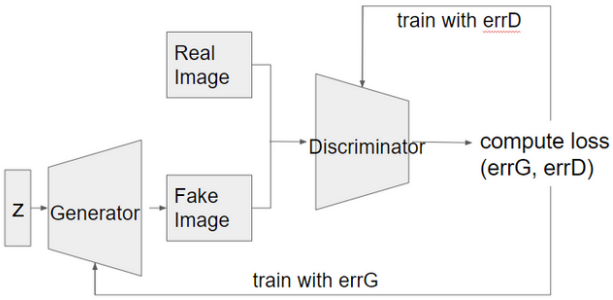
Summary Report [Week 14]. DCGAN

20190106 KimByungjun (김병준)

● Exercises

이번 Exercise 에선 실제와 가까운 (가짜) data 를 생성해내는 GAN model 중 Convolutional Network 을 기반으로 구성된 DCGAN 을 구현하고, 실제 CIFAR10 image dataset 을 input 으로 받아 image generation 을 시행한다. 우선, GAN 에 대해 알아보자.



[Fig 1] Structure of GAN

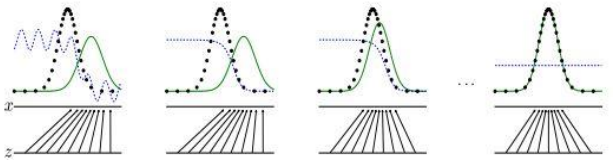
GAN 의 구성은 **[Fig 1]**과 같다. 최종 목표는 실제 데이터의 분포를 학습하여 그 분포를 따르는 data(image 등)를 만드는 것이지만, 해당 목표를 달성하기 위해 두 model 인 생성자 generator 와 판별자 discriminator 를 학습시킨다. 이 때, 생성자의 목표는 실제와 유사한 데이터 생성(unsupervised)이고, 판별자의 목표는 생성자가 만든 가짜와 진짜 훈련 데이터를 완벽히 구별하는 것(supervised)이다.

$$\min_G \max_D V(D,G) = E_{x\sim p_{data}(x)}[logD(x)] + E_{z\sim P_z(z)}[\log(1-D(G(z))]$$

$D(x)=1$일 때 최대 / $D(G(z))=1$일 때 최소 / $D(G(z))=0$일 때 최대

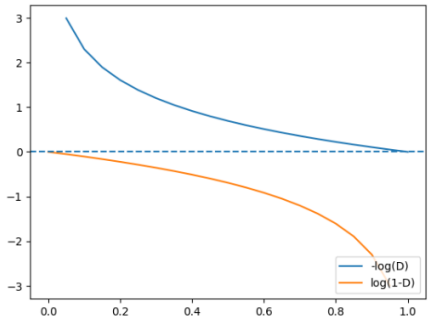
[Fig 2] Loss function for GAN

다시 말하자면, 생성자 G 의 목표는 판별자가 fake image 에 (real image 에 대응되는) label 1 을 assign 하는 것 $(\mathrm{D}(\mathrm{G}(\mathrm{z}))=1)$, 판별자 D 의 목표는 real image 에는 label 1, fake image 에는 label 0 을 assign 하는 것이다$(\mathrm{D}(\mathrm{x})=1, \mathrm{D}(\mathrm{G}(\mathrm{z}))=0)$. 해당 목표를 달성하기 위해, real or fake 를 구별하는 상황에 맞게 loss 는 Binary Cross Entropy Loss function 을 사용한다. 식은 **[Fig 2]**와 같으며, G 와 D model 이 가지는 목표가 다름에 유의하자. 이상적인 Training 의 결과로, [Fig 3]의 검은색 real distribution 를 따라하려는 초록색 fake distribution 을 생성자 G 가 만들어내고, 학습할수록 fake 가 real 을 따라가게 된다. 파랑색은 판별자 D 의 예측인데, 초반엔 진짜와 가짜를 잘 구별하다가, 학습이 진행될수록 fake or real 을 구분하지 못해 $\mathrm{D}(\mathrm{G}(\mathrm{z}))=0.5$ 에 수렴할 것이다.



