به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

درس شبکه‌های عصبی و یادگیری عمیق

تمرین ششم

نام و نام خانوادگی	عرفان باقری سولا – محمد قره حسنلو
شماره دانشجویی	810198461 - 810198361
تاریخ ارسال گزارش	1402.4.10

فهرست

پاسخ 1. شبکه های رمزگذار–رمزگشا مولد

1-1 مجموعه داده

شماره دانشجویی من فرد است و در نتیجه از دیتاست CIFAR-10 در طول این سوال استفاده کرده ام. همچنین به عنوان پیش پردازش، دادگان به روش min-max به بازه صفر و یک نرمالایز شده اند. همچنین در ابتدا برای استفاده راحت تر از الگوریتم های کاهش بعد کتابخانه sklearn ، ابعاد داده ها را reshape کرده ام چراکه توابع و کلاس های این مقاله با داده های یک بعدی کار می کنند.

1-2 انجام PCA و Isomap

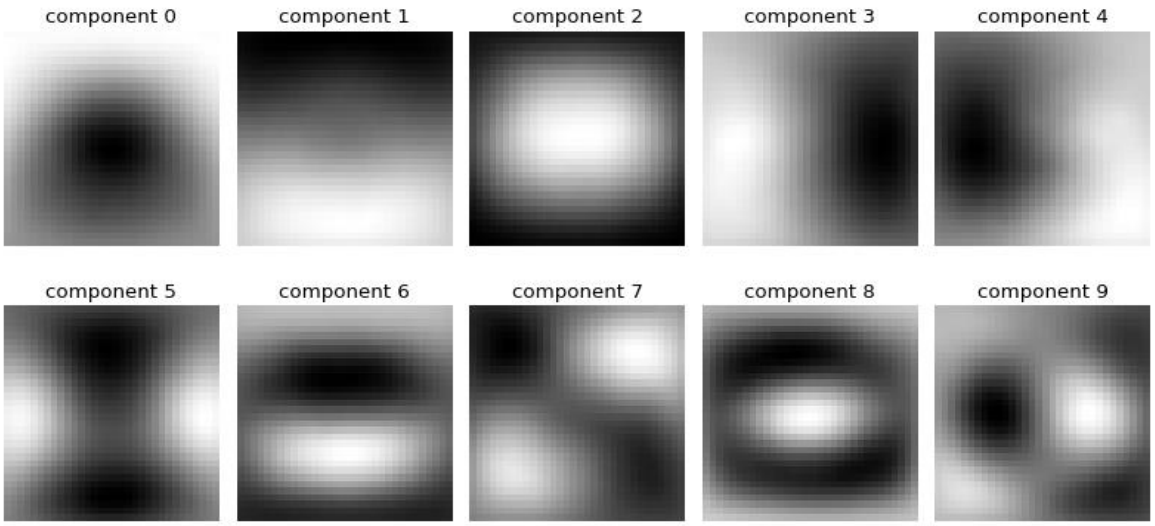
قبل از شروع کار، یک تابع ساده پیاده سازی کرده ام که دادگان را در فضای نهان می گیرد و عملکرد طبقه بند KNN را روی آنها بررسی کرده و Accuracy را گزارش می کند. در ادامه برای بررسی همه متد ها از این تابع استفاده شده است.

همچنین برای پیاده سازی random search نیز یک تابع پیاده سازی کرده ام که با گرفتن کلاس مربوط به متد کاهش بعد از کتابخانه sklearn ، الگوریتم جست و جوی تصادفی را روی پارامتر های ابعاد فضای نهان و تعداد همسایگان مورد بررسی در KNN انجام می دهد. البته چون در الگوریتم هایی مثل Isomap بیشتر بار پردازشی روی محاسبه مپینگ است و زمان زیادی را می برد، تابع طوری پیاده سازی شده است که امکان بررسی چند مقدار مختلف برای تعداد همسایگان پس از یک بار کاهش بعد مشخص وجود داشته باشد. اگر از این ویژگی استفاده نکنیم (پارامتر مربوطه را 1 قرار دهیم) و برای هر مپینگ با ابعاد رندوم فقط یک عدد رندوم هم برای تعداد همسایگان انتخاب کنیم، تابع دقیقا همان random search خواهد بود.

در ادامه از این تابع برای بررسی عملکرد الگوریتم PCA استفاده شده است.

PCA یک روش کاهش بعد است که برای تبدیل مجموعه داده های با بعد بالا به نمایشی با بعد کمتر با حفظ اطلاعات مهم استفاده می‌شود. این روش با شناسایی مؤلفه‌های اصلی، که متغیرهای جدیدی هستند و ترکیب خطی از ویژگی‌های اصلی اولیه هستند و بیشترین واریانس در داده را تشریح می‌کنند، این کار را انجام می‌دهد. مؤلفه اصلی اول، بیشترین میزان واریانس را توضیح می‌دهد و سپس مؤلفه‌های دوم و به همین ترتیب دنبال می‌شوند. با کاهش مؤلفه‌هایی که دارای واریانس کمتری هستند، PCA به ساده‌سازی مجموعه‌داده کمک می‌کند و امکان تصویرسازی، خوشه‌بندی یا طبقه‌بندی را فراهم می‌کند، در حالی که از دسترسی به اطلاعات کمتری صرفه‌جویی می‌کند.

در ادامه 10 مولفه یا component اصلی PCA روی داده های آموزش را مشاهده می کنیم که برای نمایش بهتر سیاه و سفید شده اند.



شکل 1 – 10 مولفه اصلی **PCA**

نتایج PCA پس از کاهش بعد و طبقه بندی با KNN به شکل زیر است:

```
n_components=19, n_neighbors= 8 => accuracy=40.35
n_components=19, n_neighbors=31 => accuracy=41.50
n_components=19, n_neighbors=96 => accuracy=40.30
n_components=45, n_neighbors=42 => accuracy=39.46
n_components=45, n_neighbors=74 => accuracy=38.87
n_components=45, n_neighbors=97 => accuracy=38.44
n_components=75, n_neighbors=11 => accuracy=39.71
n_components=75, n_neighbors=30 => accuracy=38.51
n_components=75, n_neighbors=60 => accuracy=37.32
n_components=91, n_neighbors=22 => accuracy=38.51
n_components=91, n_neighbors=36 => accuracy=37.70
n_components=91, n_neighbors=58 => accuracy=36.84
```

شکل 2 – نتایج طبقه بندی در فضای نهان **PCA**

همانطور که گفتم برای جست و جوی سریعتر برای هر عدد رندوم به عنوان تعداد مولفه های PCA ، یکبار PCA آموزش داده شده است و سپس سه مقدار رندوم K نیز آزمایش شده است. بیشترین دقت به دست آمده برابر 41.5% می باشد که با بهترین دقت مقاله یعنی 42.35% قابل مقایسه است و منطقی به نظر می رسد.

Isomap یک روش کاهش بعد است که مثل PCA برای تبدیل مجموعه داده های با بعد بالا به فضای با بعد پایین تر با حفظ ساختار کلی از طریق تحلیل نزدیک‌ترین همسایگی مورد استفاده قرار می‌گیرد. این روش با استفاده از ماتریس فاصله گراف، که فاصله‌های جغرافیایی روی توزیع واقعی بین نقاط نمونه را نمایش می‌دهد، توسط الگوریتم های کاهش بعد مانند مقیاس‌بندی، داده ها را از فضای با بعد بالا به فضای با بعد پایین مپ می کند به طوری که در این فضای نهان، فاصله های کارتزین به نوعی نشان دهنده فاصله جغرافیایی در فضای اصلی باشند. در واقع Isomap زیر مجموعه manifold learning است. ایزومپ اطلاعات مهم درباره ساختار و توزیع داده ها در فضای اصلی را حفظ می‌کند و برای تصویرسازی و دسته‌بندی داده‌ها مفید است.

به خاطر زمان بر بودن این روش، در این بخش فقط از 10٪ داده ها استفاده شده است. پس از کاهش بعد با متد Isomap و طبقه بندی با KNN نتایج زیر به دست آمد:

```
n_components=9, n_neighbors= 1 => accuracy=18.50
n_components=9, n_neighbors=13 => accuracy=25.20
n_components=9, n_neighbors=69 => accuracy=25.30
n_components=9, n_neighbors=99 => accuracy=26.50
n_components=23, n_neighbors=36 => accuracy=26.30
n_components=23, n_neighbors=51 => accuracy=27.10
n_components=23, n_neighbors=85 => accuracy=27.10
n_components=23, n_neighbors=89 => accuracy=26.60
n_components=27, n_neighbors=45 => accuracy=27.10
n_components=27, n_neighbors=89 => accuracy=25.90
n_components=27, n_neighbors=93 => accuracy=25.60
n_components=27, n_neighbors=95 => accuracy=26.30
```

شکل 3 – نتایج طبقه بندی در فضای نهان **Isomap**

بیشترین دقت به دست آمده برابر 27.10% می باشد که کمتر از بیشترین دقت مقاله یعنی 33.61 می باشد ولی با توجه به اینکه فقط از بخش کوچکی از مجموعه داده استفاده کردیم می توان این موضوع را توجیه کرد.

باید توجه کنیم که برای ارزیابی مدل ها از همان داده های تست استفاده شده و روش k-fold به علت اینکه زمان مورد نیاز را تقریبا 5 برابر می کند استفاده نشده است.

3-1 رمزگذار – رمزگشا

شبکه های گفته شده را ساخته و آموزش می دهیم. چون در صورت سوال خواشته نشده در این قسمت از رندوم سرچ استفاده کنیم، برای راحتی کار چند عدد تنظیم شده به صورت دستی را امتحان می کنیم.

