# LAB 8:

# UNSUPERVISED LEARNING AND GANs

University of Washington, Seattle

Spr 2025

# OUTLINE

Part 1: Unsupervised Learning

- Supervised vs unsupervised
- Unsupervised learning in with NN

Part 2: Generative Model Taxidermy

- FVBN
- Variational Autoencoder
- GAN

Part 3: Generative Adversarial Networks

- GAN architecture

Part 4: GAN Optimization and Applications

- Competing cost function
- Minmax game optimization
- GAN variations

Part 5: GAN Example

- MNIST generation with Vanilla GAN
- Generator extension with convolution

Part 6: Lab Assignment

- MNIST generation with DCGAN

# Unsupervised Learning

Supervised vs Unsupervised

Unsupervised Learning in NN

# Supervised vs Unsupervised Learning

## Supervised

**Data:**
{x} x: inputs **WITH labels**

**Neural Network Goal:**
Minimize specific **error**

**Examples:** Classification,
Regression, Detection, Prediction

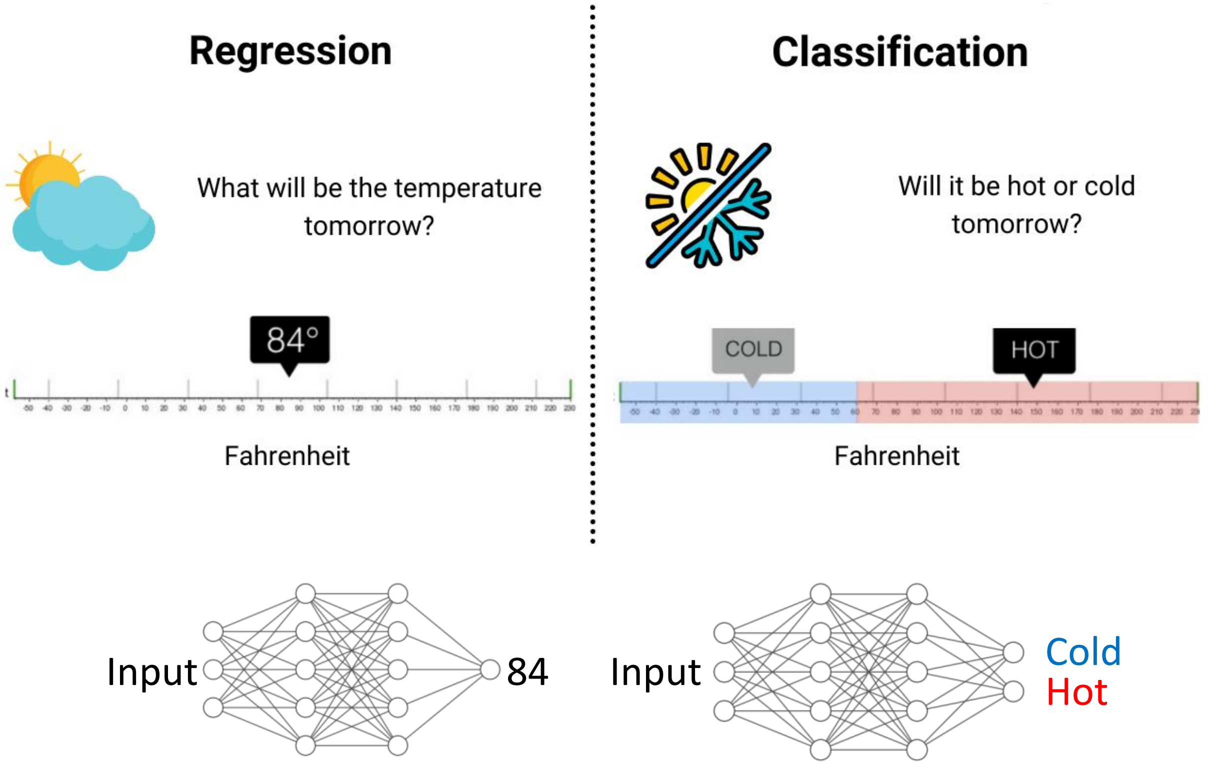
# Supervised vs Unsupervised Learning

## Supervised

**Data:**
{x} x: inputs **WITH labels**

**Neural Network Goal:**
Minimize specific **error**

**Examples:** Classification, Regression, Detection, Prediction



# Unsupervised Learning in NN

## Unsupervised

**Data:**

{x} x: inputs **WITHOUT labels**

**Neural Network Goal:**

Learn a **structure** of the data

# Unsupervised Learning in NN

Training Data

Training Data ~ $\mathbf{P_{data}(x)}$

# Unsupervised Learning in NN

## Training Data

## Generated Samples

http://www.whichfaceisreal.com/

Training Data ~ $\mathbf{P}_{\mathbf{data}}\mathbf{(x)}$

Generate Samples ~ $\mathbf{P}_{\mathbf{model}}\mathbf{(x)}$

# Unsupervised Learning in NN

## Training Data

## Generated Samples

http://www.whichfaceisreal.com/

Training Data ~ $\mathbf{P_{data}(x)}$

Generate Samples ~ $\mathbf{P_{model}(x)}$

Goal: Model estimated density ≈ Real world density

Core problem in unsupervised learning

# Unsupervised Learning in NN

## Unsupervised

**Data:**

{x} x: inputs **WITHOUT labels**

**+ No need for labeling → More data**

**- Challenge: Cost?**

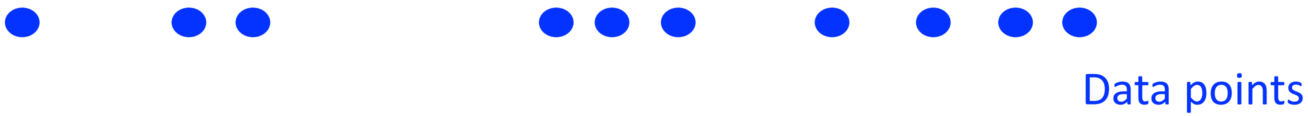
**Neural Network Goal:**

Learn a **structure** of the data

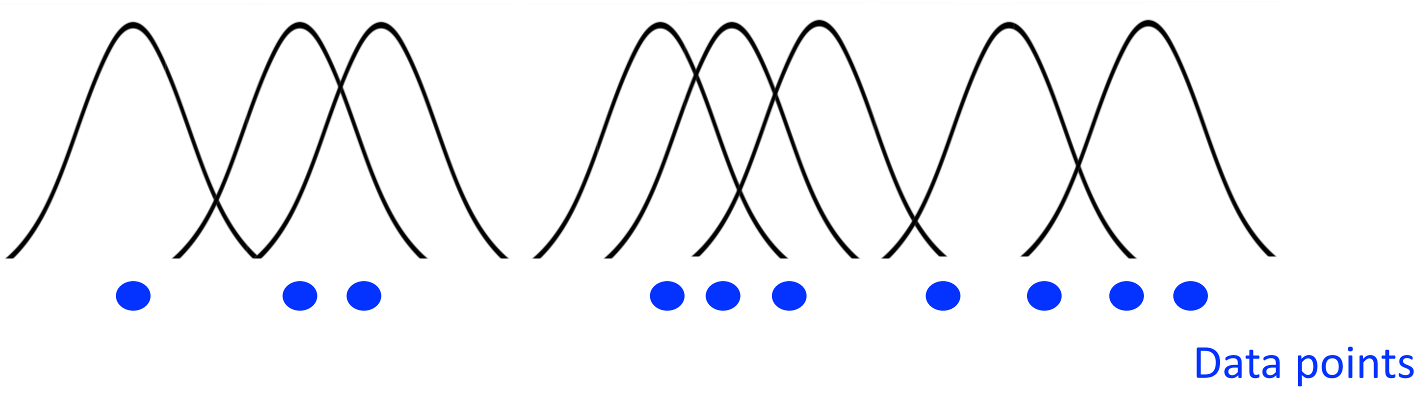
**+ Has the potential to learn the real world**

