exam 개인학습

Lec 3.3

AI summary Try on this page

Type something...

01) Generative Adversarial Networks

01_sample_noise

- GAN에 대한 이론적인 설명은 논문 리뷰와 cs231n 강의 정리에서 확인할 수 있다.
- 먼저 noise = z를 생성해보자.
- `rand` 는 (0, 1)사이의 값들을 생성해 주기 때문에 -2를 곱하고 1을 더해서 (-1, 1)로 구간을 변경시켜주었다.

```
def sample_noise(batch_size, dim, seed=None):
    """
    Generate a PyTorch Tensor of uniform random noise.

    Input:
    - batch_size: Integer giving the batch size of noise to generate.
    - dim: Integer giving the dimension of noise to generate.

    Output:
    - A PyTorch Tensor of shape (batch_size, dim) containing uniform
      random noise in the range (-1, 1).
    """
    if seed is not None:
        torch.manual_seed(seed)

    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    noise = -2 * torch.rand(batch_size, dim) + 1
    return noise

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

02_discriminator

```
model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(784, 256, bias = True),
    nn.LeakyReLU(0.01),
    nn.Linear(256, 256),
    nn.LeakyReLU(0.01),
    nn.Linear(256, 1)

)
```

- `nn.Sequential` 을 이용해서 Linear + activation로 모델을 만들었다.
- `Flatten` 으로 이미지 x: 128 x 1 x 28 x 28 를 128 x 784로 만들어준 후 layer들을 통과시켰다.

03_Generator

```
model = nn.Sequential(
    nn.Linear(noise_dim, 1024),
    nn.ReLU(),
    nn.Linear(1024, 1024),
    nn.ReLU(),
    nn.Linear(1024, 784),
    nn.Tanh()
)
```

- `tanh` 로 -1에서 1 사이로 normalization을 하기 위해 사용하였다.

04_GAN Loss

GAN Loss

Compute the generator and discriminator loss. The generator loss is:

$$\ell_G = -\mathbb{E}_{z \sim p(z)}\left[\log D(G(z))\right]$$

and the discriminator loss is:

$$\ell_D = -\mathbb{E}_{x \sim p_{\text{data}}}\left[\log D(x)\right] - \mathbb{E}_{z \sim p(z)}\left[\log(1 - D(G(z)))\right]$$

Note that these are negated from the equations presented earlier as we will be *minimizing* these losses.

HINTS: You should use the `bce_loss` function defined below to compute the binary cross entropy loss which is needed to compute the log probability of the true label given the logits output from the discriminator. Given a score $s \in \mathbb{R}$ and a label $y \in \{0, 1\}$, the binary cross entropy loss is

$$bce(s, y) = -y * \log(s) - (1 - y) * \log(1 - s)$$

A naive implementation of this formula can be numerically unstable, so we have provided a numerically stable implementation that relies on PyTorch's `nn.BCEWithLogitsLoss`.

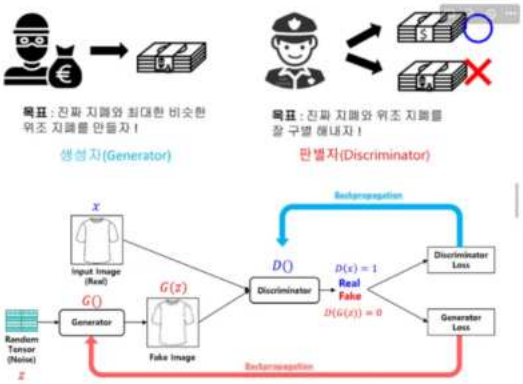
You will also need to compute labels corresponding to real or fake and use the logit arguments to determine their size. Make sure you cast these labels to the correct data type using the global `dtype` variable, for example:

```
true_labels = torch.ones(size).type(dtype)
```

Instead of computing the expectation of $\log D(G(z))$, $\log D(x)$ and $\log(1 - D(G(z)))$, we will be averaging over elements of the minibatch. This is taken care of in `bce_loss` which combines the loss by averaging.

Implement `discriminator_loss` and `generator_loss` in `cs231n/gan_pytorch.py`

- generator loss, discriminator loss는 아까 함수에서 음수를 취해서 얻을 수 있다.
- 음수를 취하는 이유는 아까는 문제를 최대화시키는 것이었는데 이를 음수를 취해서 최소화 문제로 변형시켰다.
- `bce(s,y)` : score = s, label = y, 직접 구현하면 numerically unstable하기 때문에 Pytorch에서 구현된 `nn.BCEWithLogitsLoss` 를 사용하여 Loss를 계산하자.
- `bce_loss` 에서 loss들을 averaging하여 계산하자.
- real image = 1, fake image = 0일 때의 binary cross entropy loss를 구해보자.
- **1이며 bce에서 앞부분만, 0이면 bce에서 뒷부분만 남게된다.**




```python
def discriminator_loss(logits_real, logits_fake):
    """
    Computes the discriminator loss described above.

