Study Material - Youtube

Document Information

- **Generated:** 2025-08-20 10:50:00
- **Source:** https://youtu.be/iOb8vmJd8o
- **Platform:** Youtube
- **Word Count:** 2,820 words
- **Estimated Reading Time:** ~14 minutes
- **Number of Chapters:** 4
- **Transcript Available:** Yes (analyzed from video content)

Table of Contents

Video Overview

This practical tutorial, "W2T5: Tutorial - Implementation of GAN," bridges the gap between theoretical understanding and hands-on implementation of Generative Adversarial Networks. Prof. Prathosh A P provides a comprehensive walkthrough of implementing a GAN from scratch, covering architecture design, training loops, common pitfalls, and debugging strategies. The tutorial emphasizes the practical challenges that arise when translating the elegant mathematical theory into working code, and provides concrete solutions for stable GAN training.

Learning Objectives

Upon completing this tutorial, a student will be able to: * **Implement GAN Architecture:** Build both generator and discriminator networks using modern deep learning frameworks. * **Design Training Loops:** Create stable alternating training procedures with proper gradient flow. * **Handle Training Instabilities:** Recognize and address common training problems like mode collapse and gradient vanishing. * **Apply Best Practices:** Use proven techniques for architecture design, hyperparameter tuning, and training stabilization. * **Evaluate GAN Performance:** Implement metrics and visualization techniques to assess generation quality.

Prerequisites

To fully benefit from this tutorial, students should have: * **Deep Learning Frameworks:** Proficiency in PyTorch or TensorFlow * **Python Programming:** Strong Python skills with experience in NumPy and data manipulation * **Neural Network Implementation:** Experience building and training neural networks * **GAN Theory:** Understanding of the mathematical foundations covered in previous lectures * **Computer Vision Basics:** Familiarity with image processing and convolutional networks

Key Concepts Covered

- Complete GAN Implementation
- Architecture Design Principles
- Training Loop Implementation
- Hyperparameter Tuning Strategies
- Debugging and Troubleshooting

- Performance Evaluation Metrics

GAN Implementation Fundamentals

Complete Implementation Framework

Project Structure and Dependencies

A well-organized GAN implementation follows a modular structure:

```
gan_implementation/
  models/
     generator.py
     discriminator.py
     __init__.py
  training/
     trainer.py
     losses.py
     utils.py
  data/
     dataloader.py
     preprocessing.py
  evaluation/
     metrics.py
     visualization.py
  config/
     hyperparameters.py
  main.py
```

Essential Dependencies:

```python
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import torchvision.transforms as transforms
import torchvision.datasets as datasets
import torchvision.utils as vutils
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
```

Generator Network Implementation

The generator transforms random noise into realistic data samples:

```python
class Generator(nn.Module):
    def __init__(self, nz=100, ngf=64, nc=3):
        """
        Args:
            nz: Size of latent vector
            ngf: Generator feature map size
            nc: Number of channels in output images
        """
        super(Generator, self).__init__()
```

```python
        # Initial projection and reshape
        self.fc = nn.Linear(nz, ngf * 8 * 4 * 4)

        # Transposed convolution layers
        self.conv_blocks = nn.Sequential(
            # Layer 1: (ngf*8) x 4 x 4 -> (ngf*4) x 8 x 8
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),

            # Layer 2: (ngf*4) x 8 x 8 -> (ngf*2) x 16 x 16
            nn.ConvTranspose2d(ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),

            # Layer 3: (ngf*2) x 16 x 16 -> ngf x 32 x 32
            nn.ConvTranspose2d(ngf * 2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),

            # Layer 4: ngf x 32 x 32 -> nc x 64 x 64
            nn.ConvTranspose2d(ngf, nc, 4, 2, 1, bias=False),
            nn.Tanh()
        )

        # Weight initialization
        self.apply(self._weights_init)

    def _weights_init(self, m):
        """Initialize network weights"""
        classname = m.__class__.__name__
        if classname.find('Conv') != -1:
            nn.init.normal_(m.weight.data, 0.0, 0.02)
        elif classname.find('BatchNorm') != -1:
            nn.init.normal_(m.weight.data, 1.0, 0.02)
            nn.init.constant_(m.bias.data, 0)

    def forward(self, input):
        """Forward pass through generator"""
        # Project and reshape
        x = self.fc(input)
        x = x.view(x.size(0), -1, 4, 4)  # Reshape to (batch, channels, H, W)

        # Apply convolution blocks
        output = self.conv_blocks(x)
        return output

# Usage example
generator = Generator(nz=100, ngf=64, nc=3)
noise = torch.randn(32, 100)  # Batch of 32 noise vectors
fake_images = generator(noise)
print(f"Generated images shape: {fake_images.shape}")  # [32, 3, 64, 64]
```

