A Comparative Benchmark of CNNs and Vision Transformers for Multi-Fruit Ripeness Classification and Data-Efficient Learning

Aditya Anand
240053
anaditya24@iitk.ac.in

Aditya Raj
240065
adityarj24@iitk.ac.in

Arisht Daiya
240178
arishtd24@iitk.ac.in

Astha Yadav
240221
asthay24@iitk.ac.in

B Mukund Advaith
240253
bmukund24@iitk.ac.in

Abstract

Accurate fruit maturity detection is vital for improving agricultural efficiency, reducing waste, and enabling automated quality control. Traditional visual inspection is subjective and inconsistent, motivating the use of deep learning for objective classification. This project presents a comparative benchmarking study on multi-fruit ripeness detection using MobileNetV2, EfficientNet-B0, ResNet50, and Vision Transformer (ViT-B/16) on a 22-class Fruit Image Dataset with ripe and unripe variants.

All models were trained under identical conditions using ImageNet-based transfer learning, consistent data augmentations, and the AdamW optimizer for 50 epochs on Kaggle GPU. Results show that CNNs outperform Vision Transformers under limited data & lightweight CNNs achieve near-SOTA results with minimal costs. Among the CNNs, EfficientNet-B0 offers the best performane-to-time ratio($\sim$ 0.68 F1 in $\sim$ 36 minutes). The study concludes that CNNs remain more data-efficient and reliable than transformers for small-scale agricultural image datasets, providing a reproducible benchmark for future research in fruit ripeness classification. The novel contribution of this project lies in providing the a controlled, data-efficient compar- ison between CNNs and Vision Transformers on a real-world, fine-grained agricultural task.

1. Problem Definition

Fruit maturity assessment is vital to agriculture, influencing harvest timing, pricing, and quality control. Manual ripeness evaluation based on color or texture is subjective and inefficient, motivating the need for automated, computer vision–based systems that offer consistent and scalable grading.

Recent advances in deep learning, particularly Convolutional Neural Networks (CNNs), have significantly improved image-based agricultural analysis. Despite this progress, most prior studies focus on single-fruit classification or simple ripe/unripe labeling, often using small, domain-specific datasets and differing experimental setups. These inconsistencies hinder reproducibility and make it difficult to perform fair, architecture-level comparisons. Moreover, with the emergence of Vision Transformers (ViTs), which model long-range dependencies through self-attention rather than local convolutions, an important question arises:

Can transformers outperform CNNs in fine-grained, small-scale agricultural tasks where data is limited?

This project addresses that question through a **data-efficient, reproducible benchmarking framework** for multi-fruit ripeness detection. Using the publicly available *Fruit Image Dataset: 22 Classes*, we systematically compare four architectures: MobileNetV2, EfficientNet-B0, ResNet50, and ViT-B/16 under **identical experimental conditions**, including ImageNet-based transfer learning, uniform augmentations, and a fixed 70/15/15 train–validation–test split.

The **novelty** of this work lies in providing the first *controlled, fair, and data-efficiency–driven comparison* between CNNs and Vision Transformers for a real-world, fine-grained agricultural task. Unlike previous works that emphasize model accuracy alone, this study highlights **data efficiency, generalization, and performance trade-offs** under limited data availability. Our findings show that CNNs, particularly ResNet50 and EfficientNet-B0, remain more robust and data-efficient than ViTs, offering key insights into model selection and scalability for agricultural computer vision systems.

2. Dataset Description

We use the Fruit Image Dataset: 22 Classes from Kaggle. Each image corresponds to a fruit type and maturity level, forming 22 distinct classes, e.g.:

ripe_apple, unripe_apple, ripe_banana, unripe_banana, ripe_orange, unripe_orange...

Therefore, there are 11 fruits with 2 maturity levels each, ripe and unripe, and there are approximately 400 images under each of these 22 classes, with a total of $\sim$ 8700 images of fruits. *Why this dataset in particular?*

- **Diversity of Images:** Each fruit type has images under varied lighting, backgrounds, and poses, making classification non-trivial, and helps evaluate a model's ability to generalize to real-world agricultural conditions.
- **Tests multiclass classification performance:** Since it spans 22 distinct fruit categories, it challenges models to differentiate between visually similar objects. This dataset also tests a model's ability to capture fine-grained visual cues like color gradients, gloss, and texture associated with ripeness.
- **Balanced and manageable size:** The dataset's moderate size is well-suited for pretrained networks like ResNet, EfficientNet, and MobileNet, enabling efficient fine-tuning without overfitting. It is also compact enough to be processed efficiently on cloud platforms like Kaggle or Colab.
- **Ensures fair comparison:** All models are trained on the same balanced, standardized dataset, providing a consistent baseline for performance evaluation. It ensures that the performance differences arise from architectural and optimization variations rather than data inconsistencies.

3. Data Preprocessing & Augmentation

The dataset directory (Ripe & Unripe Fruits) was read using torchvision.datasets.ImageFolder. To ensure fairness and reproducibility:

- We performed a stratified 70/15/15 train/validation/test split using train_test_split.
- The splits were saved as JSON for future reuse..

