DCGAN (Deep Convolutional Generative Adversarial Networks) Cheatsheet

Your Name

1 Concepts of DCGAN

DCGANs are an advanced type of GAN (Generative Adversarial Network) that leverage deep convolutional networks to generate high-quality images. They consist of two competing neural networks:

- **Generator:** Learns to generate fake data (e.g., images) from a random noise vector.
- **Discriminator:** Attempts to distinguish between real and fake data generated by the generator.

DCGANs, introduced by Radford et al. (2016), apply convolutional layers instead of fully connected layers in both the generator and discriminator, providing improved performance on generating high-resolution images. They were designed to address the instability and mode collapse issues of traditional GANs.

2 DCGAN Architecture

2.1 Key Components

2.1.1 Generator

- Uses fractionally-strided convolutions (sometimes referred to as "deconvolutions") to upscale random noise into a full-sized image.
- The output layer uses the tanh activation function to ensure pixel values lie between -1 and 1.
- Uses ReLU activations for all layers except the output.

2.1.2 Discriminator

- Employs strided convolutions to downsample images, distinguishing real from generated (fake) ones.
- Uses Leaky ReLU activation (with a slope of 0.2).
- Outputs a single scalar via a sigmoid function representing the probability that an input image is real.

2.2 Architectural Guidelines

- No fully connected layers in the generator or discriminator.
- Use BatchNorm in both generator and discriminator.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU in generator for all layers except the output (which uses Tanh).
- Use LeakyReLU in the discriminator for all layers.

2.3 Architectural Improvements

- **Batch Normalization:** Applied to stabilize training by normalizing input to each layer, helping gradients flow better in deeper networks.
- **Elimination of Pooling Layers:** Replaced with strided convolutions in the discriminator and fractionally-strided convolutions in the generator to allow the network to learn its own down-sampling and up-sampling operations.

3 Mathematical Foundations

3.1 Adversarial Loss Function

The DCGAN optimizes the following loss function in a min-max game:

- Discriminator Loss:

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

- Generator Loss:

$$\min_G \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))]$$

Where:

- $D(x)$: Discriminator's prediction for real data x.
- $G(z)$: Generator's output for random noise z.
- $p_z(z)$: The distribution of the input noise z.

3.2 Minimax Game

- The discriminator aims to maximize the loss function.
- The generator aims to minimize the loss function.

3.3 Alternative Loss Functions

- Wasserstein loss for improved stability.
- Least squares loss for smoother gradients.

4 Training Details

4.1 Training Algorithm

1. Sample mini-batch of real data x and random noise z.
2. Compute the discriminator loss by distinguishing real from fake data.
3. Update discriminator using gradient descent.
4. Sample a new batch of noise z and compute the generator loss based on discriminator feedback.
5. Update generator using gradient descent.

4.2 Optimization

Adam optimizer with learning rate 0.0002 and momentum parameter $\beta_1 = 0.5$ is often used for both networks.

4.3 Stabilization Techniques

- Balance generator and discriminator training.
- Label smoothing: Use soft labels (e.g., 0.9 instead of 1 for real images).
- One-sided label smoothing to prevent the discriminator from becoming too confident.

5 Applications of DCGANs

1. **Image Generation:** Generate high-quality images from random noise after training on a dataset (e.g., bedrooms, faces).
2. **Feature Extraction:** Use discriminators as unsupervised feature extractors for downstream tasks like image classification.
3. **Data Augmentation:** Generate new synthetic data in cases where datasets are small or limited.
4. **Art and Creativity:** Create art, design new objects, or generate novel images in creative fields.
5. **Vector Arithmetic on Latent Space:** Perform semantic vector arithmetic, such as adding or subtracting image features (e.g., adding ”smile” to a face).
6. **Style Transfer:** Combine features of different images.
7. **Super-resolution:** Generate high-resolution images from low-resolution inputs.
8. **Image-to-image Translation:** Convert images from one domain to another.

6 Code Implementation

```
import torch
import torch.nn as nn

# Define the generator
class Generator(nn.Module):
    def __init__(self, nz, ngf, nc):
        super(Generator, self).__init__()
        self.main = nn.Sequential(
            # Input is Z, going into a convolution
            nn.ConvTranspose2d(nz, ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # State size. (ngf*8) x 4 x 4
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # State size. (ngf*4) x 8 x 8
            nn.ConvTranspose2d(ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # State size. (ngf*2) x 16 x 16
            nn.ConvTranspose2d(ngf * 2, nc, 4, 2, 1, bias=False),
            nn.Tanh()
            # State size. (nc) x 32 x 32
        )

    def forward(self, input):
        return self.main(input)

# Define the discriminator
class Discriminator(nn.Module):
    def __init__(self, nc, ndf):
        super(Discriminator, self).__init__()
        self.main = nn.Sequential(
            # Input is (nc) x 32 x 32
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # State size. (ndf) x 16 x 16
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # State size. (ndf*2) x 8 x 8
            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # State size. (ndf*4) x 4 x 4
            nn.Conv2d(ndf * 4, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, input):
        return self.main(input)

# Hyperparameters
nz = 100  # Size of input latent vector (noise)
ngf = 64  # Size of generator feature maps
ndf = 64  # Size of discriminator feature maps
nc = 3    # Number of channels in the training images (3 for RGB)

# Create instances of the generator and discriminator
netG = Generator(nz, ngf, nc)
netD = Discriminator(nc, ndf)

# Loss and optimizers
criterion = nn.BCELoss()
optimizerD = torch.optim.Adam(netD.parameters(), lr=0.0002, betas=(0.5
optimizerG = torch.optim.Adam(netG.parameters(), lr=0.0002, betas=(0.5

# Function to generate images from noise
def generate_images(netG, nz, num_images):
    noise = torch.randn(num_images, nz, 1, 1)
    fake_images = netG(noise)
    return fake_images
```

7 Visualizing DCGAN Outputs

- **Latent Space Interpolation:** Visualize how small changes in the input noise z lead to smooth transformations in generated images.
- **Guided Backpropagation:** Useful for interpreting what specific layers in the discriminator are learning (e.g., learning to detect objects like windows, beds, etc., in image datasets).
- **t-SNE Visualization:** Visualize the latent space in 2D or 3D.
- **Class Activation Maps:** Understand which parts of the image contribute to the discriminator's decision.

8 Evaluation Metrics

- **Inception Score (IS):** Measures both quality and diversity of generated images.
- **Fréchet Inception Distance (FID):** Compares the statistics of generated images to real images.
- **Precision and Recall:** Assesses the trade-off between sample quality and diversity.

9 Recent Advancements

- **StyleGAN and its variations:** For controllable image generation.
- **BigGAN:** For high-fidelity image synthesis.
- **Self-Attention GANs:** For capturing long-range dependencies.

10 Challenges and Future Directions

- Improving training stability further.
- Addressing mode collapse more effectively.
- Extending DCGANs to other domains (e.g., video, audio).
- Ethical considerations in using GANs for image synthesis.
- Integrating DCGANs with other deep learning techniques for more complex tasks.