Discriminator Network Implementation

The discriminator distinguishes between real and fake samples:

```python
class Discriminator(nn.Module):
    def __init__(self, nc=3, ndf=64):
        """
        Args:
            nc: Number of channels in input images
            ndf: Discriminator feature map size
        """
        super(Discriminator, self).__init__()

        self.conv_blocks = nn.Sequential(
            # Layer 1: nc x 64 x 64 -> ndf x 32 x 32
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),

            # Layer 2: ndf x 32 x 32 -> (ndf*2) x 16 x 16
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),

            # Layer 3: (ndf*2) x 16 x 16 -> (ndf*4) 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),

            # Layer 4: (ndf*4) x 8 x 8 -> (ndf*8) x 4 x 4
            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),
        )

        # Final classification layer
        self.classifier = nn.Sequential(
            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

        # Weight initialization
        self.apply(self._weights_init)

    def _weights_init(self, m):
        """Initialize network weights"""
        classname = m.__class__.__name__
        if classname.find('Conv') != -1:
            nn.init.normal_(m.weight.data, 0.0, 0.02)
        elif classname.find('BatchNorm') != -1:
            nn.init.normal_(m.weight.data, 1.0, 0.02)
            nn.init.constant_(m.bias.data, 0)

    def forward(self, input):
        """Forward pass through discriminator"""
        features = self.conv_blocks(input)
```

```python
        output = self.classifier(features)
        return output.view(-1, 1).squeeze(1)  # Flatten to [batch_size]

# Usage example
discriminator = Discriminator(nc=3, ndf=64)
real_images = torch.randn(32, 3, 64, 64)
predictions = discriminator(real_images)
print(f"Discriminator output shape: {predictions.shape}")  # [32]
```

Loss Functions and Optimization

Standard GAN Loss Implementation:

```python
class GANLoss:
    def __init__(self, device):
        self.device = device
        self.criterion = nn.BCELoss()

        # Labels for real and fake data
        self.real_label = 1.0
        self.fake_label = 0.0

    def discriminator_loss(self, real_output, fake_output):
        """Compute discriminator loss"""
        batch_size = real_output.size(0)

        # Real data loss
        real_labels = torch.full((batch_size,), self.real_label,
                                 dtype=torch.float, device=self.device)
        loss_real = self.criterion(real_output, real_labels)

        # Fake data loss
        fake_labels = torch.full((batch_size,), self.fake_label,
                                 dtype=torch.float, device=self.device)
        loss_fake = self.criterion(fake_output, fake_labels)

        return loss_real + loss_fake, loss_real, loss_fake

    def generator_loss(self, fake_output):
        """Compute generator loss"""
        batch_size = fake_output.size(0)

        # Generator wants discriminator to think fakes are real
        real_labels = torch.full((batch_size,), self.real_label,
                                 dtype=torch.float, device=self.device)
        loss_g = self.criterion(fake_output, real_labels)

        return loss_g
```

Alternative Generator Loss (Non-saturating):

```python
def non_saturating_generator_loss(fake_output):
    """
    Alternative generator loss to avoid vanishing gradients
    max log(D(G(z))) instead of min log(1-D(G(z)))
    """
```

```python
    return -torch.mean(torch.log(fake_output + 1e-8))  # Add epsilon for numerical stability
```

Data Loading and Preprocessing

Dataset Preparation

```python
class GANDataLoader:
    def __init__(self, dataset_path, image_size=64, batch_size=128):
        self.dataset_path = dataset_path
        self.image_size = image_size
        self.batch_size = batch_size

        # Define transforms
        self.transforms = transforms.Compose([
            transforms.Resize(image_size),
            transforms.CenterCrop(image_size),
            transforms.ToTensor(),
            transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  # Normalize to [-1, 1]
        ])

    def get_dataloader(self):
        """Create and return dataloader"""
        # For common datasets
        if self.dataset_path == 'CIFAR10':
            dataset = datasets.CIFAR10(
                root='./data', train=True, download=True,
                transform=self.transforms
            )
        elif self.dataset_path == 'CelebA':
            dataset = datasets.ImageFolder(
                root=self.dataset_path,
                transform=self.transforms
            )
        else:
            # Custom dataset
            dataset = datasets.ImageFolder(
                root=self.dataset_path,
                transform=self.transforms
            )

        dataloader = torch.utils.data.DataLoader(
            dataset, batch_size=self.batch_size,
            shuffle=True, num_workers=4, drop_last=True
        )

        return dataloader

# Usage
data_loader = GANDataLoader('CIFAR10', image_size=64, batch_size=128)
dataloader = data_loader.get_dataloader()
```