**- Challenge: Optimization?**

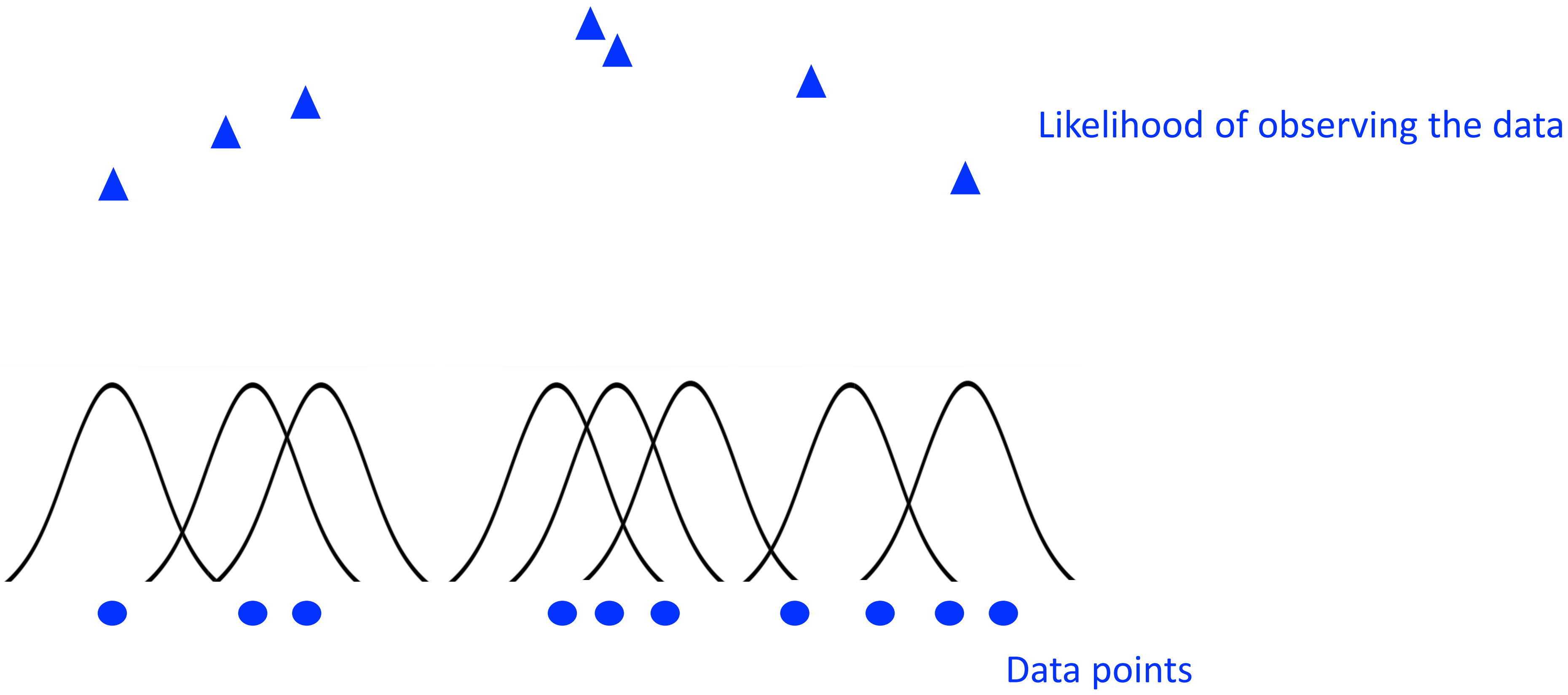
# Maximum Likelihood Estimation



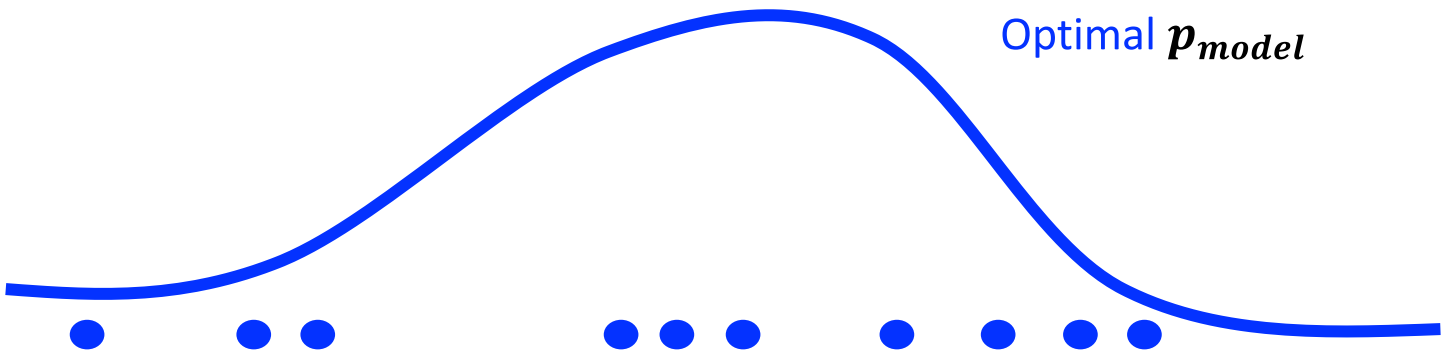
# Maximum Likelihood Estimation



# Maximum Likelihood Estimation



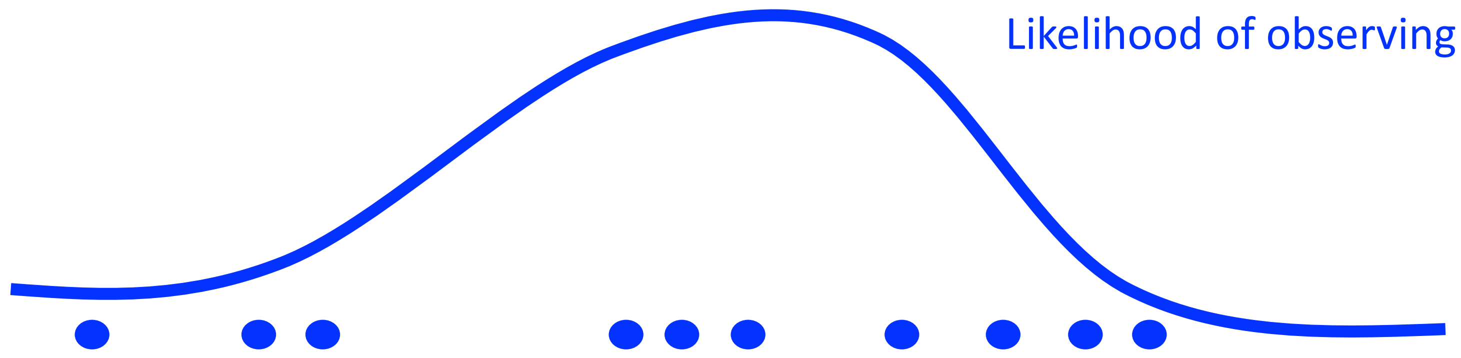
# Unsupervised Learning in NN




Model params

$$\theta^* = argmax_\theta \, E_{x \sim p_{data}} \log p_{model}(x|\theta)$$

# Unsupervised Learning in NN

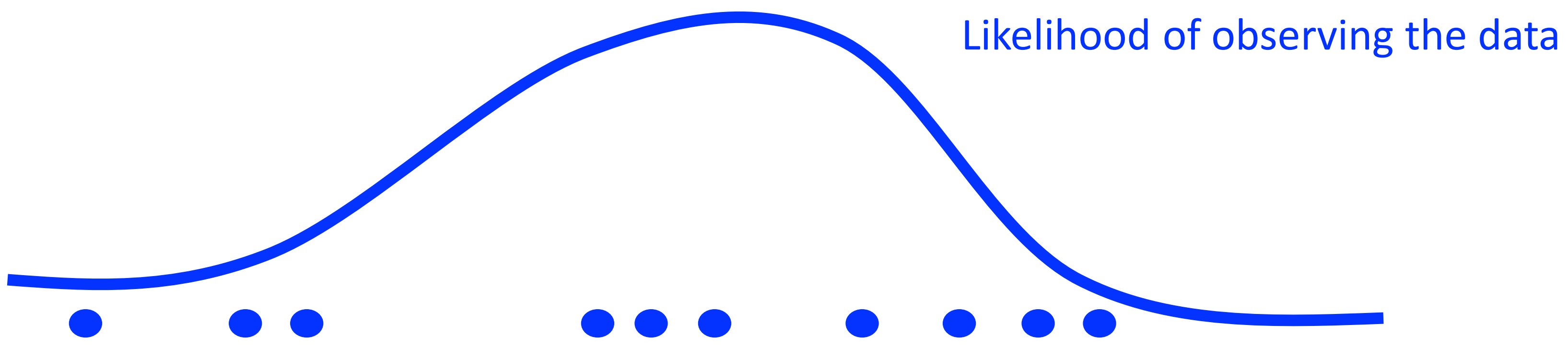



Model params

$$\theta^* = argmax_\theta \, E_{x \sim p_{data}} \, log \, p_{model}(x|\theta)$$

Goal: Find the optimal distribution $p_{model}(x|\theta)$ that best fit the data

# Unsupervised Learning in NN




Model params

$$\theta^* = argmax_\theta \, E_{x \sim p_{data}} \, log \, p_{model}(x|\theta)$$

**Explicit –** explicitly define and generate $P_{model}$

**Implicit -** generate $P_{model}$ without defining $P_{model}$ exactly

# Generative Model Taxidermy

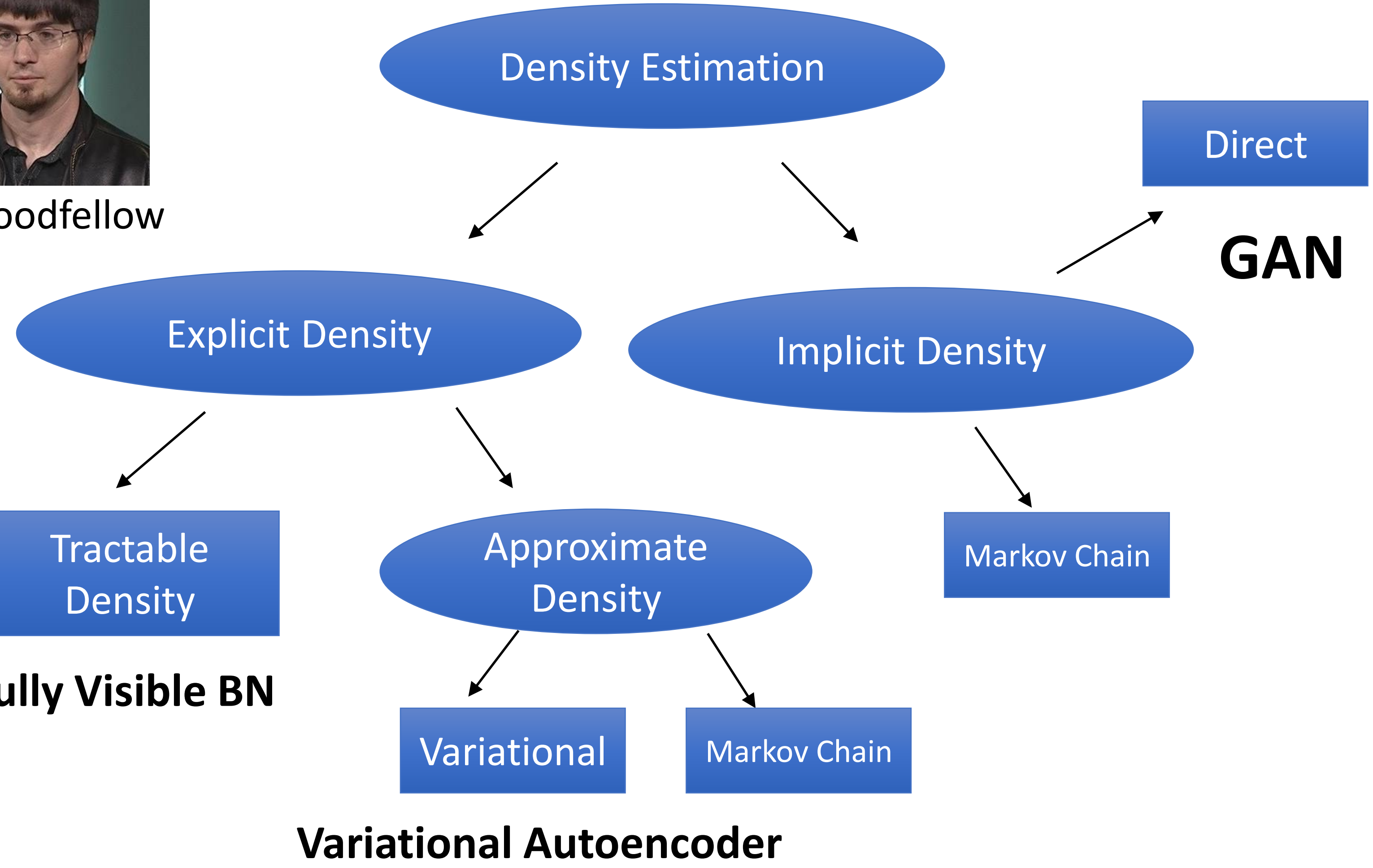
Fully Visible BN

Variational Autoencoder

Generative Adversarial Network

# Unsupervised Learning in NN

Ian Goodfellow



# Fully Visible BN

- Explicitly formula based on chain rule:

$$\boldsymbol{p_{model}(x) = p_{model}(x_1)\Pi_{i=2}^{n} p_{model}\,(x_i\,|x_1, x_2, \ldots, x_{i-1})}$$

- O(n) generation cost

- No control through hidden variables

# Language Model

**Language model:** probability distribution over sequences of words. Given such a sequence, say of length m, it assigns a probability to the whole sequence.

# Language Model

**Language model:** probability distribution over sequences of words. Given such a sequence, say of length m, it assigns a probability to the whole sequence.

P(W=NASA will **take** me to Moon) = p1
P(W= NASA will **bake** me to Moon) = p2
p2 << p1

# Language Model

**Language model:** probability distribution over sequences of words. Given such a sequence, say of length m, it assigns a probability to the whole sequence.

P(W=NASA will **take** me to Moon) = p1
P(W= NASA will **bake** me to Moon) = p2
p2 << p1

Chain rule is used to estimate probability:

$$P(w_1 w_2 \ldots w_n) = \prod_i P(w_i | w_1 w_2 \ldots w_{i-1})$$

P(W) = P(**NASA**) P(**will**| NASA) P(**take**| NASA will) P(**me**| NASA will take)
P(**to**| NASA will take me) P(**Moon** | NASA will take me to)

# Variational Autoencoder

$$p_{model}(x) = \prod_{i=2}^{n} p_{model}(x_i \mid x_1, x_2, \dots, x_{i-1})$$

$$\downarrow$$

$$p_{model}(x) = \int p_{model}(z) p_{model}(x|z)\, dz$$

# Variational Autoencoder

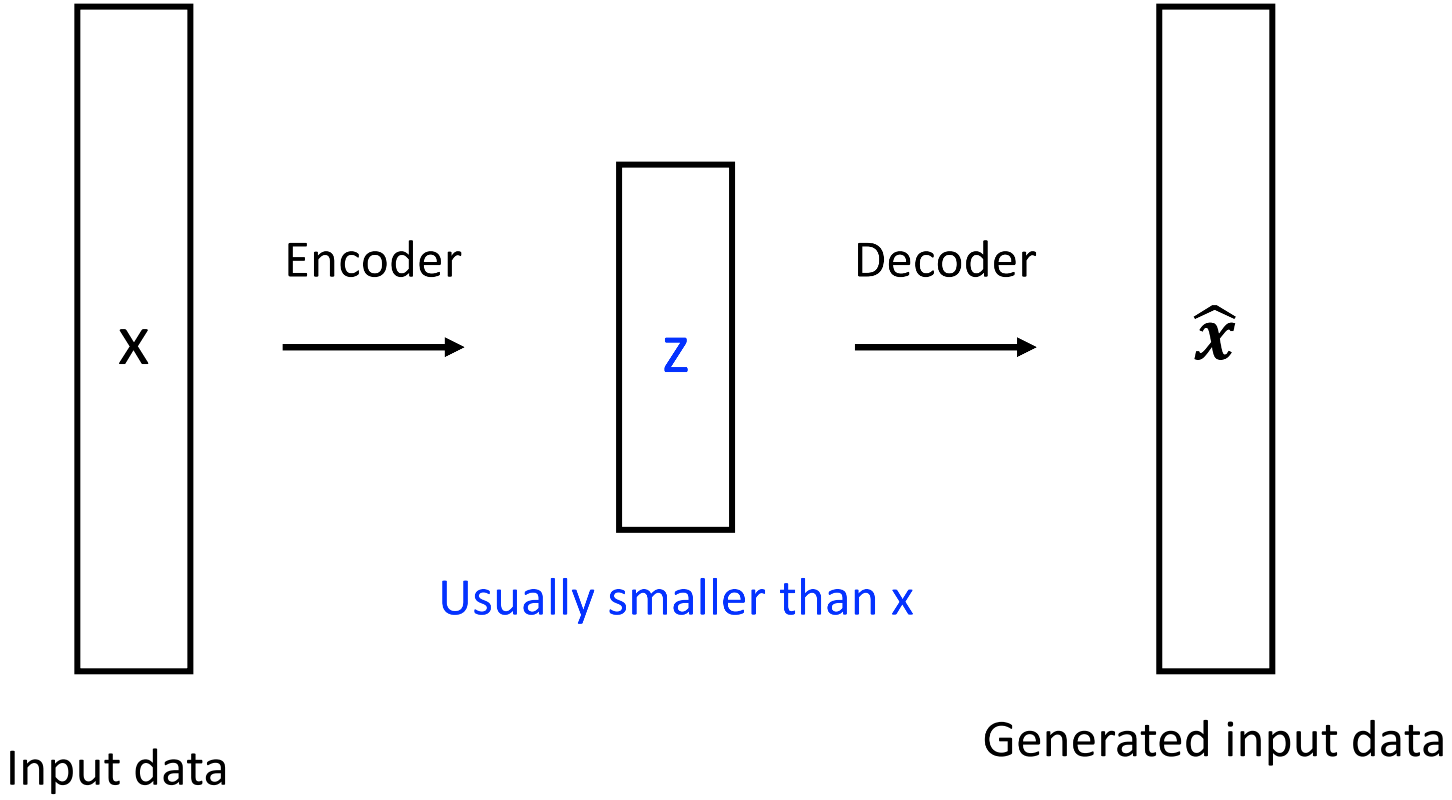
$$p_{model}(x) = \prod_{i=2}^{n} p_{model}(x_i \mid x_1, x_2, \ldots, x_{i-1})$$

$$\downarrow$$

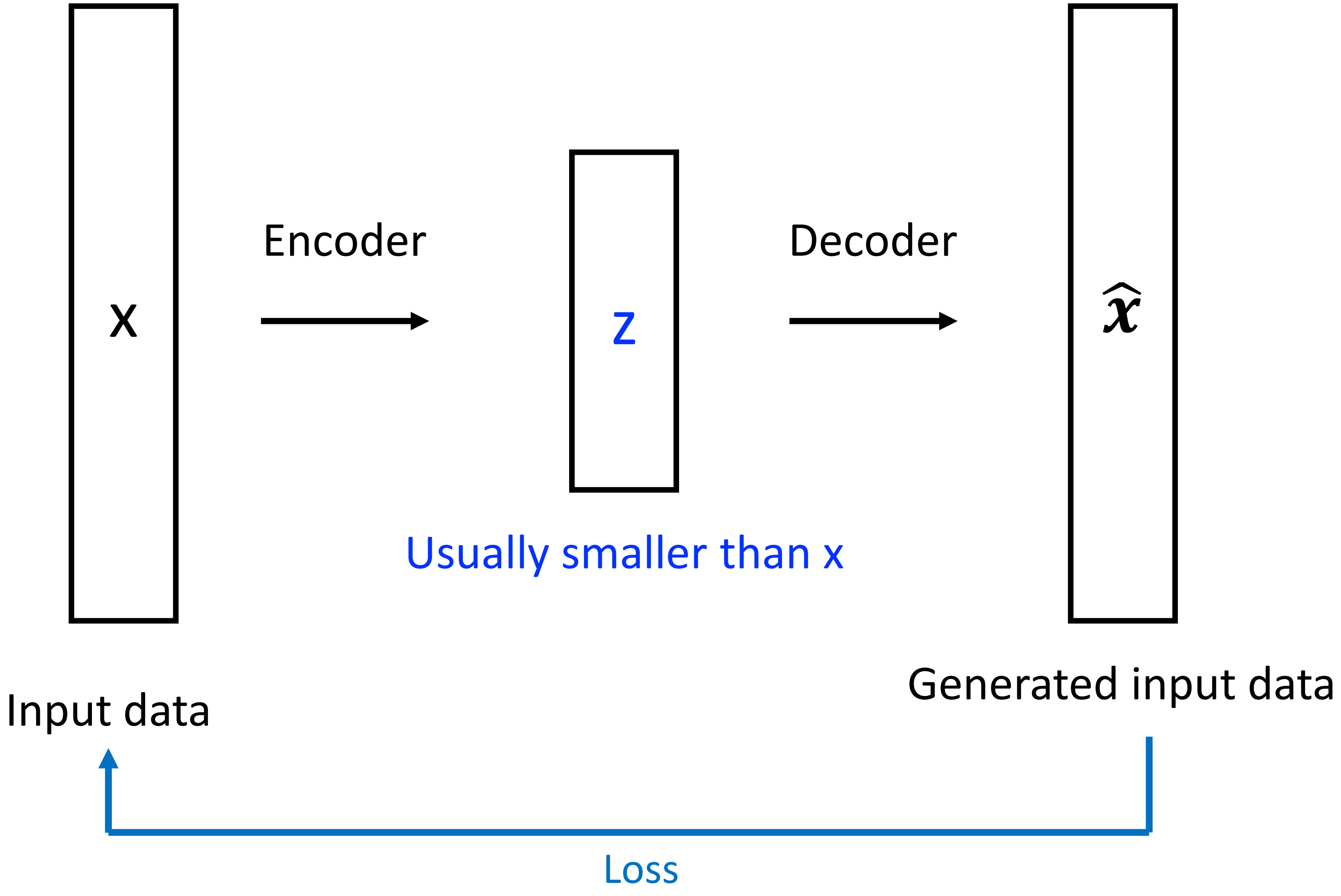
$$p_{model}(x) = \int p_{model}(z) p_{model}(x|z)\, dz$$

$p_{model}(x)$ is controlled by hidden state $z$

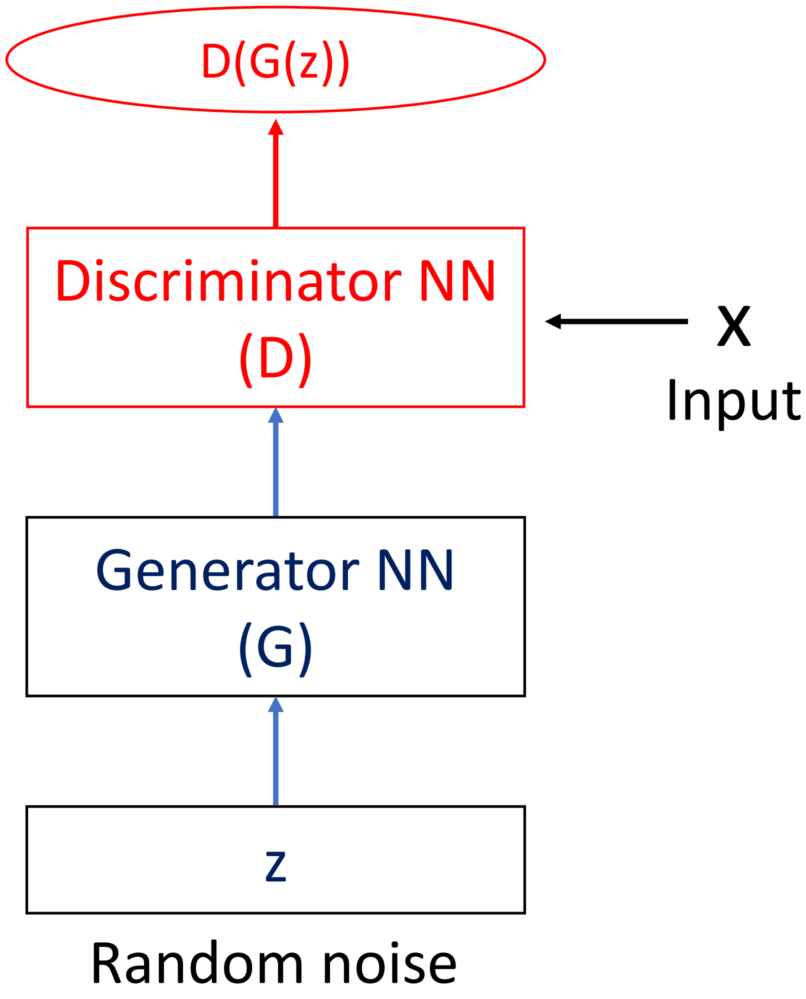
# Variational Autoencoder



# Variational Autoencoder



# GAN




- Instead of sampling from **high dimensional, complex and unknown distribution**
- Sample from **simple distribution**, e.g. normal distribution (random noise) and **find transformation** to the distribution we want to learn.
- **Learn the transformation using a NN**

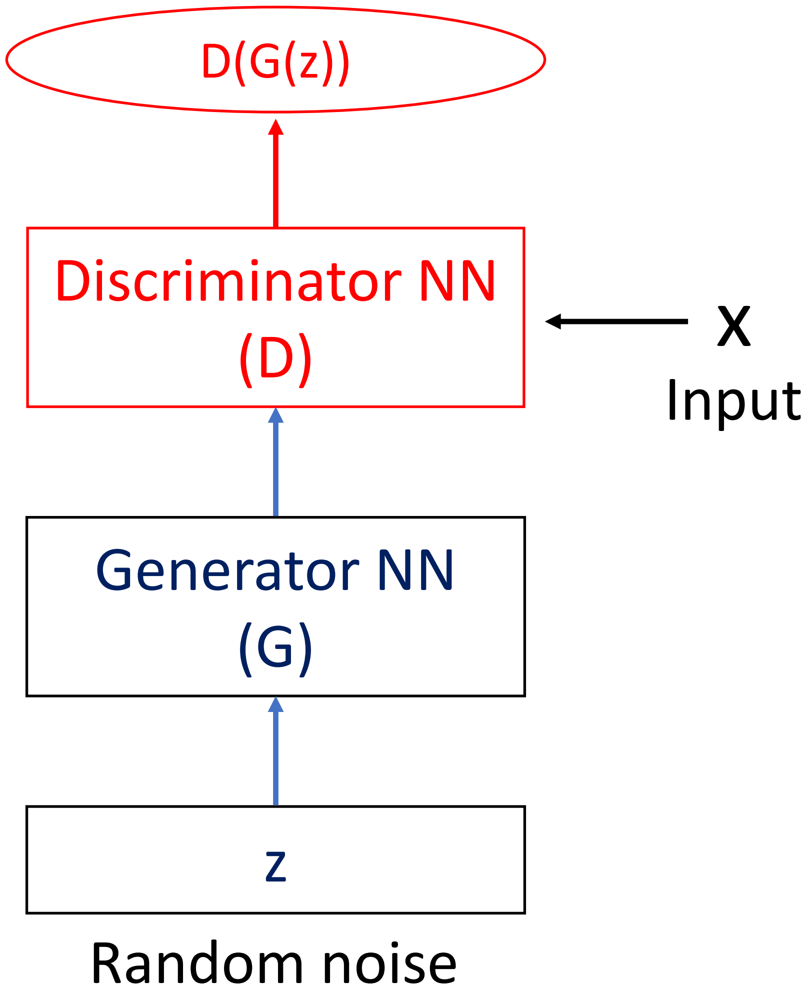
# Generative Adversarial Networks

GAN Architecture

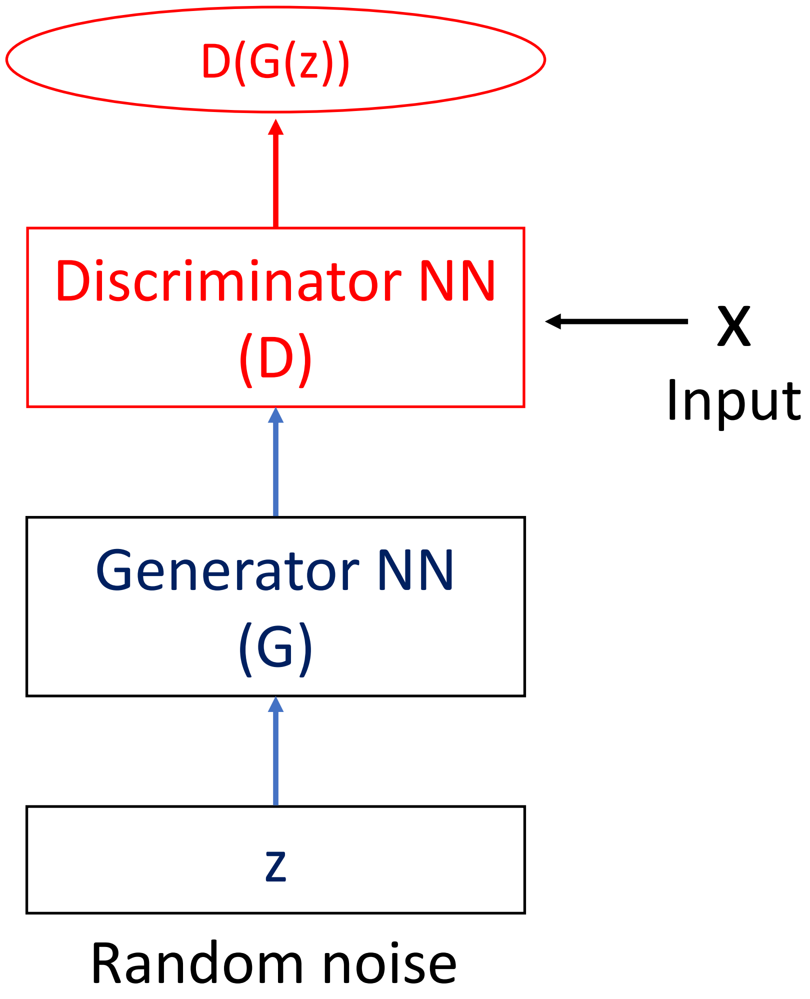
Generator Network

Discriminator Network

# GAN



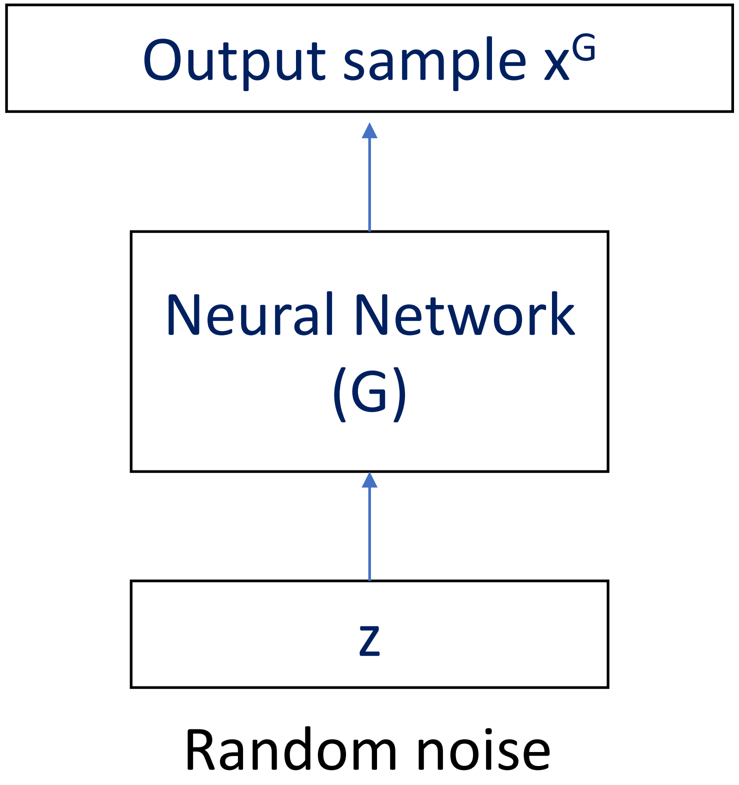
# GAN




Discriminator – try to distinguish between **x (real) and generated (fake)** images

Generator – try to generate **samples and present them as real world** and fool the discriminator

# Generator (G)




Training data has distribution $\mathbf{p}_{\mathbf{data}}$.
Sample $\mathbf{x} \sim \mathbf{p}_{\mathbf{data}}$.

