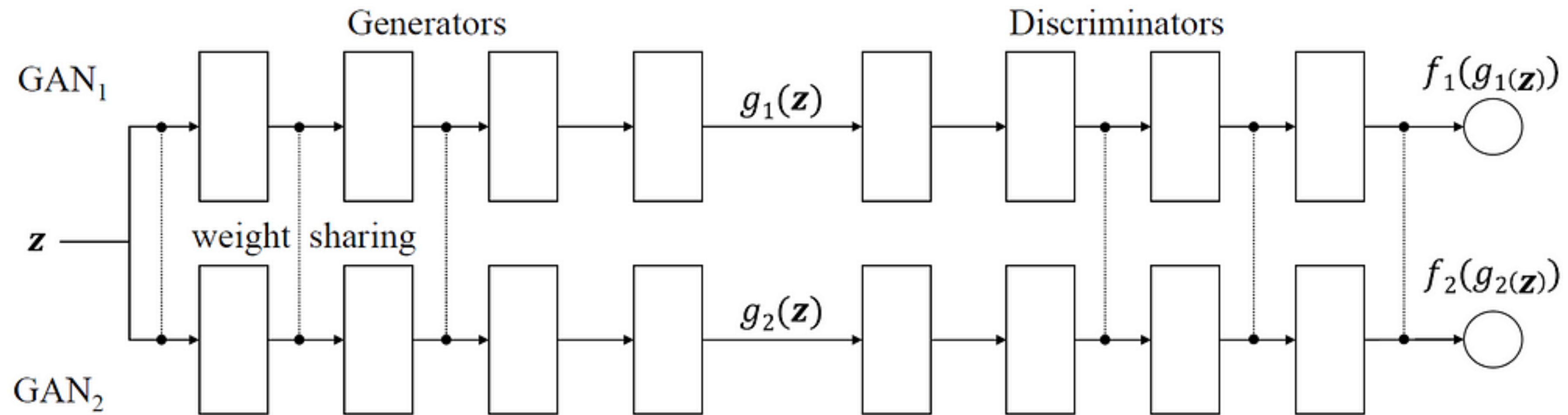
CoGAN: Coupled Generative Adversarial Networks (GAN)

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).

