دانشگاه تهران
دانشکده مهندسی برق و
کامپیوتر

درس شبکه‌های عصبی و یادگیری عمیق

تمرین دوم

متین بذرافشان - 810100093
شهریار عطار - 810100186

فهرست

پرسش 1. تشخیص آلزایمر با استفاده از تصویر برداری مغزی ADNI[1]

1-1. معرفی مقاله

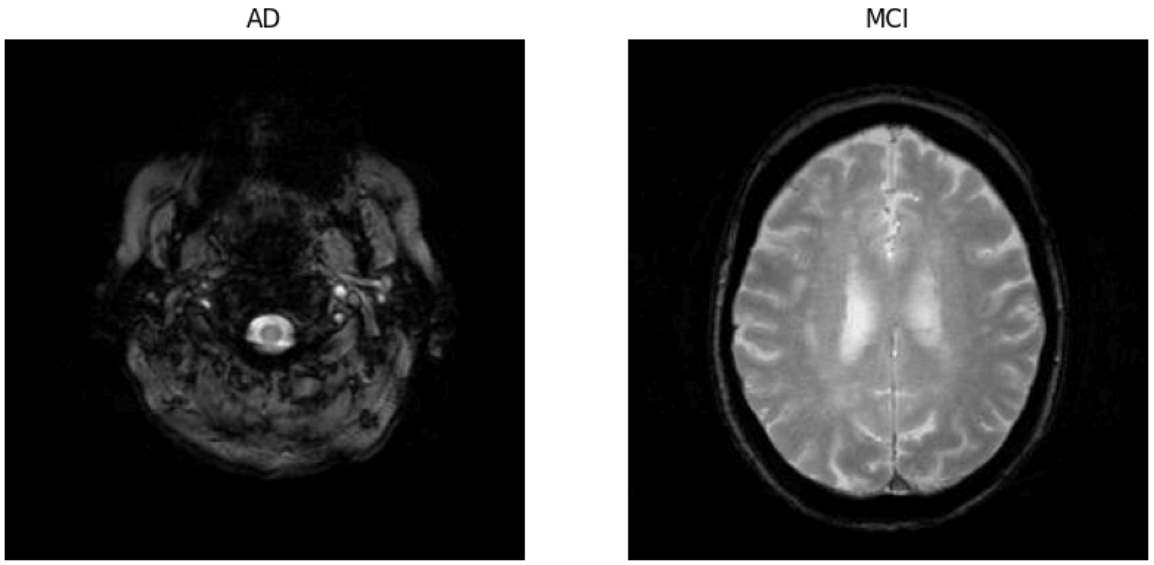
در این پرسش، یک دسته‌بند برای پیش‌بینی اینکه آیا یک بیمار دچار بیماری آلزایمر است یا خیر، ایجاد خواهیم کرد. از یک مجموعه داده استفاده خواهیم کرد که شامل تصاویر MRI مغز است. پیاده‌سازی بر اساس این مقاله است. در این مقاله سه معماری مختلف برای انجام این کار پیشنهاد شده که به بررسی این سه می‌پردازیم. بیماری آلزایمر (AD) یک اختلال عصبی‌زایی است که باعث از دست دادن حافظه و کاهش شناختی می‌شود. این بیماری شایع‌ترین علت آسیب مغزی است و حدود 60-70% از موارد آن را تشکیل می‌دهد. علت دقیق بروز آلزایمر نامشخص است، اما باور بر این است که ترکیبی از عوامل ژنتیکی، محیطی و سبک زندگی نقش دارند. در حال حاضر درمان کاملی برای آلزایمر وجود ندارد، اما شناسایی زودرس و درمان می‌تواند کمک کند تا پیشرفت بیماری کند. بنابراین، نیاز به یک روش دقیق و قابل اعتماد برای تشخیص آلزایمر بسیار حیاتی است.

[1] Alzheimer's Disease Neuroimaging Initiative

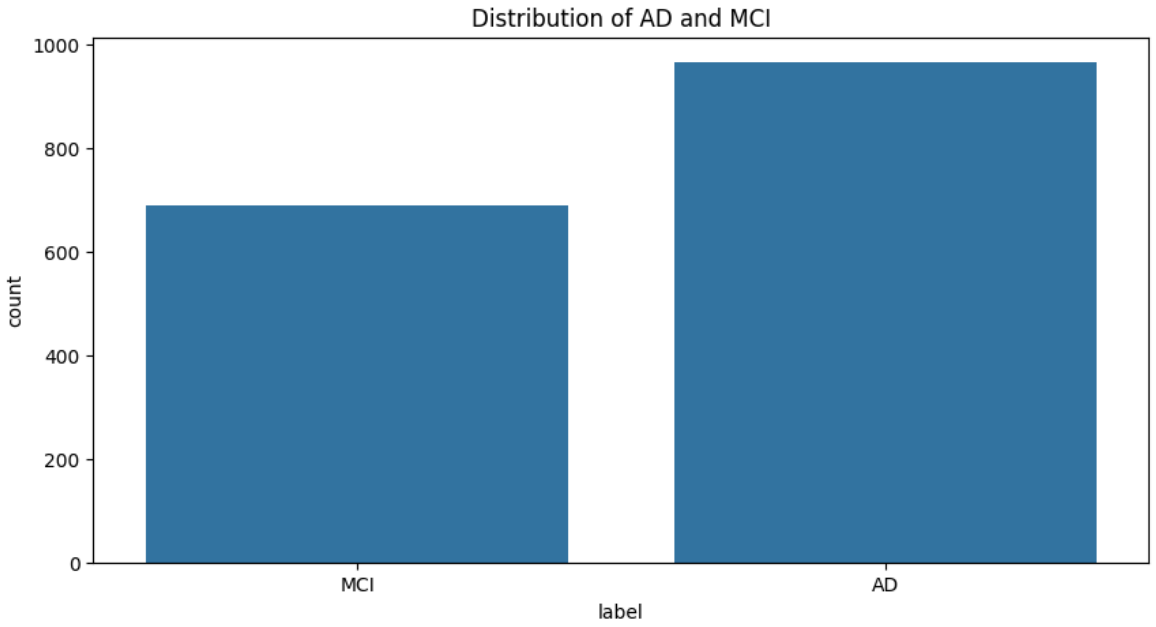
2-1. پیش‌پردازش

ابتدا باید بر روی دیتاست پیش‌پردازش انجام دهیم تا شبکه سریع‌تر همگرا شود. مجموعه‌داده استفاده شده در اینجا مجموعه‌داده ADNI از سایت Kaggle است. این مجموعه داده شامل تصاویر MRI از مغز بیماران مبتلا به بیماری آلزایمر و کنترل‌های سالم است. این مجموعه داده به دو کلاس تقسیم شده است: نقص شناختی ملایم (MCI) و بیماری آلزایمر (AD). این مجموعه داده شامل مجموعا 1654 تصویر است، که شامل 965 تصویر از بیماران آلزایمر و 689 تصویر از افراد مبتلا به نقص شناختی ملایم است. برای پیش‌پردازش، زیرا بیشتر امور از پیش انجام شده‌اند، فقط نیاز به تغییر اندازه تصاویر به یک اندازه ثابت داریم.

```
def _load_image(self, path: str) -> np.ndarray:
      img = cv2.imread(path)
      img = cv2.resize(img, self.img_size)
      img = img / 255.0
      return img

      def _load(self):
      MCI_files = glob(os.path.join(self.MCI_path, "*.jpg"))
      AD_files = glob(os.path.join(self.AD_path, "*.jpg"))

      MCI_data = []
      for file in tqdm.tqdm(MCI_files, desc="Loading MCI data"):
            img = self._load_image(file)
            MCI_data.append(img)

      AD_data = []
      for file in tqdm.tqdm(AD_files, desc="Loading AD data"):
            img = self._load_image(file)
            AD_data.append(img)

      self.MCI_data = MCI_data
      self.AD_data = AD_data
```



شکل 1-1. تصویر نمونه‌ای از دو کلاس موجود در دیتاست

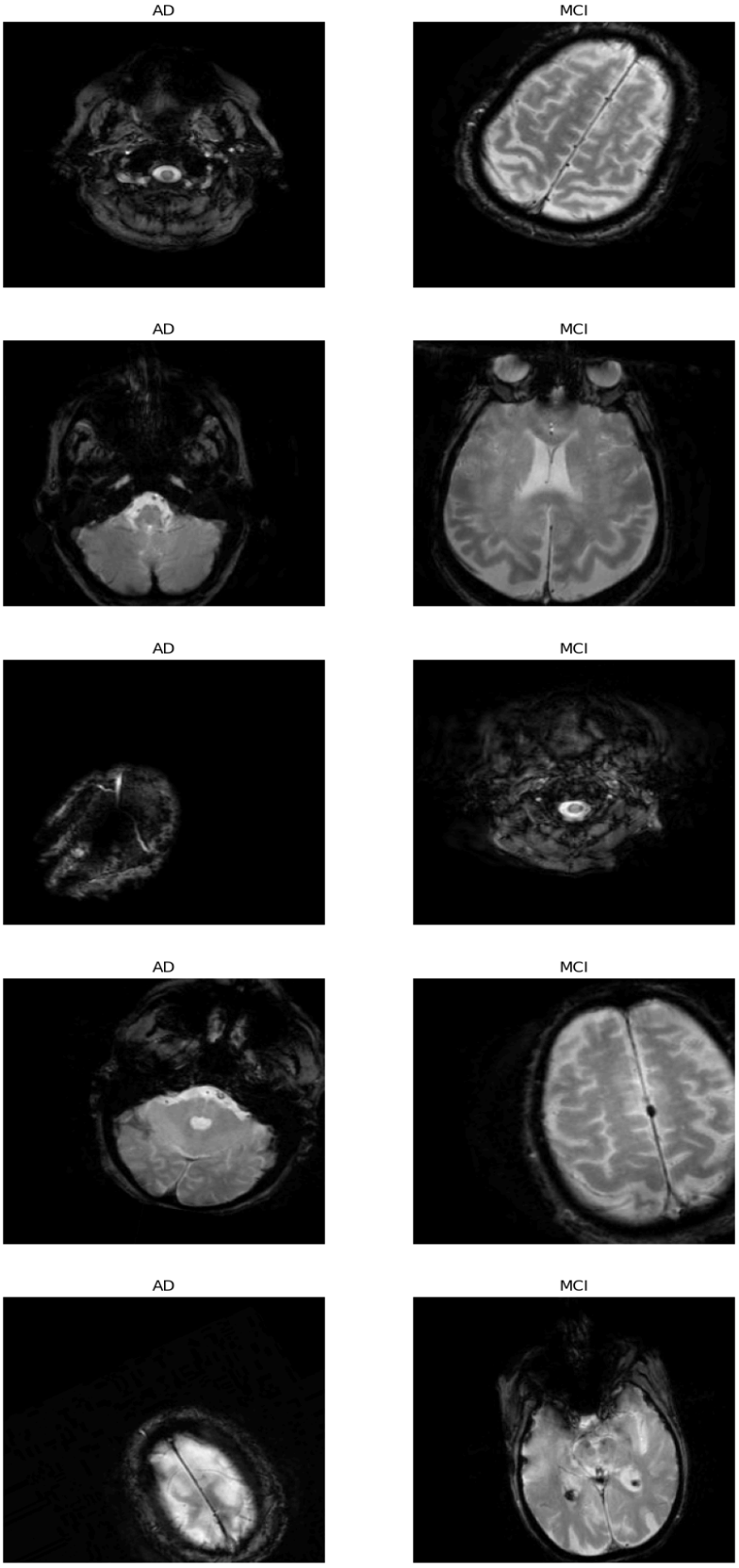


شکل 1-2. نمودار توزیع از دو کلاس موجود در دیتاست

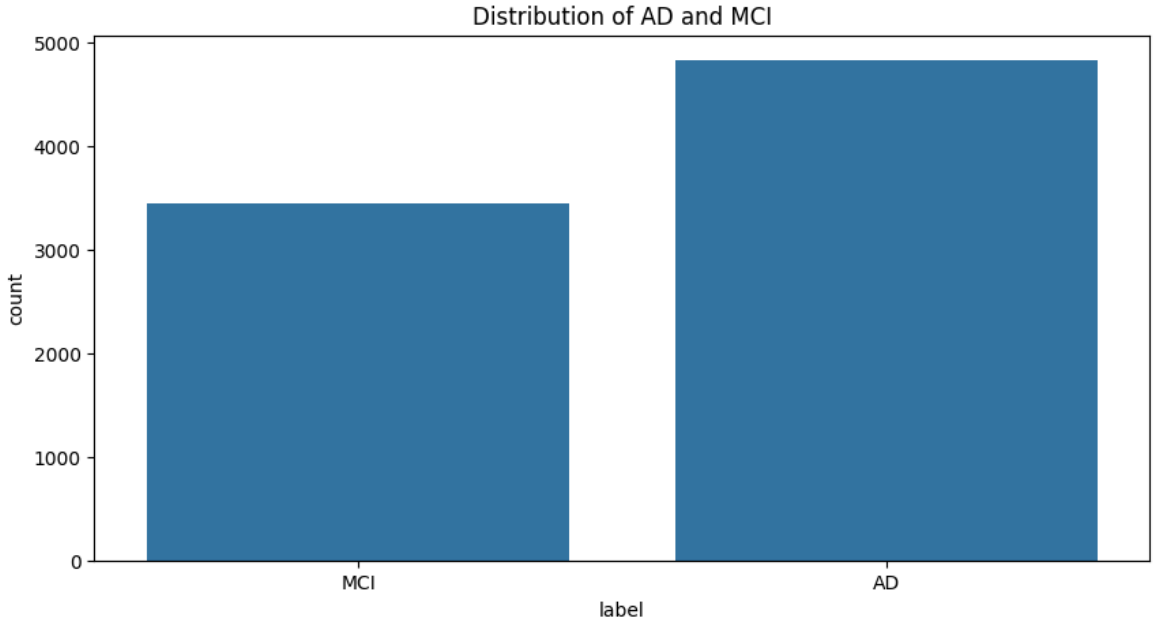
3-1. داده‌افزایی (Data Augmentation)

تکنیک افزایش داده (Data augmentation) یک روش است تا اندازه مجموعه داده‌ی آموزشی را به صورت مصنوعی افزایش داد، و این کار با اعمال تبدیلات تصادفی به تصاویر انجام می‌شود. این کار به بهبود تعمیم مدل و کاهش بیش‌برازش (overfitting) کمک می‌کند. ما تغییرات تصادفی از جمله چرخش‌ها (rotation)، برگرداندن‌ها (flip)، جابجایی‌ها (shift) و بزرگنمایی‌ها (zoom) را به تصاویر اعمال خواهیم کرد. از آنجایی که می‌خواهیم تعداد تصویرها پنج برابر شود به ازای هر تصویر، چهار تصویر جدید ایجاد می‌کنیم. برای اینکه توزیع کلاس‌ها حفظ شود به شکل زیر عمل می‌کنیم

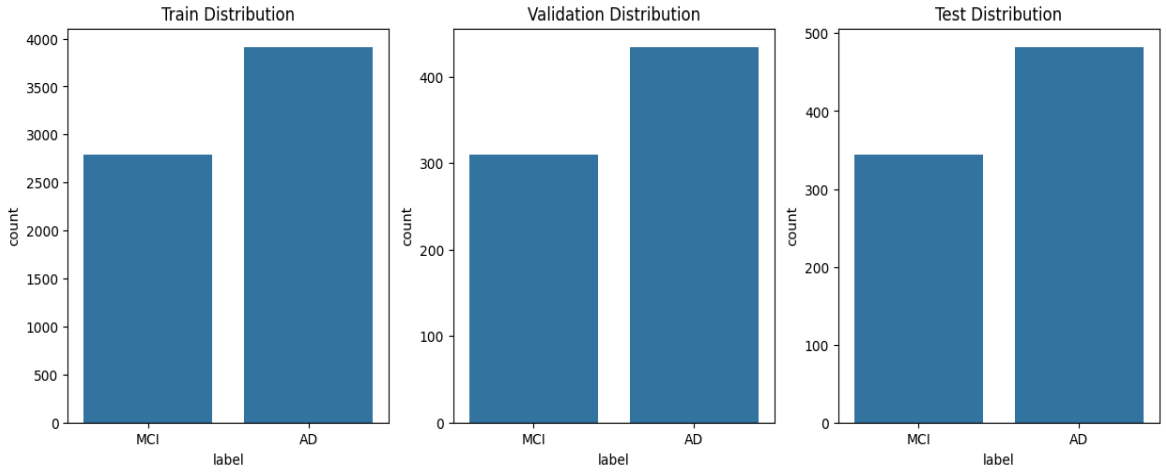
```
def augment_image(self, img: np.ndarray, datagen:
ImageDataGenerator, num: int = 4):
      augmented_img = datagen.flow(img[np.newaxis, :, :, :],
batch_size=num)
      augmented_images = [next(augmented_img)[0] for _ in
range(num)]
      all_img = [img] + augmented_images
      return all_img

def augment_dataset(self, num_samples: int = 4):
      datagen = ImageDataGenerator(
            horizontal_flip=True,
            shear_range=0.1,
            zoom_range=0.1,
            rotation_range=20,
            width_shift_range=0.1,
            height_shift_range=0.1
      )
      AD_data = []
      for img in tqdm.tqdm(self.AD_data):
            augmented_imgs = self.augment_image(img, datagen,
num_samples)
            AD_data.extend(augmented_imgs)
      MCI_data = []
      for img in tqdm.tqdm(self.MCI_data):
            augmented_imgs = self.augment_image(img, datagen,
num_samples)
            MCI_data.extend(augmented_imgs)
```



شکل 1-3. تصویرهای نمونه‌ای از دو کلاس موجود در دیتاست پس از افزایش داده



شکل 1-4. نمودار توزیع از دو کلاس موجود در دیتاست پس از افزایش داده



شکل 1-5. نمودار توزیع از دو کلاس موجود در هر یک از دیتاست پس از افزایش داده

همانگونه که مشاهده می‌شود توزیع کلاس‌ها پس از داده‌افزایی و تقسیم به دیتاست آموزش و آزمایش با توزیع اولیه کلاس‌ها یکسان است.

4-1. پیاده‌سازی

در ابتدا کلیتی از ساختار گفته می‌شود بعد سه مدل پیشنهادی توضیح داده می‌شود.

Glorot Initialization

روش Glorot Initialization یا Xavier Initialization یک روش موثر برای مقداردهی اولیه وزن‌های یک شبکه عصبی است. این روش توسط Xavier Glorot و Yoshua Bengio در سال 2010 معرفی شد. ایده اصلی این است که وزن‌ها باید به گونه‌ای مقداردهی شوند که واریانس ورودی و خروجی هر لایه از شبکه تقریباً یکسان باشد. این کمک می‌کند تا در فرآیند آموزش، گرادیان‌ها نه خیلی بزرگ و نه خیلی کوچک شوند که ممکن است منجر به مشکلاتی مانند گرادیان‌های ناپدید شونده یا بزرگ شونده شود.

با استفاده از این فرمول، مقادیر وزن‌ها به گونه‌ای مقداردهی می‌شوند که واریانس ورودی و خروجی لایه‌ها تقریباً یکسان باشد. این اتفاق باعث می‌شود که گرادیان‌ها در فرآیند آموزش مناسب و بهینه باشند و به سرعت به نقاط بیشینه یا کمینه محلی همگرا شوند.

مزایای این روش عبارتند از:

- پیشگیری از مشکل گرادیان‌های ناپدید شونده یا بزرگ شونده.
- افزایش سرعت آموزش شبکه‌های عصبی.
- بهبود عملکرد شبکه در مسائل دشوارتر.

فرمول این روش به شکل زیر می‌باشد:

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right]$$

Activation Function

ما در لایه‌های پنهان شبکه از تابع فعال‌سازی ReLU استفاده خواهیم کرد. تابع ReLU یک تابع فعال‌سازی غیرخطی است که غیرخطیت را به شبکه معرفی می‌کند و کمک می‌کند تا الگوهای پیچیده‌تری در داده‌ها یاد گرفته شود. توابع فعال‌سازی tanh و sigmoid نیز در شبکه استفاده می‌شوند. برای لایه خروجی از تابع فعال‌سازی softmax استفاده خواهیم کرد.

فرمول‌های هرکدام از توابع فعال‌سازی به شرح زیر است:

$$ReLU(x) = max(0, x)$$

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$

$$softmax(x) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Loss Function

تابع هزینه خطی دسته‌ای برای آموزش مدل استفاده شده. تابع هزینه خطی دسته‌ای برای مسائل دسته‌بندی چندکلاسه استفاده می‌شود و اختلاف بین توزیع احتمال پیش‌بینی شده و توزیع واقعی احتمال کلاس‌ها را اندازه‌گیری می‌کند. فرمول تابع هزینه خطی دسته‌ای به شرح زیر است:

$$L(y, \hat{y}) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{C} (y_i = j) log(\hat{y}_i)$$

یک تابع هزینه دیگر ممکن است استفاده شود تابع هزینه منفی لگاریتمی باشد. تابع هزینه منفی لگاریتمی برای مسائل دسته‌بندی چندکلاسه استفاده می‌شود و اختلاف بین توزیع احتمال پیش‌بینی شده و توزیع واقعی احتمال کلاس‌ها را اندازه‌گیری می‌کند. فرمول تابع هزینه منفی لگاریتمی به شرح زیر است:

$$L(y, \hat{y}) = -\frac{1}{N} \sum_{i} log(\hat{y}_i)$$

دلیل انتخاب تابع هزینه خطی دسته‌ای به دلیل مشکل گرادیان ناپدیدشونده است که استفاده از این روش به همراه بهینه‌ساز `Adam` باعث تسریع همگرایی شبکه عصبی می‌شود.

Optimizer

بهینه‌ساز `Adam` یک الگوریتم بهینه‌سازی نرخ یادگیری تطبیقی است که مزایای دو الگوریتم بهینه‌سازی محبوب دیگر، یعنی `AdaGrad` و `RMSProp` را ترکیب می‌کند. بهینه‌ساز `Adam` نرخ یادگیری را برای هر پارامتر بر اساس اولین و دومین مومنت گرادیان‌ها تطبیق می‌دهد. این کمک می‌کند تا همگرایی مدل بهبود یابد و عملکرد آن بهبود یابد.

Regularization

رگولاریزاسیون L2 یک عبارت جریمه را به تابع هزینه اضافه می‌کند که وزن‌های بزرگ را مجازات می‌کند. این کمک می‌کند تا مدل از بیش‌برازش در داده‌های آموزش جلوگیری کند و عملکرد تعمیمی آن بهبود یابد. عبارت رگولاریزاسیون `L2` به صورت زیر است:

$$L2 = \lambda \sum_{i} w_i^2$$

Dropout

ما از رگولاریزاسیون `dropout` برای جلوگیری از بیش‌برازش در مدل استفاده خواهیم کرد. این تکنیک در طول آموزش تعدادی از واحدهای ورودی را به صورت تصادفی صفر می‌کند. این کمک می‌کند تا مدل بیش‌ازحد به هر واحد ورودی تکیه نکند و عملکرد تعمیمی آن بهبود یابد. ما `dropout` را به لایه‌های پنهان شبکه اعمال خواهیم کرد.

Batch Normalization

نرمال‌سازی دسته‌ای یک تکنیک است که ورودی به هر لایه از شبکه را به میانگین صفر و انحراف معیار یک نرمال می‌کند. این کمک می‌کند تا فرآیند آموزش پایدار شود و سرعت همگرایی آن افزایش یابد. ما نرمال‌سازی دسته‌ای را به لایه‌های پنهان شبکه اعمال خواهیم کرد.

معماری مدل

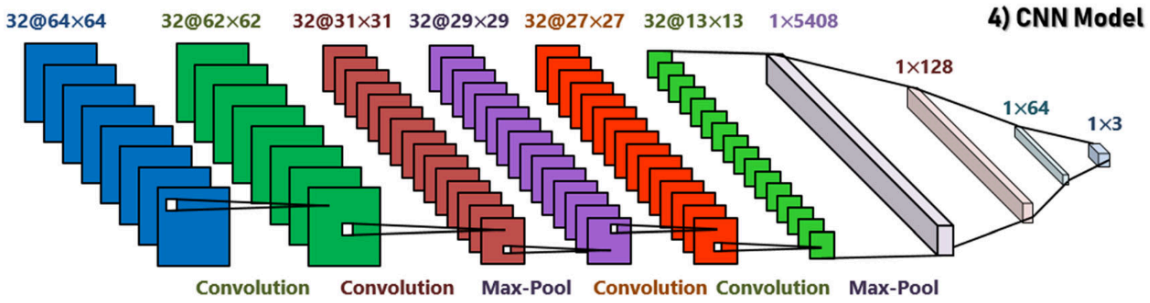
معماری مدل شامل یک سری لایه‌های پیچشی (Convolution) دنبال شده از لایه‌های حداکثرگیری (MaxPooling) است. لایه‌های پیچشی ویژگی‌ها را از تصاویر ورودی استخراج می‌کنند، در حالی که لایه‌های حداکثرگیری ابعاد فضایی را کاهش می‌دهند. خروجی لایه‌های پیچشی به صورت فلت شده و از طریق یک سری لایه‌های کاملا متصل (Fully Connected) برای انجام پیش‌بینی نهایی ارسال می‌شود. مدل از توابع فعال‌سازی `ReLU` در لایه‌های پنهان و تابع فعال‌سازی `softmax` در لایه خروجی استفاده می‌کند. مدل با استفاده از تابع هزینه خطی دسته‌ای و بهینه‌ساز Adam آموزش داده می‌شود. در کد زیر معماری مدل را می‌توانید مشاهده کنید:

```
proposed_model = Sequential([
      Input(shape=(256, 256, 3)),
      Conv2D(
      32,
      (3, 3),
      strides=(1, 1),
      padding="same",
      kernel_initializer=initializer,
      kernel_regularizer=regularizer,
      ),
```

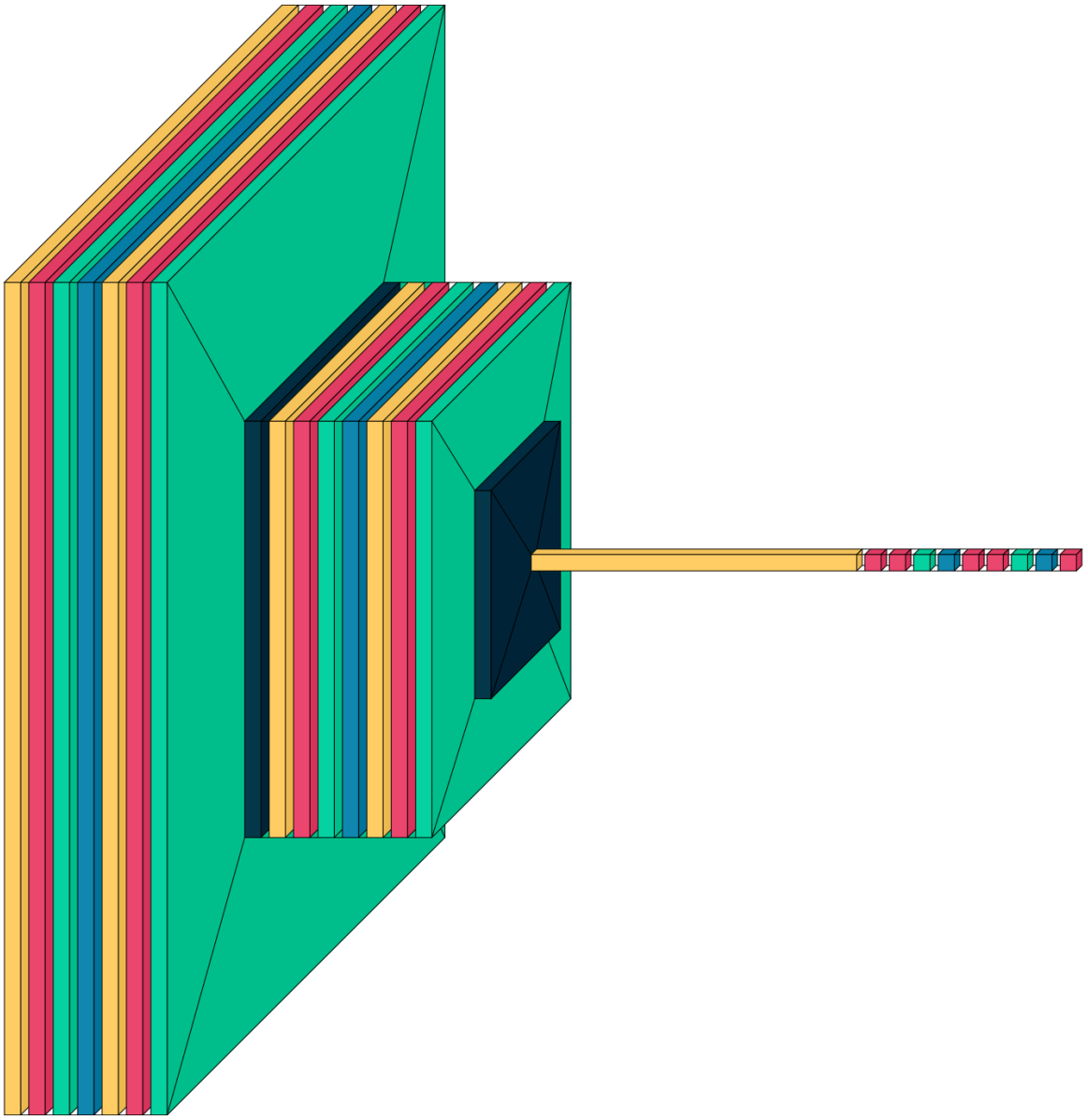
```
    BatchNormalization(),
    ReLU(),
    Dropout(0.5),
    Conv2D(
    32,
    (3, 3),
    strides=(1, 1),
    padding="same",
    kernel_initializer=initializer,
    kernel_regularizer=regularizer,
    ),
    BatchNormalization(),
    ReLU(),
    MaxPooling2D((2, 2)),
    Conv2D(
    32,
    (3, 3),
    strides=(1, 1),
    padding="same",
    kernel_initializer=initializer,
    kernel_regularizer=regularizer,
    ),
    BatchNormalization(),
    ReLU(),
    Dropout(0.5),
    Conv2D(
    32,
    (3, 3),
    strides=(1, 1),
    padding="same",
    kernel_initializer=initializer,
    kernel_regularizer=regularizer,
    ),
    BatchNormalization(),
    ReLU(),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(
    128,
    activation="relu",
    kernel_initializer=initializer,
    kernel_regularizer=regularizer,
```

```
        ),
        BatchNormalization(),
        ReLU(),
        Dropout(0.5),
        Dense(
        64,
        activation="relu",
        kernel_initializer=initializer,
        kernel_regularizer=regularizer,
        ),
        BatchNormalization(),
        ReLU(),
        Dropout(0.5),
        Dense(
        2,
        activation="softmax",
        kernel_initializer=initializer,
        kernel_regularizer=regularizer,
        ),
])

proposed_model.compile(optimizer=Adam(learning_rate=.1),
loss=categorical_crossentropy, metrics=["accuracy"])
```

```
Total params: 16,815,650
Trainable params: 16,815,010
Non-trainable params: 640
```



شکل 1-6. ساختار مدل پیشنهادی



شکل 1-7. ساختار مدل نوشته شده در کد، ساخته شده توسط visualkeras

همچنین دو مدل دیگر در مقاله پیشنهاد شده بود که با پارامترهای متفاوت با این مدل مقایسه شوند و همچنین عملکرد مدل پیشنهادی نیز با حالت‌های مختلف (batch_size و train_test_split_size و یا بقیه پارامترها) امتحان شود و مقایسه شود. در شکل زیر خلاصه معماری این سه مدل را می‌توانید مشاهده کنید.