Goal: Output sample $\mathbf{x}^{\mathbf{G}}$ is of similar dimensions as $\mathbf{x}$ and distribution $\mathbf{p}_{\mathbf{data}}$.

# Examples

Face:

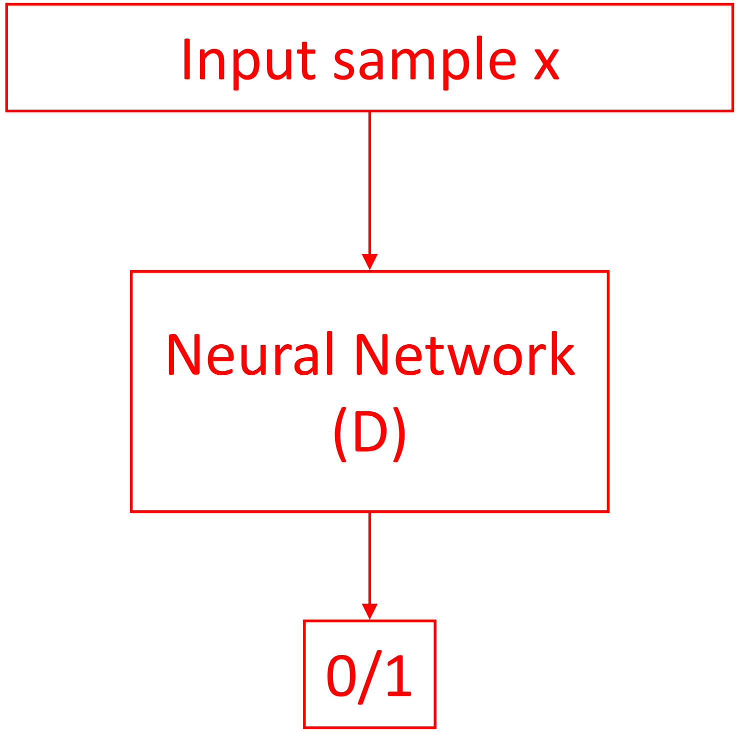
Car:

Bedroom:

# Discriminator (D)

Receives input of same dimensions as $\mathbf{p}_{\mathbf{data}}$.

Goal: Distinguish sample from $\mathbf{p}_{\mathbf{data}}$ (1) or not (0).



# Examples

Discriminator

Face (gen):

0

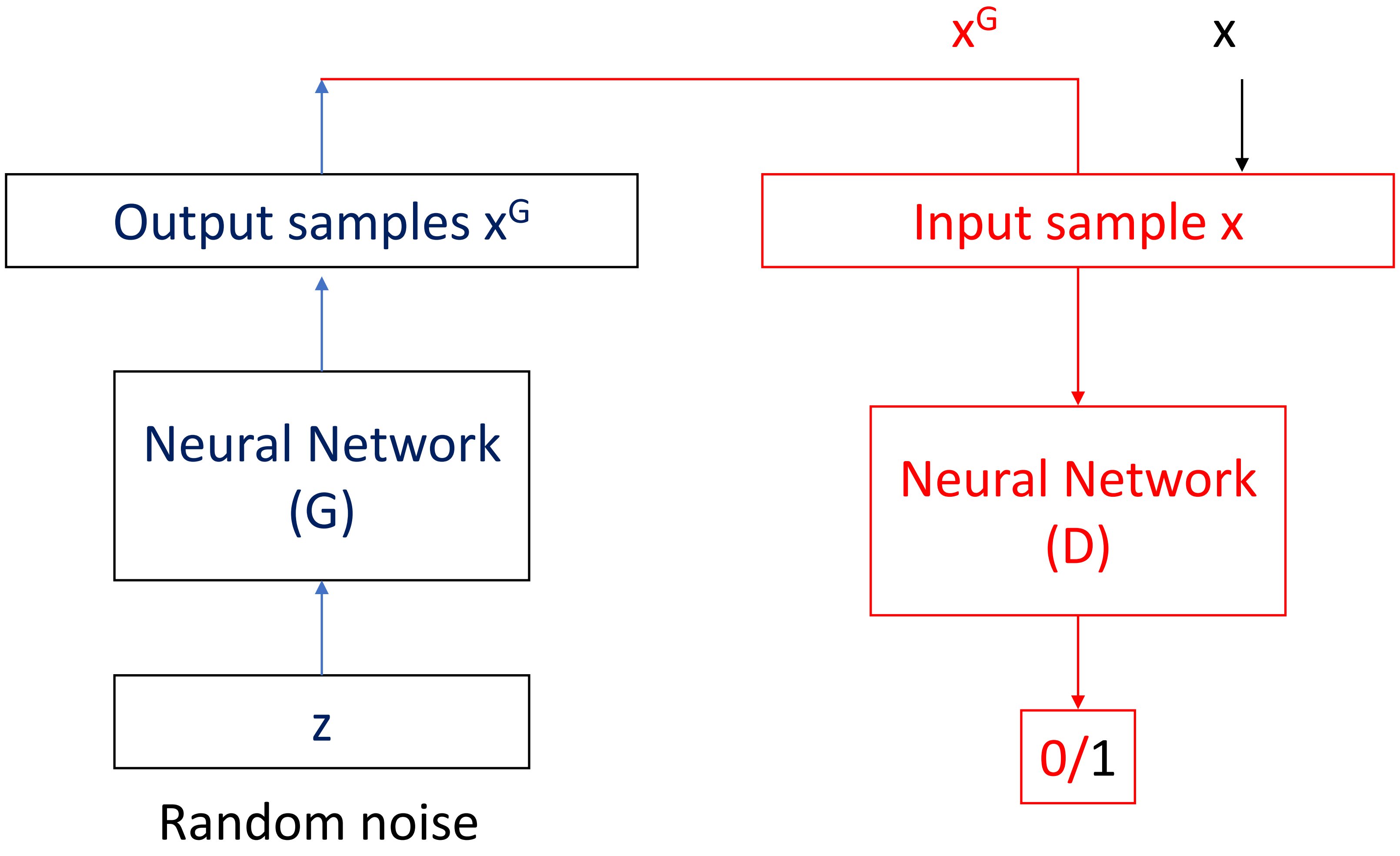
Car (real):

1

Bedroom (gen):

0

# Full Architecture



# GAN Optimization and Applications

Competing Loss Functions

Minmax Game Optimization

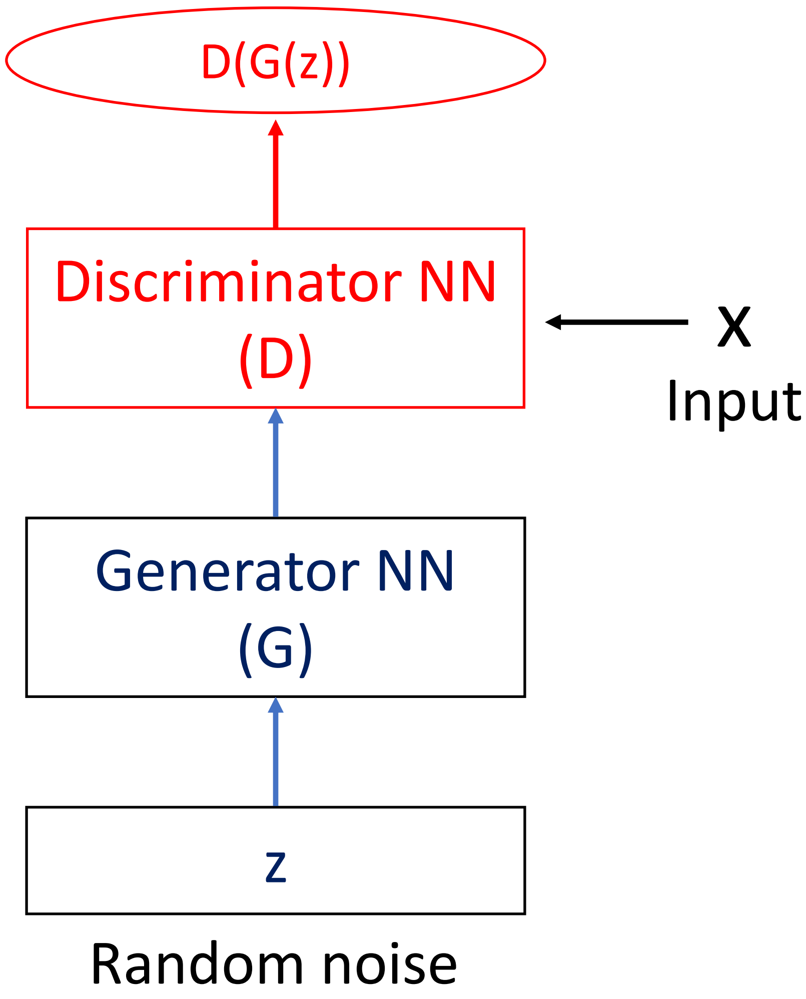
Optimization in NN

GAN Applications

# Competing cost functions

"Cost": $\|D(G(z))\|$

"Cost": $\|D(G(z)) - 1\|$



# Competing cost functions

Binary Cross Entropy Loss

$$J^{(D)} = -\frac{1}{2}\mathbb{E}_{x\sim p_{data}} \log D_{\theta_d}(x) - \frac{1}{2}\mathbb{E}_{z\sim p_{model}} \log\left(1 - D_{\theta_d}(G_{\theta_g}(z))\right)$$

$$J^{(G)} = -J^{(D)}$$

$$J^{(D)} = -\frac{1}{2}\int p_{data}(x)\log D(x)dx - \frac{1}{2}\int p_{model}(x)\log\big(1 - D(x)\big)dx$$

# Competing cost functions

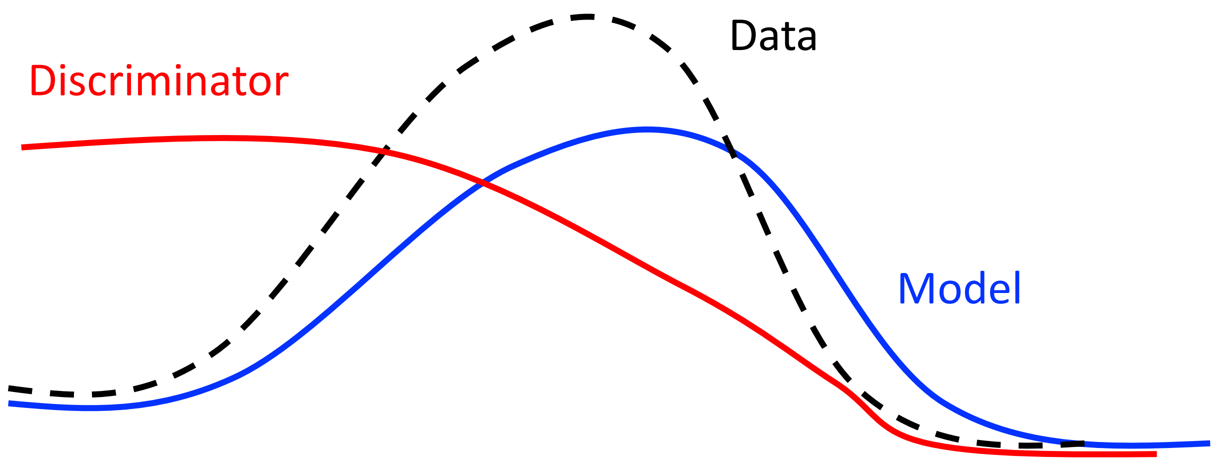
Optimal $D(x)$ is

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

Assumption: $p_{model}, p_{data}$ are nonzero everywhere

Equilibrium: $p_{model} = p_{data}$ then $\mathrm{E}(D(x)) = \frac{1}{2}$

# Competing cost functions




Discriminator learns an approximation of $p_{data}(x)/p_{model}(x)$

vs

learning $p_{model}$ (x) directly (or indirectly via latent variable models).

