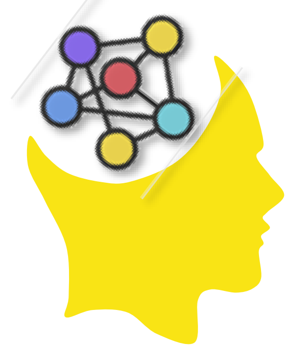


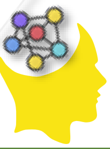
Adv ML for Gen-AI

Generative Adversarial Network

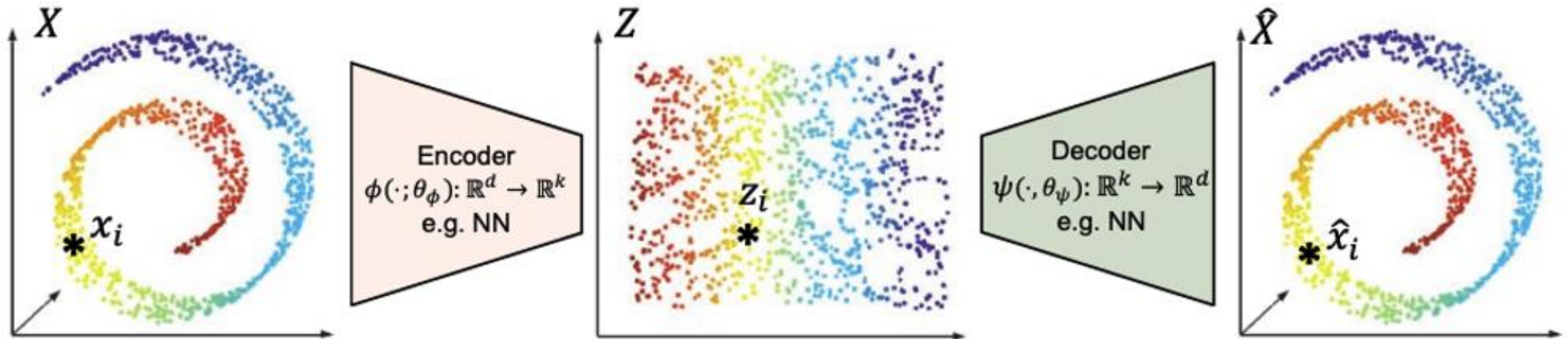
https://ml-graph.github.io/spring-2026/

Yu Wang, Ph.D.
Assistant Professor
Department of Computer Science
University of Oregon
Personal: https://yuwang0103.github.io/
Lab: https://kindlab-fly.github.io/

