

1. 모델 구축하기

Pix2PixGAN은 이미지를 이미지로 Generator을 학습시키고, fake이미지를 출력하고, 그 이미지를 discriminator가 완성된 그림을 식별하도록 목적 함수를 설계하여, 학습 시키면서 서서히 generator은 완성된 fake그림을 출력하여, ground truth에 가까운 그림을 만들어 discriminator가 구별하지 못하도록 한다,

논문을 통해, UNet과 PacthGAN모델로 구성했을 때, 높은 성능의 결과로 보인다고하여, Pix2PixGAN을 구성할 때 활용했다. Train과 Test를 조건대로 8:2로 나누어 테스트했다.

□ Generator model

Fake이미지를 만드는 Generator의 UNet을 활용

generator: 가짜 이미지를 생성합니다.

class GeneratorUNet(nn.Module):

```
def __init__(self, in_channels=3, out_channels=3):
    super().__init__()
```

```
self.down1 = UNetDown(in_channels, 64, normalize=False)
self.down2 = UNetDown(64,128)
self.down3 = UNetDown(128,256)
self.down4 = UNetDown(256,512,dropout=0.5)
self.down5 = UNetDown(512,512,dropout=0.5)
self.down6 = UNetDown(512,512,dropout=0.5)
self.down7 = UNetDown(512,512,dropout=0.5)
self.down8 = UNetDown(512,512,normalize=False,dropout=0.5)
```

```
self.up1 = UNetUp(512,512,dropout=0.5)
self.up2 = UNetUp(1024,512,dropout=0.5)
self.up3 = UNetUp(1024,512,dropout=0.5)
self.up4 = UNetUp(1024,512,dropout=0.5)
self.up5 = UNetUp(1024,256)
self.up6 = UNetUp(512,128)
self.up7 = UNetUp(256,64)
self.up8 = nn.Sequential(
    nn.ConvTranspose2d(128,3,4,stride=2,padding=1),
    nn.Tanh()
)
```

```
def forward(self, x):
```

```
    d1 = self.down1(x)
    d2 = self.down2(d1)
    d3 = self.down3(d2)
    d4 = self.down4(d3)
    d5 = self.down5(d4)
    d6 = self.down6(d5)
    d7 = self.down7(d6)
    d8 = self.down8(d7)
```

```
    u1 = self.up1(d8,d7)
```

<pre> u2 = self.up2(u1,d6) u3 = self.up3(u2,d5) u4 = self.up4(u3,d4) u5 = self.up5(u4,d3) u6 = self.up6(u5,d2) u7 = self.up7(u6,d1) u8 = self.up8(u7) return u8 </pre>	
<pre> class UNetUp(nn.Module): def __init__(self, in_channels, out_channels, dropout=0.0): super().__init__() layers = [nn.ConvTranspose2d(in_channels, out_channels,4,2,1,bias=False), nn.InstanceNorm2d(out_channels), nn.LeakyReLU()] if dropout: layers.append(nn.Dropout(dropout)) self.up = nn.Sequential(*layers) def forward(self,x,skip): x = self.up(x) x = torch.cat((x,skip),1) return x </pre>	<pre> class UNetDown(nn.Module): def __init__(self, in_channels, out_channels, normalize=True, dropout=0.0): super().__init__() layers = [nn.Conv2d(in_channels, out_channels, 4, stride=2, padding=1, bias=False)] if normalize: layers.append(nn.InstanceNorm2d(out_channels)), layers.append(nn.LeakyReLU(0.2)) if dropout: layers.append(nn.Dropout(dropout)) self.down = nn.Sequential(*layers) def forward(self, x): x = self.down(x) return x </pre>