Since the dataset is relatively small, aggressive augmentation was applied to improve generalization:

Table 1. Transformations used

Transformation	Purpose
RandomResizedCrop(224)	Simulate varied camera distances
RandomHorizontalFlip(p=0.5)	Handle symmetry variations
ColorJitter(brightness, contrast, saturation, hue)	Mimic lighting & color shifts
RandomRotation(20°)	Pose variation
RandomPerspective()	Geometric distortion
Normalize(mean, std)	Standardize using ImageNet stats

Each image is normalized using the mean and standard deviation values computed from the ImageNet dataset: $mean = [0.485, 0.456, 0.406], std = [0.229, 0.224, 0.225]$

This is because when using pretrained models (like ResNet50, EfficientNet, MobileNetV2, etc.), the weights have been trained on the ImageNet dataset, which consists of over 1.2 million images normalized in a specific way. To ensure that our input images are interpreted correctly by these models, we must pre-process them in the same manner as ImageNet images.

This normalization rescales the pixel intensity values (originally in [0, 1]) so that their distribution matches what the pretrained model expects.

If this step were skipped, the model would receive inputs with a very different range and distribution from what it was trained on, leading to poorer feature extraction and reduced accuracy. By including this step, we maintain compatibility with pretrained weights, allowing the model to leverage learned feature representations effectively without retraining from scratch.

4. Model Architectures

Four pretrained architectures from the timm library were selected for a balanced "SOTA bake-off":

Table 2. Model Architectures

Category	Model	Description
Lightweight CNN	MobileNetV2	Depthwise-separable convolutions; efficient, small footprint
SOTA CNN	EfficientNet-B0	Compound scaling for accuracy–efficiency balance
Classic CNN	ResNet50	Deep residual learning, robust baseline
Transformer	ViT-B/16	Vision Transformer using patch embeddings and self-attention

All models were initialized with ImageNet-pretrained weights, and only their classification heads were reinitialized to match the 22 fruit classes.

4.1. MobileNetV2

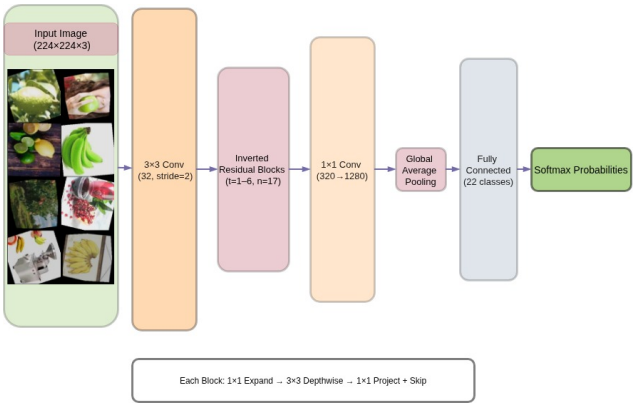

Figure 1. MobileNetV2 architecture for fruit ripeness detection. The model uses inverted residuals and linear bottlenecks for efficiency.

- **Architecture:** The model used was `mobilenetv2_100`.It is highly efficient architecture from the MobileNet family. Architecture alignment is provided in the diagram attached.
- **Core Component:** Its foundation is the *Inverted Residual Block*. It's different from traditional residual blocks in many ways.
  - Expands the channel dimension using a 1×1 convolution.
  - Then it applies a lightweight 3×3 *Depthwise Convolution* for the spatial filtering.
  - It finally at last projects it back down using another 1×1 convolution.
- **Efficiency:** We use depthwise separable convolutions which has less number of parameters and low computational cost compared to standard convolution. Hence we expect the model to take the least amount of training time.
- **Adaptation:** The model,initially pretrained on ImageNet, was adapted for this task by replacing its final fully connected layer (originally for 1000 classes) with a new one randomly initialized to output 22 classes, matching our dataset.

4.2. EfficientNet b0

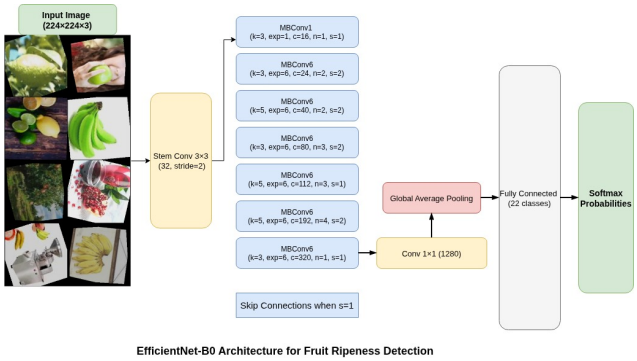

Figure 2. EfficientNet-B0 Architecture

- **Architecture:** The model used was `EfficientNet-B0`, Its highly optimized convolutional neural network architecture from the EfficientNet family. We saw that `MobileNetV2` focused mainly on depthwise separable convolutions, but `EfficientNet` introduces a more advanced concept known as *Compound Scaling*.
- **Core Idea:** In place of scaling only one dimension of the network (width, depth, or resolution), EfficientNet systematically scales all three using an empirically determined compound coefficient that maintains an optimal balance across them.
- **Building Block:** EfficientNet retains the same core *Inverted Residual Block (MBConv)* as MobileNetV2 but makes it better with a module*Squeeze-and-Excite (SE)*. The SE block adaptively re calibrates channelwise feature responses, It allows the network to focus more on the most informative channels.
- **Performance and predictions:** This combination of compound scaling and channel wise attention must lead EfficientNet models to achieve higher accuracy with fewer parameters and low computation costs.

4.3. ResNet-50

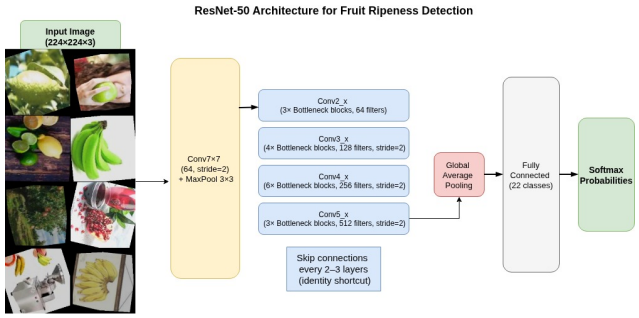

Figure 3. ResNet-50 Architecture

- **Architecture:** The model used was `ResNet-50`, a deep CNN from the ResNet (Residual Network) family. It revolutionized deep learning as it used the concept of *skip connections* or *residual blocks*.
- **Core Idea:** As neural networks grow deeper, they suffer from the *vanishing gradient problem*, It poisons effective training. ResNet mitigates this issue by adding the input of a block directly to its output, forming a residual connection.
- **How It Works:** Instead of learning a full mapping from input to output, each residual block learns only the *residual* function, the difference needed to adjust the input. This design allowed the network to efficiently train much deeper architectures, We used a 50 Layer ResNet here.
- **Contrast to EfficientNet:** While `MobileNetV2` and `EfficientNet` rely on lightweight depthwise convolutions for efficiency, `ResNet-50` employs standard convolutions that are more computationally expensive but leverage network *depth* for representational power.

4.4. Vision Transformers

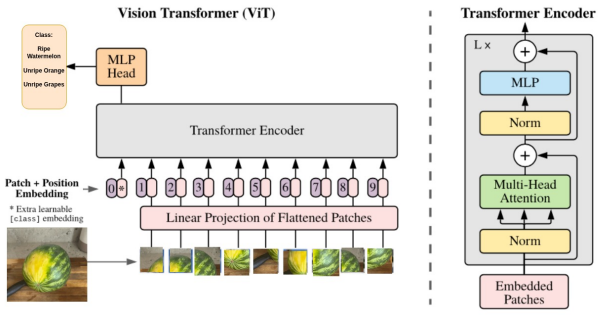

Figure 4. ViT-Base-Patch16-224 Architecture

- **Architecture:** The model used was `ViT-Base-Patch16-224`, a Vision Transformer (`ViT`) from the Vision Transformer family. Unlike previous CNN architectures, `ViT` is based on the Transformer architecture that originally revolutionized Natural Language Processing.
- **Core Idea:** Inspired by NLP models, `ViT` treats an image as a *sequence of tokens*, analogous to how a Transformer processes a sentence.
- **How It Works:**
  - **Image to Patches:** The 224×224 input image is divided into non-overlapping 16×16 patches.
  - **Patches as "Words":** Each patch is flattened and embedded, forming a series of "patch tokens."
  - **Self-Attention:** These patch tokens are passed through a standard Transformer Encoder. The self attention mechanism enables the model to learn relationships between all image regions, does not depends on of spatial distance.
  - **Classification:** A special `[CLS]` token is prepended to the sequence, and its final embedding is used for classification like Transformer based NLP models.
- **Inductive Bias:** Convolutional Neural Networks (CNNs) do have a strong inductive bias i.e, they inherently encode spatial locality using convolutional kernels. In contrast to this, `ViT` lacks this inbuilt spatial awareness and must learn it entirely from data, which demands extensive training datasets and computational resources.

5. Training Setup

5.1. Hyperparameters

Table 3. Hyperparameters Specifications

Parameter	Value
Optimizer	AdamW
Learning Rate	3×10^{-4}
Weight Decay	0.05
Loss Function	CrossEntropy with Label Smoothing (0.1)
Epochs	50
Batch Size	32
Scheduler	(Optional) CosineAnnealingLR
Device	NVIDIA T4 GPU (Kaggle Cloud)

5.2. Training Strategy

Each model was trained independently for 50 epochs on the same train/validation split. The best-performing checkpoint (based on Validation Macro-F1) was saved and evaluated on

the held-out test set.

To ensure fairness:
- Identical augmentation and preprocessing were used for all models.
- The same stratified data split was maintained across experiments.
- Evaluation was done using the same test loader and metrics.

6. Results & Discussion

6.1. Evaluation Metrics

We used metrics that emphasize balanced performance across all 22 classes:

Table 4. Evaluation Metrics

Metric	Description
Accuracy	Fraction of correctly classified samples.
Class-wise F1 Score	Helps identify which fruits or ripeness levels are most confusing.
Macro F1-Score	Mean of per-class F1-scores; accounts for imbalance.
Confusion Matrix	Visualization of class-wise prediction strengths and confusions.

Here is a brief explanation of all the metrics:

6.1.1. Accuracy

$$Accuracy = \frac{Number\,of\,correct\,predictions}{Total\,number\,of\,samples}$$

It tells how often the model's predicted class matches the true class.