# Minmax Game Optimization

$$\min_{\theta_g} \max_{\theta_d} \left[ \mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p_{model}} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

Discriminator output for real data

Discriminator output for generated data

Solution:

Saddle point in the parameter space (Nash Equillibrium)

- One player (Discriminator) is at **maximum**,
- Other player (Generator) is at **minimum**

# Optimization in NN

- **Gradient ascent** for the discriminator on J

$$J^{(D)} = \frac{1}{2}\mathbb{E}_{x\sim p_{data}} \log D_{\theta_d}(x) + \frac{1}{2}\mathbb{E}_{z\sim p_{model}} \log\left(1 - D_{\theta_d}(G_{\theta_g}(z))\right)$$

$$\theta_d \leftarrow \arg\min_{\theta_d} J^{(D)}$$

- **Gradient descent** for the generator

$$J^{(G)} = -\frac{1}{2}\mathbb{E}_{z\sim p_{model}} \log\left(1 - D_{\theta_d}(G_{\theta_g}(z))\right)$$

$$\theta_g \leftarrow \arg\min_{\theta_g} J^{(G)}$$

# Optimization in NN

Take ***k*** gradient steps for the discriminator (k a hyperparameter), each doing the following:

- Sample *m* noise samples, $\{z^{(1)},z^{(2)},\ldots,z^{(m)}\}$ from $p_{model}(z)$.
- Sample *m* actual samples, $\{x^{(1)},x^{(2)},\ldots,x^{(m)}\}$ from $p_{data}(x)$: (a minibatch of your input data.)
- Perform an optimization step on the discriminator:

# Optimization in NN

Take ***k*** gradient steps for the discriminator (k a hyperparameter), each doing the following:

- Sample *m* noise samples, $\{z^{(1)},z^{(2)},\ldots,z^{(m)}\}$ from $p_{model}(z)$.
- Sample *m* actual samples, $\{x^{(1)},x^{(2)},\ldots,x^{(m)}\}$ from $p_{data}(x)$: (a minibatch of your input data.)
- Perform an optimization step on the discriminator:

Do gradient descent step for the generator:

- Sample *m* noise samples, $\{z^{(1)},z^{(2)},\ldots,z^{(m)}\}$ from $p_{model}(z)$.
- Perform an optimization step on the **generator**:

# Optimization in NN

For epoch in range(epochs):

For batch in batches:

Compute $D_{\theta_d}(x)$ vs Real labels
Compute $D_{\theta_d}(G_{\theta_g}(z))$ vs Fake labels
Compute $J^{(D)}$

Backpropagation
Update network

**Discriminator Network**

For epoch in range(epochs):

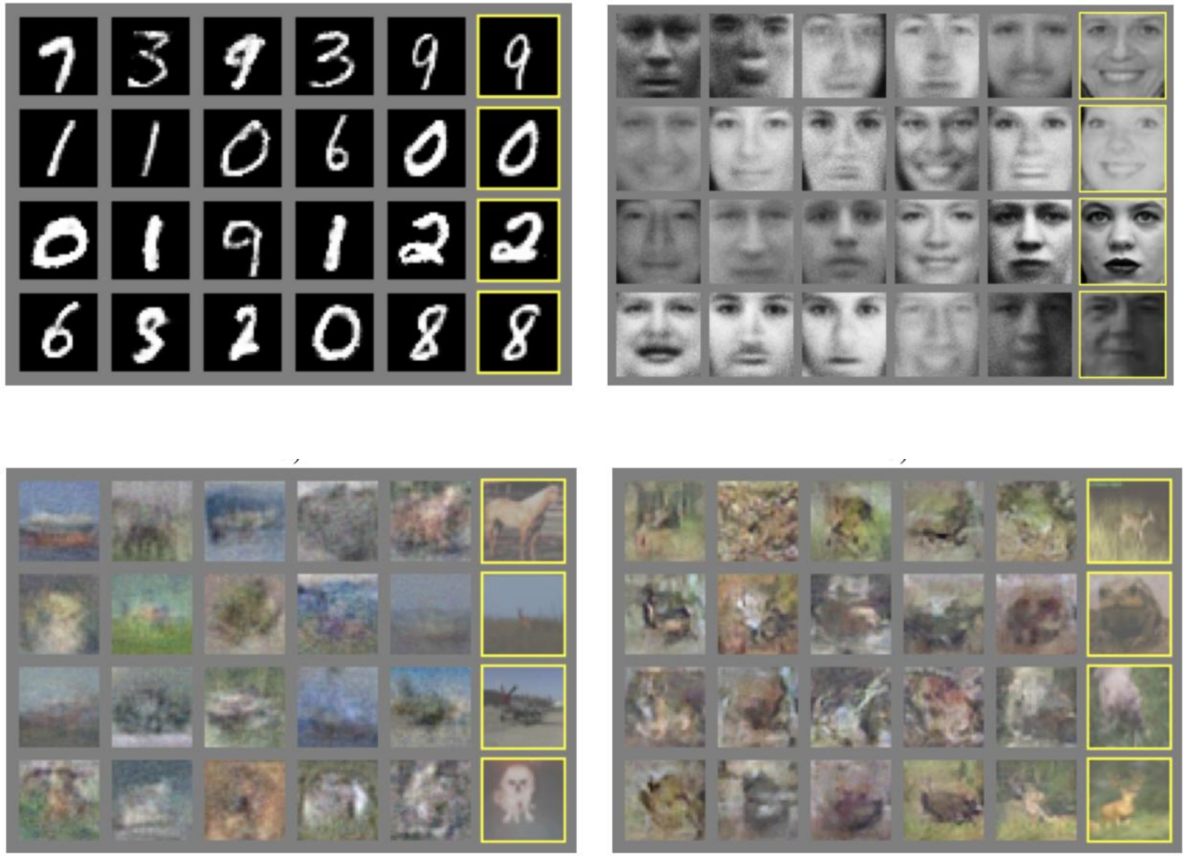
For batch in batches:

Compute $D_{\theta_d}(G_{\theta_g}(z))$ vs Real labels
Compute $J^{(G)}$

Backpropagation
Update network

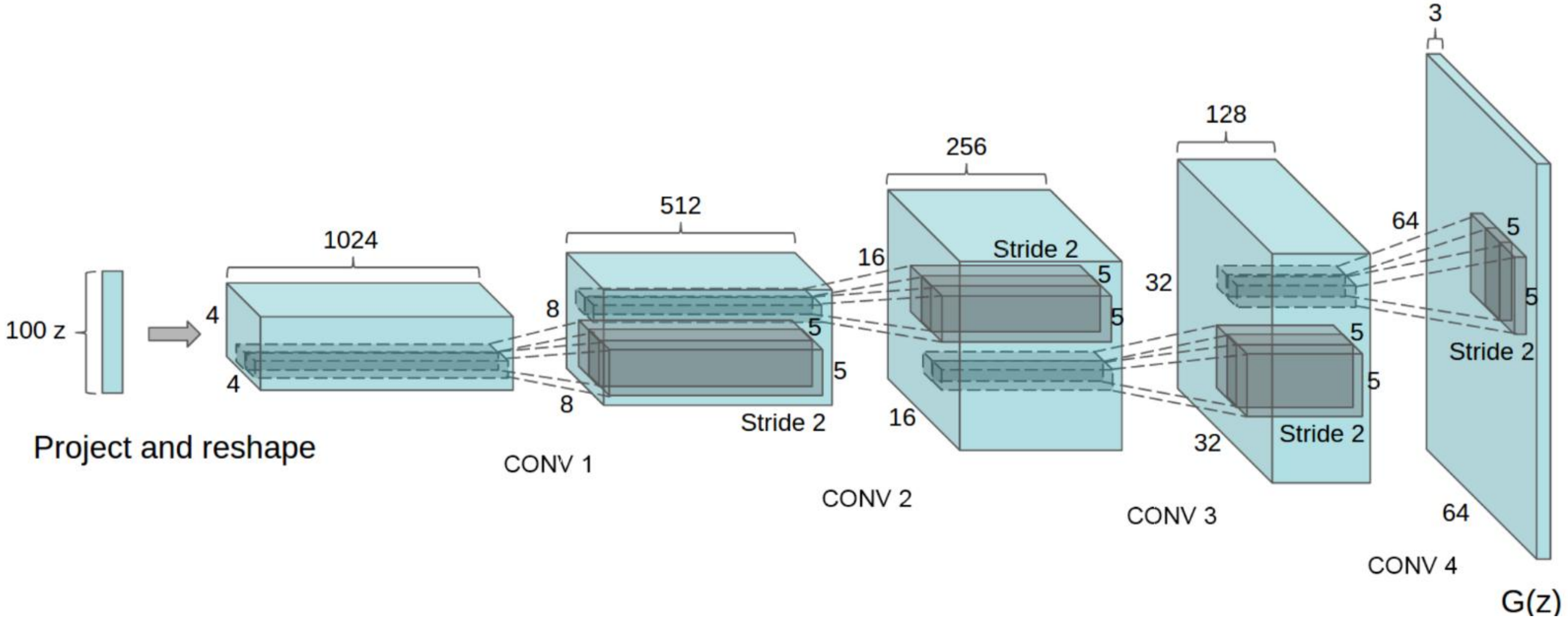
**Generator Network**

# GAN Applications: Original GAN



*Goodfellow et al. (2014) Generative Adversarial Nets*

# DCGAN

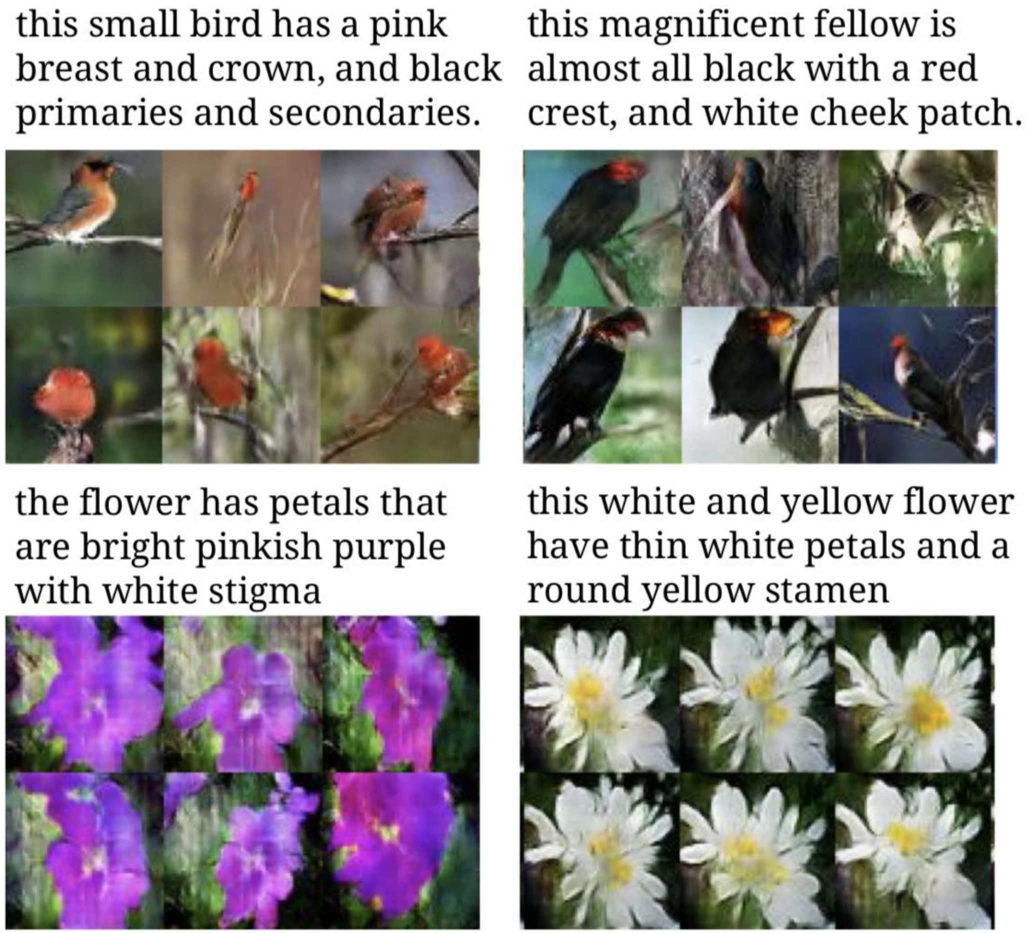



Radford, Alec, Luke Metz, and Soumith Chintala. "Unsupervised representation learning with deep convolutional generative adversarial networks." (2015).

