|  |  |  |
| --- | --- | --- |
| **پرسش ۱** | **نام و نام خانوادگی** | سیدرضا مسلمی |
| **شماره دانشجویی** | 810103326 |
| **پرسش ۲** | **نام و نام خانوادگی** | Bibi Ruqia |
| **شماره دانشجویی** | 810102053 |
|  | **مهلت ارسال پاسخ** |  |

|  |  |  |
| --- | --- | --- |
|  | **به نام خدا**  **دانشگاه تهران**  **دانشکده‌ مهندسی برق و کامپیوتر** |  |
| **درس شبکه‌های عصبی و یادگیری عمیق**  **تمرین دوم** | | |

**فهرست**

[**قوانین** 1](#_Toc183209487)

[**پرسش 1**. **تشخیص ضایعه سرطانی با استفاده از CNN** 1](#_Toc183209488)

[۱-۱. معرفی مقاله 1](#_Toc183209489)

[۱-2. پیش پردازش تصاویر 1](#_Toc183209490)

[۱-3. داده افزایی (Data Augmentation) 4](#_Toc183209491)

[۱-4. پیاده‌سازی 5](#_Toc183209492)

[۱-5. تحلیل نتایج 7](#_Toc183209493)

[۱-6. مقایسه نتایج 11](#_Toc183209494)

[۱-7. مدل عمیق‌تر 11](#_Toc183209495)

[۱-6 مقایسه نتایج. 14](#_Toc183209496)

[**پرسش ۲** **- تشخیص بیماریهای برگ لوبیا با شبکههای عصبی** 15](#_Toc183209497)

[۱-۲. **پیش پردازش تصاویر** 15](#_Toc183209498)

[۲-۲. **پیاده سازی** 15](#_Toc183209499)

[۱-۲-۲. **انتخاب مدل ها** 38](#_Toc183209500)

[۲-۲-۲. **تقویت داده** 39](#_Toc183209501)

[۳-۲-۲. **تقویت داده** 40](#_Toc183209502)

[۴-۲-۲. **بهینه سازها** 45](#_Toc183209503)

[۵-۲-۲. **آموزش مدل** 46](#_Toc183209504)

[۳-۲. **تحلیل نتایج** 51](#_Toc183209505)

**شکل‌ها**

Figure 1: Class Distribution 2

Figure 2: Pixel Intensity Distribution 3

Figure 3: Sample Images from Dataset 3

Figure 4: Remove Hair and Contrast Enhansment Implementation 4

Figure 5: Define Augmentations 5

Figure 6: Base Model Structure 6

Figure 7: Modified Model Structure 7

Figure 8 : Base Model Plots 8

Figure 9: Modified Model Plots 10

Figure 10: Deeper Model Structure 12

Figure 11: Deeper model Plots 13

Figure 12: Accuracies and Losses for Bean Leaf Classification 30

**جدول‌ها**

[1: Base Model and Modified Model Result 7](#_Toc183209340)

[2: Deeper Model Results 12](#_Toc183209341)

[3: Final Results and Comparisions 14](#_Toc183209342)

[4: Bean Leaf Results Comparision 53](#_Toc183209343)

# **قوانین**

قبل از پاسخ دادن به پرسش‌ها،‌ موارد زیر را با دقت مطالعه نمایید:

* از پاسخ‌های خود یک گزارش در قالبی که در صفحه‌ی درس در سامانه‌ی Elearn با نام ***REPORTS\_TEMPLATE.docx*** قرار داده شده تهیه نمایید.
* پیشنهاد می‌شود تمرین‌ها را در قالب گروه‌های دو نفره انجام دهید. (بیش از دو نفر مجاز نیست و تحویل تک نفره نیز نمره‌ی اضافی ندارد) توجه نمایید الزامی در یکسان ماندن اعضای گروه تا انتهای ترم وجود ندارد. (یعنی، می‌توانید تمرین اول را با شخص A و تمرین‌ دوم را با شخص B و ... انجام دهید)
* **کیفیت گزارش شما در فرآيند تصحيح از اهميت ويژه­اي برخوردار است**؛ بنابراین، لطفا تمامی نکات و فرض­هایی را كه در پیاده­سازی­ها و محاسبات خود در نظر مي­گيريد در گزارش ذکر کنید.
* در گزارش خود مطابق با آنچه در قالب نمونه قرار داده شده، برای شکل‌ها زیرنویس و برای جدول‌ها بالانویس در نظر بگیرید.
* الزامی به ارائه توضیح جزئیات کد در گزارش نیست، اما باید نتایج بدست آمده از آن را گزارش و تحلیل کنید.
* **تحلیل نتایج الزامی می‌باشد، حتی اگر در صورت پرسش اشاره‌ای به آن نشده باشد.**
* **دستیاران آموزشی ملزم به اجرا کردن کدهای شما نیستند**؛ بنابراین، هرگونه نتیجه و یا تحلیلی که در صورت پرسش از شما خواسته شده را به طور واضح و کامل در گزارش بیاورید. در صورت عدم رعایت این مورد، بدیهی است که از نمره تمرین کسر می­شود.
* **کدها حتما باید در قالب نوت‌بوک با پسوند .ipynb تهیه شوند، در پایان کار، تمامی کد اجرا شود و خروجی هر سلول حتما در این فایل ارسالی شما ذخیره شده باشد.** بنابراین برای مثال اگر خروجی سلولی یک نمودار است که در گزارش آورده‌اید، این نمودار باید هم در گزارش هم در نوت‌بوک کد‌ها وجود داشته باشد.
* **در صورت مشاهده‌ی تقلب امتیاز تمامی افراد شرکت­کننده در آن، 100- لحاظ می­شود.**
* تنها زبان برنامه نویسی مجاز **Python** است.
* **استفاده از کدهای آماده برای تمرین­ها به­ هیچ ­وجه مجاز نیست. در صورتی که دو گروه از یک منبع مشترک استفاده کنند و کدهای مشابه تحویل دهند، تقلب محسوب می‌شود.**
* نحوه محاسبه­ تاخیر به این شکل است: پس از پایان رسیدن مهلت ارسال گزارش، حداکثر تا یک هفته امکان ارسال با تاخیر وجود دارد، پس از این یک هفته نمره آن تکلیف برای شما صفر خواهد شد.
  + سه روز اول: بدون جریمه
  + روز چهارم: ۵ درصد
  + روز پنجم: ۱۰ درصد
  + روز ششم: ۱۵ درصد
  + روز هفتم: ۲۰ درصد
* حداکثر نمره‌ای که برای هر سوال می‌توان اخد کرد ۱۰۰ بوده و اگر مجموع بارم یک **سوال** بیشتر از ۱۰۰ باشد، در صورت اخد نمره بیشتر از ۱۰۰، اعمال نخواهد شد.
  + برای مثال: اگر نمره اخذ شده از سوال ۱ برابر ۱۰۵ و نمره سوال ۲ برابر ۹۵ باشد، نمره نهایی تمرین ۹۷.۵ خواهد بود و نه ۱۰۰.
* لطفا گزارش، کدها و سایر ضمایم را به در یک پوشه با نام زیر قرار داده و آن را فشرده سازید، سپس در سامانه‌ی Elearn بارگذاری نمایید:

HW[Number] \_[Lastname]\_[StudentNumber]\_[Lastname]\_[StudentNumber].zip

(مثال: HW1\_Ahmadi\_810199101\_Bagheri\_810199102.zip)

* برای گروه‌های دو نفره، بارگذاری تمرین از جانب یکی از اعضا کافی است ولی پیشنهاد می‌شود هر دو نفر بارگذاری نمایند.

# **پرسش 1**. **تشخیص ضایعه سرطانی با استفاده از CNN**

۱-۱. معرفی مقاله

۱-2. پیش پردازش تصاویر

**Preprocessing Steps Explained and Their Relevance to the ISIC Skin Cancer Dataset**

1. Image Resizing

Resizing adjusts all images to a certain size. The ISIC dataset contains images of varying resolutions, so this is essential because CNNs require fixed-sized inputs. Resizing also reduces computational complexity, and allows batch processing.

1. Normalization

Normalization scales pixel values to a smaller range, usually between 0 and 1 .It helps speed up convergence during model training by preventing large gradients and ensuring numerical stability. Skin cancer images often have subtle color variations in lesions, and normalization ensures these variations aren't dominated by large raw pixel values.

1. Data Augmentation

Techniques like rotation, flipping, zooming, and brightness adjustments are applied to artificially increase the dataset size. The ISIC dataset has a bit class imbalance (e.g., fewer malignant cases compared to benign lesions). Augmentation helps prevent overfitting by exposing the model to varied representations of the same data, making it more robust to real-world variations like lighting or orientation.

**Exploratory Data Analysis (EDA)**

1. Class Distribution

Checking the calss distribution, we find out the frequency of Benign and Malignant classes is imbalanced a bit. We can simply address this issue via data augmentation.

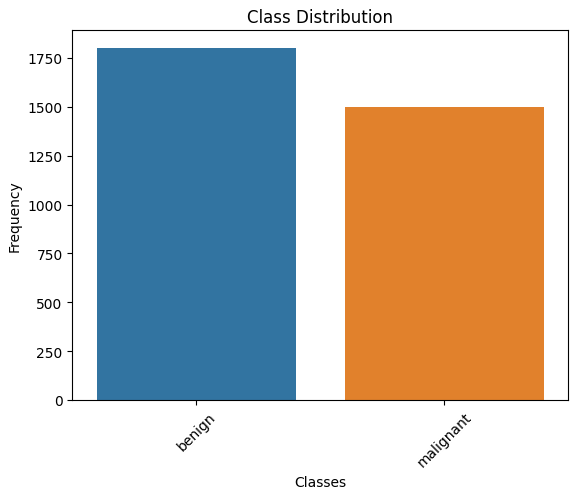


Figure : Class Distribution

1. Pixel Intensity Distribution

Analyzing the pixel intensity distributions, we conclude that the uneven lighting and color artifacts suggest the need for preprocessings like histogram equalization.

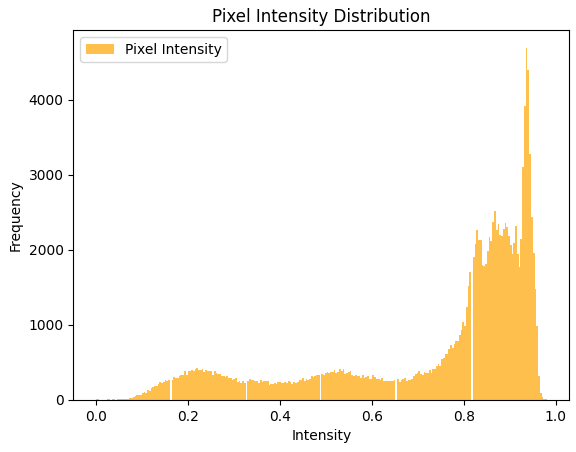


Figure : Pixel Intensity Distribution

1. Sample Images from Dataset

As shown in Figure 3, we see some images consisting of hairs on lesions, lead to crucial chalanges in prediction; Therefore it is necessary to remove this hairs by morphological transformation.



Figure : Sample Images from Dataset

**Additional Preprocessing Recommendations**

1. Hair Removal:

As mentioned before, skin cancer images often include hair artifacts. A preprocessing step to remove hair using morphological operations can improve feature clarity.

1. Contrast Enhancement:

Adjusting contrast using histogram equalization is necessary after Hair Removal.

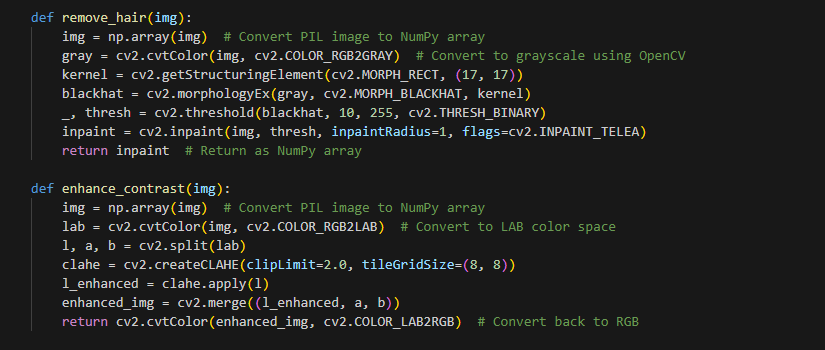


Figure : Remove Hair and Contrast Enhansment Implementation

۱-3. داده افزایی (Data Augmentation)

Dataset augmentation is a critical strategy in machine learning that enriches training data diversity, particularly vital in image processing. It simulates various senarios in real-world and enhance the model’s ability to generalize across new, unseen data, thereby improving its predictive accuracy. These all not only increase the dataset size, but also reduce overfitting and give generalization ability to the model.

