

```
In [1]: import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

import os, warnings, pathlib, glob, random
warnings.filterwarnings("ignore")
```

```
In [2]: import tensorflow as tf
from tensorflow import keras # this allows <keras.> instead of <tf.keras>
from tensorflow.keras import layers # this allows <layers.> instead of <tf.keras.layers>
tf.keras.utils.set_random_seed(111) # set random seed

import sklearn.metrics as metrics
```

```
In [4]: def load_data(base_dir="/Users/Ramv/Downloads/ninjacart_data"):
# checking if the data folders are present
assert os.path.exists(f"{base_dir}/train") and os.path.exists(f"{base_dir}/test")

train_data = tf.keras.utils.image_dataset_from_directory(
    f"{base_dir}/train", shuffle=True, label_mode='categorical'
)

test_data = tf.keras.utils.image_dataset_from_directory(
    f"{base_dir}/test", shuffle=False, label_mode='categorical'
)
return train_data, test_data, train_data.class_names
```

```
In [5]: train_data, test_data, class_names = load_data()
```

Found 3135 files belonging to 4 classes.
Found 351 files belonging to 4 classes.

```
In [8]: # Convert to DataFrames
df_train = pd.DataFrame(train_data)
df_test = pd.DataFrame(test_data)

# Preview
print("Train Data:")
display(df_train.head())

print("Test Data:")
display(df_test.head())
```

2025-07-17 14:10:10.802209: I tensorflow/core/framework/local_rendezvous.cc:407] Local rendezvous is aborting with status: OUT_OF_RANGE: End of sequence

Train Data:

	0	1
0	((tf.Tensor([38.852783 69.85278 10.852783], ...	((tf.Tensor(0.0, shape=(), dtype=float32), tf....
1	((tf.Tensor([255. 255. 255.], shape=(3,), dtype=...	((tf.Tensor(0.0, shape=(), dtype=float32), tf....
2	((tf.Tensor([185. 168. 152.], shape=(3,), dtype=...	((tf.Tensor(0.0, shape=(), dtype=float32), tf....
3	((tf.Tensor([55.804688 96.80469 38.804688], ...	((tf.Tensor(0.0, shape=(), dtype=float32), tf....
4	((tf.Tensor([170. 175. 169.], shape=(3,), dtype=...	((tf.Tensor(1.0, shape=(), dtype=float32), tf....

Test Data:

	0	1
0	((tf.Tensor([117.81372 106.81372 75.30078], ...	((tf.Tensor(1.0, shape=(), dtype=float32), tf....
1	((tf.Tensor([176.66406 198.66406 221.66406], ...	((tf.Tensor(1.0, shape=(), dtype=float32), tf....
2	((tf.Tensor([22.074219 20.074219 7.0742188...]	((tf.Tensor(1.0, shape=(), dtype=float32), tf....
3	((tf.Tensor([236. 236. 236.], shape=(3,), dtype=...	((tf.Tensor(0.0, shape=(), dtype=float32), tf....
4	((tf.Tensor([43.925323 29.753448 29.783493], ...	((tf.Tensor(0.0, shape=(), dtype=float32), tf....

```
In [9]: from pathlib import Path

def count_files(rootdir):
    """Counts and prints the number of files in each subfolder of a dir
    root_path = Path(rootdir)

    for subfolder in root_path.iterdir():
        if subfolder.is_dir():
            file_count = sum(1 for file in subfolder.iterdir() if file)
            print(f"There are {file_count} files in '{subfolder.name}'")
```

```
In [10]: count_files("/Users/Ramv/Downloads/ninjacart_data/train")
```

```
There are 849 files in 'onion'
There are 898 files in 'potato'
There are 599 files in 'indian market'
There are 789 files in 'tomato'
```

```
In [11]: count_files("/Users/Ramv/Downloads/ninjacart_data/test")
```

```

There are 83 files in 'onion'
There are 81 files in 'potato'
There are 81 files in 'indian market'
There are 106 files in 'tomato'

```

```
In [13]: image_dict
```

```

Out[13]: {'onion': <PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=200x
200>,
'potato': <PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=261
x193>,
'indian market': <PIL.JpegImagePlugin.JpegImageFile image mode=RGB s
ize=310x162>,
'tomato': <PIL.PngImagePlugin.PngImageFile image mode=RGB size=400x5
00>}

```

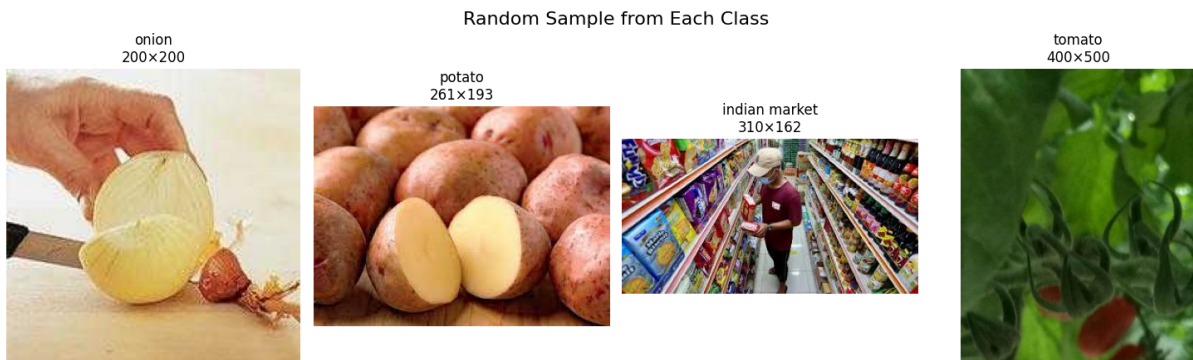
```

In [14]: # Set up the figure size and layout
plt.figure(figsize=(15, 12))

# Iterate over image_dict to plot one image per class
for i, (cls, img) in enumerate(image_dict.items()):
    ax = plt.subplot(3, 4, i + 1) # Adjust grid as needed
    plt.imshow(img)
    plt.title(f'{cls}\n{img.size[0]}x{img.size[1]}') # Width x Height
    plt.axis("off")

plt.suptitle("Random Sample from Each Class", fontsize=16)
plt.tight_layout()
plt.show()

```



```

In [17]: # Create DataFrame
df_count_train = pd.DataFrame({
    "class": list(count_dict.keys()),
    "count": list(count_dict.values())
})

# Display the class counts
print("Count of training samples per class:\n")
display(df_count_train)

# Generate one unique color per class using a colormap
num_classes = len(df_count_train)

```

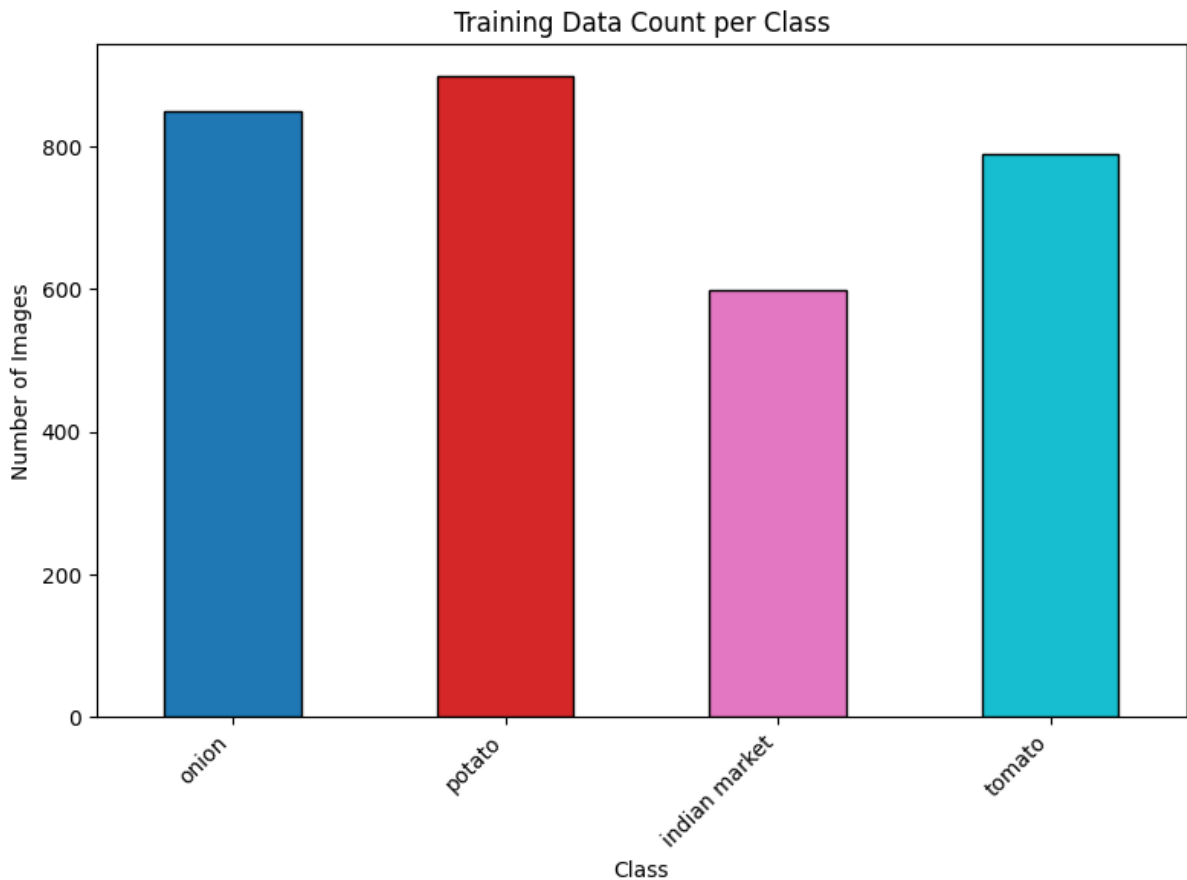
```
color_map = plt.cm.get_cmap('tab10', num_classes) # Use 'tab20', 'Set
colors = [color_map(i) for i in range(num_classes)] # Get a list of R

# Plot bar chart with individual colors
ax = df_count_train.plot.bar(
    x='class',
    y='count',
    legend=False,
    color=colors,
    edgecolor='black',
    figsize=(8, 6),
    title="Training Data Count per Class"
)

# Formatting
ax.set_xlabel("Class")
ax.set_ylabel("Number of Images")
ax.set_xticklabels(df_count_train["class"], rotation=45, ha='right')
plt.tight_layout()
plt.show()
```

Count of training samples per class:

	class	count
0	onion	849
1	potato	898
2	indian market	599
3	tomato	789



```
In [20]: import tensorflow as tf

# Parameters
seed_train_validation = 1
validation_split = 0.2
batch_size = 32
img_size = (224, 224) # You can adjust based on model needs (e.g., 299, 299)

# Load Training Dataset
print('\nLoading Train Data...')
train_data = tf.keras.utils.image_dataset_from_directory(
    '/Users/Ramv/Downloads/ninjacart_data/train',
    validation_split=validation_split,
    subset="training",
    seed=seed_train_validation,
    shuffle=True,
    image_size=img_size,
    batch_size=batch_size
)

# Load Validation Dataset
print('\nLoading Validation Data...')
val_data = tf.keras.utils.image_dataset_from_directory(
    '/Users/Ramv/Downloads/ninjacart_data/train',
    validation_split=validation_split,
    subset="validation",
    seed=seed_train_validation,
```

```

        shuffle=True,
        image_size=img_size,
        batch_size=batch_size
    )

    # Load Test Dataset (no split, no shuffle)
    print('\nLoading Test Data...')
    test_data = tf.keras.utils.image_dataset_from_directory(
        '/Users/Ramv/Downloads/ninjacart_data/test',
        shuffle=False,
        image_size=img_size,
        batch_size=batch_size
    )

```

Loading Train Data...
 Found 3135 files belonging to 4 classes.
 Using 2508 files for training.

Loading Validation Data...
 Found 3135 files belonging to 4 classes.
 Using 627 files for validation.

Loading Test Data...
 Found 351 files belonging to 4 classes.