    Inputs:
    - logits_real: PyTorch Tensor of shape (N,) giving scores for the real data.
    - logits_fake: PyTorch Tensor of shape (N,) giving scores for the fake data.

    Returns:
    - loss: PyTorch Tensor containing (scalar) the loss for the discriminator.
    """
    loss = None
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    # real + fake loss 더해주기
    bce_real = bce_loss(logits_real.reshape(-1), torch.ones_like(logits_real).reshape(-1))
    bce_fake = bce_loss(logits_fake.reshape(-1), torch.zeros_like(logits_fake).reshape(-1))
    loss = bce_real + bce_fake
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    return loss
```

- real data는 1과 비슷해야 하므로 1과 bce_loss를 구해주고 fake data는 0과 비슷해야 하므로 0과 bce_loss를 구해준다.

```python
def generator_loss(logits_fake):
    """
    Computes the generator loss described above.

    Inputs:
    - logits_fake: PyTorch Tensor of shape (N,) giving scores for the fake data.

    Returns:
    - loss: PyTorch Tensor containing the (scalar) loss for the generator.
    """
    loss = None
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    loss = bce_loss(logits_fake.reshape(-1), torch.ones_like(logits_fake).reshape(-1))

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    return loss
```

- generator의 경우 fake가 자신에게는 목표하는 이미지이므로 1과 비교하여 loss를 생성한다.

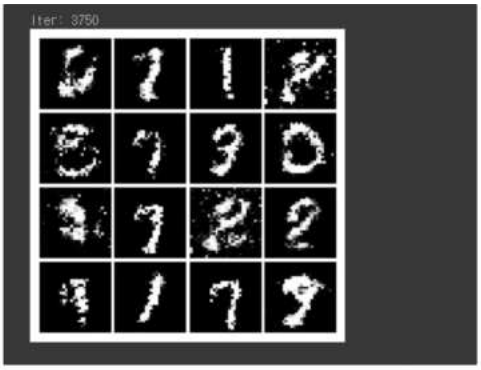
05_get_optimizer

```python
def get_optimizer(model):
    """
    Construct and return an Adam optimizer for the model with learning rate 1e-3,
    beta1=0.5, and beta2=0.999.

    Input:
    - model: A PyTorch model that we want to optimize.

    Returns:
    - An Adam optimizer for the model with the desired hyperparameters.
    """
    optimizer = None
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    optimizer = torch.optim.Adam(model.parameters(), lr = 1e-3, betas = (0.5, 0.999))

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    return optimizer
```

- optimizer는 `Adam` 을 사용하였다.
- model.parameters(), learning_rate, beta를 넣어주었다.

```python
def run_a_gan(D, G, D_solver, G_solver, discriminator_loss, generator_loss, loader_train, show_every=250,
              batch_size=128, noise_size=96, num_epochs=10):
    """
    Train a GAN!

    Inputs:
    - D, G: PyTorch models for the discriminator and generator
    - D_solver, G_solver: torch.optim Optimizers to use for training the
      discriminator and generator.
    - discriminator_loss, generator_loss: Functions to use for computing the generator and
      discriminator loss, respectively.
    - show_every: Show samples after every show_every iterations.
    - batch_size: Batch size to use for training.
    - noise_size: Dimension of the noise to use as input to the generator.
    - num_epochs: Number of epochs over the training dataset to use for training.
    """
    images = []
    iter_count = 0
    for epoch in range(num_epochs):
        for x, _ in loader_train:
            if len(x) != batch_size:
                continue
            D_solver.zero_grad()
            real_data = x.type(dtype)
            logits_real = D((2* (real_data - 0.5)).type(dtype))

            g_fake_seed = sample_noise(batch_size, noise_size).type(dtype)
            fake_images = G(g_fake_seed).detach()
            logits_fake = D(fake_images.view(batch_size, 1, 28, 28))

            d_total_error = discriminator_loss(logits_real, logits_fake)
            d_total_error.backward()
            D_solver.step()