The augmentations provided in code enhance the ISIC dataset. **Resizing** standardizes dimensions, while **color jitter**, **flipping**, **cropping**, and **rotation** handle lighting, orientation, and positional differences. **Affine** transformations mimic distortions caused by imaging or patient movement. Converting to **tensors** and **normalizing** aligns data with pre-trained model expectations for stability and faster convergence.

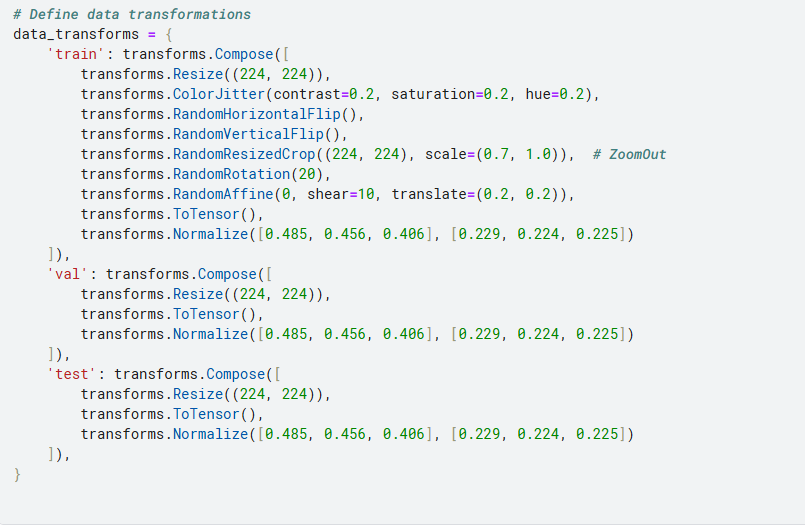


Figure : Define Augmentations

۱-4. پیاده‌سازی

**Strengths and Weaknesses of Dataloaders**

* **Strengths**:

Efficient, scalable, and integrates with PyTorch. Handles transformations, batching, and shuffling out of the box.

* **Weaknesses**:

Can introduce bottlenecks, debugging challenges, and memory overhead if not configured properly.

In this section, we have used *ReduceLROnPlateau & EarlyStopping* as the schaduler, but we implemented early stopping in different way: we will not break the training loop if the models stop improvement, we just make a checkpoint in each epoch which the validation Loss succeed the previous validation results. For the final evaluation the best model got engaged.

**Reasons to use BatchNormalization, Dropouts and new layers in new Model:**

Dropout:

Dropout randomly sets a fraction of the neurons to zero during training, preventing the model from becoming too reliant on specific neurons. This reduces overfitting, especially useful in this datasets, where the high variability of skin lesion patterns might lead to over-complex models memorizing the training data instead of generalizing to new data.

BatchNormalization:

Batch normalization normalizes the inputs to each layer, stabilizing and accelerating training. It mitigates the effects of internal covariate shift, ensuring the model is robust to initialization and learning rate selection.

**Base Model and Modified Model Structures:**

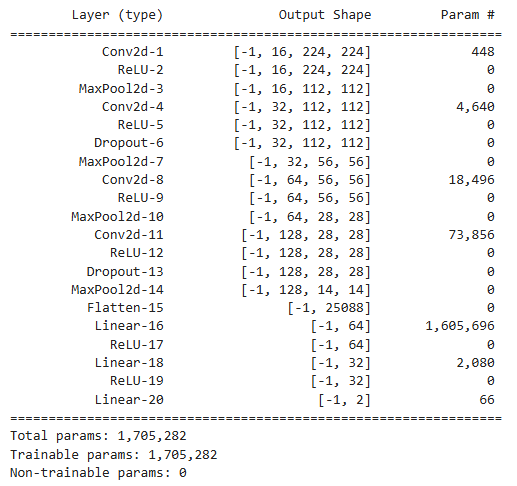
****

Figure : Base Model Structure

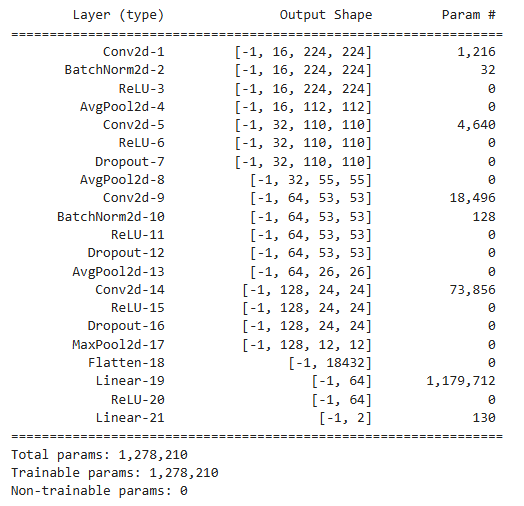


Figure : Modified Model Structure

**Base Model and Modified Model Result:**

: Base Model and Modified Model Result

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Model** | **Train Acc** | **Train Loss** | **Val Acc** | **Val Loss** | **Test Acc** | **Test Loss** |
| **Base Model** | 81.84% | 0.3927 | 83.28% | 0.3942 | 82.18% | 0.3505 |
| **Modified Model** | 80.39% | 0.4101 | 83.28% | 0.3894 | 82.78% | 0.3547 |

۱-5. تحلیل نتایج

**ROC (Receiver Operating Characteristic) Plot:**

The ROC plot visualizes the trade-off between the true positive rate and the false positive rate at various classification thresholds. The area under the ROC curve (AUC) measures the model's ability to distinguish between classes. A value close to 1.0 indicates excellent performance, while 0.5 suggests random guessing. In medical datasets, ROC plots help evaluate the model's effectiveness in identifying diseases, particularly when dealing with imbalanced classes.

**F1 Score, Recall, Precision, and Accuracy:**

* **Accuracy**: Measures the proportion of correctly predicted instances out of all predictions. For imbalanced datasets, accuracy may be incorrect, as it can be high even if the model fails to detect rare cases.

* **Precision**: Indicates the proportion of true positives among all positive predictions. High Precision and cosequently Few false positives, meaning the model is reliable when it predicts a positive case.
* **Recall (Sensitivity)**: Measures the proportion of true positives detected out of all actual positive cases. High Recall means that the model effectively identifies most positive cases (critical in medical diagnosis).
* **F1 Score**: The harmonic mean of precision and recall, providing a balanced measure, especially for imbalanced datasets. High F1 Score means that the model achieves a good trade-off between precision and recall.

Base Model:

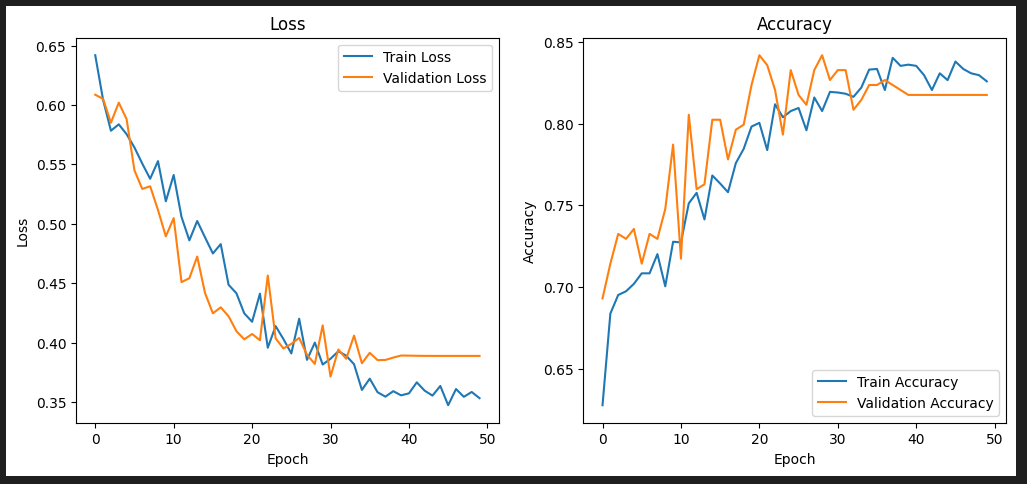
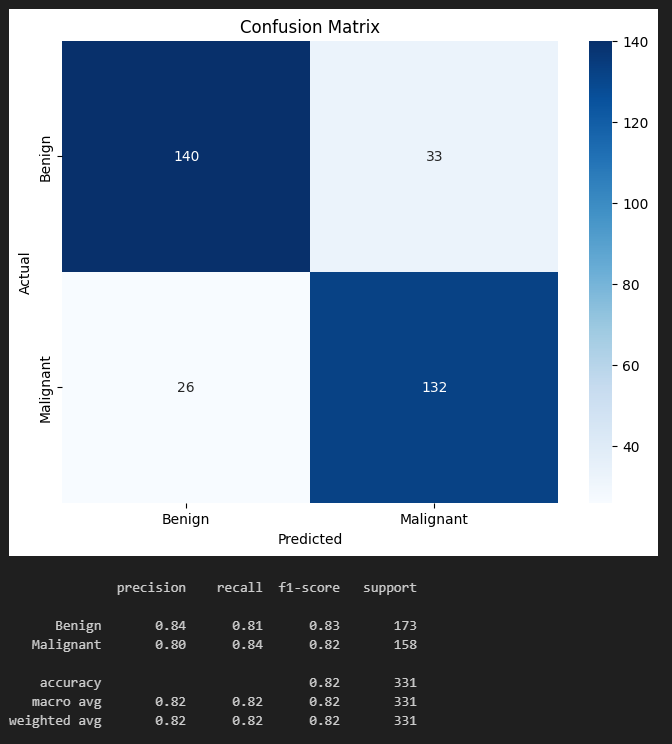
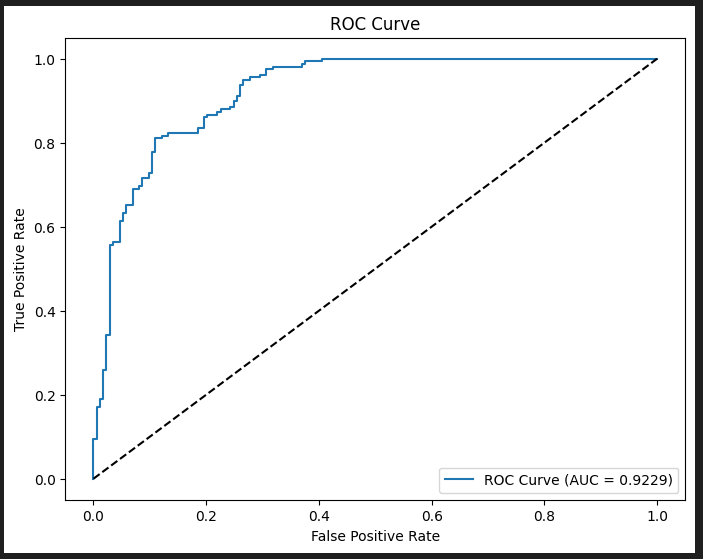


Figure : Base Model Plots

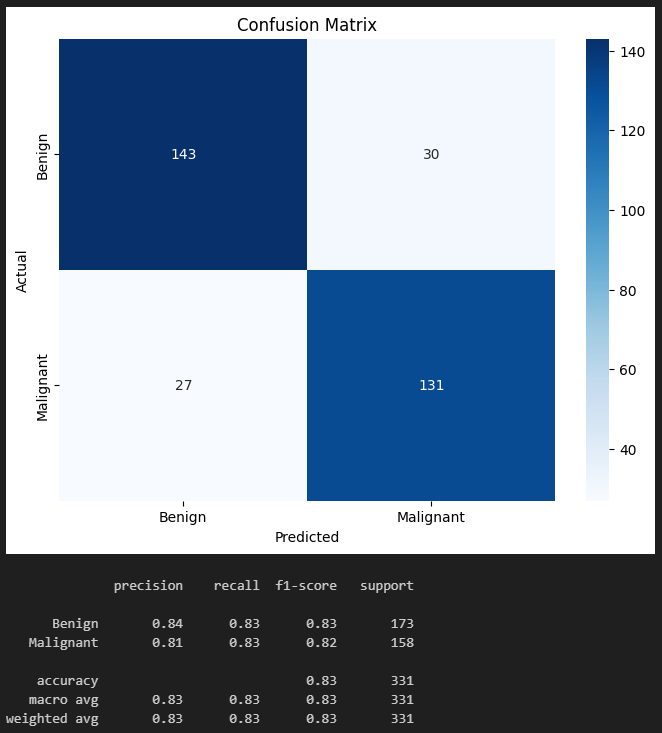


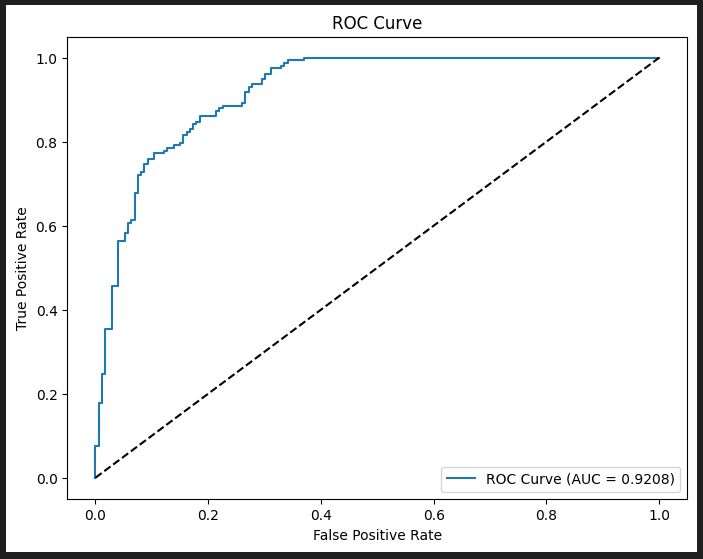


Modified Model:



Figure : Modified Model Plots





As demonstrated by the provided images, generally both models has the equall outputs, but the modified model slightly better results on Recall; Therefore it is better to be used in medical tasks where Recall is important. The ROC curve shows that both models are significantly better than Random with the AUC of 92.08%.

