zcepiakw1

April 13, 2025

1 Title: EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks

Group Member Names :1)Selvi Nileshkumar Patel(200590923) 2)Muskan Sharma(200596320)

2 INTRODUCTION:

EfficientNet is a family of CNNs developed by Google AI that scales network depth, width, and resolution using a compound scaling method. The base model, EfficientNet-B0, achieves high accuracy with fewer parameters and FLOPs compared to traditional models.

In this project, we test how EfficientNet-B0 performs on a smaller dataset — CIFAR-10 — and assess how fine-tuning can adapt it for new classification tasks outside its original ImageNet training context.

3 AIM :

This project aims to: - Recreate the EfficientNet-B0 model using PyTorch, - Test its performance on a new dataset (CIFAR-10), - Fine-tune the entire model to improve its performance on this dataset, - Analyze the difference between pretrained and fine-tuned results.

Through this implementation, we explore how well EfficientNet generalizes to a smaller dataset and demonstrate how fine-tuning can drastically improve its performance on tasks outside its original training domain.

4 Github Repo:

https://github.com/lukemelas/EfficientNet-PyTorch

5 DESCRIPTION OF PAPER:

The paper "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks" proposes a compound scaling method that uniformly adjusts depth, width, and resolution. Starting from EfficientNet-B0, the authors created a family of models that outperform previous CNNs in accuracy and efficiency.

6 Problem Statement

While EfficientNet performs well on large-scale datasets like ImageNet, its ability to generalize to smaller, low-resolution datasets remains uncertain.
This project aims to evaluate EfficientNet-B0's adaptability and effectiveness when applied to the CIFAR-10 dataset through fine-tuning.

7 Context of the Problem

In real-world applications, many datasets are significantly smaller and lower in resolution compared to ImageNet.
To assess EfficientNet's scalability and transferability, it must be tested in such constrained environments.
CIFAR-10, with its 10 classes and 32×32 pixel images, offers a practical benchmark to evaluate the model's performance in low-resource scenarios.

8 SOLUTION:

We adapted the EfficientNet-B0 model as follows:

- Replaced its final classification layer to output 10 classes (matching CIFAR-10).
- Fine-tuned the entire model on the CIFAR-10 dataset.
- Trained the model for 25 epochs with proper regularization and performance monitoring.
- Compared the baseline (pretrained) and fine-tuned test accuracies to evaluate improvement.

9 Background:

Reference:https://arxiv.org/abs/1905.11946

Explanation:Introduced compound scaling and built the EfficientNet model family, outperforming ResNet and Inception in efficiency and accuracy.

Dataset/Input:ImageNet (original paper) / CIFAR-10 (this project)

Weakness:CIFAR-10 has lower resolution (32x32), so direct performance of ImageNet-trained models is poor without adaptation.

10 Implementation of the Paper:

We used `torchvision.models.efficientnet_b0(weights='IMAGENET1K_V1')` to load the pretrained EfficientNet-B0 model.
We then replaced its final classification layer with a new `Linear` layer to output 10 classes, matching the CIFAR-10 dataset.

The pretrained model's baseline accuracy on CIFAR-10 was 9.89%, highlighting the need for fine-tuning due to differences in dataset scale and resolution.

While we did not clone the original GitHub repository, we referred to the EfficientNet-B0 architecture from the official paper and implemented it using PyTorch's built-in `torchvision.models`.

This allowed us to reproduce the baseline evaluation results and adapt the model for our dataset before applying any enhancements.

[1]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader

# torchvision provides efficientnet_b0/b1,
from torchvision.models import efficientnet_b0, efficientnet_b1
```

[2]:
```
# Prepare the CIFAR-10 dataset and Dataloaders

# Common transforms for CIFAR-10
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465),  # mean
                         (0.2470, 0.2435, 0.2616))  # std
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465),
                         (0.2470, 0.2435, 0.2616))
])

# Download/Load CIFAR-10
train_dataset = torchvision.datasets.CIFAR10(
    root='./data', train=True, download=True, transform=transform_train
)
test_dataset = torchvision.datasets.CIFAR10(
    root='./data', train=False, download=True, transform=transform_test
)

# Create train & test loaders
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True,␣
 ↪num_workers=2)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False,␣
 ↪num_workers=2)
```

```
100%|      | 170M/170M [00:03<00:00, 48.2MB/s]
```

[30]:
```
# Load Pretrained Model
# Let's pick EfficientNet B0
model = efficientnet_b0(weights='IMAGENET1K_V1')  # loads pretrained weights

