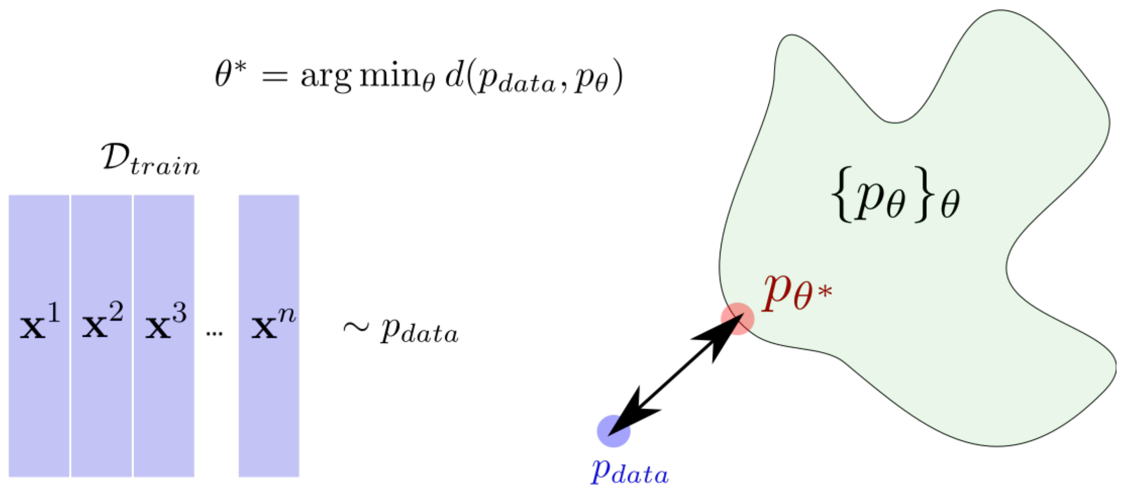
Generative Adversarial Networks

November 15, 2022



$\theta^* = \arg\min_\theta d(p_{data}, p_\theta)$
$\mathcal{D}_{train}$
$\mathbf{x}^1$ $\mathbf{x}^2$ $\mathbf{x}^3$ ... $\mathbf{x}^n$
$\sim p_{data}$
$\{p_\theta\}_\theta$
p_{θ^*}
p_{data}

We saw different ways of determining p_θ, with different design choices:

- Auto-regressive models,
- Variational autoencoders,
- Normalizing flows.

For all of them, determining θ is done, in some way or another, by **maximizing the likelihood** of the data from $\mathcal{D}_{train}$ over p_θ,

$$\theta^* = \arg\max_\theta \log L(\theta, \mathcal{D}_{train}).$$

Is maximizing the likelihood always the **best choice**? Is it really what we can/want?

What alternatives do we have?

There are some theoretical guarantees that exist for the **optimal models** under the ML criterion.

But the models that we train under a ML criterion are in general **not** the optimal ones, they are only locally maximal (non-convex optimization functions). For these sub-optimal models, a high log-likelihood value **does not necessarily mean that the samples will be of high quality (and vice-versa)**.

There may also have some **numerical considerations to handle**. Take the example of:

$$p_\theta(\mathbf{x}) = 0.001 p_{data}(\mathbf{x}) + 0.999 p_{noise}(\mathbf{x}).$$

It is clear that our samples will be of **very bad quality** (all generated through noise).

We can verify:

$$\log p_{data}(\mathbf{x}) - \log 1000 \leq \log p_\theta(\mathbf{x})$$

and deduce:

$$E_{p_{data}(\mathbf{x})}[\log p_{data}(\mathbf{x})] - \log 1000 \leq E_{p_{data}(\mathbf{x})}[\log p_\theta(\mathbf{x})] \leq E_{p_{data}(\mathbf{x})}[\log p_{data}(\mathbf{x})].$$

The previous set of inequalities show that when the **dimension** of $\mathbf{x}$ increases, since we will have

$$E_{p_{data}(\mathbf{x})}[\log p_{data}(\mathbf{x})] \to -\infty$$

(think in Gaussian distributions for example)

then it means that

$$E_{p_{data}(\mathbf{x})}[\log p_\theta(\mathbf{x})] \approx E_{p_{data}(\mathbf{x})}[\log p_{data}(\mathbf{x})].$$

This means that the **log-likelihoods will be numerically similar**!

But samples will be poor!

Another extreme example: Imagine a $p_\theta(\mathbf{x})$ **that fits exactly** $\mathcal{D}_{train}$ (memorizing the training set).

- Our samples will be of very good quality.
- The likelihoods on samples from p_{data} that are not on $\mathcal{D}_{train}$ will be zero.

Let us **change the focus** and forget the likelihood maximization to focus on the "similarity" between:

- the samples we have from $\mathcal{D}_{train}$
- samples we can produce from p_θ (assuming our design choices allow to do the sampling easily).

Given two set of samples $\mathcal{D}_1 = \{\mathbf{x} \sim p\}$ and $\mathcal{D}_2 = \{\mathbf{x} \sim q\}$, for two distributions p and q, a "two-sample test" evaluates the following hypotheses

1. Null hypothesis $H_0 : p = q$
2. Alternative hypothesis $H_1 : p \neq q$.

With some statistic (difference in mean,...) we compare $\mathcal{D}_1$ and $\mathcal{D}_2$ and based on the result, we can reject H_0 or say that H_0 is consistent with the observations.

Does not imply using likelihoods, **only samples**!

To implement the test above, we will use a **discriminator d**, i.e. a function of the data (that may be approximated by a neural network) in charge of **distinguishing** samples coming from p_{data} (the real samples from the dataset $\mathcal{D}_{train}$) and samples generated from our generative model p_θ.

The objective for training this test will be to **maximize** the test objective (to go in support of the hypothesis $p_{data} \neq p_\theta$).

In our case, we will get a way to **train the generator** by trying to minimize the output of this statistical test (i.e. we will try to become better at "tricking" the test).

Now, think that we will also need to go learning the test itself.

A good choice is to make $\mathbf{d}$ outputs probabilities (binary classifier). The training objective for the discriminator is the **binary cross-entropy**:

$$\max_{\mathbf{d}} V(p_\theta, \mathbf{d}) = E_{\mathbf{x} \sim p_{data}}[\log \mathbf{d}(\mathbf{x})] + E_{\mathbf{x} \sim p_\theta}[\log(1 - \mathbf{d}(\mathbf{x}))].$$

Ideally:

- Set probability 1 to points sampled from p_{data} (i.e., $\mathcal{D}_{train}$).
- Set probability 0 to points sampled from p_θ.

$$\begin{aligned} V(p_\theta, \mathbf{d}) = & \int_{\mathbf{x}} p_{data}(\mathbf{x}) \log \mathbf{d}(\mathbf{x}) d\mathbf{x} + \int_{\mathbf{x}} p_\theta(\mathbf{x}) \log(1 - \mathbf{d}(\mathbf{x})) d\mathbf{x}, & (1) \\ = & \int_{\mathbf{x}} (p_{data}(\mathbf{x}) \log \mathbf{d}(\mathbf{x}) + p_\theta(\mathbf{x}) \log(1 - \mathbf{d}(\mathbf{x}))) d\mathbf{x} & (2) \\ & & (3) \end{aligned}$$

At a given $\mathbf{x}$, we have a maximum when

$$p_{data}(\mathbf{x}) \frac{1}{\mathbf{d}(\mathbf{x})} - p_\theta(\mathbf{x}) \frac{1}{1 - \mathbf{d}(\mathbf{x})} = 0$$

i.e.

$$p_{data}(\mathbf{x})(1 - \mathbf{d}(\mathbf{x})) - p_\theta(\mathbf{x}) \mathbf{d}(\mathbf{x}) = 0$$

Hence, the **optimal discriminator**, given our current generator p_θ is

$$\mathbf{d}^*_{p_\theta}(\mathbf{x}) = \frac{p_{data}(\mathbf{x})}{p_{data}(\mathbf{x}) + p_\theta(\mathbf{x})}.$$

When trying to optimize both, the generator p_θ and the discriminator $\mathbf{d}$, we do it for both at the same time, in a two player **minimax game**:

- the **generator** tries to produce "fake" samples that should confuse the discriminator (**lower performance** in the binary classification test),
- the **discriminator** tries to recognize these "fake" samples (**higher performance** in the binary classification test).

The training loss function for the generator and disciminator:

$$\min_{p_\theta} \max_{\mathbf{d}} V(p_\theta, \mathbf{d}) = E_{x \sim pdata}[\log \mathbf{d}(\mathbf{x})] + E_{\mathbf{x} \sim p_\theta}[\log(1 - \mathbf{d}(\mathbf{x}))].$$

minimax is a classical setup for game theoretic problems.

For the optimal discriminator $\mathbf{d}^*_{p_\theta}(\mathbf{x})$, we have

$$\begin{aligned}
V(p_\theta, \mathbf{d}) = & \quad E_{x\sim pdata}[\log \frac{p_{data}(\mathbf{x})}{p_{data}(\mathbf{x}) + p_\theta(\mathbf{x})}] + E_{\mathbf{x}\sim p_\theta}[\log(1 - \frac{p_{data}(\mathbf{x})}{p_{data}(\mathbf{x}) + p_\theta(\mathbf{x})})], && (4)\\
= & \quad E_{x\sim pdata}[\log \frac{p_{data}(\mathbf{x})}{p_{data}(\mathbf{x}) + p_\theta(\mathbf{x})}] + E_{\mathbf{x}\sim p_\theta}[\log(\frac{p_\theta(\mathbf{x})}{p_{data}(\mathbf{x}) + p_\theta(\mathbf{x})})], && (5)\\
= & \quad E_{x\sim pdata}[\log \frac{p_{data}(\mathbf{x})}{\frac{1}{2}(p_{data}(\mathbf{x}) + p_\theta(\mathbf{x}))}] + E_{\mathbf{x}\sim p_\theta}[\log(\frac{p_\theta(\mathbf{x})}{\frac{1}{2}(p_{data}(\mathbf{x}) + p_\theta(\mathbf{x}))})] - \log 4, && (6)\\
= & \quad D_{KL}\left[p_{data}, \frac{1}{2}(p_{data} + p_\theta)\right] + D_{KL}\left[p_\theta, \frac{1}{2}(p_{data} + p_\theta)\right] - \log 4 && (7)\\
\triangleq & \quad 2D_{JSG}\left[p_{data}, p_\theta\right] - \log 4. && (8)
\end{aligned}$$

This is the **Jenson-Shanon divergence**:

$$D_{JSG}\left[p_{data}, p_\theta\right] \triangleq \frac{1}{2}(D_{KL}\left[p_{data}, \frac{1}{2}(p_{data} + p_\theta)\right] + D_{KL}\left[p_\theta, \frac{1}{2}(p_{data} + p_\theta)\right]).$$

It has some nicer properties than the KL divergnce:

- $D_{JSG}\left[p, q\right] \geq 0$.
- $D_{JSG}\left[p, q\right] = D_{JSG}\left[q, p\right]$.
- $D_{JSG}\left[p, q\right] = 0 \Leftrightarrow p = q$.
- Triangular inequality satisfied by $\sqrt{D_{JSG}\left[p, q\right]}$.

Properties of a **distance** between distributions.

Because of the anterior, the **optimal generator** p_θ, given the current optimal discriminator is

$$p_{data}.$$

The implementation of the generator with a latent variable $\mathbf{z}$ distributed under a fixed distribution $p_\mathbf{z}$

$$\begin{aligned}
\mathbf{z} \sim & \quad p_\mathbf{z}(\mathbf{z}) && (9)\\
\mathbf{x} = & \quad f_\theta(\mathbf{z}). && (10)
\end{aligned}$$

The discriminator is implemented with an another neural network, so that:

$$\min_\theta \max_\phi V(\mathbf{g}_\theta, \mathbf{d}_\phi) = E_{x\sim pdata}[\log \mathbf{d}_\phi(\mathbf{x})] + E_{\mathbf{z}\sim p_\mathbf{z}}[\log(1 - \mathbf{d}_\phi(\mathbf{g}_\theta(\mathbf{z}))].$$

https://imageio.forbes.com/blogs-images/williamfalcon/files/2018/10/edmond-de-belamy-framed-cropped.jpg?format=jpg&width=960

1. Sample batch of true training data $\mathbf{x}^1, \mathbf{x}^2, .., ., \mathbf{x}^s$ from $\mathcal{D}_{train}$.
2. Sample vectors $\mathbf{z}^1, \mathbf{z}^2, .., ., \mathbf{z}^s$ from $p_{\mathbf{z}}$.
3. Update the **discriminator parameters** ϕ by gradient **ascent**

$$\frac{1}{s}\nabla_\phi \sum_{k=1}^{s}\left[\log \mathbf{d}_\phi(\mathbf{x}^k) + \log(1 - \mathbf{d}_\phi(f_\theta(\mathbf{z}^k)))\right].$$

4. Update θ by gradient **descent**

$$-\frac{1}{s}\nabla_\theta \sum_{k=1}^{s}\left[\log(1 - \mathbf{d}_\phi(f_\theta(\mathbf{z}^k)))\right].$$

0.1 Example

[1]:
```python
import torch
import torch.nn as nn
import torchvision.transforms as transforms
import torch.optim as optim
import torchvision.datasets as datasets
import numpy as np
from torchvision.utils import make_grid
from torch.utils.data import DataLoader
from tqdm import tqdm
import matplotlib.pyplot as plt
```

[2]:
```python
batch_size  = 1024
epochs      = 100
sample_size = 16
latent_dim  = 128
leaky_slope = 0.2

# Access to CUDA
device     = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(device)
transform = transforms.Compose([transforms.ToTensor(),transforms.Normalize((0.
↪5,), (0.5,)),])
```

```
cuda
```

[3]:
```python
train_data = datasets.MNIST(
   root     ='data',
   train    =True,
   download =True,
   transform=transform
)
train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
```

[4]:
```
# Image generator
class Generator(nn.Module):

    def __init__(self, latent_dimension):
        super(Generator, self).__init__()
        self.latent_dimension = latent_dimension
        self.linear =  nn.Linear(in_features=self.latent_dimension,␣
 ↪out_features=128*7*7)
        self.deconv = nn.Sequential(
            nn.ConvTranspose2d(in_channels=128, out_channels=128,␣
 ↪kernel_size=(4, 4), stride=(2, 2), padding=(1, 1)),
            nn.LeakyReLU(leaky_slope),
            nn.ConvTranspose2d(in_channels=128, out_channels=128,␣
 ↪kernel_size=(4, 4), stride=(2, 2), padding=(1, 1)),
            nn.LeakyReLU(leaky_slope),
            nn.Conv2d(in_channels=128, out_channels=1, kernel_size=(3, 3),␣
 ↪padding=(1, 1)),
            # Generates numbers between -1 and 1
            nn.Tanh()
       )

    def forward(self, z):
        z = self.linear(z)
        z = nn.LeakyReLU(leaky_slope)(z)
        z = z.view(-1, 128, 7, 7)
        x = self.deconv(z)
        return x
```

[5]:
```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.convolutions = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=64, kernel_size=(4, 4),␣
 ↪stride=(2, 2), padding=(1, 1)),
            nn.LeakyReLU(leaky_slope),
            nn.Conv2d(in_channels=64, out_channels=64, kernel_size=(4, 4),␣
 ↪stride=(2, 2), padding=(1, 1)),
            nn.LeakyReLU(leaky_slope)
        )
        self.classifier = nn.Sequential(
            nn.Linear(in_features=3136, out_features=1),
            # Returns probability of being a real image
            nn.Sigmoid()
        )

    def forward(self, x):
        x = self.convolutions(x)
        x = x.view(x.size(0), -1)
        p = self.classifier(x)
        return p
```

[6]:
```
generator     = Generator(latent_dim).to(device)
discriminator = Discriminator().to(device)

optim_g = optim.Adam(generator.parameters(),     lr=0.0002, betas=(0.5, 0.999))
optim_d = optim.Adam(discriminator.parameters(), lr=0.0002, betas=(0.5, 0.999))

criterion = nn.BCELoss()

# Generate tensor of ones (real)
def label_real(size):
    labels = torch.ones(size, 1)
    return labels.to(device)

# Generate tensor of zeros (fake)
def label_fake(size):
    labels = torch.zeros(size, 1)
    return labels.to(device)

def sample_z(sample_size, latent_dimension):
    return torch.randn(sample_size, latent_dim).to(device)
```

[7]:
```
def generator_train_step(data_fake):
    batch_size  = data_fake.size(0)
    # Reference
    real_label  = label_real(batch_size)
    optim_g.zero_grad()
    # Evaluate loss
    loss        = criterion(discriminator(data_fake), real_label)
    loss.backward()
    optim_g.step()
    return loss
```

