



دانشگاه تهران
دانشکده مهندسی برق و
کامپیوتر



درس شبکه‌های عصبی و یادگیری عمیق
تمرین دوم

- 810100206 محمدرضا محمدهاشمی
810100216 مبینا مهرآذر -

فهرست

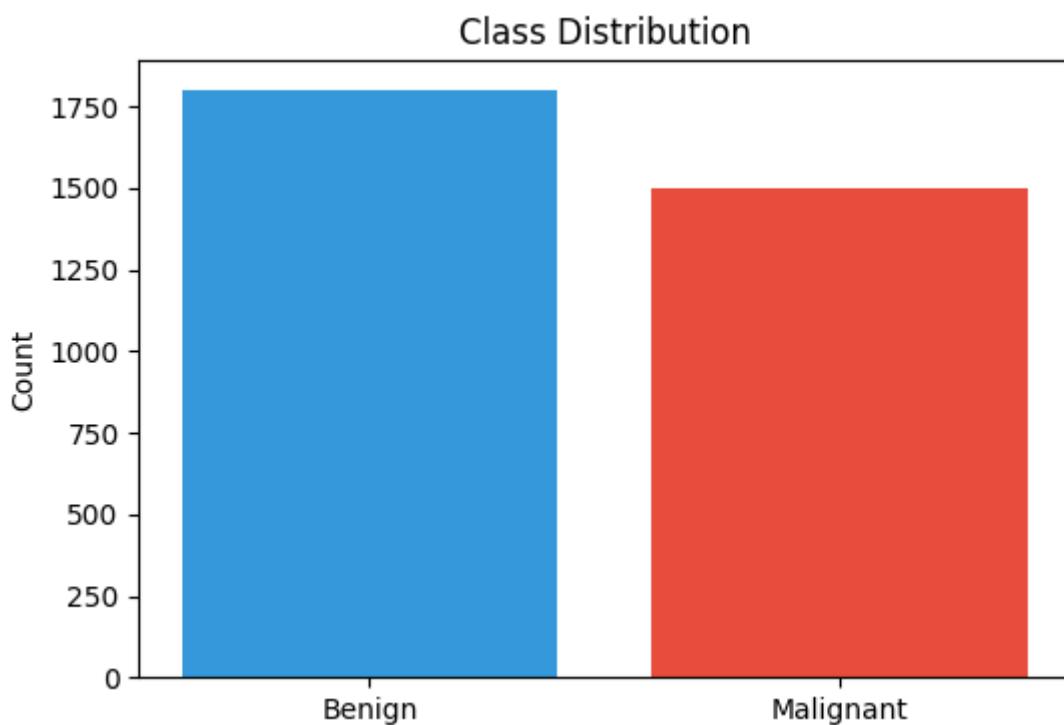
1.....	پرسش 1 . تشخیص ضایعه سرطانی با استفاده از CNN
1.....	1-1. معرفی مقاله
1.....	1-1-1. توزیع دادهها
1.....	1-2-1. پیش پردازش تصاویر
1.....	1-2-1. متداول کردن تعداد دادههای مجموعه داده
2.....	1-2-1. یکسانسازی سایز تصاویر مجموعه داده
3.....	1-2-1. نرمالسازی پیکسلهای تصاویر مجموعه داده
3.....	1-3. داده افزایی (Data augmentation)
5.....	4-1. پیاده‌سازی
15.....	4-5. تحلیل نتایج
15.....	4-6. مقایسه نتایج
15.....	4-7. مدل عمیق‌تر
22.....	پرسش 2 - تشخیص بیماری‌های برگ لوبيا با شبکه‌های عصبی
22.....	2-1. پیش‌پردازش تصاویر
24.....	2-2. پیاده‌سازی
24.....	2-2-1. انتخاب مدل‌ها
27.....	2-2-2. تقویت داده
28.....	3-2-2. بهینه‌سازها
28.....	2-2-4. آموزش مدل
32.....	2-3. تحلیل نتایج
32.....	nasNet: مدل
35.....	EfficientNetB6: مدل
37.....	MobileNetV2 : مدل

پرسش 1. تشخیص ضایعه سرطانی با استفاده از CNN

1-1. معرفی مقاله

1-1-1. توزیع داده‌ها

مجموعه داده استفاده شده در این سوال، از Kaggle یا به طور مخفف، ISIC Images می‌باشد. این مجموعه داده شامل نمونه‌های تصویری از ضایعات پوستی خوش خیم (benign) و بدخیم (malignant) است. توزیع کلاس‌های این مجموعه داده به صورت زیر می‌باشد:



شکل 1-1. توزیع کلاس‌های مجموعه داده

در این بخش، مدلی مانند مدل مقاله پیاده‌سازی کردہ‌ایم که ساختار زیر را دارد:

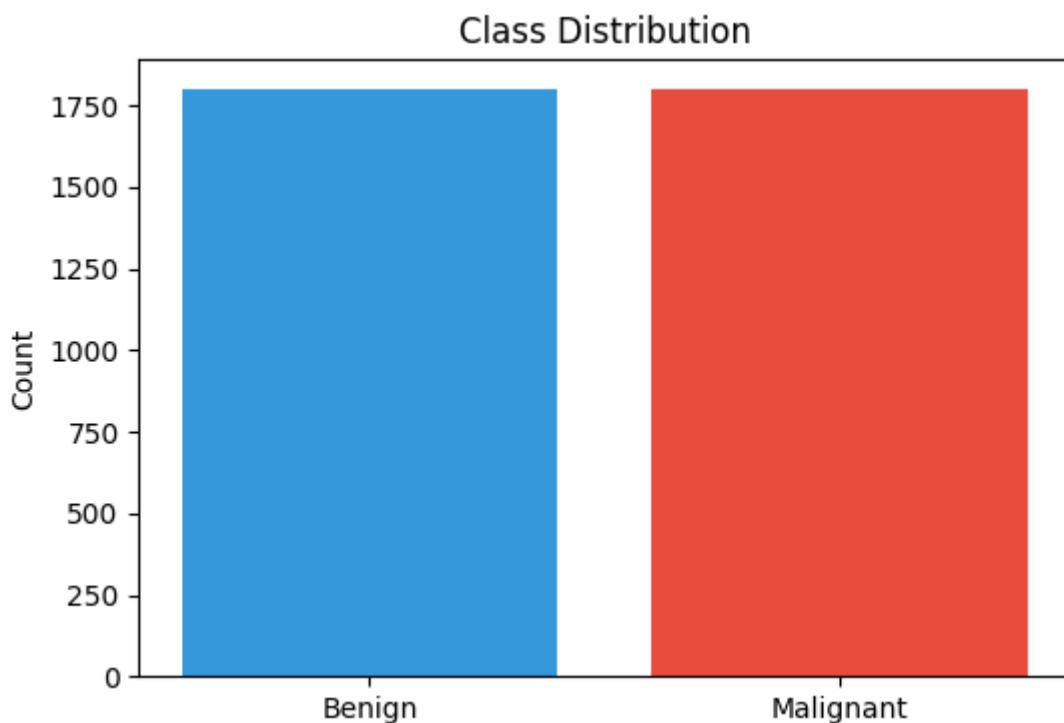
2-1. پیش‌پردازش تصاویر

2-1-1. متعادل کردن تعداد داده‌های مجموعه داده

از آنجایی که در بخش توزیع داده‌ها مشاهده کردیم، مجموعه داده ما بالانس نبود. این اتفاق در مدل‌های CNN مشکل Bias ایجاد می‌کند. در واقع مدل به سمت کلاس پر جمعیت‌تر

سوگیری کرده و باعث می‌شود نتواند کلاس اقلیت را به درستی شناسایی کند و پیش‌بینی مدل را ضعیف می‌کند. در این حالت معیارهای Accuracy و Loss هم نتیجه خوبی نخواهند داشت.

برای انجام upsample، یا همان oversampling، در کلاس Dataset، متدهای upsample را تعریف کردہ‌ایم. اگر تعداد داده‌های دو کلاس برابر نباشد، کلاس با داده کمتر را در نظر گرفته و و به اندازه‌ی اختلاف تعداد آن با تعداد داده‌های کلاس دیگر، به آن تصویر از همان مجموعه داده اضافه می‌کند. توجه کنید که به علت اعمال transform بر روی تصاویر این مجموعه داده و نحوه تصادفی بودن نوع transform انجام شده، در نهایت تصاویر تکراری، با فرمت متفاوتی وارد مدل می‌شوند.



شکل ۱-۱. توزیع کلاس‌های مجموعه داده بعد از oversampling

۱-۲-۱. یکسان‌سازی سایز تصاویر مجموعه داده

ابعاد تمامی تصاویر ورودی مدل‌های CNN باید یکسان باشند تا مدل ساختار معماري خود را در بخش‌های pooling و ابعاد filter، ثابت کند و بر روی تمامی تصاویر پاسخگو باشد و در نهایت ویژگی‌های تصاویر را به صورت خودکار و با روش یکسان از تمامی تصاویر بگیرد و فرآیند یادگیری انجام شود. برای اطمینان یافتن از این مورد، در مدل اول و دوم، سایز تصاویر را به 28×28 و در مدل GoogLeNet به 224×224 تغییر می‌دهیم. علت این مورد هم این است که در انتخاب

دیزاین گوگل نت، برای اینکه ورودی خیلی برای مدل سنگین نباشد و همینطور رزو لوشن تصاویر خیلی پایین نباشد، ابعاد ورودی را به این مقدار تغییر می‌دهند.

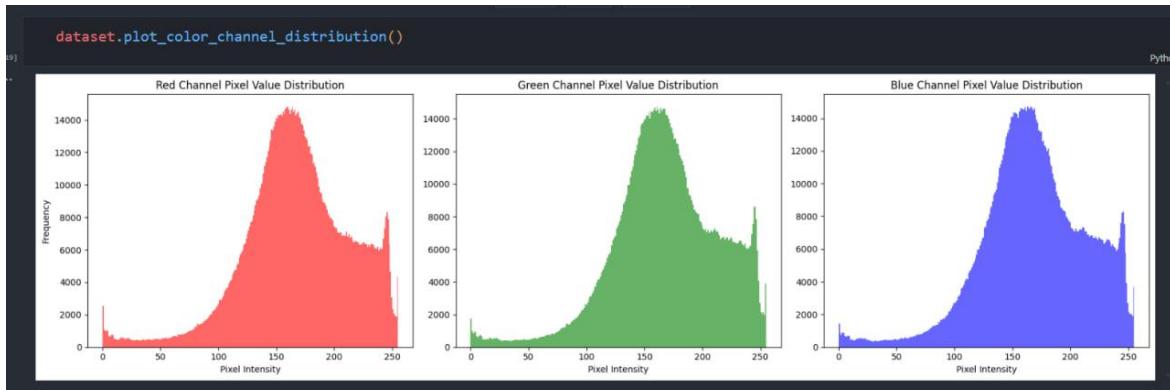
1-2-1. نرمال‌سازی پیکسل‌های تصاویر مجموعه داده

برای فراهم کردن شرایط برای یادگیری دقیق‌تر و سریع‌تر مدل‌های CNN، مقادیر پیکسل‌های موجود در هر سه رنگ Red, Green, Blue در تصاویر را به بازه‌ی (0,1) تغییر می‌دهیم. انجام این نرمال‌سازی به مدل کمک می‌کند تا ویژگی‌های مهم‌تر و موثرتری را از تصاویر استخراج کند و یاد بگیرد. همینطور واریانس موجود در مقادیر ورودی کاهش یافته و در بازه‌ی مطلوب قرار می‌گیرند. در نتیجه مدل سریع‌تر در فرایند Back Propagation به وزن‌های بهتر همگرا می‌شود. برای نرمال‌سازی متدهای Dataset normalize در کلاس normalize مقدار 255 تقسیم کردہ‌ایم.

```
def normalize(self):
    self.images = self.images/255.0
```

شکل 1-1. نرمال‌سازی ابعاد پیکسل‌های تصاویر

توزیع مقادیر پیکسل‌ها را بعد از نرمال‌سازی بررسی می‌کنیم:



همانطور که مشاهده می‌شود این مقادیر علیرغم چولگی موجود توزیع نسبتاً مناسبی دارند.

1-3. داده افزایی (Data augmentation)

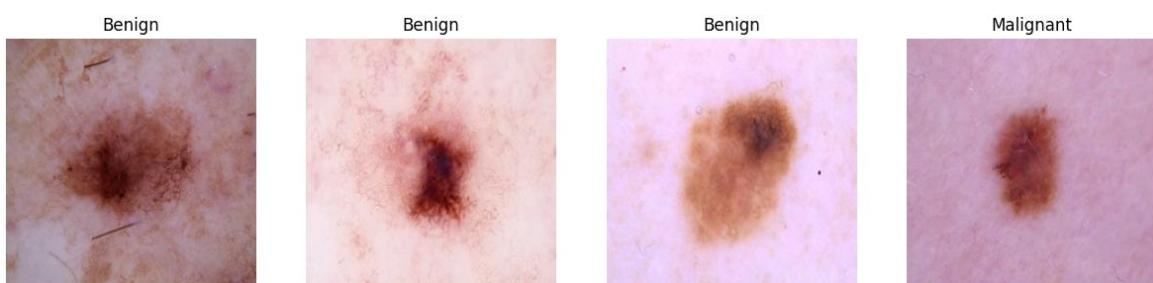
به صورت کلی تکنیک‌های تقویت داده با هدف ایجاد تغییراتی در داده ورودی به نوعی که شرایط مختلف و دامنه وسیع ورودی در شرایط کلی را بهتر مدل کند انجام می‌شود. مختلفی بر

اساس ماهیت و شرایط دیتاست جهت تقویت داده قابل انجام است. در عکس های جنرال تر تغییر زوایا و زوم می تواند تاثیر مناسبی داشته باشد و در عکس های پزشکی عکس ها را کمی دفرمه میکنند میتوانند کمک کنند تا شرایط مختلف بیشتری در عکس ها به دست آید و مدل جنرال تر ترین شود.

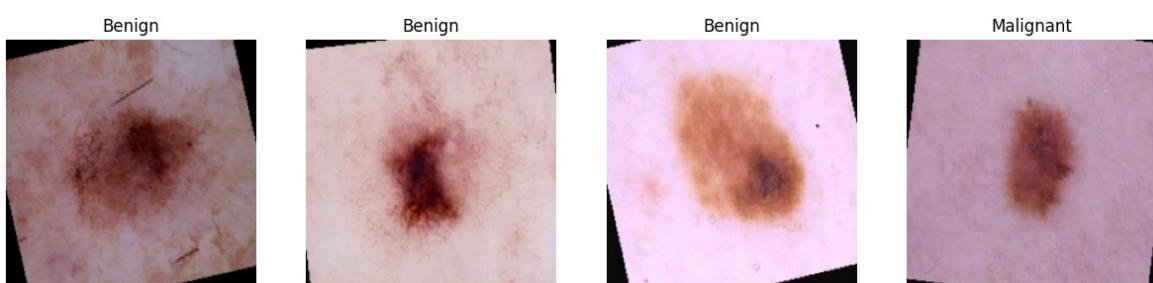
ترنسفورمیشن های انجام شده بر روی مجموعه داده جهت ترین کردن مدل عبارت اند از:

- **RandomHorizontalFlip, RandomVerticalFlip**: در این ترنسفورمیشن به صورت رندوم عکس را در جهت افقی و عمودی فلیپ می کنیم تا حساسیت به جهت عکس در مدل ایجاد نشود و مدل بر اساس جهتی خاص ترین نشود.
- **(15)RandomRotation**: به صورت رندوم تا 15 درجه در یک جهتی می چرخاند که باعث میشود حساسیت مدل به زاویه عکس از بین برود.
- **transforms.RandomResizedCrop**: به عنوان ورودی این تابع، از 0.9 (28, $scale = (0.9, 1.1)$) استفاده می کنیم. در این ترنسفورمیشن عکس را تا 1.1 انژوم و تا 1.1 زوم این میکنیم تا مدل حساس به سایز سوژه ترین نشود و حالت جنرال تری را مدل کند.
- **ColorJitter**: در این ترنسفورمیشن ویژگی هایی مانند برایتنس و کنتراست عکس را به مقدار داده شده به صورت رندوم تغییر میدهیم تا مدل به شرایط نوری و کنتراست های مختلف اشنا شود.

در این بخش، نمونه ای از چند تصویر بعد از فرایند داده افزایی را نشان می دهد:



شکل 1-2. نمونه عکس های قبل از داده افزایی شده



شکل 1-2. نمونه عکس های بعد از داده افزایی شده

4-1. پیاده‌سازی

در مرحله پیاده‌سازی مدل، کلاس دیتا ستی که از کلاس دیتابست اینهیریت شده است و نیازمندی‌های ما را پیاده کرده است پیاده سازی کردیم.

```
class Data(Dataset):
    Tabnine | Edit | Test | Explain | Document | Ask
    def __init__(self, device, img_size_w = 224, img_size_h = 224, up_sample=True):
        self.img_size_w = img_size_w
        self.img_size_h = img_size_h
        self.device = device
        self.mode = 'train'

        self.load_data(up_sample)
        self.split_data()

    Tabnine | Edit | Test | Explain | Document | Ask
    def set_mode(self, mode):
        self.mode = mode

    Tabnine | Edit | Test | Explain | Document | Ask
    def set_transformers(self, train_transform=None, val_transform=None):
        self.train_transform = train_transform if train_transform else None
        self.val_transform = val_transform if val_transform else None
```

```
def split_data(self):
    X_train, X_test, y_train, y_test = train_test_split(
        self.images, self.labels, test_size=0.2, random_state=42
    )

    X_train, X_val, y_train, y_val = train_test_split(
        X_train, y_train, test_size=0.2, random_state=42
    )

    self.X_train, self.y_train = X_train, y_train
    self.X_val, self.y_val = X_val, y_val
    self.X_test, self.y_test = X_test, y_test
```

```
def load_data(self, up_sample):
    benign = load_from_dataset(BENIGN_PATH, self.img_size_w, self.img_size_h)
    malignant = load_from_dataset(MALIGNANT_PATH, self.img_size_w, self.img_size_h)

    benign = np.array(benign, dtype=np.float32)
    malignant = np.array(malignant, dtype=np.float32)
    print(f'dataset shape is : {benign.shape}')

    benign_labels = np.zeros(benign.shape[0], dtype=np.float32)
    malignant_labels = np.full(malignant.shape[0], 1.0)

    if (up_sample):
        benign, malignant, benign_labels, malignant_labels = \
            self.upsample(benign, malignant, benign_labels, malignant_labels)

    self.images = np.concatenate((benign, malignant), axis=0)
    self.labels = np.concatenate((benign_labels, malignant_labels), axis=0)
    self.analyze_class_distribution()
```

```

def upsample(self, benign, malignant, benign_labels, malignant_labels):
    diff = abs(benign.shape[0] - malignant.shape[0])

    if benign.shape[0] > malignant.shape[0]:
        malignant_upsampled = np.concatenate([malignant] * (diff // malignant.shape[0]) +
                                              [malignant[:diff % malignant.shape[0]]], axis=0)
        malignant_labels_upsampled = np.concatenate([malignant_labels] * (diff // malignant.shape[0]) +
                                                   [malignant_labels[:diff % malignant.shape[0]]], axis=0)
        malignant = np.concatenate([malignant, malignant_upsampled], axis=0)
        malignant_labels = np.concatenate([malignant_labels, malignant_labels_upsampled], axis=0)
    else:
        benign_upsampled = np.concatenate([benign] * (diff // benign.shape[0]) +
                                         [benign[:diff % benign.shape[0]]], axis=0)
        benign_labels_upsampled = np.concatenate([benign_labels] * (diff // benign.shape[0]) +
                                               [benign_labels[:diff % benign.shape[0]]], axis=0)
        benign = np.concatenate([benign, benign_upsampled], axis=0)
        benign_labels = np.concatenate([benign_labels, benign_labels_upsampled], axis=0)

    return benign, malignant, benign_labels, malignant_labels

```

```

def train_test_split(self):
    self.X_train, self.X_test, self.y_train, self.y_test = \
        train_test_split(self.images, self.labels, test_size=0.2, random_state=42)

Tabnine | Edit | Test | Explain | Document | Ask
def train_val_split(self):
    self.X_train, self.X_val, self.y_train, self.y_val = \
        train_test_split(self.images, self.labels, test_size=0.2, random_state=42)

```

```

def __len__(self):
    if self.mode == 'train':
        return self.X_train.shape[0]
    elif self.mode == 'val':
        return self.X_val.shape[0]
    elif self.mode == 'test':
        return self.X_test.shape[0]

Tabnine | Edit | Test | Explain | Document | Ask
def __getitem__(self, idx):
    if self.mode == 'train':
        image, label = self.X_train[idx], self.y_train[idx]
        image = self.train_transform(image)
    elif self.mode == 'val':
        image, label = self.X_val[idx], self.y_val[idx]
        image = self.val_transform(image)
    elif self.mode == 'test':
        image, label = self.X_test[idx], self.y_test[idx]
        image = self.val_transform(image)

    label = torch.tensor(label, dtype=torch.float32).to(device)
    return {'image': image, 'label': label}

```

```

def normalize(self):
    self.images = self.images/255.0

Tabnine | Edit | Test | Explain | Document | Ask
def analyze_class_distribution(self):
    benign_count = np.sum(self.labels == 0)
    malignant_count = np.sum(self.labels == 1)

    print(f"Class Distribution:")
    print(f" Benign: {benign_count}")
    print(f" Malignant: {malignant_count}")

    plt.figure(figsize=(6, 4))
    colors = ['#3498db', '#e74c3c'] # Hex colors: blue for benign, red for malignant
    plt.bar(['Benign', 'Malignant'], [benign_count, malignant_count], color=colors)
    plt.title('Class Distribution')
    plt.ylabel('Count')
    plt.show()

```

```

Tabnine | Edit | Test | Explain | Document | Ask
def show_random_images(self, indices):
    num_images = len(indices)
    fig, axes = plt.subplots(1, num_images, figsize=(15, 5))
    for i in range(num_images):
        idx = indices[i]
        image = self.images[idx].astype(np.uint8)
        label = 'Benign' if self.labels[idx] == 0.0 else 'Malignant'
        axes[i].imshow(image)
        axes[i].set_title(label)
        axes[i].axis('off')
    plt.show()

Tabnine | Edit | Test | Explain | Document | Ask
def visualize_augmentations(self, indices):
    num_images = len(indices)
    fig, axes = plt.subplots(1, num_images, figsize=(15, 5))
    for i in range(num_images):
        idx = indices[i]
        image = self.images[idx].astype(np.uint8)
        image = self.train_transform(image)
        image = image.permute(1, 2, 0).cpu().numpy()
        label = 'Benign' if self.labels[idx] == 0.0 else 'Malignant'
        axes[i].imshow(image)
        axes[i].set_title(label)
        axes[i].axis('off')
    plt.show()

def plot_color_channel_distribution(self):
    red_channel = []
    green_channel = []
    blue_channel = []
    if self.mode == 'train':
        images = self.X_train
    elif self.mode == 'val':
        images = self.X_val
    elif self.mode == 'test':
        images = self.X_test
    for img in images:
        img = torch.from_numpy(img).permute(1, 2, 0).numpy()
        red_channel.extend(img[:, :, 0].flatten())
        green_channel.extend(img[:, :, 1].flatten())
        blue_channel.extend(img[:, :, 2].flatten())
    plt.figure(figsize=(18, 5))
    plt.subplot(1, 3, 1)
    plt.hist(red_channel, bins=256, color='red', alpha=0.6)
    plt.title('Red Channel Pixel Value Distribution')
    plt.xlabel('Pixel Intensity')
    plt.ylabel('Frequency')
    plt.subplot(1, 3, 2)
    plt.hist(green_channel, bins=256, color='green', alpha=0.6)
    plt.title('Green Channel Pixel Value Distribution')
    plt.xlabel('Pixel Intensity')
    plt.subplot(1, 3, 3)
    plt.hist(blue_channel, bins=256, color='blue', alpha=0.6)
    plt.title('Blue Channel Pixel Value Distribution')
    plt.xlabel('Pixel Intensity')
    plt.tight_layout()
    plt.show()

```

همانطور که مشاهده میشود در این کلاس به کمک متدهای `show_random_images` و `visualize_augmentations` میتوانیم تصاویر را نمایش داده و تغییراتی که بر روی آنها اعمال شده باشند را مشاهده کنیم. همچنین میتوانیم توزیع پیکسلی را برای هر یک کانال رنگی (R, G, B) در حالت train، val و test مشاهده کنیم.

سپس همانطور که مشاهده می شود دیتا لودر ها از متاد گت ایتم برای دریافت سمپل ها استفاده میکنند که ما در متاد گت آیتم هر بار ترسفورمیشن ها را انجام میدهیم که این موضوع منجر به آن میشود که داده ها در هر ایپاک دوباره ترسفورم جدید بشوند و مدل به صورت جنرال تری ترین شود.

همچنین بخش هایی از فرایند EDA شامل نشان دادن توزیع داده ها در کلاس های ان در کلاس دیتاست، نشان دادن چند تصویر به صورت رندهم، نشان دادن چند تصویر پس از ترسفورمیشن ها و پلات کردن دیستربیوشن مقادیر پیکسل های کانال های مختلف عکس ها در این کلاس قرار داده شده است.

و با استفاده از آنها از Data Loader به صورت زیر استفاده شده است:

```
dataset1 = Data(device=device, img_size_w=28, img_size_h=28)
dataset1.set_transformers(train_transform, val_transform)
dataset1.normalize()

dataset1.set_mode('train')
train_loader1 = DataLoader(dataset1, batch_size=64, shuffle=True)
train_size1 = count_batches(train_loader1)

dataset1.set_mode('val')
val_loader1 = DataLoader(dataset1, batch_size=64, shuffle=False)
val_size1 = count_batches(val_loader1)

dataset1.set_mode('test')
test_loader1 = DataLoader(dataset1, batch_size=64, shuffle=False)
test_size1 = count_batches(test_loader1)

print(f"Train Loader Size: {train_size1} batches")
print(f"Val Loader Size: {val_size1} batches")
print(f"Test Loader Size: {test_size1} batches")
```

شكل 1-1. نحوه استفاده از دیتا لودر در مجموعه های داده

همانطور که در تصویر مشخص است، از مقدار 64 استفاده شده است. به این طریق، در هر ایپاک، مجموعه داده های ورودی در batch هایی به اندازه 64 وارد مدل می شود و یکجا تمامی داده ها را که مقدار خیل بزرگی دارند را به مدل وارد نمی کند. این امر به علت ram محدود سخت افزار می باشد. همینطور با این مکانیزم از ویژگی shuffle در مجموعه train استفاده می کنیم تا ترتیب داده ها روی خروجی تاثیر نگذارند و مدل روی تصاویر ورودی overfit نکند و دقت مدل بالاتر برود.

داده های ورودی را به سه دسته validation، train، test تقسیم کرده ایم. این تقسیم سازی باعث می شود در انتهای هر ایپاک train روی مدل، امکان validation روی بخشی از مجموعه داده وجود داشته باشد تا در ایپاک بعدی، مدل یادگیری بهتری داشته

باشد. در واقع به مدل این امکان را داده‌ایم که قبلاً از validation نهایی تست، هایی روی مجموعه داده کوچک‌تر داشته باشد و به دقت بهتری برسد. مجموعه داده را به صورت 20 درصد داده‌ی تست، 16 درصد داده train، و 64 درصد داده‌ی validation اختصاص داده‌ایم.

سایز خروجی هر لایه کانولوشن در مدل‌های CNN، با فرمول زیر محاسبه می‌شود:

$$\text{Output size} = \left\lfloor \frac{\text{Input size} - \text{Filter size} + 2 \times \text{Padding}}{\text{Stride}} \right\rfloor + 1$$

