FIT3181/5215 Deep Learning

Generative Adversarial Networks and Diffusion Models

Trung Le and Teaching team

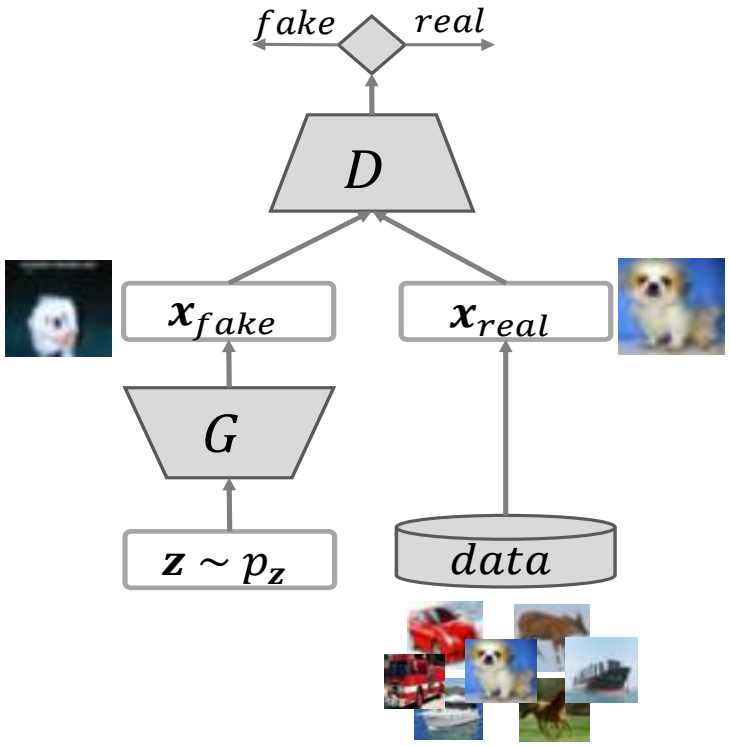
Department of Data Science and AI
Faculty of Information Technology, Monash University
Email: **trunglm@monash.edu**

Outline

- Tutorial 11a
  - Overview of GANs (*****)
  - Implementation of GANs (*****)
  - Implementation of WGAN with gradient penalty (***)
- Tutorial 11b
  - Diffusion models (***)

Overview of GANs

- Min-max game between two players
  - $\min_G \max_D J(G,D) = \mathbb{E}_{\boldsymbol{x} \sim p_d(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim p(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z}))]$
  - **The discriminator** D tries to distinguish real and fake samples, while **the generator** G tries to fool D by generating fake samples indistinguishable from real samples
  - $D(x) \in [0,1]$ is the probability that x is a real sample
- Issues and challenges
  - Mode collapsing problems, generating unrealistic images for complex dataset, no single loss to train
- WS GAN with penalty
  - $\min_G \max_D \left(\mathbb{E}_{\boldsymbol{x} \sim p_d}[D(\boldsymbol{x})] - \mathbb{E}_{\boldsymbol{z} \sim p_z}[D(G(\boldsymbol{z}))] - \alpha \mathbb{E}_{\hat{\boldsymbol{x}} \sim \hat{p}}[(\|\nabla_{\hat{\boldsymbol{x}}} D(\hat{\boldsymbol{x}})\| - 1)^2]\right)$
  - $D(x) \in \mathbb{R}$ where D is called '**critic**'



Implementation of GANs

```python
class GAN(nn.Module):
    def __init__(self, data_loader=None, batch_size=32, epochs=10, optimizer=None,
                 noise_dim=30, device='cpu'):
        super(GAN, self).__init__()
        self.batch_size = batch_size  # Batch size for training
        self.epochs = epochs  # Number of training epochs
        self.optimizer = optimizer  # Optimizer class for training
        self.noise_dim = noise_dim  # Dimension of noise vector
        self.device = device  # Device to run the model (CPU/GPU)
        self.generator = None  # Placeholder for generator model
        self.discriminator = None  # Placeholder for discriminator model
        self.gen_optimizer = None  # Placeholder for generator optimizer
        self.dis_optimizer = None  # Placeholder for discriminator optimizer
        self.data_loader = data_loader  # Data loader for training dataset
```