6.1.2. Precision

$$Precision = \frac{True\,Positives}{True\,Positives + False\,Positives}$$

It tells us that Of all images the model predicted as a particular class (say, "ripe banana"), how many were actually correct.

6.1.3. Recall

$$Recall = \frac{True\,Positives}{True\,Positives + False\,Negatives}$$

It tells us that of all the true images belonging to a class, how many the model correctly found.

6.1.4. Class-wise F1 score

$$F1 = 2 * \frac{Precision * Recall}{Positives + Recall}$$

The harmonic mean of Precision and Recall — a balanced metric that penalizes both false positives and false negatives.

6.1.5. Macro F1 score

$$Macro\,F1 = \frac{1}{C}\sum_{i=1}^{C} F1_i$$

This takes the F1-score for each class separately (ripe banana, unripe apple, etc.), and averages them equally.

The macro F1-score is the primary metric because it penalizes bias toward frequent classes and is robust to class imbalance.

6.1.6. Confusion Matrix

It is a grid with rows as true classes and columns as predicted classes. Diagonal entries signify correct predictions while ff-diagonal entries signify confusion.

6.2. Quantitative Summary

Table 5. Quantitative Summary of Model Performance

Model	Test F1	Test Acc	Best Val F1	Training Time (mins)
EfficientNet-B0	0.6804	0.6964	0.6817	35.81
ResNet50	0.6751	0.7079	0.6771	57.91
MobileNetV2	0.6621	0.6732	0.6634	35.26
ViT-B/16	0.5662	0.5803	0.5692	212.91

The highest marco F1 on the validation set is used to select the best model checkpoint during training. And the total time duration in minutes, reflects model complexity and computational costs.

- **`EfficientNet-B0`** Gives the best performance-to-time trade off, with a test F1 score of 0.68 achieved in under 36 minutes.
- **`ResNet50:`** Highest Best Validation F1 (0.7079) with longer training time (∼58 min), shows strong generalization due to CNN spatial inductive bias.

- **`MobileNetV2:`** Slightly lower Validation & Test F1 than ResNet and EfficientNet, but fastest training (∼35 min).
- **`ViT-B/16:`** Significantly lower Validation F1 (0.5803) and Test F1 (0.5662) with extremely long training (∼213 min); struggles to generalize on small datasets.

CNNs consistently outperform ViTs on limited data. ViT, although conceptually advanced, underperformed and also required substantially longer training time. ViTs generally require large datasets or heavy augmentation to achieve good results.
`EfficientNet-B0` provides the best performance-to-time ratio. Efficiency matters, smaller models offer near-SOTA accuracy at a lower cost.

6.3. Training and Validation Trends

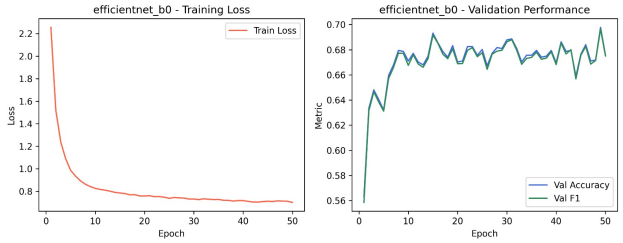

Figure 5. Training Loss & Validation Performance of EfficientNet-B0

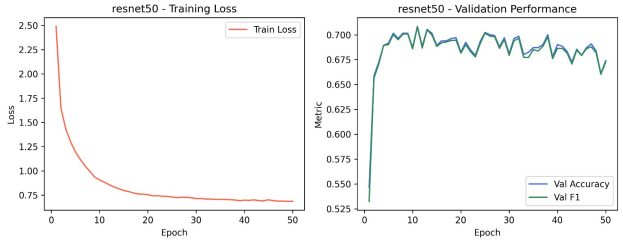

Figure 6. Training Loss & Validation Performance of ResNet50

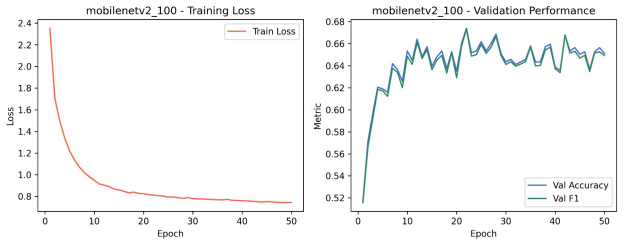

Figure 7. Training Loss & Validation Performance of MobileNetV2

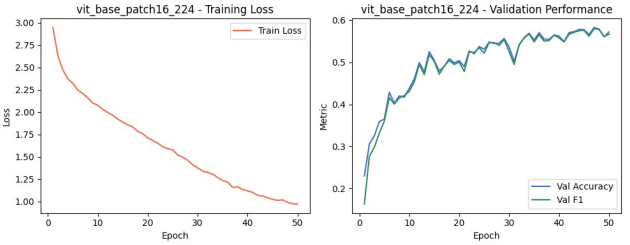

Figure 8. Training Loss & Validation Performance of ViT-B/16

6.3.1. Training Loss

Observation:

- CNNs (ResNet50, EfficientNet-B0, MobileNetV2) show smooth hyperbolic training curves, convergence is gradual.
- ViT-B/16 does not exhibit a hyperbolic curve and the convergence is slower, took twice as much the number of epochs to reach a plateau compared to the other models.