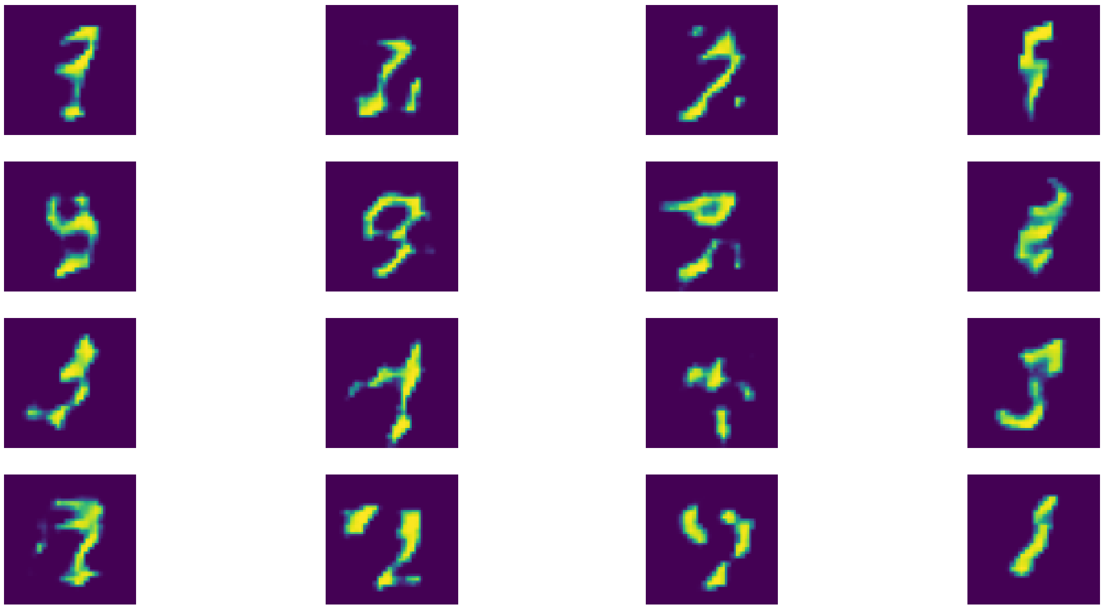
[8]:
```
def discriminator_train_step(data_real, data_fake):
    batch_size  = data_real.size(0)
    real_label  = label_real(batch_size)
    fake_label  = label_fake(batch_size)
    optim_d.zero_grad()
    # Evaluate losses
    loss_real   = criterion(discriminator(data_real), real_label)
    loss_fake   = criterion(discriminator(data_fake), fake_label)
    loss_real.backward()
    loss_fake.backward()
    optim_d.step()
    return loss_real, loss_fake
```

```
[9]: sampled_z = sample_z(sample_size, latent_dim)
generator.train()
discriminator.train()
plt.rcParams['figure.figsize'] = [10, 5]

for epoch in range(epochs):
    # Losses
    loss_g      = 0.0
    loss_d_real = 0.0
    loss_d_fake = 0.0

    # Scan batches
    for idx, data in tqdm(enumerate(train_loader), total=int(len(train_data) /␣
 ↪train_loader.batch_size)):
        data_real, _ = data
        data_real    = data_real.to(device)
        batch_size   = len(data_real)

        data_fake        = generator(sample_z(batch_size, latent_dim)).detach()
        loss_d_fake_real = discriminator_train_step(data_real, data_fake)
        loss_d_real     += loss_d_fake_real[0]
        loss_d_fake     += loss_d_fake_real[1]
        data_fake        = generator(sample_z(batch_size, latent_dim))
        loss_g          += generator_train_step(data_fake)

    if epoch%20==19:
        # Inference and observations
        generated_img = generator(sampled_z).cpu().detach()
        figure, axis = plt.subplots(4, 4)
        for i in range(16):
            axis[i//4, i%4].imshow(generated_img[i,0].numpy())
            axis[i//4, i%4].axis('off')

        #generated_img = make_grid(generated_img)
        #generated_img = np.moveaxis(generated_img.numpy(), 0, -1)
        plt.show()

    epoch_loss_g      = loss_g / idx
    epoch_loss_d_real = loss_d_real/idx
    epoch_loss_d_fake = loss_d_fake/idx

    print(f"Epoch {epoch} of {epochs}")
    print(f"Generator loss: {epoch_loss_g:.8f}, Discriminator loss fake:␣
 ↪{epoch_loss_d_fake:.8f}, Discriminator loss real: {epoch_loss_d_real:.8f}")
```

```
59it [00:24,  2.36it/s]

Epoch 0 of 100
Generator loss: 1.48855567, Discriminator loss fake: 0.41813803, Discriminator
loss real: 0.25311449

59it [00:25,  2.34it/s]

Epoch 1 of 100
Generator loss: 3.00320268, Discriminator loss fake: 0.25976944, Discriminator
loss real: 0.10784458

59it [00:25,  2.33it/s]

Epoch 2 of 100
Generator loss: 1.21443868, Discriminator loss fake: 0.48695087, Discriminator
loss real: 0.50437337

59it [00:25,  2.31it/s]

Epoch 3 of 100
Generator loss: 0.90925395, Discriminator loss fake: 0.64732164, Discriminator
loss real: 0.68965769

59it [00:25,  2.31it/s]

Epoch 4 of 100
Generator loss: 0.77056116, Discriminator loss fake: 0.70578206, Discriminator
loss real: 0.75214666

59it [00:25,  2.31it/s]

Epoch 5 of 100
Generator loss: 0.85527700, Discriminator loss fake: 0.69394225, Discriminator
loss real: 0.73092079

59it [00:26,  2.26it/s]

Epoch 6 of 100
Generator loss: 0.82547361, Discriminator loss fake: 0.66669869, Discriminator
loss real: 0.70544857

59it [00:25,  2.29it/s]

Epoch 7 of 100
Generator loss: 0.91261315, Discriminator loss fake: 0.63100016, Discriminator
loss real: 0.66071808

59it [00:26,  2.25it/s]

Epoch 8 of 100
Generator loss: 0.90869761, Discriminator loss fake: 0.59311241, Discriminator
loss real: 0.61927688

59it [00:26,  2.24it/s]
```

```
Epoch 9 of 100
Generator loss: 0.89870161, Discriminator loss fake: 0.61903781, Discriminator
loss real: 0.63989604

59it [00:26,  2.26it/s]

Epoch 10 of 100
Generator loss: 0.82695270, Discriminator loss fake: 0.65519595, Discriminator
loss real: 0.64827687

59it [00:25,  2.28it/s]

Epoch 11 of 100
Generator loss: 0.85617501, Discriminator loss fake: 0.62087679, Discriminator
loss real: 0.61180204

59it [00:25,  2.28it/s]

Epoch 12 of 100
Generator loss: 0.92566007, Discriminator loss fake: 0.61511242, Discriminator
loss real: 0.63241893

59it [00:26,  2.21it/s]

Epoch 13 of 100
Generator loss: 0.90189666, Discriminator loss fake: 0.58442706, Discriminator
loss real: 0.60637993

59it [00:26,  2.26it/s]

Epoch 14 of 100
Generator loss: 0.94292384, Discriminator loss fake: 0.57699972, Discriminator
loss real: 0.58954573

59it [00:26,  2.26it/s]

Epoch 15 of 100
Generator loss: 0.98959690, Discriminator loss fake: 0.57228386, Discriminator
loss real: 0.59039140

59it [00:25,  2.28it/s]

Epoch 16 of 100
Generator loss: 0.95899218, Discriminator loss fake: 0.56251723, Discriminator
loss real: 0.57839864

59it [00:25,  2.29it/s]

Epoch 17 of 100
Generator loss: 0.99635404, Discriminator loss fake: 0.55170798, Discriminator
loss real: 0.57072282

59it [00:25,  2.27it/s]

Epoch 18 of 100
Generator loss: 1.03365672, Discriminator loss fake: 0.54199213, Discriminator
loss real: 0.56153065
```

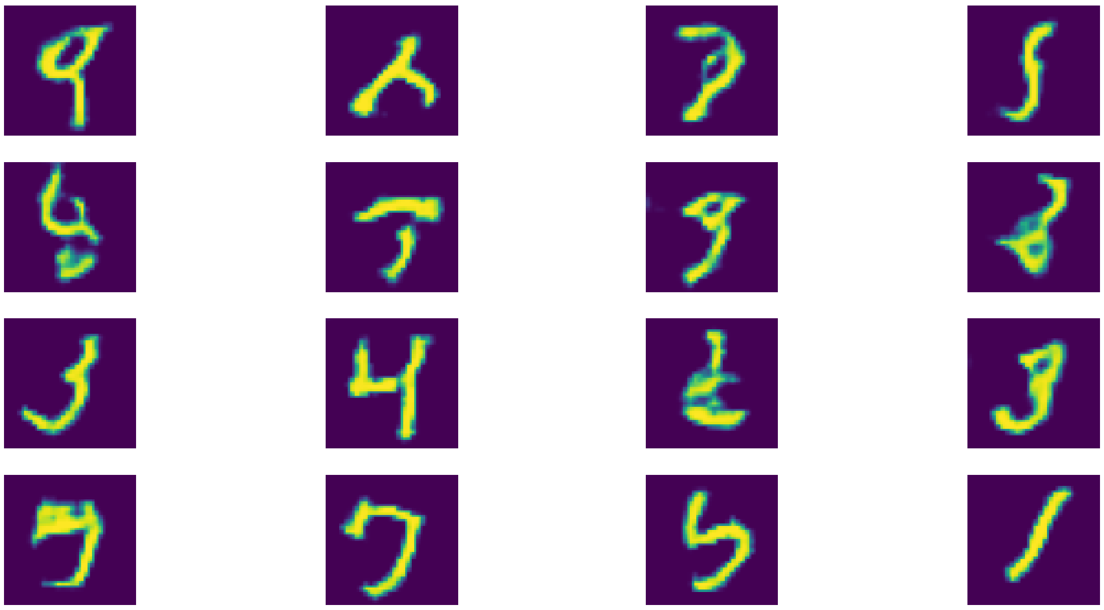
```
59it [00:26,  2.25it/s]
```



```
Epoch 19 of 100
Generator loss: 1.06702554, Discriminator loss fake: 0.54477382, Discriminator
loss real: 0.57521886

59it [00:26,  2.26it/s]

Epoch 20 of 100
Generator loss: 1.06489694, Discriminator loss fake: 0.55473602, Discriminator
loss real: 0.58375400

59it [00:25,  2.27it/s]

Epoch 21 of 100
Generator loss: 1.10099578, Discriminator loss fake: 0.57739604, Discriminator
loss real: 0.60372710

59it [00:25,  2.27it/s]

Epoch 22 of 100
Generator loss: 1.13731134, Discriminator loss fake: 0.60437924, Discriminator
loss real: 0.63204920

59it [00:26,  2.21it/s]

Epoch 23 of 100
Generator loss: 1.24187291, Discriminator loss fake: 0.59479094, Discriminator
loss real: 0.62565666

59it [00:27,  2.15it/s]
```

```
Epoch 24 of 100
Generator loss: 1.01654875, Discriminator loss fake: 0.58716083, Discriminator
loss real: 0.61409503

59it [00:27,  2.15it/s]

Epoch 25 of 100
Generator loss: 0.94117492, Discriminator loss fake: 0.59471232, Discriminator
loss real: 0.60600173

59it [00:26,  2.23it/s]

Epoch 26 of 100
Generator loss: 0.94569659, Discriminator loss fake: 0.60358489, Discriminator
loss real: 0.62232810

59it [00:27,  2.17it/s]

Epoch 27 of 100
Generator loss: 0.99801749, Discriminator loss fake: 0.62916547, Discriminator
loss real: 0.65558326

59it [00:26,  2.23it/s]

Epoch 28 of 100
Generator loss: 0.90929782, Discriminator loss fake: 0.62810659, Discriminator
loss real: 0.64472455

59it [00:26,  2.25it/s]

Epoch 29 of 100
Generator loss: 0.90167373, Discriminator loss fake: 0.63503152, Discriminator
loss real: 0.65309924

59it [00:26,  2.24it/s]

Epoch 30 of 100
Generator loss: 0.90722734, Discriminator loss fake: 0.64211327, Discriminator
loss real: 0.66250563

59it [00:27,  2.15it/s]

Epoch 31 of 100
Generator loss: 0.87792903, Discriminator loss fake: 0.64034361, Discriminator
loss real: 0.66016287

59it [00:27,  2.15it/s]

Epoch 32 of 100
Generator loss: 0.88755786, Discriminator loss fake: 0.64312953, Discriminator
loss real: 0.66100031

59it [00:27,  2.17it/s]

Epoch 33 of 100
Generator loss: 0.92387271, Discriminator loss fake: 0.64302242, Discriminator
loss real: 0.66496915
```

