Exercise 5: Generative Adversarial Networks

Erik Stolt

What you did and how

Generative Adversarial Networks (GANs) work by incorporating two neural network architectures: a generator (G) and a discriminator (D). The generator learns to produce realistic images from an abstract input distribution, typically Gaussian noise. The discriminator learns to determine whether an image is real, meaning from the training dataset, or fake, meaning generated by G.

The discriminator's loss function is the binary cross-entropy (BCE) loss. Real images are labeled as 1 and fake images as 0. During training, the discriminator is presented with both real and fake images in each batch, along with their corresponding labels. It updates its weights to improve its ability to classify them correctly. The discriminator's total loss is computed as the average of the BCE loss on real and fake samples.

The generator is trained indirectly through the discriminator. To train the generator, fake images are generated and passed to the discriminator, but this time with the label 1, as if they were real. The generator's objective is to produce fake images that the discriminator classifies as real. When the discriminator is fooled by a fake image and predicts a value close to 1, the generator receives a lower loss. Therefore, the generator learns to generate fake images that closely resemble real images from the training data.

For this exercise, we defined a generator and a discriminator network, and trained them in an adversarial setup where the generator learned to produce increasingly realistic digit images while the discriminator tried to distinguish them from real MNIST digits. We monitored the training progress by generating sample images at different epochs by fixing the noise beforehand, which clearly showed how the image quality improved over time. In practice we first normalized the input images s.t. the pixel values transformed like $[0, 1] \to [-1, 1]$ which we also later denormalized for visualization according to image $=$ image $* 0.5 + 0.5$. Regarding the network structures for the generator and discriminator, we settled on two standard NN. The generator had tanh activation in the output layer to transform the values to the $[-1, 1]$ range. The discriminator instead used sigmoid activation since its task is a binary classification task. The networks were then trained for 30 epochs with a batch size of 32, monitoring the performances using Tensorboard.

What results you obtained

Example print-out during training:

```
Epoch [28/30] | Loss D: 0.6806 | lr D: 5e-05 | Loss G: 0.7323 | lr G: 0.0001 | 135.27s
Epoch [29/30] | Loss D: 0.6682 | lr D: 5e-05 | Loss G: 0.7079 | lr G: 0.0001 | 122.10s
Epoch [30/30] | Loss D: 0.6654 | lr D: 5e-05 | Loss G: 0.7033 | lr G: 0.0001 | 126.16s
```

I also present the evolution of the digits and the loss as a function of epoch:

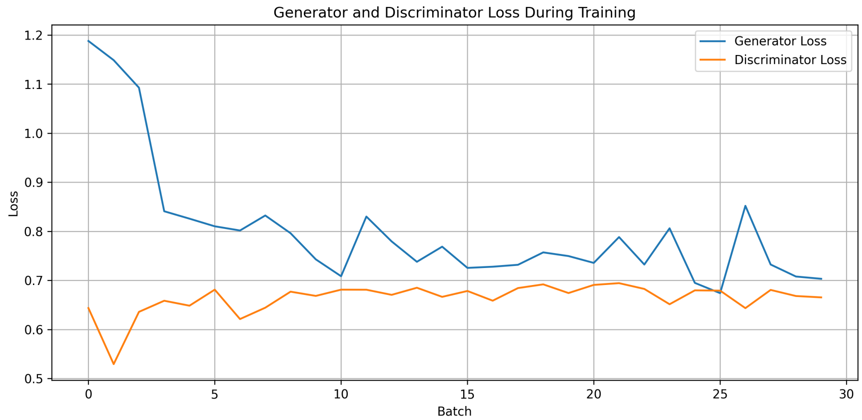

Figure 1: Loss vs epoch for both G and D.