Input size: the size (width or height) of the input

Filter size: the size (width or height) of the filter or kernel

Padding: the number of pixels added to the input along the width and height

Stride: the number of pixels by which the filter is moved across the input

محاسبات فوق منجر به پیاده سازی زیر در پای تورج گشت :

```
class SkinCancerCNN(nn.Module):
    Tabnine | Edit | Test | Explain | Document | Ask
    def __init__(self):
        super(SkinCancerCNN, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, padding=1, stride=1)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, padding=1, stride=1)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv3 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1, stride=1)
        self.pool3 = nn.MaxPool2d(kernel_size=1, stride=2)

        self.conv4 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1, stride=1)
        self.pool4 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.flatten = nn.Flatten()

        self.fc1 = nn.Linear(2 * 2 * 128, 64)
        self.fc2 = nn.Linear(64, 32)
        self.fc3 = nn.Linear(32, 2)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool1(x)

        x = F.relu(self.conv2(x))
        x = self.pool2(x)

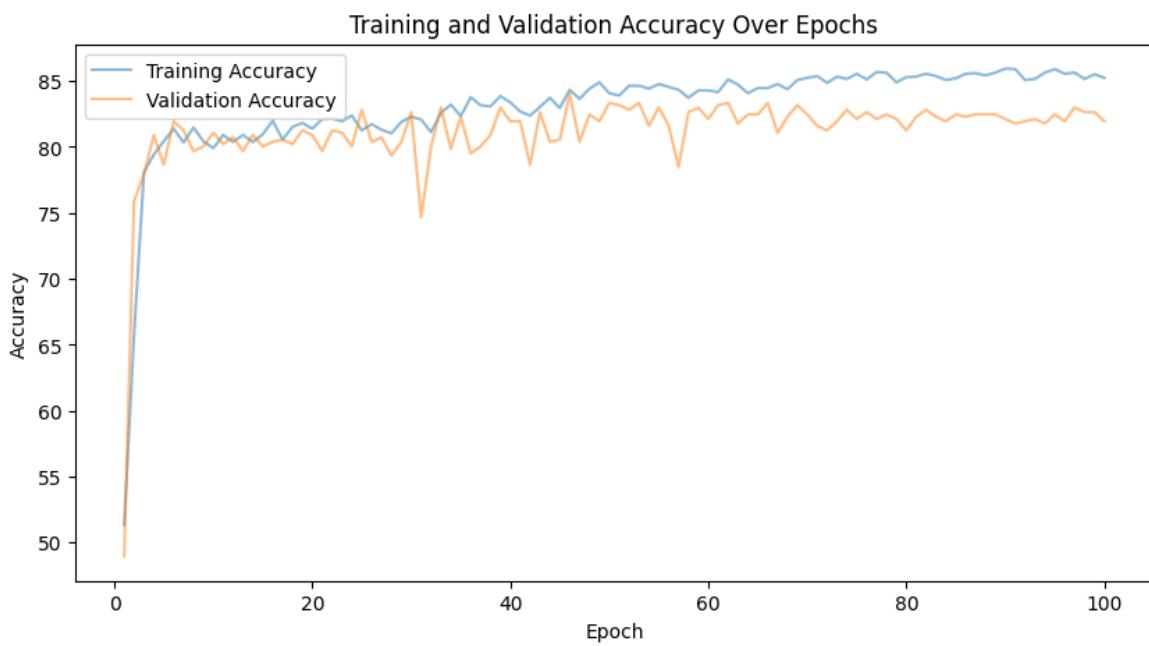
        x = F.relu(self.conv3(x))
        x = self.pool3(x)

        x = F.relu(self.conv4(x))
        x = self.pool4(x)

        x = self.flatten(x)

        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

در تصویر زیر نتایج دقت و خطای مدل اول نشان داده شده است:



شکل ۱-۱. نمودار دقت مدل در طول ایپاک‌ها

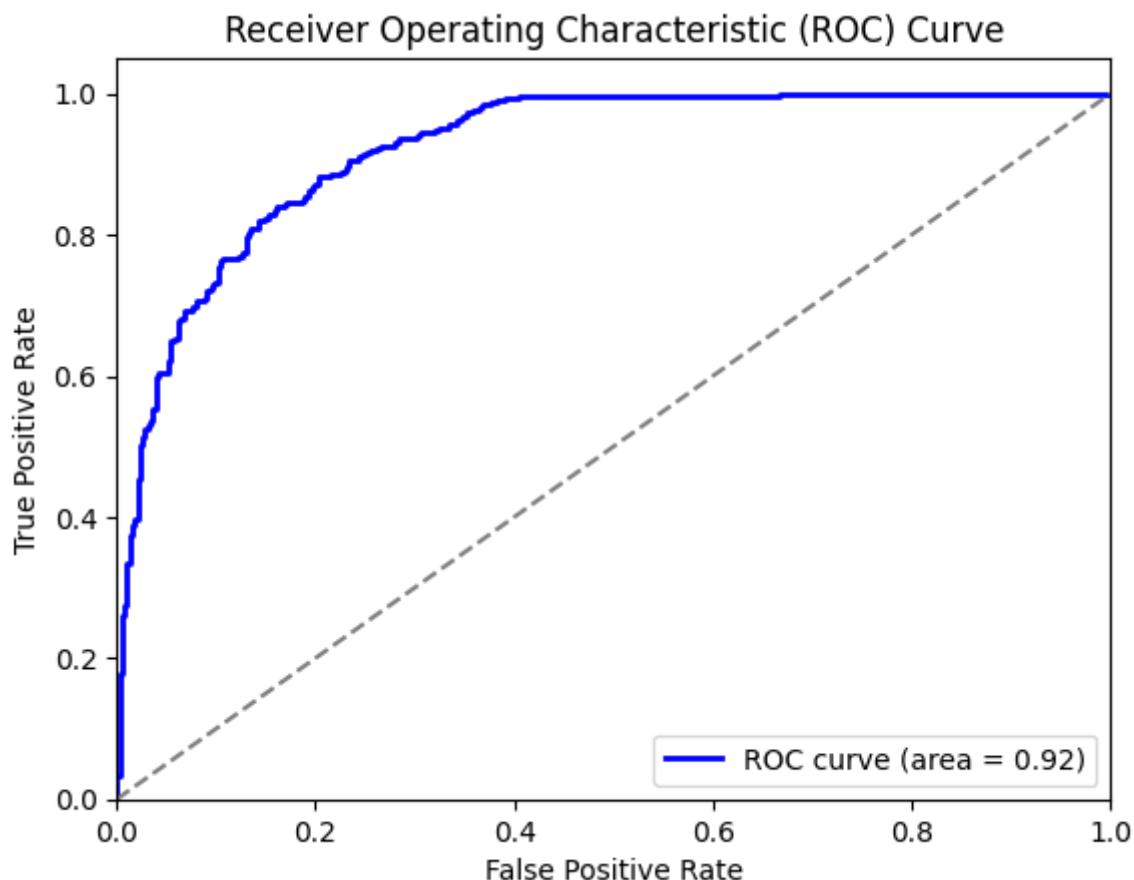


شکل ۱-۱. نمودار خطای مدل در طول ایپاک‌ها

منحنی مشخصه عملکرد گیرنده، یا همان منحنی ROC، یک نمودار گرافیکی است که عملکرد یک مدل طبقه‌بندی‌کننده باینری (می‌تواند برای طبقه‌بندی چند طبقه نیز استفاده شود) را در مقادیر آستانه متفاوت نشان می‌دهد. همینطور در این نمودار متغیری به نام Area Under Curve مقداردهی می‌شود که همان Area Under Curve است. معیاری

عددی که عملکرد کلی مدل را نشان می‌دهد. این مقدار بین ۰ تا ۱ است. هرچه این معیار بیشتر باشد، مدل ما بهتر عمل کرده است. اگر مدل ما پایین‌تر از خط چین این نمودار عمل کند، یعنی از حالت پیش‌بینی تصادفی بدتر و ضعیف تر است. این نمودار برای داده‌های نامتوازن کاربرد خیلی خوبی دارد چرا که فارغ از تعداد داده‌های هر کلاس، رابطه بین True Pos Rate و Recall را مورد بررسی قرار میدهد. رابطه بین False Pos Rate و False Neg Rate نشان داده می‌شود.

این نمودار برای مدل اول به صورت زیر رسم شده است:



شکل ۱-۱. نمودار ROC مدل

در این مدل، از جریاناتیک (Batch Normalization) و دropout استفاده شده است:

DropOut : یک تکنیک منظم سازی است که برای بهبود تعمیم و جلوگیری از برازش بیش از حد مدل های بزرگ و پیچیده طراحی شده است. در واقع مدل به صورت تصادفی، تعدادی نورون را در لایه نهان و تنها در مرحله Train غیر فعال می کند. برای مثال، اگر نرخ ورودی متد 0.5 تنظیم شده باشد، در طول هر تکرار train، نیمی از نورون های لایه مربوطه را به طور تصادفی غیرفعال می کند. درواقع مدل الگوهای گستردہ و کلی را جستجو می کند که الگوهای وزنی آنها قوی تر هستند.

Batch Normalization : نرمال سازی دسته ای روشی است که برای آموزش شبکه های عصبی سریعتر و پایدارتر استفاده می شود. این روش برای تغییرات توزیع داده ای انتقال یافته بین لایه های شبکه استفاده می شود. با نرمال سازی داده ها به مقیاس مشخص، فرآیند یادگیری مدل سریع تر شده و دقت مدل بهتر می شود. این نرمالیزیشن شامل اعمال ضرب و جمع می باشد.

در نهایت مدل حاصل به شکل زیر است :

```
class conv_block(nn.Module):
    Tabnine | Edit | Test | Explain | Document | Ask
    def __init__(self, in_channels, out_channels, **kwargs):
        super(conv_block, self).__init__()
        self.relu = nn.ReLU()
        self.conv = nn.Conv2d(in_channels, out_channels, **kwargs)
        self.batchnorm = nn.BatchNorm2d(out_channels)

Tabnine | Edit | Test | Explain | Document | Ask
def forward(self, x):
    return self.relu(self.batchnorm(self.conv(x)))
```

این کلاس پایه یه لایه های کانولوشنی ما است که در هر لایه بج نرم داریم.

```
class ImprovedSkinCancerCNN(nn.Module):
    Tabnine | Edit | Test | Explain | Document | Ask
    def __init__(self):
        super(ImprovedSkinCancerCNN, self).__init__()

        self.conv1 = conv_block(in_channels=3, out_channels=16, kernel_size=3, padding=1, stride=1)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.dropout1 = nn.Dropout(p=0.15)

        self.conv2 = conv_block(in_channels=16, out_channels=32, kernel_size=3, padding=1, stride=1)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.dropout2 = nn.Dropout(p=0.15)

        self.conv3 = conv_block(in_channels=32, out_channels=64, kernel_size=3, padding=1, stride=1)
        self.pool3 = nn.MaxPool2d(kernel_size=1, stride=2)
        self.dropout3 = nn.Dropout(p=0.15)

        self.conv4 = conv_block(in_channels=64, out_channels=128, kernel_size=3, padding=1, stride=1)
        self.pool4 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.dropout4 = nn.Dropout(p=0.15)

        self.flatten = nn.Flatten()

        self.fc1 = nn.Linear(2 * 2 * 128, 64)
        self.bnrm1 = nn.BatchNorm1d(64)
        self.fc2 = nn.Linear(64, 32)
        self.bnrm2 = nn.BatchNorm1d(32)
        self.fc3 = nn.Linear(32, 2)
```

```

def forward(self, x):
    x = self.conv1(x)
    x = self.pool1(x)
    x = self.dropout1(x)

    x = self.conv2(x)
    x = self.pool2(x)
    x = self.dropout2(x)

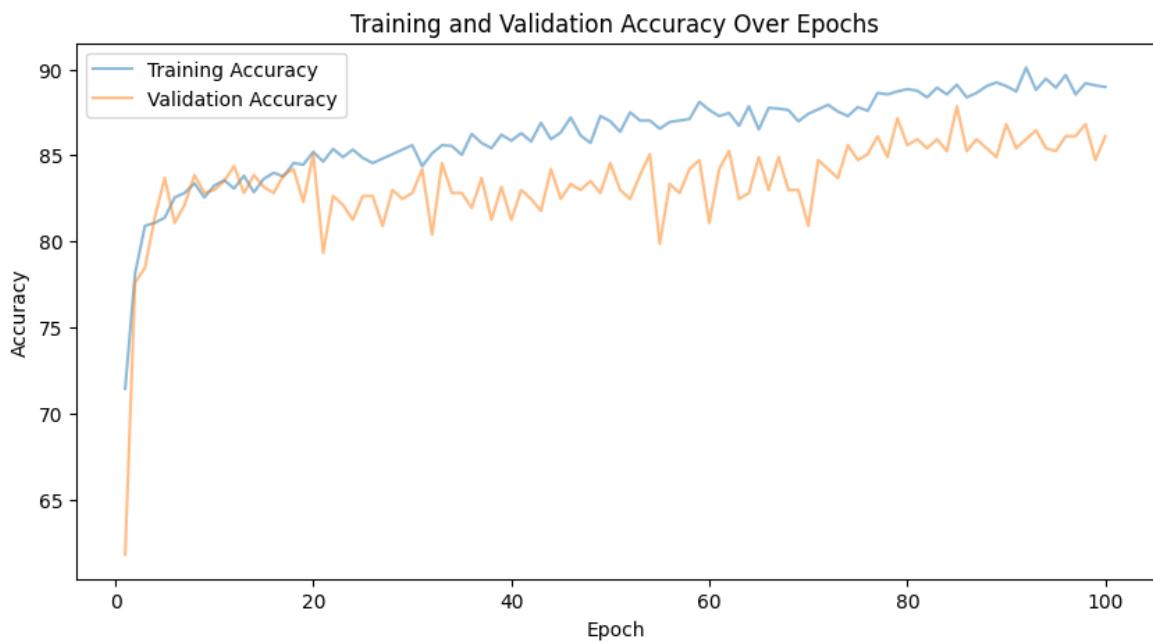
    x = self.conv3(x)
    x = self.pool3(x)
    x = self.dropout3(x)

    x = self.conv4(x)
    x = self.pool4(x)
    x = self.dropout4(x)

    x = self.flatten(x)

    x = F.relu(self.bnorm1(self.fc1(x)))
    x = F.relu(self.bnorm2(self.fc2(x)))
    x = self.fc3(x)
    return x