Abstract GAN class

StandardGAN class

```python
class StandardGAN(GAN):
    def __init__(self, data_loader=None, batch_size=32, epochs=10, optimizer=None, noise_dim=30, device='cpu'):
        super(StandardGAN, self).__init__(data_loader, batch_size, epochs, optimizer, noise_dim, device)

    def build(self):
        # Generator network definition
        self.generator = nn.Sequential(
            nn.Linear(self.noise_dim, 256),  # Maps noise to 256 dimensions
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(256, 512),  # Upsamples to 512 dimensions
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(512, 1024),  # Upsamples to 1024 dimensions
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(1024, 784),  # Outputs 784 dimensions (28x28 images)
            nn.Tanh(),
            nn.Unflatten(1, torch.Size([1, 28, 28]))  # Reshapes to image format
        ).to(self.device)

        # Discriminator network definition
        self.discriminator = nn.Sequential(
            nn.Flatten(),  # Flattens input images
            nn.Linear(784, 1024),  # Processes 784-dimensional input
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.3),
            nn.Linear(1024, 512),  # Reduces to 512 dimensions
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.3),
            nn.Linear(512, 256),  # Further reduces to 256 dimensions
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.3),
            nn.Linear(256, 128),  # Reduces to 128 dimensions
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.3),
            nn.Linear(128, 1),  # Outputs a single score (real or fake)
            nn.Sigmoid()  # Probability output
        ).to(self.device)
```

```python
def train(self):
    self.history = {'disc-loss': [], 'gen-loss': []}  # Track losses
    criterion = nn.BCELoss()  # Binary Cross Entropy loss

    for epoch in range(self.epochs):
        dis_losses = []  # List to store discriminator losses
        gen_losses = []  # List to store generator losses
        for batch_idx, (X_batch, y_batch) in enumerate(self.data_loader):
            X_batch = X_batch.to(self.device)  # Move batch to device

            # PHASE 1 - Train Discriminator
            z = torch.randn(X_batch.size(0), self.noise_dim, device=self.device)  # Random noise
            X_gen = self.generator(z)  # Generate fake images

            X_fake_and_real = torch.cat([X_gen, X_batch], dim=0)  # Combine real and fake images

            y1 = torch.cat([torch.zeros(X_gen.size(0), 1), torch.ones(X_batch.size(0), 1)], dim=0).to(self.device)

            self.discriminator.train()  # Set discriminator to training mode
            self.dis_optimizer.zero_grad()  # Zero gradients
            dis_loss = criterion(self.discriminator(X_fake_and_real), y1)  # Compute discriminator loss
            dis_loss.backward()  # Backpropagation
            self.dis_optimizer.step()  # Update discriminator weights
            dis_losses.append(dis_loss.item())  # Store loss

            # PHASE 2 - Train Generator
            z = torch.randn(self.batch_size, self.noise_dim, device=self.device)  # New random noise
            y2 = torch.ones(self.batch_size, 1, device=self.device)  # Labels for generator

            self.discriminator.eval()  # Freeze discriminator weights
            self.gen_optimizer.zero_grad()  # Zero gradients
            gen_loss = criterion(self.discriminator(self.generator(z)), y2)  # Compute generator loss
            gen_loss.backward()  # Backpropagation
            self.gen_optimizer.step()  # Update generator weights
            gen_losses.append(gen_loss.item())  # Store loss
```

```python
class DCGAN(GAN):
    def __init__(self, data_loader=None, batch_size=32, epochs=10, optimizer=None, noise_dim=30, device='cpu'):
        super(DCGAN, self).__init__(data_loader, batch_size, epochs, optimizer, noise_dim, device)

    def build(self):
        # Generator: Upscales noise to images
        self.generator = nn.Sequential(
            nn.Linear(self.noise_dim, 7 * 7 * 128),  # Upscale noise
            nn.BatchNorm1d(7 * 7 * 128),
            nn.ReLU(inplace=True),
            nn.Unflatten(1, (128, 7, 7)),
            nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(64, 1, kernel_size=4, stride=2, padding=1),
            nn.Tanh()  # Output between -1 and 1
        ).to(self.device)

        # Discriminator: Classifies images as real or fake
        self.discriminator = nn.Sequential(
            nn.Conv2d(1, 64, kernel_size=4, stride=2, padding=1),  # Downsample
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.4),
            nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1),  # Further downsample
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.4),
            nn.Flatten(),
            nn.Linear(128 * 7 * 7, 1),  # Single output
            nn.Sigmoid()  # Probability output
        ).to(self.device)
```

DCGAN class

Abstract GAN class

```python
class GAN(nn.Module):
    def __init__(self, data_loader=None, batch_size=32, epochs=10, optimizer=None,
                 noise_dim=30, device='cpu'):
        super(GAN, self).__init__()
        self.batch_size = batch_size  # Batch size for training
        self.epochs = epochs  # Number of training epochs
        self.optimizer = optimizer  # Optimizer class for training
        self.noise_dim = noise_dim  # Dimension of noise vector
        self.device = device  # Device to run the model (CPU/GPU)
        self.generator = None  # Placeholder for generator model
        self.discriminator = None  # Placeholder for discriminator model
        self.gen_optimizer = None  # Placeholder for generator optimizer
        self.dis_optimizer = None  # Placeholder for discriminator optimizer
        self.data_loader = data_loader  # Data loader for training dataset
```

```python
@staticmethod
def plot_multiple_images(images, n_cols=None):
    # Plot a grid of images
    n_cols = n_cols or len(images)
    n_rows = (len(images) - 1) // n_cols + 1
    if images.shape[1] == 1:
        images = images.squeeze(1)  # Remove channel dimension if grayscale
    plt.figure(figsize=(n_cols, n_rows))
    for index, image in enumerate(images):
        plt.subplot(n_rows, n_cols, index + 1)
        plt.imshow(image.cpu().detach().numpy(), cmap="gray")  # Display image
        plt.axis("off")  # Hide axes
    plt.show()

def build(self):
    pass  # To be implemented in subclasses

def generate(self, num_examples=None):
    # Generate fake images from random noise
    num_examples = self.batch_size if num_examples is None else num_examples
    z = torch.randn(num_examples, self.noise_dim, device=self.device)  # Random noise
    X_gen = self.generator(z)  # Generate images
    return X_gen
```