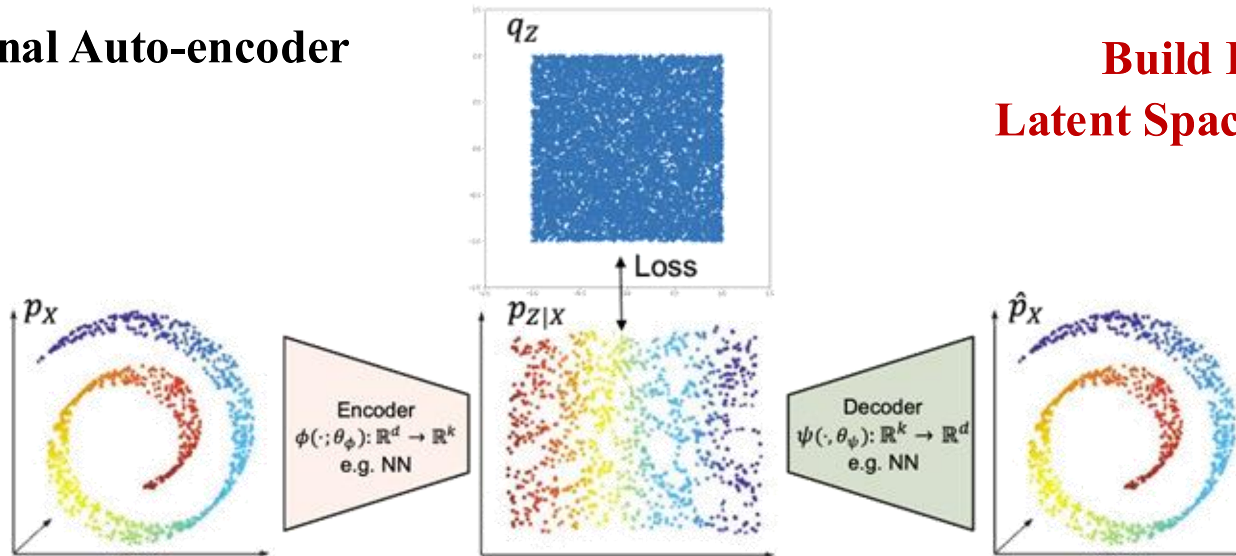
Recap



Auto-encoder based generative modeling



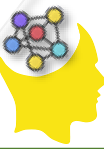
Variational Auto-encoder



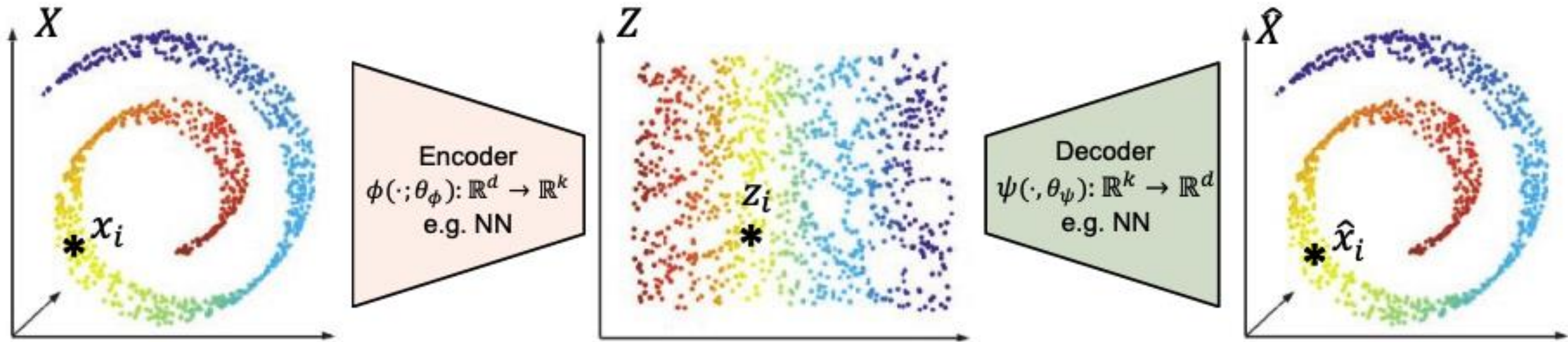
Build Latent Space

Latent Space – Decode Image

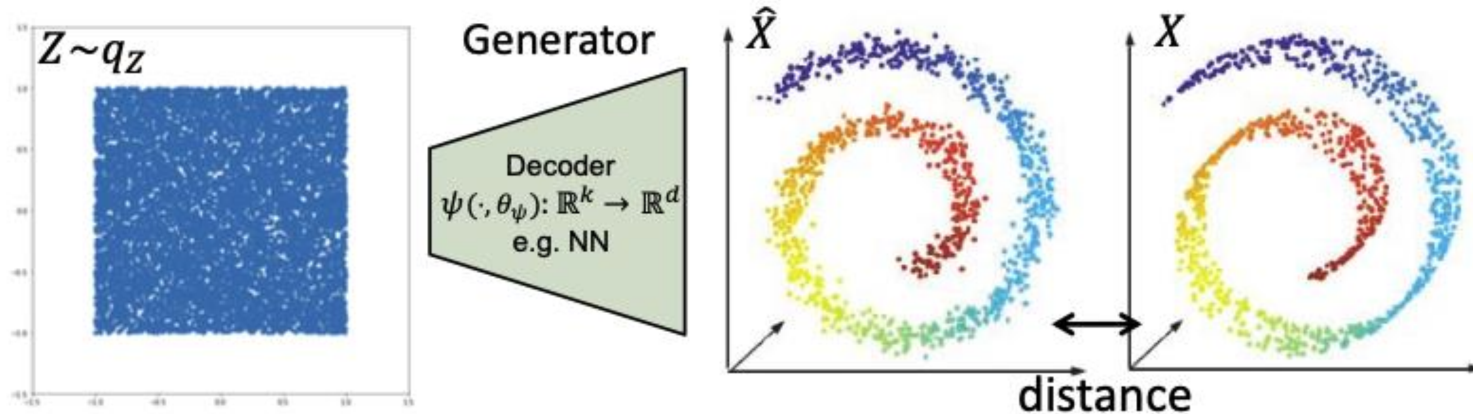
Question



Auto-encoder based generative modeling

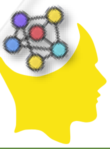


Encoder-less Generative Modeling

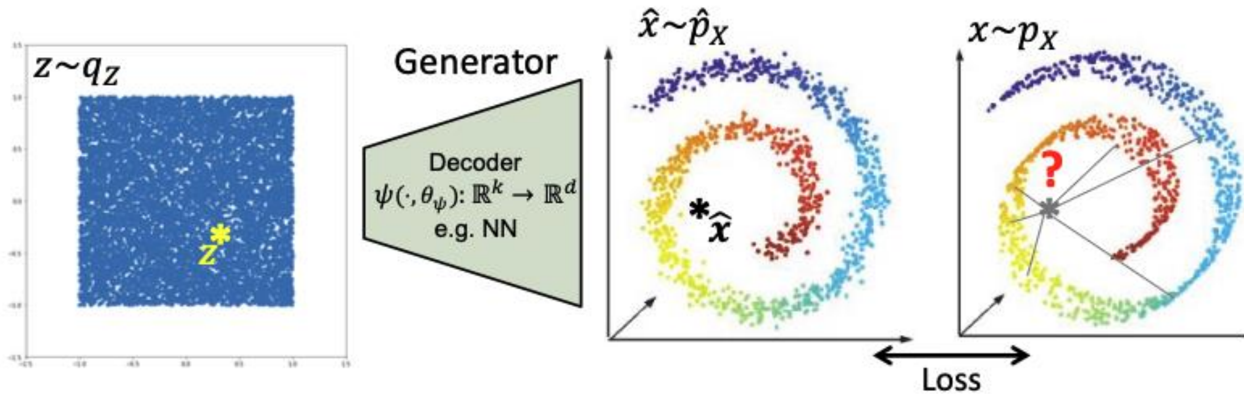


Can we remove encoder, but directly start from latent space and map?

Question



Generative Adversarial Networks: Main Idea



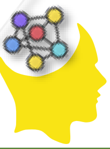
We sample $z \sim q_Z$ and generate $\hat{x} = \psi(z; \theta_\psi)$. Note that we want $\hat{x}$ to sit on the manifold of the original data, but we do not know the corresponding point for $\hat{x}$ in X!

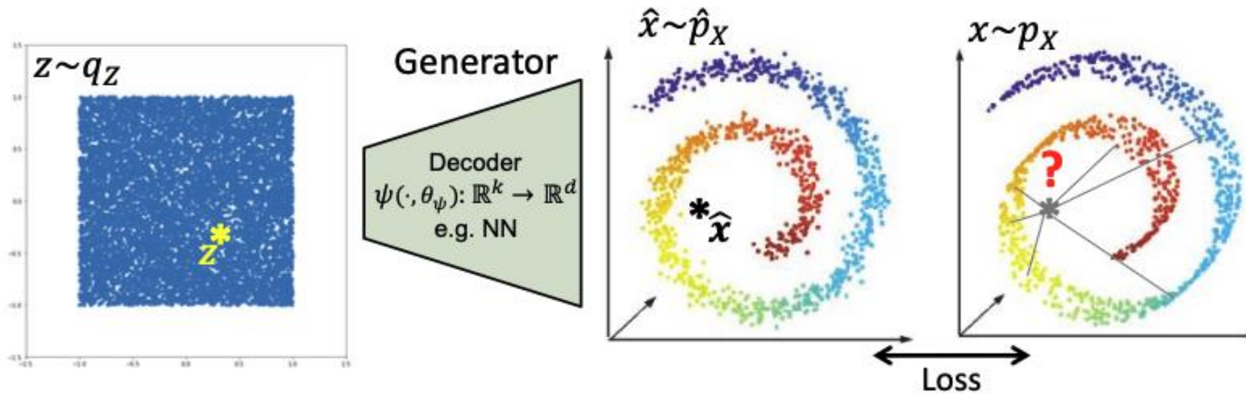
How to compute the likelihood?

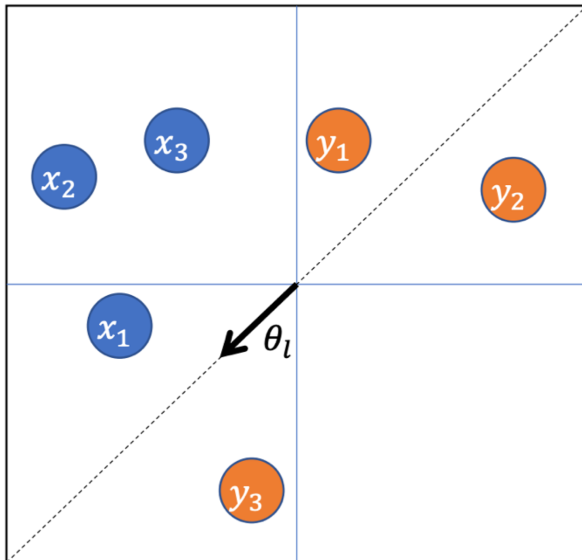
Sample-wise distances won't work in this setting, and one must rely on distribution-based distances.

Can we train a neural network to tell the difference and try to minimize that difference?

Question





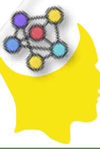


Imagine your neural network is some feature perspective extractor

$$y = w^T x + b$$

If two distributions are different, NN can always find some perspective.

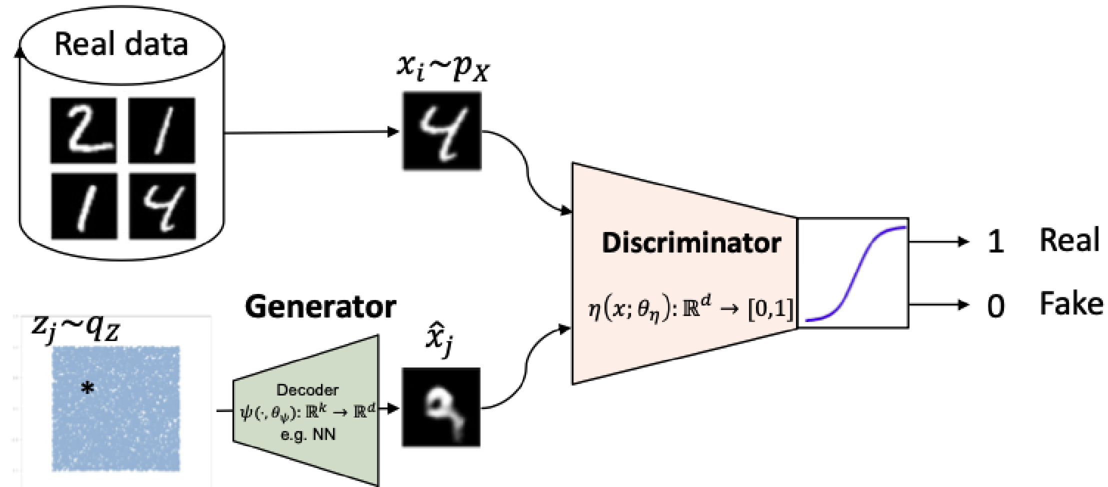
Generative Adversarial Networks



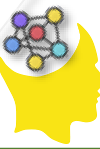
Generative Adversarial Networks (GAN)

The idea in GANs is to use an adversarial network that tries to distinguish the points on the manifold of the data from any other points in $\mathbb{R}^d$. Let $\eta(x;\theta_\eta):\mathbb{R}^d \to [0,1]$ be the discriminator:

$$\min_{\theta_\psi}\max_{\theta_\eta} \frac{1}{N}\sum_i \log(\eta(x_i;\theta_\eta)) + \frac{1}{M}\sum_j \log\left(1-\eta(\hat{x}_j;\theta_\eta)\right), \qquad where\ \ \hat{x}_j = \psi(z_j;\theta_\psi)$$



