CoGAN: Coupled Generative Adversarial Networks (GAN)

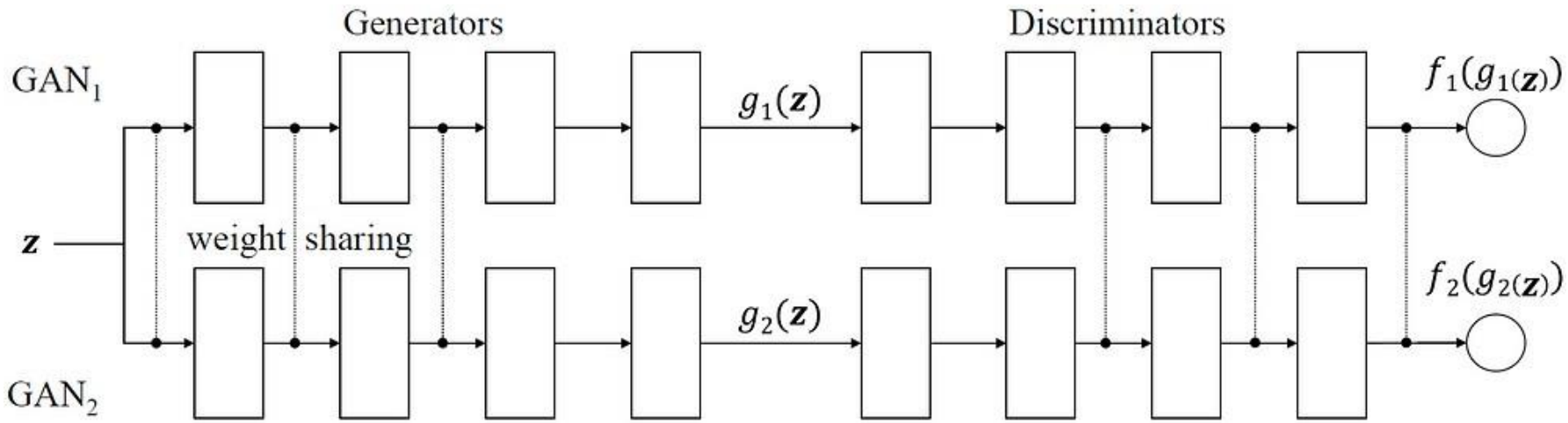
Main Idea

- Coupled Generative Adversarial Networks (CoGANs) consist of two GANs that share parameters in their layers (the lower-level layers of the generators and discriminators) to learn a joint distribution of data from two related domains without requiring paired data samples from the domains.

- Example: human faces with blond hair and dark hair

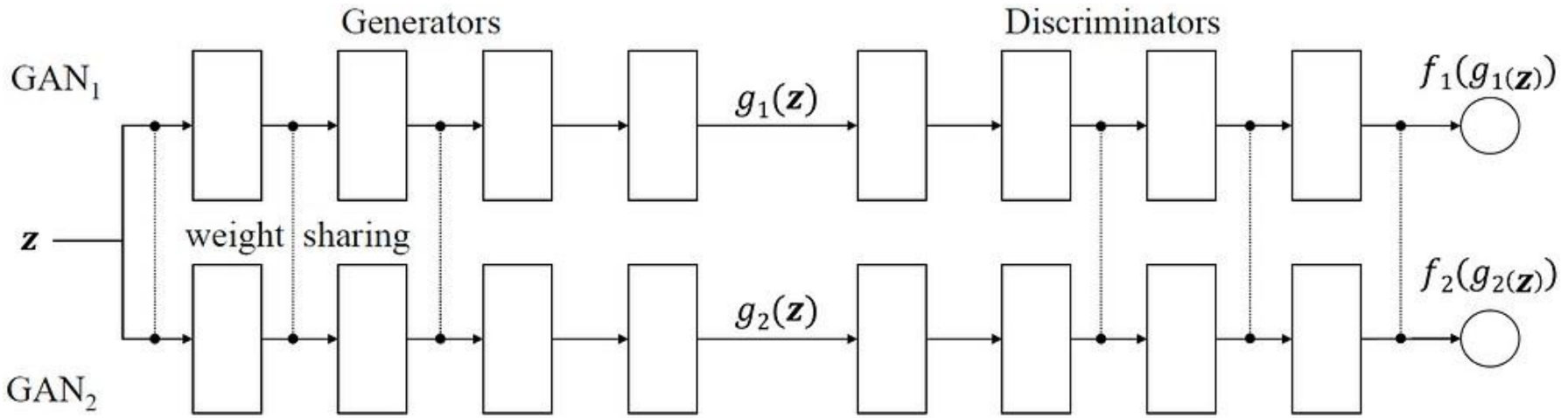
Main Idea

- A single input vector can generates correlated outputs in different domains through multiple GANs with weight sharing.



