Generative Adversarial Networks Zoo

with mathematics deduction

Yuang (Dennis) Guo

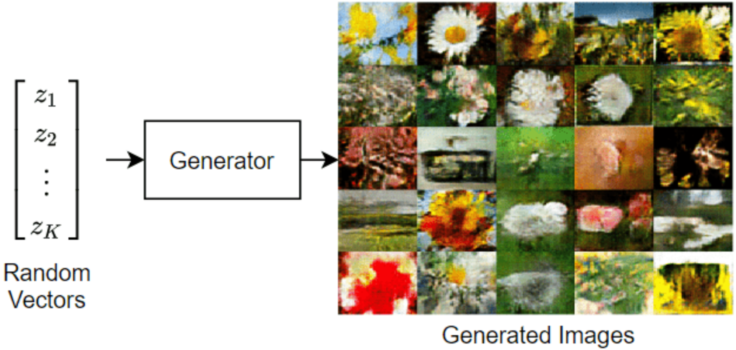
Boston College

May 14th

What is GAN?

Generative Adversarial Networks (GANs) are a class of artificial intelligence algorithms used in **unsupervised** machine learning, implemented by a system of two neural networks contesting with each other in a zero-sum game framework.

Example



Architecture of GANs

- **Generator:** Creates new data instances.
- **Discriminator:** Evaluates them for authenticity; accepts or rejects the generator output.

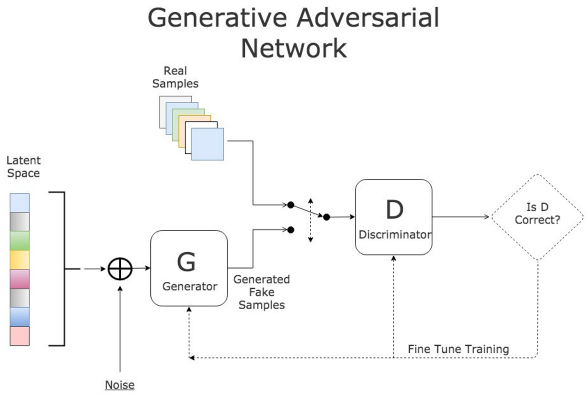
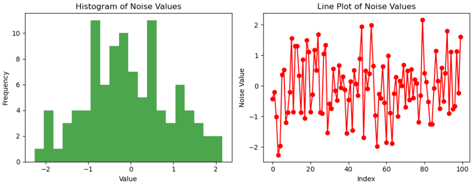
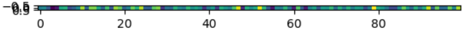


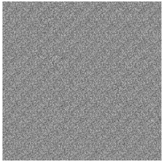
Figure: Basic GAN architecture

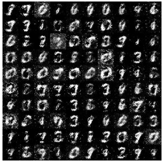
Generator Architecture and Visualization

```python
class Generator(nn.Module):

    def __init__(self, input_size=100, num_classes=784):
        super(Generator, self).__init__()
        self.layer = nn.Sequential(
            nn.Linear(input_size, 128),
            nn.LeakyReLU(0.2),
            nn.Linear(128, 256),
            nn.BatchNorm1d(256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 512),
            nn.BatchNorm1d(512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 1024),
            nn.BatchNorm1d(1024),
            nn.LeakyReLU(0.2),
            nn.Linear(1024, num_classes),
            nn.Tanh()
        )

    def forward(self, x):

        y_ = self.layer(x)
        y_ = y_.view(x.size(0), 1, 28, 28)
        return y_
```









Discriminator Architecture and Visualization

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

    def __init__(self, input_size=784, num_classes=1):

        super(Discriminator, self).__init__()
        self.layer = nn.Sequential(
            nn.Linear(input_size, 512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, num_classes),
            nn.Sigmoid(),
        )

    def forward(self, x):

        y_ = x.view(x.size(0), -1)
        y_ = self.layer(y_)
        return y_
```

```python
criterion = nn.BCELoss()
D_opt = torch.optim.Adam(D.parameters(), lr=0.0002,
      betas=(0.5, 0.999))
G_opt = torch.optim.Adam(G.parameters(), lr=0.0002,
      betas=(0.5, 0.999))

for epoch in range(max_epoch):

    for idx, (images, _) in enumerate(data_loader):

        # Training Discriminator
        x = images.to(DEVICE)
        x_outputs = D(x)
        D_x_loss = criterion(x_outputs, D_labels)

        z = torch.randn(batch_size, n_noise).to(DEVICE)
        z_outputs = D(G(z))
        D_z_loss = criterion(z_outputs, D_fakes)
        D_loss = D_x_loss + D_z_loss