[Fig 3] Theorectial Result of GAN training

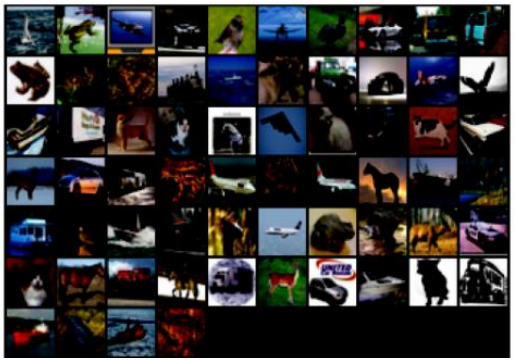
Note. [Fig 2]의 두번째 항에서 $\log(1-\mathrm{D}(\mathrm{G}(\mathrm{z})))$ 는 **[Fig 4]**와 같이 $\mathrm{D}(\mathrm{G}(\mathrm{z}))$가 0 에 가까울 때 gradient 가 0 에 근접하고, 이는 생성자 G 의 학습에 방해될 수 있다. 초반 traning 에 $\mathrm{D}(\mathrm{G}(\mathrm{z}))$는 0 와 매우 유사한 값을 가지게 되므로 해당 gradient vanishing 문제가 발생할 여지가 크고, 이로 인해 $-\log \mathrm{D}(\mathrm{G}(\mathrm{z}))$를 대신 사용하는 것이 오히려 도움될 수 있다.



[Fig 4] Graph for log(1-D) and -log(D)

여러 GAN model 중 두 model G 와 D 의 안정화에 크게 기여한 model 이 Convolutional Layer 를 응용한 DCGAN model 이며, 생성자 G 에서는 De-convolution 을 통해 차원을 증가시킨 뒤, 판별자 D 에선 Convolution 으로 차원 축소를 한다. 해당 model 에서 생성자 G 는 noise 를 이용하여 fake image 를 생성하고, training 을 거쳐 noise 를 real data 와 유사하게 만드는 distribution 을 배워간다.

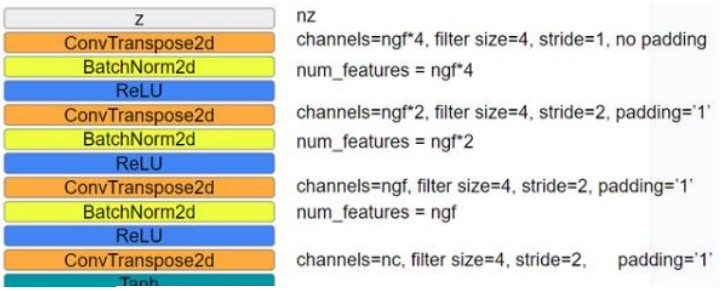
학습은 GPU 를 이용하여 진행되며, 사용되는 dataset 은 **[Fig 5]**와 같은 RGB 채널의 CIFAR10 image 이다. 이제 Generator G 를 implement 해보자.



[Fig 5] CIFAR10 Dataset Images

1) Construct generator. Define self.main using nn.Sequential. Detailed structure is suggested in [Fig 6]. The last layer is Tanh activation.

<Explanation>



[Fig 6] Structure of Generator netG

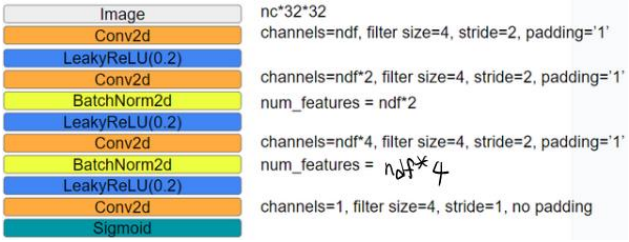
Generator 에선 앞서 설명하였듯이 De-convolution 이 진행되며, nn.ConvTranspose2d 함수로 구현된다. 각각의 De-convolution 사이에 batch normalization 과 ReLU activation 이 적용되며, 마지막 de-convolution 이후에는 tanh activation 이 진행된다. 이 흐름을 그대로 code 로 구현한 것이 **[Fig 7]**이다. Model 을 print 하면 **[Fig 10]** 과 같이 최종 channel 3 의 output 을 반환한다.

```
nc = 3 # number of channels. RGB
nz = 100 # input noise dimension
ngf = 64 # number of generator filters
ndf = 64 #number of discriminator filters

class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        ###################################
        ###### Task 1. Construct generator ####
        ### Define self.main ###
        self.main = nn.Sequential(
            nn.ConvTranspose2d(nz, ngf*4, kernel_size=4, stride=1, padding=0),
            nn.BatchNorm2d(ngf*4),
            nn.ReLU(),

            nn.ConvTranspose2d(ngf*4, ngf*2, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(ngf*2),
            nn.ReLU(),

            nn.ConvTranspose2d(ngf*2, ngf, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(ngf),
            nn.ReLU(),

            nn.ConvTranspose2d(ngf, nc, kernel_size=4, stride=2, padding=1),
            nn.Tanh()
        )
        ###################################
        ###################################

    def forward(self, input):
        output = self.main(input)
        return output
```