Main Idea

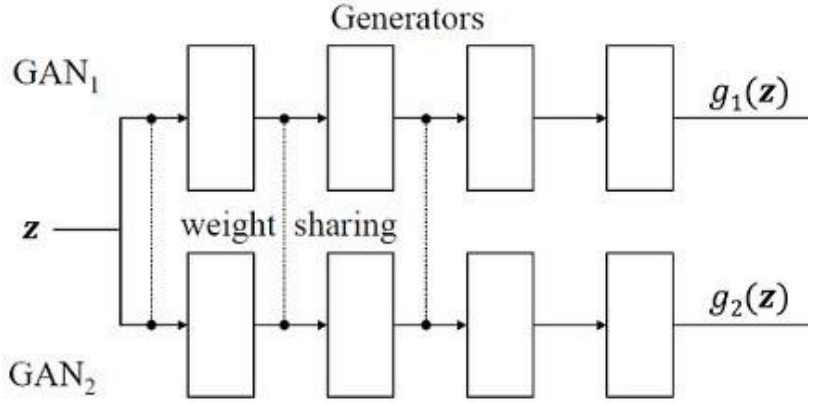
- CoGAN is designed for learning a joint distribution of images in two different domains.
- It consists of a pair of GANs — GAN1 and GAN2; each is responsible for synthesizing images in one domain.



Application

- Possible applications: Producing color image and depth image where these two images are highly correlated, i.e. describing the same scene, or images of the same face with different attributes (smiling and non-smiling).

Generators



- Both *g*1 and *g*2 are realized as multilayer neural network:

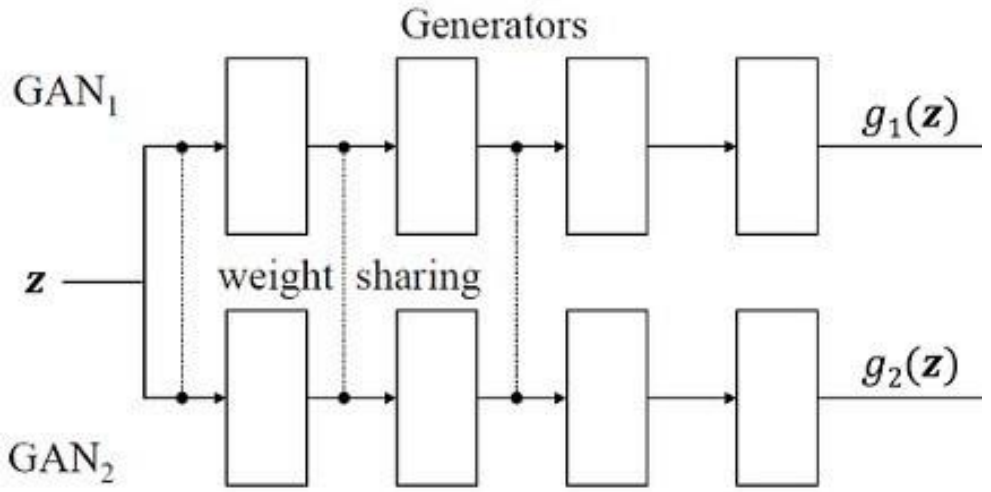
$$g_1(\mathbf{z}) = g_1^{(m_1)}(g_1^{(m_1-1)}(\ldots g_1^{(2)}(g_1^{(1)}(\mathbf{z})))), \quad g_2(\mathbf{z}) = g_2^{(m_2)}(g_2^{(m_2-1)}(\ldots g_2^{(2)}(g_2^{(1)}(\mathbf{z}))))$$

- Through layers, the generative models gradually decode information from more abstract concepts to more material details.
- The first layers decode high-level semantics and the last layers decode low-level details.
- No constraints are enforced to the last layers.

