Generative Adversarial Network (GAN) - Image Generation

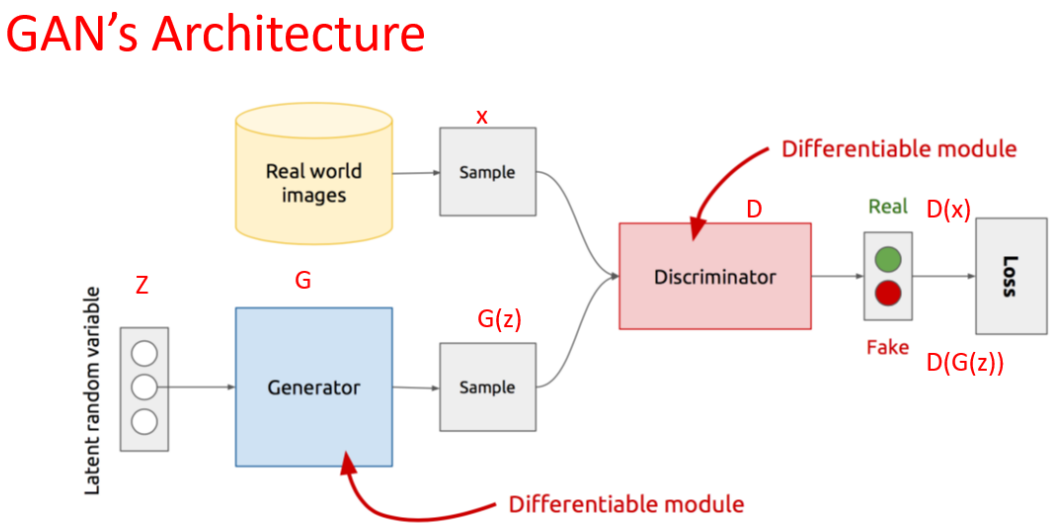
Submitting: Orel Zamler

Introduction

Generative Adversarial Networks (GANs) are a powerful class of deep learning models that have been successfully used for image generation tasks.
This document explores the implementation of a GAN using PyTorch to generate realistic images of handwritten digits. The code provided demonstrates the training process of the GAN using the MNIST dataset.

Architecture

The GAN consists of two main components: a generator and a discriminator. The generator takes random noise as input and generates realistic data, in our case fake images, while the discriminator aims to distinguish between real and fake images. Through an adversarial training process, the generator learns to produce more realistic images, while the discriminator learns to become better at distinguishing real from fake.



- **Z** is some random noise (Gaussian/Uniform).
- **Z** can be thought as the latent representation of the image.

Generative Adversarial Network (GAN) - Image Generation

Code Structure Overview

You can find the full code, explanations, and results in my colab notebook I submitted as well.

Dataset:

The MNIST dataset is loaded using the defined transformations, and a data loader is created for batched training.

GAN

The Generator and Discriminator models are defined using PyTorch's nn.Module class. The generator network architecture is defined using nn.Sequential, which allows for a sequential arrangement of layers.

Generator class:

The Generator takes random noise as input and passes it through a series of layers & activation functions:

2 * [CONV >> Batch Normalization >> Relu] >> CONV >> Tanh

Convolutional Transpose Layers:

- The first convolutional transpose layer upsamples the input noise (latent dim) to a higher resolution and increases the number of channels to 128. The kernel size, stride, and padding are set to achieve the desired output size.
- Next, Batch normalization is applied to normalize the activations of the previous layer, aiding in the stability and speed of training.
- Next, ReLU activation function is applied to introduce non-linearity and enhance the representation capacity of the network.
- This process is preformed twice, the second convolutional reduces the number of channels to 64.
- The final convolutional transpose layer upsamples the features to the desired output size, producing a single channel image.

Discriminator class:

The Discriminator takes images as input and passes them through layers & activation functions:

CONV >> LeakyReLU >> 2 * [CONV >> BatchNorm >> LeakyReLU] >> CONV >> Sigmoid

Convolutional Layers:

Generative Adversarial Network (GAN) - Image Generation

The first convolutional layer takes in a single-channel image as input (in this case, the generated or real image) and applies 64 filters with a kernel size of 4x4, stride of 2, and padding of 1. This layer reduces the spatial dimensions of the input while increasing the number of channels.

Next, LeakyReLU activation function is applied to introduce non-linearity.

The second convolutional layer takes the features from the previous layer and applies 128 filters with the same kernel size, stride, and padding. This layer further reduces the spatial dimensions and increases the number of channels. Next, Batch normalization is applied to normalize the activations of the previous layer, aiding in the stability and speed of training.