□ Discriminator model

Discriminator은 patch gan을 활용했다. Patch Gan을 활용하면, 이미지를 16x16의 패치로 분할하여 각 패치가 진짜인지 가짜인지 식별한다. high-frequency에서 정확도가 향상된다.

```
class Dis_block(nn.Module):
```

```
    def __init__(self, in_channels, out_channels, normalize=True):
        super().__init__()
```

```
        layers = [nn.Conv2d(in_channels, out_channels, 3, stride=2, padding=1)]
        if normalize:
            layers.append(nn.InstanceNorm2d(out_channels))
        layers.append(nn.LeakyReLU(0.2))
```

```
        self.block = nn.Sequential(*layers)
```

```
    def forward(self, x):
        x = self.block(x)
        return x
```

```
class Discriminator(nn.Module):
```

```
    def __init__(self, in_channels=3):
        super().__init__()
```

```
        self.stage_1 = Dis_block(in_channels*2,64,normalize=False)
        self.stage_2 = Dis_block(64,128)
        self.stage_3 = Dis_block(128,256)
        self.stage_4 = Dis_block(256,512)
```

```
        self.patch = nn.Conv2d(512,1,3,padding=1) # 16x16 패치 생성
```

```
    def forward(self,a,b):
        x = torch.cat((a,b),1)
        x = self.stage_1(x)
        x = self.stage_2(x)
        x = self.stage_3(x)
        x = self.stage_4(x)
        x = self.patch(x)
        x = torch.sigmoid(x)
        return x
```

2. 학습 및 결과

□ 학습하기

```
model_gen = GeneratorUNet().to(DEVICE)
model_dis = Discriminator().to(DEVICE)
# 가중치 초기화 적용
model_gen.apply(initialize_weights)
model_dis.apply(initialize_weights)

# 손실함수
loss_func_gen = nn.BCELoss()
loss_func_pix = nn.L1Loss()

# loss_func_pix 가중치
lambda_pixel = 100

# patch 수
patch = (1,256//2**4,256//2**4)

# 최적화 파라미터
from torch import optim
lr = 2e-4
beta1 = 0.5
beta2 = 0.999

opt_dis = optim.Adam(model_dis.parameters(),lr=lr,betas=(beta1,beta2))