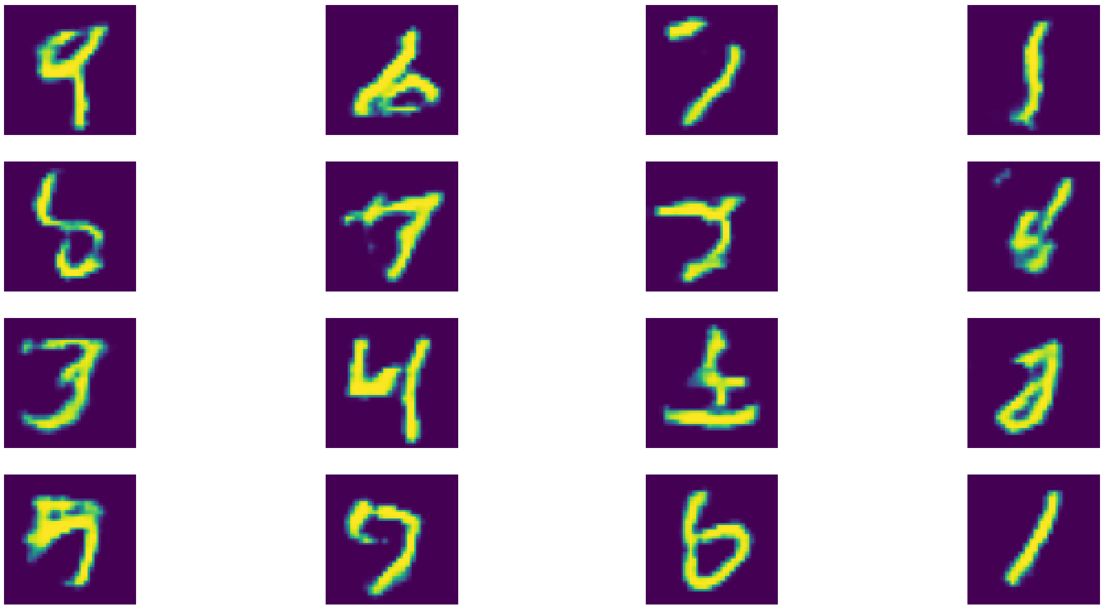
```
59it [00:26,  2.24it/s]

Epoch 34 of 100
Generator loss: 0.88455623, Discriminator loss fake: 0.64401692, Discriminator
loss real: 0.66262645

59it [00:26,  2.25it/s]

Epoch 35 of 100
Generator loss: 0.87649548, Discriminator loss fake: 0.64803207, Discriminator
loss real: 0.66030264

59it [00:26,  2.25it/s]

Epoch 36 of 100
Generator loss: 0.88821006, Discriminator loss fake: 0.64931613, Discriminator
loss real: 0.66522163

59it [00:26,  2.22it/s]

Epoch 37 of 100
Generator loss: 0.89511746, Discriminator loss fake: 0.64961314, Discriminator
loss real: 0.66691077

59it [00:26,  2.24it/s]

Epoch 38 of 100
Generator loss: 0.89819777, Discriminator loss fake: 0.64843017, Discriminator
loss real: 0.66682649

59it [00:26,  2.20it/s]
```



```
Epoch 39 of 100
Generator loss: 0.89741957, Discriminator loss fake: 0.64951509, Discriminator
loss real: 0.66399902

59it [00:26,  2.20it/s]

Epoch 40 of 100
Generator loss: 0.87018627, Discriminator loss fake: 0.65798533, Discriminator
loss real: 0.67370081

59it [00:26,  2.22it/s]

Epoch 41 of 100
Generator loss: 0.87310803, Discriminator loss fake: 0.65199167, Discriminator
loss real: 0.66286701

59it [00:26,  2.21it/s]

Epoch 42 of 100
Generator loss: 0.86788452, Discriminator loss fake: 0.65580016, Discriminator
loss real: 0.67008835

59it [00:26,  2.22it/s]

Epoch 43 of 100
Generator loss: 0.86783564, Discriminator loss fake: 0.64919645, Discriminator
loss real: 0.66426730

59it [00:26,  2.24it/s]

Epoch 44 of 100
Generator loss: 0.85583878, Discriminator loss fake: 0.65869409, Discriminator
loss real: 0.66898310

59it [00:26,  2.24it/s]

Epoch 45 of 100
Generator loss: 0.85273802, Discriminator loss fake: 0.65415132, Discriminator
loss real: 0.66243827

59it [00:26,  2.21it/s]

Epoch 46 of 100
Generator loss: 0.85275787, Discriminator loss fake: 0.65107834, Discriminator
loss real: 0.66750157

59it [00:26,  2.24it/s]

Epoch 47 of 100
Generator loss: 0.84213704, Discriminator loss fake: 0.66148585, Discriminator
loss real: 0.66773790

59it [00:26,  2.25it/s]

Epoch 48 of 100
Generator loss: 0.84906697, Discriminator loss fake: 0.65328860, Discriminator
loss real: 0.65852678
```

```
59it [00:26,  2.24it/s]

Epoch 49 of 100
Generator loss: 0.83550918, Discriminator loss fake: 0.65857309, Discriminator
loss real: 0.66883105

59it [00:27,  2.16it/s]

Epoch 50 of 100
Generator loss: 0.83616561, Discriminator loss fake: 0.65370631, Discriminator
loss real: 0.66262954

59it [00:27,  2.12it/s]

Epoch 51 of 100
Generator loss: 0.83979076, Discriminator loss fake: 0.65918845, Discriminator
loss real: 0.66918540

59it [00:27,  2.16it/s]

Epoch 52 of 100
Generator loss: 0.85085046, Discriminator loss fake: 0.65695703, Discriminator
loss real: 0.66088122

59it [00:27,  2.18it/s]

Epoch 53 of 100
Generator loss: 0.83700520, Discriminator loss fake: 0.65663421, Discriminator
loss real: 0.66535312

59it [00:26,  2.22it/s]

Epoch 54 of 100
Generator loss: 0.84636968, Discriminator loss fake: 0.65041995, Discriminator
loss real: 0.66156030

59it [00:26,  2.23it/s]

Epoch 55 of 100
Generator loss: 0.83377004, Discriminator loss fake: 0.65609211, Discriminator
loss real: 0.66378236

59it [00:26,  2.22it/s]

Epoch 56 of 100
Generator loss: 0.83880132, Discriminator loss fake: 0.65352374, Discriminator
loss real: 0.66202974

59it [00:26,  2.24it/s]

Epoch 57 of 100
Generator loss: 0.84682989, Discriminator loss fake: 0.65501010, Discriminator
loss real: 0.66108346

59it [00:26,  2.23it/s]
```