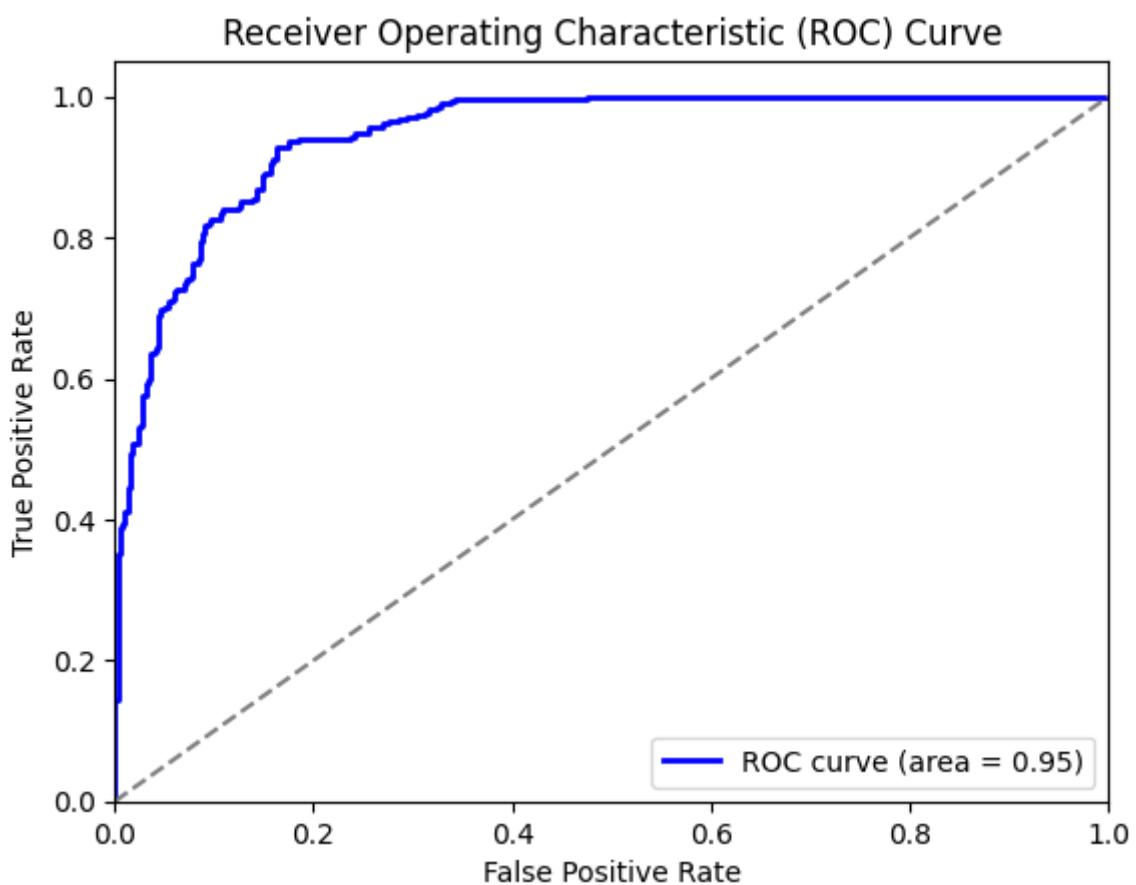
```



شکل 1-1. نمودار دقت مدل در طول ایپاکها



شکل ۱-۱. نمودار خطای مدل در طول ایپاکها



شکل ۱-۱. نمودار ROC مدل

1. تحلیل نتایج

CNN Model	Train-Acc	Test-Acc	Train-Loss	Test-Loss	AUC
Skin Cancer CNN	85.24	81.81	0.3193	0.3477	0.92
Improved Skin Cancer CNN	88.98	85.70	0.2420	0.2826	0.95
GoogLeNet	96.40	87.50	0.2658	0.3857	0.96

1-6. مقایسه نتایج

همانطور که انتظار می رفت، مدل سوم، دقت نهایی بالاتری نسبت به دو مدل قبلی دارد. در مدل دوم هم به علت اضافه کردن ویژگی دراپ اوت و نرمالیزیشن روی بچ ها، نتیجه نهایی بهتری نسبت به مدل اول داشته است.

تفاوت مدل اول و دوم روی بیشپردازی می باشد. در مدل دوم، بیشپردازش بیشتری نسبت به مدل اول دارد.

هر سه مدل، مقدار AUC بالای 90 درصد دارند و نسبت به مدل تصادفی بهتر عمل می کنند. مساحت زیر منحنی ROC در مدل دوم، 0.03 بیشتر از مدل اول است و نشان از عملکرد بهتر آن مدل روی مجموعه داده تست می باشد.

همینطور AUC در مدل گوگل نت از هر دو مدل قبلی بالاتر است و نشاندهنده عملکرد بهتر نسبت به دو مدل قبلی در پیشビینی برچسب مجموعه داده تست میباشد.

1-7. مدل عمیقتر

در این بخش، مدل GoogLeNet یا InceptionNet را پیاده کرده ایم.¹ به طور کلی معماری های مختلفی مانند LeNet-5 - AlexNet - VGGNet در اوایل دهه 2010-2020 معرفی شدند که عملکرد قابل قبول را در مسائل طبقه بندی عکس ارائه کردند. در سال 2014 مدل

¹ <https://arxiv.org/abs/1409.4842>

ارائه شد که تغییر قابل توجهی در ساختار شبکه convolutional ایجاد کرد. در این مدل از تابع فعال سازی RELU استفاده شده است. یکی از مهم ترین روش ها در این مدل که باعث عملکرد بهتر مدل می شود، امکان استفاده از کرنل با اندازه های مختلف است.

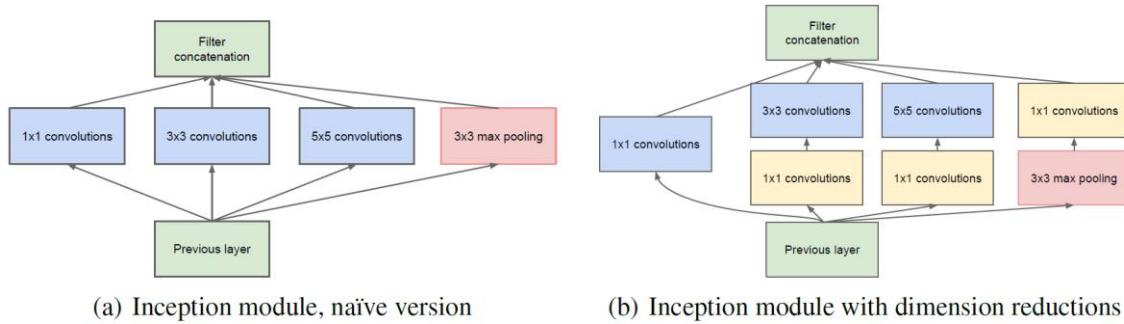
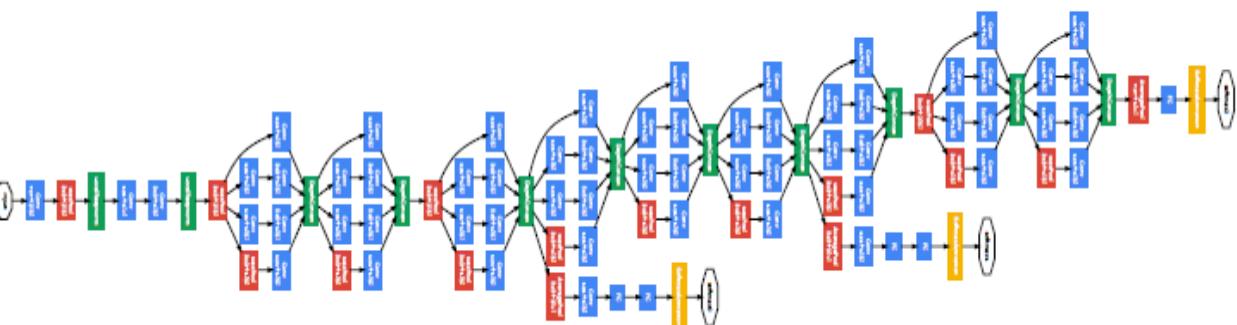


Figure 2: Inception module

شکل 1-1. مدل Inception

همانطور که در شکل بالا مشاهده می شود برای کاهش تعداد کرنل ها از کانولوشن هایی با اندازه هسته 1×1 استفاده می شود. به عنوان مثال 100 هسته وارد شده و 50 هسته از آنجایی که محاسبه کرنل های 5×5 و حتی کرنل های 3×3 تولید می شود. در مقیاس CNN ها می تواند پرهزینه باشد، بنابراین در هر لایه با توجه به تصویر، تعداد کمی از فیلترهای 5×5 ، تعداد بیشتری فیلتر 3×3 و تعدادی فیلتر ادغام کننده که به ما امکان می دهد وزن مشخص شده آنها را یاد بگیریم. این اجزه می دهد تا از هسته هایی با اندازه های مختلف استفاده کنیم.



شکل 1-1. گراف لایه های مدل

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	$7 \times 7 / 2$	$112 \times 112 \times 64$	1							2.7K	34M
max pool	$3 \times 3 / 2$	$56 \times 56 \times 64$	0								
convolution	$3 \times 3 / 1$	$56 \times 56 \times 192$	2		64	192				112K	360M
max pool	$3 \times 3 / 2$	$28 \times 28 \times 192$	0								
Inception (3a)		$28 \times 28 \times 256$	2	64	96	128	16	32	32	159K	128M
Inception (3b)		$28 \times 28 \times 480$	2	128	128	192	32	96	64	380K	304M
max pool	$3 \times 3 / 2$	$14 \times 14 \times 480$	0								
Inception (4a)		$14 \times 14 \times 512$	2	192	96	208	16	48	64	364K	73M
Inception (4b)		$14 \times 14 \times 512$	2	160	112	224	24	64	64	437K	88M
Inception (4c)		$14 \times 14 \times 512$	2	128	128	256	24	64	64	463K	100M
Inception (4d)		$14 \times 14 \times 528$	2	112	144	288	32	64	64	580K	119M
Inception (4e)		$14 \times 14 \times 832$	2	256	160	320	32	128	128	840K	170M
max pool	$3 \times 3 / 2$	$7 \times 7 \times 832$	0								
Inception (5a)		$7 \times 7 \times 832$	2	256	160	320	32	128	128	1072K	54M
Inception (5b)		$7 \times 7 \times 1024$	2	384	192	384	48	128	128	1388K	71M
avg pool	$7 \times 7 / 1$	$1 \times 1 \times 1024$	0								
dropout (40%)		$1 \times 1 \times 1024$	0								
linear		$1 \times 1 \times 1000$	1							1000K	1M
softmax		$1 \times 1 \times 1000$	0								

Table 1: GoogLeNet incarnation of the Inception architecture

شکل 1-1. جدولی از

در تصویر فوق، جدولی که در مقاله اندازه ها و ساختار این CNN آورده ایم. نکته مهم دیگری که از دو تصویر فوق مشاهده می شود، وجود دو زیرشاخه It است که به softmax یا Inception Aux منتهی می شود.

مشکل Vanishing Gradient یکی از مشکلات اصلی در فرآیند یادگیری یک CNN است، زیرا در شبکه های عمیق، در فرآیند عقبگرد یادگیری یا همان Backward Process، خطای محاسبه شده پس از عبور از چندین لایه بسیار کم می شود و زمان همگرایی الگوریتم را به بهینه می رساند. اما اضافه شدن شاخه ای اشاره شده که فقط در فرآیند آموزش مدل فعال هستند، به ما کمک می کند تا با کمک این خروجی ها، 2 مورد از خطای اضافه تر را محاسبه کنیم و در نتیجه مجموع این 3 مقادیر خطای مدل ، می توانیم تاثیر مثبت تری در نتایج مدل مشاهده کنیم.

پیاده سازی انجام شده برای این معماری در ادامه آمده است:

```

def __init__(self, aux_logits=True, num_classes=4):
    super(GoogleNet, self).__init__()
    assert aux_logits == True or aux_logits == False
    self.aux_logits = aux_logits
    self.conv1 = conv_block(
        in_channels=3, out_channels=64, kernel_size=7, stride=2, padding=3)
    self.maxpool1 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
    self.conv2 = conv_block(64, 192, kernel_size=3, stride=1, padding=1)
    self.maxpool2 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
    # In this order: in_channels, out_1x1, red_3x3, out_3x3, red_5x5, out_5x5, out_1x1pool
    self.inception3a = Inception_block(192, 64, 96, 128, 16, 32, 32)
    self.inception3b = Inception_block(256, 128, 128, 192, 32, 96, 64)
    self.maxpool3 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
    self.inception4a = Inception_block(480, 192, 96, 208, 16, 48, 64)
    self.inception4b = Inception_block(512, 160, 112, 224, 24, 64, 64)
    self.inception4c = Inception_block(512, 128, 128, 256, 24, 64, 64)
    self.inception4d = Inception_block(512, 112, 144, 288, 32, 64, 64)
    self.inception4e = Inception_block(528, 256, 160, 320, 32, 128, 128)
    self.maxpool4 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
    self.inception5a = Inception_block(832, 256, 160, 320, 32, 128, 128)
    self.inception5b = Inception_block(832, 384, 192, 384, 48, 128, 128)
    self.avgpool = nn.AvgPool2d(kernel_size=7, stride=1)
    self.dropout = nn.Dropout(p=0.4)
    self.fc1 = nn.Linear(1024, num_classes)
    if self.aux_logits:
        self.aux1 = InceptionAux(512, num_classes)
        self.aux2 = InceptionAux(528, num_classes)
    else:
        self.aux1 = self.aux2 = None

```

```

def forward(self, x):
    x = self.conv1(x) #
    x = self.maxpool1(x) #
    x = self.conv2(x) #
    x = self.maxpool2(x) #
    x = self.inception3a(x) # 192, 64
    x = self.inception3b(x) # 256, 128
    x = self.maxpool3(x) #
    x = self.inception4a(x) #
    # Auxiliary Softmax classifier 1
    if self.aux_logits and self.training:
        aux1 = self.aux1(x)
        x = self.inception4b(x) # 512, 160
        x = self.inception4c(x) # 512, 128
        x = self.inception4d(x) # 512, 112
    # Auxiliary Softmax classifier 2
    if self.aux_logits and self.training:
        aux2 = self.aux2(x)
        x = self.inception4e(x)
        x = self.maxpool4(x)
        x = self.inception5a(x)
        x = self.inception5b(x)
        x = self.avgpool(x)
        x = x.reshape(x.shape[0], -1)
        x = self.dropout(x)
        x = self.fc1(x)
    if self.aux_logits and self.training:
        return aux1, aux2, x
    else:
        return x

```

```

class Inception_block(nn.Module):
    Tabnine | Edit | Test | Explain | Document | Ask
    def __init__(self, in_channels, out_1x1, red_3x3, out_3x3, red_5x5, out_5x5, out_1x1pool):
        super(Inception_block, self).__init__()
        self.branch1 = conv_block(in_channels, out_1x1, kernel_size=1)

        self.branch2 = nn.Sequential(
            conv_block(in_channels, red_3x3, kernel_size=1),
            conv_block(red_3x3, out_3x3, kernel_size=(3, 3), padding=1),
        )

        self.branch3 = nn.Sequential(
            conv_block(in_channels, red_5x5, kernel_size=1),
            conv_block(red_5x5, out_5x5, kernel_size=5, padding=2),
        )

        self.branch4 = nn.Sequential(
            nn.MaxPool2d(kernel_size=3, stride=1, padding=1),
            conv_block(in_channels, out_1x1pool, kernel_size=1),
        )
    Tabnine | Edit | Test | Explain | Document | Ask
    def forward(self, x):
        return torch.cat([self.branch1(x), self.branch2(x), self.branch3(x), self.branch4(x)], 1)

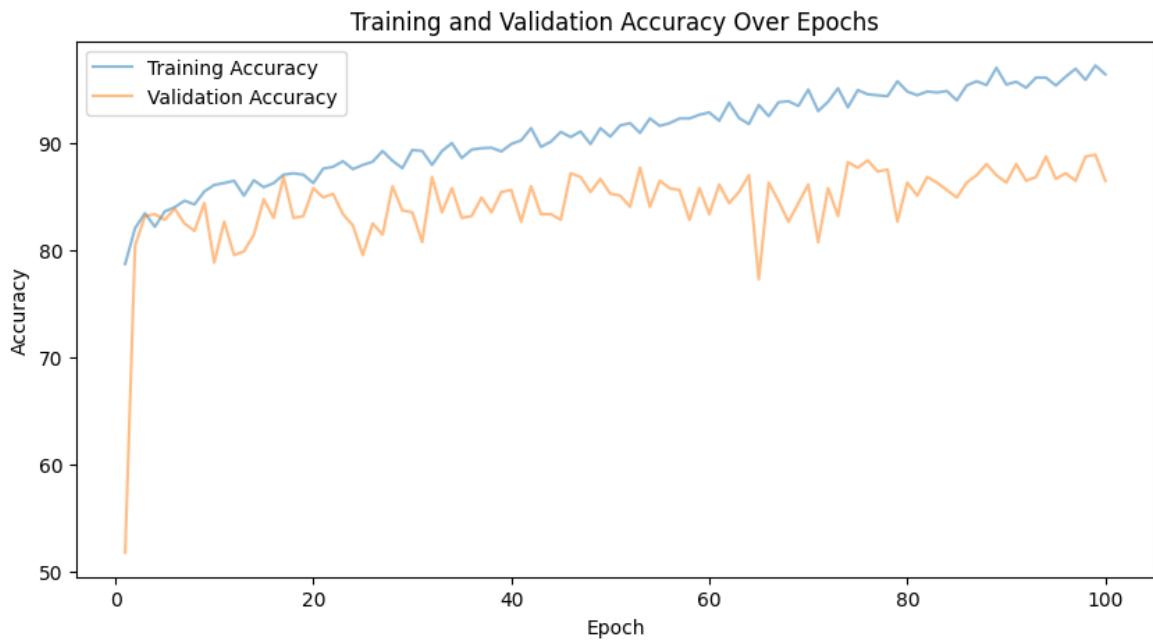
```

```

class InceptionAux(nn.Module):
    Tabnine | Edit | Test | Explain | Document | Ask
    def __init__(self, in_channels, num_classes):
        super(InceptionAux, self).__init__()
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(p=0.7)
        self.pool = nn.AvgPool2d(kernel_size=5, stride=3)
        self.conv = conv_block(in_channels, 128, kernel_size=1)
        self.fc1 = nn.Linear(2048, 1024)
        self.fc2 = nn.Linear(1024, num_classes)