        D.zero_grad()
        D_loss.backward()
        D_opt.step()

        if step % n_critic == 0:
            # Training Generator
            z = torch.randn(batch_size, n_noise).to(
                DEVICE)
            z_outputs = D(G(z))
            G_loss = criterion(z_outputs, D_labels)

            G.zero_grad()
            G_loss.backward()
            G_opt.step()
```

MinMax Loss Function for GANs

The MinMax loss function for a GAN is expressed as:

$$\min_G \max_D V(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})}[\log(1 - D(G(\mathbf{z})))]$$

- G is the generator, which tries to minimize this function against D.
- D is the discriminator, which tries to maximize this function.
- $\mathbf{x}$ are samples from the real data distribution p_{data}.
- $\mathbf{z}$ are input noise variables from distribution $p_{\mathbf{z}}$.

Derivation for the Value Function

The value function for a GAN is given by:

$$V(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})}[\log(1 - D(G(\mathbf{z})))]$$

To find the optimal discriminator, we calculate the derivative of $V(D, G)$ with respect to D and set it to zero. The derivative is given by:

$$\frac{\partial V}{\partial D} = \frac{\partial}{\partial D}\left(\mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})}[\log D(\mathbf{x})]\right) + \frac{\partial}{\partial D}\left(\mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})}[\log(1 - D(G(\mathbf{z})))]\right)$$

This simplifies to:

$$\frac{\partial V}{\partial D} = \frac{p_{\text{data}}(\mathbf{x})}{D(\mathbf{x})} - \frac{p_{\mathbf{z}}(\mathbf{z})}{1 - D(G(\mathbf{z}))}$$

Setting $\frac{\partial V}{\partial D} = 0$ for optimality, setting y $=$ $G(z)$, we find:

$$\frac{p_{\text{data}}(\mathbf{x})}{D(\mathbf{x})} = \frac{p_{\text{g}}(\mathbf{x})}{1 - D(x)}$$

From the optimality condition, the optimal $D(x)$ that discriminates between real data $\mathbf{x}$ and generated data $G(\mathbf{z})$ is:

$$D(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)}$$

where $p_g(x)$ is the density of generated data. This form of $D(x)$ maximizes the probability of correctly identifying real and generated samples.

Replace Value Function using JS-Divergence

By substituting the optimal discriminator $D(x)$ into the objective function, we have:

$$V(D^*, G) = \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}(\mathbf{x})}\left[\log\left(\frac{p_{\text{data}}(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_g(\mathbf{x})}\right)\right] + \mathbb{E}_{\mathbf{x}\sim p_{\mathrm{g}}(\mathbf{x})}\left[\log\left(1 - \frac{p_g(x)}{p_{\text{data}}(x) + p_g(x)}\right)\right]$$