Next, LeakyReLU activation is applied again. The final convolutional layer reduces the spatial dimensions to 1x1 using a kernel size of 7x7, stride of 1, and no padding. This layer aims to capture the global information from the features.

The third convolutional layer with 128 input channels (output from the previous layer), 256 output channels, a kernel size of 4x4, a stride of 2, a padding of 1, and no bias term. This layer further extracts more complex features from the previous layer's output.

Batch and LeakyReLU are applied as well for the same resones as before.

Finally, Sigmoid activation function is applied to squash the output to a probability score between 0 and 1.

Training

Instances of the generator and discriminator are created and moved to the specified device.

The binary cross entropy loss function and Adam optimizers are defined for training the GAN.

Training the Generator and Discriminator:

Training loop iterating over the specified number of epochs and batches.

Within each iteration, the discriminator is trained first by calculating the loss on real and fake images, and then updating its parameters using the optimizer.

The generator is trained by generating fake images, passing them through the discriminator, calculating the loss, and updating its parameters.

Progress and losses are printed at regular intervals to monitor the training process.

At the end of each epoch, a set of generated images is displayed.

Saving & Loading

Lastly, I saved the trained model so we can use it later. In addition, I took it for a test by loading it back and presenting the results(the full results appears at the end of the notebook I submitted as well).

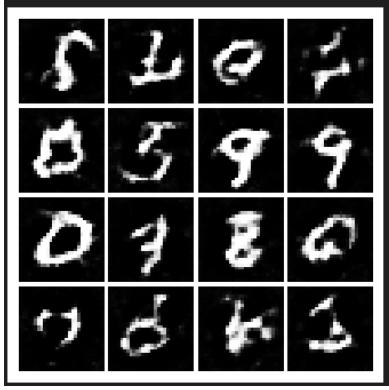
Generative Adversarial Network (GAN) - Image Generation

Results

- **You can see the full code and result in GAN.ipynb file I submitted as well.**

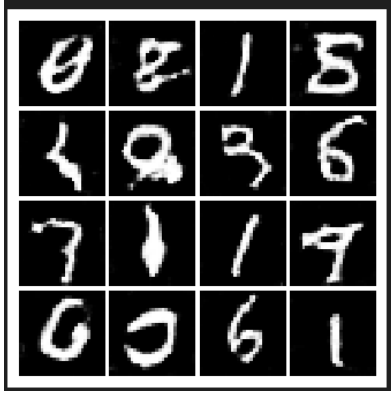
After first epoch:

```
Training Started!
Epoch [1/10], Step [250/750], D_loss: 0.3905, G_loss: 1.8542, D(x): 0.77, D(G(z)): 0.10
Epoch [1/10], Step [500/750], D_loss: 0.3576, G_loss: 1.5322, D(x): 0.75, D(G(z)): 0.03
Epoch [1/10], Step [750/750], D_loss: 0.1952, G_loss: 3.3735, D(x): 0.88, D(G(z)): 0.05
```



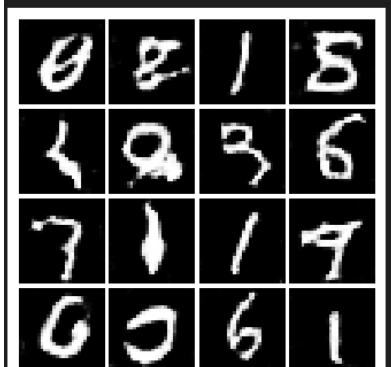
After 5 epochs:

```
Epoch [5/10], Step [250/750], D_loss: 0.0796, G_loss: 4.1870, D(x): 0.94, D(G(z)): 0.02
Epoch [5/10], Step [500/750], D_loss: 0.1177, G_loss: 4.2142, D(x): 0.97, D(G(z)): 0.08
Epoch [5/10], Step [750/750], D_loss: 0.0430, G_loss: 4.5027, D(x): 0.98, D(G(z)): 0.02
```



After the final epoch(10):

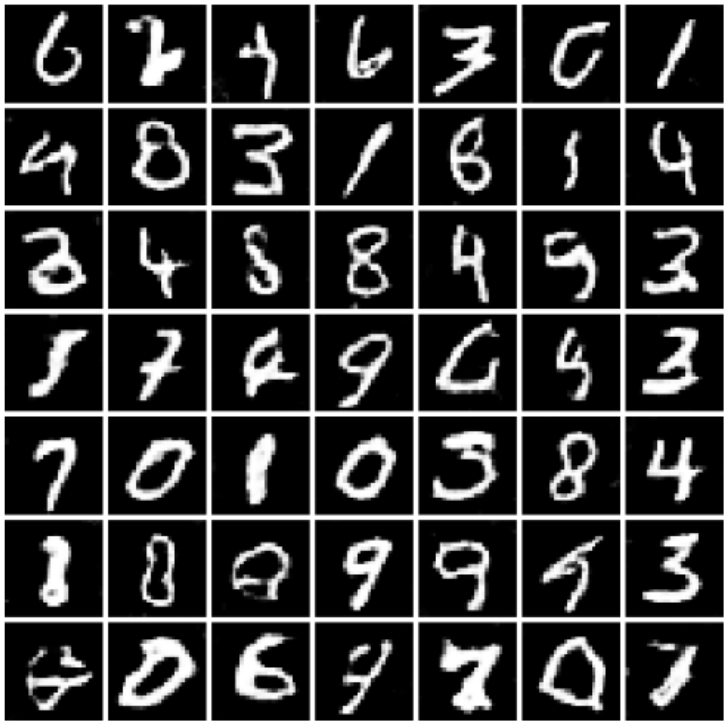
```
Epoch [5/10], Step [250/750], D_loss: 0.0796, G_loss: 4.1870, D(x): 0.94, D(G(z)): 0.02
Epoch [5/10], Step [500/750], D_loss: 0.1177, G_loss: 4.2142, D(x): 0.97, D(G(z)): 0.08
Epoch [5/10], Step [750/750], D_loss: 0.0430, G_loss: 4.5027, D(x): 0.98, D(G(z)): 0.02
```



Generative Adversarial Network (GAN) - Image Generation

When the training was done, I saved the Generator and Discriminator models and printed our pretty numbers. These are the results:

```
generate_num = 49
with torch.no_grad():
    fake_images = loaded_generator(torch.randn(generate_num, latent_dim).to(device))
    fake_images = fake_images * 0.5 + 0.5  # Denormalize the generated images
    imgs_numpy = fake_images.data.cpu().numpy()
    display_images(imgs_numpy[0:generate_num])
    plt.show()
```



Generative Adversarial Network (GAN) - Image Generation

I also printed tested our Discriminator and pretrained its outputs on the data I saved for tests:

```
# Set the discriminator to evaluation mode
loaded_discriminator.eval()

# Test the discriminator
num_samples = 10

# Generate fake images
with torch.no_grad():
    noise = torch.randn(num_samples, latent_dim, 1, 1).to(device)
    fake_images = loaded_generator(noise)

# Load a batch of real images from the dataset
real_images, _ = next(iter(train_loader))
real_images = real_images[:num_samples].to(device)

# Pass both real and fake images through the discriminator
with torch.no_grad():
    real_output = loaded_discriminator(real_images)
    fake_output = loaded_discriminator(fake_images)

# Print the discriminator outputs
print("Discriminator Outputs:\n", "Real Images:\n", real_output, "\nFake Images:\n", fake_output)
```

```
Discriminator Outputs:
 Real Images:
 tensor([[0.9810],
        [0.8953],
        [0.9694],
        [0.9912],
        [0.8853],
        [0.8803],
        [0.8496],
        [0.9492],
        [0.9213],
        [0.8311]])
Fake Images:
 tensor([[0.6576],
        [0.2884],
        [0.3611],
        [0.9159],
        [0.5985],
        [0.2250],
        [0.7511],
        [0.1585],
        [0.9011],
        [0.5314]])
```

Generative Adversarial Network (GAN) - Image Generation

```python
from sklearn.metrics import precision_score, recall_score, f1_score

# Create an empty list to store the discriminator's predictions
predictions = []

# Set the discriminator to evaluation mode
loaded_discriminator.eval()

# Iterate over the test DataLoader
with torch.no_grad():
    for real_images, _ in test_loader:
        batch_size = real_images.size(0)
        real_images = real_images.to(device)

        # Get the discriminator's output for real images
        real_output = loaded_discriminator(real_images)
        predictions.extend(real_output.cpu().numpy())

    # Convert the predictions list to a binary numpy array
    predictions = (np.array(predictions) >= 0.5).astype(int)

# Create ground truth labels (assuming real images have label 1)
ground_truth = np.ones_like(predictions)

# Calculate precision, recall, and F1-score
precision = precision_score(ground_truth, predictions)
recall = recall_score(ground_truth, predictions)
f1 = f1_score(ground_truth, predictions)

# Print the evaluation metrics
print("Results:\n\tPrecision:", precision, "\n\tRecall:", recall, "\n\tF1-Score:", f1)
```

```
Results:
        Precision: 1.0
        Recall: 0.9643333333333334
        F1-Score: 0.9818428644154081
```