۱-6. مقایسه نتایج

We will discuss after next part.

۱-7. مدل عمیق‌تر

In our new structure, we used skip connections to concat the previous layer’s data and new layer informations, this also avoid gradiant vanishing while the model’s depth grows.

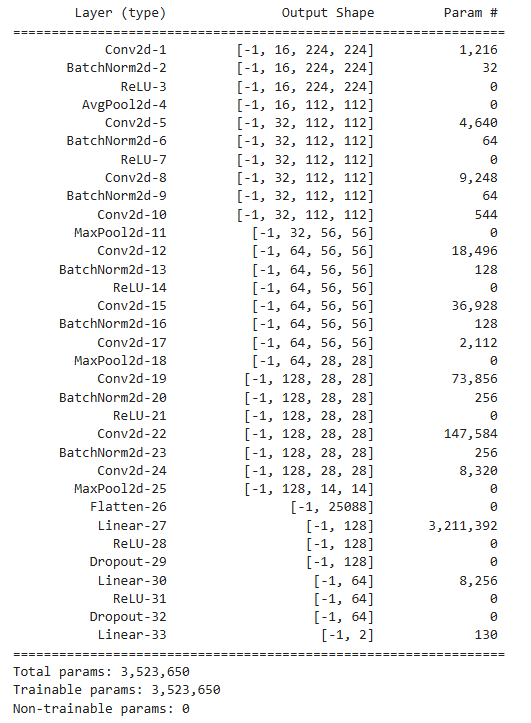


Figure : Deeper Model Structure

**Deeper Model Results:**

: Deeper Model Results

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Train Acc** | **Train Loss** | **Val Acc** | **Val Loss** | **Test Acc** | **Test Loss** |
| **Deeper Model** | 80.28% | 0.3976 | 86.63% | 0.3426 | 87.61% | 0.2894 |

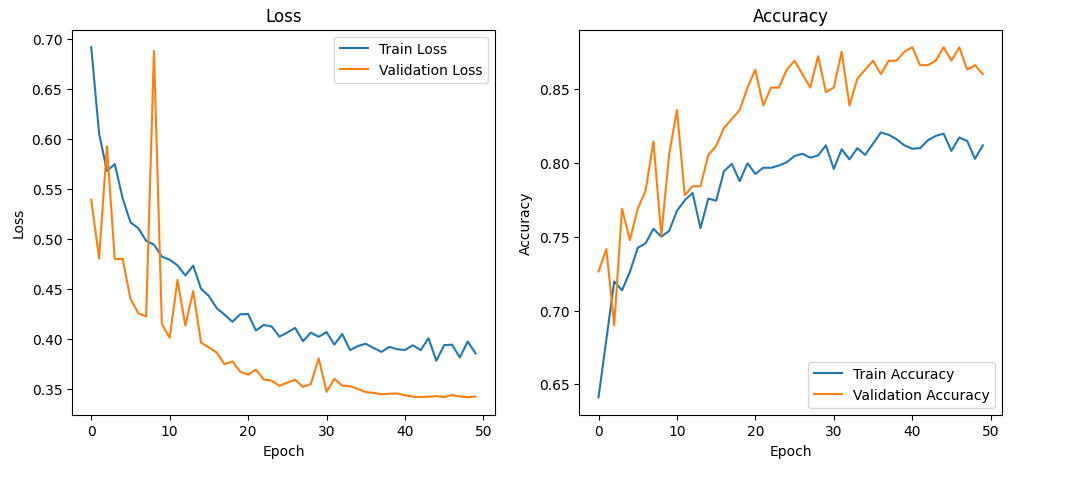
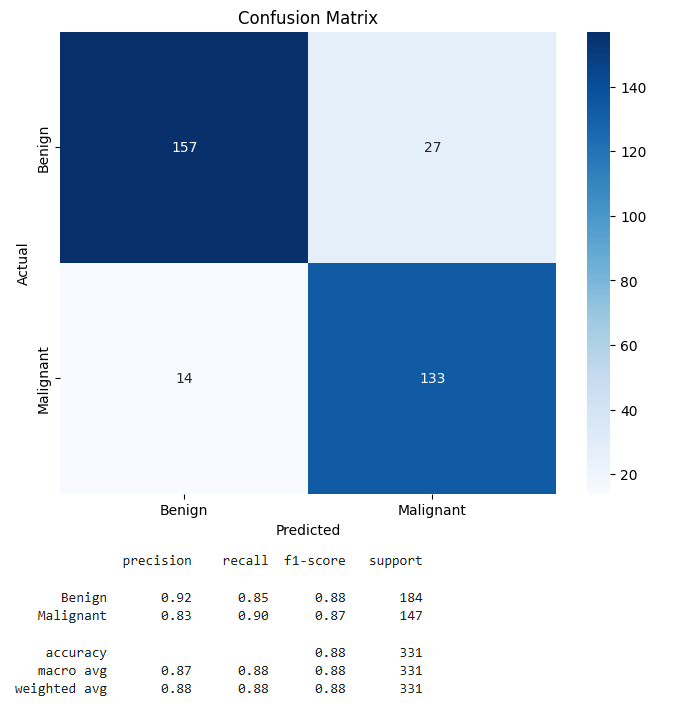
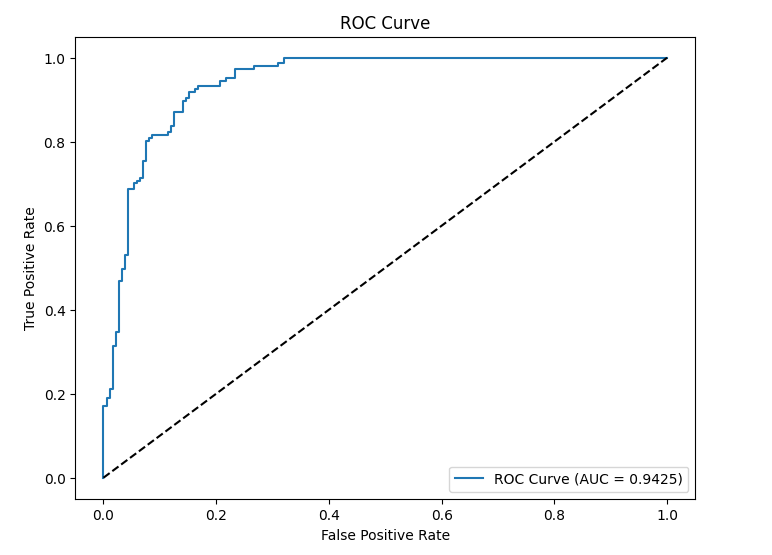


Figure : Deeper model Plots





۱-6. مقایسه نتایج

: Final Results and Comparisions

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Model** | **Train Acc** | **Train Loss** | **Val Acc** | **Val Loss** | **Test Acc** | **Test Loss** | **Wheight Precision** | **Whei. Recall** | **Whei. F1 Score** |
| **Base Model** | 81.84% | 0.3927 | 83.28% | 0.3942 | 82.18% | 0.3505 | 82% | 82% | 82% |
| **Modified Model** | 80.39% | 0.4101 | 83.28% | 0.3894 | 82.78% | 0.3547 | 83% | 83% | 83% |
| **Deeper Model** | 80.28% | 0.3976 | 86.63% | 0.3426 | 87.61% | 0.2894 | 88% | 88% | 88% |

.

# **پرسش ۲** **- تشخیص بیماریهای برگ لوبیا با شبکههای عصبی**

## ۱-۲. **پیش پردازش تصاویر**

Plant disease has been a serious concern in agriculture for years. Smart agriculture has enabled early disease identification and loss minimization by making the best decisions possible based on results. Beans leaf dataset of 1295 images captured by smartphone cameras from the field. Beans can be affected by several diseases, such as angular leaf spot disease and bean rust disease, which can harm bean leaves, causing serious damage to bean crops and reducing the beans crop yield. So to improve the quality and quantity of the product, identification of disease is necessary at an early stage.

## ۲-۲. **پیاده سازی**

**EfficientNetB6**

A comprehensive pipeline for training and evaluating the **EfficientNet-B6** model for image classification tasks, with adaptations to overcome memory limitations, and evaluates performance using various metrics and visualizations.

Due to the large number of parameters in the EfficientNetB6 model, training it posed significant challenges, including long training times and frequent GPU crashes due to memory limitations. To address these issues, we chose to train the model on the PyTorch framework. This allowed us to take full advantage of PyTorch’s efficient data loaders and provided greater control over resource management, helping to mitigate memory-related problems and improve overall training stability.

import torch

import torchvision.transforms as transforms

import torchvision.datasets as datasets

import torch.nn as nn

import torch.optim as optim

from torchvision import models

from torch.utils.data import DataLoader, Dataset

import matplotlib.pyplot as plt

import os

import numpy as np

from tqdm import tqdm

# Device Configuration

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

print(f"Using device: {device}")

# Hyperparameters

batch\_size = 16

img\_height, img\_width = 224, 224

num\_classes = 3

epochs = 25

lr = 0.001

checkpoint\_dir = 'checkpoints/'

os.makedirs(checkpoint\_dir, exist\_ok=True)

from albumentations import (

    Resize, Normalize, HorizontalFlip, VerticalFlip, RandomBrightnessContrast, ShiftScaleRotate, Compose

)

from albumentations.pytorch import ToTensorV2

input\_sizes = {

    "EfficientNet": (528, 528),

    "NASNet": (331, 331),

    "MobileNet": (224, 224)

}

batch\_sizes = {

    "EfficientNet": 6,

    "NASNet": 12,

    "MobileNet": 32

}

# Custom dataset with dynamic resizing

class DynamicResizeDataset(Dataset):

    def \_\_init\_\_(self, image\_folder\_dataset, model\_name, input\_sizes, mean, std):

        self.dataset = image\_folder\_dataset

        self.input\_size = input\_sizes[model\_name]

        self.transform = Compose([

            Resize(\*self.input\_size),  # Dynamic resizing

            HorizontalFlip(p=0.5),                           # Random horizontal flip

            VerticalFlip(p=0.2),                             # Random vertical flip

            ShiftScaleRotate(shift\_limit=0.05, scale\_limit=0.05, rotate\_limit=15, p=0.5),  # Geometric transforms

            RandomBrightnessContrast(p=0.2),                # Adjust brightness and contrast

            Normalize(mean=mean, std=std),  # Normalize to ImageNet standards

            ToTensorV2()

        ])

    def \_\_len\_\_(self):

        return len(self.dataset)

    def \_\_getitem\_\_(self, idx):

        image, label = self.dataset[idx]

        image = np.array(image)

        transformed = self.transform(image=image)

        return transformed['image'], label

# Mean and std for normalization

mean = [0.485, 0.456, 0.406]

std = [0.229, 0.224, 0.225]

# Initialize datasets and loaders

datasets\_dictionary = {

    f'{model\_name}\_{split}': DynamicResizeDataset(

        image\_folder\_dataset=datasets.ImageFolder(root=f"/kaggle/input/bean-leaf-dataset/{split}/{split}"),

        model\_name=model\_name,

        input\_sizes=input\_sizes,

        mean=mean,

        std=std

    )

    for model\_name in input\_sizes

    for split in ["train", "validation", "test"]

}

# DataLoaders

loaders = {

    model\_name: {

        split: DataLoader(datasets\_dictionary[f"{model\_name}\_{split}"], batch\_size=batch\_sizes[model\_name], shuffle=(split == "train"), num\_workers=4)

        for split in ["train", "validation", "test"]

