Denoising Diffusion Probabilistic Models (DDPM) - A Complete Guide with PyTorch Implementation

GitHub: https://github.com/musawir124/Denoising-Diffusion-Probabilistic-Models.git

Diffusion models are a class of **generative models** that have gained **massive popularity** due to their ability to generate high-quality, diverse, and realistic images. These models **reverse a noise process**, gradually converting random noise into meaningful images.

This tutorial will guide you through the **theory and implementation** of DDPMs, explaining the mathematical foundations and coding the model in **PyTorch** step by step.

1. Introduction to Generative Models

Generative models are **AI models** that can create **new data** similar to a given dataset. Some popular generative models include:

- **GANs (Generative Adversarial Networks)**
- **VAEs (Variational Autoencoders)**
- **Flow-based Models**
- **Diffusion Models (DDPMs, Stable Diffusion, etc.)**

💡 **Why use Diffusion Models?**

- **Stable Training** compared to GANs.
- **High-Quality Image Generation**.
- **Better Diversity** (avoids mode collapse seen in GANs).

2. What is a Diffusion Model?

A **Diffusion Model** consists of two processes:

🔹 **Forward Process (Adding Noise)**

A clean image is **gradually corrupted** by adding Gaussian noise at each time step until it becomes **pure noise**.

🔹 **Reverse Process (Denoising)**

A neural network is trained to **undo the noise** step-by-step, reconstructing the original image.

Forward Process (Noise Addition):

The **goal** of the forward process is to **progressively destroy** an image by adding noise step by step.

At each step t, we add Gaussian noise:

$$q(x_t|x_{t-1}) = \mathcal{N}(x_t; \sqrt{\alpha_t}x_{t-1}, (1-\alpha_t)I)$$

where:

- x_t is the noisy image at time step t.
- α_t controls the noise level.
- I is the identity matrix, ensuring noise follows a Gaussian distribution.

By applying the **reparameterization trick**, we can directly compute x_t from x_0 (original image):

$$x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon$$

where:

- $\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s$ (cumulative product of noise schedule).
- $\epsilon \sim \mathcal{N}(0, I)$ (random Gaussian noise).

Reverse Process (Denoising):

The reverse process **reconstructs** the original image from pure noise.

$$p_\theta(x_{t-1}|x_t) = \mathcal{N}(x_{t-1}; \mu_\theta(x_t, t), \sigma^2 I)$$

- A **neural network** $\epsilon_\theta(x_t, t)$ is trained to predict noise at each step.
- The predicted noise is then used to gradually **denoise** the image.

Mathematical Formulation of DDPM:

Loss Function:

Instead of learning p(xt−1|xt)p(x_{t-1} | x_t)p(xt−1|xt) directly, we train a neural network to predict the noise ϵ\epsilonϵ that was added at each step.

$$L_{\text{simple}} = \mathbb{E}_{x_0,\epsilon,t}\left[||\epsilon - \epsilon_\theta(x_t, t)||^2\right]$$

The model learns to predict the noise **accurately** so that we can subtract it and reconstruct the original image.

Implementing DDPM in PyTorch

Let's code a basic DDPM from scratch in PyTorch!

Import Necessary Libraries

In [8]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
```

Define the Diffusion Model

In [9]:
```
class SimpleDiffusionModel(nn.Module):
    def __init__(self):
        super(SimpleDiffusionModel, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 32, 3, padding=1),
            nn.ReLU(),
            nn.Conv2d(32, 64, 3, padding=1),
            nn.ReLU()
        )
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(64, 32, 3, padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(32, 1, 3, padding=1),
            nn.Sigmoid()
        )

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

In [ ]:

Prepare Dataset (MNIST)

In [10]:
```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

train_dataset = datasets.MNIST(root='./data', train=True, transform=transform, download=True)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
```

```
100%|██████████████████████████████████████████████████████████████████████████████| 9.91M/9.91M [00:18<00:00, 5
36kB/s]
100%|█████████████████████████████████████████████████████████████████████████████| 28.9k/28.9k [00:00<00:00, 67
.7kB/s]
100%|█████████████████████████████████████████████████████████████████████████████| 1.65M/1.65M [00:17<00:00, 94
.4kB/s]
100%|██████████████████████████████████████████████████████████████████████████████| 4.54k/4.54k [00:00<00:00, 3
97kB/s]
```

Define Noise Schedule & Forward Diffusion Process

In [11]:
```
def add_noise(images, noise_level=0.5):
    noise = torch.randn_like(images) * noise_level
    return images + noise
```

Train the Diffusion Model

In [ ]:

In [12]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = SimpleDiffusionModel().to(device)
optimizer = optim.Adam(model.parameters(), lr=0.001)
criterion = nn.MSELoss()
```

```
num_epochs = 5

for epoch in range(num_epochs):
    for images, _ in train_loader:
        images = images.to(device)
        noisy_images = add_noise(images)

        optimizer.zero_grad()
        output = model(noisy_images)

        loss = criterion(output, images)
        loss.backward()
        optimizer.step()

    print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}")

# Save trained model
torch.save(model.state_dict(), "diffusion_model.pth")
```

```
Epoch [1/5], Loss: 0.8375
Epoch [2/5], Loss: 0.8345
Epoch [3/5], Loss: 0.8551
Epoch [4/5], Loss: 0.8354
Epoch [5/5], Loss: 0.8418
```

Generate New Images

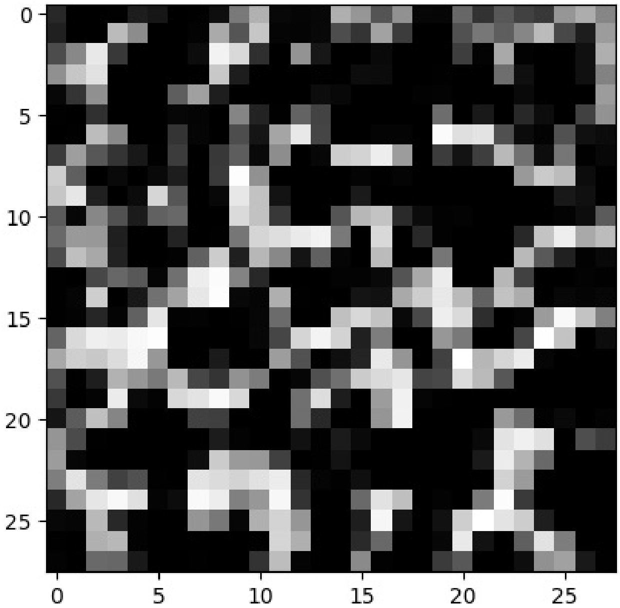
In [13]:

```
def generate_images(model, noise_level=0.5):
    model.eval()
    noise = torch.randn((1, 1, 28, 28)).to(device)
    with torch.no_grad():
        denoised_image = model(noise)
    return denoised_image

# Load Model
model.load_state_dict(torch.load("diffusion_model.pth"))
model.to(device)

# Generate Image
generated_img = generate_images(model).cpu().squeeze().numpy()

plt.imshow(generated_img, cmap="gray")
plt.show()
```



Final Summary

Defined a simple U-Net model for denoising.

Added noise to images and trained the model.

Saved the trained model for future use.

Generated new images by reversing the noise process.

In [ ]: