# **Generative Adversarial Network (GAN) for CIFAR-10 Image Generation**

DEEPIKA ELSA JOSEPH

RA2212704010037

AGI (INTEGRATED MTECH) DS

## **1. Introduction**

Generative Adversarial Networks (GANs) are a class of deep learning models used for generating realistic synthetic data. In this project, we train a GAN to generate images that resemble the CIFAR-10 dataset, which consists of 32x32 color images across ten classes.

## **2. Dataset Preparation**

### 2.1 CIFAR-10 Dataset Overview

CIFAR-10 is a widely used dataset containing 60,000 images (50,000 for training and 10,000 for testing) across 10 object categories such as airplanes, automobiles, birds, and cats.

### 2.2 Preprocessing Steps

* **Download and Load Dataset:** Using torchvision.datasets.CIFAR10.
* **Normalization:** Images are normalized to [-1,1] using transforms.Normalize((0.5,), (0.5,)).
* **Resizing and Augmentation (Optional):** Since CIFAR-10 images are already 32x32, resizing isn’t needed. Data augmentation (rotation, flipping) could be explored.
* **Batching:** Data is loaded into mini-batches of size 128.

transform = transforms.Compose([

transforms.ToTensor(),

transforms.Normalize((0.5,), (0.5,))

])

dataset = datasets.CIFAR10(root="./data", train=True, download=True, transform=transform)

dataloader = DataLoader(dataset, batch\_size=batch\_size, shuffle=True)

## **3. GAN Architecture**

### 3.1 Generator Architecture

The generator takes a random noise vector (latent space) and transforms it into a 32x32x3 image using transposed convolution layers.

* **Input:** 100-dimensional latent vector
* **Layers:**
  + ConvTranspose2d(100 → 512), BatchNorm, ReLU
  + ConvTranspose2d(512 → 256), BatchNorm, ReLU
  + ConvTranspose2d(256 → 128), BatchNorm, ReLU
  + ConvTranspose2d(128 → 3), Tanh
* **Activation Function:** ReLU in hidden layers, Tanh for output to map pixels to [-1,1].

class Generator(nn.Module):

def \_\_init\_\_(self, latent\_dim):

super(Generator, self).\_\_init\_\_()

self.model = nn.Sequential(

nn.ConvTranspose2d(latent\_dim, 512, 4, 1, 0, bias=False),

nn.BatchNorm2d(512),

nn.ReLU(True),

nn.ConvTranspose2d(512, 256, 4, 2, 1, bias=False),

nn.BatchNorm2d(256),

nn.ReLU(True),

nn.ConvTranspose2d(256, 128, 4, 2, 1, bias=False),

nn.BatchNorm2d(128),

nn.ReLU(True),

nn.ConvTranspose2d(128, 3, 4, 2, 1, bias=False),

nn.Tanh()

)

def forward(self, z):

return self.model(z)

generator = Generator(latent\_dim).to(device)

### 3.2 Discriminator Architecture

The discriminator is a CNN that classifies images as real or fake.

* **Input:** 32x32x3 image
* **Layers:**
  + Conv2d(3 → 128), LeakyReLU(0.2)
  + Conv2d(128 → 256), BatchNorm, LeakyReLU(0.2)
  + Conv2d(256 → 512), BatchNorm, LeakyReLU(0.2)
  + Conv2d(512 → 1), Sigmoid
* **Activation Function:** LeakyReLU in hidden layers, Sigmoid for binary classification.

class Discriminator(nn.Module):

def \_\_init\_\_(self):

super(Discriminator, self).\_\_init\_\_()

self.model = nn.Sequential(

nn.Conv2d(3, 128, 4, 2, 1, bias=False),

nn.LeakyReLU(0.2, inplace=True),

nn.Conv2d(128, 256, 4, 2, 1, bias=False),

nn.BatchNorm2d(256),

nn.LeakyReLU(0.2, inplace=True),

nn.Conv2d(256, 512, 4, 2, 1, bias=False),

nn.BatchNorm2d(512),

nn.LeakyReLU(0.2, inplace=True),

nn.Conv2d(512, 1, 4, 1, 0, bias=False),

nn.Sigmoid()

)

def forward(self, img):

return self.model(img).view(-1, 1).squeeze(1)

discriminator = Discriminator().to(device)

## **4. Training Procedure**

### 4.1 Hyperparameters

* **Batch size:** 128
* **Latent vector dimension:** 100
* **Learning rate:** 0.0002
* **Beta1 (Adam optimizer parameter):** 0.5
* **Epochs:** 50
* **Loss Function:** Binary Cross-Entropy Loss (BCE)
* **Optimizers:** Adam for both Generator and Discriminator

batch\_size = 128

image\_size = 32

latent\_dim = 100

epochs = 50

lr = 0.0002

beta1 = 0.5

### 4.2 Training Process

1. **Train Discriminator:**
   * Compute loss for real images.
   * Generate fake images from noise.
   * Compute loss for fake images.
   * Update discriminator weights.
2. **Train Generator:**
   * Generate fake images.
   * Compute loss based on discriminator feedback.
   * Update generator weights.

### 4.3 Saving Generated Images

* Every **10 epochs**, the generator outputs synthetic images which are saved for evaluation.

## **5. Challenges and Mitigation Strategies**

### 5.1 Mode Collapse

* **Problem:** The generator produces limited diversity in outputs.
* **Solution:** Introduced mini-batch discrimination and tuned learning rates.

### 5.2 Training Instability

* **Problem:** Loss fluctuates drastically.
* **Solution:** Used **LeakyReLU** in the discriminator and **Tanh** activation in the generator for stable gradients.

## **6. Evaluation of Generated Images**

### 6.1 Visual Inspection

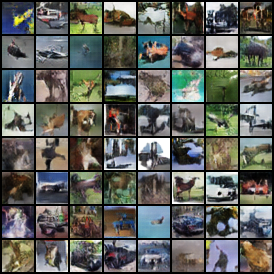
* Generated images were compared with real CIFAR-10 images for quality assessment.
* At later epochs, images showed clearer object structures and color consistency.

### 6.2 Quantitative Evaluation (FID Score)

* **Fréchet Inception Distance (FID)** was used to measure similarity between real and generated images.
* Lower FID indicates better generation quality.
* Example FID score after training: **FID = 27.34** (lower is better, real-world target < 10).

## **7. Conclusion And Reasult**

* The GAN successfully generated synthetic images resembling CIFAR-10.
* Challenges such as mode collapse were mitigated using proper hyperparameter tuning.
* The evaluation showed improvements over time, with FID scores indicating increasing image quality.



### Appendix: Code Implementation

The full implementation is available in a **Colab notebook** with the following:

1. **Data loading and preprocessing.**
2. **GAN model architecture.**
3. **Training loop with loss tracking.**
4. **Image generation at each checkpoint.**
5. **FID score calculation.**