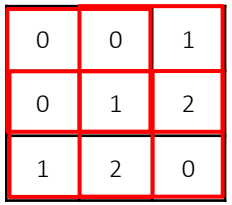
```python
def train(self):
    self.history = {'disc-loss': [], 'gen-loss': []}  # Track losses
    criterion = nn.BCELoss()  # Binary Cross Entropy loss

    for epoch in range(self.epochs):
        dis_losses = []  # List to store discriminator losses
        gen_losses = []  # List to store generator losses
        for batch_idx, (X_batch, y_batch) in enumerate(self.data_loader):
            X_batch = X_batch.to(self.device)  # Move batch to device

            # PHASE 1 - Train Discriminator
            z = torch.randn(X_batch.size(0), self.noise_dim, device=self.device)  # Random noise
            X_gen = self.generator(z)  # Generate fake images

            X_fake_and_real = torch.cat([X_gen, X_batch], dim=0)  # Combine real and fake images

            y1 = torch.cat([torch.zeros(X_gen.size(0), 1), torch.ones(X_batch.size(0), 1)], dim=0).to(self.device)  #

            self.discriminator.train()  # Set discriminator to training mode
            self.dis_optimizer.zero_grad()  # Zero gradients
            dis_loss = criterion(self.discriminator(X_fake_and_real), y1)  # Compute discriminator loss
            dis_loss.backward()  # Backpropagation
            self.dis_optimizer.step()  # Update discriminator weights
            dis_losses.append(dis_loss.item())  # Store loss

            # PHASE 2 - Train Generator
            z = torch.randn(self.batch_size, self.noise_dim, device=self.device)  # New random noise
            y2 = torch.ones(self.batch_size, 1, device=self.device)  # Labels for generator

            self.discriminator.eval()  # Freeze discriminator weights
            self.gen_optimizer.zero_grad()  # Zero gradients
            gen_loss = criterion(self.discriminator(self.generator(z)), y2)  # Compute generator loss
            gen_loss.backward()  # Backpropagation
            self.gen_optimizer.step()  # Update generator weights
            gen_losses.append(gen_loss.item())  # Store loss

        # Log and plot losses
        print(f"Epoch {epoch + 1}/{self.epochs}: disc-loss = {np.mean(dis_losses)}, gen-loss = {np.mean(gen_losses)}")
        self.history['disc-loss'].append(np.mean(dis_losses))  # Record average loss
        self.history['gen-loss'].append(np.mean(gen_losses))
```

StandardGAN

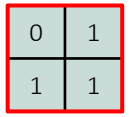
```python
class StandardGAN(GAN):
    def __init__(self, data_loader=None, batch_size=32, epochs=10, optimizer=None, noise_dim=30, device='cpu'):
        super(StandardGAN, self).__init__(data_loader, batch_size, epochs, optimizer, noise_dim, device)

    def build(self):
        # Generator network definition
        self.generator = nn.Sequential(
            nn.Linear(self.noise_dim, 256),  # Maps noise to 256 dimensions
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(256, 512),  # Upsamples to 512 dimensions
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(512, 1024),  # Upsamples to 1024 dimensions
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(1024, 784),  # Outputs 784 dimensions (28x28 images)
            nn.Tanh(),
            nn.Unflatten(1, torch.Size([1, 28, 28]))  # Reshapes to image format
        ).to(self.device)

        # Discriminator network definition
        self.discriminator = nn.Sequential(
            nn.Flatten(),  # Flattens input images
            nn.Linear(784, 1024),  # Processes 784-dimensional input
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.3),
            nn.Linear(1024, 512),  # Reduces to 512 dimensions
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.3),
            nn.Linear(512, 256),  # Further reduces to 256 dimensions
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.3),
            nn.Linear(256, 128),  # Reduces to 128 dimensions
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.3),
            nn.Linear(128, 1),  # Outputs a single score (real or fake)
            nn.Sigmoid()  # Probability output
        ).to(self.device)

        # Set optimizers for generator and discriminator
        self.dis_optimizer = self.optimizer(self.discriminator.parameters(), lr=2e-4, betas=(0.5, 0.999))
        self.gen_optimizer = self.optimizer(self.generator.parameters(), lr=2e-4, betas=(0.5, 0.999))
```

Introducing Conv2DTranspose

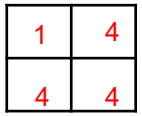
- Standard Conv2D
  - **Conv2D:** input = [3,3], kernel = [2,2], stride = [1,1] → output = [2,2]