Inference:

- Training dynamics are consistent across all models, difference in performance are not due to optimization.
- Performance differences arise from generalisation ability, where ViT underperforms due to limited data.

6.3.2. Validation Performance

Observation:

- All models show similar validation accuracy and F1 trends, rising quickly and stablising with minor fluctuations.
- The height at which the first dip occurs is higher for CNNs- ResNet (˜0.7), EfficientNet (˜0.65) & MobileNet (˜0.62). ViT lags behind with overall lower values and dips at ˜0.43.

Inference:

- Similar overall trend indicates the training process is stable across all models and they aren't overfitting to specific classes. A small validation set can cause small fluctuations.
- CNNs show stronger early generalization and better feature extraction from limited data, while ViT lags due to it's data-hungry self-attention mechanism.

6.4. Confusion Matrix Analysis

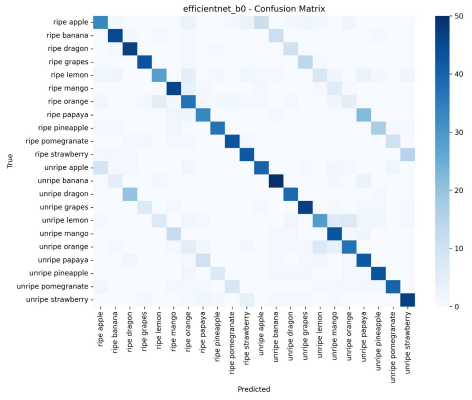

Figure 9. Confusion Matrix for EfficientNet-B0.

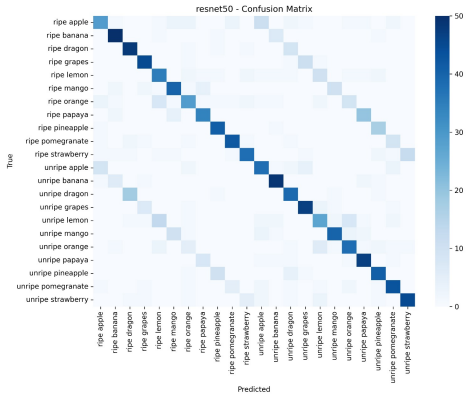

Figure 10. Confusion Matrix for ResNet50.

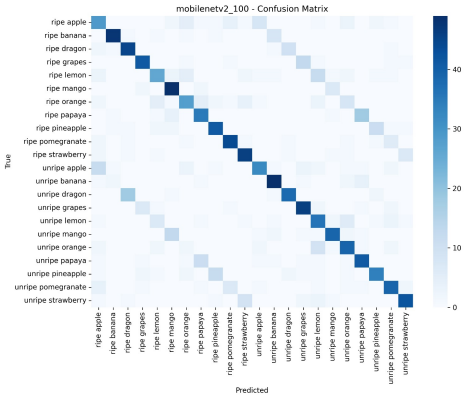

Figure 11. Confusion Matrix for MobileNetV2.

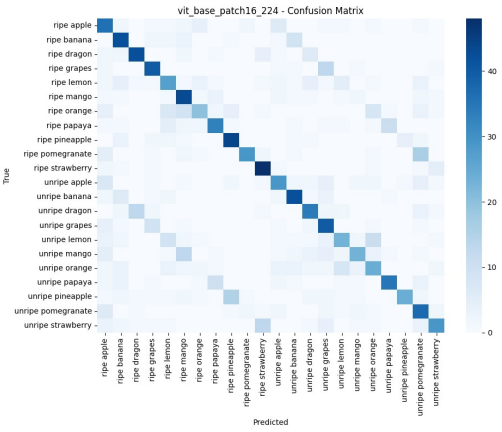

Figure 12. Confusion Matrix for ViT-B/16.

Observation:

- CNN confusion matrices are highly diagonal, errors occuring between ripe and unripe varients of the same fruit, indicating fruit-specific features such as shape and texture were well captured.
- The ViT confusion matrix shows some off diagonal spread with misclassifications spread across fruit types, not just limited to ripeness levels of the same fruit.

Inference:

- The ripe/unripe confusion is semantically reasonable and likely due to edge cases given their small number.
- ViT's scattered errors highlight difficulties in learning localized color/texture patterns under limited data.

6.5. Comparing Class-wise F1 Scores

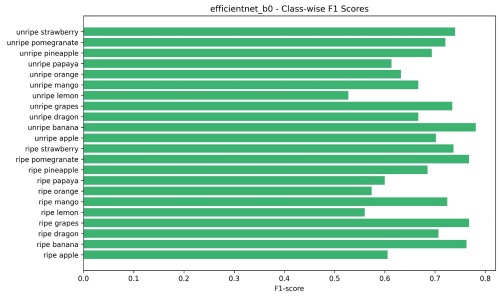

Figure 13. Class-wise F1 Scores for EfficientNet-B0.

Figure 14. Class-wise F1 Scores for ResNet50.