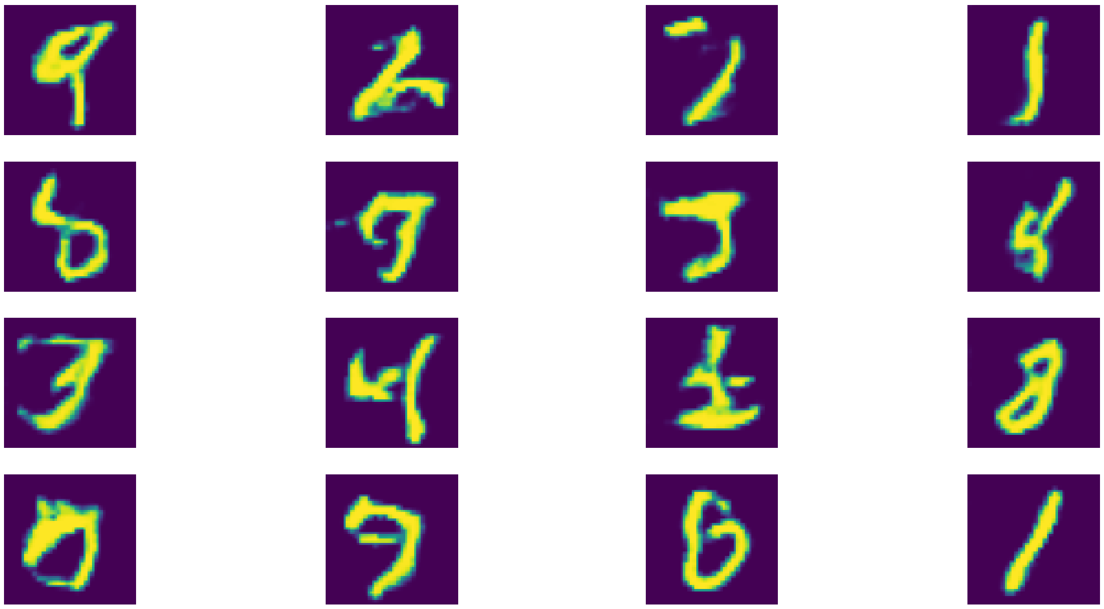
```
Epoch 58 of 100
Generator loss: 0.83634561, Discriminator loss fake: 0.65600991, Discriminator
loss real: 0.66554689

59it [00:27,  2.17it/s]
```



```
Epoch 59 of 100
Generator loss: 0.84001446, Discriminator loss fake: 0.64875656, Discriminator
loss real: 0.65773004

59it [00:26,  2.25it/s]

Epoch 60 of 100
Generator loss: 0.83675855, Discriminator loss fake: 0.65024382, Discriminator
loss real: 0.65982002

59it [00:26,  2.20it/s]

Epoch 61 of 100
Generator loss: 0.84314626, Discriminator loss fake: 0.65568286, Discriminator
loss real: 0.66411364

59it [00:26,  2.22it/s]

Epoch 62 of 100
Generator loss: 0.85397077, Discriminator loss fake: 0.65170252, Discriminator
loss real: 0.65993428

59it [00:26,  2.20it/s]

Epoch 63 of 100
Generator loss: 0.86455959, Discriminator loss fake: 0.65153235, Discriminator
```

```
loss real: 0.66197270

59it [00:26,  2.24it/s]

Epoch 64 of 100
Generator loss: 0.84867841, Discriminator loss fake: 0.65345323, Discriminator
loss real: 0.66393673

59it [00:27,  2.17it/s]

Epoch 65 of 100
Generator loss: 0.85033321, Discriminator loss fake: 0.64768386, Discriminator
loss real: 0.65889972

59it [00:26,  2.23it/s]

Epoch 66 of 100
Generator loss: 0.85686880, Discriminator loss fake: 0.65223753, Discriminator
loss real: 0.66459638

59it [00:26,  2.22it/s]

Epoch 67 of 100
Generator loss: 0.85042393, Discriminator loss fake: 0.65134001, Discriminator
loss real: 0.66034555

59it [00:26,  2.20it/s]

Epoch 68 of 100
Generator loss: 0.85852665, Discriminator loss fake: 0.64759743, Discriminator
loss real: 0.65551269

59it [00:26,  2.20it/s]

Epoch 69 of 100
Generator loss: 0.85148042, Discriminator loss fake: 0.65260059, Discriminator
loss real: 0.66602033

59it [00:26,  2.25it/s]

Epoch 70 of 100
Generator loss: 0.86540210, Discriminator loss fake: 0.64950311, Discriminator
loss real: 0.65837228

59it [00:25,  2.28it/s]

Epoch 71 of 100
Generator loss: 0.85400480, Discriminator loss fake: 0.64965183, Discriminator
loss real: 0.66146922

59it [00:26,  2.23it/s]

Epoch 72 of 100
Generator loss: 0.85517883, Discriminator loss fake: 0.65301657, Discriminator
loss real: 0.66212398

59it [00:26,  2.21it/s]
```

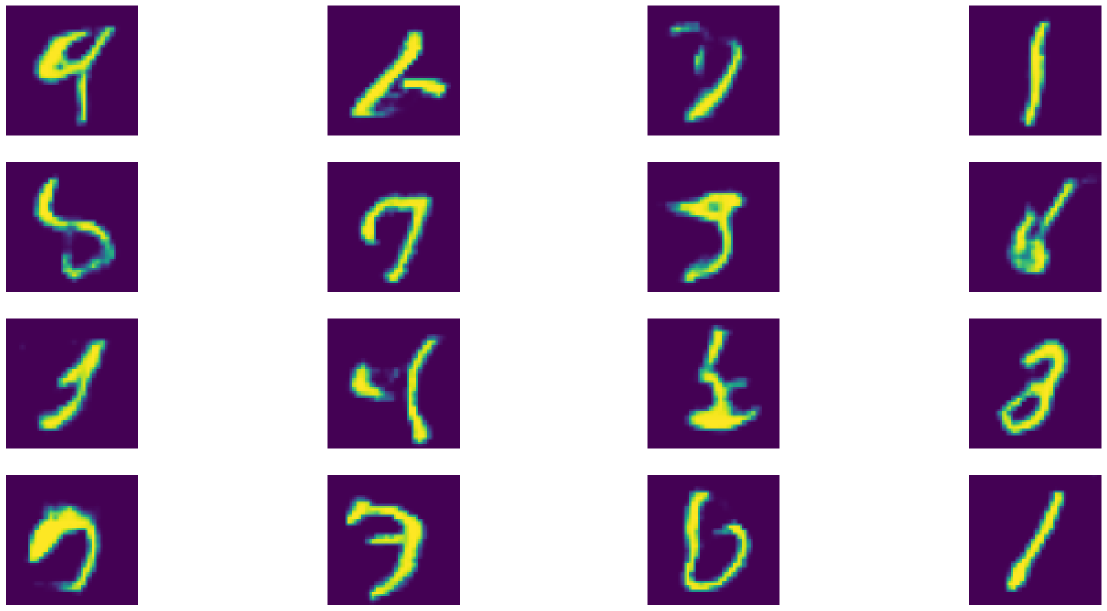
```
Epoch 73 of 100
Generator loss: 0.85226738, Discriminator loss fake: 0.64861631, Discriminator
loss real: 0.65772349

59it [00:26,  2.24it/s]

Epoch 74 of 100
Generator loss: 0.84408164, Discriminator loss fake: 0.65061122, Discriminator
loss real: 0.65933776

59it [00:25,  2.28it/s]

Epoch 75 of 100
Generator loss: 0.85043615, Discriminator loss fake: 0.64947045, Discriminator
loss real: 0.66103750

59it [00:26,  2.22it/s]

Epoch 76 of 100
Generator loss: 0.85960603, Discriminator loss fake: 0.64811832, Discriminator
loss real: 0.65743202

59it [00:27,  2.17it/s]

Epoch 77 of 100
Generator loss: 0.86901021, Discriminator loss fake: 0.65163910, Discriminator
loss real: 0.66541433

59it [00:26,  2.20it/s]

Epoch 78 of 100
Generator loss: 0.87529564, Discriminator loss fake: 0.65059996, Discriminator
loss real: 0.66473347

59it [00:26,  2.19it/s]
```



```
Epoch 79 of 100
Generator loss: 0.86338490, Discriminator loss fake: 0.65136397, Discriminator
loss real: 0.66276079

59it [00:26,  2.20it/s]

Epoch 80 of 100
Generator loss: 0.86182678, Discriminator loss fake: 0.64446807, Discriminator
loss real: 0.66311771

59it [00:26,  2.22it/s]

Epoch 81 of 100
Generator loss: 0.85046113, Discriminator loss fake: 0.64971209, Discriminator
loss real: 0.65827203

59it [00:26,  2.20it/s]

Epoch 82 of 100
Generator loss: 0.84378874, Discriminator loss fake: 0.65228963, Discriminator
loss real: 0.66067451

59it [00:26,  2.20it/s]

Epoch 83 of 100
Generator loss: 0.85306752, Discriminator loss fake: 0.64469820, Discriminator
loss real: 0.65820462

59it [00:26,  2.19it/s]

Epoch 84 of 100
Generator loss: 0.84666377, Discriminator loss fake: 0.65419084, Discriminator
loss real: 0.66291863

59it [00:26,  2.22it/s]

Epoch 85 of 100
Generator loss: 0.85473073, Discriminator loss fake: 0.64969325, Discriminator
loss real: 0.65826255

59it [00:27,  2.15it/s]

Epoch 86 of 100
Generator loss: 0.84949780, Discriminator loss fake: 0.65400249, Discriminator
loss real: 0.66667438

59it [00:27,  2.16it/s]

Epoch 87 of 100
Generator loss: 0.84920806, Discriminator loss fake: 0.64879107, Discriminator
loss real: 0.65945965

59it [00:26,  2.19it/s]
```

```
Epoch 88 of 100
Generator loss: 0.84456056, Discriminator loss fake: 0.65522224, Discriminator
loss real: 0.66111213

59it [00:27,  2.16it/s]

Epoch 89 of 100
Generator loss: 0.83727419, Discriminator loss fake: 0.65306348, Discriminator
loss real: 0.65878242

59it [00:28,  2.06it/s]

Epoch 90 of 100
Generator loss: 0.83071518, Discriminator loss fake: 0.65330988, Discriminator
loss real: 0.66193187

59it [00:27,  2.14it/s]

Epoch 91 of 100
Generator loss: 0.84186602, Discriminator loss fake: 0.65568310, Discriminator
loss real: 0.66267425

59it [00:27,  2.18it/s]

Epoch 92 of 100
Generator loss: 0.85008043, Discriminator loss fake: 0.65737182, Discriminator
loss real: 0.67109931

59it [00:27,  2.18it/s]

Epoch 93 of 100
Generator loss: 0.84976059, Discriminator loss fake: 0.65488219, Discriminator
loss real: 0.66596627

59it [00:26,  2.21it/s]

Epoch 94 of 100
Generator loss: 0.84561694, Discriminator loss fake: 0.65733981, Discriminator
loss real: 0.66496384

59it [00:26,  2.23it/s]

Epoch 95 of 100
Generator loss: 0.84540254, Discriminator loss fake: 0.66184044, Discriminator
loss real: 0.67253631

59it [00:27,  2.14it/s]

Epoch 96 of 100
Generator loss: 0.83633363, Discriminator loss fake: 0.65725356, Discriminator
loss real: 0.66573441

59it [00:26,  2.22it/s]

Epoch 97 of 100
Generator loss: 0.81687796, Discriminator loss fake: 0.65935850, Discriminator
loss real: 0.66459340
```

```
59it [00:26,  2.24it/s]

Epoch 98 of 100
Generator loss: 0.82273877, Discriminator loss fake: 0.66169125, Discriminator
loss real: 0.66714042

59it [00:26,  2.25it/s]
```