Generators

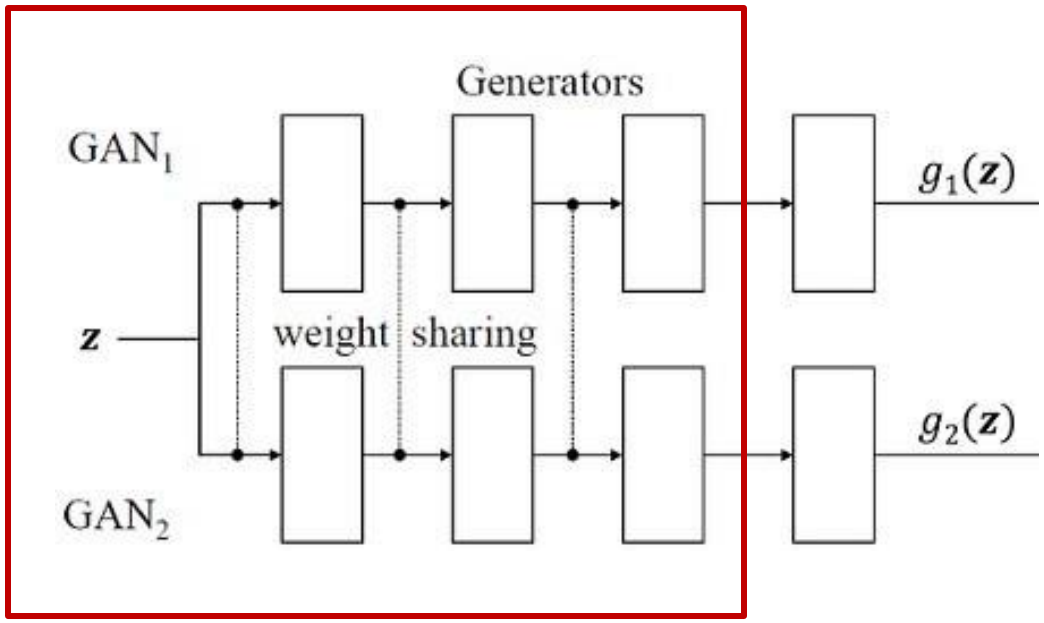
- The idea is to force the first layers of *g1* and *g2* to have identical structure and share the weights.

- With weight sharing, the pair of images can share the same high-level abstraction but having different low-level realizations.



Generators

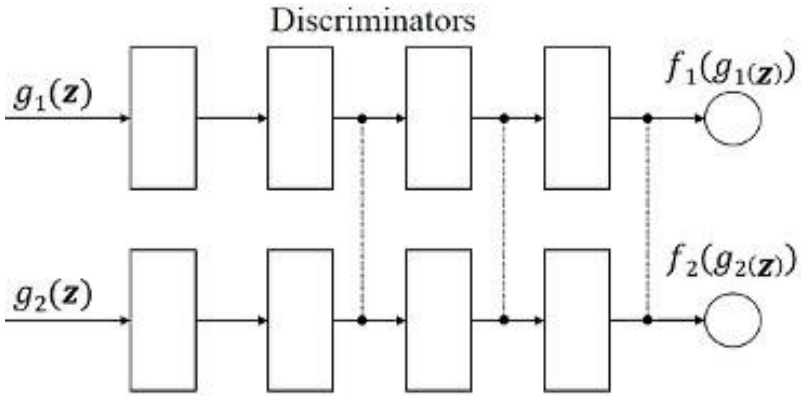
Shared layer




```python
# Coupled Generators
class CoupledGenerators(nn.Module):
    def __init__(self, z_dim, output_dim):
        super(CoupledGenerators, self).__init__()

        # Shared layer
        self.shared_fc = nn.Sequential(
            ...
        )

        # Specific output layers for each generator
        self.G1 = nn.Sequential(
            ...
        )
        self.G2 = nn.Sequential(
            ...
        )

    def forward(self, z):
        shared_out = self.shared_fc(z)
        gen1_out = self.G1(shared_out)
        gen2_out = self.G2(shared_out)
        return gen1_out, gen2_out
```