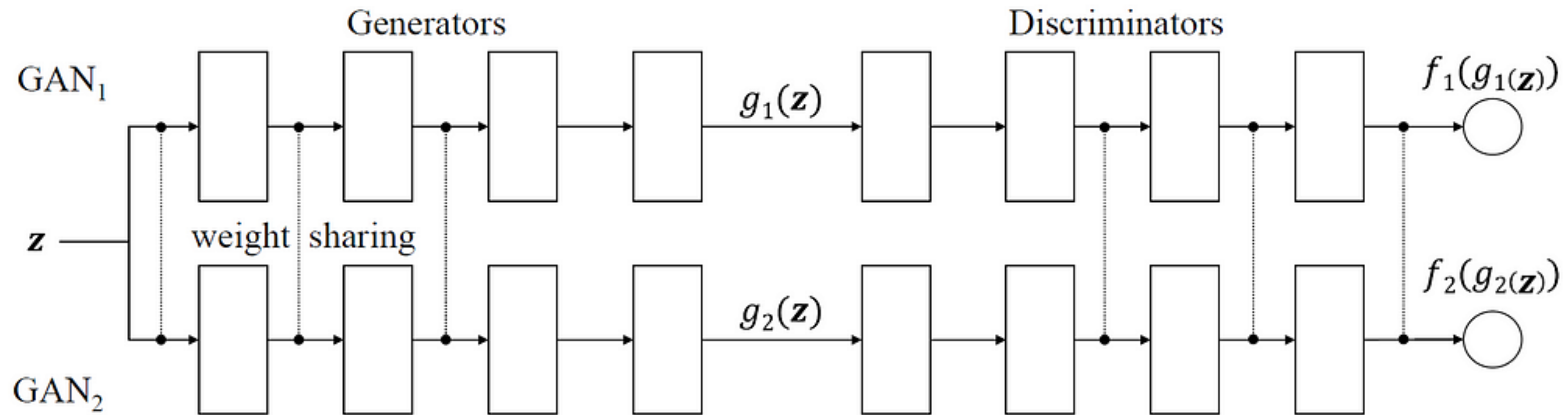
Main Idea

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

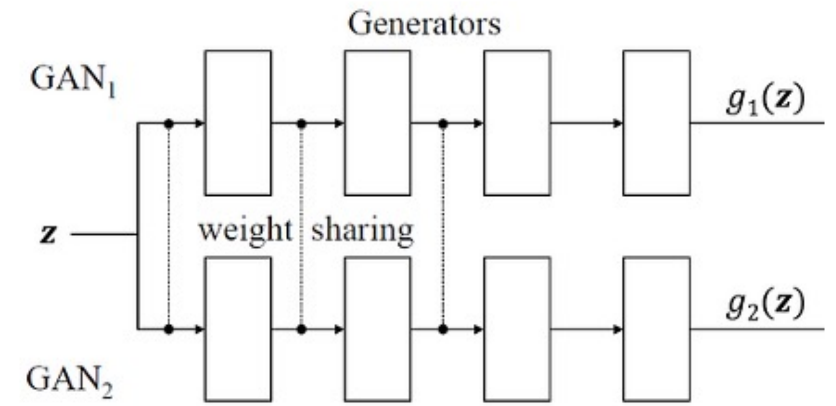


Main Idea

- 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.



Generators



- Both $g1$ and $g2$ are realized as multilayer neural network:

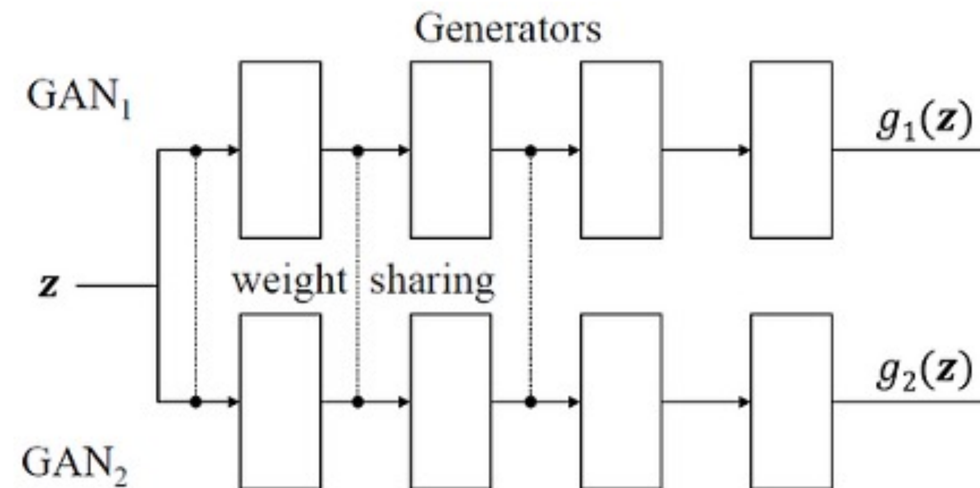
$$g_1(\mathbf{z}) = g_1^{(m_1)}(g_1^{(m_1-1)}(\dots g_1^{(2)}(g_1^{(1)}(\mathbf{z})))), \quad g_2(\mathbf{z}) = g_2^{(m_2)}(g_2^{(m_2-1)}(\dots g_2^{(2)}(g_2^{(1)}(\mathbf{z}))))$$

- 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.
- 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