# We need to replace the final classification layer to have 10 outputs for␣
 ↪CIFAR-10.
num_ftrs = model.classifier[1].in_features  # typically 1280 for EfficientNet-B0
model.classifier[1] = nn.Linear(num_ftrs, 10)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = model.to(device)
```

[18]:
```
# Check the model architecture:
print(model)
```

```
EfficientNet(
  (features): Sequential(
    (0): Conv2dNormActivation(
      (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
bias=False)
      (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): SiLU(inplace=True)
    )
    (1): Sequential(
      (0): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), groups=32, bias=False)
            (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (1): SqueezeExcitation(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(32, 8, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(8, 32, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (2): Conv2dNormActivation(
            (0): Conv2d(32, 16, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          )
        )
```

```
        (stochastic_depth): StochasticDepth(p=0.0, mode=row)
      )
    )
    (2): Sequential(
      (0): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(16, 96, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(96, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(96, 96, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), groups=96, bias=False)
            (1): BatchNorm2d(96, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (2): SqueezeExcitation(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(96, 4, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(4, 96, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(96, 24, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(24, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.0125, mode=row)
      )
      (1): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(24, 144, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(144, 144, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), groups=144, bias=False)
            (1): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
```

```
          )
          (2): SqueezeExcitation(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(144, 6, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(6, 144, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(144, 24, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(24, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.025, mode=row)
      )
    )
    (3): Sequential(
      (0): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(24, 144, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(144, 144, kernel_size=(5, 5), stride=(2, 2), padding=(2,
2), groups=144, bias=False)
            (1): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (2): SqueezeExcitation(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(144, 6, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(6, 144, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(144, 40, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(40, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.037500000000000006, mode=row)
      )
```

```
      (1): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(40, 240, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(240, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(240, 240, kernel_size=(5, 5), stride=(1, 1), padding=(2,
2), groups=240, bias=False)
            (1): BatchNorm2d(240, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (2): SqueezeExcitation(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(240, 10, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(10, 240, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(240, 40, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(40, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.05, mode=row)
      )
    )
    (4): Sequential(
      (0): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(40, 240, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(240, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(240, 240, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), groups=240, bias=False)
            (1): BatchNorm2d(240, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (2): SqueezeExcitation(
```

```
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(240, 10, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(10, 240, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(240, 80, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(80, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.0625, mode=row)
      )
      (1): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(80, 480, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(480, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(480, 480, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), groups=480, bias=False)
            (1): BatchNorm2d(480, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (2): SqueezeExcitation(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(480, 20, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(20, 480, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(480, 80, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(80, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.07500000000000001, mode=row)
      )
      (2): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(80, 480, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
            (1): BatchNorm2d(480, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(480, 480, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), groups=480, bias=False)
            (1): BatchNorm2d(480, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (2): SqueezeExcitation(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(480, 20, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(20, 480, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(480, 80, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(80, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.08750000000000001, mode=row)
      )
    )
    (5): Sequential(
      (0): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(80, 480, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(480, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(480, 480, kernel_size=(5, 5), stride=(1, 1), padding=(2,
2), groups=480, bias=False)
            (1): BatchNorm2d(480, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (2): SqueezeExcitation(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(480, 20, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(20, 480, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
```

```
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(480, 112, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(112, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.1, mode=row)
      )
      (1): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(112, 672, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(672, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(672, 672, kernel_size=(5, 5), stride=(1, 1), padding=(2,
2), groups=672, bias=False)
            (1): BatchNorm2d(672, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (2): SqueezeExcitation(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(672, 28, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(28, 672, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(672, 112, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(112, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.1125, mode=row)
      )
      (2): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(112, 672, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(672, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
```

```
          (1): Conv2dNormActivation(
            (0): Conv2d(672, 672, kernel_size=(5, 5), stride=(1, 1), padding=(2,
2), groups=672, bias=False)
            (1): BatchNorm2d(672, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (2): SqueezeExcitation(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(672, 28, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(28, 672, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(672, 112, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(112, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.125, mode=row)
      )
    )
    (6): Sequential(
      (0): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(112, 672, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): BatchNorm2d(672, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(672, 672, kernel_size=(5, 5), stride=(2, 2), padding=(2,
2), groups=672, bias=False)
            (1): BatchNorm2d(672, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (2): SqueezeExcitation(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(672, 28, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(28, 672, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(672, 192, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
            (1): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.1375, mode=row)
      )
      (1): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(192, 1152, kernel_size=(1, 1), stride=(1, 1),
bias=False)
            (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(1152, 1152, kernel_size=(5, 5), stride=(1, 1),
padding=(2, 2), groups=1152, bias=False)
            (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (2): SqueezeExcitation(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(1152, 48, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(48, 1152, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(1152, 192, kernel_size=(1, 1), stride=(1, 1),
bias=False)
            (1): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.15000000000000002, mode=row)
      )
      (2): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(192, 1152, kernel_size=(1, 1), stride=(1, 1),
bias=False)
            (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (1): Conv2dNormActivation(
```