[Fig 7] Code for Exercise 1

2) Construct discriminator. Define self.main using nn.Sequential. Detailed structure is suggested in [Fig 8].

<Explanation>



[Fig 8] Structure of Discriminator netD

```
[10] class Discriminator(nn.Module):
        def __init__(self):
            super(Discriminator, self).__init__()
            ###################################
            ###### Task 2. Construct discriminator #
            ### Define self.main ###
            self.main = nn.Sequential(
              nn.Conv2d(nc,ndf,kernel_size=4, stride=2, padding=1),
              nn.LeakyReLU(0.2),

              nn.Conv2d(ndf,ndf*2,kernel_size=4, stride=2, padding=1),
              nn.BatchNorm2d(ndf*2),
              nn.LeakyReLU(0.2),

              nn.Conv2d(ndf*2,ndf*4,kernel_size=4, stride=2, padding=1),
              nn.BatchNorm2d(ndf*4),
              nn.LeakyReLU(0.2),

              nn.Conv2d(ndf*4,1,kernel_size=4, stride=1, padding=0),
              nn.Sigmoid()
            )
            ###################################
            ###################################
        def forward(self, input):
            output = self.main(input)
            return output.view(-1, 1).squeeze(1)
```

[Fig 9] Code for Exercise 2

Discriminator 에서는 Image 에 대한 convolution 을 진행하는 general 한 CNN model 이다. 다만, ReLU 의 value 가 0 으로 knock out 되어 향후 학습이 진행되지 않는 문제를 방지하기 위해 x<0 영역에선 0.2 의 기울기를 부여하는 nn.LeakyReLU(0.2) layer 를 적용한다(대신 연산의 양이 증가한다). 마지막으로, channel 을 하나로 줄이고, sigmoid activation 을 적용해 [0,1] 사이의 값을 반환하게 한다. Code 는 **[Fig 9]**와 같다.

구현된 model 의 상태를 확인하기 위해, model 을 출력해보면 **[Fig 10]**처럼 내부 layer 구성을 알 수 있다. Loss function 으로는 앞서 말한 Binary Cross-Entropy loss (nn.BCELoss)를, optimizer 로는 Adam 을 learning rate = 0.0002, batch size = 64, epochs = 20 의 hyperparameters 와 함께 training 으로 설정한다. 이제 model 을 학습시킨 뒤, generator 가 만들어내는 fake image 의 quality 를 보도록 하자.

[Fig 11B]을 보면 두 model 이 경쟁적으로 학습하는 것을 볼 수 있으며, 어느 한 쪽이 확실히 작아진다는 경향은 크게 보이지 않는다. 진동하는 폭 역시 크며, 직관적인 해석이 어려운 결과를 보여준다. 여담으로, generator 의 loss 는 전체적으로 소폭 감소하려는 경향을 보였다.

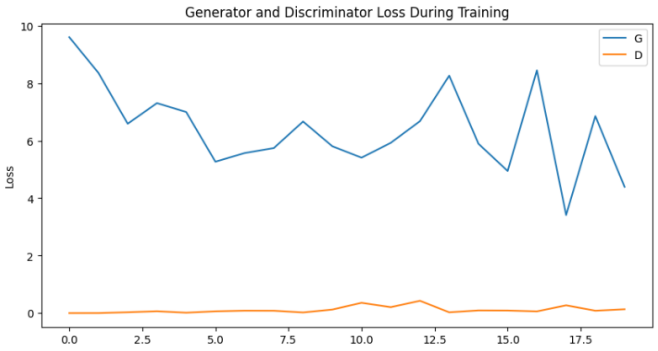
가장 중요한 training 이후 generator 의 fake image quality 이다. **[Fig 12]**는 훈련 초기(처음)에 generator 가 만들어낸 image 이며, 그냥 noise 에 가깝다고 봐도 무방할 정도로 엉망임을 알 수 있다.

```
print(netG)
print(netD)

Generator(
  (main): Sequential(
    (0): ConvTranspose2d(100, 256, kernel_size=(4, 4), stride=(1, 1))
    (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU()
    (6): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (7): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU()
    (9): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (10): Tanh()
  )
)
Discriminator(
  (main): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (1): LeakyReLU(negative_slope=0.2)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2)
    (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2)
    (8): Conv2d(256, 1, kernel_size=(4, 4), stride=(1, 1))
    (9): Sigmoid()
  )
)
```

[Fig 10] Layer components of Generator and Discriminator

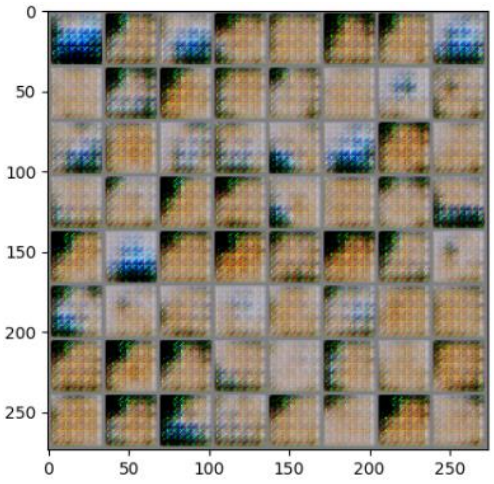
```
[0/20] Loss_D: 0.0029 Loss_G: 9.6012 D(x): 1.0000 D(G(z)): 0.0028 / 0.0005
[1/20] Loss_D: 0.0041 Loss_G: 8.3547 D(x): 0.9998 D(G(z)): 0.0039 / 0.0007
[2/20] Loss_D: 0.0323 Loss_G: 6.5885 D(x): 0.9762 D(G(z)): 0.0044 / 0.0021
[3/20] Loss_D: 0.0668 Loss_G: 7.3058 D(x): 0.9986 D(G(z)): 0.0575 / 0.0065
[4/20] Loss_D: 0.0170 Loss_G: 6.9976 D(x): 0.9934 D(G(z)): 0.0101 / 0.0036
[5/20] Loss_D: 0.0646 Loss_G: 5.2682 D(x): 0.9951 D(G(z)): 0.0545 / 0.0154
[6/20] Loss_D: 0.0871 Loss_G: 5.5691 D(x): 0.9339 D(G(z)): 0.0097 / 0.0090
[7/20] Loss_D: 0.0849 Loss_G: 5.7427 D(x): 0.9976 D(G(z)): 0.0726 / 0.0098
[8/20] Loss_D: 0.0251 Loss_G: 6.6663 D(x): 0.9811 D(G(z)): 0.0051 / 0.0072
[9/20] Loss_D: 0.1256 Loss_G: 5.8060 D(x): 0.9747 D(G(z)): 0.0785 / 0.0124
[10/20] Loss_D: 0.3600 Loss_G: 5.4084 D(x): 0.8515 D(G(z)): 0.0268 / 0.0343
[11/20] Loss_D: 0.2091 Loss_G: 5.9243 D(x): 0.9255 D(G(z)): 0.0676 / 0.0222
[12/20] Loss_D: 0.4301 Loss_G: 6.6790 D(x): 0.7996 D(G(z)): 0.0745 / 0.0111
[13/20] Loss_D: 0.0304 Loss_G: 8.2604 D(x): 0.9717 D(G(z)): 0.0008 / 0.0006
[14/20] Loss_D: 0.0956 Loss_G: 5.8931 D(x): 0.9467 D(G(z)): 0.0143 / 0.0077
[15/20] Loss_D: 0.0921 Loss_G: 4.9443 D(x): 0.9621 D(G(z)): 0.0351 / 0.0207
[16/20] Loss_D: 0.0623 Loss_G: 8.4460 D(x): 0.9773 D(G(z)): 0.0292 / 0.0064
[17/20] Loss_D: 0.2729 Loss_G: 3.4117 D(x): 0.9582 D(G(z)): 0.1712 / 0.0805
[18/20] Loss_D: 0.0860 Loss_G: 6.8538 D(x): 0.9357 D(G(z)): 0.0076 / 0.0044
[19/20] Loss_D: 0.1362 Loss_G: 4.3933 D(x): 0.9836 D(G(z)): 0.1019 / 0.0322
```

