the model to filter irrelevant images.

2 EXPLORATORY DATA ANALYSIS

2.1 VISUALISING RANDOM IMAGES FROM DATASET

[99]:
```
# Define the path to the training dataset
train_dir = r"unzipped_files\ninjacart_data\train"

# Set random seed for reproducibility
random.seed(42)

# Get class names (folder names) dynamically
class_names = [d for d in os.listdir(train_dir) if os.path.isdir(os.path.
 ↪join(train_dir, d))]

# Number of images to display per class
num_images = 4

# Set up the figure
fig, axes = plt.subplots(len(class_names), num_images, figsize=(12, 10))
fig.suptitle("Visualising the random images from dataset", fontsize=16,␣
 ↪fontweight="bold")

for i, class_name in enumerate(class_names):
    class_path = os.path.join(train_dir, class_name)

    # Get all image file names in the class folder
    image_files = [f for f in os.listdir(class_path) if f.lower().endswith(('.
 ↪png', '.jpg', '.jpeg'))]

    # Select random images
    selected_images = random.sample(image_files, min(num_images,␣
 ↪len(image_files)))

    for j, image_file in enumerate(selected_images):
        image_path = os.path.join(class_path, image_file)
        image = cv2.imread(image_path)
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)  # Convert BGR (OpenCV␣
 ↪default) to RGB

        # Display the image
        axes[i, j].imshow(image)
        axes[i, j].axis("off")

    # Set class name as row title
    axes[i, 0].set_title(class_name, fontsize=12, fontweight="bold", loc="left")

# Adjust layout and show the plot
```

```
plt.tight_layout(rect=[0, 0, 1, 0.96])  # Adjust layout to accommodate the␣
↪overall title
plt.show()
```



2.2 CREATING RANDOM IMAGE DICTIONARY AND COUNT DICTIONARY

[2]:
```
# Define dataset directories
dataset_dirs = {
    "train": r"unzipped_files/ninjacart_data/train",
    "test": r"unzipped_files/ninjacart_data/test"
}

# Store class directories separately for future use in dictionary format
class_dir = {"train": [], "test": []}

# Dictionaries to store image counts and random images
image_dict = {"train": {}, "test": {}}
```

```
count_dict = {"train": {}, "test": {}}

# Set random seed for reproducibility
random.seed(42)

# Process both train and test datasets
for dataset in ["train", "test"]:
    path = dataset_dirs[dataset]  # Get dataset path
    class_dir[dataset] = [d for d in os.listdir(path) if os.path.isdir(os.path.
 ↪join(path, d))]  # Get class names

    for cls in class_dir[dataset]:
        # Get all image paths in the class folder
        file_paths = glob.glob(os.path.join(path, cls, '*'))

        # Store the number of images per class
        count_dict[dataset][cls] = len(file_paths)

        # Select a random image from the class and store it
        if file_paths:  # Ensure there are images before choosing
            image_path = random.choice(file_paths)
            image_dict[dataset][cls] = tf.keras.utils.load_img(image_path)

print("class_dir :",class_dir)
print("image_dict :",image_dict)
print("count_dict :",count_dict)
```

```
class_dir : {'train': ['indian market', 'onion', 'potato', 'tomato'], 'test':
['indian market', 'onion', 'potato', 'tomato']}
image_dict : {'train': {'indian market': <PIL.JpegImagePlugin.JpegImageFile
image mode=RGB size=275x183 at 0x1BF02899880>, 'onion':
<PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=640x355 at
0x1BF02899FD0>, 'potato': <PIL.JpegImagePlugin.JpegImageFile image mode=RGB
size=259x194 at 0x1BF422D5790>, 'tomato': <PIL.PngImagePlugin.PngImageFile image
mode=RGB size=400x500 at 0x1BF422D6690>}, 'test': {'indian market':
<PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=259x194 at
0x1BF422D6930>, 'onion': <PIL.JpegImagePlugin.JpegImageFile image mode=RGB
size=470x470 at 0x1BF422D6990>, 'potato': <PIL.JpegImagePlugin.JpegImageFile
image mode=RGB size=286x176 at 0x1BF422D6AB0>, 'tomato':
<PIL.PngImagePlugin.PngImageFile image mode=RGB size=400x500 at 0x1BF422D6B40>}}
count_dict : {'train': {'indian market': 599, 'onion': 849, 'potato': 898,
'tomato': 789}, 'test': {'indian market': 81, 'onion': 83, 'potato': 81,
'tomato': 106}}
```

[101]:
```
# Plot the selected random images
num_classes = len(class_dir["train"])  # Number of classes (assuming same for␣
 ↪train & test)
```

```
fig, axes = plt.subplots(num_classes, 2, figsize=(8, 4 * num_classes))  # 2 columns (Train & Test)

# Main Title
fig.suptitle("Visualising the Selected Random Images", fontsize=16, fontweight="bold")

# Plot images
for i, cls in enumerate(class_dir["train"]):  # Iterate through train classes (assuming same for test)
    # Train Image
    if cls in image_dict["train"]:
        axes[i, 0].imshow(image_dict["train"][cls])
        axes[i, 0].set_title(f"Train - {cls}", fontsize=12, fontweight="bold")
        axes[i, 0].axis("off")

    # Test Image
    if cls in image_dict["test"]:
        axes[i, 1].imshow(image_dict["test"][cls])
        axes[i, 1].set_title(f"Test - {cls}", fontsize=12, fontweight="bold")
        axes[i, 1].axis("off")

# Adjust layout and show plot
plt.tight_layout(rect=[0, 0, 1, 0.96])
plt.show()
```

Visualising the Selected Random Images

Train - indian market

Test - indian market

Test - onion

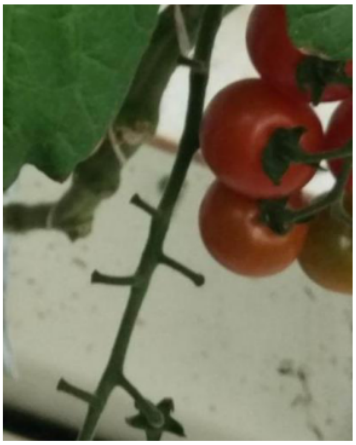
Train - onion

Train - potato

Test - potato

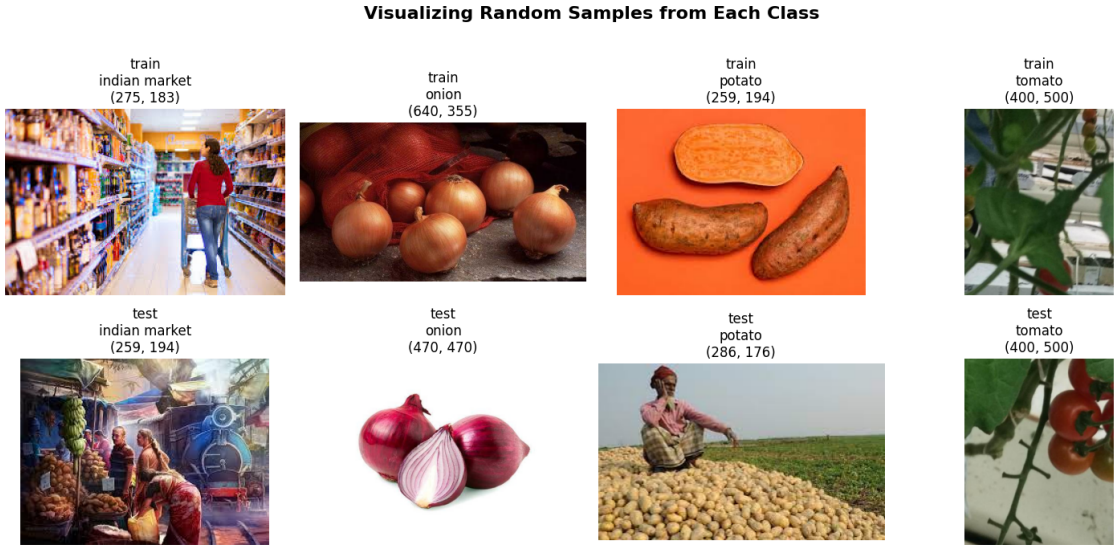
Train - tomato

Test - tomato



2.3 IMAGE DIMENSIONS

[3]:
```
# Define number of classes (assuming same for train & test)
num_classes = len(image_dict["train"])

# Create a figure with 2 rows (train & test), num_classes columns
fig, axes = plt.subplots(2, num_classes, figsize=(15, 7))

# Set overall title
fig.suptitle("Visualizing Random Samples from Each Class", fontsize=16,␣
 ↪fontweight="bold")

# Iterate over train and test datasets
for row, dataset in enumerate(["train", "test"]):  # Row 0: Train, Row 1: Test
    for col, cls in enumerate(image_dict[dataset]):  # Iterate through class␣
 ↪names
        img = image_dict[dataset][cls]  # Get the selected image

        # Plot image
        ax = axes[row, col]
        ax.imshow(img)
        ax.set_title(f'{dataset}\n{cls}\n{img.size}', fontsize=12)
        ax.axis("off")

    # Add dataset title on the leftmost side of each row
    axes[row, 0].set_ylabel(dataset.upper(), fontsize=14, fontweight="bold",␣
 ↪rotation=90, labelpad=20)

# Adjust layout and show plot
plt.tight_layout(rect=[0, 0, 1, 0.96])
plt.show()
```



Observation > Clearly, we can observe that every image of different dimensions.

We need to resize them to some square shape before applying the CNN models

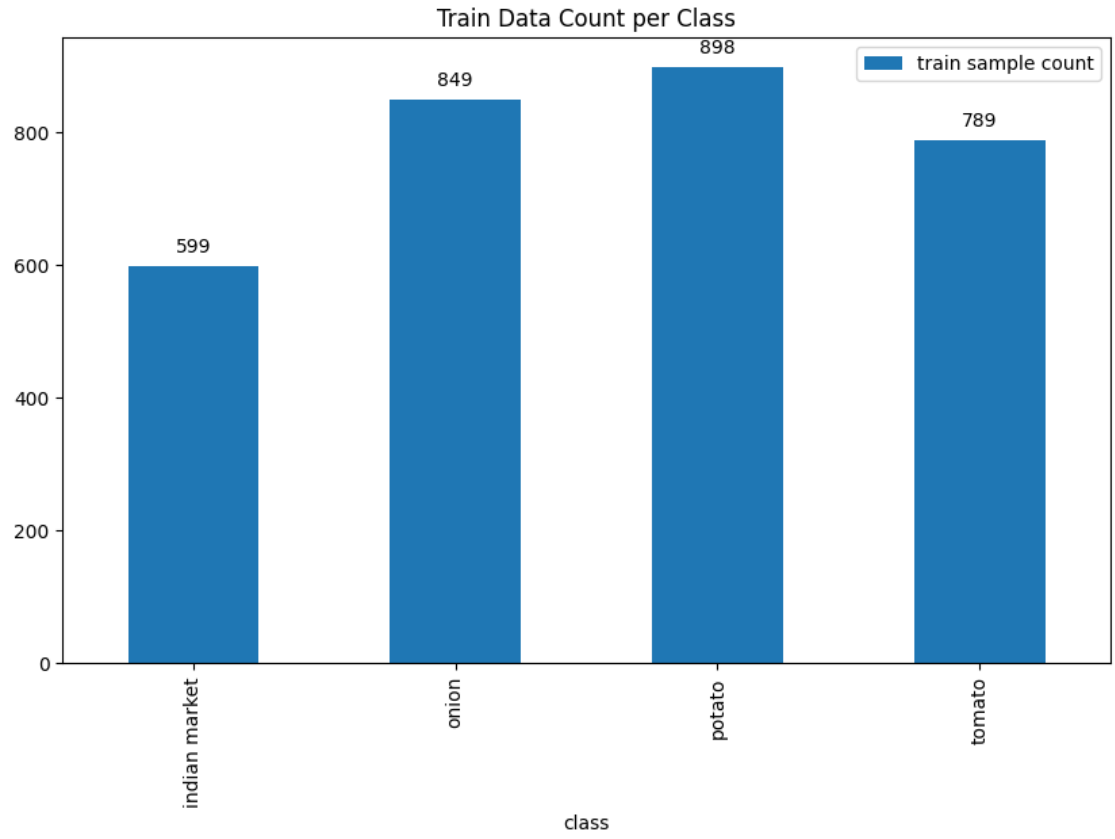
As we are doing vegetable classification, `indian market` images will be treated as `noise` images.

2.4 CLASS DISTRIBUTION ANALYSIS

[103]:
```
# Function to plot data distribution
def plot_data_distribution(dataset_type, count_dict):
    # Convert count_dict to DataFrame
    df_count = pd.DataFrame({
        "class": list(count_dict.keys()),     # Get class names
        "count": list(count_dict.values()),   # Get image counts
    })

    print(f"Count of {dataset_type} samples per class:\n", df_count, "\n")

    # Create bar plot
    ax = df_count.plot.bar(x='class', y='count', title=f"{dataset_type.
 ↪capitalize()} Data Count per Class",
                           label=f'{dataset_type} sample count', figsize=(10,␣
 ↪6))

    # Annotate bars with their counts
    for bar in ax.patches:
        ax.annotate(f'{bar.get_height()}', (bar.get_x() + bar.get_width() / 2.,␣
 ↪bar.get_height()),
```

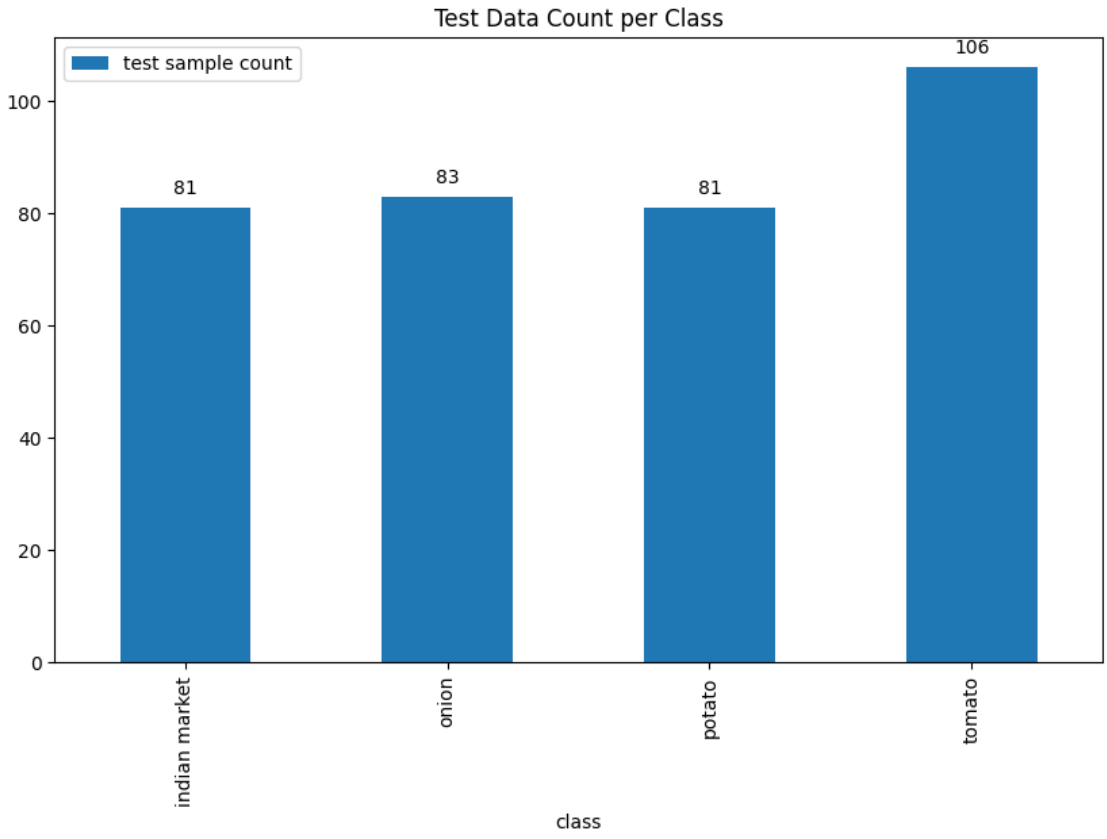
```
                    ha='center', va='center', xytext=(0, 10),␣
 ↪textcoords='offset points')

    plt.show()

# Plot Training Data Distribution
plot_data_distribution("train", count_dict["train"])

# Plot Test Data Distribution
plot_data_distribution("test", count_dict["test"])
```

```
Count of train samples per class:
            class  count
0  indian market    599
1          onion    849
2         potato    898
3         tomato    789
```




```
Count of test samples per class:
            class  count
```

```
0  indian market     81
1          onion     83
2         potato     81
3         tomato    106
```



Observations

- **Image Dimensions**:
- Original sizes varied widely (e.g., 259x194 for potatoes, 640x355 for onions).
- Resizing to **256x256** was critical for model compatibility.
- **Class Distribution**: - We have unbalanced Dataset in both training and testing dataset - **Train**: Potato (898) > Onion (849) > Tomato (789) > Indian market (599).
- **Test**: Tomato (106) > Onion (83) > Indian market (81) = Potato (81).
- Imbalance risks biased predictions (e.g., favoring “potato” or “tomato”).
- **Visual Insights**: Market scenes contained cluttered backgrounds, while vegetable images focused on single objects.

3 DATA PREPROCESSING

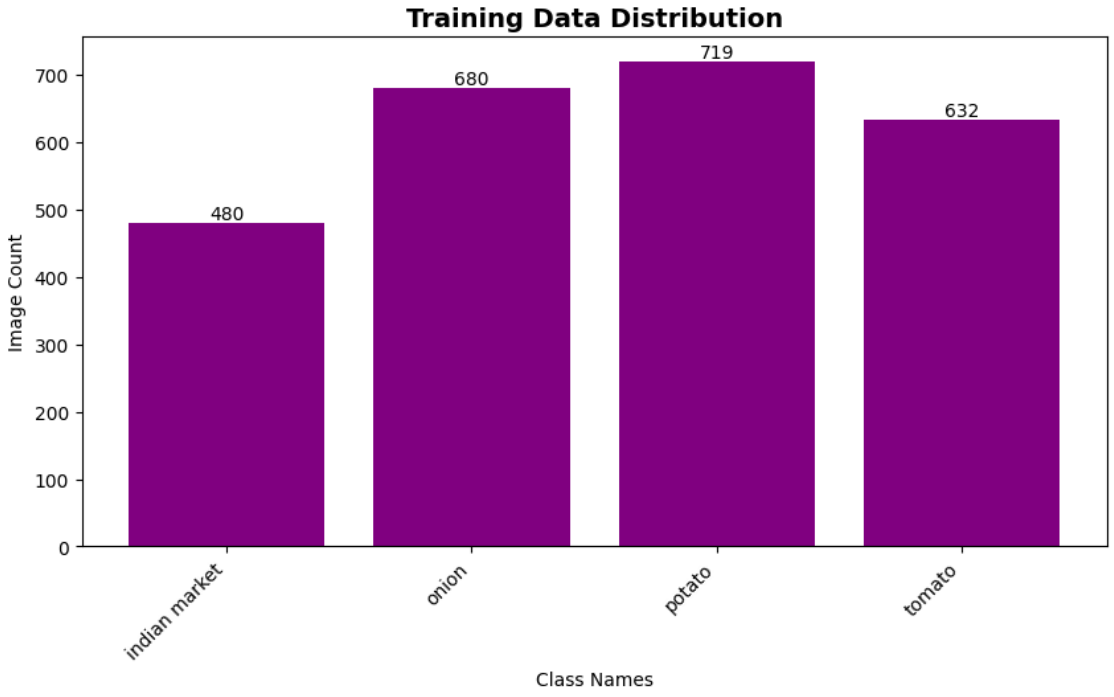
3.1 TRAIN, VALIDATION ,TEST SPLIT AND RESIZING TO (256,256)

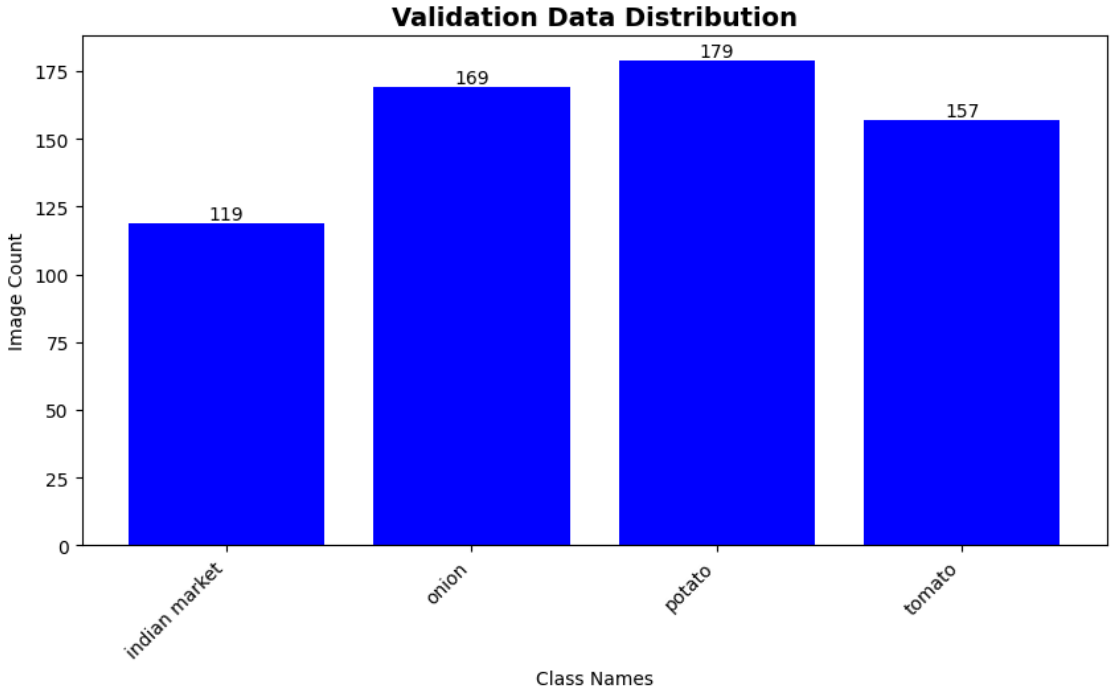
[3]:
```
def load_data(base_dir="unzipped_files/ninjacart_data", validation_split=0.2):
    # Ensure data folders exist
    assert os.path.exists(f"{base_dir}/train") and os.path.exists(f"{base_dir}/
 ↪test")

    print('\nLoading Data...\n')

    # Function to count images in each class
    def count_images(dataset_type):
        dataset_path = f"{base_dir}/{dataset_type}"
        class_dirs = [d for d in os.listdir(dataset_path) if os.path.isdir(os.
 ↪path.join(dataset_path, d))]
        class_counts = {cls: len(glob.glob(os.path.join(dataset_path, cls,␣
 ↪'*'))) for cls in class_dirs}
        return class_counts

    # Count images in train and test sets before splitting
    train_counts = count_images("train")
    test_counts = count_images("test")  # Added explicitly for clarity

    # Load the train dataset and apply validation split
    train_data = tf.keras.preprocessing.image_dataset_from_directory(
        f"{base_dir}/train",
        shuffle=True,
        label_mode='categorical',
        batch_size=32,
        validation_split=validation_split,
        subset='training',
        image_size = (256,256),
        seed=123
    )

    # Validation data (from the same directory, using validation_split)
    val_data = tf.keras.preprocessing.image_dataset_from_directory(
        f"{base_dir}/train",
        shuffle=True,
        label_mode='categorical',
        batch_size=32,
        validation_split=validation_split,
        subset='validation',
        image_size = (256,256),
        seed=123
    )
```

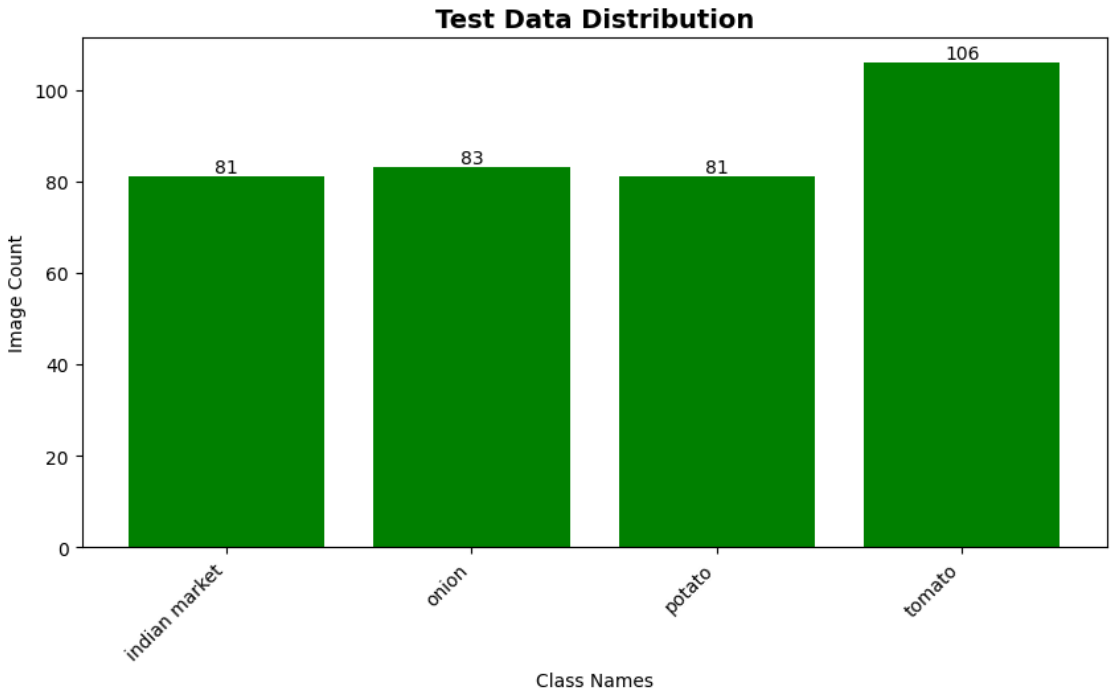
```
    # Load the test data (no split applied)
    test_data = tf.keras.preprocessing.image_dataset_from_directory(
        f"{base_dir}/test",
        shuffle=False,
        label_mode='categorical',
        batch_size=32,
        image_size = (256,256)
    )

    # Get class names
    class_names = train_data.class_names

    # Compute validation image counts using the split ratio
    val_counts = {cls: int(train_counts[cls] * validation_split) for cls in␣
↪class_names}

    # Adjust training counts after validation split
    train_counts = {cls: train_counts[cls] - val_counts[cls] for cls in␣
↪class_names}

    # Print dataset statistics
    print("Dataset Image Counts")

        # Convert counts into DataFrames
    df_train = pd.DataFrame(list(train_counts.items()), columns=["Class",␣
↪"Count"])
    df_val = pd.DataFrame(list(val_counts.items()), columns=["Class", "Count"])
    df_test = pd.DataFrame(list(test_counts.items()), columns=["Class",␣
↪"Count"])

    # Function to plot bar charts
    def plot_counts(df, title, color):
        plt.figure(figsize=(10, 5))
        bars = plt.bar(df["Class"], df["Count"], color=color)
        plt.xlabel("Class Names")
        plt.ylabel("Image Count")
        plt.title(title, fontsize=14, fontweight="bold")
        plt.xticks(rotation=45, ha="right")

        # Annotate bar heights
        for bar in bars:
            plt.text(bar.get_x() + bar.get_width() / 2, bar.get_height(),␣
↪f"{bar.get_height()}",
                     ha='center', va='bottom', fontsize=10)

        plt.show()
```

```
    # Plot training, validation, and test image distributions
    plot_counts(df_train, "Training Data Distribution", "purple")
    plot_counts(df_val, "Validation Data Distribution", "blue")
    plot_counts(df_test, "Test Data Distribution", "green")

    return train_data, val_data, test_data, class_names

# Call the function to load data
train_data, val_data, test_data, class_names = load_data()
```

```
Loading Data…

Found 3135 files belonging to 4 classes.
Using 2508 files for training.
Found 3135 files belonging to 4 classes.
Using 627 files for validation.
Found 351 files belonging to 4 classes.
Dataset Image Counts
```




Validation Data Distribution
Image Count
Class Names
indian market
onion
potato
tomato
119
169
179
157
0
25
50
75
100
125
150
175


Test Data Distribution
Image Count
Class Names
indian market
onion
potato
tomato
81
83
81
106
0
20
40
60
80
100

[5]:
```
# Function to extract one random image per class from a dataset
def get_random_images(dataset, class_names):
    class_images = {}  # Dictionary to store random images
    for images, labels in dataset:  # Iterate over batches
        for img, lbl in zip(images.numpy(), labels.numpy()):
            class_idx = np.argmax(lbl)  # Get class index
            class_name = class_names[class_idx]  # Map index to class name
            if class_name not in class_images:  # Store only one image per class
                class_images[class_name] = img
            if len(class_images) == len(class_names):  # Stop when all classes␣
 ↪are covered
                return class_images
    return class_images

# Fetch one random image per class from each dataset
train_images = get_random_images(train_data, class_names)
val_images = get_random_images(val_data, class_names)
test_images = get_random_images(test_data, class_names)

# Define number of classes
num_classes = len(class_names)

# Create a figure with 3 rows (Train, Validation, Test) and num_classes columns
fig, axes = plt.subplots(3, num_classes, figsize=(15, 10))

# Set overall title
fig.suptitle("Random Samples from Each Class with Image Sizes", fontsize=16,␣
 ↪fontweight="bold")

# Iterate over train, val, and test datasets
for row, (dataset_name, dataset_images) in enumerate(zip(["Train",␣
 ↪"Validation", "Test"], [train_images, val_images, test_images])):
    for col, cls in enumerate(class_names):  # Iterate through class names
        img = dataset_images.get(cls)  # Get the selected image
        if img is not None:
            ax = axes[row, col]
            ax.imshow(img.astype("uint8"))  # Convert image to displayable␣
 ↪format

            # Extract image size (height, width, channels)
            img_size = img.shape  # (height, width, channels)
            size_text = f"{img_size[0]}x{img_size[1]}"  # Format size

            # Set title with dataset name, class, and image size
            ax.set_title(f"{dataset_name}\n{cls}\nSize: {size_text}",␣
 ↪fontsize=10)
            ax.axis("off")
```

```
    # Add dataset title on the leftmost side of each row
    axes[row, 0].set_ylabel(dataset_name.upper(), fontsize=14,␣
 ↪fontweight="bold", rotation=90, labelpad=20)

# Adjust layout and show plot
plt.tight_layout(rect=[0, 0, 1, 0.96])
plt.show()
```

Observation > Dataset splitted into train, validation and test split

Image sizes are resized to (256,256,3)

3.2 RESIZING AND RESCALING

[4]:
```
def preprocess(train_data, val_data, test_data, target_height=256,␣
 ↪target_width=256):

    # Data Processing Stage with resizing and rescaling operations
    data_preprocess = keras.Sequential(
        name="data_preprocess",
        layers=[
```

```
            layers.Resizing(target_height, target_width),
            layers.Rescaling(1.0/255),
        ]
    )

    # Perform Data Processing on the train, val, test dataset
    train_ds = train_data.map(lambda x, y: (data_preprocess(x), y),␣
↪num_parallel_calls=tf.data.AUTOTUNE)
    val_ds = val_data.map(lambda x, y: (data_preprocess(x), y),␣
↪num_parallel_calls=tf.data.AUTOTUNE)
    test_ds = test_data.map(lambda x, y: (data_preprocess(x), y),␣
↪num_parallel_calls=tf.data.AUTOTUNE)

    return train_ds, val_ds, test_ds
```

[5]:
```
train_ds, val_ds, test_ds = preprocess(train_data, val_data, test_data)
```

Observation - Resizing & Normalization:
- All images resized to **256x256** to ensure uniformity.
- Pixel values normalized to **[0, 1]** to stabilize training.
- **Data Splitting**:
- Train: 80% (2508 images), Validation: 20% (627 images), Test: 351 images.
- Validation set used to tune hyperparameters and prevent overfitting. - **Augmentation can be done**:
- Techniques: Random rotation (±10%), translation (±10%), brightness/contrast adjustments.
- Goal: Improve generalization to unseen lighting/angles.

4 CUSTOMISED CNN MODELS FROM SCRATCH

4.1 SIMPLE CNN MODEL ARCHITECTURE WITH ONE CONV2D BLOCK

[108]:
```
def baseline(height=256, width=256, num_classes=4):
    hidden_size = 256

    model = keras.Sequential(
        name="simple_cnn1",
        layers=[
            layers.Conv2D(filters=16, kernel_size=3, padding="same",␣
↪activation='relu', input_shape=(height, width, 3)),
            layers.MaxPooling2D(pool_size=2),
            layers.Flatten(),
            layers.Dense(units=hidden_size, activation='relu'),
            layers.Dense(units=num_classes, activation='softmax')  # Final␣
↪output layer
        ]
    )
    return model
```

```
# Create and summarize the model
model = baseline()
model.summary()
```

```
C:\Users\saina\AppData\Roaming\Python\Python312\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: "simple_cnn1"

Layer (type)	Output Shape	Param #
conv2d_16 (Conv2D)	(None, 256, 256, 16)	448
max_pooling2d_13 (MaxPooling2D)	(None, 128, 128, 16)	0
flatten_1 (Flatten)	(None, 262144)	0
dense_8 (Dense)	(None, 256)	67,109,120
dense_9 (Dense)	(None, 4)	1,028

Total params: 67,110,596 (256.01 MB)

Trainable params: 67,110,596 (256.01 MB)

Non-trainable params: 0 (0.00 B)

4.1.1 Model Compilation and Training

[ ]:
```
def compile_train(model, train_ds, val_ds, epochs=10, log_dir = f"logs/{model.
↪name}",ckpt_path=f"checkpoints/{model.name}.weights.h5"):
    # Compile the model
    model.compile(
        optimizer='adam',
        loss='categorical_crossentropy',
        metrics=['accuracy', tf.keras.metrics.Precision(), tf.keras.metrics.
↪Recall()]
```

```
    )

    # Define Callbacks
    callbacks = [
        keras.callbacks.ModelCheckpoint(
            ckpt_path,
            save_weights_only=True,  # Saves only model weights
            monitor='val_accuracy',
            mode='max',
            save_best_only=True  # Saves only the best model based on␣
↪validation accuracy
        ),
        keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)  #␣
↪TensorBoard Logging
    ]

    # Train the model
    model_fit = model.fit(
        train_ds,
        validation_data=val_ds,
        epochs=epochs,  # Now customizable
        callbacks=callbacks
    )

    return model_fit
```

[111]:
```
model_fit = compile_train(model, train_ds, val_ds, epochs=10)
```

```
Epoch 1/10
79/79             151s 2s/step -
accuracy: 0.4589 - loss: 12.4082 - precision_3: 0.4692 - recall_3: 0.4351 -
val_accuracy: 0.7640 - val_loss: 0.7298 - val_precision_3: 0.7993 -
val_recall_3: 0.7049
Epoch 2/10
79/79             135s 2s/step -
accuracy: 0.8415 - loss: 0.4920 - precision_3: 0.8682 - recall_3: 0.7960 -
val_accuracy: 0.8230 - val_loss: 0.5489 - val_precision_3: 0.8468 -
val_recall_3: 0.8022
Epoch 3/10
79/79             142s 2s/step -
accuracy: 0.9299 - loss: 0.2492 - precision_3: 0.9384 - recall_3: 0.9186 -
val_accuracy: 0.8054 - val_loss: 0.5695 - val_precision_3: 0.8198 -
val_recall_3: 0.7911
Epoch 4/10
79/79             147s 2s/step -
accuracy: 0.9430 - loss: 0.2048 - precision_3: 0.9515 - recall_3: 0.9379 -
val_accuracy: 0.8246 - val_loss: 0.5674 - val_precision_3: 0.8361 -
```

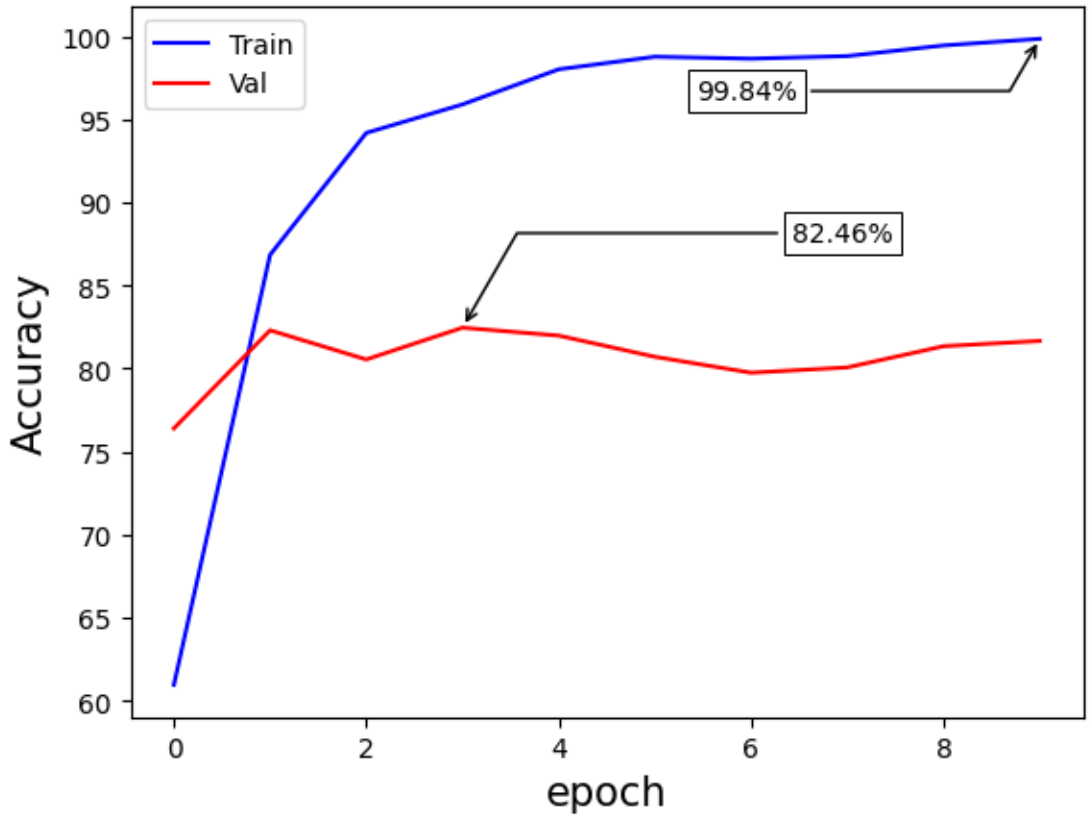
```
val_recall_3: 0.8134
Epoch 5/10
79/79            142s 2s/step -
accuracy: 0.9748 - loss: 0.1040 - precision_3: 0.9792 - recall_3: 0.9714 -
val_accuracy: 0.8198 - val_loss: 0.6250 - val_precision_3: 0.8271 -
val_recall_3: 0.8086
Epoch 6/10
79/79            143s 2s/step -
accuracy: 0.9813 - loss: 0.0732 - precision_3: 0.9845 - recall_3: 0.9805 -
val_accuracy: 0.8070 - val_loss: 0.7415 - val_precision_3: 0.8224 -
val_recall_3: 0.7974
Epoch 7/10
79/79            150s 2s/step -
accuracy: 0.9863 - loss: 0.0632 - precision_3: 0.9865 - recall_3: 0.9863 -
val_accuracy: 0.7974 - val_loss: 0.8137 - val_precision_3: 0.8013 -
val_recall_3: 0.7911
Epoch 8/10
79/79            155s 2s/step -
accuracy: 0.9874 - loss: 0.0477 - precision_3: 0.9900 - recall_3: 0.9863 -
val_accuracy: 0.8006 - val_loss: 0.6726 - val_precision_3: 0.8136 -
val_recall_3: 0.8006
Epoch 9/10
79/79            148s 2s/step -
accuracy: 0.9921 - loss: 0.0325 - precision_3: 0.9921 - recall_3: 0.9921 -
val_accuracy: 0.8134 - val_loss: 0.7373 - val_precision_3: 0.8177 -
val_recall_3: 0.8086
Epoch 10/10
79/79            154s 2s/step -
accuracy: 0.9976 - loss: 0.0180 - precision_3: 0.9976 - recall_3: 0.9976 -
val_accuracy: 0.8166 - val_loss: 0.7328 - val_precision_3: 0.8231 -
val_recall_3: 0.8086
```

4.1.2 Plot Training and Validation Accuracy wrt Epoch

[7]:
```python
# helper function to annotate maximum values in the plots
def annot_max(x,y, xytext=(0.94,0.96), ax=None, only_y=True):
    xmax = x[np.argmax(y)]
    ymax = max(y)
    if only_y:
        text = "{:.2f}%".format(ymax)
    else:
        text= "x={:.2f}, y={:.2f}%".format(xmax, ymax)
    if not ax:
        ax=plt.gca()
    bbox_props = dict(boxstyle="square,pad=0.3", fc="w", ec="k", lw=0.72)
    arrowprops=dict(arrowstyle="->",connectionstyle="angle,angleA=0,angleB=60")
    kw = dict(xycoords='data',textcoords="axes fraction",
```

```
                arrowprops=arrowprops, bbox=bbox_props, ha="right", va="top")
    ax.annotate(text, xy=(xmax, ymax), xytext=xytext, **kw)

def plot_accuracy(model_fit):
    #accuracy graph
    x = range(0,len(model_fit.history['accuracy']))
    y_train = [acc * 100 for acc in model_fit.history['accuracy']]
    y_val = [acc * 100 for acc in model_fit.history['val_accuracy']]

    plt.plot(x, y_train, label='Train', color='b')
    annot_max(x, y_train, xytext=(0.7,0.9))
    plt.plot(x, y_val, label='Val', color='r')
    annot_max(x, y_val, xytext=(0.8,0.7))
    plt.ylabel('Accuracy', fontsize=15)
    plt.xlabel('epoch', fontsize=15)
    plt.legend()
    plt.show()
```

[113]:
```
plot_accuracy(model_fit)
```



4.1.3 Tensorboard Visualisation

[114]:
```
%load_ext tensorboard
```

```
The tensorboard extension is already loaded. To reload it, use:
  %reload_ext tensorboard
```

[116]:
```
%tensorboard --logdir logs/simple_cnn1/
```

```
Launching TensorBoard...
```

4.1.4 Model Evaluation, Mlflow Logging and Printing Metrics

[8]:
```
def evaluate_model(model, train_ds, val_ds, test_ds, class_names, run_name,␣
↪ckpt_path):
    mlflow.set_experiment("Vegetable_image_classification")

    with mlflow.start_run(run_name=run_name):
        model.load_weights(ckpt_path)
        mlflow.tensorflow.log_model(model, "model")
        mlflow.log_param("checkpoint_path", ckpt_path)

        def get_metrics(ds, dataset_name):
            y_true, y_pred = [], []
            for x, y in ds:
                y_true.extend(tf.argmax(y, axis=1).numpy().tolist())
                preds = model(x, training=False)
                y_pred.extend(tf.argmax(preds, axis=1).numpy().tolist())

            acc = accuracy_score(y_true, y_pred) * 100
            prec = precision_score(y_true, y_pred, average="macro") * 100
            rec = recall_score(y_true, y_pred, average="macro") * 100

            mlflow.log_metric(f"{dataset_name}_accuracy", acc)
            mlflow.log_metric(f"{dataset_name}_precision", prec)
            mlflow.log_metric(f"{dataset_name}_recall", rec)

            print(f"\n{dataset_name} Metrics:")
            print(f" Accuracy: {acc:.2f}%")
            print(f" Precision: {prec:.2f}%")
            print(f" Recall: {rec:.2f}%")

            return y_true, y_pred

        train_y_true, train_y_pred = get_metrics(train_ds, "train")
        val_y_true, val_y_pred = get_metrics(val_ds, "val")
        test_y_true, test_y_pred = get_metrics(test_ds, "test")
```

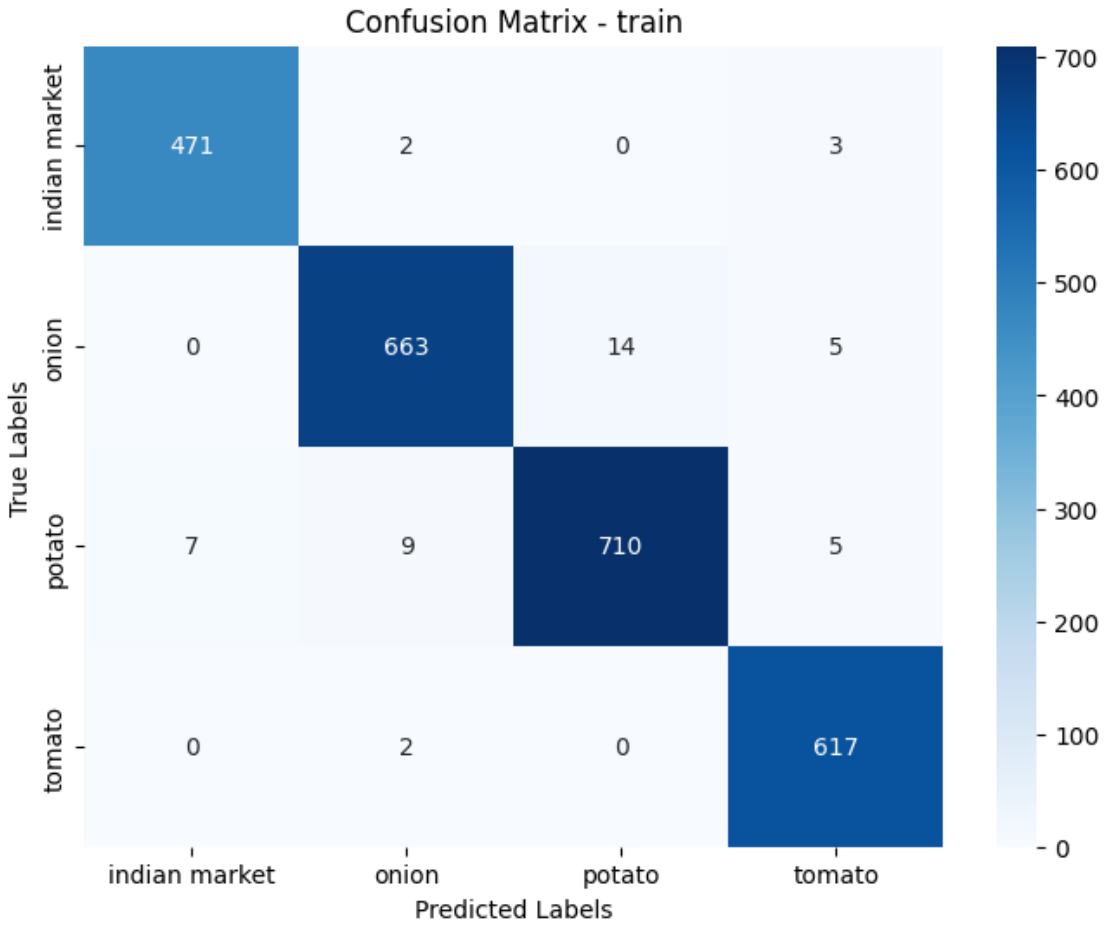
```
        def plot_confusion_matrix(y_true, y_pred, dataset_name):
            cm = confusion_matrix(y_true, y_pred)
            plt.figure(figsize=(8, 6))
            sns.heatmap(cm, annot=True, fmt='g', cmap="Blues",␣
↪xticklabels=class_names, yticklabels=class_names)
            plt.xlabel('Predicted Labels')
            plt.ylabel('True Labels')
            plt.title(f'Confusion Matrix - {dataset_name}')
            plt.savefig(f"conf_matrix_{dataset_name}.png")
            mlflow.log_artifact(f"conf_matrix_{dataset_name}.png")
            plt.show()
            plt.close()

        plot_confusion_matrix(train_y_true, train_y_pred, "train")
        plot_confusion_matrix(val_y_true, val_y_pred, "val")
        plot_confusion_matrix(test_y_true, test_y_pred, "test")

        def log_classification_report(y_true, y_pred, dataset_name):
            report = classification_report(y_true, y_pred,␣
↪target_names=class_names, output_dict=True)
            mlflow.log_dict(report, f"classification_report_{dataset_name}.
↪json")
            print(f"\nClassification Report - {dataset_name}")
            print(classification_report(y_true, y_pred,␣
↪target_names=class_names))

        log_classification_report(train_y_true, train_y_pred, "train")
        log_classification_report(val_y_true, val_y_pred, "val")
        log_classification_report(test_y_true, test_y_pred, "test")

        print("\nClass-wise Accuracy:")
        for dataset_name, y_true, y_pred in zip(["train", "val", "test"],␣
↪[train_y_true, val_y_true, test_y_true], [train_y_pred, val_y_pred,␣
↪test_y_pred]):
            print(f"\n{dataset_name.capitalize()} Class-wise Accuracy:")
            for i, class_name in enumerate(class_names):
                total_count = y_true.count(i)
                correct_count = sum(1 for true, pred in zip(y_true, y_pred) if␣
↪true == pred and true == i)
                class_acc = (correct_count / total_count) * 100 if total_count␣
↪> 0 else 0
                mlflow.
↪log_metric(f"{dataset_name}_class_accuracy_{class_name}", class_acc)
                print(f" {class_name}: {class_acc:.2f}% ({correct_count}/
↪{total_count})")
```

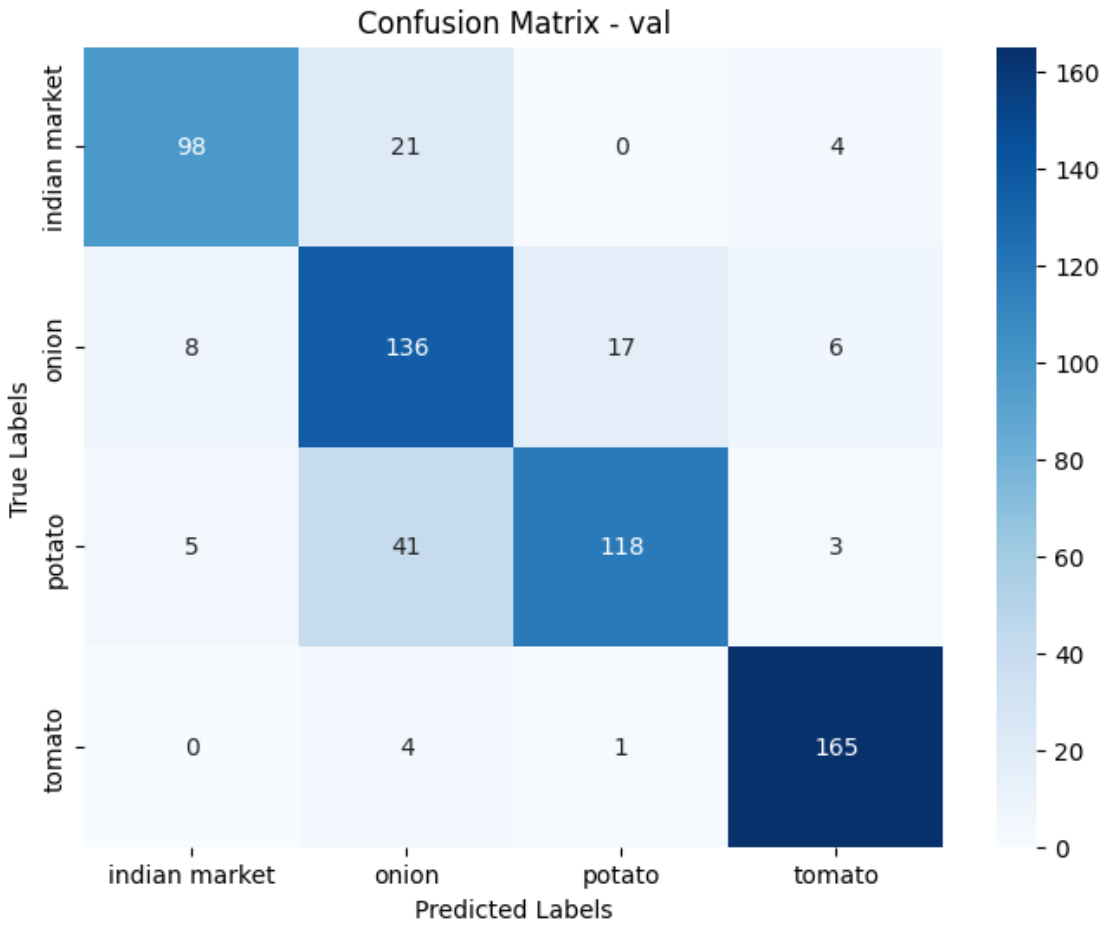
```
        print("\nRandom Test Predictions:")
        samples_per_class = 2
        fig, axes = plt.subplots(len(class_names), samples_per_class,␣
↪figsize=(10, 2 * len(class_names)))

        test_images = list(test_ds.unbatch().as_numpy_iterator())
        grouped_by_class = {i: [] for i in range(len(class_names))}

        for img, label in test_images:
            class_idx = np.argmax(label)
            grouped_by_class[class_idx].append((img, class_idx))

        for row, cls_idx in enumerate(grouped_by_class):
            selected_samples = random.sample(grouped_by_class[cls_idx],␣
↪min(samples_per_class, len(grouped_by_class[cls_idx])))

            for col, (img, true_label) in enumerate(selected_samples):
                img_input = np.expand_dims(img, axis=0)
                pred = model(img_input, training=False)
                predicted_label = np.argmax(pred)

                ax = axes[row, col]

