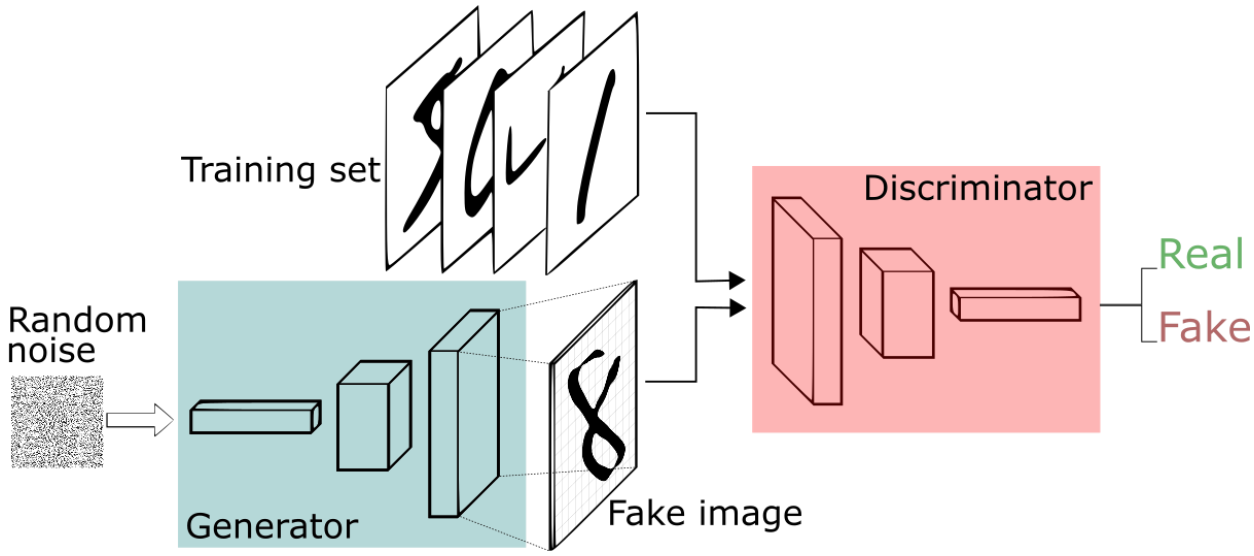
دانشگاه صنعتی شریف
دانشکده‌ی مهندسی کامپیوتر

تمرین سری پنجم
گزارش کار GAN

امیرحسین محمدی
۹۹۲۰۱۰۸۱

بهمن ۱۴۰۰

پیاده سازی معماری GAN:



تابع هزینه ی GAN:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right]$$

الگوریتم آموزش GAN:

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

  for k steps **do**

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
  - Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

پیاده سازی

- اضافه کردن کتابخانه های مورد نیاز :

```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms

import matplotlib.pyplot as plt
%matplotlib inline
plt.rcParams['figure.figsize'] = (10, 3)
```

- دانلود مجموعه داده های mnist و ترنسفورم کردن آنها، داده ها Resize می شوند به ابعاد 64 در 64، همچنین نرمال می شوند با پارامتر 0.5 برای mean و std و به شکل تنسور در می آیند.

```
img_transform = transforms.Compose([
    transforms.Resize(64),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])
train_dataset = datasets.MNIST(root='./mnist/', train=True, transform=img_
transform, download=True)
test_dataset = datasets.MNIST(root='./mnist/', train=False, transform=img_
transform, download=True)
```

- تعریف پارمترهای اولیه مثل ابعاد لایه ی latent، Epoch و batch_size و نرخ یادگیری برای generator و discriminator و چک کردن اینکه آیا امکان این وجود دارد که از GPU استفاده کنیم یا نه. در صورت عدم استفاده از GPU، سیستم پردازشی CPU در نظر گرفته می شود:

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

################ Problem 01 (5 pts) ################
# define hyper parameters
batch_size = 64
d_lr = 2e-4
g_lr = 2e-4
n_epochs = 20
######################## End ##########################
z_dim = 100
```

❖ تعریف dataloader برا خواندن دیتاست و انجام عملیت بر روی آن‌ها:

```
################ Problem 02 (5 pts) ################
# Define Dataloaders
train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_si
ze=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size
=batch_size, shuffle=False)
######################## End ########################
```

❖ تعریف معماری Discriminator، معماری Discriminator برای تعیین کیفیت تصاویر تولید شده توسط generator طراحی شده است. این معماری شاملی تعداد لایه 5 لایه کانولوشنی است که با فاکتور 2 سایز ابعاد ورودی را افزایش می دهند، همچنین برای Activation function از تابع LeakyRelu استفاده شده است به دلیل مزایای این تابع فعال ساز نسبت به دیگر توابع فعال ساز، همچنین پس از توابع Conv از تابع batchnorm استفاده شده است . در انتها برای تسهیل در فرآیند تابع هزینه از یک تابع Sigmoid استفاده شده است و در نهایت استفاده از یک تابع forward برای انجام عملیات پیاده سازی شده در Discriminator. در این معماری سایز کرنل 4، اندازه ی Stride 2 و اندازه padding 1 در نظر گرفته شده است:

```
class Discriminator(nn.Module):
    def __init__(self,d=8):
        super().__init__()

        self.discriminator = nn.Sequential(
            ################ Problem 03 (15 pts) ################
            # use linear or convolutional layer
            # use arbitrary techniques to stabilize training
            nn.Conv2d(1, d, 4, 2, 1),
            nn.LeakyReLU(),
            nn.Conv2d(d, d*2, 4, 2, 1),
            nn.BatchNorm2d(d*2),
            nn.LeakyReLU(),
            nn.Conv2d(d*2, d*4, 4, 2, 1),
            nn.BatchNorm2d(d*4),
            nn.LeakyReLU(),
            nn.Conv2d(d*4, d*8, 4, 2, 1),
            nn.BatchNorm2d(d*8),
            nn.LeakyReLU(),
            nn.Conv2d(d*8, 1, 4, 1, 0),
            nn.Sigmoid()
            ######################## End ########################
        )

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

❖ پیاده سازی معماری generator، همانطور که در این معماری می بینیم، این معماری شامل بخش های ConvTranspose2d، برای ایجاد تصویر از لایه ی latent است. پس از هر لایه ی ConvTranspose2d، یک لایه ی BatchNorm2d قرار دارد و بعد از آن یک تابع فعال ساز ReLU. در این معماری خروجی ConvTranspose2d با فاکتور 2 کاهش می یابد. همچنین در لایه ی آخر از یکتابع فعال ساز Tanh برای تسهیل در فرآیند تابع هزینه استفاده شده است. در این معماری سایز کرنل 4، Stride 2 و padding 1 در نظر گرفته شده است به جز لایه ی اول و در نهایت یک تابع forward برای انجام عملیات پیاده سازی شده در generator:

```
class Generator(nn.Module):
    def __init__(self,d=8):
        super().__init__()

        self.generator = nn.Sequential(
            ################ Problem 04 (15 pts) ################
            # use linear or convolutional layer
            # use arbitrary techniques to stabilize training
            nn.ConvTranspose2d(100, d*8, 4, 1, 0),
            nn.BatchNorm2d(d*8),
            nn.ReLU(True),
            nn.ConvTranspose2d(d*8, d*4, 4, 2, 1),
            nn.BatchNorm2d(d*4),
            nn.ReLU(True),
            nn.ConvTranspose2d(d*4, d*2, 4, 2, 1),
            nn.BatchNorm2d(d*2),
            nn.ReLU(True),
            nn.ConvTranspose2d(d*2, d, 4, 2, 1),
            nn.BatchNorm2d(d),
            nn.ReLU(True),
            nn.ConvTranspose2d(d, 1, 4, 2, 1),
            nn.Tanh()
            ######################## End ########################
        )

    def forward(self, z):
        return self.generator(z)
```

❖ ایجاد کلاس های generator و Discriminator و انتقال این کلاس ها بر روی device تعریق شده، همچنین فعال کردن requires_grad آنها:

```
discriminator = Discriminator()
generator = Generator()
generator = generator.to(device)
discriminator = discriminator.to(device)
num_params_gen = sum(p.numel() for p in generator.parameters() if p.requir
es_grad)
num_params_disc = sum(p.numel() for p in discriminator.parameters() if p.r
equires_grad)
```

- تعریف تابع بهینه ساز برای Generator و Disciminator ما نرخ یادگیری که در ابتدا تعریف شده و نرخ beta 0.5 0.999 است:

```
d_optimizer = torch.optim.Adam(params=discriminator.parameters(), lr=d_lr,
 betas=(0.5, 0.999))
g_optimizer = torch.optim.Adam(params=generator.parameters(), lr=g_lr, bet
as=(0.5, 0.999))
```

- قرار دادن Generator و Disciriminator در حالت train همچنین تعریف تو لیست برای محاسبه میانیگن خطای مولد و تمیز دهنده بعد از هر ایپاک و محاسبه ی میانگین آنها در نهایت رسم نمودار تابع هزینه در نظر گرفته شده است:

```
generator.train()
discriminator.train()
g_loss_avg = []
d_loss_avg = []
```

- شروع انجام فرآیند آموزش مقداردهی اولیه صفر به لیست ها برای تسهیل در عملیات محاسبه میانگین خطا، خواندن یک batch از trainloader قرار دادن آن برای روی device مشخص شده و ایجاد لیبل های یک برای داده های real و لیبل های صفر برای داده های fake و قرار دادن آنها بر روی device ایجاد یک لایه ی لایه latent متناسب با ابعاد اولیه در نظر گرفته شده. دادن لایه ی latent اولیه به generator و تولید تصاویر fake پیش بینی discriminator با داده ها های واقعی و تولدی شده توسط generator. محاسبه ی تابع هزینه ی generator و discriminator و بهینه سازی آن و عملیات backward. سپس محاسبه ی میانگین تابع هزینه و نمایش مقدار هزینه آخرین گام و میانگین تا آن بخش و در نهایت یک نمایش از generator در انتهای هر ایپاک:

```
print('Training ...')
for epoch in range(n_epochs):
    g_loss_avg.append(0)
    d_loss_avg.append(0)
    num_batches = 0
    for images, _ in train_loader:
        ################ Problem 07 (15 pts) ################
        # put your inputs on device
        # Prepare what you need for training, like inputs for modules and
variables for computing loss
        images = images.to(device)
        label_real = torch.ones(images.size(0), device=device)
        label_fake = torch.zeros(images.size(0), device=device)
        z = torch.randn(images.size(0), z_dim, 1, 1, device=device)
        fake_images = generator(z)
        real_pred = discriminator(images).squeeze()
        fake_pred = discriminator(fake_images.detach()).squeeze()

        ######################## End ########################



        ################ Problem 08 (10 pts) ################
        # calculate discriminator loss and update it
        d_loss = 0.5 * (
```

```
            F.binary_cross_entropy(real_pred, label_real) +
            F.binary_cross_entropy(fake_pred, label_fake))

        d_optimizer.zero_grad()
        d_loss.backward()
        d_optimizer.step()
        ######################## End ########################



        ################ Problem 09 (10 pts) ################
        # calculate generator loss and update it
        fake_pred = discriminator(fake_images).squeeze()
        g_loss = F.binary_cross_entropy(fake_pred, label_real)

        g_optimizer.zero_grad()
        g_loss.backward()
        g_optimizer.step()

        ######################## End ########################


    ################ Problem 10 (10 pts) ################
    # plot some of the generated pictures based on plot frequency variable
        g_loss_avg[-1] += g_loss.item()
        d_loss_avg[-1] += d_loss.item()
        num_batches += 1

    g_loss_avg[-1] /= num_batches
    d_loss_avg[-1] /= num_batches
    print("epoch: {} \t discriminator last batch loss: {} \t generator las
t batch loss: {}".format(epoch + 1,

                    d_loss.item(),

                    g_loss.item())
    )
    print('epoch: %d average loss generator vs. discrim.: %f vs. %f' %
          (epoch+1, g_loss_avg[-1], d_loss_avg[-1]))
    fakeimg = fake_images.detach().cpu()
    fakeimg = fakeimg.view(-1, 64, 64)
    fakeimg = fakeimg.numpy()
    cols = 5
    plt.rcParams['figure.figsize'] = (cols, 2)
    for i in range(10):
```

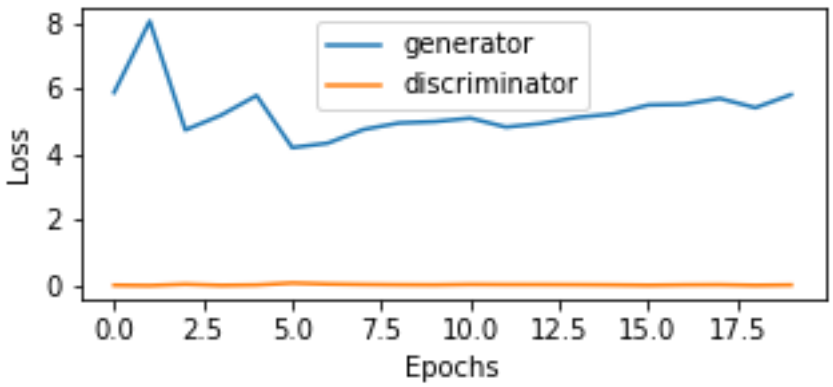
```
            plt.subplot(2, cols, i + 1)
            plt.imshow(fakeimg[i], cmap="gray", vmin=0, vmax=1)
            plt.axis('off')
        plt.show()
        ######################## End ########################
print('Finished :)')
```

❖ نمونه ای از خروجی های گام آموزش:

```
epoch: 10       discriminator last batch loss: 0.04833817481994629
generator last batch loss: 7.233254432678223
epoch: 10 average loss generator vs. discrim.: 4.993389 vs. 0.029124
```

❖ نماش مقادیر خطا به ازای generator و discriminator :

```
❖ import matplotlib.pyplot as plt
❖ plt.ion()
❖
❖ fig = plt.figure()
❖ plt.plot(g_loss_avg, label='generator')
❖ plt.plot(d_loss_avg, label='discriminator')
❖ plt.xlabel('Epochs')
❖ plt.ylabel('Loss')
❖ plt.legend()
❖ plt.show()
```



❖ ذخیره ی پارامترها و وزن های آموزش دیده شده برای generator به ازای 15 ایپوک:

```
torch.save(generator.state_dict(), 'generator.pth')
```