    }

    for model\_name in input\_sizes

}

# Model Definition

def create\_models(pretrained\_model, model\_name, dropout\_rate=0.3, num\_classes=num\_classes):

    model = pretrained\_model

    # Attempt to find and replace the final classification layer dynamically for NASNet

    if model\_name == "NASNet":

        found = False

        for name, module in model.named\_children():

            if isinstance(module, nn.Linear):

                in\_features = module.in\_features

                setattr(model, name, nn.Sequential(

                    nn.Dropout(dropout\_rate),

                    nn.Linear(in\_features, num\_classes)

                ))

                found = True

                break

            elif isinstance(module, nn.Sequential):

                for sub\_name, sub\_module in module.named\_children():

                    if isinstance(sub\_module, nn.Linear):

                        in\_features = sub\_module.in\_features

                        module.\_modules[sub\_name] = nn.Sequential(

                            nn.Dropout(dropout\_rate),

                            nn.Linear(in\_features, num\_classes)

                        )

                        found = True

                        break

        if not found:

            raise AttributeError(f"NASNet model does not have a compatible classification layer.")

        return model.to(device)

    # Handle other models like EfficientNet, MobileNet, ResNet, etc.

    if hasattr(model, 'classifier'):

        in\_features = model.classifier[1].in\_features

        model.classifier = nn.Sequential(

            nn.Dropout(dropout\_rate),

            nn.Linear(in\_features, num\_classes)

        )

    elif hasattr(model, 'fc'):

        in\_features = model.fc.in\_features

        model.fc = nn.Sequential(

            nn.Dropout(dropout\_rate),

            nn.Linear(in\_features, num\_classes)

        )

    else:

        raise AttributeError(f"Unsupported model type for {type(model)}; could not locate a classification layer.")

    return model.to(device)

def train\_model(

    model\_name, optimizer\_name, model, train\_loader, val\_loader, optimizer, criterion, num\_epochs,

    accumulation\_steps=32, early\_stopping\_patience=3

):

    best\_accuracy = 0.0

    best\_val\_loss = float('inf')

    history = {'train\_loss': [], 'val\_loss': [], 'train\_acc': [], 'val\_acc': []}

    patience\_counter = 0  # Tracks epochs without improvement for early stopping

    for epoch in range(num\_epochs):

        model.train()

        train\_loss, correct = 0, 0

        optimizer.zero\_grad()  # Clear gradients at the start

        for step, (images, labels) in enumerate(tqdm(train\_loader)):

            images, labels = images.to(device), labels.to(device)

            outputs = model(images)

            loss = criterion(outputs, labels) / accumulation\_steps  # Scale loss

            loss.backward()  # Accumulate gradients

            train\_loss += loss.item() \* images.size(0) \* accumulation\_steps

            correct += (outputs.argmax(1) == labels).sum().item()

            # Perform optimizer step every `accumulation\_steps`

            if (step + 1) % accumulation\_steps == 0 or (step + 1) == len(train\_loader):

                optimizer.step()

                optimizer.zero\_grad()

        train\_loss /= len(train\_loader.dataset)

        train\_acc = correct / len(train\_loader.dataset)

        # Validation phase

        model.eval()

        val\_loss, correct = 0, 0

        with torch.no\_grad():

            for images, labels in val\_loader:

                images, labels = images.to(device), labels.to(device)

                outputs = model(images)

                loss = criterion(outputs, labels)

                val\_loss += loss.item() \* images.size(0)

                correct += (outputs.argmax(1) == labels).sum().item()

        val\_loss /= len(val\_loader.dataset)

        val\_acc = correct / len(val\_loader.dataset)

        history['train\_loss'].append(train\_loss)

        history['val\_loss'].append(val\_loss)

        history['train\_acc'].append(train\_acc)

        history['val\_acc'].append(val\_acc)

        print(f"Epoch [{epoch+1}/{num\_epochs}] - Train Loss: {train\_loss:.4f}, Train Acc: {train\_acc:.4f}, "

              f"Val Loss: {val\_loss:.4f}, Val Acc: {val\_acc:.4f}")

        # Save the best model based on validation accuracy

        if val\_loss < best\_val\_loss:

            best\_val\_loss = val\_loss

            best\_accuracy = val\_acc

            torch.save(model.state\_dict(), os.path.join(checkpoint\_dir, f'{model\_name}\_{optimizer\_name}\_best\_model.h5'))

            print(f"New best model saved with Val Loss: {best\_val\_loss:.4f} and Val Acc: {best\_accuracy:.4f}")

            patience\_counter = 0  # Reset patience counter

        else:

            patience\_counter += 1

            print(f"No improvement in val\_loss. Patience: {patience\_counter}")

        # Early stopping condition

        if patience\_counter >= early\_stopping\_patience:

            print(f"Early stopping triggered after {patience\_counter} epochs without improvement.")

            break

    return history

from sklearn.metrics import roc\_curve, auc, confusion\_matrix, precision\_score, recall\_score, f1\_score

from timm import create\_model

# Evaluation Function

def evaluate\_model(model, test\_loader, model\_name, optimizer\_name, num\_classes=3):

    model.eval()

    all\_labels = []

    all\_preds = []

    all\_probs = []

    criterion = nn.CrossEntropyLoss()

    test\_loss, correct = 0, 0

    with torch.no\_grad():

        for images, labels in test\_loader:

            images, labels = images.to(device), labels.to(device)

            outputs = model(images)

            # Softmax to get probabilities

            probs = torch.softmax(outputs, dim=1)

            preds = outputs.argmax(1)

            all\_probs.extend(probs.cpu().numpy())  # Store probabilities

            all\_labels.extend(labels.cpu().numpy())

            all\_preds.extend(preds.cpu().numpy())

            test\_loss += criterion(outputs, labels).item() \* images.size(0)

            correct += (preds == labels).sum().item()

    # Calculate metrics

    test\_loss /= len(test\_loader.dataset)

    test\_acc = correct / len(test\_loader.dataset)

    precision = precision\_score(all\_labels, all\_preds, average='weighted', zero\_division=0)

    recall = recall\_score(all\_labels, all\_preds, average='weighted', zero\_division=0)

    f1 = f1\_score(all\_labels, all\_preds, average='weighted', zero\_division=0)

    conf\_matrix = confusion\_matrix(all\_labels, all\_preds)

    # ROC Curve for multiclass

    fpr = dict()

    tpr = dict()

    roc\_auc = dict()

    # For each class, calculate the ROC curve

    for i in range(num\_classes):

        fpr[i], tpr[i], \_ = roc\_curve(np.array(all\_labels) == i, np.array([p[i] for p in all\_probs]))

        roc\_auc[i] = auc(fpr[i], tpr[i])

    # Macro-average ROC curve

    all\_fpr = np.unique(np.concatenate([fpr[i] for i in range(num\_classes)]))

    mean\_tpr = np.zeros\_like(all\_fpr)

    for i in range(num\_classes):

        mean\_tpr += np.interp(all\_fpr, fpr[i], tpr[i])

    mean\_tpr /= num\_classes

    roc\_auc["macro"] = auc(all\_fpr, mean\_tpr)

    print(f"{model\_name} with {optimizer\_name} - Test Loss: {test\_loss:.4f}, Test Accuracy: {test\_acc:.4f}")

    print(f"Precision: {precision:.4f}, Recall: {recall:.4f}, F1 Score: {f1:.4f}")

    # Return metrics for visualization

    return {

        "test\_loss": test\_loss,

        "test\_acc": test\_acc,

        "precision": precision,

        "recall": recall,

        "f1": f1,

        "conf\_matrix": conf\_matrix,

        "fpr": fpr,

        "tpr": tpr,

        "roc\_auc": roc\_auc,

    }

pretrained\_models = {

    "NASNet": lambda: create\_model('nasnetalarge', pretrained=True),  # NASNet Large

    "EfficientNet": lambda: models.efficientnet\_b6(pretrained=True),

    "MobileNet": lambda: models.mobilenet\_v2(pretrained=True)

}

optimizers = {

    "Adam": optim.Adam,

    "RMSprop": optim.RMSprop,

    "Nadam": lambda params, lr: optim.NAdam(params, lr=lr)  # Nadam isn't directly available in all PyTorch versions

}

# Loop over each model

for model\_name, model\_fn in pretrained\_models.items():

    print(f"Training {model\_name}...")

    # Dynamically set data loaders based on the model's input size

    train\_loader = loaders[model\_name]["train"]

    val\_loader = loaders[model\_name]["validation"]

    test\_loader = loaders[model\_name]["test"]

    for optimizer\_name, optimizer\_fn in optimizers.items():

        # Reload the model for each optimizer

        pretrained\_model = model\_fn()  # Create a fresh model instance

        model = create\_models(pretrained\_model, model\_name)

        # Set optimizer and loss function

        optimizer = optimizer\_fn(model.parameters(), lr=lr)

        criterion = nn.CrossEntropyLoss()

        print(f"Using Optimizer: {optimizer\_name}")

        # Training the model

        history = train\_model(model\_name, optimizer\_name, model, train\_loader, val\_loader, optimizer, criterion, epochs)

        histories[f"{model\_name}\_{optimizer\_name}"] = history  # Store history

        # Evaluating the model

        eval\_result = evaluate\_model(model, test\_loader, model\_name, optimizer\_name)

        evaluation\_results[f"{model\_name}\_{optimizer\_name}"] = eval\_result

        # Store the trained model

        models\_dictionary[f"{model\_name}\_{optimizer\_name}"] = model

# Plotting training and validation accuracy for each model and optimizer

for key, history in histories.items():

    plt.plot(history['train\_acc'], label=f'{key} Training Accuracy')

    plt.plot(history['val\_acc'], label=f'{key} Validation Accuracy')

    plt.xlabel("Epochs")

    plt.ylabel("Accuracy")

    plt.title(f"{key} Training and Validation Accuracy")

    plt.legend()

    plt.show()

# Plotting training and validation accuracy for each model and optimizer

for key, history in histories.items():

    plt.plot(history['train\_loss'], label=f'{key} Training Accuracy')

    plt.plot(history['val\_loss'], label=f'{key} Validation Accuracy')

    plt.xlabel("Epochs")

    plt.ylabel("Accuracy")

    plt.title(f"{key} Training and Validation Accuracy")

    plt.legend()

    plt.show()

import seaborn as sns

def plot\_confusion\_matrix(conf\_matrix, model\_name, optimizer\_name, num\_classes=3):

    plt.figure(figsize=(8, 6))

    ax = plt.subplot()

    # Create heatmap

    sns.heatmap(conf\_matrix, annot=True, fmt='d', cmap='Blues', cbar=False,

                xticklabels=["Angular Leaf Spot", "Bean Rust", "Healthy"],

                yticklabels=["Angular Leaf Spot", "Bean Rust", "Healthy"])

    # Add labels and title

    ax.set\_xlabel('Predicted Labels')

    ax.set\_ylabel('True Labels')

    ax.set\_title(f'Confusion Matrix for {model\_name} with {optimizer\_name}')

    plt.show()

for key, metrics in evaluation\_results.items():

    print(f"Results for {key}:")

    print(f"Test Accuracy: {metrics['test\_acc']:.4f}")

    print(f"Precision: {metrics['precision']:.4f}")

    print(f"Recall: {metrics['recall']:.4f}")

    print(f"F1 Score: {metrics['f1']:.4f}")

    # Visualize the confusion matrix heatmap

    plot\_confusion\_matrix(metrics['conf\_matrix'], model\_name=key.split('\_')[0], optimizer\_name=key.split('\_')[1])

    # ROC Curve for each class and macro-average

    plt.figure(figsize=(8, 6))

    for i, class\_label in enumerate(["Angular Leaf Spot", "Bean Rust", "Healthy"]):

        plt.plot(metrics['fpr'][i], metrics['tpr'][i], label=f"{class\_label} AUC = {metrics['roc\_auc'][i]:.2f}")

    plt.plot([0, 1], [0, 1], 'k--', label="Random Guess")

    plt.xlabel("False Positive Rate")

    plt.ylabel("True Positive Rate")

    plt.title(f"ROC Curve for {key} (Macro-AUC = {metrics['roc\_auc']['macro']:.2f})")

    plt.legend()

    plt.show()

import random

# Visualization Function

def plot\_predictions(images, labels, predictions, model\_name):

    plt.figure(figsize=(15, 15))

    for i in range(5):

        ax = plt.subplot(3, 3, i + 1)

        img = images[i].cpu().permute(1, 2, 0).numpy()

        img = (img \* np.array([0.229, 0.224, 0.225])) + np.array([0.485, 0.456, 0.406])  # Denormalize

        img = np.clip(img, 0, 1)

        plt.imshow(img)

        actual\_label = labels[i]

        predicted\_label = predictions[i]

        color = "green" if actual\_label == predicted\_label else "red"

        plt.title(f"{model\_name}\nActual: {actual\_label}\nPredicted: {predicted\_label}", color=color)

        plt.axis("off")

    plt.show()

for model\_name, model in models\_dictionary.items():

    print(model\_name, ":")