                # **Fix: Rescale image to original format (0-255) and convert␣
↪to uint8**
                img = (img * 255).astype("uint8")

                ax.imshow(img)
                ax.set_title(f"True: {class_names[true_label]}\nPred:␣
↪{class_names[predicted_label]}")
                ax.axis("off")

        plt.tight_layout()
        plt.savefig("random_test_predictions.png")
        mlflow.log_artifact("random_test_predictions.png")
        plt.show()

        mlflow.end_run()
```

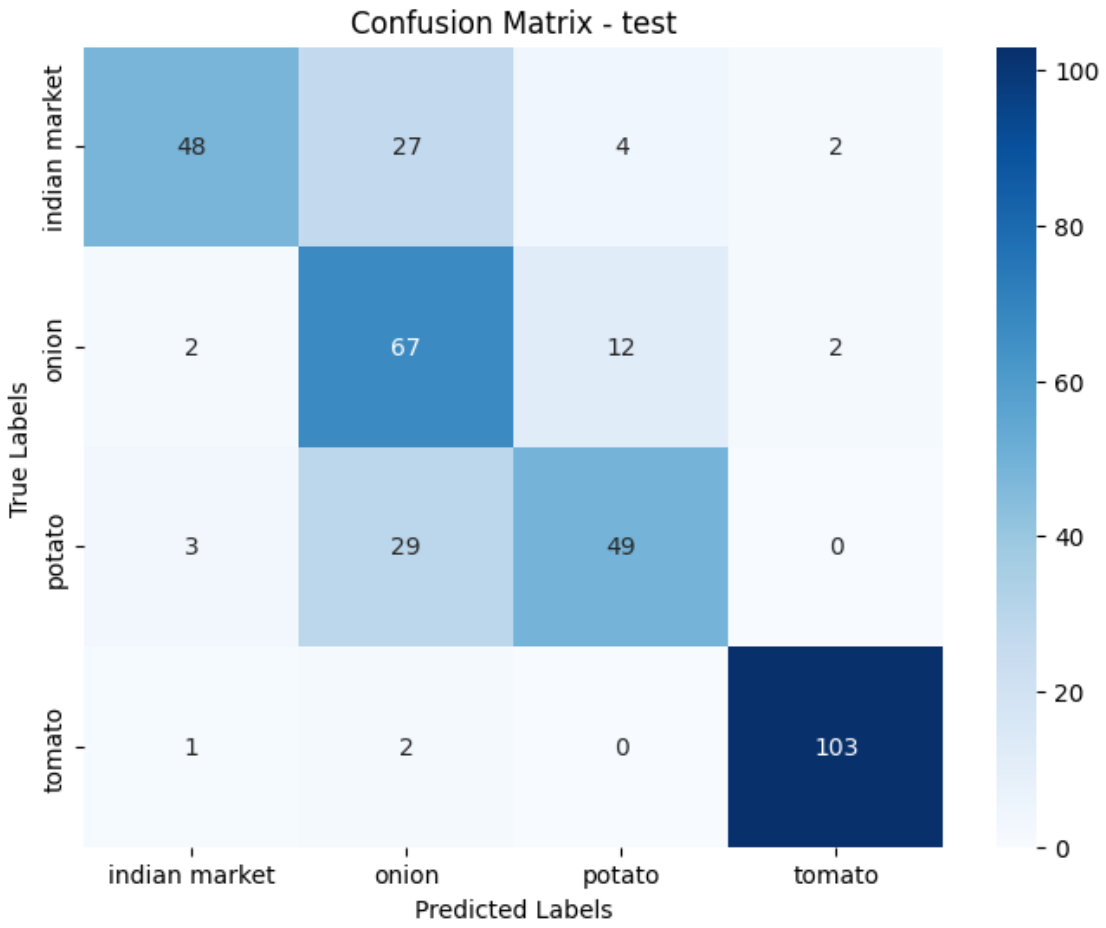
[119]:
```
evaluate_model(model, train_ds, val_ds, test_ds, class_names,run_name=f"{model.
↪name}",ckpt_path= f"checkpoints/{model.name}.weights.h5")
```

```
2025/02/08 16:22:50 WARNING mlflow.tensorflow: You are saving a TensorFlow Core
model or Keras model without a signature. Inference with
mlflow.pyfunc.spark_udf() will not work unless the model's pyfunc representation
accepts pandas DataFrames as inference inputs.
2025/02/08 16:23:19 WARNING mlflow.models.model: Model logged without a
```

```
signature and input example. Please set `input_example` parameter when logging
the model to auto infer the model signature.


train Metrics:
 Accuracy: 98.13%
 Precision: 98.15%
 Recall: 98.24%

val Metrics:
 Accuracy: 82.46%
 Precision: 83.77%
 Recall: 82.21%

test Metrics:
 Accuracy: 76.07%
 Precision: 78.53%
 Recall: 74.41%
```




Confusion Matrix - val
True Labels
Predicted Labels
indian market
onion
potato
tomato
98
21
0
4
8
136
17
6
5
41
118
3
0
4
1
165
160
140
120
100
80
60
40
20
0




```
Classification Report - train
              precision    recall  f1-score   support

indian market       0.99      0.99      0.99       476
        onion       0.98      0.97      0.98       682
       potato       0.98      0.97      0.98       731
       tomato       0.98      1.00      0.99       619

     accuracy                           0.98      2508
    macro avg       0.98      0.98      0.98      2508
 weighted avg       0.98      0.98      0.98      2508


Classification Report - val
              precision    recall  f1-score   support

indian market       0.88      0.80      0.84       123
```

```
        onion       0.67      0.81      0.74       167
       potato       0.87      0.71      0.78       167
       tomato       0.93      0.97      0.95       170

     accuracy                           0.82       627
    macro avg       0.84      0.82      0.83       627
 weighted avg       0.83      0.82      0.83       627


Classification Report - test
               precision    recall  f1-score   support

indian market       0.89      0.59      0.71        81
        onion       0.54      0.81      0.64        83
       potato       0.75      0.60      0.67        81
       tomato       0.96      0.97      0.97       106

     accuracy                           0.76       351
    macro avg       0.79      0.74      0.75       351
 weighted avg       0.80      0.76      0.76       351


Class-wise Accuracy:

Train Class-wise Accuracy:
 indian market: 98.95% (471/476)
 onion: 97.21% (663/682)
 potato: 97.13% (710/731)
 tomato: 99.68% (617/619)

Val Class-wise Accuracy:
 indian market: 79.67% (98/123)
 onion: 81.44% (136/167)
 potato: 70.66% (118/167)
 tomato: 97.06% (165/170)

Test Class-wise Accuracy:
 indian market: 59.26% (48/81)
 onion: 80.72% (67/83)
 potato: 60.49% (49/81)
 tomato: 97.17% (103/106)

Random Test Predictions:
```

True: indian market
Pred: onion

True: indian market
Pred: onion

True: onion
Pred: onion

True: onion
Pred: onion

True: potato
Pred: potato

True: potato
Pred: potato

True: tomato
Pred: tomato

True: tomato
Pred: tomato

4.1.5 Save the model

[120]:
```
model.save("saved_models/simple_cnn1_model.keras")
```

4.2 COMPLEX CNN MODEL FROM SCRATCH

[181]:
```
def complex_cnn(height=256, width=256):
    num_classes = 4

    model = keras.Sequential(
        name="complex_cnn",
        layers=[
            # Block 1
            layers.Conv2D(filters=32, kernel_size=3, padding="same",␣
 ↪input_shape=(height, width, 3),
                          kernel_regularizer=regularizers.l2(5e-4)),
            layers.Activation("relu"),
            layers.BatchNormalization(),
            layers.MaxPooling2D(),

            # Block 2
            layers.Conv2D(filters=64, kernel_size=3, padding="same",
                          kernel_regularizer=regularizers.l2(5e-4)),
            layers.Activation("relu"),
            layers.BatchNormalization(),
            layers.MaxPooling2D(),

            # Block 3
            layers.Conv2D(filters=128, kernel_size=3, padding="same",
                          kernel_regularizer=regularizers.l2(5e-4)),
            layers.Activation("relu"),
            layers.BatchNormalization(),
            layers.MaxPooling2D(),

            # Block 4
            layers.Conv2D(filters=256, kernel_size=3, padding="same",
                          kernel_regularizer=regularizers.l2(5e-4)),
            layers.Activation("relu"),
            layers.BatchNormalization(),
            layers.MaxPooling2D(),

            # Block 5
            layers.Conv2D(filters=512, kernel_size=3, padding="same",
                          kernel_regularizer=regularizers.l2(5e-4)),
            layers.Activation("relu"),
            layers.BatchNormalization(),
            layers.MaxPooling2D(),
```

```
            # **Global Average Pooling Instead of Flatten**
            layers.GlobalAveragePooling2D(),

            # Fully Connected Layers
            layers.Dense(units=512, kernel_regularizer=regularizers.l2(5e-4)),
            layers.Activation("relu"),
            layers.BatchNormalization(),
            layers.Dropout(0.3),  # Keeping only this dropout

            # Fully Connected Layers
            layers.Dense(units=256, kernel_regularizer=regularizers.l2(5e-4)),
            layers.Activation("relu"),
            layers.BatchNormalization(),

            layers.Dense(units=num_classes, activation='softmax')
        ]
    )
    return model
```

[182]:
```
model = complex_cnn()
model.summary()
```

```
C:\Users\saina\AppData\Roaming\Python\Python312\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: "complex_cnn"

Layer (type)	Output Shape	Param #
conv2d_62 (Conv2D)	(None, 256, 256, 32)	896
activation_49 (Activation)	(None, 256, 256, 32)	0
batch_normalization_49 (BatchNormalization)	(None, 256, 256, 32)	128
max_pooling2d_51 (MaxPooling2D)	(None, 128, 128, 32)	0
conv2d_63 (Conv2D)	(None, 128, 128, 64)	18,496
activation_50 (Activation)	(None, 128, 128, 64)	0

```
batch_normalization_50          (None, 128, 128, 64)          256
(BatchNormalization)

max_pooling2d_52 (MaxPooling2D) (None, 64, 64, 64)            0

conv2d_64 (Conv2D)              (None, 64, 64, 128)           73,856

activation_51 (Activation)      (None, 64, 64, 128)           0

batch_normalization_51          (None, 64, 64, 128)           512
(BatchNormalization)

max_pooling2d_53 (MaxPooling2D) (None, 32, 32, 128)           0

conv2d_65 (Conv2D)              (None, 32, 32, 256)           295,168

activation_52 (Activation)      (None, 32, 32, 256)           0

batch_normalization_52          (None, 32, 32, 256)           1,024
(BatchNormalization)

max_pooling2d_54 (MaxPooling2D) (None, 16, 16, 256)           0

conv2d_66 (Conv2D)              (None, 16, 16, 512)           1,180,160

activation_53 (Activation)      (None, 16, 16, 512)           0

batch_normalization_53          (None, 16, 16, 512)           2,048
(BatchNormalization)

max_pooling2d_55 (MaxPooling2D) (None, 8, 8, 512)             0

global_average_pooling2d_12     (None, 512)                   0
(GlobalAveragePooling2D)

dense_29 (Dense)                (None, 512)                   262,656

activation_54 (Activation)      (None, 512)                   0

batch_normalization_54          (None, 512)                   2,048
(BatchNormalization)

dropout_8 (Dropout)             (None, 512)                   0

dense_30 (Dense)                (None, 256)                   131,328

activation_55 (Activation)      (None, 256)                   0
```

```
batch_normalization_55          (None, 256)                 1,024
(BatchNormalization)

dense_31 (Dense)                (None, 4)                   1,028
```

```
Total params: 1,970,628 (7.52 MB)
```

```
Trainable params: 1,967,108 (7.50 MB)
```

```
Non-trainable params: 3,520 (13.75 KB)
```

4.2.1 Model Compilation and Training including all Callback Features

[20]:
```python
def compile_train(model, train_ds, val_ds, epochs=10, log_dir= f"logs/{model.name}", ckpt_path = f"checkpoints/{model.name}.weights.h5"):
    # Compile the model
    model.compile(
        optimizer=tf.keras.optimizers.Adam(),  # Default Adam optimizer
        loss="categorical_crossentropy",
        metrics=["accuracy", tf.keras.metrics.Precision(), tf.keras.metrics.Recall()]
    )

    # Define Callbacks
    callbacks = [
        # Reduce learning rate if validation loss stops improving
        keras.callbacks.ReduceLROnPlateau(
            monitor="val_loss", factor=0.3, patience=5, min_lr=1e-5
        ),
        # Save only the best model based on validation accuracy
        keras.callbacks.ModelCheckpoint(
            ckpt_path,
            save_weights_only=True,
            monitor="val_accuracy",
            mode="max",
            save_best_only=True
        ),
        # Stop training early if validation loss does not improve
        keras.callbacks.EarlyStopping(
            monitor="val_loss",
            patience=10,
            min_delta=0.001,
            mode="min",
```

```
            restore_best_weights=True  # Restore the best model
        ),
        # TensorBoard logging
        keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)
    ]

    # Train the model
    model_fit = model.fit(
        train_ds,
        validation_data=val_ds,
        epochs=epochs,
        callbacks=callbacks
    )

    return model_fit
```

[184]:
```
model_fit = compile_train(model, train_ds, val_ds, epochs=100)
```

```
Epoch 1/100
79/79             249s 3s/step -
accuracy: 0.6586 - loss: 1.7183 - precision_10: 0.6893 - recall_10: 0.6247 -
val_accuracy: 0.2663 - val_loss: 3.2225 - val_precision_10: 0.2663 -
val_recall_10: 0.2663 - learning_rate: 0.0010
Epoch 2/100
79/79             205s 3s/step -
accuracy: 0.7908 - loss: 1.3226 - precision_10: 0.8190 - recall_10: 0.7581 -
val_accuracy: 0.2679 - val_loss: 3.2999 - val_precision_10: 0.2679 -
val_recall_10: 0.2679 - learning_rate: 0.0010
Epoch 3/100
79/79             165s 2s/step -
accuracy: 0.8076 - loss: 1.2507 - precision_10: 0.8275 - recall_10: 0.7805 -
val_accuracy: 0.5167 - val_loss: 2.2550 - val_precision_10: 0.5429 -
val_recall_10: 0.5152 - learning_rate: 0.0010
Epoch 4/100
79/79             164s 2s/step -
accuracy: 0.8120 - loss: 1.2325 - precision_10: 0.8409 - recall_10: 0.7964 -
val_accuracy: 0.3748 - val_loss: 3.4611 - val_precision_10: 0.3785 -
val_recall_10: 0.3652 - learning_rate: 0.0010
Epoch 5/100
79/79             163s 2s/step -
accuracy: 0.8211 - loss: 1.1355 - precision_10: 0.8335 - recall_10: 0.7948 -
val_accuracy: 0.5678 - val_loss: 2.1000 - val_precision_10: 0.5892 -
val_recall_10: 0.5582 - learning_rate: 0.0010
Epoch 6/100
79/79             164s 2s/step -
accuracy: 0.8475 - loss: 1.0540 - precision_10: 0.8620 - recall_10: 0.8404 -
val_accuracy: 0.6140 - val_loss: 1.7427 - val_precision_10: 0.6221 -
```

```
val_recall_10: 0.6013 - learning_rate: 0.0010
Epoch 7/100
79/79            163s 2s/step -
accuracy: 0.8322 - loss: 1.0228 - precision_10: 0.8486 - recall_10: 0.8195 -
val_accuracy: 0.7400 - val_loss: 1.4993 - val_precision_10: 0.7488 -
val_recall_10: 0.7321 - learning_rate: 0.0010
Epoch 8/100
79/79            162s 2s/step -
accuracy: 0.8556 - loss: 0.9486 - precision_10: 0.8718 - recall_10: 0.8372 -
val_accuracy: 0.7624 - val_loss: 1.4855 - val_precision_10: 0.7725 -
val_recall_10: 0.7528 - learning_rate: 0.0010
Epoch 9/100
79/79            163s 2s/step -
accuracy: 0.8520 - loss: 0.9232 - precision_10: 0.8613 - recall_10: 0.8407 -
val_accuracy: 0.8182 - val_loss: 1.0101 - val_precision_10: 0.8322 -
val_recall_10: 0.8070 - learning_rate: 0.0010
Epoch 10/100
79/79            166s 2s/step -
accuracy: 0.8605 - loss: 0.9081 - precision_10: 0.8704 - recall_10: 0.8416 -
val_accuracy: 0.7815 - val_loss: 1.0830 - val_precision_10: 0.7864 -
val_recall_10: 0.7751 - learning_rate: 0.0010
Epoch 11/100
79/79            169s 2s/step -
accuracy: 0.8667 - loss: 0.8249 - precision_10: 0.8843 - recall_10: 0.8542 -
val_accuracy: 0.8054 - val_loss: 0.9794 - val_precision_10: 0.8176 -
val_recall_10: 0.7863 - learning_rate: 0.0010
Epoch 12/100
79/79            167s 2s/step -
accuracy: 0.8509 - loss: 0.8021 - precision_10: 0.8647 - recall_10: 0.8395 -
val_accuracy: 0.8389 - val_loss: 0.9606 - val_precision_10: 0.8465 -
val_recall_10: 0.8357 - learning_rate: 0.0010
Epoch 13/100
79/79            201s 3s/step -
accuracy: 0.8542 - loss: 0.7675 - precision_10: 0.8686 - recall_10: 0.8433 -
val_accuracy: 0.7432 - val_loss: 1.5208 - val_precision_10: 0.7528 -
val_recall_10: 0.7432 - learning_rate: 0.0010
Epoch 14/100
79/79            225s 3s/step -
accuracy: 0.8510 - loss: 0.7848 - precision_10: 0.8651 - recall_10: 0.8422 -
val_accuracy: 0.8038 - val_loss: 0.9026 - val_precision_10: 0.8141 -
val_recall_10: 0.7895 - learning_rate: 0.0010
Epoch 15/100
79/79            223s 3s/step -
accuracy: 0.8924 - loss: 0.6650 - precision_10: 0.8973 - recall_10: 0.8820 -
val_accuracy: 0.8405 - val_loss: 0.9559 - val_precision_10: 0.8493 -
val_recall_10: 0.8357 - learning_rate: 0.0010
Epoch 16/100
79/79            215s 3s/step -
```

```
accuracy: 0.8828 - loss: 0.6449 - precision_10: 0.8933 - recall_10: 0.8769 -
val_accuracy: 0.8612 - val_loss: 0.6929 - val_precision_10: 0.8686 -
val_recall_10: 0.8437 - learning_rate: 0.0010
Epoch 17/100
79/79            217s 3s/step -
accuracy: 0.8807 - loss: 0.6516 - precision_10: 0.8882 - recall_10: 0.8746 -
val_accuracy: 0.8070 - val_loss: 1.0548 - val_precision_10: 0.8169 -
val_recall_10: 0.8038 - learning_rate: 0.0010
Epoch 18/100
79/79            214s 3s/step -
accuracy: 0.8822 - loss: 0.6250 - precision_10: 0.8921 - recall_10: 0.8752 -
val_accuracy: 0.7751 - val_loss: 0.8857 - val_precision_10: 0.7914 -
val_recall_10: 0.7624 - learning_rate: 0.0010
Epoch 19/100
79/79            214s 3s/step -
accuracy: 0.8914 - loss: 0.5892 - precision_10: 0.9033 - recall_10: 0.8850 -
val_accuracy: 0.8453 - val_loss: 0.8096 - val_precision_10: 0.8502 -
val_recall_10: 0.8325 - learning_rate: 0.0010
Epoch 20/100
79/79            214s 3s/step -
accuracy: 0.9036 - loss: 0.5373 - precision_10: 0.9124 - recall_10: 0.8946 -
val_accuracy: 0.7384 - val_loss: 0.9917 - val_precision_10: 0.7562 -
val_recall_10: 0.7273 - learning_rate: 0.0010
Epoch 21/100
79/79            213s 3s/step -
accuracy: 0.8847 - loss: 0.5632 - precision_10: 0.8989 - recall_10: 0.8794 -
val_accuracy: 0.8389 - val_loss: 0.8826 - val_precision_10: 0.8430 -
val_recall_10: 0.8309 - learning_rate: 0.0010
Epoch 22/100
79/79            214s 3s/step -
accuracy: 0.9198 - loss: 0.4733 - precision_10: 0.9266 - recall_10: 0.9136 -
val_accuracy: 0.9139 - val_loss: 0.4732 - val_precision_10: 0.9148 -
val_recall_10: 0.9075 - learning_rate: 3.0000e-04
Epoch 23/100
79/79            215s 3s/step -
accuracy: 0.9199 - loss: 0.4340 - precision_10: 0.9272 - recall_10: 0.9177 -
val_accuracy: 0.9266 - val_loss: 0.4371 - val_precision_10: 0.9293 -
val_recall_10: 0.9219 - learning_rate: 3.0000e-04
Epoch 24/100
79/79            213s 3s/step -
accuracy: 0.9342 - loss: 0.3962 - precision_10: 0.9376 - recall_10: 0.9324 -
val_accuracy: 0.9234 - val_loss: 0.4435 - val_precision_10: 0.9247 -
val_recall_10: 0.9203 - learning_rate: 3.0000e-04
Epoch 25/100
79/79            212s 3s/step -
accuracy: 0.9268 - loss: 0.4026 - precision_10: 0.9329 - recall_10: 0.9238 -
val_accuracy: 0.9059 - val_loss: 0.4857 - val_precision_10: 0.9079 -
val_recall_10: 0.8963 - learning_rate: 3.0000e-04
```

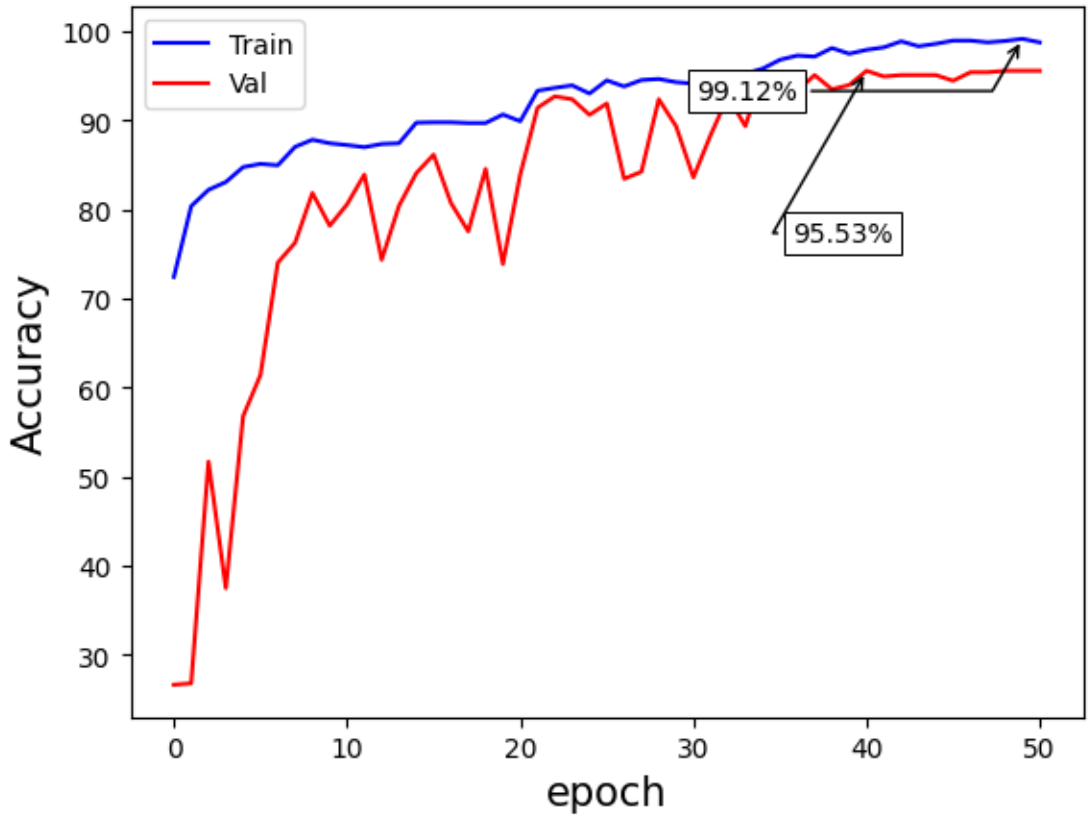
```
Epoch 26/100
79/79            213s 3s/step -
accuracy: 0.9329 - loss: 0.3985 - precision_10: 0.9348 - recall_10: 0.9308 -
val_accuracy: 0.9187 - val_loss: 0.4369 - val_precision_10: 0.9286 -
val_recall_10: 0.9123 - learning_rate: 3.0000e-04
Epoch 27/100
79/79            215s 3s/step -
accuracy: 0.9326 - loss: 0.3873 - precision_10: 0.9399 - recall_10: 0.9288 -
val_accuracy: 0.8341 - val_loss: 0.6911 - val_precision_10: 0.8352 -
val_recall_10: 0.8325 - learning_rate: 3.0000e-04
Epoch 28/100
79/79            214s 3s/step -
accuracy: 0.9331 - loss: 0.4022 - precision_10: 0.9363 - recall_10: 0.9262 -
val_accuracy: 0.8421 - val_loss: 0.6268 - val_precision_10: 0.8429 -
val_recall_10: 0.8389 - learning_rate: 3.0000e-04
Epoch 29/100
79/79            214s 3s/step -
accuracy: 0.9369 - loss: 0.3850 - precision_10: 0.9379 - recall_10: 0.9351 -
val_accuracy: 0.9234 - val_loss: 0.4010 - val_precision_10: 0.9262 -
val_recall_10: 0.9203 - learning_rate: 3.0000e-04
Epoch 30/100
79/79            260s 3s/step -
accuracy: 0.9406 - loss: 0.3518 - precision_10: 0.9457 - recall_10: 0.9381 -
val_accuracy: 0.8931 - val_loss: 0.5388 - val_precision_10: 0.8942 -
val_recall_10: 0.8900 - learning_rate: 3.0000e-04
Epoch 31/100
79/79            214s 3s/step -
accuracy: 0.9289 - loss: 0.3619 - precision_10: 0.9337 - recall_10: 0.9248 -
val_accuracy: 0.8357 - val_loss: 0.6457 - val_precision_10: 0.8368 -
val_recall_10: 0.8341 - learning_rate: 3.0000e-04
Epoch 32/100
79/79            214s 3s/step -
accuracy: 0.9458 - loss: 0.3378 - precision_10: 0.9482 - recall_10: 0.9425 -
val_accuracy: 0.8836 - val_loss: 0.5433 - val_precision_10: 0.8903 -
val_recall_10: 0.8804 - learning_rate: 3.0000e-04
Epoch 33/100
79/79            215s 3s/step -
accuracy: 0.9471 - loss: 0.3423 - precision_10: 0.9513 - recall_10: 0.9458 -
val_accuracy: 0.9266 - val_loss: 0.4201 - val_precision_10: 0.9280 -
val_recall_10: 0.9250 - learning_rate: 3.0000e-04
Epoch 34/100
79/79            222s 3s/step -
accuracy: 0.9454 - loss: 0.3340 - precision_10: 0.9477 - recall_10: 0.9417 -
val_accuracy: 0.8931 - val_loss: 0.4875 - val_precision_10: 0.8931 -
val_recall_10: 0.8931 - learning_rate: 3.0000e-04
Epoch 35/100
79/79            216s 3s/step -
accuracy: 0.9416 - loss: 0.3340 - precision_10: 0.9432 - recall_10: 0.9402 -
```

```
val_accuracy: 0.9537 - val_loss: 0.3396 - val_precision_10: 0.9552 -
val_recall_10: 0.9522 - learning_rate: 9.0000e-05
Epoch 36/100
79/79            215s 3s/step -
accuracy: 0.9656 - loss: 0.2744 - precision_10: 0.9679 - recall_10: 0.9626 -
val_accuracy: 0.9474 - val_loss: 0.3418 - val_precision_10: 0.9489 -
val_recall_10: 0.9474 - learning_rate: 9.0000e-05
Epoch 37/100
79/79            181s 2s/step -
accuracy: 0.9604 - loss: 0.2723 - precision_10: 0.9631 - recall_10: 0.9565 -
val_accuracy: 0.9330 - val_loss: 0.3556 - val_precision_10: 0.9345 -
val_recall_10: 0.9330 - learning_rate: 9.0000e-05
Epoch 38/100
79/79            159s 2s/step -
accuracy: 0.9641 - loss: 0.2649 - precision_10: 0.9653 - recall_10: 0.9634 -
val_accuracy: 0.9506 - val_loss: 0.3432 - val_precision_10: 0.9505 -
val_recall_10: 0.9490 - learning_rate: 9.0000e-05
Epoch 39/100
79/79            159s 2s/step -
accuracy: 0.9823 - loss: 0.2325 - precision_10: 0.9830 - recall_10: 0.9806 -
val_accuracy: 0.9346 - val_loss: 0.3617 - val_precision_10: 0.9345 -
val_recall_10: 0.9330 - learning_rate: 9.0000e-05
Epoch 40/100
79/79            159s 2s/step -
accuracy: 0.9772 - loss: 0.2331 - precision_10: 0.9774 - recall_10: 0.9752 -
val_accuracy: 0.9394 - val_loss: 0.3829 - val_precision_10: 0.9394 -
val_recall_10: 0.9394 - learning_rate: 9.0000e-05
Epoch 41/100
79/79            158s 2s/step -
accuracy: 0.9711 - loss: 0.2433 - precision_10: 0.9731 - recall_10: 0.9702 -
val_accuracy: 0.9553 - val_loss: 0.3282 - val_precision_10: 0.9553 -
val_recall_10: 0.9553 - learning_rate: 2.7000e-05
Epoch 42/100
79/79            158s 2s/step -
accuracy: 0.9750 - loss: 0.2268 - precision_10: 0.9768 - recall_10: 0.9724 -
val_accuracy: 0.9490 - val_loss: 0.3377 - val_precision_10: 0.9489 -
val_recall_10: 0.9474 - learning_rate: 2.7000e-05
Epoch 43/100
79/79            157s 2s/step -
accuracy: 0.9850 - loss: 0.2104 - precision_10: 0.9851 - recall_10: 0.9841 -
val_accuracy: 0.9506 - val_loss: 0.3360 - val_precision_10: 0.9521 -
val_recall_10: 0.9506 - learning_rate: 2.7000e-05
Epoch 44/100
79/79            157s 2s/step -
accuracy: 0.9782 - loss: 0.2222 - precision_10: 0.9786 - recall_10: 0.9777 -
val_accuracy: 0.9506 - val_loss: 0.3394 - val_precision_10: 0.9521 -
val_recall_10: 0.9506 - learning_rate: 2.7000e-05
Epoch 45/100
```

```
79/79            158s 2s/step -
accuracy: 0.9817 - loss: 0.2233 - precision_10: 0.9819 - recall_10: 0.9808 -
val_accuracy: 0.9506 - val_loss: 0.3413 - val_precision_10: 0.9506 -
val_recall_10: 0.9506 - learning_rate: 2.7000e-05
Epoch 46/100
79/79            157s 2s/step -
accuracy: 0.9893 - loss: 0.2013 - precision_10: 0.9901 - recall_10: 0.9880 -
val_accuracy: 0.9442 - val_loss: 0.3403 - val_precision_10: 0.9441 -
val_recall_10: 0.9426 - learning_rate: 2.7000e-05
Epoch 47/100
79/79            159s 2s/step -
accuracy: 0.9876 - loss: 0.2088 - precision_10: 0.9876 - recall_10: 0.9869 -
val_accuracy: 0.9537 - val_loss: 0.3367 - val_precision_10: 0.9537 -
val_recall_10: 0.9522 - learning_rate: 1.0000e-05
Epoch 48/100
79/79            162s 2s/step -
accuracy: 0.9876 - loss: 0.2028 - precision_10: 0.9878 - recall_10: 0.9876 -
val_accuracy: 0.9537 - val_loss: 0.3397 - val_precision_10: 0.9553 -
val_recall_10: 0.9537 - learning_rate: 1.0000e-05
Epoch 49/100
79/79            159s 2s/step -
accuracy: 0.9867 - loss: 0.1961 - precision_10: 0.9867 - recall_10: 0.9866 -
val_accuracy: 0.9553 - val_loss: 0.3399 - val_precision_10: 0.9553 -
val_recall_10: 0.9553 - learning_rate: 1.0000e-05
Epoch 50/100
79/79            159s 2s/step -
accuracy: 0.9849 - loss: 0.1952 - precision_10: 0.9849 - recall_10: 0.9842 -
val_accuracy: 0.9553 - val_loss: 0.3407 - val_precision_10: 0.9553 -
val_recall_10: 0.9553 - learning_rate: 1.0000e-05
Epoch 51/100
79/79            159s 2s/step -
accuracy: 0.9862 - loss: 0.1946 - precision_10: 0.9862 - recall_10: 0.9862 -
val_accuracy: 0.9553 - val_loss: 0.3416 - val_precision_10: 0.9584 -
val_recall_10: 0.9553 - learning_rate: 1.0000e-05
```

4.2.2 Plot accuracy for training and validation wrt epoch

[185]:
```
plot_accuracy(model_fit)
```



4.2.3 Tensorboard

[186]:
```
# !rm -rf logs/
# rmdir /s /q logs
```

[187]:
```
%load_ext tensorboard
```

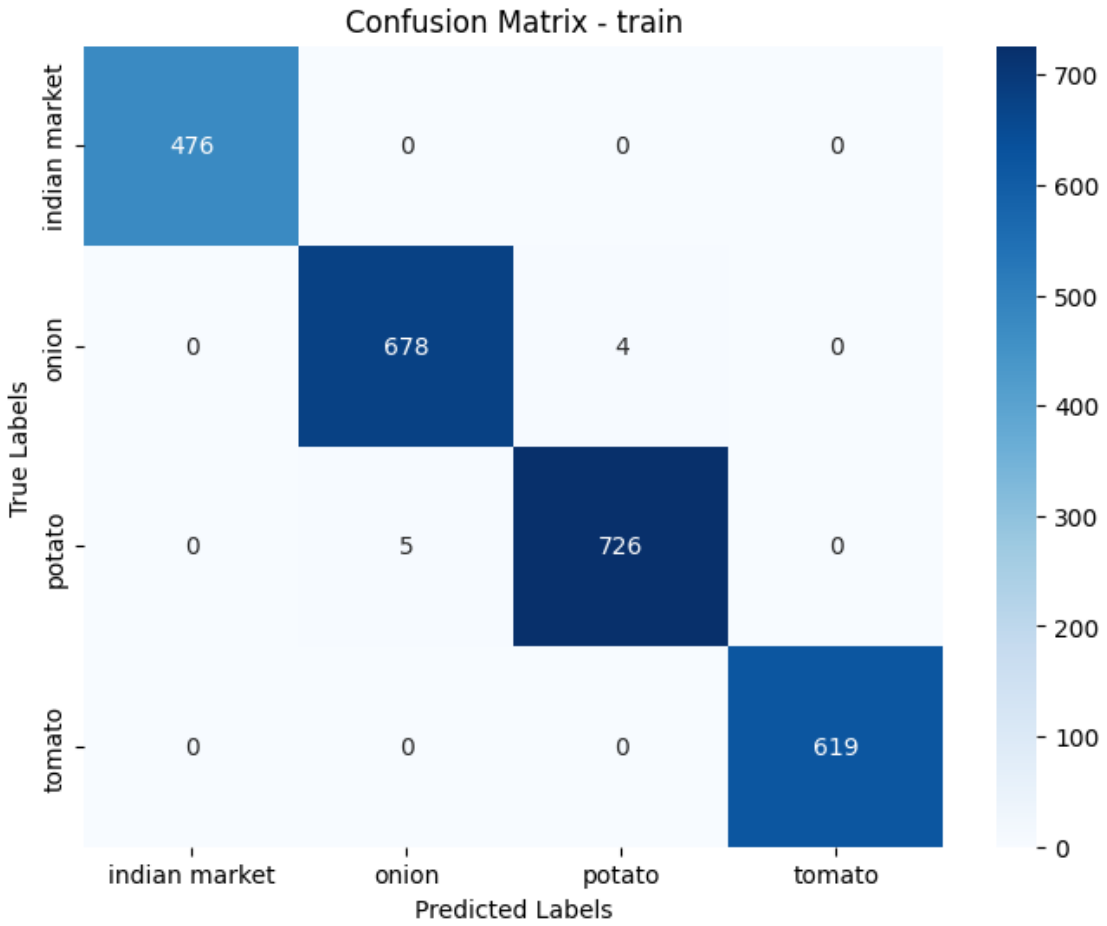
```
The tensorboard extension is already loaded. To reload it, use:
  %reload_ext tensorboard
```

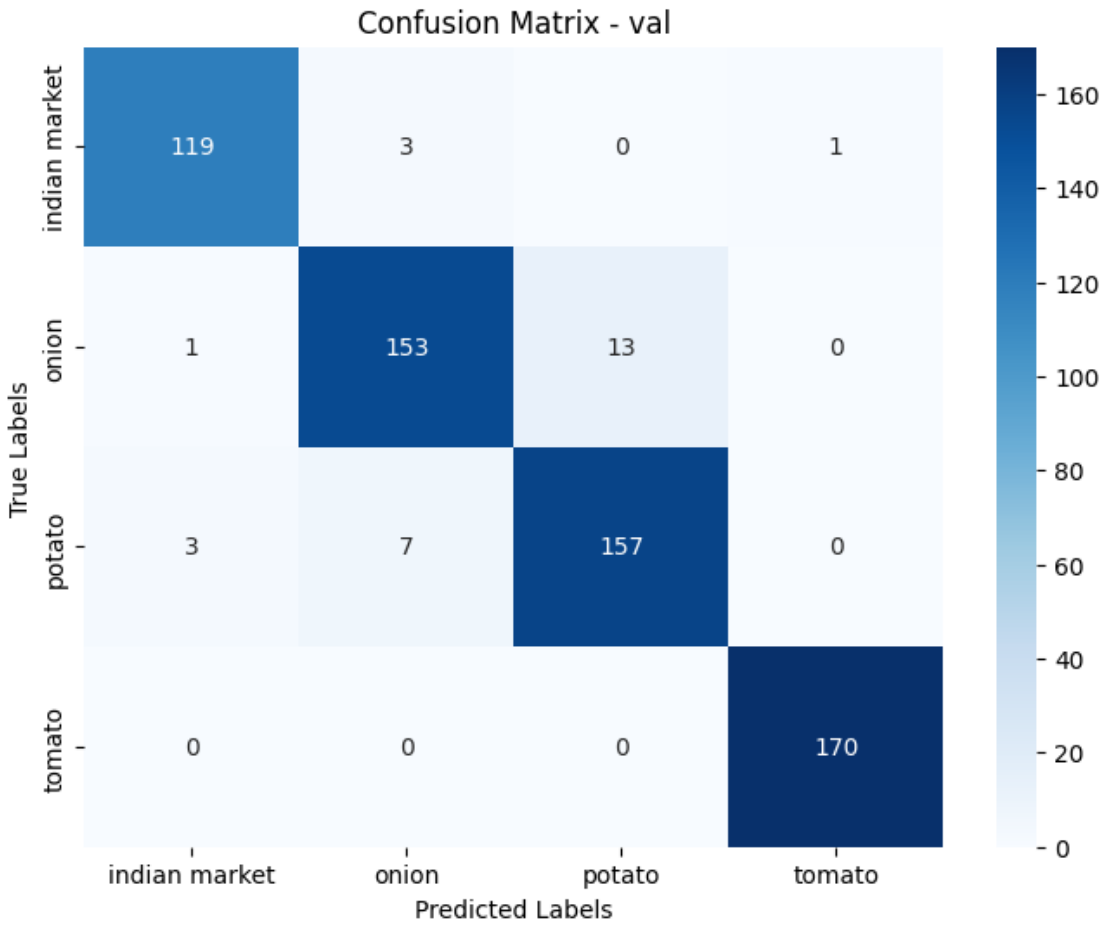
4.2.4 Model Evaluation, Mlflow Logging and Printing Metrics

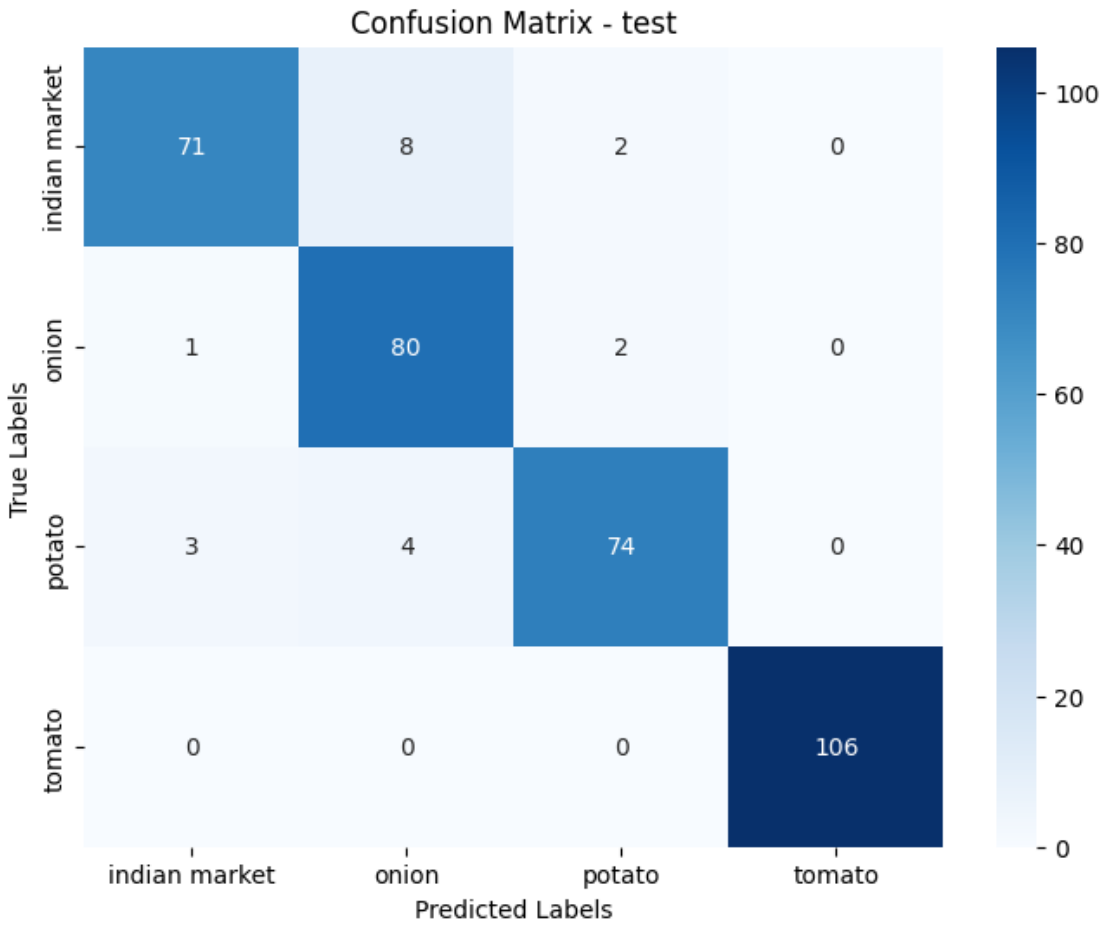
[188]:
```
evaluate_model(model, train_ds, val_ds, test_ds, class_names,run_name=f"{model.
↪name}",ckpt_path= f"checkpoints/{model.name}.weights.h5")
```

```
2025/02/09 00:52:16 WARNING mlflow.tensorflow: You are saving a TensorFlow Core
model or Keras model without a signature. Inference with
mlflow.pyfunc.spark_udf() will not work unless the model's pyfunc representation
accepts pandas DataFrames as inference inputs.
2025/02/09 00:52:27 WARNING mlflow.models.model: Model logged without a
```

```
signature and input example. Please set `input_example` parameter when logging
the model to auto infer the model signature.


train Metrics:
 Accuracy: 99.64%
 Precision: 99.68%
 Recall: 99.68%

val Metrics:
 Accuracy: 95.53%
 Precision: 95.60%
 Recall: 95.59%

test Metrics:
 Accuracy: 94.30%
 Precision: 94.12%
 Recall: 93.85%
```




Confusion Matrix - val
True Labels
indian market
onion
potato
tomato
119
3
0
1
1
153
13
0
3
7
157
0
0
0
0
170
indian market
onion
potato
tomato
Predicted Labels
160
140
120
100
80
60
40
20
0




```
Classification Report - train
              precision    recall  f1-score   support

indian market       1.00      1.00      1.00       476
        onion       0.99      0.99      0.99       682
       potato       0.99      0.99      0.99       731
       tomato       1.00      1.00      1.00       619

     accuracy                           1.00      2508
    macro avg       1.00      1.00      1.00      2508
 weighted avg       1.00      1.00      1.00      2508


Classification Report - val
              precision    recall  f1-score   support

indian market       0.97      0.97      0.97       123
```

```
        onion       0.94      0.92      0.93       167
       potato       0.92      0.94      0.93       167
       tomato       0.99      1.00      1.00       170

     accuracy                           0.96       627
    macro avg       0.96      0.96      0.96       627
 weighted avg       0.96      0.96      0.96       627


Classification Report - test
               precision    recall  f1-score   support

indian market       0.95      0.88      0.91        81
        onion       0.87      0.96      0.91        83
       potato       0.95      0.91      0.93        81
       tomato       1.00      1.00      1.00       106

     accuracy                           0.94       351
    macro avg       0.94      0.94      0.94       351
 weighted avg       0.95      0.94      0.94       351


Class-wise Accuracy:

Train Class-wise Accuracy:
 indian market: 100.00% (476/476)
 onion: 99.41% (678/682)
 potato: 99.32% (726/731)
 tomato: 100.00% (619/619)

Val Class-wise Accuracy:
 indian market: 96.75% (119/123)
 onion: 91.62% (153/167)
 potato: 94.01% (157/167)
 tomato: 100.00% (170/170)

Test Class-wise Accuracy:
 indian market: 87.65% (71/81)
 onion: 96.39% (80/83)
 potato: 91.36% (74/81)
 tomato: 100.00% (106/106)

Random Test Predictions:
```

True: indian market
Pred: indian market

True: indian market
Pred: indian market

True: onion
Pred: onion

True: onion
Pred: onion

True: potato
Pred: onion

True: potato
Pred: potato

True: tomato
Pred: tomato

True: tomato
Pred: tomato

4.2.5 Save the model

[189]:
```
model.save("saved_models/complex_cnn_model.keras")
```

Observations - Simple CNN:
- **Architecture**: 1 Conv layer → MaxPooling → Flatten → Dense layers. - **Compilation and training**: adam → [accuracy,precision,recall] → Model Checkpoint (Save best only) - **Performance**:
- Train Accuracy: **98.13%** and Validation Accuracy **82.46%** (overfit).
- Test Accuracy: **76.07%** (poor generalization). - Epochs - **10** - Training Time - **25 min (approx)**
- Misclassified "Indian market" as "onion/potato" due to noise.
- **Complex CNN**:
- **Architecture**: 5 Conv blocks with BatchNorm, Relu activation, Dropout, and L2 regularization.
- **Compilation and training**: adam → [accuracy,precision,recall] → Model Checkpoint (Save best only) → ReduceLROnPlateau → EarlyStopping → Tensorboard logging - **Performance**:
- Train Accuracy: **99.64%** and Validation Accuracy **95.53%** (Reduced Overfitting compared to simple CNN) - Test Accuracy: **94.30%**. - Epochs - **51** - Training Time - **170 min (approx)** (Training time increased 7X times) - Effective at filtering noise (e.g., 97% recall for "tomato").