# Similarities in Hidden Space

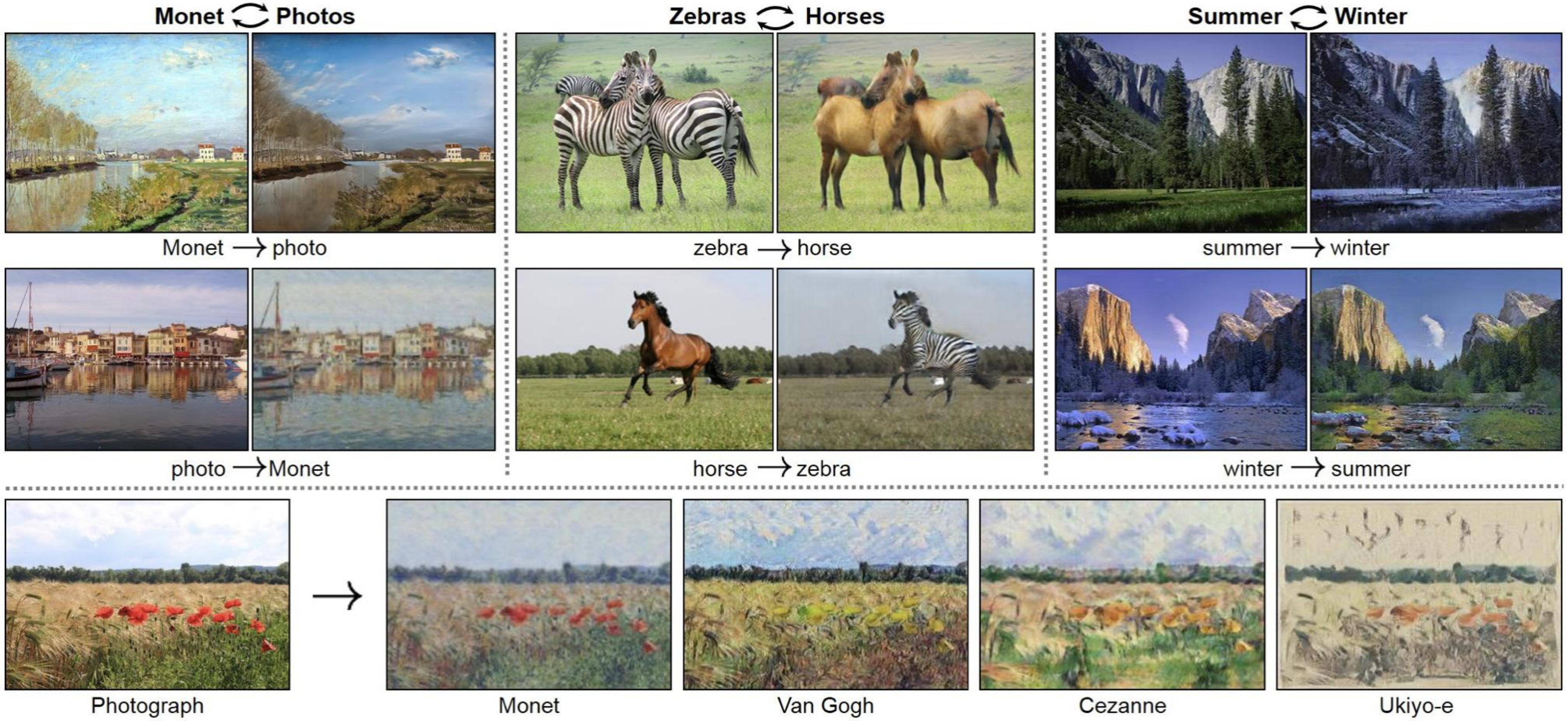

woman with glasses

# Text to Image Synthesis



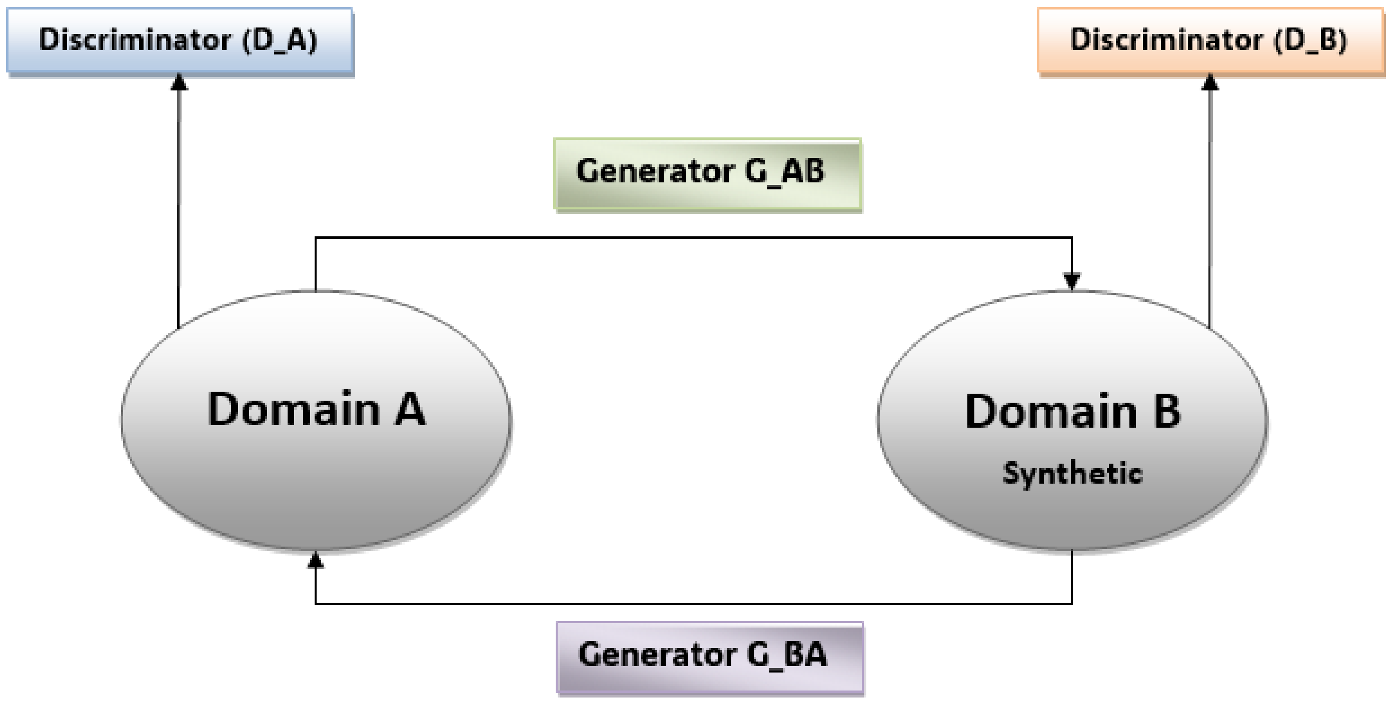

Reed et al. Generative Adversarial Text to Image Synthesis (2017)

# CycleGAN



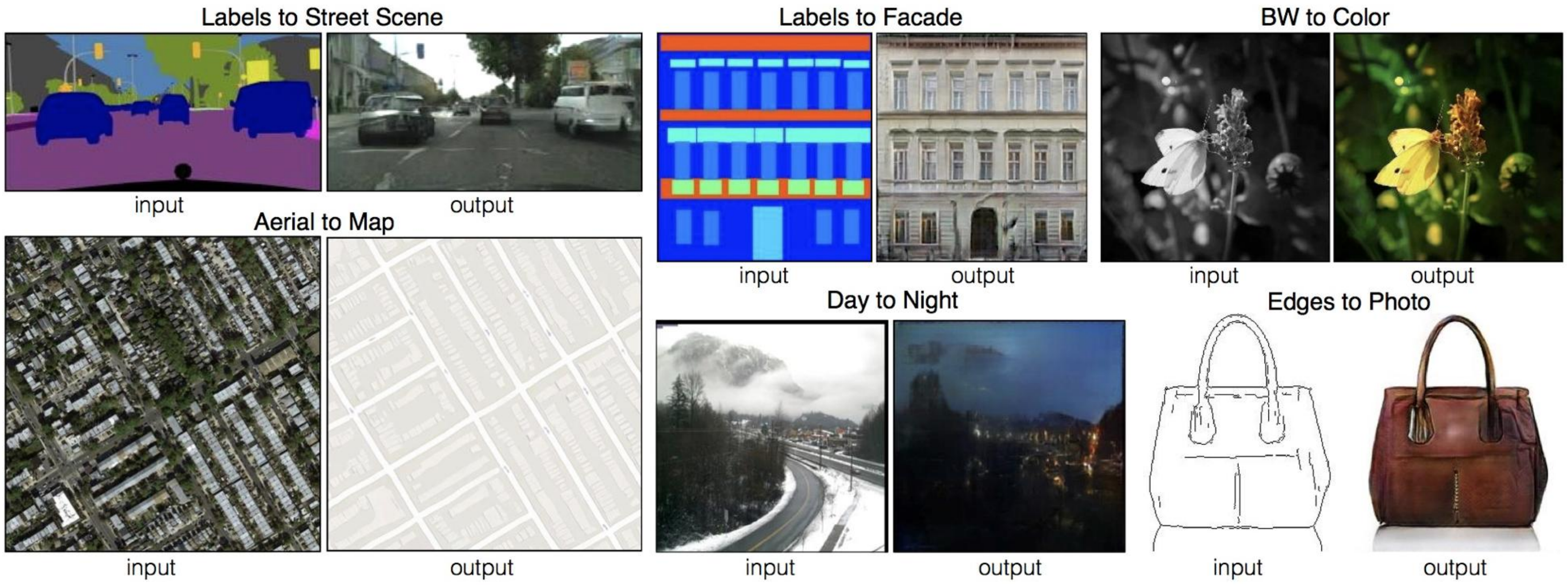

Zhu et al., Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks, ICCV 2017

# CycleGAN



Discriminator (D_A)
Discriminator (D_B)
Generator G_AB
Domain A
Domain B
Synthetic
Generator G_BA

# Pix2Pix

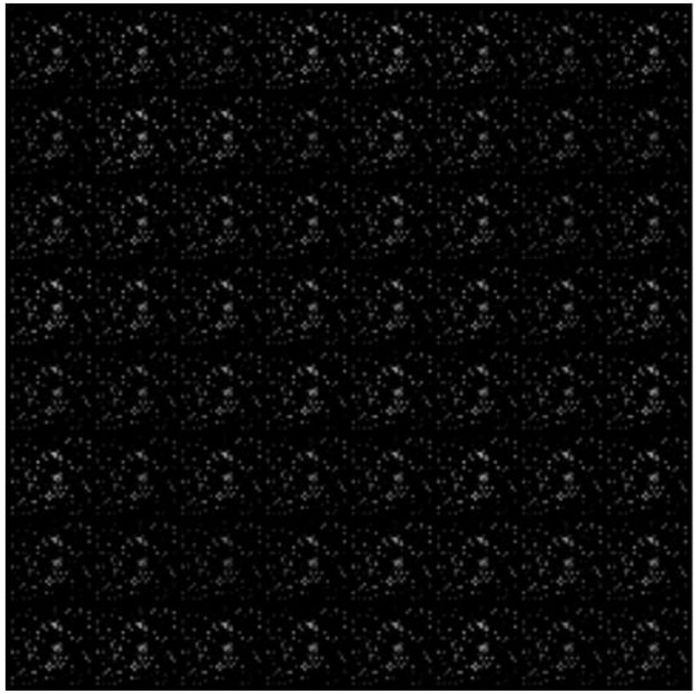



*P. Isola et al. Image-to-Image Translation with Conditional Adversarial Nets, CVPR 2017*

# GAN Example:

## MNIST Generation with Vanilla GAN

# MNIST Generation with GAN



**Before Training**

**After Training**

# Prepare Data

```
from torchvision.datasets import MNIST
from torch.utils.data import DataLoader
from torchvision import transforms

# Define a transformation to convert the data into Tensors
train_transforms = transforms.Compose([transforms.ToTensor()])

# Download the train and test MNIST data and transform it into Tensors
train_data = MNIST(root="./train.", train=True, download=True, transform=train_transforms)
```

Load MNIST dataset and in built DataLoader from torch

Configure the data transformation (to PyTorch tensors)

Download the training and testing data
N = 60000

# Define Model (Generator)

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

    def __init__(self, batchsize, input_noise_dim):

        super(Generator, self).__init__()

        self.batchsize = batchsize  # Batch size for input data
        self.input_noise_dim = input_noise_dim  # Dimension of the input data

        self.fc1 = torch.nn.Linear(input_noise_dim, 128)  # Fully connected layer 1
        self.LReLU = torch.nn.LeakyReLU()  # Leaky ReLU activation function
        self.fc2 = torch.nn.Linear(128, 1 * 28 * 28)  # Fully connected layer 2
        self.output = torch.nn.Tanh()  # Hyperbolic Tangent activation function

    def forward(self, x):

        layer1 = self.LReLU(self.fc1(x))  # Apply Leaky ReLU to the first fully connected layer
        layer2 = self.output(self.fc2(layer1))  # Apply Tanh to the second fully connected layer
        out = layer2.view(self.batchsize, 1, 28, 28)  # Reshape the output to match image dimensions

        return out
```

Takes batchsize and input noise dimension as inputs

Define FC1, FC2 layers
Uses LeakyReLU() as hidden layer activation
Uses Tanh() as output layer activation

Define signal propagation
input noise -> FC1 -> FC2 -> Output

# Define Model (Discriminator)

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

    def __init__(self, batchsize):

        super(Discriminator, self).__init__()

        self.batchsize = batchsize  # Batch size for input data

        self.fc1 = torch.nn.Linear(1 * 28 * 28, 128)  # Fully connected layer 1
        self.LReLU = torch.nn.LeakyReLU()  # Leaky ReLU activation function
        self.fc2 = torch.nn.Linear(128, 1)  # Fully connected layer 2
        self.output = torch.nn.Sigmoid()  # Sigmoid activation function

    # Function for forward propagation
    def forward(self, x):

        flat = x.view(self.batchsize, -1)  # Flatten the input image
        layer1 = self.LReLU(self.fc1(flat))  # Apply Leaky ReLU to the first fully connected layer
        out = self.output(self.fc2(layer1))  # Apply Sigmoid to the second fully connected layer

        return out.view(-1, 1).squeeze(1)  # Flatten the output and remove unnecessary dimension
```

Takes batchsize as inputs

Define FC1, FC2 layers
Uses LeakyReLU() as hidden layer activation
Uses sigmoid() as output layer activation