Generative Adversarial Networks



Minimax objective

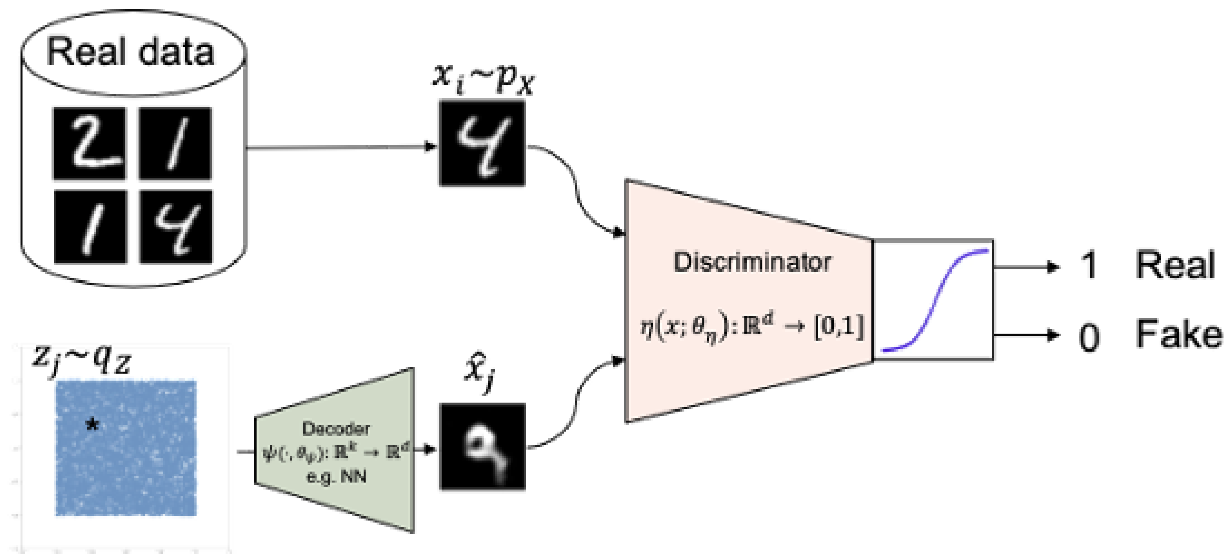
A GAN has two networks:

- **Generator** $G_\theta : \mathbb{R}^d \to \mathbb{R}^{784}$ — maps noise $z \sim \mathcal{N}(0, I)$ to fake images
- **Discriminator** $D_\phi : \mathbb{R}^{784} \to [0, 1]$ — predicts probability that an image is real

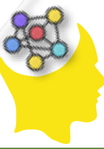
They play a **minimax game**:

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

- D wants to output 1 for real, 0 for fake → **maximise** V
- G wants D to output 1 for its fakes → **minimise** V (equivalently, maximise $\log D(G(z))$)



Generative Adversarial Networks



The Batch > Interviews & Essays > Article

Ian Goodfellow

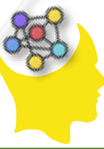
A Man, A Plan, A GAN

Interviews & Essays | Generative Adversarial Network (GAN) | GANs | Ian Goodfellow

Published Sep 30, 2020 | Reading time 3 min read

Brilliant ideas strike at unlikely moments. Ian Goodfellow conceived generative adversarial networks while spitballing programming techniques with friends at a bar. Goodfellow, who views himself as "someone who works on the core technology, not the applications," started at Stanford as a premed before switching to computer science and studying machine learning with Andrew Ng. "I realized that would be a faster path to impact more things than working on specific medical applications one