opt_gen = optim.Adam(model_gen.parameters(),lr=lr,betas=(beta1,beta2))

# 학습
model_gen.train()
model_dis.train()

for epoch in range(num_epochs):
    print("success DataSet")
    for idx, data in enumerate(DataLoader):
        noise, gt = data['img'], data['gt']

        noise = noise.type(torch.FloatTensor) / 255
        gt = gt.type(torch.FloatTensor) / 255
        ba_si = noise.size(0)

        #print("real image")
        real_a = noise.to(DEVICE)
        real_b = gt.to(DEVICE)

        # patch label
        #print("patch label")
        real_label = torch.ones(ba_si, *patch, requires_grad=False).to(DEVICE)
        fake_label = torch.zeros(ba_si, *patch, requires_grad=False).to(DEVICE)
```

```

# generator
#print("generator")
model_gen.zero_grad()

#print("model")
fake_b = model_gen(real_a)
out_dis = model_dis(fake_b, real_b)

gen_loss = loss_func_gan(out_dis, real_label)
pixel_loss = loss_func_pix(fake_b, real_b)

g_loss = gen_loss + lambda_pixel * pixel_loss
g_loss.backward()
opt_gen.step()

# discriminator
#print("discriminator")
model_dis.zero_grad()

out_dis = model_dis(real_b, real_a)
real_loss = loss_func_gan(out_dis, real_label)

out_dis = model_dis(fake_b.detach(), real_a)
fake_loss = loss_func_gan(out_dis, fake_label)

d_loss = (real_loss + fake_loss) / 2.
d_loss.backward()
opt_dis.step()

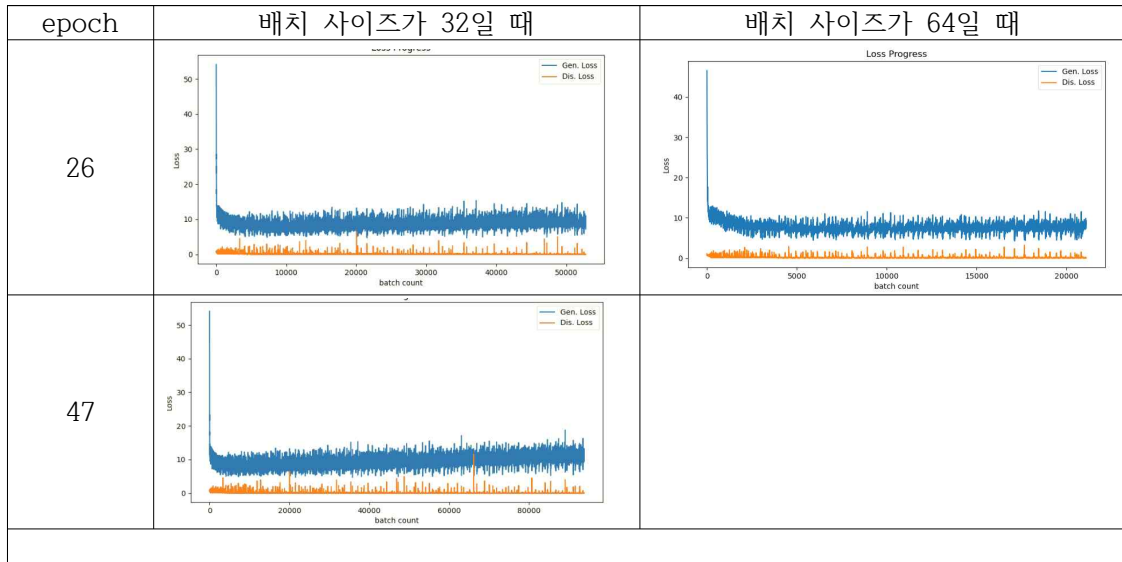
#print("loss_hist")
loss_hist['gen'].append(g_loss.item())
loss_hist['dis'].append(d_loss.item())

#batch_count += 1
#if batch_count % 100 == 0:
print('Epoch: %.0f, G_Loss: %.6f, D_Loss: %.6f, time: %.2f min' % (
    epoch, g_loss.item(), d_loss.item(), (time.time() - start_time) / 60))

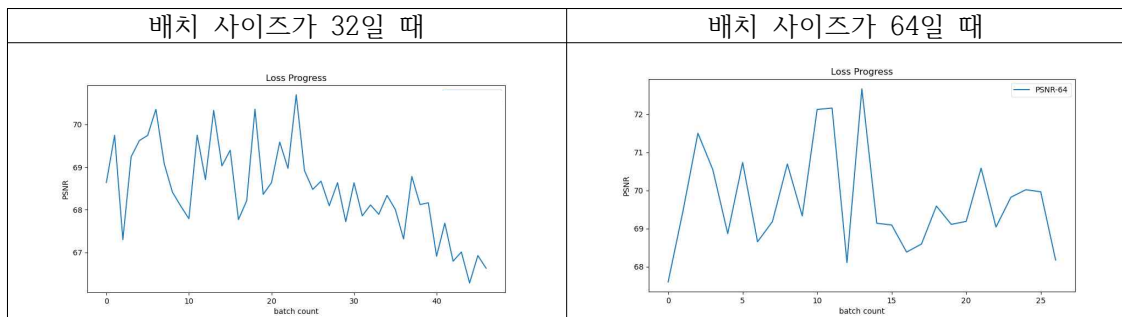
```

Generator는 BCELoss함수를 사용하여, 클래스가 2개인 경우로 사용하고, Discriminator은 L1loss로 실제값과 예측값 사이의 차이값에 절대값을 취해 오차 합을 최소화하는 방향으로 loss를 구하여 사용한다.

□ batch size 32/64 비교



□ Generator이 생성한 PSNR로 비교하기

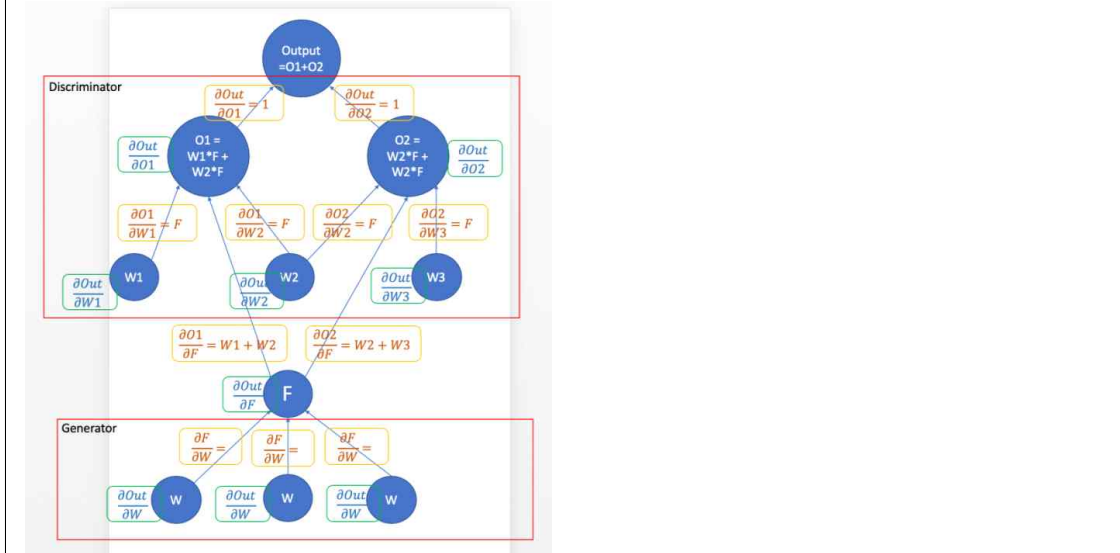


배치 사이즈가 32/64일 때의 PSNR을 비교 했을 때, 학습 시간에 비해 64일 때가 효과적인 것을 알 수있다. 32일 때는 epoch이 5, 14, 18, 23일 때가 높았고, epoch이 23 일 때 **PSNR =70.3541** 로 제일 높았고, 64일 때, epoch이 10~13 사이가 높았으며, **PSNR=73.9459**로 제일 높았다.

□ train 시, discriminator의 입력에 생성영상이 들어갈 때 어째서 detach()가 붙는지에 대해서도 고민!!

fake 이미지를 Discriminator에 넣어 real/fake를 판단하는데, 이 과정에서 위의 real 이미지와 달리 detach()가 붙는다.

.detach()는 원래 requires_grad = False로 만드는 역할을 한다. 밑에 그림을 참고하면, fake 이미지는 Generator로부터 만들어지기 때문에 fake에 .detach()를 붙이지 않을 경우, Generator weight들의 grad까지 계산하게 된다. 하여 net만 업데이트하고 Generator에까지 backpropagation을 할 수 없도록 하기 위해 .detach()를 붙여야 한다.



□ PSNR 연산

최대신호대잡음비(peak signal-to-noise ratio, PSNR)는 신호가 가질 수 있는 최대 전력에 대한 잡음의 전력을 나타낸 것이다. 주로 영상 또는 동영상 손실 압축에서 화질 손실 정보를 평가할 때 사용된다. 아래 밑에 수식으로 계산한다.

```
import numpy as np
```

```
import math
```

```
import cv2
```

```
from skimage import io, transform
```

```
from PIL import Image
```

```
def psnr(orgin_img, pred_img):
```

```
    #orgin = cv2.imread(orgin_img)
```

```
    #pred = cv2.imread(pred_img)
```

```
    #orgin = np.array(Image.open(orgin_img).convert('RGB'))
```

```
    #pred = np.array(Image.open(pred_img).convert('RGB'))
```

```
    print(type(orgin))
```

```
    mse = np.mean((orgin - pred) ** 2)
```

```
    print("mse : ", mse)
```

```
    if mse == 0:
```

```
        return 100
```

$$\begin{aligned} PSNR &= 10 \log_{10} \left(\frac{MAX_I^2}{MSE} \right) \\ &= 20 \log_{10} \left(\frac{MAX_I}{\sqrt{MSE}} \right) \\ &= 20 \log_{10}(MAX_I) - 10 \log_{10}(MSE) \end{aligned}$$

```










PIXEL_MAX = 255.0
return 20 * math.log10(PIXEL_MAX / math.sqrt(mse))

fake_imgs = "input.png"
real_imgs = "gt_test.png"

d = psnr(real_imgs, fake_imgs)
print(d)

```

□ 각 32, 64 배치사이즈마다 각 에폭의 모델을 저장하여, 비교한 결과

epoch	BATCH 32 일때		BATCH 64 일 때		
	(GT/generator)순서로 ground truth = target(정답) generator = gerator model 결과		(input/GT/generator)순서로 ground truth = target(정답) generator = gerator model 결과		
1			1		
5			3		
14			5		
18			10		
23			11		
30			12		
40			13		
47			16		



□ 각 에폭마다 테스트한 결과 모습