            G_solver.zero_grad()
            g_fake_seed = sample_noise(batch_size, noise_size).type(dtype)
            fake_images = G(g_fake_seed)

            gen_logits_fake = D(fake_images.view(batch_size, 1, 28, 28))
            g_error = generator_loss(gen_logits_fake)
            g_error.backward()
            G_solver.step()

            if (iter_count % show_every == 0):
                print('Iter: {}, D: {:.4}, G:{:.4}'.format(iter_count,d_total_error.item(),g_error.item()))
                imgs_numpy = fake_images.data.cpu().numpy()
                images.append(imgs_numpy[0:16])

            iter_count += 1
```

- `run_a_gan` 함수로 train을 해보자.
- 입력으로는 D, G, D_solver, G_solver, discriminator loss, generator loss를 받는다.
- **loader_train으로 x값을 받은 후 Discriminator을 통과시켜 logits_real를 얻는다.**
- g_fake_seed의 noise 이미지를 Generator을 통과시켜 fake_image를 얻는다.
- **fake_image를 Discriminator를 통과시켜 logits_fake를 얻는다.**
- **각각의 에러에 대하여 backpropagation을 진행시킨 후 업데이트를 시킨다.**
- Discriminator → Generator 순서로 업데이트를 진행한다.
- **G 고정 → D update로 최적으로 수렴 → G 업데이트 반복**



- 반복할수록 숫자가 잘 나오는 것을 확인할 수 있다.

06_Least Squares GAN(LSGAN)

Least Squares GAN

We'll now look at Least Squares GAN, a newer, more stable alernative to the original GAN loss function. For this part, all we have to do is change the loss function and retrain the model. We'll implement equation (9) in the paper, with the generator loss:

$$\ell_G = \frac{1}{2}\mathbb{E}_{z\sim p(z)}\left[(D(G(z))-1)^2\right]$$

and the discriminator loss:

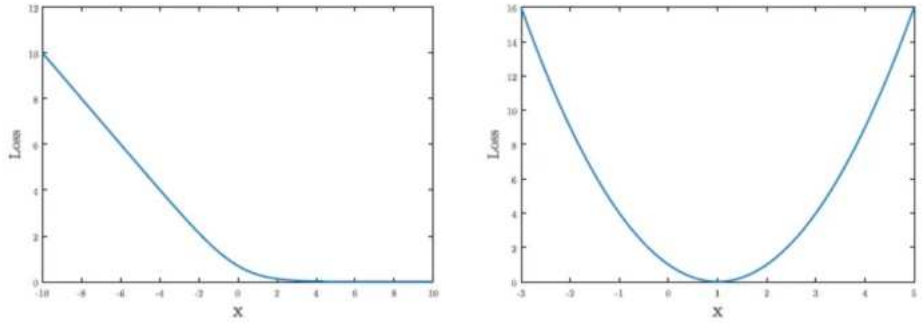
$$\ell_D = \frac{1}{2}\mathbb{E}_{x\sim p_{\text{data}}}\left[(D(x)-1)^2\right] + \frac{1}{2}\mathbb{E}_{z\sim p(z)}\left[(D(G(z)))^2\right]$$

HINTS: Instead of computing the expectation, we will be averaging over elements of the minibatch, so make sure to combine the loss by averaging instead of summing. When plugging in for $D(x)$ and $D(G(z))$ use the direct output from the discriminator (`scores_real` and `scores_fake`).

Implement `ls_discriminator_loss`, `ls_generator_loss` in `cs231n/gan_pytorch.py`

Before running a GAN with our new loss function, let's check it:

- 원래 GAN에서 log 대신 1을 뺀 후 제곱하고 0.5를 곱하는 방식으로 만든 Least Square GAN을 구현할 것이다.



(a) (b)

Figure 2. (a): The sigmoid cross entropy loss function. (b): The least squares loss function.

- 원래의 GAN보다 결정 경계의 맞는 부분에 길게 놓여져 있는 샘플들을 잘 구별한다는 장점이 있다.
- 그로 인해 vanishing gradient 문제를 완화시킨다.

```
    loss_real = 0.5 * torch.mean((scores_real - 1) ** 2)
    loss_fake = 0.5 * torch.mean(scores_fake ** 2)
    loss = loss_real + loss_fake

    # ****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    return loss
```

```
loss = None
# ****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)****

loss = 0.5 * torch.mean((scores_fake - 1) ** 2)

# ****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)****
return loss
```

- discriminator와 generator의 loss를 각각 구현해주었다.

07_Deep Convolution GAN(DCGAN)

- 2014년에 처음 나온 기존의 GAN은 MNIST같은 단순한 이미지에는 잘 작동하였지만 CIFAR-10같이 조금만 이미지가 복잡해져서도 성능이 그닥 좋지 않았다.
- 그래서 FC와 pooling layer를 최대한 배제하고 Strided Convoultion과 Transpose Convolution(UPsampling)으로 네트워크 구조를 만들었다.
- 그 이유는 이미지의 위치 정보를 잃어 버리지 않게 하기 위함이다.

Discriminator

We will use a discriminator inspired by the TensorFlow MNIST classification tutorial, which is able to get above 99% accuracy on the MNIST dataset fairly quickly.

- Conv2D: 32 Filters, 5x5, Stride 1
- Leaky ReLU(alpha=0.01)
- Max Pool 2x2, Stride 2
- Conv2D: 64 Filters, 5x5, Stride 1
- Leaky ReLU(alpha=0.01)
- Max Pool 2x2, Stride 2
- Flatten
- Fully Connected with output size 4 x 4 x 64
- Leaky ReLU(alpha=0.01)
- Fully Connected with output size 1

- `conv2d-> Leaky ReLU -> MaxPooling -> Conv2D-> Leaky ReLu-> MaxPool -> Flatten -> FC -> Leaky ReLU -> FC` 구조로 Discriminator를 만들 것이다.

```
model = nn.Sequential(
  nn.Conv2d(1, 32, kernel_size = 5, stride = 1), # 24 * 24
  nn.LeakyReLU(0.01),
  nn.MaxPool2d(2, stride = 2), # 12 * 12
  nn.Conv2d(32, 64, kernel_size = 5, stride = 1), # 8 * 8
  nn.LeakyReLU(0.01),
  nn.MaxPool2d(2, stride = 2), # 4 * 4
  nn.Flatten(),
  nn.Linear(4 * 4 * 64, 4 * 4 * 64),
  nn.LeakyReLU(0.01),
  nn.Linear(4 * 4 * 64, 1)

)

return model
```

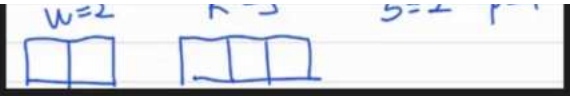
- 이미지를 28 * 28 → 4 * 4 로 압축시키고 channel을 64개로 늘린다.

Generator

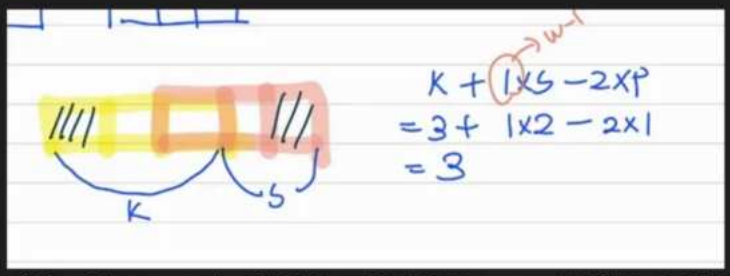
For the generator, we will copy the architecture exactly from the InfoGAN paper. See Appendix C.1 MNIST. See the documentation for nn.ConvTranspose2d. We are always "training" in GAN mode.

- Fully connected with output size 1024
- `ReLU`
- BatchNorm
- Fully connected with output size 7 x 7 x 128
- ReLU
- BatchNorm
- Use `Unflatten()` to reshape into Image Tensor of shape 7, 7, 128
- ConvTranspose2d: 64 filters of 4x4, stride 2, 'same' padding (use `padding=1`)
- `ReLU`
- BatchNorm
- ConvTranspose2d: 1 filter of 4x4, stride 2, 'same' padding (use `padding=1`)
- `TanH`
- Should have a 28x28x1 image, reshape back into 784 vector (using `Flatten()`)

- `FC->ReLU->BatchNorm->FC->ReLU->BatchNorm->Unflatten->ConvTranspose2d->ReLU->BatchNorm->ConvTranspose2d-> TanH-> Flatten`



- `input`의 크기 `w`를 2,
- `kernel size`를 3,
- `stride`의 크기를 2,
- `padding`의 크기를 1이라고 가정해보자

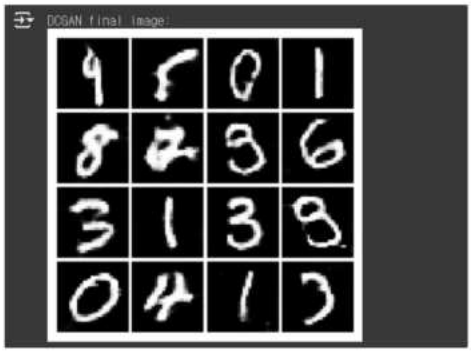


- 직접 그림으로 그려보면 위의 그림과 같은 `shape`을 얻을 수 있다.
- 처음의 `k`만큼의 `shape`이 생기고 그 다음 남은 `input`의 크기(`w-1`)만큼 `s`가 추가된다.
- 그 후 양쪽 끝에서 `p`만큼 사이즈가 줄어든다.
- 따라서 $k + (w-1) \times s - 2 \times p$라는 식을 세울 수 있고, 위의 그림의 값을 대입해보면 $3 + 1 \times 2 - 2 \times 1 = 3$이라는 결과를 얻을 수 있다.

- size = k + (w - 1) x s - 2 x p
- k: kernel_size, w: input_size, s: stride,
- p: padding → 여기서는 반대로 축소

```
model = nn.Sequential(
    nn.Linear(noise_dim, 1024),
    nn.ReLU(),
    nn.BatchNorm1d(1024),
    nn.Linear(1024, 7*7*128),
    nn.ReLU(),
    nn.BatchNorm1d(7*7*128),
    Unflatten(-1, 128, 7, 7),
    nn.ConvTranspose2d(128, 64, 4, 2, 1),
    nn.ReLU(),
    nn.BatchNorm2d(64),
    nn.ConvTranspose2d(64, 1, 4, 2, 1),
    nn.Tanh(),
    Flatten()
)

return model
```

- BatchNorm차원이 잘못되어서 힘들었다.
- 처음에는 2차원 입력이었기 때문에 `BatchNorm1d`를 이용해서 normalization을 진행하였다.
- 나중 Conv를 통과한 후에는 4차원이므로 `BatchNorm2d`를 이용해서 Channel수를 입력해주었다.



- 이미지가 이전 버전들에 비해 확실히 선명해지는 것을 확인할 수 있다.

1. Inline Question 1

Inline Question 4

We will look at an example to see why alternating minimization of the same objective (like in a GAN) can be tricky business.

Consider $f(x, y) = xy$. What does $\min_x \max_y f(x, y)$ evaluate to? (Hint: minmax tries to minimize the maximum value achievable.)

Now try to evaluate this function numerically for 6 steps, starting at the point $(1, 1)$, by using alternating gradient (first updating y, then updating x using that updated y) with step size 1. **Here step size is the learning_rate, and steps will be learning_rate * gradient.** You'll find that writing out the update step in terms of $x_t, y_t, x_{t+1}, y_{t+1}$ will be useful.

Breifly explain what $\min_x \max_y f(x, y)$ evaluates to and record the six pairs of explicit values for (x_t, y_t) in the table below.

Your answer:

y_0	y_1	y_2	y_3	y_4	y_5	y_6
1	2	1	-1	-2	-1	1
x_0	x_1	x_2	x_3	x_4	x_5	x_6
1	-1	-2	-1	1	2	1

- 위 과정을 반복하다 보면 처음 값으로 되돌아온다.
- y: gradient ascend, x: gradient descend

2. Inline Question 5

Inline Question 5

Using this method, will we ever reach the optimal value? Why or why not?

Your answer:

- 값이 계속 반복되기 때문에 최적의 값에 도달하지 못한다.

3. Inline Question 6

Inline Question 6

If the generator loss decreases during training while the discriminator loss stays at a constant high value from the start, is this a good sign? Why or why not? A qualitative answer is sufficient.

Your answer:

- generator loss가 감소하는데 discriminator가 계속 큰 값을 유지한다면 어떻게 될까?
- discriminator loss가 높다는 것은 이미지가 real인지 fake인지 판별을 못한다는 것이다.
- 그러면 generator가 overfit한 이미지를 만들어도 discriminator가 판단할 수 없다.
- 그렇다면 generator는 계속 이상한 이미지를 생성하게 될 것이다.