    # Select a random batch from the test loader

    test\_batches = list(test\_loader)  # Convert test\_loader to a list of batches

    random\_batch = random.choice(test\_batches)  # Choose a random batch

    images, labels = random\_batch  # Unpack the selected batch

    images, labels = images[:5], labels[:5]  # Take the first 9 images for visualization

    outputs = model(images.to(device))  # Get model predictions

    predictions = outputs.argmax(1).cpu()  # Get predicted classes

    # Visualize the predictions

    plot\_predictions(images, labels, predictions, f"{model\_name.split('\_')[0]} with Optimizer={model\_name.split('\_')[1]}")

import os

os.listdir('/kaggle/working/checkpoints')

import shutil

shutil.make\_archive('/kaggle/working/models', 'zip', '/kaggle/working/checkpoints')

**Accuracies:**

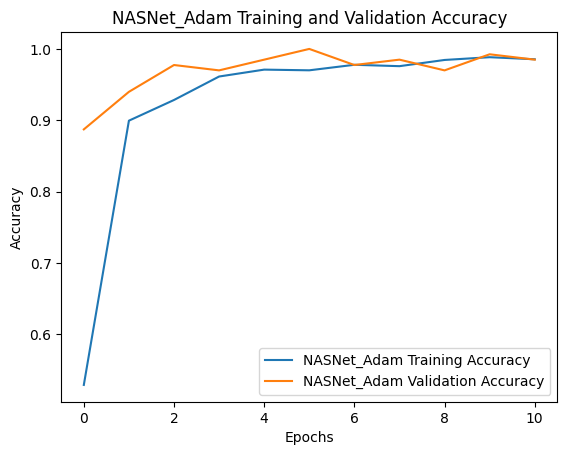
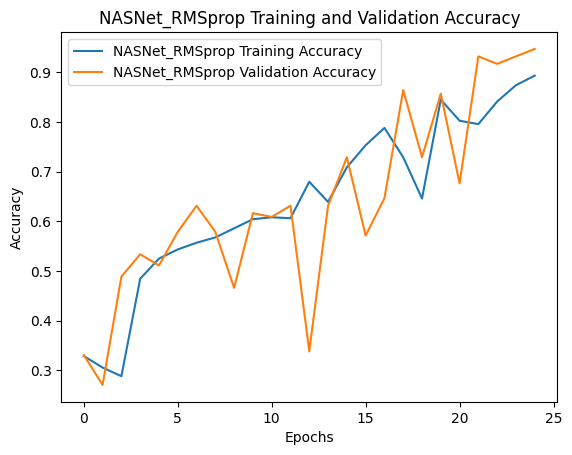
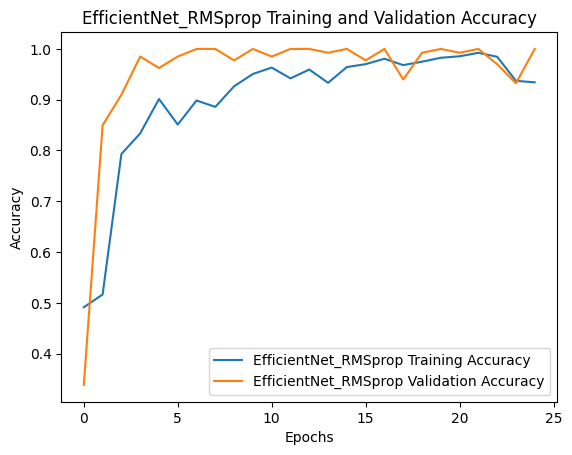
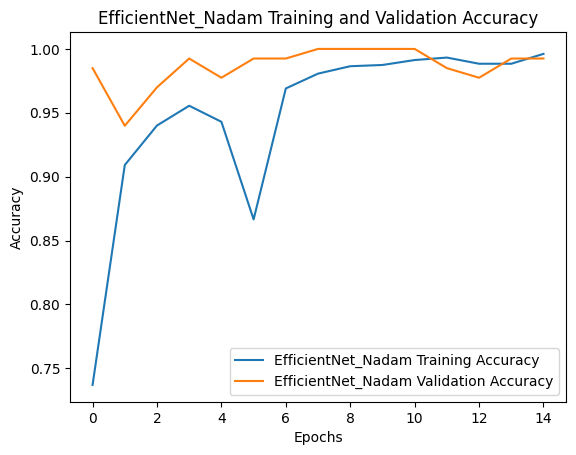


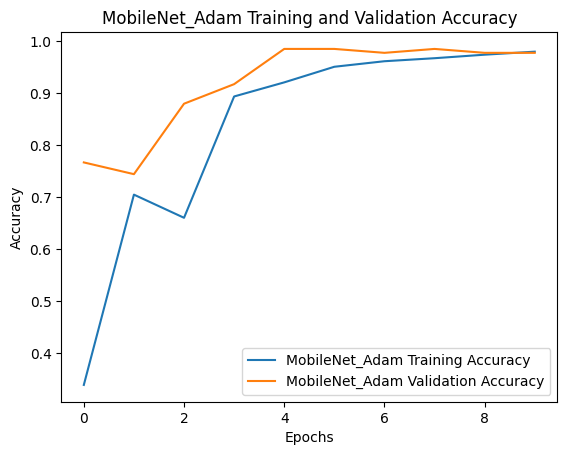
Figure : Accuracies and Losses for Bean Leaf Classification

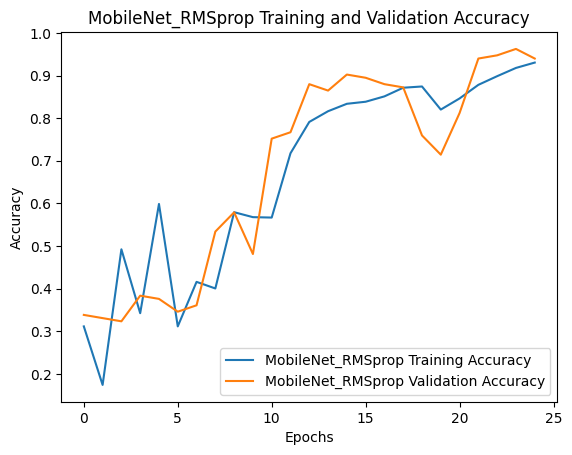


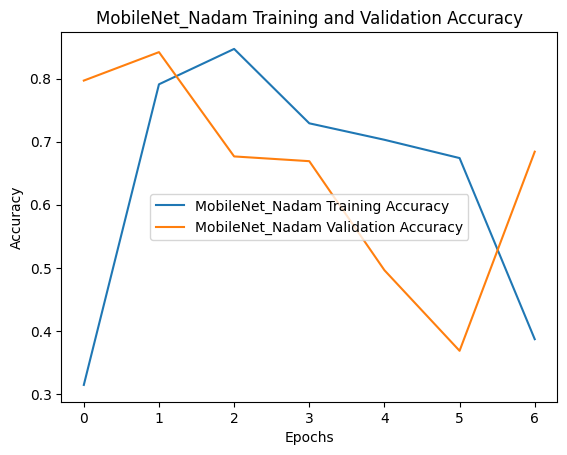




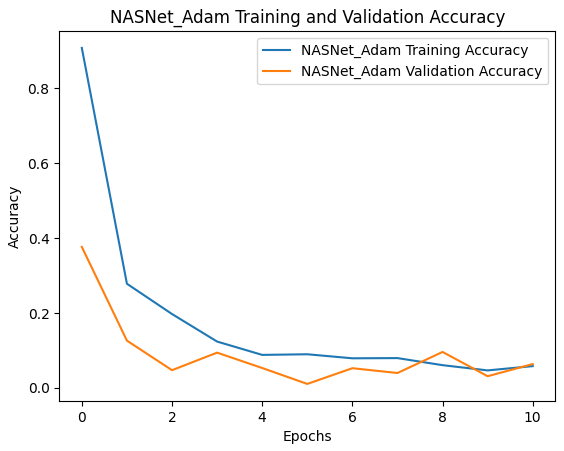


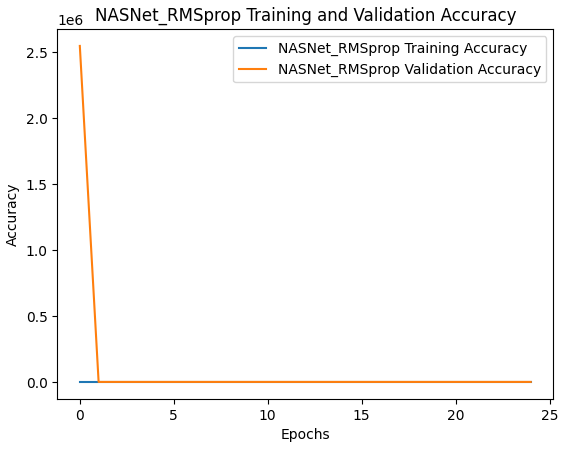


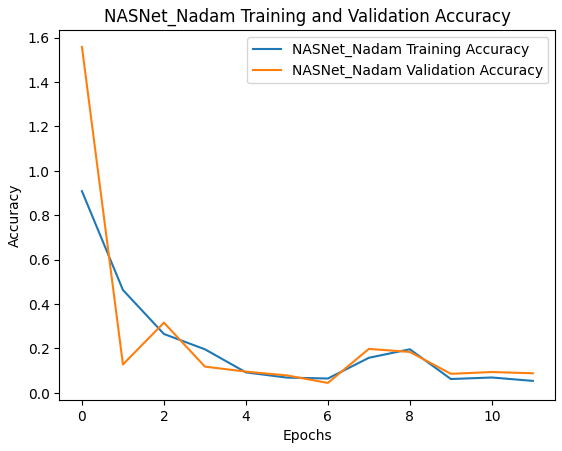


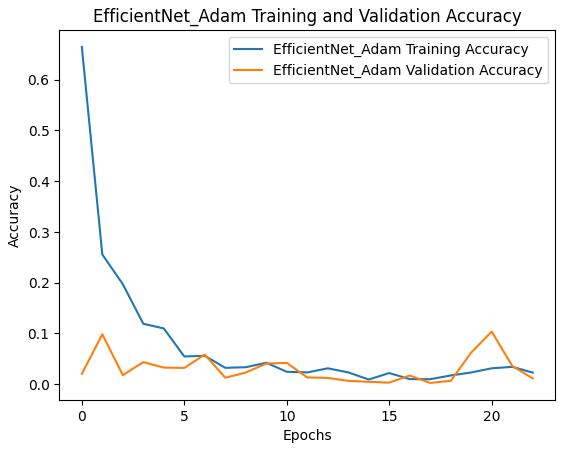


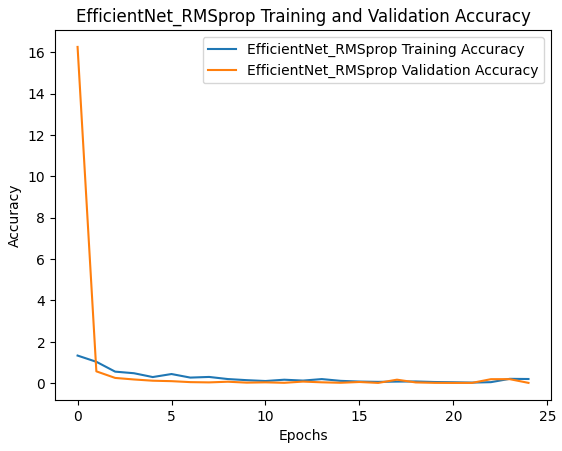
**Losses (I wrote Accuracy by mistake, these are losses):**

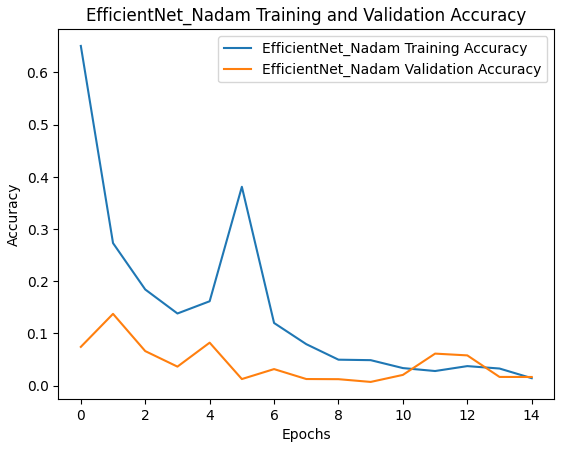


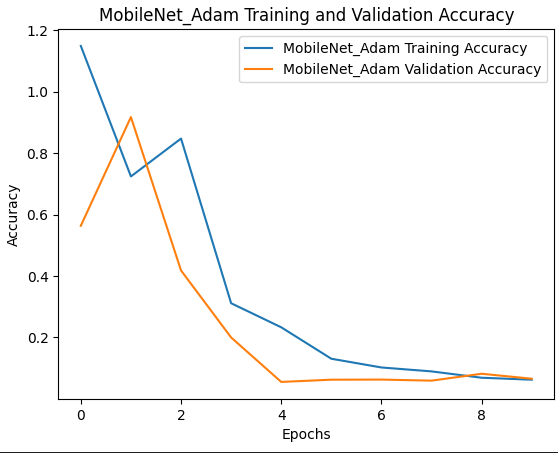


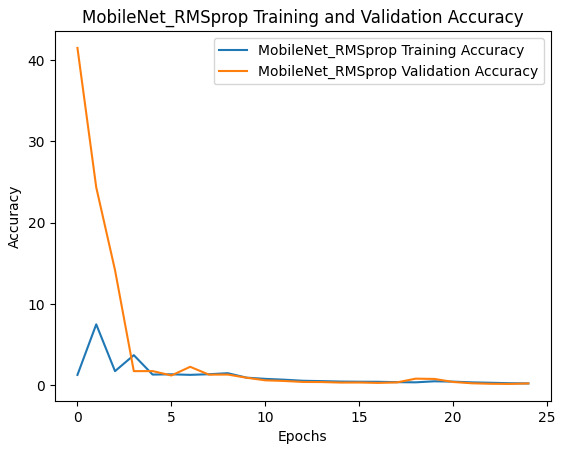


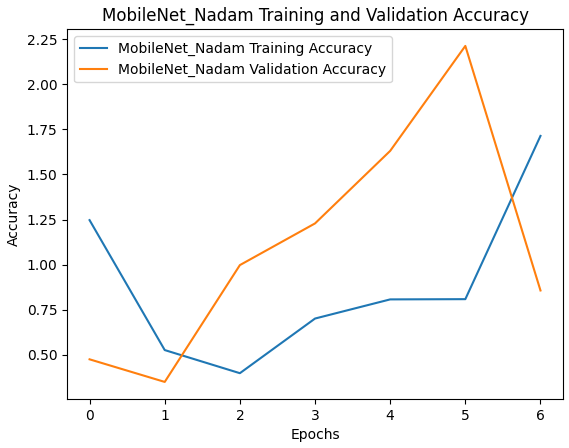












The bad results of Mobilenet on Nadam is for early stopping, we got more than 90% before, but we could not run it again due to being time consuming and gpu problems.

## ۱-۲-۲. **انتخاب مدل ها**

***Transfer Learning***Different pre-trained CNN models were applied for transfer learning over Beans datasets.  
**MobileNetV2**, the inverted residual structure of this model is built on residual connections between bottleneck layers. In the intermediate expansion layer filters, light-weight depth-wise convolutions are utilized as a source of non-linearity. The architecture of MobileNetV2 incorporates a fully convolutional layer with 32 filters, followed by 19 residual bottleneck layers.  
**EfficientNetB6**, is a CNN that uses a compound coefficient to adjust the depth, breadth, and resolution of the network consistently for improved performance. Unlike the traditional method, which changes these parameters at random, the EfficientNet scaling strategy uses a set of preset scaling coefficients to increase  
network breadth, depth, and resolution.  
**NasNet**, Neural Search Architecture (NAS) deep networks have been used in several applications where the right design is essential for optimum performance. The first study in this area was NASNet, which represents the construction of CNN as a multi-step selection problem that can be solved through deep learning.

## ۲-۲-۲. **تقویت داده**

When strengthening (augmenting) data using libraries like **Albumentations**, the goal is to improve the model's generalization by diversifying the training data.

**1. Reasons for Choosing Specific Augmentation Methods**

**a. Random Rotation and Flip**

* + Many real-world objects can appear at different angles.
  + Helps the model become invariant to orientation changes.
  + Prevents overfitting by simulating different perspectives.
  + Useful for datasets where object orientation is not fixed.

**b. Brightness, Contrast, and Hue Adjustments**

* + Real-world images often have varying lighting conditions.
  + Models trained on a single lighting condition may fail to generalize.
  + Simulates various lighting scenarios to improve robustness.

**c. Random Cropping and Resizing**

* + Forces the model to focus on different parts of the object in the image.
  + Handles cases where objects are partially visible.
  + Encourages learning of local features.

**d. Gaussian Blur or Noise**

* + Images in real applications can be blurry or noisy due to poor resolution or motion.
  + Improves the model's ability to handle noisy data.
  + Images captured under different sensors may vary in color intensity.
  + Increases generalization across devices.

**f. Elastic Transformations and Affine Transformations**

* + Applicable when the object shape can vary slightly (e.g., handwritten text or biological structures).
  + Encourages shape invariance.

**g. Random Erasing**

* + Simulates occlusion where parts of the object are hidden.
  + Forces the model to rely on global context rather than specific regions.

**Effects of Augmentation on Model Performance**

* **Robustness**: Enhances the model's ability to generalize to unseen scenarios (e.g., different lighting, orientation).
* **Prevent Overfitting**: Provides a more diverse training dataset, reducing dependency on specific patterns.
* **Resilience to Noise**: Trains the model to handle imperfections in real-world data.

## ۳-۲-۲. **تقویت داده**

Adapting the size of input images to match the requirements of deep learning models such as MobileNetV, EfficientNetB6, and NASNet is critical for achieving high accuracy and efficient processing. Adapting image sizes effectively ensures that the model utilizes its architecture optimally, balancing accuracy and computational demands. Correct image sizing aligns the dataset with the model’s requirements, ensuring consistent performance, optimal utilization of computational resources, and preservation of critical features for accurate predictions.Correct sizing of input images is crucial for achieving optimal accuracy and efficiency in deep learning models.

**Understanding Model Input Size Requirements**

Each deep learning model has specific input size requirements:

* **MobileNetV2**: Requires input sizes such as 224×224224 \times 224224×224.
* **EfficientNetB6**: Scales input sizes based on its version (e.g., EfficientNet-B0 uses 224×224224 \times 224224×224, while B6 uses 528×528528 \times 528528×528).
* **NasNet**: Typically expects 331×331331 \times 331331×331.

These sizes are determined during model design to balance feature extraction and computational cost.

**Steps to Adapt Image Sizes**

**a. Image Resizing**

Resize all images to match the required dimensions for each model. Common resizing methods include:

* **Bilinear interpolation** (default in most libraries): Smooth resizing for natural images.
* **Nearest-neighbor interpolation**: Preserves sharpness, useful for small images.
* **Library Tools**:
  + PyTorch: transforms.Resize((height, width))
  + TensorFlow: tf.image.resize(images, (height, width))

**b. Maintaining Aspect Ratio**

If resizing without distortion is essential:

* Use padding to maintain the aspect ratio, followed by resizing.
* Tools like transforms.Pad (PyTorch) or tf.image.pad\_to\_bounding\_box (TensorFlow) are helpful.

**c. Data Augmentation**

While resizing:

* Incorporate random cropping, flipping, and rotation to enhance generalization.

transform = transforms.Compose([

transforms.Resize((image\_height, image\_width)),

transforms.RandomHorizontalFlip(),

transforms.RandomCrop(size=(image\_height, image\_width)),

transforms.ToTensor(),

])

**d. Normalization**

Normalize pixel values to match the preprocessing done during the model’s training:

* MobileNet, EfficientNet, and NasNet often use the ImageNet normalization mean and std

transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

**e. Handling Varying Input Sizes**

If you need to use a model with varying input sizes:

* **Dynamic resizing**: EfficientNet supports scalable image sizes (e.g., 224×224224 \times 224224×224 to 600×600600 \times 600600×600).
* **Batches with different sizes**:
  + Use tools like torch.nn.functional.interpolate to resize batches on the fly.

images = torch.nn.functional.interpolate(images, size=(required\_height, required\_width))

**Automated Resizing Pipelines**

Automate the resizing process using:

* **PyTorch Datasets**:

python

Copy code

dataset = datasets.ImageFolder(root='data', transform=transforms.Compose([

transforms.Resize((height, width)),

transforms.ToTensor()

]))

* **TensorFlow Pipelines**:

python

Copy code

dataset = tf.data.Dataset.from\_tensor\_slices(file\_paths)

dataset = dataset.map(lambda x: preprocess\_and\_resize(x, height, width))

**5. Model-Specific Considerations**

* **MobileNetV2**:
  + Lightweight and efficient; stick to 224×224224 \times 224224×224 for fast processing.
* **EfficientNetB**:
  + Match input size to the version (e.g., B0: 224224224, B6: 528528528).
  + Ensure batch sizes align with memory constraints.
* **NasNet**:
  + Larger sizes like 331×331331 \times 331331×331 maximize performance but demand more resources.

**Model-Specific Considerations**

* **MobileNetV2**:
  + Lightweight and efficient; stick to 224×224224 \times 224224×224 for fast processing.
* **EfficientNetB**:
  + Match input size to the version (e.g., B0: 224224224, B6: 528528528).
  + Ensure batch sizes align with memory constraints.
* **NasNet**:
  + Larger sizes like 331×331331 \times 331331×331 maximize performance but demand more resources.

**1. Impact on Accuracy**

**a. Information Preservation**

* **Too Small Images**:
  + Reducing the image size too much during resizing can lead to the loss of important features (e.g., edges, textures).
  + Missing critical details negatively affects the model’s ability to learn meaningful patterns.
* **Too Large Images**:
  + Overly large images may include irrelevant or redundant details, introducing noise that hampers model performance.

**b. Model Architecture Design**

* Deep learning models are designed with specific input sizes in mind:
  + For example, **EfficientNet-B6** expects 528×528528 \times 528528×528, while **MobileNetV2** expects 224×224224 \times 224224×224.
  + Feeding images of incorrect sizes disrupts layer compatibility, leading to poor feature extraction and degraded performance.

**c. Feature Representation**

* Correct image sizing ensures that features are represented in a way that aligns with how the model was pre-trained (e.g., on ImageNet).
  + If the input size differs significantly, the model might fail to recognize patterns it was trained on, reducing transfer learning benefits.

**d. Uniformity Across the Dataset**

* Models perform better when the input data is consistent.
  + If images vary in size, the model faces difficulty learning consistent patterns.
  + Standardizing image sizes prevents this issue.

**2. Impact on Efficiency**

**a. Computational Costs**

* **Larger Images**:
  + Require more memory and computational power during training and inference.
  + Increase processing time, as convolutional operations scale with input size.
* **Smaller Images**:
  + Reduce computational requirements but at the risk of losing important details, leading to a trade-off.

**b. GPU/Memory Utilization**

* Incorrectly sized images can lead to:
  + **Out-of-memory errors**: Large image sizes, especially in models like **EfficientNet-B6**, can exhaust GPU memory.
  + **Underutilization**: Too-small images may lead to inefficient GPU usage, wasting computational capacity.

**c. Batch Processing**

* Uniformly sized images allow efficient batching of data.
  + If images vary in size, padding or resizing on-the-fly can slow down training or inference pipelines.

**3. Preprocessing and Consistency with Model Training**

Deep learning models are often pre-trained on large datasets (e.g., ImageNet) with specific input sizes. If the input images during fine-tuning or inference do not match these sizes:

* **Feature mismatch**: The features extracted during training may not align with those extracted from resized inputs.
* **Reduced transfer learning benefits**: Pre-trained weights lose effectiveness, leading to slower convergence and lower accuracy.

**4. Trade-Offs Between Accuracy and Efficiency**

Choosing the correct input size involves balancing:

**Accuracy**:

* + Larger images often improve accuracy for complex datasets but may increase the risk of overfitting.

**Efficiency**:

* + Smaller sizes enhance speed and reduce resource demands but may result in poorer feature representation.

**5. Tools for Effective Image Sizing**

To ensure correct sizing:

**Resizing during Preprocessing**:

* + Use libraries like PyTorch (transforms.Resize) or TensorFlow (tf.image.resize).

**Data Augmentation**:

* + Crop, flip, or pad images to preserve details while maintaining the target size.

**Dynamic Resizing**:

* + For models like EfficientNet, which support flexible input sizes, adjust image sizes based on computational resources.

## ۴-۲-۲. **بهینه سازها**

Pretrained Models:

1. EfficientNetB6:
   * Part of the EfficientNet family, EfficientNetB6 is designed for high performance and efficiency. These models scale up more effectively than traditional convolutional neural networks (CNNs), making them excellent for tasks like image classification. EfficientNet models are known for their accuracy and low computational costs.
2. MobileNetV2:
   * MobileNetV2 is optimized for mobile and embedded vision applications. It's designed to be lightweight and efficient, making it suitable for devices with limited computational power. Despite being more compact, MobileNetV2 delivers reasonably good performance for image classification tasks.
3. NasNet:
   * NasNet (Neural Architecture Search Network) models are the result of a novel approach called neural architecture search, which automates the process of designing neural networks. NasNet models are optimized for image classification and achieve high accuracy. They are known for being both powerful and flexible.