Layer	Shape	Filters
Proposed model		
CONV	64 × 64	32 (3 × 3)
CONV	64 × 64	32 (3 × 3)
MAX_POOL		32 (2 × 2)
CONV	64 × 64	32 (3 × 3)
CONV	64 × 64	32 (3 × 3)
MAX_POOL		32 (2 × 2)
FLATTEN		
DENSE	128	
DENSE	64	
DENSE	2	
Testing model 1		
CONV	64 × 64	32 (3 × 3)
MAX_POOL		32 (2 × 2)
CONV	64 × 64	32 (3 × 3)
MAX_POOL		32 (2 × 2)
FLATTEN		
DENSE	128	
DENSE	2	
Testing model 2		
CONV	64 × 64	32 (3 × 3)
CONV	64 × 64	32 (3 × 3)
MAX_POOL		32 (2 × 2)
FLATTEN		
DENSE	128	
DENSE	64	
DENSE	2	

شکل 1-8. خلاصه معماری سه مدل پیشنهاد شده در مقاله

5-1. ابزار تحلیل نتایج

ما مدل را روی مجموعه داده آموزشی آموزش می‌دهیم و عملکرد آن را بر روی مجموعه داده آزمون ارزیابی می‌کنیم. ما از پارامترهای Recall ،Precision، Accuracy و امتیاز F1 برای ارزیابی مدل استفاده خواهیم کرد. همچنین با Confusion Matrix را برای تجسم عملکرد مدل نیز رسم خواهیم کرد. منحنی ROC و امتیاز AUC نیز برای ارزیابی عملکرد مدل محاسبه خواهد شد. فرمول‌های به شرح زیر است:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

	Predicted Negative	Predicted Positive
Actual Negative (True Negative)	TN	FP
Actual Positive (True Positive)	FN	TP

منحنی ROC:

منحنی ROC یک نمودار است که نرخ مثبت‌های واقعی (True Positive Rate) را در مقابل نرخ مثبت‌های اشتباه (False Positive Rate) (معکوس خصوصیت) به تفکیک مختلفی از آستانه‌های تصمیم مدل نشان می‌دهد. به طور کلی، منحنی ROC به ما اطلاعاتی در مورد توازن بین نرخ درستی و نرخ اشتباهی مدل در همه آستانه‌های تصمیم ارائه می‌دهد.

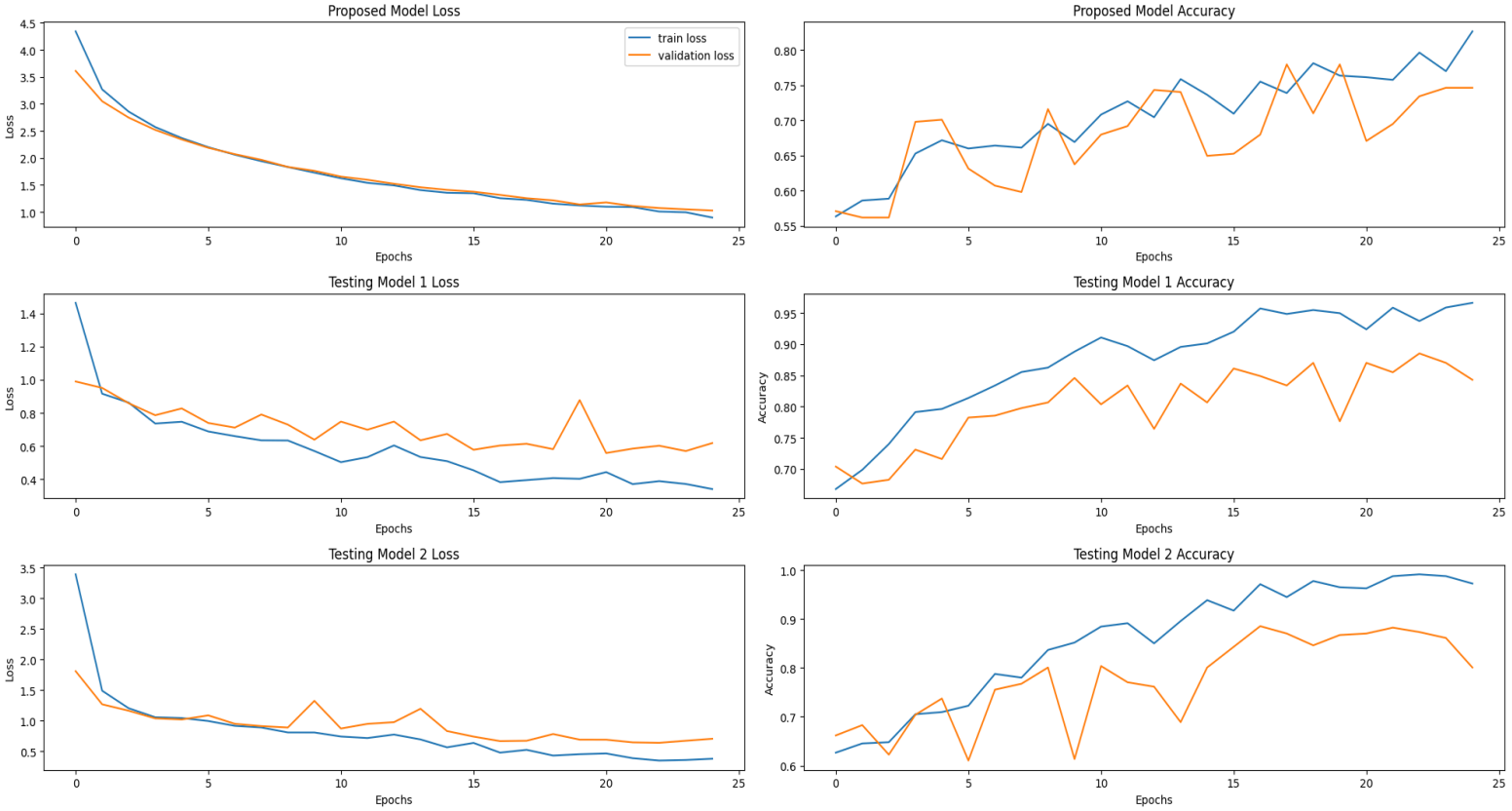
امتیاز AUC:

امتیاز AUC مساحت زیر منحنی ROC است و یک معیار خوب از کیفیت و عملکرد یک مدل طبقه‌بندی است. اگر امتیاز AUC برابر با 1 باشد، به این معنی است که مدل توانایی کاملی در تمایز بین دو کلاس دارد، در حالی که اگر امتیاز AUC برابر با 0.5 باشد، مدل هیچ توانایی در تمایز بین کلاس‌ها ندارد و عملکرد آن معادل با یک تصادفی است. پس امتیاز AUC به ما ایده‌ای از عملکرد مدل در تمام آستانه‌های تصمیم می‌دهد. به عبارت دیگر، این امتیاز نمایانگر میزان توانایی مدل در جداسازی نمونه‌های مثبت و منفی است، بدون درنظر گرفتن آستانه‌های خاصی که ممکن است برای تصمیم‌گیری استفاده شود.

1-6. مقایسه نتایج

مقایسه معماری‌ها

نمودارهای loss و accuracy:



شکل 1-9. خلاصه عملکرد معماری سه مدل پیشنهاد شده در مقاله

	Train		Test	
model	accuracy	loss	accuracy	loss
proposed	98.75%	0.0896	89%	0.0901
testing_1	96.63%	0.3414	84%	0.6183
testing_2	97.23%	0.3760	80%	0.7029

جدول 1-1. خلاصه نتیجه معماری سه مدل پیشنهاد شده در مقاله

همانگونه که دیده می‌شود مدل اولیه دقت و تعمیم‌پذیری بهتری نسبت به دو مدل پیشنهادی دیگر دارد که دلیل آن وجود لایه‌های کانولوشنال زیاد برای استخراج اطلاعات از عکس می‌باشد.

Classification Reports:

proposed model:

	precision	recall	f1-score	support
0	0.81	0.93	0.87	126
1	0.95	0.87	0.91	205
accuracy			0.89	331
macro avg	0.88	0.90	0.89	331
weighted avg	0.90	0.89	0.89	331

testing model 1:

	precision	recall	f1-score	support
0	0.88	0.73	0.80	142
1	0.82	0.93	0.87	189
accuracy			0.84	331
macro avg	0.85	0.83	0.84	331
weighted avg	0.85	0.84	0.84	331

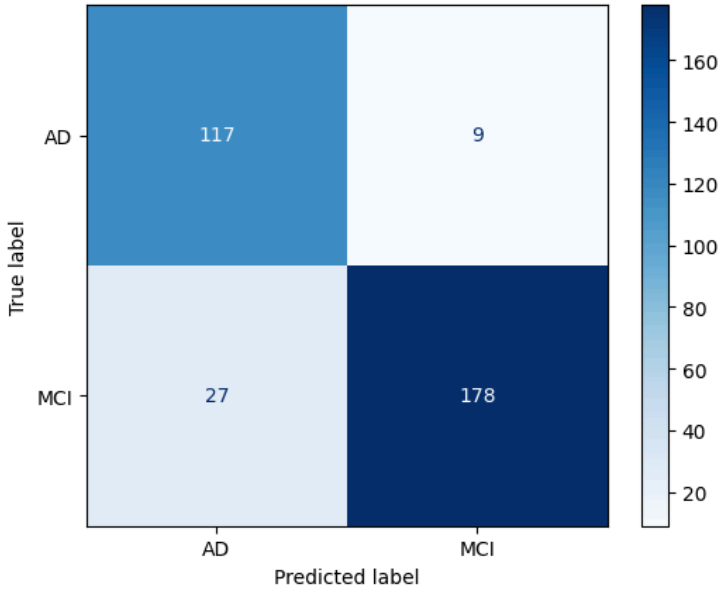
testing model 2:

	precision	recall	f1-score	support
0	0.72	0.88	0.79	142
1	0.89	0.74	0.81	189
accuracy			0.80	331
macro avg	0.81	0.81	0.80	331
weighted avg	0.82	0.80	0.80	331

همانگونه که بالاتر اشاره شد مدل اولیه دقت و عملکرد بهتری دارد هر چند دو مدل دیگر نیز عملکرد قابل قبول دارند. مدل اول دارای precision و recall و در نتیجه f1 بالایی می‌باشد و عملکرد بسیار مناسبی دارد و در تعداد کمی داده با خطا مواجه می‌شود.

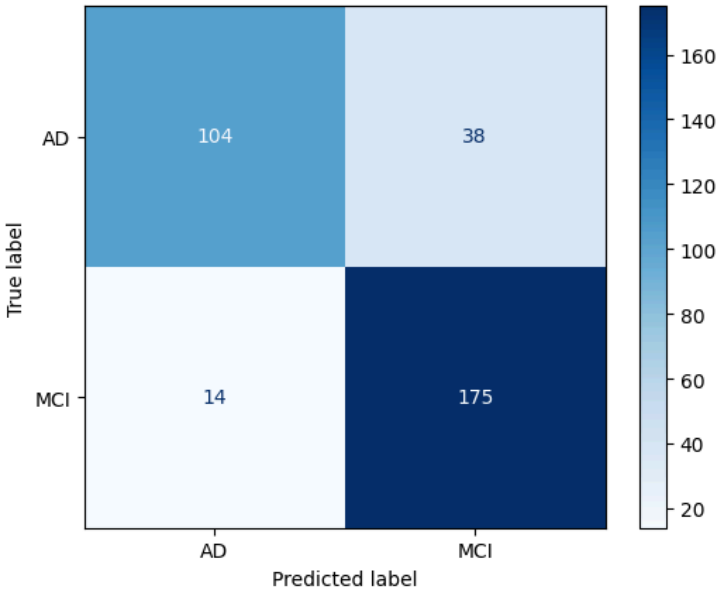
Confustion Matrixs:

proposed model:



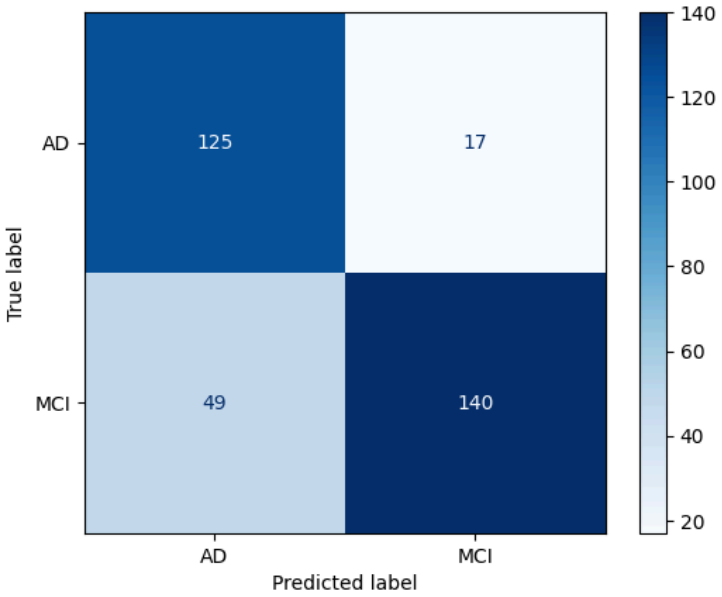
شکل 1-10. `confusion matrix` برای `proposed model`

testing model 1:



شکل 1-11. `confusion matrix` برای `testing model 1`

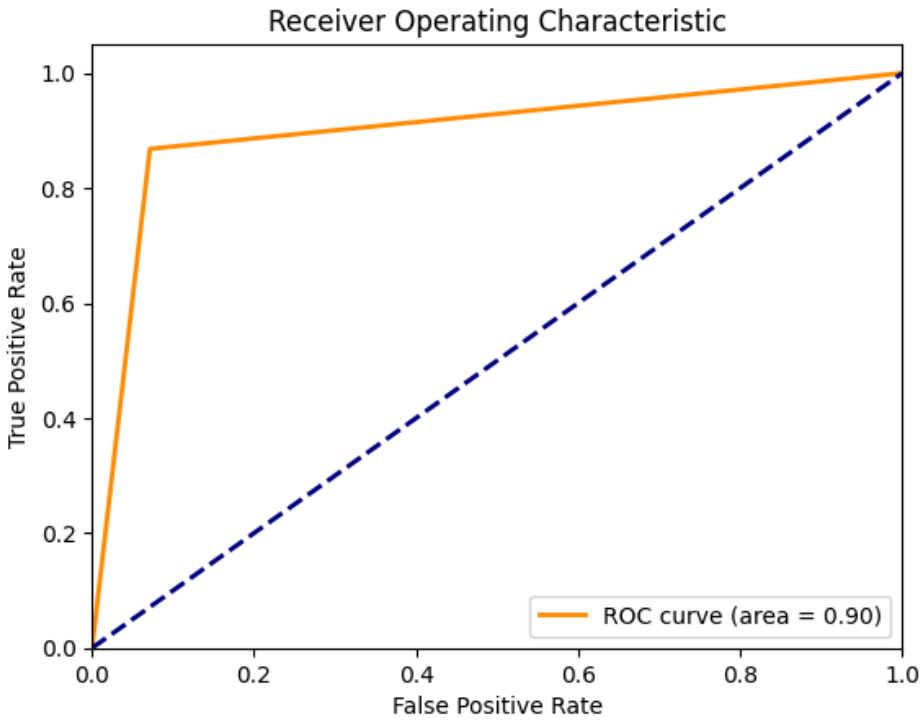
testing model 2:



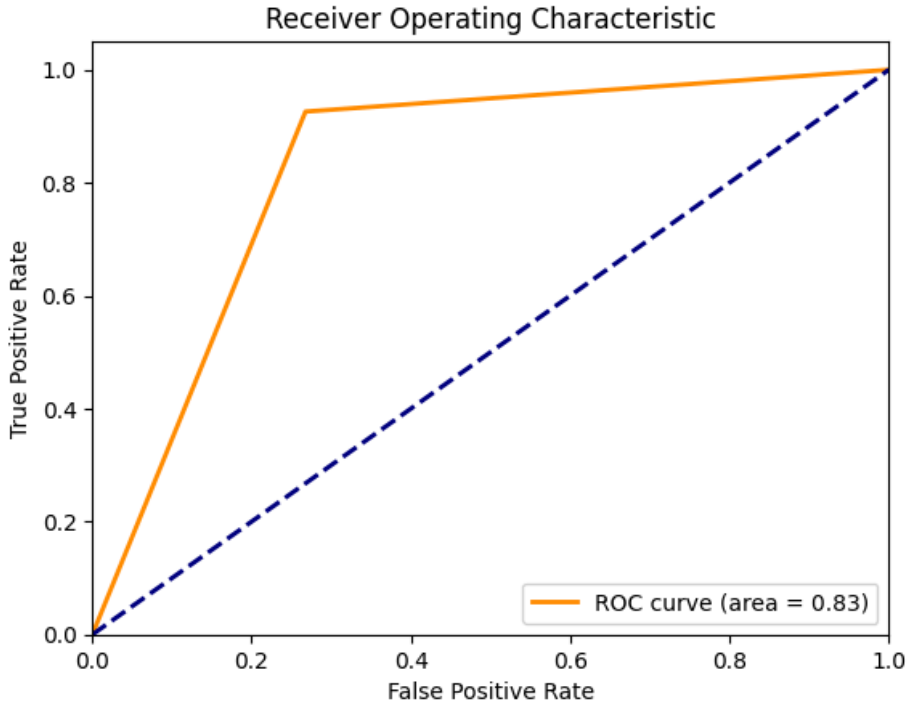
شکل 1-12. confusion matrix برای testing model 2

همانگونه که دیده می‌شود مدل testing model 1 در شناسایی AD دارایی مشکل است و بسیاری از عکس‌های اینگونه را به اشتباه تشخیص می‌دهد (که خطای بسیار بدی محسوب می‌شود و می‌تواند باعث مرگ یک بیمار شود) ولی مدل testing model 2 به اشتباه عکس‌هایی که دارای بیماری را نیستند را دارای بیماری تشخیص می‌دهد. همچنین در مدل پیشنهادی مقدار recall برای کلاس AD بسیار بالا می‌باشد که برای تشخیص پزشکی اهمیت بالایی دارد. در مجموع مدل پیشنهادی در همه پارامترها (به خصوص accuracy) از بقیه مدل‌ها بهتر عمل می‌کند و گزینه مورد اعتمادتری می‌باشد.

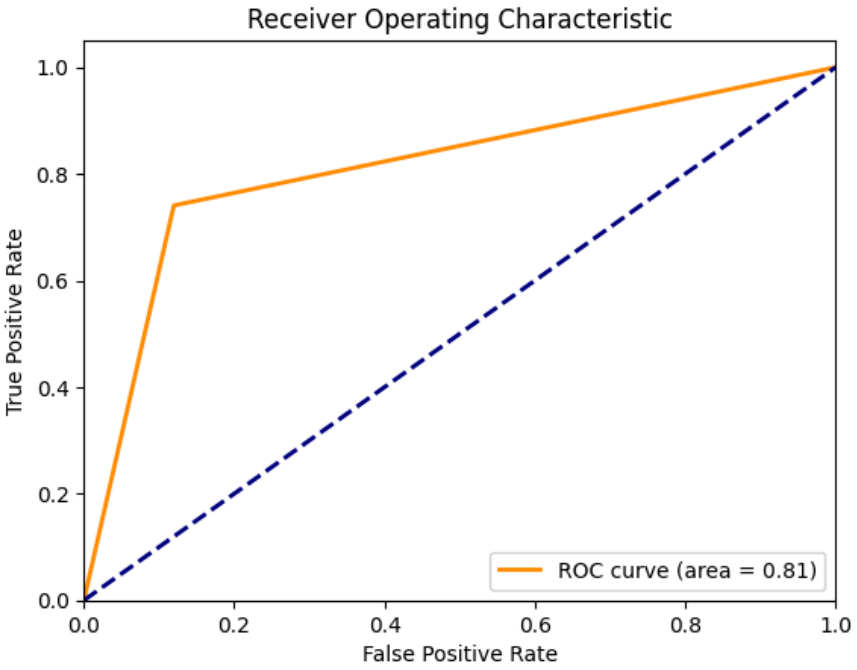
ROC:



شکل 1-13. نمودار ROC برای `proposed model`



شکل 1-14. نمودار ROC برای `testing model 1`



شکل 1-15. نمودار ROC برای `testing model 2`

همانگونه که انتظار داشتیم مساحت زیر نمودار ROC (مقدار AUC) برای مدل پیشنهادی از دو مدل دیگر با اختلاف بهتر است و دو مدل دیگر تقریبا عملکرد نزدیکی دارند.

مقایسه نسبت توزیع داده آموزش و آزمایش

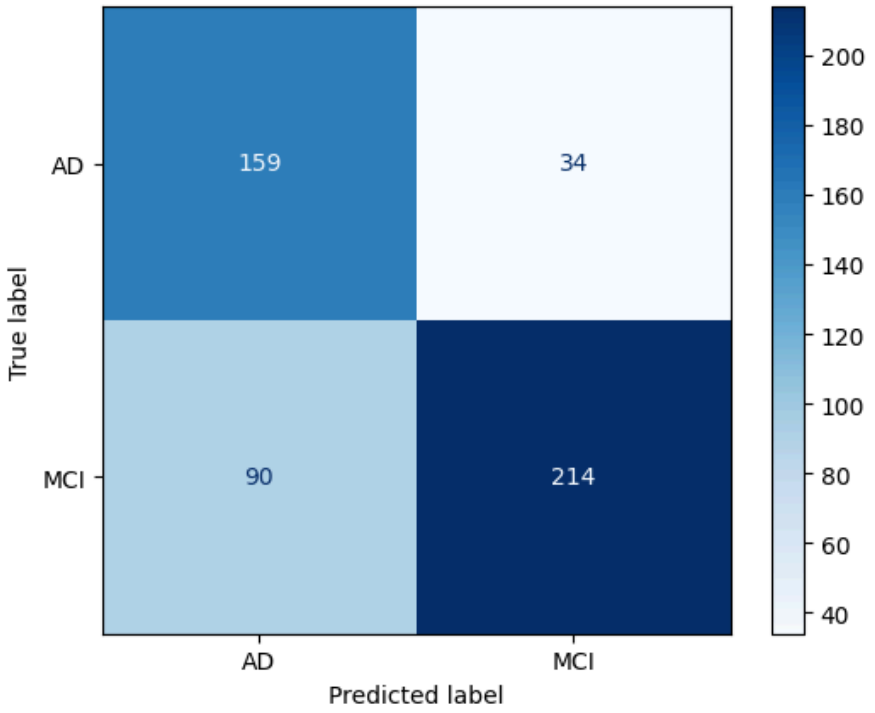
حال به بررسی اثر پارامترهای مختلف بر روی عملکرد شبکه `proposed_model` که پیشتر توضیح داده‌شده می‌پردازیم. ابتدا اثر استفاده از نسبت متفاوت برای تقسیم‌بندی را بررسی می‌کنیم

```
dataset.split_train_test_validation(split_size=.3,
validation=True)

proposed_model_cnn.train(dataset.train_gen, dataset.val_gen,
epochs=5)

proposed_model_cnn.analyze(dataset.test_data,
dataset.test_label)
```

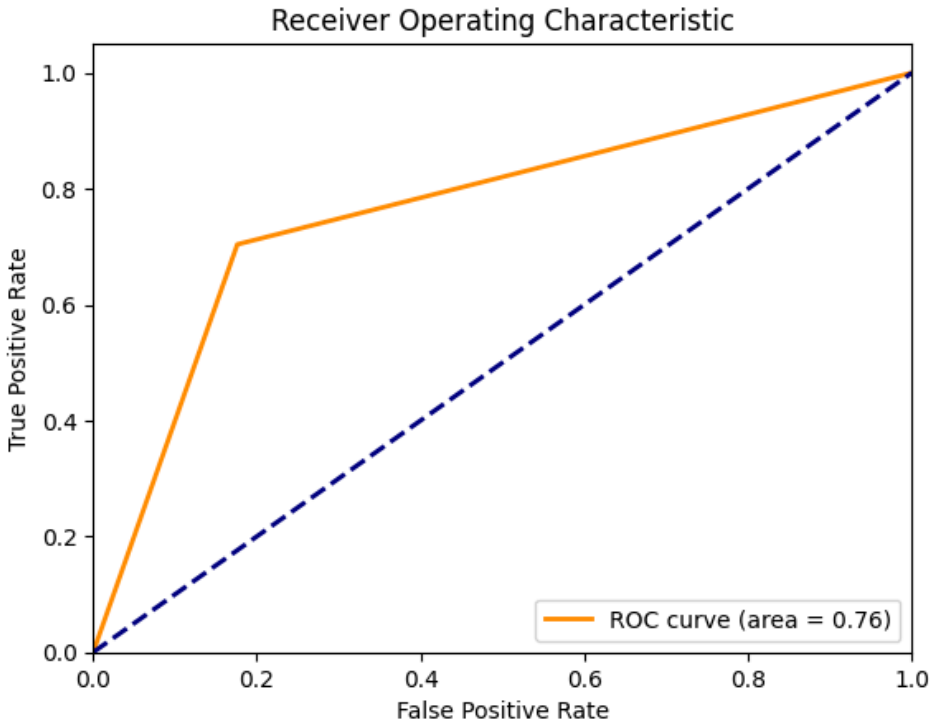
```
dataset.split_train_test_validation(split_size=.3,
validation=True)

proposed_model_cnn.train(dataset.train_gen, dataset.val_gen,
epochs=5)

proposed_model_cnn.analyze(dataset.test_data,
dataset.test_label)
```

split_size: 0.3:

```
              precision    recall  f1-score   support

           0       0.64      0.82      0.72       193
           1       0.86      0.70      0.78       304

    accuracy                           0.75       497
   macro avg       0.75      0.76      0.75       497
weighted avg       0.78      0.75      0.75       497
```



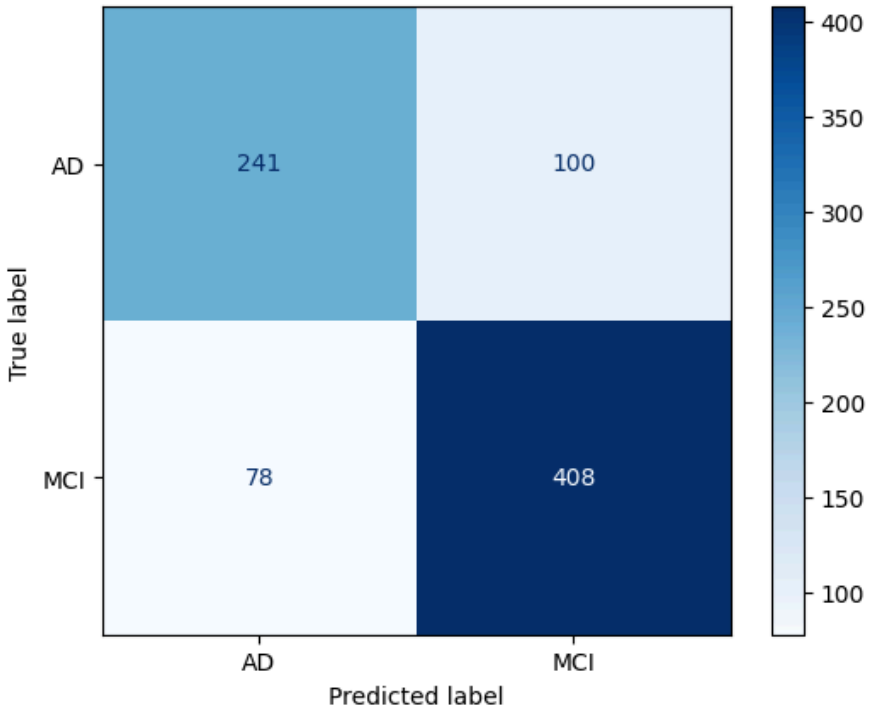
شکل 1-16. confusion matrix برای proposed model با split_size=0.3



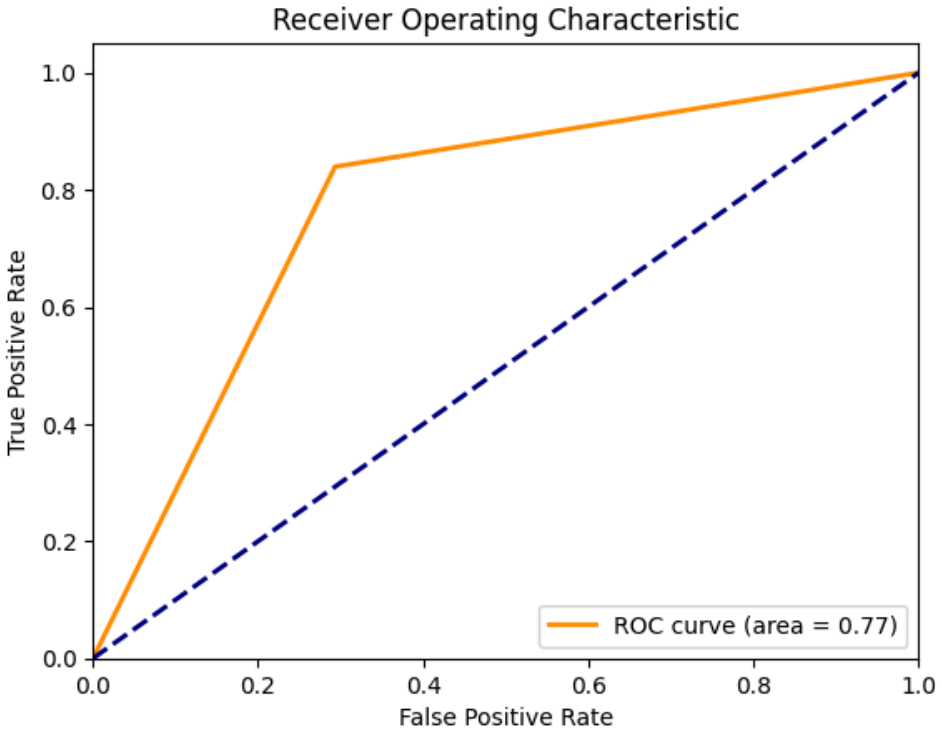
شکل 1-17. نمودار ROC برای proposed model با split_size=0.3

split_size: 0.5:

	precision	recall	f1-score	support
0	0.76	0.71	0.73	341
1	0.80	0.84	0.82	486
accuracy			0.78	827
macro avg	0.78	0.77	0.78	827
weighted avg	0.78	0.78	0.78	827



شکل 1-18. `confusion matrix` برای `proposed model` با `split_size=0.5`



شکل 1-19. نمودار ROC برای `proposed model` با `split_size=0.5`

	Train		Test	
split_size	accuracy	loss	accuracy	loss
0.2	98.75%	0.0896	89%	0.0901
0.3	75.67%	0.9452	75.05%	0.9913
0.5	78.62%	0.8684	78.43%	0.9446

جدول 1-2. خلاصه نتیجه تاثیر `split_size` بر روی عملکرد `proposed_model`

همانگونه که از نمودارها و مقدارها برداشت می‌شود مدل در زمانی که `split_size=0.5` باشد عملکرد نسبتا بهتری دارد، هر چند خروجی‌ها تفاوت چندانی ندارند هر چند اگر با زمانی که با نسبت `0.2` تقسیم شدند مقایسه کنیم در می‌یابیم که نتیجه‌ای که با آن نسبت میگرفتیم تفاوت فاحشی (حدود 10 درصد) با خروجی‌ها در این بقیه نسبت‌ها دارد.

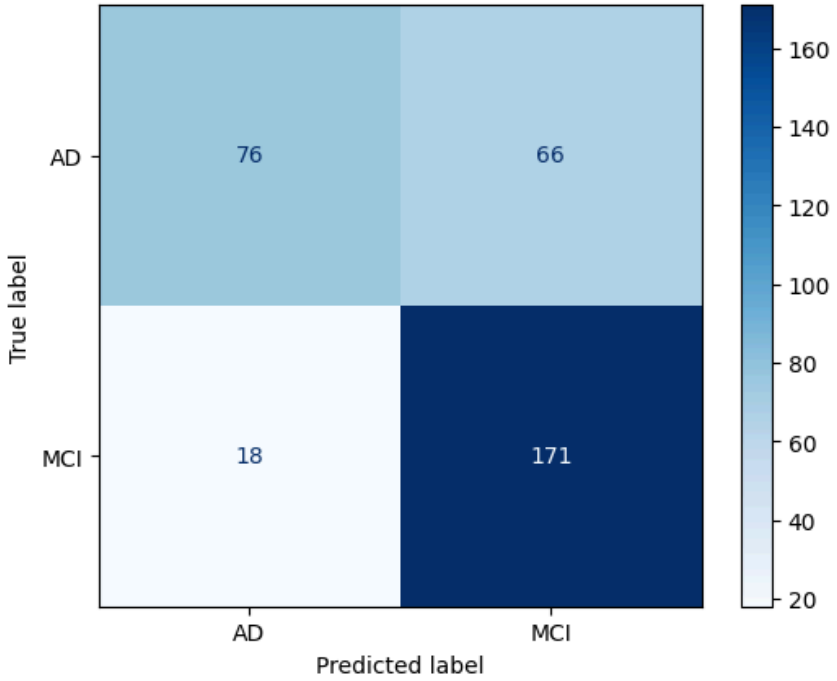
اثر Dropout:

سپس به بررسی اثر استفاده و استفاده نکردن از لایه‌های `Dropout` برای جلوگیری از بیش‌برازش می‌پردازیم. برای این کار ابتدا به شکل زیر مدل جدید را می‌سازیم:

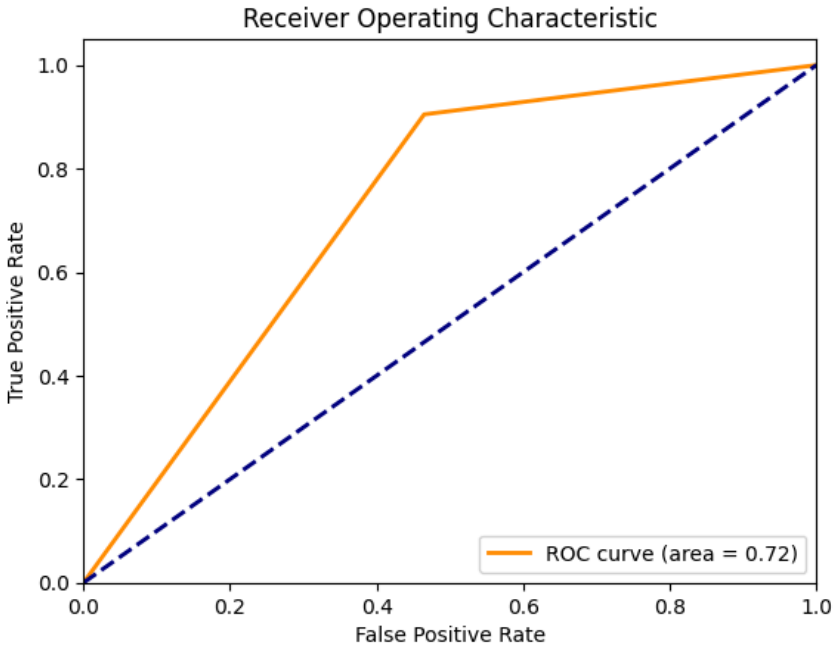
```
proposed_model_without_dropout = Sequential([
      Input(shape=(256, 256, 3)),
      Conv2D(
      32,
      (3, 3),
      strides=(1, 1),
      padding="same",
      kernel_initializer=initializer,
      ),
      BatchNormalization(),
      ReLU(),
      Conv2D(
      32,
      (3, 3),
      strides=(1, 1),
      padding="same",
      kernel_initializer=initializer,
      ),
      BatchNormalization(),
      ReLU(),
      MaxPooling2D((2, 2)),
      Conv2D(
      32,
      (3, 3),
      strides=(1, 1),
      padding="same",
      kernel_initializer=initializer,
      ),
      BatchNormalization(),
      ReLU(),
      Conv2D(
      32,
      (3, 3),
      strides=(1, 1),
      padding="same",
```

```
        kernel_initializer=initializer,
        ),
        BatchNormalization(),
        ReLU(),
        MaxPooling2D((2, 2)),
        Flatten(),
        Dense(
        128,
        activation="relu",
        kernel_initializer=initializer,
        ),
        BatchNormalization(),
        ReLU(),
        Dense(
        64,
        activation="relu",
        kernel_initializer=initializer,
        ),
        BatchNormalization(),
        ReLU(),
        Dense(
        2,
        activation="softmax",
        kernel_initializer=initializer,
        kernel_regularizer=regularizer,
        ),
])

proposed_model_without_dropout.compile(optimizer=Adam(learning_r
ate=.1), loss=categorical_crossentropy, metrics=["accuracy"])
```

```
              precision    recall  f1-score   support

           0       0.81      0.54      0.64       142
           1       0.72      0.90      0.80       189

    accuracy                           0.75       331
   macro avg       0.77      0.72      0.72       331
weighted avg       0.76      0.75      0.73       331
```



شکل 1-20. `confusion matrix` برای `proposed model` بدون `Dropout`



شکل 1-21. نمودار ROC برای `proposed model` بدون `Dropout`

	Train		Test	
regularization	accuracy	loss	accuracy	loss
with dropout	98.75%	0.0896	89%	0.0901
without dropout	97.77%	0.1555	75.1%	0.9671

جدول 1-3. خلاصه نتیجه تاثیر `Dropout` بر روی عملکرد `proposed_model`

مشاهدات برای تاثیر حذف `Dropout` نشان می‌دهد که در صورتی که از این روش استفاده نشود مدل دچار بیش‌برازش (overfitting) شده و با اینکه روی داده آموزش به خوبی عمل می‌کند اما قدرت تعمیم‌دهی زیادی ندارد و در نتیجه برای داده‌های جدید از پیش دیده‌نشده عملکرد خوبی ندارد. در اینجا برای مشاهده بیشتر اثر عدم کنترل بیش‌برازش (overfitting) از روش `L2` نیز استفاده نشده.

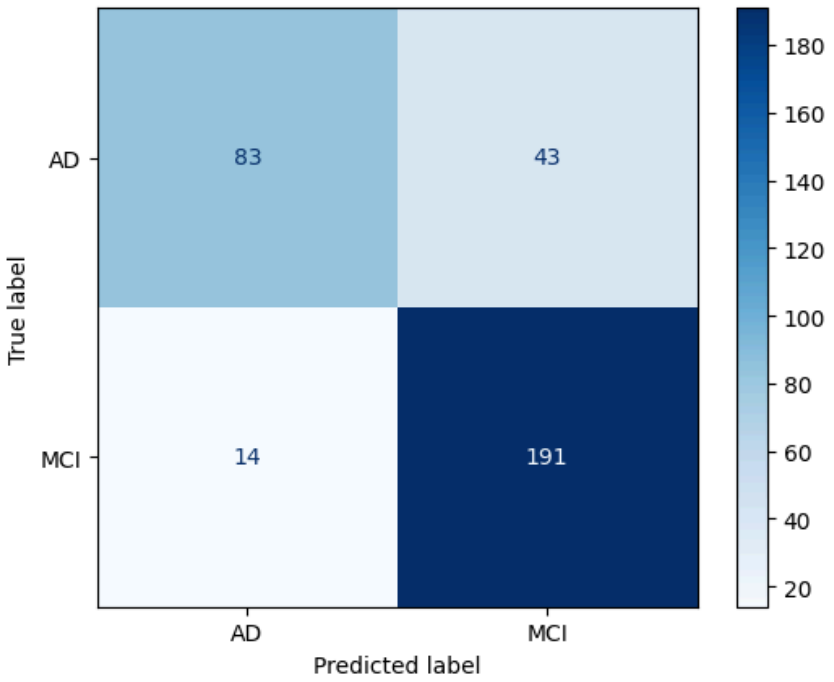
اثر Glorot Initialization:

پس از آن بررسی می‌کنیم که چقدر استفاده از روش `Glorot Initialization` بر روی سرعت همگرایی تاثیر داشته. پس یک بار دیگر مدل را این بار بدون این روش پیاده‌سازی می‌کنیم. نسبت و بقیه پارامترها در این مرحله مانند بقیه مراحل است و تغییر نکرده است.

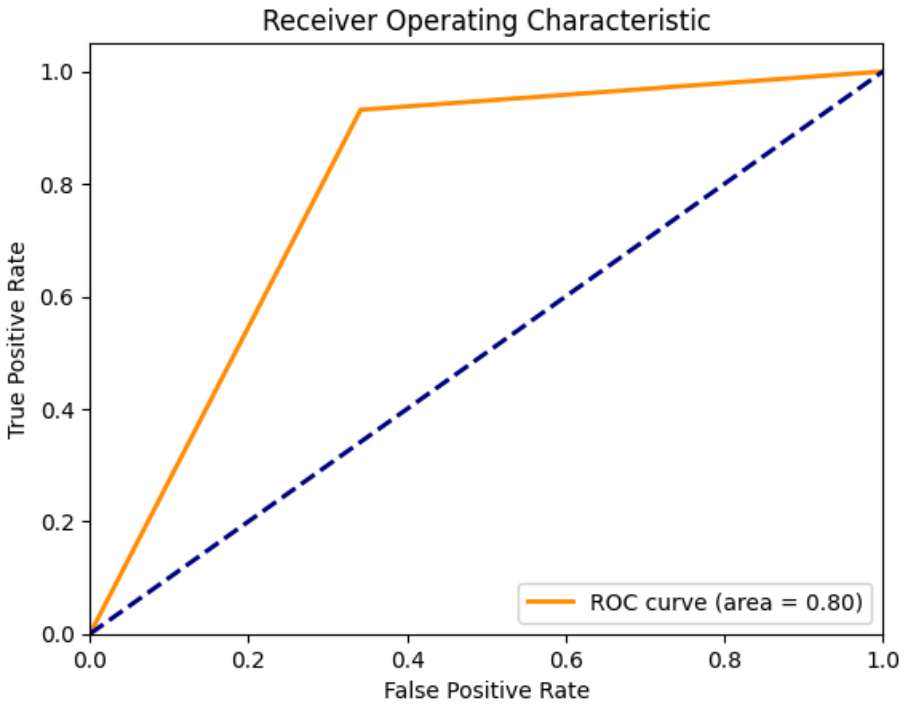
```
proposed_model_without_glorot_uniform = Sequential([
      Input(shape=(256, 256, 3)),
      Conv2D(
      32,
      (3, 3),
      strides=(1, 1),
      padding="same",
      kernel_regularizer=regularizer,
      ),
      BatchNormalization(),
      ReLU(),
      Dropout(0.5),
      Conv2D(
      32,
      (3, 3),
      strides=(1, 1),
      padding="same",
      kernel_regularizer=regularizer,
      ),
      BatchNormalization(),
      ReLU(),
      MaxPooling2D((2, 2)),
      Conv2D(
      32,
      (3, 3),
      strides=(1, 1),
      padding="same",
      kernel_regularizer=regularizer,
      ),
      BatchNormalization(),
      ReLU(),
      Dropout(0.5),
      Conv2D(
      32,
```

```
        (3, 3),
        strides=(1, 1),
        padding="same",
        kernel_regularizer=regularizer,
        ),
        BatchNormalization(),
        ReLU(),
        MaxPooling2D((2, 2)),
        Flatten(),
        Dense(
        128,
        activation="relu",
        kernel_regularizer=regularizer,
        ),
        BatchNormalization(),
        ReLU(),
        Dropout(0.5),
        Dense(
        64,
        activation="relu",
        kernel_regularizer=regularizer,
        ),
        BatchNormalization(),
        ReLU(),
        Dropout(0.5),
        Dense(
        2,
        activation="softmax",
        kernel_regularizer=regularizer,
        ),
])

proposed_model_without_glorot_uniform.compile(optimizer=Adam(lea
rning_rate=.1), loss=categorical_crossentropy,
metrics=["accuracy"])
```

```
              precision    recall  f1-score   support

           0       0.86      0.66      0.74       126
           1       0.82      0.93      0.87       205

    accuracy                           0.83       331
   macro avg       0.84      0.80      0.81       331
weighted avg       0.83      0.83      0.82       331
```



شکل 1-22. confusion matrix برای proposed model بدون Glorot Initialization



شکل 1-23. نمودار ROC برای `proposed model` بدون `Glorot Initialization`

	Train		Test	
initialization	accuracy	loss	accuracy	loss
with Glorot	98.75%	0.0896	89%	0.0901
without Glorot	81.15%	0.9219	74.46%	0.8288

جدول 1-4. خلاصه نتیجه تاثیر `Glorot Initialization` بر روی عملکرد `proposed_model`

همانگونه که انتظار داشتیم اگر از این روش استفاده نشود ممکن است با گرادیان‌های بزرگی روبرو شویم (در آموزش شبکه مجبور شدیم از نرخ یادگیری کوچک‌تری برای همگرایی استفاده کنیم) و همچنین مدل با سرعت بسیار کمتری همگرا می‌شود.

پرسش 2 - بررسی تاثیر افزایش داده بر عملکرد Fine-Tuned CNN

2-1. معرفی مقاله

در این مقاله ما به بررسی تاثیر استفاده از داده‌افزایی (data augmentation) در نحوه عملکرد مدل می‌پردازیم. برای این کار ما از دو مدل معروف از پیش آماده‌شده (pre-trained) به نام‌های VGG16 و ResNet-50 استفاده می‌کنیم و بعد به Fine-Tune کردن مدل شبکه عصبی کانولوشنال خودمان می‌پردازیم. Fine-tuning یک روش مهم در آموزش شبکه‌های عصبی است که به ما اجازه می‌دهد از اطلاعات آموزشی موجود در یک شبکه عصبی پیش‌آموزش‌دیده استفاده کنیم و سپس شبکه را برای یک وظیفه خاص دیگر، مثلاً دسته‌بندی تصاویر، بازآموزی دهیم. دلیل اصلی استفاده از Fine-tuning این است که معماری شبکه‌های عصبی پیش‌آموزش‌دیده معمولا دارای توانایی استخراج ویژگی‌های عمومی از داده‌ها است. این ویژگی‌های عمومی، می‌توانند برای وظایف دیگری نیز مفید باشند. مثلا برای مسئله ما شبکه‌های نام برده شده دقت خوبی روی مسائل طبقه‌بندی دارند و می‌توان از آن‌ها کمک گرفت و ویژگی‌های اصلی عکس‌های دیتاست خودمان را بدست آوریم سپس برای طبقه‌بندی از مدل دیگری استفاده می‌کنیم و آموزش مدل را روی آن قسمت انجام می‌دهیم.

معماری بخش پیچشی VGG16:

1. لایه ورودی
2. بلوک پیچشی 1
    a. لایه پیچشی با 64 فیلتر 3 × 3 با گام 1 و فعالساز RELU
    b. لایه پیچشی با 64 فیلتر 3 × 3 با گام 1 و فعالساز RELU
    c. لایه حداکثر گیر با سایز ادغام 2 × 2 و گام 2
3. بلوک پیچشی 2
    a. لایه پیچشی با 128 فیلتر 3 × 3 با گام 1 و فعالساز RELU
    b. لایه پیچشی با 128 فیلتر 3 × 3 با گام 1 و فعالساز RELU
    c. لایه حداکثر گیر با سایز ادغام 2 × 2 و گام 2
4. بلوک پیچشی 3
    a. لایه پیچشی با 256 فیلتر 3 × 3 با گام 1 و فعالساز RELU
    b. لایه پیچشی با 256 فیلتر 3 × 3 با گام 1 و فعالساز RELU
    c. لایه پیچشی با 256 فیلتر 3 × 3 با گام 1 و فعالساز RELU
    d. لایه حداکثر گیر با سایز ادغام 2 × 2 و گام 2
5. بلوک پیچشی 4
    a. لایه پیچشی با 512 فیلتر 3 × 3 با گام 1 و فعالساز RELU
    b. لایه پیچشی با 512 فیلتر 3 × 3 با گام 1 و فعالساز RELU
    c. لایه پیچشی با 512 فیلتر 3 × 3 با گام 1 و فعالساز RELU
    d. لایه حداکثر گیر با سایز ادغام 2 × 2 و گام 2
6. بلوک پیچشی 5
    a. لایه پیچشی با 512 فیلتر 3 × 3 با گام 1 و فعالساز RELU
    b. لایه پیچشی با 512 فیلتر 3 × 3 با گام 1 و فعالساز RELU
    c. لایه پیچشی با 512 فیلتر 3 × 3 با گام 1 و فعالساز RELU
    d. لایه حداکثر گیر با سایز ادغام 2 × 2 و گام 2

معماری ResNet-50:

معماری ResNet-50 پیچیده تر از VGG16 می‌باشد (و به همین علت نتیجه بهتری را نیز به همراه دارد. در ادامه کاملا توضیح داده خواهد شد.) تفاوت‌های اصلی آن با VGG16 عبارت‌اند از:

1. وجود لایه‌های BatchNormalization

2. وجود اتصال‌های میانبر[2] از برخی نورون لایه‌های پیچشی به نورن های لایه‌های پیچشی چند لایه بعدتر.

برای Fine-tuning، معمولاً به دو مرحله نیاز است:

1. **پیش‌آموزش (Pretraining):** در این مرحله، شبکه عصبی را بر روی یک مجموعه داده بزرگ و عمومی، مانند ImageNet، آموزش داده‌شده. در اینجا، شبکه ویژگی‌های عمومی را از تصاویر یاد می‌گیرد. دو نمونه از این شبکه‌ها VGG16 و ResNet-50 هستند.

خصوصیات اصلی مدل VGG16 عبارتند از:

I. عمق زیاد: این مدل از 16 لایه عمیق پیچشی و پرسپترون‌های کاملاً متصل[3] تشکیل شده است.
II. لایه‌های پیچشی: VGG16 شامل تعداد زیادی لایه پیچشی متوالی با هسته‌های کوچک 3x3 است که با استفاده از حاشیه گذاری[4] به اندازه‌ی همان‌طور که ابعاد جلوی ورودی را حفظ می‌کند، اعمال می‌شود.
III. ادغام[5] با حرکت کردن (گام[6]) 2x2: بین لایه‌های پیچشی، لایه‌های پولینگ با حرکت کردن (گام) 2x2 و حالت حداکثر گیر[7] وجود دارند که کاهش ابعاد فضایی تصاویر را انجام می‌دهند و اطلاعات مهم را از تصاویر استخراج می‌کنند.
IV. لایه‌های کاملاً متصل: پس از لایه‌های پیچشی و پولینگ، تعدادی لایه کاملاً متصل وجود دارد که باعث تبدیل ویژگی‌های استخراج شده از تصاویر به بردار ویژگی‌های پیوسته می‌شوند.
V. تابع فعال‌سازی ReLU: در تمام لایه‌های پیچشی و کاملا متصل، تابع فعال‌سازی ReLU استفاده می‌شود که به عنوان یکی از توابع فعال‌سازی

[2] Shortcut
[3] Fully Connected
[4] Padding
[5] Pooling
[6] Stride
[7] Max Pooling

محبوب استفاده می‌شود و به شبکه امکان یادگیری رفتار غیرخطی می‌دهد.

VI. توابع فعال‌سازی softmax در لایه خروجی: برای دسته‌بندی چند کلاسه، VGG16 از توابع فعال‌سازی softmax در لایه خروجی استفاده می‌کند که احتمال تعلق تصویر به هر یک از کلاس‌ها را محاسبه می‌کند.

خصوصیات اصلی مدل ResNet-50 عبارتند از:

I. بلوک‌های مابین اتصال (Skip Connection Blocks): یکی از ویژگی‌های مهم مدل ResNet از این بلوک‌ها استفاده است که امکان عبور اطلاعات از لایه‌های پایین‌تر به لایه‌های بالاتر را فراهم می‌کند. این بلوک‌ها کمک می‌کنند تا مشکل محو شدن گرادیان[8] در شبکه‌های عمیق را حل کرده و اجازه می‌دهند که شبکه‌ها با عمق بالا بیشتر یاد بگیرند.

II. بلوک‌های اساسی (Basic Blocks): مدل ResNet-50 از بلوک‌های اساسی که شامل لایه‌های کانولوشنالی و تابع فعال‌سازی ReLU هستند، استفاده می‌کند. این بلوک‌ها به عنوان واحدهای ساختمانی اصلی شبکه عمل می‌کنند و وظیفه استخراج ویژگی‌های تصویر را انجام می‌دهند.

III. لایه‌های اولیه (Initial Layers): مدل ResNet-50 با استفاده از لایه‌های اولیه از جمله لایه پیچشی و لایه پولینگ به استخراج ویژگی‌های ابتدایی از تصاویر می‌پردازد.

IV. لایه‌های پایانی (Final Layers): در انتهای مدل، یک لایه کاملاً متصل (Fully Connected) با تابع فعال‌سازی softmax برای دسته‌بندی تصاویر وجود دارد که احتمال تعلق تصویر به هر یک از کلاس‌ها را محاسبه می‌کند.

2. **بازآموزی (Fine-tuning):** در این مرحله، شبکه را بر روی داده‌های مخصوص وظیفه خاصی که قصد داریم برای آن استفاده کنیم، آموزش می‌دهیم. اینجاست که وزن‌های شبکه به‌روزرسانی می‌شوند تا بهترین نتایج برای وظیفه مورد نظر به دست آوریم. مفهوم Fine-tuning

[8] Vanishing Gradient

این است که شبکه عصبی پیش‌آموزش‌دیده، با تغییر و بهینه‌سازی وزن‌های خود برای وظایف خاص دیگری نیز قابل استفاده است. به این ترتیب، می‌توانیم از تمامی اطلاعات آموخته شبکه در مراحل پیشین (پیش‌آموزش) بهره‌بری کنیم و فقط وزن‌های مربوط به لایه‌های آخر شبکه را برای وظیفه جدید تنظیم کنیم. این کار امکان دارد زمان و میزان داده‌آموزی لازم برای دستیابی به عملکرد خوب را کاهش دهد. در اینجا ما با کمک دو مدل توضیح داده‌شده یک شبکه عصبی کانولوشنال روی دیتاست جدید ایجاد می‌کنیم و به بررسی عملکرد مدل می‌پردازیم.

2-2. پیش‌پردازش تصاویر

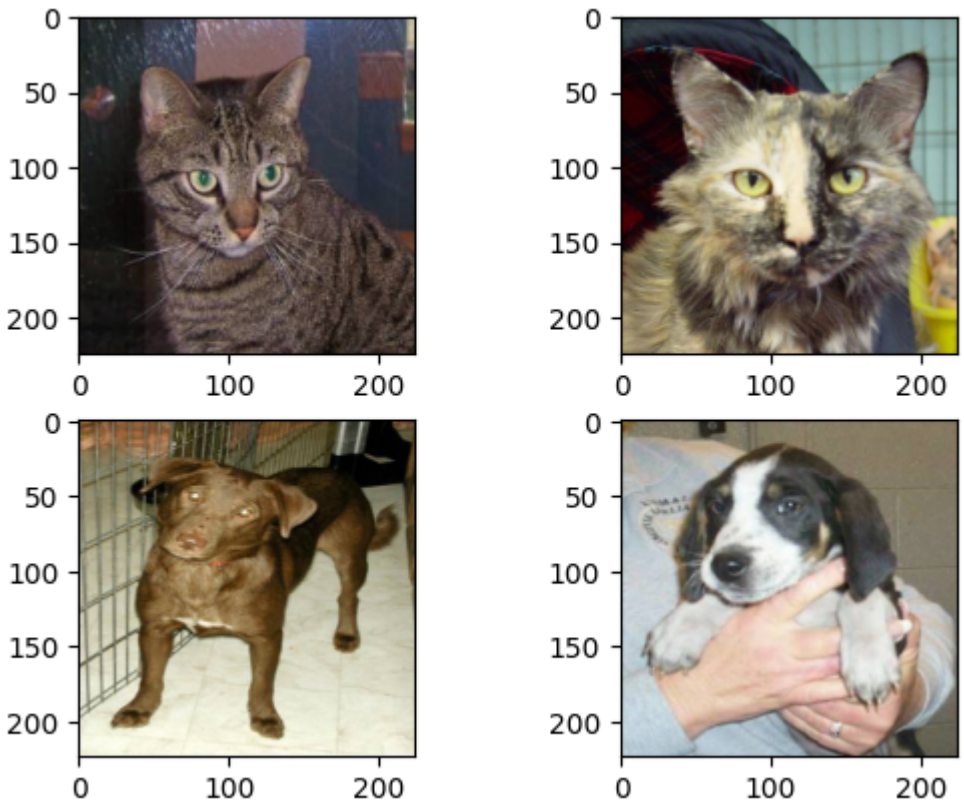
دیتاست جدید ما از 700 عکس (350 عکس گربه و 350 عکس سگ) برای آموزش[9] و 100 عکس (50 عکس گربه و 50 عکس سگ) برای آزمون[10] تشکیل شده که از طریق این لینک می‌توانید به آن دسترسی داشته باشید. برای خواندن عکس‌ها و اضافه کردن لیبل‌ها به شکل زیر عمل کردیم.

```
def read_images_of(folder_path):
      images = []
      for filename in os.listdir(folder_path):
      if filename.endswith(".jpg"):
            img_path = os.path.join(folder_path, filename)
            image = cv2.imread(img_path)
            image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
            image = cv2.resize(image, (224, 224))
            image = image / 255.0
            if image is not None:
                  images.append(image)
      images = np.array(images)
      return images


cats_train = read_images_of("../data/HW2_Dataset/Train/Cats")
cats_test = read_images_of("../data/HW2_Dataset/Test/Cats")
dogs_train = read_images_of("../data/HW2_Dataset/Train/Dogs")
dogs_test = read_images_of("../data/HW2_Dataset/Test/Dogs")

encoder = OneHotEncoder(sparse_output=False)

train_data = np.concatenate((cats_train, dogs_train), axis=0)
cats_label = np.ones(cats_train.shape[0], dtype=int)
dogs_label = np.zeros(dogs_train.shape[0], dtype=int)
train_labels = np.concatenate((cats_label, dogs_label),
axis=0).reshape(-1, 1)


shuffle_indices = np.random.permutation(train_data.shape[0])
train_data = train_data[shuffle_indices]
```

[9] Train

[10] Test

```
train_labels = train_labels[shuffle_indices]
train_labels = encoder.fit_transform(train_labels)


test_data = np.concatenate((cats_test, dogs_test), axis=0)
cats_label = np.ones(cats_test.shape[0], dtype=int)
dogs_label = np.zeros(dogs_test.shape[0], dtype=int)
test_labels = np.concatenate((cats_label, dogs_label),
axis=0).reshape(-1, 1)

shuffle_indices = np.random.permutation(test_data.shape[0])
test_data = test_data[shuffle_indices]
test_labels = test_labels[shuffle_indices]
test_labels = encoder.fit_transform(test_labels)
```



شکل 2-1. نمونه‌هایی از عکس‌های موجود در دیتاست

سپس داده‌های آزمایش را به دو بخش آزمایش و صحت سنجی[11] با نسبت 70-30 شکستیم و برای این کار به شکل زیر عمل کردیم. از آنجا که تصاویر دارای ابعاد یکسان و پیش‌پردازش‌های لازم بودند کار زیادی در این قسمت انجام نشده و صرفا داده‌ها را به چند مجموعه برای آموزش و صحت سنجی تقسیم کردیم.

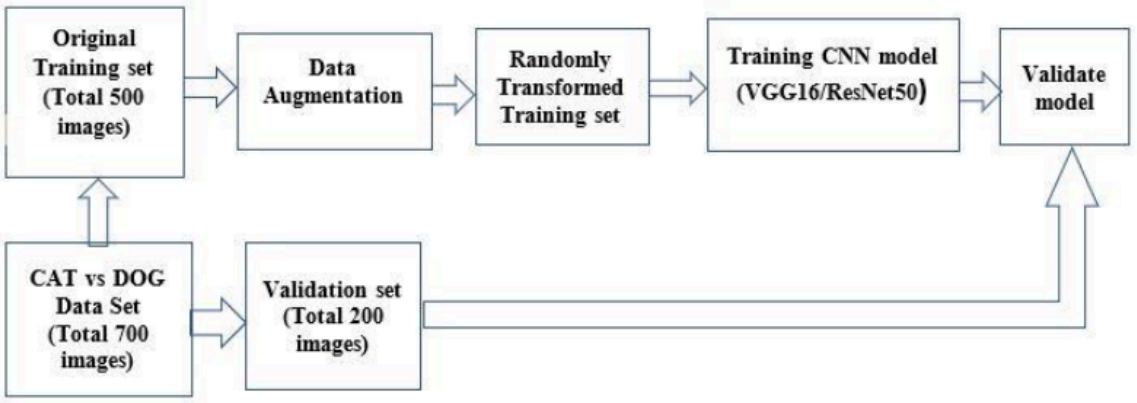
برای افزایش داده‌ها از سه روش ذکر شده در مقاله استفاده کردیم:

1. Horizontal Flipping: در این روش عکس حول محور عمودی خود می‌چرخد. دقت کنید که در داده‌ی مورد بررسی استفاده از Vertical Flipping توجیه‌پذیر نیست.
2. Zooming: در این روش عکس‌ها به مقدار 0.75 تا 1.25 برابر اندازه اولیه خود می‌شوند.
3. Rotation: در این روش عکس تا 30 درجه می‌چرخد.(چرخش به راست و به چپ می‌باشد - در نتیجه درجه آزادی 60 درجه ای وجود دارد.)

برای این کار از `keras.utils.ImageDataGenerator` بهره بردیم. با این کار ما مجموعه دیتاست آموزش را تا ده برابر افزایش دادیم.

```
train_datagen = ImageDataGenerator(
      horizontal_flip = True,
      rotation_range = 30,
      zoom_range = (0.75, 1.25)
).flow(
      x = X_train,
      y = y_train,
      batch_size = 10
)


valid_datagen = ImageDataGenerator().flow(
      x = X_valid,
      y = y_valid,
      batch_size = 10
)
```

[11] Validation



شکل 2-2. توضیحات کلی از مراحل انجام آزمایش

2-3. پیاده‌سازی

از آنجا که ما می‌خواهیم از روش Fine-Tuning استفاده کنیم پس محاسبات نسبتا سبک‌تری داریم و کاری که باید انجام دهیم این است که بعضی از پارامترهای VGG16 و ResNet-59 را آموزش دهیم تا به طبقه‌بندی دیتاست ما بپردازد. مراحلی که برای این کار باید انجام دهیم عبارت است از:

1. شبکه‌ی VGG16 یا ResNet50 را با وزن‌های پیش‌آموزش‌دیده بر روی مجموعه داده ImageNet بارگیری کنیم.
2. لایه‌های کاملاً متصل[12] اصلی را با لایه‌های جدید برای تعداد کلاس‌های تصویر جدید جایگزین کنیم.
3. تمامی لایه‌های پیچشی[13] را منجمد[14] کنیم تا تمامی اطلاعاتی که یاد گرفته‌اند منتقل شوند. در این حالت، وزن این این لایه‌ها در حین آموزش تغییر نمی‌کنند و ثابت می‌مانند.
4. شبکه را آموزش دهیم.

[12] Fully-Connected
[13] Convolutional
[14] Freeze

5. بلوک پیچشی آخر را که وظیفه استخراج ویژگی‌های خاص[15] را دارند را آزاد می‌کنیم.
6. دوباره شبکه را آموزش دهیم تا پارامترهای بلوک پیچشی آخرین لایه‌ها آموزش داده شوند.

در شکل زیر می‌توانید تعداد پارامترهای این دو شبکه را مشاهده کنید.

برای آموزش شبکه VGG16 از معماری زیر استفاده شده است، توجه کنید که در مقاله اصلی فقط به تعداد نورون‌های لایه‌های کاملا متصل اشاره شده و معماری آن شرح داده نشده است، برای همین معماری مناسب دلخواهی انتخاب شده است.

1. لایه صاف‌کننده[16]:

این لایه، وظیفه این این را دارد که خروجی های لایه قبلی را صاف کند(بعد های آن را به یک بعد کاهش دهد.) این لایه معمولا بعد از آخرین لایه پیچشی قرار می‌گیرد تا خروجی آن را برای ورودی لایه‌های کاملا متصل آماده کند.

2. لایه‌های کاملا متصل:

این لایه‌ها ویژگی‌های استخراج شده را یاد می‌گیرند.

3. لایه Batch Normalization:

این لایه برای تنظیم کردن شکل توزیع است. در حین آموزش ممکن است پارمترهای توزیع ورودی به هم بخورد، در نتیجه این لایه خروجی لایه قبل را می‌گیرد و آن را نرمال سازی می‌کند تا اثر این برهم‌ریختگی را بگیرد.

4. لایه‌های Dropout:

این لایه به طور تصادفی برخی از نورون ها را غیر فعال می‌کند تا استقلال نورون‌ها بیشتر شود و یادگیری مقاوم‌تر شد.

[15] Features

[16] Flatten

```
def create_model_vgg16():

      cnn_base = VGG16(weights="imagenet", include_top=False,
input_shape=(224, 224, 3))

      model = Sequential()
      model.add(cnn_base)
      model.add(Layers.Flatten())
      model.add(Layers.Dense(512, activation="relu"))
      model.add(Layers.BatchNormalization())
      model.add(Layers.Dropout(0.5))
      model.add(Layers.Dense(512, activation="relu"))
      model.add(Layers.Dropout(0.5))
      model.add(Layers.Dense(2, activation="softmax"))
      return cnn_base, model
```

برای آموزش شبکه ResNet50، پس از لایه‌های پیچشی از معماری زیر استفاده کردیم:

1. لایه GlobalAveragePooling2D:

این لایه همانند لایه Flatten می‌باشد با این تفاوت که بین خروجی‌های لایه قبل میانگین‌گیری می‌کند و بدین ترتیب تعداد پارامترها را کاهش می‌دهد.

2. لایه BatchNormalization
3. لایه کاملا متصل با 2 نورون و فعالساز softmax برای دسته بندی به 2 کلاس موجود در داده.

طبق مقاله با این معماری دارای 8192 پارامتر batch normalization و 4096 نورون کاملا متصل می‌باشد.

```
def create_model_resnet50():
      cnn_base = ResNet50(weights="imagenet", include_top=False,
input_shape=(224, 224, 3))

      x = (Layers.GlobalAveragePooling2D())(cnn_base.output)
      x = (Layers.BatchNormalization())(x)
      output = (Layers.Dense(2, activation="softmax"))(x)

      model = Model(inputs=cnn_base.input, outputs=output)
      return cnn_base, model
```

پارامترها	VGG16	ResNet50
کل پارامترها	27,626,178	23,600,002
پارامترهای قابل آموزش	19,910,914	1,111,938
پارامترهای غیر قابل آموزش	7,635,256	22,488,064

جدول 2-4. تعداد پارامترهای مدل‌های مورد بررسی

همانگونه که از کدها مشخص است ما برای بخش از پیش آموزش داده‌شده، از بخش‌های کانولوشنال مدل‌های VGG16 و ResNet-50 استفاده کردیم و در ادامه آن شبکه کاملا متصل (fully connected) را اضافه کردیم که برای ما طبقه‌بندی روی ویژگی‌های بدست آمده از بخش‌های کانولوشنال را انجام می‌دهد. همچنین برای آموزش شبکه از تابع هزینه cross-entropy استفاده شده است. برای بروزرسانی وزن‌ها از روش mini-batch gradient descent استفاده شده که نسبت به روش stochastic gradient descent همگرایی سریع‌تری دارد به طور معمول. همچنین در شکل 2-5 می‌توانید بقیه هایپر پارامترهای استفاده شده در آموزش شبکه را مشاهده کنید.

فرمول تغییر وزن‌ها با این روش به شکل زیر است که بعد از هر mini-batch تکرار می‌شود:

$$w = w - lr \times dw$$

همچنین نرخ آموزش[17] با این فرمول به‌روزرسانی می‌شود:

$$lr = lr \times \frac{1}{1 + decay \times epoch}$$

[17] Learning Rate

مقادیر پارامترهای استفاده شده در جدول زیر آمده است. دقت کنید که در مقاله منبع از نرخ آموزش اولیه 0.1 استفاده شده است،که احتمالا به خاطر تفاوت معماری لایه‌های کاملا متصل می‌باشد. در معماری استفاده شده، با استفاده از این نرخ، تابع هزینه[18] واگرا می‌شود و به بی‌نهایت میل می‌کند.

پارامتر	نرخ آموزش اولیه	نرخ decay	momentum	سایز mini-batch	تعداد دوره‌ها
مقدار	2e-5 , 1e-2	0.002	0.9	10	50

جدول 2-5. هایپرپارامترهای استفاده شده برای آموزش مدل

در کد زیر نحوه پیاده‌سازی مدل را مشاهده می‌کنید:

```
optimizer = keras.optimizers.SGD(learning_rate=2e-5,
momentum=0.9)

model.compile(
      optimizer=optimizer, loss="categorical_crossentropy",
metrics=["accuracy"]
)
```

[18] Loss Function

پس از آموزش دادن لایه‌های کاملا متصل، برای fine-tune کردن شبکه، آخرین بلاک پیچشی را آزاد می‌کنیم و بار دیگر شبکه را به داده‌ها آموزش می‌دهیم. این کار در دو حالت انجام داده شده است.

1. استفاده از داده‌های اولیه
2. استفاده از داده‌ها اضافه شده

برای باز کردن انجماد آخرین بلوک پیچشی به طور زیر عمل می‌کنیم.

برای VGG16:

```
set_trainable = False
for layer in cnn_base.layers:
     if layer.name == "block5_conv1":
          set_trainable = True
     if set_trainable:
          layer.trainable = True
     else:
          layer.trainable = False
```

برای ResNet-50:

می‌دانیم در معماری ResNet-50 از لایه های BatchNormalization استفاده شده است. این لایه‌ها با استفاده دیتاست ImageNet آموزش داده شده‌اند در نتیجه ویژگی‌های آماری متناسب با توزیع آن داده‌ها را یاد گرفته اند. در هنگام استفاده از این لایه‌ها برای مجموعه داده جدید، اگر توزیع داده‌های جدید با توزیع داده‌های آموزش دیده متفاوت باشد، مدل دچار خطای زیادی می‌شود، زیرا این لایه‌ها سعی می‌کنند ورودی را با استفاده از متغیر های توزیع اولیه(میانگین و واریانس) scale کنند و به علت تفاوت توزیع، داده‌ها دچار انحراف می‌شوند. در نتیجه برای fine-tune کردن این مدل، لایه‌های BatchNormalization را **قابل آموزش** می‌کنیم تا ویژگی‌های توزیع جدید را یاد بگیرند. شایان ذکر است که در حالت منجمد بودن این لایه‌ها، دقت مدل از 65 درصد عبور نمی‌کرد.

```
set_trainable = False
for layer in cnn_base.layers:
    if layer.name == "conv5_block3_1_conv":
            set_trainable = True
    if set_trainable:
            layer.trainable = True
    else:
            layer.trainable = False

for layer in model.layers:
    if layer.name.endswith("bn"):
      layer.trainable = True
```

استفاده از واحد پردازشی گرافیکی[19] برای آموزش:

برای تسریع آموزش دادها تصویری، به علت سنگین بودن پردازش این نوع داده‌ها می‌توان از واحد پردازشی گرافیکی استفاده کرد. برای این کار می توان از کودا[20] و cuDNN برای کارت‌های گرافیک Nvidia بهره برد. البته در این استفاده محدودیت هایی وجود دارد، از مهم‌ترین آن‌ها محدودیت حافظه کارت گرافیک می‌باشد. برای برطرف کردن این محدودیت از روش زیر استفاده شده است:

1. آموزش مدل را به چند قسمت تقسیم شده است.
2. بخشی از آموزش انجام داده می‌شود.
3. وزن‌های مدل در حافظه اصلی ذخیره می‌شود.
4. حافظه کارت گرافیک را خالی می‌کنیم.
5. مدل را دوباره با وزن‌های آموزش داده شده، بارگیری می‌کنیم.
6. آموزش را از سر می‌گیریم.

همچنین استفاده از batch size های کوچک‌تر نیز برای مدیریت حافظه استفاده می‌شود که در این مقاله از batch size به مقدار 10 استفاده شده است که این مشکل بوجود نمی‌آید.

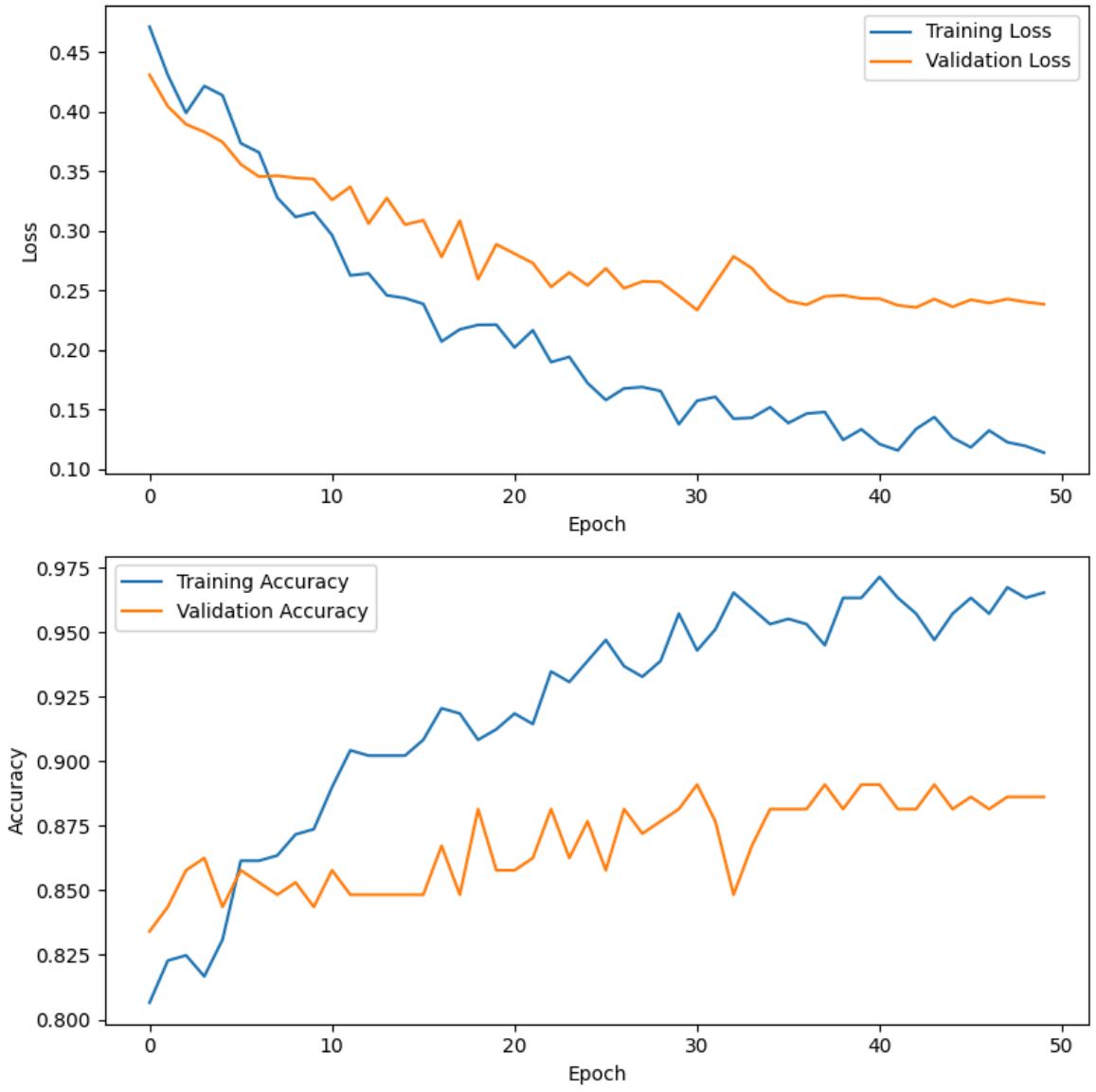
[19] GPU

[20] CUDA

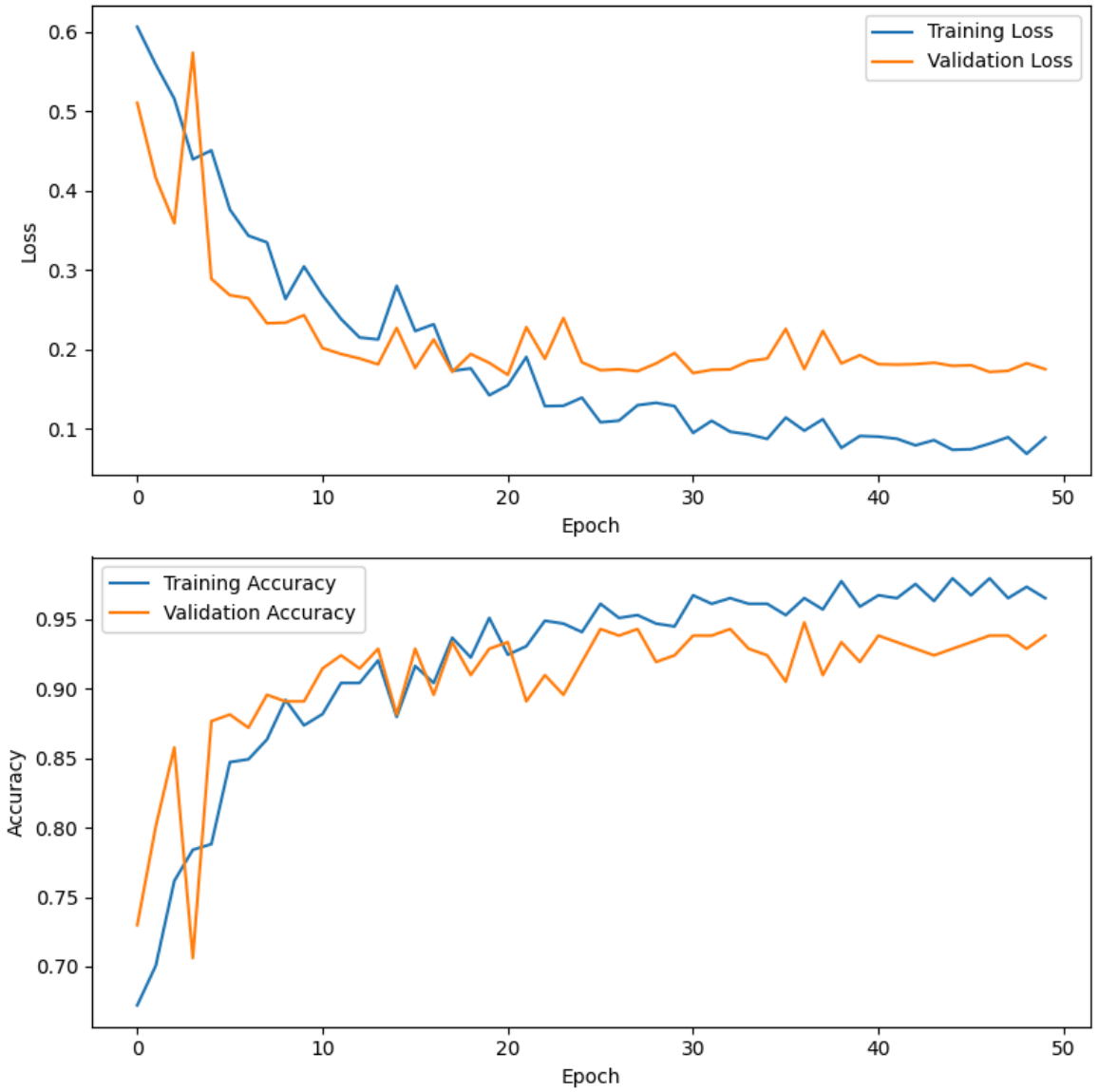
2-4. مقایسه نتایج

برای مقایسه نتایج استفاده از Data Augmentation، با هر دو روش شبکه را آموزش می‌دهیم و نتایج آن‌ها را بررسی می‌کنیم. توجه کنید که در این مرحله بقیه پارامترها تغییری نداشتند و صرفا از دو دیتاست متفاوت (augmented و not augmented) استفاده شده.

:VGG16

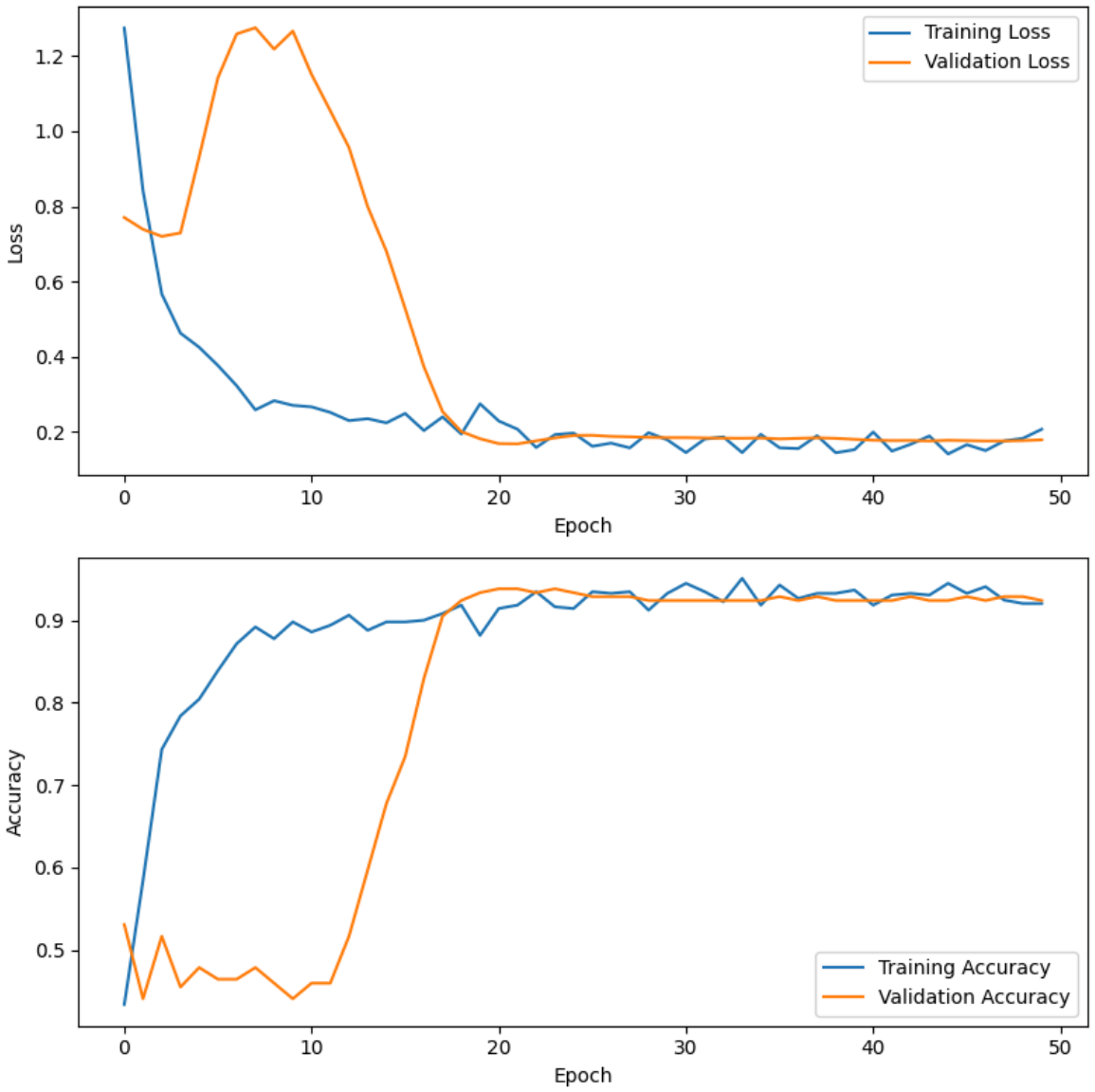


شکل 2-6. نمودارهای accuracy و loss برای حالت داده خام برای Fine-Tuned VGG16

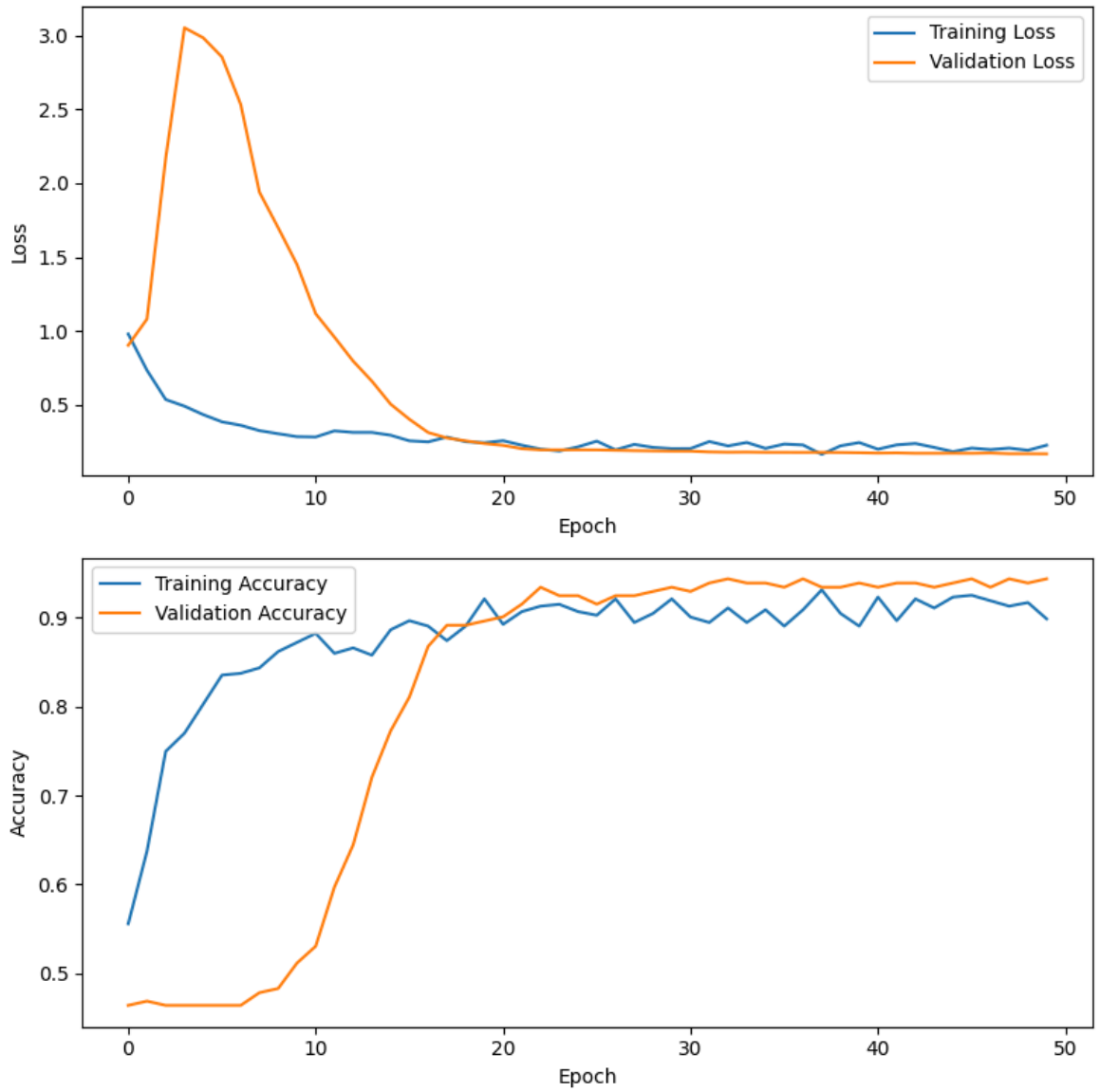


شکل 2-7. نمودارهای accuracy و loss برای حالت داده افزوده‌شده برای Fine-Tuned VGG16

:ResNet-50



شکل 2-8. نمودارهای accuracy و loss برای حالت داده خام برای Fine-Tuned ResNet-50



شکل 2-9. نمودارهای accuracy و loss برای حالت داده افزوده‌شده برای Fine-Tuned ResNet-50

از نمودارها می‌توان نتیجه گرفت که مدل ResNet-50 سریع‌تر همگرا شده و نیاز به ایپاک کمتری دارد همچنین دقت بیشتری نیز به ما می‌دهید. همچنین خود مدل ResNet-50 زمانی که داده بیشتری داشته باشد می‌تواند با دقت بهتری طبقه‌بندی کند.

مقایسه:

		Train		Validation		Test
model	data	accuracy	loss	accuracy	loss	accuracy
VGG16	with augmentation	97.35%	0.0892	93.84	0.1753	94%
	without augmentation	96.54%	0.1139	88.63%	0.2384	92%
ResNet-50	with augmentation	91.65%	0.1913	94.31%	0.1674	97%
	without augmentation	92.06%	0.2061	92.42%	0.1779	94%

جدول 2-1. خلاصه نتیجه معماری سه مدل پیشنهاد شده در مقاله

همانگونه که مشاهده می‌شود همانگونه که انتظار داشتیم در حالتی که از تکنیک داده‌افزایی (data augmentation) استفاده کردیم مدل قابلیت تعمیم‌پذیری بهتری دارد و روی داده‌هایی که تا الان مشاهده نکرده بهتر عمل می‌کند و این را در هر دو مدل مشاهده می‌کنیم. علت این اتفاق این است که مدل داده‌های متفاوت بیشتری دریافت کرده و در نتیجه بهتر می‌تواند برای نمونه‌های جدید تصمیم بگیرد.

علت تفاوت و عملکرد بهتر مدل ResNet-50 نسبت به مدل VGG16 می‌تواند این باشد که مدل ResNet-50 از تعداد لایه و پیچیدگی بسیار بیشتری برخوردار است (که پیش‌تر توضیح داده شده است) در نتیجه می‌تواند ویژگی‌های بیشتر و دقیق‌تری از ورودی به دست آورد و در نتیجه در بخش کاملا متصل (fully connected) با وجود مدل ساده‌تر، می‌توانیم با دقت بیشتری عکس‌ها را طبقه‌بندی کنیم.