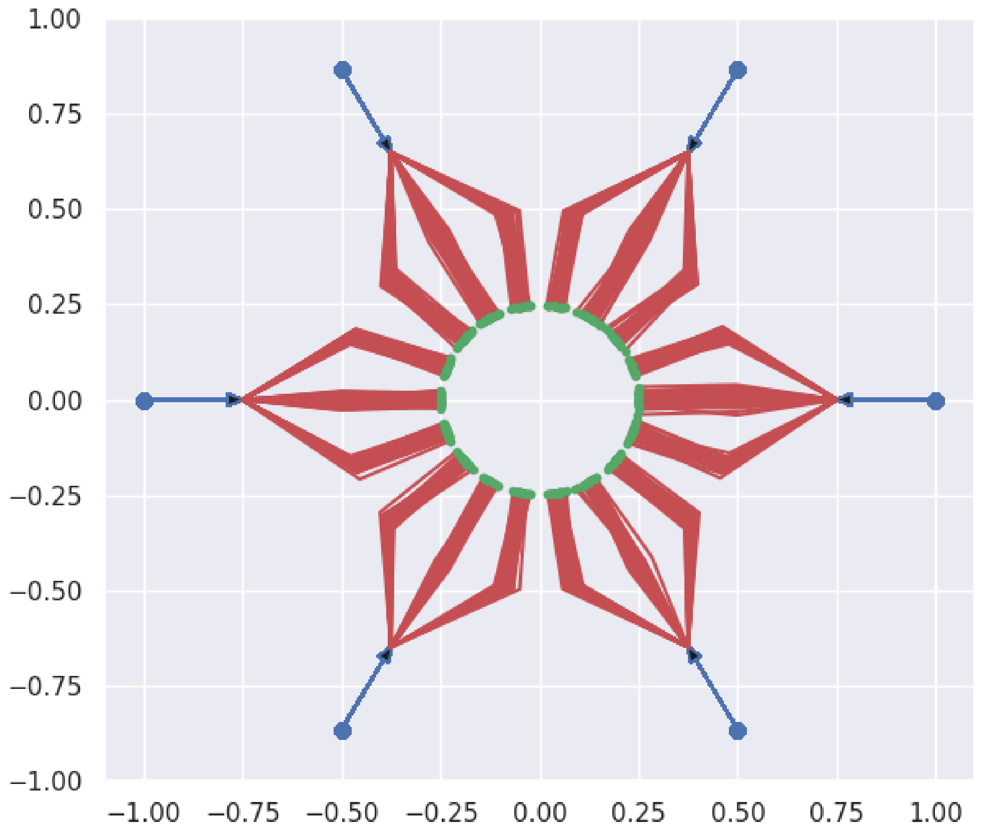
```
Epoch 99 of 100
Generator loss: 0.82344109, Discriminator loss fake: 0.66266972, Discriminator
loss real: 0.67016757
```

0.1.1 Difficulties in the optimization process

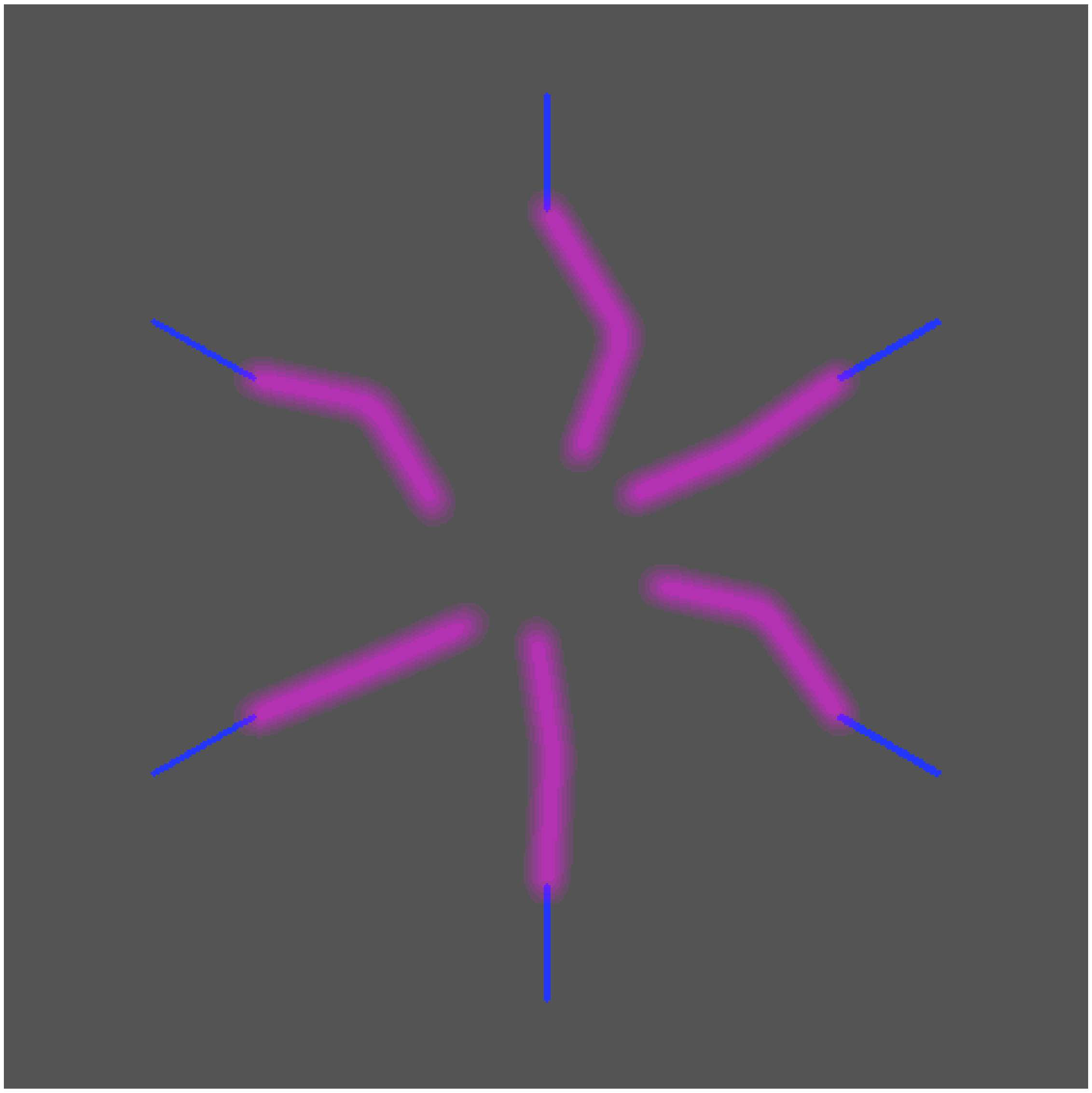
The optimization process is **more difficult** than with more traditional networks

- Not obvious to decide when to **stop**!
- In theory, if the discriminator is optimal after each step, we should converge. But this is not the case, and **oscillations** for both the discriminator and generator.

A very common problem with GANs is known as **mode collapse**.



Typical results:



Social Ways: Learning Multi-Modal Distributions of Pedestrian Trajectories with GANs Javad Amirian, Jean-Bernard Hayet, Julien Pettre. CVPR 2019 Precognition Workshop

- **KL divergence**: we saw it in autoregressive Models, normalizing flow models. And it translates into maximizing
- **Jenson-Shannon divergence**: this is what the "vanilla" GAN uses (assuming a perfect discriminator)

$$2D_{JSG}\left[p_{data}, p_\theta\right] - \log 4.$$

In fact,

$$D_{JSG}\left[p_{data}, p_\theta\right] \triangleq \frac{1}{2}(D_{KL}\left[p_{data}, \frac{1}{2}(p_{data}+p_\theta)\right] + D_{KL}\left[p_\theta, \frac{1}{2}(p_{data}+p_\theta)\right]),$$

is one particular case of a divergence called $f-$**divergence**

$$D_f\,[p,q] \triangleq E_q[f(\frac{p}{q})],$$

with $f(x) = x\log x - (x+1)\log(\frac{1}{2}(x+1))$.

A problem with these divergences is that the support of q needs to **cover the support of** p. Otherwise you have discontinuities.

Consider the example of

$$p_{data}(\mathbf{x}) = \begin{cases} \infty & \text{if } \mathbf{x} = 0 \\ 0 & \text{otherwise,} \end{cases}$$

and

$$p_\theta(\mathbf{x}) = \begin{cases} \infty & \text{if } \mathbf{x} = \theta \\ 0 & \text{otherwise.} \end{cases}$$

$$D_{KL}(p_{data}, p_\theta) = \int_{\mathbf{x}} p_{data}(\mathbf{x})\log(\frac{p_{data}(\mathbf{x})}{p_{data}(\mathbf{x})})d\mathbf{x} = \begin{cases} 0 & \text{if } \theta = 0 \\ \infty & \text{otherwise,} \end{cases}$$

and

$$D_{JS}(p_{data}, p_\theta) = \begin{cases} 0 & \text{if } \theta = 0 \\ \log 2 & \text{otherwise,} \end{cases}$$

We may have discontinuities that are not good for the optimization process.

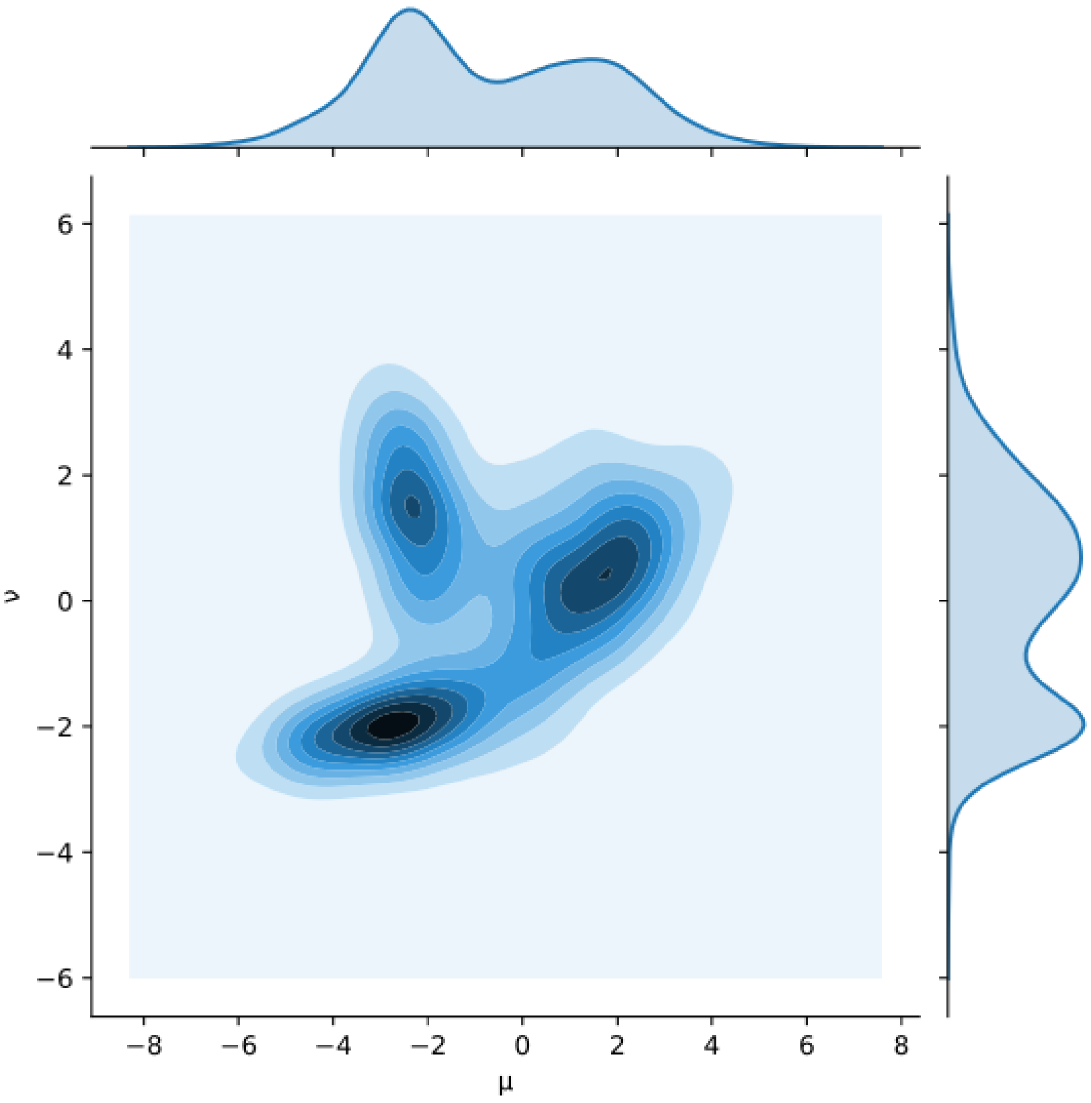
0.2 Wasserstein GANs

Let us introduce the **Wasserstein distance**:

$$D_W(p,q) = \inf_{\gamma \in \Pi(p,q)} E_{(\mathbf{x},\mathbf{y})\sim\gamma}\|\mathbf{x}-\mathbf{y}\|_1,$$

where $\Pi(p,q)$ is the set of all **joint distributions of (x, y)** such that the marginal of $\mathbf{x}$ is $p(\mathbf{x})$, and the marginal of $\mathbf{y}$ is $q(\mathbf{y})$.

A pair of p and q and a possible joint distribution in $\Pi(p,q)$.

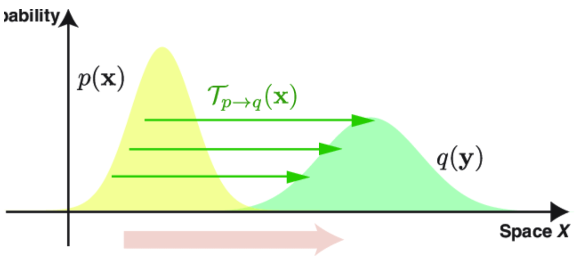


By Lambdabadger - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=64872543

Interpretation:

$$E_{(\mathbf{x},\mathbf{y})\sim\gamma}\|\mathbf{x}-\mathbf{y}\|_1$$

represents the average displacement ("transport")/effort that you need to move some probability mass from $\mathbf{x}$ to $\mathbf{y}$, weighted by $\gamma(\mathbf{x},\mathbf{y})$. There are many, many possible γ!



Let us retak the example above:

$$p(\mathbf{x}) = \begin{cases} \infty & \text{if } \mathbf{x} = 0 \\ 0 & \text{otherwise,} \end{cases}$$

and

$$q_\theta(\mathbf{x}) = \begin{cases} \infty & \text{if } \mathbf{x} = \theta \\ 0 & \text{otherwise.} \end{cases}$$

In that case the **only possible transport γ is from zero to θ**, so

$$D_W(p, q) = E_{(\mathbf{x},\mathbf{y})\sim\gamma} \|\mathbf{x} - \mathbf{y}\|_1 = |\theta - 0| = |\theta|.$$

Note how it is **continuous in θ**!

Now one big problem:

$$D_W(p, q) = \inf_{\gamma \in \Pi(p,q)} E_{(\mathbf{x},\mathbf{y})\sim\gamma} \|\mathbf{x} - \mathbf{y}\|_1,$$

is **intractable in general** (think in the search space, it is huge!).

Wasserstein Generative Adversarial Networks. International Conference on Machine Learning. PMLR: 214–223 Arjovsky, Martin; Chintala, Soumith; Bottou, Léon.

Kantorovich-Rubinstein duality:

$$D_W(p, q) = \sup_{\|g\|_L \leq 1} E_{\mathbf{x}\sim p}[g(\mathbf{x})] - E_{\mathbf{x}\sim q}[g(\mathbf{x})]$$

where $\|g\|_L \leq 1$ means that the Lipschitz constant of $g(\mathbf{x})$ is 1, i.e.

$$\forall \mathbf{x}, \mathbf{y} : |g(\mathbf{x}) - g(\mathbf{y})| \leq \|\mathbf{x} - \mathbf{y}\|_1.$$

The authors first emphasize that by replacing $\|g\|_L \leq 1$ by $\|g\|_L \leq K$ (for some constant K), then evaluating the same supremum leads to $KD_W(p, q)$.

Then, the idea is to train a neural network **parameterized with weights** ϕ lying in some **compact space** $\mathcal{W}$ and determine through this network a parameterized function g_ϕ to solve:

$$D_W(p, q) = \sup_{\phi \in \mathcal{W}} E_{\mathbf{x}\sim p}[g_\phi(\mathbf{x})] - E_{\mathbf{x}\sim q}[g_\phi(\mathbf{x})].$$

Note that $\mathcal{W}$ being compact implies that all the functions g_ϕ will be K-Lipschitz **for some** K that only depends on $\mathcal{W}$. So, by forcing ϕ to live in a compact space, we will implicitly evaluate a multiple of the D_W.