Optimizers:

1. Adam (Adaptive Moment Estimation):
   * Adam is a popular optimization algorithm in the field of machine learning. It combines the advantages of two other extensions of stochastic gradient descent (SGD), namely AdaGrad and RMSProp. Adam is well-suited for problems with sparse gradients and noisy data, making it effective for a wide range of applications.
2. RMSProp (Root Mean Square Propagation):
   * RMSProp is an adaptive learning rate method designed to tackle diminishing learning rates. It works by adjusting the learning rate for each parameter based on recent gradients, which helps in speeding up the training process. RMSProp is particularly useful for recurrent neural networks and tasks involving non-stationary targets.
3. Nadam (Nesterov-accelerated Adaptive Moment Estimation):
   * Nadam combines the features of Adam and Nesterov accelerated gradient. It uses the momentum concept from Nesterov and adapts it into Adam’s framework, resulting in faster convergence and improved performance over standard Adam in some cases. This makes Nadam a powerful optimizer for training deep learning models.

Each of these models and optimizers has its strengths and is suited for different types of tasks and hardware constraints. The choice of model and optimizer can significantly impact the performance of a neural network.

## ۵-۲-۲. **آموزش مدل**

1. **EfficientNetB6**:
   * When paired with the Adam optimizer, it achieves outstanding results with very low losses and very high accuracies, making it a standout performer.
   * With the RMSProp optimizer, the validation accuracy hits a perfect 1.0000, but the training loss is significantly higher, suggesting possible overfitting.
   * The Nadam optimizer yields similar excellent results as Adam, with very low losses and high accuracies.
2. **MobileNetV2**:
   * The performance varies significantly across different rows, with the optimizer not being specified for some entries.
   * With RMSProp, it shows the highest losses and the lowest accuracies, indicating poor performance with this configuration.
3. **NasNet**:
   * Displays reasonably good performance, with losses and accuracies better than MobileNetV2 but generally not as high-performing as EfficientNetB6.
   * The RMSProp optimizer yields slightly better results in terms of validation accuracy compared to MobileNetV2 but still not on par with EfficientNetB6.

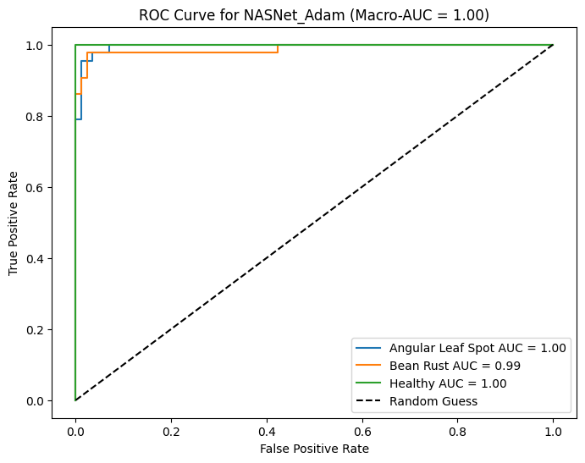
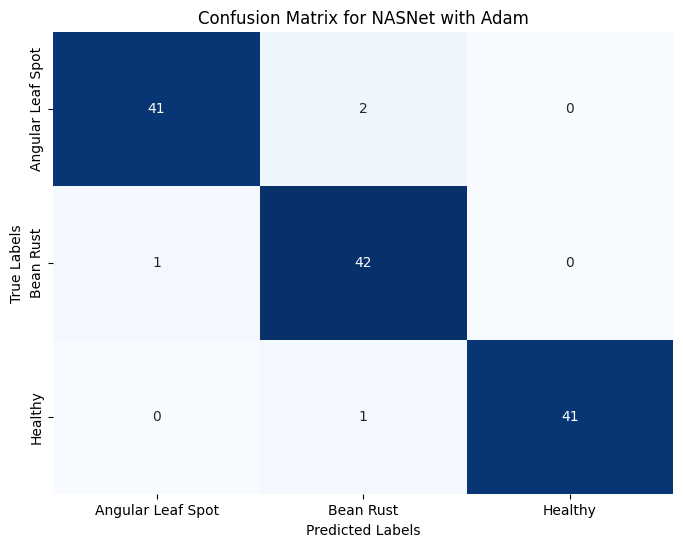
**Results for NASNet\_Adam:**

Test Accuracy: 0.9688

Precision: 0.9696

Recall: 0.9688

F1 Score: 0.9689



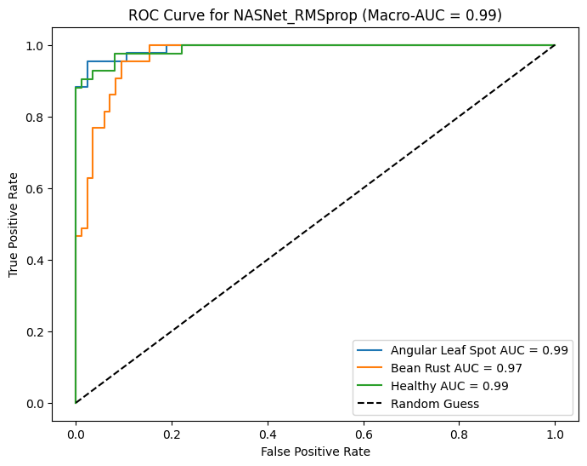
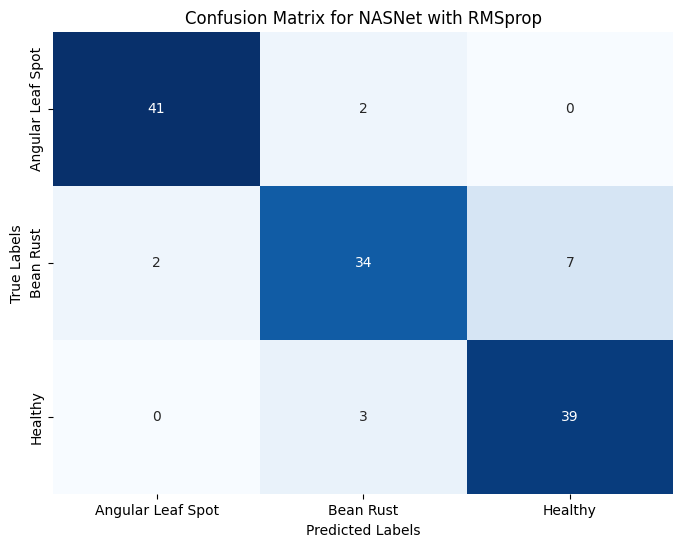
**Results for NASNet\_RMSprop:**

Test Accuracy: 0.8906

Precision: 0.8914

Recall: 0.8906

F1 Score: 0.8897



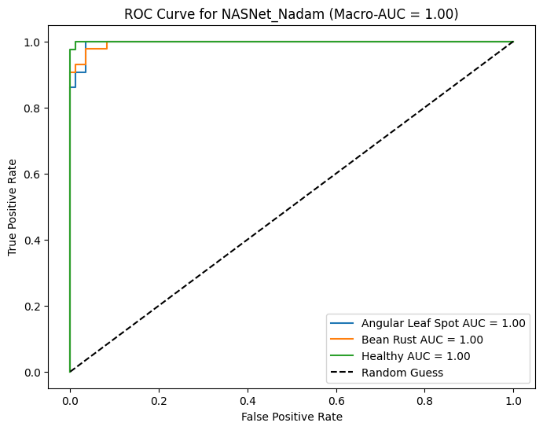
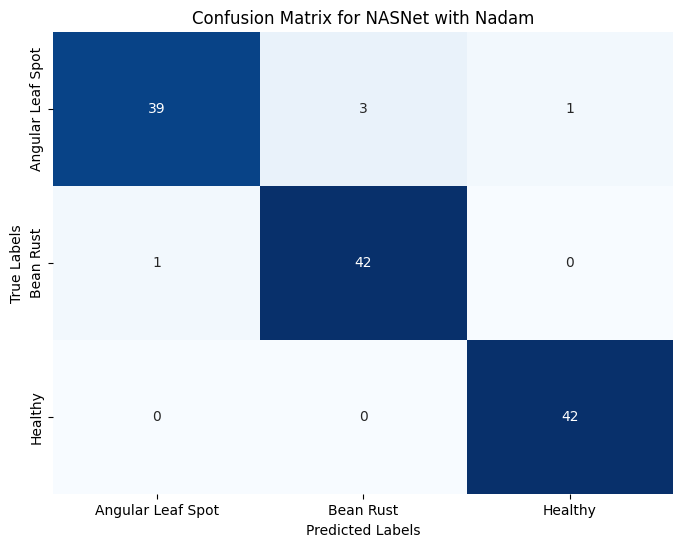
**Results for NASNet\_Nadam:**

Test Accuracy: 0.9609

Precision: 0.9616

Recall: 0.9609

F1 Score: 0.9606



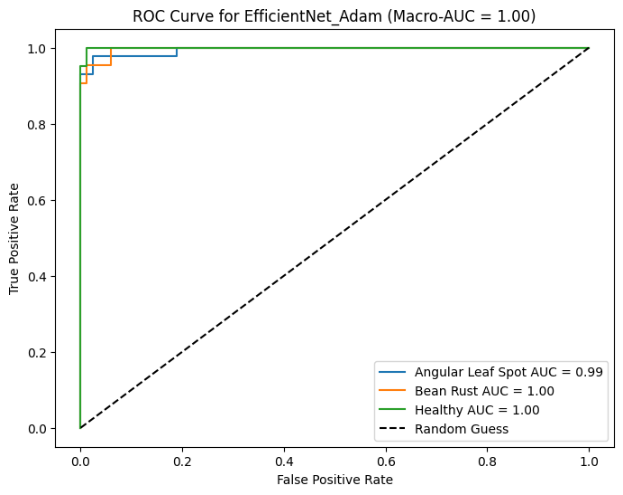
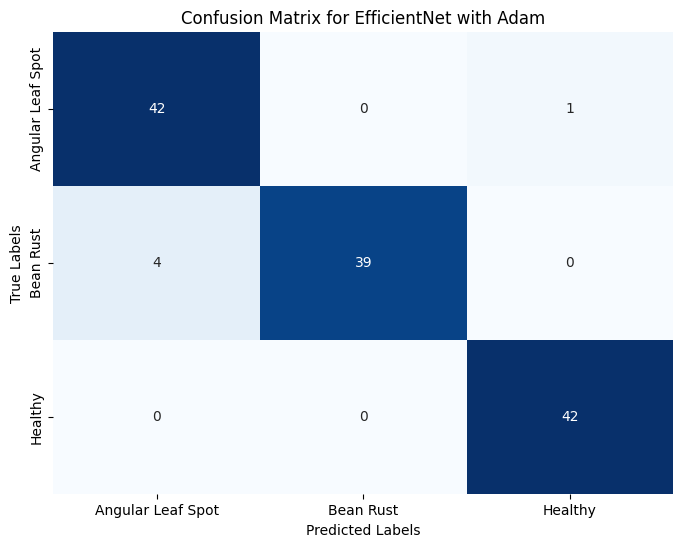
**Results for EfficientNet\_Adam:**

Test Accuracy: 0.9609

Precision: 0.9632

Recall: 0.9609

F1 Score: 0.9609



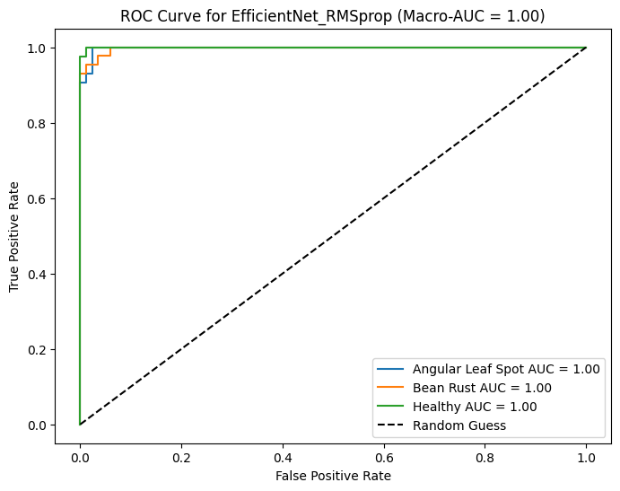
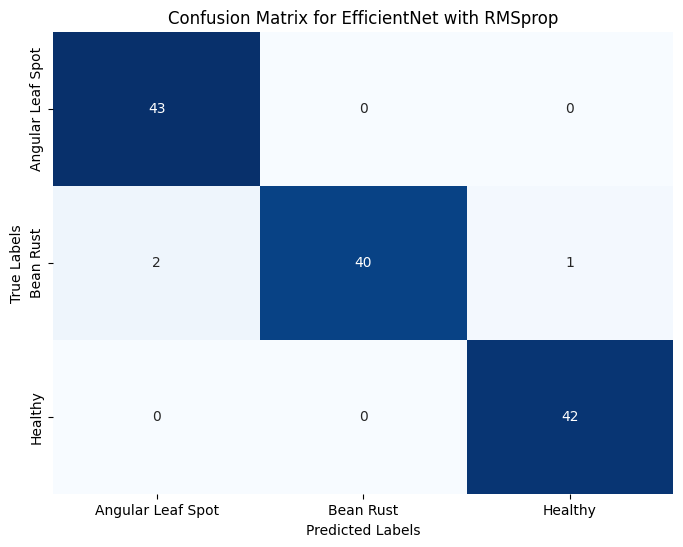
**Results for EfficientNet\_RMSprop:**