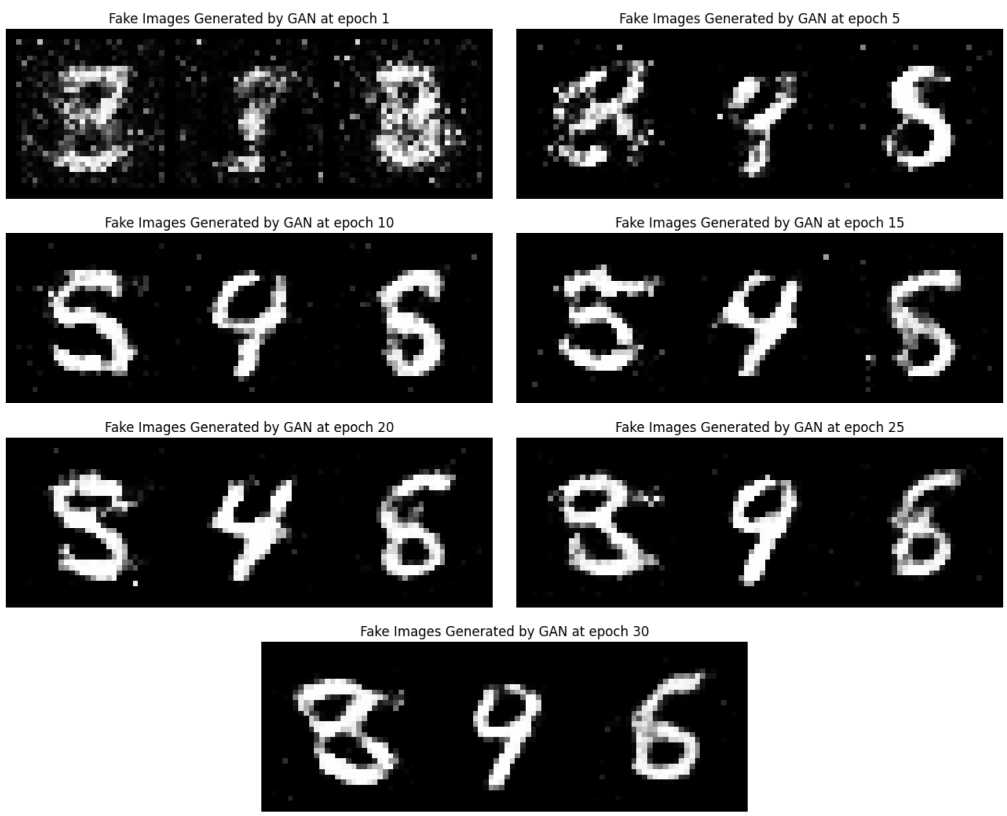

Figure 2: Evolution of G generating digits.

We clearly see that the loss stays the same but the digits obviously get better. This is the result of a good trade-off between G and D improving at the same rate.

What challenges you encountered and what could be improved

GANs are notoriously unstable which lead to the discriminant having zero loss, hence, no learning. This is fixed by setting different initial lrs for the two models. Furthermore, I also replaced the true labels ones and zeros with 0.9 and 0.1 (instead of 1 and 0) in order to trick D not to be overconfident. This works because it penalizes the model in being overconfident (see Appendix 1). I also added dropout and batch normalization in order for G not to be deterministic as this also lead to model collapse. Batch normalization is a critical part of G but often left out in D since it can cause numerical intabilities [1]. To improve the model one could use WGANs to get even better results and prevent model collapse.

This exercise also taught me a lot about interpreting losses. In GANs, monitoring losses is not enough and even misleading. If one only were to monitor the loss, model collapse seems good since it happens when the discriminators loss is zero. However, we also have the two scenarios

- Discriminator loss may go to zero when it is dominating, but that means G is not learning.
- Generator loss may go up even when it is improving image quality because D is getting better.

This means that in adversarial setups, both networks are playing a game, so, their losses do not converge in the usual sense.

Link to GitHub

Link: `https://github.com/erst6955/Advanced-Applied-Deep-Learning-in-Physics-And-Engineering`

Appendix 1

BCE is given by

$$\text{BCE}(p, y) = -(y \cdot \log(p) + (1 - y) \cdot \log(1 - p)),$$

which means that if $p = 0.99$ and $y = 1$ we would get $\text{BCE} \approx 0.01$ which is very small loss, hence, the gradient would vanish. However, if we let $y \to 0.9$ we would instead get $\text{BCE} \approx 0.47$ which is higher! Therefore, more learning is possible with this small change. The same goes for label $y \to 0.1$. This is a common trick which works well when D is learning too strong.

Code (for backup)

~\OneDrive\Desktop\Exercise 5 - Generative Adversarial Networks\GANs for MNIST.py

```python


import torchvision
from torchvision import transforms, datasets
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torch.utils.tensorboard import SummaryWriter
import matplotlib.pyplot as plt
import time
import random



# Hyperparameters
device = "cuda" if torch.cuda.is_available() else "cpu"
lr = 3e-4
batchSize = 32
logStep = 625
latent_dimension = 128
image_dimension = 784

myTransforms = transforms.Compose([ # we define a tranform that converts the image to tensor and normalizes it with mean and std of 0.5
    transforms.ToTensor(),          # which will convert the image range from [0, 1] to [-1, 1]
    transforms.Normalize((0.5,), (0.5,))
])

dataset = datasets.MNIST(root="dataset/", transform=myTransforms, download=True)
loader = DataLoader(dataset, batch_size=batchSize, shuffle=True)


class Generator(nn.Module):
    """
    Generator Model which takes a random noise vector and generates a fake image. The input noise in latent space is an abstract space where each point corresponds to a different kind of image.
    The noise vector doesn't directly mean anything in the beginning, but after training, it gets mapped by the generator to a specific kind of image — like a "7", or a "2", or a "3". We make
    assumptions about how the input noise is distributed.
    """
    def __init__(self):
        super().__init__()
        self.gen = nn.Sequential( # simple NN
            nn.Linear(latent_dimension, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(),

            nn.Linear(256, 512),
```

```
            nn.BatchNorm1d(512),
            nn.ReLU(),

            nn.Linear(512, 1024),
            nn.BatchNorm1d(1024),
            nn.ReLU(),

            nn.Linear(1024, image_dimension),
            nn.Tanh(), # tanh activation function to get the output in the range of [-1, 1]
        )

    def forward(self, x):
        return self.gen(x)

class Discriminator(nn.Module):
    """
    Discriminator Model which takes an image and outputs a probability of it being real or fake. Furthermore,
    the discriminator is a binary classifier which uses the sigmoid activation and the BCE loss function.
    """
    def __init__(self):
        super().__init__()
        self.disc = nn.Sequential(
            nn.Linear(image_dimension, 1024),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),  # add this
            nn.Linear(1024, 512),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3), # add this
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 1),
            nn.Sigmoid(),
        )


    def forward(self, x):
        return self.disc(x)


def train_models(discriminator, generator, fixed_noise, num_examples, opt_discriminator, opt_generator, criterion, epochs, writer):
    """
    Trains the GAN models using the MNIST dataset.
    The generator uses BCE where 1 is the label for fake images and 0 is the label for real images. This mean that it learns
    to generate fake images that are similar to the real images in the dataset since the discriminator has real images labeled as
    1 and fake images labeled as 0. I.e., the generator aims to minimize the loss by producing images that are classified as real by the discriminator.
    The disciminator on the other hand is simply trained to identify the real and fake images.
    """
```

```python
    step = 0 # for tensorboard logging
    gen_losses = []
    disc_losses = []

    # ===================== Training Loop ========================================
    for epoch in range(epochs):

        start_time = time.time()

        for batch_idx, (real, _) in enumerate(loader):
            real = real.view(-1, image_dimension).to(device) # flatten the image in order to pass it to the discriminator
            batch_size = real.shape[0] # get the batch size

            noise = torch.randn(batch_size, latent_dimension).to(device) # generate random noise from a normal distribution
            fake = generator(noise) # generate fake images from the noise

            # Discriminator Loss
            disc_real = discriminator(real).view(-1) # get the discriminator output for real images
            loss_real = criterion(disc_real, torch.full_like(disc_real, 0.9)) # pass through BCE where real images are labeled as 1

            disc_fake = discriminator(fake.detach()).view(-1) # get the discriminator output for fake images
            loss_fake = criterion(disc_fake, torch.full_like(disc_fake, 0.1)) # pass through BCE where fake images are labeled as 0

            loss_discriminator = (loss_real + loss_fake) / 2 # average the loss for real and fake images

            discriminator.zero_grad() # zero the gradients
            loss_discriminator.backward(retain_graph=True) # backpropagate the loss
            opt_discriminator.step() # update the discriminator weights

            # ===== Generator Loss =====
            # The generator tries to fool the discriminator, so we want the discriminator to think that the fake images are real
            output = discriminator(fake).view(-1) # get the discriminator output for fake images
            loss_generator = criterion(output, torch.ones_like(output)) # pass through BCE where fake images are labeled as 1
            # The generator tries to maximize the probability of the discriminator being wrong

            generator.zero_grad()
            loss_generator.backward()
            opt_generator.step()

            if batch_idx % logStep == 0: # tensorboard logging
                with torch.no_grad():
                    fake_images = generator(fixed_noise).reshape(-1, 1, 28, 28) # reshape the fake images to 1 channel and 28x28
```

```python
                    real_images = real.reshape(-1, 1, 28, 28) # reshape the real images to 1 channel and 28x28

                    imgGridFake = torchvision.utils.make_grid(fake_images, normalize=True) # make a grid of fake images
                    imgGridReal = torchvision.utils.make_grid(real_images, normalize=True) # make a grid of real images

                    # Denormalize: [-1, 1] → [0, 1]
                    fake_images = denormalize(fake_images)
                    real_images = denormalize(fake_images)

                    writer.add_image("MNIST Fake Images", imgGridFake, global_step=step)
                    writer.add_image("MNIST Real Images", imgGridReal, global_step=step)
                    writer.add_scalar("Loss Discriminator", loss_discriminator.item(), step)
                    writer.add_scalar("Loss Generator", loss_generator.item(), step)

                    step += 1
        for param_group in opt_generator.param_groups:
            lr_g = param_group['lr']
        for param_group in opt_discriminator.param_groups:
            lr_d = param_group['lr']

        # At the end of each epoch
        gen_losses.append(loss_generator.item())
        disc_losses.append(loss_discriminator.item())
        elapsed = time.time() - start_time
        print(f"Epoch [{epoch+1}/{epochs}] | Loss D: {loss_discriminator:.4f} | lr D: {lr_d} | Loss G: {loss_generator:.4f} | lr G: {lr_g} | {elapsed:.2f}s")

        get_fake_images(generator, fixed_noise, latent_dimension, num_examples, epoch) # get some fake images for the epoch

    return gen_losses, disc_losses


def denormalize(images):
    return images * 0.5 + 0.5



def get_fake_images(generator, fixed_noise, latent_dimension, num_examples, epoch):
    """
    Returns a batch of fake images from the generator.
    """

    generator.eval()

    # Generate images from noise
    with torch.no_grad():
        generated_images = generator(fixed_noise).reshape(-1, 1, 28, 28)
        generated_images = denormalize(generated_images) # denormalize the images

```

```python
    # Create a grid for visualization
    grid = torchvision.utils.make_grid(generated_images.cpu(), nrow=4, normalize=True)

    # Plot the generated images
    plt.figure(figsize=(8, 8))
    plt.title(f"Fake Images Generated by GAN at epoch {epoch+1}")
    plt.axis("off")
    plt.imshow(grid.permute(1, 2, 0).squeeze())
    plt.savefig(f"Figures/fake_images_{epoch+1}.png", bbox_inches='tight')
    plt.close()



def plot_losses(gen_losses, disc_losses):
    plt.figure(figsize=(10, 5))
    plt.plot(gen_losses, label="Generator Loss")
    plt.plot(disc_losses, label="Discriminator Loss")
    plt.xlabel("Batch")
    plt.ylabel("Loss")
    plt.title("Generator and Discriminator Loss During Training")
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.savefig("Figures/loss_plot.png", dpi=300)



# =========================================== Run Code =========================================================
discriminator = Discriminator().to(device)
generator = Generator().to(device)

lr_G = 1e-4
lr_D = 5e-5

opt_generator = optim.Adam(generator.parameters(), lr=lr_G, betas=(0.5, 0.999))
opt_discriminator = optim.Adam(discriminator.parameters(), lr=lr_D, betas=(0.5, 0.999))
criterion = nn.BCELoss() # Binary Cross Entropy Loss

epochs = 30
num_examples = 3

fixed_noise = torch.randn(num_examples, latent_dimension).to(device) # fixed noise for reproducibility after every epoch
writer = SummaryWriter("runs/GAN_MNIST")

gen_losses, disc_losses = train_models(discriminator, generator, fixed_noise, num_examples, opt_discriminator, opt_generator, criterion, epochs, writer)
plot_losses(gen_losses, disc_losses)

writer.close()
print("Complete!")
```

References

[1] Alec Radford, Luke Metz, and Soumith Chintala. "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks". In: *arXiv preprint arXiv:1511.06434* (2016). URL: `https://arxiv.org/abs/1511.06434`.