    Tabnine | Edit | Test | Explain | Document | Ask
    def forward(self, x):
        x = self.pool(x)
        x = self.conv(x)
        x = x.reshape(x.shape[0], -1)
        x = self.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x

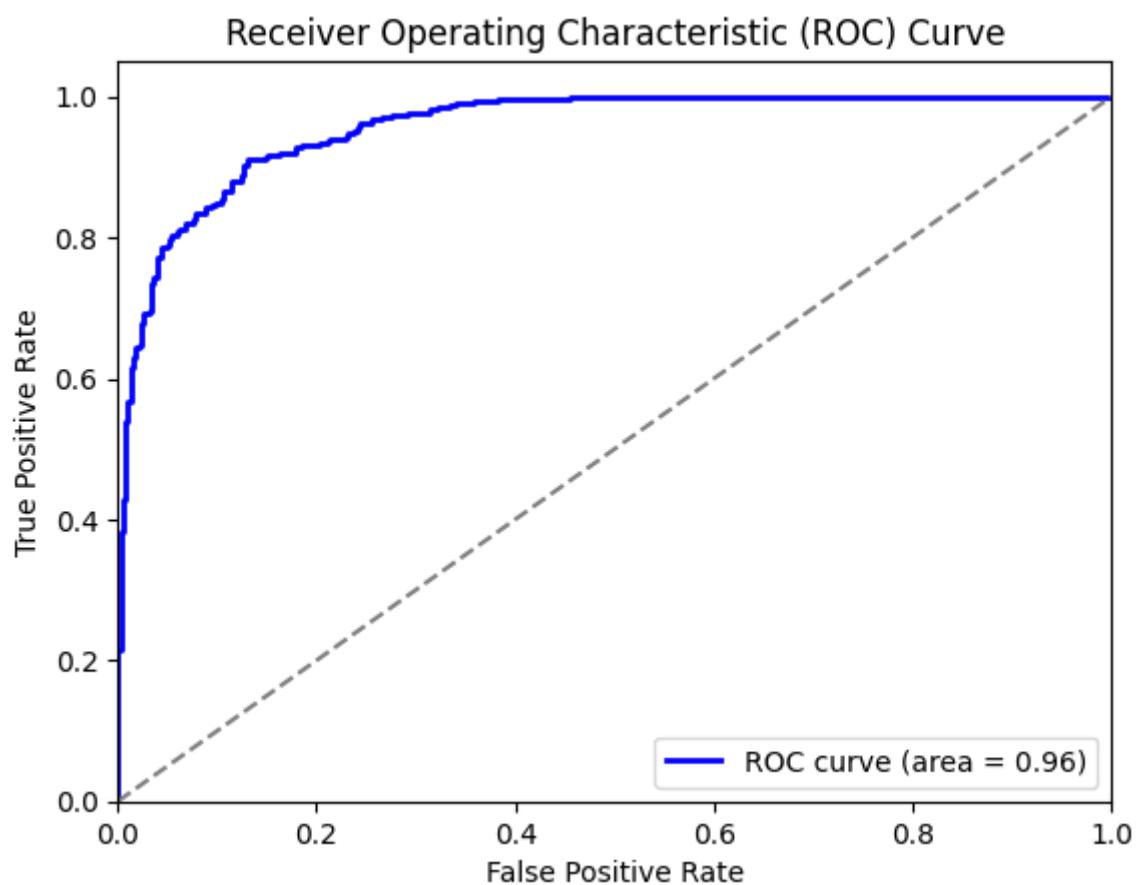
```



شکل ۱-۱. نمودار دقت مدل در طول ایپاکها



شکل ۱-۱. نمودار خطای مدل در طول ایپاکها



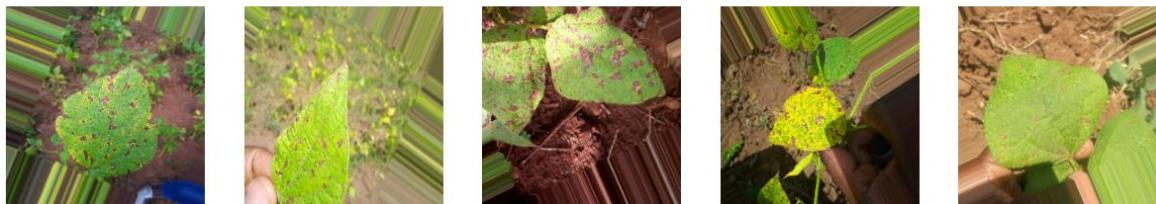
شكل 1-1. نمودار ROC مدل

پرسش 2 - تشخیص بیماری‌های برگ لوبیا با شبکه‌های عصبی

2-1. پیش‌پردازش تصاویر

شکل زیر، چند نمونه از تصویر هر سه نوع برگ سالم، بیماری زنگ لوبیا، و لکه‌برگی زاویه‌دار را نشان می‌دهد.

Class: angular_leaf_spot



شکل 1-1. پنج تصویر از مجموعه برگ‌های دارای لکه برگی زاویه‌دار

Class: bean_rust



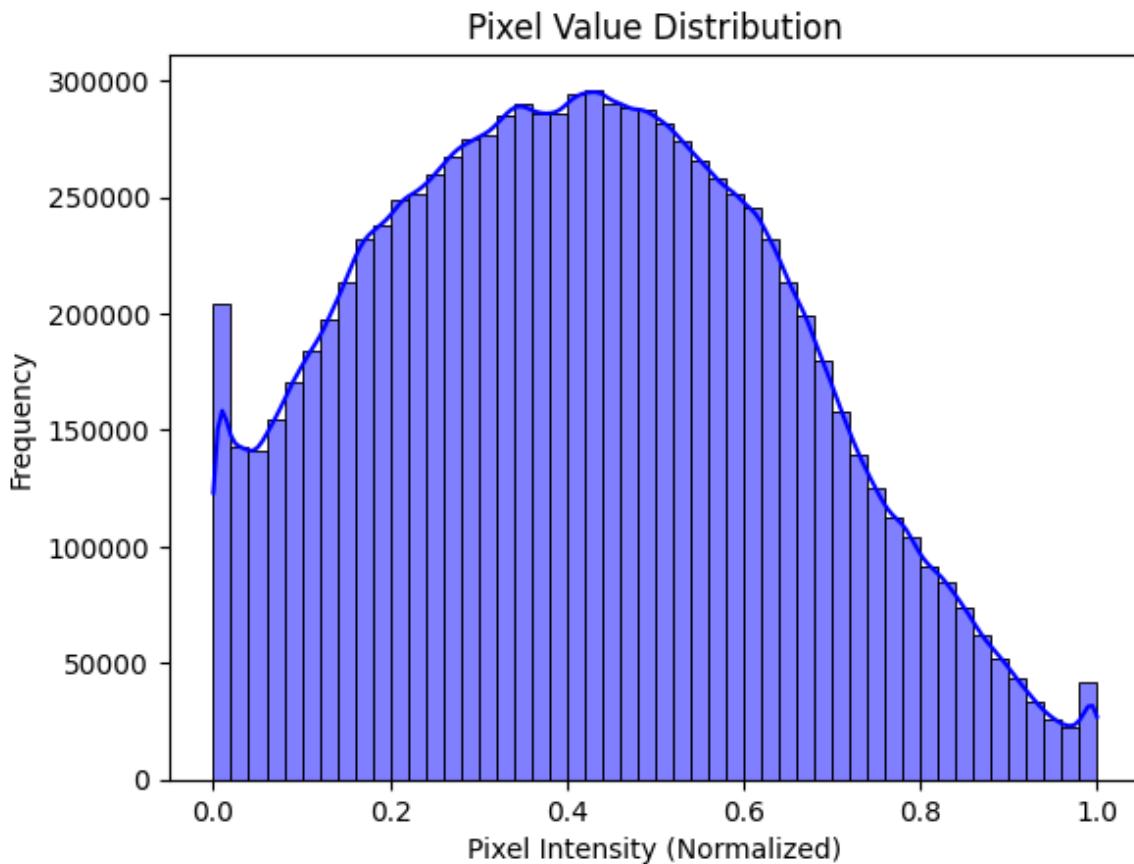
شکل 1-1. پنج تصویر از مجموعه برگ‌های دارای بیماری زنگ لوبیا

Class: healthy



شکل 1-1. پنج تصویر از مجموعه برگ‌های سالم

در شکل زیر به بررسی توزیع پیکسل‌های موجود در داده ترین می‌پردازیم.



شکل ۱-۱. توزیع پیکسل‌های داده‌ی ترین

همانطور که مشاهده می‌شود، توزیع خیلی شبیه به توزیع نرمال دارند.

در مقاله مربوطه برای یکنواختی داده‌های ورودی مدل و استخراج بهتر ویژگی از آنها، از پیش پردازش تغییر اندازه تصویر های مجموعه داده به سایز مطلوب $224 \times 224 \times 3$ می‌باشد. این اعداد با مدل‌های پیچیده بررسی شده در این سوال سازگاری قابل قبولی دارند.

همینطور روی داده‌های ورودی مدل اعمال زیر پیاده می‌شود. ترانسفورمیشن های انجام شده بر روی مجموعه داده جهت ترین کردن مدل در بخش بعدی تفسیر شده‌اند.

تغییرات اعمال شده روی داده‌های تست و ولیدیشن تنها شامل rescale شدن می‌باشد.

توجه شود که rescale برای همگی مدل‌ها به جز مدل EfficientNetB6 انجام شده است. دلیل این مورد هم صورت گرفتن اسکیل در این مدل می‌باشد که انجام مجدد آن توصیه نمی‌شود.

مجموعه‌های Data loader را با استفاده از train، validation و test می‌خوانیم. مزایای دیتا لودر در سوال اول ذکر شده است. توجه شود ویژگی شافل را تنها برای داده train فعال کرده‌ایم.

2-2. پیاده‌سازی

2-2-1. انتخاب مدل‌ها

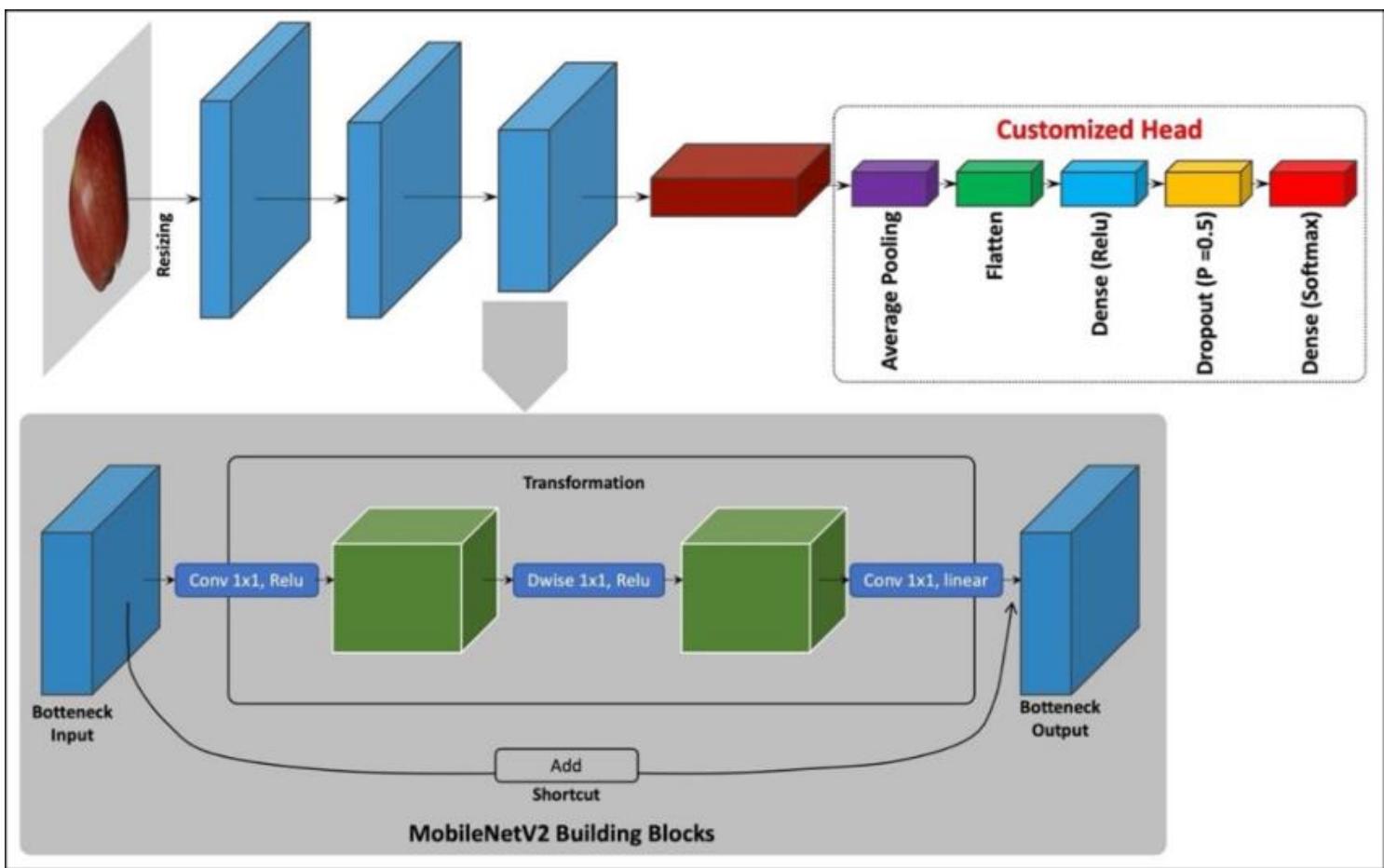
در ابتدا به توضیح ساختار مدل‌های ارائه شده می‌پردازیم:

MobileNetV2 •
یکی از عوامل افزایش بسیار پارامترهای مدل‌های دیپ لایه‌های کانولوشن کلاسیک است که به صورت کلی هزینه‌ای برابر با $H \times W \times K^2$ دارد که در آن H ارتفاع ورودی، W عرض ورودی، K اندازه کرنل، $C_{in} \times C_{out}$ نماینده تعداد فیلترهای ورودی و C_{out} نماد تعداد فیلترهای خروجی است.

در ورژن اولیه این مدل تمرکز بر روی کاهش این هزینه قرار داشت که به کانولوشنی به جای انجام یک کانولوشن کلاسیک به معنای ضرب یک کرنل $K \times K \times C_{in}$ یک کرنل $K \times K^2 \times C_{in}$ می‌گویند و هزینه‌ای برابر $H \times W \times K^2 \times C_{in}$ خواهد داشت و پس از آن بر روی خروجی یک کانولوشن 1×1 انجام می‌شود تا داده‌ها را به تعداد کanal‌های خروجی برساند و در این فرایند داده‌های کanal‌ها را به صورت مناسب تلفیق کند. و به آن Pointwise Convolution می‌گویند که هزینه‌ای برابر $H \times W \times C_{in} \times C_{out}$ خواهد داشت. که تلفیق این دو منجر به ذخیره

توان محاسباتی با اندازه $\frac{K^2 \times C_{in} \times C_{out}}{K^2 \times C_{in} + C_{in} \times C_{out}}$ می‌شود.

حال که با این مفاهیم آشنا شدیم به سراغ MobileNetV2 می‌رویم که مطابق شکل زیر است:

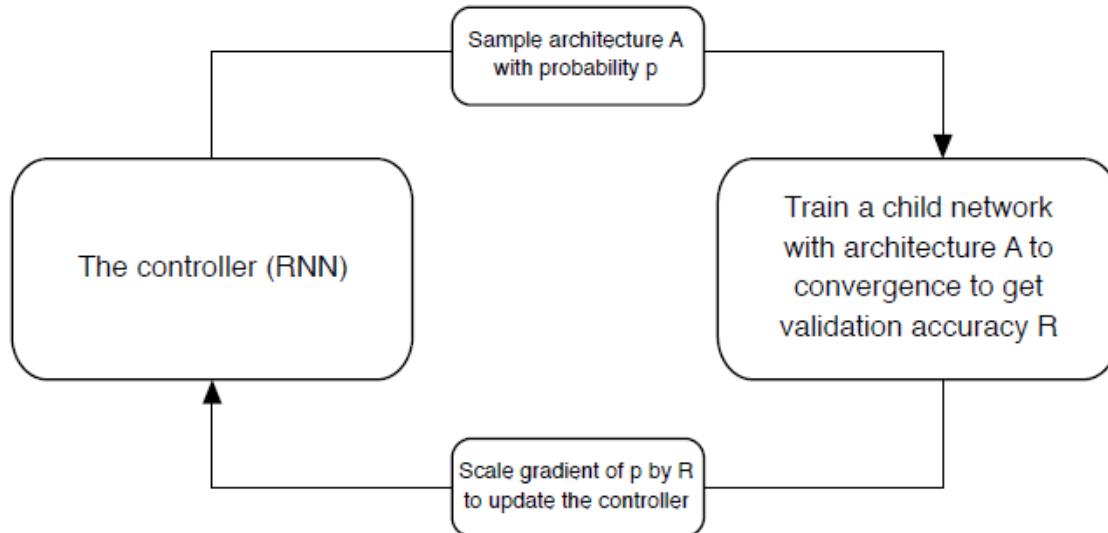


شکل 2-1. ساختار لایه‌های مدل 2MobileNetV2

همانطور که در شکل بالا مشاهده می‌کنید ابتدا یک Pointwise Convolution داریم و پس از آن یک Depthwise Convolution و سپس یک Pointwise Convolution اول Pointwise Convolution انجام می‌شود که در این کانال‌ها Depthwise Convolution ویژگی‌های جدید را با این کرنل‌ها افزایش داده و با Pointwise Convolution دوم ابعاد را به ابعاد ورودی بر می‌گرداند. و پس از آن اسکیپ کانکشنی در حالتی که در عکس ارائه شد مابین ورودی و خروجی برقرار می‌شود.

- این مدل در حقیقت معماری ساخته شده توسط ام ال است و در کتگوری nasNet
- اتو ام ال قرار دارد. این مدل از دو نوع بلاک reduction cell و normal cell تشکیل شده است. normal cell ها ابعاد را حفظ می‌کنند و reduction cell ها ابعاد را کاهش میدهند.

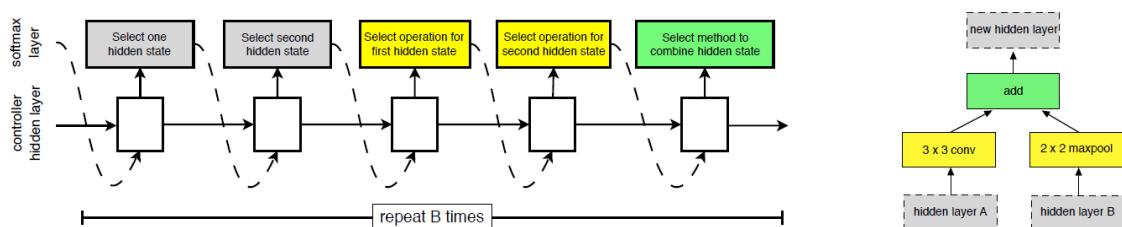
اما این مدل چگونه ترین میشود؟ ابتدا بر روی دیتاست کوچکی مانند CIFAR10 ترین شده و سپس با تغییراتی کوچک همان مدل عملکرد مناسبی بر روی نشان میدهد. فرایند ترین این مدل به شرح زیر است.



شکل 2-2. فرایند train مدل NasNet

همانطور که گفته شد این معماری از سل های نرمآل و ریداکشن تشکیل می شود که معماری داخل این سل ها را یک ار ان ان انجام میدهد و سپس بر اساس نتیجه حاصل این RNN بهینه سازی میشود تا سل های بهتری را انتخاب کند.

فرایند انتخاب این سل ها به این صورت است که در هر مرحله مجموعه ای از هیدن استیت ها که شامل نتایج لایه های قبل و ورودی است وجود دارد که 2 تا از آنها انتخاب شده و برای آنها اوپریشنی که در واقع کرنل کانولوشنی است انتخاب میشود و پس از آن برای نتایج این دو یک مکانیزم تجمعی انتخاب میشود.



شکل 2-3. مکانیزم ساختار لایه های NasNet

• EfficientNetB6: این مدل تمرکز خود را بیش از آنکه بر روی معماری خاصی بگذارد بر روی فرایندی گذاشته است که منجر به اسکیل شدن موثر مدل شود و و مدل نه تنها از جهت عمق یا عرض و یا رزولوشن ورودی بلکه از تمامی این جهات به نوعی اسکیل شود که بهترین نتیجه را بگیرند. به این منظور ضرایب $\alpha \square \beta \square \gamma$ تعریف میشود به گونه ای که $\alpha > 1 \square \beta > 1 \square \gamma > 1$ و همچنین $\alpha \cdot \beta^2 \approx 2$ باشد حال بر اساس این ضرایب خواهیم داشت :

$$d = \alpha^\phi \square w = \beta^\phi \square r = \gamma^\phi$$

که d بیانگر عمق شبکه w بیانگر عرض شبکه و r بیانگر رزولوشن ورودی است و فای ضریب اسکیل کردن مدل است و با بهینه سازی این پارامتر ها برای مدل میتوانند به مدلی برسند که به صورت افیشنتی اسکیل میشود .

2-2-2. تقویت داده

به صورت کلی تکنیک های تقویت داده با هدف ایجاد تغییراتی در داده ورودی به نوعی که شرایط مختلف و دامنه وسیع ورودی در شرایط کلی را بهتر مدل کند انجام میشود. مختلفی بر اساس ماهیت و شرایط دیتاست جهت تقویت داده قابل انجام است. در عکس های جنرال تر تغییر زوایا و زوم می تواند تاثیر مناسبی داشته باشد و در عکس های پزشکی عکس ها را کمی دفرمه میکنند میتوانند کمک کنند تا شرایط مختلف بیشتری در عکس ها به دست آید و مدل جنرال تر ترین شود.

ترنسفورمیشن های انجام شده بر روی مجموعه داده جهت ترین کردن مدل عبارت اند از: Normalizer: این ترنسفورمیشن با تقسیم تمامی مقادیر بر 255 نقش rescale دارد .

: در این ترنسفورمیشن محدوده ای تا 30 درجه جهت روتیت کردن عکس قرار میدهیم که موجب میشود تا عکس از زوایای مختلف در فرایند ترین دیده شود و مدل روی داده ی تست بتواند حساسیت کمتری نسبت به زوایا داشته باشد.

: این دو مورد با شیفت دادن عکس در جهت افقی و عمودی باعث میشود تا مدل قرار گیری سوژه در مکان های مختلفی از عکس را تجربه کند و حساسیت کمتری نسبت به موقعیت سوژه پیدا کند.

: این مورد نیز پرسپکتیو عکس را تغییر میدهد که حساسیت مدل و یادگیری آن بر اساس پرسپکتیو عکس شکل نگیرد.

0.2=zoom_range؛ این مورد نیز باعث کاهش حساسیت مدل به اندازه سوژه میشود که کمک به جنرالیزیشن بیشتر میشود.

3-2-2. بهینه‌سازها

ADAM میانگین متحرک یا همان First Moment Gradients و مربع Second Moment Gradients را در نظر می‌گیرد. نرخ یادگیری برای هر پارامتر به صورت تطبیقی بر اساس این موارد تنظیم می‌شود. این روش ترکیبی از مزایای Momentum برای سرعت‌بخشی به همگرایی و RMSprop برای نرخ یادگیری تطبیق را به مدل اضافه می‌کند. این اپتیمایزر به دلیل پایداری و سرعت همگرایی بالا عملکرد خوبی دارد. همینطور از آنجایی که نرخ یادگیری را به صورت خودکار تنظیم می‌کند، نیاز کمتری به تنظیم دستی هایپرامترها دارد.

RMSprop تقسیم نرخ یادگیری به ریشه میانگین مربعات gradients، نرخ یادگیری را برای پارامترهایی که تغییرات زیادی دارند، کاهش می‌دهد. این ویژگی باعث می‌شود که مدل به سرعت در مسیر بهینه حرکت کند و در نواحی پرتلاطم پایدارتر باشد. RMSprop برای مسائل با داده‌های غیر ایستاتیکارایی مناسب‌تری دارد و در مسائل پیچیده‌ای که نیاز به تنظیم دقیق نرخ یادگیری دارند، عملکرد خوبی نشان می‌دهد. اما در مسائلی که ممکن است نرخ یادگیری بیش از حد کاهش یابد و روی مدل تاثیر منفی بگذارد، همگرایی مدل را کنترل می‌کند.

Nadam نسخه بهینه شده Adam میباشد. این اپتیمایزر از تکنیک Nesterov استفاده می‌کند یعنی با اضافه کردن یک گام Lookahead به اپتیمایزر Adam، سرعت و دقت همگرایی را افزایش میدهد. علت سریعتر بودن Nadam این است که با ترکیب ویژگی‌های Adam و Nesterov، مسیر حرکت در فضای پارامترهای تعریف شده را سریعتر پیش‌بینی می‌کند و نوسانات موجود در نتیجه هم کاهش می‌یابد.

علت تفاوت عملکرد بهینه‌سازها در نحوه کانفیگ کردن نرخ یادگیری استفاده شده، نحوه تغییر وزن‌ها و هایپر پارامترها و می‌باشد. این الگوریتم‌ها هر کدام روش خودشان را پیاده می‌کنند و در نتیجه سرعت همگرایی در مدل‌ها با بهینه‌سازهای مختلف متفاوت می‌باشد. در این سوال ما عملکرد هرکدام را روی سه مدل بررسی می‌کنیم.

2-2-4. آموزش مدل

جهت انجام transfer learning و آموزش مدل‌های ذکر شده از تابع زیر استفاده کردیم :

```

def train_model(base_model, dense_layers, dense_layers_dropout, train_generator, validation_generator,
               optimizer, loss, metrics, freezed_epochs, fine_tune_epochs, learning_rate=3e-4):
    base_model.trainable = False

    x = base_model.output
    x = GlobalAveragePooling2D()(x)
    for i in range(len(dense_layers)) :
        x = Dense(dense_layers[i], activation='relu')(x)
        # x = BatchNormalization()(x)
        x = Dropout(dense_layers_dropout[i])(x)

    num_classes = train_generator.num_classes
    output = Dense(num_classes, activation='softmax')(x)

    model = Model(inputs=base_model.input, outputs=output)

    model.compile(
        optimizer=create_optimizer(optimizer, learning_rate),
        loss=loss,
        metrics=metrics
    )

    early_stopping_freeze = tf.keras.callbacks.EarlyStopping(
        monitor='val_loss',
        patience=5,
        restore_best_weights=True
    )

```

```

history = model.fit(
    train_generator,
    validation_data=validation_generator,
    epochs=freezed_epochs,
    callbacks=[early_stopping_freeze]
)

base_model.trainable = True

model.compile(
    optimizer=create_optimizer(optimizer, learning_rate),
    loss=loss,
    metrics=metrics
)

early_stopping_fine = tf.keras.callbacks.EarlyStopping(
    monitor='val_loss',
    patience=5,
    restore_best_weights=True
)

history_fine = model.fit(
    train_generator,
    validation_data=validation_generator,
    epochs=fine_tune_epochs,
    callbacks=[early_stopping_fine]
)

return model, history, history_fine

```

که اولین مدل قابل مشاهده `base_model` است. از آنجا که سه مدل متفاوت ذکر شده را ترین میکردیم مدل لود شده را از این طریق به تابع پاس میدهیم. سایر ورودی ها شامل ترین و ولیدیشن دیتا جنریتور ها است که داده را برای مدل امداده میکنند و ترانسفرمیشن های لازم را روی آن انجام میدهند. سایر فیلد های ورودی شامل `optimizer` که استرینگی است که نوع بهینه ساز هدف را مشخص میکند که به کمک تابع زیر اپتیمایزر ها را میسازیم .

```

def create_optimizer(optimizer, lr=3e-4):
    if optimizer == 'adam':
        return Adam(learning_rate=lr)
    elif optimizer == 'nadam':
        return Nadam(learning_rate=lr)
    elif optimizer == 'rmsprop':
        return RMSprop(learning_rate=lr)
    else:
        raise ValueError("Unsupported optimizer")

```

همچنین فیلد های لاس و متريک در مدل استفاده می شوند تا در حین ترین شدن از لاس معين شده استفاده شود و متريک های مشخص شده را در هر ايپاک محاسبه کند.

از آنجا که در حال یادگیری انتقالی هستیم پس قسمت فولی کانکتد مدل ها را لود نمی کنیم. در نتیجه سر کلسيفایر دلخواهی داریم که میتوان از هر تعداد لایه دلخواه با هر ضریبی از dense_layers_dropout و dense_layers اوت دلخواه تشکیل شوند که به کمک فیلد های CNN اکتفا کرده و بیس مدل این مقادیر را در ورودی دریافت کرده و در یک حلقه سر کلسيفایر مدل را می سازیم و سپس مدل را کامپایل میکنیم. در گام نخست ابتدا به وزن های پریترین شده ی CNN اکتفا کرده و بیس مدل را فریز میکنیم و تنها سر کلسيفایر را به تعداد ايپاک های مشخص شده ترین میکنیم و پس از آن fineTune لایه های بیس مدل را trainable میکنیم و به تعداد ايپاک مشخص شده کل مدل را میکنیم. در هر دو مرحله نیز از earlystop 5 استفاده کرده ایم که جلوی اور فیت را بگیریم و به بهترین مدل دست یابیم.

راه حل دیگری نیز برای مرحله fineTuning وجود دارد که در تابع پایین انجام شده است :

```

def train_model2(base_model, dense_layers, dense_layers_dropout, train_generator, validation_generator,
                 optimizer, loss, metrics, freezed_epochs, fine_tune_epochs, learning_rate=3e-4):
    base_model.trainable = False

    x = base_model.output
    x = GlobalAveragePooling2D()(x)
    for i in range(len(dense_layers)):
        x = Dense(dense_layers[i], activation='relu')(x)
        # x = BatchNormalization()(x)
        x = Dropout(dense_layers_dropout[i])(x)

    num_classes = train_generator.num_classes
    output = Dense(num_classes, activation='softmax')(x)

    model = Model(inputs=base_model.input, outputs=output)

    model.compile(
        optimizer=create_optimizer(optimizer, learning_rate),
        loss=loss,
        metrics=metrics
    )

    early_stopping_freeze = tf.keras.callbacks.EarlyStopping(
        monitor='val_loss',
        patience=5,
        restore_best_weights=True
    )

```

```

history = model.fit(
    train_generator,
    validation_data=validation_generator,
    epochs=freezed_epochs,
    callbacks=[early_stopping_freeze]
)

base_model.trainable = True
for layer in base_model.layers[:-len(base_model.layers)//10]:
    layer.trainable = False

model.compile(
    optimizer=create_optimizer(optimizer, learning_rate),
    loss=loss,
    metrics=metrics
)

early_stopping_fine = tf.keras.callbacks.EarlyStopping(
    monitor='val_loss',
    patience=5,
    restore_best_weights=True
)

history_fine = model.fit(
    train_generator,
    validation_data=validation_generator,
    epochs=fine_tune_epochs,
    callbacks=[early_stopping_fine]
)

return model, history, history_fine

```

همانطور که مشاهده می شود تفاوت تابع بالا با تابع قبل در مرحله fineTune کردن است که به جای trainable کردن تمام بیس مدل به trainable کردن تنها 10 درصد پایانی لایه های بیس مدل اکتفا میکند و اجازه میدهد تا مدل فیچر های سطح پایین پریترین شده را حفظ کند و تنها در فیچر های سطح بالا تغییراتی ایجاد کند و ریسک اورفیت شدن مدل را باز هم کاهش دهد. در فرایند آموزش از هر دو این روش ها استفاده شده و بهترین مورد انتخاب شده است. همچنین یک راهکار دیگر 0 وارد کردن تعداد ایپاک های fineTuning است که این مورد نیز اعمال شد و در نهایت بهترین موارد انتخاب شد.

جهت لود مدل های پایه از کد های زیر استفاده شده است.

```

with tfl.device('/gpu:0'):
    nasNet_base_model = NASNetMobile(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
    NasNetMobile = Model(nasNet_base_model.input, nasNet_base_model.get_layer('avg_pool').output)

with tfl.device('/gpu:0'):
    EfficientNet_base_model = EfficientNetB6(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
    EfficientNetB6 = Model(EfficientNet_base_model.input, EfficientNet_base_model.get_layer('avg_pool').output)

```

```

with tfl.device('/gpu:0'):
    MobileNetV2_base_model = MobileNetV2(weights='imagenet', include_top=False, input_shape=(224, 224, 3), alpha=1.4)
    MobileNetV2 = Model(MobileNetV2_base_model.input, MobileNetV2_base_model.get_layer('avg_pool').output)

```

شایان ذکر است که مدل 6EfficientNetB در حالت اصلی خود ورودی با ابعاد 528*528 دریافت میکند که به علت تعداد پارامتر های بالا و کندی فرایند آموزش و مشکلات حافظه پیش آمده سایز ورودی کوچک تری مانند 224*224 را برای آن مشخص کردیم. همچنین برای مدل موبایل نت نیز از پارامتر آلفا برابر با 1.4 که بزرگترین مقدار آن است استفاده کردیم که در واقع بزرگترین حالت مدل موبایل نت را لود میکند و به دریافت دقیقی بالاتر کمک میکند.

همچنین در طی فرایند آموزش مدل از لرنینگ ریت اسکجولر نیز استفاده کرده بودیم که پس از استفاده از ارلی استاپینگ آن را حذف کردیم.

2-3. تحلیل نتایج

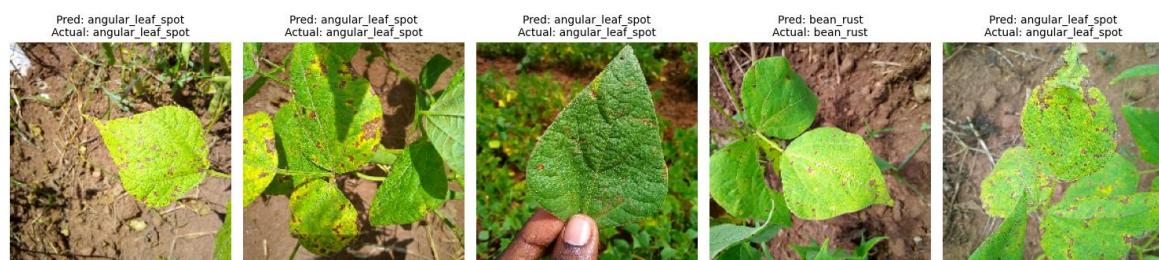
توجه شود تصاویر شامل نتیجه مدل روی 5 تصویر تصادفی، قابل تعمیم به عملکرد مدل نبوده و تنها دیدی نسبت به خروجی مدل می‌دهد. در این بخش، 9 مدل بررسی شده که

nasNet: مدل

Adam Optimizer:

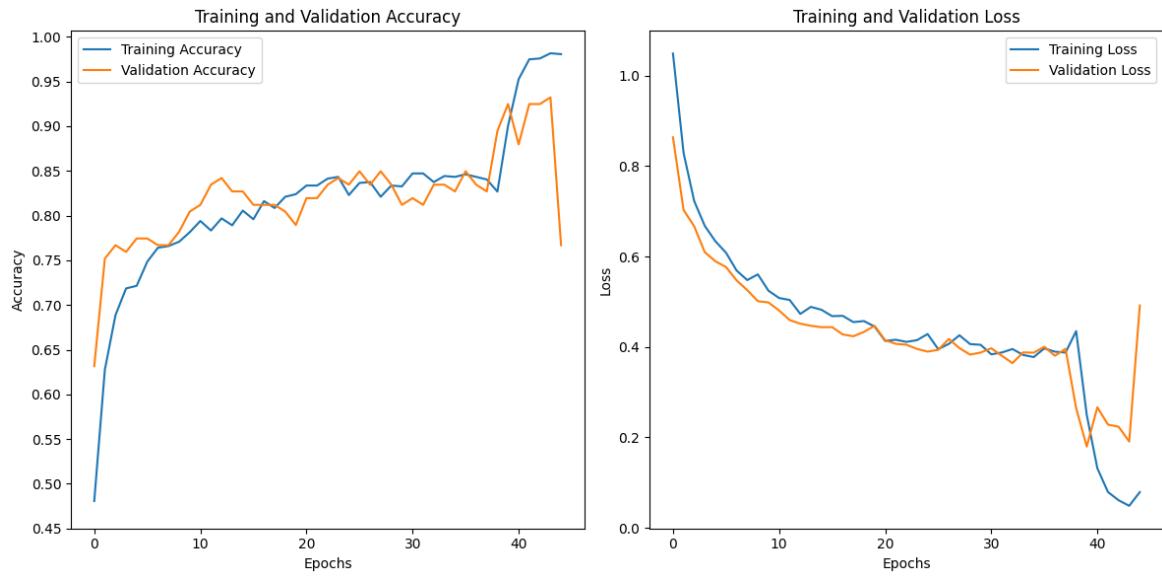


شکل 2-3. نمودار دقت و خطای مدل



شکل 2-3. نتیجه پیشینی مدل روی 5 تصویر همانطور که مشاهده می‌شود، این مدل روی تمامی 5 تصویر تصادفی نتیجه درست داده است.

Nadam Optimizer:



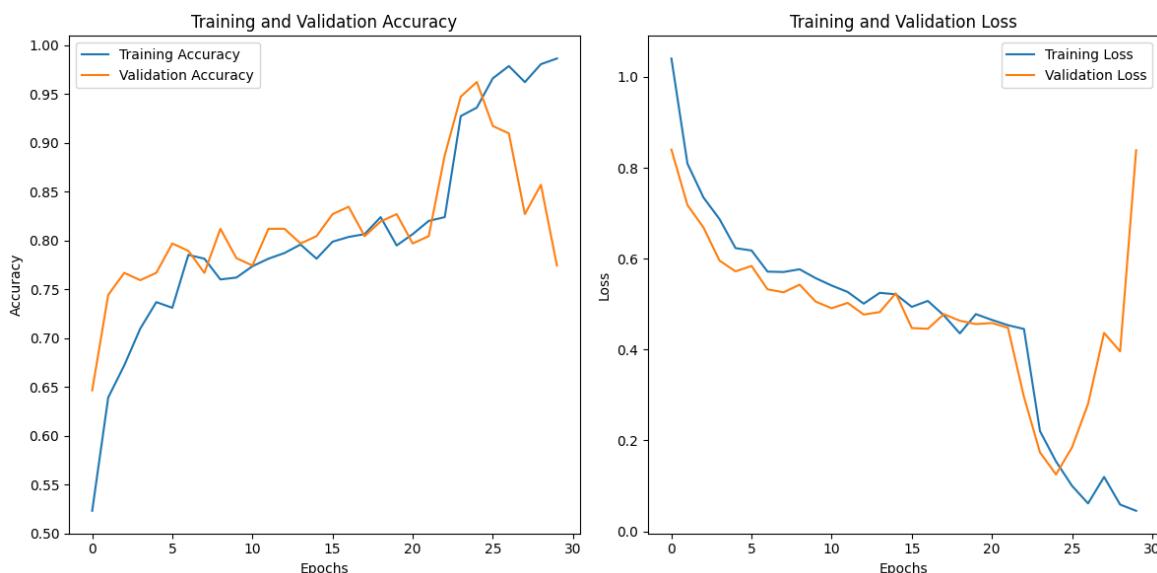
شکل 2-3. نمودار دقت و خطای مدل



شکل 2-3. نتیجه پیشینی مدل روی 5 تصویر

همانطور که مشاهده می شود، این مدل روی تنها 2 تصویر از بین 5 تصویر تصادفی نتیجه نادرست داده است.

RMS Optimizer:



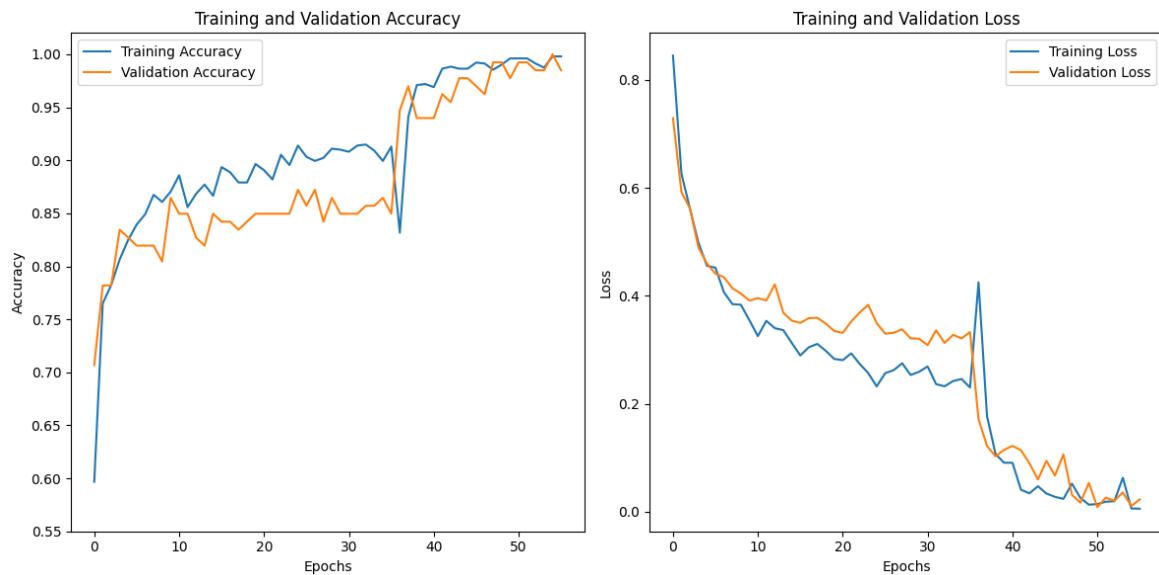
شکل 2-3. نمودار دقت و خطای مدل



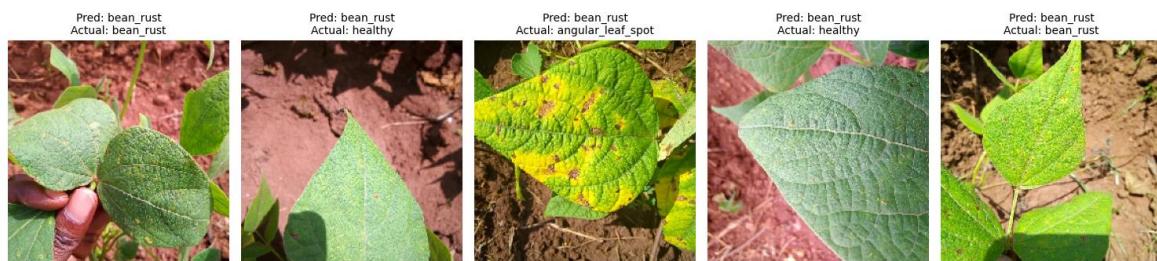
شکل 2-3. نتیجه پیشینی مدل روی 5 تصویر

همانطور که مشاهده می‌شود، این مدل روی تمامی این ۵ تصویر تصادفی نتیجه درست داده است.

EfficientNetB6: مدل RMS Optimizer:



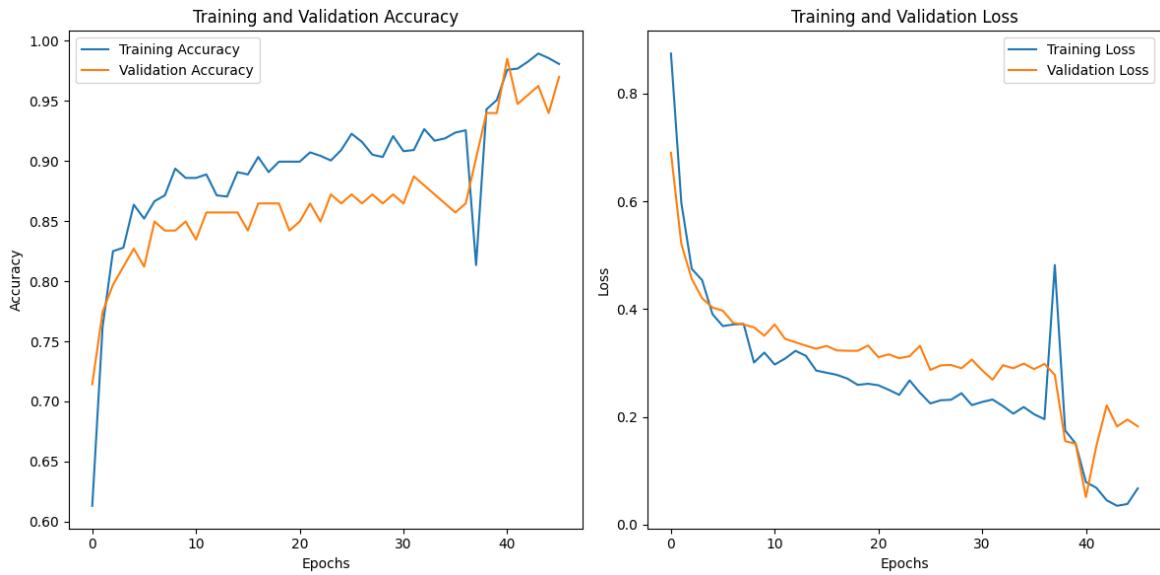
شکل 2-3. نمودار دقت و خطای مدل



شکل 2-3. نتیجه پیش‌بینی مدل روی ۵ تصویر

همانطور که مشاهده می‌شود، این مدل روی تنها ۳ تصویر از بین ۵ تصویر تصادفی نتیجه نادرست داده است.

Nadam Optimizer:



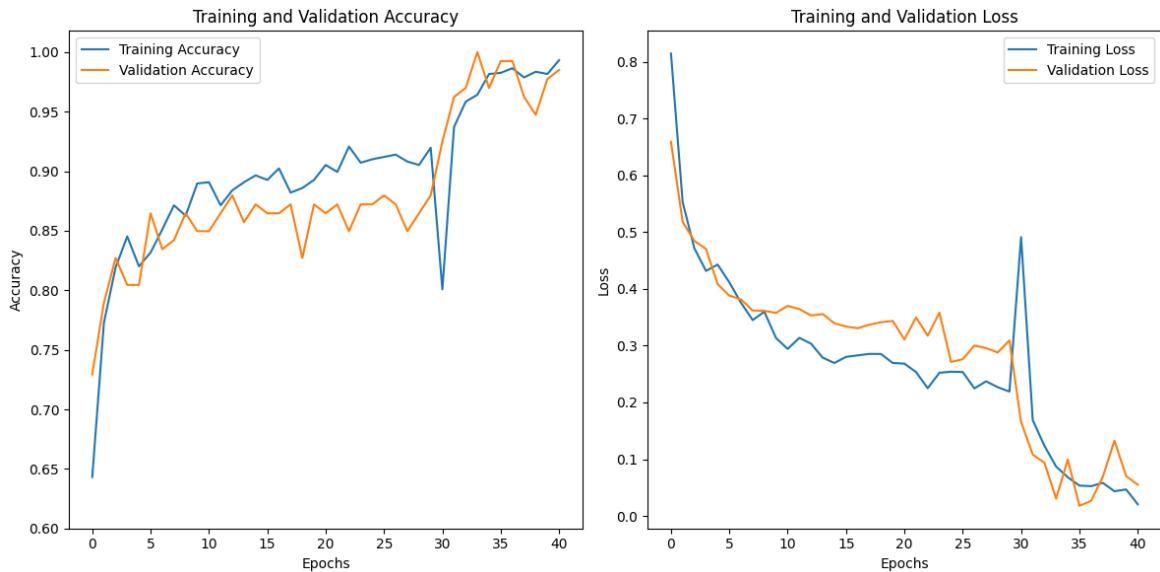
شکل 2-3. نمودار دقت و خطای مدل



شکل 2-3. نتیجه پیش‌بینی مدل روی 5 تصویر

همانطور که مشاهده می‌شود، این مدل روی تنها 2 تصویر از بین 5 تصویر تصادفی نتیجه نادرست داده است.

Adam Optimizer:



شکل 2-3. نمودار دقت و خطای مدل

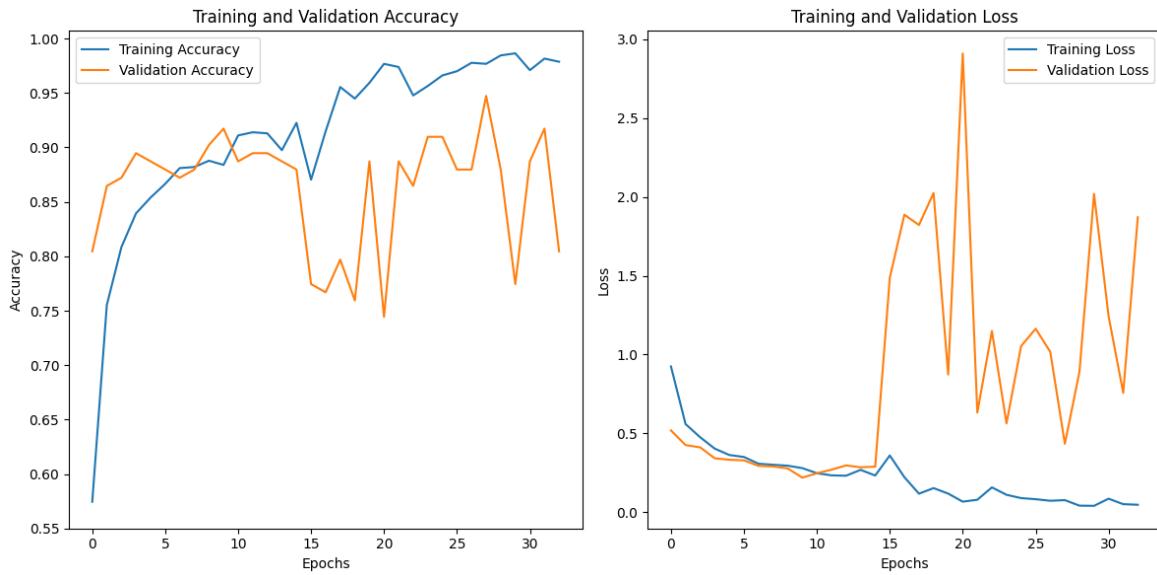


شکل 2-3. نتیجه پیش‌بینی مدل روی 5 تصویر

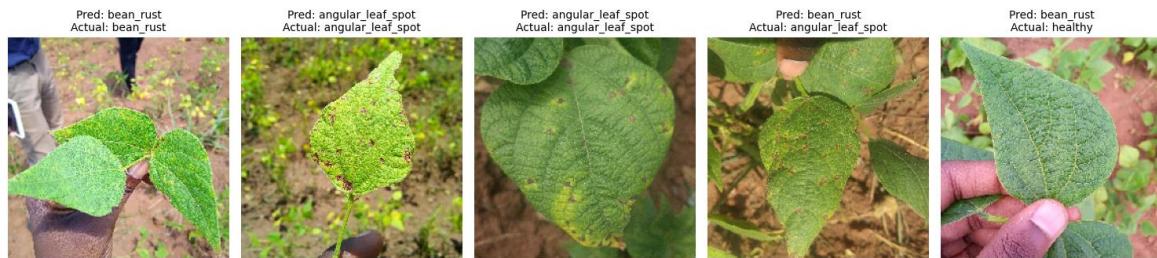
همانطور که مشاهده می‌شود، این مدل روی تنها 1 تصویر از بین 5 تصویر تصادفی نتیجه نادرست داده است.

MobileNetV2 : مدل

Adam Optimizer:



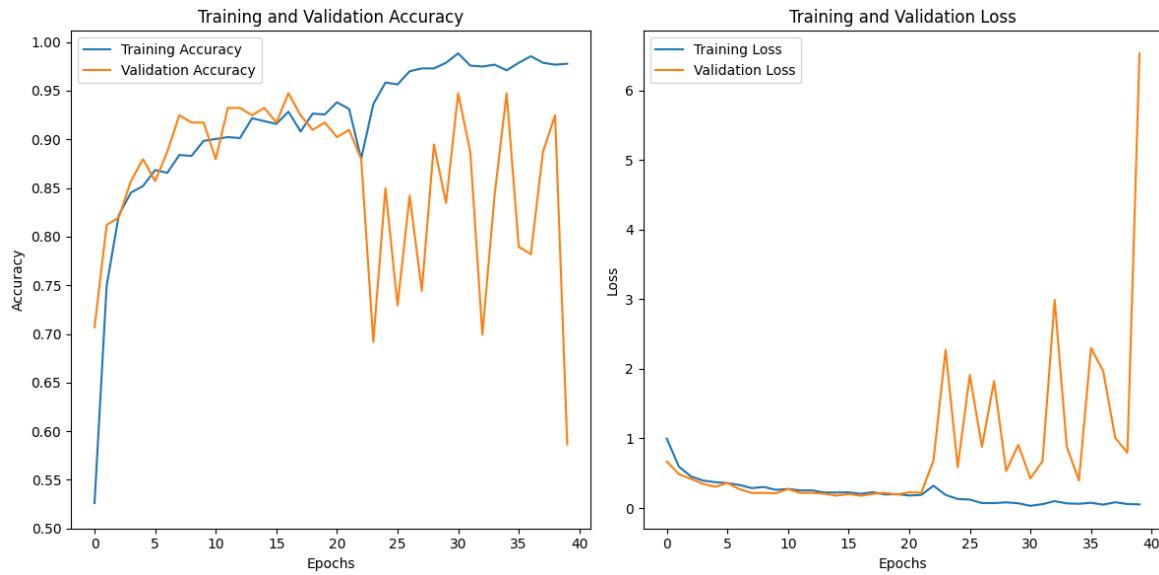
شکل 2-3. نمودار دقت و خطای مدل



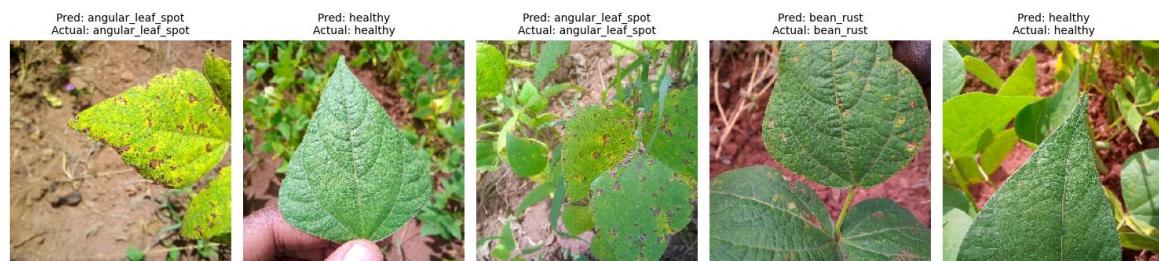
شکل 2-3. نتیجه پیش‌بینی مدل روی 5 تصویر

همانطور که مشاهده می‌شود، این مدل روی تنها 2 تصویر از بین 5 تصویر تصادفی نتیجه نادرست داده است.

Nadam Optimizer:



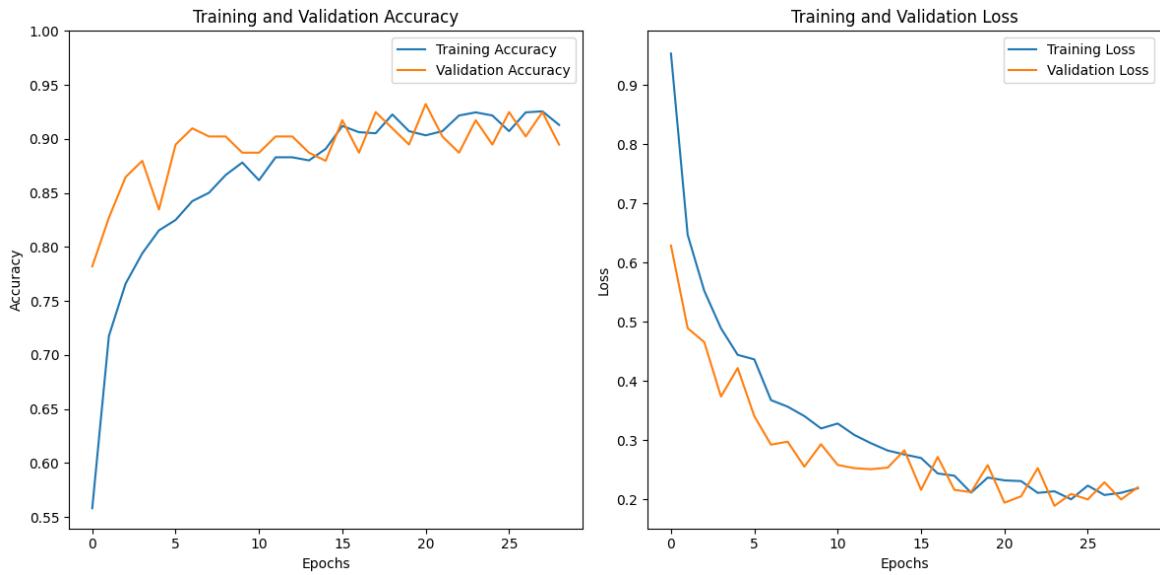
شکل 2-3. نمودار دقت و خطای مدل



شکل 2-3. نتیجه پیش‌بینی مدل روی 5 تصویر

همانطور که مشاهده می‌شود، این مدل روی تمامی این 5 تصویر تصادفی نتیجه درست داده است.

RMS Optimizer:



شکل 2-3. نمودار دقت و خطای مدل



شکل 2-3. نتیجه پیش‌بینی مدل روی 5 تصویر

همانطور که مشاهده می‌شود، این مدل تنها روی آخرین نمونه از این 5 تصویر نتیجه اشتباه داده است.

به بررسی دقت و خطای مدل‌های مختلف می‌پردازیم:

جدول 2-3. دقت و خطای مدل‌ها در داده آموزش و ارزیابی با بهینه‌سازهای مختلف

Optimizer	CNN Model	Train-Acc	Test-Acc	Train-Loss	Test-Loss
Adam	EfficientNetB6	99.30	92.97	0.0242	0.1690
	MobileNetV2	98.20	93.75	0.0403	0.2326
	NasNet	98.96	91.41	0.0420	0.2327
RMSprop	EfficientNetB6	99.79	95.31	0.0063	0.2984
	MobileNetV2	91.30	89.06	0.2070	0.4364
	NasNet	98.52	89.84	0.0437	0.3424
Nadam	EfficientNetB6	97.99	92.97	0.0601	0.2150
	MobileNetV2	97.88	94.53	0.0587	0.2735
	NasNet	97.92	86.72	0.0835	0.2892

همانطور که مشاهده می‌شود هر اپتیمایزر در یکی از مدل‌ها بهترین عملکرد را داشتند که این مسئله به دلیل randomness موجود در فرایند ترین و augmentation ها ای است که منجر به این رندوم نس می‌شوند. ولی به صورت کلی قابل مشاهده است که مدل adam به صورت پایسته

در تمامی مدل ها دقتی بالای 90 درصد دریافت کرده در حالی که دو مدل دیگر موارد پایین تری نیز داشتند پس از آن اپتیمایزر RMSprop عملکرد بهتری داشت و در نهایت .RMSprop

همچنین همانطور که قابل مشاهده است دو مدل MobileNetV2 و مدل EfficientNetB6 عملکرد نزدیکی داشتند و هردو قدری از NasNet عملکرد بهتری داشتند.

و به صورت کلی دستیابی به درصد های بالای 90 در تمامی مدل ها نشان دهنده ترین مناسب و فاین تیون شدن مناسب مدل ها است.