- 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

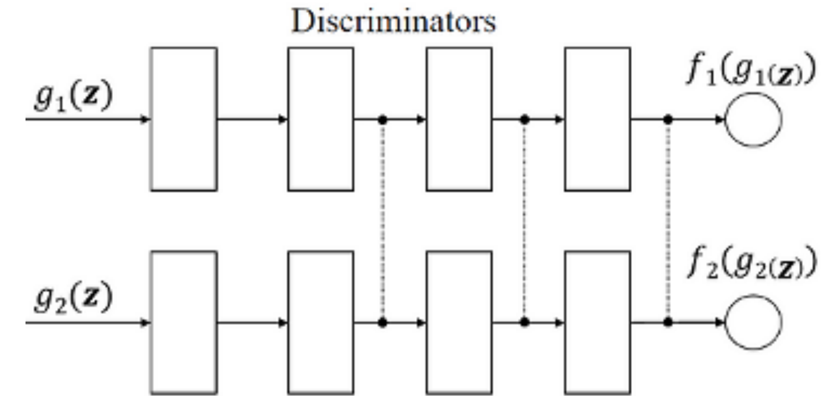
```
class CoupledGenerators(nn.Module):
    def __init__(self):
        super(CoupledGenerators, self).__init__()

        self.init_size = img_size // 4
        self.fc = nn.Sequential(nn.Linear(z_dim, 128 * self.init_size ** 2))

        self.shared_conv = nn.Sequential(
            ...
        )
        self.G1 = nn.Sequential(
            ...
        )
        self.G2 = nn.Sequential(
            ...
        )

    def forward(self, z):
        out = self.fc(z)
        out = out.view(out.shape[0], 128, self.init_size, self.init_size)
        img_emb = self.shared_conv(out)
        img1 = self.G1(img_emb)
        img2 = self.G2(img_emb)
        return img1, img2
```

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.

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

- The first layers of the discriminative models extract low-level features, while the last layers extract high-level features.
- Similar to generator, the last layers are weight shared.

Discriminators

```
class CoupledDiscriminators(nn.Module):
    def __init__(self):
        super(CoupledDiscriminators, self).__init__()

        def discriminator_block(in_filters, out_filters, bn=True):
            block = [nn.Conv2d(in_filters, out_filters, 3, 2, 1)]
            if bn:
                block.append(nn.BatchNorm2d(out_filters, 0.8))
            block.extend([nn.LeakyReLU(0.2, inplace=True), nn.Dropout2d(0.25
            return block

        self.shared_conv = nn.Sequential(
            *discriminator_block(channels, 16, bn=False),
            *discriminator_block(16, 32),
            *discriminator_block(32, 64),
            *discriminator_block(64, 128),
        )
        # The height and width of downsampled image
        ds_size = img_size // 2 ** 4
        self.D1 = nn.Linear(128 * ds_size ** 2, 1)
        self.D2 = nn.Linear(128 * ds_size ** 2, 1)

    def forward(self, img1, img2):
        # Determine validity of first image
        out = self.shared_conv(img1)
        out = out.view(out.shape[0], -1)
        validity1 = self.D1(out)
        # Determine validity of second image
        out = self.shared_conv(img2)
        out = out.view(out.shape[0], -1)
        validity2 = self.D2(out)

        return validity1, validity2
```

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

```python
# 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()
```

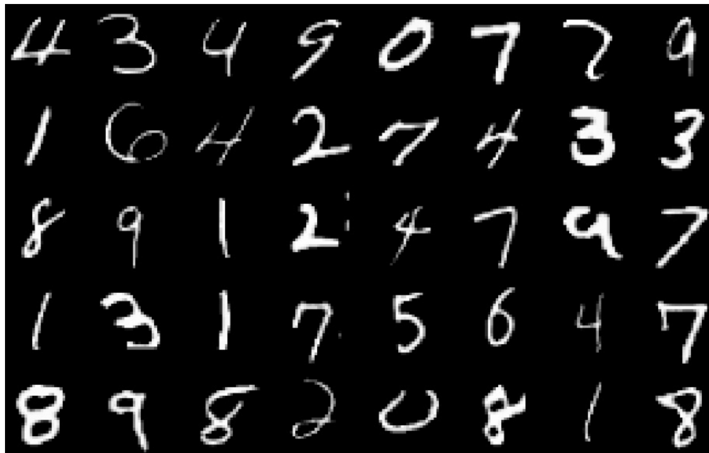
```python
# Generate a batch of images
gen_imgs1, gen_imgs2 = coupled_generators(z)
# Determine validity of generated images
validity1, validity2 = coupled_discriminators(gen_imgs1, gen_imgs2)

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

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

Full code

- Generate MNIST on MNIST-Modified images



https://drive.google.com/file/d/1vovpwO9h5CJAtOpJFAfRWnTyTcOjLY5H/view?usp=sharing