The Jensen-Shannon divergence between two distributions p_{data} and p_g is defined as:

$$JS(p_{\text{data}} \| p_g) = \frac{1}{2}\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}\left[\log\frac{2p_{\text{data}}(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_g(\mathbf{x})}\right] + \frac{1}{2}\mathbb{E}_{\mathbf{x}\sim p_g}\left[\log\frac{2p_g(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_g(\mathbf{x})}\right]$$

The optimal $V(D, G)$ can be linked to the Jensen-Shannon divergence:

$$V(D, G) = -2\log 2 + 2JS(p_{\text{data}} \| p_g)$$

When $p_g = p_{\text{data}}$, the JS divergence reaches its minimum of 0, and hence:

$$\min V(D, G) = -2\log 2$$

Backpropagation with Gradient

The Binary Cross-Entropy Loss for a single data point with true label y and predicted probability $\hat{y}$ is defined as follows: For a batch of data, the loss is usually computed as the average over all instances:

$$\text{BCELoss} = -\frac{1}{N}\sum_{i=1}^{N}\left(y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right) \tag{1}$$

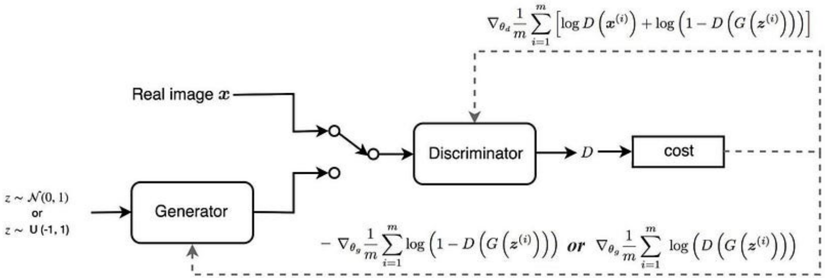


Figure: Backpropagation for GAN

Shortcoming and Improvement

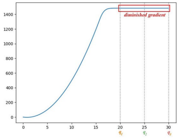


Figure: 1. Diminished Gradient

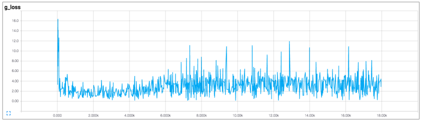


Figure: 2. No Convergence

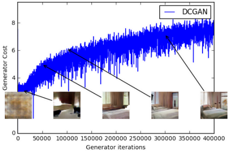


Figure: 3. Loss $\neq$ quality

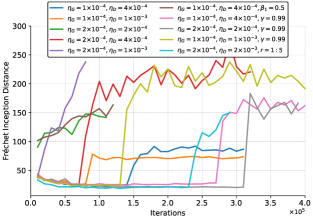


Figure: 4. Highly sensitive to hyperparameters

WGAN and WGAN-GP

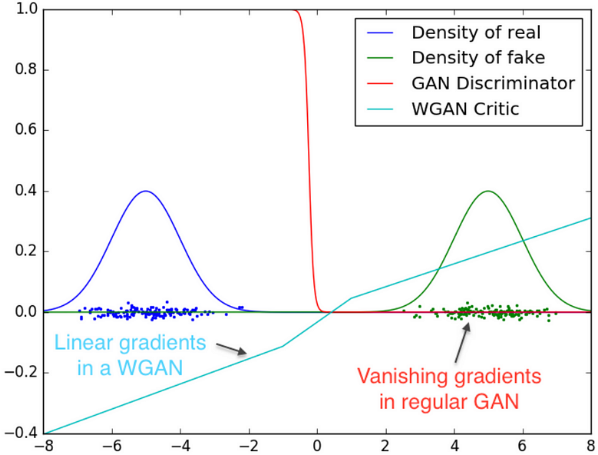


Figure: Visualization for Vanishing Gradient

Mathematics Derivation for Wasserstein GAN

The Wasserstein distance is the minimum cost of transporting mass in converting the data distribution p to the data distribution q. It is mathematically defined as the greatest lower bound (infimum) for any transport plan:

$$W(P_r, P_g) = \inf_{\gamma \in \Pi(P_r, P_g)} \mathbb{E}_{(x,y)\sim\gamma}[\|x - y\|]$$

where $\Pi(P_r, P_g)$ denotes the set of all joint distributions $\gamma(x, y)$ whose marginals are P_r and P_g, respectively.

$$L = \underbrace{\mathbb{E}_{x\sim P_r}[D(x)] - \mathbb{E}_{z\sim P_z}[D(G(z))]}_{\text{Original WGAN Loss}} + \lambda \underbrace{\mathbb{E}_{\hat{x}\sim P_{\hat{x}}}[(\|\nabla_{\hat{x}} D(\hat{x})\|_2 - 1)^2]}_{\text{Gradient Penalty}} \tag{2}$$

Hint: Kantorovich-Rubinstein duality and 1-Lipschitz

DCGAN

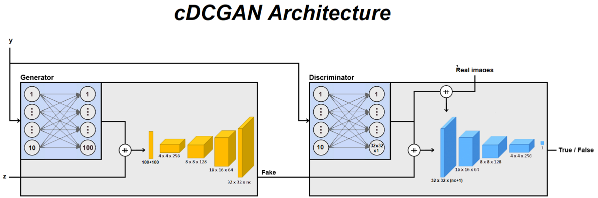


Figure: Deep Convolution Generative Adversarial Network

Conditional-XXX-GAN

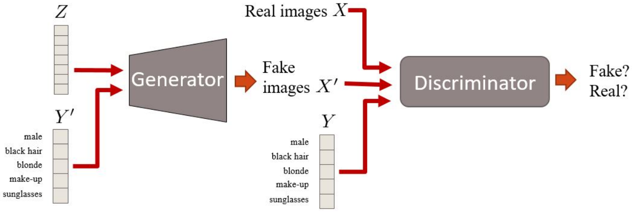


Figure: Conditional + ANY GAN

Hint: From unsupervised model to semi-supervised model

Enjoy the GAN zoooooooooo!