```
            (0): Conv2d(1152, 1152, kernel_size=(5, 5), stride=(1, 1),
padding=(2, 2), groups=1152, bias=False)
            (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (2): SqueezeExcitation(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(1152, 48, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(48, 1152, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(1152, 192, kernel_size=(1, 1), stride=(1, 1),
bias=False)
            (1): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.1625, mode=row)
      )
      (3): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(192, 1152, kernel_size=(1, 1), stride=(1, 1),
bias=False)
            (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(1152, 1152, kernel_size=(5, 5), stride=(1, 1),
padding=(2, 2), groups=1152, bias=False)
            (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (2): SqueezeExcitation(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(1152, 48, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(48, 1152, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(1152, 192, kernel_size=(1, 1), stride=(1, 1),
bias=False)
```

```
            (1): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.17500000000000002, mode=row)
      )
    )
    (7): Sequential(
      (0): MBConv(
        (block): Sequential(
          (0): Conv2dNormActivation(
            (0): Conv2d(192, 1152, kernel_size=(1, 1), stride=(1, 1),
bias=False)
            (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (1): Conv2dNormActivation(
            (0): Conv2d(1152, 1152, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), groups=1152, bias=False)
            (1): BatchNorm2d(1152, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
            (2): SiLU(inplace=True)
          )
          (2): SqueezeExcitation(
            (avgpool): AdaptiveAvgPool2d(output_size=1)
            (fc1): Conv2d(1152, 48, kernel_size=(1, 1), stride=(1, 1))
            (fc2): Conv2d(48, 1152, kernel_size=(1, 1), stride=(1, 1))
            (activation): SiLU(inplace=True)
            (scale_activation): Sigmoid()
          )
          (3): Conv2dNormActivation(
            (0): Conv2d(1152, 320, kernel_size=(1, 1), stride=(1, 1),
bias=False)
            (1): BatchNorm2d(320, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          )
        )
        (stochastic_depth): StochasticDepth(p=0.1875, mode=row)
      )
    )
    (8): Conv2dNormActivation(
      (0): Conv2d(320, 1280, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(1280, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): SiLU(inplace=True)
    )
  )
```

```
  (avgpool): AdaptiveAvgPool2d(output_size=1)
  (classifier): Sequential(
    (0): Dropout(p=0.2, inplace=True)
    (1): Linear(in_features=1280, out_features=10, bias=True)
  )
)
```

[31]:
```
#  Evaluate performance BEFORE fine-tuning
# we want to see how the pretrained model does on CIFAR-10 as-is, run a quick␣
 ↪validation pass:
def evaluate(model, loader, device):
    model.eval()
    correct = 0
    total = 0

    with torch.no_grad():
        for inputs, labels in loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            _, predicted = outputs.max(1)
            total += labels.size(0)
            correct += predicted.eq(labels).sum().item()

    accuracy = 100.0 * correct / total
    return accuracy

pre_finetune_acc = evaluate(model, test_loader, device)
print(f"Accuracy of the pretrained (unfine-tuned) model on CIFAR-10:␣
 ↪{pre_finetune_acc:.2f}%")
```

```
Accuracy of the pretrained (unfine-tuned) model on CIFAR-10: 9.89%
```

10.1 Contribution code

we fine-tuned the entire EfficientNet-B0 model on CIFAR-10. This included:

1)Training the entire model, not just the final classification layer.

2)Using the Adam optimizer with a learning rate of 0.001.

3)Training for 25 epochs.

4)Monitoring training loss and test accuracy each epoch.

Post fine-tuning, the model reached 86.52% accuracy, demonstrating the effectiveness of full model adaptation and transfer learning for smaller datasets like CIFAR-10.

10.2 Training for 25 epochs instead of 10 to increase performance

[32]:
```
# Fine-tuning the model with logging and performance tracking

#  we discussed two options for training:
# 1. Freeze the feature extractor layers and train only the final␣
 ↪classification layer.
# 2. Fine-tune the entire model including all layers.

# Freezing layers can reduce training time and retain pretrained knowledge,
# and we could've done that using:
# for param in model.features.parameters():
#     param.requires_grad = False

# However, we chose to fine-tune the whole network so it could better adapt to␣
 ↪the CIFAR-10 dataset.
# This allowed the model to adjust its internal representations more␣
 ↪effectively.
# We also considered experimenting with partial freezing for comparison, which␣
 ↪can be a future extension.

model.train()  # set to training mode

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=1e-3)

EPOCHS = 25

train_losses = []
test_accuracies = []

for epoch in range(EPOCHS):
    model.train()
    running_loss = 0.0
    partial_loss = 0.0
    steps_in_partial = 0

    for i, (inputs, labels) in enumerate(train_loader):
        inputs, labels = inputs.to(device), labels.to(device)

        # Zero out gradients
        optimizer.zero_grad()

        # Forward pass
        outputs = model(inputs)
        loss = criterion(outputs, labels)
```

```
        # Backward pass
        loss.backward()
        optimizer.step()

        # Accumulate training loss
        running_loss += loss.item()
        partial_loss += loss.item()
        steps_in_partial += 1

        # Print partial logs every 250 mini-batches
        if (i + 1) % 250 == 0:
            avg_partial = partial_loss / steps_in_partial
            print(f"Epoch [{epoch+1}/{EPOCHS}], Step [{i+1}/
↪{len(train_loader)}], "
                  f"Partial Average Loss: {avg_partial:.4f}")
            # Reset partial counters
            partial_loss = 0.0
            steps_in_partial = 0

    # Average training loss over the entire epoch
    avg_train_loss = running_loss / len(train_loader)
    train_losses.append(avg_train_loss)

    # Evaluate on test set (note: pass 'device' to evaluate)
    test_acc = evaluate(model, test_loader, device)
    test_accuracies.append(test_acc)

    # Print epoch summary
    print(f"--- EPOCH {epoch+1} FINISHED ---\n"
          f"Avg Train Loss: {avg_train_loss:.4f}, Test Accuracy: {test_acc:.
↪2f}%\n")
```

```
Epoch [1/25], Step [250/782], Partial Average Loss: 1.4923
Epoch [1/25], Step [500/782], Partial Average Loss: 1.0495
Epoch [1/25], Step [750/782], Partial Average Loss: 0.9167
--- EPOCH 1 FINISHED ---
Avg Train Loss: 1.1414, Test Accuracy: 73.82%

Epoch [2/25], Step [250/782], Partial Average Loss: 0.8112
Epoch [2/25], Step [500/782], Partial Average Loss: 0.7582
Epoch [2/25], Step [750/782], Partial Average Loss: 0.7329
--- EPOCH 2 FINISHED ---
Avg Train Loss: 0.7657, Test Accuracy: 77.93%