Discriminators

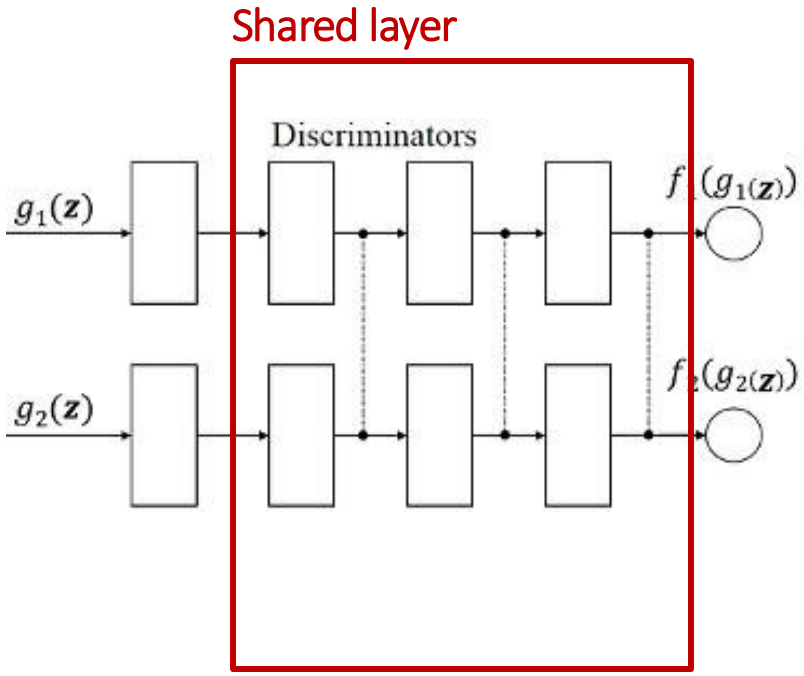
- The discriminative models map an input image to a probability score, estimating the likelihood that the input is drawn from a true data distribution.



$$f_1(\mathbf{x}_1) = f_1^{(n_1)}\left(f_1^{(n_1-1)}\left(\dots f_1^{(2)}\left(f_1^{(1)}(\mathbf{x}_1)\right)\right)\right),\ f_2(\mathbf{x}_2) = f_2^{(n_2)}\left(f_2^{(n_2-1)}\left(\dots f_2^{(2)}\left(f_2^{(1)}(\mathbf{x}_2)\right)\right)\right)$$

- The first layers of the discriminative models extract low-level features, while the last layers extract high-level features.
  - Sharing first layers allows the discriminators to learn a common representation for the two domains, capturing shared features between them.
  - Last layers allow the discriminators to specialize in the nuances of their respective domains.

Discriminators




```python
# Coupled Discriminators
class CoupledDiscriminators(nn.Module):
    def __init__(self, input_dim):
        super(CoupledDiscriminators, self).__init__()

        # Shared layers for discriminators
        self.shared_fc = nn.Sequential(
            ...
        )

        # Specific output layers for each discriminator
        self.D1 = nn.Sequential(
            ...
        )
        self.D2 = nn.Sequential(
            ...
        )

    def forward(self, x1, x2):
        shared_out1 = self.shared_fc(x1)
        shared_out2 = self.shared_fc(x2)
        disc1_out = self.D1(shared_out1)
        disc2_out = self.D2(shared_out2)
        return disc1_out, disc2_out
```

Learning

- Similar to minmax GAN, CoGAN can be trained by back propagation with the alternating gradient update steps.
- In the game, there are two teams and each team has two players.

$$\max_{g_1,g_2} \min_{f_1,f_2} V(f_1, f_2, g_1, g_2), \quad \text{subject to} \quad \begin{aligned} &\boldsymbol{\theta}_{g_1^{(i)}} = \boldsymbol{\theta}_{g_2^{(i)}}, && \text{for } i = 1, 2, ..., k \\ &\boldsymbol{\theta}_{f_1^{(n_1-j)}} = \boldsymbol{\theta}_{f_2^{(n_2-j)}}, && \text{for } j = 0, 1, ..., l-1 \end{aligned}$$

$$\begin{aligned} V(f_1, f_2, g_1, g_2) = &E_{\mathbf{x}_1 \sim p_{\mathbf{X}_1}}[-\log f_1(\mathbf{x}_1)] + E_{\mathbf{z} \sim p_{\mathbf{Z}}}[-\log(1 - f_1(g_1(\mathbf{z})))] \\ &+ E_{\mathbf{x}_2 \sim p_{\mathbf{X}_2}}[-\log f_2(\mathbf{x}_2)] + E_{\mathbf{z} \sim p_{\mathbf{Z}}}[-\log(1 - f_2(g_2(\mathbf{z})))]. \end{aligned}$$

- Basically, the alternating gradient update steps are to train 2 discriminators one by one, then to train 2 generators one by one alternatively.