Define signal propagation
input image -> FC1 -> FC2 -> (0/1)

Use .squeeze() to reduce output to 0/1

# Define Hyperparameters

```
# Fix random seed
torch.manual_seed(55)

# Define learning rate + epochs
learning_rate = 0.001
epochs = 5

# Define batch size and num_features/timestep (this is simply the last dimension of train_output_seqs)
batchsize = 128
input_noise_dim = 100

# Create a Discriminator model
disc = Discriminator(batchsize)
gen = Generator(batchsize, input_noise_dim)

# Binary Cross Entropy (BCE) loss function
loss_func = torch.nn.BCELoss()
optimizer_disc = torch.optim.Adam(disc.parameters(), lr=learning_rate, weight_decay=1e-05)
optimizer_gen = torch.optim.Adam(gen.parameters(), lr=learning_rate, weight_decay=1e-05)

# Determine the device for training (GPU if available, otherwise CPU)
device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
disc.to(device)
gen.to(device)
```

Define learning rate and epoch number

Define batchsize and input noise dimensions

Define Discriminator and Generator networks

Using Binary Cross-Entropy loss and Adam Optimizer with L2 regularization

Device for training (GPU/CPU)

# Identify Tracked Values

```
gen_train_loss_list = []
disc_train_loss_list = []
```

Lists for storing generator/discriminator training loss

# Train Model

```python
# Create DataLoader objects to efficiently load the training and test data in batches
train_loader = DataLoader(train_data, batch_size=batchsize, shuffle=False, drop_last=True)
```

```python
# Set the device as CUDA or CPU based on availability
if torch.cuda.is_available():
    device = torch.device("cuda")
else:
    torch.device("cpu")

# Run training for each epoch
for epoch in range(epochs):

    print('Epoch {}/{}'.format(epoch + 1, epochs))
    running_loss_D = 0
    running_loss_G = 0

    for inputs, labels in train_loader:

        inputs = inputs.to(device)

        # Convert labels into torch tensors with the proper size as per the batch size
        real_label = torch.full((batchsize,), 1, dtype=inputs.dtype, device=device)
        fake_label = torch.full((batchsize,), 0, dtype=inputs.dtype, device=device)
```

Define the data loader for training and testing

Define device for training

Initialize Discriminator/Generator loss for given epoch

Create labels for real (1) vs fake (0) data

# Train Model (Discriminator)

```
# Zero the gradients of the Discriminator optimizer
optimizer_disc.zero_grad()

# Compute output from the Discriminator
output = disc(inputs)

# Discriminator real loss
D_real_loss = loss_func(output, real_label)
D_real_loss.backward()

# Generate random noise data as input to the Generator
noise = torch.randn(batchsize, input_noise_dim, device=device)

# Generate fake images using the Generator
fake = gen(noise)

# Pass fake images through the Discriminator with gradient detachment
output = disc(fake.detach())

# Discriminator fake loss
D_fake_loss = loss_func(output, fake_label)
D_fake_loss.backward()

# Total loss for the Discriminator
Disc_loss = D_real_loss + D_fake_loss
running_loss_D += Disc_loss

# Update Discriminator's parameters
optimizer_disc.step()
```

Clear the gradient for discriminator network

Compute the discriminator outputs for given real data inputs

Compute the loss with respect to real labels (1)
Perform back propagation

Initialize noise to be fed into generator network

Compute the fake images from the generator

Feed the generated images to discriminator and get discriminator outputs

Compute the loss with respect to fake labels (0)
Perform back propagation

Final discriminator loss is **loss from real labels + fake labels**

Update discriminator network

# Train Model (Generator)

```
# Zero the gradients of the Generator optimizer
optimizer_gen.zero_grad()

# Pass fake images obtained from the Generator to the Discriminator
output = disc(fake)

# Calculate Generator loss by giving fake images as input but providing real labels
Gen_loss = loss_func(output, real_label)
running_loss_G += Gen_loss

# Backpropagation for the Generator
Gen_loss.backward()

# Update Generator's parameters
optimizer_gen.step()

disc_train_loss_list.append(Disc_loss.item())
gen_train_loss_list.append(Gen_loss.item())
```

Clear the gradient for generator network

Feed the generated image to discriminator

Compute complementary discriminator loss with respect to **real labels**

Perform back-propagation

Update generator network

Store the loss the respective lists

# Visualize & Evaluate Model

```python
import matplotlib.pyplot as plt

# Function to plot an image
def show_image(img):
    # Convert the image from a tensor to a NumPy array
    npimg = img.numpy()
    # Transpose the NumPy array to the correct format for displaying
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
```

Define the function for converting torch image array to numpy array for plotting

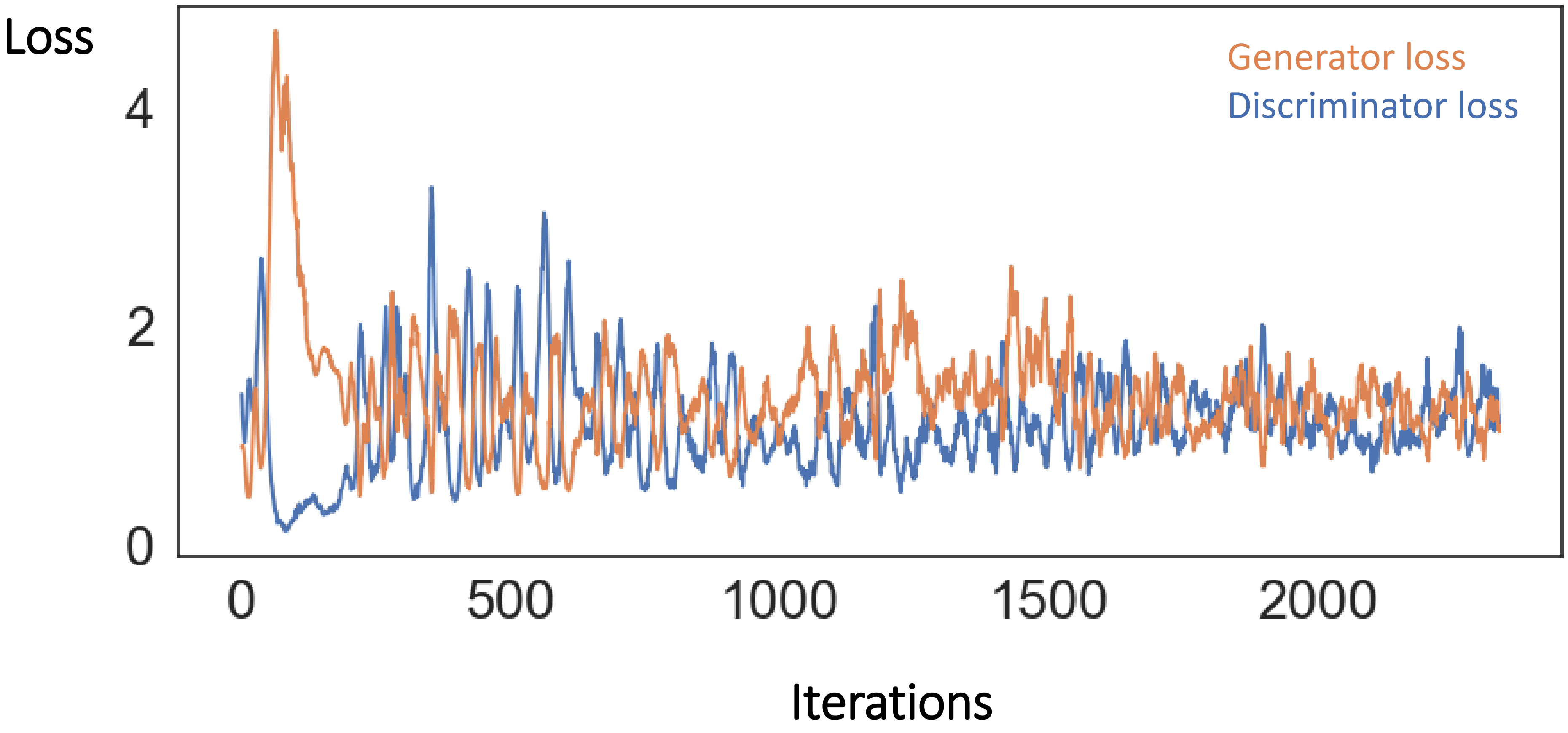
```python
import torchvision

# Generate random noise for generating fake images
random_noise = torch.randn(128, input_noise_dim, device=device)

# Generate fake images from the random noise using the Generator
fake = gen(random_noise)
fake = fake.cpu()  # Move the generated fake images to the CPU for displaying

# Create a Matplotlib figure and axis for displaying the fake images
fig, ax = plt.subplots(figsize=(20, 8.5))

# Display the fake images in a grid (e.g., 10x5 grid)
show_image(torchvision.utils.make_grid(fake[0:50], 10, 5))

plt.show()
```

Define the random noise to be fed to generator for testing purpose

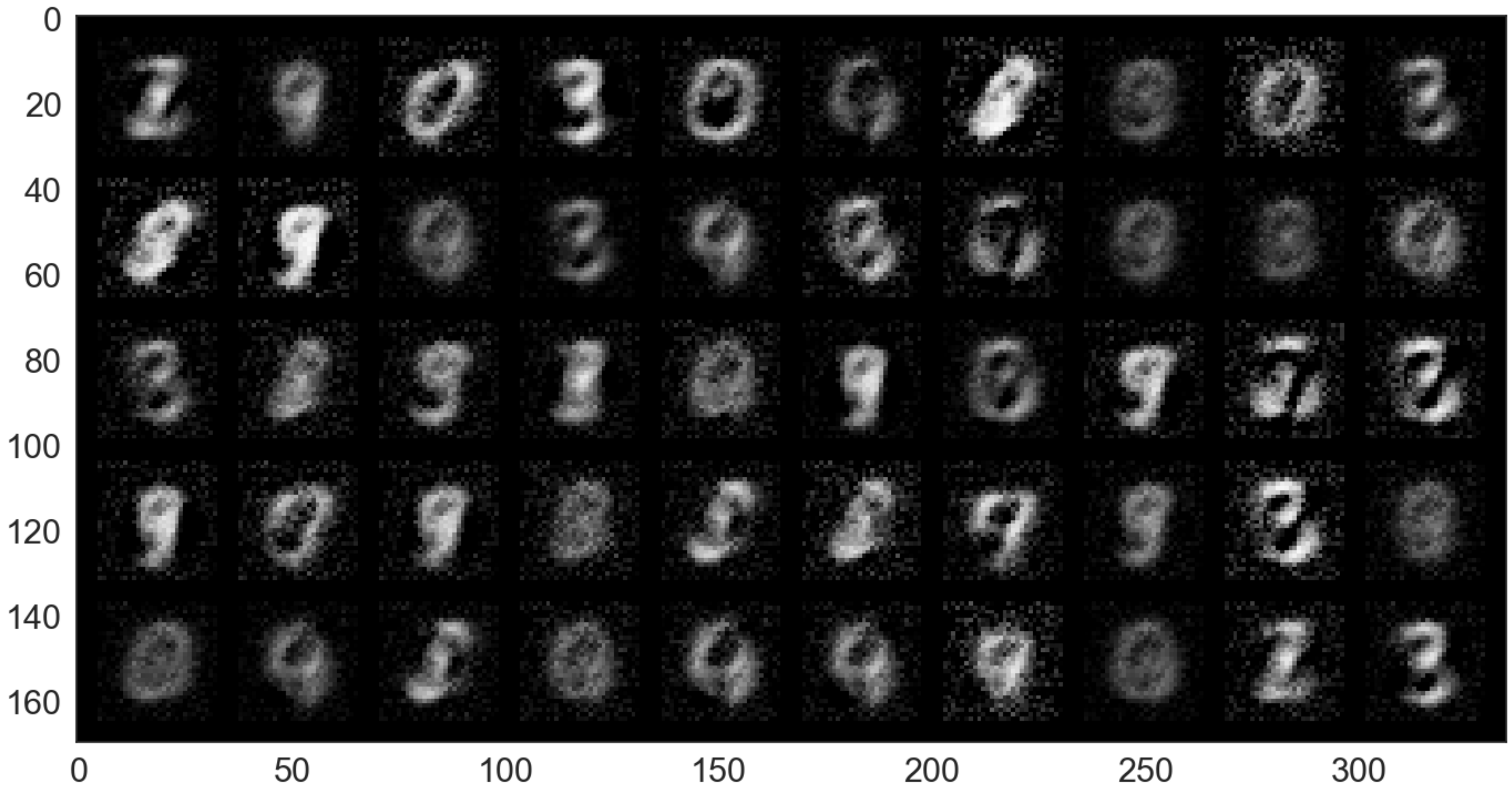
Feed the noise the generator to produce outputs and move them to cpu