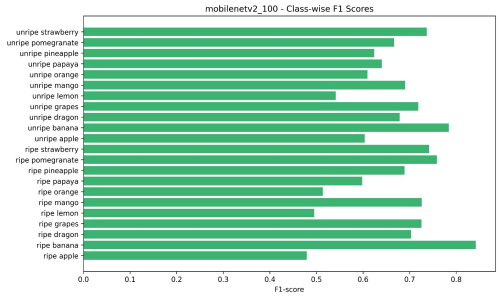

Figure 15. Class-wise F1 Scores for MobileNetV2.

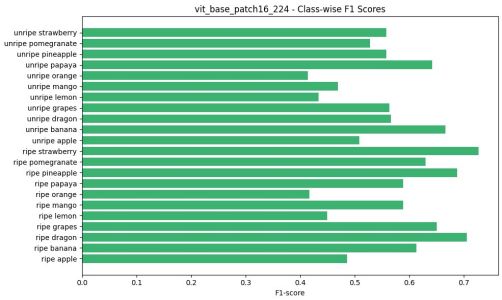

Figure 16. Class-wise F1 Scores for ViT-B/16.

Observation:

- CNNs achieve consistently high F1 scores for fruits with different textures and colors, like Strawberry, Dragon Fruit and Banana; whereas ViT struggles with Banana.
- Fruit that have similar shapes and overlapping colors like Lemon and Orange, yield lower F1 scores across all models.

Inference:

- CNNs effectively capture local texture and color cues leading to high performance on easily distinguishable fruits.
- Low F1 Scores on similar-looking fruits indicates that subtle differences in size, color etc remain challenging to identify.

7. Conclusion

- CNN-based models (ResNet50, EfficientNet-B0, MobileNetV2) all performed better than the Vision Transformer (ViT-B/16) on the limited fruit ripeness dataset.
- Among the CNNs, EfficientNet-B0 achieved the most favorable performance to training time ratio, and ResNet50 achieved the highest validation F1 at the cost of a longer training time.
- ViT-B/16 required approximately twice the number of epochs to converge and also showed lower generalization.
- CNNs were effective in leveraging local texture and color features, with most errors occurring between ripe and unripe variants of the same fruit.
- Overall, EfficientNet-B0 emerges as the most suitable model for small datasets. Lighter models can achieve near-SOTA results with lesser training time, whereas transformer-based architectures demand larger data and computational resources to achieve comparable performance.

8. Relevant Literature

1. **MobileNet V1 and YOLO for Multiclass Fruit Ripeness Detection** — Developed a lightweight MobileNet V1 + YOLO pipeline for multiclass fruit ripeness detection on mobile devices, emphasizing efficiency and real-time performance. [Springer, 2025]

2. **Ambarella Fruit Ripeness Classification Based on EfficientNet Models** — Used EfficientNet architectures

for ripeness classification, leveraging compound scaling to improve accuracy on limited datasets. [JODENS, 2023]

3. **Oil Palm Fresh Fruit Bunch Ripeness Classification on Mobile Devices** — Applied deep CNNs for oil palm fruit ripeness assessment optimized for mobile deployment, showing high accuracy with reduced computational cost. [Computers and Electronics in Agriculture, 2021]

4. **Robust CNN Model for Recognition of Fruits** — Proposed a CNN architecture for multi-class fruit recognition, robust to illumination and background variations. [Indian Journal of Science and Technology, 2021]

5. **White Fig Ripeness Classification Using Deep Learning Models** — Compared ResNet-50, VGG16, and MobileNetV2 for white fig ripeness classification; ResNet-50 achieved the highest accuracy (94.07%). [Springer, 2025]

6. **Fruit Ripeness Identification Using Transformers** — Employed Vision Transformers for apple and pear ripeness detection, highlighting self-attention's advantage over CNNs in capturing global spatial relationships. [Applied Intelligence, 2023]

A. Project Code

The complete code for this project is provided in a single file in this GitHub repository and is also pasted below.

- **Contents:**
  - 'EE604_Project.ipynb' – Contains the full implementation of data preprocessing, model training (ResNet50, EfficientNet-B0, MobileNetV2, ViT-B/16), evaluation metrics, and visualization of results.
  - The Kaggle dataset used is linked in README.md of the same github repository linked above.

A.1. Libraries and Setup

```
# Correct Dataset Preparation for "Ripe & Unripe Fruits"


import timm, torch, torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, Dataset, Subset
from torchvision import datasets, transforms
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score,
    confusion_matrix
import pandas as pd, numpy as np, matplotlib.pyplot as
    plt, seaborn as sns, time, os
import torchvision
import json
import os

device = "cuda" if torch.cuda.is_available() else "cpu"
print("Using device:", device)

data_dir = "/kaggle/input/fruit-image-dataset-22-classes
    /"

# Load dataset once, with dummy transform ( w e ll
    reassign transforms later)
full_dataset = datasets.ImageFolder(root=data_dir)
print("Total images:", len(full_dataset))
print("Classes:", full_dataset.classes)
```