input



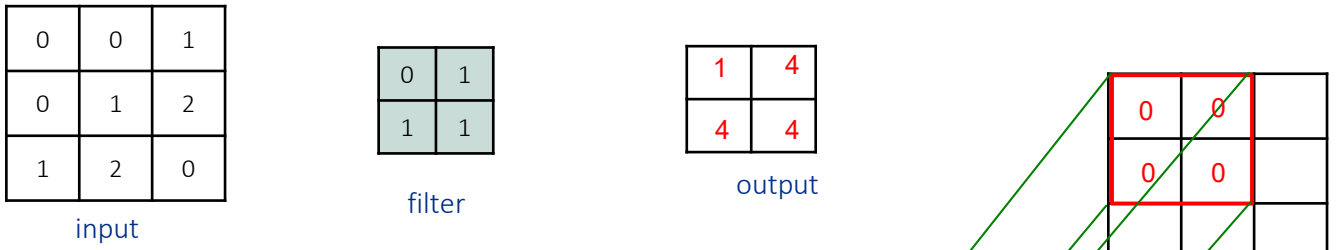
filter



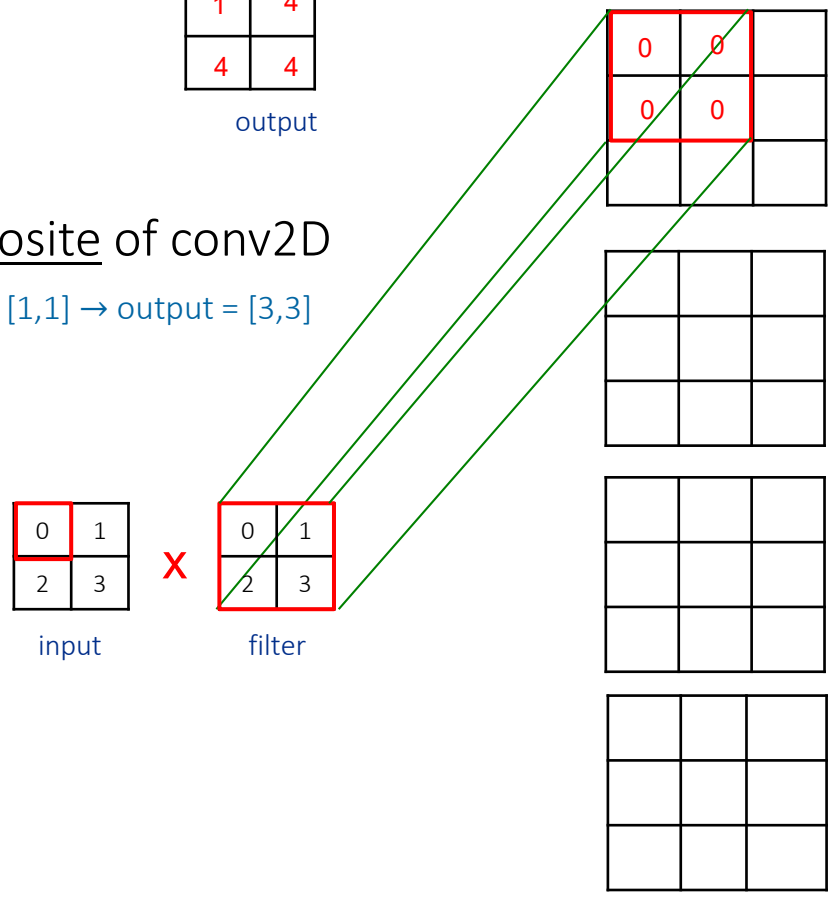
output

Introducing Conv2DTranspose

- Standard Conv2D
  - **Conv2D:** input = [3,3], kernel = [2,2], stride = [1,1] → output = [2,2]

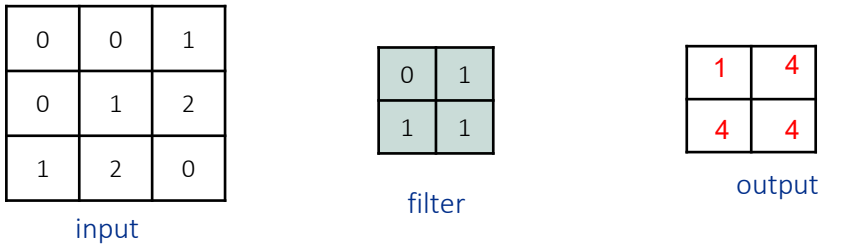


- Conv2Dtranpose: it is <u>opposite</u> of conv2D
  - Input =[2,2], kernel = [2,2], stride = [1,1] → output = [3,3]

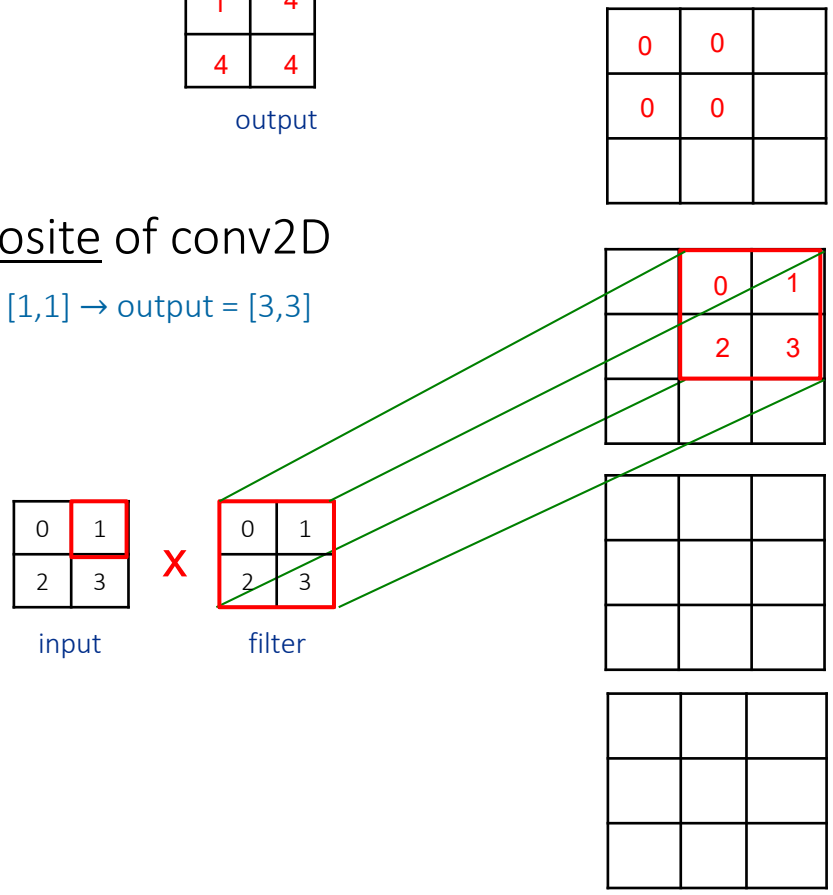


Introducing Conv2DTranspose

- Standard Conv2D
  - **Conv2D:** input = [3,3], kernel = [2,2], stride = [1,1] → output = [2,2]