نتایج را در ادامه مشاهده می کنیم:

```
latent_dim=40, n_neighbors= 5 => accuracy=39.09
latent_dim=40, n_neighbors=10 => accuracy=41.44
latent_dim=40, n_neighbors=15 => accuracy=41.93
latent_dim=40, n_neighbors=25 => accuracy=42.31
latent_dim=40, n_neighbors=35 => accuracy=42.27
latent_dim=40, n_neighbors=50 => accuracy=42.17
```

شکل 4 – نتیجه طبقه بندی روی فضای نهان **Autoencoder** با لایه های **Dense** و فضای پسین 40

```
latent_dim=80, n_neighbors= 5 => accuracy=42.04
latent_dim=80, n_neighbors=10 => accuracy=43.89
latent_dim=80, n_neighbors=15 => accuracy=44.64
latent_dim=80, n_neighbors=25 => accuracy=44.91
latent_dim=80, n_neighbors=35 => accuracy=44.66
latent_dim=80, n_neighbors=50 => accuracy=44.31
```

شکل 5 – نتیجه طبقه بندی روی فضای نهان **Autoencoder** با لایه های **Dense** و فضای پسین 80

```
latent_dim=30, n_neighbors= 5 => accuracy=39.14
latent_dim=30, n_neighbors=10 => accuracy=40.62
latent_dim=30, n_neighbors=15 => accuracy=40.73
latent_dim=30, n_neighbors=25 => accuracy=40.84
latent_dim=30, n_neighbors=35 => accuracy=40.69
latent_dim=30, n_neighbors=50 => accuracy=40.28
```

شکل 6 – نتیجه طبقه بندی روی فضای نهان **Autoencoder** با لایه های **Convolutional** و فضای پسین 30

```
latent_dim=90, n_neighbors= 5 => accuracy=41.53
latent_dim=90, n_neighbors=10 => accuracy=41.66
latent_dim=90, n_neighbors=15 => accuracy=42.02
latent_dim=90, n_neighbors=25 => accuracy=40.98
latent_dim=90, n_neighbors=35 => accuracy=40.74
latent_dim=90, n_neighbors=50 => accuracy=39.77
```

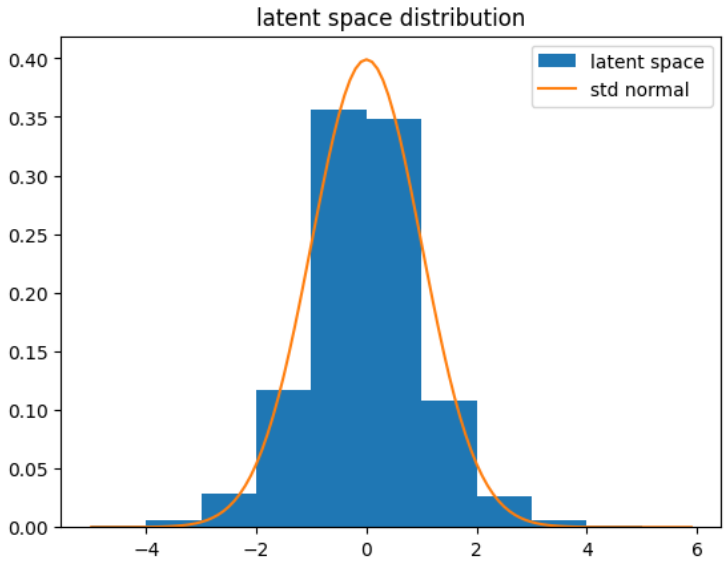
شکل 7 – نتیجه طبقه بندی روی فضای نهان **Autoencoder** با لایه های **Convolutional** و فضای پسین 90

مشاهده می کنیم که بیشترین دقت مربوط به Autoencoder با لایه های Dense می باشد که برابر 44.91% است و خیلی نزدیک به بیشترین دقت مقاله در این بخش یعنی 45.77% می باشد. در این معماری از لایه های Dropout به عنوان رگولاریزیشن استفاده شده است که عملکرد مدل را بهبود می بخشد. برای Autoencoder با لایه های Convolution نیز از لایه های Batch Normalization به عنوان رگولاریزیشن استفاده شده است. مشاهده می کنیم که علی رغم reconstruction loss پایین تر در مدل Convolutional ، در طبقه بندی مدل های Dense عملکرد بهتری داشتند.

4-1 خود رمزگذار متغیر

خود رمزگذار های متغیر (VAE) مدل‌های تولیدی هستند که با ترکیب یادگیری عمیق و مدل‌سازی احتمالی، فرایند رمزنگاری و رمزگشایی داده‌ها را یاد می‌گیرند. ایده اصلی پشت VAE ها تبدیل داده‌های ورودی به یک فضای پسین با بعد کم است، به گونه‌ای که هر نقطه در این فضا نمایانگر یک ترکیب مختلف از داده ها است. VAE ها فضای پسین را به عنوان یک توزیع، به طور معمول یک توزیع گاوسی، مدل می‌کنند. در طول فرایند آموزش، VAE ها سعی می کنند داده های اصلی را دقیقا بازسازی کنند و در عین حال، توزیع فضای پسین را به یک توزیع پیشین مورد نظر، معمولا یک نرمال استاندارد، نزدیک کنند. این کار باعث می‌شود VAE ها یاد بگیرند با نمونه برداری از فضای پسین، داده‌های جدیدی تولید کنند. تابع هزینه در فرایند آموزش با استفاده از ترکیبی از خطای بازسازی و انحراف KL معرفی می‌شود که به کنترل ساختار فضای پسین کمک می‌کند.

برای اطمینان از صحت عملکرد، پس از آموزش اولین VAE ، توزیع فضای پسین را برای داده های آموزش در آن رسم می کنیم که به شکل زیر است:



شکل 8 – توزیع فضای پسین **VAE**

در حالت ایده آل، VAE سعی می کند همه میانگین ها را در ابعاد مختلف به 0 نزدیک کند، ولی برای اینکه مدل قدرت تفکیک داشته باشد و خطای بازسازی را کاهش دهد، نمی تواند همه میانگین ها را دقیقا 0 بکند، به همین علت خود میانگین ها نیز یک توزیع شبیه توزیع نرمال اطراف 0 دارند. شکل بالا به خوبی نتیجه استفاده از خطای KL divergence را نمایش می دهد. چراکه بر خلاف VAE ، در AE ها توزیع فضای پسین بسیار پراکنده و Sparse می باشد.

همچنین در این قسمت حتی برای مدل با لایه های Dense نیز از Batch Normalization استفاده شده است چراکه استفاده از Dropout فرآیند آموزش VAE را ناپایدار می کرد و نتوانستم با آن مدل ها را آموزش دهم. ولی BN این مشکل را ندارد.

```
latent_dim=40, n_neighbors= 5 => accuracy=41.60
latent_dim=40, n_neighbors=10 => accuracy=43.03
latent_dim=40, n_neighbors=15 => accuracy=43.36
latent_dim=40, n_neighbors=25 => accuracy=43.61
latent_dim=40, n_neighbors=35 => accuracy=43.35
latent_dim=40, n_neighbors=50 => accuracy=42.92
```

شکل 9 - نتیجه طبقه بندی روی فضای نهان **VAE** با لایه های **Dense** و فضای پسین 40

```
latent_dim=70, n_neighbors= 5 => accuracy=41.15
latent_dim=70, n_neighbors=10 => accuracy=42.80
latent_dim=70, n_neighbors=15 => accuracy=42.43
latent_dim=70, n_neighbors=25 => accuracy=42.55
latent_dim=70, n_neighbors=35 => accuracy=42.40
latent_dim=70, n_neighbors=50 => accuracy=41.85
```

شکل 10 - نتیجه طبقه بندی روی فضای نهان **VAE** با لایه های **Dense** و فضای پسین 70

```
latent_dim=50, n_neighbors= 5 => accuracy=42.21
latent_dim=50, n_neighbors=10 => accuracy=43.04
latent_dim=50, n_neighbors=15 => accuracy=42.79
latent_dim=50, n_neighbors=25 => accuracy=41.95
latent_dim=50, n_neighbors=35 => accuracy=41.61
latent_dim=50, n_neighbors=50 => accuracy=40.56
```

شکل 11 - نتیجه طبقه بندی روی فضای نهان **VAE** با لایه های **Convolutional** و فضای پسین 50

```
latent_dim=90, n_neighbors= 5 => accuracy=40.18
latent_dim=90, n_neighbors=10 => accuracy=40.69
latent_dim=90, n_neighbors=15 => accuracy=40.13
latent_dim=90, n_neighbors=25 => accuracy=38.75
latent_dim=90, n_neighbors=35 => accuracy=38.18
latent_dim=90, n_neighbors=50 => accuracy=37.36
```

شکل 12 - نتیجه طبقه بندی روی فضای نهان **VAE** با لایه های **Convolutional** و فضای پسین 90

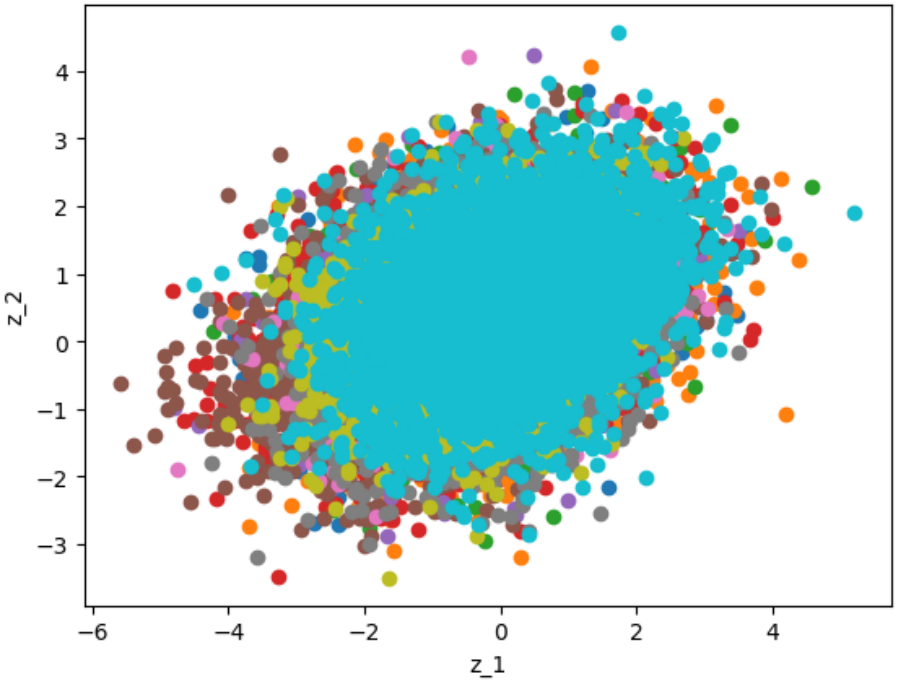
در اینجا هم بیشترین دقت مربوط به مدل با لایه های Dense و برابر 43.61% می باشد با بهترین دقت مربوط به مقاله یعنی 44.39% قابل مقایسه می باشد. اینجا هم تابع هزینه برای مدل های Convolutional بسیار کمتر است ولی با اینحال فضای پسین آنها برای طبقه بندی ضعیف تر است. مدل آخر با فضای پسین 90 تابع هزینه را به نسبت بقیه مدل ها بهتر پایین آورده است و به همین خاطر بیشتر از بقیه و 20 epoch آموزش داده شده است تا در بخش بعد از آن استفاده کنیم.

