Build Basic Generative Adversarial Networks (GANs)

Intro to Generative Adversarial Networks (GANs)

Learning Objectives:

1. Construct Your First GAN
2. Develop Intuition Behind GANs and Their Components
3. Examine Real-Life Applications of GANs

1. Understanding GANs

What is a GAN?

A Generative Adversarial Network (GAN) (Goodfellow et al., 2014) is a type of machine learning framework that consists of two neural networks: the Generator (G) and the Discriminator (D). These networks are trained simultaneously in a zero-sum game, where one network tries to outsmart the other.

- Generator (G): This network generates synthetic data from random noise (latent space). Its goal is to produce data that is indistinguishable from real data.
- Discriminator (D): This network evaluates data and determines whether it is real (from the training set) or fake (generated by G). Its goal is to maximize its ability to distinguish between the two.

GAN Architecture

The GAN training process can be visualized as follows:

1. The Generator produces a batch of fake data samples.
2. The Discriminator evaluates both real and fake samples and assigns probabilities indicating whether each sample is real or fake.
3. The feedback from the Discriminator is used to update both networks:
    - The Discriminator aims to maximize its ability to differentiate real from fake data.
    - The Generator aims to minimize the Discriminator's ability to correctly identify the fake data.

Mathematical Formulation

The objective of GANs can be defined using a minimax game:

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

Where:

- $p_{data}(x)$ is the distribution of real data.
- $p_z(z)$ is the distribution of the input noise (latent space).
- $D(x)$ is the Discriminator's output probability that x is real.
- $G(z)$ is the output of the Generator.

2. Building Your First GAN with PyTorch

Step-by-Step Implementation

In this section, you'll build a simple GAN using PyTorch. We'll create a GAN to generate handwritten digits similar to those found in the MNIST dataset.

1. Import Required Libraries

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.datasets as dsets
import torchvision.transforms as transforms
```

2. Define the Generator and Discriminator Models

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(100, 256),
            nn.ReLU(),
            nn.Linear(256, 512),
            nn.ReLU(),
            nn.Linear(512, 1024),
            nn.ReLU(),
            nn.Linear(1024, 28*28),
            nn.Tanh()  # Output should be between -1 and 1
        )

    def forward(self, z):
        return self.model(z).view(-1, 1, 28, 28)

class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Flatten(),
            nn.Linear(28*28, 512),
            nn.LeakyReLU(0.2),
```

```
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 1),
            nn.Sigmoid()  # Output between 0 and 1
        )

    def forward(self, img):
        return self.model(img)
```

3. Initialize Models, Loss Function, and Optimizers

```
generator = Generator()
discriminator = Discriminator()

criterion = nn.BCELoss()
optimizer_G = optim.Adam(generator.parameters(), lr=0.0002)
optimizer_D = optim.Adam(discriminator.parameters(), lr=0.0002)
```

4. Training Loop

```
# Load the MNIST dataset
mnist = dsets.MNIST(root='./data', train=True, download=True,
transform=transforms.ToTensor())
data_loader = torch.utils.data.DataLoader(mnist, batch_size=64, shuffle=True)

# Training the GAN
num_epochs = 50
for epoch in range(num_epochs):
    for real_images, _ in data_loader:
        batch_size = real_images.size(0)

        # Train Discriminator
        z = torch.randn(batch_size, 100)  # Random noise
        fake_images = generator(z)
        real_labels = torch.ones(batch_size, 1)  # Labels for real images
        fake_labels = torch.zeros(batch_size, 1)  # Labels for fake images

        optimizer_D.zero_grad()
        outputs = discriminator(real_images)
        d_loss_real = criterion(outputs, real_labels)
        outputs = discriminator(fake_images.detach())
        d_loss_fake = criterion(outputs, fake_labels)
        d_loss = d_loss_real + d_loss_fake
        d_loss.backward()
        optimizer_D.step()

        # Train Generator
```

```
        optimizer_G.zero_grad()
        outputs = discriminator(fake_images)
        g_loss = criterion(outputs, real_labels)
        g_loss.backward()
        optimizer_G.step()

      print(f'Epoch [{epoch+1}/{num_epochs}], d_loss: {d_loss.item():.4f},
g_loss: {g_loss.item():.4f}')
```