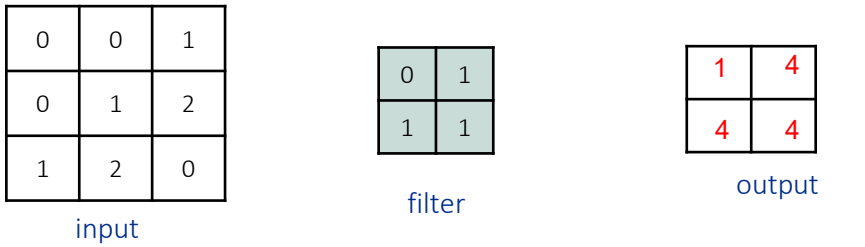


- Conv2Dtranpose: it is <u>opposite</u> of conv2D
  - Input =[2,2], kernel = [2,2], stride = [1,1] → output = [3,3]

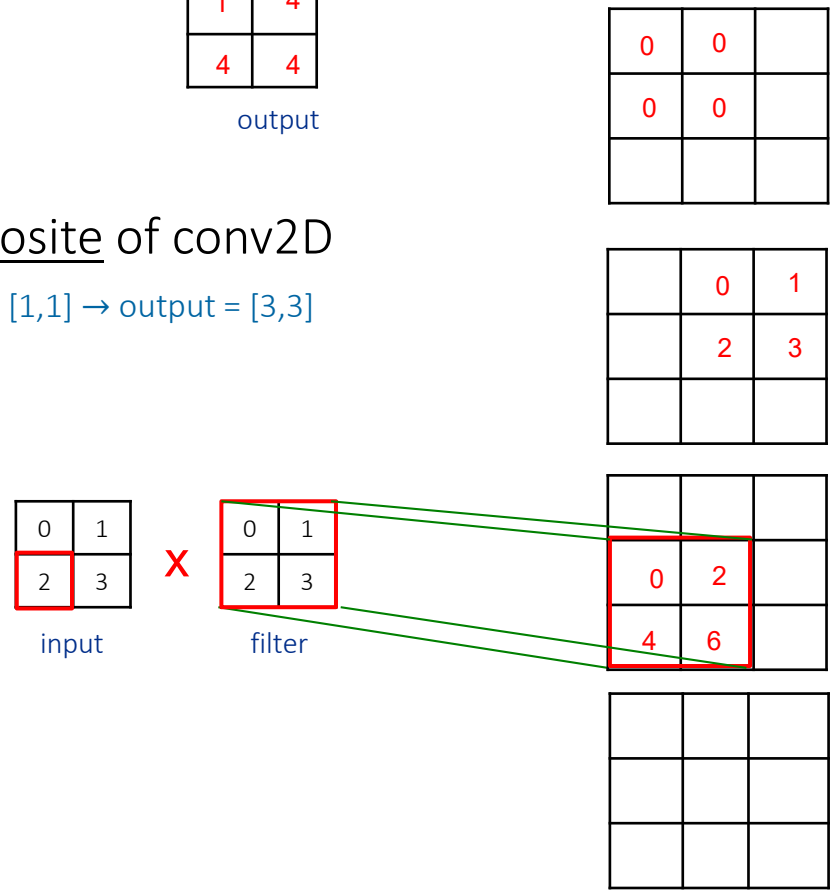


Introducing Conv2DTranspose

- Standard Conv2D
  - **Conv2D:** input = [3,3], kernel = [2,2], stride = [1,1] → output = [2,2]

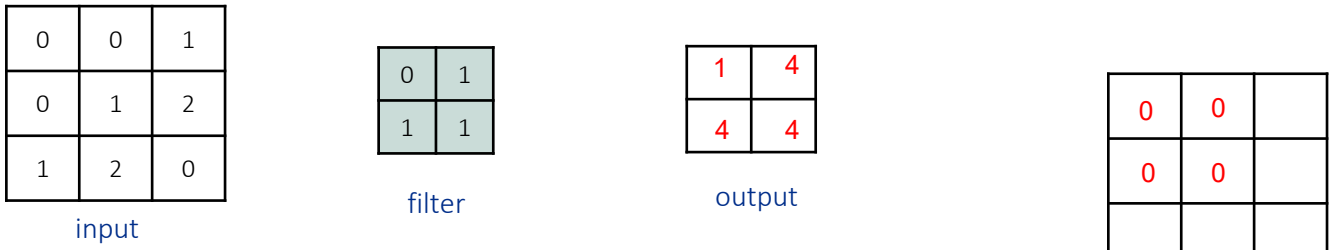


- Conv2Dtranpose: it is <u>opposite</u> of conv2D
  - Input =[2,2], kernel = [2,2], stride = [1,1] → output = [3,3]

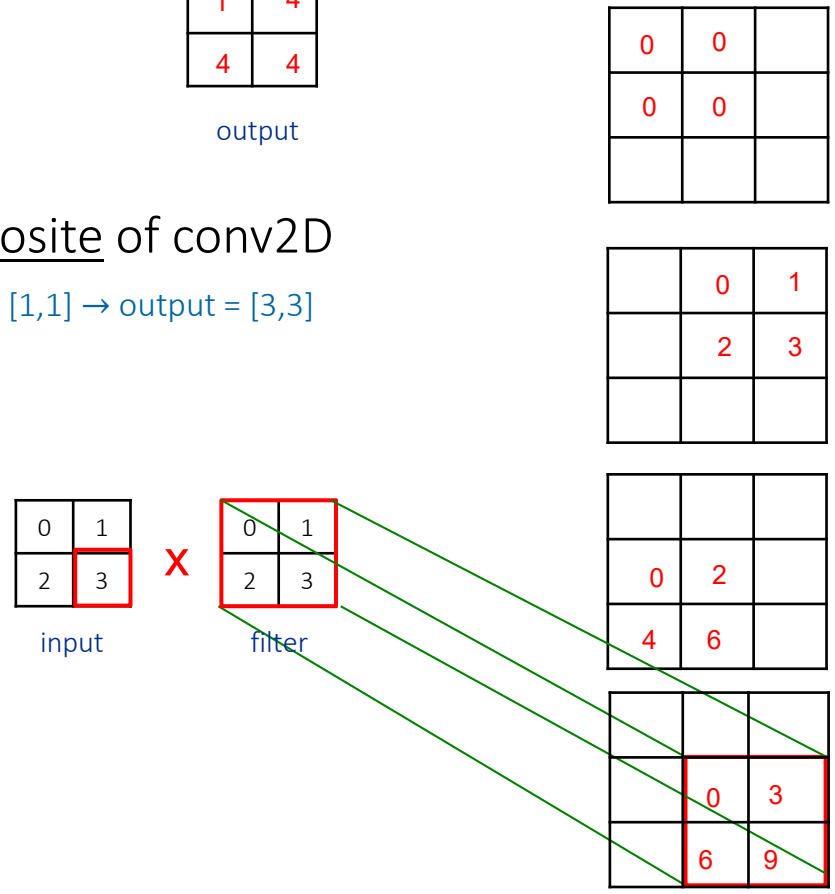


Introducing Conv2DTranspose

- Standard Conv2D
  - **Conv2D:** input = [3,3], kernel = [2,2], stride = [1,1] → output = [2,2]

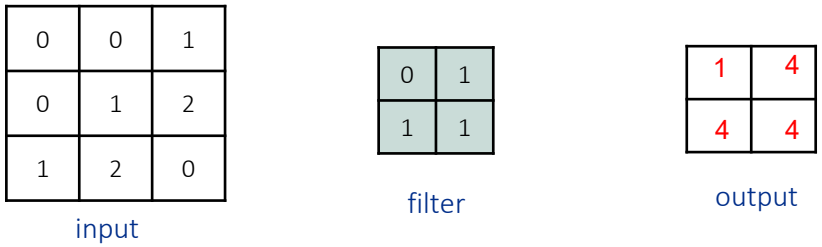


- Conv2Dtranpose: it is <u>opposite</u> of conv2D
  - Input =[2,2], kernel = [2,2], stride = [1,1] → output = [3,3]

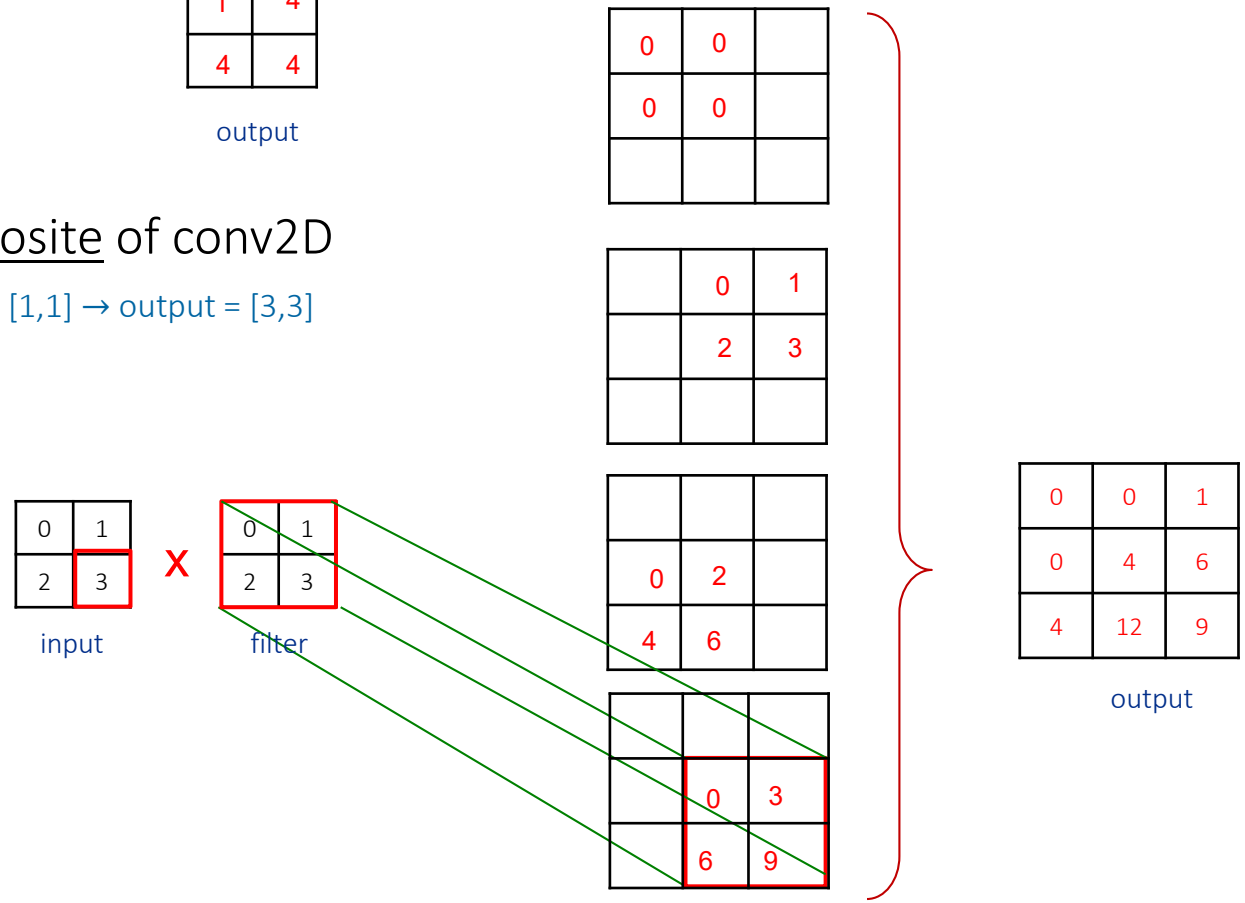


Introducing Conv2DTranspose

- Standard Conv2D
  - **Conv2D:** input = [3,3], kernel = [2,2], stride = [1,1] → output = [2,2]



- Conv2Dtranpose: it is <u>opposite</u> of conv2D
  - Input =[2,2], kernel = [2,2], stride = [1,1] → output = [3,3]



Conv2DTranspose

- Basic syntax

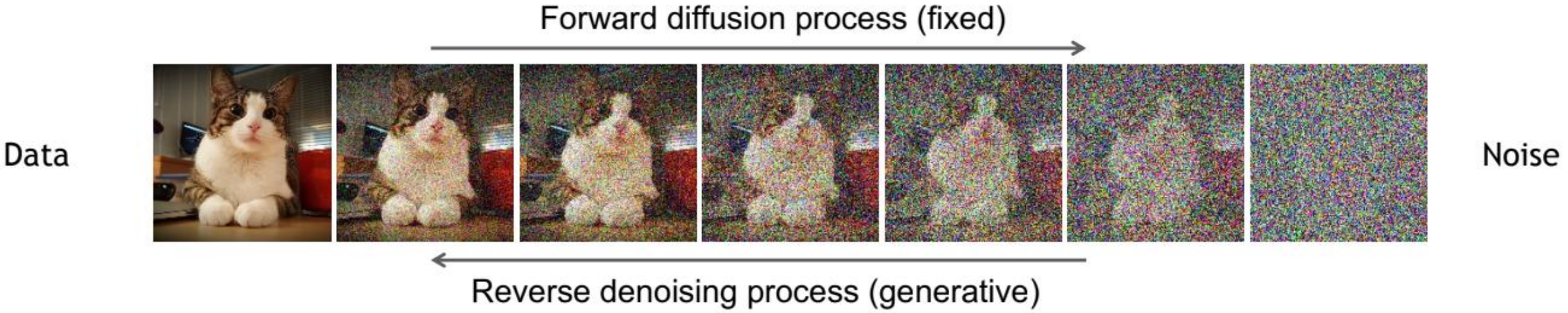
```python
torch.nn.ConvTranspose2d(in_channels, out_channels, kernel_size, stride, padding=0)
```

- Arguments
  - **in_channels**: Number of input channels (depth of the input feature map).
  - **out_channels**: Number of output channels (depth of the output feature map).
  - **kernel_size**: Size of the convolving kernel (e.g., 3 for a 3x3 filter).
  - **stride**: Step size of the transposed convolution (determines the upsampling factor).
  - **padding**: Implicit padding added on both sides of the input.
- Output sizes
  - **Output Size**=$(I-1)\times S-2P+K+$output padding
  - Input size (I), stride (S), padding (P), kernel size (K), output padding is optional

DCGAN

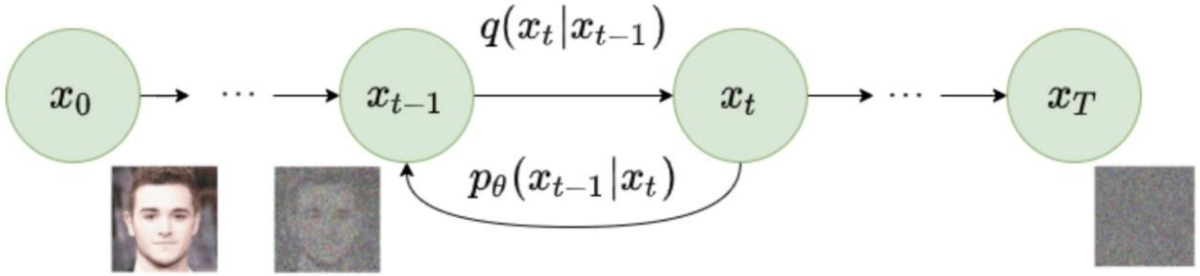
```python
class DCGAN(GAN):
    def __init__(self, data_loader=None, batch_size=32, epochs=10, optimizer=None, noise_dim=30, device='cpu'):
        super(DCGAN, self).__init__(data_loader, batch_size, epochs, optimizer, noise_dim, device)

    def build(self):
        # Generator: Upscales noise to images
        self.generator = nn.Sequential(
            nn.Linear(self.noise_dim, 7 * 7 * 128),  # Upscale noise
            nn.BatchNorm1d(7 * 7 * 128),
            nn.ReLU(inplace=True),
            nn.Unflatten(1, (128, 7, 7)),
            nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(64, 1, kernel_size=4, stride=2, padding=1),
            nn.Tanh()  # Output between -1 and 1
        ).to(self.device)

        # Discriminator: Classifies images as real or fake
        self.discriminator = nn.Sequential(
            nn.Conv2d(1, 64, kernel_size=4, stride=2, padding=1),  # Downsample
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.4),
            nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1),  # Further downsample
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.4),
            nn.Flatten(),
            nn.Linear(128 * 7 * 7, 1),  # Single output
            nn.Sigmoid()  # Probability output
        ).to(self.device)

        # Optimizers for both networks
        self.dis_optimizer = self.optimizer(self.discriminator.parameters(), lr=2e-4, betas=(0.5, 0.999))
        self.gen_optimizer = self.optimizer(self.generator.parameters(), lr=2e-4, betas=(0.5, 0.999))
```

Diffusion Model: Probabilistic Viewpoint



- **Forward process**
  - Shrinking and adding Gaussian noises
  - $\mathbf{x}_t = \sqrt{1-\beta_t}\mathbf{x}_{t-1} + \sqrt{\beta_t}\boldsymbol{\epsilon}$ where $\beta_t \uparrow 1$
- **Backward process**
  - Learning to generate data by denoising
  - $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\right) + \sigma_t \mathbf{z}$

Overview of Diffusion Models

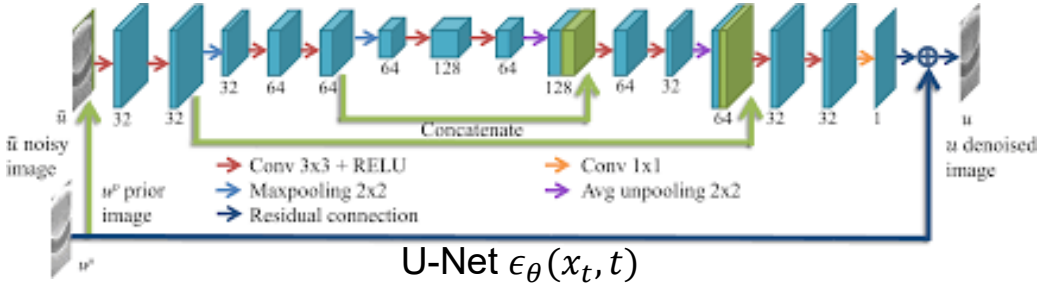


Reverse diffusion process. Image modified by Ho et al. 2020

$\mathbf{x}_t = \sqrt{1-\beta_t}\mathbf{x}_{t-1} + \sqrt{\beta_t}\boldsymbol{\epsilon}$ and $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}$ where $\alpha_t = 1-\beta_t$ and $\bar{\alpha}_t = \prod_{s=1}^{t}\alpha_s$

- $\alpha_t \downarrow 0$ and $\bar{\alpha}_t \downarrow 0$ → $\mathbf{x}_T = \sqrt{\bar{\alpha}_T}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_T}\boldsymbol{\epsilon} = \sqrt{\prod_{s=1}^{T}\alpha_s}\,\mathbf{x}_0 + \sqrt{1-\prod_{s=1}^{T}\alpha_s}\,\boldsymbol{\epsilon} \xrightarrow{T\to\infty} \boldsymbol{N}(\mathbf{0},\mathbf{I})$
- $q(x_{t-1}|x_t, x_0) = N\left(\frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}\right), \sigma_t^2 I\right)$ with $\sigma_t = \sqrt{\frac{\beta_t(1-\bar{\alpha}_t)}{1-\alpha_t}}$
- $p_\theta(x_{t-1} \mid x_t) = N(\mu_t(x_t, t;\theta), \sigma_t^2 I)$

  with $\mu_t(x_t, t;\theta) = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\right)$ and $\sigma_t = \sqrt{\frac{\beta_t(1-\bar{\alpha}_t)}{1-\alpha_t}}$



U-Net $\epsilon_\theta(x_t, t)$

Algorithm 1 Training

1: **repeat**
2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3: $t \sim \text{Uniform}(\{1, \ldots, T\})$
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5: Take gradient descent step on
$\nabla_\theta \left\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t)\right\|^2$
6: **until** converged

Use $\epsilon_\theta(x_t, t)$ to predict the noise ϵ injected by **minimizing**

$$\left|\left|\epsilon_\theta(x_t, t) - \epsilon\right|\right|^2$$

Algorithm 2 Sampling

1: $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
2: **for** $t = T, \ldots, 1$ **do**
3: $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
4: $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\right) + \sigma_t\mathbf{z}$
5: **end for**
6: **return** $\mathbf{x}_0$

Implementation of Diffusion Models

- Step 1: Data Preparation
- Step 2: Forward Diffusion Process
- Step 3: Building a U-net Model for $\epsilon_\theta(\mathbf{x}_t, t)$
- Step 4: Model Training

$$L = \mathbb{E}_{\mathbf{x_0} \sim q(\mathbf{x}), \boldsymbol{\epsilon} \sim N(0,I), \mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}} \left[\left|\left| \epsilon - \epsilon_\theta(\mathbf{x}_t, t) \right|\right|^2 \right]$$

- Step 5: Generation by Sampling

Thanks for your attention!