Understanding Denoising Diffusion Probabilistic Models: A Code-Based Implementation Guide

Implementation Analysis of Ho et al. (2020)

August 27, 2025

Abstract

This document provides a comprehensive analysis of a PyTorch implementation of Denoising Diffusion Probabilistic Models (DDPM) as introduced by Ho, Jain, and Abbeel in 2020. We explain the mathematical foundations, implementation details, and relationship between the theoretical framework and practical code. The implementation demonstrates key concepts including forward diffusion, reverse denoising, U-Net architecture with time conditioning, and the noise prediction training objective that makes DDPM effective for high-quality image generation.

1 Introduction

Denoising Diffusion Probabilistic Models (DDPM) [1] represent a breakthrough in generative modeling, achieving state-of-the-art results in image synthesis. Unlike GANs or VAEs, diffusion models learn to reverse a gradual noising process, generating samples by progressively denoising random noise.

The core insight of DDPM is to model the data distribution by learning to reverse a fixed forward process that gradually adds Gaussian noise to data. This approach provides stable training and high-quality generation capabilities.

2 Mathematical Foundation

2.1 Forward Diffusion Process

The forward process gradually corrupts data $\mathbf{x}_0 \sim q(\mathbf{x}_0)$ by adding Gaussian noise over T timesteps:

$$q(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I}) \tag{1}$$

$$q(\mathbf{x}_{1:T}|\mathbf{x}_0) = \prod_{t=1}^{T} q(\mathbf{x}_t|\mathbf{x}_{t-1}) \tag{2}$$

where $\beta_1, \ldots, \beta_T$ is a variance schedule. A key property is that we can sample $\mathbf{x}_t$ directly from $\mathbf{x}_0$:

$$q(\mathbf{x}_t|\mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I}) \tag{3}$$

where $\alpha_t = 1 - \beta_t$ and $\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s$.

2.2 Reverse Process

The reverse process learns to denoise:

$$p_\theta(\mathbf{x}_{0:T}) = p(\mathbf{x}_T)\prod_{t=1}^{T} p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) \tag{4}$$

$$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \boldsymbol{\mu}_\theta(\mathbf{x}_t, t), \Sigma_\theta(\mathbf{x}_t, t)) \tag{5}$$

2.3 Training Objective

DDPM's key insight is to parameterize the model to predict noise $\boldsymbol{\epsilon}$ rather than the mean directly:

$$L_{\text{simple}} = \mathbb{E}_{t,\mathbf{x}_0,\boldsymbol{\epsilon}}\left[\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\|^2\right] \tag{6}$$

where $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}$.

3 Implementation Analysis

3.1 Dataset Loading and Preprocessing

The implementation uses the CelebA dataset with specific transformations:

```python
def load_transformed_dataset():
    data_transforms = transforms.Compose([
        transforms.Resize((IMG_SIZE, IMG_SIZE)),      # 64x64 resolution
        transforms.RandomHorizontalFlip(),            # Data augmentation
        transforms.ToTensor(),                        # Convert to [0,1]
        transforms.Lambda(lambda t: (t * 2) - 1)     # Scale to [-1, 1]
    ])
    train = torchvision.datasets.CelebA(root=".", download=True,
                                       transform=data_transforms)
    return torch.utils.data.ConcatDataset([train])
```

Listing 1: Dataset preprocessing pipeline

The scaling to $[-1, 1]$ is crucial as it centers the data distribution, improving training stability and generation quality.

3.2 Diffusion Schedule Implementation

```python
def linear_beta_schedule(timesteps, start=0.0001, end=0.02):
    return torch.linspace(start, end, timesteps)

# Pre-calculate terms for efficiency
T = 300
betas = linear_beta_schedule(timesteps=T)
alphas = 1. - betas
alphas_cumprod = torch.cumprod(alphas, axis=0)
sqrt_alphas_cumprod = torch.sqrt(alphas_cumprod)
sqrt_one_minus_alphas_cumprod = torch.sqrt(1. - alphas_cumprod)
```

Listing 2: Beta schedule and derived terms

The linear schedule $\beta_t \in [0.0001, 0.02]$ ensures gradual noise addition. Pre-computing terms like $\sqrt{\bar{\alpha}_t}$ enables efficient forward sampling.

3.3 Forward Diffusion Implementation

The forward diffusion directly implements Equation (3):

```python
def forward_diffusion_sample(x_0, t, device="cuda"):
    noise = torch.randn_like(x_0)
    sqrt_alphas_cumprod_t = get_index_from_list(sqrt_alphas_cumprod, t, x_0.shape)
    sqrt_one_minus_alphas_cumprod_t = get_index_from_list(
        sqrt_one_minus_alphas_cumprod, t, x_0.shape
    )
    # Implements: x_t = sqrt(alpha_bar_t) * x_0 + sqrt(1-alpha_bar_t) * epsilon
    return (sqrt_alphas_cumprod_t.to(device) * x_0.to(device) +
            sqrt_one_minus_alphas_cumprod_t.to(device) * noise.to(device)), noise.to(device)
```

Listing 3: Forward diffusion sampling

This function enables direct sampling of $\mathbf{x}_t$ from $\mathbf{x}_0$ without iterating through intermediate steps, crucial for efficient training.

3.4 U-Net Architecture with Time Conditioning

The neural network architecture combines a U-Net with time embeddings:

3.4.1 Time Embeddings

```python
class SinusoidalPositionEmbeddings(nn.Module):
    def forward(self, time):
        device = time.device
        half_dim = self.dim // 2
        embeddings = math.log(10000) / (half_dim - 1)
        embeddings = torch.exp(torch.arange(half_dim, device=device) * -embeddings)
        embeddings = time[:, None] * embeddings[None, :]
        embeddings = torch.cat((embeddings.sin(), embeddings.cos()), dim=-1)
        return embeddings
```

Listing 4: Sinusoidal position embeddings

Time embeddings allow the network to understand which diffusion timestep it's processing, similar to positional encodings in Transformers.

3.4.2 U-Net Building Blocks

```python
class Block(nn.Module):
    def __init__(self, in_ch, out_ch, time_emb_dim, up=False):
        super().__init__()
        self.time_mlp = nn.Linear(time_emb_dim, out_ch)
        if up:
            self.conv1 = nn.Conv2d(2*in_ch, out_ch, 3, padding=1)  # Skip connections
            self.transform = nn.ConvTranspose2d(out_ch, out_ch, 4, 2, 1)  # Upsample
        else:
            self.conv1 = nn.Conv2d(in_ch, out_ch, 3, padding=1)
            self.transform = nn.Conv2d(out_ch, out_ch, 4, 2, 1)  # Downsample

    def forward(self, x, t):
        h = self.bnorm1(self.relu(self.conv1(x)))
        # Inject time information
        time_emb = self.relu(self.time_mlp(t))
        time_emb = time_emb[(..., ) + (None, ) * 2]  # Broadcast to spatial dims
        h = h + time_emb  # Add time conditioning
        h = self.bnorm2(self.relu(self.conv2(h)))
        return self.transform(h)
```

Listing 5: U-Net block with time conditioning

The time embedding injection allows each layer to adapt its processing based on the current diffusion timestep.

3.5 Training Objective

The loss function implements the simplified DDPM objective:

```
def get_loss(model, x_0, t):
    x_noisy, noise = forward_diffusion_sample(x_0, t, device)
    noise_pred = model(x_noisy, t)
    return F.l1_loss(noise, noise_pred)  # Predict the added noise
```

Listing 6: DDPM training loss

This trains the model to predict the noise ϵ that was added during the forward process, rather than directly predicting $\mathbf{x}_0$ or $\boldsymbol{\mu}$.

3.6 Reverse Sampling

The reverse sampling implements the denoising process:

```
@torch.no_grad()
def sample_timestep(x, t):
    betas_t = get_index_from_list(betas, t, x.shape)
    sqrt_one_minus_alphas_cumprod_t = get_index_from_list(
        sqrt_one_minus_alphas_cumprod, t, x.shape)
    sqrt_recip_alphas_t = get_index_from_list(sqrt_recip_alphas, t, x.shape)

    # Predict x_0 from noise prediction
    model_mean = sqrt_recip_alphas_t * (
        x - betas_t * model(x, t) / sqrt_one_minus_alphas_cumprod_t
    )

    if t == 0:
        return model_mean  # No noise at final step
    else:
        posterior_variance_t = get_index_from_list(posterior_variance, t, x.shape)
        noise = torch.randn_like(x)
        return model_mean + torch.sqrt(posterior_variance_t) * noise
```

Listing 7: Reverse diffusion sampling

This implements the reverse process mean calculation from DDPM theory, using the predicted noise to estimate the denoised image.

4 Training Process

```
for epoch in range(epochs):
    for step, batch in enumerate(dataloader):
        optimizer.zero_grad()
        t = torch.randint(0, T, (BATCH_SIZE,), device=device).long()
```

Algorithm 1 DDPM Training Algorithm

```
1: repeat
2:     Sample $\mathbf{x}_0 \sim q(\mathbf{x}_0)$ from dataset
3:     Sample $t \sim \text{Uniform}(\{1, \ldots, T\})$
4:     Sample $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5:     Compute $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t}\boldsymbol{\epsilon}$
6:     Take gradient descent step on $\nabla_\theta \|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\|^2$
7: until converged
```

```
        loss = get_loss(model, batch[0], t)
        loss.backward()
        optimizer.step()
```

Listing 8: Training loop implementation

The training randomly samples timesteps for each batch, ensuring the model learns to denoise at all noise levels.

5 Generation Process

Algorithm 2 DDPM Sampling Algorithm

```
1: Sample $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
2: for $t = T, T - 1, \ldots, 1$ do
3:     $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\boldsymbol{\epsilon} = \mathbf{0}$
4:     $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\right) + \sigma_t\boldsymbol{\epsilon}$
5: end for
6: return $\mathbf{x}_0$
```

Generation starts with pure noise and progressively denoises over T steps to produce a sample from the learned data distribution.

6 Key Implementation Insights

6.1 Efficiency Optimizations

- **Pre-computed Terms**: All α, β related terms are pre-calculated to avoid repeated computation during training.
- **Direct Sampling**: The forward process allows direct sampling of any timestep without iterating through all previous steps.
- **Batch Processing**: Random timesteps are sampled for each training batch, enabling parallel processing.

6.2 Architectural Choices

- **U-Net Backbone**: Provides the necessary spatial hierarchies for image generation with skip connections preserving fine details.
- **Time Conditioning**: Sinusoidal embeddings inject timestep information throughout the network.

- **Residual Connections**: Skip connections in the U-Net help gradient flow and preserve information across scales.

6.3 Training Stability

- **Noise Prediction**: Predicting noise rather than images provides more stable gradients.
- **L1 Loss**: Less sensitive to outliers compared to L2 loss, improving training robustness.
- **Data Normalization**: Scaling images to $[-1, 1]$ centers the distribution and aids convergence.

7 Comparison with Original DDPM

This implementation faithfully reproduces the core DDPM algorithm:

Component	**DDPM Paper**	**Implementation**
Forward Process	$q(\mathbf{x}_t \vert \mathbf{x}_0) = \mathcal{N}(\sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I})$	`forward_diffusion_sample`
Noise Schedule	Linear $\beta \in [10^{-4}, 0.02]$	`linear_beta_schedule`
Training Loss	$\Vert\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\Vert^2$	`F.l1_loss(noise, noise_pred)`
Architecture	Time-conditioned U-Net	`SimpleUnet` with time embeddings
Sampling	Algorithm 2 from paper	`sample_timestep`

Table 1: Correspondence between DDPM theory and implementation

8 Potential Improvements

Several enhancements could improve this implementation:

- **Attention Mechanisms**: Adding self-attention layers as in later diffusion models
- **Advanced Schedules**: Cosine or learned noise schedules for better generation quality
- **Classifier Guidance**: Conditional generation with class or text conditioning
- **Fast Sampling**: DDIM or other accelerated sampling techniques
- **Progressive Training**: Starting with lower resolutions and increasing during training

9 Conclusion

This implementation provides a clear, educational example of DDPM in action. The code closely follows the mathematical framework from Ho et al. (2020), demonstrating how theoretical advances in generative modeling translate to practical implementations. The modular design makes it easy to understand each component's role in the overall diffusion process.

The success of this approach lies in its simplicity: by framing generation as a learned denoising process, DDPM provides stable training and high-quality results. This implementation serves as an excellent foundation for understanding and extending diffusion-based generative models.

References

[1] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. In *Advances in Neural Information Processing Systems*, volume 33, pages 6840–6851, 2020.

[2] Lilian Weng. What are diffusion models? *Lil'Log*, 2021. `https://lilianweng.github.io/posts/2021-07-11-diffusion-models/`

[3] Niels Rogge and Kashif Rasul. The annotated diffusion model. *Hugging Face Blog*, 2022.

[4] U-Net: Convolutional Networks for Biomedical Image Segmentation *Olaf Ronneberger, Philipp Fischer, and Thomas Brox* `https://arxiv.org/pdf/1505.04597`