In [21]: height, width = 128, 128

```

# Data Processing Stage with resizing and rescaling operations
data_preprocess = keras.Sequential(
    name="data_preprocess",
    layers=[
        layers.Resizing(height, width), # Shape Preprocessing
        layers.Rescaling(1.0/255), # Value Preprocessing
    ]
)

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

```

Creating Baseline Model : CNN Sratch

```

In [22]: def build_baseline_model(input_height=128, input_width=128, num_classes=4):
        """Builds a simple CNN baseline model."""

        model = keras.Sequential(name="baseline_cnn")

        # Convolution + MaxPooling
        model.add(layers.Conv2D(
            filters=16, kernel_size=3, padding='same', activation='relu',
            input_shape=(input_height, input_width, 3)

```

```

))
model.add(layers.MaxPooling2D())

# Fully Connected Layers
model.add(layers.Flatten())
model.add(layers.Dense(units=hidden_units, activation='relu'))
model.add(layers.Dense(units=num_classes, activation='softmax'))

return model

```

```

In [23]: model = build_baseline_model()
model.summary()

```

Model: "baseline_cnn"

Layer (type)	Output Shape	
conv2d (Conv2D)	(None, 128, 128, 16)	
max_pooling2d (MaxPooling2D)	(None, 64, 64, 16)	
flatten (Flatten)	(None, 65536)	
dense (Dense)	(None, 256)	1
dense_1 (Dense)	(None, 4)	

Total params: 16,778,948 (64.01 MB)

Trainable params: 16,778,948 (64.01 MB)

Non-trainable params: 0 (0.00 B)

```

In [24]: from tensorflow import keras

def compile_and_train_model(model, train_ds, val_ds, ckpt_path="/tmp/c
    """
    Compiles and trains the given Keras model using the Adam optimizer
    sparse categorical crossentropy loss.

    Args:
        model: Keras model to train.
        train_ds: Training dataset (tf.data.Dataset).
        val_ds: Validation dataset (tf.data.Dataset).
        ckpt_path: Path to save the best model weights.
        epochs: Number of training epochs.

    Returns:
        model_fit: History object containing training and validation m
    """

    # Compile the model
    model.compile(
        optimizer='adam',

```

```

        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )

    # Define the checkpoint callback
    checkpoint_cb = keras.callbacks.ModelCheckpoint(
        filepath=ckpt_path,
        save_weights_only=True,
        monitor='val_accuracy',
        mode='max',
        save_best_only=True,
        verbose=1
    )

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

    return model_fit

```

In [26]: `model_fit = compile_and_train_model(model, train_ds, val_ds)`

```

Epoch 1/10
78/79 ————— 0s 49ms/step - accuracy: 0.4744 - loss: 2.99
62
Epoch 1: val_accuracy improved from -inf to 0.80542, saving model to /t
mp/checkpoint.weights.h5
79/79 ————— 5s 56ms/step - accuracy: 0.4788 - loss: 2.95
77 - val_accuracy: 0.8054 - val_loss: 0.5086
Epoch 2/10
79/79 ————— 0s 53ms/step - accuracy: 0.8275 - loss: 0.47
94
Epoch 2: val_accuracy improved from 0.80542 to 0.83094, saving model to
/tmp/checkpoint.weights.h5
79/79 ————— 5s 58ms/step - accuracy: 0.8278 - loss: 0.47
87 - val_accuracy: 0.8309 - val_loss: 0.4502
Epoch 3/10
78/79 ————— 0s 51ms/step - accuracy: 0.8814 - loss: 0.32
33
Epoch 3: val_accuracy did not improve from 0.83094
79/79 ————— 4s 56ms/step - accuracy: 0.8820 - loss: 0.32
22 - val_accuracy: 0.8293 - val_loss: 0.4695
Epoch 4/10
78/79 ————— 0s 50ms/step - accuracy: 0.9379 - loss: 0.19
52
Epoch 4: val_accuracy did not improve from 0.83094
79/79 ————— 4s 55ms/step - accuracy: 0.9382 - loss: 0.19
47 - val_accuracy: 0.8230 - val_loss: 0.4553
Epoch 5/10

```



```

78/79 ————— 0s 51ms/step - accuracy: 0.9654 - loss: 0.11
37
Epoch 5: val_accuracy did not improve from 0.83094
79/79 ————— 4s 55ms/step - accuracy: 0.9655 - loss: 0.11
35 - val_accuracy: 0.8198 - val_loss: 0.5112
Epoch 6/10
79/79 ————— 0s 51ms/step - accuracy: 0.9832 - loss: 0.07
23
Epoch 6: val_accuracy did not improve from 0.83094
79/79 ————— 4s 56ms/step - accuracy: 0.9832 - loss: 0.07
22 - val_accuracy: 0.8182 - val_loss: 0.5469
Epoch 7/10
79/79 ————— 0s 50ms/step - accuracy: 0.9937 - loss: 0.05
28
Epoch 7: val_accuracy improved from 0.83094 to 0.83413, saving model to
/tmp/checkpoint.weights.h5
79/79 ————— 4s 55ms/step - accuracy: 0.9937 - loss: 0.05
27 - val_accuracy: 0.8341 - val_loss: 0.5255
Epoch 8/10
79/79 ————— 0s 49ms/step - accuracy: 0.9920 - loss: 0.03
13
Epoch 8: val_accuracy did not improve from 0.83413
79/79 ————— 4s 54ms/step - accuracy: 0.9920 - loss: 0.03
13 - val_accuracy: 0.7847 - val_loss: 0.8682
Epoch 9/10
78/79 ————— 0s 52ms/step - accuracy: 0.9889 - loss: 0.03
61
Epoch 9: val_accuracy did not improve from 0.83413
79/79 ————— 5s 56ms/step - accuracy: 0.9891 - loss: 0.03
59 - val_accuracy: 0.8214 - val_loss: 0.5438
Epoch 10/10
78/79 ————— 0s 50ms/step - accuracy: 0.9972 - loss: 0.01
65
Epoch 10: val_accuracy improved from 0.83413 to 0.83732, saving model t
o /tmp/checkpoint.weights.h5
79/79 ————— 4s 56ms/step - accuracy: 0.9972 - loss: 0.01
64 - val_accuracy: 0.8373 - val_loss: 0.5939

```

```

In [50]: fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(15,5))
         ax = axes.ravel()

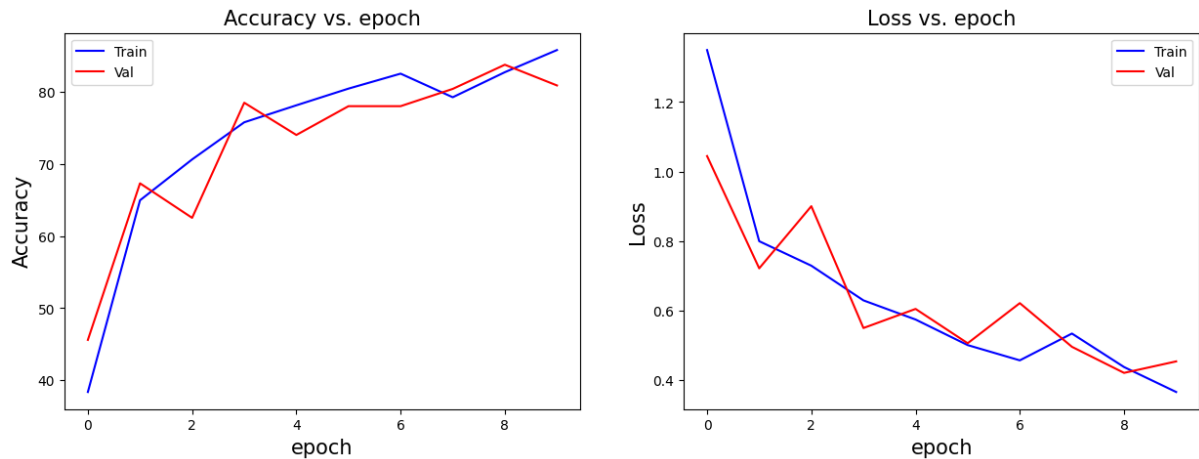
         #accuracy graph
         ax[0].plot(range(0,model_fit.params['epochs']), [acc * 100 for acc in
         ax[0].plot(range(0,model_fit.params['epochs']), [acc * 100 for acc in
         ax[0].set_title('Accuracy vs. epoch', fontsize=15)
         ax[0].set_ylabel('Accuracy', fontsize=15)
         ax[0].set_xlabel('epoch', fontsize=15)
         ax[0].legend()

         #loss graph
         ax[1].plot(range(0,model_fit.params['epochs']), model_fit.history['los
         ax[1].plot(range(0,model_fit.params['epochs']), model_fit.history['val
         ax[1].set_title('Loss vs. epoch', fontsize=15)

```

```
ax[1].set_ylabel('Loss', fontsize=15)
ax[1].set_xlabel('epoch', fontsize=15)
ax[1].legend()

#display the graph
plt.show()
```



Accuracy vs Epoch:

Training Accuracy steadily increases and nearly reaches 100% by the final epoch.

Validation Accuracy improves initially, then fluctuates and plateaus around 80–84%.

Loss vs Epoch:

Training Loss drops consistently, reaching near zero — indicating the model is fitting the training data extremely well.

Validation Loss, however, shows no meaningful improvement after the first 2–3 epochs and even increases intermittently.

```
In [28]: # Load best model weights from checkpoint (optional)
checkpoint_path = "/tmp/checkpoint.weights.h5"
model.load_weights(checkpoint_path)

# Predict probabilities on test dataset
y_pred_probs = model.predict(test_ds)

# Convert predicted probabilities to class labels
predicted_categories = tf.argmax(y_pred_probs, axis=1)

# Get true labels from test_ds
true_categories = tf.concat([y for x, y in test_ds], axis=0)

# Get class names from the dataset (useful for further reporting or co
class_names = test_data.class_names
```

```
# Calculate accuracy
test_acc = metrics.accuracy_score(true_categories, predicted_categories)
print(f"\n✅ Test Accuracy: {test_acc:.2f}%")

# Optional: Print classification report
print("\n📊 Classification Report:")
print(metrics.classification_report(true_categories, predicted_categories))
```

11/11 ————— 0s 18ms/step

✅ Test Accuracy: 73.79%

📊 Classification Report:

	precision	recall	f1-score	support
indian market	0.88	0.65	0.75	81
onion	0.62	0.53	0.57	83
potato	0.54	0.78	0.64	81
tomato	0.96	0.93	0.95	106
accuracy			0.74	351
macro avg	0.75	0.72	0.73	351
weighted avg	0.76	0.74	0.74	351

```
In [29]: def ConfusionMatrix(model, ds, label_list, title="Confusion Matrix"):
        """
        Plots a confusion matrix from a model and a dataset.

        Args:
            model: Trained tf.keras model.
            ds: Test or validation dataset (tf.data.Dataset).
            label_list: List of class names in order.
            title: Title for the plot (optional).
        """
        # Disable shuffle if needed to preserve order
        y_pred_probs = model.predict(ds)
        y_pred = tf.argmax(y_pred_probs, axis=1)

        # Gather true labels from dataset
        y_true = tf.concat([y for x, y in ds], axis=0)

        # Convert to NumPy arrays (required by sklearn)
        y_true = y_true.numpy()
        y_pred = y_pred.numpy()

        # Generate confusion matrix
        cm = metrics.confusion_matrix(y_true, y_pred)

        # Plot heatmap
        plt.figure(figsize=(8, 6))
        sns.heatmap(cm, annot=True, fmt='g', cmap='YlGnBu',
```

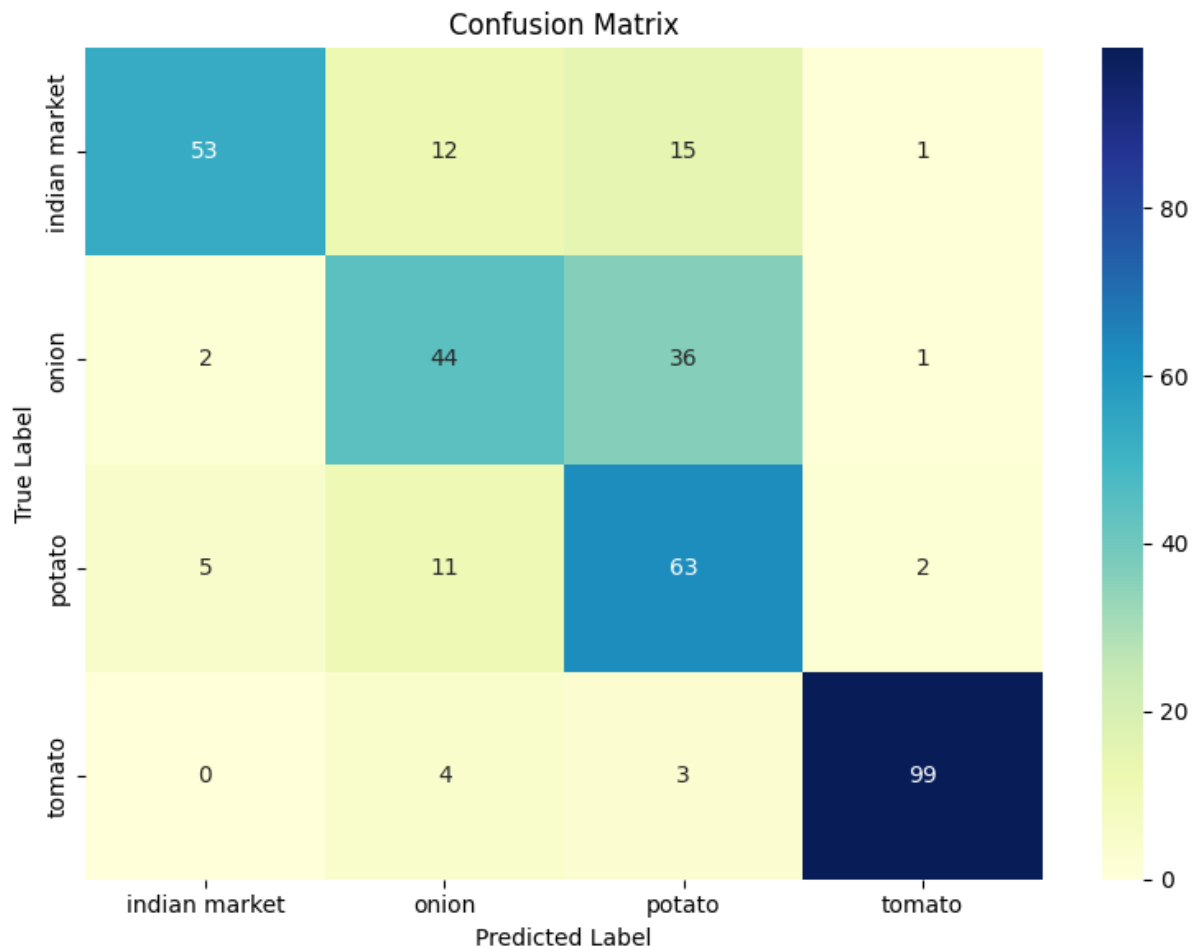
```

        xticklabels=label_list, yticklabels=label_list)
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title(title)
plt.tight_layout()
plt.show()

```

In [30]: ConfusionMatrix(model, test_ds, test_data.class_names)

11/11 ————— 0s 19ms/step



Training Model with Data Augmentation

```

In [31]: # 1. Set basic parameters
img_height, img_width = 128, 128
batch_size = 32
num_classes = 4

# 2. Define the data augmentation pipeline
data_augmentation = keras.Sequential(
    name="data_augmentation",
    layers=[
        layers.RandomFlip("horizontal"),
        layers.RandomRotation(0.1),
        layers.RandomZoom(0.1),
    ]
)

```

```

        layers.RandomContrast(0.1),
        layers.RandomTranslation(0.1, 0.1),
    ]
)

# 3. Define preprocessing (resizing + rescaling)
preprocess = keras.Sequential([
    layers.Resizing(img_height, img_width),
    layers.Rescaling(1./255)
])

# 4. Apply preprocessing + augmentation to training dataset only
def prepare_dataset(dataset, training=False):
    dataset = dataset.map(lambda x, y: (preprocess(x), y))
    if training:
        dataset = dataset.map(lambda x, y: (data_augmentation(x), y))
    return dataset.prefetch(buffer_size=tf.data.AUTOTUNE)

train_ds_aug = prepare_dataset(train_data, training=True)
val_ds_proc = prepare_dataset(val_data, training=False)
test_ds_proc = prepare_dataset(test_data, training=False)

```

```

In [32]: def build_model(input_shape=(img_height, img_width, 3), num_classes=4)
        model = keras.Sequential(name="cnn_augmented")
        model.add(layers.Conv2D(32, 3, activation='relu', padding='same',
        model.add(layers.MaxPooling2D())
        model.add(layers.Conv2D(64, 3, activation='relu', padding='same'))
        model.add(layers.MaxPooling2D())
        model.add(layers.Flatten())
        model.add(layers.Dense(256, activation='relu'))
        model.add(layers.Dropout(0.5))
        model.add(layers.Dense(num_classes, activation='softmax'))
        return model

```

```

In [34]: model = build_model()

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

# Optional: Add callbacks
checkpoint_cb = keras.callbacks.ModelCheckpoint(
    filepath="/tmp/best_model_augmented.weights.h5",
    save_weights_only=True,
    monitor='val_accuracy',
    mode='max',
    save_best_only=True,
    verbose=1
)

```

```
earlystop_cb = keras.callbacks.EarlyStopping(  
    monitor='val_loss',  
    patience=3,  
    restore_best_weights=True  
)  
  
# Train the model  
history = model.fit(  
    train_ds_aug,  
    validation_data=val_ds_proc,  
    epochs=10,  
    callbacks=[checkpoint_cb, earlystop_cb]  
)
```

Epoch 1/10
79/79 ————— 0s 113ms/step - accuracy: 0.4236 - loss: 2.0052
Epoch 1: val_accuracy improved from -inf to 0.76236, saving model to /tmp/best_model_augmented.weights.h5
79/79 ————— 10s 123ms/step - accuracy: 0.4253 - loss: 1.9959 - val_accuracy: 0.7624 - val_loss: 0.6105
Epoch 2/10
79/79 ————— 0s 117ms/step - accuracy: 0.7440 - loss: 0.6653
Epoch 2: val_accuracy improved from 0.76236 to 0.77193, saving model to /tmp/best_model_augmented.weights.h5
79/79 ————— 10s 127ms/step - accuracy: 0.7444 - loss: 0.6648 - val_accuracy: 0.7719 - val_loss: 0.5692
Epoch 3/10
79/79 ————— 0s 114ms/step - accuracy: 0.7964 - loss: 0.5530
Epoch 3: val_accuracy improved from 0.77193 to 0.81499, saving model to /tmp/best_model_augmented.weights.h5
79/79 ————— 10s 124ms/step - accuracy: 0.7964 - loss: 0.5529 - val_accuracy: 0.8150 - val_loss: 0.4742
Epoch 4/10
79/79 ————— 0s 113ms/step - accuracy: 0.7979 - loss: 0.5100
Epoch 4: val_accuracy did not improve from 0.81499
79/79 ————— 10s 121ms/step - accuracy: 0.7981 - loss: 0.5098 - val_accuracy: 0.7097 - val_loss: 0.8868
Epoch 5/10
79/79 ————— 0s 113ms/step - accuracy: 0.8142 - loss: 0.4741
Epoch 5: val_accuracy improved from 0.81499 to 0.82775, saving model to /tmp/best_model_augmented.weights.h5
79/79 ————— 10s 123ms/step - accuracy: 0.8144 - loss: 0.4738 - val_accuracy: 0.8278 - val_loss: 0.4648
Epoch 6/10
79/79 ————— 0s 112ms/step - accuracy: 0.8446 - loss: 0.4012
Epoch 6: val_accuracy did not improve from 0.82775
79/79 ————— 10s 120ms/step - accuracy: 0.8445 - loss: 0.4016 - val_accuracy: 0.7927 - val_loss: 0.5036
Epoch 7/10
79/79 ————— 0s 113ms/step - accuracy: 0.8330 - loss: 0.4279
Epoch 7: val_accuracy did not improve from 0.82775
79/79 ————— 10s 121ms/step - accuracy: 0.8331 - loss: 0.4278 - val_accuracy: 0.8166 - val_loss: 0.4867
Epoch 8/10
79/79 ————— 0s 112ms/step - accuracy: 0.8415 - loss: 0.4209
Epoch 8: val_accuracy did not improve from 0.82775
79/79 ————— 10s 120ms/step - accuracy: 0.8415 - loss: 0.4208 - val_accuracy: 0.7895 - val_loss: 0.6241

```
In [36]: def plot_training_metrics(history):
        """
        Plots Accuracy and Loss over Epochs for training and validation.

        Args:
            history: History object returned by model.fit()
        """
        # Extract metrics
        epochs = range(1, len(history.history['accuracy']) + 1)
        train_acc = history.history['accuracy']
        val_acc = history.history['val_accuracy']
        train_loss = history.history['loss']
        val_loss = history.history['val_loss']

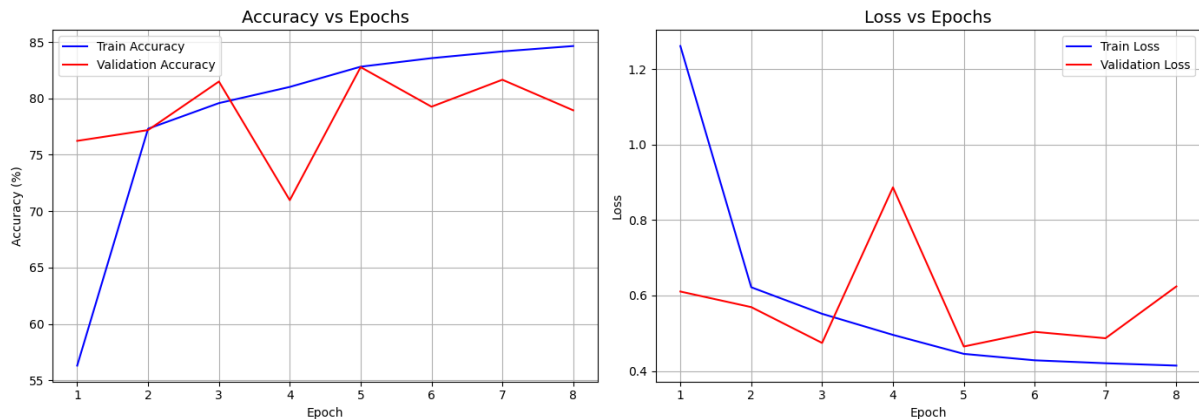
        # Plot Accuracy
        plt.figure(figsize=(14, 5))

        plt.subplot(1, 2, 1)
        plt.plot(epochs, [a * 100 for a in train_acc], 'b-', label='Train')
        plt.plot(epochs, [a * 100 for a in val_acc], 'r-', label='Validation')
        plt.title('Accuracy vs Epochs', fontsize=14)
        plt.xlabel('Epoch')
        plt.ylabel('Accuracy (%)')
        plt.legend()
        plt.grid(True)

        # Plot Loss
        plt.subplot(1, 2, 2)
        plt.plot(epochs, train_loss, 'b-', label='Train Loss')
        plt.plot(epochs, val_loss, 'r-', label='Validation Loss')
        plt.title('Loss vs Epochs', fontsize=14)
        plt.xlabel('Epoch')
        plt.ylabel('Loss')
        plt.legend()
        plt.grid(True)

        # Show plots
        plt.tight_layout()
        plt.show()

        # Usage
        plot_training_metrics(history)
```

model with data augmentation performs well, reaching ~83% validation accuracy, with training steadily improving. However, the validation instability after Epoch 5 suggests that generalization can still be enhanced

Accuracy vs Epochs: Training Accuracy improves steadily from ~56% to ~85%, indicating that the model is learning the patterns in the training data well.

Validation Accuracy starts around 76% and fluctuates across epochs. It peaks at Epoch 5 (~83%), but then declines or stabilizes, ending slightly lower at ~79%.

Interpretation: The gap between training and validation accuracy is small, suggesting data augmentation helped reduce overfitting by introducing more variation in training images. However, validation accuracy fluctuations hint at some instability or sensitivity in generalization, possibly due to: Insufficient augmentation strength or dataset size Model capacity being marginal for the task Label noise or ambiguous examples

Loss vs Epochs: Training Loss shows a smooth decline from 1.25 to ~0.42, indicating effective learning. Validation Loss, however, is non-monotonic — it: Decreases till Epoch 3 Spikes at Epoch 4, then improves again Ends higher than its best point (similar to Epoch 1)

Interpretation: The spike in validation loss (despite decent accuracy) suggests the model was less confident or made wrong predictions with high certainty, leading to a higher loss penalty. The increase in final validation loss may signal onset of overfitting, which early stopping could help mitigate.

In []:

In []:

In []:

Deep CNN Model

```
In [37]: def deep_cnn(height=128, width=128, num_classes=10, hidden_size=256, dropout_rate=0.5):
        """
        Builds a deeper CNN model with progressively increasing filters and layers.

        Args:
            height (int): Height of the input image.
            width (int): Width of the input image.
            num_classes (int): Number of output classes.
            hidden_size (int): Units in the dense layer.
            dropout_rate (float): Dropout rate after dense layer.

        Returns:
            keras.Model: A compiled CNN model.
        """
        model = keras.Sequential(name="model_cnn_deep")

        # Convolutional feature extractor
        model.add(layers.Conv2D(16, kernel_size=3, padding="same", activation='relu'))
        model.add(layers.MaxPooling2D())

        model.add(layers.Conv2D(32, kernel_size=3, padding="same", activation='relu'))
        model.add(layers.MaxPooling2D())

        model.add(layers.Conv2D(64, kernel_size=3, padding="same", activation='relu'))
        model.add(layers.MaxPooling2D())

        model.add(layers.Conv2D(128, kernel_size=3, padding="same", activation='relu'))
        model.add(layers.MaxPooling2D())

        model.add(layers.Conv2D(256, kernel_size=3, padding="same", activation='relu'))

        # Replace Flatten with GlobalAveragePooling for better generalization
        model.add(layers.GlobalAveragePooling2D())

        # Dense layer for classification
        model.add(layers.Dense(hidden_size, activation='relu'))
        model.add(layers.Dropout(dropout_rate)) # Regularization
        model.add(layers.Dense(num_classes, activation='softmax'))

        return model
```

```
In [38]: model = deep_cnn()
        model.summary()
```

Model: "model_cnn_deep"

Layer (type)	Output Shape	
conv2d_5 (Conv2D)	(None, 128, 128, 16)	
max_pooling2d_5 (MaxPooling2D)	(None, 64, 64, 16)	
conv2d_6 (Conv2D)	(None, 64, 64, 32)	
max_pooling2d_6 (MaxPooling2D)	(None, 32, 32, 32)	
conv2d_7 (Conv2D)	(None, 32, 32, 64)	
max_pooling2d_7 (MaxPooling2D)	(None, 16, 16, 64)	
conv2d_8 (Conv2D)	(None, 16, 16, 128)	
max_pooling2d_8 (MaxPooling2D)	(None, 8, 8, 128)	
conv2d_9 (Conv2D)	(None, 8, 8, 256)	
global_average_pooling2d (GlobalAveragePooling2D)	(None, 256)	
dense_6 (Dense)	(None, 256)	
dropout_2 (Dropout)	(None, 256)	
dense_7 (Dense)	(None, 10)	

Total params: 460,970 (1.76 MB)

Trainable params: 460,970 (1.76 MB)

Non-trainable params: 0 (0.00 B)

```
In [39]: model_fit = compile_and_train_model(model, train_ds, val_ds)
```

Epoch 1/10

78/79 ————— 0s 76ms/step - accuracy: 0.2838 - loss: 1.6465

Epoch 1: val_accuracy improved from -inf to 0.45614, saving model to /tmp/checkpoint.weights.h5

79/79 ————— 7s 83ms/step - accuracy: 0.2863 - loss: 1.6390 - val_accuracy: 0.4561 - val_loss: 1.0444

Epoch 2/10

78/79 ————— 0s 79ms/step - accuracy: 0.6253 - loss: 0.8565

Epoch 2: val_accuracy improved from 0.45614 to 0.67305, saving model to /tmp/checkpoint.weights.h5

79/79 ————— 7s 86ms/step - accuracy: 0.6259 - loss: 0.8551 - val_accuracy: 0.6730 - val_loss: 0.7213

Epoch 3/10

78/79 ————— 0s 79ms/step - accuracy: 0.6865 - loss: 0.7305

```

Epoch 3: val_accuracy did not improve from 0.67305
79/79 ————— 7s 86ms/step - accuracy: 0.6870 - loss: 0.73
04 - val_accuracy: 0.6252 - val_loss: 0.9001
Epoch 4/10
78/79 ————— 0s 76ms/step - accuracy: 0.7282 - loss: 0.70
01
Epoch 4: val_accuracy improved from 0.67305 to 0.78469, saving model to
/tmp/checkpoint.weights.h5
79/79 ————— 7s 83ms/step - accuracy: 0.7289 - loss: 0.69
84 - val_accuracy: 0.7847 - val_loss: 0.5495
Epoch 5/10
78/79 ————— 0s 76ms/step - accuracy: 0.7671 - loss: 0.60
80
Epoch 5: val_accuracy did not improve from 0.78469
79/79 ————— 7s 83ms/step - accuracy: 0.7674 - loss: 0.60
72 - val_accuracy: 0.7400 - val_loss: 0.6045
Epoch 6/10
78/79 ————— 0s 78ms/step - accuracy: 0.7900 - loss: 0.53
46
Epoch 6: val_accuracy did not improve from 0.78469
79/79 ————— 7s 85ms/step - accuracy: 0.7904 - loss: 0.53
37 - val_accuracy: 0.7799 - val_loss: 0.5057
Epoch 7/10
78/79 ————— 0s 78ms/step - accuracy: 0.8005 - loss: 0.48
48
Epoch 7: val_accuracy did not improve from 0.78469
79/79 ————— 7s 85ms/step - accuracy: 0.8011 - loss: 0.48
41 - val_accuracy: 0.7799 - val_loss: 0.6212
Epoch 8/10
78/79 ————— 0s 79ms/step - accuracy: 0.7909 - loss: 0.54
21
Epoch 8: val_accuracy improved from 0.78469 to 0.80383, saving model to
/tmp/checkpoint.weights.h5
79/79 ————— 7s 86ms/step - accuracy: 0.7909 - loss: 0.54
19 - val_accuracy: 0.8038 - val_loss: 0.4955
Epoch 9/10
78/79 ————— 0s 78ms/step - accuracy: 0.8151 - loss: 0.46
80
Epoch 9: val_accuracy improved from 0.80383 to 0.83732, saving model to
/tmp/checkpoint.weights.h5
79/79 ————— 7s 85ms/step - accuracy: 0.8154 - loss: 0.46
73 - val_accuracy: 0.8373 - val_loss: 0.4204
Epoch 10/10
78/79 ————— 0s 75ms/step - accuracy: 0.8490 - loss: 0.39
16
Epoch 10: val_accuracy did not improve from 0.83732
79/79 ————— 7s 82ms/step - accuracy: 0.8492 - loss: 0.39
10 - val_accuracy: 0.8086 - val_loss: 0.4535

```

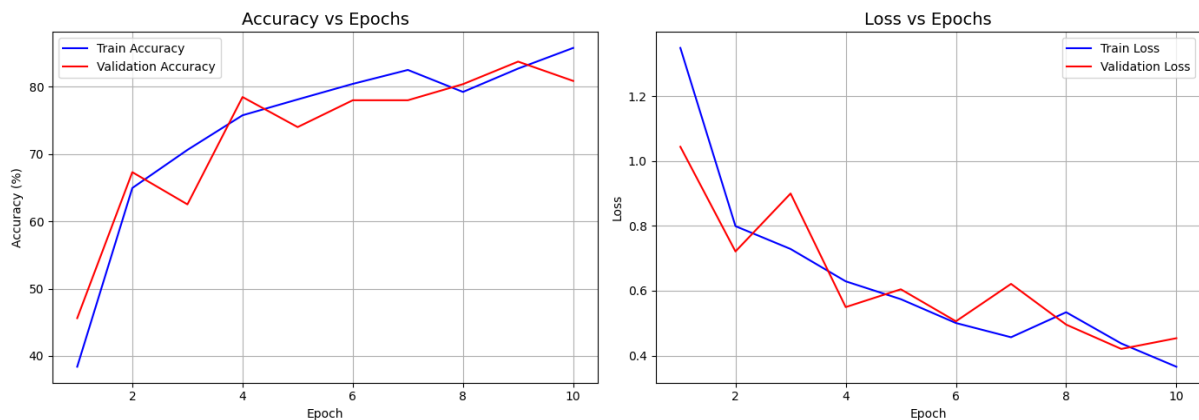
```

In [40]: print(model_fit.history.keys())

dict_keys(['accuracy', 'loss', 'val_accuracy', 'val_loss'])

```

```
In [41]: plot_training_metrics(model_fit)
```



Accuracy vs Epochs:

Training Accuracy increases consistently from ~38% to ~86% over 10 epochs — a strong sign that the model is learning effectively. Validation Accuracy also improves, though with some fluctuations around epochs 3–5, eventually reaching a stable performance near 82–84%.

Interpretation:

Both training and validation accuracy trends are upward, with minimal gap — indicating: Good generalization No severe overfitting (thanks to data augmentation and model depth) Slight fluctuations in validation accuracy may be due to: Limited validation data Natural noise or complexity in the dataset

Loss vs Epochs:

Training Loss steadily decreases from ~1.3 to ~0.36. Validation Loss decreases overall but shows more volatility, especially at epochs 3, 5, and 7. Despite fluctuations, validation loss ends slightly above training loss, at around 0.45, which is acceptable.

Interpretation:

The model is learning effectively on training data (clean downward trend). Spikes in validation loss (even when validation accuracy is stable) suggest: The model sometimes makes confident but incorrect predictions (high-penalty mistakes) Data augmentation may have helped prevent overfitting, but there's still some noise or class overlap in the data

```
In [47]: def calculate_testing_accuracy(model, test_ds):
# load model from pretrained checkpoints (optional)
model.load_weights("/tmp/checkpoint.weights.h5")
```

```
# run model prediction and obtain probabilities
y_pred = model.predict(test_ds)

# get list of predicted classes by taking argmax of the probabilities
predicted_categories = tf.argmax(y_pred, axis=1)

# get list of class names
class_names = test_data.class_names

# create list of all "y"s labels, by iterating over test dataset
true_categories = tf.concat([y for x, y in test_ds], axis=0)

# calculate accuracy
test_acc = metrics.accuracy_score(true_categories, predicted_categories)
print(f'\nTest Accuracy: {test_acc:.2f}%\n')
```

In [48]: calculate_testing_accuracy(model, test_ds)

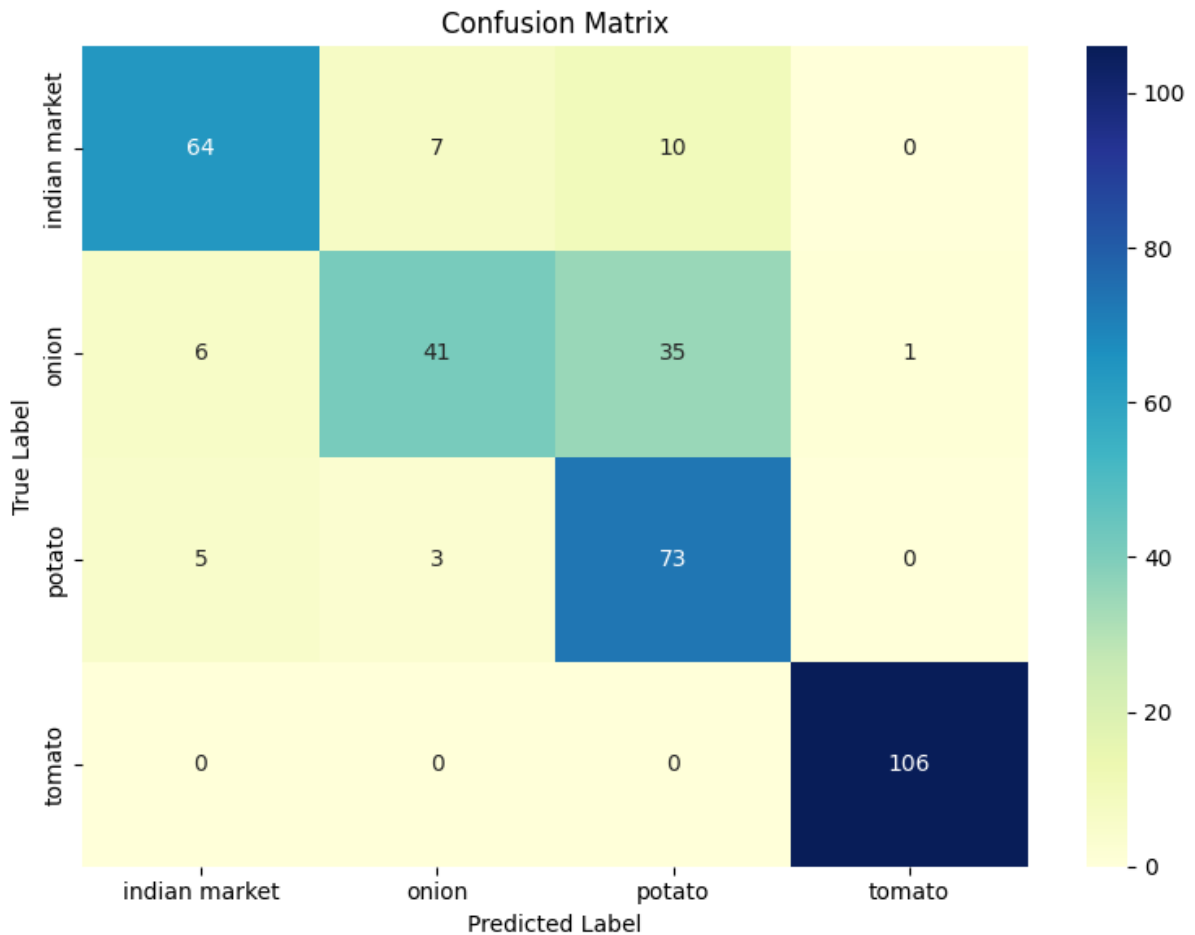
11/11 ————— 0s 26ms/step

Test Accuracy: 80.91%

In [49]: ConfusionMatrix(model, test_ds, test_data.class_names)

11/11 ————— 0s 29ms/step

2025-07-22 00:28:57.203788: I tensorflow/core/framework/local_rendezvous.cc:407] Local rendezvous is aborting with status: OUT_OF_RANGE: End of sequence



Deeper CNN Model

```
In [51]: def compile_and_train_v2(model, train_ds, val_ds, epochs=10, ckpt_path
         callbacks = [
             keras.callbacks.ReduceLROnPlateau(
                 monitor="val_loss", factor=0.3, patience=5, min_lr=0.00001
             ),
             keras.callbacks.ModelCheckpoint(ckpt_path, save_weights_only=True),
             keras.callbacks.EarlyStopping(
                 monitor="val_loss", patience=10, min_delta=0.001, mode='min'
             )
         ]
         model.compile(optimizer='adam',
                       loss='sparse_categorical_crossentropy',
                       metrics=['accuracy'])
         model_fit = model.fit(train_ds, validation_data=val_ds, epochs=epochs)
         return model_fit
```

```
In [52]: from tensorflow.keras import layers, models

         def conv_block(filters, kernel_size=3, pool=True):
             """
             Builds a convolutional block with Conv2D → ReLU → BatchNorm → (Optional Pooling)
             """
```

```

    block = [
        layers.Conv2D(filters, kernel_size, padding="same"),
        layers.Activation("relu"),
        layers.BatchNormalization()
    ]
    if pool:
        block.append(layers.MaxPooling2D())
    return block

def deeper_cnn(height=128, width=128, num_classes=10, hidden_size=256,
    """
    Builds a deeper CNN model with batch normalization and global average pooling.

    Args:
        height (int): Input image height.
        width (int): Input image width.
        num_classes (int): Number of output classes.
        hidden_size (int): Units in the dense layer.
        dropout_rate (float): Dropout rate after dense layer.

    Returns:
        tf.keras.Model: Compiled CNN model.
    """
    model = models.Sequential(name="model_cnn_deeper")

    # Input layer
    model.add(layers.Input(shape=(height, width, 3)))

    # Convolutional blocks
    for filters in [16, 32, 64, 128, 256]:
        model.add(layers.Conv2D(filters, kernel_size=3, padding="same"))
        model.add(layers.Activation("relu"))
        model.add(layers.BatchNormalization())
        model.add(layers.MaxPooling2D())

    # Global Average Pooling instead of Flatten
    model.add(layers.GlobalAveragePooling2D())

    # Fully connected layer
    model.add(layers.Dense(hidden_size))
    model.add(layers.Activation("relu"))
    model.add(layers.BatchNormalization())
    model.add(layers.Dropout(dropout_rate))

    # Output layer
    model.add(layers.Dense(num_classes, activation='softmax'))

    return model

```

```

In [53]: model = deeper_cnn()
         model.summary()

```


Model: "model_cnn_deeper"

Layer (type)	Output Shape	
conv2d_10 (Conv2D)	(None, 128, 128, 16)	
activation (Activation)	(None, 128, 128, 16)	
batch_normalization (BatchNormalization)	(None, 128, 128, 16)	
max_pooling2d_9 (MaxPooling2D)	(None, 64, 64, 16)	
conv2d_11 (Conv2D)	(None, 64, 64, 32)	
activation_1 (Activation)	(None, 64, 64, 32)	
batch_normalization_1 (BatchNormalization)	(None, 64, 64, 32)	
max_pooling2d_10 (MaxPooling2D)	(None, 32, 32, 32)	
conv2d_12 (Conv2D)	(None, 32, 32, 64)	
activation_2 (Activation)	(None, 32, 32, 64)	
batch_normalization_2 (BatchNormalization)	(None, 32, 32, 64)	
max_pooling2d_11 (MaxPooling2D)	(None, 16, 16, 64)	
conv2d_13 (Conv2D)	(None, 16, 16, 128)	
activation_3 (Activation)	(None, 16, 16, 128)	
batch_normalization_3 (BatchNormalization)	(None, 16, 16, 128)	
max_pooling2d_12 (MaxPooling2D)	(None, 8, 8, 128)	
conv2d_14 (Conv2D)	(None, 8, 8, 256)	
activation_4 (Activation)	(None, 8, 8, 256)	
batch_normalization_4 (BatchNormalization)	(None, 8, 8, 256)	
max_pooling2d_13 (MaxPooling2D)	(None, 4, 4, 256)	
global_average_pooling2d_1 (GlobalAveragePooling2D)	(None, 256)	
dense_8 (Dense)	(None, 256)	
activation_5 (Activation)	(None, 256)	

batch_normalization_5 (BatchNormalization)	(None, 256)	
dropout_3 (Dropout)	(None, 256)	
dense_9 (Dense)	(None, 10)	

Total params: 463,978 (1.77 MB)

Trainable params: 462,474 (1.76 MB)

Non-trainable params: 1,504 (5.88 KB)

```
In [54]: model_fit = compile_and_train_v2(model, train_ds, val_ds,100 )
```

Epoch 1/100

79/79 ————— 12s 127ms/step - accuracy: 0.6037 - loss: 1.4434 - val_accuracy: 0.2488 - val_loss: 3.2716 - learning_rate: 0.0010

Epoch 2/100

79/79 ————— 10s 125ms/step - accuracy: 0.7791 - loss: 0.6683 - val_accuracy: 0.2663 - val_loss: 3.0384 - learning_rate: 0.0010

Epoch 3/100

79/79 ————— 10s 123ms/step - accuracy: 0.8053 - loss: 0.5706 - val_accuracy: 0.2584 - val_loss: 5.1662 - learning_rate: 0.0010

Epoch 4/100

79/79 ————— 10s 122ms/step - accuracy: 0.8260 - loss: 0.5008 - val_accuracy: 0.4593 - val_loss: 1.6724 - learning_rate: 0.0010

Epoch 5/100

79/79 ————— 10s 126ms/step - accuracy: 0.8309 - loss: 0.4857 - val_accuracy: 0.5630 - val_loss: 0.9868 - learning_rate: 0.0010

Epoch 6/100

79/79 ————— 10s 124ms/step - accuracy: 0.8371 - loss: 0.4384 - val_accuracy: 0.5885 - val_loss: 0.9795 - learning_rate: 0.0010

Epoch 7/100

79/79 ————— 10s 127ms/step - accuracy: 0.8475 - loss: 0.4088 - val_accuracy: 0.8022 - val_loss: 0.5298 - learning_rate: 0.0010

Epoch 8/100

79/79 ————— 10s 126ms/step - accuracy: 0.8638 - loss: 0.3672 - val_accuracy: 0.7209 - val_loss: 0.9004 - learning_rate: 0.0010

Epoch 9/100

79/79 ————— 10s 128ms/step - accuracy: 0.8812 - loss: 0.3623 - val_accuracy: 0.7735 - val_loss: 0.7089 - learning_rate: 0.0010

Epoch 10/100

79/79 ————— 10s 124ms/step - accuracy: 0.8715 - loss: 0.3055 - val_accuracy: 0.7033 - val_loss: 0.7199 - learning_rate: 0.0010

Epoch 11/100















79/79 ————— 10s 128ms/step - accuracy: 0.8961 - loss: 0.2714 - val_accuracy: 0.8788 - val_loss: 0.3629 - learning_rate: 0.0010

Epoch 12/100

79/79 ————— 12s 153ms/step - accuracy: 0.8869 - loss: 0.2946 - val_accuracy: 0.7831 - val_loss: 0.5952 - learning_rate: 0.0010

Epoch 13/100

79/79 ————— 10s 127ms/step - accuracy: 0.8914 - loss: 0.

2970 - val_accuracy: 0.8246 - val_loss: 0.5170 - learning_rate: 0.0010
Epoch 14/100
79/79  **10s** 125ms/step - accuracy: 0.9109 - loss: 0.
2255 - val_accuracy: 0.8453 - val_loss: 0.4245 - learning_rate: 0.0010
Epoch 15/100
79/79  **9s** 118ms/step - accuracy: 0.9152 - loss: 0.2
328 - val_accuracy: 0.8533 - val_loss: 0.4164 - learning_rate: 0.0010
Epoch 16/100
79/79  **10s** 120ms/step - accuracy: 0.8919 - loss: 0.
2597 - val_accuracy: 0.7847 - val_loss: 0.8568 - learning_rate: 0.0010
Epoch 17/100
79/79  **10s** 128ms/step - accuracy: 0.9442 - loss: 0.
1811 - val_accuracy: 0.9123 - val_loss: 0.2265 - learning_rate: 3.0000e
-04
Epoch 18/100
79/79  **10s** 126ms/step - accuracy: 0.9472 - loss: 0.
1385 - val_accuracy: 0.9059 - val_loss: 0.2751 - learning_rate: 3.0000e
-04
Epoch 19/100
79/79  **10s** 121ms/step - accuracy: 0.9608 - loss: 0.
1136 - val_accuracy: 0.9091 - val_loss: 0.2839 - learning_rate: 3.0000e
-04
Epoch 20/100
79/79  **10s** 126ms/step - accuracy: 0.9661 - loss: 0.
0872 - val_accuracy: 0.9155 - val_loss: 0.2653 - learning_rate: 3.0000e
-04
Epoch 21/100
79/79  **10s** 126ms/step - accuracy: 0.9718 - loss: 0.
0890 - val_accuracy: 0.8852 - val_loss: 0.3211 - learning_rate: 3.0000e
-04
Epoch 22/100
79/79  **10s** 130ms/step - accuracy: 0.9708 - loss: 0.
0826 - val_accuracy: 0.9123 - val_loss: 0.2703 - learning_rate: 3.0000e
-04
Epoch 23/100
79/79  **10s** 125ms/step - accuracy: 0.9830 - loss: 0.
0556 - val_accuracy: 0.9187 - val_loss: 0.2530 - learning_rate: 9.0000e
-05
Epoch 24/100
79/79  **10s** 120ms/step - accuracy: 0.9734 - loss: 0.
0678 - val_accuracy: 0.9282 - val_loss: 0.2317 - learning_rate: 9.0000e
-05
Epoch 25/100
79/79  **10s** 122ms/step - accuracy: 0.9777 - loss: 0.
0628 - val_accuracy: 0.9203 - val_loss: 0.2460 - learning_rate: 9.0000e
-05
Epoch 26/100
79/79  **10s** 132ms/step - accuracy: 0.9889 - loss: 0.
0459 - val_accuracy: 0.9250 - val_loss: 0.2380 - learning_rate: 9.0000e
-05
Epoch 27/100
79/79  **10s** 127ms/step - accuracy: 0.9908 - loss: 0.