5 DATA AUGMENTATION

5.0.1 Data Augmentation Preprocessing

[65]:
```
def preprocess_aug(train_data, val_data, test_data, target_height=256,␣
↪target_width=256):
    # Data preprocessing pipeline for validation and test datasets
    data_preprocess = Sequential(
        name="data_preprocess",
        layers=[
            layers.Resizing(target_height, target_width),  # Resize to target␣
↪size
            layers.Rescaling(1.0 / 255),  # Normalize pixel values to [0, 1]
        ],
    )

    # Data augmentation pipeline for training dataset
    data_augmentation = Sequential(
        name="data_augmentation",
        layers=[
            layers.Resizing(300, 300),  # Resize to a larger size first
            layers.RandomCrop(target_height, target_width),  # Randomly crop to␣
↪target size
            layers.RandomTranslation(height_factor=0.1, width_factor=0.1),  #␣
↪Random translation
            layers.RandomRotation(0.1),  # Random rotation (in radians)
            layers.RandomBrightness(0.1),  # Random brightness adjustment
            layers.RandomContrast(0.1),  # Random contrast adjustment
```

```
            layers.Rescaling(1.0 / 255),  # Normalize pixel values to [0, 1]
        ],
    )

    # Apply data augmentation to the training dataset
    train_aug_ds = train_data.map(
        lambda x, y: (data_augmentation(x), y), num_parallel_calls=tf.data.
 ↪AUTOTUNE
    ).prefetch(tf.data.AUTOTUNE)

    # Apply preprocessing to the validation dataset
    val_aug_ds = val_data.map(
        lambda x, y: (data_preprocess(x), y), num_parallel_calls=tf.data.
 ↪AUTOTUNE
    ).prefetch(tf.data.AUTOTUNE)

    # Apply preprocessing to the test dataset
    test_aug_ds = test_data.map(
        lambda x, y: (data_preprocess(x), y), num_parallel_calls=tf.data.
 ↪AUTOTUNE
    ).prefetch(tf.data.AUTOTUNE)

    return train_aug_ds, val_aug_ds, test_aug_ds
```

[66]:
```
train_aug_ds, val_aug_ds, test_aug_ds = preprocess_aug(train_data, val_data,␣
 ↪test_data)
```

[192]:
```
model = complex_cnn()
model.summary()
```

```
C:\Users\saina\AppData\Roaming\Python\Python312\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: "complex_cnn"

Layer (type)	Output Shape	Param #
conv2d_67 (Conv2D)	(None, 256, 256, 32)	896
activation_56 (Activation)	(None, 256, 256, 32)	0
batch_normalization_56	(None, 256, 256, 32)	128

```
(BatchNormalization)

max_pooling2d_56 (MaxPooling2D)   (None, 128, 128, 32)              0

conv2d_68 (Conv2D)                (None, 128, 128, 64)          18,496

activation_57 (Activation)        (None, 128, 128, 64)              0

batch_normalization_57            (None, 128, 128, 64)            256
(BatchNormalization)

max_pooling2d_57 (MaxPooling2D)   (None, 64, 64, 64)                0

conv2d_69 (Conv2D)                (None, 64, 64, 128)           73,856

activation_58 (Activation)        (None, 64, 64, 128)               0

batch_normalization_58            (None, 64, 64, 128)             512
(BatchNormalization)

max_pooling2d_58 (MaxPooling2D)   (None, 32, 32, 128)               0

conv2d_70 (Conv2D)                (None, 32, 32, 256)          295,168

activation_59 (Activation)        (None, 32, 32, 256)               0

batch_normalization_59            (None, 32, 32, 256)           1,024
(BatchNormalization)

max_pooling2d_59 (MaxPooling2D)   (None, 16, 16, 256)               0

conv2d_71 (Conv2D)                (None, 16, 16, 512)       1,180,160

activation_60 (Activation)        (None, 16, 16, 512)               0

batch_normalization_60            (None, 16, 16, 512)           2,048
(BatchNormalization)

max_pooling2d_60 (MaxPooling2D)   (None, 8, 8, 512)                 0

global_average_pooling2d_13       (None, 512)                       0
(GlobalAveragePooling2D)

dense_32 (Dense)                  (None, 512)                 262,656

activation_61 (Activation)        (None, 512)                       0

batch_normalization_61            (None, 512)                   2,048
```

```
 (BatchNormalization)

 dropout_9 (Dropout)              (None, 512)                    0

 dense_33 (Dense)                 (None, 256)              131,328

 activation_62 (Activation)       (None, 256)                    0

 batch_normalization_62           (None, 256)                1,024
 (BatchNormalization)

 dense_34 (Dense)                 (None, 4)                  1,028


 Total params: 1,970,628 (7.52 MB)

 Trainable params: 1,967,108 (7.50 MB)

 Non-trainable params: 3,520 (13.75 KB)
```

5.0.2 Model Compilation and Training including all Callback Features

[193]:
```
model_fit = compile_train(model, train_aug_ds, val_aug_ds, log_dir=f"logs/
↪{model.name}_aug",epochs=100,ckpt_path=f"checkpoints/{model.name}_aug.
↪weights.h5")
```

```
Epoch 1/100
79/79             167s 2s/step -
accuracy: 0.6657 - loss: 1.7372 - precision_11: 0.7089 - recall_11: 0.6227 -
val_accuracy: 0.3222 - val_loss: 2.0990 - val_precision_11: 0.3509 -
val_recall_11: 0.1483 - learning_rate: 0.0010
Epoch 2/100
79/79             162s 2s/step -
accuracy: 0.7852 - loss: 1.3563 - precision_11: 0.8104 - recall_11: 0.7534 -
val_accuracy: 0.3573 - val_loss: 3.2594 - val_precision_11: 0.3583 -
val_recall_11: 0.3509 - learning_rate: 0.0010
Epoch 3/100
79/79             162s 2s/step -
accuracy: 0.7890 - loss: 1.3161 - precision_11: 0.8173 - recall_11: 0.7580 -
val_accuracy: 0.2775 - val_loss: 2.8202 - val_precision_11: 0.2790 -
val_recall_11: 0.2759 - learning_rate: 0.0010
Epoch 4/100
79/79             164s 2s/step -
accuracy: 0.8162 - loss: 1.1929 - precision_11: 0.8466 - recall_11: 0.7973 -
val_accuracy: 0.4338 - val_loss: 2.8151 - val_precision_11: 0.4376 -
```

```
val_recall_11: 0.4306 - learning_rate: 0.0010
Epoch 5/100
79/79            167s 2s/step -
accuracy: 0.8158 - loss: 1.1308 - precision_11: 0.8479 - recall_11: 0.7908 -
val_accuracy: 0.5582 - val_loss: 2.0908 - val_precision_11: 0.5783 -
val_recall_11: 0.5183 - learning_rate: 0.0010
Epoch 6/100
79/79            164s 2s/step -
accuracy: 0.8040 - loss: 1.1579 - precision_11: 0.8253 - recall_11: 0.7801 -
val_accuracy: 0.5774 - val_loss: 1.7055 - val_precision_11: 0.5972 -
val_recall_11: 0.5534 - learning_rate: 0.0010
Epoch 7/100
79/79            165s 2s/step -
accuracy: 0.8415 - loss: 1.0036 - precision_11: 0.8562 - recall_11: 0.8177 -
val_accuracy: 0.7161 - val_loss: 1.3001 - val_precision_11: 0.7452 -
val_recall_11: 0.6762 - learning_rate: 0.0010
Epoch 8/100
79/79            164s 2s/step -
accuracy: 0.8209 - loss: 1.0052 - precision_11: 0.8425 - recall_11: 0.7962 -
val_accuracy: 0.7209 - val_loss: 1.2546 - val_precision_11: 0.7337 -
val_recall_11: 0.6986 - learning_rate: 0.0010
Epoch 9/100
79/79            162s 2s/step -
accuracy: 0.8230 - loss: 1.0024 - precision_11: 0.8290 - recall_11: 0.7971 -
val_accuracy: 0.8230 - val_loss: 0.9214 - val_precision_11: 0.8416 -
val_recall_11: 0.8134 - learning_rate: 0.0010
Epoch 10/100
79/79            161s 2s/step -
accuracy: 0.8338 - loss: 0.9234 - precision_11: 0.8559 - recall_11: 0.8119 -
val_accuracy: 0.7799 - val_loss: 1.0266 - val_precision_11: 0.7912 -
val_recall_11: 0.7496 - learning_rate: 0.0010
Epoch 11/100
79/79            162s 2s/step -
accuracy: 0.8530 - loss: 0.8602 - precision_11: 0.8699 - recall_11: 0.8363 -
val_accuracy: 0.7719 - val_loss: 1.0335 - val_precision_11: 0.7901 -
val_recall_11: 0.7624 - learning_rate: 0.0010
Epoch 12/100
79/79            161s 2s/step -
accuracy: 0.8468 - loss: 0.8214 - precision_11: 0.8679 - recall_11: 0.8302 -
val_accuracy: 0.8325 - val_loss: 0.8714 - val_precision_11: 0.8474 -
val_recall_11: 0.8150 - learning_rate: 0.0010
Epoch 13/100
79/79            162s 2s/step -
accuracy: 0.8487 - loss: 0.7996 - precision_11: 0.8658 - recall_11: 0.8259 -
val_accuracy: 0.2600 - val_loss: 5.0937 - val_precision_11: 0.2625 -
val_recall_11: 0.2520 - learning_rate: 0.0010
Epoch 14/100
79/79            163s 2s/step -
```

```
accuracy: 0.8320 - loss: 0.7910 - precision_11: 0.8500 - recall_11: 0.8204 -
val_accuracy: 0.8341 - val_loss: 0.7807 - val_precision_11: 0.8522 -
val_recall_11: 0.8182 - learning_rate: 0.0010
Epoch 15/100
79/79            162s 2s/step -
accuracy: 0.8402 - loss: 0.7691 - precision_11: 0.8588 - recall_11: 0.8241 -
val_accuracy: 0.8278 - val_loss: 0.8021 - val_precision_11: 0.8449 -
val_recall_11: 0.8166 - learning_rate: 0.0010
Epoch 16/100
79/79            163s 2s/step -
accuracy: 0.8543 - loss: 0.7281 - precision_11: 0.8729 - recall_11: 0.8376 -
val_accuracy: 0.8469 - val_loss: 0.6885 - val_precision_11: 0.8539 -
val_recall_11: 0.8389 - learning_rate: 0.0010
Epoch 17/100
79/79            163s 2s/step -
accuracy: 0.8417 - loss: 0.7236 - precision_11: 0.8554 - recall_11: 0.8256 -
val_accuracy: 0.8772 - val_loss: 0.6057 - val_precision_11: 0.8929 -
val_recall_11: 0.8644 - learning_rate: 0.0010
Epoch 18/100
79/79            161s 2s/step -
accuracy: 0.8525 - loss: 0.6754 - precision_11: 0.8740 - recall_11: 0.8298 -
val_accuracy: 0.7352 - val_loss: 0.9247 - val_precision_11: 0.7612 -
val_recall_11: 0.7065 - learning_rate: 0.0010
Epoch 19/100
79/79            164s 2s/step -
accuracy: 0.8512 - loss: 0.6794 - precision_11: 0.8644 - recall_11: 0.8330 -
val_accuracy: 0.6699 - val_loss: 1.0651 - val_precision_11: 0.6880 -
val_recall_11: 0.6507 - learning_rate: 0.0010
Epoch 20/100
79/79            216s 3s/step -
accuracy: 0.8540 - loss: 0.6418 - precision_11: 0.8716 - recall_11: 0.8423 -
val_accuracy: 0.8549 - val_loss: 0.6423 - val_precision_11: 0.8648 -
val_recall_11: 0.8469 - learning_rate: 0.0010
Epoch 21/100
79/79            221s 3s/step -
accuracy: 0.8534 - loss: 0.6181 - precision_11: 0.8709 - recall_11: 0.8453 -
val_accuracy: 0.8437 - val_loss: 0.6150 - val_precision_11: 0.8541 -
val_recall_11: 0.8309 - learning_rate: 0.0010
Epoch 22/100
79/79            222s 3s/step -
accuracy: 0.8507 - loss: 0.5980 - precision_11: 0.8682 - recall_11: 0.8335 -
val_accuracy: 0.8054 - val_loss: 0.7211 - val_precision_11: 0.8157 -
val_recall_11: 0.7974 - learning_rate: 0.0010
Epoch 23/100
79/79            220s 3s/step -
accuracy: 0.8708 - loss: 0.5704 - precision_11: 0.8773 - recall_11: 0.8579 -
val_accuracy: 0.9123 - val_loss: 0.4785 - val_precision_11: 0.9157 -
val_recall_11: 0.9011 - learning_rate: 3.0000e-04
```

```
Epoch 24/100
79/79            222s 3s/step -
accuracy: 0.9004 - loss: 0.4712 - precision_11: 0.9087 - recall_11: 0.8915 -
val_accuracy: 0.9107 - val_loss: 0.4716 - val_precision_11: 0.9175 -
val_recall_11: 0.9043 - learning_rate: 3.0000e-04
Epoch 25/100
79/79            208s 3s/step -
accuracy: 0.8819 - loss: 0.5105 - precision_11: 0.8897 - recall_11: 0.8750 -
val_accuracy: 0.8979 - val_loss: 0.4867 - val_precision_11: 0.8989 -
val_recall_11: 0.8931 - learning_rate: 3.0000e-04
Epoch 26/100
79/79            163s 2s/step -
accuracy: 0.9004 - loss: 0.4870 - precision_11: 0.9060 - recall_11: 0.8911 -
val_accuracy: 0.9027 - val_loss: 0.4529 - val_precision_11: 0.9125 -
val_recall_11: 0.8979 - learning_rate: 3.0000e-04
Epoch 27/100
79/79            163s 2s/step -
accuracy: 0.9083 - loss: 0.4474 - precision_11: 0.9179 - recall_11: 0.9003 -
val_accuracy: 0.8931 - val_loss: 0.5033 - val_precision_11: 0.9003 -
val_recall_11: 0.8931 - learning_rate: 3.0000e-04
Epoch 28/100
79/79            162s 2s/step -
accuracy: 0.8870 - loss: 0.5066 - precision_11: 0.8938 - recall_11: 0.8747 -
val_accuracy: 0.8884 - val_loss: 0.4744 - val_precision_11: 0.8934 -
val_recall_11: 0.8820 - learning_rate: 3.0000e-04
Epoch 29/100
79/79            163s 2s/step -
accuracy: 0.8957 - loss: 0.4604 - precision_11: 0.9023 - recall_11: 0.8872 -
val_accuracy: 0.8995 - val_loss: 0.4633 - val_precision_11: 0.9032 -
val_recall_11: 0.8931 - learning_rate: 3.0000e-04
Epoch 30/100
79/79            162s 2s/step -
accuracy: 0.8896 - loss: 0.4563 - precision_11: 0.8981 - recall_11: 0.8817 -
val_accuracy: 0.8884 - val_loss: 0.4749 - val_precision_11: 0.8939 -
val_recall_11: 0.8868 - learning_rate: 3.0000e-04
Epoch 31/100
79/79            163s 2s/step -
accuracy: 0.9050 - loss: 0.4395 - precision_11: 0.9118 - recall_11: 0.8951 -
val_accuracy: 0.8341 - val_loss: 0.6016 - val_precision_11: 0.8429 -
val_recall_11: 0.8214 - learning_rate: 3.0000e-04
Epoch 32/100
79/79            162s 2s/step -
accuracy: 0.9110 - loss: 0.4128 - precision_11: 0.9180 - recall_11: 0.9052 -
val_accuracy: 0.9043 - val_loss: 0.4081 - val_precision_11: 0.9069 -
val_recall_11: 0.9011 - learning_rate: 9.0000e-05
Epoch 33/100
79/79            162s 2s/step -
accuracy: 0.9231 - loss: 0.3821 - precision_11: 0.9287 - recall_11: 0.9126 -
```

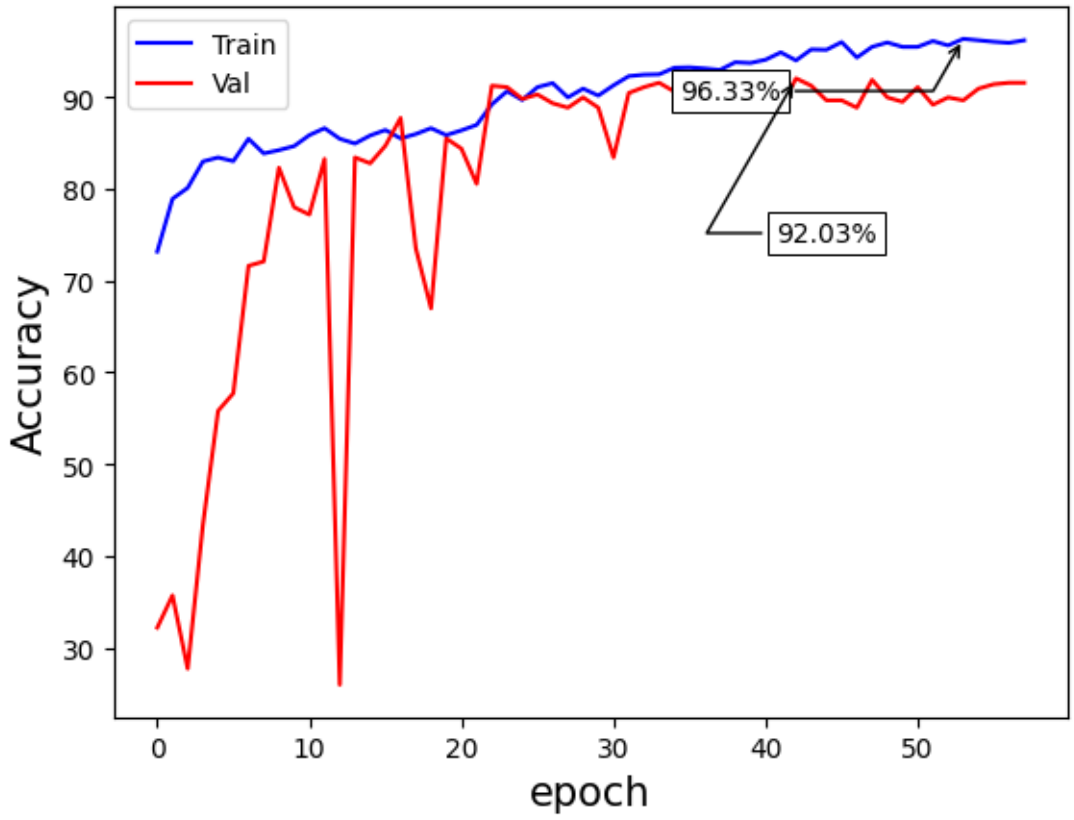
```
val_accuracy: 0.9107 - val_loss: 0.4061 - val_precision_11: 0.9130 -
val_recall_11: 0.9043 - learning_rate: 9.0000e-05
Epoch 34/100
79/79            162s 2s/step -
accuracy: 0.9178 - loss: 0.3805 - precision_11: 0.9214 - recall_11: 0.9088 -
val_accuracy: 0.9155 - val_loss: 0.3895 - val_precision_11: 0.9181 -
val_recall_11: 0.9123 - learning_rate: 9.0000e-05
Epoch 35/100
79/79            162s 2s/step -
accuracy: 0.9217 - loss: 0.3679 - precision_11: 0.9256 - recall_11: 0.9155 -
val_accuracy: 0.9059 - val_loss: 0.4111 - val_precision_11: 0.9085 -
val_recall_11: 0.9027 - learning_rate: 9.0000e-05
Epoch 36/100
79/79            163s 2s/step -
accuracy: 0.9273 - loss: 0.3562 - precision_11: 0.9319 - recall_11: 0.9228 -
val_accuracy: 0.8963 - val_loss: 0.4284 - val_precision_11: 0.8990 -
val_recall_11: 0.8947 - learning_rate: 9.0000e-05
Epoch 37/100
79/79            163s 2s/step -
accuracy: 0.9285 - loss: 0.3539 - precision_11: 0.9334 - recall_11: 0.9223 -
val_accuracy: 0.9139 - val_loss: 0.3878 - val_precision_11: 0.9136 -
val_recall_11: 0.9107 - learning_rate: 9.0000e-05
Epoch 38/100
79/79            162s 2s/step -
accuracy: 0.9219 - loss: 0.3607 - precision_11: 0.9242 - recall_11: 0.9191 -
val_accuracy: 0.8884 - val_loss: 0.4464 - val_precision_11: 0.8898 -
val_recall_11: 0.8884 - learning_rate: 9.0000e-05
Epoch 39/100
79/79            164s 2s/step -
accuracy: 0.9287 - loss: 0.3421 - precision_11: 0.9335 - recall_11: 0.9264 -
val_accuracy: 0.9187 - val_loss: 0.3941 - val_precision_11: 0.9187 -
val_recall_11: 0.9187 - learning_rate: 9.0000e-05
Epoch 40/100
79/79            162s 2s/step -
accuracy: 0.9236 - loss: 0.3673 - precision_11: 0.9295 - recall_11: 0.9210 -
val_accuracy: 0.9043 - val_loss: 0.4246 - val_precision_11: 0.9087 -
val_recall_11: 0.9043 - learning_rate: 9.0000e-05
Epoch 41/100
79/79            162s 2s/step -
accuracy: 0.9279 - loss: 0.3511 - precision_11: 0.9291 - recall_11: 0.9232 -
val_accuracy: 0.9107 - val_loss: 0.4037 - val_precision_11: 0.9121 -
val_recall_11: 0.9107 - learning_rate: 9.0000e-05
Epoch 42/100
79/79            163s 2s/step -
accuracy: 0.9438 - loss: 0.3095 - precision_11: 0.9457 - recall_11: 0.9425 -
val_accuracy: 0.8884 - val_loss: 0.4598 - val_precision_11: 0.8925 -
val_recall_11: 0.8868 - learning_rate: 9.0000e-05
Epoch 43/100
```

```
79/79            162s 2s/step -
accuracy: 0.9328 - loss: 0.3295 - precision_11: 0.9348 - recall_11: 0.9290 -
val_accuracy: 0.9203 - val_loss: 0.3750 - val_precision_11: 0.9215 -
val_recall_11: 0.9171 - learning_rate: 2.7000e-05
Epoch 44/100
79/79            162s 2s/step -
accuracy: 0.9455 - loss: 0.2966 - precision_11: 0.9483 - recall_11: 0.9415 -
val_accuracy: 0.9123 - val_loss: 0.3760 - val_precision_11: 0.9180 -
val_recall_11: 0.9107 - learning_rate: 2.7000e-05
Epoch 45/100
79/79            163s 2s/step -
accuracy: 0.9481 - loss: 0.3114 - precision_11: 0.9525 - recall_11: 0.9452 -
val_accuracy: 0.8963 - val_loss: 0.4063 - val_precision_11: 0.8973 -
val_recall_11: 0.8915 - learning_rate: 2.7000e-05
Epoch 46/100
79/79            164s 2s/step -
accuracy: 0.9559 - loss: 0.2825 - precision_11: 0.9579 - recall_11: 0.9542 -
val_accuracy: 0.8963 - val_loss: 0.4063 - val_precision_11: 0.8974 -
val_recall_11: 0.8931 - learning_rate: 2.7000e-05
Epoch 47/100
79/79            162s 2s/step -
accuracy: 0.9336 - loss: 0.3181 - precision_11: 0.9383 - recall_11: 0.9324 -
val_accuracy: 0.8884 - val_loss: 0.4192 - val_precision_11: 0.8926 -
val_recall_11: 0.8884 - learning_rate: 2.7000e-05
Epoch 48/100
79/79            163s 2s/step -
accuracy: 0.9470 - loss: 0.2901 - precision_11: 0.9506 - recall_11: 0.9446 -
val_accuracy: 0.9187 - val_loss: 0.3740 - val_precision_11: 0.9201 -
val_recall_11: 0.9187 - learning_rate: 2.7000e-05
Epoch 49/100
79/79            162s 2s/step -
accuracy: 0.9583 - loss: 0.2824 - precision_11: 0.9620 - recall_11: 0.9561 -
val_accuracy: 0.8995 - val_loss: 0.4126 - val_precision_11: 0.8992 -
val_recall_11: 0.8963 - learning_rate: 2.7000e-05
Epoch 50/100
79/79            162s 2s/step -
accuracy: 0.9553 - loss: 0.2804 - precision_11: 0.9568 - recall_11: 0.9533 -
val_accuracy: 0.8947 - val_loss: 0.4033 - val_precision_11: 0.8960 -
val_recall_11: 0.8931 - learning_rate: 2.7000e-05
Epoch 51/100
79/79            162s 2s/step -
accuracy: 0.9439 - loss: 0.2953 - precision_11: 0.9483 - recall_11: 0.9423 -
val_accuracy: 0.9107 - val_loss: 0.3970 - val_precision_11: 0.9119 -
val_recall_11: 0.9075 - learning_rate: 2.7000e-05
Epoch 52/100
79/79            162s 2s/step -
accuracy: 0.9579 - loss: 0.2681 - precision_11: 0.9594 - recall_11: 0.9551 -
val_accuracy: 0.8915 - val_loss: 0.4479 - val_precision_11: 0.8928 -
```

```
val_recall_11: 0.8900 - learning_rate: 2.7000e-05
Epoch 53/100
79/79             163s 2s/step -
accuracy: 0.9556 - loss: 0.2791 - precision_11: 0.9560 - recall_11: 0.9544 -
val_accuracy: 0.8995 - val_loss: 0.3998 - val_precision_11: 0.9051 -
val_recall_11: 0.8979 - learning_rate: 2.7000e-05
Epoch 54/100
79/79             162s 2s/step -
accuracy: 0.9563 - loss: 0.2703 - precision_11: 0.9587 - recall_11: 0.9541 -
val_accuracy: 0.8963 - val_loss: 0.4104 - val_precision_11: 0.9006 -
val_recall_11: 0.8963 - learning_rate: 1.0000e-05
Epoch 55/100
79/79             163s 2s/step -
accuracy: 0.9605 - loss: 0.2627 - precision_11: 0.9618 - recall_11: 0.9570 -
val_accuracy: 0.9091 - val_loss: 0.3855 - val_precision_11: 0.9119 -
val_recall_11: 0.9075 - learning_rate: 1.0000e-05
Epoch 56/100
79/79             162s 2s/step -
accuracy: 0.9645 - loss: 0.2593 - precision_11: 0.9650 - recall_11: 0.9626 -
val_accuracy: 0.9139 - val_loss: 0.3929 - val_precision_11: 0.9153 -
val_recall_11: 0.9139 - learning_rate: 1.0000e-05
Epoch 57/100
79/79             162s 2s/step -
accuracy: 0.9557 - loss: 0.2736 - precision_11: 0.9569 - recall_11: 0.9536 -
val_accuracy: 0.9155 - val_loss: 0.3843 - val_precision_11: 0.9168 -
val_recall_11: 0.9139 - learning_rate: 1.0000e-05
Epoch 58/100
79/79             163s 2s/step -
accuracy: 0.9550 - loss: 0.2691 - precision_11: 0.9592 - recall_11: 0.9538 -
val_accuracy: 0.9155 - val_loss: 0.3859 - val_precision_11: 0.9183 -
val_recall_11: 0.9139 - learning_rate: 1.0000e-05
```

5.0.3 Plot accuracy for training and validation wrt epoch

[194]:
```
plot_accuracy(model_fit)
```



5.0.4 Tensorboard

[195]:
```
%load_ext tensorboard
```

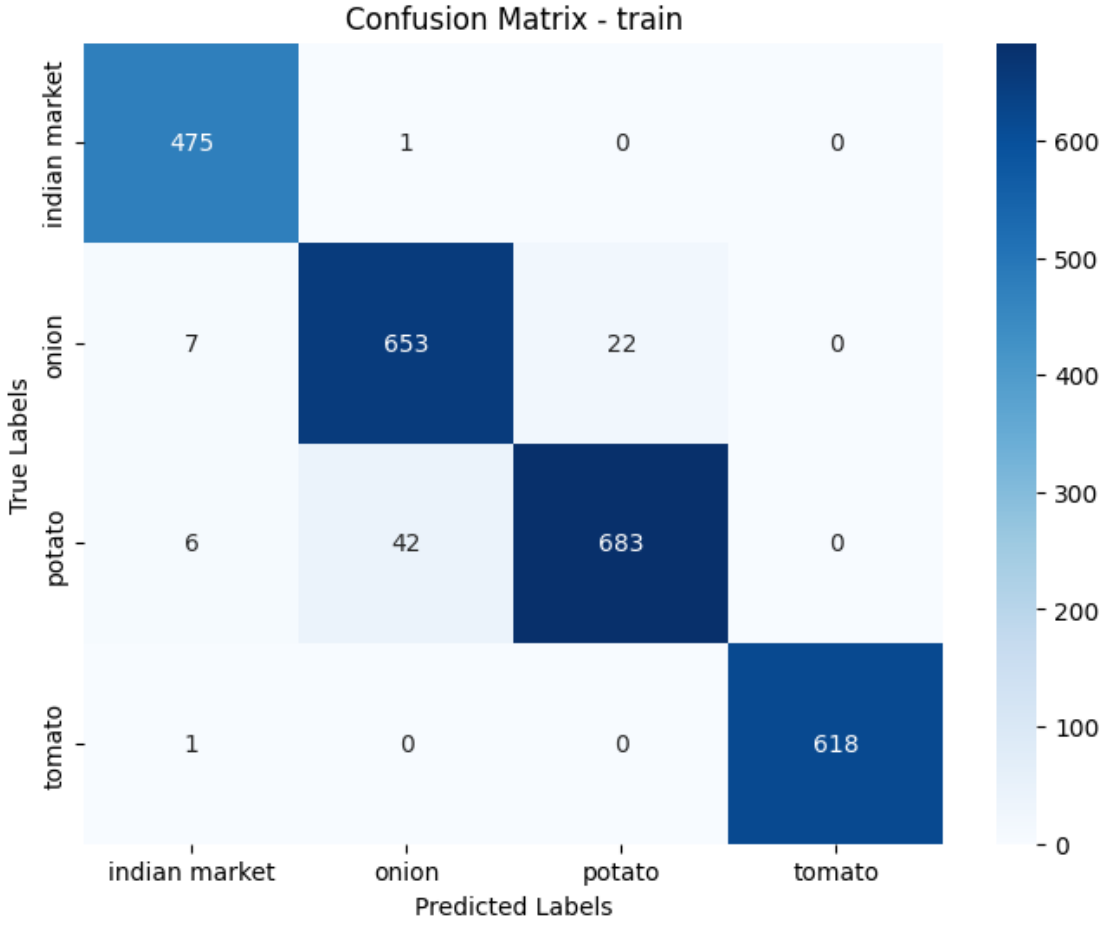
```
The tensorboard extension is already loaded. To reload it, use:
  %reload_ext tensorboard
```

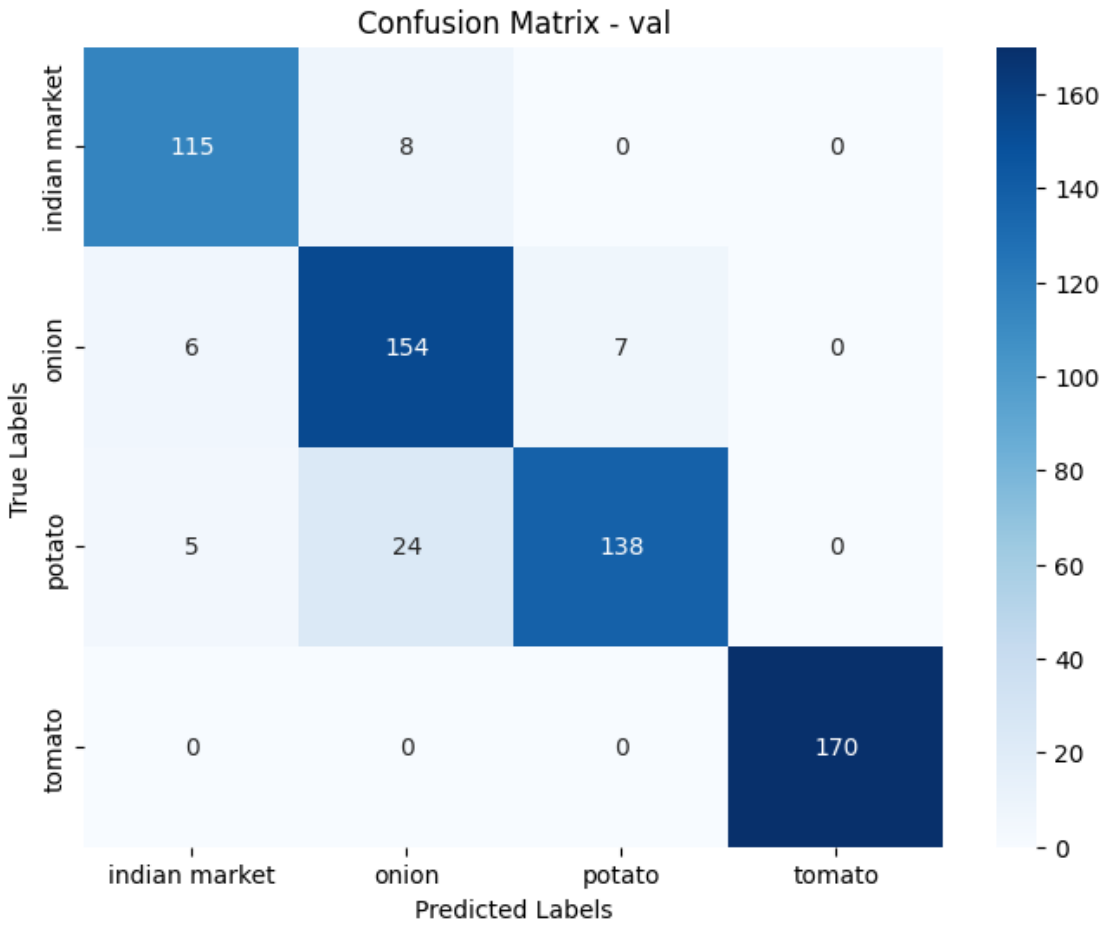
5.0.5 Model Evaluation, Mlflow Logging and Printing Metrics

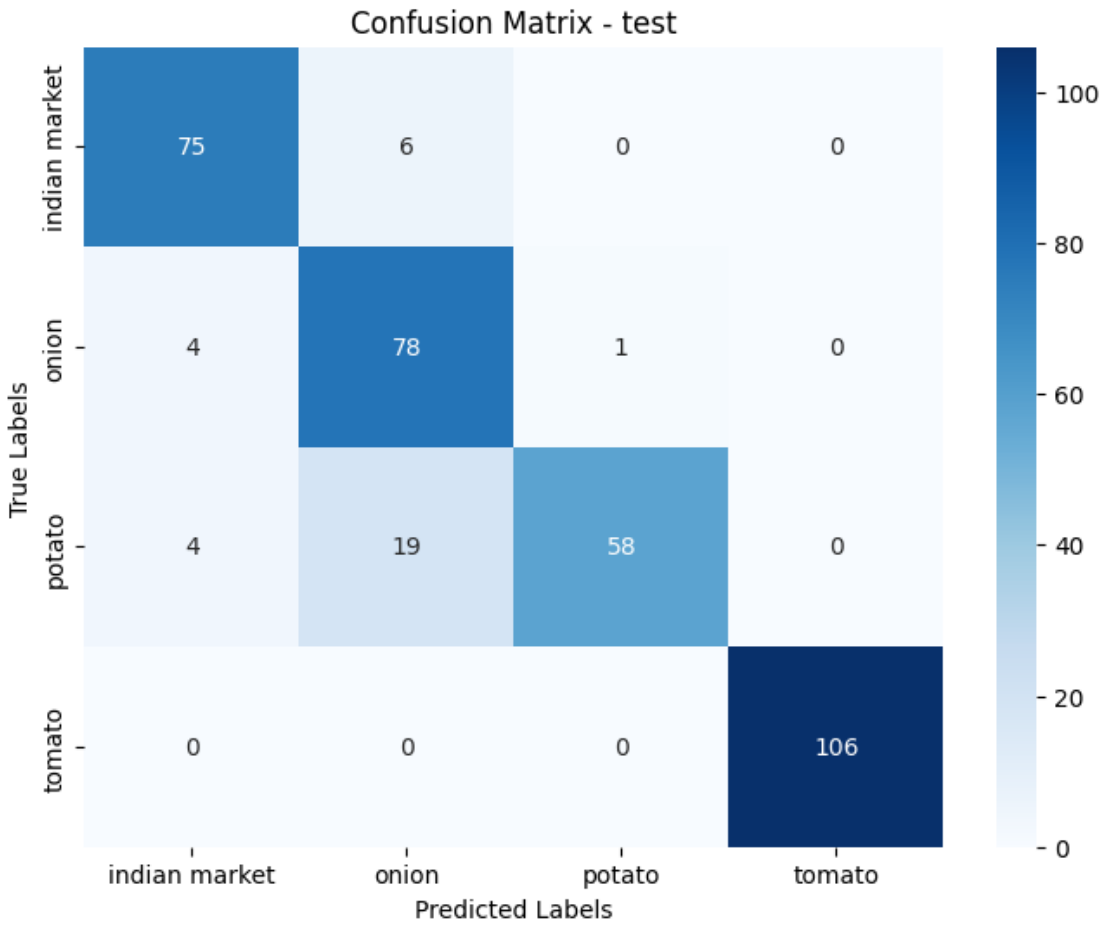
[196]:
```
evaluate_model(model, train_aug_ds, val_aug_ds, test_aug_ds,␣
↪class_names,run_name=f"{model.name}_aug",ckpt_path= f"checkpoints/{model.
↪name}_aug.weights.h5")
```

```
2025/02/09 03:36:22 WARNING mlflow.tensorflow: You are saving a TensorFlow Core
model or Keras model without a signature. Inference with
mlflow.pyfunc.spark_udf() will not work unless the model's pyfunc representation
accepts pandas DataFrames as inference inputs.
2025/02/09 03:36:32 WARNING mlflow.models.model: Model logged without a
signature and input example. Please set `input_example` parameter when logging
the model to auto infer the model signature.
```

```
train Metrics:
 Accuracy: 96.85%
 Precision: 96.96%
 Recall: 97.20%

val Metrics:
 Accuracy: 92.03%
 Precision: 92.31%
 Recall: 92.09%

test Metrics:
 Accuracy: 90.31%
 Precision: 91.10%
 Recall: 89.54%
```




Confusion Matrix - val
True Labels
Predicted Labels
indian market
onion
potato
tomato
115
8
0
0
6
154
7
0
5
24
138
0
0
0
0
170
160
140
120
100
80
60
40
20
0



Classification Report - train

	precision	recall	f1-score	support
indian market	0.97	1.00	0.98	476
onion	0.94	0.96	0.95	682
potato	0.97	0.93	0.95	731
tomato	1.00	1.00	1.00	619
accuracy			0.97	2508
macro avg	0.97	0.97	0.97	2508
weighted avg	0.97	0.97	0.97	2508

Classification Report - val

	precision	recall	f1-score	support
indian market	0.91	0.93	0.92	123

```
        onion       0.83      0.92      0.87       167
       potato       0.95      0.83      0.88       167
       tomato       1.00      1.00      1.00       170

     accuracy                           0.92       627
    macro avg       0.92      0.92      0.92       627
 weighted avg       0.92      0.92      0.92       627


Classification Report - test
               precision    recall  f1-score   support

indian market       0.90      0.93      0.91        81
        onion       0.76      0.94      0.84        83
       potato       0.98      0.72      0.83        81
       tomato       1.00      1.00      1.00       106

     accuracy                           0.90       351
    macro avg       0.91      0.90      0.90       351
 weighted avg       0.92      0.90      0.90       351


Class-wise Accuracy:

Train Class-wise Accuracy:
 indian market: 99.79% (475/476)
 onion: 95.75% (653/682)
 potato: 93.43% (683/731)
 tomato: 99.84% (618/619)

Val Class-wise Accuracy:
 indian market: 93.50% (115/123)
 onion: 92.22% (154/167)
 potato: 82.63% (138/167)
 tomato: 100.00% (170/170)

Test Class-wise Accuracy:
 indian market: 92.59% (75/81)
 onion: 93.98% (78/83)
 potato: 71.60% (58/81)
 tomato: 100.00% (106/106)

Random Test Predictions:
```

True: indian market
Pred: indian market

True: indian market
Pred: onion

True: onion
Pred: onion

True: onion
Pred: onion

True: potato
Pred: potato

True: potato
Pred: potato

True: tomato
Pred: tomato

True: tomato
Pred: tomato

5.0.6 Save the model

[197]:
```
model.save("saved_models/complex_cnn_aug_model.keras")
```

Observations
- **Impact on Complex CNN**:
- Without Augmentation: Test Accuracy = **94.30%**.
- With Augmentation: Test Accuracy = **90.31%** (slight drop but better robustness). - With Augmentation: Training Accuracy = **96.85%** and Validation Accuracy = **92.03%** (Better generalisation with data augmentation) - Epochs = **58** - Training time = **160 min (approx)**. - Augmentation reduced overfitting but introduced variability, requiring more epochs to improve the accuracies.
- **Key Augmentation Techniques**:
- Random cropping,translation, rotation, brightness, contrast adjustments with ±10% will simulate real-world variability. - More augmentation will bring accuracy down.

6 PRE TRAINED MODELS

6.1 VGG16

6.1.1 Building the model

[77]:
```
def pretrained_vgg16(height=256, width=256, num_classes=4, trainable_layers=0,␣
↪ckpt_path=None):
    base_model = applications.VGG16(weights="imagenet", include_top=False,␣
↪input_shape=(height, width, 3))

    # Freeze all layers initially
    base_model.trainable = False

    # If fine-tuning, unfreeze last `trainable_layers`
    if trainable_layers > 0:
        for layer in base_model.layers[-trainable_layers:]:
            layer.trainable = True

    model = keras.Sequential([
        base_model,
        layers.GlobalAveragePooling2D(),
        layers.Dense(512, activation="relu", kernel_regularizer=regularizers.
↪l2(5e-4)),
        layers.BatchNormalization(),
        layers.Dropout(0.3),

        layers.Dense(256, activation="relu", kernel_regularizer=regularizers.
↪l2(5e-4)),
        layers.BatchNormalization(),

        layers.Dense(num_classes, activation="softmax")
```

```
    ], name="pretrained_vgg16")

    # Load weights only if a valid checkpoint is provided
    if ckpt_path and os.path.exists(ckpt_path):
        print(f"Loading weights from {ckpt_path}")
        model.load_weights(ckpt_path)

    return model
```

[17]:
```
ckpt_path = "checkpoints/1_pretrained_vgg16_trainable0.weights.h5"
log_dir = "logs/1_pretrained_vgg16_trainable0"

# Load pretrained VGG16 model
model = pretrained_vgg16(height=256, width=256, num_classes=4,␣
 ↪trainable_layers=0,ckpt_path=ckpt_path)
```

[22]:
```
model.summary()
```

Model: "pretrained_vgg16"

Layer (type)	Output Shape	Param #
vgg16 (Functional)	(None, 8, 8, 512)	14,714,688
global_average_pooling2d_2 (GlobalAveragePooling2D)	(None, 512)	0
dense_6 (Dense)	(None, 512)	262,656
batch_normalization_4 (BatchNormalization)	(None, 512)	2,048
dropout_2 (Dropout)	(None, 512)	0
dense_7 (Dense)	(None, 256)	131,328
batch_normalization_5 (BatchNormalization)	(None, 256)	1,024
dense_8 (Dense)	(None, 4)	1,028

Total params: 15,905,870 (60.68 MB)

Trainable params: 396,548 (1.51 MB)

Non-trainable params: 14,716,224 (56.14 MB)

Optimizer params: 793,098 (3.03 MB)

6.1.2 Model Compilation and Training

[23]:
```
model_fit = compile_train(model, train_ds, val_ds,␣
↪log_dir=log_dir,epochs=100,ckpt_path=ckpt_path)
```

```
Epoch 1/100
79/79             479s 6s/step -
accuracy: 0.8973 - loss: 0.5742 - precision_1: 0.9081 - recall_1: 0.8901 -
val_accuracy: 0.7767 - val_loss: 0.7906 - val_precision_1: 0.8390 -
val_recall_1: 0.7065 - learning_rate: 0.0010
Epoch 2/100
79/79             474s 6s/step -
accuracy: 0.9092 - loss: 0.4707 - precision_1: 0.9153 - recall_1: 0.9064 -
val_accuracy: 0.8772 - val_loss: 0.5810 - val_precision_1: 0.9008 -
val_recall_1: 0.8549 - learning_rate: 0.0010
Epoch 3/100
79/79             471s 6s/step -
accuracy: 0.9406 - loss: 0.4082 - precision_1: 0.9477 - recall_1: 0.9373 -
val_accuracy: 0.9298 - val_loss: 0.4384 - val_precision_1: 0.9338 -
val_recall_1: 0.9219 - learning_rate: 0.0010
Epoch 4/100
79/79             471s 6s/step -
accuracy: 0.9468 - loss: 0.3834 - precision_1: 0.9528 - recall_1: 0.9408 -
val_accuracy: 0.8931 - val_loss: 0.4756 - val_precision_1: 0.8957 -
val_recall_1: 0.8900 - learning_rate: 0.0010
Epoch 5/100
79/79             471s 6s/step -
accuracy: 0.9407 - loss: 0.3766 - precision_1: 0.9459 - recall_1: 0.9344 -
val_accuracy: 0.9075 - val_loss: 0.4503 - val_precision_1: 0.9087 -
val_recall_1: 0.9043 - learning_rate: 0.0010
Epoch 6/100
79/79             471s 6s/step -
accuracy: 0.9494 - loss: 0.3421 - precision_1: 0.9512 - recall_1: 0.9452 -
val_accuracy: 0.9107 - val_loss: 0.4315 - val_precision_1: 0.9149 -
val_recall_1: 0.9091 - learning_rate: 0.0010
Epoch 7/100
79/79             471s 6s/step -
accuracy: 0.9525 - loss: 0.3260 - precision_1: 0.9552 - recall_1: 0.9494 -
val_accuracy: 0.8628 - val_loss: 0.8039 - val_precision_1: 0.8626 -
val_recall_1: 0.8612 - learning_rate: 0.0010
Epoch 8/100
```

```
79/79            407s 5s/step -
accuracy: 0.9441 - loss: 0.3440 - precision_1: 0.9441 - recall_1: 0.9408 -
val_accuracy: 0.9282 - val_loss: 0.4121 - val_precision_1: 0.9296 -
val_recall_1: 0.9266 - learning_rate: 0.0010
Epoch 9/100
79/79            287s 4s/step -
accuracy: 0.9536 - loss: 0.3156 - precision_1: 0.9580 - recall_1: 0.9529 -
val_accuracy: 0.9027 - val_loss: 0.4871 - val_precision_1: 0.9054 -
val_recall_1: 0.9011 - learning_rate: 0.0010
Epoch 10/100
79/79            287s 4s/step -
accuracy: 0.9638 - loss: 0.2892 - precision_1: 0.9649 - recall_1: 0.9617 -
val_accuracy: 0.9346 - val_loss: 0.3636 - val_precision_1: 0.9361 -
val_recall_1: 0.9346 - learning_rate: 0.0010
Epoch 11/100
79/79            289s 4s/step -
accuracy: 0.9561 - loss: 0.2855 - precision_1: 0.9579 - recall_1: 0.9544 -
val_accuracy: 0.9426 - val_loss: 0.3630 - val_precision_1: 0.9424 -
val_recall_1: 0.9394 - learning_rate: 0.0010
Epoch 12/100
79/79            416s 5s/step -
accuracy: 0.9720 - loss: 0.2414 - precision_1: 0.9734 - recall_1: 0.9720 -
val_accuracy: 0.9266 - val_loss: 0.4415 - val_precision_1: 0.9309 -
val_recall_1: 0.9234 - learning_rate: 0.0010
Epoch 13/100
79/79            469s 6s/step -
accuracy: 0.9793 - loss: 0.2232 - precision_1: 0.9804 - recall_1: 0.9792 -
val_accuracy: 0.8979 - val_loss: 0.4224 - val_precision_1: 0.8994 -
val_recall_1: 0.8979 - learning_rate: 0.0010
Epoch 14/100
79/79            470s 6s/step -
accuracy: 0.9723 - loss: 0.2364 - precision_1: 0.9740 - recall_1: 0.9709 -
val_accuracy: 0.8884 - val_loss: 0.5179 - val_precision_1: 0.8892 -
val_recall_1: 0.8836 - learning_rate: 0.0010
Epoch 15/100
79/79            440s 6s/step -
accuracy: 0.9616 - loss: 0.2490 - precision_1: 0.9628 - recall_1: 0.9607 -
val_accuracy: 0.8963 - val_loss: 0.4918 - val_precision_1: 0.8973 -
val_recall_1: 0.8915 - learning_rate: 0.0010
Epoch 16/100
79/79            274s 3s/step -
accuracy: 0.9638 - loss: 0.2560 - precision_1: 0.9642 - recall_1: 0.9636 -
val_accuracy: 0.9346 - val_loss: 0.3729 - val_precision_1: 0.9359 -
val_recall_1: 0.9314 - learning_rate: 0.0010
Epoch 17/100
79/79            273s 3s/step -
accuracy: 0.9793 - loss: 0.2161 - precision_1: 0.9800 - recall_1: 0.9781 -
val_accuracy: 0.9506 - val_loss: 0.3182 - val_precision_1: 0.9521 -
```

```
val_recall_1: 0.9506 - learning_rate: 3.0000e-04
Epoch 18/100
79/79            275s 3s/step -
accuracy: 0.9815 - loss: 0.2041 - precision_1: 0.9833 - recall_1: 0.9794 -
val_accuracy: 0.9458 - val_loss: 0.3351 - val_precision_1: 0.9458 -
val_recall_1: 0.9458 - learning_rate: 3.0000e-04
Epoch 19/100
79/79            274s 3s/step -
accuracy: 0.9884 - loss: 0.1904 - precision_1: 0.9899 - recall_1: 0.9884 -
val_accuracy: 0.9442 - val_loss: 0.3280 - val_precision_1: 0.9442 -
val_recall_1: 0.9442 - learning_rate: 3.0000e-04
Epoch 20/100
79/79            274s 3s/step -
accuracy: 0.9889 - loss: 0.1786 - precision_1: 0.9899 - recall_1: 0.9889 -
val_accuracy: 0.9458 - val_loss: 0.3066 - val_precision_1: 0.9472 -
val_recall_1: 0.9442 - learning_rate: 3.0000e-04
Epoch 21/100
79/79            275s 3s/step -
accuracy: 0.9915 - loss: 0.1733 - precision_1: 0.9915 - recall_1: 0.9915 -
val_accuracy: 0.9490 - val_loss: 0.2986 - val_precision_1: 0.9490 -
val_recall_1: 0.9490 - learning_rate: 3.0000e-04
Epoch 22/100
79/79            274s 3s/step -
accuracy: 0.9910 - loss: 0.1723 - precision_1: 0.9910 - recall_1: 0.9901 -
val_accuracy: 0.9330 - val_loss: 0.3462 - val_precision_1: 0.9360 -
val_recall_1: 0.9330 - learning_rate: 3.0000e-04
Epoch 23/100
79/79            275s 3s/step -
accuracy: 0.9919 - loss: 0.1644 - precision_1: 0.9919 - recall_1: 0.9919 -
val_accuracy: 0.9490 - val_loss: 0.3202 - val_precision_1: 0.9505 -
val_recall_1: 0.9490 - learning_rate: 3.0000e-04
Epoch 24/100
79/79            436s 6s/step -
accuracy: 0.9865 - loss: 0.1777 - precision_1: 0.9881 - recall_1: 0.9843 -
val_accuracy: 0.9458 - val_loss: 0.3163 - val_precision_1: 0.9488 -
val_recall_1: 0.9458 - learning_rate: 3.0000e-04
Epoch 25/100
79/79            471s 6s/step -
accuracy: 0.9926 - loss: 0.1594 - precision_1: 0.9926 - recall_1: 0.9926 -
val_accuracy: 0.9458 - val_loss: 0.2891 - val_precision_1: 0.9458 -
val_recall_1: 0.9458 - learning_rate: 3.0000e-04
Epoch 26/100
79/79            472s 6s/step -
accuracy: 0.9919 - loss: 0.1573 - precision_1: 0.9932 - recall_1: 0.9902 -
val_accuracy: 0.9553 - val_loss: 0.2937 - val_precision_1: 0.9553 -
val_recall_1: 0.9537 - learning_rate: 3.0000e-04
Epoch 27/100
79/79            470s 6s/step -
```

```
accuracy: 0.9865 - loss: 0.1664 - precision_1: 0.9865 - recall_1: 0.9864 -
val_accuracy: 0.9458 - val_loss: 0.3039 - val_precision_1: 0.9458 -
val_recall_1: 0.9458 - learning_rate: 3.0000e-04
Epoch 28/100
79/79            330s 4s/step -
accuracy: 0.9930 - loss: 0.1474 - precision_1: 0.9934 - recall_1: 0.9919 -
val_accuracy: 0.9474 - val_loss: 0.3130 - val_precision_1: 0.9488 -
val_recall_1: 0.9458 - learning_rate: 3.0000e-04
Epoch 29/100
79/79            275s 3s/step -
accuracy: 0.9882 - loss: 0.1563 - precision_1: 0.9897 - recall_1: 0.9882 -
val_accuracy: 0.9490 - val_loss: 0.3073 - val_precision_1: 0.9488 -
val_recall_1: 0.9458 - learning_rate: 3.0000e-04
Epoch 30/100
79/79            275s 3s/step -
accuracy: 0.9925 - loss: 0.1407 - precision_1: 0.9925 - recall_1: 0.9925 -
val_accuracy: 0.9506 - val_loss: 0.3066 - val_precision_1: 0.9520 -
val_recall_1: 0.9490 - learning_rate: 3.0000e-04
Epoch 31/100
79/79            277s 4s/step -
accuracy: 0.9935 - loss: 0.1387 - precision_1: 0.9935 - recall_1: 0.9935 -
val_accuracy: 0.9506 - val_loss: 0.3072 - val_precision_1: 0.9506 -
val_recall_1: 0.9506 - learning_rate: 9.0000e-05
Epoch 32/100
79/79            275s 3s/step -
accuracy: 0.9971 - loss: 0.1323 - precision_1: 0.9971 - recall_1: 0.9971 -
val_accuracy: 0.9537 - val_loss: 0.2941 - val_precision_1: 0.9537 -
val_recall_1: 0.9522 - learning_rate: 9.0000e-05
Epoch 33/100
79/79            277s 4s/step -
accuracy: 0.9919 - loss: 0.1423 - precision_1: 0.9919 - recall_1: 0.9919 -
val_accuracy: 0.9537 - val_loss: 0.2781 - val_precision_1: 0.9553 -
val_recall_1: 0.9537 - learning_rate: 9.0000e-05
Epoch 34/100
79/79            289s 4s/step -
accuracy: 0.9918 - loss: 0.1364 - precision_1: 0.9918 - recall_1: 0.9917 -
val_accuracy: 0.9537 - val_loss: 0.2924 - val_precision_1: 0.9537 -
val_recall_1: 0.9537 - learning_rate: 9.0000e-05
Epoch 35/100
79/79            472s 6s/step -
accuracy: 0.9979 - loss: 0.1244 - precision_1: 0.9979 - recall_1: 0.9974 -
val_accuracy: 0.9569 - val_loss: 0.2928 - val_precision_1: 0.9567 -
val_recall_1: 0.9522 - learning_rate: 9.0000e-05
Epoch 36/100
79/79            473s 6s/step -
accuracy: 0.9968 - loss: 0.1277 - precision_1: 0.9968 - recall_1: 0.9968 -
val_accuracy: 0.9537 - val_loss: 0.2858 - val_precision_1: 0.9537 -
val_recall_1: 0.9522 - learning_rate: 9.0000e-05
```

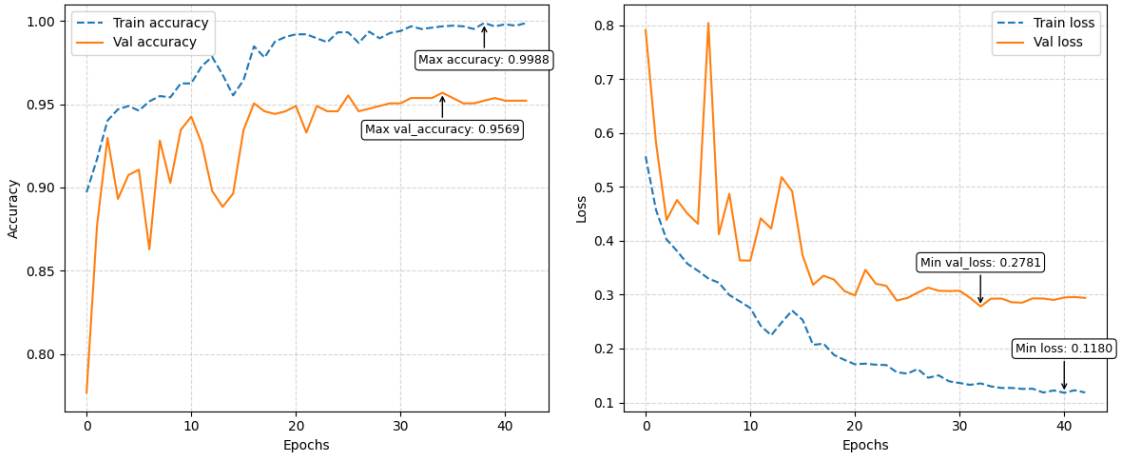
```
Epoch 37/100
79/79             474s 6s/step -
accuracy: 0.9988 - loss: 0.1246 - precision_1: 0.9988 - recall_1: 0.9985 -
val_accuracy: 0.9506 - val_loss: 0.2852 - val_precision_1: 0.9521 -
val_recall_1: 0.9506 - learning_rate: 9.0000e-05
Epoch 38/100
79/79             472s 6s/step -
accuracy: 0.9945 - loss: 0.1266 - precision_1: 0.9946 - recall_1: 0.9945 -
val_accuracy: 0.9506 - val_loss: 0.2931 - val_precision_1: 0.9521 -
val_recall_1: 0.9506 - learning_rate: 9.0000e-05
Epoch 39/100
79/79             422s 5s/step -
accuracy: 0.9991 - loss: 0.1188 - precision_1: 0.9997 - recall_1: 0.9991 -
val_accuracy: 0.9522 - val_loss: 0.2928 - val_precision_1: 0.9536 -
val_recall_1: 0.9506 - learning_rate: 2.7000e-05
Epoch 40/100
79/79             273s 3s/step -
accuracy: 0.9967 - loss: 0.1241 - precision_1: 0.9977 - recall_1: 0.9967 -
val_accuracy: 0.9537 - val_loss: 0.2901 - val_precision_1: 0.9537 -
val_recall_1: 0.9537 - learning_rate: 2.7000e-05
Epoch 41/100
79/79             277s 4s/step -
accuracy: 0.9975 - loss: 0.1183 - precision_1: 0.9975 - recall_1: 0.9975 -
val_accuracy: 0.9522 - val_loss: 0.2948 - val_precision_1: 0.9521 -
val_recall_1: 0.9506 - learning_rate: 2.7000e-05
Epoch 42/100
79/79             279s 4s/step -
accuracy: 0.9973 - loss: 0.1225 - precision_1: 0.9973 - recall_1: 0.9973 -
val_accuracy: 0.9522 - val_loss: 0.2956 - val_precision_1: 0.9521 -
val_recall_1: 0.9506 - learning_rate: 2.7000e-05
Epoch 43/100
79/79             276s 3s/step -
accuracy: 0.9996 - loss: 0.1180 - precision_1: 0.9996 - recall_1: 0.9996 -
val_accuracy: 0.9522 - val_loss: 0.2939 - val_precision_1: 0.9521 -
val_recall_1: 0.9506 - learning_rate: 2.7000e-05
```

6.1.3 Plot loss & accuracy for training and validation wrt epoch

[ ]:
```python
# Function to annotate max accuracy and min loss
def annot_max_min(x, y, metric, ax=None):
    if not ax:
        ax = plt.gca()

    idx = np.argmax(y) if "accuracy" in metric else np.argmin(y)
    text = f"{'Max' if 'accuracy' in metric else 'Min'} {metric}: {y[idx]:.4f}"

    # Get y-axis limits
```

```
    ymin, ymax = ax.get_ylim()
    range_y = ymax - ymin  # Total height of the plot

    # Dynamic offset to avoid overlap
    offset = 0.1 * range_y  # 10% of total height

    # Move annotation position
    if "accuracy" in metric:
        xytext_offset = (idx, y[idx] - offset)  # Move below for accuracy
    else:
        xytext_offset = (idx, y[idx] + offset)  # Move above for loss

    ax.annotate(text, xy=(idx, y[idx]), xytext=xytext_offset,
                textcoords="data",
                arrowprops=dict(arrowstyle="->", connectionstyle="angle3"),
                bbox=dict(boxstyle="round,pad=0.3", fc="w", ec="k", lw=0.8),
                fontsize=9, ha="center")

# Function to plot training curves
def plot_loss_and_accuracy(metrics, history):
    fig, ax = plt.subplots(1, len(metrics), figsize=(6 * len(metrics), 5))

    if len(metrics) == 1:
        ax = [ax]

    for idx, metric in enumerate(metrics):
        x = range(len(history.history[metric]))
        y_train = history.history[metric]
        y_val = history.history["val_" + metric]

        ax[idx].plot(x, y_train, linestyle="dashed", label=f"Train {metric}")
        ax[idx].plot(x, y_val, label=f"Val {metric}")

        # Annotate max accuracy & min loss with dynamic positioning
        annot_max_min(x, y_train, metric, ax[idx])
        annot_max_min(x, y_val, "val_" + metric, ax[idx])

        ax[idx].set_xlabel("Epochs")
        ax[idx].set_ylabel(metric.capitalize())
        ax[idx].legend()
        ax[idx].grid(True, linestyle="--", alpha=0.5)

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

[48]:
```
plot_loss_and_accuracy(["accuracy","loss"],model_fit)
```



6.1.4 Tensorboard

[49]:
```
%load_ext tensorboard
```

6.1.5 Model Evaluation, Mlflow Logging and Printing Metrics

[50]:
```
ckpt_path
```

[50]: 'checkpoints/1_pretrained_vgg16_trainable0.weights.h5'

[51]:
```
log_dir
```

[51]: 'logs/1_pretrained_vgg16_trainable0'

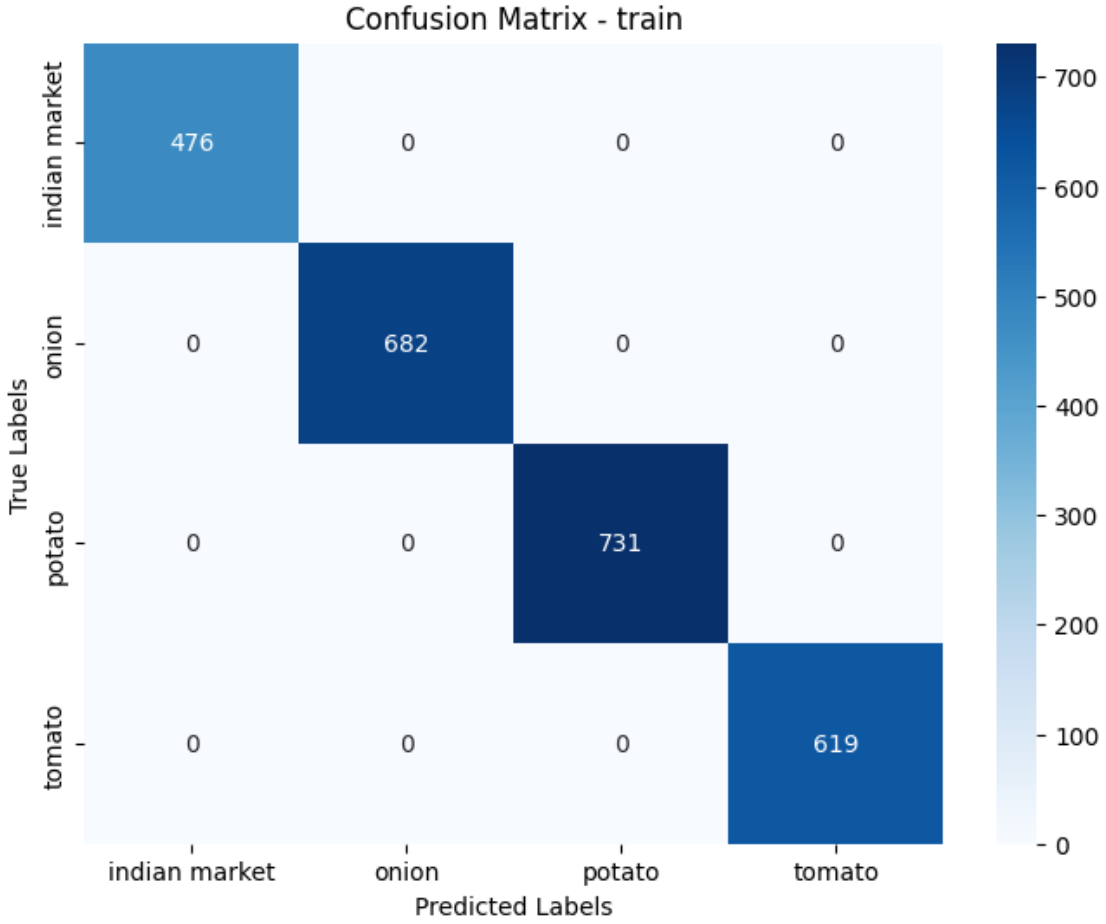
[52]:
```
run_name = "1_pretrained_vgg16_trainable0"
run_name
```

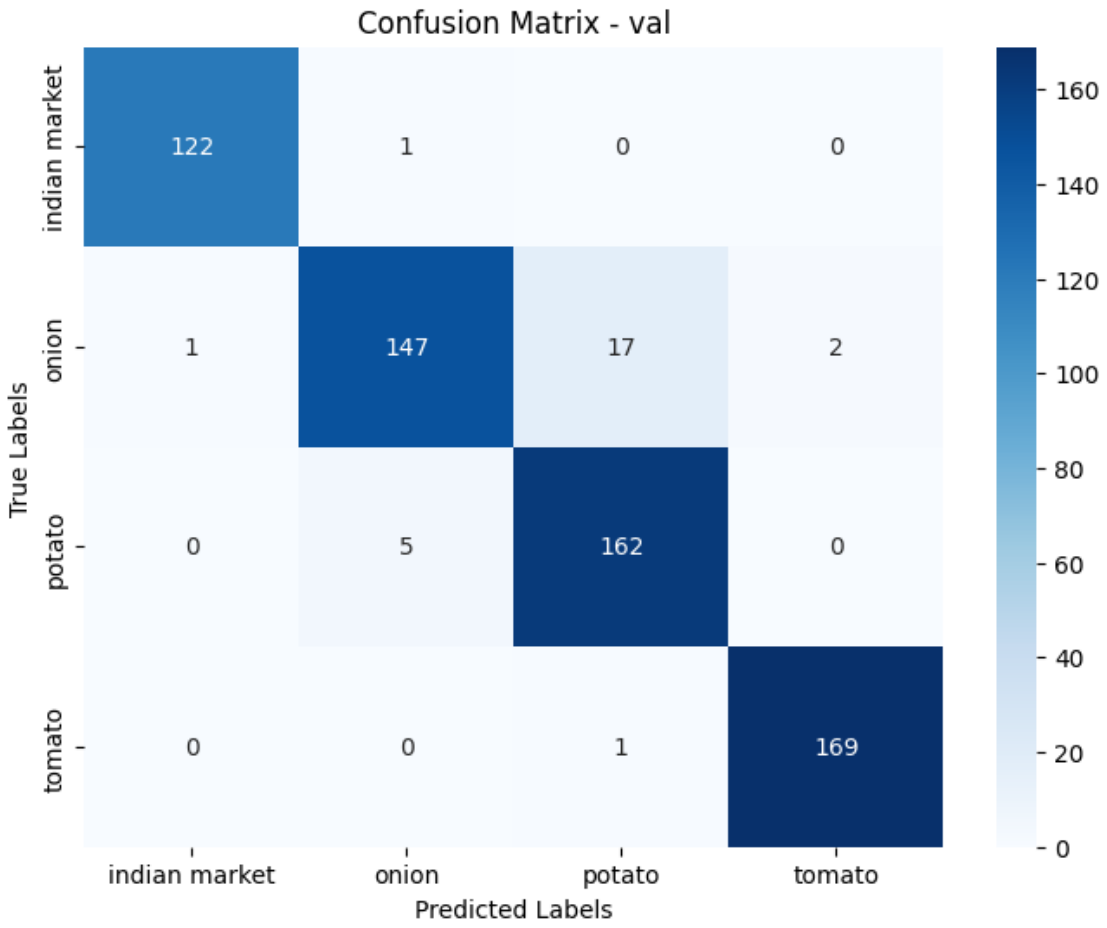
[52]: '1_pretrained_vgg16_trainable0'

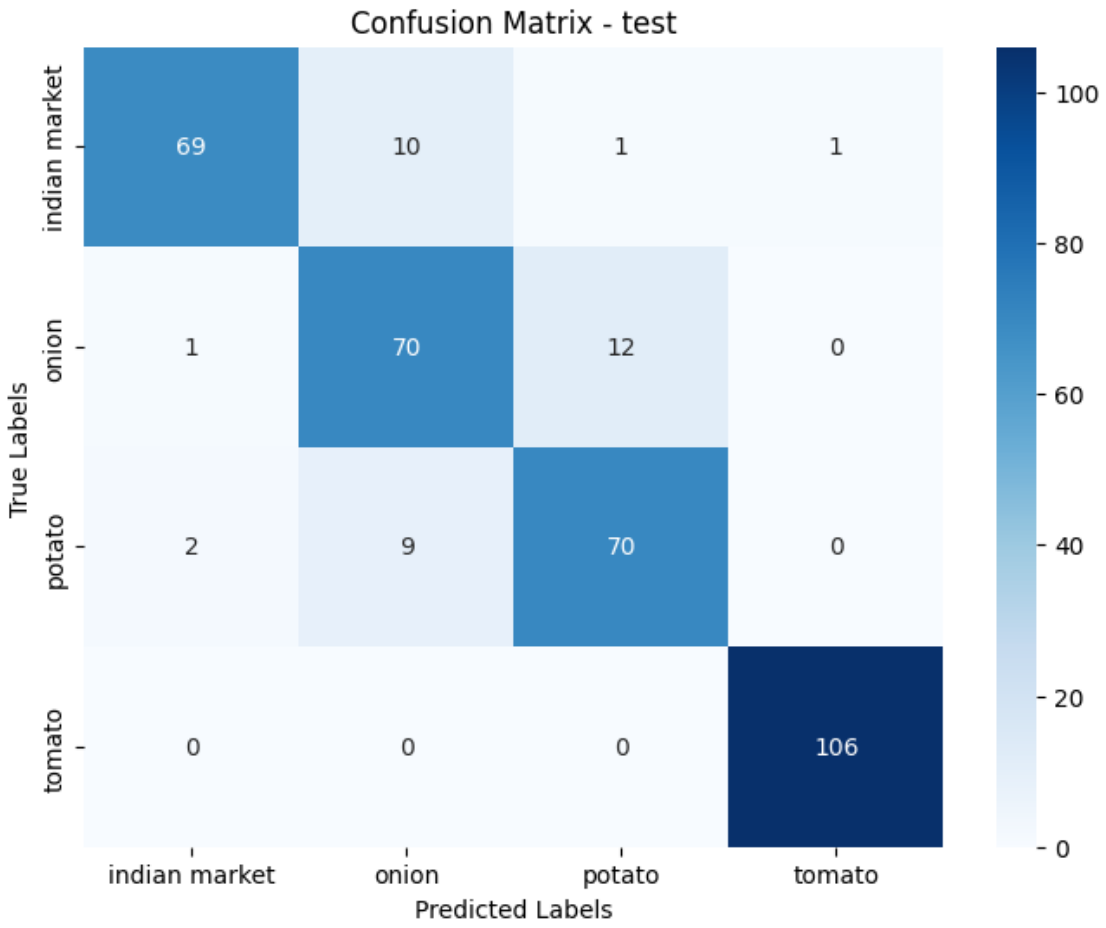
[53]:
```
evaluate_model(model, train_ds, val_ds, test_ds,␣
↪class_names,run_name=run_name,ckpt_path= ckpt_path)
```

```
2025/02/10 16:32:09 WARNING mlflow.tensorflow: You are saving a TensorFlow Core
model or Keras model without a signature. Inference with
mlflow.pyfunc.spark_udf() will not work unless the model's pyfunc representation
accepts pandas DataFrames as inference inputs.
2025/02/10 16:32:45 WARNING mlflow.models.model: Model logged without a
signature and input example. Please set `input_example` parameter when logging
the model to auto infer the model signature.
```

```
train Metrics:
 Accuracy: 100.00%
 Precision: 100.00%
 Recall: 100.00%

val Metrics:
 Accuracy: 95.69%
 Precision: 96.02%
 Recall: 95.91%

test Metrics:
 Accuracy: 89.74%
 Precision: 89.47%
 Recall: 88.99%
```




Confusion Matrix - val
indian market
onion
potato
tomato
True Labels
122
1
0
0
1
147
17
2
0
5
162
0
0
0
1
169
indian market
onion
potato
tomato
Predicted Labels
160
140
120
100
80
60
40
20
0



Classification Report - train

	precision	recall	f1-score	support
indian market	1.00	1.00	1.00	476
onion	1.00	1.00	1.00	682
potato	1.00	1.00	1.00	731
tomato	1.00	1.00	1.00	619
accuracy			1.00	2508
macro avg	1.00	1.00	1.00	2508
weighted avg	1.00	1.00	1.00	2508

Classification Report - val

	precision	recall	f1-score	support
indian market	0.99	0.99	0.99	123

```
        onion       0.96      0.88      0.92       167
       potato       0.90      0.97      0.93       167
       tomato       0.99      0.99      0.99       170

     accuracy                           0.96       627
    macro avg       0.96      0.96      0.96       627
 weighted avg       0.96      0.96      0.96       627


Classification Report - test
               precision    recall  f1-score   support

indian market       0.96      0.85      0.90        81
        onion       0.79      0.84      0.81        83
       potato       0.84      0.86      0.85        81
       tomato       0.99      1.00      1.00       106

     accuracy                           0.90       351
    macro avg       0.89      0.89      0.89       351
 weighted avg       0.90      0.90      0.90       351


Class-wise Accuracy:

Train Class-wise Accuracy:
 indian market: 100.00% (476/476)
 onion: 100.00% (682/682)
 potato: 100.00% (731/731)
 tomato: 100.00% (619/619)

Val Class-wise Accuracy:
 indian market: 99.19% (122/123)
 onion: 88.02% (147/167)
 potato: 97.01% (162/167)
 tomato: 99.41% (169/170)

Test Class-wise Accuracy:
 indian market: 85.19% (69/81)
 onion: 84.34% (70/83)
 potato: 86.42% (70/81)
 tomato: 100.00% (106/106)

Random Test Predictions:
```

True: indian market
Pred: indian market

True: indian market
Pred: indian market

True: onion
Pred: onion

True: onion
Pred: potato

True: potato
Pred: onion

True: potato
Pred: potato

True: tomato
Pred: tomato

True: tomato
Pred: tomato

6.1.6 Save the model

[54]:
```
model.save("saved_models/1_pretrained_vgg16_trainable0_model.keras")
```

6.2 DATA AUGMENTED VGG16

6.2.1 Load the previously trained model architecture and weights

[55]:
```
# Load the entire model (architecture + weights)
model = keras.models.load_model("saved_models/
  ↪1_pretrained_vgg16_trainable0_model.keras")
```

[62]:
```
model.summary()
```

Model: "pretrained_vgg16"

Layer (type)	Output Shape	Param #
vgg16 (Functional)	(None, 8, 8, 512)	14,714,688
global_average_pooling2d_2 (GlobalAveragePooling2D)	(None, 512)	0
dense_6 (Dense)	(None, 512)	262,656
batch_normalization_4 (BatchNormalization)	(None, 512)	2,048
dropout_2 (Dropout)	(None, 512)	0
dense_7 (Dense)	(None, 256)	131,328
batch_normalization_5 (BatchNormalization)	(None, 256)	1,024
dense_8 (Dense)	(None, 4)	1,028

Total params: 15,905,870 (60.68 MB)

Trainable params: 396,548 (1.51 MB)

Non-trainable params: 14,716,224 (56.14 MB)

```
Optimizer params: 793,098 (3.03 MB)
```

6.2.2 Model Compilation and Training

[78]:
```
ckpt_path = "checkpoints/2_pretrained_vgg16_aug_trainable0.weights.h5"
```

[79]:
```
log_dir = 'logs/1_pretrained_vgg16_aug_trainable0'
```

[80]:
```
run_name = '1_pretrained_vgg16_aug_trainable0'
```

[67]:
```
model_fit = compile_train(model, train_aug_ds, val_aug_ds,␣
↪log_dir=log_dir,epochs=20,ckpt_path=ckpt_path)
```

```
Epoch 1/20
79/79             486s 6s/step -
accuracy: 0.9189 - loss: 0.3640 - precision_2: 0.9238 - recall_2: 0.9141 -
val_accuracy: 0.9155 - val_loss: 0.3776 - val_precision_2: 0.9155 -
val_recall_2: 0.9155 - learning_rate: 0.0010
Epoch 2/20
79/79             479s 6s/step -
accuracy: 0.9289 - loss: 0.2944 - precision_2: 0.9336 - recall_2: 0.9269 -
val_accuracy: 0.9362 - val_loss: 0.3149 - val_precision_2: 0.9362 -
val_recall_2: 0.9362 - learning_rate: 0.0010
Epoch 3/20
79/79             478s 6s/step -
accuracy: 0.9367 - loss: 0.2796 - precision_2: 0.9407 - recall_2: 0.9353 -
val_accuracy: 0.9314 - val_loss: 0.3472 - val_precision_2: 0.9329 -
val_recall_2: 0.9314 - learning_rate: 0.0010
Epoch 4/20
79/79             479s 6s/step -
accuracy: 0.9429 - loss: 0.2564 - precision_2: 0.9469 - recall_2: 0.9403 -
val_accuracy: 0.9298 - val_loss: 0.3202 - val_precision_2: 0.9327 -
val_recall_2: 0.9282 - learning_rate: 0.0010
Epoch 5/20
79/79             294s 4s/step -
accuracy: 0.9236 - loss: 0.2949 - precision_2: 0.9295 - recall_2: 0.9229 -
val_accuracy: 0.9155 - val_loss: 0.3996 - val_precision_2: 0.9184 -
val_recall_2: 0.9155 - learning_rate: 0.0010
Epoch 6/20
79/79             273s 3s/step -
accuracy: 0.9346 - loss: 0.2642 - precision_2: 0.9366 - recall_2: 0.9299 -
val_accuracy: 0.9234 - val_loss: 0.3450 - val_precision_2: 0.9233 -
val_recall_2: 0.9219 - learning_rate: 0.0010
Epoch 7/20
79/79             296s 4s/step -
accuracy: 0.9403 - loss: 0.2739 - precision_2: 0.9458 - recall_2: 0.9390 -
val_accuracy: 0.9139 - val_loss: 0.3546 - val_precision_2: 0.9167 -
```

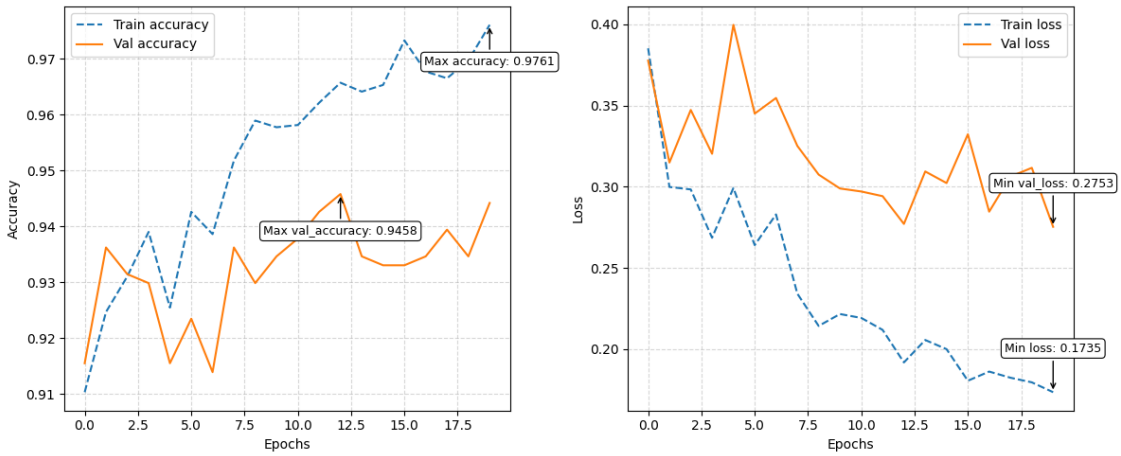
```
val_recall_2: 0.9123 - learning_rate: 0.0010
Epoch 8/20
79/79            282s 4s/step -
accuracy: 0.9456 - loss: 0.2394 - precision_2: 0.9515 - recall_2: 0.9406 -
val_accuracy: 0.9362 - val_loss: 0.3251 - val_precision_2: 0.9377 -
val_recall_2: 0.9362 - learning_rate: 3.0000e-04
Epoch 9/20
79/79            282s 4s/step -
accuracy: 0.9621 - loss: 0.2111 - precision_2: 0.9667 - recall_2: 0.9589 -
val_accuracy: 0.9298 - val_loss: 0.3074 - val_precision_2: 0.9312 -
val_recall_2: 0.9282 - learning_rate: 3.0000e-04
Epoch 10/20
79/79            341s 4s/step -
accuracy: 0.9579 - loss: 0.2220 - precision_2: 0.9624 - recall_2: 0.9574 -
val_accuracy: 0.9346 - val_loss: 0.2989 - val_precision_2: 0.9375 -
val_recall_2: 0.9330 - learning_rate: 3.0000e-04
Epoch 11/20
79/79            474s 6s/step -
accuracy: 0.9582 - loss: 0.2154 - precision_2: 0.9589 - recall_2: 0.9571 -
val_accuracy: 0.9378 - val_loss: 0.2970 - val_precision_2: 0.9392 -
val_recall_2: 0.9362 - learning_rate: 3.0000e-04
Epoch 12/20
79/79            473s 6s/step -
accuracy: 0.9665 - loss: 0.2012 - precision_2: 0.9670 - recall_2: 0.9656 -
val_accuracy: 0.9426 - val_loss: 0.2942 - val_precision_2: 0.9426 -
val_recall_2: 0.9426 - learning_rate: 3.0000e-04
Epoch 13/20
79/79            472s 6s/step -
accuracy: 0.9656 - loss: 0.1935 - precision_2: 0.9690 - recall_2: 0.9645 -
val_accuracy: 0.9458 - val_loss: 0.2771 - val_precision_2: 0.9473 -
val_recall_2: 0.9458 - learning_rate: 3.0000e-04
Epoch 14/20
79/79            471s 6s/step -
accuracy: 0.9571 - loss: 0.2210 - precision_2: 0.9580 - recall_2: 0.9555 -
val_accuracy: 0.9346 - val_loss: 0.3094 - val_precision_2: 0.9346 -
val_recall_2: 0.9346 - learning_rate: 3.0000e-04
Epoch 15/20
79/79            472s 6s/step -
accuracy: 0.9733 - loss: 0.1903 - precision_2: 0.9771 - recall_2: 0.9706 -
val_accuracy: 0.9330 - val_loss: 0.3022 - val_precision_2: 0.9359 -
val_recall_2: 0.9314 - learning_rate: 3.0000e-04
Epoch 16/20
79/79            472s 6s/step -
accuracy: 0.9683 - loss: 0.1820 - precision_2: 0.9732 - recall_2: 0.9664 -
val_accuracy: 0.9330 - val_loss: 0.3323 - val_precision_2: 0.9345 -
val_recall_2: 0.9330 - learning_rate: 3.0000e-04
Epoch 17/20
79/79            470s 6s/step -
```

```
accuracy: 0.9679 - loss: 0.1878 - precision_2: 0.9701 - recall_2: 0.9663 -
val_accuracy: 0.9346 - val_loss: 0.2846 - val_precision_2: 0.9360 -
val_recall_2: 0.9330 - learning_rate: 3.0000e-04
Epoch 18/20
79/79              299s 4s/step -
accuracy: 0.9738 - loss: 0.1732 - precision_2: 0.9745 - recall_2: 0.9722 -
val_accuracy: 0.9394 - val_loss: 0.3062 - val_precision_2: 0.9393 -
val_recall_2: 0.9378 - learning_rate: 3.0000e-04
Epoch 19/20
79/79              270s 3s/step -
accuracy: 0.9761 - loss: 0.1653 - precision_2: 0.9782 - recall_2: 0.9759 -
val_accuracy: 0.9346 - val_loss: 0.3116 - val_precision_2: 0.9360 -
val_recall_2: 0.9330 - learning_rate: 9.0000e-05
Epoch 20/20
79/79              272s 3s/step -
accuracy: 0.9773 - loss: 0.1744 - precision_2: 0.9791 - recall_2: 0.9761 -
val_accuracy: 0.9442 - val_loss: 0.2753 - val_precision_2: 0.9440 -
val_recall_2: 0.9410 - learning_rate: 9.0000e-05
```

[ ]:
```
model_fit = compile_train(model, train_aug_ds, val_aug_ds,␣
↪log_dir=log_dir,epochs=1,ckpt_path=ckpt_path)
```

6.2.3 Plot loss & accuracy for training and validation wrt epoch

[68]:
```
plot_loss_and_accuracy(["accuracy","loss"],model_fit)
```



6.2.4 Tensorboard

[69]:
```
%load_ext tensorboard
```

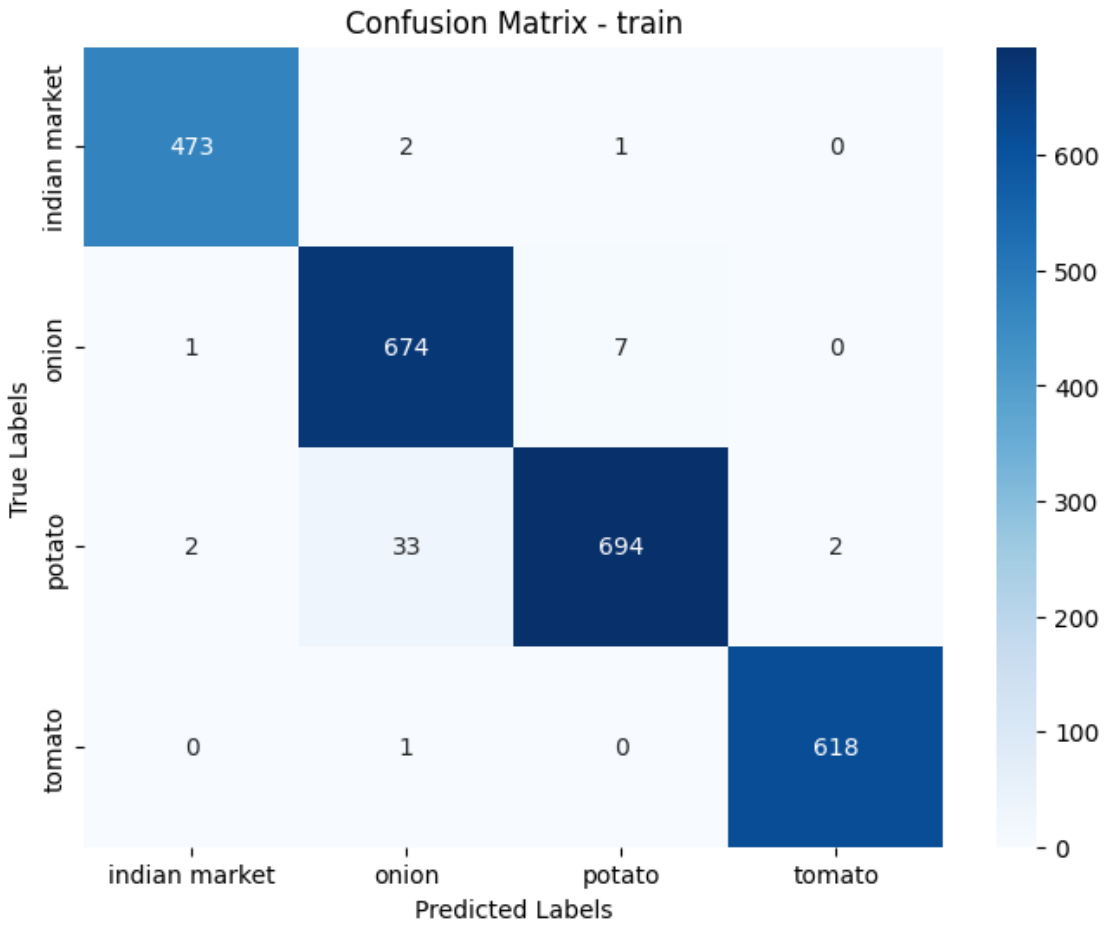
```
The tensorboard extension is already loaded. To reload it, use:
```

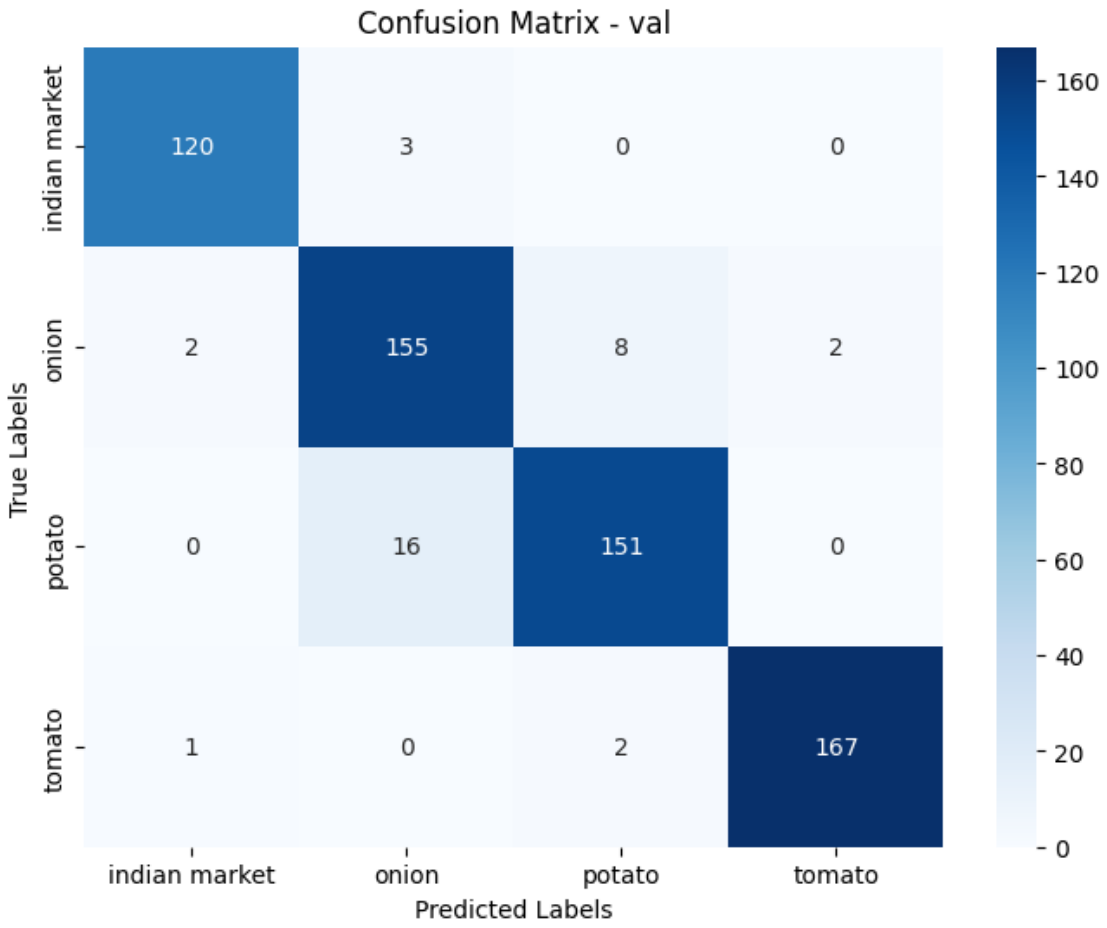
```
%reload_ext tensorboard
```

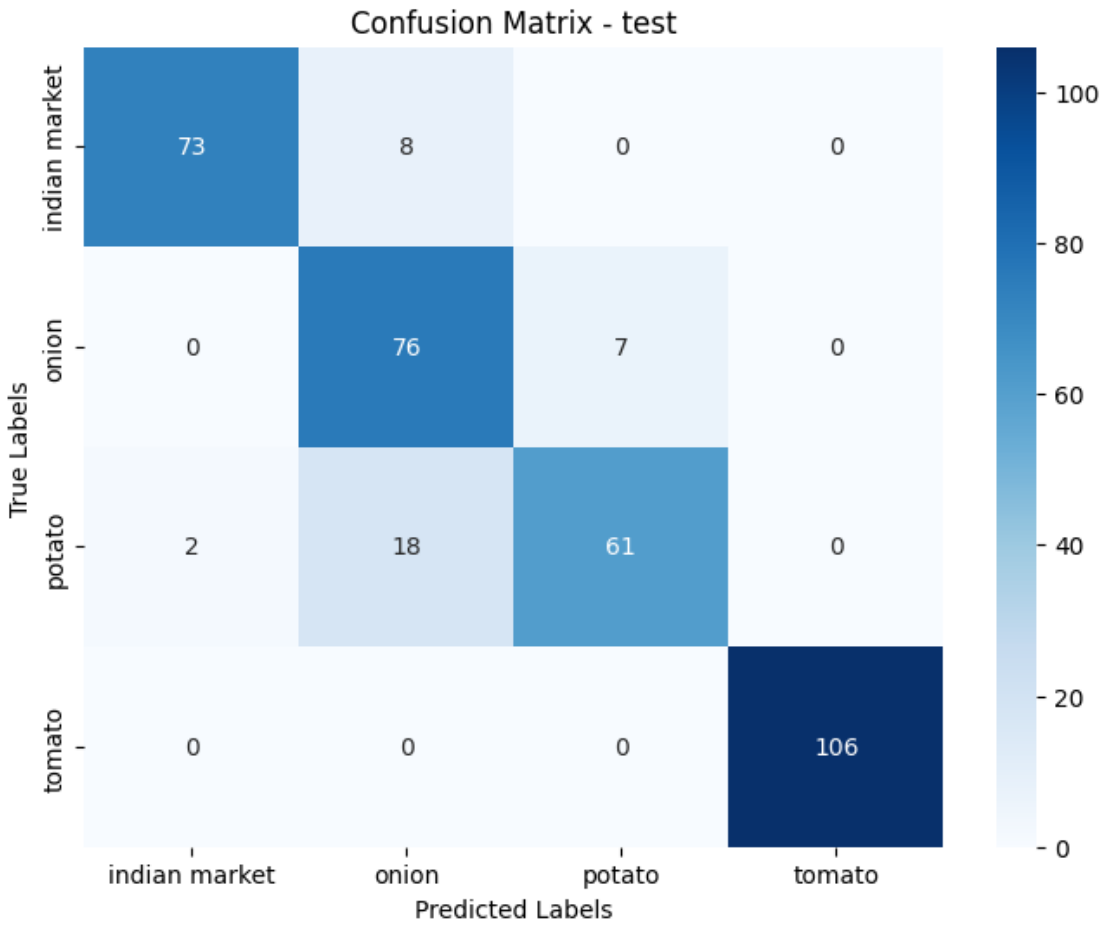
6.2.5 Model Evaluation, Mlflow Logging and Printing Metrics

[83]:
```
evaluate_model(model, train_aug_ds, val_aug_ds, test_aug_ds,␣
↪class_names,run_name=run_name,ckpt_path= ckpt_path)
```

```
2025/02/10 19:31:30 WARNING mlflow.tensorflow: You are saving a TensorFlow Core
model or Keras model without a signature. Inference with
mlflow.pyfunc.spark_udf() will not work unless the model's pyfunc representation
accepts pandas DataFrames as inference inputs.
2025/02/10 19:31:39 WARNING mlflow.models.model: Model logged without a
signature and input example. Please set `input_example` parameter when logging
the model to auto infer the model signature.


train Metrics:
 Accuracy: 98.05%
 Precision: 98.21%
 Recall: 98.24%

val Metrics:
 Accuracy: 94.58%
 Precision: 94.81%
 Recall: 94.76%

test Metrics:
 Accuracy: 90.03%
 Precision: 90.39%
 Recall: 89.25%
```


Confusion Matrix - train
True Labels
Predicted Labels
indian market
onion
potato
tomato
473
2
1
0
1
674
7
0
2
33
694
2
0
1
0
618
600
500
400
300
200
100
0


Confusion Matrix - val
indian market
onion
potato
tomato
True Labels
120
3
0
0
2
155
8
2
0
16
151
0
1
0
2
167
indian market
onion
potato
tomato
Predicted Labels
160
140
120
100
80
60
40
20
0



Classification Report - train

	precision	recall	f1-score	support
indian market	0.99	0.99	0.99	476
onion	0.95	0.99	0.97	682
potato	0.99	0.95	0.97	731
tomato	1.00	1.00	1.00	619
accuracy			0.98	2508
macro avg	0.98	0.98	0.98	2508
weighted avg	0.98	0.98	0.98	2508

Classification Report - val

	precision	recall	f1-score	support
indian market	0.98	0.98	0.98	123

```
        onion       0.89      0.93      0.91       167
       potato       0.94      0.90      0.92       167
       tomato       0.99      0.98      0.99       170

     accuracy                           0.95       627
    macro avg       0.95      0.95      0.95       627
 weighted avg       0.95      0.95      0.95       627


Classification Report - test
               precision    recall  f1-score   support

indian market       0.97      0.90      0.94        81
        onion       0.75      0.92      0.82        83
       potato       0.90      0.75      0.82        81
       tomato       1.00      1.00      1.00       106

     accuracy                           0.90       351
    macro avg       0.90      0.89      0.89       351
 weighted avg       0.91      0.90      0.90       351


Class-wise Accuracy:

Train Class-wise Accuracy:
 indian market: 99.37% (473/476)
 onion: 98.83% (674/682)
 potato: 94.94% (694/731)
 tomato: 99.84% (618/619)

Val Class-wise Accuracy:
 indian market: 97.56% (120/123)
 onion: 92.81% (155/167)
 potato: 90.42% (151/167)
 tomato: 98.24% (167/170)

Test Class-wise Accuracy:
 indian market: 90.12% (73/81)
 onion: 91.57% (76/83)
 potato: 75.31% (61/81)
 tomato: 100.00% (106/106)

Random Test Predictions:
```

True: indian market
Pred: indian market

True: indian market
Pred: onion

True: onion
Pred: onion

True: onion
Pred: onion

True: potato
Pred: onion

True: potato
Pred: onion

True: tomato
Pred: tomato

True: tomato
Pred: tomato

6.2.6 Save the model

[82]:
```
model.save("saved_models/2_pretrained_vgg16_aug_trainable0_model.keras")
```

6.2.7 Observations

- **Architecture**: Deep 16-layer VGG16 CNN pretrained on ImageNet along with two dense layers with batch norm and one dropout

- **Performance Before augmentation**:
  - **Train Accuracy**: **100%**
  - **Validation Accuracy**: **95.69%**
  - **Test Accuracy**: **89.7%** (Better training but lack of generalisation).

  - **Test Precision/Recall**: ~89% macro-average.
  - **Epochs**: **43**
  - **Training time**: **320 min (approx)**
  - **Class-Specific Issues**:
    * Struggled with **"Indian market"** (noise) class with **Onions and Potatoes**. misclassifying it as vegetables due to background clutter.

    * Excelled at **"tomato"** (**100% recall**) due to distinct color/texture features.

- **Strengths**:
  - Captured fine-grained details (e.g., vegetable textures).

  - Generalizable to similar agricultural datasets.

- **Weaknesses**:
  - Slow inference speed due to dense layers.

  - Overfit on training data (99% train vs. 89% test accuracy).

- **Impact of Augmentation**:
  - **Train Accuracy**: **98.05%**
  - **Validation Accuracy**: **94.58%**
  - **Test Accuracy**: **90.03%** (Slightly robust than previous).

  - **Test Precision/Recall**: ~89% macro-average.
  - **Epochs**: **20** + initialised with VGG16-no augmentation weights
  - **Training time**: **320 min (approx)**

6.3 RESNET50

6.3.1 Building the model

[88]:
```
def pretrained_resnet50(height=256, width=256, num_classes=4,␣
↪trainable_layers=0, ckpt_path=None):
    base_model = applications.ResNet50(weights="imagenet", include_top=False,␣
↪input_shape=(height, width, 3))

    # Freeze all layers initially
    base_model.trainable = False

    # If fine-tuning, unfreeze last `trainable_layers`
    if trainable_layers > 0:
        for layer in base_model.layers[-trainable_layers:]:
            layer.trainable = True

    model = keras.Sequential([
        base_model,
        layers.GlobalAveragePooling2D(),
        layers.Dense(512, activation="relu", kernel_regularizer=regularizers.
↪l2(5e-4)),
        layers.BatchNormalization(),
        layers.Dropout(0.3),

        layers.Dense(256, activation="relu", kernel_regularizer=regularizers.
↪l2(5e-4)),
        layers.BatchNormalization(),

        layers.Dense(num_classes, activation="softmax")
    ], name="pretrained_resnet50")

    # Load weights only if a valid checkpoint is provided
    if ckpt_path and os.path.exists(ckpt_path):
        print(f"Loading weights from {ckpt_path}")
        model.load_weights(ckpt_path)

    return model
```

[89]:
```
ckpt_path = "checkpoints/1_pretrained_resnet50_trainable0.weights.h5"
log_dir = "logs/1_pretrained_resnet50_trainable0"

# Load pretrained resnet50 model
model = pretrained_resnet50(height=256, width=256, num_classes=4,␣
↪trainable_layers=0,ckpt_path=ckpt_path)
```

[90]:
```
model.summary()
```

Model: "pretrained_resnet50"

Layer (type)	Output Shape	Param #
resnet50 (Functional)	(None, 8, 8, 2048)	23,587,712
global_average_pooling2d_6 (GlobalAveragePooling2D)	(None, 2048)	0
dense_18 (Dense)	(None, 512)	1,049,088
batch_normalization_12 (BatchNormalization)	(None, 512)	2,048
dropout_6 (Dropout)	(None, 512)	0
dense_19 (Dense)	(None, 256)	131,328
batch_normalization_13 (BatchNormalization)	(None, 256)	1,024
dense_20 (Dense)	(None, 4)	1,028

Total params: 24,772,228 (94.50 MB)

Trainable params: 1,182,980 (4.51 MB)

Non-trainable params: 23,589,248 (89.99 MB)

6.3.2 Model Compilation and Training

[91]:
```
model_fit = compile_train(model, train_ds, val_ds,␣
↪log_dir=log_dir,epochs=50,ckpt_path=ckpt_path)
```

```
Epoch 1/50
79/79              174s 2s/step -
accuracy: 0.5286 - loss: 1.6048 - precision_3: 0.5718 - recall_3: 0.4438 -
val_accuracy: 0.2663 - val_loss: 2.7746 - val_precision_3: 0.2668 -
val_recall_3: 0.2663 - learning_rate: 0.0010
Epoch 2/50
79/79              183s 2s/step -
accuracy: 0.6496 - loss: 1.0667 - precision_3: 0.6930 - recall_3: 0.5660 -
val_accuracy: 0.3381 - val_loss: 2.3900 - val_precision_3: 0.3523 -
```

```
val_recall_3: 0.3349 - learning_rate: 0.0010
Epoch 3/50
79/79            181s 2s/step -
accuracy: 0.6865 - loss: 0.9255 - precision_3: 0.7257 - recall_3: 0.6292 -
val_accuracy: 0.4545 - val_loss: 1.3855 - val_precision_3: 0.5517 -
val_recall_3: 0.4083 - learning_rate: 0.0010
Epoch 4/50
79/79            183s 2s/step -
accuracy: 0.7336 - loss: 0.8573 - precision_3: 0.7680 - recall_3: 0.6741 -
val_accuracy: 0.3939 - val_loss: 1.6536 - val_precision_3: 0.3989 -
val_recall_3: 0.3429 - learning_rate: 0.0010
Epoch 5/50
79/79            183s 2s/step -
accuracy: 0.7286 - loss: 0.8414 - precision_3: 0.7629 - recall_3: 0.6847 -
val_accuracy: 0.3110 - val_loss: 4.1335 - val_precision_3: 0.3124 -
val_recall_3: 0.3094 - learning_rate: 0.0010
Epoch 6/50
79/79            186s 2s/step -
accuracy: 0.7418 - loss: 0.8370 - precision_3: 0.7724 - recall_3: 0.7076 -
val_accuracy: 0.5152 - val_loss: 2.0395 - val_precision_3: 0.5362 -
val_recall_3: 0.4960 - learning_rate: 0.0010
Epoch 7/50
79/79            183s 2s/step -
accuracy: 0.7446 - loss: 0.7882 - precision_3: 0.7740 - recall_3: 0.7047 -
val_accuracy: 0.3014 - val_loss: 3.6226 - val_precision_3: 0.3013 -
val_recall_3: 0.2998 - learning_rate: 0.0010
Epoch 8/50
79/79            182s 2s/step -
accuracy: 0.7536 - loss: 0.7763 - precision_3: 0.7810 - recall_3: 0.7222 -
val_accuracy: 0.2935 - val_loss: 5.4834 - val_precision_3: 0.2923 -
val_recall_3: 0.2919 - learning_rate: 0.0010
Epoch 9/50
79/79            183s 2s/step -
accuracy: 0.7813 - loss: 0.7004 - precision_3: 0.8134 - recall_3: 0.7450 -
val_accuracy: 0.5486 - val_loss: 1.3409 - val_precision_3: 0.5662 -
val_recall_3: 0.5183 - learning_rate: 3.0000e-04
Epoch 10/50
79/79            181s 2s/step -
accuracy: 0.8249 - loss: 0.6420 - precision_3: 0.8442 - recall_3: 0.7917 -
val_accuracy: 0.3892 - val_loss: 2.0319 - val_precision_3: 0.3947 -
val_recall_3: 0.3796 - learning_rate: 3.0000e-04
Epoch 11/50
79/79            182s 2s/step -
accuracy: 0.8182 - loss: 0.6275 - precision_3: 0.8456 - recall_3: 0.7989 -
val_accuracy: 0.5726 - val_loss: 1.2986 - val_precision_3: 0.5823 -
val_recall_3: 0.5359 - learning_rate: 3.0000e-04
Epoch 12/50
79/79            183s 2s/step -
```

```
accuracy: 0.8281 - loss: 0.5832 - precision_3: 0.8434 - recall_3: 0.8039 -
val_accuracy: 0.7416 - val_loss: 0.7650 - val_precision_3: 0.7795 -
val_recall_3: 0.7049 - learning_rate: 3.0000e-04
Epoch 13/50
79/79            180s 2s/step -
accuracy: 0.8324 - loss: 0.5723 - precision_3: 0.8550 - recall_3: 0.8134 -
val_accuracy: 0.7049 - val_loss: 1.0640 - val_precision_3: 0.7224 -
val_recall_3: 0.6890 - learning_rate: 3.0000e-04
Epoch 14/50
79/79            182s 2s/step -
accuracy: 0.8221 - loss: 0.6057 - precision_3: 0.8423 - recall_3: 0.8002 -
val_accuracy: 0.5965 - val_loss: 1.2404 - val_precision_3: 0.6067 -
val_recall_3: 0.5758 - learning_rate: 3.0000e-04
Epoch 15/50
79/79            181s 2s/step -
accuracy: 0.8462 - loss: 0.5733 - precision_3: 0.8582 - recall_3: 0.8226 -
val_accuracy: 0.3365 - val_loss: 2.4530 - val_precision_3: 0.3355 -
val_recall_3: 0.3349 - learning_rate: 3.0000e-04
Epoch 16/50
79/79            182s 2s/step -
accuracy: 0.8394 - loss: 0.5771 - precision_3: 0.8588 - recall_3: 0.8169 -
val_accuracy: 0.5167 - val_loss: 1.4620 - val_precision_3: 0.5509 -
val_recall_3: 0.5008 - learning_rate: 3.0000e-04
Epoch 17/50
79/79            181s 2s/step -
accuracy: 0.8326 - loss: 0.5642 - precision_3: 0.8508 - recall_3: 0.8183 -
val_accuracy: 0.5710 - val_loss: 1.3393 - val_precision_3: 0.5671 -
val_recall_3: 0.5327 - learning_rate: 3.0000e-04
Epoch 18/50
79/79            180s 2s/step -
accuracy: 0.8484 - loss: 0.5238 - precision_3: 0.8659 - recall_3: 0.8330 -
val_accuracy: 0.7384 - val_loss: 0.7899 - val_precision_3: 0.7703 -
val_recall_3: 0.7113 - learning_rate: 9.0000e-05
Epoch 19/50
79/79            182s 2s/step -
accuracy: 0.8572 - loss: 0.5040 - precision_3: 0.8779 - recall_3: 0.8407 -
val_accuracy: 0.7081 - val_loss: 0.8874 - val_precision_3: 0.7372 -
val_recall_3: 0.6890 - learning_rate: 9.0000e-05
Epoch 20/50
79/79            182s 2s/step -
accuracy: 0.8481 - loss: 0.5189 - precision_3: 0.8642 - recall_3: 0.8348 -
val_accuracy: 0.7512 - val_loss: 0.7878 - val_precision_3: 0.7711 -
val_recall_3: 0.7416 - learning_rate: 9.0000e-05
Epoch 21/50
79/79            182s 2s/step -
accuracy: 0.8528 - loss: 0.5185 - precision_3: 0.8665 - recall_3: 0.8317 -
val_accuracy: 0.7065 - val_loss: 0.8887 - val_precision_3: 0.7285 -
val_recall_3: 0.6890 - learning_rate: 9.0000e-05
```

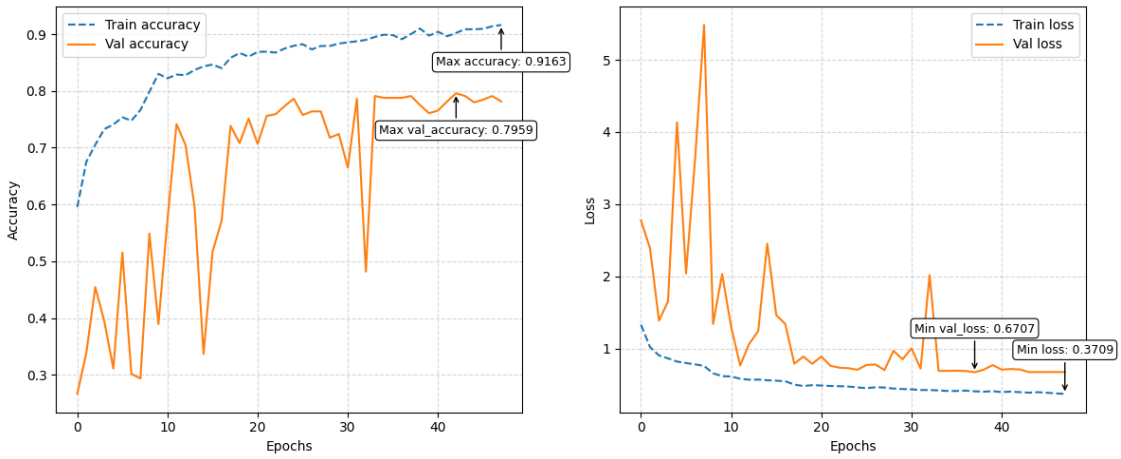
```
Epoch 22/50
79/79            185s 2s/step -
accuracy: 0.8594 - loss: 0.4947 - precision_3: 0.8757 - recall_3: 0.8439 -
val_accuracy: 0.7560 - val_loss: 0.7611 - val_precision_3: 0.7668 -
val_recall_3: 0.7448 - learning_rate: 9.0000e-05
Epoch 23/50
79/79            185s 2s/step -
accuracy: 0.8657 - loss: 0.4972 - precision_3: 0.8745 - recall_3: 0.8485 -
val_accuracy: 0.7592 - val_loss: 0.7336 - val_precision_3: 0.7732 -
val_recall_3: 0.7448 - learning_rate: 9.0000e-05
Epoch 24/50
79/79            186s 2s/step -
accuracy: 0.8684 - loss: 0.4918 - precision_3: 0.8845 - recall_3: 0.8549 -
val_accuracy: 0.7735 - val_loss: 0.7267 - val_precision_3: 0.7906 -
val_recall_3: 0.7528 - learning_rate: 9.0000e-05
Epoch 25/50
79/79            182s 2s/step -
accuracy: 0.8731 - loss: 0.4844 - precision_3: 0.8814 - recall_3: 0.8561 -
val_accuracy: 0.7863 - val_loss: 0.7056 - val_precision_3: 0.8068 -
val_recall_3: 0.7592 - learning_rate: 9.0000e-05
Epoch 26/50
79/79            180s 2s/step -
accuracy: 0.8719 - loss: 0.4720 - precision_3: 0.8908 - recall_3: 0.8579 -
val_accuracy: 0.7576 - val_loss: 0.7716 - val_precision_3: 0.7754 -
val_recall_3: 0.7432 - learning_rate: 9.0000e-05
Epoch 27/50
79/79            178s 2s/step -
accuracy: 0.8601 - loss: 0.4959 - precision_3: 0.8734 - recall_3: 0.8419 -
val_accuracy: 0.7640 - val_loss: 0.7789 - val_precision_3: 0.7738 -
val_recall_3: 0.7528 - learning_rate: 9.0000e-05
Epoch 28/50
79/79            180s 2s/step -
accuracy: 0.8698 - loss: 0.4793 - precision_3: 0.8807 - recall_3: 0.8548 -
val_accuracy: 0.7640 - val_loss: 0.7004 - val_precision_3: 0.7837 -
val_recall_3: 0.7512 - learning_rate: 9.0000e-05
Epoch 29/50
79/79            132s 2s/step -
accuracy: 0.8751 - loss: 0.4636 - precision_3: 0.8903 - recall_3: 0.8588 -
val_accuracy: 0.7177 - val_loss: 0.9682 - val_precision_3: 0.7407 -
val_recall_3: 0.6970 - learning_rate: 9.0000e-05
Epoch 30/50
79/79            126s 2s/step -
accuracy: 0.8847 - loss: 0.4480 - precision_3: 0.8937 - recall_3: 0.8725 -
val_accuracy: 0.7241 - val_loss: 0.8508 - val_precision_3: 0.7364 -
val_recall_3: 0.7129 - learning_rate: 9.0000e-05
Epoch 31/50
79/79            126s 2s/step -
accuracy: 0.8773 - loss: 0.4429 - precision_3: 0.8907 - recall_3: 0.8683 -
```

```
val_accuracy: 0.6651 - val_loss: 1.0020 - val_precision_3: 0.6744 -
val_recall_3: 0.6475 - learning_rate: 9.0000e-05
Epoch 32/50
79/79            127s 2s/step -
accuracy: 0.8727 - loss: 0.4475 - precision_3: 0.8825 - recall_3: 0.8610 -
val_accuracy: 0.7863 - val_loss: 0.7200 - val_precision_3: 0.7987 -
val_recall_3: 0.7719 - learning_rate: 9.0000e-05
Epoch 33/50
79/79            127s 2s/step -
accuracy: 0.8888 - loss: 0.4308 - precision_3: 0.9019 - recall_3: 0.8790 -
val_accuracy: 0.4817 - val_loss: 2.0158 - val_precision_3: 0.4885 -
val_recall_3: 0.4753 - learning_rate: 9.0000e-05
Epoch 34/50
79/79            126s 2s/step -
accuracy: 0.8870 - loss: 0.4439 - precision_3: 0.9007 - recall_3: 0.8719 -
val_accuracy: 0.7911 - val_loss: 0.6892 - val_precision_3: 0.7997 -
val_recall_3: 0.7767 - learning_rate: 2.7000e-05
Epoch 35/50
79/79            127s 2s/step -
accuracy: 0.8894 - loss: 0.4299 - precision_3: 0.9035 - recall_3: 0.8785 -
val_accuracy: 0.7879 - val_loss: 0.6901 - val_precision_3: 0.8003 -
val_recall_3: 0.7735 - learning_rate: 2.7000e-05
Epoch 36/50
79/79            127s 2s/step -
accuracy: 0.8970 - loss: 0.4205 - precision_3: 0.9066 - recall_3: 0.8842 -
val_accuracy: 0.7879 - val_loss: 0.6921 - val_precision_3: 0.7997 -
val_recall_3: 0.7703 - learning_rate: 2.7000e-05
Epoch 37/50
79/79            126s 2s/step -
accuracy: 0.8868 - loss: 0.4243 - precision_3: 0.9018 - recall_3: 0.8770 -
val_accuracy: 0.7879 - val_loss: 0.6883 - val_precision_3: 0.8013 -
val_recall_3: 0.7719 - learning_rate: 2.7000e-05
Epoch 38/50
79/79            126s 2s/step -
accuracy: 0.8904 - loss: 0.4337 - precision_3: 0.8983 - recall_3: 0.8774 -
val_accuracy: 0.7911 - val_loss: 0.6707 - val_precision_3: 0.8046 -
val_recall_3: 0.7815 - learning_rate: 2.7000e-05
Epoch 39/50
79/79            126s 2s/step -
accuracy: 0.9039 - loss: 0.4232 - precision_3: 0.9125 - recall_3: 0.8936 -
val_accuracy: 0.7751 - val_loss: 0.7062 - val_precision_3: 0.7840 -
val_recall_3: 0.7640 - learning_rate: 2.7000e-05
Epoch 40/50
79/79            126s 2s/step -
accuracy: 0.8921 - loss: 0.4170 - precision_3: 0.9032 - recall_3: 0.8756 -
val_accuracy: 0.7608 - val_loss: 0.7694 - val_precision_3: 0.7702 -
val_recall_3: 0.7432 - learning_rate: 2.7000e-05
Epoch 41/50
```

```
79/79            126s 2s/step -
accuracy: 0.9094 - loss: 0.4033 - precision_3: 0.9149 - recall_3: 0.8947 -
val_accuracy: 0.7656 - val_loss: 0.7067 - val_precision_3: 0.7789 -
val_recall_3: 0.7528 - learning_rate: 2.7000e-05
Epoch 42/50
79/79            126s 2s/step -
accuracy: 0.8997 - loss: 0.4065 - precision_3: 0.9141 - recall_3: 0.8861 -
val_accuracy: 0.7815 - val_loss: 0.7163 - val_precision_3: 0.7980 -
val_recall_3: 0.7624 - learning_rate: 2.7000e-05
Epoch 43/50
79/79            127s 2s/step -
accuracy: 0.8915 - loss: 0.4094 - precision_3: 0.9053 - recall_3: 0.8816 -
val_accuracy: 0.7959 - val_loss: 0.7092 - val_precision_3: 0.8036 -
val_recall_3: 0.7831 - learning_rate: 2.7000e-05
Epoch 44/50
79/79            127s 2s/step -
accuracy: 0.9012 - loss: 0.4126 - precision_3: 0.9134 - recall_3: 0.8879 -
val_accuracy: 0.7911 - val_loss: 0.6742 - val_precision_3: 0.8036 -
val_recall_3: 0.7767 - learning_rate: 1.0000e-05
Epoch 45/50
79/79            127s 2s/step -
accuracy: 0.9063 - loss: 0.4052 - precision_3: 0.9182 - recall_3: 0.8941 -
val_accuracy: 0.7799 - val_loss: 0.6729 - val_precision_3: 0.7951 -
val_recall_3: 0.7735 - learning_rate: 1.0000e-05
Epoch 46/50
79/79            126s 2s/step -
accuracy: 0.9027 - loss: 0.3984 - precision_3: 0.9118 - recall_3: 0.8943 -
val_accuracy: 0.7847 - val_loss: 0.6733 - val_precision_3: 0.8003 -
val_recall_3: 0.7735 - learning_rate: 1.0000e-05
Epoch 47/50
79/79            127s 2s/step -
accuracy: 0.9010 - loss: 0.4047 - precision_3: 0.9129 - recall_3: 0.8919 -
val_accuracy: 0.7911 - val_loss: 0.6740 - val_precision_3: 0.8072 -
val_recall_3: 0.7815 - learning_rate: 1.0000e-05
Epoch 48/50
79/79            126s 2s/step -
accuracy: 0.9063 - loss: 0.3857 - precision_3: 0.9113 - recall_3: 0.8944 -
val_accuracy: 0.7815 - val_loss: 0.6720 - val_precision_3: 0.8000 -
val_recall_3: 0.7719 - learning_rate: 1.0000e-05
```

6.3.3 Plot loss & accuracy for training and validation wrt epoch

[92]:
```
plot_loss_and_accuracy(["accuracy","loss"],model_fit)
```



6.3.4 Tensorboard

[93]:
```
%load_ext tensorboard
```

```
The tensorboard extension is already loaded. To reload it, use:
  %reload_ext tensorboard
```

6.3.5 Model Evaluation, Mlflow Logging and Printing Metrics

[94]:
```
ckpt_path
```

[94]:
```
'checkpoints/1_pretrained_resnet50_trainable0.weights.h5'
```

[95]:
```
log_dir
```

[95]:
```
'logs/1_pretrained_resnet50_trainable0'
```

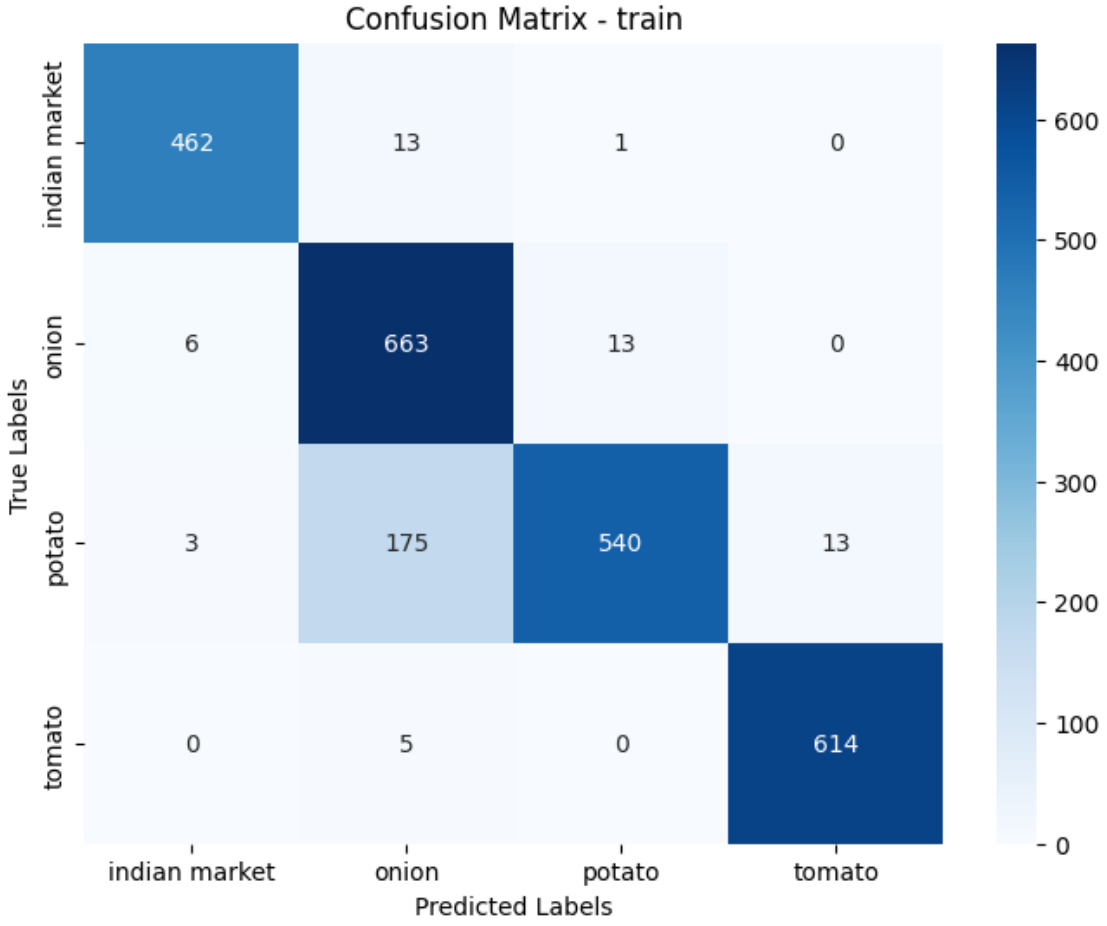
[96]:
```
run_name = "1_pretrained_resnet50_trainable0"
run_name
```

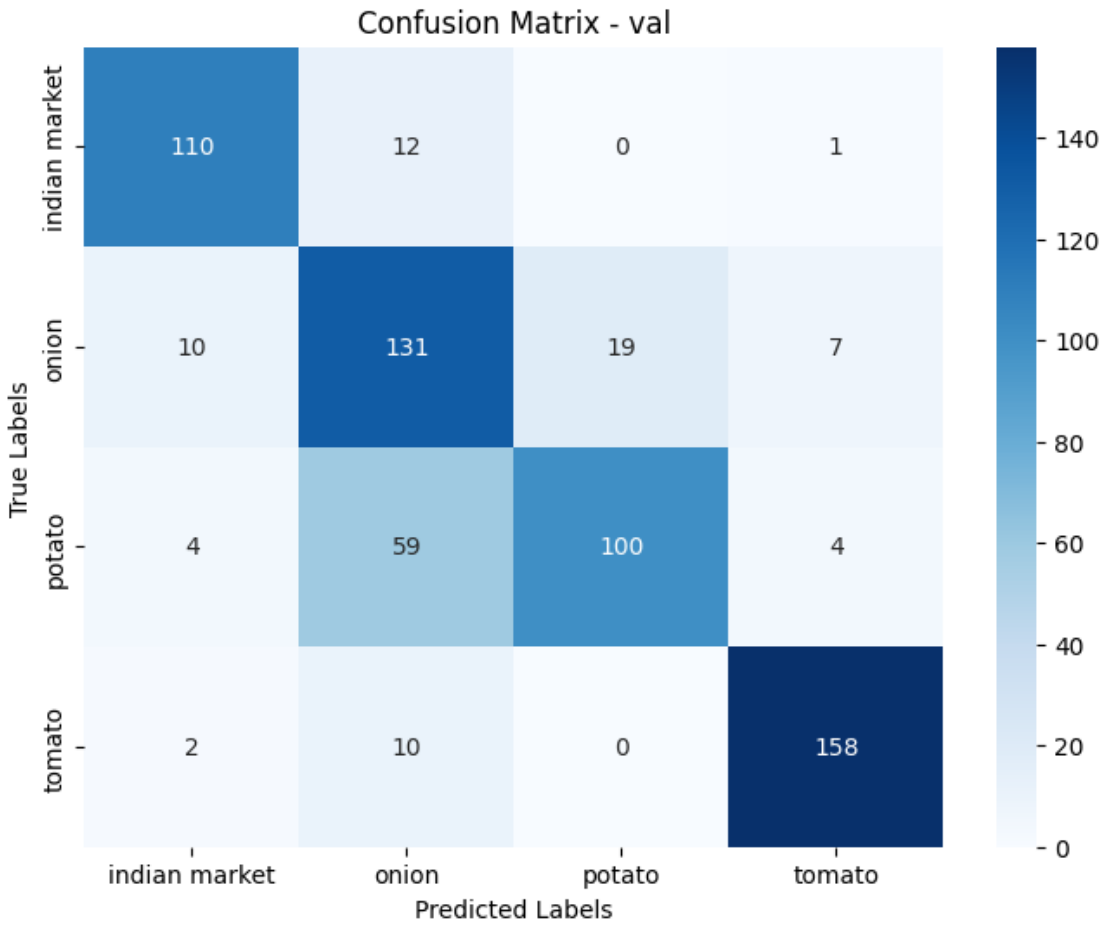
[96]:
```
'1_pretrained_resnet50_trainable0'
```

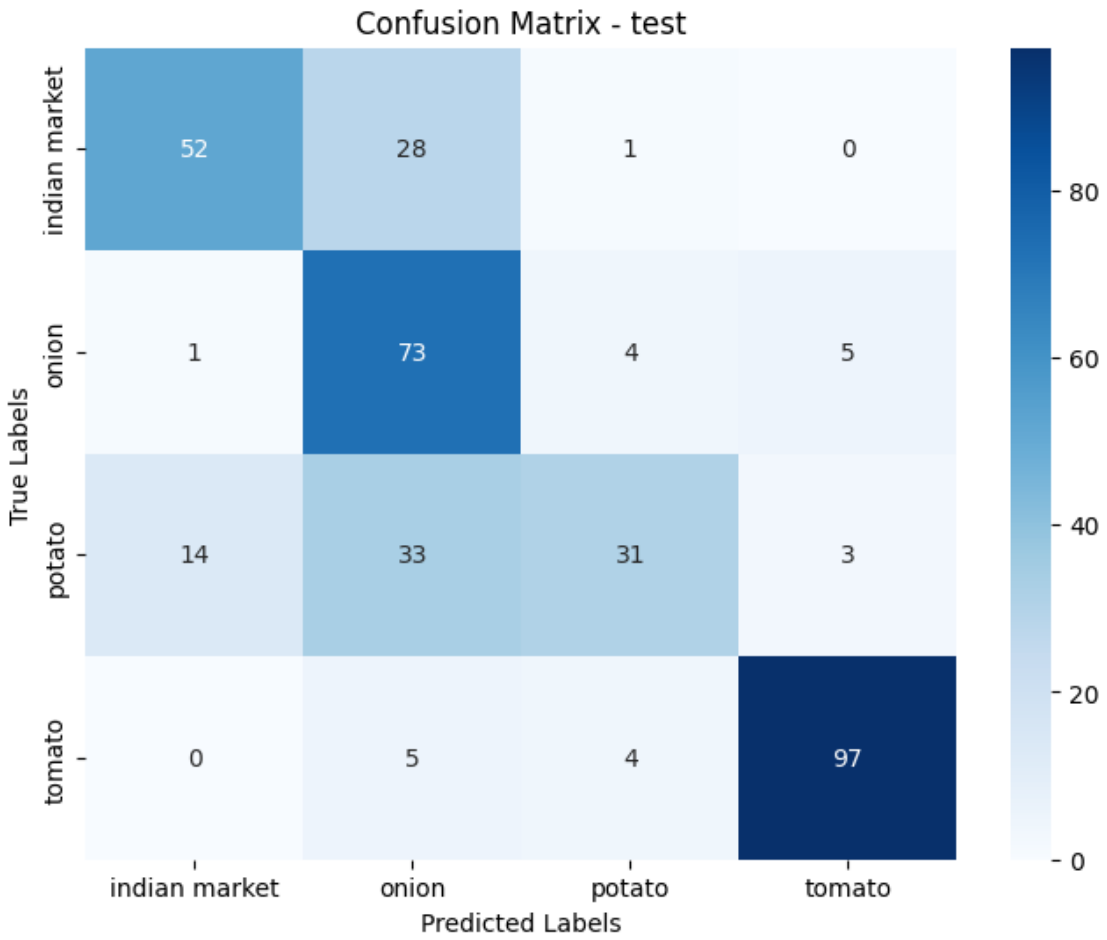
[97]:
```
evaluate_model(model, train_ds, val_ds, test_ds,␣
↪class_names,run_name=run_name,ckpt_path= ckpt_path)
```

```
2025/02/10 21:58:15 WARNING mlflow.tensorflow: You are saving a TensorFlow Core
model or Keras model without a signature. Inference with
mlflow.pyfunc.spark_udf() will not work unless the model's pyfunc representation
accepts pandas DataFrames as inference inputs.
2025/02/10 21:58:26 WARNING mlflow.models.model: Model logged without a
signature and input example. Please set `input_example` parameter when logging
the model to auto infer the model signature.
```

```
train Metrics:
 Accuracy: 90.87%
 Precision: 92.74%
 Recall: 91.83%

val Metrics:
 Accuracy: 79.59%
 Precision: 81.52%
 Recall: 80.17%

test Metrics:
 Accuracy: 72.08%
 Precision: 75.00%
 Recall: 70.48%
```




Confusion Matrix - val
indian market
onion
potato
tomato
True Labels
110
12
0
1
10
131
19
7
4
59
100
4
2
10
0
158
indian market
onion
potato
tomato
Predicted Labels
140
120
100
80
60
40
20
0




```
Classification Report - train
              precision    recall  f1-score   support

indian market       0.98      0.97      0.98       476
        onion       0.77      0.97      0.86       682
       potato       0.97      0.74      0.84       731
       tomato       0.98      0.99      0.99       619

     accuracy                           0.91      2508
    macro avg       0.93      0.92      0.92      2508
 weighted avg       0.92      0.91      0.91      2508


Classification Report - val
              precision    recall  f1-score   support

indian market       0.87      0.89      0.88       123
```

```
        onion       0.62      0.78      0.69       167
       potato       0.84      0.60      0.70       167
       tomato       0.93      0.93      0.93       170

     accuracy                           0.80       627
    macro avg       0.82      0.80      0.80       627
 weighted avg       0.81      0.80      0.80       627


Classification Report - test
               precision    recall  f1-score   support

indian market       0.78      0.64      0.70        81
        onion       0.53      0.88      0.66        83
       potato       0.78      0.38      0.51        81
       tomato       0.92      0.92      0.92       106

     accuracy                           0.72       351
    macro avg       0.75      0.70      0.70       351
 weighted avg       0.76      0.72      0.71       351


Class-wise Accuracy:

Train Class-wise Accuracy:
 indian market: 97.06% (462/476)
 onion: 97.21% (663/682)
 potato: 73.87% (540/731)
 tomato: 99.19% (614/619)

Val Class-wise Accuracy:
 indian market: 89.43% (110/123)
 onion: 78.44% (131/167)
 potato: 59.88% (100/167)
 tomato: 92.94% (158/170)

Test Class-wise Accuracy:
 indian market: 64.20% (52/81)
 onion: 87.95% (73/83)
 potato: 38.27% (31/81)
 tomato: 91.51% (97/106)

Random Test Predictions:
```

True: indian market
Pred: indian market

True: indian market
Pred: indian market

True: onion
Pred: potato

True: onion
Pred: indian market

True: potato
Pred: tomato

True: potato
Pred: onion

True: tomato
Pred: onion

True: tomato
Pred: tomato

6.3.6 Save the model

[98]:
```
model.save("saved_models/1_pretrained_resnet50_trainable0_model.keras")
```

6.4 DATA AUGMENTED RESNET50

6.4.1 Load the previously trained model architecture and weights

[99]:
```
# Load the entire model (architecture + weights)
model = keras.models.load_model("saved_models/
  ↪1_pretrained_resnet50_trainable0_model.keras")
```

[100]:
```
model.summary()
```

Model: "pretrained_resnet50"

Layer (type)	Output Shape	Param #
resnet50 (Functional)	(None, 8, 8, 2048)	23,587,712
global_average_pooling2d_6 (GlobalAveragePooling2D)	(None, 2048)	0
dense_18 (Dense)	(None, 512)	1,049,088
batch_normalization_12 (BatchNormalization)	(None, 512)	2,048
dropout_6 (Dropout)	(None, 512)	0
dense_19 (Dense)	(None, 256)	131,328
batch_normalization_13 (BatchNormalization)	(None, 256)	1,024
dense_20 (Dense)	(None, 4)	1,028

Total params: 27,138,190 (103.52 MB)

Trainable params: 1,182,980 (4.51 MB)

Non-trainable params: 23,589,248 (89.99 MB)

```
 Optimizer params: 2,365,962 (9.03 MB)
```

6.4.2 Model Compilation and Training

[101]:
```
ckpt_path = "checkpoints/1_pretrained_resnet50_aug_trainable0.weights.h5"
```

[102]:
```
log_dir = 'logs/1_pretrained_resnet50_aug_trainable0'
```

[103]:
```
run_name = '1_pretrained_resnet50_aug_trainable0'
```

[104]:
```
model_fit = compile_train(model, train_aug_ds, val_aug_ds,␣
 ↪log_dir=log_dir,epochs=20,ckpt_path=ckpt_path)
```

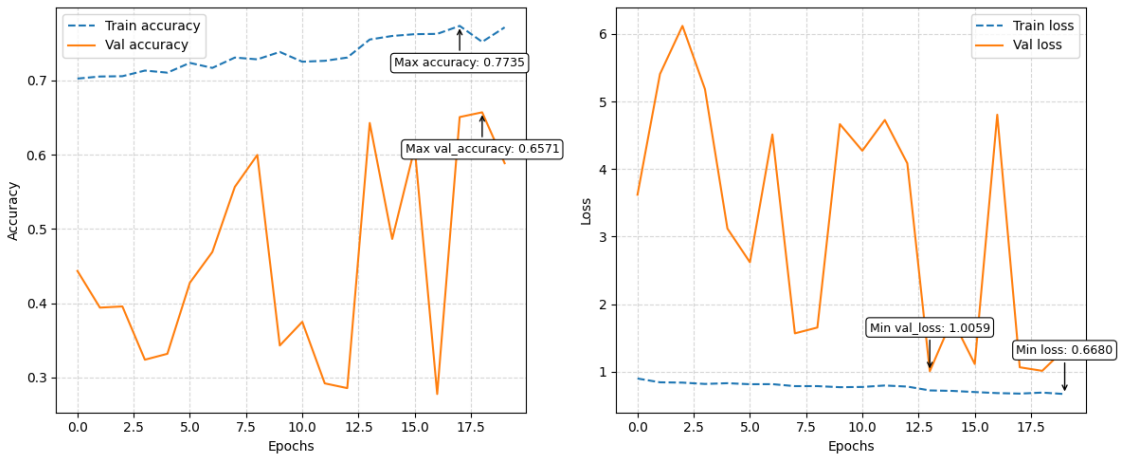
```
Epoch 1/20
79/79             138s 2s/step -
accuracy: 0.7046 - loss: 0.9642 - precision_4: 0.7288 - recall_4: 0.6683 -
val_accuracy: 0.4434 - val_loss: 3.6197 - val_precision_4: 0.4457 -
val_recall_4: 0.4386 - learning_rate: 0.0010
Epoch 2/20
79/79             129s 2s/step -
accuracy: 0.7112 - loss: 0.8268 - precision_4: 0.7432 - recall_4: 0.6651 -
val_accuracy: 0.3939 - val_loss: 5.4068 - val_precision_4: 0.3939 -
val_recall_4: 0.3939 - learning_rate: 0.0010
Epoch 3/20
79/79             128s 2s/step -
accuracy: 0.6950 - loss: 0.8709 - precision_4: 0.7396 - recall_4: 0.6384 -
val_accuracy: 0.3955 - val_loss: 6.1206 - val_precision_4: 0.3971 -
val_recall_4: 0.3939 - learning_rate: 0.0010
Epoch 4/20
79/79             133s 2s/step -
accuracy: 0.6953 - loss: 0.8382 - precision_4: 0.7365 - recall_4: 0.6324 -
val_accuracy: 0.3238 - val_loss: 5.1845 - val_precision_4: 0.3237 -
val_recall_4: 0.3222 - learning_rate: 0.0010
Epoch 5/20
79/79             131s 2s/step -
accuracy: 0.7122 - loss: 0.8262 - precision_4: 0.7551 - recall_4: 0.6668 -
val_accuracy: 0.3317 - val_loss: 3.1185 - val_precision_4: 0.3350 -
val_recall_4: 0.3270 - learning_rate: 0.0010
Epoch 6/20
79/79             131s 2s/step -
accuracy: 0.7328 - loss: 0.7921 - precision_4: 0.7695 - recall_4: 0.6700 -
val_accuracy: 0.4274 - val_loss: 2.6197 - val_precision_4: 0.4272 -
val_recall_4: 0.4163 - learning_rate: 0.0010
Epoch 7/20
79/79             131s 2s/step -
accuracy: 0.7107 - loss: 0.8195 - precision_4: 0.7416 - recall_4: 0.6646 -
val_accuracy: 0.4689 - val_loss: 4.5126 - val_precision_4: 0.4689 -
```

```
val_recall_4: 0.4689 - learning_rate: 0.0010
Epoch 8/20
79/79            130s 2s/step -
accuracy: 0.7222 - loss: 0.8058 - precision_4: 0.7617 - recall_4: 0.6734 -
val_accuracy: 0.5566 - val_loss: 1.5669 - val_precision_4: 0.5590 -
val_recall_4: 0.5439 - learning_rate: 0.0010
Epoch 9/20
79/79            128s 2s/step -
accuracy: 0.7359 - loss: 0.7748 - precision_4: 0.7727 - recall_4: 0.6799 -
val_accuracy: 0.5997 - val_loss: 1.6530 - val_precision_4: 0.6050 -
val_recall_4: 0.5837 - learning_rate: 0.0010
Epoch 10/20
79/79            165s 2s/step -
accuracy: 0.7360 - loss: 0.7911 - precision_4: 0.7706 - recall_4: 0.6818 -
val_accuracy: 0.3429 - val_loss: 4.6655 - val_precision_4: 0.3435 -
val_recall_4: 0.3429 - learning_rate: 0.0010
Epoch 11/20
79/79            185s 2s/step -
accuracy: 0.7221 - loss: 0.7943 - precision_4: 0.7686 - recall_4: 0.6722 -
val_accuracy: 0.3748 - val_loss: 4.2717 - val_precision_4: 0.3728 -
val_recall_4: 0.3668 - learning_rate: 0.0010
Epoch 12/20
79/79            183s 2s/step -
accuracy: 0.7111 - loss: 0.8148 - precision_4: 0.7534 - recall_4: 0.6591 -
val_accuracy: 0.2919 - val_loss: 4.7266 - val_precision_4: 0.2919 -
val_recall_4: 0.2919 - learning_rate: 0.0010
Epoch 13/20
79/79            188s 2s/step -
accuracy: 0.7177 - loss: 0.7980 - precision_4: 0.7624 - recall_4: 0.6699 -
val_accuracy: 0.2855 - val_loss: 4.0851 - val_precision_4: 0.2859 -
val_recall_4: 0.2855 - learning_rate: 0.0010
Epoch 14/20
79/79            187s 2s/step -
accuracy: 0.7376 - loss: 0.7565 - precision_4: 0.7827 - recall_4: 0.6839 -
val_accuracy: 0.6427 - val_loss: 1.0059 - val_precision_4: 0.6667 -
val_recall_4: 0.6061 - learning_rate: 3.0000e-04
Epoch 15/20
79/79            187s 2s/step -
accuracy: 0.7504 - loss: 0.7256 - precision_4: 0.7835 - recall_4: 0.7107 -
val_accuracy: 0.4864 - val_loss: 1.7771 - val_precision_4: 0.4934 -
val_recall_4: 0.4801 - learning_rate: 3.0000e-04
Epoch 16/20
79/79            184s 2s/step -
accuracy: 0.7562 - loss: 0.7041 - precision_4: 0.7937 - recall_4: 0.7085 -
val_accuracy: 0.6093 - val_loss: 1.1126 - val_precision_4: 0.6241 -
val_recall_4: 0.5694 - learning_rate: 3.0000e-04
Epoch 17/20
79/79            182s 2s/step -
```

```
accuracy: 0.7558 - loss: 0.7005 - precision_4: 0.7862 - recall_4: 0.7105 -
val_accuracy: 0.2775 - val_loss: 4.8054 - val_precision_4: 0.2775 -
val_recall_4: 0.2775 - learning_rate: 3.0000e-04
Epoch 18/20
79/79              183s 2s/step -
accuracy: 0.7733 - loss: 0.6748 - precision_4: 0.8140 - recall_4: 0.7374 -
val_accuracy: 0.6507 - val_loss: 1.0644 - val_precision_4: 0.6644 -
val_recall_4: 0.6252 - learning_rate: 3.0000e-04
Epoch 19/20
79/79              184s 2s/step -
accuracy: 0.7408 - loss: 0.7053 - precision_4: 0.7774 - recall_4: 0.7087 -
val_accuracy: 0.6571 - val_loss: 1.0117 - val_precision_4: 0.6785 -
val_recall_4: 0.6427 - learning_rate: 3.0000e-04
Epoch 20/20
79/79              183s 2s/step -
accuracy: 0.7528 - loss: 0.7177 - precision_4: 0.7832 - recall_4: 0.7126 -
val_accuracy: 0.5885 - val_loss: 1.3327 - val_precision_4: 0.5940 -
val_recall_4: 0.5694 - learning_rate: 9.0000e-05
```

6.4.3 Plot loss & accuracy for training and validation wrt epoch

[105]:
```
plot_loss_and_accuracy(["accuracy","loss"],model_fit)
```



6.4.4 Tensorboard

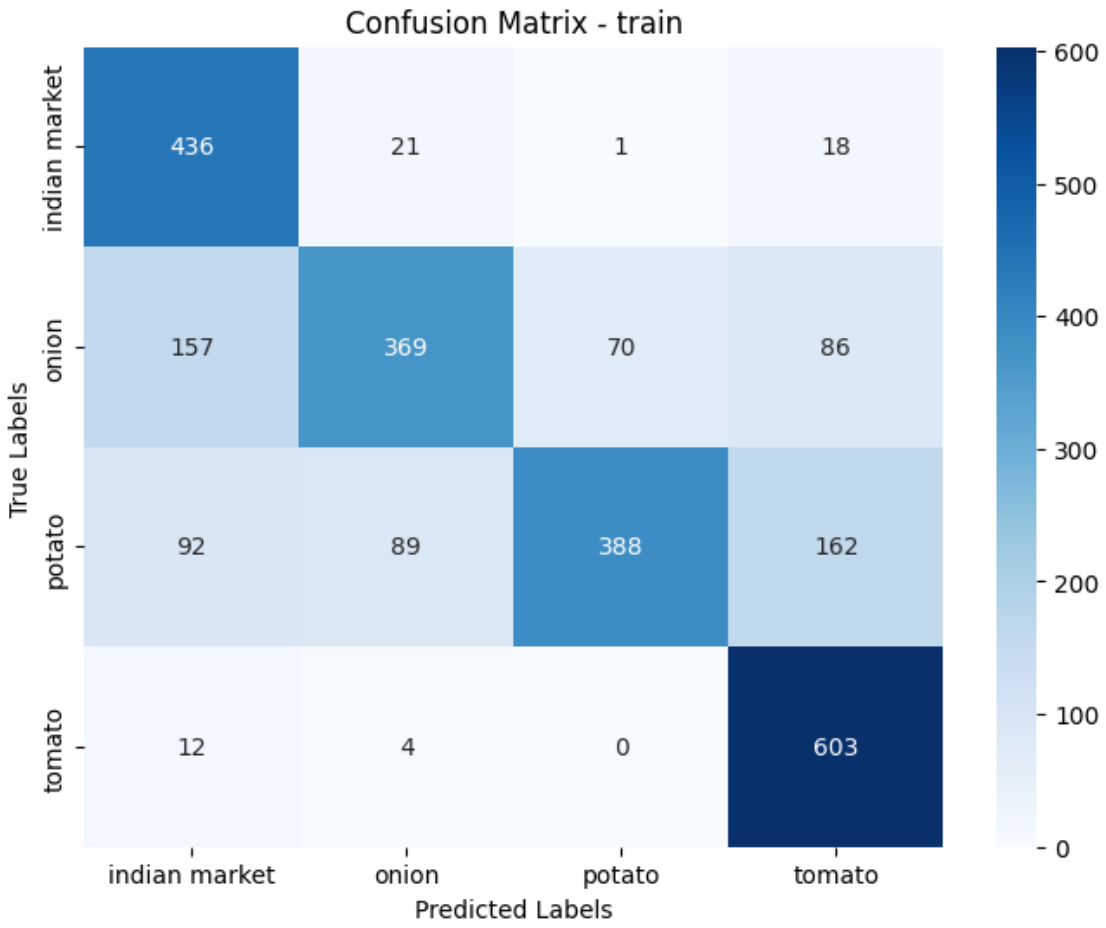
[106]:
```
%load_ext tensorboard
```

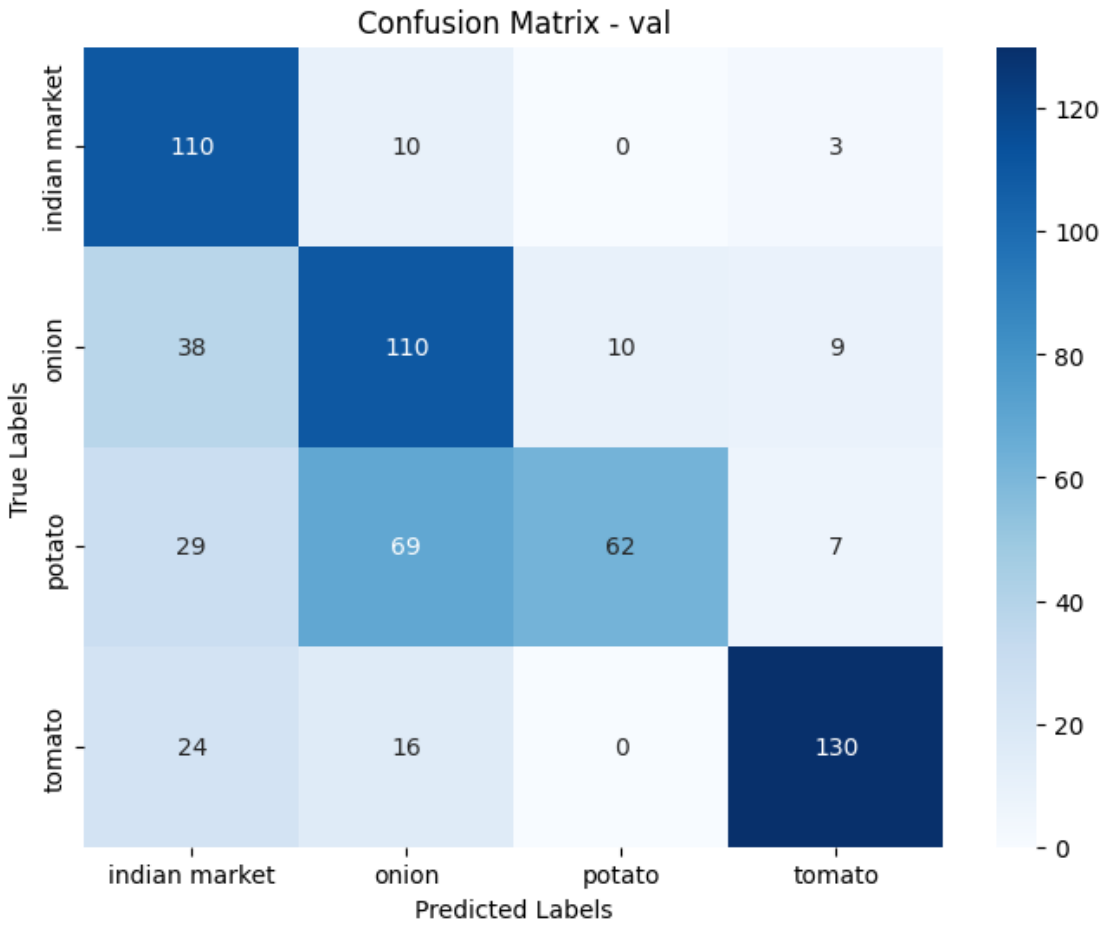
```
The tensorboard extension is already loaded. To reload it, use:
  %reload_ext tensorboard
```

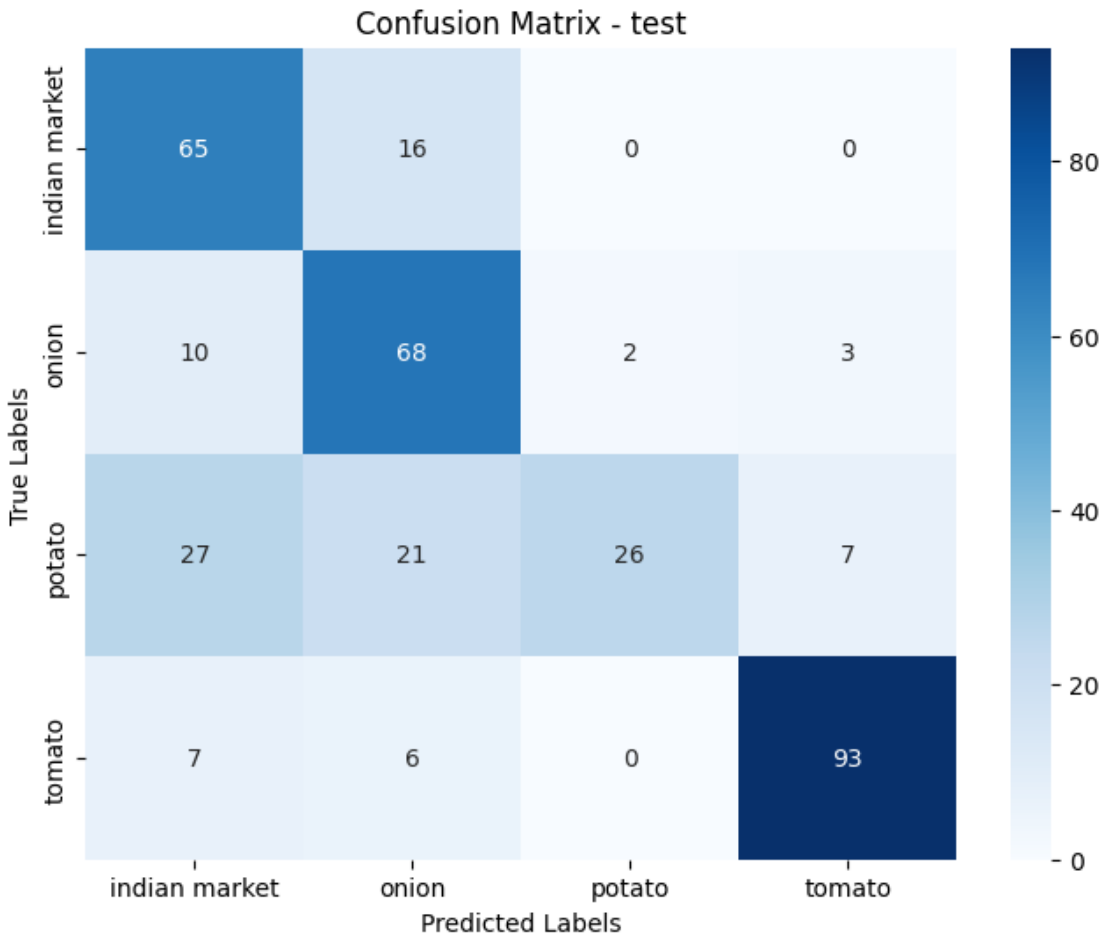
6.4.5 Model Evaluation, Mlflow Logging and Printing Metrics

[107]:
```
evaluate_model(model, train_aug_ds, val_aug_ds, test_aug_ds,␣
↪class_names,run_name=run_name,ckpt_path= ckpt_path)
```

```
2025/02/10 22:55:08 WARNING mlflow.tensorflow: You are saving a TensorFlow Core
model or Keras model without a signature. Inference with
mlflow.pyfunc.spark_udf() will not work unless the model's pyfunc representation
accepts pandas DataFrames as inference inputs.
2025/02/10 22:55:34 WARNING mlflow.models.model: Model logged without a
signature and input example. Please set `input_example` parameter when logging
the model to auto infer the model signature.


train Metrics:
 Accuracy: 71.61%
 Precision: 73.22%
 Recall: 74.05%

val Metrics:
 Accuracy: 65.71%
 Precision: 70.44%
 Recall: 67.22%

test Metrics:
 Accuracy: 71.79%
 Precision: 76.01%
 Recall: 70.50%
```


Confusion Matrix - train
True Labels
Predicted Labels
indian market
onion
potato
tomato
436
21
1
18
157
369
70
86
92
89
388
162
12
4
0
603
600
500
400
300
200
100
0


Confusion Matrix - val
indian market
onion
potato
tomato
True Labels
110
10
0
3
38
110
10
9
29
69
62
7
24
16
0
130
indian market
onion
potato
tomato
Predicted Labels
120
100
80
60
40
20
0




```
Classification Report - train
              precision    recall  f1-score   support

indian market       0.63      0.92      0.74       476
        onion       0.76      0.54      0.63       682
       potato       0.85      0.53      0.65       731
       tomato       0.69      0.97      0.81       619

     accuracy                           0.72      2508
    macro avg       0.73      0.74      0.71      2508
 weighted avg       0.74      0.72      0.70      2508


Classification Report - val
              precision    recall  f1-score   support

indian market       0.55      0.89      0.68       123
```

```
        onion       0.54      0.66      0.59       167
       potato       0.86      0.37      0.52       167
       tomato       0.87      0.76      0.82       170

     accuracy                           0.66       627
    macro avg       0.70      0.67      0.65       627
 weighted avg       0.72      0.66      0.65       627


Classification Report - test
               precision    recall  f1-score   support

indian market       0.60      0.80      0.68        81
        onion       0.61      0.82      0.70        83
       potato       0.93      0.32      0.48        81
       tomato       0.90      0.88      0.89       106

     accuracy                           0.72       351
    macro avg       0.76      0.71      0.69       351
 weighted avg       0.77      0.72      0.70       351


Class-wise Accuracy:

Train Class-wise Accuracy:
 indian market: 91.60% (436/476)
 onion: 54.11% (369/682)
 potato: 53.08% (388/731)
 tomato: 97.42% (603/619)

Val Class-wise Accuracy:
 indian market: 89.43% (110/123)
 onion: 65.87% (110/167)
 potato: 37.13% (62/167)
 tomato: 76.47% (130/170)

Test Class-wise Accuracy:
 indian market: 80.25% (65/81)
 onion: 81.93% (68/83)
 potato: 32.10% (26/81)
 tomato: 87.74% (93/106)

Random Test Predictions:
```

True: indian market
Pred: indian market

True: indian market
Pred: indian market

True: onion
Pred: onion

True: onion
Pred: onion

True: potato
Pred: indian market

True: potato
Pred: indian market

True: tomato
Pred: tomato

True: tomato
Pred: tomato

6.4.6 Save the model

[108]:
```
model.save("saved_models/1_pretrained_resnet50_aug_trainable0_model.keras")
```

6.4.7 Observations

- **Architecture**: Deep 50-layer Resnet50 CNN with skip connections pretrained on ImageNet along with two dense layers with batch norm and one dropout.
- **Performance Before augmentation**:
  - **Train Accuracy**: **90.87%** (Require many epochs to improve accuracy)
  - **Validation Accuracy**: **79.59%**
  - **Test Accuracy**: **72.08%** (Highly Overfitted model).
  - **Test Precision/Recall**: ~75% macro-average.
  - **Epochs**: **48**
  - **Training time**: **135 min (approx)**
  - **Class-Specific Issues**:
    - Poor performance on **"potato"** (**38% accuracy**) due to similarity with cluttered "Indian market" scenes.
    - Misclassified **"onion"** as **"potato"**(shape confusion).
- **Strengths**:
  - Handled deeper feature hierarchies better than simpler CNNs.
- **Weaknesses**:
  - Severe overfitting (train accuracy **90.8%** vs. test **72%**).
  - Struggled with noisy backgrounds and class imbalance.
- **Impact of Augmentation**:
  - **Train Accuracy**: **71.61%** (Require change in architecture and epochs)
  - **Validation Accuracy**: **65.71%**
  - **Test Accuracy**: **71.79%** (Overfitting reduced but Underfitting increased).
  - **Test Precision/Recall**: ~75% macro-average.
  - **Epochs**: **20** + load weights from previous RESNET50
  - **Training time**: **60 min (approx)** (Require more epochs to get good model)

6.5 INCEPTIONV3

6.5.1 Building the model

[144]:
```
def pretrained_inceptionv3(height=256, width=256, num_classes=4,␣
↪trainable_layers=0, ckpt_path=None):
    base_model = applications.InceptionV3(weights="imagenet",␣
↪include_top=False, input_shape=(height, width, 3))

    # Freeze all layers initially
```

```
    base_model.trainable = False

    # If fine-tuning, unfreeze last `trainable_layers`
    if trainable_layers > 0:
        for layer in base_model.layers[-trainable_layers:]:
            layer.trainable = True

    model = keras.Sequential([
        base_model,
        layers.GlobalAveragePooling2D(),
        layers.Dense(512, activation="relu", kernel_regularizer=regularizers.
 ↪l2(5e-4)),
        layers.BatchNormalization(),
        layers.Dropout(0.3),

        layers.Dense(256, activation="relu", kernel_regularizer=regularizers.
 ↪l2(5e-4)),
        layers.BatchNormalization(),

        layers.Dense(num_classes, activation="softmax")
    ], name="pretrained_inceptionv3")

    # Load weights only if a valid checkpoint is provided
    if ckpt_path and os.path.exists(ckpt_path):
        print(f"Loading weights from {ckpt_path}")
        model.load_weights(ckpt_path)

    return model
```

[145]:
```
ckpt_path = "checkpoints/1_pretrained_inceptionv3_trainable0.weights.h5"
log_dir = "logs/1_pretrained_inceptionv3_trainable0"

# Load pretrained VGG16 model
model = pretrained_inceptionv3(height=256, width=256, num_classes=4,␣
 ↪trainable_layers=0,ckpt_path=ckpt_path)
```

[146]:
```
model.summary()
```

Model: "pretrained_inceptionv3"

Layer (type)	Output Shape	Param #
inception_v3 (Functional)	(None, 6, 6, 2048)	21,802,784
global_average_pooling2d_12 (GlobalAveragePooling2D)	(None, 2048)	0

dense_36 (Dense)	(None, 512)	1,049,088
batch_normalization_110 (BatchNormalization)	(None, 512)	2,048
dropout_14 (Dropout)	(None, 512)	0
dense_37 (Dense)	(None, 256)	131,328
batch_normalization_111 (BatchNormalization)	(None, 256)	1,024
dense_38 (Dense)	(None, 4)	1,028

Total params: 22,987,300 (87.69 MB)

Trainable params: 1,182,980 (4.51 MB)

Non-trainable params: 21,804,320 (83.18 MB)

6.5.2 Model Compilation and Training

[147]:
```
model_fit = compile_train(model, train_ds, val_ds,␣
↪log_dir=log_dir,epochs=50,ckpt_path=ckpt_path)
```

```
Epoch 1/50
79/79            88s 1s/step -
accuracy: 0.8402 - loss: 1.0339 - precision_11: 0.8697 - recall_11: 0.8139 -
val_accuracy: 0.9346 - val_loss: 0.7284 - val_precision_11: 0.9463 -
val_recall_11: 0.9282 - learning_rate: 0.0010
Epoch 2/50
79/79            112s 1s/step -
accuracy: 0.9688 - loss: 0.6379 - precision_11: 0.9717 - recall_11: 0.9667 -
val_accuracy: 0.9458 - val_loss: 0.6981 - val_precision_11: 0.9485 -
val_recall_11: 0.9394 - learning_rate: 0.0010
Epoch 3/50
79/79            115s 1s/step -
accuracy: 0.9803 - loss: 0.5795 - precision_11: 0.9807 - recall_11: 0.9787 -
val_accuracy: 0.9346 - val_loss: 0.7433 - val_precision_11: 0.9374 -
val_recall_11: 0.9314 - learning_rate: 0.0010
Epoch 4/50
79/79            116s 1s/step -
accuracy: 0.9778 - loss: 0.5403 - precision_11: 0.9781 - recall_11: 0.9769 -
```

```
val_accuracy: 0.9442 - val_loss: 0.7112 - val_precision_11: 0.9456 -
val_recall_11: 0.9426 - learning_rate: 0.0010
Epoch 5/50
79/79            116s 1s/step -
accuracy: 0.9834 - loss: 0.5035 - precision_11: 0.9837 - recall_11: 0.9825 -
val_accuracy: 0.9601 - val_loss: 0.5410 - val_precision_11: 0.9601 -
val_recall_11: 0.9585 - learning_rate: 0.0010
Epoch 6/50
79/79            116s 1s/step -
accuracy: 0.9847 - loss: 0.4618 - precision_11: 0.9848 - recall_11: 0.9844 -
val_accuracy: 0.9314 - val_loss: 0.6270 - val_precision_11: 0.9313 -
val_recall_11: 0.9298 - learning_rate: 0.0010
Epoch 7/50
79/79            117s 1s/step -
accuracy: 0.9924 - loss: 0.4155 - precision_11: 0.9924 - recall_11: 0.9924 -
val_accuracy: 0.9474 - val_loss: 0.5656 - val_precision_11: 0.9489 -
val_recall_11: 0.9474 - learning_rate: 0.0010
Epoch 8/50
79/79            117s 1s/step -
accuracy: 0.9931 - loss: 0.3736 - precision_11: 0.9933 - recall_11: 0.9931 -
val_accuracy: 0.9490 - val_loss: 0.5359 - val_precision_11: 0.9519 -
val_recall_11: 0.9474 - learning_rate: 0.0010
Epoch 9/50
79/79            121s 2s/step -
accuracy: 0.9857 - loss: 0.3715 - precision_11: 0.9857 - recall_11: 0.9857 -
val_accuracy: 0.9569 - val_loss: 0.4644 - val_precision_11: 0.9615 -
val_recall_11: 0.9553 - learning_rate: 0.0010
Epoch 10/50
79/79            116s 1s/step -
accuracy: 0.9891 - loss: 0.3274 - precision_11: 0.9894 - recall_11: 0.9891 -
val_accuracy: 0.9410 - val_loss: 0.5002 - val_precision_11: 0.9410 -
val_recall_11: 0.9410 - learning_rate: 0.0010
Epoch 11/50
79/79            116s 1s/step -
accuracy: 0.9873 - loss: 0.3340 - precision_11: 0.9878 - recall_11: 0.9873 -
val_accuracy: 0.9139 - val_loss: 0.5525 - val_precision_11: 0.9152 -
val_recall_11: 0.9123 - learning_rate: 0.0010
Epoch 12/50
79/79            117s 1s/step -
accuracy: 0.9889 - loss: 0.2901 - precision_11: 0.9890 - recall_11: 0.9889 -
val_accuracy: 0.9091 - val_loss: 0.6122 - val_precision_11: 0.9098 -
val_recall_11: 0.9011 - learning_rate: 0.0010
Epoch 13/50
79/79            116s 1s/step -
accuracy: 0.9871 - loss: 0.2798 - precision_11: 0.9875 - recall_11: 0.9871 -
val_accuracy: 0.9362 - val_loss: 0.4498 - val_precision_11: 0.9377 -
val_recall_11: 0.9362 - learning_rate: 0.0010
Epoch 14/50
```

```
79/79            116s 1s/step -
accuracy: 0.9822 - loss: 0.2908 - precision_11: 0.9833 - recall_11: 0.9810 -
val_accuracy: 0.9266 - val_loss: 0.5284 - val_precision_11: 0.9265 -
val_recall_11: 0.9250 - learning_rate: 0.0010
Epoch 15/50
79/79            117s 1s/step -
accuracy: 0.9754 - loss: 0.3034 - precision_11: 0.9775 - recall_11: 0.9754 -
val_accuracy: 0.9378 - val_loss: 0.4553 - val_precision_11: 0.9378 -
val_recall_11: 0.9378 - learning_rate: 0.0010
Epoch 16/50
79/79            116s 1s/step -
accuracy: 0.9898 - loss: 0.2477 - precision_11: 0.9909 - recall_11: 0.9896 -
val_accuracy: 0.9394 - val_loss: 0.4391 - val_precision_11: 0.9408 -
val_recall_11: 0.9378 - learning_rate: 0.0010
Epoch 17/50
79/79            116s 1s/step -
accuracy: 0.9926 - loss: 0.2278 - precision_11: 0.9926 - recall_11: 0.9926 -
val_accuracy: 0.9442 - val_loss: 0.3898 - val_precision_11: 0.9472 -
val_recall_11: 0.9442 - learning_rate: 0.0010
Epoch 18/50
79/79            116s 1s/step -
accuracy: 0.9942 - loss: 0.2032 - precision_11: 0.9948 - recall_11: 0.9941 -
val_accuracy: 0.9442 - val_loss: 0.4018 - val_precision_11: 0.9456 -
val_recall_11: 0.9426 - learning_rate: 0.0010
Epoch 19/50
79/79            116s 1s/step -
accuracy: 0.9834 - loss: 0.2198 - precision_11: 0.9850 - recall_11: 0.9833 -
val_accuracy: 0.9474 - val_loss: 0.4001 - val_precision_11: 0.9489 -
val_recall_11: 0.9474 - learning_rate: 0.0010
Epoch 20/50
79/79            117s 1s/step -
accuracy: 0.9941 - loss: 0.1923 - precision_11: 0.9941 - recall_11: 0.9941 -
val_accuracy: 0.9314 - val_loss: 0.4691 - val_precision_11: 0.9314 -
val_recall_11: 0.9314 - learning_rate: 0.0010
Epoch 21/50
79/79            115s 1s/step -
accuracy: 0.9932 - loss: 0.1877 - precision_11: 0.9932 - recall_11: 0.9932 -
val_accuracy: 0.9474 - val_loss: 0.3815 - val_precision_11: 0.9487 -
val_recall_11: 0.9442 - learning_rate: 0.0010
Epoch 22/50
79/79            120s 2s/step -
accuracy: 0.9874 - loss: 0.1976 - precision_11: 0.9874 - recall_11: 0.9864 -
val_accuracy: 0.9522 - val_loss: 0.3465 - val_precision_11: 0.9552 -
val_recall_11: 0.9522 - learning_rate: 0.0010
Epoch 23/50
79/79            115s 1s/step -
accuracy: 0.9824 - loss: 0.2282 - precision_11: 0.9836 - recall_11: 0.9822 -
val_accuracy: 0.9458 - val_loss: 0.3607 - val_precision_11: 0.9473 -
```

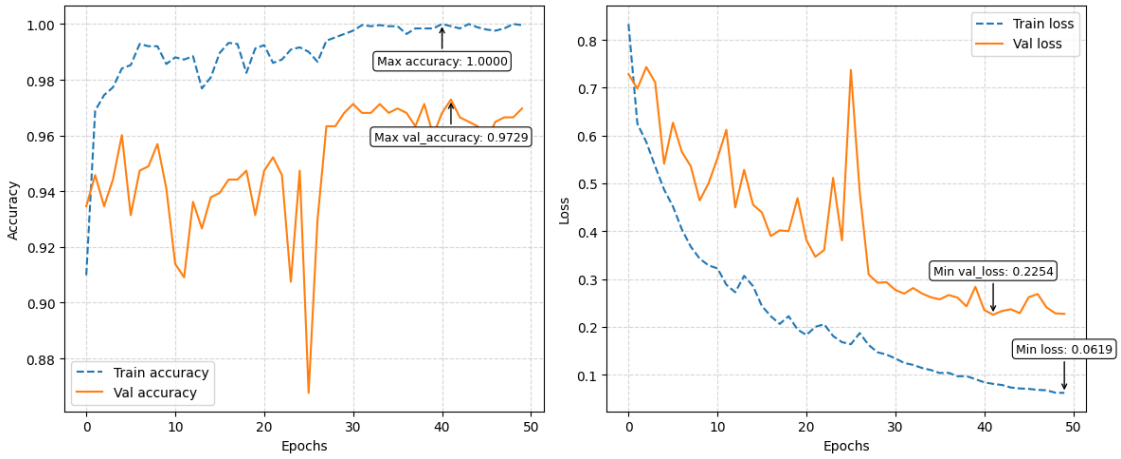
```
val_recall_11: 0.9458 - learning_rate: 0.0010
Epoch 24/50
79/79            115s 1s/step -
accuracy: 0.9915 - loss: 0.1833 - precision_11: 0.9915 - recall_11: 0.9915 -
val_accuracy: 0.9075 - val_loss: 0.5115 - val_precision_11: 0.9075 -
val_recall_11: 0.9075 - learning_rate: 0.0010
Epoch 25/50
79/79            116s 1s/step -
accuracy: 0.9895 - loss: 0.1728 - precision_11: 0.9900 - recall_11: 0.9895 -
val_accuracy: 0.9474 - val_loss: 0.3808 - val_precision_11: 0.9489 -
val_recall_11: 0.9474 - learning_rate: 0.0010
Epoch 26/50
79/79            117s 1s/step -
accuracy: 0.9919 - loss: 0.1630 - precision_11: 0.9919 - recall_11: 0.9915 -
val_accuracy: 0.8676 - val_loss: 0.7373 - val_precision_11: 0.8712 -
val_recall_11: 0.8628 - learning_rate: 0.0010
Epoch 27/50
79/79            116s 1s/step -
accuracy: 0.9850 - loss: 0.1868 - precision_11: 0.9849 - recall_11: 0.9845 -
val_accuracy: 0.9298 - val_loss: 0.4813 - val_precision_11: 0.9297 -
val_recall_11: 0.9282 - learning_rate: 0.0010
Epoch 28/50
79/79            117s 1s/step -
accuracy: 0.9925 - loss: 0.1657 - precision_11: 0.9930 - recall_11: 0.9924 -
val_accuracy: 0.9633 - val_loss: 0.3095 - val_precision_11: 0.9633 -
val_recall_11: 0.9633 - learning_rate: 3.0000e-04
Epoch 29/50
79/79            116s 1s/step -
accuracy: 0.9971 - loss: 0.1463 - precision_11: 0.9971 - recall_11: 0.9971 -
val_accuracy: 0.9633 - val_loss: 0.2921 - val_precision_11: 0.9633 -
val_recall_11: 0.9633 - learning_rate: 3.0000e-04
Epoch 30/50
79/79            117s 1s/step -
accuracy: 0.9988 - loss: 0.1394 - precision_11: 0.9988 - recall_11: 0.9988 -
val_accuracy: 0.9681 - val_loss: 0.2934 - val_precision_11: 0.9681 -
val_recall_11: 0.9681 - learning_rate: 3.0000e-04
Epoch 31/50
79/79            117s 1s/step -
accuracy: 0.9972 - loss: 0.1369 - precision_11: 0.9972 - recall_11: 0.9972 -
val_accuracy: 0.9713 - val_loss: 0.2771 - val_precision_11: 0.9712 -
val_recall_11: 0.9697 - learning_rate: 3.0000e-04
Epoch 32/50
79/79            116s 1s/step -
accuracy: 0.9996 - loss: 0.1257 - precision_11: 0.9996 - recall_11: 0.9996 -
val_accuracy: 0.9681 - val_loss: 0.2694 - val_precision_11: 0.9696 -
val_recall_11: 0.9681 - learning_rate: 3.0000e-04
Epoch 33/50
79/79            116s 1s/step -
```

```
accuracy: 0.9990 - loss: 0.1243 - precision_11: 0.9990 - recall_11: 0.9990 -
val_accuracy: 0.9681 - val_loss: 0.2812 - val_precision_11: 0.9681 -
val_recall_11: 0.9681 - learning_rate: 3.0000e-04
Epoch 34/50
79/79            145s 1s/step -
accuracy: 1.0000 - loss: 0.1145 - precision_11: 1.0000 - recall_11: 1.0000 -
val_accuracy: 0.9713 - val_loss: 0.2697 - val_precision_11: 0.9713 -
val_recall_11: 0.9713 - learning_rate: 3.0000e-04
Epoch 35/50
79/79            116s 1s/step -
accuracy: 0.9988 - loss: 0.1111 - precision_11: 0.9988 - recall_11: 0.9988 -
val_accuracy: 0.9681 - val_loss: 0.2618 - val_precision_11: 0.9681 -
val_recall_11: 0.9665 - learning_rate: 3.0000e-04
Epoch 36/50
79/79            116s 1s/step -
accuracy: 0.9994 - loss: 0.1040 - precision_11: 0.9994 - recall_11: 0.9994 -
val_accuracy: 0.9697 - val_loss: 0.2575 - val_precision_11: 0.9697 -
val_recall_11: 0.9697 - learning_rate: 3.0000e-04
Epoch 37/50
79/79            116s 1s/step -
accuracy: 0.9953 - loss: 0.1060 - precision_11: 0.9959 - recall_11: 0.9953 -
val_accuracy: 0.9681 - val_loss: 0.2666 - val_precision_11: 0.9681 -
val_recall_11: 0.9681 - learning_rate: 3.0000e-04
Epoch 38/50
79/79            116s 1s/step -
accuracy: 0.9996 - loss: 0.0960 - precision_11: 0.9996 - recall_11: 0.9996 -
val_accuracy: 0.9633 - val_loss: 0.2610 - val_precision_11: 0.9633 -
val_recall_11: 0.9633 - learning_rate: 3.0000e-04
Epoch 39/50
79/79            116s 1s/step -
accuracy: 0.9985 - loss: 0.0978 - precision_11: 0.9985 - recall_11: 0.9985 -
val_accuracy: 0.9713 - val_loss: 0.2429 - val_precision_11: 0.9713 -
val_recall_11: 0.9713 - learning_rate: 3.0000e-04
Epoch 40/50
79/79            116s 1s/step -
accuracy: 0.9996 - loss: 0.0893 - precision_11: 0.9996 - recall_11: 0.9996 -
val_accuracy: 0.9601 - val_loss: 0.2837 - val_precision_11: 0.9601 -
val_recall_11: 0.9601 - learning_rate: 3.0000e-04
Epoch 41/50
79/79            115s 1s/step -
accuracy: 1.0000 - loss: 0.0850 - precision_11: 1.0000 - recall_11: 1.0000 -
val_accuracy: 0.9681 - val_loss: 0.2350 - val_precision_11: 0.9681 -
val_recall_11: 0.9665 - learning_rate: 3.0000e-04
Epoch 42/50
79/79            116s 1s/step -
accuracy: 0.9996 - loss: 0.0812 - precision_11: 0.9996 - recall_11: 0.9996 -
val_accuracy: 0.9729 - val_loss: 0.2254 - val_precision_11: 0.9729 -
val_recall_11: 0.9729 - learning_rate: 3.0000e-04
```

```
Epoch 43/50
79/79            117s 1s/step -
accuracy: 0.9990 - loss: 0.0780 - precision_11: 0.9990 - recall_11: 0.9990 -
val_accuracy: 0.9665 - val_loss: 0.2333 - val_precision_11: 0.9665 -
val_recall_11: 0.9665 - learning_rate: 3.0000e-04
Epoch 44/50
79/79            116s 1s/step -
accuracy: 1.0000 - loss: 0.0740 - precision_11: 1.0000 - recall_11: 1.0000 -
val_accuracy: 0.9649 - val_loss: 0.2367 - val_precision_11: 0.9649 -
val_recall_11: 0.9649 - learning_rate: 3.0000e-04
Epoch 45/50
79/79            116s 1s/step -
accuracy: 0.9984 - loss: 0.0720 - precision_11: 0.9984 - recall_11: 0.9984 -
val_accuracy: 0.9633 - val_loss: 0.2286 - val_precision_11: 0.9633 -
val_recall_11: 0.9633 - learning_rate: 3.0000e-04
Epoch 46/50
79/79            116s 1s/step -
accuracy: 0.9977 - loss: 0.0710 - precision_11: 0.9977 - recall_11: 0.9977 -
val_accuracy: 0.9585 - val_loss: 0.2621 - val_precision_11: 0.9601 -
val_recall_11: 0.9585 - learning_rate: 3.0000e-04
Epoch 47/50
79/79            116s 1s/step -
accuracy: 0.9978 - loss: 0.0684 - precision_11: 0.9991 - recall_11: 0.9978 -
val_accuracy: 0.9649 - val_loss: 0.2686 - val_precision_11: 0.9649 -
val_recall_11: 0.9649 - learning_rate: 3.0000e-04
Epoch 48/50
79/79            116s 1s/step -
accuracy: 0.9987 - loss: 0.0671 - precision_11: 0.9988 - recall_11: 0.9987 -
val_accuracy: 0.9665 - val_loss: 0.2407 - val_precision_11: 0.9665 -
val_recall_11: 0.9665 - learning_rate: 9.0000e-05
Epoch 49/50
79/79            116s 1s/step -
accuracy: 1.0000 - loss: 0.0629 - precision_11: 1.0000 - recall_11: 1.0000 -
val_accuracy: 0.9665 - val_loss: 0.2282 - val_precision_11: 0.9681 -
val_recall_11: 0.9665 - learning_rate: 9.0000e-05
Epoch 50/50
79/79            116s 1s/step -
accuracy: 0.9999 - loss: 0.0614 - precision_11: 0.9999 - recall_11: 0.9999 -
val_accuracy: 0.9697 - val_loss: 0.2271 - val_precision_11: 0.9697 -
val_recall_11: 0.9697 - learning_rate: 9.0000e-05
```

6.5.3 Plot loss & accuracy for training and validation wrt epoch

[148]:
```
plot_loss_and_accuracy(["accuracy","loss"],model_fit)
```



6.5.4 Tensorboard

[149]:
```
%load_ext tensorboard
```

```
The tensorboard extension is already loaded. To reload it, use:
  %reload_ext tensorboard
```

6.5.5 Model Evaluation, Mlflow Logging and Printing Metrics

[150]:
```
ckpt_path
```

[150]: 'checkpoints/1_pretrained_inceptionv3_trainable0.weights.h5'

[151]:
```
log_dir
```

[151]: 'logs/1_pretrained_inceptionv3_trainable0'

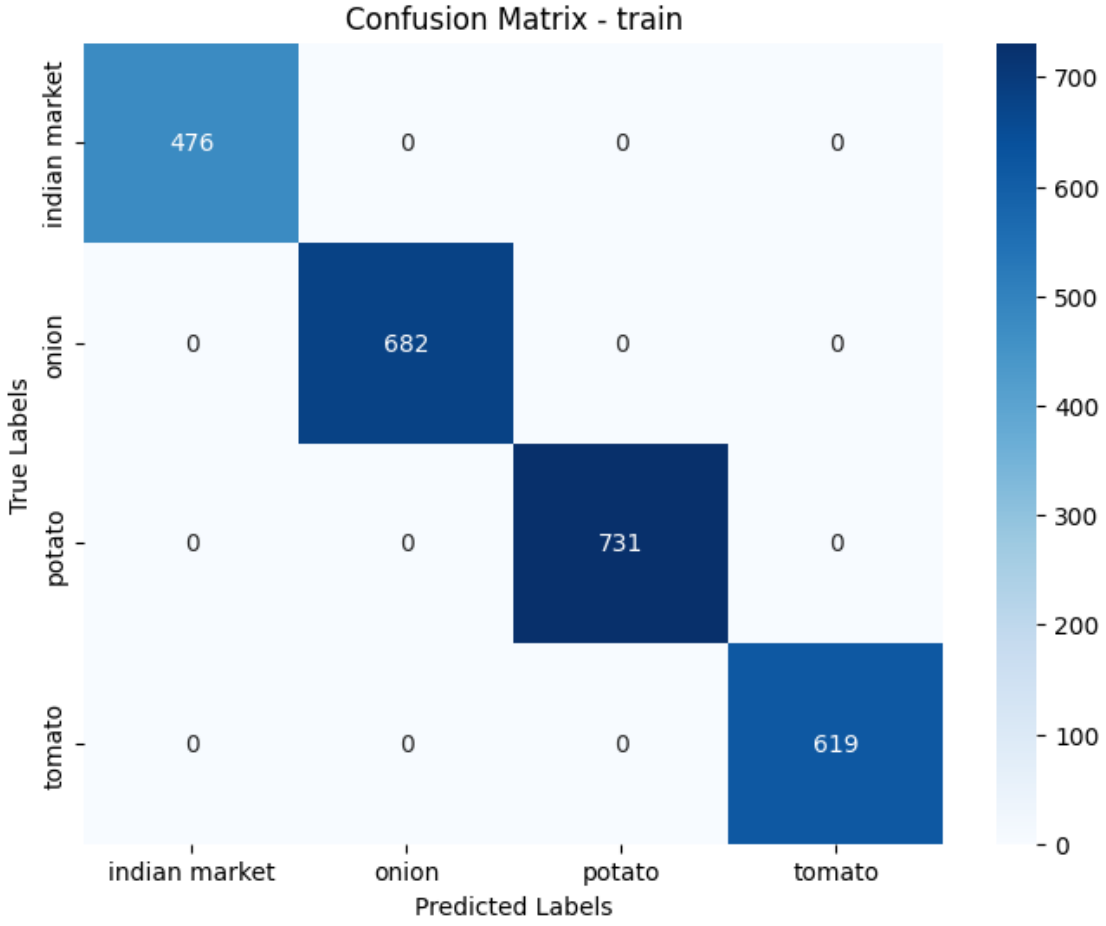
[152]:
```
run_name = "1_pretrained_inceptionv3_trainable0"
run_name
```

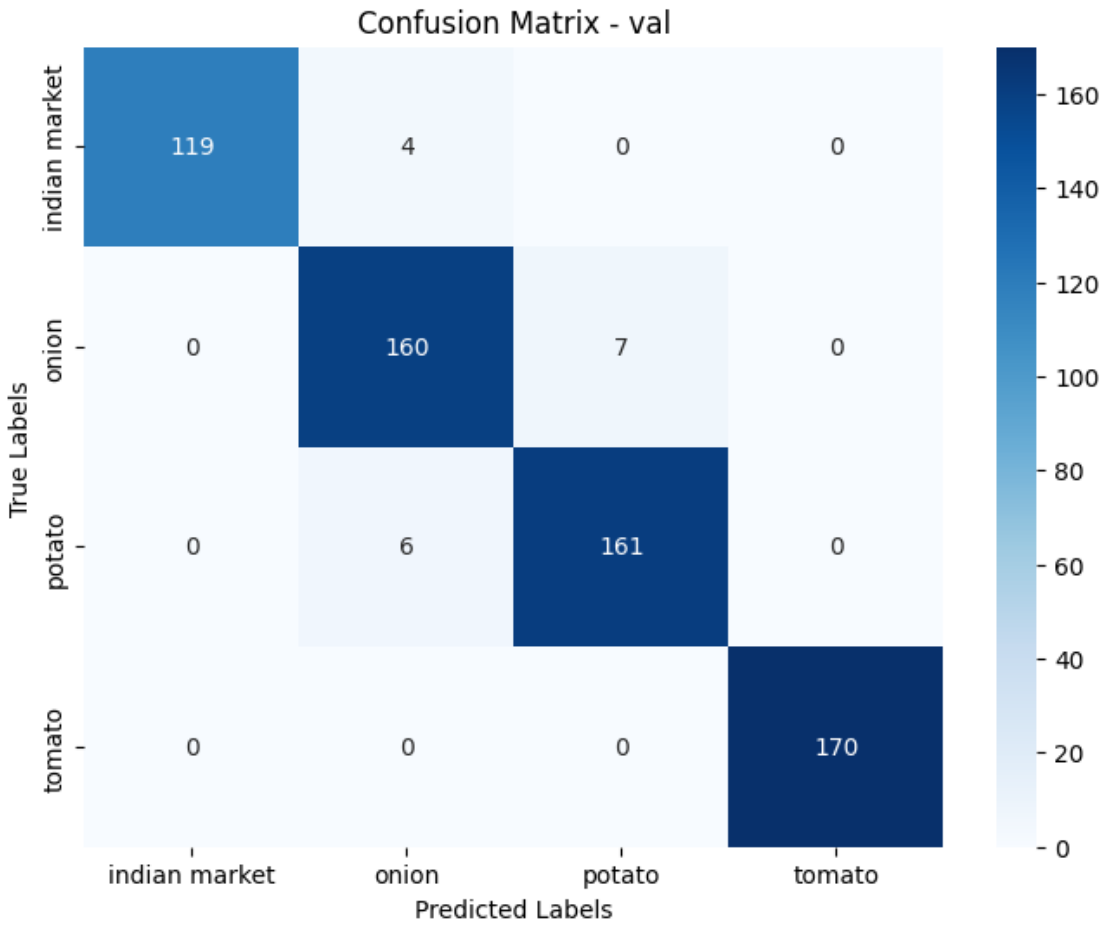
[152]: '1_pretrained_inceptionv3_trainable0'

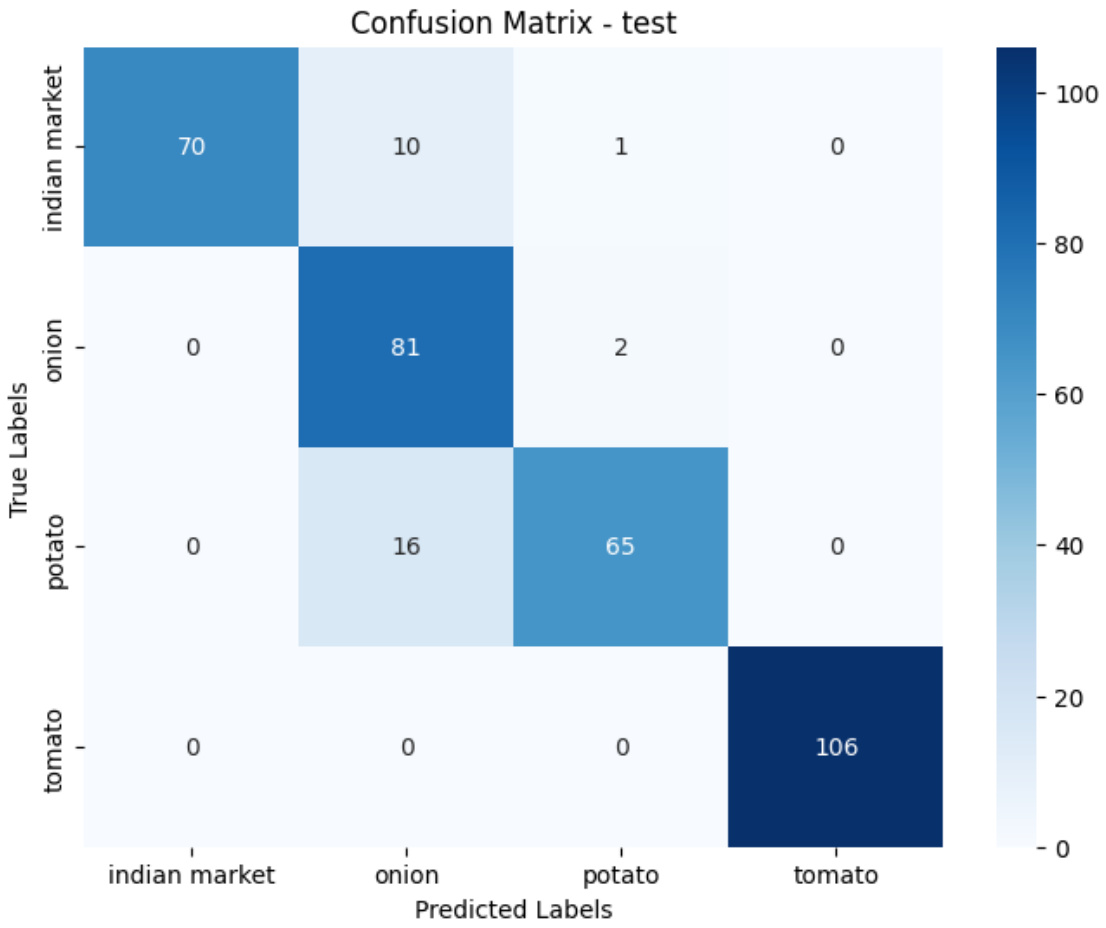
[153]:
```
evaluate_model(model, train_ds, val_ds, test_ds,␣
  ↪class_names,run_name=run_name,ckpt_path= ckpt_path)
```

```
2025/02/11 02:03:20 WARNING mlflow.tensorflow: You are saving a TensorFlow Core
model or Keras model without a signature. Inference with
mlflow.pyfunc.spark_udf() will not work unless the model's pyfunc representation
accepts pandas DataFrames as inference inputs.
2025/02/11 02:03:50 WARNING mlflow.models.model: Model logged without a
signature and input example. Please set `input_example` parameter when logging
the model to auto infer the model signature.
```

```
train Metrics:
 Accuracy: 100.00%
 Precision: 100.00%
 Recall: 100.00%

val Metrics:
 Accuracy: 97.29%
 Precision: 97.49%
 Recall: 97.24%

test Metrics:
 Accuracy: 91.74%
 Precision: 92.82%
 Recall: 91.06%
```




Confusion Matrix - val
True Labels
Predicted Labels
indian market
onion
potato
tomato
119
4
0
0
0
160
7
0
0
6
161
0
0
0
0
170
160
140
120
100
80
60
40
20
0



Classification Report - train

	precision	recall	f1-score	support
indian market	1.00	1.00	1.00	476
onion	1.00	1.00	1.00	682
potato	1.00	1.00	1.00	731
tomato	1.00	1.00	1.00	619
accuracy			1.00	2508
macro avg	1.00	1.00	1.00	2508
weighted avg	1.00	1.00	1.00	2508

Classification Report - val

	precision	recall	f1-score	support
indian market	1.00	0.97	0.98	123

```
        onion       0.94      0.96      0.95       167
       potato       0.96      0.96      0.96       167
       tomato       1.00      1.00      1.00       170

     accuracy                           0.97       627
    macro avg       0.97      0.97      0.97       627
 weighted avg       0.97      0.97      0.97       627


Classification Report - test
               precision    recall  f1-score   support

indian market       1.00      0.86      0.93        81
        onion       0.76      0.98      0.85        83
       potato       0.96      0.80      0.87        81
       tomato       1.00      1.00      1.00       106

     accuracy                           0.92       351
    macro avg       0.93      0.91      0.91       351
 weighted avg       0.93      0.92      0.92       351


Class-wise Accuracy:

Train Class-wise Accuracy:
 indian market: 100.00% (476/476)
 onion: 100.00% (682/682)
 potato: 100.00% (731/731)
 tomato: 100.00% (619/619)

Val Class-wise Accuracy:
 indian market: 96.75% (119/123)
 onion: 95.81% (160/167)
 potato: 96.41% (161/167)
 tomato: 100.00% (170/170)

Test Class-wise Accuracy:
 indian market: 86.42% (70/81)
 onion: 97.59% (81/83)
 potato: 80.25% (65/81)
 tomato: 100.00% (106/106)

Random Test Predictions:
```

True: indian market
Pred: indian market

True: indian market
Pred: indian market

True: onion
Pred: onion

True: onion
Pred: onion

True: potato
Pred: potato

True: potato
Pred: potato

True: tomato
Pred: tomato

True: tomato
Pred: tomato

6.5.6 Save the model

[154]:
```
model.save("saved_models/1_pretrained_inceptionv3_trainable0_model.keras")
```

6.6 DATA AUGMENTED INCEPTION_V3

6.6.1 Load the previously trained model architecture and weights

[189]:
```
# Load the entire model (architecture + weights)
model = keras.models.load_model("saved_models/
 ↪1_pretrained_inceptionv3_trainable0_model.keras")
```

[190]:
```
model.summary()
```

Model: "pretrained_inceptionv3"

Layer (type)	Output Shape	Param #
inception_v3 (Functional)	(None, 6, 6, 2048)	21,802,784
global_average_pooling2d_12 (GlobalAveragePooling2D)	(None, 2048)	0
dense_36 (Dense)	(None, 512)	1,049,088
batch_normalization_110 (BatchNormalization)	(None, 512)	2,048
dropout_14 (Dropout)	(None, 512)	0
dense_37 (Dense)	(None, 256)	131,328
batch_normalization_111 (BatchNormalization)	(None, 256)	1,024
dense_38 (Dense)	(None, 4)	1,028

Total params: 25,353,262 (96.72 MB)

Trainable params: 1,182,980 (4.51 MB)

Non-trainable params: 21,804,320 (83.18 MB)

```
 Optimizer params: 2,365,962 (9.03 MB)
```

6.6.2 Model Compilation and Training

[191]:
```
ckpt_path = "checkpoints/2_pretrained_inceptionv3_aug_trainable0.weights.h5"
```

[192]:
```
log_dir = 'logs/2_pretrained_inceptionv3_aug_trainable0'
```

[193]:
```
run_name = '2_pretrained_inceptionv3_aug_trainable0'
```

[194]:
```
model_fit = compile_train(model, train_aug_ds, val_aug_ds,␣
 ↪log_dir=log_dir,epochs=20,ckpt_path=ckpt_path)
```

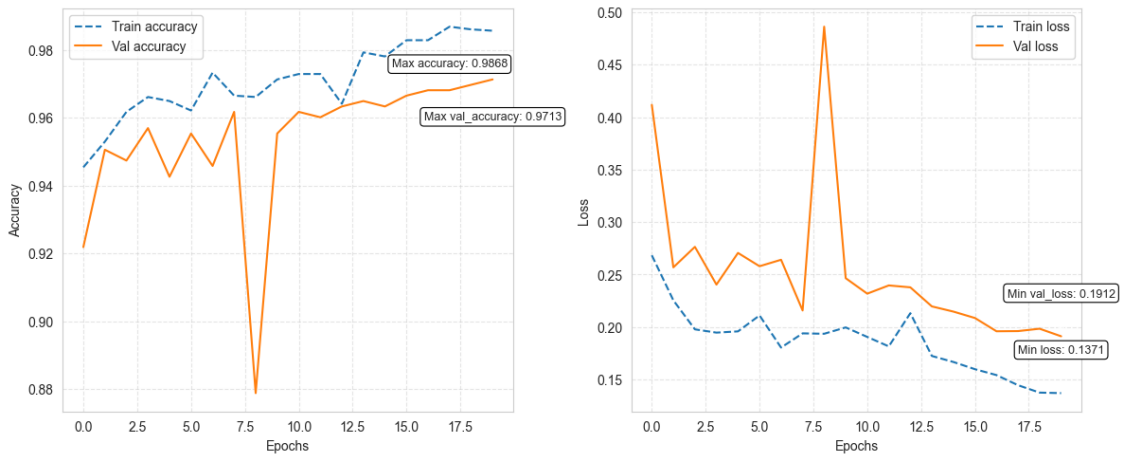
```
Epoch 1/20
79/79             176s 2s/step -
accuracy: 0.9476 - loss: 0.2690 - precision_16: 0.9480 - recall_16: 0.9476 -
val_accuracy: 0.9219 - val_loss: 0.4117 - val_precision_16: 0.9232 -
val_recall_16: 0.9203 - learning_rate: 0.0010
Epoch 2/20
79/79             132s 2s/step -
accuracy: 0.9526 - loss: 0.2313 - precision_16: 0.9539 - recall_16: 0.9526 -
val_accuracy: 0.9506 - val_loss: 0.2569 - val_precision_16: 0.9520 -
val_recall_16: 0.9490 - learning_rate: 0.0010
Epoch 3/20
79/79             132s 2s/step -
accuracy: 0.9697 - loss: 0.1735 - precision_16: 0.9706 - recall_16: 0.9696 -
val_accuracy: 0.9474 - val_loss: 0.2764 - val_precision_16: 0.9473 -
val_recall_16: 0.9458 - learning_rate: 0.0010
Epoch 4/20
79/79             133s 2s/step -
accuracy: 0.9637 - loss: 0.1954 - precision_16: 0.9670 - recall_16: 0.9636 -
val_accuracy: 0.9569 - val_loss: 0.2405 - val_precision_16: 0.9569 -
val_recall_16: 0.9569 - learning_rate: 0.0010
Epoch 5/20
79/79             130s 2s/step -
accuracy: 0.9655 - loss: 0.1924 - precision_16: 0.9674 - recall_16: 0.9629 -
val_accuracy: 0.9426 - val_loss: 0.2706 - val_precision_16: 0.9426 -
val_recall_16: 0.9426 - learning_rate: 0.0010
Epoch 6/20
79/79             132s 2s/step -
accuracy: 0.9631 - loss: 0.2091 - precision_16: 0.9653 - recall_16: 0.9622 -
val_accuracy: 0.9553 - val_loss: 0.2579 - val_precision_16: 0.9553 -
val_recall_16: 0.9553 - learning_rate: 0.0010
Epoch 7/20
79/79             131s 2s/step -
accuracy: 0.9723 - loss: 0.1822 - precision_16: 0.9723 - recall_16: 0.9723 -
val_accuracy: 0.9458 - val_loss: 0.2641 - val_precision_16: 0.9473 -
```

```
val_recall_16: 0.9458 - learning_rate: 0.0010
Epoch 8/20
79/79            133s 2s/step -
accuracy: 0.9699 - loss: 0.1856 - precision_16: 0.9698 - recall_16: 0.9659 -
val_accuracy: 0.9617 - val_loss: 0.2158 - val_precision_16: 0.9617 -
val_recall_16: 0.9617 - learning_rate: 0.0010
Epoch 9/20
79/79            131s 2s/step -
accuracy: 0.9651 - loss: 0.1927 - precision_16: 0.9653 - recall_16: 0.9642 -
val_accuracy: 0.8788 - val_loss: 0.4861 - val_precision_16: 0.8812 -
val_recall_16: 0.8756 - learning_rate: 0.0010
Epoch 10/20
79/79            132s 2s/step -
accuracy: 0.9685 - loss: 0.2010 - precision_16: 0.9687 - recall_16: 0.9674 -
val_accuracy: 0.9553 - val_loss: 0.2465 - val_precision_16: 0.9569 -
val_recall_16: 0.9553 - learning_rate: 0.0010
Epoch 11/20
79/79            132s 2s/step -
accuracy: 0.9674 - loss: 0.1991 - precision_16: 0.9680 - recall_16: 0.9652 -
val_accuracy: 0.9617 - val_loss: 0.2318 - val_precision_16: 0.9617 -
val_recall_16: 0.9601 - learning_rate: 0.0010
Epoch 12/20
79/79            132s 2s/step -
accuracy: 0.9701 - loss: 0.1889 - precision_16: 0.9718 - recall_16: 0.9692 -
val_accuracy: 0.9601 - val_loss: 0.2396 - val_precision_16: 0.9617 -
val_recall_16: 0.9601 - learning_rate: 0.0010
Epoch 13/20
79/79            133s 2s/step -
accuracy: 0.9686 - loss: 0.1994 - precision_16: 0.9697 - recall_16: 0.9686 -
val_accuracy: 0.9633 - val_loss: 0.2378 - val_precision_16: 0.9649 -
val_recall_16: 0.9633 - learning_rate: 0.0010
Epoch 14/20
79/79            132s 2s/step -
accuracy: 0.9785 - loss: 0.1748 - precision_16: 0.9785 - recall_16: 0.9764 -
val_accuracy: 0.9649 - val_loss: 0.2197 - val_precision_16: 0.9649 -
val_recall_16: 0.9649 - learning_rate: 3.0000e-04
Epoch 15/20
79/79            132s 2s/step -
accuracy: 0.9741 - loss: 0.1726 - precision_16: 0.9752 - recall_16: 0.9736 -
val_accuracy: 0.9633 - val_loss: 0.2147 - val_precision_16: 0.9649 -
val_recall_16: 0.9633 - learning_rate: 3.0000e-04
Epoch 16/20
79/79            134s 2s/step -
accuracy: 0.9863 - loss: 0.1569 - precision_16: 0.9877 - recall_16: 0.9844 -
val_accuracy: 0.9665 - val_loss: 0.2086 - val_precision_16: 0.9681 -
val_recall_16: 0.9665 - learning_rate: 3.0000e-04
Epoch 17/20
79/79            134s 2s/step -
```

```
accuracy: 0.9838 - loss: 0.1547 - precision_16: 0.9852 - recall_16: 0.9825 -
val_accuracy: 0.9681 - val_loss: 0.1960 - val_precision_16: 0.9696 -
val_recall_16: 0.9681 - learning_rate: 3.0000e-04
Epoch 18/20
79/79              131s 2s/step -
accuracy: 0.9849 - loss: 0.1522 - precision_16: 0.9848 - recall_16: 0.9809 -
val_accuracy: 0.9681 - val_loss: 0.1961 - val_precision_16: 0.9681 -
val_recall_16: 0.9681 - learning_rate: 3.0000e-04
Epoch 19/20
79/79              132s 2s/step -
accuracy: 0.9869 - loss: 0.1362 - precision_16: 0.9872 - recall_16: 0.9868 -
val_accuracy: 0.9697 - val_loss: 0.1984 - val_precision_16: 0.9697 -
val_recall_16: 0.9697 - learning_rate: 3.0000e-04
Epoch 20/20
79/79              133s 2s/step -
accuracy: 0.9854 - loss: 0.1391 - precision_16: 0.9858 - recall_16: 0.9852 -
val_accuracy: 0.9713 - val_loss: 0.1912 - val_precision_16: 0.9712 -
val_recall_16: 0.9697 - learning_rate: 3.0000e-04
```

6.6.3 Plot loss & accuracy for training and validation wrt epoch

[195]:
```
plot_loss_and_accuracy(["accuracy","loss"],model_fit)
```



6.6.4 Tensorboard

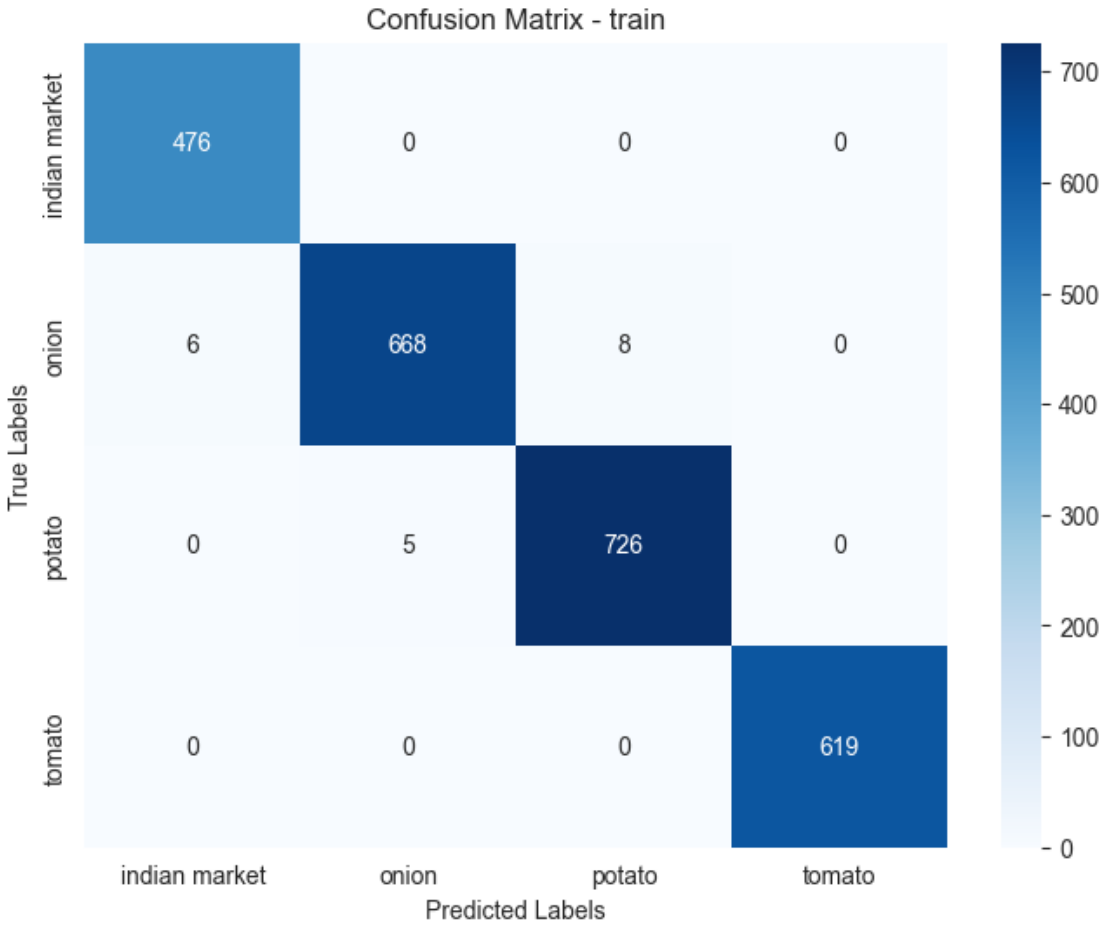
[163]:
```
%load_ext tensorboard
```

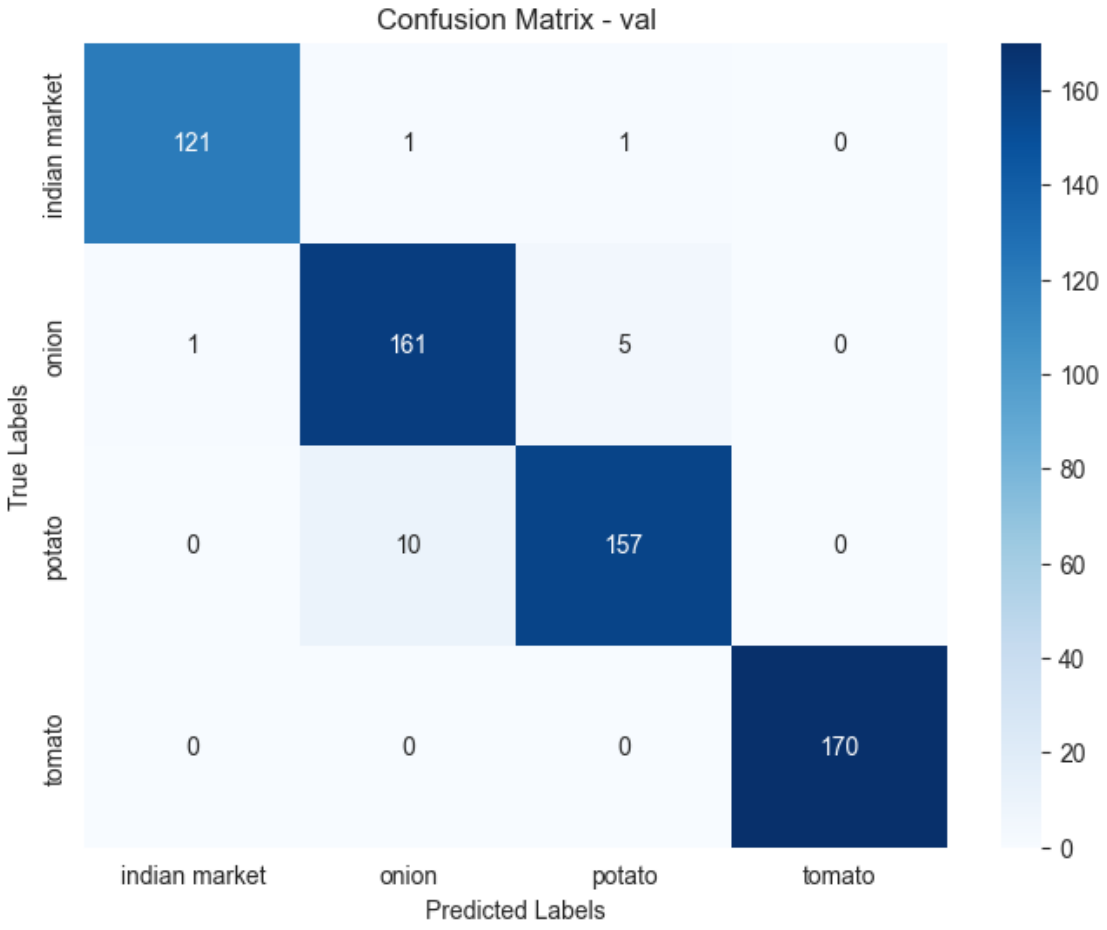
```
The tensorboard extension is already loaded. To reload it, use:
  %reload_ext tensorboard
```

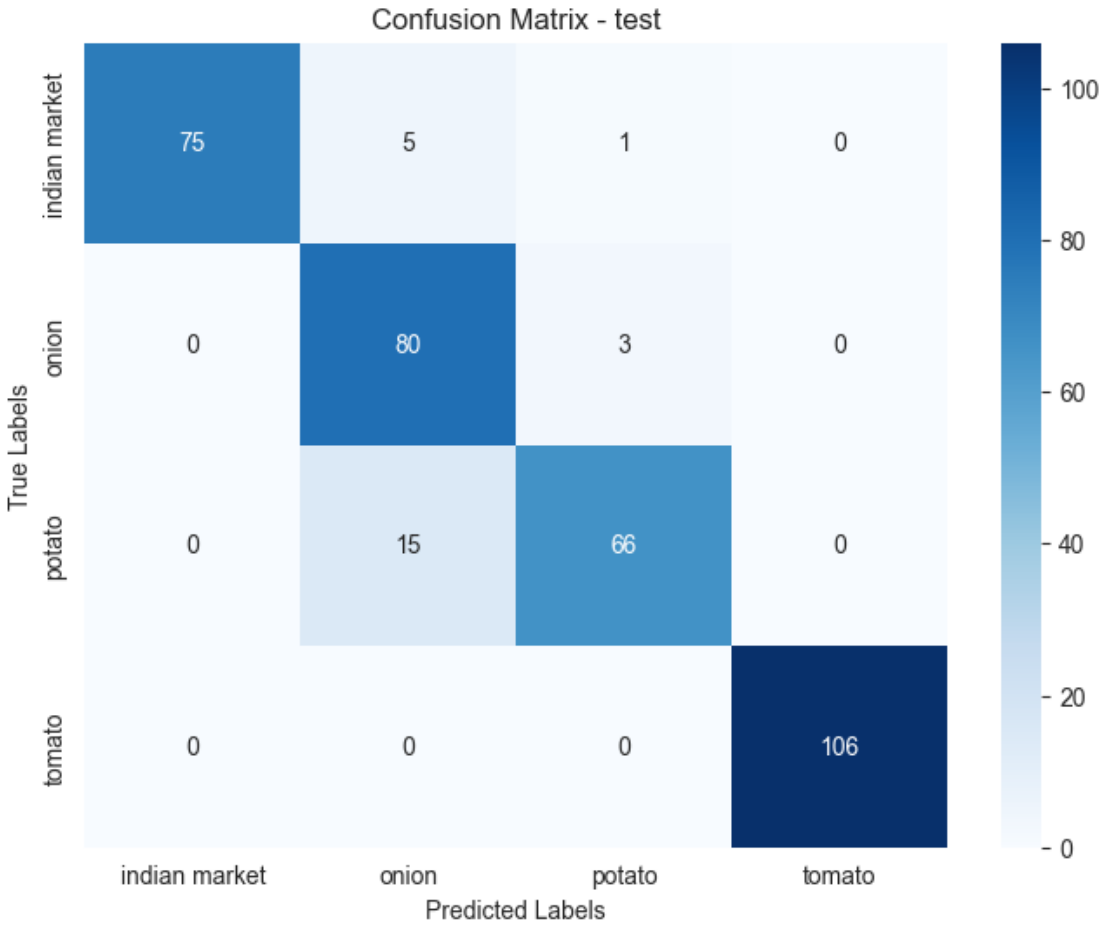
6.6.5 Model Evaluation, Mlflow Logging and Printing Metrics

[196]:
```
evaluate_model(model, train_aug_ds, val_aug_ds, test_aug_ds,␣
↪class_names,run_name=run_name,ckpt_path= ckpt_path)
```

```
2025/02/11 05:56:43 WARNING mlflow.tensorflow: You are saving a TensorFlow Core
model or Keras model without a signature. Inference with
mlflow.pyfunc.spark_udf() will not work unless the model's pyfunc representation
accepts pandas DataFrames as inference inputs.
2025/02/11 05:57:11 WARNING mlflow.models.model: Model logged without a
signature and input example. Please set `input_example` parameter when logging
the model to auto infer the model signature.


train Metrics:
 Accuracy: 99.24%
 Precision: 99.23%
 Recall: 99.32%

val Metrics:
 Accuracy: 97.13%
 Precision: 97.28%
 Recall: 97.20%

test Metrics:
 Accuracy: 93.16%
 Precision: 93.57%
 Recall: 92.61%
```


Confusion Matrix - train
True Labels
indian market
onion
potato
tomato
476
0
0
0
6
668
8
0
0
5
726
0
0
0
0
619
indian market
onion
potato
tomato
Predicted Labels
700
600
500
400
300
200
100
0


Confusion Matrix - val
True Labels
indian market
onion
potato
tomato
121
1
1
0
1
161
5
0
0
10
157
0
0
0
0
170
indian market
onion
potato
tomato
Predicted Labels
160
140
120
100
80
60
40
20
0




```
Classification Report - train
              precision    recall  f1-score   support

indian market       0.99      1.00      0.99       476
        onion       0.99      0.98      0.99       682
       potato       0.99      0.99      0.99       731
       tomato       1.00      1.00      1.00       619

     accuracy                           0.99      2508
    macro avg       0.99      0.99      0.99      2508
 weighted avg       0.99      0.99      0.99      2508


Classification Report - val
              precision    recall  f1-score   support

indian market       0.99      0.98      0.99       123
```

```
        onion       0.94      0.96      0.95       167
       potato       0.96      0.94      0.95       167
       tomato       1.00      1.00      1.00       170

     accuracy                           0.97       627
    macro avg       0.97      0.97      0.97       627
 weighted avg       0.97      0.97      0.97       627


Classification Report - test
               precision    recall  f1-score   support

indian market       1.00      0.93      0.96        81
        onion       0.80      0.96      0.87        83
       potato       0.94      0.81      0.87        81
       tomato       1.00      1.00      1.00       106

     accuracy                           0.93       351
    macro avg       0.94      0.93      0.93       351
 weighted avg       0.94      0.93      0.93       351


Class-wise Accuracy:

Train Class-wise Accuracy:
 indian market: 100.00% (476/476)
 onion: 97.95% (668/682)
 potato: 99.32% (726/731)
 tomato: 100.00% (619/619)

Val Class-wise Accuracy:
 indian market: 98.37% (121/123)
 onion: 96.41% (161/167)
 potato: 94.01% (157/167)
 tomato: 100.00% (170/170)

Test Class-wise Accuracy:
 indian market: 92.59% (75/81)
 onion: 96.39% (80/83)
 potato: 81.48% (66/81)
 tomato: 100.00% (106/106)

Random Test Predictions:
```

True: indian market
Pred: indian market

True: indian market
Pred: indian market

True: onion
Pred: onion

True: onion
Pred: onion

True: potato
Pred: potato

True: potato
Pred: onion

True: tomato
Pred: tomato

True: tomato
Pred: tomato

6.6.6 Save the model

[197]:
```
model.save("saved_models/2_pretrained_inceptionv3_aug_trainable0_model.keras")
```

6.6.7 Observations

- **Architecture**: Deep Multi-Scale Inception V3 CNN with parallel connections pretrained on ImageNet along with two dense layers with batch norm and one dropout.
- **Performance Before augmentation**:
  - **Train Accuracy**: **100%**
  - **Validation Accuracy**: **97.29%**
  - **Test Accuracy**: **91.74%**

  - **Test Precision/Recall**: ~92% macro-average.
  - **Epochs**: **50**
  - **Training time**: **97 min (approx)**
  - **Observed Issues**:
    * Training instability (fluctuating validation accuracy between **30–60%**).

    * Likely suffered from **vanishing gradients** due to complex architecture.

- **Strengths**:
  - Designed for multi-scale feature extraction (theoretically suitable for varied vegetable sizes).

- **Weaknesses**:
  - Resource-intensive for deployment
- **Impact of Augmentation**:
  - **Train Accuracy**: **99.24%** (Robustness increased with augmentation)
  - **Validation Accuracy**: **97.13%**
  - **Test Accuracy**: **93.16%** (Best pretrained heavy weight model).

  - **Test Precision/Recall**: ~94% macro-average.
  - **Epochs**: **20** + load weights from previous InceptionV3
  - **Training time**: **45 min (approx)** (Can increase the epochs to improve more)

6.7 MOBILENETV2

6.7.1 Building the model

[166]:
```
def pretrained_mobilenetv2(height=256, width=256, num_classes=4,␣
↪trainable_layers=0, ckpt_path=None):
    base_model = applications.MobileNetV2(weights="imagenet",␣
↪include_top=False, input_shape=(height, width, 3))

    # Freeze all layers initially
    base_model.trainable = False
```

```
    # If fine-tuning, unfreeze last `trainable_layers`
    if trainable_layers > 0:
        for layer in base_model.layers[-trainable_layers:]:
            layer.trainable = True

    model = keras.Sequential([
        base_model,
        layers.GlobalAveragePooling2D(),
        layers.Dense(512, activation="relu", kernel_regularizer=regularizers.
 ↪l2(5e-4)),
        layers.BatchNormalization(),
        layers.Dropout(0.3),

        layers.Dense(256, activation="relu", kernel_regularizer=regularizers.
 ↪l2(5e-4)),
        layers.BatchNormalization(),

        layers.Dense(num_classes, activation="softmax")
    ], name="pretrained_mobilenetv2")

    # Load weights only if a valid checkpoint is provided
    if ckpt_path and os.path.exists(ckpt_path):
        print(f"Loading weights from {ckpt_path}")
        model.load_weights(ckpt_path)

    return model
```

[167]:
```
ckpt_path = "checkpoints/1_pretrained_mobilenetv2_trainable0.weights.h5"
log_dir = "logs/1_pretrained_mobilenetv2_trainable0"

# Load pretrained mobilenetv2 model
model = pretrained_mobilenetv2(height=256, width=256, num_classes=4,␣
 ↪trainable_layers=0,ckpt_path=ckpt_path)
```

```
C:\Users\saina\AppData\Local\Temp\ipykernel_11564\411930353.py:2: UserWarning:
`input_shape` is undefined or non-square, or `rows` is not in [96, 128, 160,
192, 224]. Weights for input shape (224, 224) will be loaded as the default.
  base_model = applications.MobileNetV2(weights="imagenet", include_top=False,
input_shape=(height, width, 3))

Downloading data from https://storage.googleapis.com/tensorflow/keras-applicatio
ns/mobilenet_v2/mobilenet_v2_weights_tf_dim_ordering_tf_kernels_1.0_224_no_top.h
5
9406464/9406464            2s
0us/step
```

[168]:
```
model.summary()
```

```
Model: "pretrained_mobilenetv2"
```

Layer (type)	Output Shape	Param #
mobilenetv2_1.00_224 (Functional)	(None, 8, 8, 1280)	2,257,984
global_average_pooling2d_13 (GlobalAveragePooling2D)	(None, 1280)	0
dense_39 (Dense)	(None, 512)	655,872
batch_normalization_112 (BatchNormalization)	(None, 512)	2,048
dropout_15 (Dropout)	(None, 512)	0
dense_40 (Dense)	(None, 256)	131,328
batch_normalization_113 (BatchNormalization)	(None, 256)	1,024
dense_41 (Dense)	(None, 4)	1,028

```
Total params: 3,049,284 (11.63 MB)

Trainable params: 789,764 (3.01 MB)

Non-trainable params: 2,259,520 (8.62 MB)
```

6.7.2 Model Compilation and Training

[169]:
```
model_fit = compile_train(model, train_ds, val_ds,␣
↪log_dir=log_dir,epochs=50,ckpt_path=ckpt_path)
```

```
Epoch 1/50
79/79             57s 667ms/step -
accuracy: 0.8482 - loss: 0.9569 - precision_14: 0.8636 - recall_14: 0.8352 -
val_accuracy: 0.9633 - val_loss: 0.6440 - val_precision_14: 0.9633 -
val_recall_14: 0.9617 - learning_rate: 0.0010
Epoch 2/50
79/79             50s 633ms/step -
accuracy: 0.9781 - loss: 0.5865 - precision_14: 0.9791 - recall_14: 0.9764 -
```

```
val_accuracy: 0.9745 - val_loss: 0.5820 - val_precision_14: 0.9760 -
val_recall_14: 0.9729 - learning_rate: 0.0010
Epoch 3/50
79/79            50s 632ms/step -
accuracy: 0.9878 - loss: 0.5278 - precision_14: 0.9878 - recall_14: 0.9875 -
val_accuracy: 0.9697 - val_loss: 0.6175 - val_precision_14: 0.9696 -
val_recall_14: 0.9665 - learning_rate: 0.0010
Epoch 4/50
79/79            50s 637ms/step -
accuracy: 0.9903 - loss: 0.4948 - precision_14: 0.9911 - recall_14: 0.9896 -
val_accuracy: 0.9649 - val_loss: 0.6096 - val_precision_14: 0.9649 -
val_recall_14: 0.9649 - learning_rate: 0.0010
Epoch 5/50
79/79            52s 657ms/step -
accuracy: 0.9914 - loss: 0.4669 - precision_14: 0.9914 - recall_14: 0.9914 -
val_accuracy: 0.9809 - val_loss: 0.4762 - val_precision_14: 0.9808 -
val_recall_14: 0.9793 - learning_rate: 0.0010
Epoch 6/50
79/79            51s 645ms/step -
accuracy: 0.9901 - loss: 0.4354 - precision_14: 0.9901 - recall_14: 0.9900 -
val_accuracy: 0.9729 - val_loss: 0.5023 - val_precision_14: 0.9744 -
val_recall_14: 0.9697 - learning_rate: 0.0010
Epoch 7/50
79/79            51s 647ms/step -
accuracy: 0.9946 - loss: 0.3968 - precision_14: 0.9946 - recall_14: 0.9926 -
val_accuracy: 0.9856 - val_loss: 0.4153 - val_precision_14: 0.9856 -
val_recall_14: 0.9856 - learning_rate: 0.0010
Epoch 8/50
79/79            51s 648ms/step -
accuracy: 0.9931 - loss: 0.3662 - precision_14: 0.9939 - recall_14: 0.9931 -
val_accuracy: 0.9777 - val_loss: 0.4443 - val_precision_14: 0.9777 -
val_recall_14: 0.9777 - learning_rate: 0.0010
Epoch 9/50
79/79            51s 645ms/step -
accuracy: 0.9947 - loss: 0.3435 - precision_14: 0.9947 - recall_14: 0.9947 -
val_accuracy: 0.9777 - val_loss: 0.3905 - val_precision_14: 0.9777 -
val_recall_14: 0.9777 - learning_rate: 0.0010
Epoch 10/50
79/79            51s 640ms/step -
accuracy: 0.9870 - loss: 0.3349 - precision_14: 0.9871 - recall_14: 0.9870 -
val_accuracy: 0.9713 - val_loss: 0.4037 - val_precision_14: 0.9713 -
val_recall_14: 0.9713 - learning_rate: 0.0010
Epoch 11/50
79/79            51s 639ms/step -
accuracy: 0.9928 - loss: 0.3069 - precision_14: 0.9929 - recall_14: 0.9926 -
val_accuracy: 0.9585 - val_loss: 0.4291 - val_precision_14: 0.9601 -
val_recall_14: 0.9585 - learning_rate: 0.0010
Epoch 12/50
```

```
79/79            51s 641ms/step -
accuracy: 0.9929 - loss: 0.2990 - precision_14: 0.9929 - recall_14: 0.9926 -
val_accuracy: 0.9809 - val_loss: 0.3325 - val_precision_14: 0.9809 -
val_recall_14: 0.9809 - learning_rate: 0.0010
Epoch 13/50
79/79            51s 641ms/step -
accuracy: 0.9927 - loss: 0.2729 - precision_14: 0.9927 - recall_14: 0.9927 -
val_accuracy: 0.9809 - val_loss: 0.3055 - val_precision_14: 0.9809 -
val_recall_14: 0.9809 - learning_rate: 0.0010
Epoch 14/50
79/79            50s 635ms/step -
accuracy: 0.9901 - loss: 0.2665 - precision_14: 0.9904 - recall_14: 0.9901 -
val_accuracy: 0.9601 - val_loss: 0.3600 - val_precision_14: 0.9632 -
val_recall_14: 0.9601 - learning_rate: 0.0010
Epoch 15/50
79/79            50s 636ms/step -
accuracy: 0.9860 - loss: 0.2667 - precision_14: 0.9860 - recall_14: 0.9860 -
val_accuracy: 0.9793 - val_loss: 0.3043 - val_precision_14: 0.9808 -
val_recall_14: 0.9793 - learning_rate: 0.0010
Epoch 16/50
79/79            51s 638ms/step -
accuracy: 0.9889 - loss: 0.2464 - precision_14: 0.9893 - recall_14: 0.9889 -
val_accuracy: 0.9825 - val_loss: 0.2916 - val_precision_14: 0.9825 -
val_recall_14: 0.9825 - learning_rate: 0.0010
Epoch 17/50
79/79            50s 636ms/step -
accuracy: 0.9960 - loss: 0.2146 - precision_14: 0.9963 - recall_14: 0.9954 -
val_accuracy: 0.9793 - val_loss: 0.2851 - val_precision_14: 0.9792 -
val_recall_14: 0.9777 - learning_rate: 0.0010
Epoch 18/50
79/79            50s 637ms/step -
accuracy: 0.9957 - loss: 0.2020 - precision_14: 0.9957 - recall_14: 0.9957 -
val_accuracy: 0.9745 - val_loss: 0.2747 - val_precision_14: 0.9760 -
val_recall_14: 0.9745 - learning_rate: 0.0010
Epoch 19/50
79/79            51s 641ms/step -
accuracy: 0.9902 - loss: 0.2076 - precision_14: 0.9909 - recall_14: 0.9899 -
val_accuracy: 0.9777 - val_loss: 0.2714 - val_precision_14: 0.9777 -
val_recall_14: 0.9777 - learning_rate: 0.0010
Epoch 20/50
79/79            51s 647ms/step -
accuracy: 0.9895 - loss: 0.2001 - precision_14: 0.9895 - recall_14: 0.9885 -
val_accuracy: 0.9888 - val_loss: 0.2130 - val_precision_14: 0.9888 -
val_recall_14: 0.9872 - learning_rate: 0.0010
Epoch 21/50
79/79            50s 638ms/step -
accuracy: 0.9948 - loss: 0.1813 - precision_14: 0.9948 - recall_14: 0.9948 -
val_accuracy: 0.9809 - val_loss: 0.2593 - val_precision_14: 0.9809 -
```

```
val_recall_14: 0.9809 - learning_rate: 0.0010
Epoch 22/50
79/79            51s 638ms/step -
accuracy: 0.9971 - loss: 0.1693 - precision_14: 0.9975 - recall_14: 0.9971 -
val_accuracy: 0.9569 - val_loss: 0.3318 - val_precision_14: 0.9569 -
val_recall_14: 0.9553 - learning_rate: 0.0010
Epoch 23/50
79/79            52s 658ms/step -
accuracy: 0.9915 - loss: 0.1844 - precision_14: 0.9917 - recall_14: 0.9915 -
val_accuracy: 0.9793 - val_loss: 0.2264 - val_precision_14: 0.9793 -
val_recall_14: 0.9793 - learning_rate: 0.0010
Epoch 24/50
79/79            51s 638ms/step -
accuracy: 0.9907 - loss: 0.1773 - precision_14: 0.9910 - recall_14: 0.9907 -
val_accuracy: 0.9713 - val_loss: 0.2572 - val_precision_14: 0.9713 -
val_recall_14: 0.9713 - learning_rate: 0.0010
Epoch 25/50
79/79            51s 646ms/step -
accuracy: 0.9937 - loss: 0.1741 - precision_14: 0.9947 - recall_14: 0.9937 -
val_accuracy: 0.9697 - val_loss: 0.2787 - val_precision_14: 0.9697 -
val_recall_14: 0.9697 - learning_rate: 0.0010
Epoch 26/50
79/79            51s 639ms/step -
accuracy: 0.9929 - loss: 0.1683 - precision_14: 0.9929 - recall_14: 0.9929 -
val_accuracy: 0.9793 - val_loss: 0.2126 - val_precision_14: 0.9793 -
val_recall_14: 0.9793 - learning_rate: 3.0000e-04
Epoch 27/50
79/79            51s 640ms/step -
accuracy: 0.9969 - loss: 0.1469 - precision_14: 0.9969 - recall_14: 0.9969 -
val_accuracy: 0.9841 - val_loss: 0.2013 - val_precision_14: 0.9841 -
val_recall_14: 0.9841 - learning_rate: 3.0000e-04
Epoch 28/50
79/79            51s 642ms/step -
accuracy: 0.9983 - loss: 0.1398 - precision_14: 0.9983 - recall_14: 0.9983 -
val_accuracy: 0.9809 - val_loss: 0.2020 - val_precision_14: 0.9809 -
val_recall_14: 0.9809 - learning_rate: 3.0000e-04
Epoch 29/50
79/79            51s 640ms/step -
accuracy: 0.9981 - loss: 0.1348 - precision_14: 0.9981 - recall_14: 0.9981 -
val_accuracy: 0.9841 - val_loss: 0.1888 - val_precision_14: 0.9841 -
val_recall_14: 0.9841 - learning_rate: 3.0000e-04
Epoch 30/50
79/79            51s 640ms/step -
accuracy: 1.0000 - loss: 0.1264 - precision_14: 1.0000 - recall_14: 1.0000 -
val_accuracy: 0.9809 - val_loss: 0.1849 - val_precision_14: 0.9809 -
val_recall_14: 0.9809 - learning_rate: 3.0000e-04
Epoch 31/50
79/79            51s 644ms/step -
```

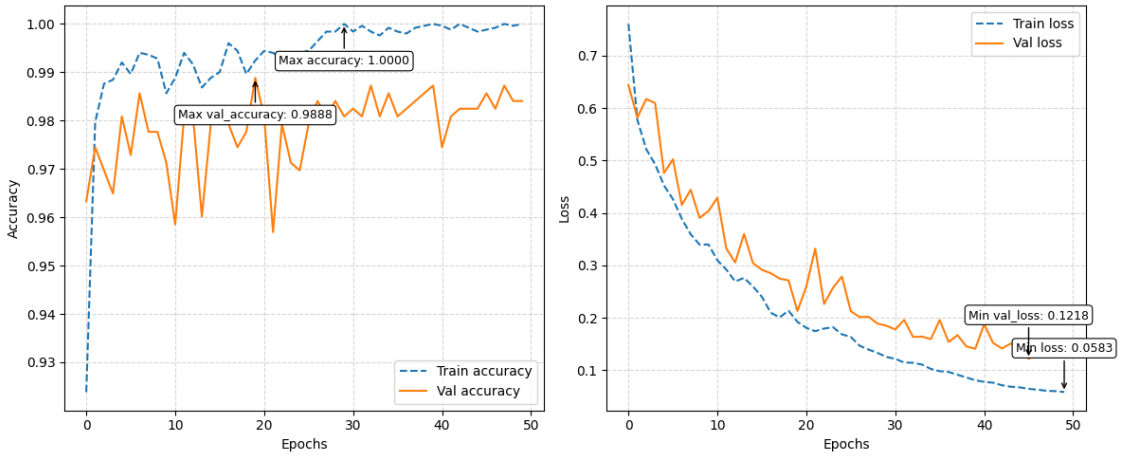
```
accuracy: 0.9989 - loss: 0.1224 - precision_14: 0.9989 - recall_14: 0.9989 -
val_accuracy: 0.9825 - val_loss: 0.1777 - val_precision_14: 0.9825 -
val_recall_14: 0.9825 - learning_rate: 3.0000e-04
Epoch 32/50
79/79            51s 651ms/step -
accuracy: 0.9988 - loss: 0.1168 - precision_14: 0.9988 - recall_14: 0.9988 -
val_accuracy: 0.9809 - val_loss: 0.1959 - val_precision_14: 0.9809 -
val_recall_14: 0.9809 - learning_rate: 3.0000e-04
Epoch 33/50
79/79            51s 643ms/step -
accuracy: 0.9987 - loss: 0.1135 - precision_14: 0.9987 - recall_14: 0.9987 -
val_accuracy: 0.9872 - val_loss: 0.1634 - val_precision_14: 0.9872 -
val_recall_14: 0.9872 - learning_rate: 3.0000e-04
Epoch 34/50
79/79            51s 639ms/step -
accuracy: 0.9975 - loss: 0.1110 - precision_14: 0.9975 - recall_14: 0.9975 -
val_accuracy: 0.9809 - val_loss: 0.1641 - val_precision_14: 0.9809 -
val_recall_14: 0.9809 - learning_rate: 3.0000e-04
Epoch 35/50
79/79            51s 642ms/step -
accuracy: 0.9998 - loss: 0.1040 - precision_14: 0.9998 - recall_14: 0.9998 -
val_accuracy: 0.9856 - val_loss: 0.1592 - val_precision_14: 0.9856 -
val_recall_14: 0.9856 - learning_rate: 3.0000e-04
Epoch 36/50
79/79            50s 637ms/step -
accuracy: 0.9986 - loss: 0.0990 - precision_14: 0.9986 - recall_14: 0.9986 -
val_accuracy: 0.9809 - val_loss: 0.1957 - val_precision_14: 0.9809 -
val_recall_14: 0.9809 - learning_rate: 3.0000e-04
Epoch 37/50
79/79            51s 643ms/step -
accuracy: 0.9978 - loss: 0.0990 - precision_14: 0.9978 - recall_14: 0.9976 -
val_accuracy: 0.9825 - val_loss: 0.1539 - val_precision_14: 0.9824 -
val_recall_14: 0.9809 - learning_rate: 3.0000e-04
Epoch 38/50
79/79            51s 639ms/step -
accuracy: 0.9991 - loss: 0.0921 - precision_14: 0.9991 - recall_14: 0.9991 -
val_accuracy: 0.9841 - val_loss: 0.1669 - val_precision_14: 0.9841 -
val_recall_14: 0.9841 - learning_rate: 3.0000e-04
Epoch 39/50
79/79            51s 642ms/step -
accuracy: 0.9997 - loss: 0.0866 - precision_14: 0.9997 - recall_14: 0.9997 -
val_accuracy: 0.9856 - val_loss: 0.1452 - val_precision_14: 0.9856 -
val_recall_14: 0.9856 - learning_rate: 3.0000e-04
Epoch 40/50
79/79            51s 642ms/step -
accuracy: 1.0000 - loss: 0.0819 - precision_14: 1.0000 - recall_14: 1.0000 -
val_accuracy: 0.9872 - val_loss: 0.1410 - val_precision_14: 0.9872 -
val_recall_14: 0.9872 - learning_rate: 3.0000e-04
```

```
Epoch 41/50
79/79            51s 648ms/step -
accuracy: 0.9999 - loss: 0.0780 - precision_14: 0.9999 - recall_14: 0.9999 -
val_accuracy: 0.9745 - val_loss: 0.1876 - val_precision_14: 0.9745 -
val_recall_14: 0.9745 - learning_rate: 3.0000e-04
Epoch 42/50
79/79            51s 647ms/step -
accuracy: 0.9990 - loss: 0.0779 - precision_14: 0.9991 - recall_14: 0.9990 -
val_accuracy: 0.9809 - val_loss: 0.1521 - val_precision_14: 0.9809 -
val_recall_14: 0.9809 - learning_rate: 3.0000e-04
Epoch 43/50
79/79            51s 640ms/step -
accuracy: 1.0000 - loss: 0.0724 - precision_14: 1.0000 - recall_14: 1.0000 -
val_accuracy: 0.9825 - val_loss: 0.1414 - val_precision_14: 0.9825 -
val_recall_14: 0.9825 - learning_rate: 3.0000e-04
Epoch 44/50
79/79            51s 639ms/step -
accuracy: 0.9993 - loss: 0.0688 - precision_14: 0.9993 - recall_14: 0.9993 -
val_accuracy: 0.9825 - val_loss: 0.1509 - val_precision_14: 0.9825 -
val_recall_14: 0.9825 - learning_rate: 3.0000e-04
Epoch 45/50
79/79            51s 638ms/step -
accuracy: 0.9989 - loss: 0.0663 - precision_14: 0.9989 - recall_14: 0.9989 -
val_accuracy: 0.9825 - val_loss: 0.1417 - val_precision_14: 0.9825 -
val_recall_14: 0.9825 - learning_rate: 3.0000e-04
Epoch 46/50
79/79            51s 643ms/step -
accuracy: 0.9986 - loss: 0.0646 - precision_14: 0.9986 - recall_14: 0.9986 -
val_accuracy: 0.9856 - val_loss: 0.1218 - val_precision_14: 0.9856 -
val_recall_14: 0.9856 - learning_rate: 9.0000e-05
Epoch 47/50
79/79            51s 639ms/step -
accuracy: 0.9995 - loss: 0.0631 - precision_14: 0.9995 - recall_14: 0.9995 -
val_accuracy: 0.9825 - val_loss: 0.1301 - val_precision_14: 0.9825 -
val_recall_14: 0.9825 - learning_rate: 9.0000e-05
Epoch 48/50
79/79            51s 646ms/step -
accuracy: 1.0000 - loss: 0.0611 - precision_14: 1.0000 - recall_14: 1.0000 -
val_accuracy: 0.9872 - val_loss: 0.1284 - val_precision_14: 0.9872 -
val_recall_14: 0.9872 - learning_rate: 9.0000e-05
Epoch 49/50
79/79            52s 658ms/step -
accuracy: 0.9996 - loss: 0.0604 - precision_14: 1.0000 - recall_14: 0.9996 -
val_accuracy: 0.9841 - val_loss: 0.1274 - val_precision_14: 0.9841 -
val_recall_14: 0.9841 - learning_rate: 9.0000e-05
Epoch 50/50
79/79            51s 641ms/step -
accuracy: 1.0000 - loss: 0.0584 - precision_14: 1.0000 - recall_14: 1.0000 -
```

```
val_accuracy: 0.9841 - val_loss: 0.1238 - val_precision_14: 0.9841 -
val_recall_14: 0.9841 - learning_rate: 9.0000e-05
```

6.7.3 Plot loss & accuracy for training and validation wrt epoch

[170]:
```
plot_loss_and_accuracy(["accuracy","loss"],model_fit)
```



6.7.4 Tensorboard

[171]:
```
%load_ext tensorboard
```

```
The tensorboard extension is already loaded. To reload it, use:
  %reload_ext tensorboard
```

6.7.5 Model Evaluation, Mlflow Logging and Printing Metrics

[172]:
```
ckpt_path
```

[172]:
```
'checkpoints/1_pretrained_mobilenetv2_trainable0.weights.h5'
```

[173]:
```
log_dir
```

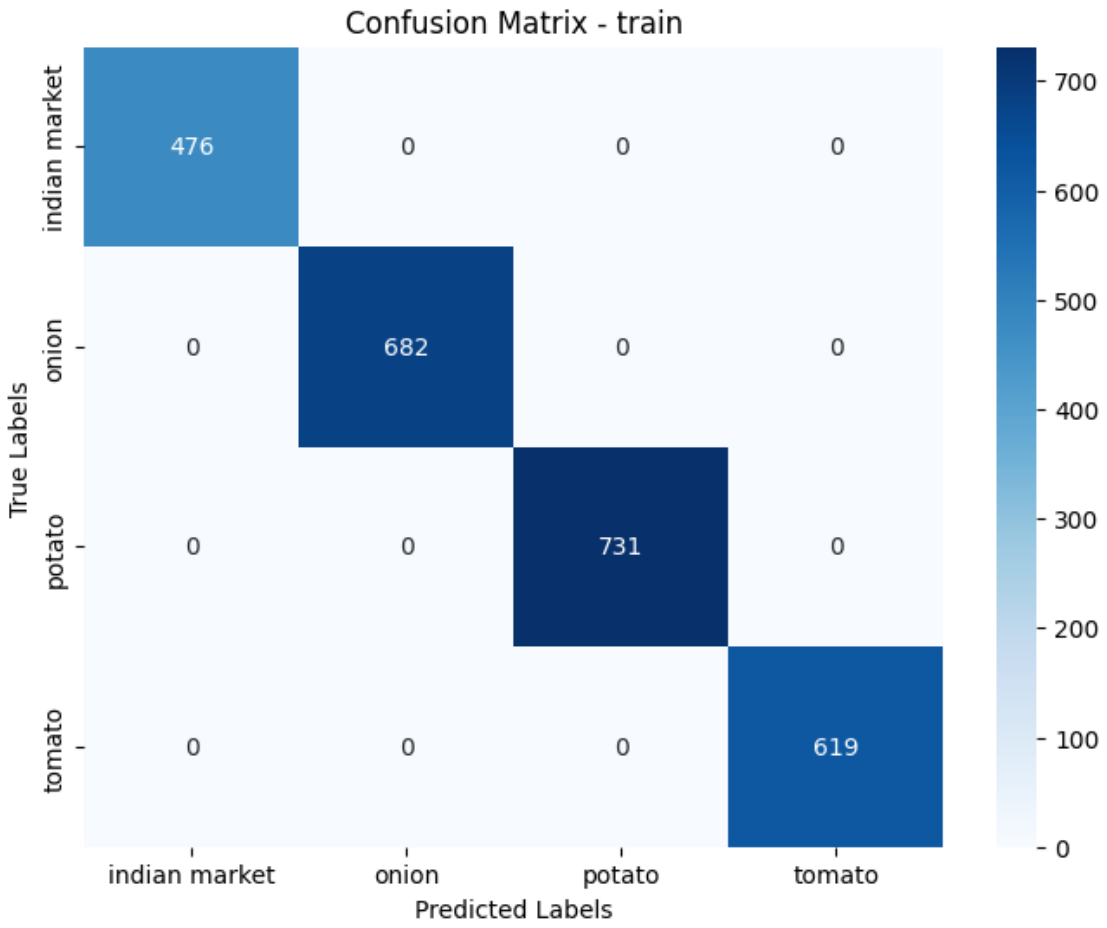
[173]:
```
'logs/1_pretrained_mobilenetv2_trainable0'
```

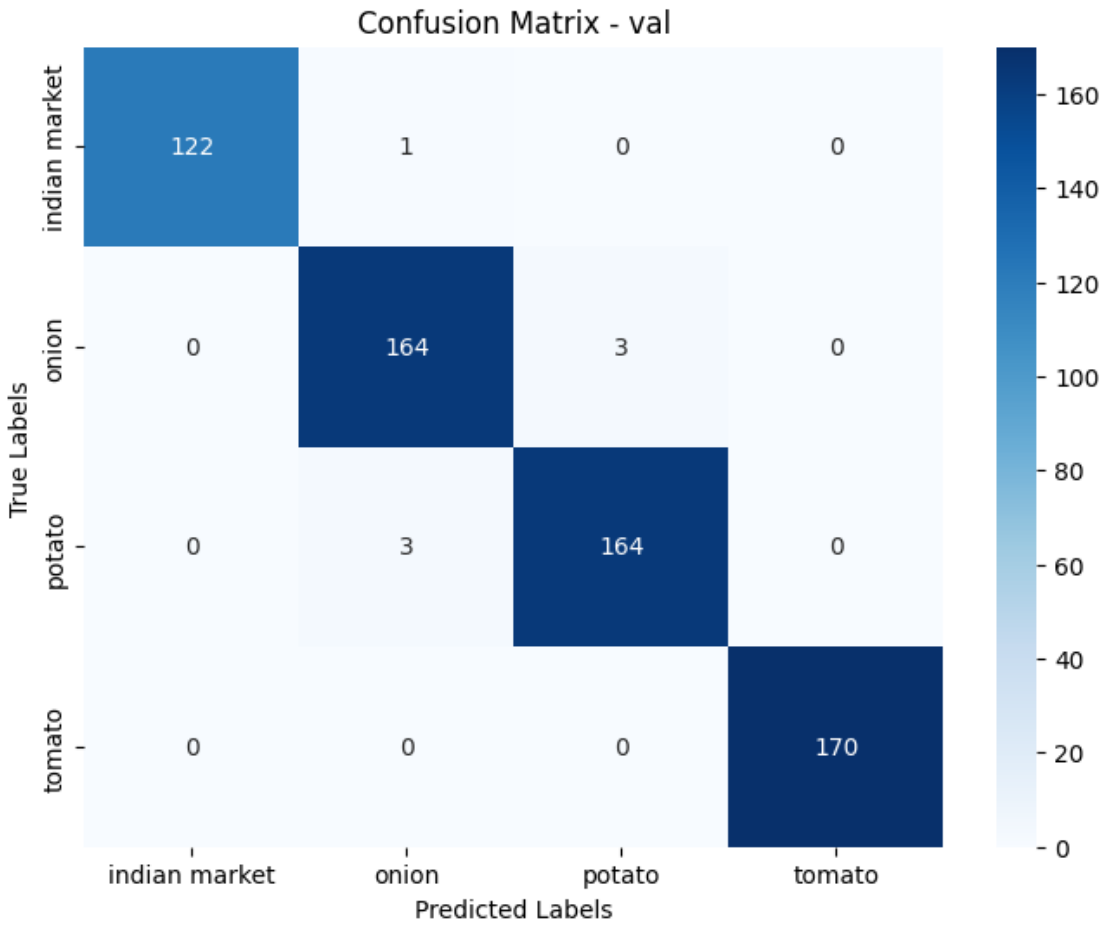
[174]:
```
run_name = "1_pretrained_mobilenetv2_trainable0"
run_name
```