5-1 کاوش در فضای latent

شکل 13 – یک گرید از تصاویر تولید شده با پیمایش در فضای پسین

شکل بالا سمت راست یک اسب، بالا سمت چپ یک هواپیما، پایین سمت راست یک قورباغه و پایین سمت چپ یک truck می باشد.

در ادامه Scatter Plot داده های آموزش را در فضای نهان رسم می کنیم. مشاهده می شود که تمام نقاط در هم تنیده هستند و این نمودار برای چند جفت از ابعاد دیگر هم که امتحان کردم همین گونه بود. این یعنی توزیع کلاس ها در فضای نهان بسیار به هم نزدیک هستند و این یعنی VAE به خوبی آموزش دیده است و ما میتوانیم به راحتی در این فضای نهان بین کلاس ها کاوش کنیم.



شکل 14 – **Scatter Plot** داده های آموزش در فضای نهان

1-6 Diffusion Models

مدل های دیفیوژن به این صورت عمل می کنند که توضیع احتمال یک تصویر با نویز کمتر را به شرط یک ورودی با نویز بیشتر یاد می گیرند. با استفاده زنجیره از این توابع احتمال، در نهایت قادر هستیم که توزیع احتمال را در فضای اصلی مدل کنیم. در عمل این مدل ها به این صورت کار می کنند که در فرآیند آموزش به یک تصویر مثلا 1000 بار نویز (گاوسی) می دهند به صورتی که بار هزارم فقط یک تصویر با نویز خالص می ماند. سپس یاد میگیرند هر بار یک تصویر نویزی را در ورودی گرفته و یک قدم دینویز کنند. پس از فرآیند آموزش، مدل قادر است با گرفتن یک نویز گاوسی به عنوان ورودی، پس از 1000 بار دینویز کردن آن یک تصویر جدید تولید کند. در واقع در طول آموزش، مدل یاد می‌گیرد که مراحل دیفیوژن را معکوس کند و توزیع داده اصلی را بازسازی کند. با انجام این عمل، مدل می‌تواند با اعمال فرآیند معکوس دیفیوژن یادگرفته شده به نویز تصادفی، نمونه‌های جدیدی تولید کند. این مدل ها توانایی بسیار بالایی در یادگیری توزیع های پیچیده دارند و می توانند خروجی های متنوع و با کیفیت تولید کنند. مشکل این مدل ها کند بودن آنها به دلیل فرآیند تکراری در آنهاست.

تفاوت این مدل ها با VAE در اینجاست VAE از یک فضای با بعد کمتر سعی می کند نمونه ها را تولید کند در صورتی که در مدل ها دیفیوژن فضای پسین هم سایز با فضای اصلی است و همچنین فضای پسین معنای خاصی را ندارد که مثلا بتوان از آن برای کاهش بعد یا طبقه بندی استفاده کرد. به بیان دیگر فضای پسین یک نوسز خالص است و هیچ کوریلیشنی با داده های اصی ندارد.

همچنین VAE کار تولید نمونه را در یک گام انجام می دهد در صورتی که مدل های دیفیوژن این کار را در تعداد گام های زیاد انجام می دهند. خروجی های VAE معمولا به خاطر استفاده از تابع هزینه mse حالت blur دارد ولی خروجی مدل های دیفیوژن بسیار با کیفیت هستند.

پاسخ 2. شبکه ی متخاصم مولد

2.1: بارگذاری دادهها و شبکه ی ResNet

برای این قسمت از پیش پردازش های زیر استفاده کرده ایم.

```
# preprocessing
data_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomVerticalFlip(p=0.5),
    transforms.RandomRotation(degrees=15),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),
    transforms.RandomResizedCrop(size=28, scale=(0.8, 1.0)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5], std=[0.5])
])

# load the data
train_dataset = DataClass(split='train', transform=data_transform, download=True)
val_dataset = DataClass(split='val', transform=data_transform, download=True)
test_dataset = DataClass(split='test', transform=data_transform, download=True)

# encapsulate data into dataloader form
train_loader = data.DataLoader(dataset=train_dataset, batch_size=BATCH_SIZE, shuffle=True)
val_loader = data.DataLoader(dataset=val_dataset, batch_size=2*BATCH_SIZE, shuffle=False)
test_loader = data.DataLoader(dataset=test_dataset, batch_size=2*BATCH_SIZE, shuffle=False)
```

شکل 1: پیش پردازش ها و لود کردن دیتاست ها در قسمت **resnet**

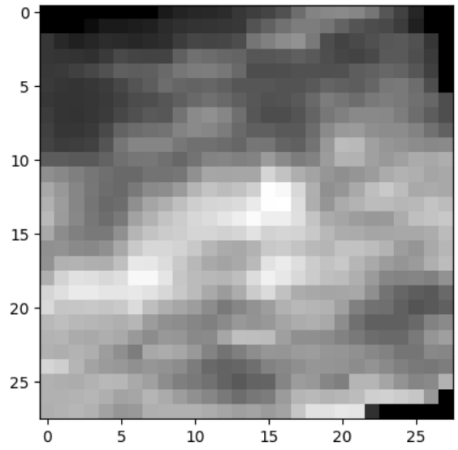
1. RandomHorizontalFlip: با احتمال ۰,۵، تصویر ورودی به صورت افقی برعکس می شود.
2. RandomVerticalFlip: با احتمال ۰,۵، تصویر ورودی به صورت عمودی برعکس می شود.
3. RandomRotation: تصویر به صورت تصادفی در یک زاویه مشخص (۱۵ درجه) چرخش داده می شود.
4. ColorJitter: با اعمال تصادفی تغییرات در روشنایی، کنتراست، اشباع رنگ و رنگ‌های تصویر، تصویر ورودی تغییر می کند.

5. RandomResizedCrop: تصویر ابتدایی برای آموزش بعد از تصویر برش داده شده به اندازه تصادفی و مشخص شده (۲۸ در ۲۸ پیکسل) با نسبت تصادفی (از ۰,۸ تا ۱) اندازه تصادفی شده است.
6. ToTensor: تصویر به فرمت تانسور تبدیل می شود.
7. Normalize: تصویر با میانگین و انحراف معیار 0.5 (برای تصاویر خاکستری) نرمال شده است.

نمونه عکس کوجود در دیتاست BreastMNIST در شکل زیر آورده شده است.

```
print(f'Following is image from class {train_dataset[0][1][0]}') # Image with index 0 had label 0, so I used train_dataset[0][1][0]

plt.imshow(train_dataset[0][0].reshape((28, 28)), cmap='gray')
plt.show()
```

```
Following is image from class 1
```



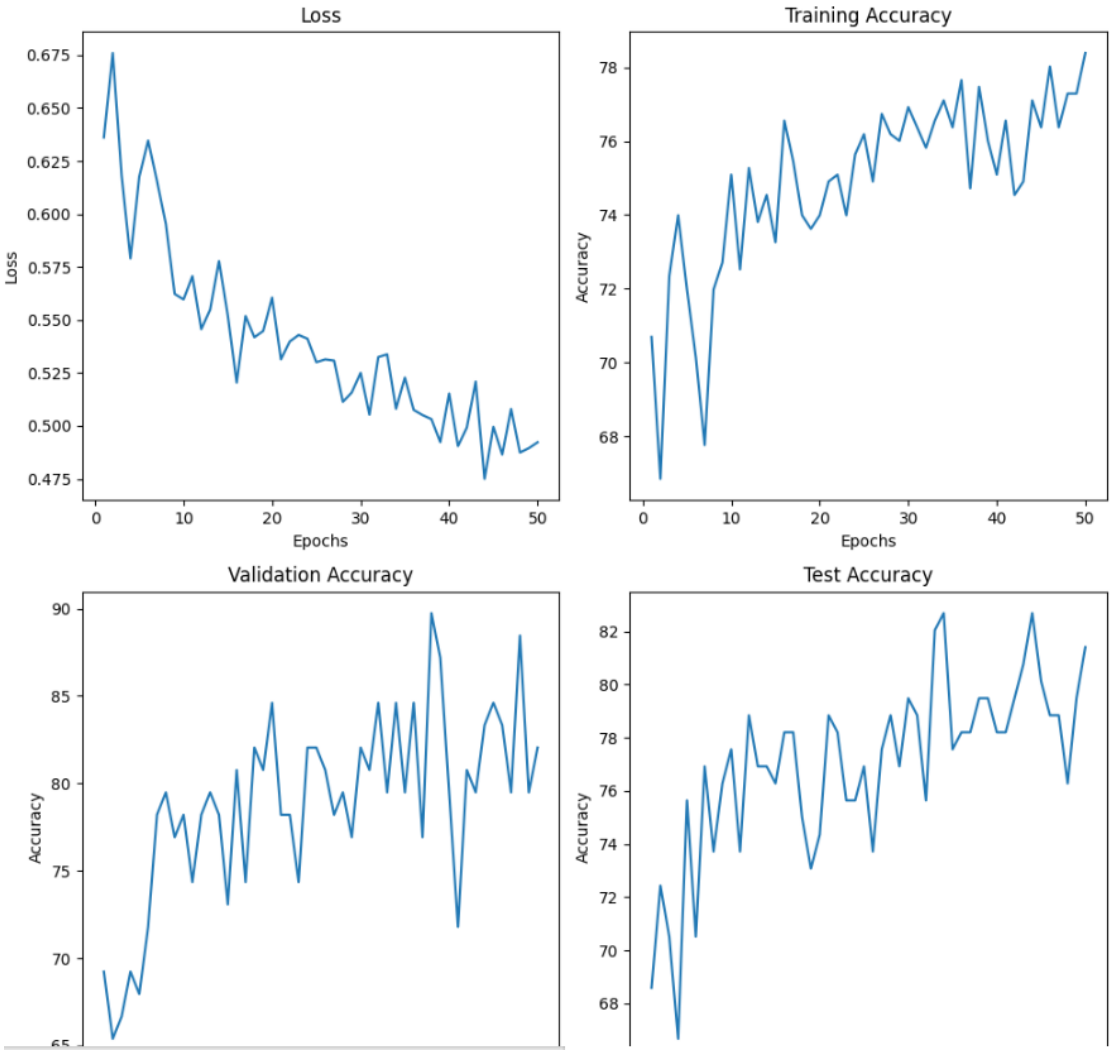
شکل 2: نمونه ای از **BreastMNIST**

با استفاده از متود montage دسته ای از خروجی ها به شکل زیر هستند.

شکل 3: دسته ای از نمونه های **BreastMNIST**

گفته شده که از شبکه resnet برای آموزش استفاده شود که ما شبکه را به شکل زیر ئرآوردیم و یک لایه کانولوشن برای یادگیری با کرنل سایز 7 و استراید 2 اضافه کردیم و در نهایت آن را به خروجی با تعداد کلاسها وصل کردیم. در اینجا از Adam و CrossEntropy استفاده کردیم، مقدار نرخ یادگیری را برابر 0.0001(پس از کلی آزمون و خطا این مقدار انتخاب شده است.)، batch size را به دلیل پایین بودن تعداد نمونه ها برابر 8 برای train set و 16 برای val set و test set در نظر گرفتیم و در آخر در 50 ایپاک آن را آموزش دادیم.

نمودار های loss و دقت داده آموزش، اعتبارسنجی و ارزیابی به شکل زیر در می آیند.



شکل 4: نمودار دقت در دادگان آموزش، اعتبارسنجی و ارزیابی و خطا با شبکه **resnet** تغییر یافته

همانطور که مشخص است، مقدار خطا به مرور زمان کم شده است(با اینکه مقدار نوسان دارد و این نوسان حتی قبل از تغییرات زیادی بسیار بیشتر بود)، و همچنین مقدار دقت در هر سه دسته داده افزایش یافته است که برای دادگان ٔموزش به 78 درصد، دادگان اعتبارسنجی به 83 درصد و دادگان ارزیابی به 80 درصد شده است که نشان دهنده عملکرد نسبتا خوب در تشخیص عکس ها میباشد.(با اینکه عکس ها بسیار نامشخص است، عملکرد خوبی دارد.)

ماتریس آشفتگی در شکل زیر قابل مشاهده است.

```python
# Calculate test accuracy and confusion matrix
correct = 0
total = 0
predictions = []
true_labels = []

with torch.no_grad():
    for images, labels in test_loader:
        images = images.to(device)
        labels = labels.view(-1).to(device)

        outputs = resnet(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

        predictions.extend(predicted.tolist())
        true_labels.extend(labels.tolist())

test_accuracy.append(100 * correct / total)

# Calculate confusion matrix
confusion_mat = confusion_matrix(true_labels, predictions)

print(f"Epoch {epoch+1}/{NUM_EPOCHS} - Loss: {total_loss[-1]} - Train Accuracy: {train_accuracy[-1]:.2f}% - Val Accuracy: {val_accuracy[-1]:.2f}% - Test Ac
print("Confusion Matrix:")
print(confusion_mat)
```

```
Epoch 50/50 - Loss: 0.49230653501075244 - Train Accuracy: 78.39% - Val Accuracy: 82.05% - Test Accuracy: 83.33%
Confusion Matrix:
[[ 21  21]
 [  5 109]]
```

شکل 5: ماتریس آشفتگی

همانطور که مشخص است، ماتریس آشفتگی به خوبی تواسته TN هارا تشخیص دهند و آنهایی که سالم بودند به خوبی تشخیص داده شده اند و تنها 21 نمونه اشتباه دارای مریضی تشخیص داده شده است در حالی که نرمال هستند. همچنین در قسمت دیگر قضیه، 21 نمونه به درستی TP تشخیص داده شده و تنها 5 نمونه نرمال تشخیص داده شده است در حالی که بیماری دارد.

2.2: شبکه Conditional DCGAN

این قسمت پروژه در دو قسمت انجام شده است، در قسمت اول cGAN را پیاده سازی کردیم و نتیج آن را آوردیم و پس از آن معماری conditional DCGAN را کامل و عین مقاله با نتایج آن آورده ایم.

ابتدا به cGAN به طور خلاصه میپردازیم(به نام فایل P2.ipynb) ولی قسمت conditional DCGAN(به نام فایل Q2.ipynb) را به طور کامل آورده ایم.

cGAN(بر گرفته از سایت keras):

در اینجا ابتدا چیش چردازش های زیر را انجام میدهیم که دیتاست را به numpy می آوریم و خروجی را به شکل one hot در می آوریم و آن را به سکل عکس های 28*28 دارای یک کانال در می آوریم و سپس تقسیم بر 255 میکنیم.

```python
def transform_to_numpy(dataset):

    X, y = [], []
    for img, label in dataset:
        img = np.array(img)
        X.append(img)
        y.append(label)

    y = np.array(y).reshape((-1,))
    y = keras.utils.to_categorical(y)
    X = np.array(X).reshape((-1, 28, 28, 1))
    return X / 255, y
```

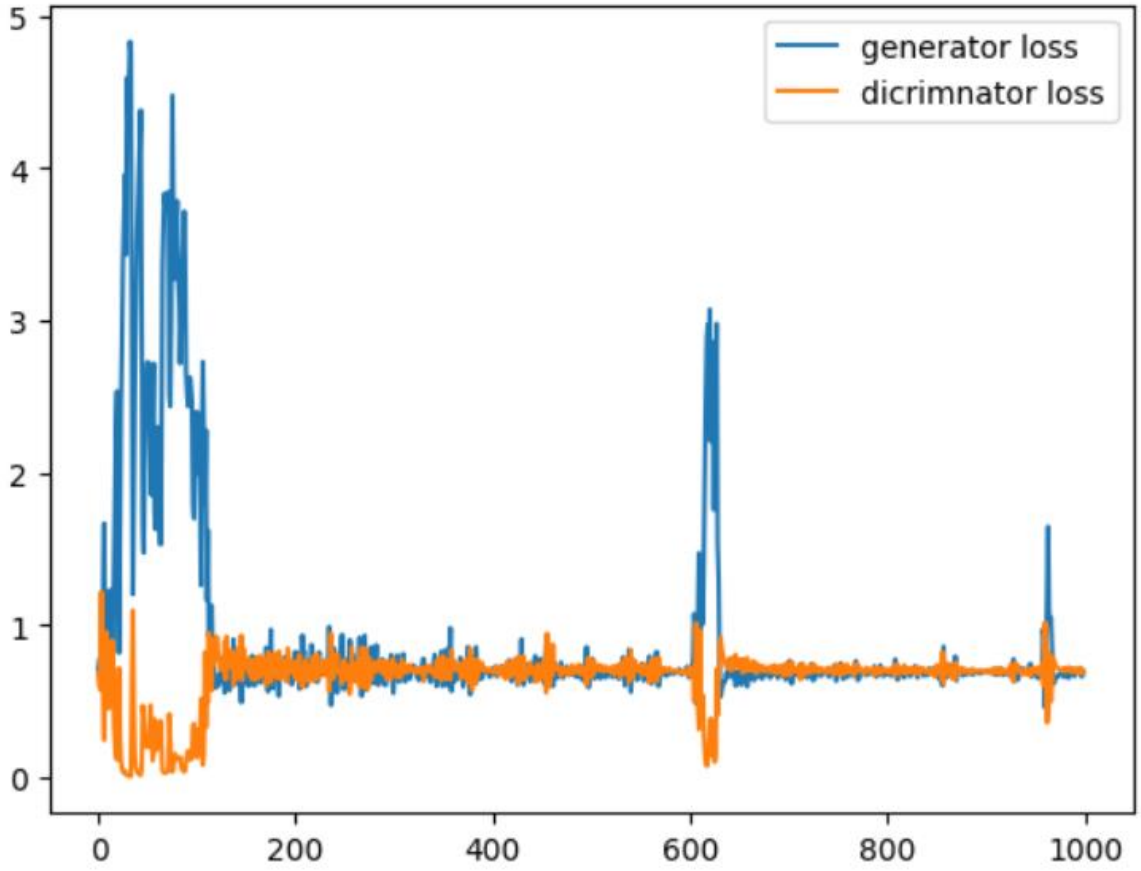
شکل 6: پیش پردازش های انجام شده در **cGAN**

معماری های generator و discriminator را به شکل زیر تعریف میکنیم:

```python
discriminator = keras.Sequential(
    [
        keras.layers.InputLayer((28, 28, discriminator_in_channels)),
        layers.Conv2D(64, (3, 3), strides=(2, 2), padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Conv2D(128, (3, 3), strides=(2, 2), padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.GlobalMaxPooling2D(),
        layers.Dense(1),
    ],
    name="discriminator",
)

# Create the generator.
generator = keras.Sequential(
    [
        keras.layers.InputLayer((generator_in_channels,)),
        # We want to generate 128 + num_classes coefficients to reshape into a
        # 7x7x(128 + num_classes) map.
        layers.Dense(7 * 7 * generator_in_channels),
        layers.LeakyReLU(alpha=0.2),
        layers.Reshape((7, 7, generator_in_channels)),
        layers.Conv2DTranspose(128, (4, 4), strides=(2, 2), padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Conv2DTranspose(128, (4, 4), strides=(2, 2), padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Conv2D(1, (7, 7), padding="same", activation="sigmoid"),
    ],
    name="generator",
)
```

شکل 7: معماری های **generator** و **discriminator** در **cGAN**

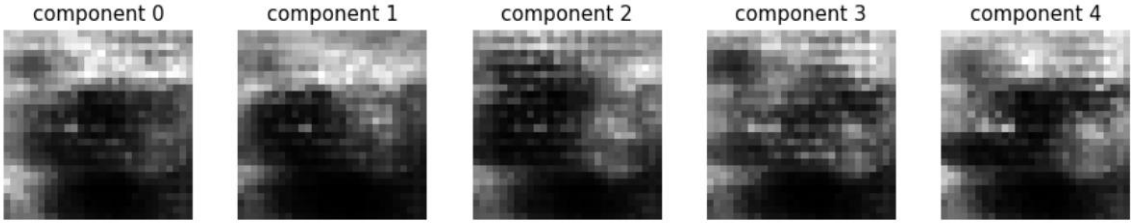
حال با استفاده از این دو، معماری cGAN را تعریف میکنیم و برای پارامترها به این شکل عمل میکنیم که learning rate را برای هر دو برابر 0.0003، لاس را برابر cross entropy و تعداد ایپاک را برابر 1000 میگذاریم. نمودار لاس این دو به شکل زیر در می آید:



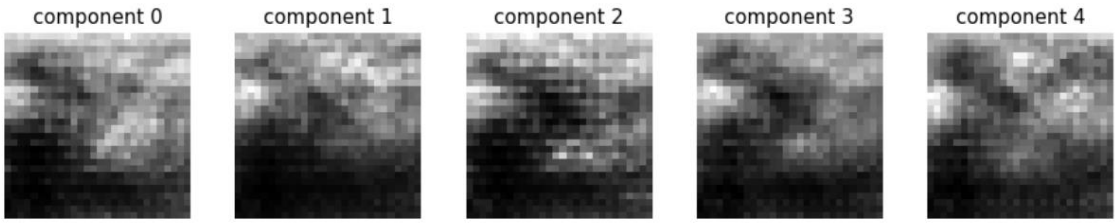
شکل 8: نمودار خطا در **generator** و **discriminator**

همانطور که مشخص است، لاس هر دو همگرا شده و به هم نزدیک میشوند و تقریبا برابر 0.7 شده اند. نزدیک بودن این دو نشان دهنده درست کار کردن شبکه است و مولد بعد از 1000 ایپاک توانسته داده هایی تولید کند که تفکیک کننده نسبت به ابتدا fake و real را تشخیص میدهد.

نمونه های تولید شده برای کلاس صفر به شکل زیر است:



شکل 9: نمونه های کلاس صفر با **cGAN**



شکل 10: نمونه های کلاس یک با **cGAN**

توضیحات کامل تر و جواب تک تک سوالات در قسمت بعد آورده شده است اما آوردن cGAN هم خالی از لطف برای مطمِن شدن چرایی عملکرد conditional DCGAN خالی از لطف نبود.

conditional DCGAN

شبکه خواسته شده کاملا مطابق با مقاله Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks نوشته شده است. این شبکه تغییراتی نسبت به مدل های قبل داشته است و تغییرات اساسی که در پیاده سازی هم انجام شده است عبارت است از:

1- همه pooling ها با strided convolution ها جایگزین شده اند.
2- از batchnorm در generator و discriminator استفاده شده است.
3- لایه های fc حذف شده اند.
4- برای generator به جز لایه آخز که از tanh استفاده شده است، از ReLU برای activation function استفاده شده است.
5- برای همه لایه های discriminator از LeakyReLU استفاده شده است.

در اینجا متغیرها و هایپرپارامتر ها به این شکل در مقاله خواسته شده اند:

1- مقدار batch size برابر 128 باشد.
2- از SGD استفاده کنند.
3- همه وزن ها به یک نرمال با مرکز صفر و انحراف معیار 0.02 مقداردهی اولیه میشوند.
4- شیب در LeakyReLU برابر 0.2 میباشد.
5- از Adam optimizer در هر دو استفاده شود.
6- از learning rate برابر 0.0002 استفاده شود؛ چون به گفته مقاله مقدار 0.001 زیاد است.
7- مقدار ممنتوم B1 در Adam به 0.5 تغییر پیدا کند تا دوره train کردن stablize شود.

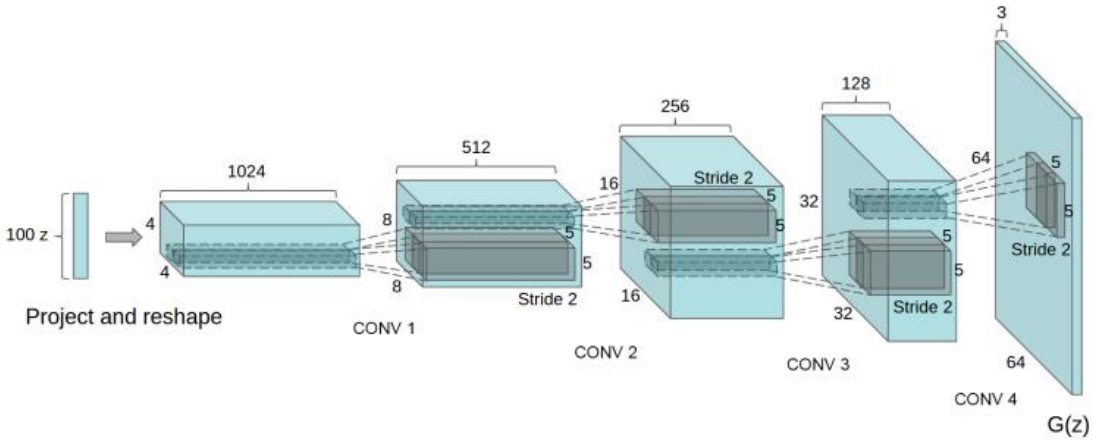
8- همچنین خواسته شده که data augmentation انجام نشود، برای همین برای این قسمت تنها resize کردن به مقدار 64 و normalize کردن به میانگین 0.5 و انحراف معیار 0.5 انجام شده است.

هایپرپارامترها و پیش پردازش انجام شدخه در این قسمت طبق نکات بالا در شکل زیر آورده شده است.

```
# Hyperparameters etc.
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
LEARNING_RATE = 2e-4  # could also use two lrs, one for gen and one for disc
BATCH_SIZE = 128
IMAGE_SIZE = 64
CHANNELS_IMG = info['n_channels']
NOISE_DIM = 100
NUM_EPOCHS = 1000
FEATURES_DISC = 64
FEATURES_GEN = 64

transforms = transforms.Compose(
    [
        transforms.Resize(IMAGE_SIZE),
        transforms.ToTensor(),
        transforms.Normalize(
            [0.5 for _ in range(CHANNELS_IMG)], [0.5 for _ in range(CHANNELS_IMG)]
        ),
    ]
)

# Load the data
train_dataset = DataClass(split='train', transform=transforms, download=True)
dataloader = data.DataLoader(dataset=train_dataset, batch_size=BATCH_SIZE, shuffle=True)
val_dataset = DataClass(split='val', transform=transforms, download=True)
val_dataloader = data.DataLoader(dataset=val_dataset, batch_size=BATCH_SIZE, shuffle=True)
test_dataset = DataClass(split='test', transform=transforms, download=True)
test_dataloader = data.DataLoader(dataset=test_dataset, batch_size=BATCH_SIZE, shuffle=True)
```

شکل 11: هایپرپارامترها و پیش پردازش های مربوط به **Conditional DCGAN**

ساختار generator این پیاد سازی عینا همان چیزی است که در مقاله آورده شده است و ساختار گفته شده در مقاله به شکل زیر است:



شکل 12: ساختار **generator**

بنابراین در شکل زیر نیز ما این پیاده سازی را انجام داده ایم که از یک فضای latent به یک فضایی با 1024 فیچر میرویم و پس از آن هر دفعه تعداد فیچرها را نصف میکنیم تا در نهایت به تعداد فیچر برابر با تعداد کانال مدنظر برسیم که در اینجا با توجه به سیاه و سفید بودن عکس ها برابر یک است. بعد هر لایه ReLU میذاریم به جز لایه آخر که بعد از آن یک tanh میگذاریم.

```python
class Generator(nn.Module):
    def __init__(self, channels_noise, channels_img, features_g):
        super(Generator, self).__init__()
        self.net = nn.Sequential(
            # Input: N x channels_noise x 1 x 1
            self._block(channels_noise, features_g * 16, 4, 1, 0),  # img: 4x4
            self._block(features_g * 16, features_g * 8, 4, 2, 1),  # img: 8x8
            self._block(features_g * 8, features_g * 4, 4, 2, 1),  # img: 16x16
            self._block(features_g * 4, features_g * 2, 4, 2, 1),  # img: 32x32
            nn.ConvTranspose2d(
                features_g * 2, channels_img, kernel_size=4, stride=2, padding=1
            ),
            # Output: N x channels_img x 64 x 64
            nn.Tanh(),
        )

    def _block(self, in_channels, out_channels, kernel_size, stride, padding):
        return nn.Sequential(
            nn.ConvTranspose2d(
                in_channels,
                out_channels,
                kernel_size,
                stride,
                padding,
                bias=False,
            ),
            # nn.BatchNorm2d(out_channels),
            nn.ReLU(),
        )

    def forward(self, x):
        return self.net(x)
```

شکل 13: ساختار **generator**

در discriminator ما برعکس ساختار generator عمل میکنیم و از عکس داده شده که دارای تعداد فیچرهایی برابر تعداد کانال عکس هستند(در اینجا یک)، هر دفعه تعداد فیچرهارا بیشتر میکنیم و در نهایت آن را به یک نورون که برای predict کردن به صفر یا یک استفاده میشود، وصل میکنیمو برای این کار در آخر از sigmoid استفاده میکنیم که آن را بین یک تا صفر چیش بینی کند، چون برد آن در این بین است.

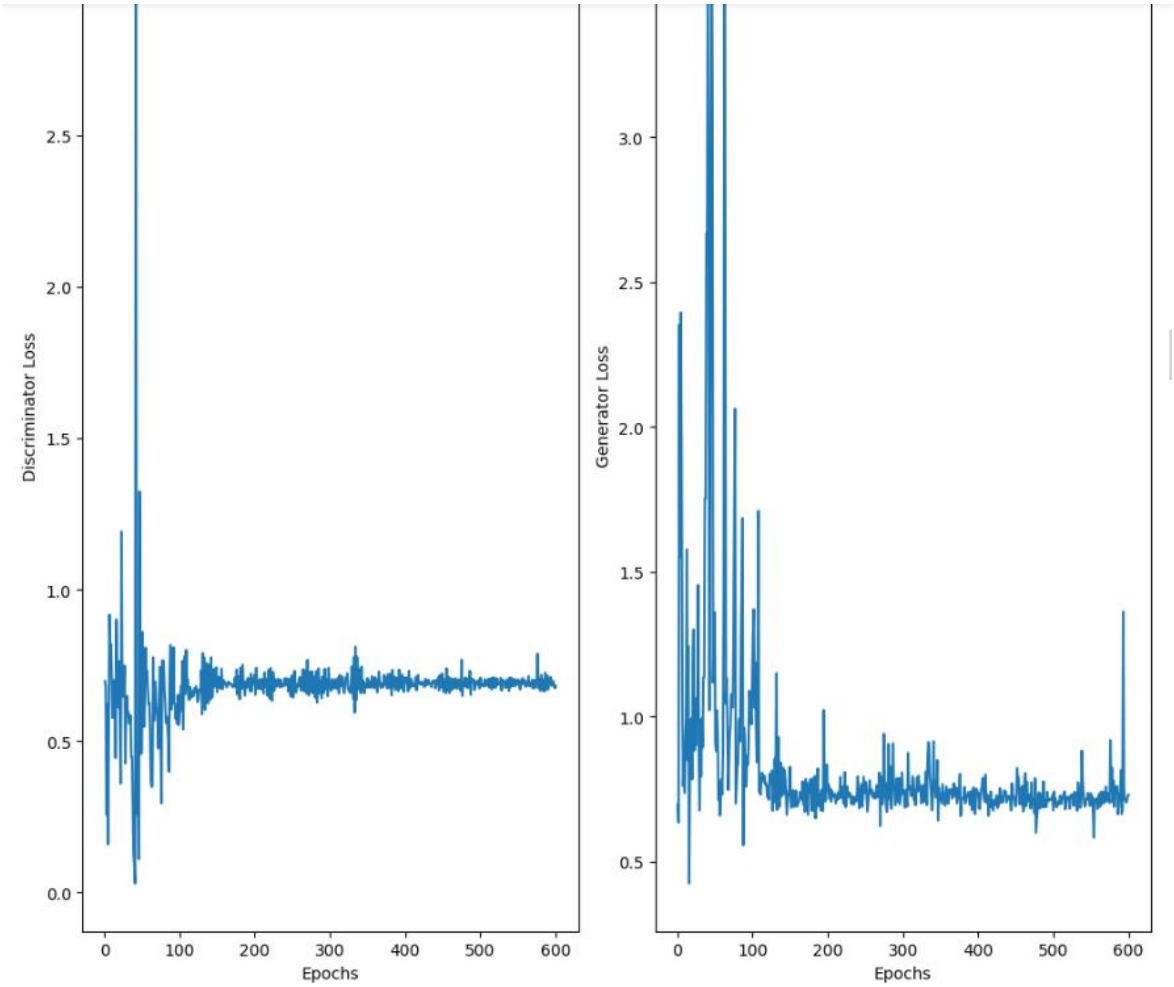
```python
class Discriminator(nn.Module):
    def __init__(self, channels_img, features_d):
        super(Discriminator, self).__init__()
        self.disc = nn.Sequential(
            # input: N x channels_img x 64 x 64
            nn.Conv2d(channels_img, features_d, kernel_size=4, stride=2, padding=1),
            nn.LeakyReLU(0.2),
            # _block(in_channels, out_channels, kernel_size, stride, padding)
            self._block(features_d, features_d * 2, 4, 2, 1),
            self._block(features_d * 2, features_d * 4, 4, 2, 1),
            self._block(features_d * 4, features_d * 8, 4, 2, 1),
            # After all _block img output is 4x4 (Conv2d below makes into 1x1)
            nn.Conv2d(features_d * 8, 1, kernel_size=4, stride=2, padding=0),
            nn.Sigmoid(),
        )

    def _block(self, in_channels, out_channels, kernel_size, stride, padding):
        return nn.Sequential(
            nn.Conv2d(
                in_channels,
                out_channels,
                kernel_size,
                stride,
                padding,
                bias=False,
            ),
            # nn.BatchNorm2d(out_channels),
            nn.LeakyReLU(0.2),
        )

    def forward(self, x):
        return self.disc(x)
```

شکل 14: ساختار **discriminator**

پیاده سازی این بخش را به دو قسمت تقسیم کردیم، در یک قسمت تا اپاک 600 جلو میرویم و دلیل ایکار آن است که در اینجا طبق گفته دستیاز آموزشی که میخواهیم دو تا loss به هم نزدیک شوند، در اینجا متوثق میشویم و جلوتز نمیرویم که دقت و خطا به شکل زیر میشوند.

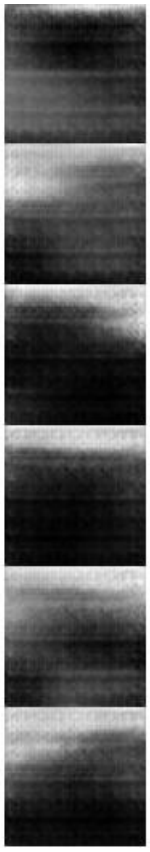
```
Epoch [592/600] Batch 0/5                   Loss D: 0.7111, loss G: 1.3627
Accuracy on validation set: 57.13%
Accuracy on train set: 59.44%
Accuracy on test set: 58.61%
Epoch [593/600] Batch 0/5                   Loss D: 0.6833, loss G: 0.7528
Accuracy on validation set: 51.06%
Accuracy on train set: 54.64%
Accuracy on test set: 55.83%
Epoch [594/600] Batch 0/5                   Loss D: 0.6826, loss G: 0.7154
Accuracy on validation set: 47.22%
Accuracy on train set: 43.91%
Accuracy on test set: 42.03%
Epoch [595/600] Batch 0/5                   Loss D: 0.6980, loss G: 0.7112
Accuracy on validation set: 53.55%
Accuracy on train set: 51.97%
Accuracy on test set: 48.64%
Epoch [596/600] Batch 0/5                   Loss D: 0.6956, loss G: 0.7033
Accuracy on validation set: 58.39%
Accuracy on train set: 59.75%
Accuracy on test set: 62.34%
Epoch [597/600] Batch 0/5                   Loss D: 0.6796, loss G: 0.7122
Accuracy on validation set: 57.22%
Accuracy on train set: 63.40%
Accuracy on test set: 63.57%
Epoch [598/600] Batch 0/5                   Loss D: 0.6739, loss G: 0.7249
Accuracy on validation set: 48.83%
Accuracy on train set: 54.88%
Accuracy on test set: 53.86%
Epoch [599/600] Batch 0/5                   Loss D: 0.6815, loss G: 0.7303
```

شکل 15: **log** های پایانی خطا و دقت برای شبکه با 600 ایپاک

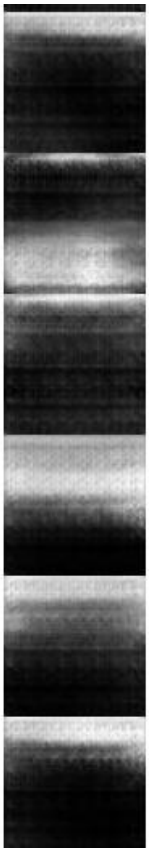


شکل 16: نمودار **loss** برای **generator** و **discriminator** برای شبکه با 600 ایپاک

همانطور که در تصویر بالا مشاهده میشود، مقدار خطای generator و discriminator در ظول این 600 ایپاک نزدیک به هم شده، هر چه جلوتر میرویم، کمتر oscillate میکند و هر دو مقدار حدودا 0.7 همگرا میشوند. شکل های به دست آمده از generator اینم شبکه به شکل زیر است.



شکل 17: کلاس صفر در شبکه با 600 ایپاک



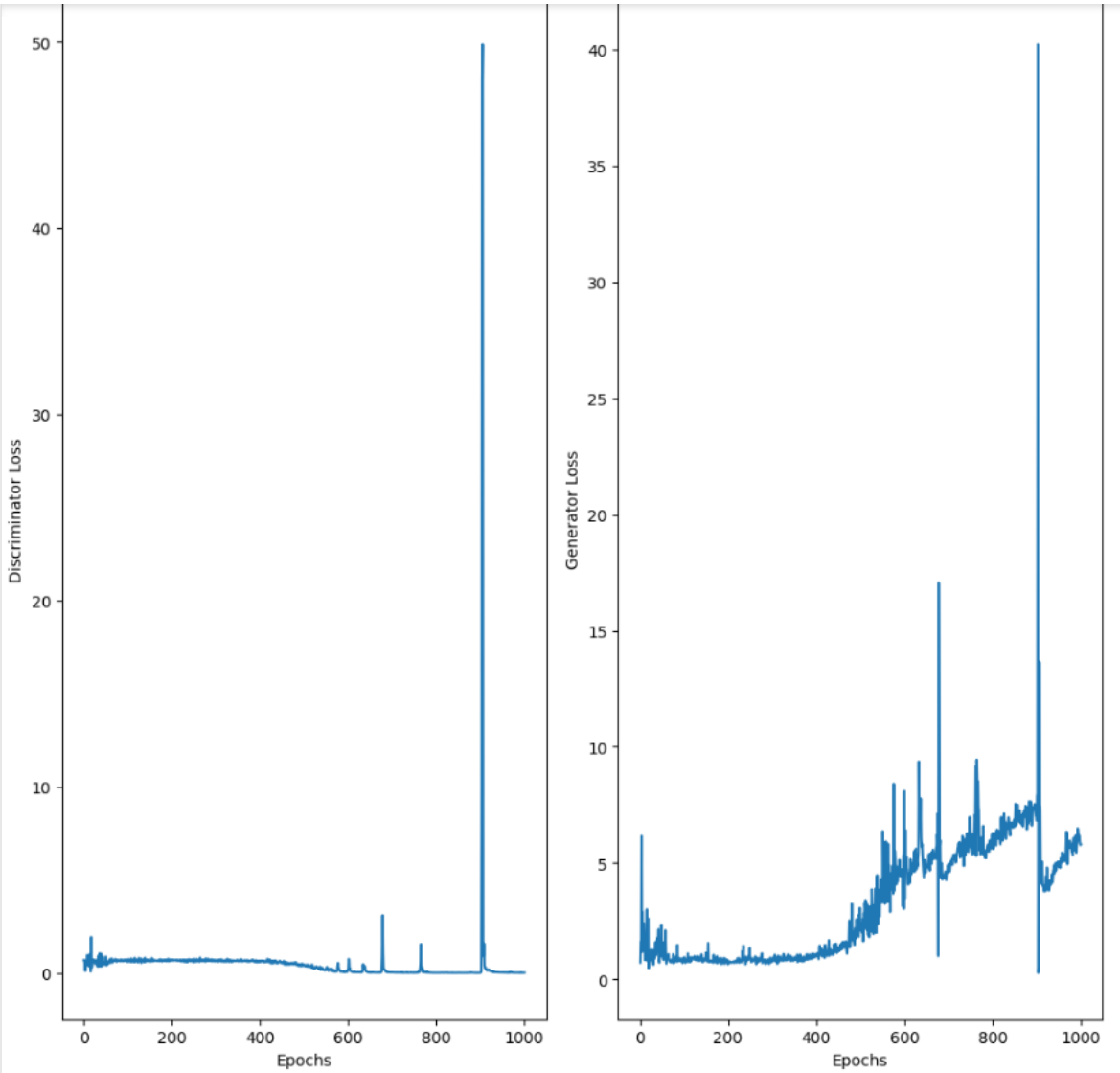
شکل 18: کلاس یک در شبکه با 600 ایپاک

همانطور که مشخص است، شکل های خوبی در نهایت به دست نیامده اند و دلیل آن میتواند تعداد ایپاک کم باشد، چ.ن هنوز به شبکه فرصت نداده ایم تا کامل همگرا شود و صرفا کمتر نوسان کردن نمیتواند جوابگو باشد. چون مقدار دقت هر سه دسته داده هنوز در حال نوسان است و هنوز به خوبی قابل تشخیص نیستند، پس توقع خروجی خوبی نباید از generator نباید داشته باشیم.

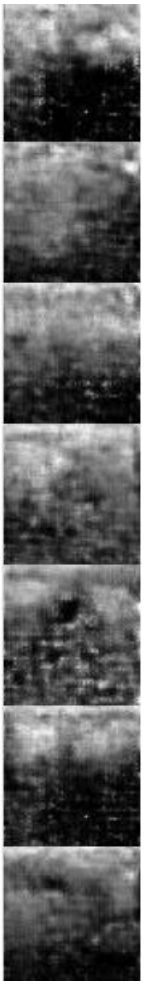
حال تا 1000 ایپاک جلو میرویم و نمودار خطا و مقدارهای خطا و دقت به شکل زیر در می آیند.

```
Epoch [991/1000] Batch 0/5                  Loss D: 0.0151, loss G: 5.8664
Accuracy on validation set: 68.59%
Accuracy on train set: 99.92%
Accuracy on test set: 67.75%
Epoch [992/1000] Batch 0/5                  Loss D: 0.0307, loss G: 6.4909
Accuracy on validation set: 69.23%
Accuracy on train set: 99.77%
Accuracy on test set: 67.63%
Epoch [993/1000] Batch 0/5                  Loss D: 0.0155, loss G: 6.2957
Accuracy on validation set: 68.84%
Accuracy on train set: 99.84%
Accuracy on test set: 67.88%
Epoch [994/1000] Batch 0/5                  Loss D: 0.0081, loss G: 6.0019
Accuracy on validation set: 69.23%
Accuracy on train set: 100.00%
Accuracy on test set: 67.75%
Epoch [995/1000] Batch 0/5                  Loss D: 0.0092, loss G: 6.1984
Accuracy on validation set: 68.59%
Accuracy on train set: 99.61%
Accuracy on test set: 67.63%
Epoch [996/1000] Batch 0/5                  Loss D: 0.0128, loss G: 6.0550
Accuracy on validation set: 68.20%
Accuracy on train set: 99.69%
Accuracy on test set: 67.75%
Epoch [997/1000] Batch 0/5                  Loss D: 0.0112, loss G: 5.8790
Accuracy on validation set: 69.23%
Accuracy on train set: 100.00%
Accuracy on test set: 67.75%
Epoch [998/1000] Batch 0/5                  Loss D: 0.0103, loss G: 5.7735
Accuracy on validation set: 68.84%
Accuracy on train set: 99.45%
Accuracy on test set: 67.56%
Epoch [999/1000] Batch 0/5                  Loss D: 0.0137, loss G: 5.8027
```

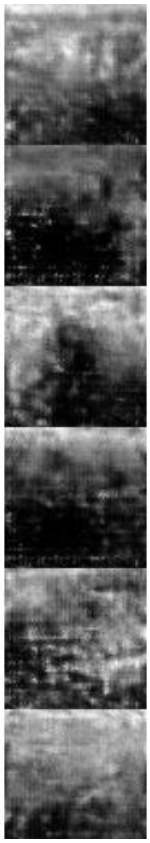
شکل 19: **log** های پایانی خطا و دقت برای شبکه با 1000 ایپاک



شکل 20: نمودار **loss** برای **generator** و **discriminator** برای شبکه با 1000 ایپاک



شکل 21: کلاس صفر در شبکه با 1000 ایپاک



شکل 22: کلاس یک در شبکه با 1000 ایپاک

همانطور که مشخص است، عکس های خیلی بهتری نسبت به دفعه قبل ایجاد شده است. با اینکه حتی دیتاست BreastMNISt عکس هایش قابل تشخیص نیستند، پس حتی خروجی generator هم با چشم غیرقابل تشخیص است. در لاگ ها هم مشخص است که دقت ها نوساتن کمتری دارند و دقت داده آموزش به 99 درصد رسیده و برای داده اعتبارسنجی و ارزیابی حدود 68 درصد شده اند. این مقدار دقت از مقداری که از ResNet به دست آوردیم برای دادگان val و test پایینتر میباشد و برای داده train بالاتر میباشد که نشان میدهد این شبکه نسبت به شبکه ResNet عمیق تر نمیباشد؛ زیر مقدار دقت دادگان ارزیابی و اعتبارسنجی نسبت به دادگان آموزش بسیار پایینتر است که نشان میدهد دچار underfit شده است و این در حالی است که در ResNet دقت دادگان ارزیابی و اعتبار سنجی حدود 80 درصد و حتی از دقت دادگان آموزش بیشتر است.

توابع خطا برای شناسایی‌کننده (Discriminator) و مولد (Generator) در GAN بسیار حائز اهمیت هستند. هدف اصلی discriminator این است که بین داده های واقعی و جعلی تفاوت‌ها را تشخیص دهد، در حالی که هدف اصلی generator، تولید داده‌های واقع‌گرایی است که discriminator را به گمراهی بکشاند.

تابع خطا برای شناسایی‌کننده به طور معمول شامل (Binary Cross-Entropy) است که میزان توانایی شناسایی‌کننده در درست تشخیص دادن داده‌های واقعی و جعلی را اندازه‌گیری می‌کند. اما تابع خطا برای مولد، به طور معمول شامل (Negative Binary Cross-Entropy) است که generator را تشویق می‌کند تا داده‌هایی تولید کند که discriminator می‌تواند به عنوان داده‌های واقعی تشخیص دهد.

در طول فرآیند آموزش، تابع خطا برای discriminator و generator باید به گونه‌ای تعادل‌یافته باشد که هیچ یک از آنها بر دیگری غلبه نکند. اگر تابع خطا برای شناسایی‌کننده بسیار بالا باشد، به این معنی است که discriminator بسیار توانا در تشخیص داده‌های واقعی و جعلی است و این ممکن است باعث شود generator سختی در بهبود خود داشته باشد. از سوی دیگر، اگر تابع خطا برای generator بسیار بالا باشد، به این معنی است که generator داده‌های واقع‌گرا تولید نمی‌کند که discriminator را به گمراهی بکشاند.

یکی از روش‌های تعادل‌یافته کردن تابع خطا برای شناسایی‌کننده و مولد، تنظیم نرخ یادگیری برای هر دو شبکه است. رویکرد دیگر، استفاده از تکنیک‌هایی مانند تنظیم گرادیان یا نرمال‌سازی طیفی است که فرآیند آموزش را پایدار تر کرده و جلوی غلبه کردن یکی از شبکه‌ها بر دیگری را می‌گیرد.

به طور خلاصه، تابع خطا برای شناسایی‌کننده و مولد در یک GAN باید در طول آموزش به گونه‌ای تعادل‌یافته باشد که هر دو شبکه به طور موثری یاد بگیرند و مولد قادر به تولید داده‌های واقع‌گرا باشد.

به طور کلی خواسته باید احتمالا شبکه با 600 ایپاک باشد جون هیچ کدام از loss ها بریکذیگر غلبه نمیکنند و هر دو نزدیک مقدار 0.7 هستند، اما تصاویر با 1000 ایپاک بهتر شده و از آن برای قسمت سوم استفاده میکنیم. تصاویر از جایی به بعد نمیتوانند بهتر تولید شوند.

برای بهتر شدن خروجی مولد و پایدارسازی شبکه، می توان از راهکارهای مختلفی استفاده کرد. در زیر به برخی از این راهکارها اشاره می کنم:

1. استفاده از تکنیک های Preprocessing داده ها: این تکنیک ها می توانند به کاهش نویز و افزایش کیفیت داده های ورودی کمک کنند. مثلاً می توان از روش های پاکسازی، نرمال سازی و تبدیل داده ها به فضای برداری استفاده کرد.
2. استفاده از تکنیک های Regularization : تکنیک هایRegularization ، مانند Dropout و L1/L2 regularization، می توانند به کاهش اورفیتینگ شبکه و افزایش پایداری آن کمک کنند.
3. افزایش تعداد داده های آموزشی: با افزایش تعداد داده های آموزشی، مولد می تواند الگوهای بیشتری را یاد بگیرد و خروجی بهتری تولید کند.

4. تغییر در معماری شبکه: با تغییر در معماری شبکه، می توان به بهبود خروجی مولد کمک کرد. شبکه های عمیقتر میتوانند بهتر عمل کنند که در بالا در مقایسه دو شبکه به این نتیجه رسیدیم و از روی دقت ها آنها را مقایسه کردیم.
5. تنظیم پارامتر های شبکه: تنظیم پارامتر های شبکه، مانند learning rate وbatch size ، می تواند به بهبود خروجی مولد کمک کند. مثلاً با افزایشlearning rate ، می توان به سرعت آموزش شبکه کمک کرد، و با کاهشbatch size ، می توان به بهبود کیفیت خروجی کمک کرد.
6. استفاده از تکنیک های :Transfer Learning با استفاده از تکنیک هایTransfer Learning ، می توان از مدل های قبلی آموزش دیده استفاده کرد و مولد را بر اساس آنها آموزش داد. این روش می تواند به بهبود خروجی مولد کمک کند و آموزش شبکه را سریعتر کند.

بر اساس این موارد، سعی شده است تا حد ممکن پیاده سازی به پیاده سازی مقاله نزدیک باشد، چون خود مقاله درصدد رفع این موضوعات بوده است. به طور مثال نرخ یادگیری را کاهش داده است، مقدار batch size را تا 128 افزایش داده است، از regularization استفاده کرده است و در معماری دست برده و کارهایی که در ابتدا گفتیم را انجام داده است مانند حذف fc ها، استفاده از strided convolution ها، حذف pooling ها و... .

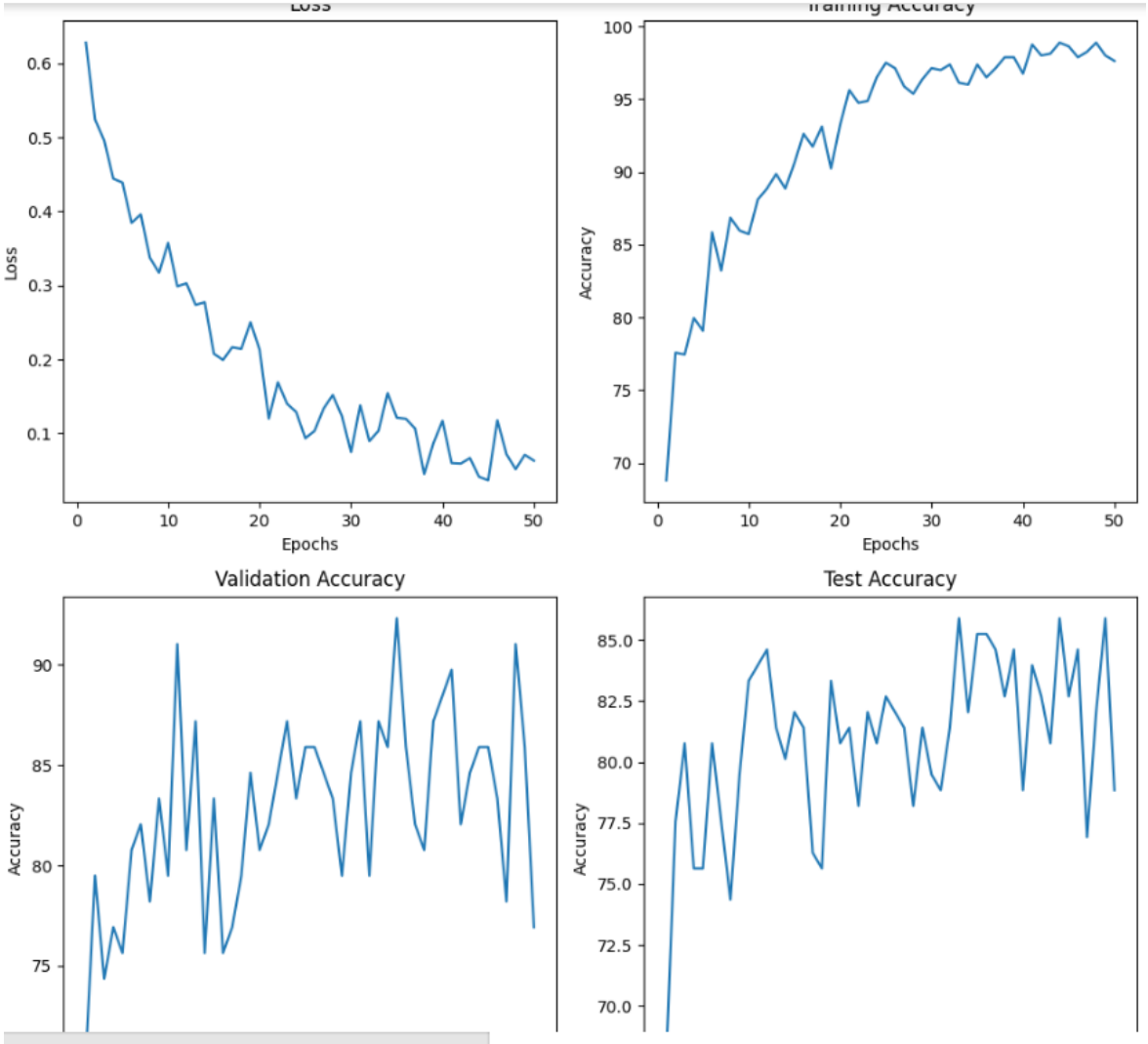
2.3: طبقه بندی به کمک داده های تولید شده توسط مولد

در اینجا ابتدا تعداد داده های هر کلاس در trains set را به به دست می آورین که نسبت آنها 399 به 147 میباشد، به همین دلیل از 2000 عکس تولید شده برای کلاسصفرم به طور تصادفی 252 عکس انتخاب میکنیم و به train set اضافه میکنیم که مجموع همه عکس ها برابر 798 تا میشود که هر کلاس 399 نمونه خواهد داشت. کد به شکلا زیر است.

```python
import random
from PIL import Image

# Step 1: Load images from the "class_0_second" folder
class_0_path = 'class_0_second'  # Replace with the actual path to the folder
class_0_data = []
for filename in os.listdir(class_0_path):
    img = np.array(Image.open(os.path.join(class_0_path, filename)))
    img = np.resize(img, (1, 28, 28))
    class_0_data.append(img)

# Step 2: Shuffle the new dataset
random.shuffle(class_0_data)

# Step 3: Select the required number of samples
num_samples = len(class_0_data)
selected_samples = class_0_data[:252]

# Step 4: Append the selected samples to the existing dataset
X_train_new = np.array(selected_samples).reshape((-1, 1, 28, 28))
y_train_new = np.array([0.] * 252)

X_train = np.concatenate((X_train, X_train_new), axis=0)
y_train = np.concatenate((y_train, y_train_new), axis=0)
```

شکل 23: اضافه کردن عکس های تولید شده به صورت تصادفی برای ایجاد تعادل بین کلاس ها

حال به همان شکلی که در قسمت اول ران کردیم، در این قسمت هم با همان پارامترها ران میکنیم تا امکان مقایسه وجود داشته باشد و نتایج زیر به دست می آید. فقط نرمالایز نکردیم چون با نرمالایز کردن به اشتباه همه را یک پیش بینی میکرد.



شکل 24: نتایج دقت و خطا

```
Epoch 50/50 - Loss: 0.06320665226550773 - Train Accuracy: 97.62% - Val Accuracy: 76.92% - Test Accuracy: 78.85%
Confusion Matrix:
[[32 10]
 [23 91]]
```

شکل 25: ماتریس آشفتگی

همانطور که مشخص است، این نتایج به دست آمده تا حدودی باعث بهتر شدن نتایج دقت دادگان آموزشی شده است اما دقت دادگان ارزیابی و اعتبارسنجی به همان شکل باقی مانده اند. در اینجا مقدارهای TP را نسبت به قبل بهتر پیش بینی کرده است ولی مقادیر FP مقداری افت کرده است.