0337 - val_accuracy: 0.9266 - val_loss: 0.2540 - learning_rate: 9.0000e-05

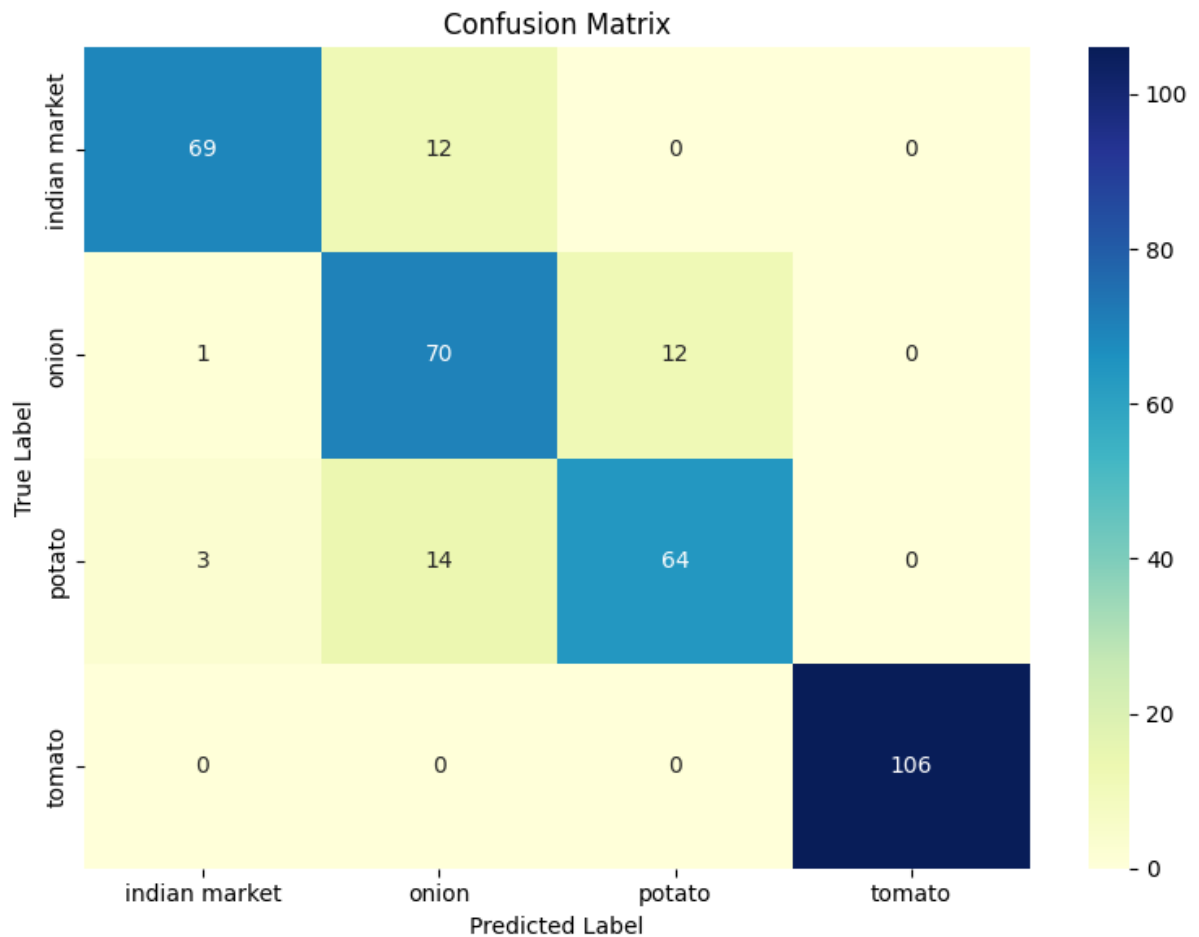
```
In [55]: calculate_testing_accuracy(model, test_ds)
```

11/11 ————— 1s 34ms/step

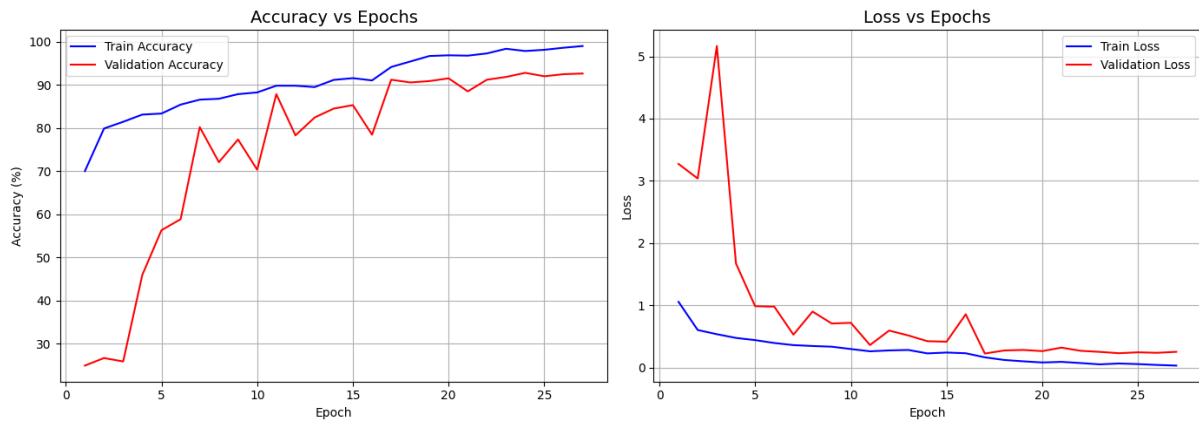
Test Accuracy: 88.03%

```
In [56]: ConfusionMatrix(model, test_ds, test_data.class_names)
```

11/11 ————— 0s 38ms/step



```
In [57]: plot_training_metrics(model_fit)
```



In []:

Accuracy vs Epochs:

Training Accuracy starts around 70%, rising steadily to nearly 100% by epoch 27. Validation Accuracy starts low (~25%), but rapidly improves — reaching above 90%, and then stabilizing around 92–94%.

Interpretation:

Strong model capacity: The network learns the training data very well. **Excellent generalization:** Validation accuracy follows training closely in the latter epochs, with minimal overfitting. Initial instability in early epochs (validation jumps up and down) likely due to: A high learning rate at the start Small or noisy validation set BatchNorm adjusting internal statistics

Loss vs Epochs:

Training Loss steadily decreases from ~1.0 to below 0.1, which is expected. Validation Loss starts high (around 3.0–5.0), but drops quickly and stabilizes around 0.3–0.4 after epoch 10. A few spikes in validation loss (epochs ~3, ~9, ~15), but no long-term divergence.

Interpretation:

Healthy training behavior overall — loss trends align with accuracy curves. Spikes in validation loss indicate occasional high-confidence misclassifications — not uncommon in deeper models with softmax outputs. Low final validation loss suggests the model is not just accurate, but also confident in its correct predictions.

Deeper_CNN model shows excellent performance in both training and generalization. With minimal tuning, it is production-worthy for image classification tasks.

Data pre-processing for models using Pre-trained models

```
In [59]: # Constants
BATCH_SIZE = 128
IMG_SIZE = (224, 224)
VALIDATION_SPLIT = 0.2
SEED = 10

# Set seeds for reproducibility
tf.random.set_seed(SEED)
np.random.seed(SEED)

# Load training and validation datasets
train_ds = tf.keras.utils.image_dataset_from_directory(
    "/Users/Ramv/Downloads/ninjacart_data/train/",
    validation_split=VALIDATION_SPLIT,
    subset="training",
    seed=SEED,
    shuffle=True,
    image_size=IMG_SIZE,
    batch_size=BATCH_SIZE
)

val_ds = tf.keras.utils.image_dataset_from_directory(
    "/Users/Ramv/Downloads/ninjacart_data/train/",
    validation_split=VALIDATION_SPLIT,
    subset="validation",
    seed=SEED,
    shuffle=True,
    image_size=IMG_SIZE,
    batch_size=BATCH_SIZE
)

# Load test dataset (no split)
test_ds = tf.keras.utils.image_dataset_from_directory(
    "/Users/Ramv/Downloads/ninjacart_data/test",
    shuffle=False,
    seed=SEED,
    image_size=IMG_SIZE,
    batch_size=BATCH_SIZE
)

# Define preprocessing (Rescaling for pre-trained models like ResNet,
preprocess_layer = tf.keras.Sequential([
    tf.keras.layers.Rescaling(1./255) # Normalize to [0, 1]
], name="data_preprocess")

# Apply preprocessing to datasets
train_ds = train_ds.map(lambda x, y: (preprocess_layer(x), y), num_parallel_calls=tf.data.experimental.DEFAULT)
val_ds = val_ds.map(lambda x, y: (preprocess_layer(x), y), num_parallel_calls=tf.data.experimental.DEFAULT)
test_ds = test_ds.map(lambda x, y: (preprocess_layer(x), y), num_parallel_calls=tf.data.experimental.DEFAULT)
```

```
# Prefetch for performance
train_ds = train_ds.prefetch(buffer_size=tf.data.AUTOTUNE)
val_ds   = val_ds.prefetch(buffer_size=tf.data.AUTOTUNE)
test_ds  = test_ds.prefetch(buffer_size=tf.data.AUTOTUNE)
```

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.

```
In [64]: from tensorflow.keras.applications import VGG16
        from tensorflow.keras import Model, Sequential
        from tensorflow.keras.layers import Flatten, Dense

        # Load pre-trained VGG16 without top layers
        pretrained_base_model = VGG16(weights='imagenet', include_top=False, i

        # Split layers into two halves
        total_layers = len(pretrained_base_model.layers)
        split_index = total_layers // 2 # halfway

        # First half (to be frozen)
        first_half = Sequential(pretrained_base_model.layers[:split_index], na
        first_half.trainable = False

        # Second half (trainable)
        second_half = Sequential(pretrained_base_model.layers[split_index:], n

        # Final model
        vgg16_model = Sequential([
            first_half,
            second_half,
            Flatten(),
            Dense(256, activation='relu'),      # Optional dense layer for feat
            Dense(4, activation='softmax')      # Final output for 4 classes
        ])
```

```
In [65]: pretrained_base_model.summary()
```

Model: "vgg16"

Layer (type)	Output Shape	
input_layer_19 (InputLayer)	(None, 224, 224, 3)	
block1_conv1 (Conv2D)	(None, 224, 224, 64)	
block1_conv2 (Conv2D)	(None, 224, 224, 64)	
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	
block2_conv1 (Conv2D)	(None, 112, 112, 128)	
block2_conv2 (Conv2D)	(None, 112, 112, 128)	
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	
block3_conv1 (Conv2D)	(None, 56, 56, 256)	
block3_conv2 (Conv2D)	(None, 56, 56, 256)	
block3_conv3 (Conv2D)	(None, 56, 56, 256)	
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	
block4_conv1 (Conv2D)	(None, 28, 28, 512)	
block4_conv2 (Conv2D)	(None, 28, 28, 512)	
block4_conv3 (Conv2D)	(None, 28, 28, 512)	
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	
block5_conv1 (Conv2D)	(None, 14, 14, 512)	
block5_conv2 (Conv2D)	(None, 14, 14, 512)	
block5_conv3 (Conv2D)	(None, 14, 14, 512)	
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	

Total params: 14,714,688 (56.13 MB)

Trainable params: 13,569,280 (51.76 MB)

Non-trainable params: 1,145,408 (4.37 MB)

```
In [66]: vgg16_model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=[
        'accuracy',
        tf.keras.metrics.SparseTopKCategoricalAccuracy(k=5, name='top_
    ]
)
```



```
In [67]: history = vgg16_model.fit(train_ds, epochs=5,
                                   validation_data=val_ds)
```

Epoch 1/5

20/20 ————— **337s** 17s/step – accuracy: 0.2594 – loss: 4.6020 – top_5_accuracy: 1.0000 – val_accuracy: 0.2392 – val_loss: 1.3910 – val_top_5_accuracy: 1.0000

Epoch 2/5

20/20 ————— **1371s** 71s/step – accuracy: 0.2836 – loss: 1.3868 – top_5_accuracy: 1.0000 – val_accuracy: 0.3365 – val_loss: 1.2920 – val_top_5_accuracy: 1.0000

Epoch 3/5

20/20 ————— **1333s** 69s/step – accuracy: 0.3923 – loss: 1.2240 – top_5_accuracy: 1.0000 – val_accuracy: 0.4896 – val_loss: 1.0438 – val_top_5_accuracy: 1.0000

