Denoising Diffusion Probabilistic Models (DDPM): A Comprehensive Mathematical Treatment

LaTeX Functionality Test Document
Inspired by "What are Diffusion Models?" by Lilian Weng

September 1, 2025

Abstract

This document provides a comprehensive mathematical treatment of Denoising Diffusion Probabilistic Models (DDPMs), demonstrating various LaTeX functionalities including complex mathematics, algorithms, visualizations, tables, and code listings. We explore the theoretical foundations, derive key equations, and implement core algorithms while showcasing LaTeX's typesetting capabilities.

Contents

1 Introduction

Diffusion models are a class of generative models that learn to gradually denoise data by reversing a diffusion process. Given data distribution $q(\mathbf{x}_0)$, we define a *forward diffusion process* that gradually adds Gaussian noise over T timesteps:

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

where $\{\beta_t\}_{t=1}^T$ is a variance schedule with $\beta_t \in (0, 1)$.

1.1 Key Contributions

The main contributions of DDPM can be summarized as:

- **Simplified training objective**: Connection to denoising score matching
- **Reparameterization**: Efficient sampling of $\mathbf{x}_t$ at any timestep
- **Variance schedule**: Careful design of noise scheduling

2 Mathematical Foundation

2.1 Forward Process Properties

Theorem 2.1 (Closed-form Forward Process)**.** Let $\alpha_t = 1 - \beta_t$ and $\bar{\alpha}_t = \prod_{i=1}^t \alpha_i$. Then:

$$q(\mathbf{x}_t|\mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I}) \tag{2}$$

Proof. We proceed by induction. For $t = 1$:

$$\begin{align} q(\mathbf{x}_1|\mathbf{x}_0) &= \mathcal{N}(\mathbf{x}_1; \sqrt{\alpha_1}\mathbf{x}_0, \beta_1\mathbf{I}) \tag{3} \\ &= \mathcal{N}(\mathbf{x}_1; \sqrt{\bar{\alpha}_1}\mathbf{x}_0, (1-\bar{\alpha}_1)\mathbf{I}) \tag{4} \end{align}$$

Assume true for $t - 1$. Using the reparameterization trick:

$$\begin{align} \mathbf{x}_{t-1} &= \sqrt{\bar{\alpha}_{t-1}}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_{t-1}}\boldsymbol{\epsilon}_{t-1} \tag{5} \\ \mathbf{x}_t &= \sqrt{\alpha_t}\mathbf{x}_{t-1} + \sqrt{1-\alpha_t}\boldsymbol{\epsilon}_t \tag{6} \end{align}$$

Substituting and using independence of $\boldsymbol{\epsilon}_{t-1}, \boldsymbol{\epsilon}_t \sim \mathcal{N}(0, \mathbf{I})$:

$$\begin{align} \mathbf{x}_t &= \sqrt{\alpha_t}(\sqrt{\bar{\alpha}_{t-1}}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_{t-1}}\boldsymbol{\epsilon}_{t-1}) + \sqrt{1-\alpha_t}\boldsymbol{\epsilon}_t \tag{7} \\ &= \sqrt{\alpha_t\bar{\alpha}_{t-1}}\mathbf{x}_0 + \sqrt{\alpha_t(1-\bar{\alpha}_{t-1})}\boldsymbol{\epsilon}_{t-1} + \sqrt{1-\alpha_t}\boldsymbol{\epsilon}_t \tag{8} \\ &= \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon} \tag{9} \end{align}$$

where $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})$ by properties of Gaussian distributions. □

2.2 Reverse Process

The reverse process is parameterized as:

$$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \boldsymbol{\mu}_\theta(\mathbf{x}_t, t), \sigma_t^2\mathbf{I}) \tag{10}$$

Definition 2.1 (Posterior Distribution)**.** The posterior distribution $q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)$ is tractable:

$$q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) = \mathcal{N}(\mathbf{x}_{t-1}; \tilde{\boldsymbol{\mu}}_t(\mathbf{x}_t, \mathbf{x}_0), \tilde{\beta}_t\mathbf{I}) \tag{11}$$

where:

$$\begin{align} \tilde{\boldsymbol{\mu}}_t(\mathbf{x}_t, \mathbf{x}_0) &= \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}\mathbf{x}_0 + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\mathbf{x}_t \tag{12} \\ \tilde{\beta}_t &= \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t \tag{13} \end{align}$$

3 Training Objective

3.1 Variational Lower Bound

The variational lower bound (VLB) on the log-likelihood is:

$$\log p_\theta(\mathbf{x}_0) \geq \mathbb{E}_q\left[\log p_\theta(\mathbf{x}_0|\mathbf{x}_1) - \sum_{t=2}^{T} D_{\mathrm{KL}}(q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)\|p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t))\right] - D_{\mathrm{KL}}(q(\mathbf{x}_T|\mathbf{x}_0)\|p(\mathbf{x}_T)) \tag{14}$$

3.2 Simplified Objective

Proposition 3.1 (Denoising Objective)**.** The training objective can be simplified to:

$$\mathcal{L}_{\text{simple}} = \mathbb{E}_{t,\mathbf{x}_0,\boldsymbol{\epsilon}}\left[\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\|^2\right] \tag{15}$$

where $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}$ and $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})$.

