



نام و نام خانوادگی	علی رنجبری - امیرحسین علیزاد
شماره دانشجویی	۸۱۰۱۹۸۵۷۰ - ۸۱۰۱۹۷۵۴۶
تاریخ ارسال گزارش	۱۴۰۱.۱۱.۰۷

	<p>به نام خدا دانشگاه تهران دانشکده مهندسی برق و کامپیوتر</p>	
<p>درس شبکه‌های عصبی و یادگیری عمیق تمرین ششم</p>		

## فهرست

۱	پاسخ 1. شبکه های مولد تخصصی کانولوشنال عمیق
۱	۱-۱. پیاده سازی مولد تصویر با استفاده از شبکه های مولد تخصصی کانولوشنال عمیق
۲	۱-۲. ارزیابی شبکه
۲	۱-۳. پایدار سازی شبکه
۴	پاسخ ۲ - شبکه متخاصم مولد طبقه بندی کمکی و شبکه Wasserstein
۲	۲-۱. شبکه متخاصم مولد طبقه بندی کمکی
۳	۲-۲. شبکه متخاصم مولد Wasserstein

## شکل‌ها

- ۱ شکل ۱. DCGAN generator و discriminator شبکه
- ۱ شکل ۲. نمونه عکس های تولید شده توسط DCGAN
- ۲ شکل ۳. نمودار های loss و accuracy برای DCGAN
- ۳ شکل ۴. نمودار Loss و Accuracy برای شبکه DCGAN بعد از پایدار سازی
- ۳ شکل ۵. نمونه عکس های تولید شده توسط شبکه DCGAN بعد از پایدار سازی
- ۴ شکل ۶. Generator و Discriminator شبکه AC-GAN
- ۴ شکل ۷. نمودار Loss و Accuracy برای AC-GAN
- ۵ شکل ۸. نمونه عکس های تولید شده توسط AC-GAN
- ۵ شکل ۹. کلاس WassersteinLoss
- ۶ شکل ۱۰. نمودار Loss و Accuracy برای شبکه WGAN
- ۶ شکل ۱۱. نمونه عکس های تولید شده توسط شبکه

## پاسخ ۱. شبکه های مولد تخصصی کانولوشنال عمیق

۱-۱. پیاده سازی مولد تصویر با استفاده از شبکه های مولد تخصصی کانولوشنال عمیق شبکه GAN را با استفاده از Generator و Discriminator بصورت زیر تولید کردیم.

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.conv_blocks = nn.Sequential(
            nn.Conv2d(1, 64, 4, stride=2, padding=1),
            nn.LeakyReLU(0.2),
            nn.Conv2d(64, 128, 4, stride=2, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2),
            nn.Conv2d(128, 256, 4, stride=2, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2),
            nn.Conv2d(256, 512, 4, stride=2, padding=1),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2),
        )
        self.fc = nn.Linear(4*4*512, 1)

    def forward(self, x):
        x = self.conv_blocks(x)
        x = x.view(-1, 4*4*512)
        x = self.fc(x)
        return x

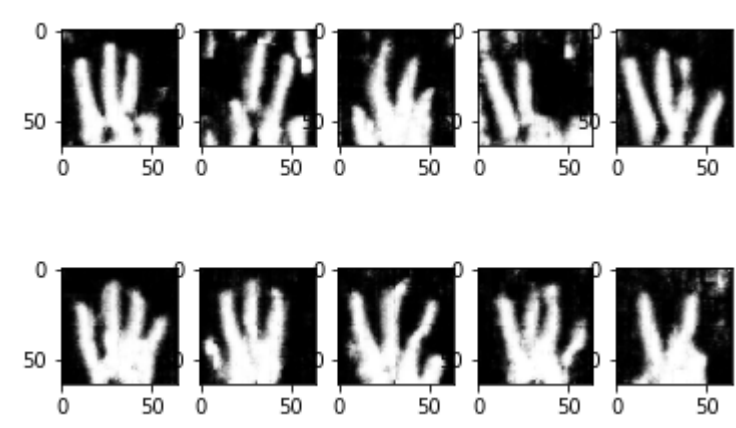
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        # formular: (h-1) * stride + (h-1) + 1

        self.fc = nn.Linear(100, 4*4*1024) # 4 * 4
        self.bn = nn.BatchNorm2d(1024)
        self.conv_blocks = nn.Sequential(
            nn.ConvTranspose2d(1024, 512, 2, stride=2), # 8 * 8
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.ConvTranspose2d(512, 256, 2, stride=2), # 16 * 16
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.ConvTranspose2d(256, 128, 2, stride=2), # 32 * 32
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.ConvTranspose2d(128, 1, 2, stride=2), # 64 * 64
            nn.Tanh()
        )

    def forward(self, x):
        x = self.fc(x)
        x = x.view(-1, 1024, 4, 4)
        x = self.bn(x)
        x = self.conv_blocks(x)
        return x
```

شکل ۱. generator و discriminator شبکه DCGAN

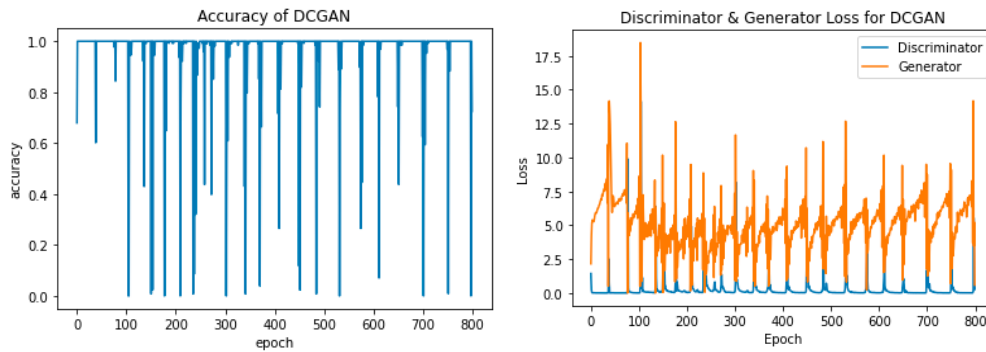
مدل را به تعداد ۱۰۰ epoch با learning rate=0.0002 و adam optimizer آموزش دادیم. همچنین تعداد batch size را ۱۲۸ در نظر گرفتیم. تعداد ۱۰ عکس با generator تولید کردی که در شکل زیر مشخص است.



شکل ۲. نمونه عکس های تولید شده توسط DCGAN

## ۱-۲. ارزیابی شبکه

نمودار Loss و Accuracy بعد از epoch ۱۰۰ بصورت زیر است.



شکل ۳. نمودار های loss و accuracy برای DCGAN (دقت فقط برای discriminator رسم شده)

## ۱-۳. پایدار سازی شبکه

برای پایدار سازی شبکه DCGAN از دو روش One-sided label Smoothing و Add Noise استفاده شد که ابتدا بصورت مختصر در زیر این دو روش را توضیح میدهیم.

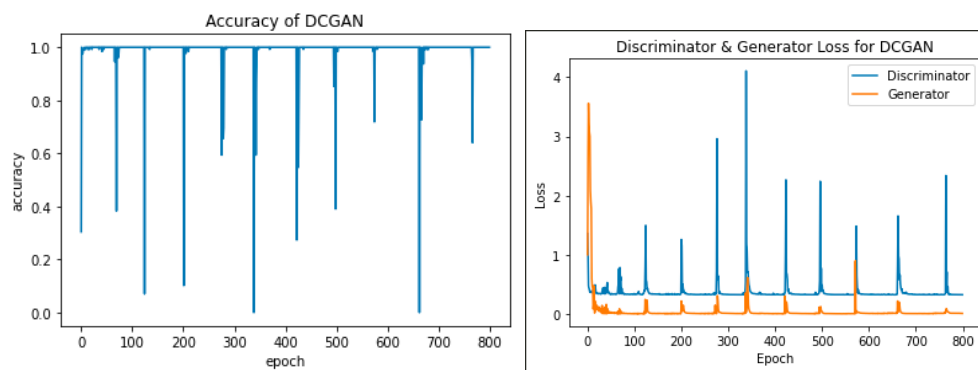
### I. One-sided label Smoothing

label smoothing روشی است که برای این استفاده می‌شود که مدل در جواب دادن و تولید کردن اطمینان کمتری داشته باشد و این کار را به این صورت انجام می‌دهد که نمونه های درست در دیتاست قطعی نیستند. one-sided label smoothing یک روش برای این کار است که تنها کلاس درست در دیتاست smooth می‌شوند به اینصورت که به مدل می‌گوییم کلاس درست 1 نیست و ممکن است 0.9 باشد. دلیل اینکه به این روش one-sided می‌گویند این است که فقط کلاس درست نرم می‌شوند و کلاس غلط یا کلاس های دیگر دست نخورده باقی میمانند.

### II. Add Noise

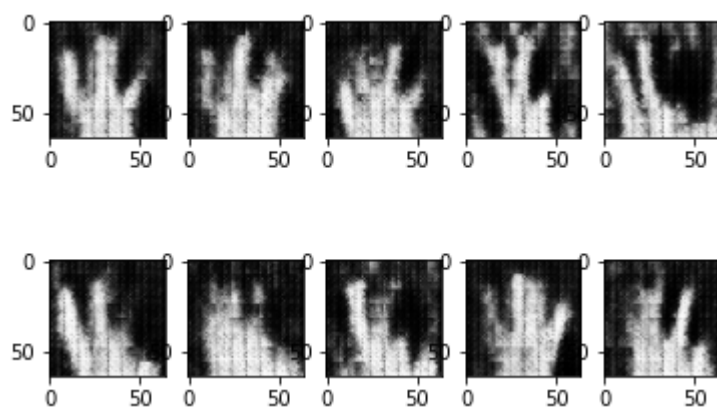
این روش با اضافه کردن نویز به دیتای ورودی باعث میشود از overfitting جلوگیری شود و همچنین مدل با حفظ کردن دیتای ورودی نمیتواند پیشبینی کند. نویز های متفاوتی وجود دارد که میتوان به دیتا اضافه کرد مثل gaussian noise یا dropout noise. اضافه کردن نویز به صورت کلی باعث میشود مدل منطقه های مختلفی از فضای ورودی را کاوش کند.

برای One-Sided label smoothing ما به جای ۱ برای تصاویر واقعی از ۰.۹ استفاده کردیم همچنین برای اضافه کردن نویز قبل از ورود عکس ها به discriminator به آنها gaussian noise نویز اضافه کردی. سپس با همان هاپیر پارامتر های قبلی مدل قبلی را آموزش دادیم نمودار Loss و Accuracy بصورت زیر شد:



شکل ۴. نمودار Loss و Accuracy برای شبکه DCGAN بعد از پایدار سازی

همچنین عکس های تولید شده با این شبکه به صورت زیر است:



شکل ۵. نمونه عکس های تولید شده توسط شبکه DCGAN بعد از پایدار سازی

## پاسخ ۲ - شبکه متخاصم مولد طبقه‌بندی کمکی و شبکه Wasserstein

### ۲-۱. شبکه متخاصم مولد طبقه‌بندی کمکی

شبکه AC-GAN برخلاف شبکه‌های معمولی GAN همزمان که واقعی یا غیر واقعی بودن ورودی‌ها را یاد می‌گیرد شماره کلاس یا دسته آنها را هم یاد می‌گیرد این کار باعث می‌شود که مدل همزمان بهتر بتواند تشخیص دهد و یادگیری را بهتر میکند.

generator و discriminator این شبکه در شکل زیر مشخص است:

```
class Discriminator(nn.Module):
    def __init__(self, classes=5):
        super(Discriminator, self).__init__()

        # formula: (n + 2*pad - b) / stride + 1
        self.conv1 = nn.Sequential( # 16
            nn.Conv2d(1, 16, 3, stride=2, padding=1),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.5)
        )

        self.conv2 = nn.Sequential( # 16
            nn.Conv2d(16, 32, 3, stride=1, padding=1),
            nn.BatchNorm2d(32),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.5)
        )

        self.conv3 = nn.Sequential( # 8
            nn.Conv2d(32, 64, 3, stride=2, padding=1),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.5)
        )

        self.conv4 = nn.Sequential( # 8
            nn.Conv2d(64, 128, 3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.5)
        )

        self.conv5 = nn.Sequential( # 4
            nn.Conv2d(128, 256, 3, stride=2, padding=1),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.5)
        )

        self.conv6 = nn.Sequential( # 4
            nn.Conv2d(256, 512, 3, stride=1, padding=1),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.5)
        )

        self.fc_source = nn.Linear(4*4*512, 1)
        self.fc_class = nn.Linear(4*4*512, classes)
        self.sig = nn.Sigmoid()
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        x = self.conv4(x)
        x = self.conv5(x)
        x = self.conv6(x)
        x = x.view(-1, 4*4*512)
        rf = self.sig(self.fc_source(x))
        c = self.softmax(self.fc_class(x))

        return rf, c

class Generator(nn.Module):
    def __init__(self):
        super().__init__()

        # formula: (h-1) * stride + (k-1) + 1

        self.fc = nn.Linear(110, 384)
        self.conv1 = nn.Sequential( # 5 * 5
            nn.ConvTranspose2d(384, 192, 5, stride=2),
            nn.BatchNorm2d(192),
            nn.ReLU()
        )

        self.conv2 = nn.Sequential( # 13 * 13
            nn.ConvTranspose2d(192, 96, 5, stride=2),
            nn.BatchNorm2d(96),
            nn.ReLU()
        )

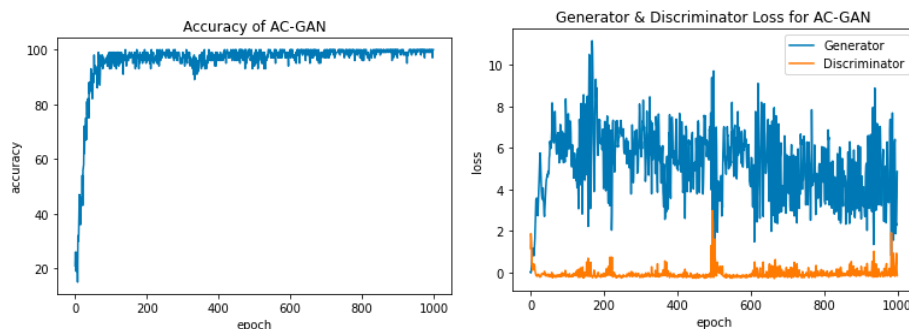
        self.conv3 = nn.Sequential( # 29 * 29
            nn.ConvTranspose2d(96, 3, 5, stride=2),
            nn.BatchNorm2d(3),
            nn.ReLU()
        )

        self.conv4 = nn.Sequential( # 32 * 32
            nn.ConvTranspose2d(3, 1, 4, stride=1),
            nn.Tanh()
        )

    def forward(self, x):
        x = x.view(-1, 110)
        x = self.fc(x)
        x = x.view(-1, 384, 1, 1)
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        x = self.conv4(x)
        # print(x.shape)
        return x
```

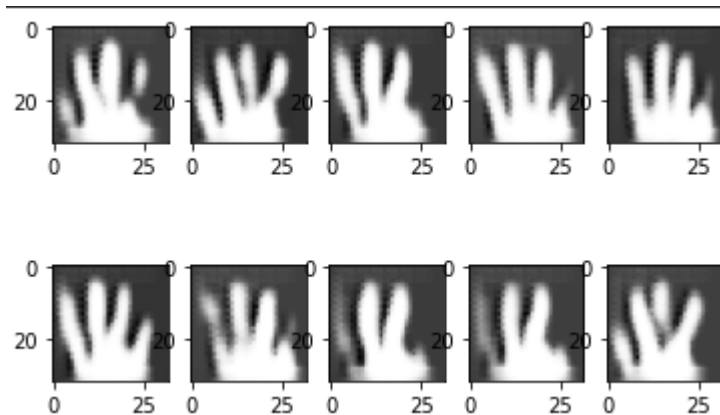
شکل ۶. Generator و Discriminator شبکه AC-GAN

سپس همان دیتاست سوال قبل را روی این دیتا آموزش دادیم با تعداد ۱۰۰ epoch و learning rate برابر 0.0002 نمودار accuracy و loss در زیر مشخص است:



شکل ۷. نمودار Accuracy و Loss برای AC-GAN

همچنین عکس های تولید توسط AC-GAN در شکل ۸ گذاشته شده است.



شکل ۸. نمونه عکس های تولید شده توسط AC-GAN

## ۲-۲. شبکه متخاصم مولد Wasserstein

Wasserstein Loss یا Earth Mover's distance یک شاخص محاسبه loss برای یادگیری عمیق است که بیشتر در شبکه های GAN استفاده می شود.

این loss فاصله بین دو توزیع احتمالی را محاسبه میکند ، معمولاً دیتا های واقعی و دیتا های تولید شده غیر واقعی. این فاصله معمولاً به صورت کمترین مقدار کار مورد نیاز برای انتقال توزیع تولید شده به توزیع واقعی سنجیده می شود. که کار در اینجا به معنای مقدار حجم ضرب در میزان جابجایی تعریف می شود.

Wasserstein loss پایدار تر و راحت تر برای optimize کردن نسبت به توابع loss دیگر که در GAN ها استفاده می شود مثل Jensen-Shannon divergence یا Kullback-Leibler divergence است که معمولاً مشکلاتی مثل mode collapse و vanishing gradient دارند.

wasserstein loss را به صورت زیر تعریف کردیم:

```
class WassersteinLoss(nn.Module):
    def __init__(self):
        super(WassersteinLoss, self).__init__()

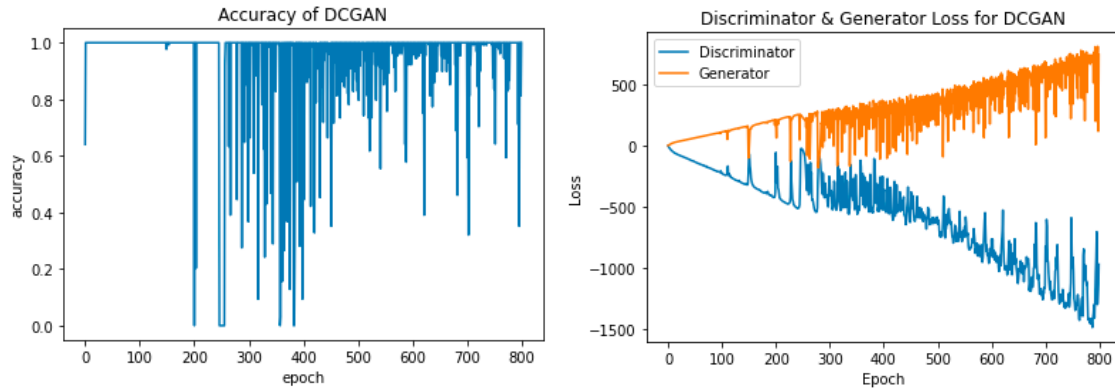
    def forward(self, real_output, fake_output):
        return -torch.mean(real_output - fake_output)
```

شکل ۹. کلاس WassersteinLoss



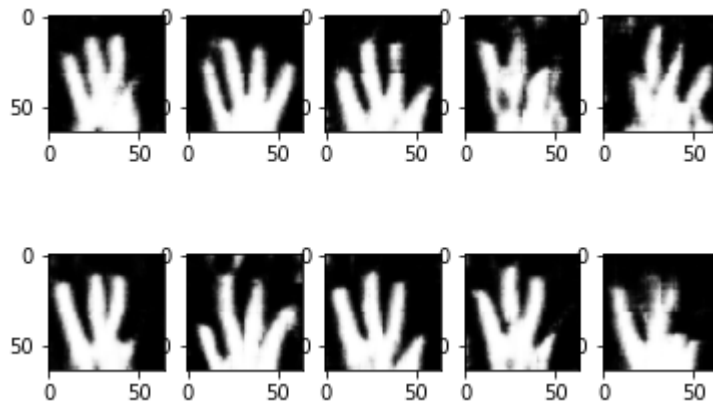
سپس همان Generator و Discriminator مدل DCGAN را برای این قسمت استفاده کردیم با این تفاوت که به جای loss قبلی از wasserstein loss استفاده کردیم.

نمودار loss و accuracy در این حالت در شکل ۱۰ مشخص است:



شکل ۱۰. نمودار Accuracy و Loss برای شبکه WGAN

و سپس ۱۰ نمونه با استفاده از این شبکه تولید شد.



شکل ۱۱. نمونه عکس های تولید شده توسط شبکه