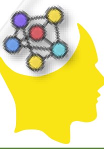
Generative Adversarial Networks

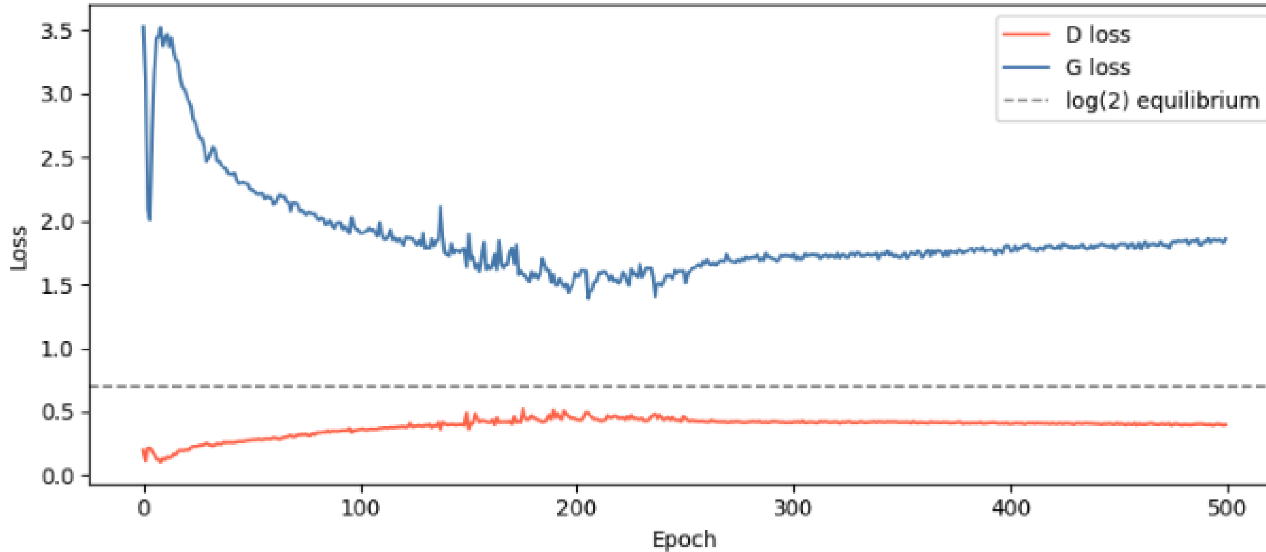


```python
class Generator(nn.Module):
    def __init__(self, latent_dim=LATENT_DIM):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(latent_dim, 256),
            nn.BatchNorm1d(256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 512),
            nn.BatchNorm1d(512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 784),
            nn.Sigmoid(),          # output in [0
        )
    def forward(self, z):
        return self.net(z)


class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(784, 512),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),
            nn.Linear(256, 1),
            nn.Sigmoid(),
        )
    def forward(self, x):
        return self.net(x)
```

```python
# Fixed noise for visualisation — same z throughout training
fixed_z = torch.randn(16, LATENT_DIM, device=device)

for epoch in range(1, N_EPOCHS + 1):
    epoch_d, epoch_g = 0.0, 0.0
    for real_imgs, _ in train_dl:
        real_imgs = real_imgs.to(device)
        B = real_imgs.size(0)

        real_labels = torch.ones (B, 1, device=device)
        fake_labels = torch.zeros(B, 1, device=device)

        # ── Train Discriminator ──────────────────────────────────
        z         = torch.randn(B, LATENT_DIM, device=device)
        fake_imgs = G(z).detach()          # detach: don't backprop into G yet

        loss_D_real = criterion(D(real_imgs), real_labels)
        loss_D_fake = criterion(D(fake_imgs), fake_labels)
        loss_D      = (loss_D_real + loss_D_fake) / 2

        opt_D.zero_grad(); loss_D.backward(); opt_D.step()

        # ── Train Generator ──────────────────────────────────────
        z         = torch.randn(B, LATENT_DIM, device=device)
        fake_imgs = G(z)

        # Non-saturating loss: G maximises log D(G(z))
        loss_G = criterion(D(fake_imgs), real_labels)

        opt_G.zero_grad(); loss_G.backward(); opt_G.step()

        epoch_d += loss_D.item()
        epoch_g += loss_G.item()

    d_losses.append(epoch_d / len(train_dl))
    g_losses.append(epoch_g / len(train_dl))

    if epoch % 5 == 0:
        print(f'Epoch {epoch:3d}/{N_EPOCHS}  D_loss={d_losses[-1]:.4f}  G_loss={g_losses[-1]:.4f}')

print('Training complete.')
```

Generative Adversarial Networks

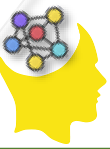


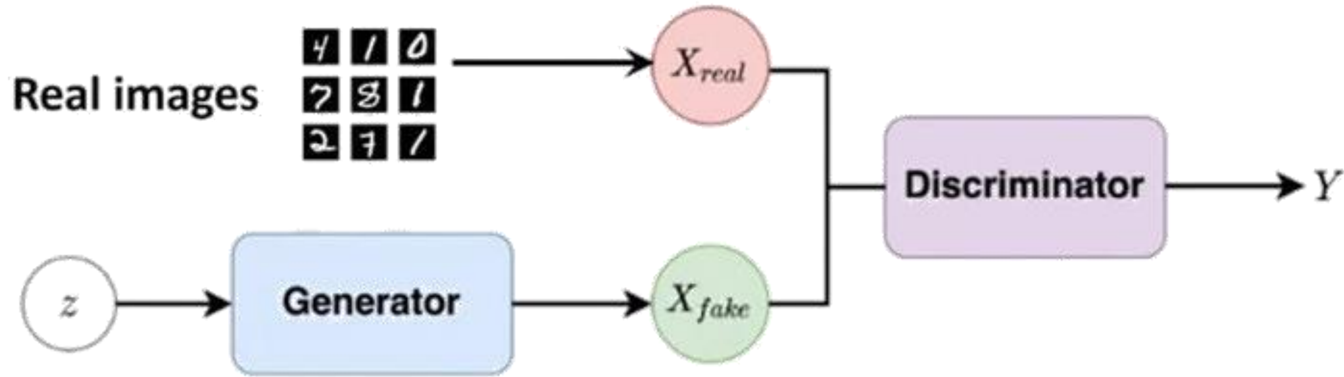


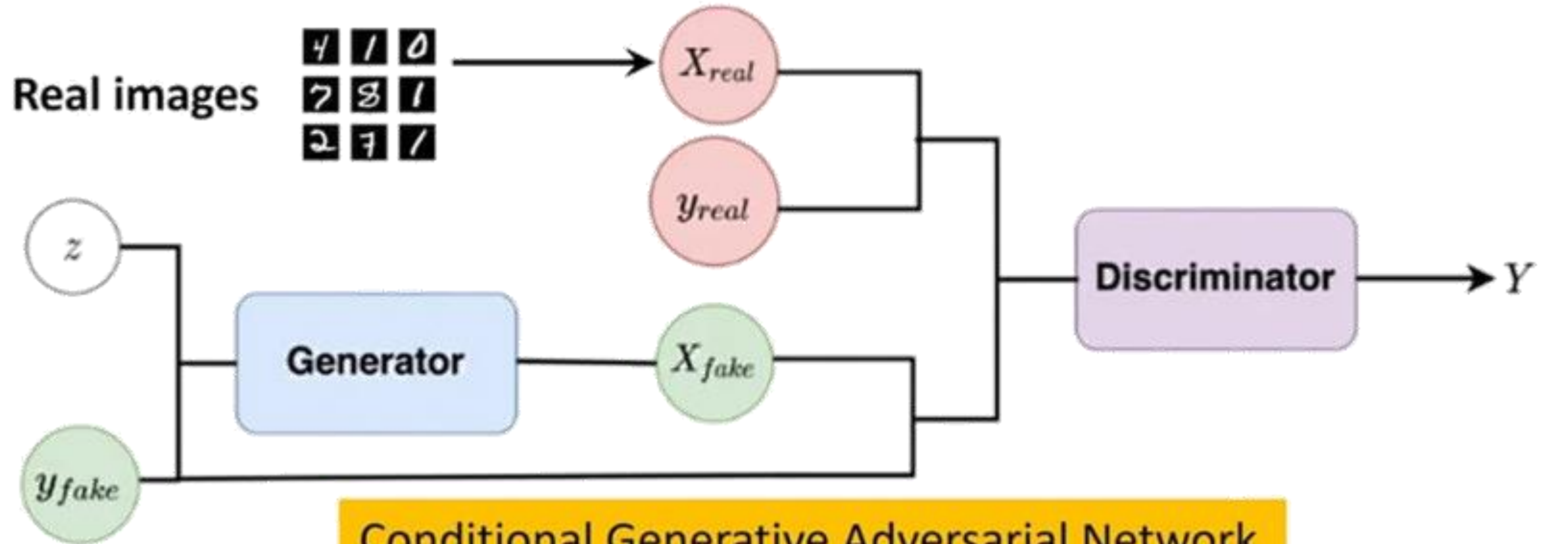
Discriminator cannot recognize what is real and what is fake

Generator continuously confuses discriminator

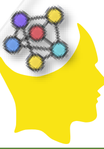
Class Conditional GAN







Class Conditional GAN



With a vanilla GAN, $G(z)$ generates a random digit — we can't ask for a specific class.

Conditional GAN (Mirza & Osindero, 2014) fixes this by conditioning both networks on a label y:

$$\min_G \max_D V(D, G) = \mathbb{E}_{x,y}[\log D(x \mid y)] + \mathbb{E}_{z,y}[\log(1 - D(G(z \mid y) \mid y))]$$

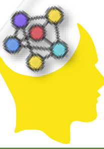
Both G and D receive the class label as an additional input.

At test time, we can choose any label y to get a digit of that class.

The standard approach: **embedding + concatenation**

```
Generator:     [z (64-d) | embed(y) (16-d)]  →  784
Discriminator: [x (784)  | embed(y) (16-d)]  →  1
```

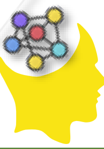
Class Conditional GAN



```python
class CondGenerator(nn.Module):
    def __init__(self, latent_dim=LATENT_DIM, embed_dim=EMBED_DIM, n_classes=N_CLASSES):
        super().__init__()
        self.label_emb = nn.Embedding(n_classes, embed_dim)
        self.net = nn.Sequential(
            nn.Linear(latent_dim + embed_dim, 256),
            nn.BatchNorm1d(256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 512),
            nn.BatchNorm1d(512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 784),
            nn.Sigmoid(),
        )
    def forward(self, z, y):
        emb = self.label_emb(y)          # (B, embed_dim)
        inp = torch.cat([z, emb], dim=1) # (B, latent_dim + embed_dim)
        return self.net(inp)


class CondDiscriminator(nn.Module):
    def __init__(self, embed_dim=EMBED_DIM, n_classes=N_CLASSES):
        super().__init__()
        self.label_emb = nn.Embedding(n_classes, embed_dim)
        self.net = nn.Sequential(
            nn.Linear(784 + embed_dim, 512),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),
            nn.Linear(256, 1),
            nn.Sigmoid(),
        )
    def forward(self, x, y):
        emb = self.label_emb(y)          # (B, embed_dim)
        inp = torch.cat([x, emb], dim=1) # (B, 784 + embed_dim)
        return self.net(inp)
```