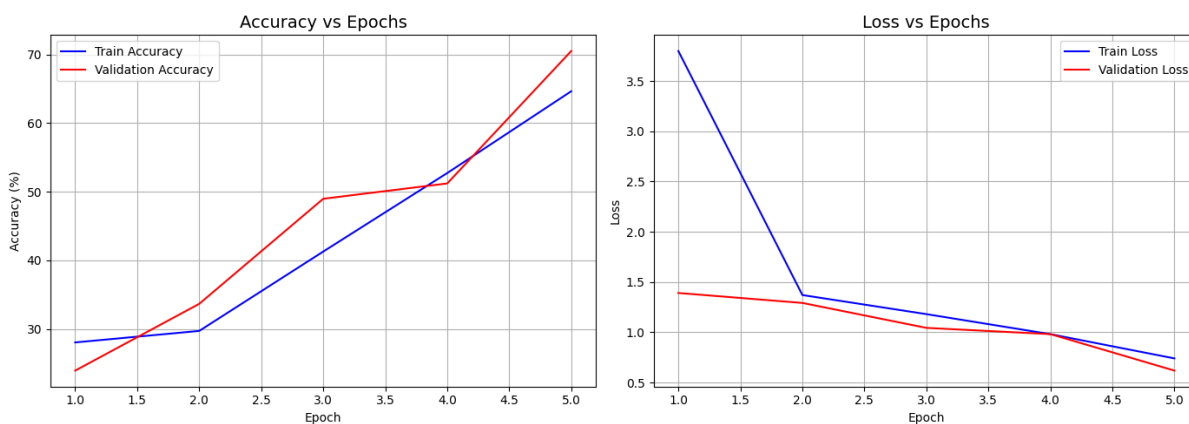
Epoch 4/5

20/20 ————— **5307s** 124s/step – accuracy: 0.5073 – loss: 1.0267 – top_5_accuracy: 1.0000 – val_accuracy: 0.5120 – val_loss: 0.9818 – val_top_5_accuracy: 1.0000

Epoch 5/5

20/20 ————— **409s** 21s/step – accuracy: 0.6038 – loss: 0.8069 – top_5_accuracy: 1.0000 – val_accuracy: 0.7049 – val_loss: 0.6186 – val_top_5_accuracy: 1.0000

```
In [68]: plot_training_metrics(history)
```



Accuracy vs Epochs:

Both training accuracy (blue) and validation accuracy (red) are steadily improving over the 5 epochs. Validation accuracy starts below training, surpasses it by epoch 3, and remains higher.

Interpretation:

The model is learning effectively and generalizing well. The steeper increase in validation accuracy indicates that the model benefits from pretrained features (you're using VGG16), even more so on validation data.

Final Accuracy: ~65% (train), ~71% (validation) — solid early performance, especially for a model with frozen convolutional layers.

Loss vs Epochs:

Both training loss (blue) and validation loss (red) are decreasing, which is a healthy sign. Training loss drops sharply from 3.7 to ~0.8, while validation loss drops from 1.4 to ~0.6.

Interpretation:

No signs of overfitting or underfitting. The pretrained model is being fine-tuned smoothly (likely just classifier layers trained). Loss is decreasing steadily — more training epochs may yield even better performance.

```
In [73]: from sklearn.metrics import accuracy_score, classification_report

# Predict class probabilities
y_pred_probs = vgg16_model.predict(test_ds)

# Get predicted class indices
predicted_categories = tf.argmax(y_pred_probs, axis=1)

# Get true labels
true_categories = tf.concat([y for x, y in test_ds], axis=0)

# Get class names (optional)
class_names = test_ds.class_names if hasattr(test_ds, 'class_names') else None

# Accuracy
test_accuracy = accuracy_score(true_categories, predicted_categories)
print(f"\n✅ Test Accuracy: {test_accuracy:.2f}%")

# Classification report
if class_names:
    print("\n📊 Classification Report:")
    print(classification_report(true_categories, predicted_categories,
                                target_names=class_names))
else:
    print("\n📊 Classification Report:")
    print(classification_report(true_categories, predicted_categories))
```

3/3 ————— 22s 7s/step

✓ Test Accuracy: 70.66%



Classification Report:

	precision	recall	f1-score	support
0	0.80	0.86	0.83	81
1	0.51	0.53	0.52	83
2	0.46	0.38	0.42	81
3	0.94	0.97	0.95	106
accuracy			0.71	351
macro avg	0.68	0.69	0.68	351
weighted avg	0.69	0.71	0.70	351

Fine tuning VGG16 Model

```
In [74]: import tensorflow as tf
from tensorflow.keras import layers, models

# 1. Load VGG16 base model (exclude top dense layers)
vgg_base_tune = tf.keras.applications.VGG16(
    include_top=False,
    weights='imagenet',
    input_shape=(224, 224, 3)
)

# 2. Freeze all layers initially
vgg_base_tune.trainable = False

# 3. Unfreeze the last 2 convolutional blocks (block4 and block5)
# VGG16 layer names: 'block1_conv1', ..., 'block5_conv3'
for layer in vgg_base_tune.layers:
    if 'block5' in layer.name or 'block4' in layer.name:
        layer.trainable = True

# 4. Build your model on top of VGG16 base
model = tf.keras.Sequential([
    vgg_base_tune,
    layers.GlobalAveragePooling2D(),
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(4, activation='softmax') # Adjust 4 to your number o
])

# 5. Compile the model (use a low learning rate)
model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=1e-5),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
```

)

```
In [75]: vgg_base_tune.summary()
```

Model: "vgg16"

Layer (type)	Output Shape	
input_layer_22 (InputLayer)	(None, 224, 224, 3)	
block1_conv1 (Conv2D)	(None, 224, 224, 64)	
block1_conv2 (Conv2D)	(None, 224, 224, 64)	
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	
block2_conv1 (Conv2D)	(None, 112, 112, 128)	
block2_conv2 (Conv2D)	(None, 112, 112, 128)	
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	
block3_conv1 (Conv2D)	(None, 56, 56, 256)	
block3_conv2 (Conv2D)	(None, 56, 56, 256)	
block3_conv3 (Conv2D)	(None, 56, 56, 256)	
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	
block4_conv1 (Conv2D)	(None, 28, 28, 512)	
block4_conv2 (Conv2D)	(None, 28, 28, 512)	
block4_conv3 (Conv2D)	(None, 28, 28, 512)	
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	
block5_conv1 (Conv2D)	(None, 14, 14, 512)	
block5_conv2 (Conv2D)	(None, 14, 14, 512)	
block5_conv3 (Conv2D)	(None, 14, 14, 512)	
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	

Total params: 14,714,688 (56.13 MB)

Trainable params: 0 (0.00 B)

Non-trainable params: 14,714,688 (56.13 MB)

```
In [81]: model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=1e-5),
    loss='sparse_categorical_crossentropy',
```

```
metrics=[
    'accuracy',
    tf.keras.metrics.SparseTopKCategoryicalAccuracy(k=5, name='top_
]
)
```

```
In [82]: history_tune = model.fit(train_ds, epochs=5, validation_data=val_ds)
```

Epoch 1/5

20/20 ————— **308s** 15s/step - accuracy: 0.3381 - loss: 1.4198 - top_5_accuracy: 1.0000 - val_accuracy: 0.7289 - val_loss: 0.8325 - val_top_5_accuracy: 1.0000

Epoch 2/5

20/20 ————— **306s** 15s/step - accuracy: 0.6818 - loss: 0.8013 - top_5_accuracy: 1.0000 - val_accuracy: 0.8485 - val_loss: 0.4510 - val_top_5_accuracy: 1.0000

Epoch 3/5

20/20 ————— **312s** 16s/step - accuracy: 0.7893 - loss: 0.5080 - top_5_accuracy: 1.0000 - val_accuracy: 0.8756 - val_loss: 0.3361 - val_top_5_accuracy: 1.0000

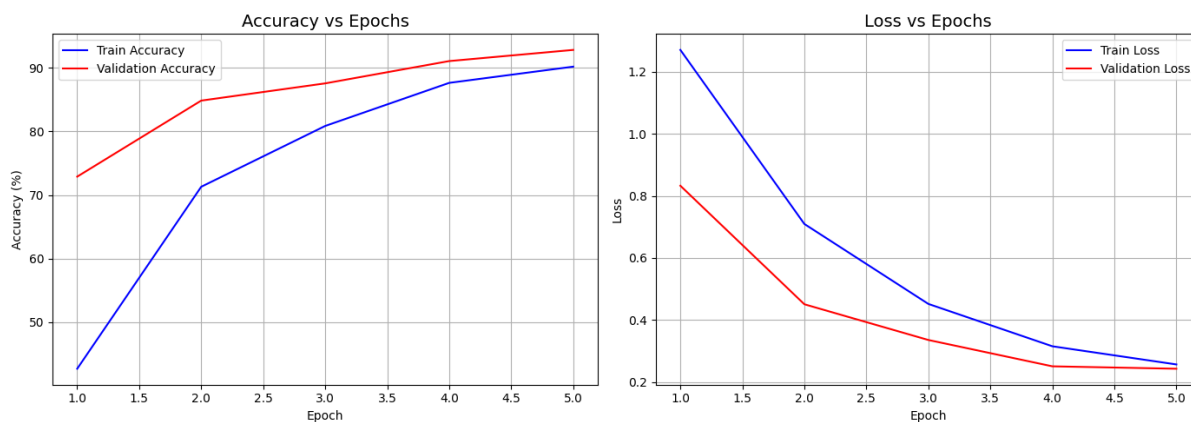
Epoch 4/5

20/20 ————— **432s** 22s/step - accuracy: 0.8686 - loss: 0.3447 - top_5_accuracy: 1.0000 - val_accuracy: 0.9107 - val_loss: 0.2513 - val_top_5_accuracy: 1.0000

Epoch 5/5

20/20 ————— **304s** 15s/step - accuracy: 0.9000 - loss: 0.2607 - top_5_accuracy: 1.0000 - val_accuracy: 0.9282 - val_loss: 0.2435 - val_top_5_accuracy: 1.0000

```
In [83]: plot_training_metrics(history_tune)
```



Accuracy vs Epochs:

Training Accuracy rises steadily from ~43% to ~90%.

Validation Accuracy starts high (~73%) and improves consistently to reach ~93%.

Validation accuracy is higher than training accuracy throughout — this is unusual but not necessarily wrong. Possible reasons: Data augmentation is applied only on training set (increasing difficulty). Validation set may be simpler/less noisy.

Pretrained features from VGG16 generalize very well to validation examples early on.

Inference:

The model is learning effectively with good generalization. No signs of overfitting within these 5 epochs.

Loss vs Epochs:

Both Training Loss and Validation Loss decrease steadily, converging to ~0.25–0.3. Smooth and aligned drop in both losses confirms: The model is not overfitting. Optimization is progressing well. Loss and accuracy are consistent.

Inference:

Fine-tuning VGG16 helped transfer learned features effectively

```
In [85]: from sklearn.metrics import accuracy_score, classification_report

# Predict class probabilities
y_pred_probs = model.predict(test_ds)

# Get predicted class indices
predicted_categories = tf.argmax(y_pred_probs, axis=1)

# Get true labels
true_categories = tf.concat([y for x, y in test_ds], axis=0)

# Get class names (optional)
class_names = test_ds.class_names if hasattr(test_ds, 'class_names') else None


# Accuracy
test_accuracy = accuracy_score(true_categories, predicted_categories)
print(f"\n✅ Test Accuracy: {test_accuracy:.2f}%")

# Classification report
if class_names:
    print("\n📊 Classification Report:")
    print(classification_report(true_categories, predicted_categories,
                                target_names=class_names))
else:
    print("\n📊 Classification Report:")
    print(classification_report(true_categories, predicted_categories))
```

WARNING:tensorflow:5 out of the last 18 calls to <function TensorFlowTrainer.make_predict_function.<locals>.one_step_on_data_distributed at 0x16901cdc0> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the loop. For (2), @tf.function has reduce_retracing=True option that can avoid unnecessary retracing. For (3), please refer to https://www.tensorflow.org/guide/function#controlling_retracing and https://www.tensorflow.org/api_docs/python/tf/function for more details.

3/3  21s 7s/step

✓ Test Accuracy: 85.47%

 Classification Report:

	precision	recall	f1-score	support
0	1.00	0.69	0.82	81
1	0.68	0.81	0.74	83
2	0.80	0.88	0.84	81
3	0.99	1.00	1.00	106
accuracy			0.85	351
macro avg	0.87	0.84	0.85	351
weighted avg	0.87	0.85	0.86	351

Test results show that your fine-tuned VGG16 model is performing very well on unseen data.