Plot the first 50 generated images

# Visualize & Evaluate Model



# Visualize & Evaluate Model

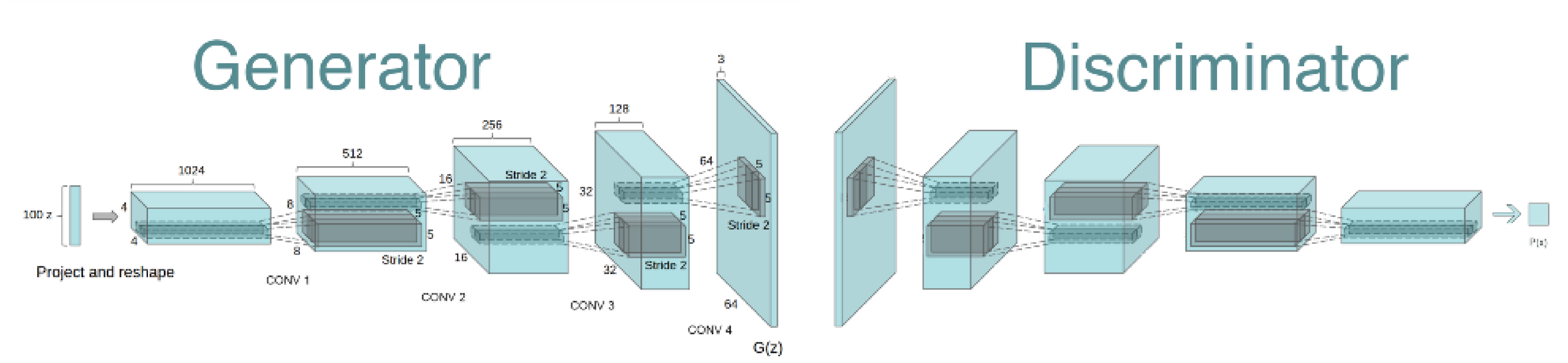



Generated samples

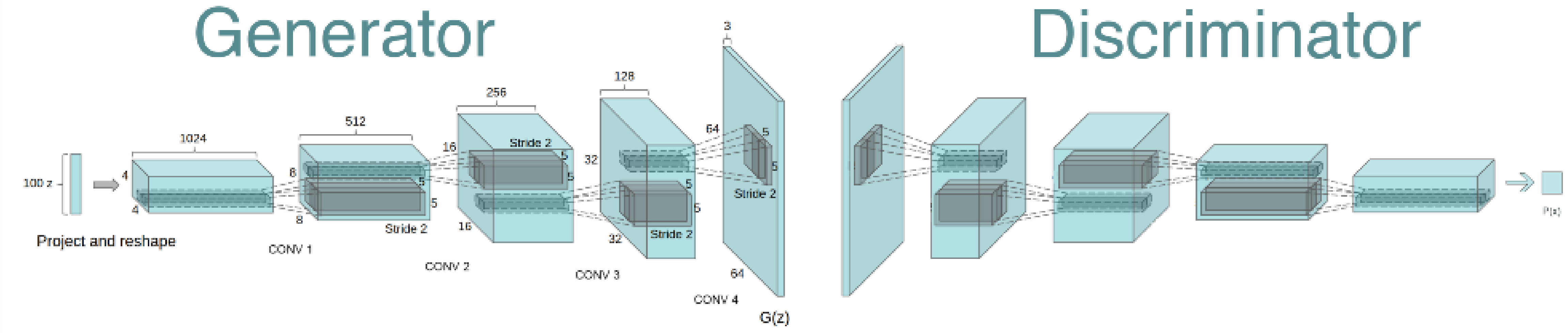
# Generator Extension with Convolution:

## 2D Transpose Convolution

# DCGAN Architecture



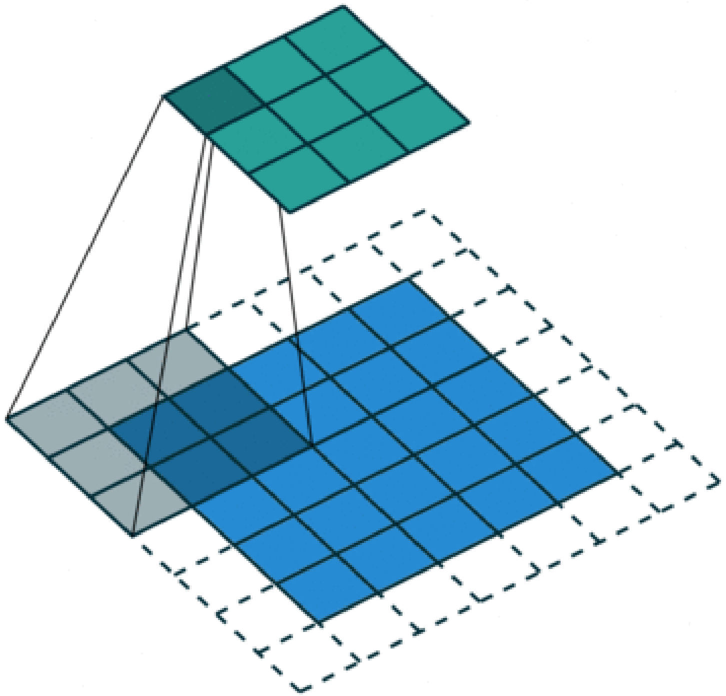
# DCGAN Architecture



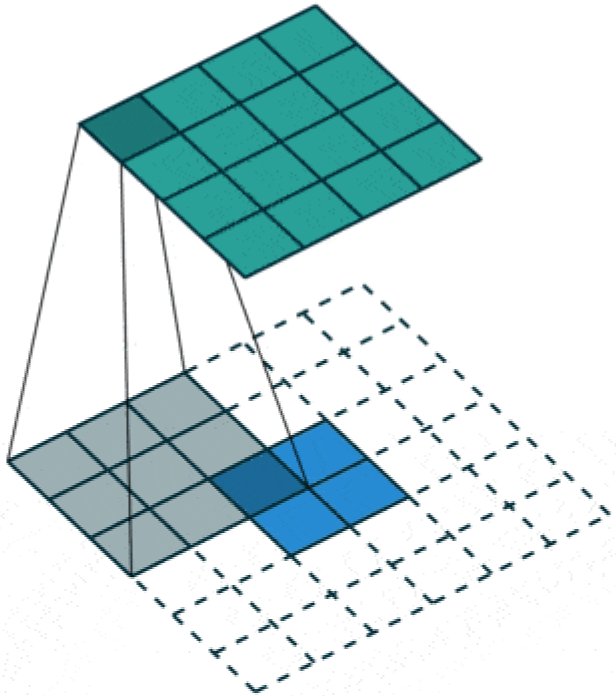

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
   ConvTranspose2d-1            [-1, 512, 4, 4]         819,200
       BatchNorm2d-2            [-1, 512, 4, 4]           1,024
              ReLU-3            [-1, 512, 4, 4]               0
   ConvTranspose2d-4            [-1, 256, 8, 8]       2,097,152
       BatchNorm2d-5            [-1, 256, 8, 8]             512
              ReLU-6            [-1, 256, 8, 8]               0
   ConvTranspose2d-7          [-1, 128, 16, 16]         524,288
       BatchNorm2d-8          [-1, 128, 16, 16]             256
              ReLU-9          [-1, 128, 16, 16]               0
  ConvTranspose2d-10           [-1, 64, 32, 32]         131,072
      BatchNorm2d-11           [-1, 64, 32, 32]             128
             ReLU-12           [-1, 64, 32, 32]               0
  ConvTranspose2d-13            [-1, 3, 64, 64]           3,072
             Tanh-14            [-1, 3, 64, 64]               0
----------------------------------------------------------------
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 64, 32, 32]           3,072
         LeakyReLU-2           [-1, 64, 32, 32]               0
            Conv2d-3          [-1, 128, 16, 16]         131,072
       BatchNorm2d-4          [-1, 128, 16, 16]             256
         LeakyReLU-5          [-1, 128, 16, 16]               0
            Conv2d-6            [-1, 256, 8, 8]         524,288
       BatchNorm2d-7            [-1, 256, 8, 8]             512
         LeakyReLU-8            [-1, 256, 8, 8]               0
            Conv2d-9            [-1, 512, 4, 4]       2,097,152
      BatchNorm2d-10            [-1, 512, 4, 4]           1,024
        LeakyReLU-11            [-1, 512, 4, 4]               0
           Conv2d-12              [-1, 1, 1, 1]           8,192
          Sigmoid-13              [-1, 1, 1, 1]               0
          Flatten-14                    [-1, 1]               0
================================================================
```

# Conv2D vs ConvTranspose2D

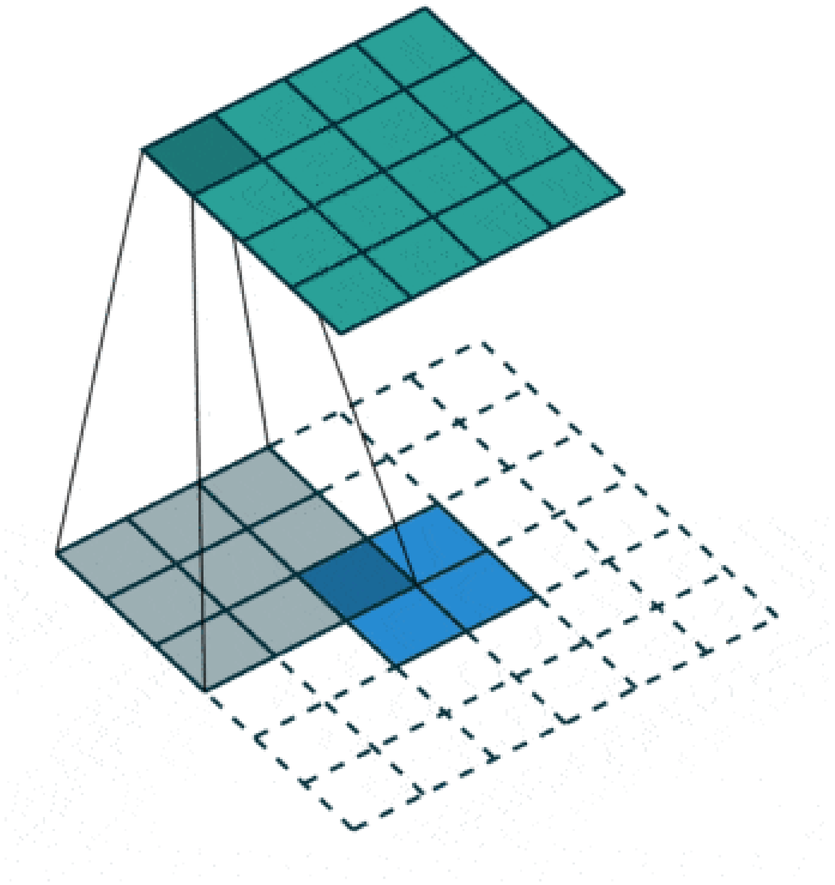


**Conv2D()**
input image = (5, 5)
Kernel size = (3, 3)
Output image = (3, 3)



**ConvTranspose2D()**
input image = (2, 2)
Kernel size = (3, 3)
Output image = (4, 4)

# Conv2D vs ConvTranspose2D



torch.nn.ConvTranspose2d(

| | |
|---|---|
| in_channels | # of channels of input |
| out_channels | # of channels of output |
| kernel_size | Size of the convolving Filter |
| stride | Stride of the convolution |
| Padding | Padding added to input |

)

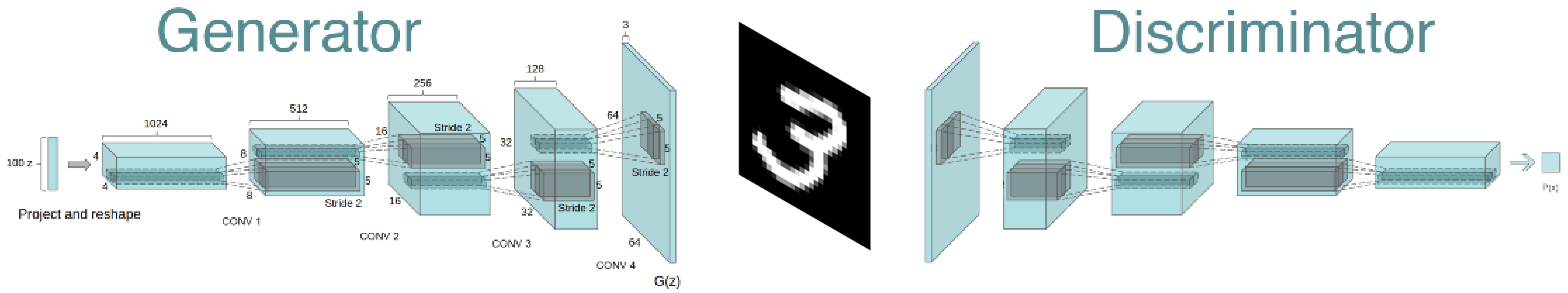
$$H\_out = (H\_in - 1)*\text{stride} - 2*\text{padding} + 1*(\text{kernel\_size} - 1) + 1$$

$$W\_out = (W\_in - 1)*\text{stride} - 2*\text{padding} + 1*(\text{kernel\_size} - 1) + 1$$

# LAB 8 ASSIGNMENT:

MNIST Generation with DCGAN

# MNIST Generation with DCGAN




In this exercise, you will use DCGAN architecture to generate **MNIST hand-written images.**

You are free to design architectures for Generator and Discriminator such as # of convolution layers, activation functions, regularization techniques etc.

You are also free to pick your own hyperparameters e.g., total epochs, batch size, learning rate, optimizer etc

Make sure to use **ConvTranspose2D()** for Generator instead of Conv2d().

After training, plot **training loss for both generator and discriminator** as well as the **50 best generated samples** similar to example task. Comment on how their qualities different from Vanilla-GAN - Are they better quality?