Learning

```
# Determine validity of real and generated images
validity1_real, validity2_real = coupled_discriminators(imgs1, imgs2)
validity1_fake, validity2_fake = coupled_discriminators(gen_imgs1.detach(), gen_imgs2.detach())

d_loss = (
    adversarial_loss(validity1_real, real_label)
    + adversarial_loss(validity1_fake, fake_label)
    + adversarial_loss(validity2_real, real_label)
    + adversarial_loss(validity2_fake, fake_label)
) / 4

d_loss.backward()
optimizer_D.step()
```

```
# Generate a batch of images
gen_imgs1, gen_imgs2 = coupled_generators(z)
# Determine validity of generated images
validity1, validity2 = coupled_discriminators(gen_imgs1, gen_imgs2)

g_loss = (adversarial_loss(validity1, real_label) + adversarial_loss(validity2, real_label)) / 2

g_loss.backward()
optimizer_G.step()
```

Key features of CoGANs

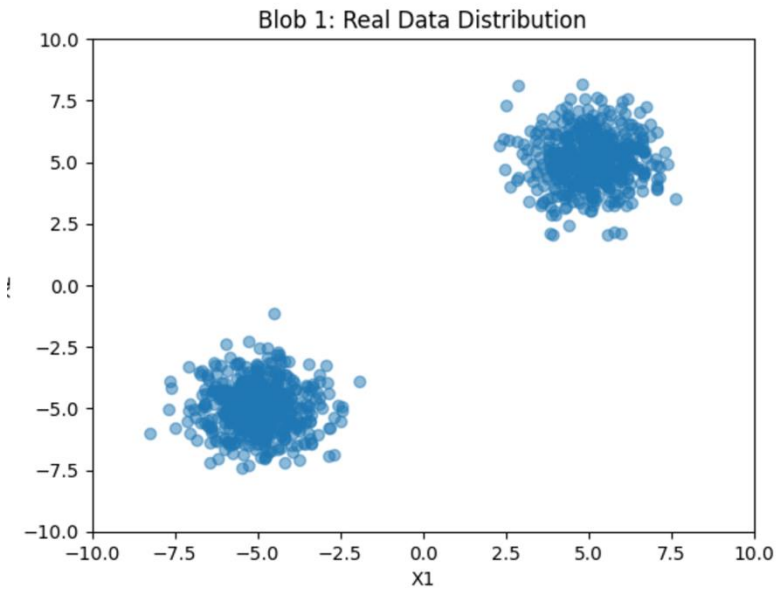
- Unpaired Data: Unlike many methods that rely on paired datasets (e.g., an image and its translation), CoGANs can learn from unpaired datasets, which are much easier to collect.

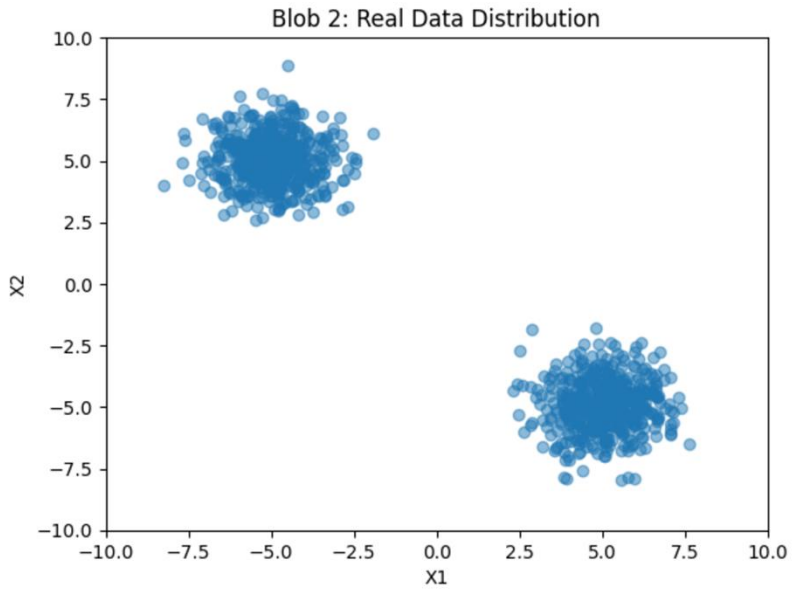
- Parameter Sharing: The lower layers of the generators and discriminators are shared across the two GANs, allowing the networks to learn common features (e.g., edges, shapes) while specializing in domain-specific features in higher layers.

- Domain Adaptation: CoGANs are particularly useful for tasks that involve learning correspondences between two related but distinct domains.

Full code

- Generate two different 2D distributions





https://colab.research.google.com/drive/1k2UuQVFFv-LNXNzJE-czDJUbRpMCNm54?usp=sharing

Full code

- Generate MNIST on MNIST-Modified images

https://drive.google.com/file/d/1oTFxWsTmmkOa7Xku0gP-q94moqLLYN5a/view?usp=sharing