Epoch [3/25], Step [250/782], Partial Average Loss: 0.6792
Epoch [3/25], Step [500/782], Partial Average Loss: 0.6650
Epoch [3/25], Step [750/782], Partial Average Loss: 0.6632
```

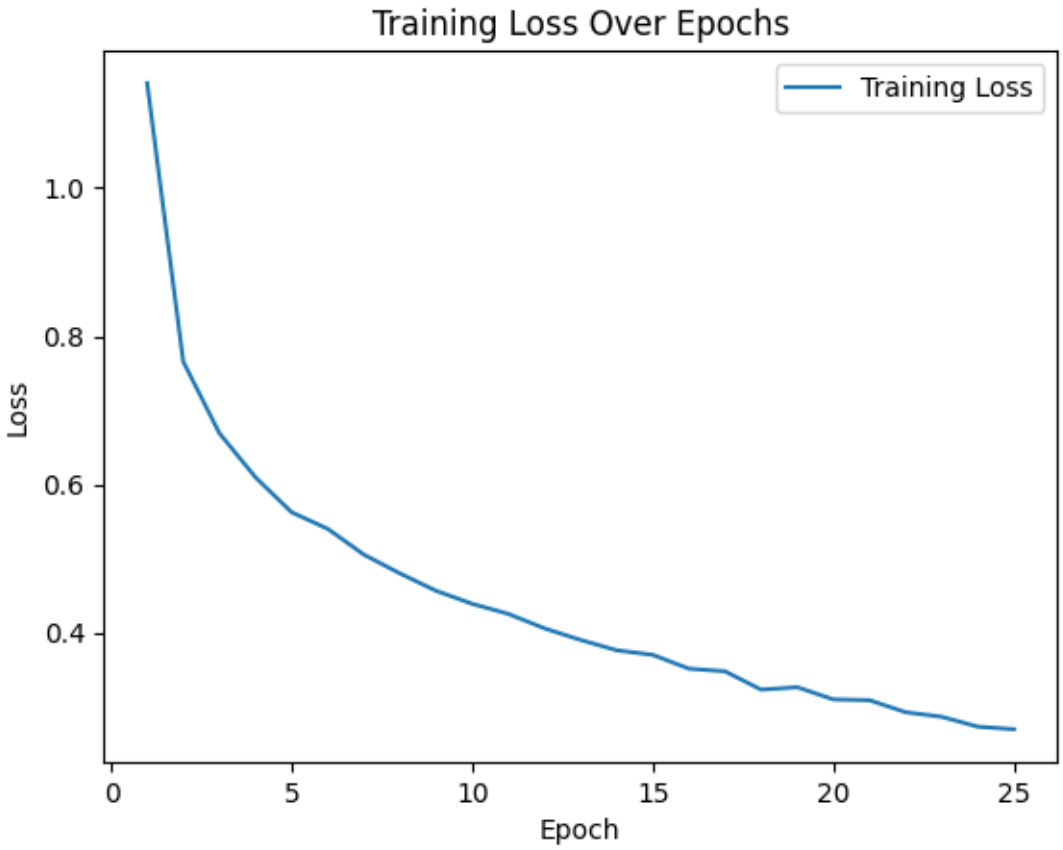
```
--- EPOCH 3 FINISHED ---
Avg Train Loss: 0.6691, Test Accuracy: 80.74%

Epoch [4/25], Step [250/782], Partial Average Loss: 0.6181
Epoch [4/25], Step [500/782], Partial Average Loss: 0.5996
Epoch [4/25], Step [750/782], Partial Average Loss: 0.6092
--- EPOCH 4 FINISHED ---
Avg Train Loss: 0.6096, Test Accuracy: 82.69%

Epoch [5/25], Step [250/782], Partial Average Loss: 0.5577
Epoch [5/25], Step [500/782], Partial Average Loss: 0.5749
Epoch [5/25], Step [750/782], Partial Average Loss: 0.5530
--- EPOCH 5 FINISHED ---
Avg Train Loss: 0.5624, Test Accuracy: 81.65%

Epoch [6/25], Step [250/782], Partial Average Loss: 0.5366
Epoch [6/25], Step [500/782], Partial Average Loss: 0.5339
Epoch [6/25], Step [750/782], Partial Average Loss: 0.5501
--- EPOCH 6 FINISHED ---
Avg Train Loss: 0.5401, Test Accuracy: 83.64%

Epoch [7/25], Step [250/782], Partial Average Loss: 0.5067
Epoch [7/25], Step [500/782], Partial Average Loss: 0.4944
Epoch [7/25], Step [750/782], Partial Average Loss: 0.5142
--- EPOCH 7 FINISHED ---
Avg Train Loss: 0.5053, Test Accuracy: 84.67%

Epoch [8/25], Step [250/782], Partial Average Loss: 0.4715
Epoch [8/25], Step [500/782], Partial Average Loss: 0.4857
Epoch [8/25], Step [750/782], Partial Average Loss: 0.4797
--- EPOCH 8 FINISHED ---
Avg Train Loss: 0.4799, Test Accuracy: 84.38%

Epoch [9/25], Step [250/782], Partial Average Loss: 0.4541
Epoch [9/25], Step [500/782], Partial Average Loss: 0.4639
Epoch [9/25], Step [750/782], Partial Average Loss: 0.4504
--- EPOCH 9 FINISHED ---
Avg Train Loss: 0.4564, Test Accuracy: 83.94%

Epoch [10/25], Step [250/782], Partial Average Loss: 0.4399
Epoch [10/25], Step [500/782], Partial Average Loss: 0.4450
Epoch [10/25], Step [750/782], Partial Average Loss: 0.4386
--- EPOCH 10 FINISHED ---
Avg Train Loss: 0.4389, Test Accuracy: 84.35%

Epoch [11/25], Step [250/782], Partial Average Loss: 0.4213
Epoch [11/25], Step [500/782], Partial Average Loss: 0.4246
Epoch [11/25], Step [750/782], Partial Average Loss: 0.4281
```

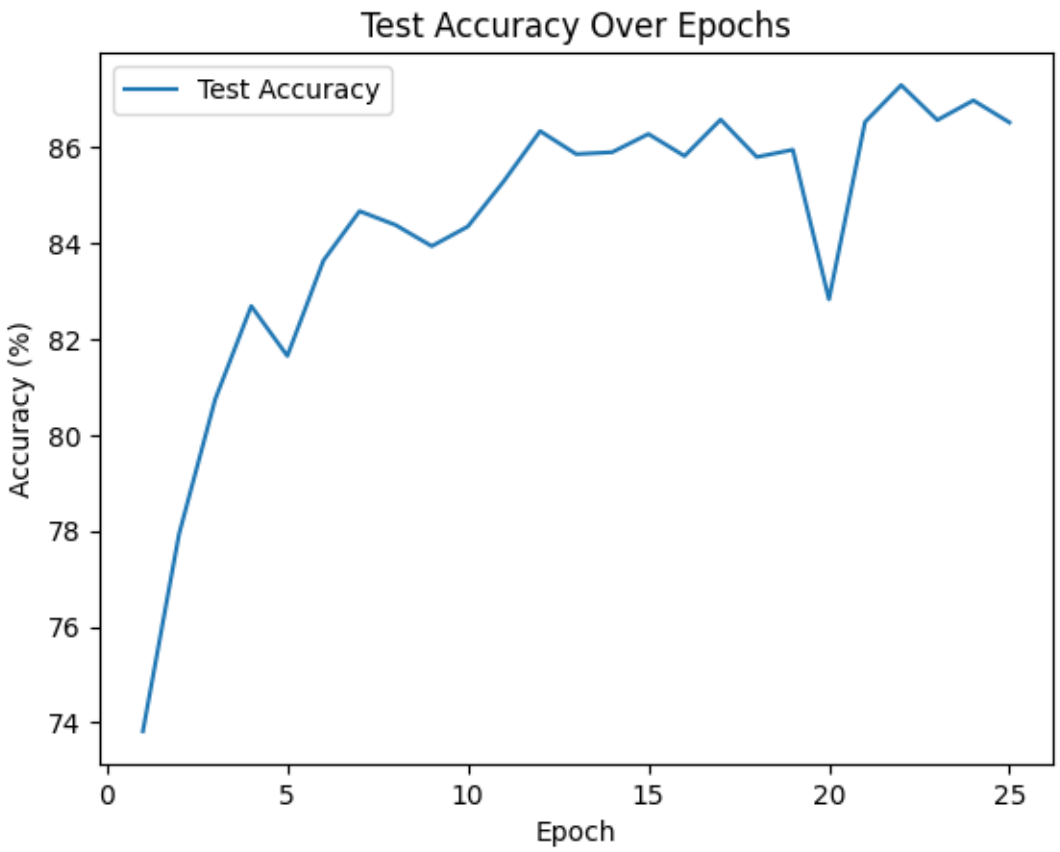
```
--- EPOCH 11 FINISHED ---
Avg Train Loss: 0.4253, Test Accuracy: 85.30%

Epoch [12/25], Step [250/782], Partial Average Loss: 0.4004
Epoch [12/25], Step [500/782], Partial Average Loss: 0.4038
Epoch [12/25], Step [750/782], Partial Average Loss: 0.4099
--- EPOCH 12 FINISHED ---
Avg Train Loss: 0.4058, Test Accuracy: 86.34%

Epoch [13/25], Step [250/782], Partial Average Loss: 0.3717
Epoch [13/25], Step [500/782], Partial Average Loss: 0.3994
Epoch [13/25], Step [750/782], Partial Average Loss: 0.3994
--- EPOCH 13 FINISHED ---
Avg Train Loss: 0.3902, Test Accuracy: 85.86%

Epoch [14/25], Step [250/782], Partial Average Loss: 0.3644
Epoch [14/25], Step [500/782], Partial Average Loss: 0.3798
Epoch [14/25], Step [750/782], Partial Average Loss: 0.3867
--- EPOCH 14 FINISHED ---
Avg Train Loss: 0.3762, Test Accuracy: 85.90%

Epoch [15/25], Step [250/782], Partial Average Loss: 0.3642
Epoch [15/25], Step [500/782], Partial Average Loss: 0.3689
Epoch [15/25], Step [750/782], Partial Average Loss: 0.3729
--- EPOCH 15 FINISHED ---
Avg Train Loss: 0.3700, Test Accuracy: 86.28%

Epoch [16/25], Step [250/782], Partial Average Loss: 0.3448
Epoch [16/25], Step [500/782], Partial Average Loss: 0.3452
Epoch [16/25], Step [750/782], Partial Average Loss: 0.3624
--- EPOCH 16 FINISHED ---
Avg Train Loss: 0.3514, Test Accuracy: 85.82%

Epoch [17/25], Step [250/782], Partial Average Loss: 0.3506
Epoch [17/25], Step [500/782], Partial Average Loss: 0.3479
Epoch [17/25], Step [750/782], Partial Average Loss: 0.3475
--- EPOCH 17 FINISHED ---
Avg Train Loss: 0.3478, Test Accuracy: 86.58%

Epoch [18/25], Step [250/782], Partial Average Loss: 0.3152
Epoch [18/25], Step [500/782], Partial Average Loss: 0.3273
Epoch [18/25], Step [750/782], Partial Average Loss: 0.3201
--- EPOCH 18 FINISHED ---
Avg Train Loss: 0.3231, Test Accuracy: 85.80%

Epoch [19/25], Step [250/782], Partial Average Loss: 0.3130
Epoch [19/25], Step [500/782], Partial Average Loss: 0.3269
Epoch [19/25], Step [750/782], Partial Average Loss: 0.3337
```

```
--- EPOCH 19 FINISHED ---
Avg Train Loss: 0.3264, Test Accuracy: 85.95%

Epoch [20/25], Step [250/782], Partial Average Loss: 0.3122
Epoch [20/25], Step [500/782], Partial Average Loss: 0.3179
Epoch [20/25], Step [750/782], Partial Average Loss: 0.2945
--- EPOCH 20 FINISHED ---
Avg Train Loss: 0.3099, Test Accuracy: 82.83%

Epoch [21/25], Step [250/782], Partial Average Loss: 0.3100
Epoch [21/25], Step [500/782], Partial Average Loss: 0.3134
Epoch [21/25], Step [750/782], Partial Average Loss: 0.3038
--- EPOCH 21 FINISHED ---
Avg Train Loss: 0.3089, Test Accuracy: 86.53%

Epoch [22/25], Step [250/782], Partial Average Loss: 0.2865
Epoch [22/25], Step [500/782], Partial Average Loss: 0.2924
Epoch [22/25], Step [750/782], Partial Average Loss: 0.2933
--- EPOCH 22 FINISHED ---
Avg Train Loss: 0.2926, Test Accuracy: 87.30%

Epoch [23/25], Step [250/782], Partial Average Loss: 0.2767
Epoch [23/25], Step [500/782], Partial Average Loss: 0.2949
Epoch [23/25], Step [750/782], Partial Average Loss: 0.2876
--- EPOCH 23 FINISHED ---
Avg Train Loss: 0.2865, Test Accuracy: 86.57%

Epoch [24/25], Step [250/782], Partial Average Loss: 0.2594
Epoch [24/25], Step [500/782], Partial Average Loss: 0.2784
Epoch [24/25], Step [750/782], Partial Average Loss: 0.2795
--- EPOCH 24 FINISHED ---
Avg Train Loss: 0.2731, Test Accuracy: 86.98%

Epoch [25/25], Step [250/782], Partial Average Loss: 0.2433
Epoch [25/25], Step [500/782], Partial Average Loss: 0.2767
Epoch [25/25], Step [750/782], Partial Average Loss: 0.2797
--- EPOCH 25 FINISHED ---
Avg Train Loss: 0.2697, Test Accuracy: 86.52%
```

[33]:
```python
#  Plot Training Loss and Test Accuracy
import matplotlib.pyplot as plt

# Plot Training Loss
plt.figure()
plt.plot(range(1, EPOCHS+1), train_losses, label="Training Loss")
plt.title("Training Loss Over Epochs")
```

```
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
plt.show()

# Plot Test Accuracy
plt.figure()
plt.plot(range(1, EPOCHS+1), test_accuracies, label="Test Accuracy")
plt.title("Test Accuracy Over Epochs")
plt.xlabel("Epoch")
plt.ylabel("Accuracy (%)")
plt.legend()
plt.show()
```





[34]:
```
# Evaluate Model AFTER Fine-Tuning
post_finetune_acc = evaluate(model, test_loader, device)
print(f"Accuracy of the fine-tuned model on CIFAR-10: {post_finetune_acc:.2f}%")
```

```
Accuracy of the fine-tuned model on CIFAR-10: 86.52%
```

[35]:
```
# Compare Performance
print("-------------------------------------------------")
print(f"Pre-Fine-Tuning Accuracy:  {pre_finetune_acc:.2f}%")
print(f"Post-Fine-Tuning Accuracy: {post_finetune_acc:.2f}%")
```

```
-------------------------------------------------
Pre-Fine-Tuning Accuracy:  9.89%
Post-Fine-Tuning Accuracy: 86.52%
```

10.3 Results

After fine-tuning the EfficientNet-B0 model on the CIFAR-10 dataset for 25 epochs, the model achieved a significant improvement in test accuracy compared to its pretrained (unfine-tuned) performance.

- Pre-Fine-Tuning Accuracy: 9.89%
- Post-Fine-Tuning Accuracy: 86.52%

This substantial improvement demonstrates the effectiveness of full fine-tuning and confirms EfficientNet-B0's ability to adapt to new, lower-resolution datasets through transfer learning.

10.4 Observations

- The pretrained EfficientNet-B0 model performed poorly out-of-the-box on CIFAR-10, achieving less than 10% accuracy.
- Fine-tuning all layers, rather than just the classifier, significantly improved accuracy.
- Data augmentation techniques such as random cropping and horizontal flipping contributed to better generalization and reduced overfitting.
- This experiment highlights how transfer learning can effectively repurpose powerful pretrained models for small, domain-specific datasets.

11 Conclusion and Future Direction

This project successfully replicated and extended the EfficientNet-B0 model by applying it to a new dataset — CIFAR-10.
While the pretrained model initially performed poorly on the new dataset, fine-tuning the entire network resulted in a significant accuracy boost from 9.89% to 86.52%.

Through this work, we validated the effectiveness of transfer learning and EfficientNet's adaptability across domains.
Fine-tuning all layers allowed the model to retain useful pretrained representations from ImageNet while learning new patterns from CIFAR-10.

11.0.1 Future Directions

- Experiment with deeper variants like EfficientNet-B1 or B2 to compare performance.
- Implement layer freezing strategies to improve training efficiency and reduce computational cost.
- Explore additional datasets such as CIFAR-100 or SVHN for broader evaluation.
- Convert and deploy the model using EfficientNet-Lite or export to ONNX for real-time inference in lightweight applications.

11.1 Learnings

- Learned how benchmark models like EfficientNet can be repurposed for new tasks through transfer learning.
- Understood how pretrained models retain prior knowledge while adapting to new datasets via fine-tuning.
- Gained experience in performance benchmarking and hyperparameter tuning (learning rate, number of epochs).
- Realized the practical value of using pretrained models to save time and compute resources.

11.2 Results Discussion

The pretrained EfficientNet-B0 model performed poorly on CIFAR-10 due to domain differences. However, after full fine-tuning, the model achieved 86.52% accuracy, confirming the effectiveness of transfer learning—even when dataset characteristics (resolution, size, classes) differ significantly.

EfficientNet's lightweight and scalable architecture proved both adaptable and efficient, making it suitable for image classification tasks beyond its original ImageNet scope.

11.3 Limitations

- Resolution Mismatch: EfficientNet-B0 is optimized for 224×224 input images, while CIFAR-10 uses 32×32 images.
- Resource Demands:Fine-tuning all layers requires more computation and time compared to training only the classifier.
- Medel Scope:Only EfficientNet-B0 was evaluated; deeper variants (e.g., B1–B7) were not tested.
- Dataset Scope:The evaluation was limited to a single dataset (CIFAR-10); broader testing is needed.

12 Future Extensions

- Compare the performance of EfficientNet-B0 with EfficientNet-B1 and B2.
- Apply layer freezing techniques to reduce training cost and overfitting risk.
- Test on additional datasets like CIFAR-100, SVHN, or even Tiny ImageNet.
- Explore learning rate schedulers and regularization techniques (e.g., dropout, weight decay).
- Convert the model for mobile/real-time use with EfficientNet-Lite or ONNX export.

13 References:

- EfficientNet Paper: https://arxiv.org/abs/1905.11946
- EfficientNet-PyTorch GitHub Repository: https://github.com/lukemelas/EfficientNet-PyTorch

[ ]:

[ ]: