به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

درس شبکه‌های عصبی و یادگیری عمیق

تمرین دوم

پرسش ۱	نام و نام خانوادگی	علی صفری
	شماره دانشجویی	۸۱۰۲۰۲۱۵۳
پرسش ۲	نام و نام خانوادگی	حمیدرضا نادی مقدم
	شماره دانشجویی	۸۱۰۱۰۳۲۶۴
	مهلت ارسال پاسخ	**۱۴۰۳.۰۸.۲۹**

فهرست

شکل‌ها

جدول‌ها

مقدمه

برای پیاده‌سازی پروژه از بستر Kaggle برای پروژه اول و Google Colab برای پروژه دوم به منظور کد نویسی و اجرا استفاده شده است. تمامی مراحل کد و اجرای آن در این گزارش به تفصیل شرح داده شده است.

کد های نوشته شده همگی در پوشه‌ی Code و با پسوند ipynd. ذخیره شده است.

پرسش ۱. تشخیص ضایعه سرطانی با استفاده از CNN

در این سوال با توجه به مقاله و دیتاست داده شد با توجه به الگوریتم مورد استفاده در مدل ابتدا دیتاست را پیش پردازش کرده، در ادامه داده افزایی می‌کنیم و با پیاده سازی مدل مقاله و سپس مدل دیگری به تحلیل و مقایسه آنها می‌پردازیم. در گزارش این سوال تلاش شده تمامی مراحل به تفصیل و دقت گزارش شود اما کد تمامی بخش ها مشخصا در گزارش آورده نشده بلکه فقط بخش های حیاتی و تاثیر گذار که مورد سوال بوده آورده شده است. بخشی از کدها صرفا برای نمایش بهتر و به جهت مقایسه ساده تر نوشته شده اند که از توضیح آنها در کد صرف نظر شده است. در این سوال در بخش چهارم یک مدل که مشابه مقاله است نوشته شده است که در ادامه تحت عنوان مدل مقاله از آن یاد خواهد شد و سپس مدلی که مشابه ساختار مقاله اما با تفاوت هایی به جهت بهبود عملکرد ساخته و آموزش داده شده است که در ادامه تحت عنوان مدل بهبود یافته از آن یاد خواهد شد.

مدل اولیه که همان مدل مقاله است در فایل HW2_1_main ذخیره شده است. بخش بهبود یافته در فایل HW2_1_improved و بخش امتیازی در فایل HW2_1_Extra می‌باشد.

۱-۱. پیش پردازش تصاویر

در بخش پیش پردازش می‌بینیم که سه عملیات استفاده شده است.

۱-۱-۱- **تغییر ابعاد عکس (Resizing):** عکس‌های گرفته شده ابعاد و resolution متفاوتی دارند، از طرفی ما برای استفاده از عکس‌ها به عنوان ورودی در شبکه خود نیاز داریم همگی عکس‌ها ابعاد یکسان و یکنواختی داشته باشند. با این کار شبکه برای ورودی خود تعداد پیکسل یکسانی دارد و برای آموزش، درگیر ابعادهای مختلف و دیتای اضافی نخواهد شد. برای این کار تمامی عکس‌های موجود در دیتاست را با توجه به الگوریتم نوشته شده در مقاله مهم این است که همه عکس‌ها یک سایز داشته باشند. با توجه به جدول ۲ مقاله ابعاد ۲۸ × ۲۸پیشنهاد شده و انتخاب شده است. برای پیاده سازی این عملیات روی عکس از کتابخانه cv2 استفاده شده است که عکس را دریافت کرده و سپس با استفاده از تابع resize آن را به ابعاد کوچکتر و ثابتی تبدیل کردیم.

۱-۱-۲- **نرمال کردن (Normalization):** نرمال کردن عکس‌ها بدین صورت است که مقدار عددی هر پیکسل عددی بین صفر و یک خواهد شد. در ابتدا هر پیکسل می‌تواند در بازه عددی ۰ تا

۲۵۵ باشد، برای اینکه عکس نرمال شود هر پیکسل به عدد ۲۵۵ تقسیم شده و این عملیات صورت می‌پزیرد. دلیل اهمیت و مفید بودن این عملیات ازین جهت است که با کوچک‌تر کردن value، بازه‌ی وزن‌های اولیه و در نهایت وزن نهایی نورون‌ها را کوچک تر کرده و در نهایت عدد وزن‌ها کوچک تر می‌شود. از آن مهم‌تر این کار باعث افزایش سرعت همگرایی مدل خواهد شد. یکی از دلایل این موضوع همان موضوع کوچک‌تر بودن وزن نورون‌ها است. با کوچک بودن وزن، عملیات گرادیان در پروسه backpropagation قابل مدیریت‌تر بوده و در نهایت به smooth تر شدن و سریع‌تر شدن فرایند بهینه‌سازی کمک خواهد کرد. برای پیاده سازی این عملیات عکس resize شده در مرحله قبل را به عدد ۲۵۵ تقسیم کرده ایم.

۱-۱-۳-**داده افزایی (Data Augmentation)**: این فرایند در بخش بعد به تفصیل مورد بحث قرار خواهد گرفت اما از آنجایی که به نوعی یکی از فرایندهای پیش پردازش و قبل از پیاده سازی مدل به شمار می‌رود در این بخش آورده شده است. از مزیت‌های اصلی این روش می‌توان به غنی‌تر شدن و کامل‌تر شدن دیتاست اشاره کرد. همچنین این عملیات می‌تواند مانند regularization عمل کند و مدل ما را از خطر overfit شدن دور نگه دارد. کلیت کار در این روش بدین صورت است که ما یک سری عملیات روی عکس‌های موجود پیاده سازی می‌کنیم تا دیتای بیشتری در اختیار داشته باشیم. این عملیات‌ها اعم است از زوم کردن، دوران تصادفی عکس و تغییراتی در adjustment عکس مانند saturation و contrast.

```python
def preprocess_images(images, target_size=(28, 28)):
    """
    Preprocess images for model input
    Args:
        images: numpy array of images
        target_size: tuple of target image dimensions
    Returns:
        preprocessed_images: numpy array of preprocessed images
    """
    preprocessed_images = []
    for img in tqdm(images):
        # Resize image
        img_resized = cv2.resize(img, target_size)

        # Normalize pixel values
        img_normalized = img_resized / 255.0

        preprocessed_images.append(img_normalized)

    return np.array(preprocessed_images)
```

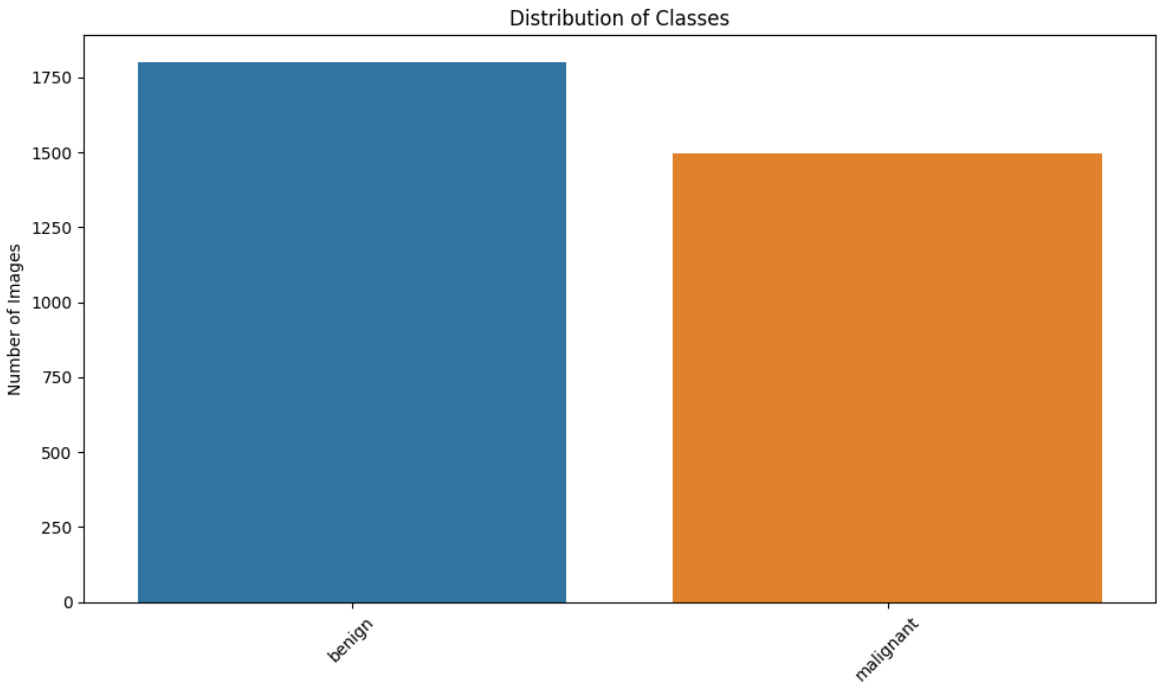
شکل ۱- پیش پردازش‌های اولیه انجام شده در مقاله - **resize** و **normalization**

در این بخش عملیات resizing و normalization را روی دیتاست انجام می‌دهیم سپس EDA و در نهایت اگر عملیات دیگری به بهبود مدل کمک می‌کرد آن را پیاده سازی خواهیم کرد.

مراحل اجرای عملیات EDA به این منظور است که یک دید مناسبی نسبت به دیتای خود پیدا کنیم و با توجه به آن تصمیماتی در رابطه با preprocessing بگیریم. برای این کار یکی از ساده ترین کارهای ممکن ترسیم نمودار توزیع داده‌ها است. بدین صورت که چند درصد از کل مجموعه در کلاس benign و چند درصد در کلاس malignant هستند.

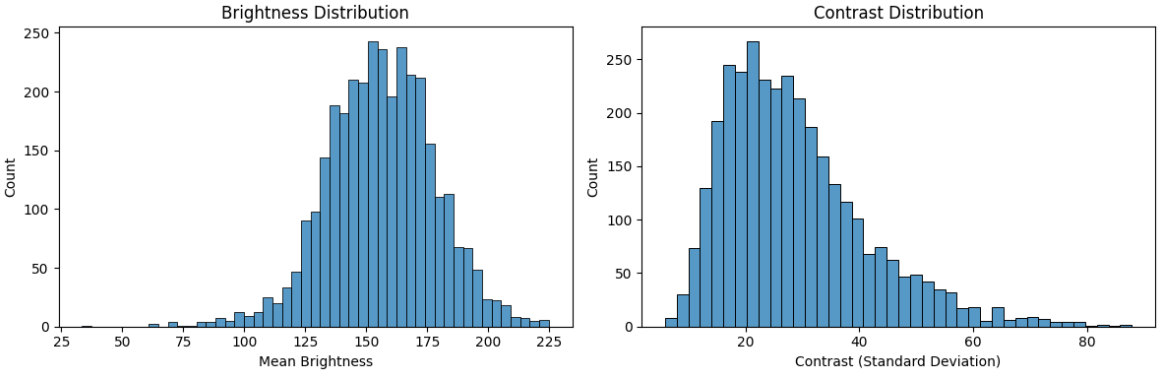
```
=== Dataset Analysis ===

Total number of images: 3297
Image dimensions: (224, 224, 3)
Number of classes: 2

Class Distribution:
benign: 1800 images (54.60%)
malignant: 1497 images (45.40%)
```

شکل ۲- آنالیز اولیه روی دیتا



شکل ۳ – نمودار میله‌ای توزیع دیتا

یکی دیگر از عملیات‌های قابل انجام برای بررسی دیتا، بررسی نحوه توزیع درخشندگی و تفاوت رنگ ها (contrast) در عکس‌ها است.



شکل ۴ - توزیع **Brightness** و **Contrast**

با توجه به نمودارهای ترسیم شده برای آنالیز دیتا می‌بینیم که اولا تعداد داده‌های از هر گروه یکسان نیست. علاوه بر این توزیع درخشنگی تا حدود خوبی نرمال است. همچنین توزیع contrast هم می‌بینیم که چندان نرمال نیست. پس از جمله پیش پردازش‌هایی که احتمالا می‌تواند کیفیت داده‌های ما را بهبود بخشد می‌توان به Balance کرده داده‌ها به معنای برابر کردن دیتای هر کلاس استفاده کرد. همچنین عملیات افزایش کیفیت یا CLAHE نیز می‌تواند به یکنواخت تر شدن توزیع و افزایش شدت تفاوت در عکس‌ها جهت تشخیص بهتر الگوها کمک کند. برای پیاده‌سازی این دو پیش پردازش می‌توانیم به صورت زیر عمل می‌کنیم.

۱) **مدیریت کلاس نامتوازن:** برای این کار ابتدا بعد از یک پردازش اولیه و تشخیص کلاسی که تعداد داده کمتری دارد به سراغ عملیات resample کردن می‌رویم. برای این کار دیتای اصلی را حفظ کرده و سپس به صورت رندوم از بین دیتای موجود کپی ایجاد می‌کنیم و داده را افزایش می‌دهیم. این مرحله تاجایی ادامه پیدا می‌کند که کلاسی که تعداد داده کمتری داشته برابر با کلاس بزرگ‌تر شود. این عملیات از این جهت به فرایند کار کمک می‌کند که دیتای ما بیشتر شده و عملیات تشخیص الگو برای هر دو کلاس بهتر انجام می‌شود.

```python
def handle_class_imbalance(images, labels, class_distribution):
    """
    Handle class imbalance by upsampling minority class to match majority class
    Args:
        images: numpy array of images
        labels: one-hot encoded labels
        class_distribution: dictionary of class distributions
    Returns:
        balanced images and labels
    """
    # Get class counts from one-hot labels
    class_counts = np.sum(labels, axis=0)
    majority_size = int(max(class_counts))

    # Get indices for each class
    class_indices = [np.where(labels[:, i] == 1)[0] for i in range(len(class_counts))]

    balanced_images = []
    balanced_labels = []

    for i, indices in enumerate(class_indices):
        current_size = len(indices)
        if current_size < majority_size:
            # Oversample minority class
            n_samples_needed = majority_size - current_size
            resampled_indices = np.random.choice(indices, n_samples_needed, replace=True)
            all_indices = np.concatenate([indices, resampled_indices])
        else:
            all_indices = indices

        balanced_images.extend(images[all_indices])
        balanced_labels.extend(labels[all_indices])

    balanced_images = np.array(balanced_images)
    balanced_labels = np.array(balanced_labels)

    print("\nAfter balancing:")
    print(f"Total samples: {len(balanced_images)}")
    print(f"Class distribution: {np.sum(balanced_labels, axis=0)}")

    return balanced_images, balanced_labels
```

شکل ۵ - مدیریت و بالانس کردن کلاس‌ها

همانطور که در کد مشخص است عملیات این تابع به این صورت است که دیتا را که به صورت one hot هم در آمده به عنوان ورودی می‌گیرید و ابتدا بزرگترین کلاس و تعداد دیتای این کلاس را انتخاب می‌کند سپس به کلاسی که تعداد دیتای کمتری دارد به صورت رندوم و از بین دیتای خود آن کلاس آنقدر دیتا اضافه می‌کنیم تا این دو کلاس با هم تعداد دیتای برابری داشته باشند.

```
Handling class imbalance...

After balancing:
Total samples: 3600
Class distribution: [1800. 1800.]
```

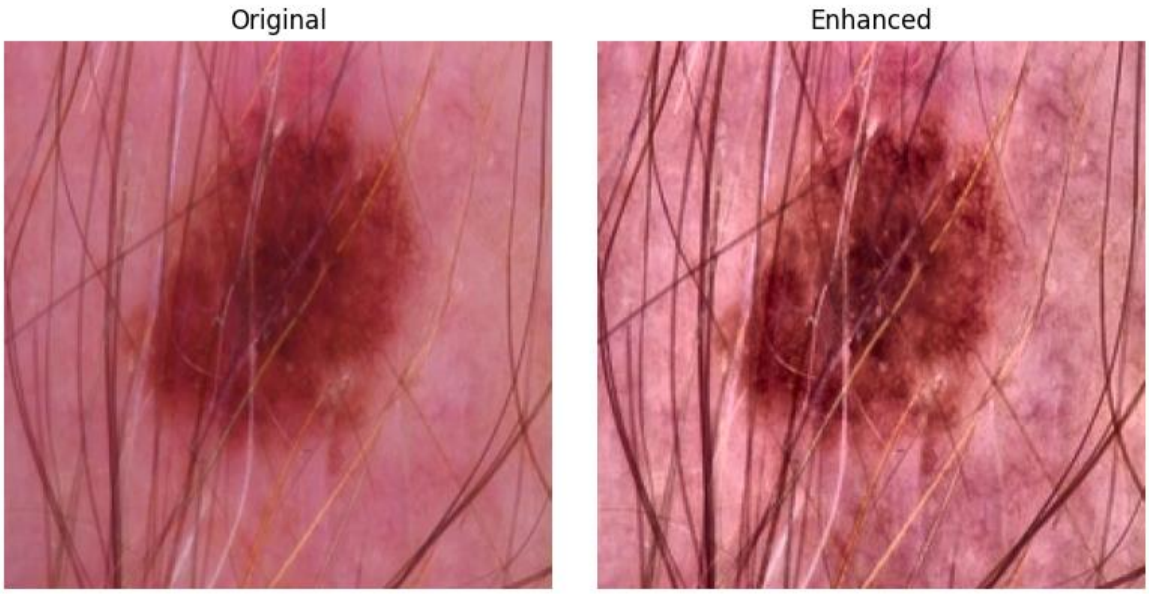
شکل ۶ - نتیجه بالانس کردن کلاس‌ها

۲) **عملیات CLAHE**: این یک عملیات پیشرفته‌تر هست که برای پیاده‌سازی آن از توابع کتابخانه استفاده شده است. عملیات این فرایند بدین صورت است که عکس از فضای RGB به فضای LAB می‌رود و در این فضا با افزایش contrast تفاوت‌های موجود در عکس و لبه‌های مهم بیشتر به چشم می‌آید و شبکه بهتر می‌تواند تفاوت‌ها را دیده و الگوی تکرار شده در تصاویر را درک و استخراج کند. نکته دیگر در این روش حفظ رنگ‌های اصلی می‌باشد. همچنین طبق نمودار توزیع brightness می‌بینیم تعدادی از عکس‌ها هستند که نور مناسبی ندارند و این عملیات به افزایش بهینه brightness این عکس‌ها در صورت نیاز هم کمک خواهد کرد.

```python
def enhance_image_quality(image):
    """
    Enhance image quality using CLAHE
    Args:
        image: input image
    Returns:
        enhanced image
    """
    # Convert to LAB color space
    lab = cv2.cvtColor(image, cv2.COLOR_RGB2LAB)

    # Apply CLAHE to L channel
    clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))
    lab[...,0] = clahe.apply(lab[...,0])

    # Convert back to RGB
    enhanced = cv2.cvtColor(lab, cv2.COLOR_LAB2RGB)

    return enhanced
```

شکل ۷ - تابع افزایش کیفیت عکس **CLAHE**

برای این عملیات از کتابخانه cv2 استفاده شده و ابتدا gridsize کوچکی مانند ۸ در ۸ انتخاب شده و سپس با استفاده از تابع این کتابخانه این عملیات انجام شده و کیفیت عکس افزایش یافته است.



شکل ۸ - نمونه‌ای از عکس افزایش کیفیت یافته به روش **CLAHE**

با انجام این دو فرایند پیش پردازش دیتا به شکل خوبی انجام شده و دیتا آماده‌ی بقیه عملیات خواهد شد.

۱-۲. داده افزایی

داده افزایی یا Data Augmentation به معنای افزایش داده‌هاست. برای انجام این کار می‌توان از عکس‌های موجود که دارای لیبل هستند عکس‌های دیگری ایجاد کرد. به عنوان مثال می‌توان دوران انجام داد، تصویر را جابه‌جا کرد، flip انجام داد و یا نور تصویر را کم و زیاد کرد و از این دست عملیات‌ها. حال چرا نیاز به انجام این کار داریم و تاثیراتی که این کار دارد می‌توان به موارد زیر اشاره کرد.

۱-۲-۱- جلوگیری از overfit شدن: با افزایش داده‌های متنوع با ویژگی‌های خاص بیشتر به مدل کمک می‌کنیم که بجای حفظ کردن الگوهای اولیه و همیشگی حالت‌های متفاوتی را ببیند و الگوهای اصلی را یاد بگیرد.

۱-۲-۲- ا فزایش اندازه مجموعه داده: خیلی اوقات به دست آوردن دیتای زیاد و به اندازه هزینه و زمان زیادی را از ما می‌گیرد. با استفاده از این تکنیک با تعداد اولیه دیتای مشخص می‌توانیم یک مجموعه داده غنی ایجاد کنیم تا حالت‌های مختلف و الگوها را درک کند.

۱-۲-۳- بهبود تعمیم پذیری (Improving Generalization): در شرایط واقعی عکس‌ها ویژگی‌ها و زاویه‌های متفاوتی دارند. با این تکنیک ما مدلی را آموزش می‌دهیم که این حالت‌ها را خیلی کامل تر دیده و قابلیت تعمیم بسیار خوبی دارد.

۱-۲-۴- رفع مشکل عدم توازن کلاس‌ها: در برخی از شرایط ما از یک کلاس خاص تعداد دیتای کمتری داریم و این باعث عدم توازن در بین کلاس‌ها می‌شود. این کار باعث می‌شود مدل در آموزش و تشخیص آن کلاس به مشکل بر بخورد اما با این کار می‌توانیم این مشکل را به خوبی حل کنیم.

نکته مهم دیگر این است که ما به صورت کلی دو نوع Augmentation داریم.

۱) static augmentation ۲) dynamic augmentation

در حالت استاتیک عملیات بدین صورت است که ابتدا و به صورت آفلاین عملیات داده افزایی روی عکس‌ها انجام شده و در قسمتی ذخیره می‌شود. با این کار می‌توان فقط یکبار عملیات داده افزایی را انجام داد و از نظر صرف زمان و هزینه وقتی منابع محدود باشد می‌تواند مفید باشد.

در حالت dynamic اما عملیات به صورت on the fly و در فرایند training بدین صورت انجام می‌شود که در هر epoch عملیات داده افزایی به صورت کاملا رندوم روی یک سری از عکس‌ها صورت گرفته و بدین ترتیب پراکندگی این عملیات داده افزایی به شکل خیلی گسترده تری انجام می‌شود که باعث بهتری شدن قابلیت تعمیم مدل ما خواهد شد. به همین جهت در این پروژه ما از Dynamic Augmentation استفاده کرده ایم.

```
Example with 5 images in dataset:

EPOCH 1:
Step 1: Original Image 1 → Gets augmented → Model trains on ONLY the augmented version
Step 2: Original Image 2 → Gets augmented → Model trains on ONLY the augmented version
Step 3: Original Image 3 → Gets augmented → Model trains on ONLY the augmented version
Step 4: Original Image 4 → Gets augmented → Model trains on ONLY the augmented version
Step 5: Original Image 5 → Gets augmented → Model trains on ONLY the augmented version

EPOCH 2:
Step 1: Original Image 1 → Gets differently augmented → Model trains on ONLY the new augmented version
Step 2: Original Image 2 → Gets differently augmented → Model trains on ONLY the new augmented version
Step 3: Original Image 3 → Gets differently augmented → Model trains on ONLY the new augmented version
Step 4: Original Image 4 → Gets differently augmented → Model trains on ONLY the new augmented version
Step 5: Original Image 5 → Gets differently augmented → Model trains on ONLY the new augmented version

=== KEY POINTS ===
1. In each epoch:
   - Each original image is used exactly once
   - It gets randomly augmented
   - Only the augmented version is used for training
   - Original image is NOT used for training

2. If you have 1000 images:
   - Each epoch still processes 1000 images (augmented versions)
   - NOT 2000 images (it doesn't use both original and augmented)

3. The benefit comes from:
   - Each epoch sees different variations of the same images
   - But the number of images per epoch stays the same
```

شکل ۹ - شفاف‌سازی عملیات داده افزایی

اما درباره این سوال که چه نوع Augmentation مناسب است و دلیل مناسب بودن آن، می‌توان به موارد زیر اشاره کرد.

- دوران عکس
- شیفت در عرض: ۰.۲ در رنج خود عکس چپ و راست می‌رود.
- شیفت در ارتفاع: ۰.۲ در رنج خود عکس بالا و پایین می‌رود.
- عملیات flip به صورت افقی

در رابطه با پیاده سازی این عملیات به دو صورت می‌توانیم عمل کنیم. روش اول استفاده از ImageDataGenerator می‌باشد. یکی از مزیت‌های این روش امکان تغییر عکس به سادگی و به صورت خیلی smooth است اما در صورت استفاده از dataloader ها امکان استفاده مستقیم از این متد وجود ندارد. به همین دلیل جایی که نیاز به استفاده از Dataloader داریم به سراغ روش دوم می‌رویم برای پیاده‌سازی.

```python
def create_data_generators(train_images, train_labels, val_images, val_labels,
                           batch_size=128):
    """
    Create data generators for training and validation
    Args:
        train_images: numpy array of training images
        train_labels: numpy array of training labels
        val_images: numpy array of validation images
        val_labels: numpy array of validation labels
        batch_size: batch size for training
    Returns:
        train_generator: training data generator
        val_generator: validation data generator
    """
    # Data augmentation for training
    train_datagen = ImageDataGenerator(
        rotation_range=20,
        width_shift_range=0.2,
        height_shift_range=0.2,
        horizontal_flip=True,
        vertical_flip=False,
        fill_mode='nearest'
    )

    # Only normalization for validation
    val_datagen = ImageDataGenerator()

    # Create generators
    train_generator = train_datagen.flow(
        train_images, train_labels,
        batch_size=batch_size,
        shuffle=True
    )

    val_generator = val_datagen.flow(
        val_images, val_labels,
        batch_size=batch_size,
        shuffle=False
    )

    return train_generator, val_generator
```

شکل ۱۰ - عملیات **data augmentation** – روش اول

روش دوم که برای جایی است که نیاز به dataloader ها است. در این روش به صورت زیر عمل می‌کنیم اما قبل از آن نکته حائز اهمیت تفاوت اندک چگونگی عملکرد این دو روش است. در این روش دوران متفاوت انجام شده است و به جای دوران اندک عکس به صورت smooth بین منفی ۲۰ و ۲۰ درجه ۹۰ درجه مثبت و منفی دوران کردیم.

```python
# Convert to float32 if needed
image = tf.cast(image, tf.float32)
# 1. Random horizontal flip
if tf.random.uniform([]) > 0.5:
    image = tf.image.flip_left_right(image)
# 2. Random width shift (20%)
if tf.random.uniform([]) > 0.5:
    w_shift = tf.random.uniform([], -0.2, 0.2)
    image = tf.image.stateless_random_crop(
        tf.pad(image, [[0, 0], [2, 2], [0, 0]], mode='SYMMETRIC'),
        size=tf.shape(image),
        seed=tf.random.uniform([2], 0, 1000, dtype=tf.int32)
    )
# 3. Random height shift (20%)
if tf.random.uniform([]) > 0.5:
    h_shift = tf.random.uniform([], -0.2, 0.2)
    image = tf.image.stateless_random_crop(
        tf.pad(image, [[2, 2], [0, 0], [0, 0]], mode='SYMMETRIC'),
        size=tf.shape(image),
        seed=tf.random.uniform([2], 0, 1000, dtype=tf.int32)
    )
# 4. Random rotation (90 and -90 degrees)
if tf.random.uniform([]) > 0.5:
    k = tf.random.uniform([], 0, 2, dtype=tf.int32) * 2 - 1  # Generate -1 or 1
    image = tf.image.rot90(image, k)  # Rotate by -90 (k=-1) or 90 (k=1)
# Ensure output shape is correct and values are in valid range
image = tf.clip_by_value(image, 0, 1)
```

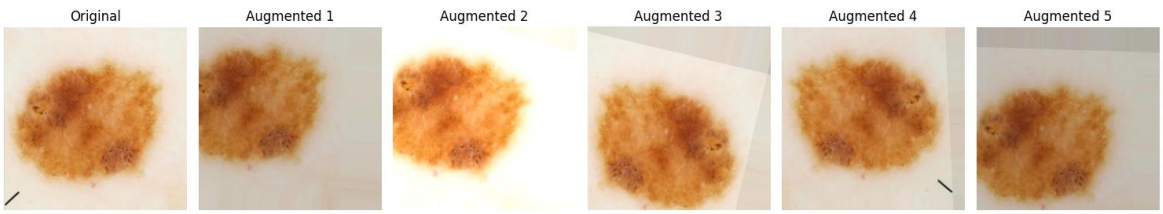
شکل ۱۱- پیاده سازی **Augmentation** در صورت استفاده از **Dataloader** – روش دوم

اینکه چرا از دیتا لودر استفاده می‌کنیم در بخش بعد مفصل توضیح داده خواهد شد.

تمام عملیات‌های گفته شده در تابع فوق پیاده سازی شده است. نکته دیگر اینکه batch size با توجه به مقاله عدد ۱۲۸ انتخاب شده است و عملیات داده افزایی به صورت رندوم و فقط برای بخش train و با استفاده از Augmentation انجام گرفته است و بخش validation شامل این عملیات نمی‌شود.

نوع دیگری از تغییرات که می‌توانست اعمال شود تغییر در brighness عکس‌ها بود که با این کار هم در ابتدا انجام شد اما شبکه به دقت بسیار کمتری نسبت به بدون تغییر این پارامتر رسید که نشان می‌دهد میزان درخشندگی و نور در عکس پارامتر بسیار مهمی در تصویر برداری است و نمی‌توان به سادگی از آن در بخش داده افزایی استفاده کرد.

همچنین عملیات flip فقط به صورت افقی انجام شده چرا که انجام flip به صورت عمودی می‌تواند ساختار و راستای ترک‌ها و زخم ایجاد شده روی پوست را تغییر دهد و به صورت کلی این راستا یکی از پارامترهایی است که می‌تواند در پیدا کردن الگو و تشخیص بهتر به مدل کمک کند پس از vertical flip استفاده نکرده ایم.



شکل ۱۲- نمونه ای از داده افزایی روی یکی از عکس‌های دیتاست

به عنوان نمونه عملیات داده افزایی روی یکی از عکس‌ها را در تصویر بالا می‌بینیم. به ترتیب از چپ به راست عملیات‌های انجام شده عبارت اند از: width shift, brightness change, rotation, horizontal flip, height shift.

البته عملیات تغییر در brightness همانطور که گفته شد در کل دیتاست اعمال نشده است چرا که همانطور که در نمونه هم مشاهده می‌شود این تغییر ماهیت و الگوهای کلی عکس را عوض کرده و کار شبکه برای تشخیص را به شدت سخت می‌کند. چنانچه یک بار این عملیات با این تغییر هم تست شد که دقت مدل به شکل چشم گیری کمتر از حالت بدون این تغییر بود. به صورتی که در صورت اعمال تغییر در brightness در عکس در بازه ۰.۸ تا ۱.۲ دقت مدل را از حدود ۸۶ درصد به ۵۵ درصد افت کرد.

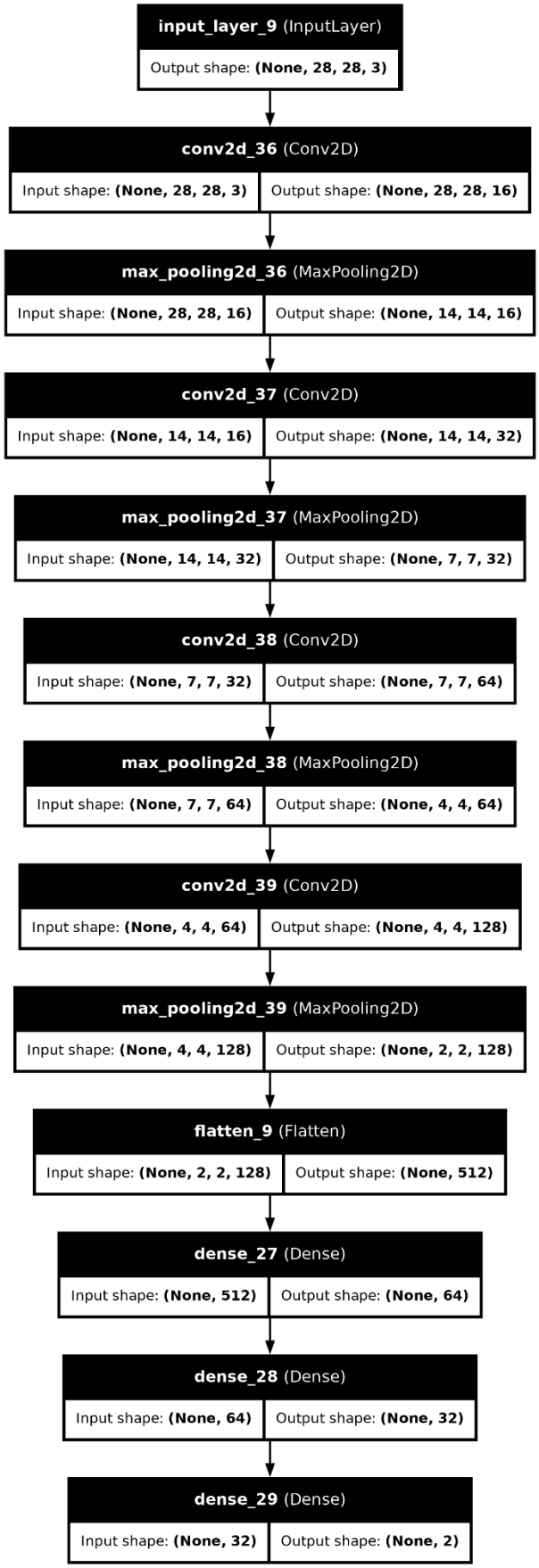
۱-۳. پیاده‌سازی

برای پیاده‌سازی مدل از مدل استفاده شده در جدول ۲ مقاله استفاده کرده‌ایم. نکته حائز اهمیت تفاوت مدل مقاله و مدلی است که ما پیاده‌سازی کرده‌ایم. اولین تفاوت در دیتاست مورد استفاده است که در مقاله دیتاستی با ۱۰۰۰۰ دیتا و همچنین با ۷ کلاس متفاوت استفاده شده است اما ما از دیتاستی استفاده می‌کنیم که ۳۲۹۷ عکس دارد و تنها ۲ کلاس. به دلیل دو کلاس یا binary بودن دیتاست مورد استفاده ما به عنوان تابع loss هم به جای categorical_crossentropy از binary_corssentropy استفاده شده است. باز هم به همین دلیل one hot موجود در مدل مقاله با ۷ کلاس بوده ولی برای ما با دو کلاس خواهد بود. محل دیگری که می‌توانستیم تغییری داشته باشیم در تابع فعال‌ساز لایه خروجی بود که اگر بجای تبدیل دیتا به one hot از دیتا به صورت binary استفاده می‌کردیم می‌توانستیم به جای softmax مورد استفاده قرار گرفته در مقاله از sigmoid استفاده کنیم اما نظر به اینکه تابع softmax گزینه پایدارتر و مورد اعتمادتری است و همینطور در تست انجام شده در نهایت دقت بالاتری را نتیجه داد، از همین تابع به عنوان فعال‌ساز لایه اخر استفاده کردیم.

در ادامه به شرح دقیق ساختار مدل و چگونگی پیاده‌سازی آن اشاره خواهیم کرد.

Layer (type)	Output Shape	Param #
input_layer_9 (InputLayer)	(None, 28, 28, 3)	0
conv2d_36 (Conv2D)	(None, 28, 28, 16)	448
max_pooling2d_36 (MaxPooling2D)	(None, 14, 14, 16)	0
conv2d_37 (Conv2D)	(None, 14, 14, 32)	4,640
max_pooling2d_37 (MaxPooling2D)	(None, 7, 7, 32)	0
conv2d_38 (Conv2D)	(None, 7, 7, 64)	18,496
max_pooling2d_38 (MaxPooling2D)	(None, 4, 4, 64)	0
conv2d_39 (Conv2D)	(None, 4, 4, 128)	73,856
max_pooling2d_39 (MaxPooling2D)	(None, 2, 2, 128)	0
flatten_9 (Flatten)	(None, 512)	0
dense_27 (Dense)	(None, 64)	32,832
dense_28 (Dense)	(None, 32)	2,080
dense_29 (Dense)	(None, 2)	66

شکل ۱۳ - ساختار مورد استفاده قرار گرفته در مقاله



شکل ۱۴- ساختار لایه‌های بکار رفته در کد

ساختار بکار رفته در کد این بخش دقیقا مشابه مقاله و مطابق نمودار فوق. کرنل سایز ۳ در ۳ و اولین لایه دارای ۱۶ فیلتر و همینطور با padding = same. نکته مهم استفاده از این پارامتر پدینگ هست که چون خروجی طبق جدول بعد از هر مرحله‌ی convolutional ثابت مانده است از این پارامتر استفاده شده (در تمام لایه های convolutional). در گام بعد maxpooling به کار گرفته شده است که با ابعاد ۲ در ۲ و Stride ۲ در ۲. به این صورت عرض و ارتفاع ما نصف خواهد شد. لایه بعدی دارای ۳۲ فیلتر است و بعد از maxpooling به ابعاد ۳۲ ×۷ × ۷ خواهیم رسید. در گام بعد ۶۴ فیلتر خواهیم داشت اما نکته در بخش maxpooling است که اگر حالت عادی پیش برویم به ابعاد ۶۴ × ۳ × ۳ خواهیم رسید که طبق جدول ۲ مدنظر ما نبوده پس در این قسمت برای maxpooling هم از padding = same استفاده می‌کنیم. با این کار به ساختار ۶۴ × ۴ × ۴ خواهیم رسید که مدنظر مقاله است. لایه آخر convolution هم با ۱۲۸ فیلتر اجرا خواهد شد و در نهایت با maxpooling اخر به یک لایه ۱۲۸ × ۲ × ۲ رسیده‌ایم. سپس این لایه را به صورت خطی در آورده و با ۵۱۲ ورودی به سراغ بخش fully connected رفته‌ایم. برای این بخش هم طبق مقاله دو لایه پنهان با ۶۴ و بعدی ۳۲ نورون است. لایه خروجی هم با ۲ نورون که همان دو کلاس ما است قرار گرفته است. activation function تمامی لایه‌ها غیر از لایه آخر ReLU و لایه خروجی Softmax است. در نهایت از binary_crossentropy به عنوان loss و از Adam به عنوان optimizer استفاده شده است.

```python
def create_model(input_shape, num_classes):
    """
    Create CNN model based on the paper's architecture
    Args:
        input_shape: tuple of input image shape
        num_classes: number of classes
    Returns:
        model: compiled keras model
    """
    # Use Input layer as recommended
    inputs = tf.keras.Input(shape=input_shape)

    # First convolutional block
    x = Conv2D(16, (3, 3), activation='relu', padding='same')(inputs)
    x = MaxPooling2D(2, 2)(x)

    # Second convolutional block
    x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D(2, 2)(x)

    # Convolutional layer with padding='same'
    x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D(pool_size=(2, 2), padding='same')(x)

    # Fourth convolutional block
    x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D(2, 2)(x)

    # Flatten and dense layers
    x = Flatten()(x)
    x = Dense(64, activation='relu')(x)
    x = Dense(32, activation='relu')(x)
    outputs = Dense(num_classes, activation='softmax')(x)
    # Create model
    model = tf.keras.Model(inputs=inputs, outputs=outputs)
    # Compile model
    model.compile(
        optimizer='adam',
        loss='binary_crossentropy',
        metrics=['accuracy']
    )

    return model
```

شکل ۱۵- کد ساخت مدل مقاله

در ادامه برای تابع train از سه کلاس مهم برای callback استفاده شده است که هم روند کار را سرعت می‌بخشد هم نتایج را بهتر می‌کند. اولین کلاس ModelCheckpoint هست که با توجه به آموزش انجام شده بهترین مقدار را با توجه به val_accuracy در نهایت انتخاب می‌کند. کلاس دوم EarlyStopping هست که کار پایان سریع تر از موعد عملیات محاسبات را قبل از رسیدن به epoch پایانی را انجام می‌دهد. با استفاده از این کلاس سرعت کار افزایش میابد چرا که به تعداد patience تعریف شده صبر می‌کند و اگر مقدار val_accuracy افزایش نیابد کار را متوقف می‌کند. کلاس آخر ReduceLROnPlateau هست که کار تغییر کلی و از پایه learning rate است. در شرایطی که مدل در یک local optimum گیر کرده باشد و به تعداد گامی که در patience تعریف می‌کنیم صبر کرده و سپس با توجه به factor و min_lr مقدار

learning rate را تغییر داده و فرایند را با عدد جدید ادامه می‌دهد. در مقاله هم به استفاده از این روش اشاره شده است.

```python
def train_model(model, train_dataset, val_dataset, epochs=50):
    """
    Train the model using tf.data.Dataset
    """
    checkpoint = ModelCheckpoint(
        'best_model.keras',
        monitor='val_accuracy',
        save_best_only=True,
        mode='max',
        verbose=1
    )

    early_stopping = EarlyStopping(
        monitor='val_loss',
        patience=10,
        restore_best_weights=True,
        verbose=1
    )

    reduce_lr = ReduceLROnPlateau(
        monitor='val_loss',
        factor=0.2,
        patience=5,
        min_lr=1e-6,
        verbose=1
    )

    # Train model using datasets
    history = model.fit(
        train_dataset,
        validation_data=val_dataset,
        epochs=epochs,
        callbacks=[checkpoint, early_stopping, reduce_lr]
    )

    return history
```

شکل ۱۶- آموزش مدل

در مرحله بعد توصیه به استفاده از dataloader ها شده است. برای استفاده از Dataloader در کتابخانه Pytorch متدی با همین نام وجود دارد اما چون در حال استفاده از ternsorflow هستیم پیاده سازی آن در قالب یک تابع و باز هم با استفاده از متدهای موجود است اما نکته مهم در این مورد است که در صورت

استفاده از dataloader نحوه پیاده سازی augmentation اندکی متفاوت می‌شود و این موضوع به تفاوت اندک نتایج هم منجر خواهد شد. برنامه در حالت اجرای مدل مقاله با دیتالودر و در بخش بعد که مربوط به اجرای مدلی مشابه مدل مقاله اما با اندکی تغییر است بدون دیتالودر و با augmentation متفاوت اجرا شده است.

اما اهمیت استفاده از dataloader ها. در ادامه به شرح برخی از آنها می‌پردازیم:

۱ .مدیریت بهینه حافظه

- در روش معمولی، کل تصاویر به یکباره در حافظه RAM بارگذاری می‌شوند که برای مجموعه داده‌های بزرگ می‌تواند مشکل‌ساز باشد
- DataLoaderها داده‌ها را به صورت batch-by-batch بارگذاری می‌کنند
- در هر لحظه فقط بخشی از داده‌ها که برای آموزش نیاز است در حافظه قرار می‌گیرد
- این روش امکان کار با مجموعه داده‌های بسیار بزرگ را فراهم می‌کند

۲ .بهینه‌سازی عملکرد

- امکان load کردن موازی داده‌ها با استفاده از چندworker
- پیش‌بارگیری batch (prefetching) بعدی در حین پردازش batch فعلی
- استفاده از pin_memory برای انتقال سریع‌تر داده بهGPU
- کاهش قابل توجه زمان آموزش به دلیل حذف تاخیرهای مربوط به بارگذاری داده

۳ .مدیریت خودکار داده‌ها

- مدیریت خودکار shuffle کردن داده‌ها در هرepoch
- تقسیم‌بندی خودکار داده‌ها به batch های مناسب
- اعمال تبدیلات و افزایش داده (data augmentation) به صورتon-the-fly
- اطمینان از توزیع متوازن کلاس‌ها در هرbatch

۴ .قابلیت توسعه و انعطاف‌پذیری

- امکان تعریف transform های سفارشی برای پیش‌پردازش داده‌ها
- سهولت در اضافه کردن تکنیک‌های جدید افزایش داده

- قابلیت تغییر پارامترهای بارگذاری داده بدون نیاز به تغییر کد اصلی

۵ .بهبود کیفیت آموزش

- اعمال تصادفی تبدیلات در هر epoch باعث افزایش تنوع داده‌های آموزشی می‌شود
- کاهش احتمال overfitting با توجه به تنوع بیشتر داده‌ها
- امکان استفاده از تکنیک‌های پیشرفته sampling برای مدیریت imbalanced data

۶ .سازگاری با استانداردهای صنعتی

- DataLoaderها بخشی از استاندارد کتابخانه‌های یادگیری عمیق هستند
- سازگاری با فریم‌ورک‌های مختلف مثل PyTorch و TensorFlow
- تسهیل در به‌روزرسانی و نگهداری کد

۷ .مزایای پیاده‌سازی

- کد تمیزتر و ساختاریافته‌تر
- جداسازی منطقی بخش‌های مختلف کد(separation of concerns)
- قابلیت استفاده مجدد از کد برای پروژه‌های مختلف
- سهولت در اشکال‌زدایی و بهینه‌سازی

۸ .مدیریت بهتر منابع سیستم

- استفاده بهینه از CPU و GPU
- مدیریت خودکار حافظه
- امکان تنظیم پارامترهای بارگذاری براساس منابع سخت‌افزاری موجود

از معایب استفاده از dataloader هم می‌توان به موارد زیر پرداخت:

۱. مصرف حافظه:

- DataLoaderها برای پیش‌پردازش موازی و prefetching نیاز به حافظه اضافی دارند
- هر چه batch size و تعداد worker ها بیشتر باشد، مصرف حافظه افزایش می‌یابد
- در برخی موارد می‌تواند منجر به Out of Memory Error شود

۲. **پیچیدگی تنظیمات**:

- تنظیم بهینه پارامترهای DataLoader مثل:
  - تعدادworkers
  - اندازهbatch
  - اندازه buffer برایprefetching
- نیاز به تجربه و آزمون و خطا دارد
- تنظیمات نامناسب می‌تواند باعث کاهش کارایی شود

۳. محدودیت augmentation درDataLoader:

- وقتی از DataLoader استفاده می‌کنیم، به دلیل مکانیزمprefetching ، در هر batch فقط یک نوع augmentation می‌تواند روی هر تصویر اعمال شود. این به دلیل آن است که DataLoader از قبل داده‌ها را در حافظه پنهان (cache) نگه می‌دارد و نمی‌تواند چندین نسخه augment شده از یک تصویر را همزمان در cache داشته باشد.

چرا بدون DataLoader محدودیت در Augmentation نداریم؟

۱. در حالت بدونDataLoader ، کنترل کامل روی فرآیند augmentation داریم

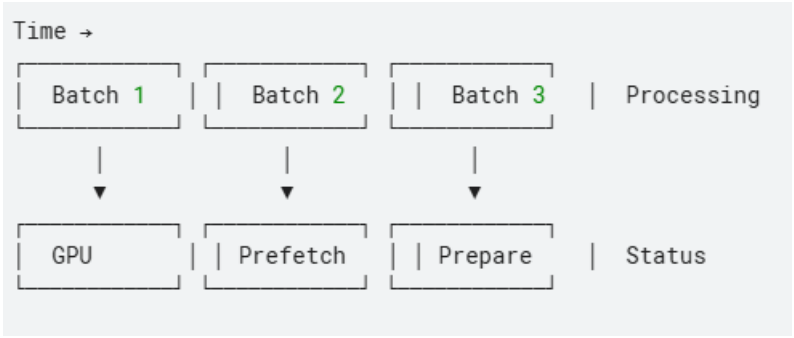
۲. می‌توانیم هر تعداد نسخه augment شده که بخواهیم از یک تصویر ایجاد کنیم

۳. محدودیت prefetching و cache نداریم

۴. می‌توانیم همه نسخه‌های augment شده را همزمان در حافظه داشته باشیم

۵. انعطاف‌پذیری بیشتری در ترکیب augmentation ها داریم.

نحوه پیاده سازی دیتالودر در مدل ما به زبان ساده بدین صورت است که در هنگام آموزش مدل به جای اینکه کل دیتاست به صورت یکجا خوانده شده و در اختیار مدل قرار بگیرد در هر epoch ابتدا به اندازه Batch size انتخاب شده عکس استخراج می‌شود و ما بقی عکس‌های مورد نیاز prefetch شده و آماده می‌شوند برای انتخاب و استفاده. به این صورت در استفاده از memory و GPU به شکل مشخصی صرفه جویی می‌شود.



شکل ۱۷ - شمایی از نحوه عملکرد **Dataloader**

اما تابعی که استفاده شده تا به این صورت گفته شده عملیات دیتالودر را انجام دهد به صورت زیر است.

```python
def create_dataloader(images, labels, batch_size=128, shuffle=True, augment=True):
    """
    Create TensorFlow dataset with your preprocessing pipeline
    Args:
        images: numpy array of images
        labels: numpy array of labels
        batch_size: batch size
        shuffle: whether to shuffle data
        augment: whether to apply augmentation
    Returns:
        tf.data.Dataset: configured dataset
    """
    # Convert to tensorflow dataset
    dataset = tf.data.Dataset.from_tensor_slices((images, labels))

    # Shuffle if requested
    if shuffle:
        dataset = dataset.shuffle(buffer_size=len(images))

    # Define preprocessing function
    def process_image(image, label):
        """Apply your preprocessing pipeline"""

        if augment:
            # Convert to float32 if needed
            image = tf.cast(image, tf.float32)
            # 1. Random horizontal flip
            if tf.random.uniform([]) > 0.5:
                image = tf.image.flip_left_right(image)
            # 2. Random width shift (20%)
            if tf.random.uniform([]) > 0.5:
                w_shift = tf.random.uniform([], -0.2, 0.2)
                image = tf.image.stateless_random_crop(
                    tf.pad(image, [[0, 0], [2, 2], [0, 0]], mode='SYMMETRIC'),
```

شکل ۱۸ - **Dataloader** - بخش اول

```
                size=tf.shape(image),
                seed=tf.random.uniform([2], 0, 1000, dtype=tf.int32)
            )
        # 3. Random height shift (20%)
        if tf.random.uniform([]) > 0.5:
            h_shift = tf.random.uniform([], -0.2, 0.2)
            image = tf.image.stateless_random_crop(
                tf.pad(image, [[2, 2], [0, 0], [0, 0]], mode='SYMMETRIC'),
                size=tf.shape(image),
                seed=tf.random.uniform([2], 0, 1000, dtype=tf.int32)
            )
        # 4. Random rotation (90 and -90 degrees)
        if tf.random.uniform([]) > 0.5:
            k = tf.random.uniform([], 0, 2, dtype=tf.int32) * 2 - 1  # Generate -1 or 1
            image = tf.image.rot90(image, k)  # Rotate by -90 (k=-1) or 90 (k=1)
        # Ensure output shape is correct and values are in valid range
        image = tf.clip_by_value(image, 0, 1)


    return image, label

# Apply preprocessing
dataset = dataset.map(
    process_image,
    num_parallel_calls=tf.data.AUTOTUNE
)

# Batch and prefetch
dataset = dataset.batch(batch_size)
dataset = dataset.prefetch(tf.data.AUTOTUNE)

return dataset
```

شکل ۱۹- **Dataloader** - بخش دوم

خروجی این تابع همان دیتاست مورد نظر ما است که باید به عنوان train dataset به مدل فرستاده شود.

برای بررسی بهتر عملکرد Dataloader تابع کلاسی به عنوان بررسی Performance هم نوشته شده است که میزان استفاده از Momory و همینطور زمان صرف شده برای هر بخش را گزارش می‌کند. از آنجایی که بررسی دقیق این پارامتر ها در صورت سوال خواسته نشده است از گزارش دقیق چگونگی عملکرد این بخش از کد صرف نظر کرده و تنها نتایج را گزارش می‌کنیم.

در نهایت برای اجرای مدل از ابتدا تا انتها به صورت زیر عمل می‌کنیم.

```python
# 1. Data Loading and EDA
with monitor.measure_time("Data Loading & EDA"):
    print("1. Loading and analyzing data...")
    images, labels, class_names = load_data(data_dir)
    print("\n2. Visualizing preprocessing steps on a sample image...")
    class_distribution, _, _ = perform_eda(images, labels, class_names)

# 2. Image Enhancement
print("\n3. Enhancing image quality...")
with monitor.measure_time("Image Enhancement"):
    enhanced_images = []
    for image in tqdm(images, desc="Enhancing images"):
        enhanced = enhance_image_quality(image)
        enhanced_images.append(enhanced)
    enhanced_images = np.array(enhanced_images)

# 3. Class Balancing
print("\n4. Handling class imbalance...")
with monitor.measure_time("Class Balancing"):
    balanced_images, balanced_labels = handle_class_imbalance(
        enhanced_images, labels, class_distribution
    )

# 4. Preprocessing
print("\n5. Preprocessing images...")
with monitor.measure_time("Preprocessing"):
    preprocessed_images = preprocess_images(balanced_images, target_size=img_size)
```

شکل ۲۰ - اجرای مدل بخش اول

در بخش اول ابتدا دیتا خوانده شده سپس عملیات های EDA و افزایش کیفیت عکس و بالانس کردن عکس و در نهایت resize و normalization انجام شده است.

```python
# 5. Data Splitting
print("\n6. Splitting data...")
with monitor.measure_time("Data Splitting"):
    train_images, temp_images, train_labels, temp_labels = train_test_split(
        preprocessed_images, balanced_labels,
        test_size=0.2, random_state=42,
        stratify=np.argmax(balanced_labels, axis=1)
    )

    val_images, test_images, val_labels, test_labels = train_test_split(
        temp_images, temp_labels,
        test_size=0.5, random_state=42,
        stratify=np.argmax(temp_labels, axis=1)
    )

# 6. Model Creation

print("\n7. Creating and compiling model...")
with monitor.measure_time("Model Creation"):
    model = create_model(input_shape=img_size + (3,), num_classes=len(class_names))
    print_simple_summary(model)
    visualize_model_architecture(model)
```

شکل ۲۱- اجرای مدل بخش دوم

در این بخش ابتدا ۸۰ درصد دیتا به بخش train و ۱۰ درصد به validation و ۱۰ درصد به test اختصاص داده شده است و در بخش بعدی مدل ایجاد شده است.

```python
# 7. Training
print("\n8. Creating data loaders...")
with monitor.measure_time("Model Training"):
    train_dataset = create_dataloader(
        train_images, train_labels,
        batch_size=batch_size,
        shuffle=True,
        augment=True
    )

    val_dataset = create_dataloader(
        val_images, val_labels,
        batch_size=batch_size,
        shuffle=False,
        augment=False
    )
    print("\n9. Training model...")
    history = train_model(model, train_dataset, val_dataset, epochs=epochs)

# Evaluate and visualize results
print("\n10. Evaluating model...")
evaluate_model(model, test_images, test_labels, class_names)

print("\n11. Visualizing predictions...")
visualize_predictions(model, test_images, test_labels, class_names)

print("\n12. Plotting training history...")
plot_training_history(history)
```

شکل ۲۲ - اجرای مدل بخش سوم

در بخش نهایی Dataloader ایجاد شده و بخش مربوط با آموزش و ارزیابی جدا شده است. نکته مهم در این بخش تصادفی بودن و دیتاافزایی در بخش آموزش و برعکس آن برای بخش validation است به این معنا که shuffle و augment برای این بخش خاموش است.

سپس مدل آموزش داده شده، ارزیابی و نمایش نتایج صورت گرفته است.

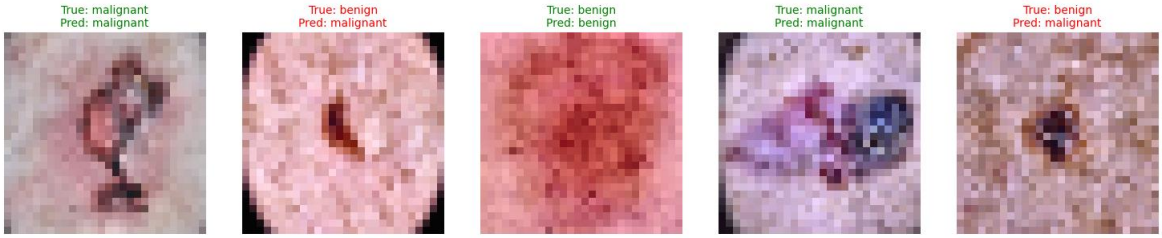
روند پیش روی مدل در چند ایپاک پایانی را در ادامه خواهیم دید اما در نهایت در بخش بعد به گزارش کامل و تحلیل نتایج بدست آمده از نتایج این مدل خواهیم پرداخت.

```
Epoch 39/50
19/23 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.8690 - loss: 0.2894
Epoch 39: val_accuracy did not improve from 0.85556
23/23 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8675 - loss: 0.2903 - val_accuracy: 0.8361 - val_loss: 0.3412 - learning_rate: 2.0000e-04
Epoch 40/50
19/23 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.8695 - loss: 0.2924
Epoch 40: val_accuracy did not improve from 0.85556
23/23 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.8685 - loss: 0.2928 - val_accuracy: 0.8306 - val_loss: 0.3322 - learning_rate: 2.0000e-04
Epoch 41/50
19/23 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.8607 - loss: 0.2955
Epoch 41: val_accuracy did not improve from 0.85556

Epoch 41: ReduceLROnPlateau reducing learning rate to 4.0000001899898055e-05.
23/23 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8619 - loss: 0.2947 - val_accuracy: 0.8444 - val_loss: 0.3367 - learning_rate: 2.0000e-04
Epoch 42/50
19/23 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.8662 - loss: 0.2994
Epoch 42: val_accuracy did not improve from 0.85556
23/23 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8661 - loss: 0.2982 - val_accuracy: 0.8361 - val_loss: 0.3330 - learning_rate: 4.0000e-05
Epoch 43/50
19/23 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.8639 - loss: 0.2832
Epoch 43: val_accuracy did not improve from 0.85556
23/23 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8637 - loss: 0.2847 - val_accuracy: 0.8444 - val_loss: 0.3277 - learning_rate: 4.0000e-05
Epoch 44/50
18/23 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.8682 - loss: 0.2849
Epoch 44: val_accuracy did not improve from 0.85556
23/23 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8677 - loss: 0.2861 - val_accuracy: 0.8417 - val_loss: 0.3262 - learning_rate: 4.0000e-05
Epoch 45/50
19/23 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.8671 - loss: 0.2889
Epoch 45: val_accuracy did not improve from 0.85556
23/23 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.8670 - loss: 0.2886 - val_accuracy: 0.8389 - val_loss: 0.3347 - learning_rate: 4.0000e-05
Epoch 46/50
19/23 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.8709 - loss: 0.2856
Epoch 46: val_accuracy did not improve from 0.85556

Epoch 46: ReduceLROnPlateau reducing learning rate to 8.000000525498762e-06.
23/23 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.8703 - loss: 0.2862 - val_accuracy: 0.8444 - val_loss: 0.3264 - learning_rate: 4.0000e-05
Epoch 46: early stopping
Restoring model weights from the end of the best epoch: 36.
```

شکل ۲۳ - بخشی از نتایج اجرای مدل مقاله

```
Detailed Metrics:

Class: benign
Accuracy: 0.8361
Precision: 0.8854
Recall: 0.7722
F1-score: 0.8249
True Positives: 139
False Positives: 18
True Negatives: 162
False Negatives: 41

Class: malignant
Accuracy: 0.8361
Precision: 0.7980
Recall: 0.9000
F1-score: 0.8460
True Positives: 162
False Positives: 41
True Negatives: 139
False Negatives: 18
```

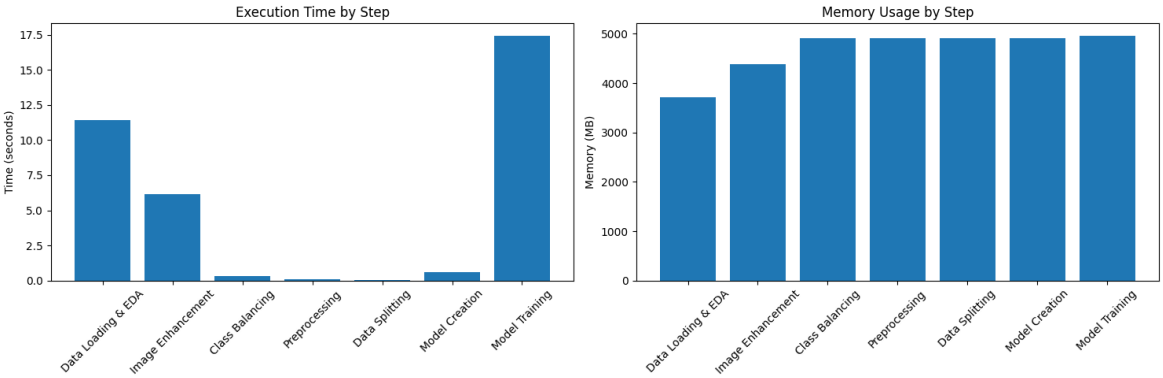
شکل ۲۴ - **Detail Metrics** مدل مقاله



شکل ۲۵ - نمونه ای از تصاویر پیشبینی شده توسط مدل مقاله

```
Performance Metrics:
--------------------------------------------------

Data Loading & EDA:
Time: 11.44 seconds
Memory: 3711.80 MB

Image Enhancement:
Time: 6.14 seconds
Memory: 4387.82 MB

Class Balancing:
Time: 0.32 seconds
Memory: 4904.66 MB

Preprocessing:
Time: 0.10 seconds
Memory: 4904.66 MB

Data Splitting:
Time: 0.02 seconds
Memory: 4904.66 MB

Model Creation:
Time: 0.57 seconds
Memory: 4904.66 MB

Model Training:
Time: 17.44 seconds
Memory: 4961.33 MB

Total Pipeline Time: 36.02 seconds
```

شکل ۲۶ -بررسی **Performance** در زمان استفاده از **Dataloader**



شکل ۲۷ - بررسی **Performance** در زمان استفاده از **Dataloader** به صورت جدول

در گام بعد مدل دیگری با ساختاری مشابه مدل مقاله اما کمی متفاوت ایجاد کرده‌ایم. در این مدل تعداد لایه‌ها و عملیات maxpooling دقیقا مشابه قبل است اما تفاوت‌های اصلی ایجاد شده از قرار زیر است:

۱) **استفاده از Nadam به جای Adam**: این گزینه تست شد و نتیجه بهتر بود. از دلایل احتمالی بهتر عملکردن این بهینه ساز نسبت به Adam می‌توان به مواردی مانند موارد ذیل اشاره کرد: متفاوت بودن عملیات momentum و بهتر عملکردن آن با توجه به تابع loss تعریف شده جدید(binary)، سریع تر همگرا شدن و مواردی از این دست اشاره کرد.

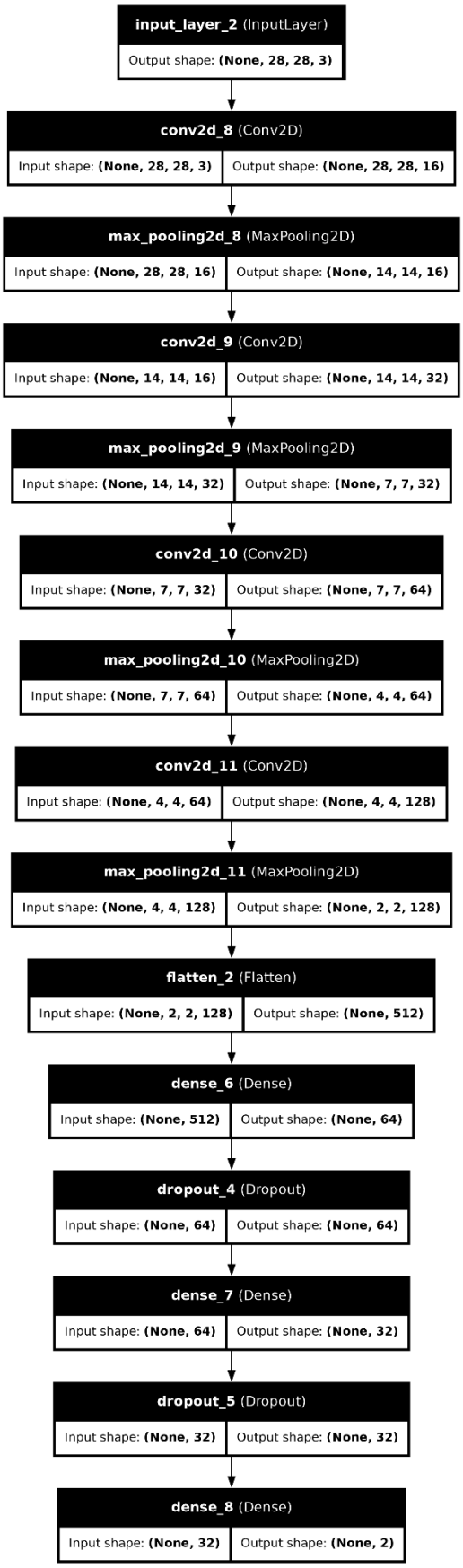
۲) **استفاده از لایه Dropout**: در بخش fully connected دو لایه dropout با نرخ به ترتیب ۲۵ و ۵۰ درصد ایجاد شده است. دلیل استفاده از این لایه ایجاد یک regularization خوب و عدم اتکا به نورون‌های خاص در مدل است. با استفاده از این لایه می‌توان به قابلیت تعمیم بهتری رسید و عملا مدل را از خطر overfit شدن دور نگه داشت.

۳) **عدم استفاده از Dataloader و استفاده از روش اول Augmentation برای آموزش**: اگر چه انجام این کار سرعت اجرای مدل را به وضوح پایین می‌اورد اما از انجایی که با استفاده از روش اول می‌توانیم به شکل بهتر و منعطف تری

عملیات داده افزایی را انجام دهیم (مانند دوران با زاویه‌های کمتر و به صورت smooth تر) به سراغ این روش رفته ایم.

Layer (type)	Output Shape	Param #
input_layer_2 (InputLayer)	(None, 28, 28, 3)	0
conv2d_8 (Conv2D)	(None, 28, 28, 16)	448
max_pooling2d_8 (MaxPooling2D)	(None, 14, 14, 16)	0
conv2d_9 (Conv2D)	(None, 14, 14, 32)	4,640
max_pooling2d_9 (MaxPooling2D)	(None, 7, 7, 32)	0
conv2d_10 (Conv2D)	(None, 7, 7, 64)	18,496
max_pooling2d_10 (MaxPooling2D)	(None, 4, 4, 64)	0
conv2d_11 (Conv2D)	(None, 4, 4, 128)	73,856
max_pooling2d_11 (MaxPooling2D)	(None, 2, 2, 128)	0
flatten_2 (Flatten)	(None, 512)	0
dense_6 (Dense)	(None, 64)	32,832
dropout_4 (Dropout)	(None, 64)	0
dense_7 (Dense)	(None, 32)	2,080
dropout_5 (Dropout)	(None, 32)	0
dense_8 (Dense)	(None, 2)	66

شکل ۲۸ - ساختار مورد استفاده قرار گرفته در مدل بهبود یافته



شکل ۲۹ - ساختار لایه های بکار رفته در کد برای مدل بهبود یافته

در نهایت کد نوشته شده برای ایجاد مدل به صورت زیر است که با ساختار گفته شده و مشابه مدل قبل پیاده سازی شده است.

```python
def create_model(input_shape, num_classes):
    """
    Create CNN model based on the paper's architecture
    Args:
        input_shape: tuple of input image shape
        num_classes: number of classes
    Returns:
        model: compiled keras model
    """
    # Use Input layer as recommended
    inputs = tf.keras.Input(shape=input_shape)

    # First convolutional block
    x = Conv2D(16, (3, 3), activation='relu', padding='same')(inputs)
    x = MaxPooling2D(2, 2)(x)

    # Second convolutional block
    x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D(2, 2)(x)

    # Convolutional layer with padding='same'
    x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D(pool_size=(2, 2), padding='same')(x)

    # Fourth convolutional block
    x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D(2, 2)(x)

    # Flatten and dense layers
    x = Flatten()(x)
    x = Dense(64, activation='relu')(x)
    x = Dropout(0.25)(x)
    x = Dense(32, activation='relu')(x)
    x = Dropout(0.5)(x)
    outputs = Dense(num_classes, activation='softmax')(x)
    # Create model
    model = tf.keras.Model(inputs=inputs, outputs=outputs)
    # Compile model
    model.compile(
        optimizer='nadam',
        loss='binary_crossentropy',
        metrics=['accuracy']
    )
    return model
```

شکل ۳۰ - ساختار مدل بهبود یافته

و اما نتایج در چند epoch پایانی را در ادامه خواهیم دید. لازم به ذکر است تمام نتایج اعم از نمودارها و مقادیر دقت در بخش بعدی به تفصل امده و توضیح داده شده اند.

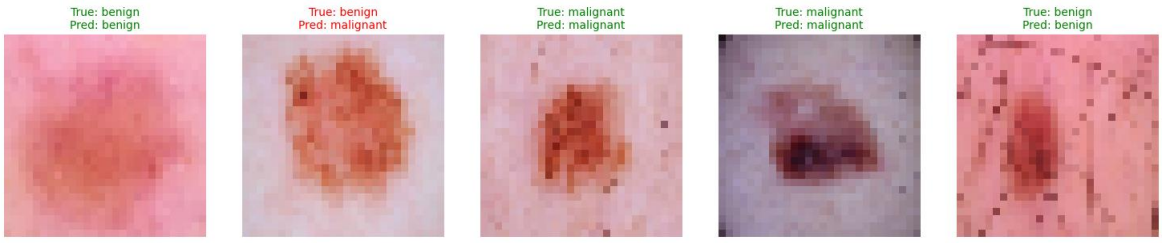
```
Epoch 29: val_accuracy did not improve from 0.84545
21/21 ━━━━━━━━━━━━━━━━━━━━ 1s 43ms/step - accuracy: 0.8379 - loss: 0.3409 - val_accuracy: 0.8455 - val_loss: 0.3290 - learning_rate: 0.0010
Epoch 30/50
14/21 ━━━━━━━━━━━━━━━━━━━━ 0s 61ms/step - accuracy: 0.8341 - loss: 0.3399
Epoch 30: val_accuracy did not improve from 0.84545
21/21 ━━━━━━━━━━━━━━━━━━━━ 1s 43ms/step - accuracy: 0.8360 - loss: 0.3388 - val_accuracy: 0.8333 - val_loss: 0.3795 - learning_rate: 0.0010
Epoch 31/50
14/21 ━━━━━━━━━━━━━━━━━━━━ 0s 66ms/step - accuracy: 0.8365 - loss: 0.3436
Epoch 31: val_accuracy did not improve from 0.84545
21/21 ━━━━━━━━━━━━━━━━━━━━ 1s 46ms/step - accuracy: 0.8370 - loss: 0.3388 - val_accuracy: 0.8333 - val_loss: 0.3754 - learning_rate: 0.0010
Epoch 32/50
14/21 ━━━━━━━━━━━━━━━━━━━━ 0s 64ms/step - accuracy: 0.8402 - loss: 0.3478
Epoch 32: val_accuracy did not improve from 0.84545
21/21 ━━━━━━━━━━━━━━━━━━━━ 1s 45ms/step - accuracy: 0.8399 - loss: 0.3489 - val_accuracy: 0.8242 - val_loss: 0.3799 - learning_rate: 0.0010
Epoch 33/50
14/21 ━━━━━━━━━━━━━━━━━━━━ 0s 65ms/step - accuracy: 0.8341 - loss: 0.3654
Epoch 33: val_accuracy did not improve from 0.84545
21/21 ━━━━━━━━━━━━━━━━━━━━ 1s 46ms/step - accuracy: 0.8389 - loss: 0.3563 - val_accuracy: 0.8394 - val_loss: 0.3584 - learning_rate: 0.0010
Epoch 34/50
14/21 ━━━━━━━━━━━━━━━━━━━━ 0s 62ms/step - accuracy: 0.8447 - loss: 0.3104
Epoch 34: val_accuracy did not improve from 0.84545

Epoch 34: ReduceLROnPlateau reducing learning rate to 0.00020000000949949026.
21/21 ━━━━━━━━━━━━━━━━━━━━ 1s 44ms/step - accuracy: 0.8436 - loss: 0.3175 - val_accuracy: 0.8424 - val_loss: 0.3460 - learning_rate: 0.0010
Epoch 35/50
14/21 ━━━━━━━━━━━━━━━━━━━━ 0s 64ms/step - accuracy: 0.8464 - loss: 0.3078
Epoch 35: val_accuracy did not improve from 0.84545
21/21 ━━━━━━━━━━━━━━━━━━━━ 1s 45ms/step - accuracy: 0.8488 - loss: 0.3085 - val_accuracy: 0.8303 - val_loss: 0.3615 - learning_rate: 2.0000e-04
Epoch 36/50
14/21 ━━━━━━━━━━━━━━━━━━━━ 0s 64ms/step - accuracy: 0.8556 - loss: 0.3153
Epoch 36: val_accuracy did not improve from 0.84545
21/21 ━━━━━━━━━━━━━━━━━━━━ 1s 45ms/step - accuracy: 0.8557 - loss: 0.3128 - val_accuracy: 0.8333 - val_loss: 0.3449 - learning_rate: 2.0000e-04
Epoch 37/50
14/21 ━━━━━━━━━━━━━━━━━━━━ 0s 62ms/step - accuracy: 0.8598 - loss: 0.3000
Epoch 37: val_accuracy did not improve from 0.84545
21/21 ━━━━━━━━━━━━━━━━━━━━ 1s 44ms/step - accuracy: 0.8582 - loss: 0.3019 - val_accuracy: 0.8364 - val_loss: 0.3384 - learning_rate: 2.0000e-04
Epoch 38/50
14/21 ━━━━━━━━━━━━━━━━━━━━ 0s 64ms/step - accuracy: 0.8550 - loss: 0.3017
Epoch 38: val_accuracy did not improve from 0.84545
21/21 ━━━━━━━━━━━━━━━━━━━━ 1s 46ms/step - accuracy: 0.8548 - loss: 0.3033 - val_accuracy: 0.8333 - val_loss: 0.3679 - learning_rate: 2.0000e-04
Epoch 39/50
14/21 ━━━━━━━━━━━━━━━━━━━━ 0s 65ms/step - accuracy: 0.8558 - loss: 0.2997
Epoch 39: val_accuracy did not improve from 0.84545

Epoch 39: ReduceLROnPlateau reducing learning rate to 4.0000001899898055e-05.
21/21 ━━━━━━━━━━━━━━━━━━━━ 1s 46ms/step - accuracy: 0.8567 - loss: 0.3003 - val_accuracy: 0.8364 - val_loss: 0.3672 - learning_rate: 2.0000e-04
Epoch 39: early stopping
Restoring model weights from the end of the best epoch: 29.
```

شکل ۳۱ بخشی از نتایج اجرای مدل بهبود یافته

در هر دو مدل سیر نتایج درست و منطقی است و ما کاهش loss و افزایش accuracy را در روند کلی مشاهده می‌کنیم.

```
Detailed Metrics:

Class: benign
Accuracy: 0.8667
Precision: 0.9250
Recall: 0.8222
F1-score: 0.8706
True Positives: 148
False Positives: 12
True Negatives: 138
False Negatives: 32

Class: malignant
Accuracy: 0.8667
Precision: 0.8118
Recall: 0.9200
F1-score: 0.8625
True Positives: 138
False Positives: 32
True Negatives: 148
False Negatives: 12
```

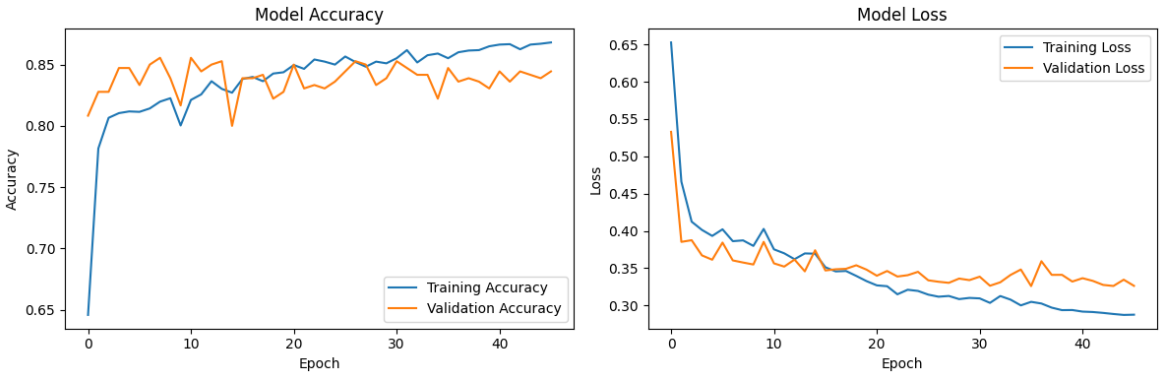
شکل ۳۲- **Detail Metrics** برای مدل بهبود یافته



شکل ۳۳ - نمونه ای از تصاویر پیشبینی شده با مدل بهبود یافته

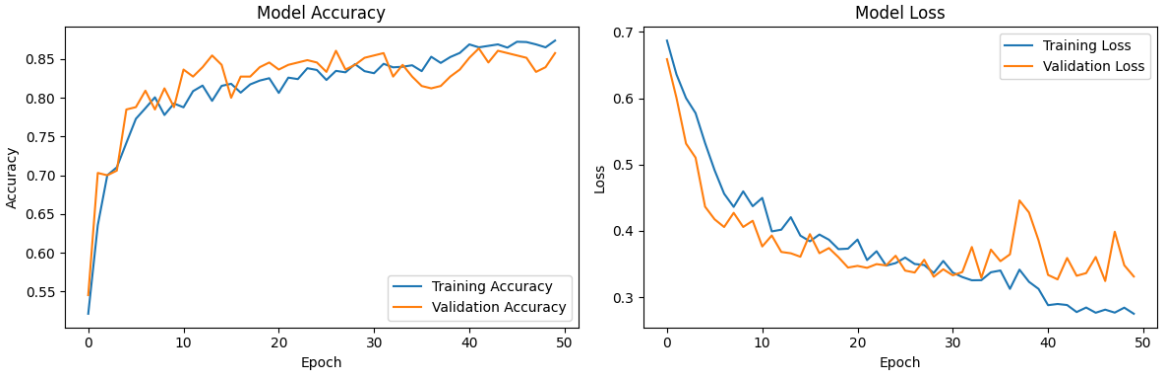
۴-۱. تحلیل نتایج

در این قسمت ابتدا نمودارهای loss و accuracy برای هر دو مدل را به نمایش می‌گذاریم. ساختار این نمودارها روند کاهش یا افزایش را در هر epoch به ما نشان می‌دهند. این نمودارها برای train و validation ترسیم می‌شود و در نهایت مدل با test ارزیابی نهایی شده و دقت نهایی گزارش می‌شود.



شکل ۳۴ - نمودار **loss** و **accuracy** مدل مقاله

همان طور که در نمودار ها مشخص است روند به درستی طی شده است. loss کاهش یافته و accuracy افزایش یافته است. تفاوتی که در بخش آموزش و ارزیابی مشاهده می‌شود نسبتا کم و قابل قبول است و از قابلیت تعمیم پذیری نسبتا خوبی برخوردار است. همچنین چون دقت به مقدار خوب و قابل قبولی رسیده نگران underfit شدن هم نخواهیم بود.



شکل ۳۵ - نمودار **loss** و **accuracy** مدل بهبود یافته

نمودار مدل بهبود یافته نشان دهنده روند بسیار درست مشابه مدل قبل است که نشان از عملکرد مناسب مدل می‌دهد. مقایسه این نمودارها در بخش بعد آمده است.

نمودار **ROC (Receiver Operating Characteristic)** یکی از ابزارهای گرافیکی برای ارزیابی عملکرد مدل‌های دسته‌بندی دوتایی است. این نمودار نشان‌دهنده‌ی رابطه بین نرخ مثبت درست **(True Positive Rate - TPR)** و نرخ مثبت کاذب **(False Positive Rate - FPR)** در تمام آستانه‌های ممکن **(Thresholds)** است.

اجزای اصلی نمودار:ROC

۱. **نرخ مثبت درست :(True Positive Rate - TPR)** این مقدار که به آن **حساسیت** یا **Recall** نیز گفته می‌شود، نشان‌دهنده درصد نمونه‌های مثبت واقعی است که به درستی توسط مدل شناسایی شده‌اند:

$$TPR = \frac{True\ Positives(TP)}{\left(True\ Positives\ (TP) + \ False\ Negatives\ (FN)\right)}$$

نرخ مثبت کاذب :(False Positive Rate - FPR) این مقدار نشان‌دهنده درصد نمونه‌های منفی واقعی است که به اشتباه توسط مدل به عنوان مثبت دسته‌بندی شده‌اند:

$$FPR = \frac{False\ Positives(FP)}{\left(\ False\ Positives(FP) + \ True\ Negatives\ (TN)\right)}$$

۲. **آستانه :(Threshold)** مدل‌های دسته‌بندی معمولاً خروجی خود را به صورت احتمالات ارائه می‌دهند. با تغییر آستانه تصمیم‌گیری (مثلاً از ۰.۱ به ۰.۵ یا ۰.۹)، مقادیر مختلفی از TPR و FPRبه دست می‌آید. نمودار ROC از رسم این مقادیر در تمام آستانه‌ها تشکیل می‌شود.

ساختار نمودار:ROC

- **محور افقی :(X)** نرخ مثبت کاذب (FPR) در بازه.[0, 1]
- **محور عمودی :(Y)** نرخ مثبت درست (TPR) در بازه.[0, 1]

ویژگی‌های نمودار:

- **خط مورب :**خط مورب از نقطه (۰, ۰) تا (۱, ۱) نشان‌دهنده یک مدل تصادفی است که توانایی تشخیص ندارد.
- **گوشه بالا-چپ :**نقطه ایده‌آل در (۰, ۱) است که نشان‌دهنده ۰ مثبت کاذب و ٪۱۰۰ مثبت درست است.
- **شکل نمودار :**هرچه نمودار به گوشه بالا-چپ نزدیک‌تر باشد، عملکرد مدل بهتر است.

مساحت زیر نمودار :(AUC - Area Under Curve)

- **AUC**معیاری برای ارزیابی توانایی مدل در تفکیک بین دو کلاس است:
  - **AUC = 1.0:**مدل کاملاً دقیق است.
  - **AUC = 0.5:**مدل کاملاً تصادفی است.
  - **AUC < 0.5:**مدل عملکردی ضعیف‌تر از تصادف دارد.

نحوه تفسیر نمودار:ROC

۱. **TPRبالا و FPR پایین :**نشان‌دهنده عملکرد خوب مدل است.

۲. **نمودار نزدیک به خط مورب :(AUC ≈ 0.5)** نشان‌دهنده عملکرد ضعیف مدل است.

۳. **مقایسه مدل‌ها :**مدلی که AUC بالاتری دارد، معمولاً عملکرد بهتری دارد.

کاربردهای عملی:

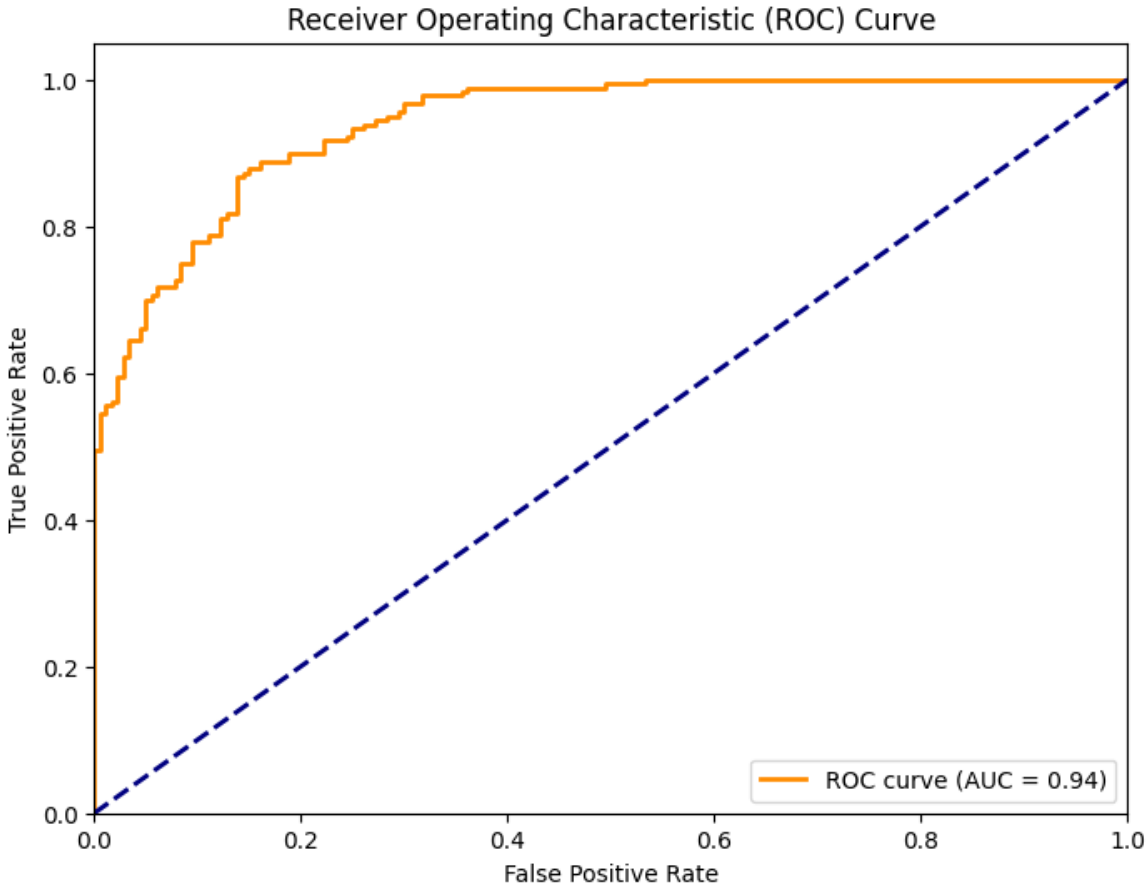
۱. **داده‌های نامتوازن :**نمودار ROC زمانی که توزیع کلاس‌ها نامتوازن باشد بسیار مفید است، زیرا به بررسی تعادل بین شناسایی درست کلاس مثبت و کاهش مثبت کاذب می‌پردازد.

۲. **تنظیم آستانه :**نمودار ROC به شناسایی بهترین آستانه برای دسته‌بندی کمک می‌کند.

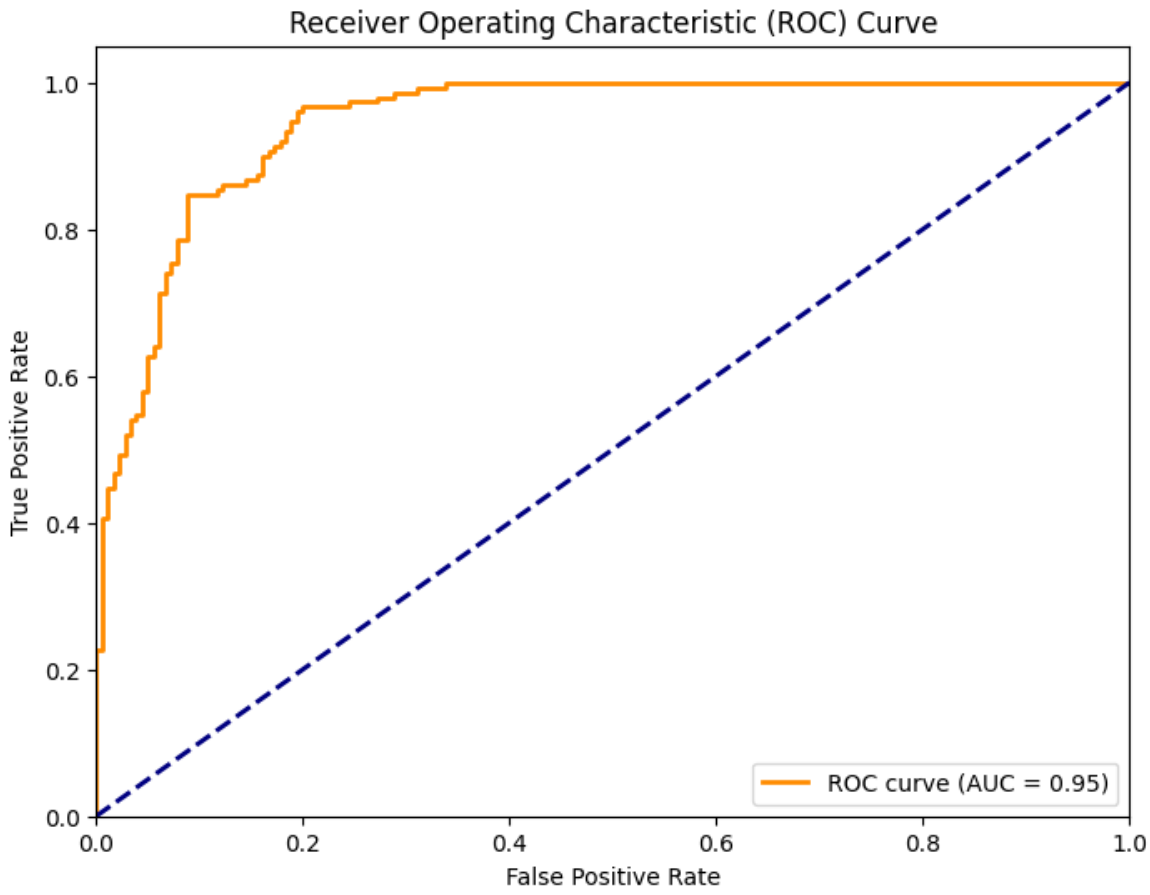
محدودیت‌ها:

۱. **حساسیت به عدم تعادل داده‌ها :**در شرایطی که کلاس منفی غالب باشد، حتی یک مدل با عملکرد ضعیف برای کلاس مثبت ممکن است AUC بالایی داشته باشد.

۲. **انتخاب آستانه :**نمودار ROC تنها عملکرد مدل را در آستانه‌های مختلف نشان می‌دهد و پیشنهاد مستقیمی برای انتخاب بهترین آستانه ارائه نمی‌دهد.



شکل ۳۶ - نمودار **ROC** مدل مقاله



شکل ۳۷ - نمودار **ROC** مدل بهبود یافته

در بخش بعد به مقایسه این دو نمودار پرداخته شده است.

ماتریس آشفتگی (Confusion Matrix) یکی از ابزارهای اصلی برای ارزیابی کیفیت دسته‌بندی در مدل‌های یادگیری ماشین است. این ماتریس، تعداد پیش‌بینی‌های درست و غلط مدل را در هر کلاس نمایش می‌دهد. با استفاده از ماتریس آشفتگی، معیارهایی مانند **Precision**، **Recall**، **Accuracy** و **F1-score** محاسبه می‌شوند که در ادامه توضیح داده شده‌اند:

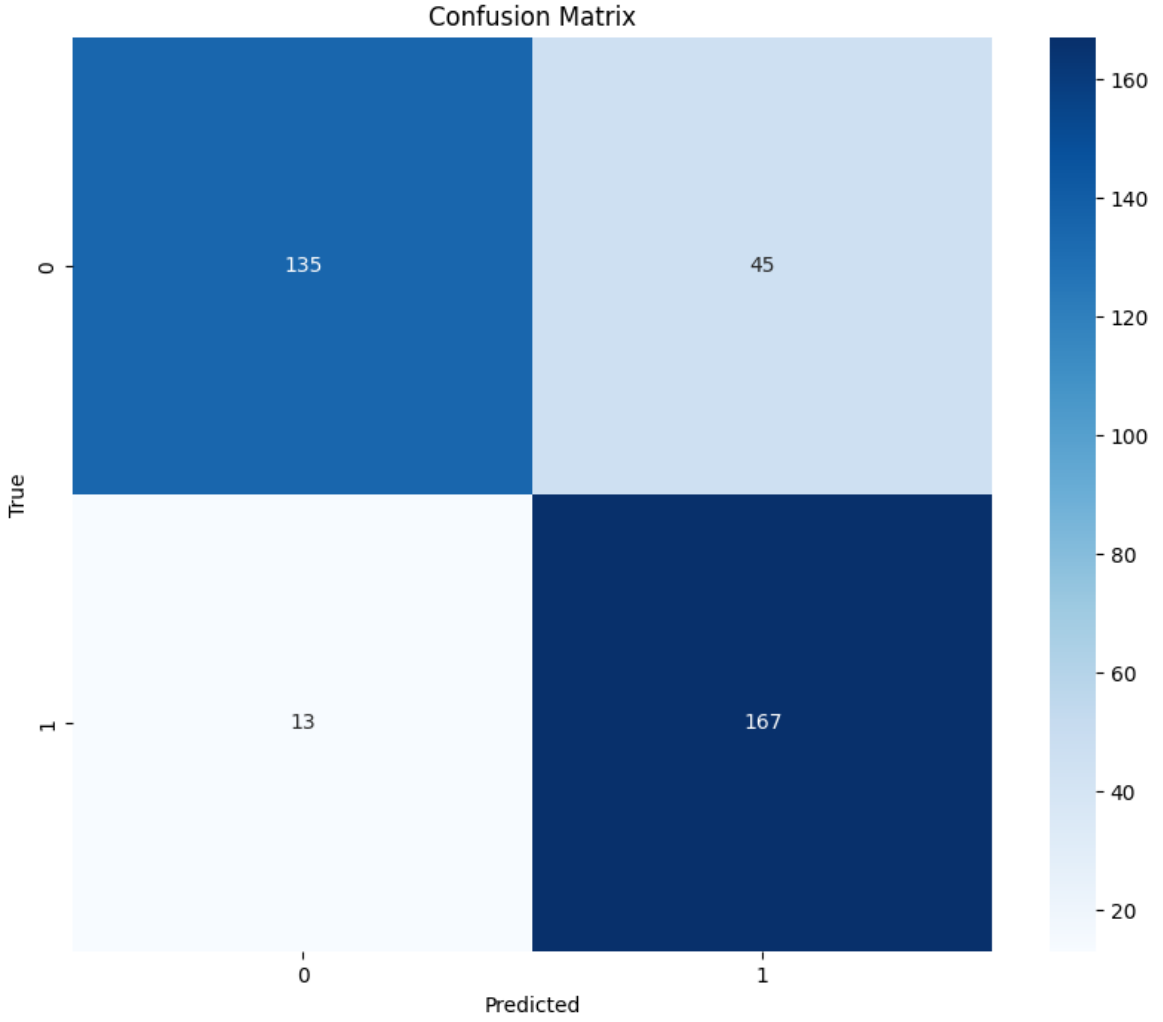
۱ .اجزای ماتریس آشفتگی:

ماتریس آشفتگی برای یک مسئله دسته‌بندی دوتایی (مانند نمونه شما) شامل چهار مقدار است:

- **True Positives (TP):** تعداد نمونه‌های مثبت که به درستی مثبت پیش‌بینی شده‌اند.
- **False Positives (FP):** تعداد نمونه‌های منفی که به اشتباه مثبت پیش‌بینی شده‌اند.
- **True Negatives (TN):** تعداد نمونه‌های منفی که به درستی منفی پیش‌بینی شده‌اند.

- **False Negatives (FN):** تعداد نمونه‌های مثبت که به اشتباه منفی پیش‌بینی شده‌اند

نکته بسیار مهم در ترسیم ماتریس آشفتگی این است که این ماتریس باید حتما با دیتای test ترسیم شود که ما هم همینکار را انجام داده ایم. به این صورت که بعد از آموزش مدل تابعی که برای evaluate کردن مدل نوشته شده است را فراخوانی کرده و با توجه به پیشبینی مدل برای این عکس های Test و لیبل واقعی آنها این ماتریس ترسیم شده است.



شکل ۳۸ - ماتریس آشفتگی مدل مقاله

به عنوان مثال در تصویر فوق:

- **FN = 13 ،TN = 135 ،FP = 45 ،TP = 167**

۲ .معیارهای ارزیابی مدل:

الف) دقت کلی (Accuracy):

Accuracyمیزان پیش‌بینی‌های درست مدل را در کل داده نشان می‌دهد:

$$\textbf{Accuracy} = (TP + TN) / (TP + TN + FP + FN)$$

- **توضیح :**این معیار زمانی مناسب است که توزیع داده متوازن باشد. اگر تعداد نمونه‌های یکی از کلاس‌ها بسیار بیشتر از دیگری باشد، Accuracyمعیار دقیقی نیست.

ب) دقت (Precision):

Precisionنسبت نمونه‌های مثبت درست پیش‌بینی‌شده به کل نمونه‌هایی است که به عنوان مثبت پیش‌بینی شده‌اند:

$$\textbf{Precision} = TP / (TP + FP)$$

- **توضیح** Precision **:**نشان می‌دهد چه درصد از پیش‌بینی‌های مثبت مدل واقعاً مثبت بوده‌اند. این معیار زمانی اهمیت دارد که هزینه مثبت کاذب (FP) بالا باشد، مثلاً در یک سیستم اسپم ایمیل.

ج) حساسیت یا بازیابی (Recall):

Recallنسبت نمونه‌های مثبت درست پیش‌بینی‌شده به کل نمونه‌های مثبت واقعی است:

$$\textbf{Recall} = TP / (TP + FN)$$

- **توضیح :**این معیار زمانی اهمیت دارد که هزینه از دست دادن مثبت‌های واقعی (FN) بالا باشد، مثلاً در شناسایی بیماران در پزشکی.

د) F1-Score:

F1-Scoreمیانگین هارمونیک Precision و Recall است و تعادلی بین این دو معیار ارائه می‌دهد:

$$\textbf{F1-Score} = 2 * (Precision * Recall) / (Precision + Recall)$$

- **توضیح :**این معیار زمانی مفید است که توزیع داده نامتوازن باشد، زیرا به صورت یکپارچه به دقت و حساسیت توجه می‌کند.

۳ .تحلیل عملکرد مدل:

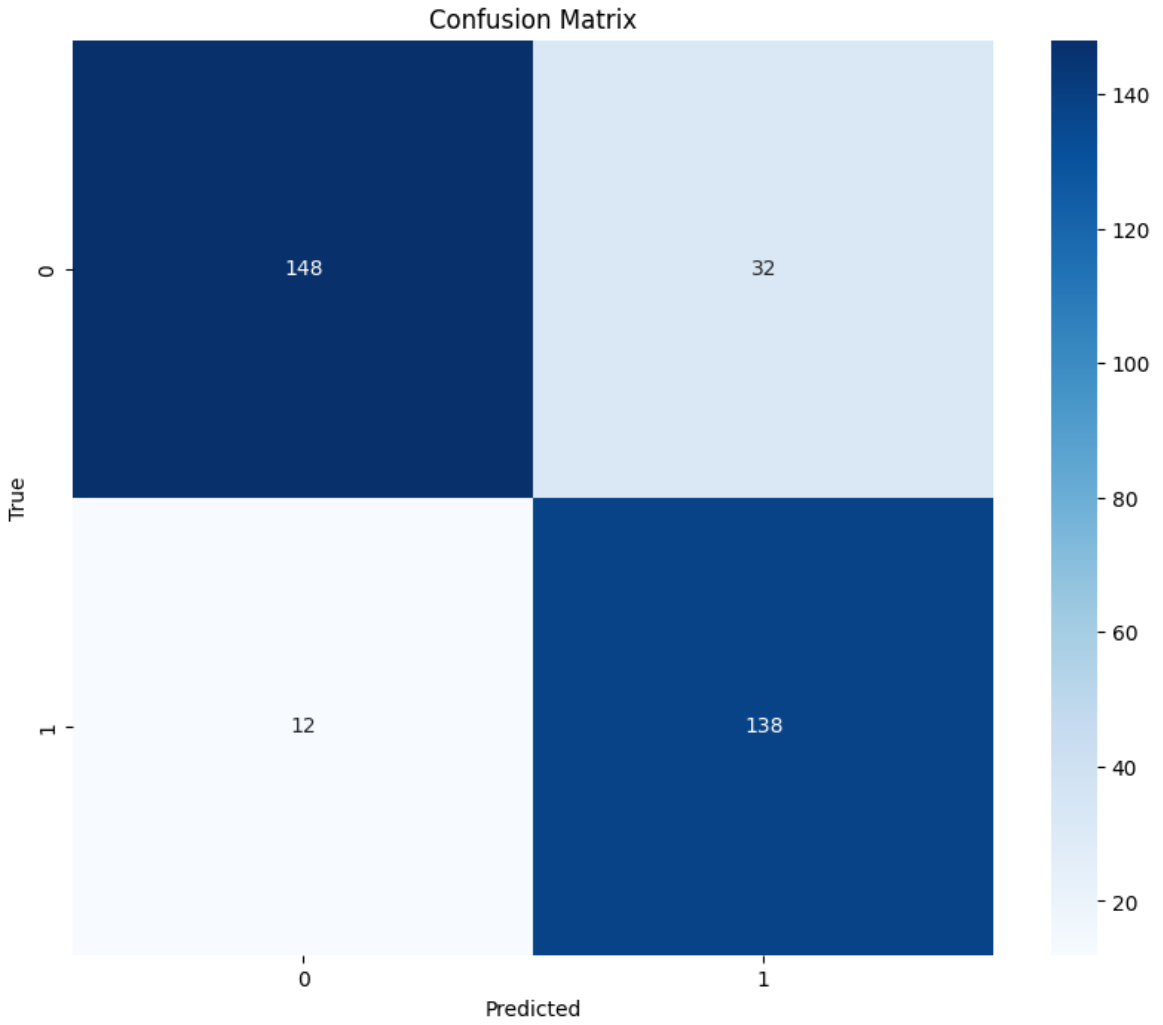
برای تحلیل کیفیت مدل:

۱. **Accuracy:** برای بررسی کلی عملکرد مدل استفاده می‌شود.

۲. **Precision:** اگر مثبت‌های کاذب (FP) مشکل‌ساز باشند، مانند اسپم یا فیلترهای امنیتی، Precision مهم‌تر است.

۳. **Recall:** اگر از دست دادن مثبت‌های واقعی (FN) پرهزینه باشد، مانند شناسایی بیماری، Recall اهمیت بیشتری دارد.

۴. **F1-Score:** برای تعادل بین Precision و Recall استفاده می‌شود و معیار مناسبی برای داده‌های نامتوازن است.



شکل ۳۹ - ماتریس آشفتگی مدل بهبود یافته

جدول ۱- بررسی معیارهای مختلف دقت مدل مقاله

precision	recall	F1- Score	Support	
۰.۹۱	۰.۷۵	۰.۸۲	۱۸۰	**Benign**
۰.۷۹	۰.۹۳	۰.۸۵	۱۸۰	**Malignant**
		۰.۸۴	۳۶۰	**Accuracy**
۰.۸۵	۰.۸۴	۰.۸۴	۳۶۰	**Macro avg**

Weighted avg	۳۶۰	۰.۸۴	۰.۸۴	۰.۸۵

جدول ۲ - بررسی معیارهای مختلف دقت مدل بهبود یافته

	Support	F1- Score	recall	precision
Benign	۱۸۰	۰.۸۷	۰.۸۲	۰.۹۳
Malignant	۱۸۰	۰.۸۶	۰.۹۲	۰.۸۱
Accuracy	۳۶۰	۰.۸۷		
Macro avg	۳۶۰	۰.۸۷	۰.۸۷	۰.۸۷
Weighted avg	۳۶۰	۰.۸۷	۰.۸۷	۰.۸۷

قسمت Macro و Weighted هم میانگین هر کدام از معیارهای precision , recall , F1-Score هست که به ترتیب میانگین ساده و میانگین وزنی را محاسبه کرده است.

۱-۵. مقایسه نتایج

از مقایسه نمودار loss و accuracy دو مدل مقاله و مدل بهبود یافته می‌توانیم ببینیم که مدل بهبود یافته ابتدا accuracy کمتری داشته اما در نهایت رشد آن بسیار مشابه train بوده و به دقتی بالاتر از دقت مقاله رسیده است. همچنین برای قسمت loss هم این مورد مشخص است. اما نکته دیگر نزدیک تر بودن نمودار داده‌های validation به train در مدل بهبود یافته نسبت به مدل مقاله است. این مورد بخصوص در accuracy به وضوح قابل مشاهده است و عملا هر این دو نمودار به هم نزدیک تر باشند نشان از تعمیم پذیری بهتر مدل می‌دهد و مدل بهتر fit شده و خطر overfit شدن کمتر بنظر می‌رسد. از این جهت عملکرد بهتر مدل بهبود یافته را مشاهده می‌کنیم. دلیل اصلی را می‌توان استفاده از dropout در این مورد دانست که به تعمیم پذیری مدل کمک شایانی می‌کند. اما در رابطه با روند بهتر و مشابه training می‌توان اثر استفاده از optimizer متفاوت Nadam را هم در نظر گرفت چرا که نحوه تغییر و adapt کردن learning rate را به شکل متفاوتی انجام می‌دهد.

جدول ۳- مقایسه مشخصات مدل مقاله و مدل بهبود یافته

	Optimizer	Augmentation	Dropout
مدل مقاله	Adam	داخل dataloader	ندارد
مدل بهبود یافته	Nadam	به صورت مجزا	دارد

جدول ۴ - مقایسه نمودار **loss** و **Accuracy**

	روند loss	روند دقت	تعمیم پذیری
مدل مقاله	نزولی	صعودی	خوب
مدل بهبود یافته	نزولی	صعودی	عالی

مورد بعدی برای مقایسه نمودار ROC دو مدل مقاله و مدل بهبود یافته است. از تصاویری که در بخش تحلیل برای این نمودار برای هر دو مدل آمده مشخص است که در نمودار مدل بهبود یافته مساحت زیر نمودار یا همان معیار AUC رشد به میزان یک درصد افزایش یافته است. همچنین نمودار مدل بهبود یافته اندکی به سمت چپ و بالا مایل تر است. این دو مقدار اگرچه زیاد نیست اما نشان از عملکرد بهتر مدل در این معیار می‌دهد.

جدول ۵ - مقایسه **AUC**

	AUC
مدل مقاله	۰.۹۴
مدل بهبود یافته	۰.۹۵

معیار بعدی اما مقایسه دقت دو مدل است که در ۴ عنوان متفاوت Accuracy و Precision و Recall و F1-Score تعریف می‌شود. برای مقایسه هر چه بهتر این دو مدل جدول زیر تهیه شده است.

جدول ۶ - مقایسه کامل دقت مدل مقاله و مدل بهبود یافته

		Support	F1-Score	recall	precision
Benign	مدل مقاله	۱۸۰	۰.۸۲	۰.۷۵	۰.۹۱
	مدل بهبود یافته	۱۸۰	۰.۸۷	۰.۸۲	۰.۹۳
Malignant	مدل مقاله	۱۸۰	۰.۸۵	۰.۹۳	۰.۷۹
	مدل بهبود یافته	۱۸۰	۰.۸۶	۰.۹۲	۰.۸۱
Accuracy	مدل مقاله	۳۶۰	۰.۸۴		
	مدل بهبود یافته	۳۶۰	۰.۸۷		
Macro avg	مدل مقاله	۳۶۰	۰.۸۴	۰.۸۴	۰.۸۵
	مدل بهبود یافته	۳۶۰	۰.۸۷	۰.۸۷	۰.۸۷
Weighted avg	مدل مقاله	۳۶۰	۰.۸۴	۰.۸۴	۰.۸۵
	مدل بهبود یافته	۳۶۰	۰.۸۷	۰.۸۷	۰.۸۷

تحلیل Benign : مدل بهبود یافته توانسته است با افزایش همزمان Precision وRecall ، معیار F1-Score را بهبود بخشد. این نشان‌دهنده آن است که مدل بهبود یافته در شناسایی نمونه‌های مثبت (Benign) عملکرد متوازن‌تری دارد و اشتباهات کمتری در طبقه‌بندی داشته است.

تحلیل Malignant : در مورد داده‌های Malignant، Recallمدل مقاله بالاتر است، اما Precision پایین‌تر است. این به این معناست که مدل مقاله نمونه‌های بیشتری را به‌عنوان مثبت شناسایی کرده است، اما برخی از این نمونه‌ها اشتباه بوده‌اند. مدل بهبود یافته با بهبود Precision و حفظ یک Recall بالا، عملکرد کلی متوازن‌تری داشته است و باعث افزایش F1-Score شده است.

تحلیل Accuracy : مدل بهبود یافته به دلیل بهبود متوازن در شناسایی نمونه‌های Benign و Magilgnant، دقت کلی بالاتری دارد. این نشان می‌دهد که تغییرات ساختاری در مدل باعث بهبود توانایی کلی در تشخیص نمونه‌های درست شده است.

Macro avg : مدل بهبود یافته در تمام معیارها(F1-Score) ، Recall، و (Precision بهبود یافته است (از ۰.۸۴ به ۰.۸۷).

- Weighted avg : نتایج مشابه بهبود در Macro avg است، که نشان‌دهنده تأثیر مثبت تغییرات ساختاری مدل بر عملکرد کلی آن است.

تفاوت اول: استفاده ازDropout

- در مدل بهبود یافته، از دو لایه Dropout با نرخ ۲۵٪ و ۵۰٪ استفاده شده است. این کار باعث کاهش **overfitting**در مدل شده است، زیرا برخی از نورون‌ها در طول فرآیند آموزش غیرفعال شده‌اند. در نتیجه، مدل بهبود یافته عملکرد بهتری روی داده‌های تست داشته و توانسته تعادل بهتری بین Recall و Precision برقرار کند.

تفاوت دوم: تغییر بهینه‌ساز از Adam بهNadam

- Nadamبا اضافه کردن شتاب نستوروف بهAdam ، توانسته سرعت همگرایی مدل را بهبود دهد و به یافتن نقاط بهینه‌تر در فضای پارامترها کمک کند. این تغییر به بهبود جزئی در دقت کلی و بهبود عملکرد در معیارهای جزئی کمک کرده است.

تفاوت سوم: نوع داده‌افزایی(Data Augmentation)

- مدل مقاله از **Dataloader**با تغییرات دستی استفاده کرده است، که ممکن است شامل تغییرات محدود باشد. در مقابل، مدل بهبود یافته از **ImageDataGenerator**استفاده کرده است، که ابزار جامعی برای ایجاد تنوع بیشتر در داده‌های آموزشی است. این باعث شده که مدل بهبود یافته تنوع بیشتری در داده‌های آموزشی ببیند و در نتیجه بهتر بتواند به داده‌های تست تعمیم یابد. البته لازم به ذکر است در مدل بهبود یافته نحوه استفاده از Memory و GPU به اندازه مدل مقاله بهینه نیست (به دلیل عدم استفاده از DataLoader).

۱-۵. مدل عمیق تر (امتیازی)

برای این بخش به منظور بهبود کیفیت مدل عملیات‌های زیر انجام شده است.

۱) **عدم استفاده از Resize:** با این کار عکس با ابعاد واقعی ۲۲۴ در ۲۲۴ در مدل به کار گرفته می‌شود. بدین صورت مدل با عکس‌هایی که کیفیت بالاتری دارند و الگوهای کلی در آنها بهتر مشخص است به کار گرفته شده و عملکرد بهتری خواهد داشت.

۲) **لایه‌های عمیق تر در بخش Convolutional:** به منظور بهبود کیفیت مدل و از آنجایی که عکس‌ها اکنون دارای جزئیات بیشتری هستند از لایه‌های بیشتری استفاده کرده‌ایم تا عملیات فیلتر کردن و پیدا کردن الگوهای عکس به شکل عمیق‌تر و بهتری انجام شود. بدین صورت که یک لایه قبل لایه ۱۶ فیلتر، با ۸ فیلتر و یک لایه بعد از ۱۲۸ فیلتر با ۵۱۲ فیتلر.

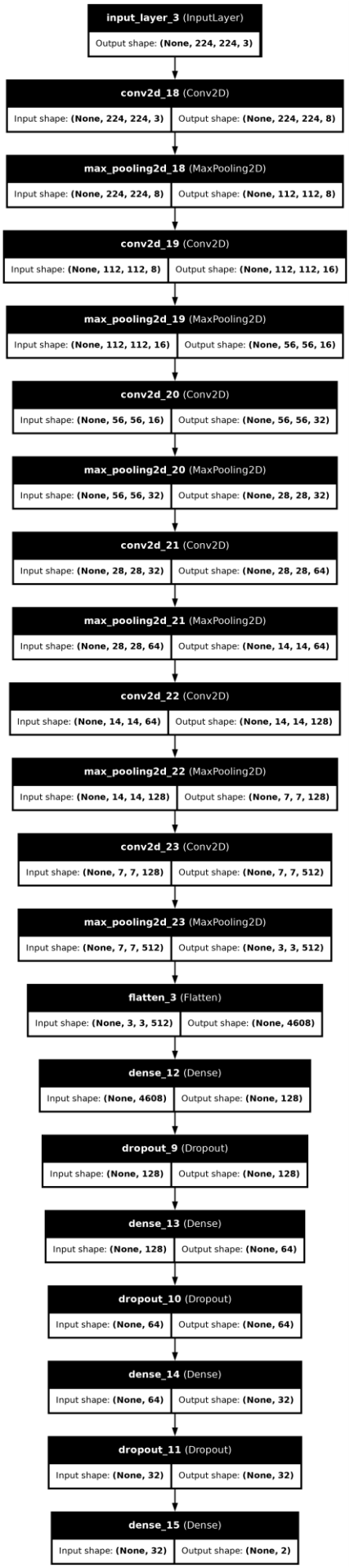
۳) **استفاده از Dropout مشابه مدل بهبود یافته در بخش قبل:** با این کار قابلیت تعمیم مدل افزایش میابد.

۴) **افزایش تعداد Epoch از ۵۰ به ۱۰۰:** با این کار مدل فرصت کافی برای نشان دادن عملکرد خود را خواهد داشت از طرفی early stop هم تعریف شده است پس اگر نیازی به این مورد هم نباشد برنامه به صورت خودکار زودتر متوقف خواهد شد. هرچند نتایج نشان می‌دهد مدل تا گام ۹۰ پیش رفته است و این افزایش هم کمک شایانی به مدل کرده است.

ساختار مدل به صورت زیر خواهد بود:

Layer (type)	Output Shape	Param #
input_layer_3 (InputLayer)	(None, 224, 224, 3)	0
conv2d_18 (Conv2D)	(None, 224, 224, 8)	224
max_pooling2d_18 (MaxPooling2D)	(None, 112, 112, 8)	0
conv2d_19 (Conv2D)	(None, 112, 112, 16)	1,168
max_pooling2d_19 (MaxPooling2D)	(None, 56, 56, 16)	0
conv2d_20 (Conv2D)	(None, 56, 56, 32)	4,640
max_pooling2d_20 (MaxPooling2D)	(None, 28, 28, 32)	0
conv2d_21 (Conv2D)	(None, 28, 28, 64)	18,496
max_pooling2d_21 (MaxPooling2D)	(None, 14, 14, 64)	0
conv2d_22 (Conv2D)	(None, 14, 14, 128)	73,856
max_pooling2d_22 (MaxPooling2D)	(None, 7, 7, 128)	0
conv2d_23 (Conv2D)	(None, 7, 7, 512)	590,336
max_pooling2d_23 (MaxPooling2D)	(None, 3, 3, 512)	0
flatten_3 (Flatten)	(None, 4608)	0
dense_12 (Dense)	(None, 128)	589,952
dropout_9 (Dropout)	(None, 128)	0
dense_13 (Dense)	(None, 64)	8,256
dropout_10 (Dropout)	(None, 64)	0
dense_14 (Dense)	(None, 32)	2,080
dropout_11 (Dropout)	(None, 32)	0
dense_15 (Dense)	(None, 2)	66

شکل ۴۰ - ساختار کلی مدل عمیق تر


input_layer_3 (InputLayer)
Output shape: (None, 224, 224, 3)
conv2d_18 (Conv2D)
Input shape: (None, 224, 224, 3)
Output shape: (None, 224, 224, 8)
max_pooling2d_18 (MaxPooling2D)
Input shape: (None, 224, 224, 8)
Output shape: (None, 112, 112, 8)
conv2d_19 (Conv2D)
Input shape: (None, 112, 112, 8)
Output shape: (None, 112, 112, 16)
max_pooling2d_19 (MaxPooling2D)
Input shape: (None, 112, 112, 16)
Output shape: (None, 56, 56, 16)
conv2d_20 (Conv2D)
Input shape: (None, 56, 56, 16)
Output shape: (None, 56, 56, 32)
max_pooling2d_20 (MaxPooling2D)
Input shape: (None, 56, 56, 32)
Output shape: (None, 28, 28, 32)
conv2d_21 (Conv2D)
Input shape: (None, 28, 28, 32)
Output shape: (None, 28, 28, 64)
max_pooling2d_21 (MaxPooling2D)
Input shape: (None, 28, 28, 64)
Output shape: (None, 14, 14, 64)
conv2d_22 (Conv2D)
Input shape: (None, 14, 14, 64)
Output shape: (None, 14, 14, 128)
max_pooling2d_22 (MaxPooling2D)
Input shape: (None, 14, 14, 128)
Output shape: (None, 7, 7, 128)
conv2d_23 (Conv2D)
Input shape: (None, 7, 7, 128)
Output shape: (None, 7, 7, 512)
max_pooling2d_23 (MaxPooling2D)
Input shape: (None, 7, 7, 512)
Output shape: (None, 3, 3, 512)
flatten_3 (Flatten)
Input shape: (None, 3, 3, 512)
Output shape: (None, 4608)
dense_12 (Dense)
Input shape: (None, 4608)
Output shape: (None, 128)
dropout_9 (Dropout)
Input shape: (None, 128)
Output shape: (None, 128)
dense_13 (Dense)
Input shape: (None, 128)
Output shape: (None, 64)
dropout_10 (Dropout)
Input shape: (None, 64)
Output shape: (None, 64)
dense_14 (Dense)
Input shape: (None, 64)
Output shape: (None, 32)
dropout_11 (Dropout)
Input shape: (None, 32)
Output shape: (None, 32)
dense_15 (Dense)
Input shape: (None, 32)
Output shape: (None, 2)

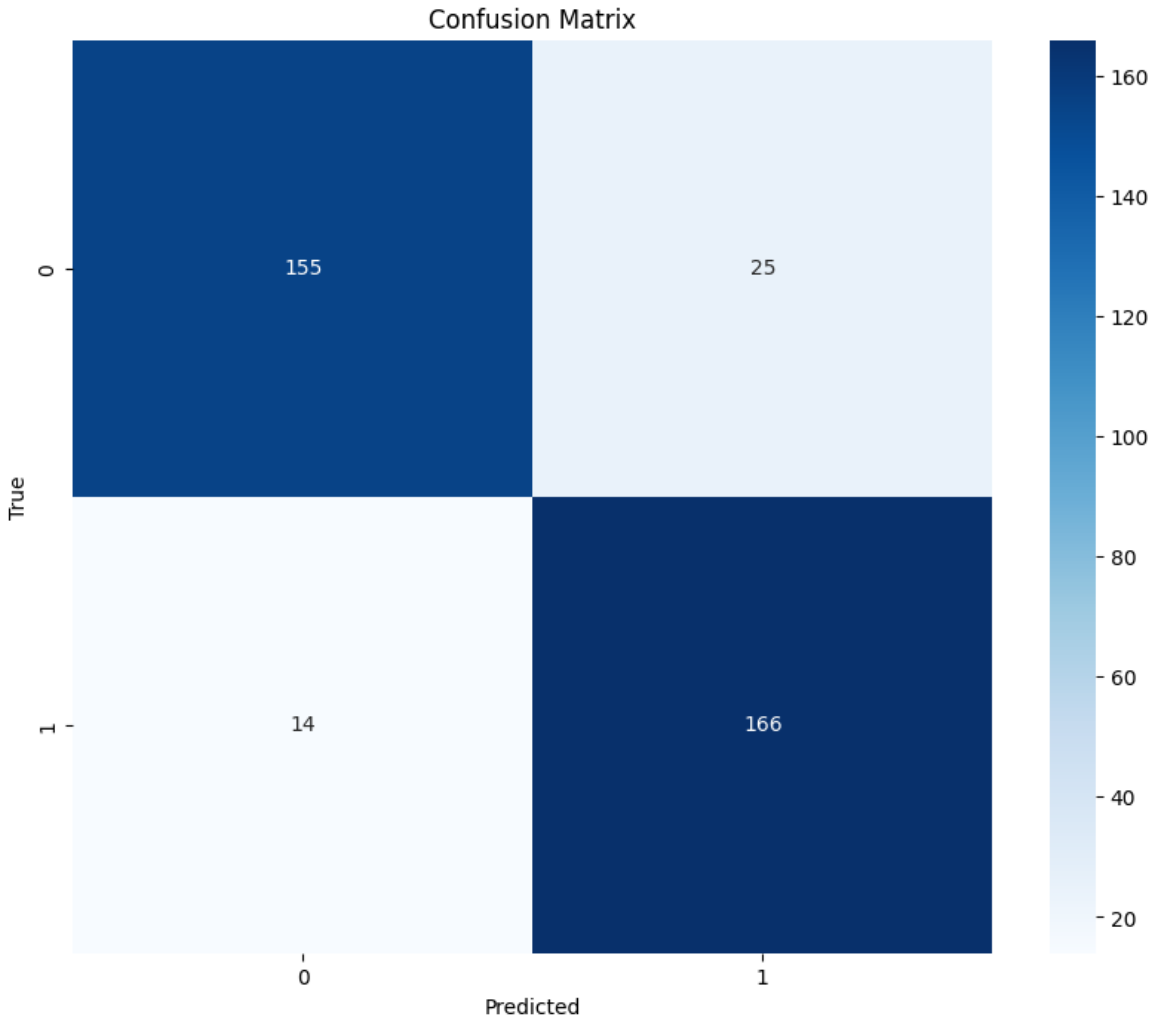
شکل ۴۱ - ساختار لایه‌های پیاده سازی شده برای مدل عمیق در کد

در نهایت نتیجه پیاده سازی مدل با این ساختار به فرم زیر است.

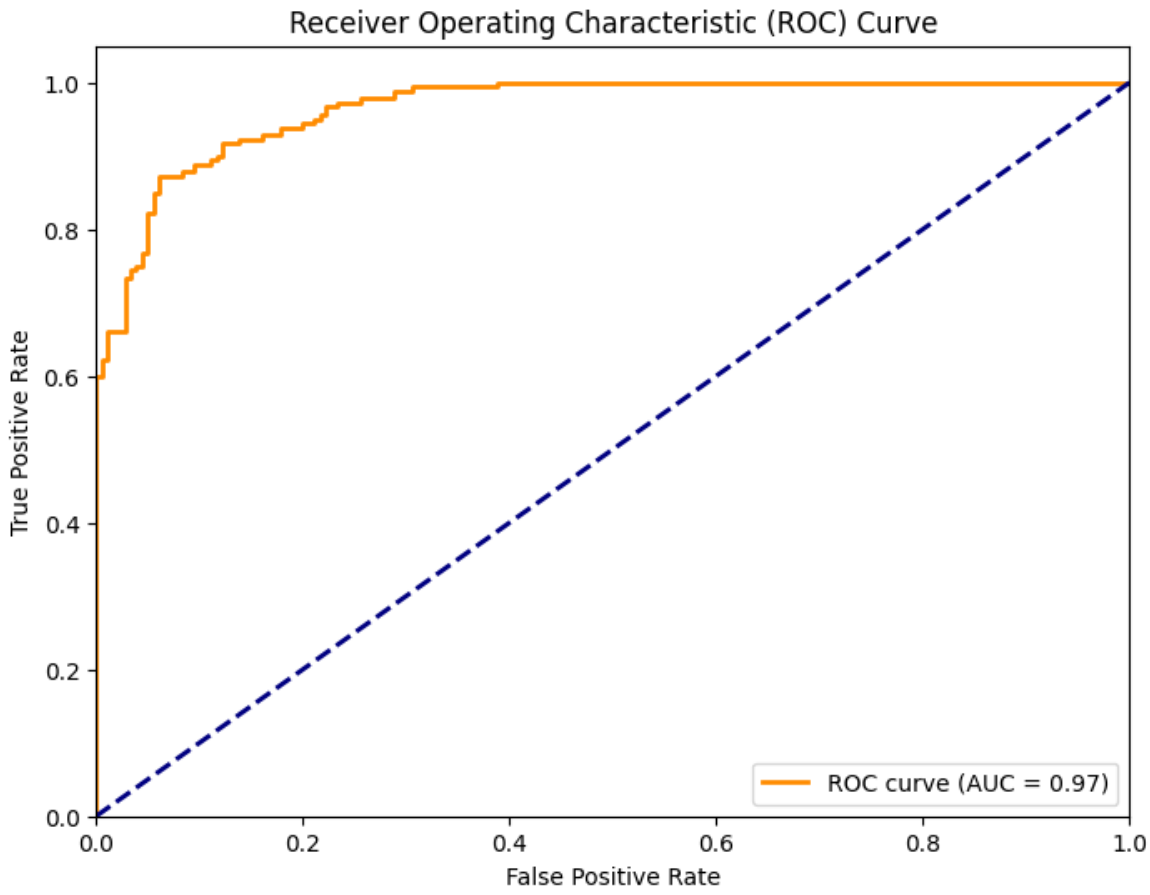
جدول ۷ - دقت مدل عمیق تر

	Support	F1- Score	recall	precision
Benign	۱۸۰	۰.۸۹	۰.۸۶	۰.۹۲
Malignant	۱۸۰	۰.۸۹	۰.۹۲	۰.۸۷
Accuracy	۳۶۰	۰.۸۹		
Macro avg	۳۶۰	۰.۸۹	۰.۸۹	۰.۸۹
Weighted avg	۳۶۰	۰.۸۹	۰.۸۹	۰.۸۹

دقت مدل در این حالت به دلیل بالا تر بودن وضوح تصویر، فیلتر بهتر به دلیل لایه های بیشتر و عمیق تر و همینطور وجود Dropout بهتر از مدل های قبلی است و به عدد **۸۹ درصد** رسیده است.

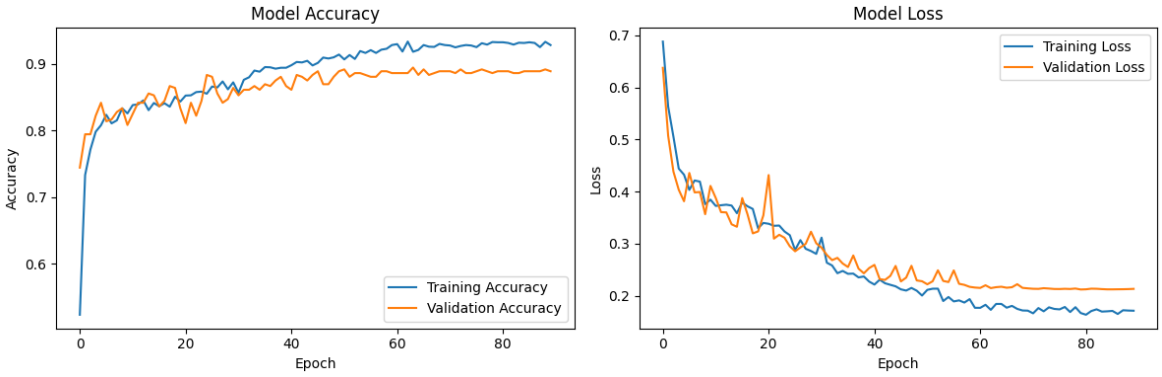


شکل ۴۲- ماتریس آشفتگی مدل عمیق تر



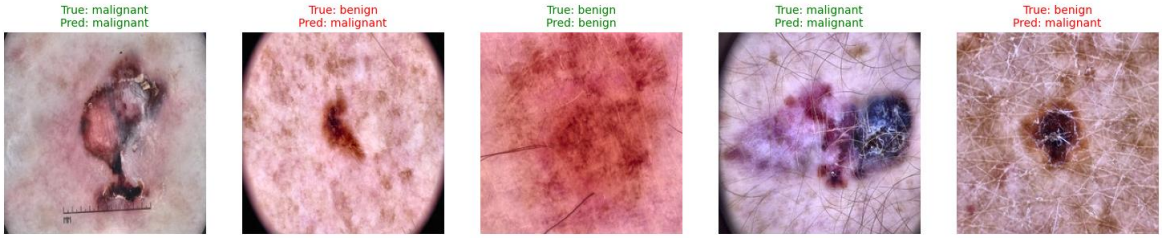
شکل ۴۳ - نمودار **ROC** مدل عمیق تر

مقدار AUC در این مدل ۹۷ درصد است که از مدل مقاله و مدل بهبود یافته عدد بالاتری را نشان می‌دهد. این موضوع نشان از دقت خوب مدل به نسبت دو مدل دیگر می‌دهد.



شکل ۴۴ - نمودار تغییرات **loss** و **accuracy**

نمودارها روند خوب و صحیحی را نشان می دهند همینطور نزدیک بودن نمودار train و validation نشان از تعمیم پذیری خوب مدل می دهد.



شکل ۴۵ - نمونه ای از پیشبینی انجام شده توسط مدل

```
Detailed Metrics:

Class: benign
Accuracy: 0.8917
Precision: 0.9172
Recall: 0.8611
F1-score: 0.8883
True Positives: 155
False Positives: 14
True Negatives: 166
False Negatives: 25

Class: malignant
Accuracy: 0.8917
Precision: 0.8691
Recall: 0.9222
F1-score: 0.8949
True Positives: 166
False Positives: 25
True Negatives: 155
False Negatives: 14
```

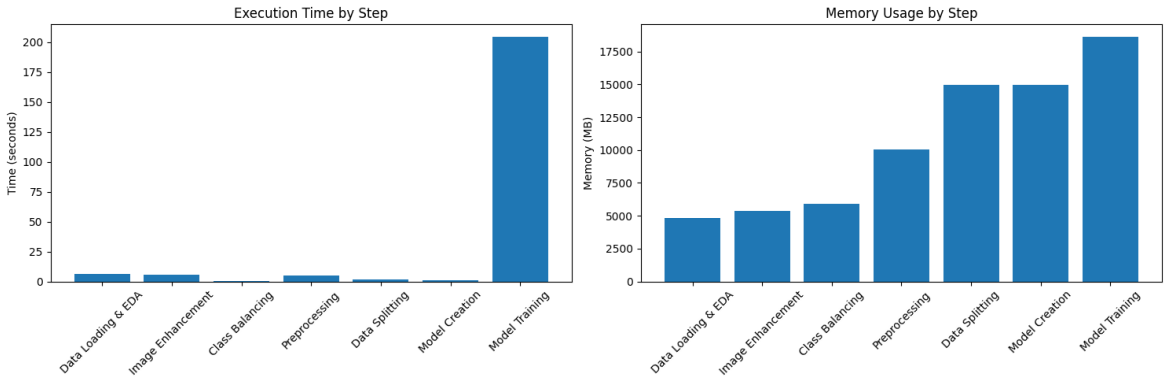
شکل ۴۶ - **Detail Metrics** برای مدل عمیق تر

```
Performance Metrics:
--------------------------------------------------

Data Loading & EDA:
Time: 6.15 seconds
Memory: 4822.27 MB

Image Enhancement:
Time: 5.97 seconds
Memory: 5371.02 MB

Class Balancing:
Time: 0.29 seconds
Memory: 5888.25 MB

Preprocessing:
Time: 4.91 seconds
Memory: 10022.77 MB

Data Splitting:
Time: 1.46 seconds
Memory: 14983.96 MB

Model Creation:
Time: 1.23 seconds
Memory: 14983.96 MB

Model Training:
Time: 204.63 seconds
Memory: 18621.46 MB

Total Pipeline Time: 224.64 seconds
```

شکل ۴۷ - **Performance** مدل عمیق تر

جدول ۸- مقایسه زمان اجرای مدل مقاله و مدل عمیق تر

	زمان اجرا (sec)
مدل مقاله	۳۶.۰۲
مدل عمیق تر	۲۲۴.۶۴



شکل ۴۸ - **performance** مدل عمیق تر به صورت جدول

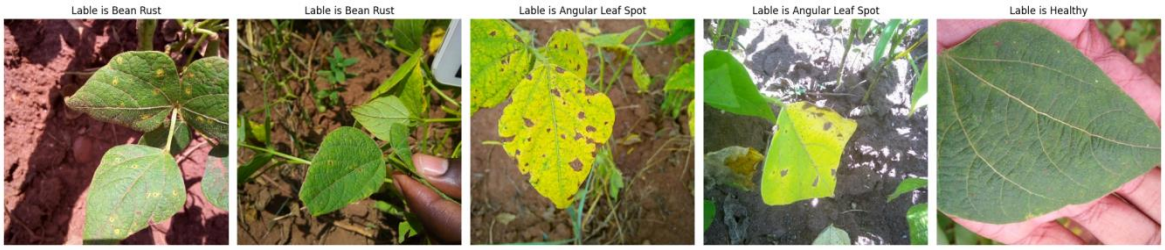
همانطور که مشخص است زمان اجرای این مدل به مراتب به نسبت مدل مقاله طولانی تر است. دلیل این اتفاق ابعاد بزرگتر عکس و تعداد لایه های بیشتر در مدل ما است.

پرسش۲–تشخیص بیماری‌های برگ لوبیا با شبکه عصبی

۲-۱.پیش پردازش تصاویر

برای پیش‌پردازش مجموعه داده در مقاله "طبقه بندی بیماری های برگ لوبیا با استفاده از مدل CNN دقیق تنظیم شده"، مراحل زیر مورد استفاده قرار گرفته است:

مجموعه داده‌ها شامل ۱۲۹۵ تصویر از برگ‌های لوبیا بوده که به سه دسته‌ی سالم، لکه زاویه‌ای و زنگ لوبیا تقسیم شده است. داده‌ها به نسبت ۸۰:۱۰:۱۰ برای مجموعه‌های آموزشی، اعتبارسنجی و آزمون تقسیم شدند. پیش پردازش های زیر روی مجموعه داده اعمال شده است که به صورت زیر هستند:



شکل ۵۰- تصویر نمونه با لیبل

خواندن و نمایش تصاویر: این مرحله برای آشنایی با ساختار داده‌ها و بررسی کیفیت تصاویر ضروری است. نمایش نمونه‌ها، دسته‌بندی اولیه داده‌ها و چک کردن خطاهای احتمالی (مانند وجود تصاویر بی‌کیفیت) را تسهیل می‌کند.

تغییر اندازه تصاویر: مدل‌هایCNN مانند MobileNetV2، EfficientNetB6 و NasNet ورودی‌هایی با ابعاد مشخص (به ترتیب به سایز ۲۲۴×۲۲۴، ۵۲۸×۵۲۸ و ۳۳۱×۳۳۱) را می‌پذیرند. تغییر اندازه تصاویر به ابعاد استاندارد هر مدل، سازگاری داده‌ها با مدل را تضمین می‌کند. خلاصه تغییر اندازه تصاویرتضمین می‌کند که داده‌ها با معماری مدل سازگار هستند و از اعوجاج تصاویر جلوگیری می‌کند و مدل ویژگی‌ها را بهتر یاد می‌گیرد.

نرمال‌سازی پیکسل‌ها: نرمال‌سازی مقادیر پیکسل‌ها به بازه [۰,۱]، باعث می‌شود مقادیر ورودی مدل محدود شوند. این کار به پایداریآموزش، کاهش حساسیت به پارامترهای بزرگ کمتر شودو به بهبود همگرایی کمک می‌کند. خلاصه اینکه سرعت یادگیری را افزایش می‌دهد و گرادیان‌ها را پایدارتر می‌کند.

تقویت داده‌ها (Data Augmentation): تقویت داده‌ها به افزایش تنوع مجموعه داده کمک می‌کند و مدل را قادر می‌سازد تا در شرایط مختلف (مانند تغییر روشنایی، چرخش، و وارونگی) بهتر عمل کند. این کار باعث کاهش بیش‌برازش (Overfitting) و افزایش تعمیم‌پذیری مدل می‌شود. این کار باعث افزایش مقاومت مدل در برابر شرایط متغیر می‌شود.

تقسیم داده‌ها به سه مجموعه: آموزش (۸۰٪): برایادگیری مدل.اعتبارسنجی (۱۰٪): برای ارزیابی عملکرد مدل در طول آموزش و تنظیم هایپر پارامترها.آزمون (۱۰٪): برای ارزیابی نهایی عملکرد مدل بر روی داده‌های دیده‌نشده. تقسیم داده‌ها از ارزیابی عملکرد مدل در شرایط دیده‌نشده اطمینان حاصل می‌کند. داده بصورت تقسیم شده بودند واین کار انجام نشده است.

۲-۲. پیاده سازی

۲-۲-۱. انتخاب مدل

این سه مدل با معماری‌های پیشرفته، برای مسائل پردازش تصویر توسعه یافته‌اند.در ادامه، جزئیات و ویژگی‌های کلیدی هر مدل توضیح داده شده است.

مدل EfficientNetB6

EfficientNetیک مدل شبکه عصبی کانولوشنی (CNN) است که توسط تیم گوگل توسعه داده شده است. هدف اصلی این مدل ارائه‌ی معماری‌هایی است که در برابر سرعت و دقت بالا، استفاده بهینه‌تری از منابع داشته باشند. در نسخه‌های مختلف این مدل، EfficientNetB6یکی از قدرتمندترین نسخه‌ها است که در مقایسه با مدل‌های پیشین مانند EfficientNetB0 تا B5 دارای دقت بالاتری است.

از ویژگی‌ها این مدل می توان به بازدهی بهینه اشاره کرد.EfficientNet از یک استراتژی به نام "compound scaling" استفاده می‌کند که در آن سه پارامتر (عمق، عرض و وضوح تصویر) به‌طور همزمان و به صورت متوازن افزایش می‌یابند تا کارایی مدل بهینه شود. لایه‌های اصلیEfficientNet شامل لایه‌های معمولی کانولوشن، لایه‌های بکار رفته در مدل‌های پیشرفته‌تر مانند SE-Block (Squeeze and Excitation)، و لایه‌های متوسط است.(Squeeze and Excitation (SE: این لایه به مدل اجازه می‌دهد تا به طور هوشمندانه ویژگی‌های مفید را برای هر کانال از ویژگی‌ها استخراج کند.

ساختار مدل شامل Depth (تعداد لایه‌های شبکه)، Width (تعداد فیلترهای کانولوشن در هر لایه) و Resolution (وضوح ورودی تصویر) است. در نهایت،EfficientNet با استفاده از ترکیب این سه بعد (عمق، عرض، و وضوح) به صورت بهینه‌تری منابع را مصرف می‌کند.EfficientNetB6 دارای ۱۵۵ میلیون پارامتر است و معمولاً برای مشکلات پیچیده‌تر مانند طبقه‌بندی تصاویر با دقت بالا استفاده می‌شود.

مدل MobileNetV2

یک مدل CNN سبک و سریع است که به ویژه برای دستگاه‌های موبایل و محیط‌هایی که منابع محدودی دارند طراحی شده است. این مدل در ادامه‌یMobileNetV1 و بهینه‌سازی‌های انجام شده بر روی آن ارائه شده است.

ویژگی‌های آن شامل موارد زیر است.

Inverted Residual Blocks: این ویژگییکی از نقاط قوت MobileNetV2 است که در آن از لایه‌های معکوس با استفاده از لایه‌های ضخیم و نازک برای افزایش کارایی و کاهش پیچیدگی استفاده می‌شود.

Depthwise Separable Convolutions: به جای استفاده از کانولوشن‌های معمولی، در MobileNetV2 از کانولوشن‌های جداگانه برای هر کانال (Depthwise) و ترکیب این کانال‌ها (Pointwise) استفاده می‌شود که باعث کاهش تعداد محاسبات و پیچیدگی مدل می‌شود.

Linear Bottleneck: در این مدل، لایه‌هایBottleneck به صورت خطی طراحی شده‌اند که باعث کاهش تعداد پارامترها و بهبود سرعت اجرا می‌شود.

کاهش حجم مدل: به دلیل معماری بهینه، این مدل می‌تواند بر روی دستگاه‌های موبایل با قدرت پردازشی محدود اجرا شود بدون اینکه دقت آن به صورت محسوسی کم شود.

ساختار مدلMobileNetV2 از ۲ لایه کانولوشن Depthwise separable، یک لایه ضخیم و سپس یک لایه ترکیب‌کننده (pointwise) استفاده می‌کند. تعداد پارامترهای مدل معمولیMobileNetV2 حدود ۳.۴ میلیون است که این مدل را بسیار سبک و سریع می‌کند.MobileNetV2 به‌طور ویژه برای کاربردهای بلادرنگ(real-time) در موبایل‌ها و دستگاه‌های با منابع محدود طراحی شده است.

مدل NasNet

NasNetیک مدل معماری است که توسط Google Brain با استفاده از روش جستجوی معماری شبکه عصبی (Neural Architecture Search) ساخته شده است. در این روش، خود مدل به صورت خودکار معماری بهینه‌تری برای شبکه عصبی را انتخاب می‌کند.

ویژگی‌های آن شامل موارد زیر می‌شود:

Neural Architecture Search (NAS): این تکنیک به طور خودکار ساختار بهینه شبکه عصبی را جستجو می‌کند. به این ترتیب، معماری‌های پیچیده‌تری نسبت به طراحی‌های دستی به دست می‌آید.

آموزش از طریق جستجو: NASNet از یک الگوریتم جستجو برایافتن معماری شبکه‌ای استفاده می‌کند که بهترین عملکرد را در یک مجموعه داده خاص داشته باشد.

Module-based: این مدل به صورت ماژولار است و از قطعاتی استفاده می‌کند که هرکدام بهینه‌سازی شده‌اند.

ساختار مدلNASNet از سلول‌ها (cells) برای ساختار شبکه استفاده می‌کند که این سلول‌ها توسط فرآیند جستجو در NAS بهینه‌سازی می‌شوند.در NASNet، سلول‌ها می‌توانند شامل ترکیب‌هایی از لایه‌های کانولوشن، Pooling، Normalization، و دیگر لایه‌ها باشند.مدل NASNet-Large دارای ۸۵ میلیون پارامتر است که از آن در پروژه‌های پیچیده‌تری مانند شناسایی اشیاء و طبقه‌بندی تصاویر استفاده می‌شود.

```python
# Transfer Learning
def build_transfer_model(base_model):
    base_model.trainable = False
    x = base_model.output
    x = GlobalAveragePooling2D()(x)
    x = Dense(64, activation='relu')(x)
    x = Dropout(0.3)(x)
    output = Dense(3, activation='softmax')(x)
    return Model(inputs=base_model.input, outputs=output)
```

```python
# MobileNetV2
base_model_mobilenet = MobileNetV2(input_shape=(224, 224, 3), include_top=False, weights='imagenet')
model_mobilenet = build_transfer_model(base_model_mobilenet)

# EfficientNetB6
base_model_efficientnet = EfficientNetB6(input_shape=(528, 528, 3), include_top=False, weights='imagenet')
model_efficientnet = build_transfer_model(base_model_efficientnet)

# NasNet
base_model_nasnet = NASNetMobile(input_shape=(331, 331, 3), include_top=False, weights='imagenet')
model_nasnet = build_transfer_model(base_model_nasnet)
```

شکل ۵۱ - پیاده سازی یادگیری انتقالی با هر سه مدل

۲-۲-۲. تقویت داده

تقویت داده

در تکنیک‌های تقویت داده، تغییراتی روی تصاویر اعمال می‌شود تا به مدل کمک شود در شرایط مختلف بهتر تعمیم دهد. این تکنیک‌ها، مخصوصاً در مسائل طبقه‌بندی تصویر، از بیش‌برازشمدل جلوگیری می‌کنند و باعث افزایش دقت مدل می‌شوند.

دلایل انتخاب تکنیک‌های تقویت داده و تاثیرات آن‌ها

چرخش (Rotation): چرخش تصاویر باعث می‌شود مدل بتواند به راحتی اجسامی که در زوایای مختلف ظاهر می‌شوند را شناسایی کند. این کار باعث می‌شود مدل نسبت به جهت‌دهی تصاویر حساسیت کمتری داشته باشد.

وارونگی افقی و عمودی (Flip): این عملیات به مدل کمک می‌کند تا توانایی شناسایی ویژگی‌های تقارنی در تصاویر را تقویت کند. این کار به ویژه برای تشخیص اشیایی که در دو جهت می‌توانند مشابه باشند، مفید است.

تغییر روشنایی و کنتراست (Brightness and Contrast Adjustment): تغییر روشنایی و کنتراست تصاویر به مدل کمک می‌کند تا بتواند تصاویر را در شرایط نوری مختلف تشخیص دهد. این عملیات به مدل در برابر تغییرات نوری محیط، مانند نور آفتاب و سایه، مقاومت می‌دهد

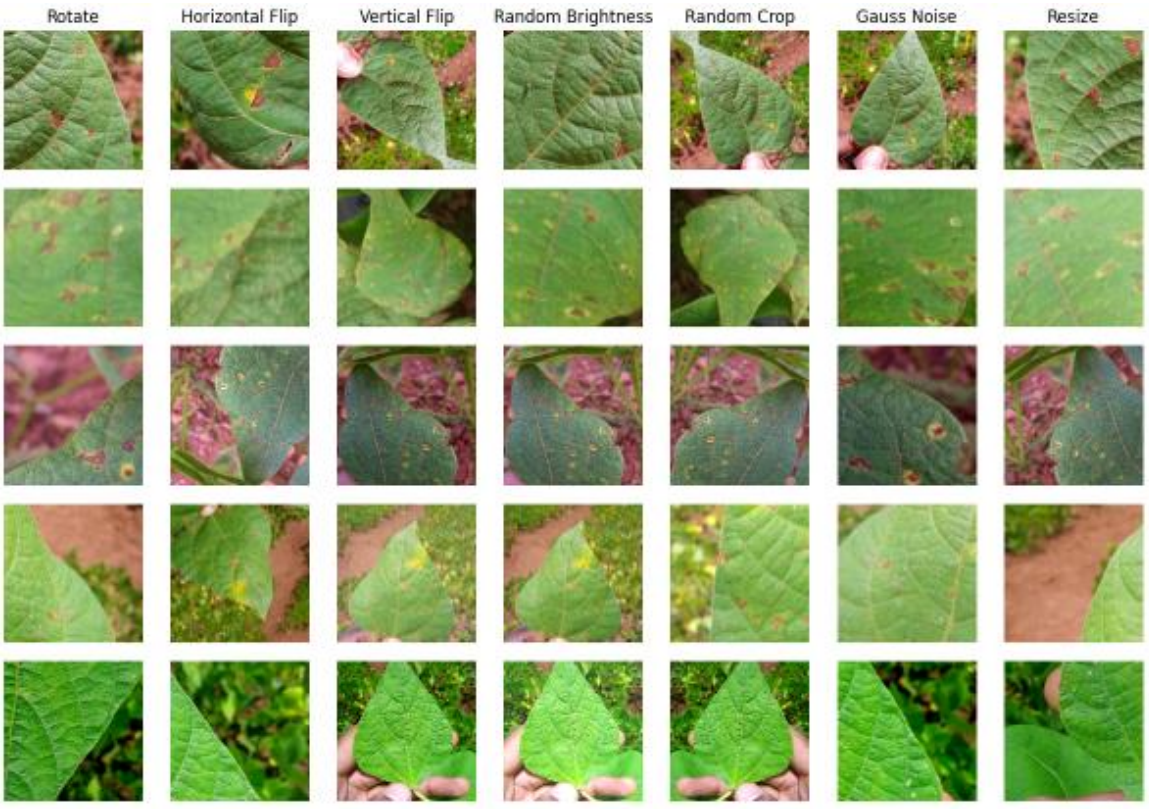
برش تصادفی (Random Crop): با انتخاب بخش‌های تصادفی از تصویر، مدل به ویژگی‌های محلی توجه بیشتری می‌کند و از تمرکز رویبک بخش خاص از تصویردوری می‌کند.

اعمال نویز (Gaussian Noise): نویز گوسی، به مدل کمک می‌کند تا نسبت به نویزهای احتمالی در تصاویر مقاوم شود. این نوع نویز، مشابه نویزهای طبیعی در تصاویر است و به مدل در یادگیری بهتر ویژگی‌های مهم کمک می‌کند.

این تکنیک‌های تقویت داده به مدل کمک می‌کنند تا تصاویر برگ لوبیا را در شرایط مختلف، مانند چرخش‌های گوناگون، شرایط نوری متفاوت و حتی وجود نویز، به درستی طبقه‌بندی کند. نمایش نمونه‌ها نشان می‌دهد که مدل چگونه به ویژگی‌های مختلف، مانند تقارن و تغییرات نوری، حساس‌تر می‌شود. این افزایش تنوع در داده‌های آموزشی به مدل کمک می‌کند تا قابلیت تعمیم‌دهی خود را افزایش دهد و دقت آن بهبود یابد.

```python
transform = A.Compose([
    A.Rotate(limit=40, p=0.5),
    A.HorizontalFlip(p=0.5),
    A.VerticalFlip(p=0.2),
    A.RandomBrightnessContrast(p=0.5),
    A.RandomCrop(width=180, height=180, p=0.5),
    A.GaussNoise(var_limit=(10, 50), p=0.3),
    A.Resize(224, 224),
    ToTensorV2()
])

augmented_title = ["Rotate", "Horizontal Flip", "Vertical Flip", "Random Brightness", "Random Crop", "Gauss Noise", "Resize"]
for j, sample_image in enumerate(random_images):
    image = cv2.imread(sample_image)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    fig, axs = plt.subplots(1, 7, figsize=(15, 7))

    if j == 0:
        for k in range(7):
            axs[k].set_title(augmented_title[k])

    for i in range(7):
        augmented = transform(image=image)
        aug_image = augmented['image']
        axs[i].imshow(aug_image.permute(1, 2, 0))
        axs[i].axis('off')
```

شکل ۵۲ - کد تقویت داده با کتابخانه **albumentations**



شکل ۵۳ - نمونه عکس های تقویت داده با کتابخانه **albumentations**

۳-۲-۲. تقویت داده

مدل‌هایCNN مانند EfficientNetB6، MobileNetV2، و NasNet هر کدام دارای اندازه ورودی مشخصی برای تصاویر هستند. برای اینکه این مدل‌ها بتوانند به درستی از داده‌ها استفاده کنند، لازم است تصاویر ورودی به ابعاد مناسب تغییر اندازه داده شوند.

سایز ورودی مناسب برای هر مدل

- MobileNetV2: اندازه ورودی پیشنهادی این مدل ۲۲۴×۲۲۴ پیکسل است.
- EfficientNetB6: اندازه ورودی مناسب برای این مدل ۵۲۸×۵۲۸ پیکسل است.
- NasNet: اندازه ورودی پیشنهادی۳۳۱×۳۳۱ پیکسل است.

```
mobile_image_size = (224, 224)
efficient_image_size = (528, 528)
nas_image_size = (331, 331)
BATCH_SIZE = 32

models = {
    "MobileNetV2": (224, 224),
    "EfficientNetB6": (528, 528),
    "NasNetMobile": (331, 331)
}


train_dir = path + "/train/train"
valid_dir = path + "/validation/validation"

data_gen = ImageDataGenerator(rescale=1./255)

train_gen = {}
valid_gen = {}

for model_name, size in models.items():
    train_gen[model_name] = data_gen.flow_from_directory(train_dir, target_size=size, batch_size=32, class_mode='categorical')
    valid_gen[model_name] = data_gen.flow_from_directory(valid_dir, target_size=size, batch_size=32, class_mode='categorical')
```

شکل ۵۴ - کد مناسب سازی ورودی برای هر مدل

تنظیم اندازه تصاویر ورودی با اندازه‌های موردنیاز هر مدل بسیار مهم است، زیرا:

افزایش دقت مدل: تنظیم صحیح اندازه تصاویر باعث می‌شود که مدل بتواند به درستی ویژگی‌های مهم را شناسایی کند. اگر تصویر ورودی به اندازه صحیح نباشد، مدل ممکن است نتواند الگوها و ویژگی‌های موجود در تصویر را به درستی تشخیص دهد، که منجر به کاهش دقت می‌شود.

بهینه‌سازی مصرف حافظه و زمان پردازش: مدل‌ها با سایزهای ورودی بزرگ‌تر (مثل EfficientNetB6 با سایز ۵۲۸×۵۲۸) به حافظه و زمان پردازش بیشتری نیاز دارند. اگر تصاویر ورودی به درستی تنظیم شوند، استفاده از منابع محاسباتی بهینه می‌شود و مدل با سرعت و کارایی بیشتری اجرا می‌شود.

حفظ نسبت ابعاد (Aspect Ratio): هنگام تغییر اندازه، حفظ نسبت ابعاد تصویر می‌تواند به شناسایی ویژگی‌های مهم تصویر کمک کند. تغییر اندازه مناسب از فشردگییا کشیدگی غیرعادی تصویر جلوگیری می‌کند و باعث می‌شود مدل بهتر بتواند ویژگی‌های اصلی را شناسایی کند.

۴-۲-۲. بهینه سازها

MSprop یا "Root Mean Square Propagation" بهینه‌سازی‌ای مبتنی بر گرادیان است که نرخ یادگیری را برای هر پارامتر به صورت تطبیقی تنظیم می‌کند. این بهینه‌ساز با حفظ میانگین مربع گرادیان‌های گذشته، به کاهش نوسانات در به‌روزرسانی‌ها کمک می‌کند و به ویژهدریادگیری مدل‌ها روی داده‌هایnoisy و با داده‌های بزرگ کاربرد دارد.این بهینه‌ساز در مسائل پیچیده با تغییرات زیاد عملکرد خوبی دارد و می‌تواند پایداری بیشتری به دقت مدل در داده‌های متنوع بدهد.

Adam (Adaptive Moment Estimation) ترکیبی از دو بهینه‌سازRMSprop و Momentum است. این بهینه‌ساز با استفاده از میانگین گرادیان و میانگین مربع گرادیان، نرخ یادگیری را به صورت تطبیقی تنظیم می‌کند. این ویژگی باعث می‌شودAdam در یادگیری الگوهای پیچیده با دقت بالا عمل کند.Adamیکی از محبوب‌ترین بهینه‌سازهاست، چون عموماً به سرعت به کمینه‌های محلی می‌رسد و نرخ یادگیری به صورت دینامیک تنظیم می‌شود. این ویژگی‌ها باعث می‌شودAdam برای اکثر مسائل یادگیری عمیق مناسب باشد.

Nadam نسخه اصلاح شده‌ای از Adam است که در آن تکنیکNesterov Momentum نیز اعمال شده است. این بهینه‌سازیک نگاه به پیش‌بینی گرادیان در گام بعدی دارد، که می‌تواند منجر به حرکت مؤثرتر به سمت کمینه‌های محلی شود. استفاده از Nesterov Momentum به Nadam امکان می‌دهد که هم سرعت و هم دقت را بهبود ببخشد، به خصوص در مسائل حساس که به بهینه‌سازی دقیق نیاز دارند.

```
# MobileNetV2 with Adam Optimizer
model_mobilenet_adam = build_transfer_model(base_model_mobilenet)
model_mobilenet_adam.compile(optimizer=Adam(learning_rate=0.001), loss='categorical_crossentropy', metrics=['accuracy'])
history_mobilenet_adam = model_mobilenet_adam.fit(train_gen['MobileNetV2'], validation_data=valid_gen['MobileNetV2'], epochs=25)
print(f"MobileNetV2 Validation Accuracy: {history_mobilenet_adam.history['val_accuracy'][-1]:.2f}")


# Save the model after training
# model_mobilenet_adam.save('mobilenetv2_adam_transfer_model.h5')
model_mobilenet_adam.save('mobilenetv2_adam_transfer_model.keras')
```

شکل ۵۵ کد مدل **MoileNetV2** با بهینه ساز **Adam**

```python
# MobileNetV2 with RMSprop Optimizer
model_mobilenet_rmsprop = build_transfer_model(base_model_mobilenet)
model_mobilenet_rmsprop.compile(optimizer=RMSprop(learning_rate=0.001), loss='categorical_crossentropy', metrics=['accuracy'])
history_mobilenet_rmsprop = model_mobilenet_rmsprop.fit(train_gen['MobileNetV2'], validation_data=valid_gen['MobileNetV2'], epochs=25)
print(f"MobileNetV2 Validation Accuracy: {history_mobilenet_rmsprop.history['val_accuracy'][-1]:.2f}")


# Save the model after training
# model_mobilenet_rmsprop.save('mobilenetv2_rmsprop_transfer_model.h5')
model_mobilenet_rmsprop.save('mobilenetv2_rmsprop_transfer_model.keras')
```

شکل ۵۶ کد مدل **MobileNetV2** با بهینه ساز **RMSprop**

```python
# MobileNetV2 with Nadam Optimizer
model_mobilenet_nadam = build_transfer_model(base_model_mobilenet)
model_mobilenet_nadam.compile(optimizer=Nadam(learning_rate=0.001), loss='categorical_crossentropy', metrics=['accuracy'])
history_mobilenet_nadam = model_mobilenet_nadam.fit(train_gen['MobileNetV2'], validation_data=valid_gen['MobileNetV2'], epochs=25)
print(f"MobileNetV2 Validation Accuracy: {history_mobilenet_nadam.history['val_accuracy'][-1]:.2f}")


# Save the model after training
# model_mobilenet_nadam.save('mobilenetv2_nadam_transfer_model.h5')
model_mobilenet_nadam.save('mobilenetv2_nadam_transfer_model.keras')
```

شکل ۵۷ - کد مدل **MobileNetV2** با بهینه ساز **Nadam**

```python
# EfficientNetB6 with RMSprop Optimizer
model_efficientnet_rmsprop = build_transfer_model(base_model_efficientnet)
model_efficientnet_rmsprop.compile(optimizer=RMSprop(learning_rate=0.001), loss='categorical_crossentropy', metrics=['accuracy'])
history_efficientnet_rmsprop = model_efficientnet_rmsprop.fit(train_gen['EfficientNetB6'], validation_data=valid_gen['EfficientNetB6'], epochs=25)
print(f"EfficientNetB6 Validation Accuracy: {history_efficientnet_rmsprop.history['val_accuracy'][-1]:.2f}")


# Save the model after training
# model_efficientnet_rmsprop.save('efficientnetb6_rmsprop_transfer_model.h5')
model_efficientnet_rmsprop.save('efficientnetb6_rmsprop_transfer_model.keras')
```

شکل ۵۸ - کد مدل **EfficientNetB6** با بهینه ساز **RMSprop**

```python
# EfficientNetB6 with Adam Optimizer
model_efficientnet_adam = build_transfer_model(base_model_efficientnet)
model_efficientnet_adam.compile(optimizer=Adam(learning_rate=0.001), loss='categorical_crossentropy', metrics=['accuracy'])
history_efficientnet_adam = model_efficientnet_adam.fit(train_gen['EfficientNetB6'], validation_data=valid_gen['EfficientNetB6'], epochs=25)
print(f"EfficientNetB6 Validation Accuracy: {history_efficientnet_adam.history['val_accuracy'][-1]:.2f}")


# Save the model after training
# model_efficientnet_adam.save('efficientnetb6_adam_transfer_model.h5')
model_efficientnet_adam.save('efficientnetb6_adam_transfer_model.keras')
```

شکل ۵۹ - کد مدل **EfficientNetB6** با بهینه ساز **Adam**

```python
# EfficientNetB6 with Nadam Optimizer
model_efficientnet_nadam = build_transfer_model(base_model_efficientnet)
model_efficientnet_nadam.compile(optimizer=Nadam(learning_rate=0.001), loss='categorical_crossentropy', metrics=['accuracy'])
history_efficientnet_nadam = model_efficientnet_nadam.fit(train_gen['EfficientNetB6'], validation_data=valid_gen['EfficientNetB6'], epochs=25)
print(f"EfficientNetB6 Validation Accuracy: {history_efficientnet_nadam.history['val_accuracy'][-1]:.2f}")


# Save the model after training
# model_efficientnet_nadam.save('efficientnetb6_nadam_transfer_model.h5')
model_efficientnet_nadam.save('efficientnetb6_nadam_transfer_model.keras')
```

شکل ۶۱ - کد مدل **MobileNetV2** با بهینه ساز **Nadam**

```python
# NasNet with RMSprop Optimizer
model_nasnet_rmsprop = build_transfer_model(base_model_nasnet)
model_nasnet_rmsprop.compile(optimizer=RMSprop(learning_rate=0.001), loss='categorical_crossentropy', metrics=['accuracy'])
history_nasnet_rmsprop = model_nasnet_rmsprop.fit(train_gen['NasNetMobile'], validation_data=valid_gen['NasNetMobile'], epochs=25)
print(f"NasNetMobile Validation Accuracy: {history_nasnet_rmsprop.history['val_accuracy'][-1]:.2f}")


# Save the model after training
# model_nasnet_rmsprop.save('nasnet_rmsprop_transfer_model.h5')
model_nasnet_rmsprop.save('nasnet_rmsprop_transfer_model.keras')
```

شکل ۶۰ - کد مدل **NasNet** با بهینه ساز **RMSprop**

```python
# NasNet with Adam Optimizer
model_nasnet_adam = build_transfer_model(base_model_nasnet)
model_nasnet_adam.compile(optimizer=Adam(learning_rate=0.001), loss='categorical_crossentropy', metrics=['accuracy'])
history_nasnet_adam = model_nasnet_adam.fit(train_gen['NasNetMobile'], validation_data=valid_gen['NasNetMobile'], epochs=25)
print(f"NasNetMobile Validation Accuracy: {history_nasnet_adam.history['val_accuracy'][-1]:.2f}")


# Save the model after training
# history_nasnet_adam.save('nasnet_adam_transfer_model.h5')
model_nasnet_adam.save('nasnet_adam_transfer_model.keras')
```

شکل ۶۲ - کد مدل **NasNet** با بهینه ساز **Adam**

```python
# NasNet with Nadam Optimizer
model_nasnet_nadam = build_transfer_model(base_model_nasnet)
model_nasnet_nadam.compile(optimizer=Nadam(learning_rate=0.001), loss='categorical_crossentropy', metrics=['accuracy'])
history_nasnet_nadam = model_nasnet_nadam.fit(train_gen['NasNetMobile'], validation_data=valid_gen['NasNetMobile'], epochs=25)
print(f"NasNetMobile Validation Accuracy: {history_nasnet_nadam.history['val_accuracy'][-1]:.2f}")


# Save the model after training
# model_nasnet_nadam.save('nasnet_nadam_transfer_model.h5')
model_nasnet_nadam.save('nasnet_nadam_transfer_model.keras')
```

شکل ۶۳ - کد مدل **NasNet** با بهینه ساز **Nadam**

تحلیل نتایج

RMSprop: عملکرد خوبی در داده‌هایnoisy دارد، اما معمولاً از Adam و Nadam دقت کمتری دارد.در این آزمایش،RMSprop برایMobileNetV2 عملکرد قابل قبولی نشان داده است.

Adam: دقت بالا و سرعت همگرایی بیشتر در اکثر مدل‌ها. بهترین عملکرد در NasNet به دلیل معماری پیچیده و نیاز به تنظیم نرخ یادگیری تطبیقی.

Nadam:دقت بالا و سرعت همگرایی مشابه Adam، اما در EfficientNetB6 عملکرد بهتری داشته است. استفاده از Nesterov Momentum ممکن است باعث همگرایی بهتر در مدل‌های عمیق‌تر شود.

۵-۲-۲. آموزش مدل

در این بخش، مراحل آموزش مدل با استفاده از مجموعه داده‌های آموزشی و ارزیابی را طبق تنظیمات مقاله پیاده‌سازی می‌کنیم. و از تکنیک Early Stopping برای جلوگیری از بیش‌برازش استفاده خواهیم کرد و نمودار تغییرات معیارهای ارزیابی (مانند دقت) و تابع هزینه را در طول آموزش و اعتبارسنجی ترسیم می‌کنیم.

بر اساس اطلاعات موجود در مقاله تعداد epoch ها ۲۵ و اندازه batch ۳۲ تا و نرخ یادگیری برابر ۰.۰۰۱، بهینه‌سازو از معیار ارزیابی دقت (accuracy) استفاده شده است.

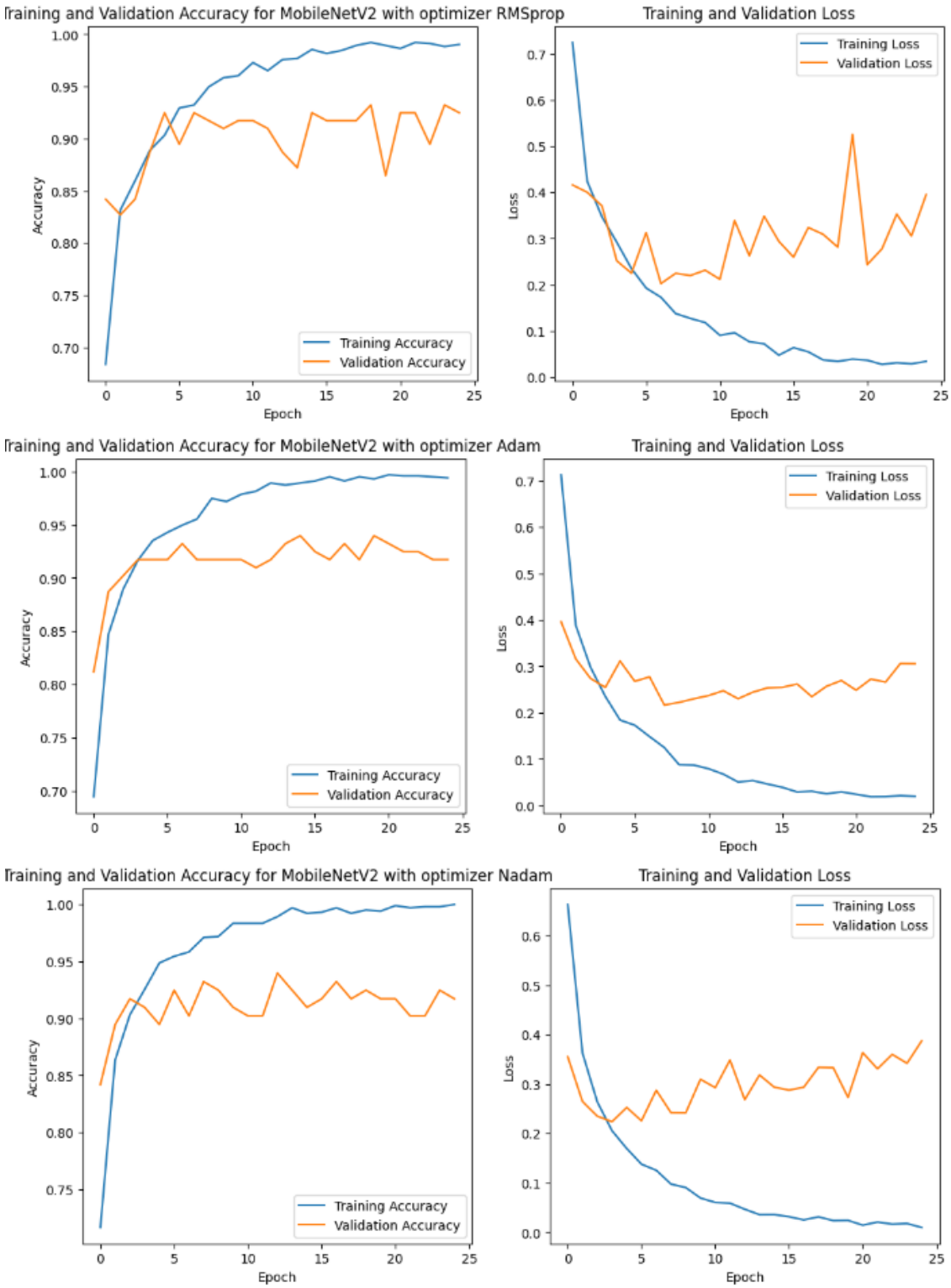
از مدل MobileNetV2 به عنوان مدل پایه استفاده کردیم و یک لایه Fully Connected با خروجی ۳ کلاس اضافه کردیم. مدل با استفاده از بهینه‌ساز Adam، نرخ یادگیری ۰.۰۰۱ و تابع هزینه categorical crossentropy کامپایل شد. با استفاده از Early Stopping این تکنیک برای جلوگیری از بیش‌برازش و بهبود عملکرد عمومی مدل مورد استفاده قرار گرفت. تنظیم patience برابر با ۵ به این معناست که اگر بهبود قابل‌توجهی در تابع هزینه اعتبارسنجی مشاهده نشود، آموزش متوقف می‌شود. ارزیابی مدل پس از اتمام آموزش، مدل با داده‌های ارزیابی تست شده و نتایج تابع هزینه و دقت نهایی ثبت می‌شود.

نمودار دقت: افزایش دقت در هر epoch نشان‌دهنده بهبود مدل است و این افزایش تا حدی ادامه می‌یابد که مدل به بیشینه دقت خود برسد.

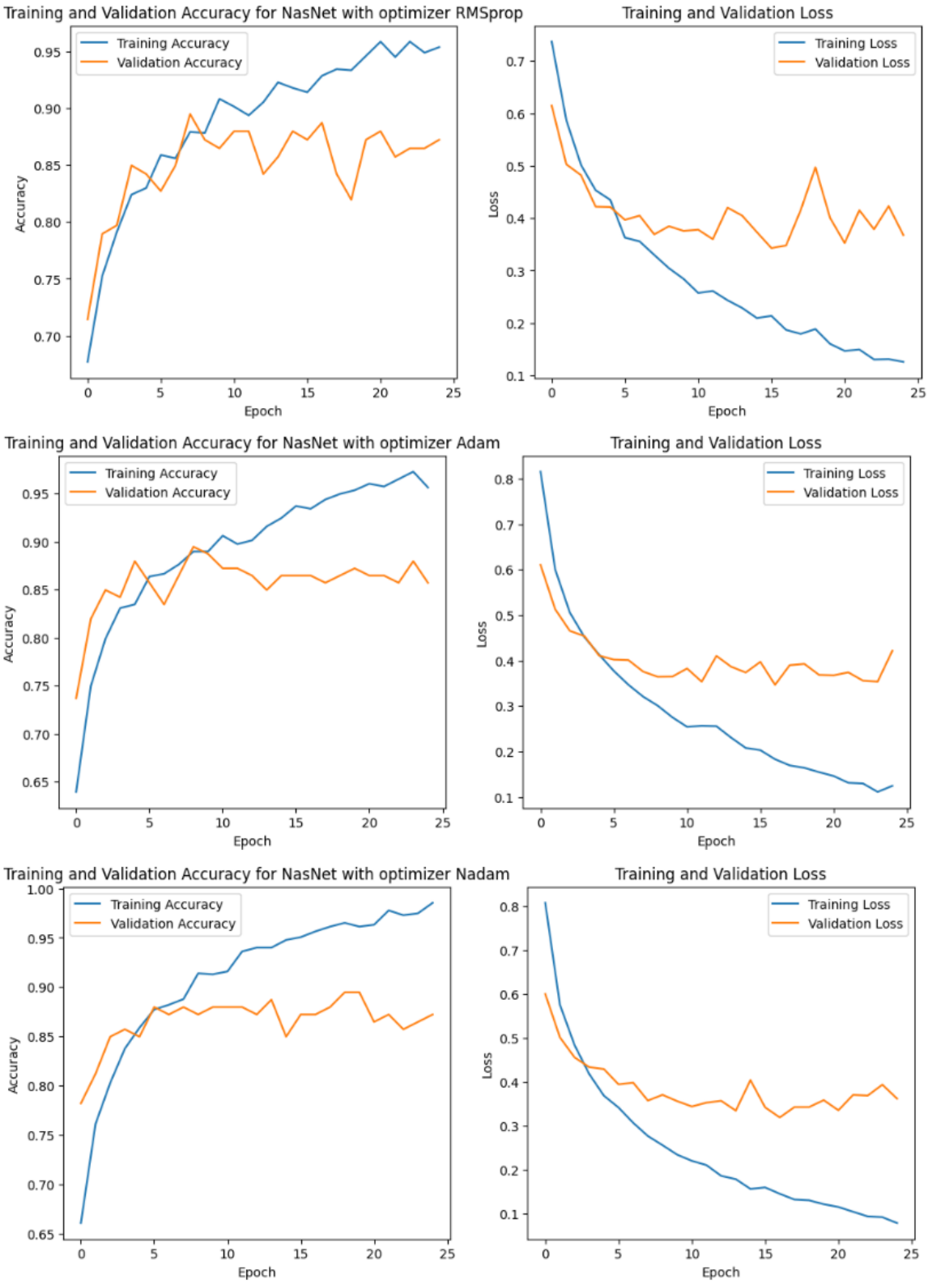
نمودار تابع هزینه :کاهش تابع هزینه نشان‌دهنده بهبود در پیش‌بینی مدل است. هرچه تابع هزینه کمتر باشد، مدل به جواب‌های بهتری نزدیک‌تر است.

اثر Early Stopping: این تکنیک با متوقف کردن آموزش در زمان مناسب باعث می‌شود مدل از بیش‌برازش جلوگیری کند و عملکرد آن در داده‌های اعتبارسنجی بهبود یابد. در این کد ما فقط یک نمونه از مدل با یک بهینه ساز را پیاده سازی کردیم و نتایج برای آموزش‌های بدون این تکنیک است.

این مراحل و تنظیمات باعث می‌شود که مدل نهایی به صورت بهینه‌تر آموزش ببیند و عملکرد بهتری در شناسایی بیماری‌های برگ لوبیا داشته باشد.



شکل ۶۴ - آموزش و ارزیابی و **loss** مدل **MobileNetV2** با هر سه بهینه ساز



شکل ۶۵ - آموزش و ارزیابی و **loss** با مدل **NasNet** و بهینه ساز **Nadam**

۲-۳. تحلیل نتایج

بعد از آموزش مدل، می‌توان از داده‌های ارزیابی برای بررسی عملکرد مدل استفاده کرد. در این مرحله، تصاویر نمونه‌ای از داده‌های ارزیابی همراه با برچسب‌های واقعی و پیش‌بینی‌شده توسط مدل نمایش داده می‌شوند. همچنین، مقادیر تابع هزینه و معیارهای ارزیابی مانند دقت گزارش شده و تحلیل کاملی از نتایج ارائه می‌شود. یه صورت تصادفی ۵ نمونه انتخاب شده و روی سه مدل با بهینه ساز Adam پیش آن نیز آمده است.

```
# Train With MobileNetV2 and Early Stopping Technique
base_model_one = MobileNetV2(input_shape=(224, 224, 3), include_top=False, weights='imagenet')
base_model_one.trainable = False
x = base_model_one.output
x = GlobalAveragePooling2D()(x)
x = Dropout(0.3)(x)
x = Dense(64, activation='relu')(x)
output = Dense(3, activation='softmax')(x)
model = Model(inputs=base_model_one.input, outputs=output)

model.compile(optimizer=Adam(learning_rate=0.001), loss='categorical_crossentropy', metrics=['accuracy'])

early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

history = model.fit(train_gen['NasNetMobile'], validation_data=valid_gen['NasNetMobile'], epochs=25, batch_size=32, callbacks=[early_stopping],

eval_results = model.evaluate(valid_gen['NasNetMobile'], batch_size=32)
print(f"Evaluation Loss: {eval_results[0]:.4f}")
print(f"Evaluation Accuracy: {eval_results[1]:.4f}")
```

شکل ۶۶ - کد آموزش با مدل **MobileNetV2** با تکنیک **Early Stopping**

جدول ۹ - درصدهای دقت آموزی و ارزیابی و **loss** در بهینه سازهای متفاوت بدون تکنیک متوقف کردن

Optimizer	CNN Model	Tr-Acc(%)	Val-Acc (%)	Tr-Loss	Val-loss
Adam	EfficientNetB6	67.31	59.55	0.5494	0.54.93
	MobileNetV2	98.42	91.73	0.201	0.3058
	NasNet	95.65	85.71	0.1239	0.4210
RMSProp	EfficientNetB6	56.09	59.55	0.5599	0.5494
	MobileNetV2	98.03	92.48	0.340	0.3956
	NasNet	95.36	87.22	0.1257	0.3678
Nadam	EfficientNetB6	65.18	60.90	0.5496	0.5493
	MobileNetV2	99.01	91.73	0.0097	0.3877
	NasNet	98.55	87.22	0.0792	0.3620

شکل ۶۷ نمونه عکس ها با لیبل پیش بینی با مدل **MobileNet**

تحلیل مدل‌ها

EfficientNetB6 عملکرد بهتری در دقت و تابع هزینه دارد که به دلیل معماری پیچیده‌تر و بلوک‌های SE (Squeeze-and-Excitation) است. MobileNetV2 به نسبت دقت بالایی را ارائه می‌کند و در شرایطی با منابع محدود مناسب است. NasNetMobile به دلیل پیچیدگی معماری و تنظیم خودکار بلوک‌ها، در شرایط خاص قابل استفاده است، اما دقت کمتری نسبت به EfficientNetB6 دارد.

مدل‌ها در دسته‌های ساده‌تر مانند برگ سالم عملکرد بهتری دارند. و در دسته‌های با شباهت زیاد (زنگ لوبیا و لکه زاویه‌دار) گاهی اوقات خطاهایی مشاهده می‌شود.