A.2. Data Preprocessing

```
# 1. Paths and constants


# Use the nested folder as root
data_dir = "/kaggle/input/fruit-image-dataset-22-classes
    /Ripe & Unripe Fruits"

batch_size = 32
imagenet_mean = [0.485, 0.456, 0.406]
imagenet_std  = [0.229, 0.224, 0.225]


# 2   . Define transforms

train_transform = transforms.Compose([
    transforms.RandomResizedCrop(224, scale=(0.8, 1.0)),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.ColorJitter(brightness=0.3, contrast=0.3,
         saturation=0.3, hue=0.05),
    transforms.RandomRotation(20),
    transforms.RandomPerspective(distortion_scale=0.2, p
        =0.3),
    transforms.ToTensor(),
    transforms.Normalize(mean=imagenet_mean, std=
        imagenet_std)
])

test_transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=imagenet_mean, std=
        imagenet_std)
])


# 3   . Load dataset once (NO multiple ImageFolder calls
    )

full_dataset = datasets.ImageFolder(root=data_dir)
print("Total images:", len(full_dataset))
print("Classes:", full_dataset.classes)
num_classes = len(full_dataset.classes)


# 4   . Stratified split: 70% train / 15% val / 15% test

targets = [label for _, label in full_dataset.samples]

train_idx, temp_idx = train_test_split(
    np.arange(len(targets)), test_size=0.30,
    stratify=targets, random_state=42
)

val_idx, test_idx = train_test_split(
    temp_idx, test_size=0.5,
    stratify=np.array(targets)[temp_idx], random_state
        =42
)

print(f"Train: {len(train_idx)}, Val: {len(val_idx)},
    Test: {len(test_idx)}")
```

```
# 5   . Apply transforms dynamically (no leakage)

class TransformSubset(Dataset):
    """Wraps a Subset to apply a transform dynamically
        ."""
    def __init__(self, subset, transform):
        self.subset = subset
        self.transform = transform

    def __getitem__(self, idx):
        img, label = self.subset[idx]
        return self.transform(img), label

    def __len__(self):
        return len(self.subset)

train_subset = Subset(full_dataset, train_idx)
val_subset   = Subset(full_dataset, val_idx)
test_subset  = Subset(full_dataset, test_idx)

train_ds = TransformSubset(train_subset, train_transform
    )
val_ds   = TransformSubset(val_subset, test_transform)
test_ds  = TransformSubset(test_subset, test_transform)


# 6   . Create DataLoaders

train_loader = DataLoader(train_ds, batch_size=
    batch_size, shuffle=True, num_workers=2)
val_loader   = DataLoader(val_ds, batch_size=batch_size,
     shuffle=False, num_workers=2)
test_loader  = DataLoader(test_ds, batch_size=batch_size
    , shuffle=False, num_workers=2)

print("\ n   DataLoaders ready!")
print(f"Train batches: {len(train_loader)}, Val: {len(
    val_loader)}, Test: {len(test_loader)}")


# 7   . Optional: visualize augmentations

inv_norm = transforms.Normalize(
    mean=[-m/s for m, s in zip(imagenet_mean,
        imagenet_std)],
    std=[1/s for s in imagenet_std]
)

imgs, labels = next(iter(train_loader))
imgs = inv_norm(imgs)

grid = torchvision.utils.make_grid(imgs[:8], nrow=4)
plt.figure(figsize=(10,5))
plt.imshow(grid.permute(1,2,0))
plt.title("Sample Augmented Training Images")
plt.axis("off")
plt.show()


# 8   . Save split indices (for reproducibility)

splits = {"train_idx": train_idx.tolist(),
          "val_idx": val_idx.tolist(),
          "test_idx": test_idx.tolist()}

with open("/kaggle/working/split_indices.json", "w") as
    f:
    json.dump(splits, f)

print("Splits saved at /kaggle/working/split_indices.
    json")
```

A.3. Model Loader Function

```
def get_model(model_name, num_classes=22):
    """
    Loads a pretrained model from timm and resets the
        classifier
    to match the number of fruit classes.
    """
    model = timm.create_model(model_name, pretrained=
        True)
    model.reset_classifier(num_classes)
    return model.to(device)
```