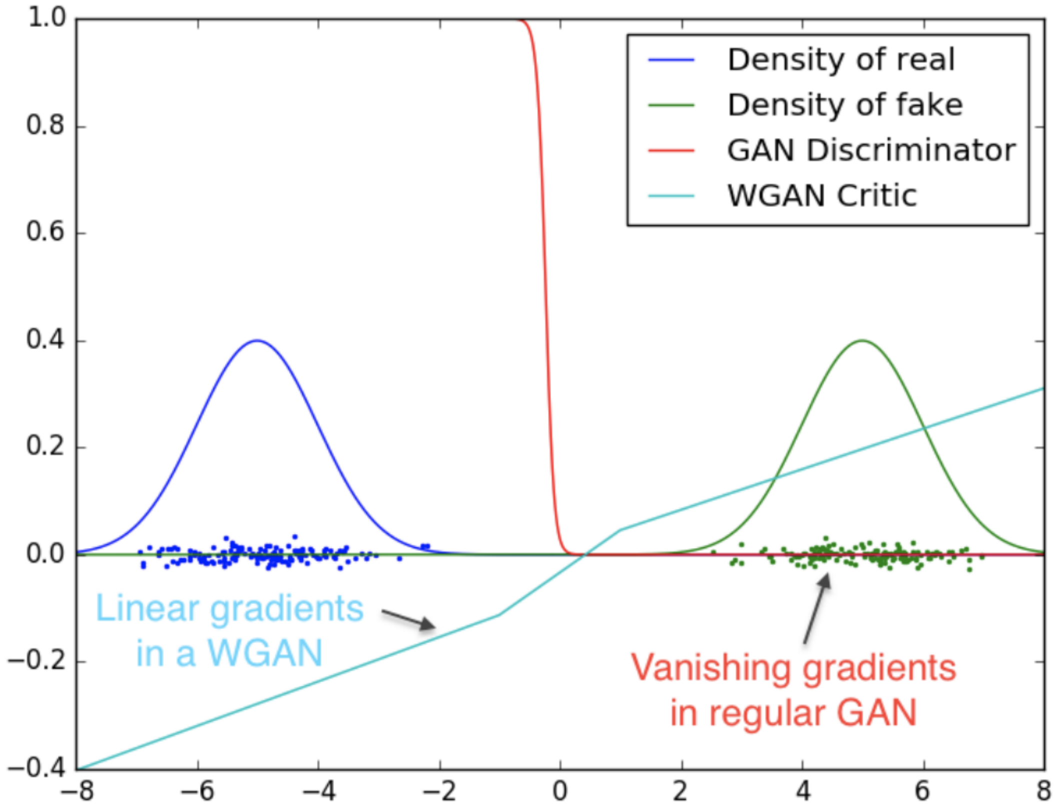
Note that since the goal is to minimize the D_W, we find the same min-max expression:

$$\min_\theta \max_\phi E_{\mathbf{x}\sim p}[g_\phi(\mathbf{x})] - E_{\mathbf{z}\sim p_\mathbf{z}}[g_\phi(f_\theta(\mathbf{z}))].$$

The interpretation of g_ϕ is different from the one of discriminator. Called **critic**.

To have parameters ϕ in a compact space, they propose to **clip the weights to a fixed box**, for example $[-0.01, 0.01]$ after each gradient update.

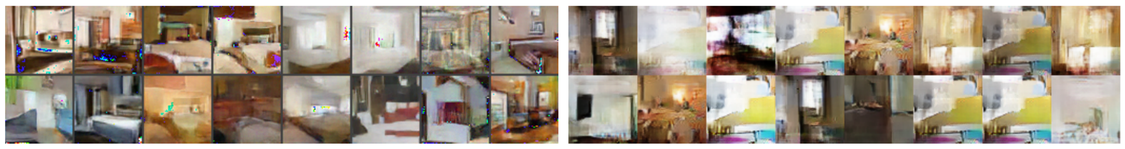
The algorithm is very similar to the original GAN, except that the add the clipping part of the parameters ϕ.



Comparison: Gan discriminator vs. WGan Critic:

- The discriminator learns **quickly to distinguish between** p_{data} **and** p_θ, and as a result cannot evolve more (gradients vanish everywhere).
- The Wgan critic converges to a **linear function with "clean gradients"**.

The **training** is more stable, and one has less **mode collapse**.



0.3 Access to the latent space

This is not trivial with GANs:

- Unlike in a **normalizing flow model**, we do not have the inverse of the direct mapping from $\mathbf{z}$ to $\mathbf{x}$.
- Unlike a **VAE**, no representation $q_\phi(\mathbf{z}|\mathbf{x})$ over the latent variables.

One option: Use the activations of the final layers of $\mathbf{d}$ as a **feature representation space**: After all, this is the feature that is used to perform the discrimination. So we can interpret it as a latent variable.

But, this is **not** the latent variable that is used by the generator.

Another option: instead of passing only $\mathbf{x}$ to the discriminator, pass both $\mathbf{x}$ and $\mathbf{z}$!

- With a fake $\mathbf{x}$, we will know the $\mathbf{z}$ that will have generated $\mathbf{x}$ through the generator.
- With a real $\mathbf{x}$, we do not observe $\mathbf{z}$, so how to get access to it? By training a "recognition" neetwork as in VAE.

Adversarial Feature Learning. Jeff Donahue, Philipp Krähenbühl, Trevor Darrell. ICLR (Poster) 2017.

BiGAN: in this system the authors add an **encoder network** E acting in parallel to the generator network f_θ.

The encoder network observes $\mathbf{x} \sim p_{data}(\mathbf{x})$ during training to learn a **mapping**

$$E : \mathbf{x} \to \mathbf{z}.$$

The rest is the same: the generator samples taking $\mathbf{z}$ from $p_\mathbf{z}$ to generate $\mathbf{x}$. But the discriminator now try to discriminate between:

- $f_\theta(\mathbf{z}), \mathbf{z}$
- $\mathbf{x}, E(\mathbf{x})$

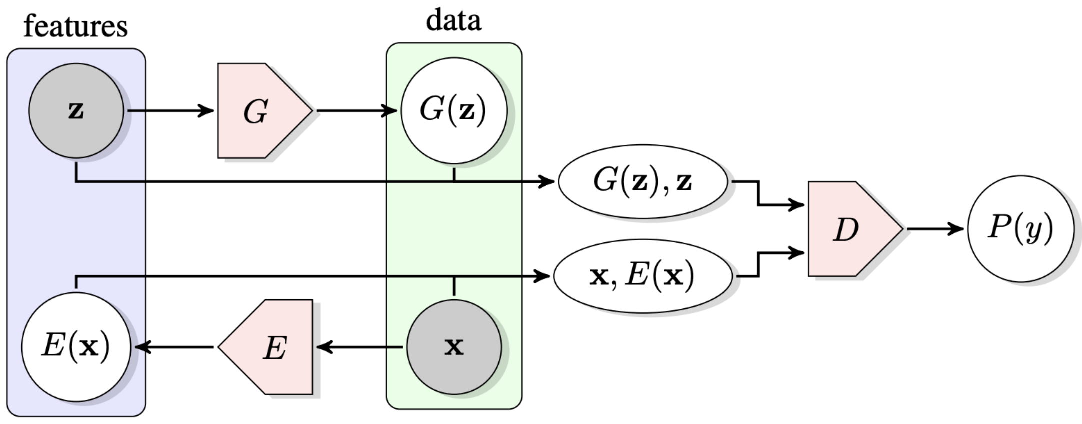
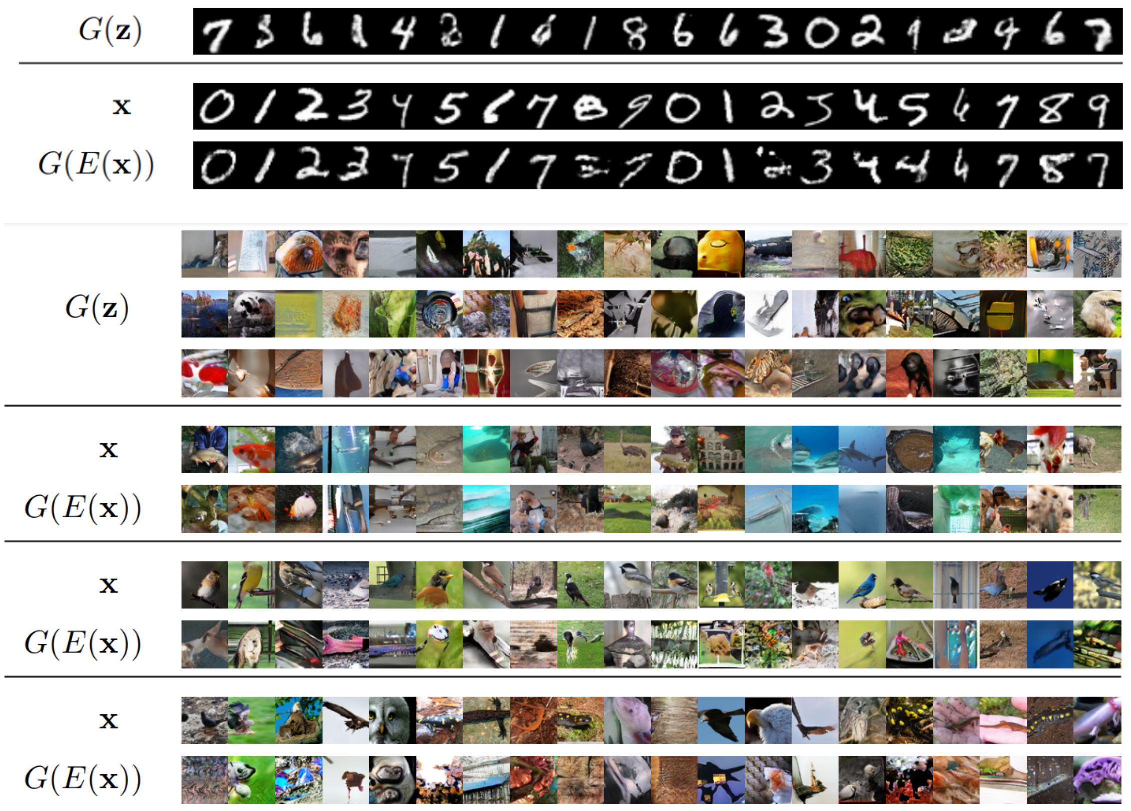


Figure 1: The structure of Bidirectional Generative Adversarial Networks (BiGAN).


$G(\mathbf{z})$
$\mathbf{x}$
$G(E(\mathbf{x}))$
$G(\mathbf{z})$
$\mathbf{x}$
$G(E(\mathbf{x}))$
$\mathbf{x}$
$G(E(\mathbf{x}))$
$\mathbf{x}$
$G(E(\mathbf{x}))$