3. Real-Life Applications of GANs

GANs have a wide range of applications across various fields:

1. **Image Generation**: Creating realistic images from random noise, used in art generation and photo editing.
2. **Super Resolution**: Enhancing the resolution of images, particularly in medical imaging and satellite imagery.
3. **Image-to-Image Translation**: Applications like converting sketches to realistic images (pix2pix), altering seasons in photos, or transforming day images to night.
4. **Data Augmentation**: Generating synthetic data for training models, especially in cases where data is scarce.
5. **Anomaly Detection**: Identifying unusual patterns in data, particularly in fraud detection and quality control in manufacturing.

Deep Convolutional GANs (DCGANs)

Learning Objectives:

1. Explain the Components of a Deep Convolutional GAN
2. Compose a Deep Convolutional GAN Using These Components
3. Examine the Difference Between Upsampling and Transposed Convolutions

1. Understanding Deep Convolutional GANs (DCGANs)

What is a DCGAN?

Deep Convolutional GANs (DCGANs) (Radford et al., 2016) extend the traditional GAN architecture by incorporating convolutional layers into the Generator and Discriminator. This makes them particularly effective for image generation tasks, as convolutional layers are adept at capturing spatial hierarchies in images.

Components of a DCGAN

- Generator (G): The generator in a DCGAN uses transposed convolutional layers (also known as deconvolutions) to upscale the input noise vector into a full-size image.
- Discriminator (D): The discriminator uses convolutional layers to downsample the input image and classify it as real or fake.

Key Components

1. Convolutional Layers: Capture spatial patterns and features in images.
2. Transposed Convolutional Layers: Upsample the feature maps in the generator.
3. Batch Normalization: Normalizes the activations of the previous layer to stabilize and speed up training.
4. Activation Functions: Functions that introduce non-linearity into the model, with ReLU and Leaky ReLU being commonly used in DCGANs.

2. Building a Deep Convolutional GAN Using PyTorch

Step-by-Step Implementation

This section outlines how to implement a DCGAN using PyTorch to generate images from the MNIST dataset.

1. Import Required Libraries

```
import torch
import torch.nn as nn
```

```
import torch.optim as optim
import torchvision.datasets as dsets
import torchvision.transforms as transforms
```

2. Define the DCGAN Generator

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
                nn.ConvTranspose2d(100, 128, 7, 1, 0, bias=False),   #
Input: (N, 100, 1, 1)
            nn.BatchNorm2d(128),
            nn.ReLU(),
                 nn.ConvTranspose2d(128, 64, 4, 2, 1, bias=False),   #
Upsample to (N, 64, 2, 2)
            nn.BatchNorm2d(64),
            nn.ReLU(),
               nn.ConvTranspose2d(64, 1, 4, 2, 1, bias=False),        #
Upsample to (N, 1, 28, 28)
            nn.Tanh()  # Output image values between -1 and 1
        )

    def forward(self, z):
        return self.model(z.view(-1, 100, 1, 1))
```

3. Define the DCGAN Discriminator

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
             nn.Conv2d(1, 64, 4, 2, 1, bias=False),  # Input: (N, 1, 28,
28)
            nn.LeakyReLU(0.2),
              nn.Conv2d(64, 128, 4, 2, 1, bias=False),  # Downsample to
(N, 128, 7, 7)
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2),
            nn.Flatten(),
                nn.Linear(128 * 7 * 7, 1),   # Output single value for
real/fake
            nn.Sigmoid()  # Output probability
        )
```

```
    def forward(self, img):
        return self.model(img)
```

4. Initialize Models, Loss Function, and Optimizers

```
generator = Generator()
discriminator = Discriminator()

criterion = nn.BCELoss()
optimizer_G = optim.Adam(generator.parameters(), lr=0.0002)
optimizer_D = optim.Adam(discriminator.parameters(), lr=0.0002)
```