Insights:

The model performs best on Class 3 — possibly due to distinctive features or class imbalance. Class 0 and 1 show precision-recall trade-off: Class 0 has perfect precision but misses some positives (lower recall). Class 1 has more false positives (lower precision) but better coverage (higher recall).

```
In [86]: vgg16_model_finetuned = model.save("vgg16_model_finetuned.keras")
```

```
In [94]: def plot_image(pred_array, true_label, img, class_names):
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])

    # Rescale [0,1] -> [0,255] and convert to uint8 for display
    plt.imshow((img * 255).numpy().astype("uint8"))

    predicted_label = np.argmax(pred_array)
    confidence = 100 * np.max(pred_array)
    color = 'blue' if predicted_label == true_label else 'red'
```

```
plt.xlabel(
    f"Pred: {class_names[predicted_label]} ({confidence:.0f}%) \n True: {class_names[true_label]}"
    color='red',
    fontsize=10
)
```

```
In [95]: true_categories = tf.concat([y for x, y in test_ds], axis=0)
images = tf.concat([x for x, y in test_ds], axis=0)
y_pred = model.predict(test_ds)
class_names = test_data.class_names

# Randomly sample 15 test images and plot it with their predicted labels
indices = random.sample(range(len(images)), 15)
# Color correct predictions in blue and incorrect predictions in red.
num_rows = 5
num_cols = 4
num_images = num_rows*num_cols
plt.figure(figsize=(4*num_cols, 2*num_rows))
for i, index in enumerate(indices):
    plt.subplot(num_rows, num_cols, i + 1)
    plot_image(y_pred_probs[index], true_categories[index], images[index])

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

3/3 ————— 22s 7s/step



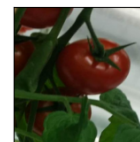
Pred: tomato (100%)
True: tomato



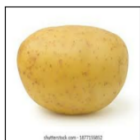
Pred: tomato (100%)
True: tomato



Pred: potato (95%)
True: potato



Pred: tomato (100%)
True: tomato



Pred: potato (90%)
True: potato



Pred: onion (48%)
True: indian market



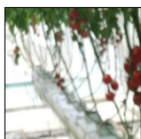
Pred: potato (85%)
True: potato



Pred: tomato (98%)
True: tomato



Pred: potato (99%)
True: onion



Pred: tomato (80%)
True: tomato



Pred: onion (90%)
True: onion



Pred: onion (52%)
True: indian market



Pred: tomato (100%)
True: tomato



Pred: onion (59%)
True: indian market



Pred: potato (73%)
True: onion

Key Observations:

High Precision on Clear-Class Images

The model correctly classified tomato and potato images with high confidence (95%–100%). Examples: "Pred: tomato (100%) / True: tomato" — several instances "Pred: potato (90%) / True: potato" — confidently predicted

Confusion with "Onion" vs "Potato"

Several onion images were predicted as potato (and vice versa). Example: "Pred: potato (99%) / True: onion" → classic misclassification May be due to visual similarity in color or shape

Misclassification of "Indian Market"

Images labeled as "indian market" are frequently misclassified: "Pred: onion (48%) / True: indian market" "Pred: potato (73%) / True: onion"

Reason: These are cluttered scenes or unstructured environments (vs. close-ups of single objects), which likely confuse the model trained mainly on object-focused examples.

Using ResNet pretrained Model

```
In [96]: # Load ResNet50V2 without top layers
resnet_base = tf.keras.applications.ResNet50V2(
    weights='imagenet',
    include_top=False,
    input_shape=(224, 224, 3)
)

# Optionally normalize input (not required if you preprocess separately)
# You can use tf.keras.applications.resnet_v2.preprocess_input on the

# Full model (no freezing or splitting)
resnet_model = models.Sequential([
    resnet_base,
    layers.GlobalAveragePooling2D(),
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(4, activation='softmax') # Adjust number of classes
])
```

Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/resnet/resnet50v2_weights_tf_dim_ordering_tf_kernels_notop.h5

94668760/94668760 ————— **10s** 0us/step

```
In [97]: resnet_model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=1e-4), # Smaller
    loss='sparse_categorical_crossentropy',
    metrics=[
```

```

        'accuracy',
        tf.keras.metrics.SparseTopKCategoryicalAccuracy(k=3, name='top_
    ]
)

```

```
In [98]: history_resnet = resnet_model.fit(train_ds, epochs=5,
      validation_data=val_ds)
```

Epoch 1/5

20/20 ————— **242s** 12s/step - accuracy: 0.6021 - loss: 0.9620 - top_3_accuracy: 0.9249 - val_accuracy: 0.9171 - val_loss: 0.2756 - val_top_3_accuracy: 0.9952

Epoch 2/5

20/20 ————— **228s** 11s/step - accuracy: 0.9748 - loss: 0.0942 - top_3_accuracy: 0.9998 - val_accuracy: 0.9410 - val_loss: 0.2660 - val_top_3_accuracy: 0.9968

Epoch 3/5

20/20 ————— **240s** 12s/step - accuracy: 0.9941 - loss: 0.0287 - top_3_accuracy: 1.0000 - val_accuracy: 0.9506 - val_loss: 0.2320 - val_top_3_accuracy: 0.9984

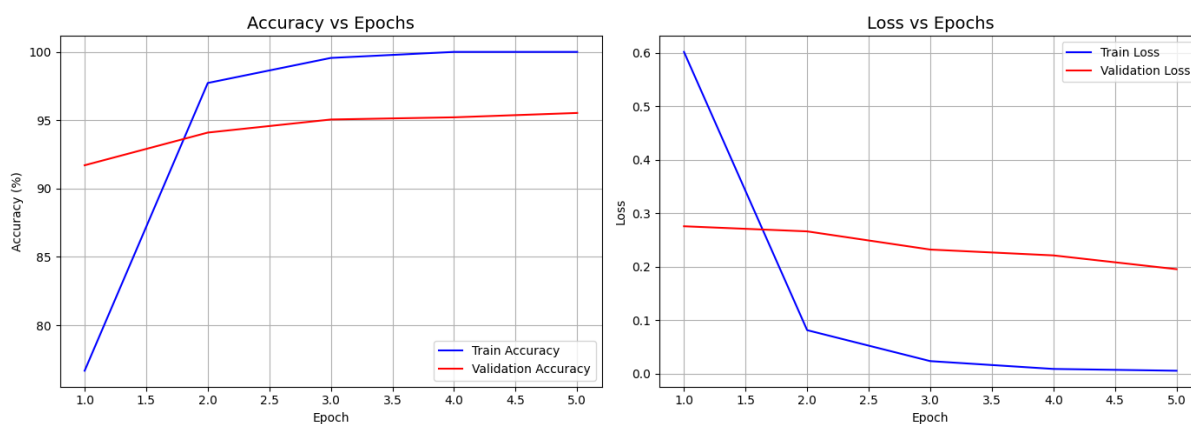
Epoch 4/5

20/20 ————— **224s** 11s/step - accuracy: 1.0000 - loss: 0.0100 - top_3_accuracy: 1.0000 - val_accuracy: 0.9522 - val_loss: 0.2210 - val_top_3_accuracy: 0.9984

Epoch 5/5

20/20 ————— **226s** 11s/step - accuracy: 1.0000 - loss: 0.0057 - top_3_accuracy: 1.0000 - val_accuracy: 0.9553 - val_loss: 0.1953 - val_top_3_accuracy: 0.9984

```
In [99]: plot_training_metrics(history_resnet)
```



In []:

In []:

In []:

```
In [101... true_categories = tf.concat([y for x, y in test_ds], axis=0)
      images = tf.concat([x for x, y in test_ds], axis=0)
```

```

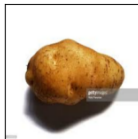
y_pred = model.predict(test_ds)
class_names = test_data.class_names

# Randomly sample 15 test images and plot it with their predicted labels
indices = random.sample(range(len(images)), 15)
# Color correct predictions in blue and incorrect predictions in red.
num_rows = 5
num_cols = 4
num_images = num_rows*num_cols
plt.figure(figsize=(4*num_cols, 2*num_rows))
for i, index in enumerate(indices):
    plt.subplot(num_rows, num_cols, i + 1)
    plot_image(y_pred_probs[index], true_categories[index], images[index])

plt.tight_layout()
plt.show()

```

3/3 ————— 20s 6s/step



Pred: potato (96%)
True: potato



Pred: tomato (100%)
True: tomato



Pred: tomato (99%)
True: tomato



Pred: tomato (100%)
True: tomato



Pred: onion (72%)
True: onion



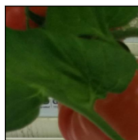
Pred: potato (49%)
True: potato



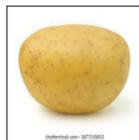
Pred: indian market (55%)
True: indian market



Pred: tomato (100%)
True: tomato



Pred: tomato (97%)
True: tomato



Pred: potato (90%)
True: potato



Pred: onion (62%)
True: potato



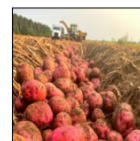
Pred: indian market (91%)
True: indian market



Pred: tomato (100%)
True: tomato



Pred: potato (99%)
True: potato



Pred: potato (77%)
True: potato

Accuracy Plot:

Train Accuracy quickly reaches near 100% by epoch 4. Validation Accuracy improves steadily, peaking around 95.5%, but plateaus afterward.

Insights: The model learns very quickly, thanks to ResNet50V2's strong pretrained weights. No severe overfitting is seen yet, as the gap between train and validation accuracy is moderate and stable.

Loss Plot:

Train Loss drops rapidly and reaches a very low value (~0.01) by epoch 5.

Validation Loss also decreases, but more slowly and stabilizes around 0.2.

Insights: Low training loss = model fits training data very well. Validation loss keeps improving → generalization is good. Train loss becoming extremely low compared to validation loss might signal that the model is starting to memorize the training data.

Observations:

Correct Predictions (Most Cases):

Tomato: Nearly all tomato images (plants, fruits) are classified correctly with ~99–100% confidence.

Potato: Several potato images are predicted correctly, including close-up shots and farm images.

Indian Market: Market scenes are also correctly predicted, even with moderate confidence (~55–91%).

Misclassifications (Few but Notable):

One potato image (with a shovel in soil) is predicted as onion with 62% confidence.

Model seems slightly confused when: Visual background is complex (e.g., farming scenes with cluttered elements). Potato variety is non-standard (red-skinned potatoes).

```
In [103... from sklearn.metrics import accuracy_score, classification_report

# Predict class probabilities
y_pred_probs = resnet_model.predict(test_ds)

# Get predicted class indices
predicted_categories = tf.argmax(y_pred_probs, axis=1)

# Get true labels
true_categories = tf.concat([y for x, y in test_ds], axis=0)

# Get class names (optional)
class_names = test_ds.class_names if hasattr(test_ds, 'class_names') else None

# Accuracy
test_accuracy = accuracy_score(true_categories, predicted_categories)
print(f"\n✅ Test Accuracy: {test_accuracy:.2f}%")

# Classification report
```

```
if class_names:
    print("\n📊 Classification Report:")
    print(classification_report(true_categories, predicted_categories,
else:
    print("\n📊 Classification Report:")
    print(classification_report(true_categories, predicted_categories)
```

3/3 6s 2s/step

✅ Test Accuracy: 92.31%

📊 Classification Report:

	precision	recall	f1-score	support
0	0.99	0.91	0.95	81
1	0.79	0.94	0.86	83
2	0.93	0.81	0.87	81
3	1.00	1.00	1.00	106
accuracy			0.92	351
macro avg	0.93	0.92	0.92	351
weighted avg	0.93	0.92	0.92	351

Overall Performance:

Test Accuracy: 92.31% — the model correctly predicts ~92% of test samples.

Macro Avg F1-score: 0.92 — indicates strong overall balance across classes.

Weighted Avg F1-score: 0.92 — reflects excellent performance considering class imbalance.

Insights:

Class 1 (possibly "onion") shows the most room for improvement, especially in precision — model occasionally mislabels other classes as class 1.

Class 2 has good precision but moderate recall — might benefit from more representative examples or fine-tuning.

Class 3 (possibly "tomato") is perfectly classified.

In []: