به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

درس شبکه‌های عصبی و یادگیری عمیق

تمرین دوم

نام و نام خانوادگی	عرفان باقری سولا – محمد قره حسنلو
شماره دانشجویی	۸۱۰۱۹۸۳۶۱ - ۸۱۰۱۹۸۴۶۱
تاریخ ارسال گزارش	۱۴۰۲.۰۱.۱۶

فهرست

شکل ها

جدول‌ها

پاسخ ۱. شبکه عصبی پیچشی کم عمق برای طبقه بندی تصویر

۱-۱. آماده سازی و پیش پردازش داده ها

بارگذاری داده ها به شکل زیر برای هر سه دسته داده انجام شده است که در آن پیش پردازش هایی انجام شده است که در ادامه به آن اشاره می کنیم.

```
train_dataset_FashionMNIST = torchvision.datasets.FashionMNIST(
    root = "data", # where to download data to
    train = True, # get training data
    download = True, # download data if it doesn't exist on disk
    transform = transform # images come as PIL format, we want to turn into Torch tensors
)

test_dataset_FashionMNIST = torchvision.datasets.FashionMNIST(
    root = "data",
    train = False, # get test data
    download = True,
    transform = transform
)
```

شکل ۱: بارگذاری داده های **FashionMNIST**

در مقاله به طور مستقیم گفته شده به طور رندوم، عکس های آموزش و تست با احتمال 0.5 flip شده و از آنها به عنوان داده های آموزش و تست استفاده می شود. با توجه به مقاله باید دقت کرد که سایز عکس ها در دسته داده های MNIST و FashionMNIST برابر ۲۸*۲۸ بوده و در دسته داده CIFAR10 برابر ۳۲*۳۲ میباشد.

این دسته داده ها علاوه بر RandomHorizontalFlip گفته شده، Normalize نیز میشوند. mean و std در نظر گرفته شده هر دو برابر ۰.۵ میباشد. این کار باعث میشود که توزیع پیکسل ها در یک بازه مشابه قرار بگیرند و یاد گرفتن ویژگی ها برای مدل راحت تر و قابل قهم تر شود. همچنین از پدیده هایی مانند ناپدید شدن گرادیان یا explode شدن گرادیان جلوگیری میکند. برای استفاده از Normalize میتوانیم به transform ها، transform.Normalize((0.5,), (0.5,)) اضافه کنیم. در نرمالایز کردن مقادیر هر داده را منهای میانگین کرده و نتیجه تقسیم بر انحراف معیار میشود. خوبی انتخاب ۰.۵ این است که داده ها را حول میانگین متقارن میکند که از مشکلاتی مانند ناپددید شدن گرادیان و انفجار گرادیان تا حدی جلوگیری میکند. همچنین از اشباع توابع فعالسازی تا حدی جلوگیری میکند.

علاوه بر دو پیش پردازش گفته شده، ()ToTensor نیز برای تبدیل داده های عکس از PIL به تنسور استفاده میشود. همچنین مقدار پیکسل هارا از بازه ۰ تا ۲۵۵ به بازه ۰ تا ۱ اسکیل میکند.

```
mean, std = 0.5, 0.5

transform = transforms.Compose([
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.ToTensor(),
    transforms.Normalize((mean,), (std,))
])
```

شکل ۲: پیش پردازش های انجام شده برای هر سه دسته داده

۱-۲. توضیح لایه های مختلف معماری شبکه

لایه ورودی: در هنگام تعریف کردن مدل shallow CNN باید به این موضوع دقت کنیم که دسته داده های MNIST و FashionMNIST، grayscale هستند، برای همین ورودی برای این دو داده دسته، یک بوده ولی چون CIFAR10، RGB میباشد، ورودی مدل سه میباشد. همچنین اندازه عکسها در MNIST و FashionMNIST، ۲۸*۲۸ بوده ولی در CIFAR10 برابر ۳۲*۳۲ میباشد که باعث میشود در لایه fully connected، تعداد hidden unit متفاوت باشد.

لایه convolutional: این لایه به منظور استخراج کردن ویژگی های داده استفاده میشود. هر چه تعداد کرنل های convolution بیشتر باشد، توانایی بیشتری برای استخراج ویژگی دارد اما با توجه به اینکه هدف این مقاله استفاده از شبکه کوچکتر با دقت بالاست، تنها از دو لایه convolitional ۳*۳ استفاده شده است که هر کدام ۳۲ و ۶۴ فیلتر دارد.

لایه max-pooling: بعد از آنکه ویژگی ها استخراج شوند، از لایه pooling برای کاهش دادن افزونگی استفاده میشود. در این مقاله از دو max-pooling ۲*۲ استفاده میشود. مزایای این لایه، کاهش بعد دیتا و سختی محاسبه است که ویژگی های مفید را نگه داری کند. همچنین تا حدی از overfitting جلوگیری میکند.

لایه fully connected: هر نورون در لایه به همه نورون های لایه بعدی متصل میشود و در شبکه convolution در انتهای شبکه استفاده میشود تا بعد از استخراج ویژگی های مفید و حذف افزونگی ها ، آنها را در یک خط arrange کنیم.

لایه softmax: برای دستیابی به multi-classification استفاده میشود تا نشان دهد با ویژگی های یادگرفته شده، با احتمال بیشتر در کدام دسته قرار میگیرد.

لایه ReLU: به عنوان لایه فعال کننده غیرخطی استفاده میشود که در مقاله بعد از لایه convolutional و BN استفاده میشود. از این لایه برای حل کردن مشکل خطی نبودنمدل استفاده میشود که مدل را قوی تر میشود.

لایه Batch Normalization: لایه ای است که در مقاله از بیشترین تاکیدها برخوردار بود و حتی مدل را به سه دسته تقسیم کرده بود:

۱- SCNNB: در این مدلبعد از هر دو لایه convolution، ازBN استفاده کرده شده بود.

۲- SCNNB-a: BN بعد از لایه convolution اول، حذف شده است.

۳- SCNNB-b: هیچ BN استفاده نشده بود.

به طور خلاصه نتیجه نشان داد که SCNNB بهتر از دوتای دیگه عمل کرد. به همین دلیل تاکید زیادی روی BN شده بود. از BN برای بالا بردن سرعت training و بهتر کردن نتیجه classification استفاده شده است.

لایه Dropout: اگر تعداد پارامترهای آموزش زیاد باشد و تعداد داده ورودی کمتر باشد، ممکن است به overfitting برخورد بکنیم. Dropout یک استراتژی ممکن برای جلوگیری از overfitting است که به طور رندوم، تعدادی از نورون ها در لایه های fully connected را در نظر نمیگیرد.

۱-۳. پیاده سازی معماری

به طور کلی شبکه SCNNB از دو لایه convolutional، دو لایه max-pooling با سایز ۲*۲، یک لایه fully connected و یک لایه softmax تشکیل شده است.

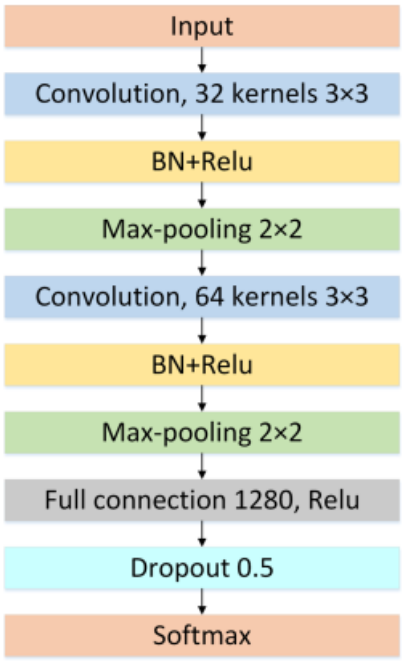
برای سرعت بخشیدن به شبکه و بهبود بخشیدن قابلیت generalization، بعد از هر لایه convolutional، Batch Normalization اضافه شده است.

برای اینکه هدف مقاله، مدل shallow از CNN است، به همین دلیل از kernel size کوچک ۳*۳ برای هر دو لایه convolutional استفاده شده است که لایه اول دارای ۳۲ فیلتر و دومی دارای ۶۴ فیلتر میباشد.

برای جلوگیری از خطی شدن مدل، بعد از هر BN، از ReLU استفاده شده است.

در انتها از لایه fully connected با ۱۲۸۰ نورون استفاده شده است که همراه با Dropout با احتمال ۰.۵ آمده است تا از overfitting جلوگیری کند.

در آخر برای multi-classification از softmax استفاده شده است.



شکل ۳: معماری برای مدل **SCNNB**

```python
class SCNNB(nn.Module):
    def __init__(self, input_shape, hidden_shape, output_shape):
        super().__init__()

        self.CNN_block = nn.Sequential(
            nn.Conv2d(input_shape, 32, kernel_size = 3),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(32, 64, kernel_size = 3),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )

        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(in_features = 64*hidden_shape*hidden_shape, out_features = 1280),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(in_features = 1280, out_features = output_shape)

        )

    def forward(self, x: torch.Tensor):
        # print(x.shape)
        x = self.CNN_block(x)
        # print(x.shape)
        x = self.classifier(x)
        # print(x.shape)
        return x
```

شکل ۴: پیاده سازی معماری **SCNNB** در کد

هایپرپارامتر ها و پارامترهای گفته شده در مقاله به صورت زیر میباشد:

۱- learning rate با اندازه ۰.۰۲

۲- optimizer استفاده شده SGD با momentum برابر ۰.۹ و weight_decay برابر ۰.۰۰۰۰۰۵

۳- نرخ Dropout برابر ۰.۵

۴- تعداد epoch برابر ۳۰۰ (که با توجه زمان بر بودن و پیشنهاد گزارش کار برابر ۱۵۰ گذاشته شده بود ولی در گزارش نتایج برای هر معماری به مقدارهای دیگری تغییر کرد که در ادامه ذکر خواهد شد.)

۵- batch size برابر ۱۲۸

۶- loss استفاده شده CrossEntropyLoss میباشد، چون با چند احتمال چند کلاس در لایه آخر و مسئله classification طرف هستیم.

سه معماری پیشنهادی در مقاله گفته شده که معماری که بالاتر گفتیم، معماری SCNNB بوده و دو معماری دیگر SCNNB-a بوده که BN بعد از لایه کانولوشن اول حذف شده و در SCNNB-b هر دو BN ها حذف شده اند.

```python
class SCNNB_a(nn.Module):
    def __init__(self, input_shape, hidden_shape, output_shape):
        super().__init__()

        self.CNN_block = nn.Sequential(
            nn.Conv2d(input_shape, 32, kernel_size = 3),
            # nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(32, 64, kernel_size = 3),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )

        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(in_features = 64*hidden_shape*hidden_shape, out_features = 1280),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(in_features = 1280, out_features = output_shape)

        )

    def forward(self, x: torch.Tensor):
        # print(x.shape)
        x = self.CNN_block(x)
        # print(x.shape)
        x = self.classifier(x)
        # print(x.shape)
        return x
```

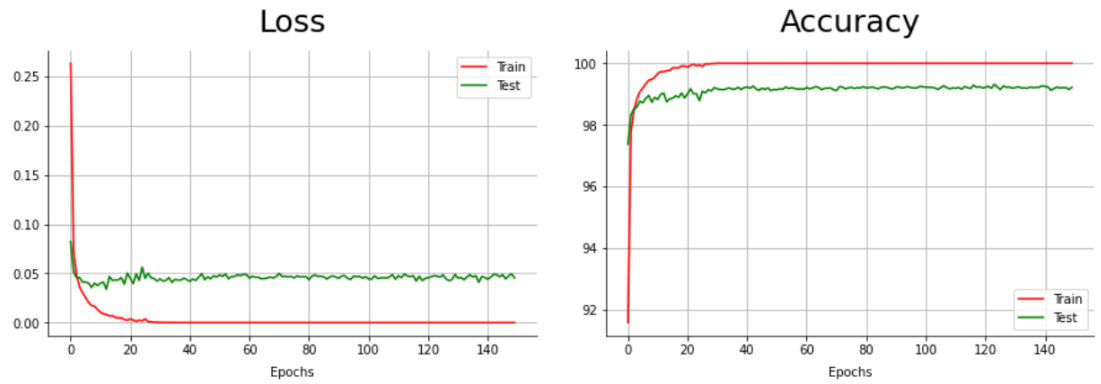
شکل ۵: پیاده سازی معماری **SCNNB-a** در کد

```python
class SCNNB_b(nn.Module):
    def __init__(self, input_shape, hidden_shape, output_shape):
        super().__init__()

        self.CNN_block = nn.Sequential(
            nn.Conv2d(input_shape, 32, kernel_size = 3),
            # nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(32, 64, kernel_size = 3),
            # nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )

        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(in_features = 64*hidden_shape*hidden_shape, out_features = 1280),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(in_features = 1280, out_features = output_shape)

        )

    def forward(self, x: torch.Tensor):
        # print(x.shape)
        x = self.CNN_block(x)
        # print(x.shape)
        x = self.classifier(x)
        # print(x.shape)
        return x
```

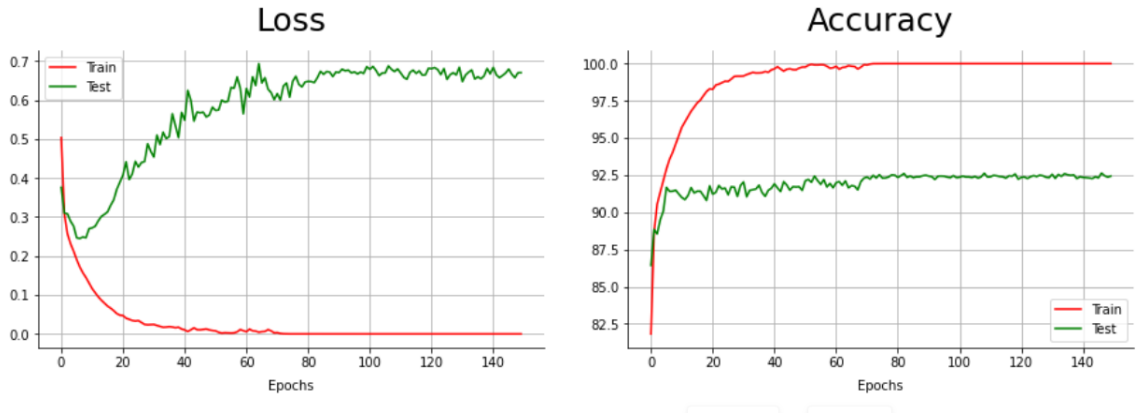
شکل ۶: پیاده سازی معماری **SCNNB-b** در کد

۱-۴. نتایج پیاده سازی

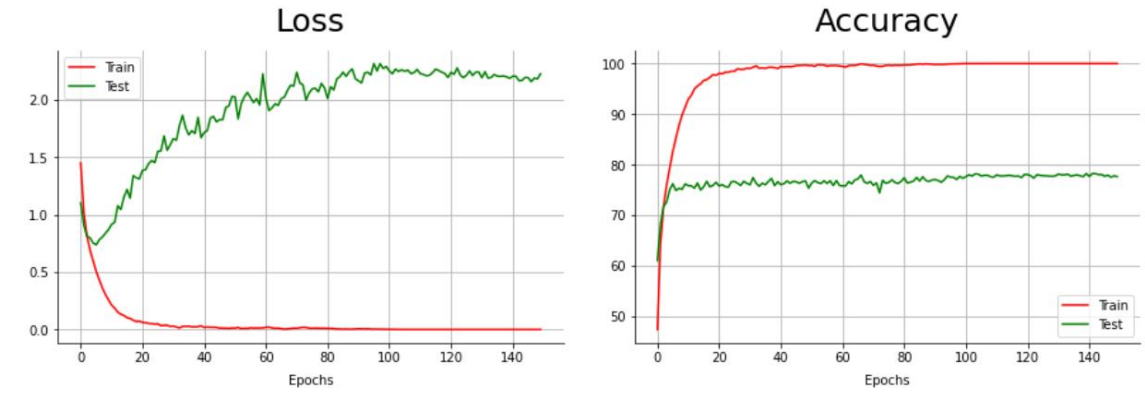
ابتدا هر سه دسته داده را برای ۱۵۰ دور برای مدل SCNNB فیت کردیم اما بسیار طول کشید (هر کدام حداقل ۵۵ دقیقه) و چون در هر مدل برای هر سه دسته داده باید نمودار جداگانه ای کشیده می شد، گوگل کولب به دلیل اتمام وقت مجاز استفاده از GPU اجازه ادامه نمی داد و به همین دلیل باید چند روز منتظر اتمام محدودیت ها می شدیم تا دوباره کارمان ادامه دهیم. به همین دلیل مجبور شدم تعداد دورها را کم کنم تا بتوان نتیجه را نمایش داد. عکس های ذیل نتایجی هست که قبل از از دست دادن نتایج آموزش داده شده بود.



شکل ۷: نمودار خطا و دقت برای دادگان آموزش و ارزیابی بر دسته داده **MNIST**



شکل ۸: نمودار خطا و دقت برای دادگان آموزش و ارزیابی بر دسته داده **FashionMNIST**



شکل ۹: نمودار خطا و دقت برای دادگان آموزش و ارزیابی بر دسته داده **CIFAR10**

در ادامه با توجه به محدودیت های ذکر شده و همچنین ثابت شدن دقت در دادگان آموزش و ارزیابی در یک epoch مشخص برای هر کدام از دسته داده ها، تعداد epoch های در نظر گرفته شده برای دسته داده MNIST برابر ۴۰، برای FashionMNIST برابر ۸۰ و برای CIFAR10 برابر ۱۰۰ می باشد.

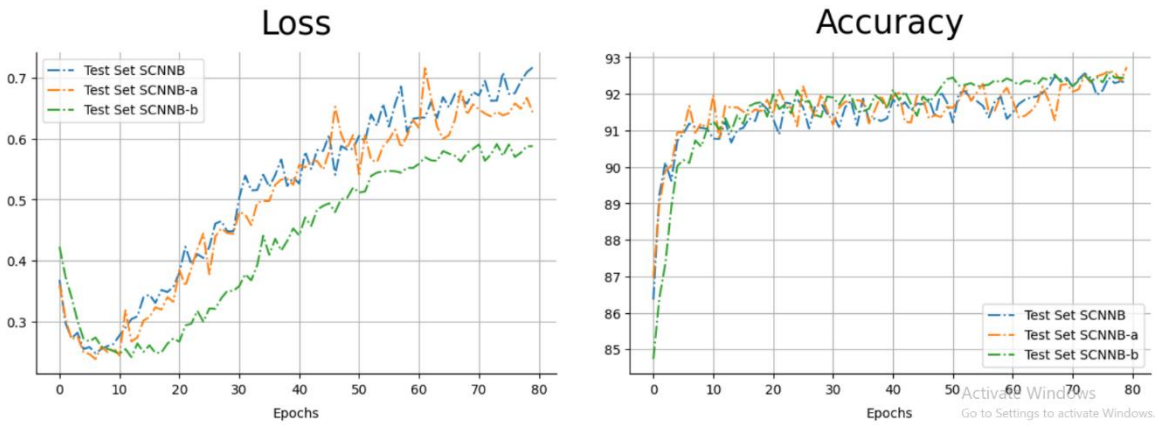
الف: نمودار دقت و خطا برای دادگان ارزیابی

در این قسمت برخلاف انتظاری که طبق مقاله داشتیم، SCNNB-a و حتی SCNNB-b در بعضی دسته داده ها نسبت به SCNNB عملکرد بهتری داشته است که دلایل مختلفی میتواند داشته باشد؛ از جمله مشخص نبودن کامل استراتژی BN برای مدل های ذکر شده، run نشدن برای ۳۰۰ دور به دلیل محدودیت های ذکر شده در قسمت قبل، چندین بار فیت کردن بر روی داده آموزش و گرفتن نتیجه از روی داده ارزیابی و در نهایت میانگین گرفتن از نتایج و... .

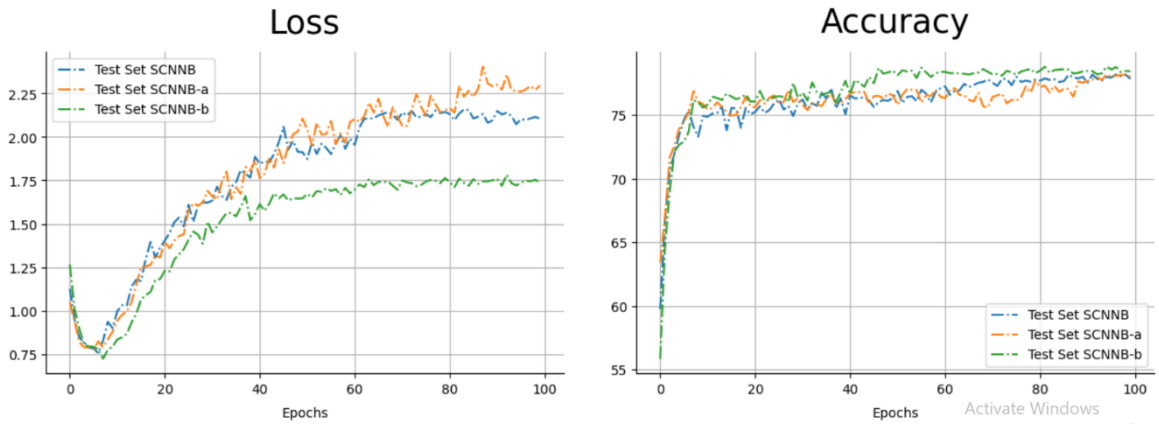
به طور کلی برای دادگان تست برای دسته داده MNIST دقتی در حدود ۹۹.۱ درصد، برای دسته داده FashionMNIST در حدود ۹۲.۳ درصد و برای CIFAR10 در حدود ۷۸ درصد اتخاذ شده است.

برخلاف دسته داده MNIST، خطا در دو دسته داده دیگر بعد از مدتی افزایش پیدا کرده اند که البته بر روی دقت نمودار تاثیری نگذاشته است؛ پس آن را یک مشکل چشمگیر قلمداد نکردیم با اینکه نباید نمودار به شکل های زیر در می آمد.

شکل ۱۰: نمودار خطا و دقت برای دادگان ارزیابی بر دسته داده **MNIST**



شکل ۱۱: نمودار خطا و دقت برای دادگان ارزیابی بر دسته داده **FashionMNIST**



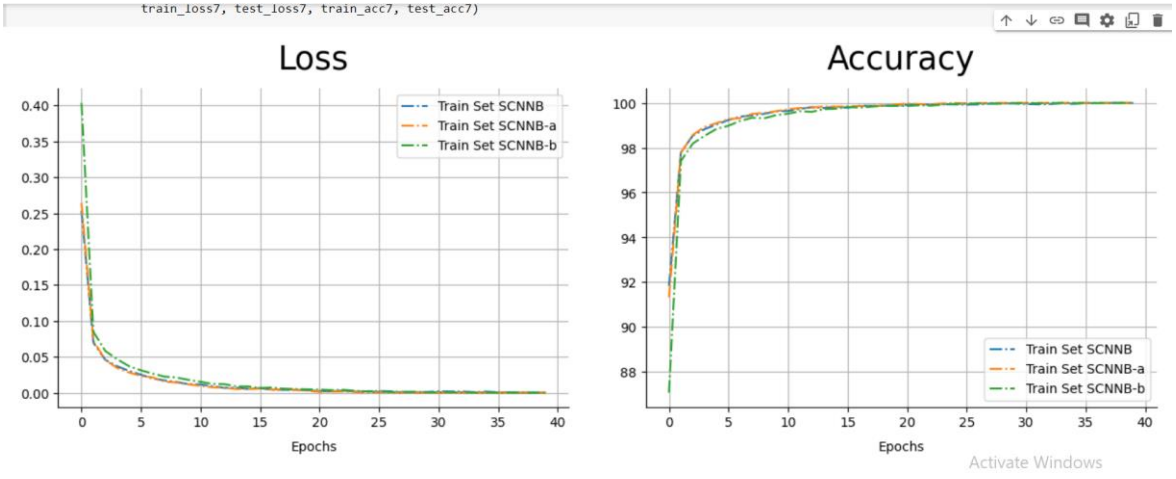
شکل ۱۲: نمودار خطا و دقت برای دادگان ارزیابی بر دسته داده **CIFAR10**

جدول ۱: دقت دادگان تست بر روی سه دسته داده و سه مدل پیشنهاد شده

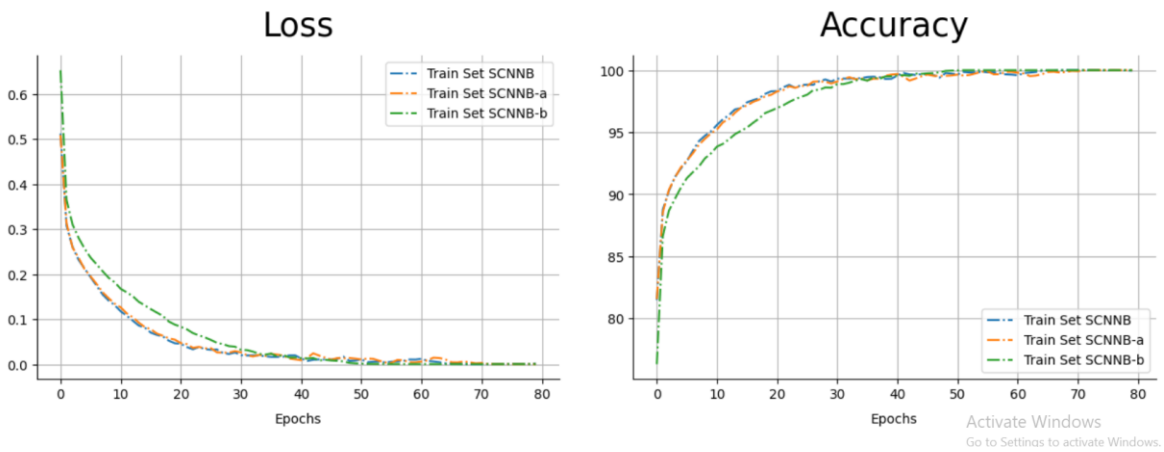
method	MNIST classification test accuracy (%)	Fashion-MNIST classification test accuracy (%)	CIFAR10 classification test accuracy (%)
SCNNB-a	۹۹.۲	۹۲.۶	۷۸
SCNNB-b	۹۹.۱	۹۲.۴	۷۸.۲
SCNNB	99.1	92.3	77.8

به همان دلایلی که بالاتر گفته شد، عملکرد دقیقی با توجه به نکات گفته شده نمیتوان از این جدول برداشت کرد اما با توجه به اینکه درصد های ذکر شده تا حد خیلی خوبی نزدیک به درصدهای مقاله است، بدون در نظر گرفتن مقایسه بین سه مدل گفته شده، میتوان درصدهارا پذیرفت و عملکرد خوبی را برای شبکه غیرعمیق ذکر شده بر سه دسته داده گفته شده در نظر گرفت.

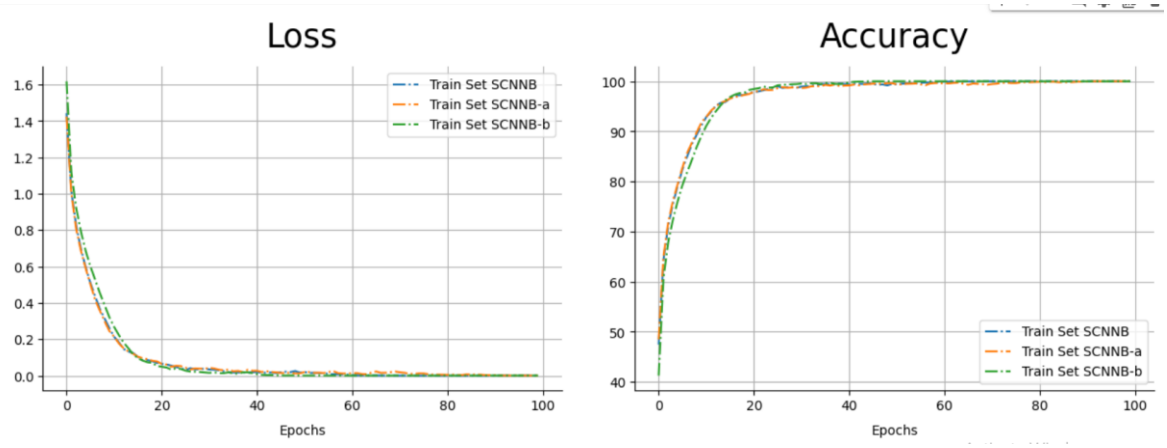
ب: نمودار دقت و خطا برای دادگان آموزش



شکل ۱۳: نمودار خطا و دقت برای دادگان آموزش بر دسته داده **MNIST**



شکل ۱۴: نمودار خطا و دقت برای دادگان ارزیابی بر دسته داده **FashionMNIST**



شکل ۱۵: نمودار خطا و دقت برای دادگان آموزش بر دسته داده **CIFAR10**

ج: تفسیر نتایج

مقداری از نتایج در قسمت الف تحلیل شد و حال به ادامه تحلیل ها میپردازیم:

برای قسمت ب، برای هر سه دسته داده، دقت بعد از مدتی به ۱۰۰ درصد میرسد و این یعنی برای دادگان آموزش این مدل به طور خیلی خوبی عمل کرده و بایاس نداشته و مدل به سختی و پیچیدگی لازم دست پیدا کرده است.

با توجه به اینکه داده های CIFAR10 نسبت به FashionMNIST و FashionMNIST نسبت به MNIST سخت تر هستند، در نتیجه مدل تعریف شده برای داده MNIST دقت بالاتری نسبت به دو دسته داده دیگر پیدا کرده است. همچنین با توجه به این سختی، در دقت آموزش، MNIST سریعتر از دو دسته داده دیگر به دقت بالای ۹۰ درصد میرسد که برای ارزیابی نیز همین گونه است.

با توجه به استفاده از BN باید SCNNB نسبت به SNNB-a و آن نیز نسبت به SCNNB-b باید عملکرد بهتری پیدا میکرد اما با توجه به دلایلی که قبل تر ذکر شد، چنین نتیجه ای کسب نشده است. از BN

برای بالا بردن سرعت training و بهتر کردن نتیجه classification استفاده شده است که با توجه به دلایل قسمت های قبل، این نتایج روی نمودار ها مشاهده نشد.

در هر سه دسته داده، پس از چندین دور(که برای MNIST کمتر از FashionMNIST و آن نیز کمتر از CIFAR10 است) به دقت ۱۰۰ درصد میرسند اما در قسمت ارزیابی هر سه دسته داده دقت پایینتری پیدا کرده که میتوان گفت overfitting رخ داده و مدل نتوانسته همه ویژگی های اصلی را به درستی یاد گرفته و جنرالایز کند. شاید با تعداد داده های بیشتر بتوان دقت بیشتری گرفت. (اینگونه میتوان مقداری از overfit را کاهش داد.) البته این اظهار نظر همواره درست نیست و در بعضی از مسائل به دلیل توضیع موجود میان دادگان کلاس ها یک سقف مشخص برای دقت مدل در دنیای واقعی وجود دارد که به طور کلی هدف ما نزدیک شدن به آن دقت ایده آل است.

پاسخ ۲. طبقه بندی تصاویر اشعه ایکس قفسه سینه

۲-۱- آماده سازی و پیش پردازش داده ها

الف)

دیتاست مورد استفاده در این مقاله از ۵۸۶۳ داده تشکیل شده است (۴۲۷۳ داده مربوط به افراد دارای بیماری و ۱۵۸۳ داده مربوط به افراد سالم) که با توجه به پیچیدگی مسئله و معماری شبکه، تعداد آن کم است. به همین دلیل تقسیم داده ها به سه بخش training ، validation و testing به نحوی انجام شده است که مطمئن باشیم نسبت دو کلاس در همه بخش ها رعایت شده است چرا که اگر علاوه بر کمبود داده یک عدم توازن را نیز به دیتاست وارد کنیم کیفیت طبقه بند پایین تر می رود. نسبت داده ها در این سه بخش به ترتیب ۶۰٪ ، ۲۰٪ و ۲۰٪ می باشد. سپس تصاویر دیتاست به اندازه های ۱۲۸*۱۲۸ پیکسل resize شده اند. برای تقویت دیتاست و جبران کمبود داده ها از روش های افزایش داده استفاده شده است. در این روش با استفاده از تکنیک هایی مانند زوم کردن، چرخاندن، شیفت افقی یا عمودی و ... تصاویر نسبتا جدیدی از روی تصاویر قبلی تولید می شود که می تواند تا حدی از overfit شدن مدل جلوگیری کند و قدرت جنرالیزیشن آن را افزایش دهد. در نهایت تمام دادگان با ضریب ۱/۲۵۵ rescale شده اند که با توجه به بازه عددی هر پیکسل، این کار معادل min-max normalization می باشد.

ب)

فایل "Kaggle.json" را برای استفاده از API از اکانت Kaggle دانلود کرده و در محیط گوگل کولب آپلود می کنیم. سپس با استفاده از دستورات زیر، دیتاست را در محیط گوگل کولب دانلود می کنیم:

```
! mkdir ~/.kaggle
! mv kaggle.json ~/.kaggle/
! chmod 600 ~/.kaggle/kaggle.json
! kaggle datasets download paultimothymooney/chest-xray-pneumonia
! unzip -q 'chest-xray-pneumonia.zip'
```

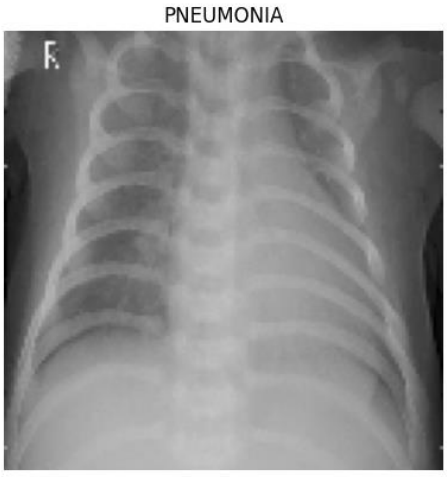
```
Downloading chest-xray-pneumonia.zip to /content
 99% 2.28G/2.29G [00:09<00:00, 233MB/s]
100% 2.29G/2.29G [00:09<00:00, 251MB/s]
```

شکل ۱۶: نحوه دانلود دیتاست در محیط گوگل کولب

سپس با استفاده از opencv تصاویر را لود کرده و به اندازه گفته شده در مقاله یعنی ۱۲۸*۱۲۸ ریسایز کرده و در یک آرایه ذخیره می کنیم. طبق پارامتر rescale ذکر شده داده ها را تقسیم بر ۲۵۵ می کنیم. سپس با استفاده از تابع StraitifiedShuffleSplit از کتابخانه sklearn دیتاست را به سه بخش تقسیم می کنیم به طوری که نسبت دو کلاس در همه بخش ها ثابت بماند. در ادامه مشاهده می کنیم که تعداد داده ها در هر بخش به همان نسبت گفته شده در مقاله می باشد.

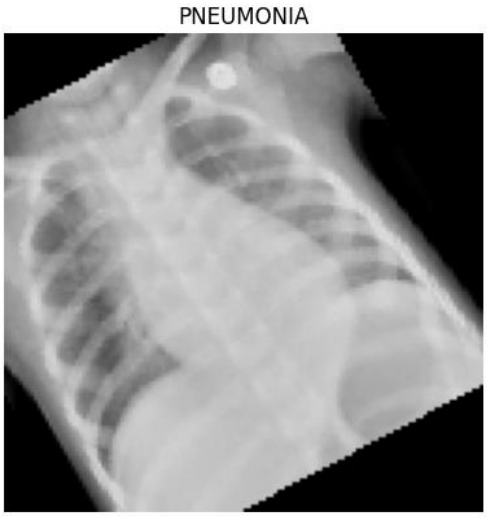
	NORMAL	PNEUMONIA
training:	950	2563
validation:	317	855
testing:	316	855

شکل ۱۷: نسبت داده ها در بخش های مختلف دیتاست



شکل ۱۸: یک نمونه از تصاویر دیتاست

حال با استفاده از ImageDataGenerator در کتابخانه keras بخش data augmentation را پیاده سازی می کنیم. همه پارامتر ها طبق مقاله ست شده اند. پارامتر fill_mode در مقاله ذکر نشده بود که هر دو حالت constant و nearest را امتحان کردیم و استفاده از مقدار constant نتایج را کمی بهتر می کرد. در اینجا هایپر پارامتر batch_size را نیز طبق گفته مقاله ۱۲۸ تنظیم می کنیم.



شکل ۱۹: نمونه ای داده های augment شده

۲-۲- توضیح لایه های مختلف معماری شبکه

EfficientNet یک معماری شبکه عصبی کانولوشنی (CNN) است که با هدف دستیابی به عملکرد بهتر در طبقه بندی تصاویر با استفاده از تعداد پارامترهای کمتر نسبت به معماری های قبلی طراحی شده است. ایده اصلی پشت EfficientNet این است که عمق، عرض و رزولوشن شبکه را به گونه‌ای مناسب مقیاس بندی (scale) کند تا عملکرد دلخواه را داشته باشد. به طور خاص، این معماری از روش Compound Scaling استفاده می‌کند که به طور همزمان عمق، عرض و رزولوشن شبکه را براساس ضرایب از پیش تعیین شده بالا می‌برد.

حال معماری کلی شبکه را بررسی می کنیم که از سه بخش مهم تشکیل شده است:

۱- لایه Stem: این بخش شامل یک لایه کانولوشنی و سپس یک لایه batch normalization و یک تابع فعال‌سازی ReLU است. هدف این لایه استخراج ویژگی های سطح پایین از تصاویر ورودی است.

۲- بلوک های تکرار شونده: این بلوک ها بخش سازنده اصلی معماری EfficientNet هستند. هر بلوک شامل یک سری لایه مانند کانولوشن، batch normalization و توابع فعال سازی می باشد. عمق، عرض و

وضوح هر بلوک براساس ضرایبی که قبل تر گفتیم اسکیل می‌شوند. تعداد این بلوک ها توسط پارامتری به نام depth coefficient تعیین می شود.

۳- Classification head : این بخش لایه نهایی معماری EfficientNet است که شامل یک لایه Global Average Pooling می باشد که به آن یک لایه Fully Connected با تابع فعال سازی softmax متصل شده است. هدف ای لایه نگاشت بردار ویژگی استخراج شده توسط بلوک های تکراری به خروجی طبقه بند و تعیین کلاس مربوطه است.

به طور کلی، EfficientNet با استفاده از یک رویکرد اصولی طراحی شده است که به آن این امکان را می‌دهد که با پارامترها و محاسبات کمتر، دقت بهتری را نسبت به معماری های قدیمی تر در شبکه های عصبی کانولوشنی (CNN) به دست آورد. این امر باعث می‌شود که EfficientNet از نظر منابع محاسباتی بسیار کارآمد بوده و با پیچیدگی کمتر برای کاربردهای واقعی مناسب باشد.

همچنین EfficientNet به طور گسترده بر روی بنچمارک های بزرگ طبقه بندی تصاویر مانند ImageNet آزمایش شده است و همواره عملکرد خوبی از خود نشان داده است. به علاوه ایده این معماری که باعث تعادل توانایی مدل و عملیات محاسباتی مورد نیاز می شود، انتخاب مناسبی برای پژوهشگران است که بتوانند با هزینه محاسباتی کم به دقت خوبی دست پیدا کنند.

البته باید توجه کنیم که در مدل استفاده شده در این مقاله، قسمت Classification head با لایه های زیر جایگزین شده است:

Layer type	Output shape	Param#
global_average_pooling2d	1408	0
Dense (with activation 'RELU')	128	180352
Dropout (30%)	128	0
Dense (with activation 'RELU')	64	8256
Dropout (20%)	64	0
Dense (with activation 'sigmoid')	1	65
Total parameters: 7,957,235		
Trainable parameters: 7,889,667		
Non-trainable parameters: 67,568		

شکل ۲۰: لایه های پایانی شبکه

۲-۳- پیاده سازی شبکه

با استفاده از کتابخانه کراس شبکه گفته شده در مقاله را با استفاده از functional API پیاده سازی می کنیم. مدل EfficientNetB2 را بدون لایه کلسیفیکیشن بالایی آن با وزن های آموزش داده شده روی دیتاست imageNet لود می کنیم. ابعاد تنسور ورودی را طبق مقاله ۱۲۸*۱۲۸ انتخاب می کنیم. سپس لایه های پایانی شبکه را طبق مقاله به انتهای EfficientNetB2 متصل می کنیم.

```
input_layer = keras.layers.Input(shape=(128, 128, 3))
efficientNet = keras.applications.EfficientNetB2(
    include_top=False, weights="imagenet", input_shape=(128, 128, 3))

# efficientNet.trainable = False

x = efficientNet(input_layer, training=False)
x = keras.layers.GlobalAveragePooling2D()(x)
x = keras.layers.Dense(128, activation='relu')(x)
x = keras.layers.Dropout(0.3)(x)
x = keras.layers.Dense(64, activation='relu')(x)
x = keras.layers.Dropout(0.2)(x)
output_layer = keras.layers.Dense(1, activation='sigmoid')(x)

model = keras.Model(input_layer, output_layer)
```

شکل ۲۱: پیاده سازی معماری شبکه

در بخش نتایج مقاله گفته شده است که بهینه ساز Adam با نرخ یادگیری 0.001 توانسته نتیجه خوبی حاصل کند. همچنین batch size نیز همانطور که گفتیم ۱۲۸ در نظر گرفته شده است. Class Weight را نیز برای دو کلاس محاسبه می کنیم تا عدم توازن نمونه های دو کلاس در دیتاست را جبران کنیم.

در مقاله گفته شده است که به تعدادی از لایه های بالایی (انتهایی) شبکه EfficientNetB2 نیز اجازه یادگیری داده اند ولی یادگیری لایه های پایینی را غیر فعال کرده اند. تعداد دقیق این لایه ها ذکر نشده است. انتخاب ما آموزش ۸۰ لایه آخر مدل بود چرا که شبکه EfficientNetB2 حدود ۳۴۰ لایه دارد و با توجه به تفاوت دیتاست ما با دیتاست imageNet ، ایده خوبی است که حداقل ۲۵٪ لایه ها اجازه یادگیری داشته باشند تا فیلتر های سطح بالاتر آپدیت شوند.

همچنین از EarlyStopping و ModelCheckpoint نیز استفاده کردیم ولی به دلیل ناپایداری مدل در گام های ابتدایی یادگیری از استفاده از EarlyStopping صرف نظر کردیم تا مدل فرصت یادگیری بیشتر داشته باشد. همچنین از ModelCheckpoint نیز به علت افزایش زیاد زمان آموزش به علت توقف های مکرر برای ذخیره مدل استفاده نکردیم. البته کد های مربوط به این دو کامنت شده اند و همچنان در کد ها موجود هستند.

مدل را با این پارامتر ها آموزش می دهیم ولی اصلا دقت خوبی به دست نمی آید. به همین جهت با آزمایش هایپر پارامتر های مختلف، با انجام تغییرات زیر توانستیم یک مدل بهبود یافته با دقت مناسب به دست آوریم:

۱- تعداد لایه هایی که اجازه یادگیری داشتند را افزایش می دهیم به طوری که فقط ۷۴ لایه ابتدایی شبکه EfficientNetB2 اجازه یادگیری ندارند و بقیه لایه ها می توانند همراه با آموزش لایه های بالایی Fine Tune شوند. علت این انتخاب طبق توضیحات بخش قبل این است که لایه های Stem که مسئول استخراج ویژگی های سطح پایین است و دو بلوک اول از بلوک های تکرار شونده اجازه یادگیری نداشته باشند. شبکه EfficientNetB2 از ۷ بلوک تکرار شونده تشکیل شده است و در نتیجه ۵ بلوک از این ۷ بلوک اجازه یادگیری دارند و ویژگی های سطح بالاتری را استخراج می کنند. پس قابل توجیه است که فریز کردن مدل را به لایه های ابتدایی محدود کنیم.

۲- نرخ یادگیری را به مراتب کاهش می دهیم چراکه با افزایش لایه ها و پارامتر های قابل آموزش، شبکه می تواند به سرعت ناپایدار یا overfit شود. همچنین از یک Learning Rate scheduler استفاده می کنیم که در دو مرحله نرخ یادگیری را بازهم کمتر می کند.

۳- پارامتر batch size را به ۳۲ کاهش می دهیم تا Convergence سریع تری داشته باشیم.

سپس در ۲۰ epoch مدل را آموزش می دهیم.

باید توجه کنیم که با توجه به پارامتر batch size باید پارامتر step per epoch را در فرآیند آموزش تعیین کنیم چرا که به علت استفاده از ImageDataGenerator تابع فیت از تعداد کل داده ها آگاه نیست و باید تعیین شود که با پیمودن چند قدم یک epoch کامل می شود.

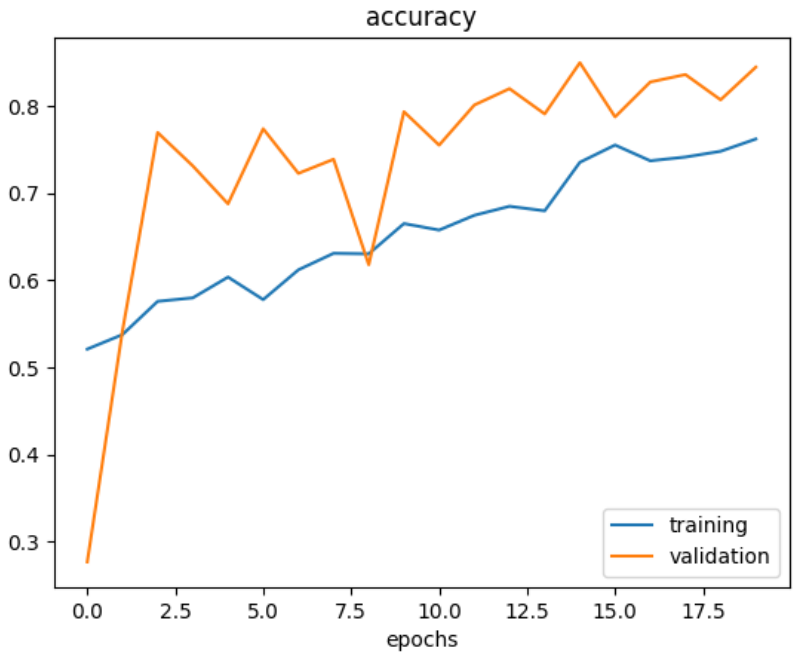
۲-۴- نتیاج پیاده سازی

الف)

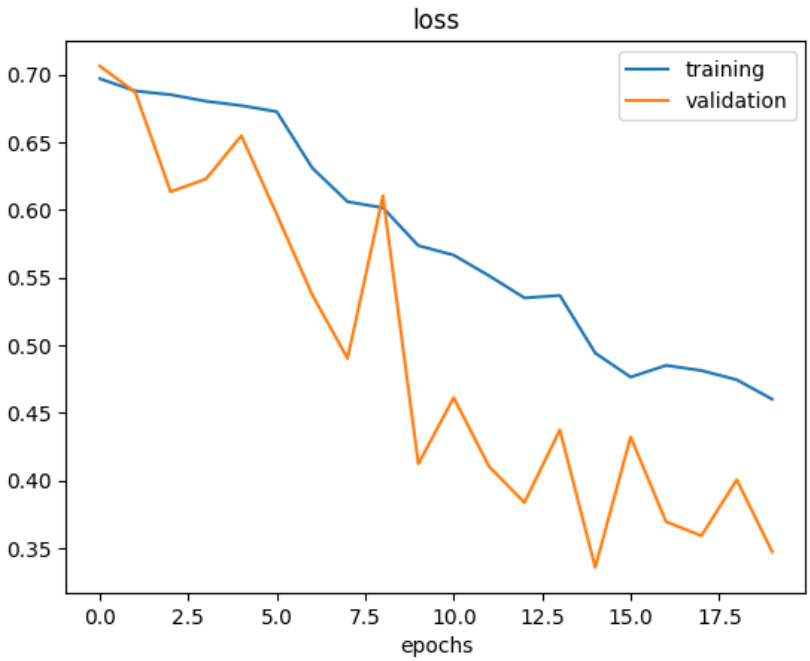
	precision	recall	f1-score	support
0	0.62	0.88	0.73	316
1	0.95	0.80	0.87	855
accuracy			0.82	1171
macro avg	0.78	0.84	0.80	1171
weighted avg	0.86	0.82	0.83	1171

شکل ۲۲: عملکرد مدل روی داده های تست

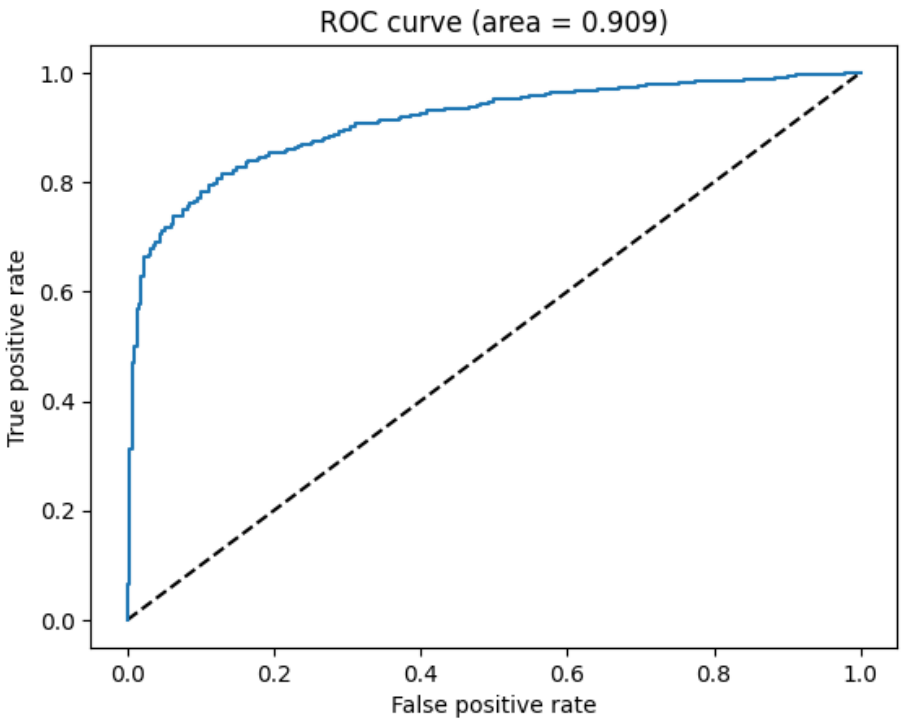
ب)



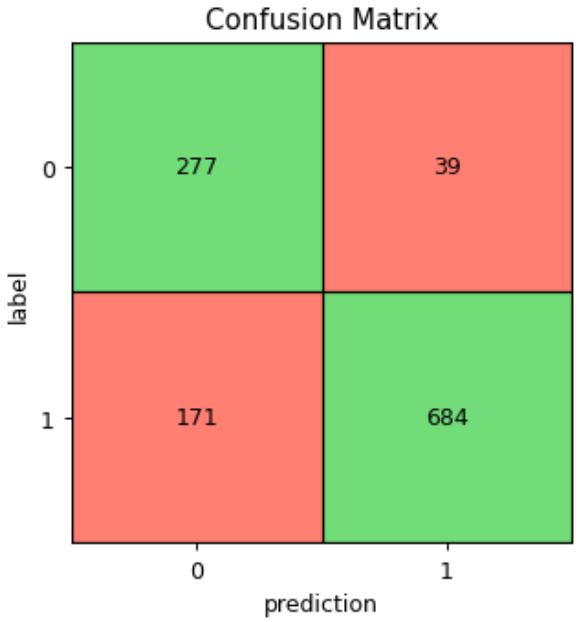
شکل ۲۳: نمودار accuracy در فرآیند یادگیری



شکل ۲۴: نمودار loss در فرآیند یادگیری



شکل ۲۵: ROC Curve



شکل ۲۶: نمودار Confusion Matrix

ج)

مشاهده می کنیم که accuracy مدل روی داده های تست برابر ۸۲٪ می باشد. در نگاه اول متوجه می شویم که نمودار accuracy برای داده های validation بالای منحنی مربوط به داده های training می باشد. این اتفاق در اینجا طبیعی است و علت آن استفاده از data augmentation روی داده های training می باشد چرا که باعث پیچیده تر شدن و اضافه شدن قسمت هایی غیر طبیعی به تصاویر می شود که کار را برای مدل سخت تر می کند چون مدل باید ویژگی های بسیار سطح بالایی را برای طبقه بندی این داده ها استفاده کند. در نتیجه مدل که روی این داده های پیچیده تر آموزش می بیند، در مواجهه با داده های validation که توزیع طبیعی دارند می تواند بسیار موفق تر عمل کند.

مدل روی هر دو کلاس recall خوبی دارد که این نشان دهنده عملکرد خوب مدل می باشد. همچنین مشاهده می کنیم که مقادیر recall دو کلاس کاملا به accuracy مدل نزدیک می باشد و واریانس زیادی ندارد. این یعنی مدل در هر دو کلاس توانسته مهارت تشخیص خوبی پیدا کند.

البته precision برای کلاس ۰ که مربوط به افراد سالم است پایین است و برای کلاس ۱ که مربوط به افراد بیمار است بالا است. در اینجا precision معیار خوبی برای بررسی عملکرد مدل نیست چرا که دیتاست نامتوازن است و اتفاقا توزیع مشابهی با دنیای واقعی هم ندارد که معمولا تعداد بیماران از تعداد افراد سالم کمتر است. پس علت precision پایین برای کلاس ۰ زیادتر بودن داده های متعلق به کلاس ۱ و در نتیجه زیاد بودن false negative ها است و علت presicion زیاد برای کلاس ۱ کمتر بودن داده های متعلق به کلاس ۰ و در نتیجه کم بودن false positive ها است.

در نتیجه بررسی f1 score یا نمودار ROC می تواند منطقی تر باشد. مساحت زیر نمودار ROC نشانگر عملکرد مدل است که هرچه بهتر باشد به ۱ نزدیک تر می شود. در اینجا مساحت زیر نمودار برابر 0.909 می باشد که نشان دهنده عملکرد خوب مدل است.