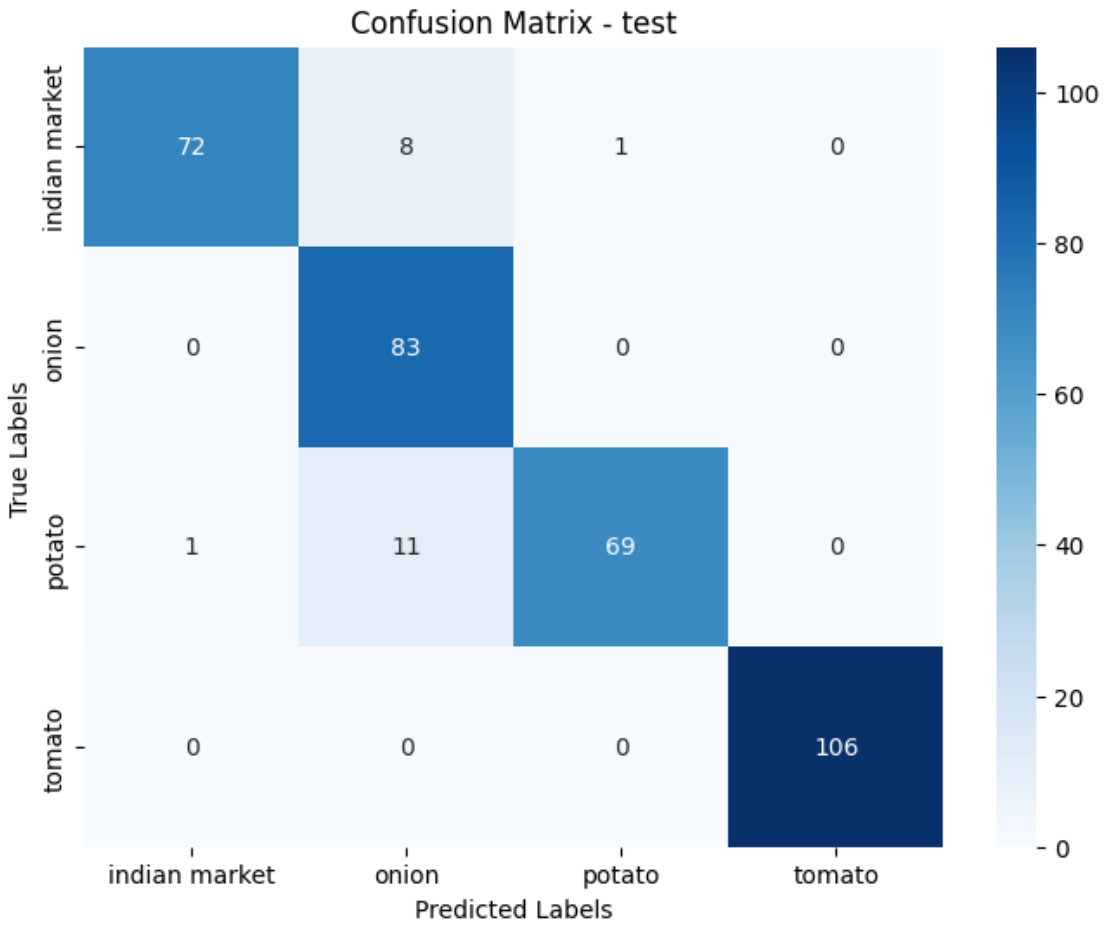
[174]:
```
'1_pretrained_mobilenetv2_trainable0'
```

[175]:
```
evaluate_model(model, train_ds, val_ds, test_ds,␣
↪class_names,run_name=run_name,ckpt_path= ckpt_path)
```

```
2025/02/11 03:19:39 WARNING mlflow.tensorflow: You are saving a TensorFlow Core
model or Keras model without a signature. Inference with
mlflow.pyfunc.spark_udf() will not work unless the model's pyfunc representation
accepts pandas DataFrames as inference inputs.
2025/02/11 03:19:50 WARNING mlflow.models.model: Model logged without a
signature and input example. Please set `input_example` parameter when logging
the model to auto infer the model signature.


train Metrics:
 Accuracy: 100.00%
 Precision: 100.00%
 Recall: 100.00%

val Metrics:
 Accuracy: 98.88%
 Precision: 98.96%
 Recall: 98.90%

test Metrics:
 Accuracy: 94.02%
 Precision: 94.64%
 Recall: 93.52%
```


Confusion Matrix - train
True Labels
Predicted Labels
indian market
onion
potato
tomato
476
0
0
0
0
682
0
0
0
0
731
0
0
0
0
619
700
600
500
400
300
200
100
0


Confusion Matrix - val
indian market
onion
potato
tomato
True Labels
122
1
0
0
0
164
3
0
0
3
164
0
0
0
0
170
indian market
onion
potato
tomato
Predicted Labels
160
140
120
100
80
60
40
20
0




```
Classification Report - train
              precision    recall  f1-score   support

indian market       1.00      1.00      1.00       476
        onion       1.00      1.00      1.00       682
       potato       1.00      1.00      1.00       731
       tomato       1.00      1.00      1.00       619

     accuracy                           1.00      2508
    macro avg       1.00      1.00      1.00      2508
 weighted avg       1.00      1.00      1.00      2508


Classification Report - val
              precision    recall  f1-score   support

indian market       1.00      0.99      1.00       123
```

```
        onion       0.98      0.98      0.98       167
       potato       0.98      0.98      0.98       167
       tomato       1.00      1.00      1.00       170

     accuracy                           0.99       627
    macro avg       0.99      0.99      0.99       627
 weighted avg       0.99      0.99      0.99       627


Classification Report - test
               precision    recall  f1-score   support

indian market       0.99      0.89      0.94        81
        onion       0.81      1.00      0.90        83
       potato       0.99      0.85      0.91        81
       tomato       1.00      1.00      1.00       106

     accuracy                           0.94       351
    macro avg       0.95      0.94      0.94       351
 weighted avg       0.95      0.94      0.94       351


Class-wise Accuracy:

Train Class-wise Accuracy:
 indian market: 100.00% (476/476)
 onion: 100.00% (682/682)
 potato: 100.00% (731/731)
 tomato: 100.00% (619/619)

Val Class-wise Accuracy:
 indian market: 99.19% (122/123)
 onion: 98.20% (164/167)
 potato: 98.20% (164/167)
 tomato: 100.00% (170/170)

Test Class-wise Accuracy:
 indian market: 88.89% (72/81)
 onion: 100.00% (83/83)
 potato: 85.19% (69/81)
 tomato: 100.00% (106/106)

Random Test Predictions:
```

True: indian market
Pred: indian market

True: indian market
Pred: indian market

True: onion
Pred: onion

True: onion
Pred: onion

True: potato
Pred: potato

True: potato
Pred: onion

True: tomato
Pred: tomato

True: tomato
Pred: tomato

6.7.6 Save the model

[176]:
```
model.save("saved_models/1_pretrained_mobilenetv2_trainable0_model.keras")
```

6.8 DATA AUGMENTED MOBILENET_V2

6.8.1 Load the previously trained model architecture and weights

[177]:
```
# Load the entire model (architecture + weights)
model = keras.models.load_model("saved_models/
  ↪1_pretrained_mobilenetv2_trainable0_model.keras")
```

[178]:
```
model.summary()
```

Model: "pretrained_mobilenetv2"

Layer (type)	Output Shape	Param #
mobilenetv2_1.00_224 (Functional)	(None, 8, 8, 1280)	2,257,984
global_average_pooling2d_13 (GlobalAveragePooling2D)	(None, 1280)	0
dense_39 (Dense)	(None, 512)	655,872
batch_normalization_112 (BatchNormalization)	(None, 512)	2,048
dropout_15 (Dropout)	(None, 512)	0
dense_40 (Dense)	(None, 256)	131,328
batch_normalization_113 (BatchNormalization)	(None, 256)	1,024
dense_41 (Dense)	(None, 4)	1,028

Total params: 4,628,814 (17.66 MB)

Trainable params: 789,764 (3.01 MB)

Non-trainable params: 2,259,520 (8.62 MB)

```
 Optimizer params: 1,579,530 (6.03 MB)
```

6.8.2 Model Compilation and Training

[179]:
```
ckpt_path = "checkpoints/1_pretrained_mobilenetv2_aug_trainable0.weights.h5"
```

[180]:
```
log_dir = 'logs/1_pretrained_mobilenetv2_aug_trainable0'
```

[181]:
```
run_name = '1_pretrained_mobilenetv2_aug_trainable0'
```

[182]:
```
model_fit = compile_train(model, train_aug_ds, val_aug_ds,␣
 ↪log_dir=log_dir,epochs=20,ckpt_path=ckpt_path)
```

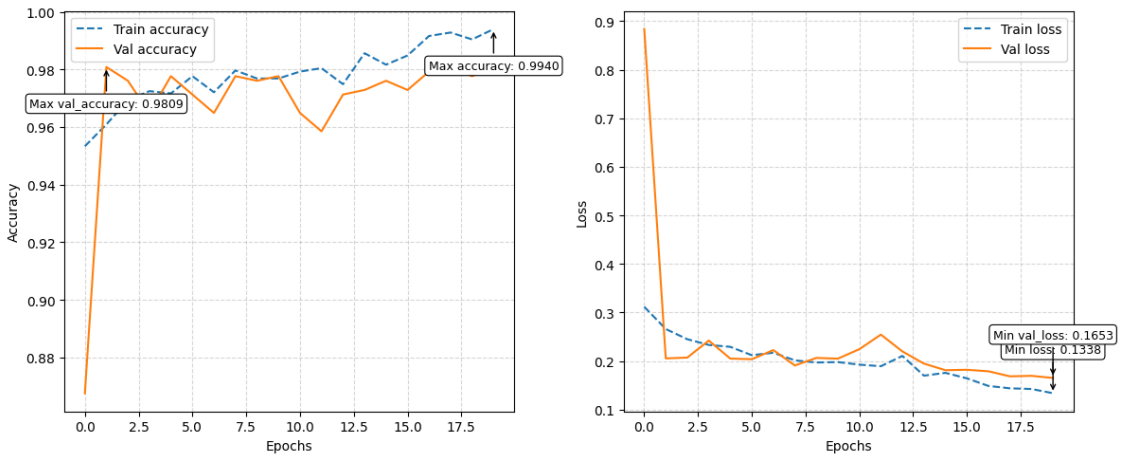
```
Epoch 1/20
79/79             59s 680ms/step -
accuracy: 0.9526 - loss: 0.3458 - precision_15: 0.9547 - recall_15: 0.9526 -
val_accuracy: 0.8676 - val_loss: 0.8836 - val_precision_15: 0.8676 -
val_recall_15: 0.8676 - learning_rate: 0.0010
Epoch 2/20
79/79             54s 680ms/step -
accuracy: 0.9604 - loss: 0.2680 - precision_15: 0.9631 - recall_15: 0.9577 -
val_accuracy: 0.9809 - val_loss: 0.2056 - val_precision_15: 0.9824 -
val_recall_15: 0.9809 - learning_rate: 0.0010
Epoch 3/20
79/79             53s 666ms/step -
accuracy: 0.9720 - loss: 0.2435 - precision_15: 0.9728 - recall_15: 0.9696 -
val_accuracy: 0.9761 - val_loss: 0.2073 - val_precision_15: 0.9760 -
val_recall_15: 0.9745 - learning_rate: 0.0010
Epoch 4/20
79/79             53s 665ms/step -
accuracy: 0.9750 - loss: 0.2250 - precision_15: 0.9750 - recall_15: 0.9748 -
val_accuracy: 0.9649 - val_loss: 0.2423 - val_precision_15: 0.9664 -
val_recall_15: 0.9633 - learning_rate: 0.0010
Epoch 5/20
79/79             53s 667ms/step -
accuracy: 0.9698 - loss: 0.2389 - precision_15: 0.9710 - recall_15: 0.9689 -
val_accuracy: 0.9777 - val_loss: 0.2052 - val_precision_15: 0.9792 -
val_recall_15: 0.9777 - learning_rate: 0.0010
Epoch 6/20
79/79             53s 659ms/step -
accuracy: 0.9786 - loss: 0.2179 - precision_15: 0.9785 - recall_15: 0.9776 -
val_accuracy: 0.9713 - val_loss: 0.2039 - val_precision_15: 0.9712 -
val_recall_15: 0.9697 - learning_rate: 0.0010
Epoch 7/20
79/79             53s 660ms/step -
accuracy: 0.9744 - loss: 0.2138 - precision_15: 0.9749 - recall_15: 0.9743 -
val_accuracy: 0.9649 - val_loss: 0.2226 - val_precision_15: 0.9665 -
```

```
val_recall_15: 0.9649 - learning_rate: 0.0010
Epoch 8/20
79/79            53s 668ms/step -
accuracy: 0.9752 - loss: 0.2198 - precision_15: 0.9773 - recall_15: 0.9752 -
val_accuracy: 0.9777 - val_loss: 0.1910 - val_precision_15: 0.9792 -
val_recall_15: 0.9777 - learning_rate: 0.0010
Epoch 9/20
79/79            53s 659ms/step -
accuracy: 0.9756 - loss: 0.1932 - precision_15: 0.9780 - recall_15: 0.9754 -
val_accuracy: 0.9761 - val_loss: 0.2066 - val_precision_15: 0.9761 -
val_recall_15: 0.9761 - learning_rate: 0.0010
Epoch 10/20
79/79            53s 661ms/step -
accuracy: 0.9800 - loss: 0.1960 - precision_15: 0.9807 - recall_15: 0.9792 -
val_accuracy: 0.9777 - val_loss: 0.2050 - val_precision_15: 0.9777 -
val_recall_15: 0.9777 - learning_rate: 0.0010
Epoch 11/20
79/79            53s 665ms/step -
accuracy: 0.9805 - loss: 0.1929 - precision_15: 0.9810 - recall_15: 0.9805 -
val_accuracy: 0.9649 - val_loss: 0.2246 - val_precision_15: 0.9665 -
val_recall_15: 0.9649 - learning_rate: 0.0010
Epoch 12/20
79/79            53s 658ms/step -
accuracy: 0.9849 - loss: 0.1721 - precision_15: 0.9861 - recall_15: 0.9837 -
val_accuracy: 0.9585 - val_loss: 0.2545 - val_precision_15: 0.9600 -
val_recall_15: 0.9569 - learning_rate: 0.0010
Epoch 13/20
79/79            53s 659ms/step -
accuracy: 0.9700 - loss: 0.2231 - precision_15: 0.9708 - recall_15: 0.9674 -
val_accuracy: 0.9713 - val_loss: 0.2199 - val_precision_15: 0.9713 -
val_recall_15: 0.9713 - learning_rate: 0.0010
Epoch 14/20
79/79            53s 659ms/step -
accuracy: 0.9876 - loss: 0.1686 - precision_15: 0.9880 - recall_15: 0.9873 -
val_accuracy: 0.9729 - val_loss: 0.1949 - val_precision_15: 0.9729 -
val_recall_15: 0.9729 - learning_rate: 3.0000e-04
Epoch 15/20
79/79            54s 673ms/step -
accuracy: 0.9792 - loss: 0.1803 - precision_15: 0.9812 - recall_15: 0.9788 -
val_accuracy: 0.9761 - val_loss: 0.1813 - val_precision_15: 0.9792 -
val_recall_15: 0.9761 - learning_rate: 3.0000e-04
Epoch 16/20
79/79            53s 666ms/step -
accuracy: 0.9835 - loss: 0.1714 - precision_15: 0.9837 - recall_15: 0.9835 -
val_accuracy: 0.9729 - val_loss: 0.1821 - val_precision_15: 0.9729 -
val_recall_15: 0.9729 - learning_rate: 3.0000e-04
Epoch 17/20
79/79            53s 660ms/step -
```

```
accuracy: 0.9891 - loss: 0.1504 - precision_15: 0.9893 - recall_15: 0.9888 -
val_accuracy: 0.9793 - val_loss: 0.1790 - val_precision_15: 0.9793 -
val_recall_15: 0.9793 - learning_rate: 3.0000e-04
Epoch 18/20
79/79              53s 660ms/step -
accuracy: 0.9923 - loss: 0.1467 - precision_15: 0.9923 - recall_15: 0.9923 -
val_accuracy: 0.9809 - val_loss: 0.1686 - val_precision_15: 0.9809 -
val_recall_15: 0.9809 - learning_rate: 3.0000e-04
Epoch 19/20
79/79              53s 658ms/step -
accuracy: 0.9881 - loss: 0.1483 - precision_15: 0.9882 - recall_15: 0.9881 -
val_accuracy: 0.9777 - val_loss: 0.1697 - val_precision_15: 0.9792 -
val_recall_15: 0.9777 - learning_rate: 3.0000e-04
Epoch 20/20
79/79              53s 660ms/step -
accuracy: 0.9945 - loss: 0.1344 - precision_15: 0.9944 - recall_15: 0.9942 -
val_accuracy: 0.9809 - val_loss: 0.1653 - val_precision_15: 0.9808 -
val_recall_15: 0.9793 - learning_rate: 3.0000e-04
```

6.8.3 Plot loss & accuracy for training and validation wrt epoch

[183]:
```
plot_loss_and_accuracy(["accuracy","loss"],model_fit)
```



6.8.4 Tensorboard

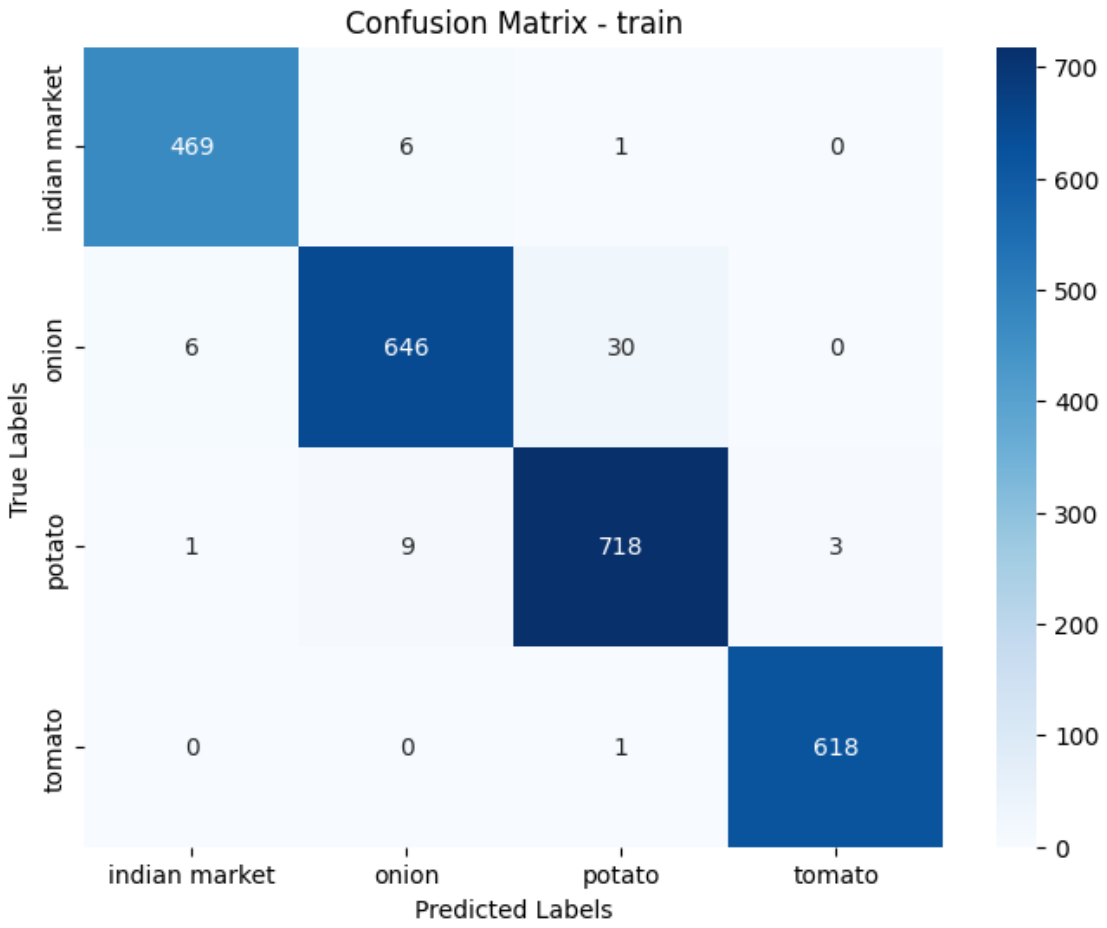
[184]:
```
%load_ext tensorboard
```

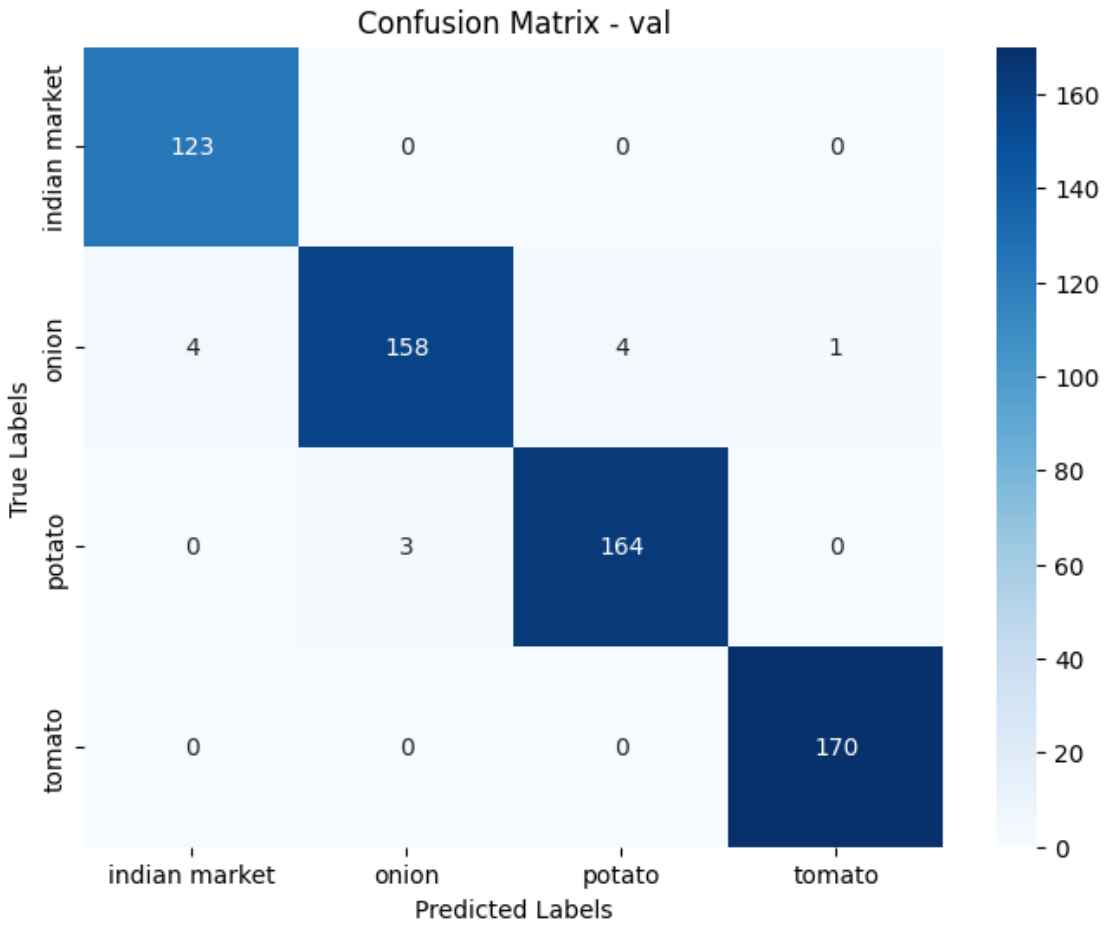
```
The tensorboard extension is already loaded. To reload it, use:
  %reload_ext tensorboard
```

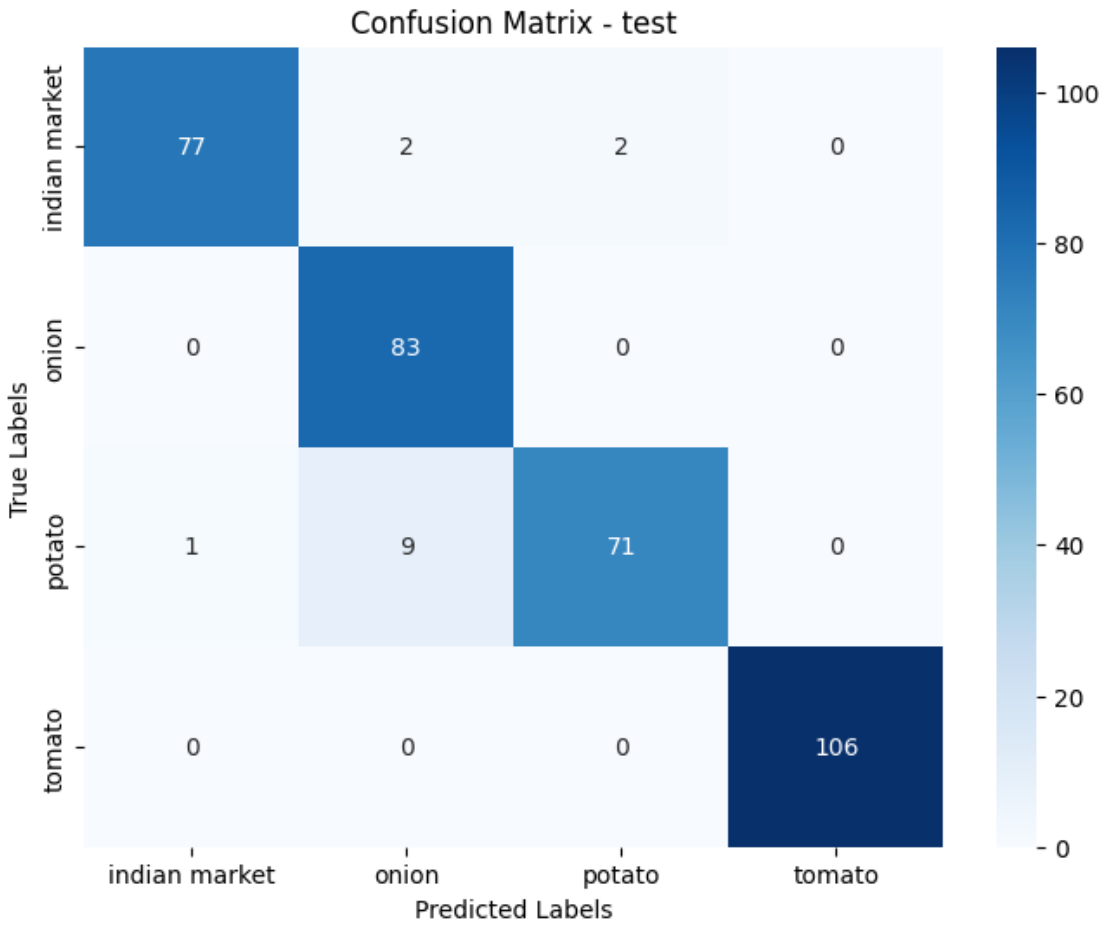
6.8.5 Model Evaluation, Mlflow Logging and Printing Metrics

[185]:
```
evaluate_model(model, train_aug_ds, val_aug_ds, test_aug_ds,␣
↪class_names,run_name=run_name,ckpt_path= ckpt_path)
```

```
2025/02/11 03:39:03 WARNING mlflow.tensorflow: You are saving a TensorFlow Core
model or Keras model without a signature. Inference with
mlflow.pyfunc.spark_udf() will not work unless the model's pyfunc representation
accepts pandas DataFrames as inference inputs.
2025/02/11 03:39:14 WARNING mlflow.models.model: Model logged without a
signature and input example. Please set `input_example` parameter when logging
the model to auto infer the model signature.


train Metrics:
 Accuracy: 97.73%
 Precision: 97.88%
 Recall: 97.83%

val Metrics:
 Accuracy: 98.09%
 Precision: 98.01%
 Recall: 98.20%

test Metrics:
 Accuracy: 96.01%
 Precision: 96.07%
 Recall: 95.68%
```


Confusion Matrix - train
True Labels
Predicted Labels
indian market
onion
potato
tomato
469
6
1
0
6
646
30
0
1
9
718
3
0
0
1
618
700
600
500
400
300
200
100
0


Confusion Matrix - val
indian market
onion
potato
tomato
True Labels
123
0
0
0
4
158
4
1
0
3
164
0
0
0
0
170
indian market
onion
potato
tomato
Predicted Labels
160
140
120
100
80
60
40
20
0



Classification Report - train

	precision	recall	f1-score	support
indian market	0.99	0.99	0.99	476
onion	0.98	0.95	0.96	682
potato	0.96	0.98	0.97	731
tomato	1.00	1.00	1.00	619
accuracy			0.98	2508
macro avg	0.98	0.98	0.98	2508
weighted avg	0.98	0.98	0.98	2508

Classification Report - val

	precision	recall	f1-score	support
indian market	0.97	1.00	0.98	123

```
        onion       0.98      0.95      0.96       167
       potato       0.98      0.98      0.98       167
       tomato       0.99      1.00      1.00       170

     accuracy                           0.98       627
    macro avg       0.98      0.98      0.98       627
 weighted avg       0.98      0.98      0.98       627


Classification Report - test
               precision    recall  f1-score   support

indian market       0.99      0.95      0.97        81
        onion       0.88      1.00      0.94        83
       potato       0.97      0.88      0.92        81
       tomato       1.00      1.00      1.00       106

     accuracy                           0.96       351
    macro avg       0.96      0.96      0.96       351
 weighted avg       0.96      0.96      0.96       351


Class-wise Accuracy:

Train Class-wise Accuracy:
 indian market: 98.53% (469/476)
 onion: 94.72% (646/682)
 potato: 98.22% (718/731)
 tomato: 99.84% (618/619)

Val Class-wise Accuracy:
 indian market: 100.00% (123/123)
 onion: 94.61% (158/167)
 potato: 98.20% (164/167)
 tomato: 100.00% (170/170)

Test Class-wise Accuracy:
 indian market: 95.06% (77/81)
 onion: 100.00% (83/83)
 potato: 87.65% (71/81)
 tomato: 100.00% (106/106)

Random Test Predictions:
```

True: indian market
Pred: indian market

True: indian market
Pred: indian market

True: onion
Pred: onion

True: onion
Pred: onion

True: potato
Pred: potato

True: potato
Pred: potato

True: tomato
Pred: tomato

True: tomato
Pred: tomato

6.8.6 Save the model

[186]:
```
model.save("saved_models/1_pretrained_mobilenetv2_aug_trainable0_model.keras")
```

6.8.7 Observations

- **Architecture**: Lightweight Mobilenet V2 CNN using depthwise separable convolutions pretrained on ImageNet along with two dense layers with batch norm and one dropout.
- **Performance Before augmentation**:
  - **Train Accuracy**: **100%**
  - **Validation Accuracy**: **98.88%**
  - **Test Accuracy**: **94.02%** (Best model of project and light weight too).
  - **Test Precision/Recall**: ~95% macro-average.
  - **Epochs**: **50**
  - **Training time**: **42 min (approx)**
  - **Observed Issues**:
    - Dataset's size and simplicity likely make MobileNet a better fit than bulkier models. Its efficiency and compatibility with augmentation give it an edge, while VGG16/ResNet overcomplicate the problem.
    - Observed Confusion between indian market and potato.
- **Strengths**:
  - Designed for mobile/embedded devices (e.g., real-time sorting systems).
  - Best model to deploy among custom and pretrained models.
- **Impact of Augmentation**:
  - **Train Accuracy**: **97.73%** (Robustness increased with augmentation)
  - **Validation Accuracy**: **98.09%**
  - **Test Accuracy**: **96.01%** (Best light weight pretrained model accuracy).
  - **Test Precision/Recall**: ~96% macro-average.
  - **Epochs**: **20** + load weights from previous Mobilenet V2
  - **Training time**: **19 min (approx)** (Can increase the epochs to improve more)

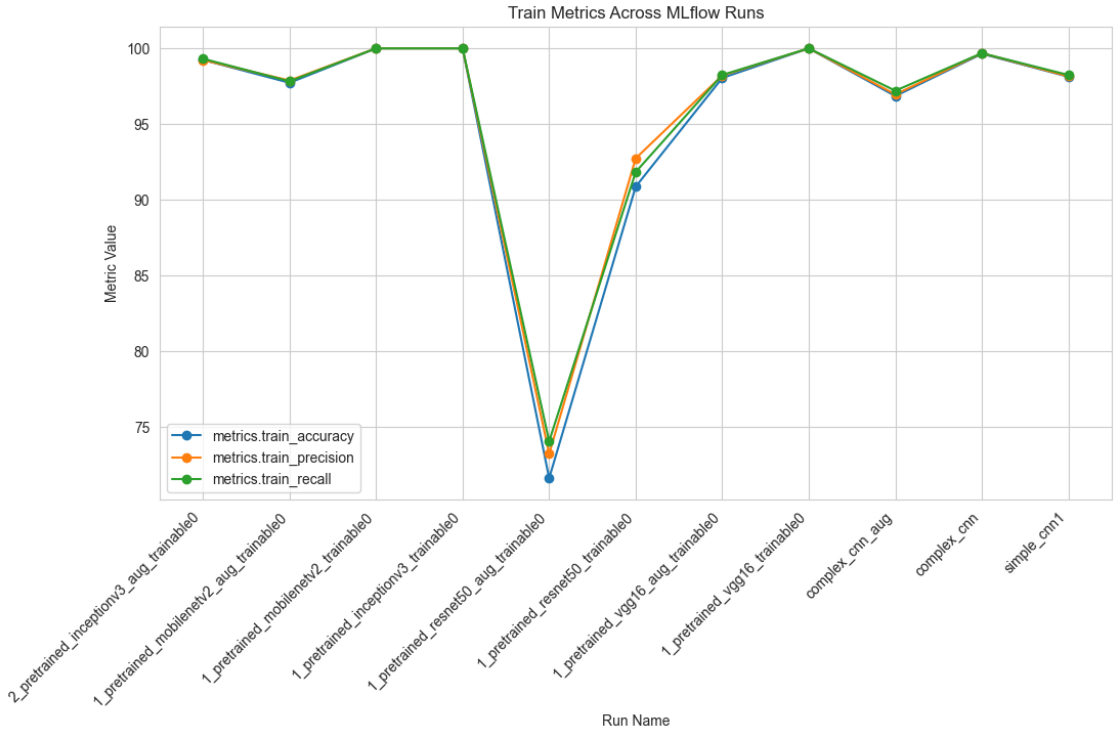
7 MODEL COMPARISONS

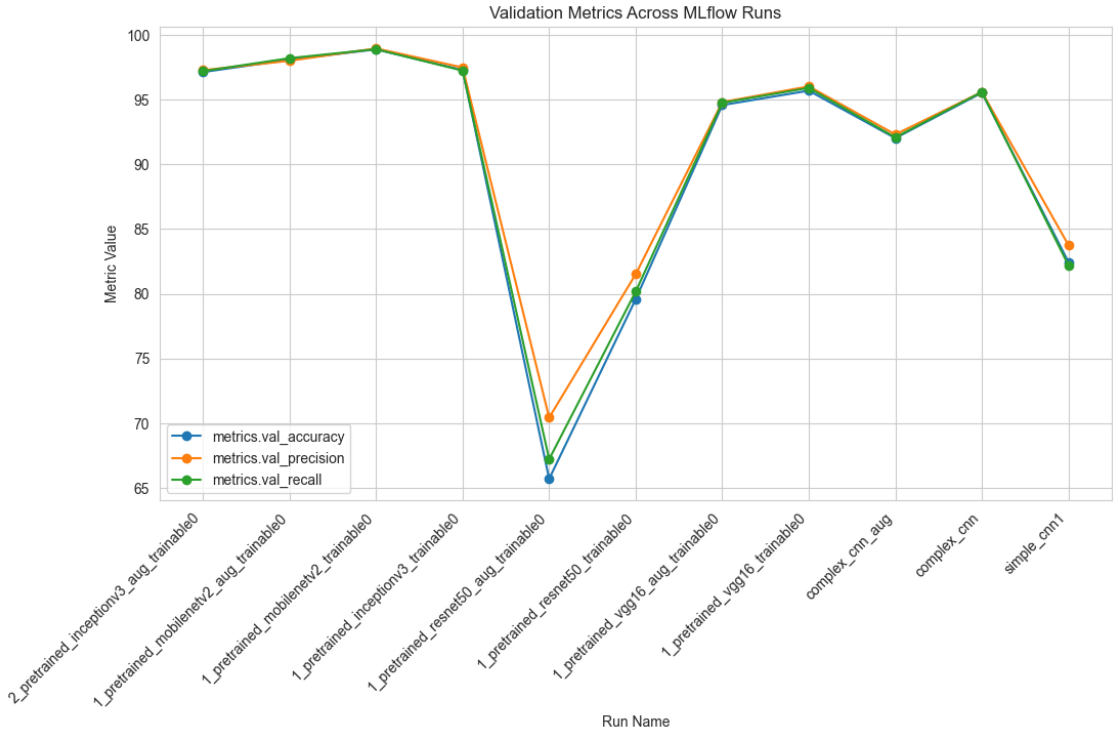
[198]:
```
# Define experiment name
experiment_name = "Vegetable_image_classification"
experiment = mlflow.get_experiment_by_name(experiment_name)

# Get all runs
runs_df = mlflow.search_runs(experiment_ids=[experiment.experiment_id])

# Print all available columns to check metric names
print("Available Columns:", runs_df.columns.tolist())

# Identify metric categories
train_metrics = [col for col in runs_df.columns if "train" in col.lower()]
```

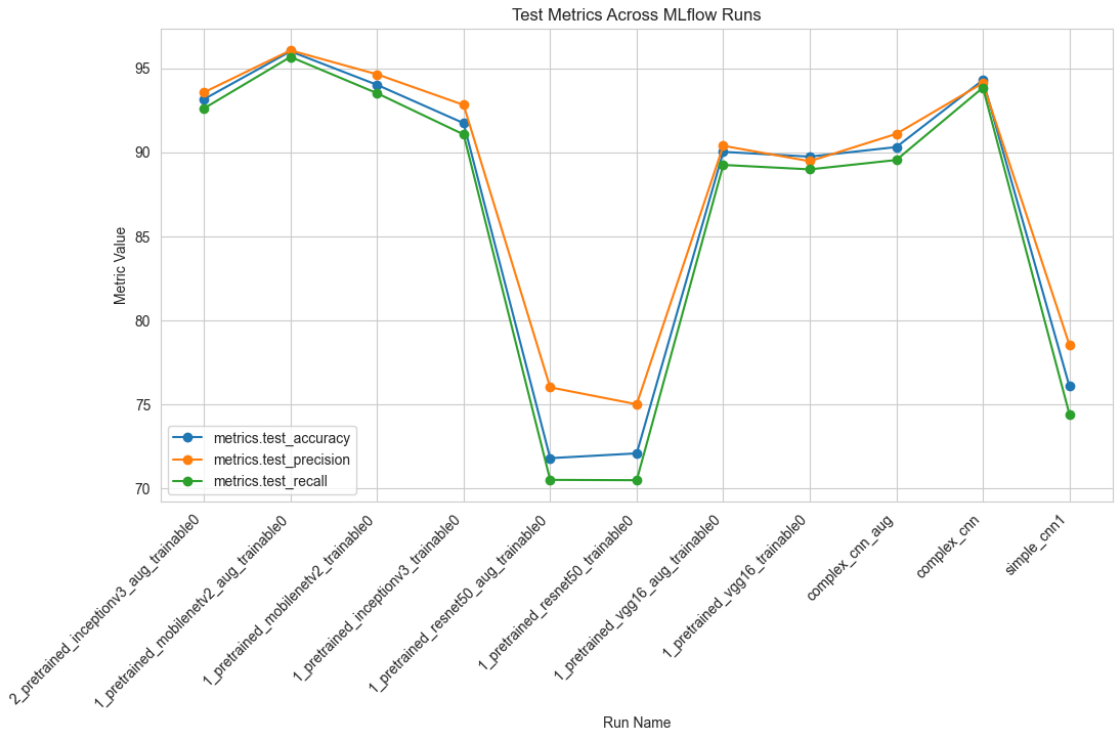
```
val_metrics = [col for col in runs_df.columns if "val" in col.lower()]
test_metrics = [col for col in runs_df.columns if "test" in col.lower()]

# Print detected metrics
print("Train Metrics:", train_metrics)
print("Validation Metrics:", val_metrics)
print("Test Metrics:", test_metrics)

# Check if "run_name" exists, else use "run_id"
run_identifier = "tags.mlflow.runName" if "tags.mlflow.runName" in runs_df.
 ↪columns else "run_id"

# Extract relevant metrics with proper indexing to avoid SettingWithCopyWarning
train_df = runs_df[[run_identifier] + train_metrics].copy()
val_df = runs_df[[run_identifier] + val_metrics].copy()
test_df = runs_df[[run_identifier] + test_metrics].copy()

# Rename run column for clarity
train_df.rename(columns={run_identifier: "Run"}, inplace=True)
val_df.rename(columns={run_identifier: "Run"}, inplace=True)
test_df.rename(columns={run_identifier: "Run"}, inplace=True)

# Convert to numeric (MLflow sometimes stores metrics as strings)
for df, metric_list in zip([train_df, val_df, test_df], [train_metrics,␣
 ↪val_metrics, test_metrics]):
    df.loc[:, metric_list] = df.loc[:, metric_list].apply(pd.to_numeric,␣
 ↪errors='coerce')

# Function to plot metrics
def plot_metrics(df, metric_list, title):
    sns.set_style("whitegrid")
    plt.figure(figsize=(12, 6))

    for metric in metric_list:
        plt.plot(df["Run"], df[metric], marker='o', label=metric)

    plt.xticks(rotation=45, ha="right")
    plt.xlabel("Run Name")
    plt.ylabel("Metric Value")
    plt.title(title)
    plt.legend()
    plt.show()

# Plot Train, Validation, and Test metrics separately
plot_metrics(train_df, ["metrics.train_accuracy", "metrics.train_precision",␣
 ↪"metrics.train_recall"], "Train Metrics Across MLflow Runs")
```

```
plot_metrics(val_df, ["metrics.val_accuracy", "metrics.val_precision", "metrics.
  ↪val_recall"], "Validation Metrics Across MLflow Runs")
plot_metrics(test_df, ["metrics.test_accuracy", "metrics.test_precision",␣
  ↪"metrics.test_recall"], "Test Metrics Across MLflow Runs")
```

```
Available Columns: ['run_id', 'experiment_id', 'status', 'artifact_uri',
'start_time', 'end_time', 'metrics.test_recall', 'metrics.train_recall',
'metrics.train_class_accuracy_indian market',
'metrics.train_class_accuracy_potato', 'metrics.train_class_accuracy_onion',
'metrics.val_accuracy', 'metrics.train_precision', 'metrics.test_precision',
'metrics.test_accuracy', 'metrics.test_class_accuracy_potato',
'metrics.val_class_accuracy_tomato', 'metrics.train_class_accuracy_tomato',
'metrics.test_class_accuracy_onion', 'metrics.test_class_accuracy_tomato',
'metrics.val_recall', 'metrics.val_precision',
'metrics.val_class_accuracy_indian market', 'metrics.test_class_accuracy_indian
market', 'metrics.train_accuracy', 'metrics.val_class_accuracy_potato',
'metrics.val_class_accuracy_onion', 'params.checkpoint_path',
'tags.mlflow.runName', 'tags.mlflow.source.type', 'tags.mlflow.log-
model.history', 'tags.mlflow.source.name', 'tags.mlflow.user']
Train Metrics: ['metrics.train_recall', 'metrics.train_class_accuracy_indian
market', 'metrics.train_class_accuracy_potato',
'metrics.train_class_accuracy_onion', 'metrics.train_precision',
'metrics.train_class_accuracy_tomato', 'metrics.train_accuracy']
Validation Metrics: ['metrics.val_accuracy',
'metrics.val_class_accuracy_tomato', 'metrics.val_recall',
'metrics.val_precision', 'metrics.val_class_accuracy_indian market',
'metrics.val_class_accuracy_potato', 'metrics.val_class_accuracy_onion']
Test Metrics: ['metrics.test_recall', 'metrics.test_precision',
'metrics.test_accuracy', 'metrics.test_class_accuracy_potato',
'metrics.test_class_accuracy_onion', 'metrics.test_class_accuracy_tomato',
'metrics.test_class_accuracy_indian market']
```


Train Metrics Across MLflow Runs
Metric Value
Run Name
metrics.train_accuracy
metrics.train_precision
metrics.train_recall
2_pretrained_inceptionv3_aug_trainable0
1_pretrained_mobilenetv2_aug_trainable0
1_pretrained_mobilenetv2_trainable0
1_pretrained_inceptionv3_trainable0
1_pretrained_resnet50_aug_trainable0
1_pretrained_resnet50_trainable0
1_pretrained_vgg16_aug_trainable0
1_pretrained_vgg16_trainable0
complex_cnn_aug
complex_cnn
simple_cnn1


Validation Metrics Across MLflow Runs
Metric Value
Run Name
metrics.val_accuracy
metrics.val_precision
metrics.val_recall
2_pretrained_inceptionv3_aug_trainable0
1_pretrained_mobilenetv2_aug_trainable0
1_pretrained_mobilenetv2_trainable0
1_pretrained_inceptionv3_trainable0
1_pretrained_resnet50_aug_trainable0
1_pretrained_resnet50_trainable0
1_pretrained_vgg16_aug_trainable0
1_pretrained_vgg16_trainable0
complex_cnn_aug
complex_cnn
simple_cnn1



7.0.1 MODEL COMPARISONS

Model	Test Accuracy	Train Accuracy	Validation Accuracy	Overfitting	Inference Speed	Epochs	Training Time (min)
Simple CNN	76.07%	98.13%	82.46%	Severe	Fast	10	25
Complex CNN	94.30%	99.64%	95.53%	Minimal	Moderate	51	170
Complex CNN + Aug	90.31%	96.85%	92.03%	Low	Moderate	58	160
VGG16	89.70%	100%	95.69%	Moderate	Slow	43	320
VGG16 + Aug	90.03%	98.05%	94.58%	Low	Slow	20	320
ResNet50	72.08%	90.87%	79.59%	High	Moderate	48	135
ResNet50 + Aug	71.79%	71.61%	65.71%	Underfitting	Moderate	20	60
InceptionV3	91.74%	100%	97.29%	Minimal	Slow	50	97
InceptionV3 + Aug	93.16%	99.24%	97.13%	Low	Slow	20	45
MobileNetV2	94.02%	100%	98.88%	Minimal	Fast	50	42
MobileNetV2 + Aug	**96.01%**	97.73%	98.09%	Low	**Fastest**	20	19

7.0.2 Best Performer

- **MobileNetV2 + Augmentation** (96.01% test accuracy, fastest inference).

7.0.3 Trade-offs

- **MobileNetV2** is the best model in terms of **accuracy, inference speed, and efficiency**.
- **Complex CNN** performed well but was heavier than MobileNetV2.
- **Pretrained models (VGG16, ResNet50, InceptionV3)** had good accuracy but were **resource-heavy**.
- **ResNet50** performed the worst due to **overfitting and underfitting**, requiring **architecture tuning**.
- **Data Augmentation** improved generalization for most models but slightly reduced accuracy in some cases.

Let me know if you need additional insights or visualizations!

8 ACTIONABLE INSIGHTS AND RECOMMENDATIONS

1. Augmentation Effectively Improved Generalization
- Augmentation reduced overfitting and increased robustness, leading to improved test accuracy across models.
- Complex CNN's accuracy dropped slightly with augmentation (**94.3% → 90.3%**), but it generalized better.
- MobileNetV2 benefited the most (**94.02% → 96.01%**), making it the best lightweight model for deployment.
- Excessive augmentation can introduce unnecessary variability, requiring fine-tuning to optimize performance.

2. Model Selection Should Prioritize Efficiency & Accuracy
- **MobileNetV2 + Augmentation** provided the **best trade-off** between accuracy (**96.01%**), training time, and inference speed.
- **InceptionV3 + Augmentation** (**93.16% test accuracy**) was the best heavyweight model but resource-intensive.
- **ResNet50 struggled** due to high overfitting and required more training epochs to perform better.
- **VGG16 performed well** (**90.03% test accuracy**) but was slow in inference.

3. Noise Filtering Needs More Attention
- The "Indian market" class (background noise) was frequently misclassified, affecting model performance.
- Models struggled with distinguishing "potato" and "onion" due to similar shapes and backgrounds.
- Attention-based architectures like **Transformers or Self-Attention CNNs** could help in filtering out irrelevant features.

4. Training Efficiency Can Be Optimized Further

- **ReduceLROnPlateau + EarlyStopping** helped control overfitting and stabilized training.
- Some models like **ResNet50** needed more epochs, while **VGG16 could converge faster** with reduced epochs.
- **Batch Normalization + Dropout** effectively controlled overfitting in Complex CNN and pre-trained models.

Final Recommendations

1. **Deploy MobileNetV2 + Augmentation** for real-world applications due to its high accuracy and efficiency.
2. **Fine-tune augmentation strategies** to avoid unnecessary variability while improving generalization.
3. **Improve noise filtering** using self-attention mechanisms or multi-scale feature extraction.
4. **Optimize training** by adjusting epochs for each model and refining learning rate schedules.