# Generative Adversarial Networks for Image Generation on the MNIST Dataset

Kelian Schulz

**Abstract**

In this paper, we implement a Generative Adversarial Network (GAN) to generate images of handwritten digits from the MNIST dataset. GANs consist of two competing neural networks, the Generator and the Discriminator, which are trained to improve simultaneously through adversarial training. The Generator learns to produce realistic images, while the Discriminator learns to distinguish real from fake data. This paper explores the architecture, training process, and challenges involved in training GANs.

## 1 Introduction

Generative Adversarial Networks (GANs), introduced by Goodfellow et al. (2014), have revolutionized the field of generative modeling by enabling the generation of realistic data from random noise. The network consists of two components:

- **Generator** ($G$): A neural network that takes a random noise vector $\mathbf{z}$ and transforms it into a synthetic image.
- **Discriminator** ($D$): A neural network that takes an image as input and outputs a probability that the image is real (from the training set) or fake (generated by $G$).

The core idea behind GANs is that both networks are trained simultaneously in a minimax game. The Generator tries to produce data that resembles the real dataset, while the Discriminator tries to distinguish real data from generated data. The loss function for this game is:

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

Where: - $p_{\text{data}}(x)$ is the distribution of real data. - $p_z(z)$ is the distribution of input noise. - $G(z)$ is the output of the Generator.

The goal is for the Generator to learn to produce realistic images that the Discriminator can no longer distinguish from real ones.

# 2 Loss Function Explanation

The training of a Generative Adversarial Network (GAN) is driven by a two-player min-max game between the Generator ($G$) and the Discriminator ($D$). The loss function for this game can be written as:

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

Where: - $p_{\text{data}}(x)$ represents the distribution of real data, which comes from the training dataset. - $p_z(z)$ represents the distribution of input noise $z$, which is a random vector sampled from a uniform or normal distribution. - $G(z)$ is the output of the Generator, which is a synthetic image generated from the input noise $z$.

## 2.1 Discriminator's Objective

The Discriminator's goal is to correctly classify real images as real and fake images as fake. It does this by maximizing the following objective:

$$E_{x\sim p_{\text{data}}(x)}[\log D(x)] + E_{z\sim p_z(z)}[\log(1 - D(G(z)))]$$

Where: - The first term $E_{x\sim p_{\text{data}}(x)}[\log D(x)]$ penalizes the Discriminator if it incorrectly classifies real images as fake. The Discriminator outputs $D(x)$, the probability that an image $x$ is real, and the goal is to maximize this probability for real images. - The second term $E_{z\sim p_z(z)}[\log(1-D(G(z)))]$ penalizes the Discriminator for incorrectly classifying fake images (generated by $G$) as real. The Discriminator should assign a probability close to 0 to generated images, and the term encourages it to do so.

The Discriminator's objective is to maximize the log-probabilities of real images being real and fake images being fake.

### 2.2 Generator's Objective

The Generator's goal is to generate realistic images that can fool the Discriminator into thinking they are real. The Generator seeks to minimize the following objective:

$$E_{z \sim p_z(z)}[\log(1 - D(G(z)))]$$

However, this objective suffers from the issue of vanishing gradients, which makes learning difficult when the Discriminator is too confident. To mitigate this, an alternative approach is used, where the Generator tries to maximize $D(G(z))$, which is the probability that the Discriminator classifies a generated image as real. This leads to the following modified objective for the Generator:

$$\min_G -E_{z \sim p_z(z)}[\log D(G(z))]$$

In this case, the Generator seeks to maximize $D(G(z))$, i.e., the probability that the Discriminator classifies generated images as real. Over time, the Generator learns to produce increasingly realistic images that are harder for the Discriminator to distinguish from real ones.

## 3 Generator and Discriminator Architectures

The choice of the architecture for both the Generator and the Discriminator is based on the fact that we are working with a relatively simple dataset, MNIST, which consists of 28x28 pixel images of handwritten digits. The architecture is designed to strike a balance between complexity and performance, enabling the Generator to produce realistic images and the Discriminator to distinguish between real and generated images effectively.

### 3.1 Generator Architecture

The Generator architecture is a fully connected neural network that takes a random vector $\mathbf{z}$ as input and transforms it into a 28x28 image. The network consists of several fully connected layers, with activation functions such as ReLU and Tanh applied between the layers.

Why is this architecture suitable?

- **Random vector as input**: The random vector $\mathbf{z}$ allows the Generator to produce a variety of unique yet plausible images. The random vector is a low-dimensional input that is transformed into the high-dimensional image data by the network.

