Generative Adversarial Network (GAN) Cheat Sheet

1 Key Concepts

1. **Generator (G)**: A neural network that generates fake samples from random noise.

$$G : z \to x, \quad z \sim p_z(z), \quad x = G(z)$$

2. **Discriminator (D)**: A neural network that distinguishes real samples from fake ones.

$$D : x \to [0, 1], \quad D(x) = \text{probability that x is real}$$

3. **Adversarial Training**: G and D are trained simultaneously in a minimax game.

4. **Nash Equilibrium**: The optimal point where G produces perfect fakes and D can't distinguish them (D(x) = 0.5 for all x).

2 GAN Objective Function

The GAN objective function is a two-player minimax game:

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

Where:

- $p_{\text{data}}(x)$ is the real data distribution
- $p_z(z)$ is the prior on input noise variables
- $G(z)$ is the generator's distribution over data x

3 Training Process

1. **Discriminator Training**:

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

2. **Generator Training**:

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

Often, we maximize $\mathbb{E}_{z \sim p_z(z)}[\log D(G(z))]$ instead to provide stronger gradients early in training.

4 Theoretical Foundations

4.1 Optimal Discriminator

For a fixed generator G, the optimal discriminator D is:

$$D^*_G(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)}$$

Where p_g is the distribution of samples generated by G.

4.2 Global Optimum

The global minimum of the virtual training criterion C(G) is achieved if and only if $p_g = p_{\text{data}}$. At that point, C(G) achieves the value $-\log 4$.

4.3 Convergence

If G and D have enough capacity, and at each step of training, the discriminator is allowed to reach its optimum given G, and p_g is updated so as to improve the criterion:

$$\mathbb{E}_{x\sim p_{\text{data}}}[\log D_G^*(x)] + \mathbb{E}_{x\sim p_g}[\log(1 - D_G^*(x))]$$

Then p_g converges to p_{data}.

5 Practical Tips

1. **Architecture**:
   - Use strided convolutions (discriminator) and fractional-strided convolutions (generator)
   - Use batchnorm in both generator and discriminator
   - Use ReLU activation in generator for all layers except for the output, which uses Tanh
   - Use LeakyReLU activation in the discriminator for all layers
2. **Training Tricks**:
   - Use label smoothing (e.g., use soft labels 0.9 and 0.1 instead of 1 and 0)
   - Add noise to inputs of discriminator
   - Use mini-batch discrimination
   - Update G twice for each D update
   - Use different learning rates for G and D
3. **Loss Functions**:
   - Original GAN: Binary cross-entropy
   - Wasserstein GAN: Wasserstein distance
   - Least Squares GAN: L2 loss
4. **Evaluation Metrics**:
   - Inception Score (IS)
   - Fréchet Inception Distance (FID)
   - Kernel Inception Distance (KID)

6 Common GAN Variants

1. **DCGAN (Deep Convolutional GAN)**:
   - Uses convolutional and convolutional-transpose layers
   - Eliminates fully connected layers for deeper architectures
2. **WGAN (Wasserstein GAN)**:
   - Uses Wasserstein distance instead of Jensen-Shannon divergence
   - Improves stability of training
   - Objective: $\min_G \max_{D\in\mathcal{D}} \mathbb{E}_{x\sim p_{\text{data}}(x)}[D(x)] - \mathbb{E}_{z\sim p(z)}[D(G(z))]$
3. **CGAN (Conditional GAN)**:
   - Incorporates additional information (e.g., class labels) in both G and D
   - Enables controlled generation of samples

4. **CycleGAN**:
    - Performs unpaired image-to-image translation
    - Uses cycle consistency loss
5. **Progressive GAN**:
    - Grows both G and D progressively, starting from low resolution
    - Improves stability and quality of high-resolution image generation
6. **BigGAN**:
    - Scales up GANs to generate high-fidelity natural images
    - Uses large batch sizes and high-dimensional noise vectors
7. **StyleGAN**:
    - Introduces style-based generator architecture
    - Enables better control over generated image features

7 Challenges and Open Problems

1. **Mode Collapse**: Generator produces limited varieties of samples.
2. **Training Instability**: Difficult to achieve Nash equilibrium.
3. **Evaluation**: Lack of a single comprehensive metric for GAN performance.
4. **Discrete Data**: GANs struggle with generating discrete data (e.g., text).
5. **Interpretability**: Understanding and controlling what GANs learn.

8 Recent Developments

- **Self-Attention GANs**: Incorporating attention mechanisms for long-range dependencies.
- **GAN Inversion**: Techniques to project real images back into the latent space.
- **Few-shot GANs**: Generating samples from limited training data.
- **Energy-based GANs**: Formulating GANs in an energy-based framework.
- **Differentiable Augmentation**: Improving GAN training with differentiable data augmentation.

9 Applications

- Image Generation
- Image-to-Image Translation
- Text-to-Image Synthesis
- Super-Resolution
- Video Generation
- Music Generation
- Drug Discovery
- Data Augmentation
- Anomaly Detection

[Previous content remains unchanged]

10 PyTorch Implementation

Here's a complete PyTorch implementation of a basic GAN for generating MNIST-like digits:

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader

# Set random seed for reproducibility
torch.manual_seed(999)

# Hyperparameters
batch_size = 64
lr = 0.0002
num_epochs = 100
latent_dim = 100

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

# MNIST Dataset
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5], std=[0.5])
])

mnist = torchvision.datasets.MNIST(root='./data', train=True, transform=transform, download=True)
dataloader = DataLoader(mnist, batch_size=batch_size, shuffle=True)

# Generator
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(latent_dim, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 1024),
            nn.LeakyReLU(0.2),
            nn.Linear(1024, 784),
            nn.Tanh()
        )

    def forward(self, z):
        img = self.model(z)
        img = img.view(img.size(0), 1, 28, 28)
        return img

# Discriminator
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(784, 1024),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),
            nn.Linear(1024, 512),
```

```
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),
            nn.Linear(256, 1),
            nn.Sigmoid()
        )

    def forward(self, img):
        img_flat = img.view(img.size(0), -1)
        validity = self.model(img_flat)
        return validity

# Initialize generator and discriminator
generator = Generator().to(device)
discriminator = Discriminator().to(device)

# Loss function
adversarial_loss = nn.BCELoss()

# Optimizers
optimizer_G = optim.Adam(generator.parameters(), lr=lr, betas=(0.5, 0.999))
optimizer_D = optim.Adam(discriminator.parameters(), lr=lr, betas=(0.5, 0.999))

# Training loop
for epoch in range(num_epochs):
    for i, (imgs, _) in enumerate(dataloader):

        # Adversarial ground truths
        real = torch.ones(imgs.size(0), 1).to(device)
        fake = torch.zeros(imgs.size(0), 1).to(device)

        # Configure input
        real_imgs = imgs.to(device)

        # -----------------
        #  Train Generator
        # -----------------

        optimizer_G.zero_grad()

        # Sample noise as generator input
        z = torch.randn(imgs.size(0), latent_dim).to(device)

        # Generate a batch of images
        gen_imgs = generator(z)

        # Loss measures generator's ability to fool the discriminator
        g_loss = adversarial_loss(discriminator(gen_imgs), real)

        g_loss.backward()
        optimizer_G.step()

        # ---------------------
        #  Train Discriminator
        # ---------------------

        optimizer_D.zero_grad()

        # Measure discriminator's ability to classify real from generated samples
```

```
        real_loss = adversarial_loss(discriminator(real_imgs), real)
        fake_loss = adversarial_loss(discriminator(gen_imgs.detach()), fake)
        d_loss = (real_loss + fake_loss) / 2

        d_loss.backward()
        optimizer_D.step()

        # Print progress
        if i % 100 == 0:
            print(f"[Epoch {epoch}/{num_epochs}] [Batch {i}/{len(dataloader)}] "
                  f"[D loss: {d_loss.item():.4f}] [G loss: {g_loss.item():.4f}]")

    # Save generated images
    if (epoch + 1) % 10 == 0:
        torchvision.utils.save_image(gen_imgs.data[:25], f"images/epoch_{epoch+1}.png",
                                     nrow=5, normalize=True)

print("Training finished!")
```

This implementation includes:

- Generator and Discriminator network architectures
- Data loading and preprocessing for MNIST dataset
- Training loop with separate optimization for Generator and Discriminator
- Periodic saving of generated images to monitor progress

Key points in the implementation:

1. The Generator transforms random noise into fake images.
2. The Discriminator tries to distinguish between real and fake images.
3. We use Binary Cross Entropy loss for both networks.
4. The training alternates between updating the Discriminator and the Generator.
5. We use the Adam optimizer with $\beta_1 = 0.5$ and $\beta_2 = 0.999$, which are common choices for GANs.

To run this code, ensure you have PyTorch and torchvision installed. The code will download the MNIST dataset automatically if it's not present in the './data' directory.

Remember that GANs can be sensitive to hyperparameters and initialization. You might need to experiment with different architectures, learning rates, or training procedures to get the best results for your specific use case.

11 Conclusion

Generative Adversarial Networks have revolutionized the field of generative modeling, enabling the creation of highly realistic synthetic data across various domains. Despite challenges in training and evaluation, GANs continue to be a vibrant area of research with numerous practical applications. As the field progresses, we can expect GANs to play an increasingly important role in artificial intelligence and machine learning.