A PyTorch Implementation of Generative Adversarial Network

Jan Burakowski

1 Introduction

To better understand the original architecture of Generative Adversarial Networks (GANs) I have decided to follow the simplest one in this field proposed by Goodfellow et al. [1], I implemented a basic GAN in PyTorch using the MNIST dataset. The goal was to explore and understand how this architecture works and learn about some challenges that may occur while implementing it.

While setting up and training the model, I encountered several challenges, particularly around training stability and mode collapse. The following sections describe these issues and how they manifest during training in more detail.

In the repository you will find two other files:"test.ipynb" which is a Jupyter notebook with all the experiments and descriptions and "main.py" which is the final, best performing architecture.

2 Mathematical Foundations of GANs

A GAN is made up of two neural networks that train together: a **generator** G that takes in random noise $z \sim p_z$ and tries to produce realistic data samples, and a **discriminator** D that learns to tell the difference between real data and fake data coming from the generator.

Their training process is framed as a minimax game, with the following objective:

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

The discriminator is trained to assign high probabilities to real samples and low ones to generated ones. The generator, meanwhile, tries to produce samples that the discriminator believes are real.

Since minimizing $\log(1 - D(G(z)))$ can lead to vanishing gradients early in training (especially if D gets too confident), alternative loss is proposed in the paper:

$$L_G = -\mathbb{E}_{z \sim p_z}[\log D(G(z))] \tag{2}$$

This encourages G to produce outputs that D classifies as real and tends to give stronger gradients when the generator is still learning.

Over time, if both networks improve in balance, the generator gets good enough that the discriminator can no longer tell real from fake. At this point, called the equilibrium, the discriminator outputs values close to 0.5 for all inputs, essentially guessing, and the generator's output becomes indistinguishable from real data.

3 Implementation in PyTorch

To explore the original GAN architecture, I implemented the model in PyTorch with almost the same parameters and trained it on the MNIST dataset of handwritten digits.

3.1 Dataset and Preprocessing

The dataset consists of 28×28 grayscale images, flattened into 784-dimensional vectors. I normalized pixel values to $[-1, 1]$ to use tanh activation in generator's case. A batch size of 100 was used during training.

3.2 Network Architecture

Generator (G): Takes a 100-dimensional noise vector and passes it through two fully connected layers with 1200 ReLU units. The final output layer uses tanh activation to produce a 784-dimensional vector representing an image. **Discriminator (D):** Accepts a 784-dimensional input and passes it through two hidden layers with 240 LeakyReLU units. A final sigmoid layer outputs the probability that the input is real. I used LeakyReLU instead of the original paper's maxout for simplicity and stability.

3.3 Training Procedure

Each training step involves:

- Sampling a batch of real images and generating fake ones from random noise.
- Training the discriminator to classify real as 1 and fake as 0 using binary cross-entropy:

$$L_D = -\log D(x_{\text{real}}) - \log(1 - D(G(z)))$$

- Training the generator to fool the discriminator using the non-saturating loss:

$$L_G = -\log D(G(z))$$

I updated D and G alternately using SGD with momentum 0.5 and a learning rate of 0.1. Before each update, gradients were zeroed, and during D updates, fake samples were detached to prevent gradients from flowing into G.

To monitor training, I tracked the loss values. Ideally, L_D approaches 0.5 when D is unsure, and a decreasing L_G suggests the generator is improving. In practice, losses often oscillated due to the adversarial nature of training.

4 Training Challenges

GANs are difficult to train due to the adversarial interaction between the generator and discriminator. Two common issues are mode collapse, where the generator produces limited variations, and instability caused by imbalance between the two networks.

In my case, I encountered mode collapse during training. The generator started producing nearly identical digit images, ignoring the diversity of the MNIST dataset. This typically happened when G found a pattern that briefly fooled D, and then kept repeating it. Monitoring loss values and inspecting generated samples helped identify the issue, though resolving it fully required further changes which I will describe below.

5 Results

In my first two attempts, I encountered mode collapse. In the first run (Figure 1), the discriminator's loss quickly saturated at its maximum while the generator's loss dropped to zero, indicating that the generator was producing outputs that D could not distinguish, but not due to meaningful learning. This led to a total lack of diversity in generated samples.

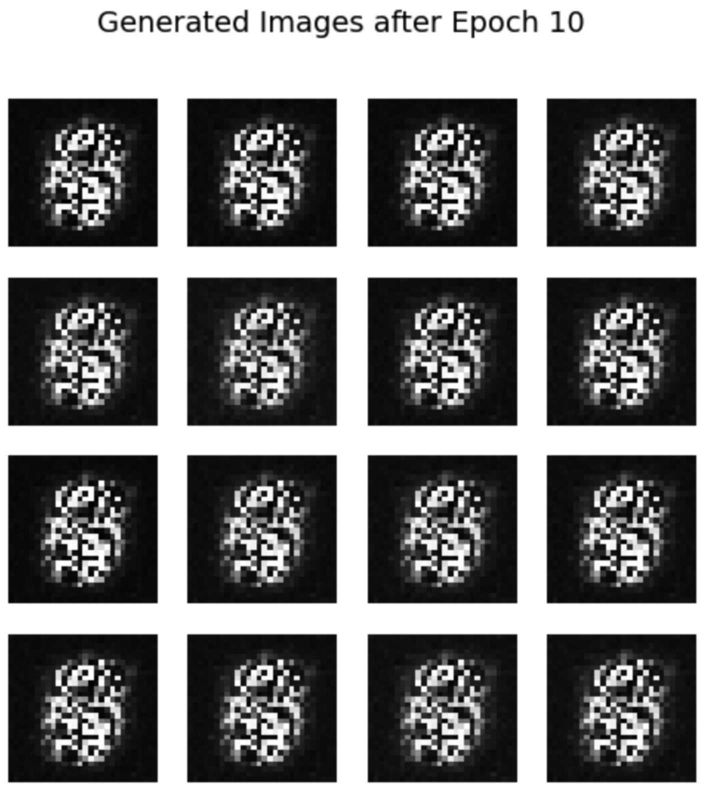

Figure 1: First failed attempt: mode collapse. The discriminator gets stuck at maximum loss while the generator loss drops to zero. The generator produces meaningless outputs.

To address this, I lowered the learning rate from 0.1 to 0.002 (as suggested in forums online). This helped prevent the losses from immediately collapsing—both G and D remained active. However, as shown in Figure 2, the generator started producing the same noisy image regardless of input, indicating a different form of mode collapse.

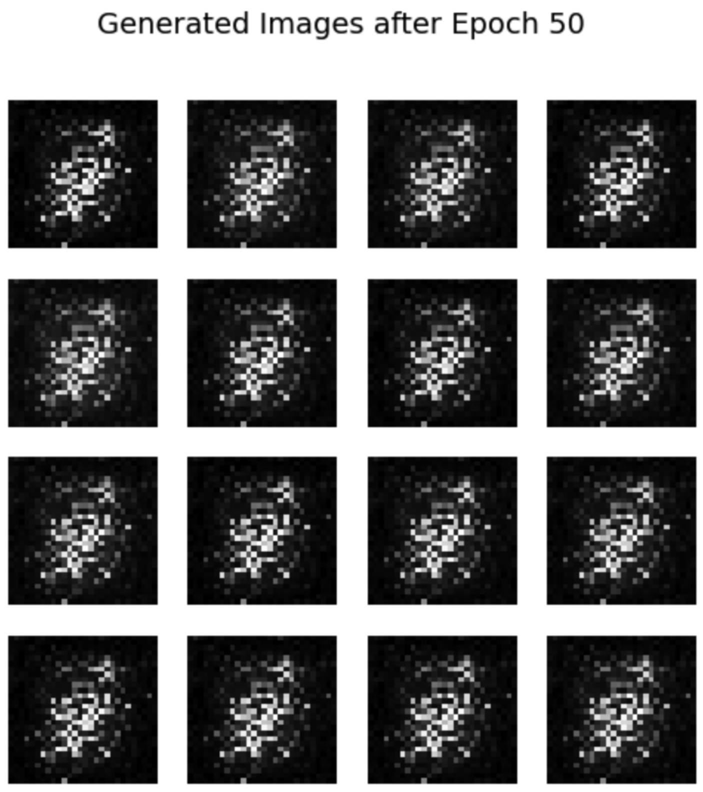

Figure 2: Second attempt with a smaller learning rate (0.002). Losses are more stable, but the generator outputs the same noise repeatedly, indicating persistent mode collapse.

In the third experiment, I kept the original architecture but switched to the Adam optimizer for both networks. As shown in Figure 3, this significantly improved performance. The generator produced diverse and realistic digits, and the training losses remained within a stable and effective range.

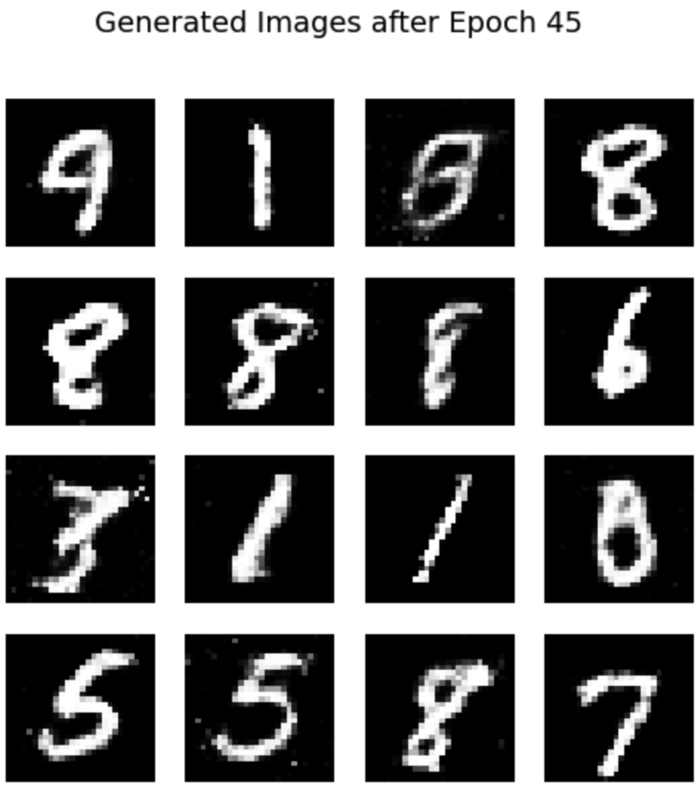

Figure 3: Successful training using Adam optimizer. The generator outputs show diverse and plausible digits. Training remained stable over time.

Finally, in Figure 4, I experimented with a setup closer to the original paper by replacing tanh with a sigmoid activation in the generator's output layer and adjusting the data range accordingly. This variant also worked reasonably well, but the results were slightly blurrier and less consistent compared to the Adam setup with tanh.

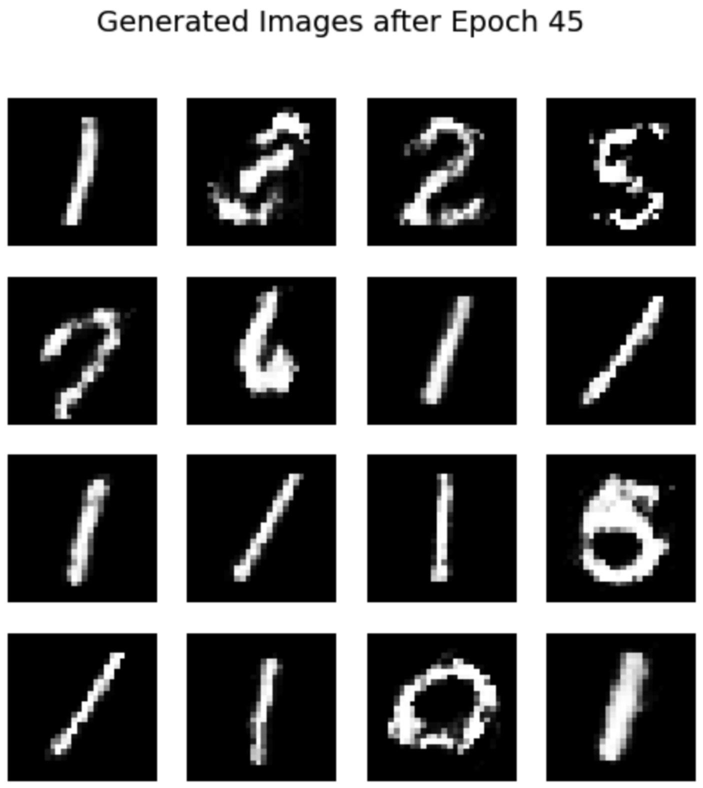

Figure 4: Variant using sigmoid activation instead of tanh, following the original GAN paper. The results are acceptable, but less sharp and consistent.

6 Conclusion

This project gave me hands-on experience with training Generative Adversarial Networks and helped me better understand both their power and complexity. Initially, I underestimated the difficulty of achieving stable training. My first mistake was using a high learning rate with SGD, which led to immediate mode collapse, where the generator produced meaningless outputs and the discriminator stopped improving. Lowering the learning rate prevented the losses from collapsing, but the generator still failed to produce varied outputs.

The most stable and successful setup used the Adam optimizer, which significantly improved both the quality and diversity of the generated digits. I also tested an architecture closer to the original GAN paper by using sigmoid activation instead of tanh, which worked reasonably well but produced slightly less consistent results.

Through this process, I learned how sensitive GAN training is to hyperparameters and architecture choices. Small changes, like optimizer selection or output activation, can make a major difference. Most importantly, I gained practical insight into diagnosing mode collapse and balancing the dynamics between generator and discriminator.

References

[1] Goodfellow, Ian, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. *"Generative Adversarial Nets."* Advances in Neural Information Processing Systems 27 (NIPS 2014).