[Fig 11A] Loss and returning values



[Fig 11B] Average loss for two models

훈련 이후, generator 가 만들어낸 image **[Fig 13]**를 살펴보자. **[Fig 5]**의 원본과 비교하면 역시 quality 가 낮지만, **[Fig 12]**에 비해 적어도 배경과 대상이 구분되는 경계는 있다고 확신할 수 있는 image 가 형성되었다. **[Fig 11A]**의 D(x) 값에서 볼 수 있듯이, 아직 Discriminator 가 헷갈릴 정도의 image 를 Generator 가 만들어낼 순 없지만, 20 epoch 만에 image quality 차이가 크게 난다는 것을 확인할 수 있었다.

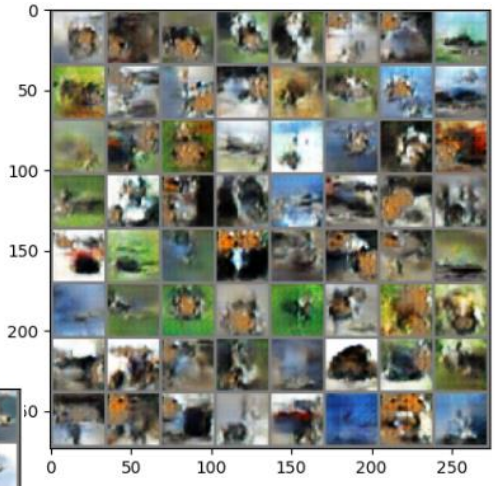


[Fig 12] Generated Image at the beginning of training

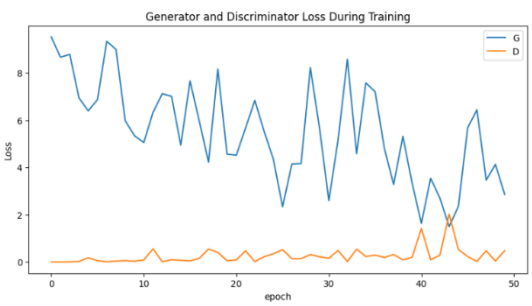
● Discussion

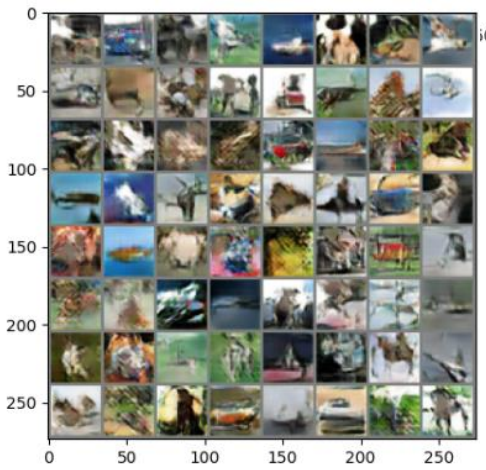
1) Training for larger epoch

Epoch 를 20 에서 50 으로 늘려서 동일하게 training 을 진행한다면, **[Fig 14]**의 결과들을 얻을 수 있다. **[Fig 13]**과 비교하면 화질이 더 좋아지는 듯하며 대상이 배경과 구별되어 더 clear 하게 보이는 것을 알 수 있다. 이는 보다 더 많은 training 을 통해 model 이 더 robust 해졌다는 것을 시사하며, 더 많은 epoch 와 model 을 정교하게 수정하면 performance 는 추가로 더 향상될 수 있음을 알려준다. 다만, GAN 의 Generator 와 Discriminator 의 경쟁적 학습 특성상 model 자체의 불안전성은 완전히 해소되기 어려우며, 해당 Exercise 의 구조만으론 원본 데이터와 거의 판박이의 image 를 생성하는 것은 어렵다고 본다.



[Fig 13] Generated Image at the end of training (epoch = 20)





[Fig 14] Average loss(above) and generated image(right) for model trained with epoch 50

● References

✓ https://velog.io/@tobigs-gm1/basicofgan