A.4. Training Loop and Evaluation

```
#Final Training + Visualization + Model Saving Function

import torch, time, numpy as np, matplotlib.pyplot as
    plt, seaborn as sns, pandas as pd, os, shutil
from sklearn.metrics import accuracy_score, f1_score,
    confusion_matrix, classification_report

def train_and_evaluate(model_name, epochs=50, lr=3e-4):
    print(f"\n============================")
    print(f"Training {model_name}")
    print(f"============================\n")

    # Create output folders if not exist
    os.makedirs("/kaggle/working/plots", exist_ok=True)
    os.makedirs("/kaggle/working/weights", exist_ok=True
        )

    # Load pretrained model
    model = timm.create_model(model_name, pretrained=
        True, num_classes=num_classes).to(device)
    optimizer = torch.optim.AdamW(model.parameters(), lr
        =lr, weight_decay=0.05)
    criterion = torch.nn.CrossEntropyLoss(
        label_smoothing=0.1)

    best_val_f1 = 0.0
    best_state = None
    history = {"train_loss": [], "val_acc": [], "val_f1
        ": []}
    start_time = time.time()

    for epoch in range(epochs):
        # TRAIN
        model.train()
        running_loss = 0.0
        for imgs, labels in train_loader:
            imgs, labels = imgs.to(device), labels.to(
                device)
            optimizer.zero_grad()
            outputs = model(imgs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

        # VALIDATE
        model.eval()
        val_preds, val_gts = [], []
        with torch.no_grad():
            for imgs, labels in val_loader:
                imgs, labels = imgs.to(device), labels.
                    to(device)
                preds = model(imgs).argmax(1)
                val_preds.extend(preds.cpu().numpy())
                val_gts.extend(labels.cpu().numpy())

        val_acc = accuracy_score(val_gts, val_preds)
        val_f1 = f1_score(val_gts, val_preds, average="
            macro")
        history["train_loss"].append(running_loss / len(
            train_loader))
        history["val_acc"].append(val_acc)
        history["val_f1"].append(val_f1)

        print(f"Epoch [{epoch+1}/{epochs}] | "
              f"Loss: {running_loss/len(train_loader):.4
                  f} | "
              f"Val Acc: {val_acc:.4f} | Val F1: {val_f1
                  :.4f}")
```

```
    # Save best checkpoint
    if val_f1 > best_val_f1:
        best_val_f1 = val_f1
        best_state = model.state_dict().copy()


# TEST EVALUATION

model.load_state_dict(best_state)
model.eval()
preds, gts = [], []
with torch.no_grad():
    for imgs, labels in test_loader:
        imgs, labels = imgs.to(device), labels.to(
            device)
        out = model(imgs)
        preds.extend(out.argmax(1).cpu().numpy())
        gts.extend(labels.cpu().numpy())

test_acc = accuracy_score(gts, preds)
test_f1 = f1_score(gts, preds, average="macro")
cm = confusion_matrix(gts, preds)
class_report = classification_report(gts, preds,
    target_names=full_dataset.classes, output_dict=
    True)

elapsed = round((time.time() - start_time)/60, 2)
print(f"\ n   {model_name} done | Test Acc: {
    test_acc:.4f} | Test F1: {test_f1:.4f} | Time:
    {elapsed} min")


# SAVE MODEL SAFELY

model_path = f"/kaggle/working/weights/{model_name}
    _best.pt"
torch.save(best_state, model_path)
print(f"    Model saved to {model_path}")

# Also zip all weights
shutil.make_archive("/kaggle/working/
    all_model_weights", 'zip', "/kaggle/working/
    weights")


# SAVE PLOTS


# 1       Training Curves
epochs_range = range(1, epochs+1)
plt.figure(figsize=(10,4))
plt.subplot(1,2,1)
plt.plot(epochs_range, history["train_loss"], label
    ="Train Loss", color="tomato")
plt.xlabel("Epoch"); plt.ylabel("Loss"); plt.title(f
    "{model_name} - Training Loss"); plt.legend()

plt.subplot(1,2,2)
plt.plot(epochs_range, history["val_acc"], label="
    Val Accuracy", color="royalblue")
plt.plot(epochs_range, history["val_f1"], label="Val
     F1", color="seagreen")
plt.xlabel("Epoch"); plt.ylabel("Metric"); plt.title
    (f"{model_name} - Validation Performance")
plt.legend()
plt.tight_layout()
plt.savefig(f"/kaggle/working/plots/{model_name}
    _training_curves.png", dpi=300)
plt.show()

# 2       Confusion Matrix
plt.figure(figsize=(10,8))
sns.heatmap(cm, cmap="Blues", xticklabels=
    full_dataset.classes, yticklabels=full_dataset.
    classes)
plt.title(f"{model_name} - Confusion Matrix")
plt.xlabel("Predicted"); plt.ylabel("True")
plt.tight_layout()
plt.savefig(f"/kaggle/working/plots/{model_name}
    _confusion_matrix.png", dpi=300)
plt.show()

# 3       Class-wise F1 Plot
class_f1 = [class_report[c]['f1-score'] for c in
    full_dataset.classes]
plt.figure(figsize=(10,6))
plt.barh(full_dataset.classes, class_f1, color="
    mediumseagreen")
plt.title(f"{model_name} - Class-wise F1 Scores")
plt.xlabel("F1-score")
plt.tight_layout()
plt.savefig(f"/kaggle/working/plots/{model_name}
    _classwise_f1.png", dpi=300)
plt.show()

print(f"       Plots saved under /kaggle/working/
    plots/{model_name}_*.png")


# Return summary for comparative analysis

return {
    "Model": model_name,
    "Best Val F1": round(best_val_f1, 4),
    "Test Acc": round(test_acc, 4),
    "Test F1": round(test_f1, 4),
    "Train Time (min)": elapsed
}
```

A.5. Visualization

```
# Example: Train and visualize mobilenetv2_100
results = []
metrics = train_and_evaluate("mobilenetv2_100", epochs
    =50, lr=3e-4)
results.append(metrics)
pd.DataFrame(results)

# Example: Train and visualize efficientnet_b0
results = []
metrics = train_and_evaluate("efficientnet_b0", epochs
    =50, lr=3e-4)
results.append(metrics)
pd.DataFrame(results)

# Example: Train and visualize ResNet50
results = []
metrics = train_and_evaluate("resnet50", epochs=50, lr=3
    e-4)
results.append(metrics)
pd.DataFrame(results)

# Example: Train and visualize ViT
results = []
metrics = train_and_evaluate("vit_base_patch16_224",
    epochs=50, lr=3e-4)
results.append(metrics)
pd.DataFrame(results)
```