Noise Generation Utilities

```python
class NoiseGenerator:
    def __init__(self, noise_dim=100, device='cuda'):
```

```python
        self.noise_dim = noise_dim
        self.device = device

    def generate_noise(self, batch_size, distribution='normal'):
        """Generate random noise vectors"""
        if distribution == 'normal':
            return torch.randn(batch_size, self.noise_dim, device=self.device)
        elif distribution == 'uniform':
            return torch.rand(batch_size, self.noise_dim, device=self.device) * 2 - 1
        else:
            raise ValueError(f"Unknown distribution: {distribution}")

    def generate_fixed_noise(self, num_samples):
        """Generate fixed noise for consistent evaluation"""
        torch.manual_seed(42)  # For reproducibility
        fixed_noise = torch.randn(num_samples, self.noise_dim, device=self.device)
        return fixed_noise

# Usage
noise_gen = NoiseGenerator(noise_dim=100)
batch_noise = noise_gen.generate_noise(32)
fixed_noise = noise_gen.generate_fixed_noise(64)
```

Training Strategies and Best Practices

Complete Training Loop Implementation

Main Training Class

```python
class GANTrainer:
    def __init__(self, generator, discriminator, dataloader, config):
        self.generator = generator
        self.discriminator = discriminator
        self.dataloader = dataloader
        self.config = config
        self.device = config.device

        # Move models to device
        self.generator.to(self.device)
        self.discriminator.to(self.device)

        # Initialize optimizers
        self.opt_g = optim.Adam(
            self.generator.parameters(),
            lr=config.lr_g, betas=(config.beta1, 0.999)
        )
        self.opt_d = optim.Adam(
            self.discriminator.parameters(),
            lr=config.lr_d, betas=(config.beta1, 0.999)
        )

        # Loss function
        self.criterion = GANLoss(self.device)
```

```python
        self.noise_gen = NoiseGenerator(config.noise_dim, self.device)

        # Fixed noise for evaluation
        self.fixed_noise = self.noise_gen.generate_fixed_noise(64)

        # Training statistics
        self.g_losses = []
        self.d_losses = []
        self.d_real_acc = []
        self.d_fake_acc = []

    def train_discriminator(self, real_batch):
        """Train discriminator for one step"""
        batch_size = real_batch.size(0)

        # Clear gradients
        self.opt_d.zero_grad()

        # Train on real data
        real_output = self.discriminator(real_batch)

        # Train on fake data
        noise = self.noise_gen.generate_noise(batch_size)
        fake_batch = self.generator(noise).detach()  # Detach to avoid training generator
        fake_output = self.discriminator(fake_batch)

        # Compute loss
        d_loss, d_loss_real, d_loss_fake = self.criterion.discriminator_loss(
            real_output, fake_output
        )

        # Backward pass and optimize
        d_loss.backward()
        self.opt_d.step()

        # Compute accuracy
        real_acc = (real_output > 0.5).float().mean()
        fake_acc = (fake_output < 0.5).float().mean()

        return d_loss.item(), real_acc.item(), fake_acc.item()

    def train_generator(self, batch_size):
        """Train generator for one step"""
        # Clear gradients
        self.opt_g.zero_grad()

        # Generate fake data
        noise = self.noise_gen.generate_noise(batch_size)
        fake_batch = self.generator(noise)
        fake_output = self.discriminator(fake_batch)

        # Compute loss
        g_loss = self.criterion.generator_loss(fake_output)
```

```python
        # Backward pass and optimize
        g_loss.backward()
        self.opt_g.step()

        return g_loss.item()

    def train_epoch(self, epoch):
        """Train for one epoch"""
        epoch_d_loss = 0
        epoch_g_loss = 0
        epoch_d_real_acc = 0
        epoch_d_fake_acc = 0
        num_batches = len(self.dataloader)

        progress_bar = tqdm(self.dataloader, desc=f'Epoch {epoch}')

        for i, (real_batch, _) in enumerate(progress_bar):
            real_batch = real_batch.to(self.device)
            batch_size = real_batch.size(0)

            # Train Discriminator
            d_loss, d_real_acc, d_fake_acc = self.train_discriminator(real_batch)

            # Train Generator (less frequently to balance training)
            if i % self.config.g_update_freq == 0:
                g_loss = self.train_generator(batch_size)
            else:
                g_loss = 0

            # Update statistics
            epoch_d_loss += d_loss
            epoch_g_loss += g_loss
            epoch_d_real_acc += d_real_acc
            epoch_d_fake_acc += d_fake_acc

            # Update progress bar
            progress_bar.set_postfix({
                'D_loss': f'{d_loss:.4f}',
                'G_loss': f'{g_loss:.4f}',
                'D_acc': f'{(d_real_acc + d_fake_acc) / 2:.4f}'
            })

        # Average statistics
        avg_d_loss = epoch_d_loss / num_batches
        avg_g_loss = epoch_g_loss / (num_batches // self.config.g_update_freq)
        avg_d_real_acc = epoch_d_real_acc / num_batches
        avg_d_fake_acc = epoch_d_fake_acc / num_batches

        return avg_d_loss, avg_g_loss, avg_d_real_acc, avg_d_fake_acc

    def train(self, num_epochs):
        """Full training procedure"""
        for epoch in range(num_epochs):
            # Train for one epoch
```

```python
            d_loss, g_loss, d_real_acc, d_fake_acc = self.train_epoch(epoch)

            # Store statistics
            self.d_losses.append(d_loss)
            self.g_losses.append(g_loss)
            self.d_real_acc.append(d_real_acc)
            self.d_fake_acc.append(d_fake_acc)

            # Generate samples for evaluation
            if epoch % self.config.eval_freq == 0:
                self.evaluate(epoch)

            # Save checkpoint
            if epoch % self.config.save_freq == 0:
                self.save_checkpoint(epoch)

            print(f'Epoch {epoch}: D_loss={d_loss:.4f}, G_loss={g_loss:.4f}, '
                  f'D_real_acc={d_real_acc:.4f}, D_fake_acc={d_fake_acc:.4f}')

    def evaluate(self, epoch):
        """Evaluate model and generate samples"""
        self.generator.eval()
        with torch.no_grad():
            fake_samples = self.generator(self.fixed_noise)

            # Save sample images
            vutils.save_image(
                fake_samples,
                f'samples/epoch_{epoch:04d}.png',
                normalize=True, nrow=8
            )
        self.generator.train()

    def save_checkpoint(self, epoch):
        """Save model checkpoint"""
        checkpoint = {
            'epoch': epoch,
            'generator_state_dict': self.generator.state_dict(),
            'discriminator_state_dict': self.discriminator.state_dict(),
            'opt_g_state_dict': self.opt_g.state_dict(),
            'opt_d_state_dict': self.opt_d.state_dict(),
            'g_losses': self.g_losses,
            'd_losses': self.d_losses,
        }
        torch.save(checkpoint, f'checkpoints/epoch_{epoch:04d}.pth')
```

Training Configuration

```python
class Config:
    def __init__(self):
        # Model parameters
        self.noise_dim = 100
        self.ngf = 64  # Generator feature map size
        self.ndf = 64  # Discriminator feature map size
```

```python
        self.nc = 3    # Number of channels

        # Training parameters
        self.lr_g = 0.0002      # Generator learning rate
        self.lr_d = 0.0002      # Discriminator learning rate
        self.beta1 = 0.5        # Adam optimizer beta1
        self.batch_size = 128
        self.num_epochs = 200

        # Training strategy
        self.g_update_freq = 1  # Update generator every n discriminator updates

        # Evaluation and saving
        self.eval_freq = 10     # Generate samples every n epochs
        self.save_freq = 50     # Save checkpoint every n epochs

        # Device
        self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Usage
config = Config()
```

Advanced Training Techniques

Learning Rate Scheduling

```python
def create_lr_scheduler(optimizer, config):
    """Create learning rate scheduler"""
    if config.scheduler_type == 'cosine':
        return optim.lr_scheduler.CosineAnnealingLR(
            optimizer, T_max=config.num_epochs
        )
    elif config.scheduler_type == 'step':
        return optim.lr_scheduler.StepLR(
            optimizer, step_size=config.step_size, gamma=config.gamma
        )
    else:
        return None

# Add to trainer
self.scheduler_g = create_lr_scheduler(self.opt_g, config)
self.scheduler_d = create_lr_scheduler(self.opt_d, config)

# Update in training loop
if self.scheduler_g:
    self.scheduler_g.step()
if self.scheduler_d:
    self.scheduler_d.step()
```

Gradient Clipping for Stability

```python
def train_generator(self, batch_size):
    """Train generator with gradient clipping"""
    self.opt_g.zero_grad()
```

```python
    noise = self.noise_gen.generate_noise(batch_size)
    fake_batch = self.generator(noise)
    fake_output = self.discriminator(fake_batch)

    g_loss = self.criterion.generator_loss(fake_output)
    g_loss.backward()

    # Gradient clipping
    torch.nn.utils.clip_grad_norm_(self.generator.parameters(), max_norm=1.0)

    self.opt_g.step()
    return g_loss.item()
```

Spectral Normalization

```python
from torch.nn.utils import spectral_norm

class SpectralNormDiscriminator(nn.Module):
    """Discriminator with spectral normalization for training stability"""
    def __init__(self, nc=3, ndf=64):
        super(SpectralNormDiscriminator, self).__init__()

        self.conv_blocks = nn.Sequential(
            # Apply spectral normalization to all convolutional layers
            spectral_norm(nn.Conv2d(nc, ndf, 4, 2, 1, bias=False)),
            nn.LeakyReLU(0.2, inplace=True),

            spectral_norm(nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False)),
            nn.LeakyReLU(0.2, inplace=True),

            spectral_norm(nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False)),
            nn.LeakyReLU(0.2, inplace=True),

            spectral_norm(nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False)),
            nn.LeakyReLU(0.2, inplace=True),
        )

        self.classifier = nn.Sequential(
            spectral_norm(nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False)),
            nn.Sigmoid()
        )

    def forward(self, input):
        features = self.conv_blocks(input)
        output = self.classifier(features)
        return output.view(-1, 1).squeeze(1)
```

Evaluation and Monitoring

Comprehensive Evaluation Metrics

```python
import numpy as np
from scipy import linalg
from torchvision.models import inception_v3
```

```python
class GANEvaluator:
    def __init__(self, device):
        self.device = device
        self.inception_model = inception_v3(pretrained=True, transform_input=False)
        self.inception_model.fc = nn.Identity()  # Remove final classifier
        self.inception_model.to(device)
        self.inception_model.eval()

    def calculate_inception_score(self, generated_images, batch_size=32, splits=10):
        """Calculate Inception Score (IS)"""
        N = len(generated_images)
        preds = []

        for i in range(0, N, batch_size):
            batch = generated_images[i:i+batch_size]
            batch = F.interpolate(batch, size=(299, 299), mode='bilinear', align_corners=False)

            with torch.no_grad():
                pred = F.softmax(self.inception_model(batch), dim=1)
                preds.append(pred.cpu().numpy())

        preds = np.concatenate(preds, axis=0)

        # Calculate IS
        scores = []
        for i in range(splits):
            part = preds[(i * N // splits):((i + 1) * N // splits)]
            kl = part * (np.log(part) - np.log(np.expand_dims(np.mean(part, axis=0), 0)))
            kl = np.mean(np.sum(kl, axis=1))
            scores.append(np.exp(kl))

        return np.mean(scores), np.std(scores)

    def calculate_fid_score(self, real_images, generated_images):
        """Calculate Fréchet Inception Distance (FID)"""

        def get_activations(images):
            activations = []
            with torch.no_grad():
                for i in range(0, len(images), 32):
                    batch = images[i:i+32]
                    batch = F.interpolate(batch, size=(299, 299), mode='bilinear', align_corners=False)
                    acts = self.inception_model(batch)
                    activations.append(acts.cpu().numpy())
            return np.concatenate(activations, axis=0)

        # Get activations
        real_acts = get_activations(real_images)
        fake_acts = get_activations(generated_images)

        # Calculate statistics
        mu1, sigma1 = real_acts.mean(axis=0), np.cov(real_acts, rowvar=False)
        mu2, sigma2 = fake_acts.mean(axis=0), np.cov(fake_acts, rowvar=False)
```

```python
        # Calculate FID
        diff = mu1 - mu2
        covmean = linalg.sqrtm(sigma1.dot(sigma2))

        if np.iscomplexobj(covmean):
            covmean = covmean.real

        fid = diff.dot(diff) + np.trace(sigma1) + np.trace(sigma2) - 2 * np.trace(covmean)
        return fid
```

Training Monitoring and Visualization

```python
import matplotlib.pyplot as plt

class TrainingMonitor:
    def __init__(self):
        self.losses = {'generator': [], 'discriminator': []}
        self.accuracies = {'real': [], 'fake': []}

    def update(self, g_loss, d_loss, d_real_acc, d_fake_acc):
        self.losses['generator'].append(g_loss)
        self.losses['discriminator'].append(d_loss)
        self.accuracies['real'].append(d_real_acc)
        self.accuracies['fake'].append(d_fake_acc)

    def plot_training_curves(self, save_path='training_curves.png'):
        fig, axes = plt.subplots(2, 2, figsize=(12, 10))

        # Loss curves
        axes[0, 0].plot(self.losses['generator'], label='Generator')
        axes[0, 0].plot(self.losses['discriminator'], label='Discriminator')
        axes[0, 0].set_title('Training Losses')
        axes[0, 0].set_xlabel('Epoch')
        axes[0, 0].set_ylabel('Loss')
        axes[0, 0].legend()
        axes[0, 0].grid(True)

        # Accuracy curves
        axes[0, 1].plot(self.accuracies['real'], label='Real Data Accuracy')
        axes[0, 1].plot(self.accuracies['fake'], label='Fake Data Accuracy')
        axes[0, 1].set_title('Discriminator Accuracy')
        axes[0, 1].set_xlabel('Epoch')
        axes[0, 1].set_ylabel('Accuracy')
        axes[0, 1].legend()
        axes[0, 1].grid(True)

        # Loss ratio
        loss_ratio = np.array(self.losses['generator']) / np.array(self.losses['discriminator'])
        axes[1, 0].plot(loss_ratio)
        axes[1, 0].set_title('Generator/Discriminator Loss Ratio')
        axes[1, 0].set_xlabel('Epoch')
        axes[1, 0].set_ylabel('Ratio')
        axes[1, 0].grid(True)
```

```
    # Overall discriminator accuracy
    overall_acc = (np.array(self.accuracies['real']) + np.array(self.accuracies['fake'])) / 2
    axes[1, 1].plot(overall_acc)
    axes[1, 1].set_title('Overall Discriminator Accuracy')
    axes[1, 1].set_xlabel('Epoch')
    axes[1, 1].set_ylabel('Accuracy')
    axes[1, 1].grid(True)

    plt.tight_layout()
    plt.savefig(save_path)
    plt.show()
```

Key Takeaways from This Video

- **Implementation Complexity:** Translating GAN theory into stable, working code requires careful attention to architecture design, initialization, and training dynamics.
- **Training Balance:** The key to successful GAN training lies in maintaining the right balance between generator and discriminator capabilities.
- **Best Practices:** Modern techniques like spectral normalization, proper weight initialization, and gradient clipping significantly improve training stability.
- **Evaluation Challenges:** Assessing GAN performance requires multiple metrics (IS, FID) beyond simple loss values due to the adversarial nature of training.
- **Hyperparameter Sensitivity:** GAN training is highly sensitive to hyperparameters, requiring systematic tuning and monitoring.
- **Debugging Skills:** Understanding common failure modes (mode collapse, training instability) is crucial for successful implementation.

Self-Assessment for This Video

1. **Architecture Design:** Explain the key design choices in the generator and discriminator architectures. Why is batch normalization used differently in each network?

2. **Training Loop Implementation:** Write a simplified version of the GAN training loop in pseudocode. What are the key steps for each network update?

3. **Loss Function Analysis:** Compare the standard GAN loss with the non-saturating generator loss. When would you choose one over the other?

4. **Stability Techniques:** List and explain at least three techniques for improving GAN training stability. How does each technique address specific problems?

5. **Evaluation Metrics:** Describe how to implement Inception Score and FID. What aspects of generation quality does each metric capture?

6. **Common Failure Modes:** Identify three common problems in GAN training and describe how to detect and address each one.

7. **Hyperparameter Tuning:** What are the most critical hyperparameters in GAN training? How would you systematically tune them?

8. **Code Organization:** Outline a modular code structure for a GAN implementation. What components should be separated and why?