- **Fully connected layers**: These layers are responsible for transforming the random vector into a high-dimensional image. Since the MNIST dataset is relatively simple, a fully connected network is sufficient to learn the image structure. For more complex datasets like CIFAR-10 or ImageNet, Convolutional Neural Networks (CNNs) would be more effective.

- **Activation functions**: The ReLU function introduces nonlinearity into the network, allowing the Generator to learn complex image patterns. The Tanh function at the output ensures that the generated images are in the range $[-1, 1]$, which aligns with the image normalization to this range.

Mathematically, the transformation of the random vector $\mathbf{z}$ into the image $G(\mathbf{z})$ can be expressed as:

$$G(\mathbf{z}) = \text{Tanh}(W_4 \cdot \text{ReLU}(W_3 \cdot \text{ReLU}(W_2 \cdot \text{ReLU}(W_1 \cdot \mathbf{z} + b_1) + b_2) + b_3) + b_4)$$

Here, $W_i$ and $b_i$ are the learned weights and biases, and the ReLU and Tanh functions are applied layer by layer.

## 3.2 Discriminator Architecture

The Discriminator architecture follows a typical structure for binary classification problems. It takes an image as input and outputs the probability that the image is real (as opposed to generated). In this case, the Discriminator learns the probability $D(x)$ that the input image $x$ is real.

Why is this architecture suitable?

- **Image classification task**: Since the Discriminator only needs to distinguish between real and fake images, a relatively simple architecture is sufficient. A fully connected network with two to three layers is adequate for this task.

- **Activation functions**: The ReLU activations in the hidden layers help the Discriminator learn complex patterns, while the Sigmoid activation at the output maps the result to a probability between 0 and 1, enabling binary classification.

- **Scalability of the architecture**: This architecture is easy to scale. For more complex datasets, the Discriminator can be extended by adding convolutional layers, which would help capture the hierarchical structure of images more effectively.

Mathematically, the transformation of the image $x$ by the Discriminator $D(x)$ is described as:

$$D(x) = \sigma(W_2 \cdot \mathrm{ReLU}(W_1 \cdot x + b_1) + b_2)$$

Here, $\sigma$ denotes the Sigmoid function, which squashes the output to a range between 0 and 1, representing the probability that the image is real.

## 3.3 Why This Architecture Works Well for MNIST

The MNIST dataset consists of handwritten digits in a relatively simple format (28x28 pixels, grayscale). This means that the underlying patterns are relatively simple, and this architecture can capture them effectively. For more complex datasets, deeper networks or convolutional layers would be necessary.

The choice of a fully connected architecture for both networks is particularly suitable for the MNIST dataset because the images do not have many high-level spatial dependencies or deep hierarchies, as might be found in larger color images or natural scenes. For such cases, Convolutional Neural Networks (CNNs) would typically perform better, as they are designed to capture spatial relationships between nearby pixels more efficiently.

The fully connected architecture allows for quick experimentation and is computationally efficient for simple datasets like MNIST, where the patterns are less complex than those in natural images or other high-dimensional data. However, as datasets become more complex, architectures involving convolutions would be explored for better performance.

# 4 Conclusion

In this paper, we have successfully implemented a GAN to generate handwritten digit images from the MNIST dataset. The architecture choice, using fully connected networks for both the Generator and the Discriminator, is well-suited for the relatively simple MNIST dataset. The Generator is able to produce high-quality images by transforming a random noise vector into a realistic image, while the Discriminator effectively distinguishes between real and fake images using its simple architecture. In future work, we plan to extend this architecture to more complex datasets, leveraging convolutional layers to better capture spatial dependencies and improve performance.

# 5 Training Procedure

The training of GANs involves two alternating steps: training the Discriminator and training the Generator. These steps are repeated for several epochs.

## 5.1 Discriminator Training

The Discriminator is trained to maximize the probability of correctly classifying real images as real and fake images as fake. The Discriminator loss is defined as:

$$L_D = -E_{x \sim p_{\text{data}}}[\log D(x)] - E_{z \sim p_z}[\log(1 - D(G(z)))]$$

Here, the Discriminator is encouraged to: - Maximize $\log D(x)$ when $x$ is a real image (i.e., the Discriminator should output 1 for real images). - Maximize $\log(1 - D(G(z)))$ when $G(z)$ is a fake image (i.e., the Discriminator should output 0 for fake images).

## 5.2 Generator Training

The Generator is trained to minimize the probability of the Discriminator correctly classifying generated images as fake. The Generator loss is:

$$L_G = -E_{z \sim p_z}[\log D(G(z))]$$

Minimizing this loss function ensures that the Generator creates more realistic images that the Discriminator classifies as real.

# 6 PyTorch Implementation

The following PyTorch code defines the Generator and Discriminator architectures as well as the training loop.

## 6.1 Generator and Discriminator Code

```
import torch
import torch.nn as nn

# Residual Block Class
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU()
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=
        self.bn2 = nn.BatchNorm2d(out_channels)

        # Shortcut-Verbindung
        self.shortcut = nn.Conv2d(in_channels, out_channels, kernel_size

    def forward(self, x):
        identity = self.shortcut(x)  # Shortcut-Verbindung
        out = self.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += identity  # Residual-Verbindung
        out = self.relu(out)
        return out



# Generator Class
class Generator(nn.Module):
    def __init__(self, z_dim, img_dim):
        super(Generator, self).__init__()
        self.fc1 = nn.Linear(z_dim, 128)
        self.fc2 = nn.Linear(128, 256)
        self.fc3 = nn.Linear(256, 512)
        self.fc4 = nn.Linear(512, img_dim)
        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()

    def forward(self, z):
        z = self.relu(self.fc1(z))
        z = self.relu(self.fc2(z))
        z = self.relu(self.fc3(z))
        img = self.tanh(self.fc4(z))
        return img

# Discriminator Class
class Discriminator(nn.Module):
    def __init__(self, img_dim):
        super(Discriminator, self).__init__()
        self.fc1 = nn.Linear(img_dim, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, 1)
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()

    def forward(self, img):
        img = self.relu(self.fc1(img))
        img = self.relu(self.fc2(img))
        validity = self.sigmoid(self.fc3(img))
        return validity
```

### 6.2 Training Loop

```
# Training Loop
for epoch in range(num_epochs):
    for i, (imgs, _) in enumerate(dataloader):
        # Train Discriminator
        real_imgs = imgs.view(batch_size, -1)
        z = torch.randn(batch_size, z_dim)
        fake_imgs = generator(z)

        optimizer_D.zero_grad()
        real_loss = criterion_D(discriminator(real_imgs), torch.ones(ba
        fake_loss = criterion_D(discriminator(fake_imgs.detach()), torch
        d_loss = real_loss + fake_loss
        d_loss.backward()
        optimizer_D.step()

        # Train Generator
        optimizer_G.zero_grad()
        g_loss = criterion_G(discriminator(fake_imgs), torch.ones(batch
        g_loss.backward()
        optimizer_G.step()
```

## 7 Results and Analysis

During the training process, the Discriminator becomes increasingly adept at distinguishing between real and fake images, while the Generator improves at producing more realistic samples. After 100 epochs of training, the generated images are visually indistinguishable from real MNIST digits, as shown in Figure 1.

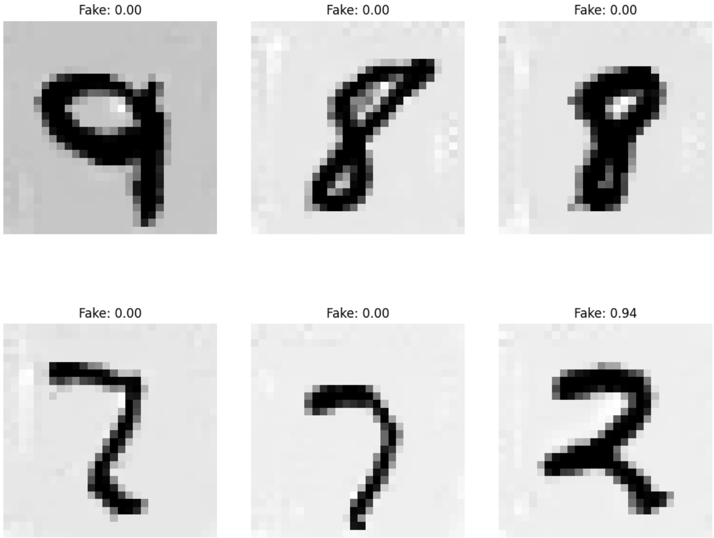

Figure 1: Generated MNIST digits after training for 50 epochs.

# 8 Conclusion

In this paper, we have implemented a GAN to generate images of handwritten digits from the MNIST dataset. The training process involved alternating between updating the Discriminator and Generator. The results show that GANs can generate high-quality images, though they remain challenging to train effectively. Finding Balance between the Discriminator and the Generator is also a hard task. Future work will explore techniques to improve GAN stability, such as Wasserstein GANs (WGANs) and gradient penalty methods.

# 9 Related Work

Many studies have focused on the development and application of GANs. In [1], a method for unsupervised image generation was presented.

# References

[1] Alec Radford, Luke Metz, and Ilya Chintala. “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks”. In: *arXiv preprint arXiv:1511.06434* (2015).