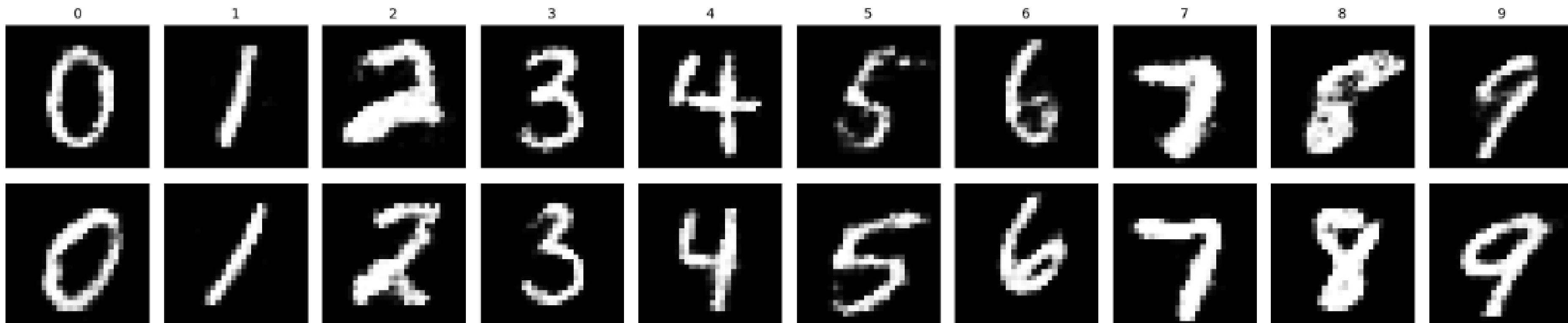
```python
for epoch in range(1, N_EPOCHS + 1):
    epoch_d, epoch_g = 0.0, 0.0
    for real_imgs, labels in train_dl:
        real_imgs = real_imgs.to(device)
        labels    = labels.to(device)
        B = real_imgs.size(0)

        real_labels = torch.ones (B, 1, device=device)
        fake_labels = torch.zeros(B, 1, device=device)

        # — Train cD ————————————————————————————————
        z         = torch.randn(B, LATENT_DIM, device=device)
        fake_imgs = cG(z, labels).detach()

        loss_D = (criterion(cD(real_imgs, labels), real_labels) +
                  criterion(cD(fake_imgs, labels), fake_labels)) / 2

        opt_cD.zero_grad(); loss_D.backward(); opt_cD.step()

        # — Train cG ————————————————————————————————
        z         = torch.randn(B, LATENT_DIM, device=device)
        fake_imgs = cG(z, labels)
        loss_G    = criterion(cD(fake_imgs, labels), real_labels)

        opt_cG.zero_grad(); loss_G.backward(); opt_cG.step()

        epoch_d += loss_D.item()
        epoch_g += loss_G.item()
```

Class Conditional GAN



```python
# —— Show class-conditioned generation ——————————————————————————————
cG.eval()
with torch.no_grad():
    cgan_samples = cG(fixed_z_cgan, fixed_y_cgan).cpu().numpy().reshape(-1, 28, 28)
cG.train()

fig, axes = plt.subplots(2, 10, figsize=(16, 4))
for i in range(2):
    for j in range(10):
        axes[i, j].imshow(cgan_samples[i*10+j], cmap='gray', vmin=0, vmax=1)
        axes[i, j].axis('off')
        if i == 0:
            axes[i, j].set_title(str(j), fontsize=10)
fig.suptitle('cGAN: each column is conditioned on digit label 0–9', fontsize=12)
plt.tight_layout()
plt.show()
```

```python
# Fixed z + labels 0–9 twice for a 2×10 display grid
fixed_z_cgan  = torch.randn(20, LATENT_DIM, device=device)
fixed_y_cgan  = torch.tensor(list(range(10)) * 2, device=device)
```



QA