Test Accuracy: 0.9766

Precision: 0.9774

Recall: 0.9766

F1 Score: 0.9764

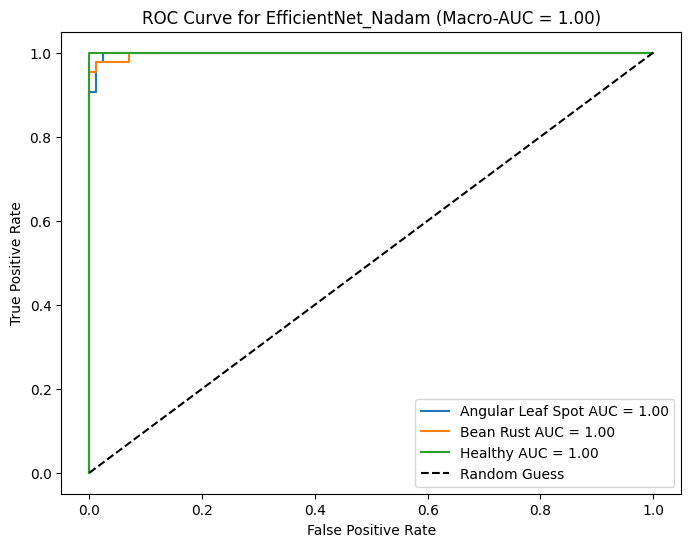
**Results for EfficientNet\_Nadam:**

Test Accuracy: 0.9844

Precision: 0.9851

Recall: 0.9844

F1 Score: 0.9844



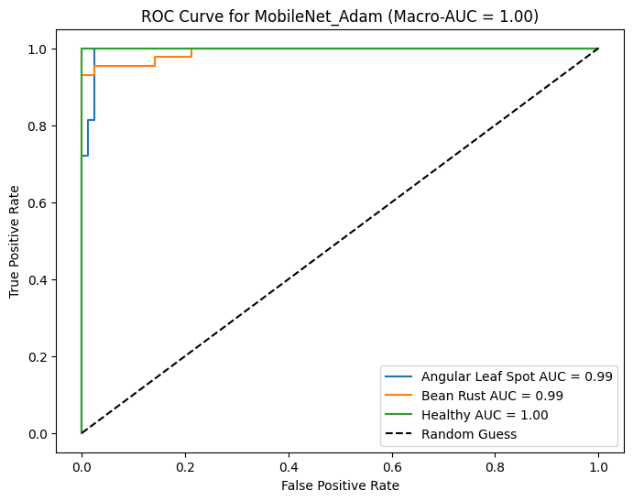
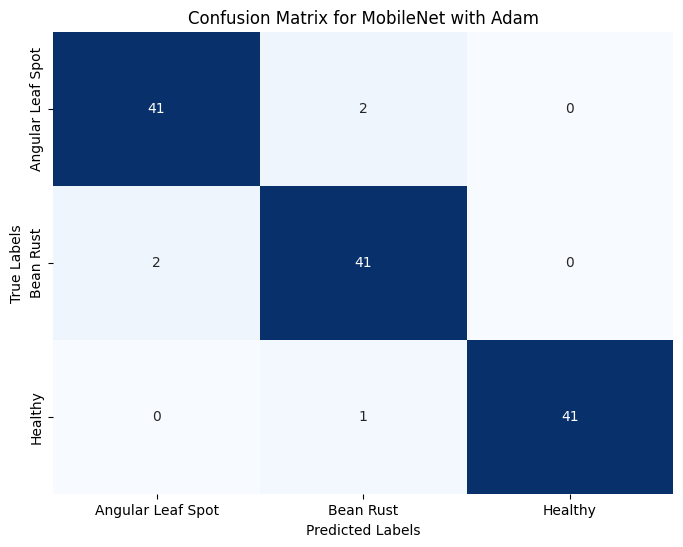
**Results for MobileNet\_Adam:**

Test Accuracy: 0.9609

Precision: 0.9615

Recall: 0.9609

F1 Score: 0.9611



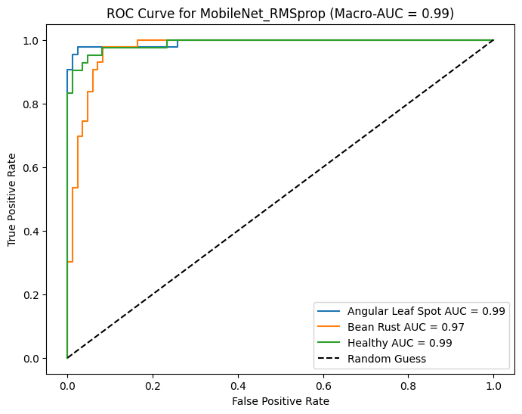
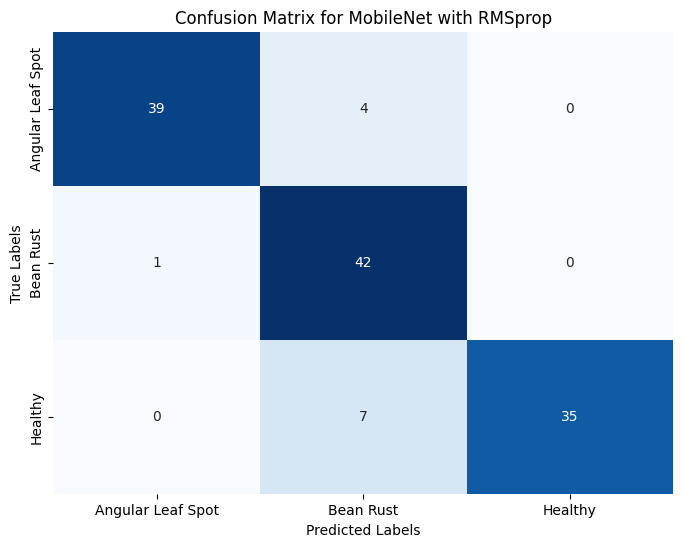
**Results for MobileNet\_RMSprop:**

Test Accuracy: 0.9062

Precision: 0.9219

Recall: 0.9062

F1 Score: 0.9079



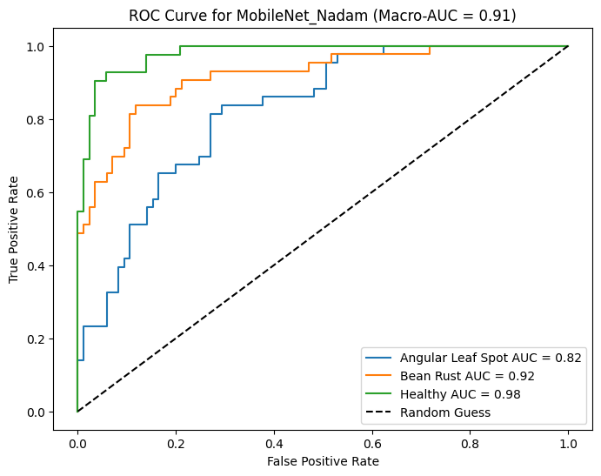
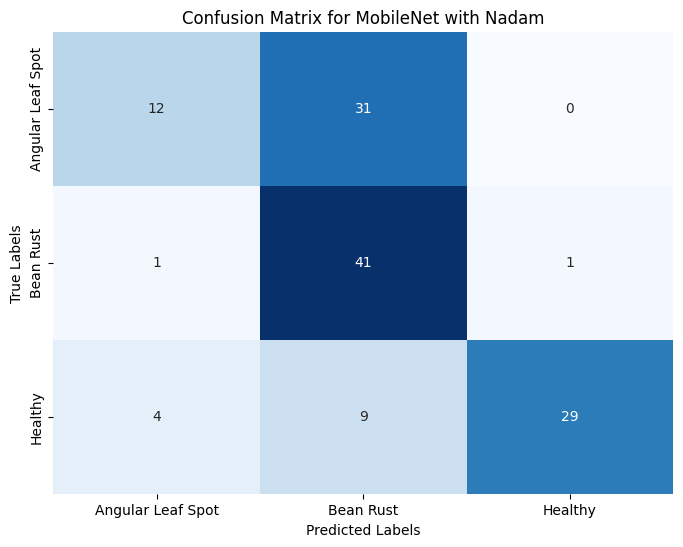
**Results for MobileNet\_Nadam:**

Test Accuracy: 0.6406

Precision: 0.7244

Recall: 0.6406

F1 Score: 0.6209



## ۳-۲. **تحلیل نتایج**

The table shows the performance metrics of various Convolutional Neural Network (CNN) models using different optimizers. The models evaluated are EfficientNetB6, MobileNetV2, and NasNet, with optimizers Adam, RMSProp, and Nadam. The performance metrics include validation loss (Val-loss), training loss (Tr-Loss), validation accuracy (Val-Acc), and training accuracy (Tr-ACC).

* **EfficientNetB6 with Adam and Nadam optimizers** shows low validation and training losses, and high validation and training accuracies, indicating very good performance.
* **EfficientNetB6 with RMSProp optimizer** also has excellent accuracy, but with higher training loss.
* **MobileNetV2 and NasNet models** generally show higher validation losses and slightly lower accuracies compared to EfficientNetB6.
* **MobileNetV2 with RMSProp optimizer** performed the worst with the highest losses and lowest accuracies.

**CNN Models and Optimizers**

1. **EfficientNetB6**:
   * **Adam Optimizer**:
     + Validation Loss: 0.0113
     + Training Loss: 0.0226
     + Validation Accuracy: 0.9925
     + Training Accuracy: 0.9952
   * **RMSProp Optimizer**:
     + Validation Loss: 0.0143
     + Training Loss: 0.2015
     + Validation Accuracy: 1.0000
     + Training Accuracy: 0.9342
   * **Nadam Optimizer**:
     + Validation Loss: 0.0163
     + Training Loss: 0.0140
     + Validation Accuracy: 0.9925
     + Training Accuracy: 0.9961
2. **MobileNetV2**:
   * **(Optimizer not specified for some rows)**:
     + Validation Loss: 0.0656, Training Loss: 0.0624
     + Validation Accuracy: 0.9774, Training Accuracy: 0.9797
     + Validation Loss: 0.2072, Training Loss: 0.2077
     + Validation Accuracy: 0.9398, Training Accuracy: 0.9304
   * **RMSProp Optimizer**:
     + Validation Loss: 0.8560, Training Loss: 1.7142
     + Validation Accuracy: 0.6842, Training Accuracy: 0.3868
3. **NasNet**:
   * **(Optimizer not specified for some rows)**:
     + Validation Loss: 0.0631, Training Loss: 0.0577
     + Validation Accuracy: 0.9850, Training Accuracy: 0.9855
     + Validation Loss: 0.1624, Training Loss: 0.3026
     + Validation Accuracy: 0.9474, Training Accuracy: 0.8936
   * **RMSProp Optimizer**:
     + Validation Loss: 0.0883, Training Loss: 0.0540
     + Validation Accuracy: 0.9699, Training Accuracy: 0.9749

**Conclusion:**

EfficientNetB6 seems to perform best, especially with the Adam optimizer. On the other hand, MobileNetV2 appears to have the least effective performance.

EfficientNetB6, particularly with the Adam and Nadam optimizers, demonstrates superior performance across the metrics, making it the most effective model-optimizer pair in this comparison. MobileNetV2 shows the least effective performance, especially with the RMSProp optimizer.

4Bean Leaf Results Comparision

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Val-loss | Tr-Loss | Val-Acc | Tr-ACC | CNN Model | Optimizer |
| 0.0113 | 0.0226 | 0.9925 | 0.9952 | EfficientNetB6 | Adam |
| 0.0656 | 0.0624 | 0.9774 | 0.9797 | MobileNetV2 |
| 0.0631 | 0.0577 | 0.9850 | 0.9855 | NasNet |
| 0.0143 | 0.2015 | 1.0000 | 0.9342 | EfficientNetB6 | RMSProp |
| 0.2072 | 0.2077 | 0.9398 | 0.9304 | MobileNetV2 |
| 0.1624 | 0.3026 | 0.9474 | 0.8936 | NasNet |
| 0.0163 | 0.0140 | 0.9925 | 0.9961 | EfficientNetB6 | Nadam |
| 0.8560 | 1.7142 | 0.6842 | 0.3868 | MobileNetV2 |
| 0.0883 | 0.0540 | 0.9699 | 0.9749 | NasNet |

Thank You