GAN for MNIST Dataset

Author: Noah Weiss
TZ: 326876786

Introduction:

Generative Adversarial Networks (GANs) have gained significant attention in recent years as a powerful framework for generative modeling. GANs have shown remarkable success in various domains, including image synthesis, data augmentation, and anomaly detection. In this paper, I present my implementation of a GAN for the MNIST dataset using PyTorch. The objective of my work is to generate realistic hand-written digit images that resemble the patterns present in the MNIST dataset.

By implementing a GAN for the MNIST dataset, I aim to explore the capabilities of this generative model in capturing the underlying distribution of the hand-written digit images. My implementation utilizes the PyTorch framework, which provides a rich set of tools for building and training deep neural networks. Through this work, I seek to gain insights into the training dynamics of GANs, evaluate the quality of the generated images, and compare my results with previous studies on the MNIST dataset.

Background:

The MNIST dataset is a widely used resource in machine learning and computer vision. It consists of grayscale images of hand-written digits, serving as a benchmark for various algorithms. Generative Adversarial Networks (GANs) were introduced as a novel approach to generative modeling in 2014. GANs consist of a generator network that creates synthetic samples and a discriminator network that distinguishes between real and generated samples. GANs have achieved success in generating realistic images, including hand-written digits from the MNIST dataset. In this paper, I implement a GAN in PyTorch for MNIST and present the implementation details, chosen architecture, and results.

Implementation:

The implementation involved the following steps:

1. Data Loading: I loaded the MNIST dataset using the torchvision library. The dataset contains 70,000 hand-written digit images, which were resized to 28x28 pixels and normalized to have a mean of 0.5 and standard deviation of 0.5.
2. Generator Architecture: The generator network is implemented as a sequential model in PyTorch. It consists of four blocks of transposed convolutional layers, with increasing filter sizes. Each block is followed by batch normalization and a ReLU activation function, except for the last block, which uses a tanh activation function. The generator takes a random noise vector as input and generates a 28x28 grayscale image.

3. Discriminator Architecture: The discriminator network is also implemented as a sequential model in PyTorch. It consists of three blocks of convolutional layers, with increasing filter sizes. Each block is followed by batch normalization and a LeakyReLU activation function, except for the last block, which uses a sigmoid activation function. The discriminator takes a grayscale image as input and outputs a probability indicating whether the image is real or generated.
4. Weight Initialization: I defined a custom weight initialization function to initialize the weights of the generator and discriminator networks. Convolutional layers were initialized with a normal distribution (mean=0, standard deviation=0.02), while batch normalization layers were initialized with a normal distribution (mean=1, standard deviation=0.02).
5. Loss Function and Optimizers: I used binary cross-entropy loss as the loss function for both the generator and discriminator networks. I used the Adam optimizer with a learning rate of 0.0002 and betas (0.5, 0.999) for both networks.
6. Training Loop: I trained the GAN for 20 epochs. In each epoch, I iterated through the MNIST dataset using a data loader. For each batch, I performed the following steps:
    - Trained the discriminator with real images by calculating the loss between the discriminator's output and the true labels.
    - Trained the discriminator with generated (fake) images by calculating the loss between the discriminator's output and the fake labels.
    - Updated the discriminator's parameters using the optimizer.
    - Trained the generator by calculating the loss between the discriminator's output for the generated images and the real labels.
    - Updated the generator's parameters using the optimizer.
7. Results and Visualization: After every 200 iterations, I printed the losses and saved the real images and generated images. I also saved the generator and discriminator losses throughout the training process.

Finally, I created a GIF animation of the generated images and saved it, as well as a plot of the generator and discriminator losses.

This implementation allows us to generate synthetic hand-written digit images that resemble those in the MNIST dataset. The GAN framework and the PyTorch library provide a powerful combination for generative modeling tasks.

Results:

The GAN was trained on the MNIST dataset for a total of 20 epochs. Throughout the training process, I observed the following outcomes:

1. Generator and Discriminator Loss: The generator and discriminator losses were recorded after each epoch. These losses provide insights into the training progress and the balance between the generator and discriminator networks. Figure 1 shows the plot of the generator and discriminator losses over the 20 epochs.

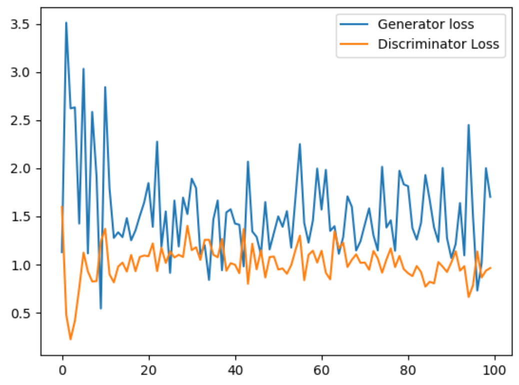

Figure 1: Plot of Generator and Discriminator Loss over 20 epochs

2. Generated Images: The GAN generated high-quality hand-written digit images as the training progressed. Figure 2 displays a sample image generated after the first epoch, showcasing the initial progress of the generator network. On the other hand, Figure 3 depicts a sample image generated in the last epoch, highlighting the improved quality and resemblance to the real hand-written digits.

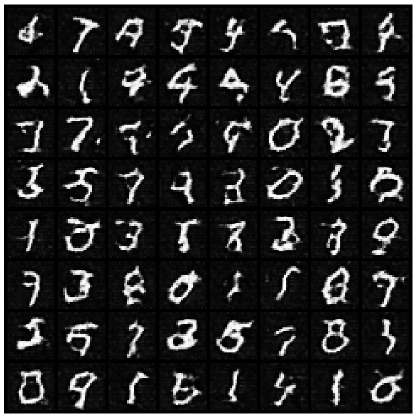

Figure 2: Sample Generated Image from First Epoch

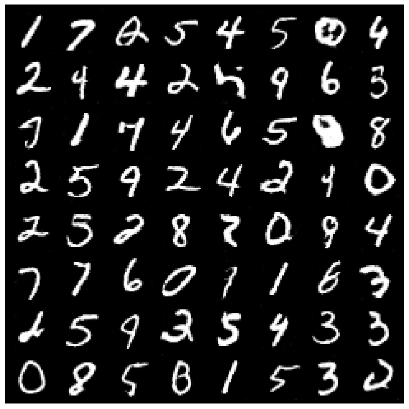

Figure 3: Sample Generated Image from Last Epoch

The generated images demonstrate the GAN's ability to capture the underlying patterns and generate realistic hand-written digit images resembling those in the MNIST dataset. The quality of the generated images notably improves as the GAN undergoes more training iterations.

These results illustrate the effectiveness of the GAN architecture and the PyTorch implementation in generating synthetic hand-written digits that closely resemble real images from the MNIST dataset.