5. Training Loop

```
# Load the MNIST dataset
mnist = dsets.MNIST(root='./data', train=True, download=True,
transform=transforms.ToTensor())
data_loader = torch.utils.data.DataLoader(mnist, batch_size=64,
shuffle=True)

# Training the DCGAN
num_epochs = 50
for epoch in range(num_epochs):
    for real_images, _ in data_loader:
        batch_size = real_images.size(0)

        # Train Discriminator
        z = torch.randn(batch_size, 100, 1, 1)  # Random noise
        fake_images = generator(z)
           real_labels = torch.ones(batch_size, 1)   # Labels for real
images
           fake_labels = torch.zeros(batch_size, 1)   # Labels for fake
images

        optimizer_D.zero_grad()
        outputs = discriminator(real_images)
        d_loss_real = criterion(outputs, real_labels)
        outputs = discriminator(fake_images.detach())
        d_loss_fake = criterion(outputs, fake_labels)
        d_loss = d_loss_real + d_loss_fake
        d_loss.backward()
        optimizer_D.step()

        # Train Generator
```

```
        optimizer_G.zero_grad()
        outputs = discriminator(fake_images)
        g_loss = criterion(outputs, real_labels)
        g_loss.backward()
        optimizer_G.step()

                print(f'Epoch    [{epoch+1}/{num_epochs}],    d_loss:
{d_loss.item():.4f}, g_loss: {g_loss.item():.4f}')
```

3. Upsampling vs. Transposed Convolutions

Upsampling

Upsampling is the process of increasing the spatial dimensions (height and width) of feature maps. Common techniques include:

- Nearest Neighbor: Duplicates values from the original feature map.
- Bilinear Interpolation: Uses linear interpolation to estimate values.

Transposed Convolutions

Transposed convolutions (or deconvolutions) are a learnable operation that performs upsampling while simultaneously applying weights. This allows the generator to learn the distribution of the output images more effectively.

Mathematical Representation

The transposed convolution operation can be represented mathematically as:

$$y_{out} = (x * k) + b \tag{2}$$

{\hspace{-1in}(1)}

Where:

- x is the input tensor,
- k is the filter/kernel,
- b is the bias term, and
- y_{out} is the output tensor after the operation.

In contrast to traditional convolution, where the kernel slides over the input, in transposed convolution, the kernel moves over the output, spreading the values of x across the output space, effectively increasing the dimensions.

Wasserstein GANs (WGANs) with Gradient Penalty

Learning Objectives:

1. Examine the Cause and Effect of Mode Collapse in GAN Training
2. Implement a Wasserstein GAN with Gradient Penalty to Mitigate Mode Collapse
3. Understand the Motivation and Conditions Needed for Wasserstein-Loss

1. Understanding Mode Collapse

What is Mode Collapse?

Mode collapse is a common issue in GAN training where the generator produces a limited variety of outputs, effectively ignoring certain modes in the target distribution. For instance, if the generator only produces a few types of images while other potential outputs are disregarded, the diversity of generated samples is compromised.

Causes of Mode Collapse

- Imbalance in Learning Rates: If the discriminator learns too quickly compared to the generator, it may overpower the generator, leading to suboptimal output.
- Loss Function Limitations: The traditional GAN loss function can create situations where the generator converges to a specific point in the data distribution.

Effects of Mode Collapse

- Reduced diversity in generated samples.
- Difficulty in generating realistic outputs that cover the entire range of the target distribution.

2. Implementing a Wasserstein GAN (WGAN) with Gradient Penalty

What is a Wasserstein GAN?

WGANs (Arjovsky et al., 2017) utilize the Earth Mover's Distance (also known as Wasserstein distance) as a measure of the difference between the real and generated data distributions. This distance provides a smoother gradient for the generator, which can alleviate the problems of mode collapse.

WGAN Loss Function

The WGAN loss function can be defined as follows:

- For the Discriminator D:

$$L_D = \mathbb{E}_{x \sim p_{data}}[D(x)] - \mathbb{E}_{z \sim p_z}[D(G(z))] \tag{3}$$

- For the Generator G:

$$L_G = -\mathbb{E}_{z \sim p_z}[D(G(z))] \tag{4}$$

Where:

- p_{data} is the distribution of real data.
- p_z is the distribution of the input noise.
- $D(x)$ is the discriminator's estimate of the real data.

Lipschitz Continuity

To ensure that the WGAN can effectively measure the Wasserstein distance, the discriminator D must satisfy Lipschitz continuity. This means that the function's output should not change too rapidly, which can be achieved by constraining the weights of D during training.

Gradient Penalty

Instead of weight clipping (the original method used to enforce Lipschitz continuity), the gradient penalty method introduces a penalty term to the loss function. The gradient penalty is defined as:

$$L_{GP} = \lambda \mathbb{E}_{\hat{x} \sim p_{\hat{x}}} \left[(\|\nabla D(\hat{x})\|_2 - 1)^2 \right] \tag{5}$$

Where:

- $\hat{x}$ is a point sampled uniformly along the line between real and generated samples.
- λ is a hyperparameter that controls the strength of the penalty.

WGAN Loss Function with Gradient Penalty

The overall loss functions for WGAN with Gradient Penalty can be summarized as follows:

- For the Discriminator:

$$L_D = -\mathbb{E}_{x \sim p_{data}}[D(x)] + \mathbb{E}_{z \sim p_z}[D(G(z))] + L_{GP} \tag{6}$$

- For the Generator:

$$L_G = -\mathbb{E}_{z \sim p_z}[D(G(z))] \tag{7}$$

3. Implementation of WGAN with Gradient Penalty in PyTorch

Step-by-Step Implementation

This section outlines how to implement a WGAN with Gradient Penalty using PyTorch.

1. Import Required Libraries

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.datasets as dsets
import torchvision.transforms as transforms
```

2. Define the WGAN Generator and Discriminator

```
class WGANGenerator(nn.Module):
    def __init__(self):
        super(WGANGenerator, self).__init__()
        self.model = nn.Sequential(
            nn.ConvTranspose2d(100, 128, 7, 1, 0),
            nn.ReLU(),
            nn.ConvTranspose2d(128, 64, 4, 2, 1),
            nn.ReLU(),
            nn.ConvTranspose2d(64, 1, 4, 2, 1),
            nn.Tanh()  # Output image values between -1 and 1
        )

    def forward(self, z):
        return self.model(z.view(-1, 100, 1, 1))


class WGANDiscriminator(nn.Module):
    def __init__(self):
        super(WGANDiscriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Conv2d(1, 64, 4, 2, 1),
            nn.LeakyReLU(0.2),
            nn.Conv2d(64, 128, 4, 2, 1),
            nn.LeakyReLU(0.2),
            nn.Flatten(),
            nn.Linear(128 * 7 * 7, 1)
        )

    def forward(self, img):
        return self.model(img)
```

3. Define the Gradient Penalty Function

```
def gradient_penalty(discriminator, real_images, fake_images,
lambda_gp):
    batch_size = real_images.size(0)
    epsilon = torch.rand(batch_size, 1, 1, 1).to(real_images.device)
      interpolated_images = epsilon * real_images + (1 - epsilon) *
fake_images
    interpolated_images.requires_grad_(True)

    d_interpolated = discriminator(interpolated_images)
    gradients = torch.autograd.grad(outputs=d_interpolated,
                                    inputs=interpolated_images,

grad_outputs=torch.ones_like(d_interpolated),
                                    create_graph=True,
                                    retain_graph=True)[0]
    gradient_norm = gradients.view(batch_size, -1).norm(2, dim=1)
    return ((gradient_norm - 1) ** 2).mean() * lambda_gp
```

4. Initialize Models, Loss Function, and Optimizers

```
generator = WGANGenerator()
discriminator = WGANDiscriminator()

optimizer_G = optim.RMSprop(generator.parameters(), lr=0.00005)
optimizer_D = optim.RMSprop(discriminator.parameters(), lr=0.00005)
```

5. Training Loop

```
# Load the MNIST dataset
mnist = dsets.MNIST(root='./data', train=True, download=True,
transform=transforms.ToTensor())
data_loader = torch.utils.data.DataLoader(mnist, batch_size=64,
shuffle=True)

# Training the WGAN
num_epochs = 50
lambda_gp = 10
for epoch in range(num_epochs):
    for real_images, _ in data_loader:
        batch_size = real_images.size(0)

        # Train Discriminator
        for _ in range(5):  # Train D more frequently than G
            z = torch.randn(batch_size, 100).view(-1, 100, 1, 1)
            fake_images = generator(z)
```

```
            optimizer_D.zero_grad()
            real_loss = -torch.mean(discriminator(real_images))
            fake_loss = torch.mean(discriminator(fake_images.detach()))
                    gp  =  gradient_penalty(discriminator,  real_images,
fake_images, lambda_gp)
            d_loss = real_loss + fake_loss + gp
            d_loss.backward()
            optimizer_D.step()

        # Train Generator
        optimizer_G.zero_grad()
        z = torch.randn(batch_size, 100).view(-1, 100, 1, 1)
        fake_images = generator(z)
        g_loss = -torch.mean(discriminator(fake_images))
        g_loss.backward()
        optimizer_G.step()

                print(f'Epoch     [{epoch+1}/{num_epochs}],     d_loss:
{d_loss.item():.4f}, g_loss: {g_loss.item():.4f}')
```

Conditional GANs and Controllable Generation

Learning Objectives:

1. Control GAN-generated Outputs by Adding Conditional Inputs
2. Control GAN-generated Outputs by Manipulating z-Vectors
3. Explain Disentanglement in a GAN

1. Controlling GAN Outputs with Conditional Inputs

What are Conditional GANs?

Conditional GANs (cGANs) (Mirza & Osindero, 2014) extend traditional GANs by conditioning the generation process on additional information, such as class labels or other relevant data. This allows the generator to produce outputs that adhere to specific categories or attributes.

cGAN Architecture

In a cGAN, both the generator G and the discriminator D receive conditional inputs. This can be a class label y or any auxiliary information that guides the generation process.

- Generator Model: The generator can be defined as:

$$G(z, y) = \text{Generator}(z, y) \tag{8}$$

Where z is the latent vector sampled from a noise distribution and y is the conditional input.

- Discriminator Model: The discriminator evaluates whether an image is real or generated, conditioned on y:

$$D(x, y) = \text{Discriminator}(x, y) \tag{9}$$

Where x is the input image.

Loss Function for cGANs

The objective of a cGAN is to minimize the following loss functions for both the generator and discriminator:

- For the Discriminator:

$$L_D = -\mathbb{E}_{(x,y)\sim p_{data}}[\log D(x, y)] - \mathbb{E}_{(z,y)\sim p_z}[D(G(z, y), y)] \tag{10}$$

- For the Generator:

$$L_G = -\mathbb{E}_{(z,y)\sim p_z}[D(G(z,y),y)] \tag{11}$$

2. Manipulating z-Vectors for Control

Latent Space Representation

In GANs, the latent vector z serves as a representation of the features in the generated data. By manipulating this z-vector, we can control the characteristics of the generated output.

Techniques for Manipulating z-Vectors

1. Vector Arithmetic: You can perform operations in the latent space to achieve desired effects. For example, if z_a generates a "smiling" face and z_b generates a "frowning" face, then $z_{new} = z_a - z_b + z_{base}$ could produce a face that has attributes of both.
2. Interpolate in Latent Space: You can generate smooth transitions between outputs by interpolating between different z-vectors:

$$z_{interp} = (1-\alpha)z_a + \alpha z_b \tag{12}$$

   where α ranges from 0 to 1.
3. Explore the Latent Space: By sampling from different regions of the latent space, you can generate diverse outputs, allowing you to observe how different z-values affect the generated images.

3. Disentanglement in GANs

What is Disentanglement?

Disentanglement refers to the property of a generative model where different dimensions of the latent space correspond to distinct and interpretable factors of variation in the data. For instance, in a face generation task, one dimension might represent "smiling" while another represents "age."

Importance of Disentanglement

Disentangled representations enable finer control over the generated outputs. If the latent space is disentangled, it becomes easier to manipulate specific attributes of the generated images without affecting others.

Techniques for Disentangled Representations

1. InfoGAN: An extension of GANs that maximizes the mutual information between the latent code and the generated output, promoting disentangled representations.

2. Variational Autoencoders (VAEs): While not GANs, VAEs provide a structured latent space that can help understand disentanglement.
3. Supervised Learning: Incorporating supervision in training can guide the model to learn disentangled representations, where the conditioning information aids in separating features in the latent space.

Implementation of Conditional GANs in PyTorch

Here's a step-by-step outline for implementing a Conditional GAN (cGAN) using PyTorch:

1. Import Required Libraries

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.datasets as dsets
import torchvision.transforms as transforms
```

2. Define the Conditional GAN Generator and Discriminator

```
class CGANGenerator(nn.Module):
    def __init__(self, num_classes):
        super(CGANGenerator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(100 + num_classes, 256),
            nn.ReLU(),
            nn.Linear(256, 512),
            nn.ReLU(),
            nn.Linear(512, 784),
            nn.Tanh()  # Output values between -1 and 1
        )

    def forward(self, z, labels):
        # Concatenate the noise vector with the label
        z = torch.cat((z, labels), dim=1)
        return self.model(z)


class CGANDiscriminator(nn.Module):
    def __init__(self, num_classes):
        super(CGANDiscriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(784 + num_classes, 512),
            nn.LeakyReLU(0.2),
```

```
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 1)
        )

    def forward(self, img, labels):
        # Concatenate the image with the label
        img = img.view(img.size(0), -1)
        x = torch.cat((img, labels), dim=1)
        return self.model(x)
```

3. Initialize Models, Loss Function, and Optimizers

```
num_classes = 10  # For example, in the MNIST dataset
generator = CGANGenerator(num_classes)
discriminator = CGANDiscriminator(num_classes)

optimizer_G = optim.Adam(generator.parameters(), lr=0.0002, betas=(0.5,
0.999))
optimizer_D    =    optim.Adam(discriminator.parameters(),    lr=0.0002,
betas=(0.5, 0.999))
criterion = nn.BCELoss()  # Binary Cross-Entropy loss
```

4. Training Loop for cGAN

```
# Load the MNIST dataset
mnist    =    dsets.MNIST(root='./data',    train=True,    download=True,
transform=transforms.ToTensor())
data_loader    =    torch.utils.data.DataLoader(mnist,    batch_size=64,
shuffle=True)

# Training the cGAN
num_epochs = 50
for epoch in range(num_epochs):
    for real_images, labels in data_loader:
        batch_size = real_images.size(0)

        # Prepare labels for the discriminator
        real_labels = torch.ones(batch_size, 1)
        fake_labels = torch.zeros(batch_size, 1)

        # Train Discriminator
        optimizer_D.zero_grad()
                                outputs    =    discriminator(real_images,
torch.nn.functional.one_hot(labels, num_classes=num_classes).float())
```

```
        d_loss_real = criterion(outputs, real_labels)

        z = torch.randn(batch_size, 100)
         fake_images = generator(z, torch.nn.functional.one_hot(labels,
num_classes=num_classes).float())
                      outputs   =   discriminator(fake_images.detach(),
torch.nn.functional.one_hot(labels, num_classes=num_classes).float())
        d_loss_fake = criterion(outputs, fake_labels)

        d_loss = d_loss_real + d_loss_fake
        d_loss.backward()
        optimizer_D.step()

        # Train Generator
        optimizer_G.zero_grad()
                            outputs    =    discriminator(fake_images,
torch.nn.functional.one_hot(labels, num_classes=num_classes).float())
        g_loss = criterion(outputs, real_labels)
        g_loss.backward()
        optimizer_G.step()

                print(f'Epoch    [{epoch+1}/{num_epochs}],    d_loss:
{d_loss.item():.4f}, g_loss: {g_loss.item():.4f}')
```

References

Arjovsky, M., Chintala, S., & Léon Bottou. (2017). Wasserstein GAN. *ArXiv Preprint*. https://doi.org/10.48550/arxiv.1701.07875

Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., & Bengio, Y. (2014). Generative Adversarial Networks. *ArXiv Preprint*, *1*. https://doi.org/10.48550/arxiv.1406.2661

Mirza, M., & Osindero, S. (2014). *Conditional Generative Adversarial Nets*. ArXiv Preprint. https://doi.org/10.48550/arXiv.1411.1784

Radford, A., Metz, L., & Chintala, S. (2016). *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*. ArXiv Preprint. https://doi.org/10.48550/arXiv.1511.06434