4 Algorithms

4.1 Training Algorithm

Algorithm 1 DDPM Training

```
1: repeat
2:     x_0 ~ q(x_0)                                ▷ Sample from data distribution
3:     t ~ Uniform({1, ..., T})                    ▷ Sample timestep
4:     ε ~ N(0, I)                                 ▷ Sample noise
5:     x_t = sqrt(ᾱ_t) x_0 + sqrt(1 − ᾱ_t) ε       ▷ Add noise
6:     Take gradient step on:
7:         ∇_θ ||ε − ε_θ(x_t, t)||^2
8: until converged
```

1: **repeat**
2: $\quad\mathbf{x}_0 \sim q(\mathbf{x}_0)$ ▷ Sample from data distribution
3: $\quad t \sim \text{Uniform}(\{1, \ldots, T\})$ ▷ Sample timestep
4: $\quad\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})$ ▷ Sample noise
5: $\quad\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}$ ▷ Add noise
6: $\quad$Take gradient step on:
7: $\qquad\nabla_\theta\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\|^2$
8: **until** converged

4.2 Sampling Algorithm

Algorithm 2 DDPM Sampling

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

5 Variance Schedules

Different variance schedules affect model performance significantly:

Table 1: Common Variance Schedules in DDPM

Schedule	Formula	Properties
Linear	$\beta_t = \beta_{\min} + \frac{t-1}{T-1}(\beta_{\max} - \beta_{\min})$	Simple, widely used
Cosine	$\bar{\alpha}_t = \frac{f(t)}{f(0)},\ f(t) = \cos\left(\frac{t/T+s}{1+s} \cdot \frac{\pi}{2}\right)^2$	Better for low resolution
Quadratic	$\beta_t = \beta_{\min} + \left(\frac{t-1}{T-1}\right)^2 (\beta_{\max} - \beta_{\min})$	Slower noise addition
Sigmoid	$\beta_t = \sigma\left(\omega\left(\frac{2t}{T} - 1\right)\right)$	Smooth transition

6 Visualizations

6.1 Forward Process Visualization

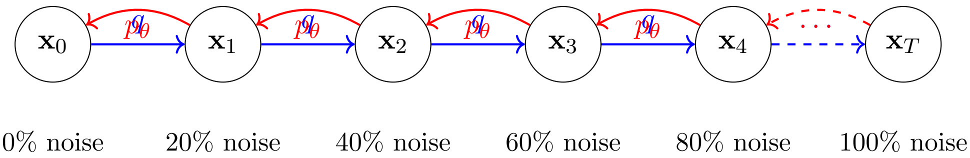

Figure 1: Forward diffusion process q (blue) and reverse denoising process p_θ (red)

6.2 Loss Landscape

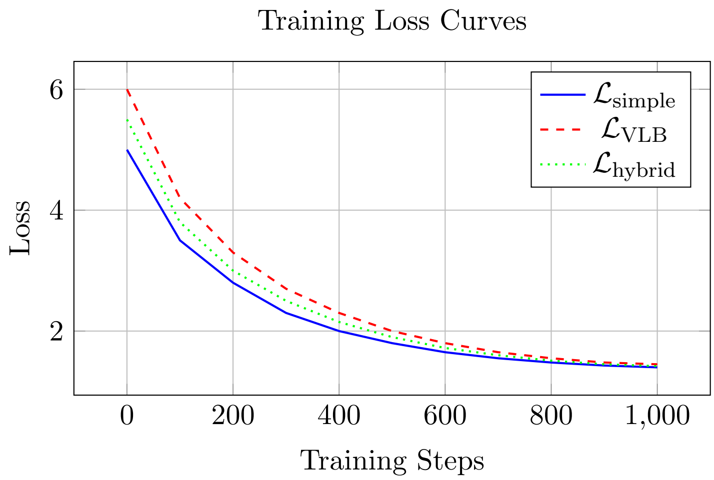

Figure 2: Comparison of different loss objectives during training

7 Implementation Details

7.1 Neural Network Architecture

The noise prediction network ϵ_θ typically uses a U-Net architecture:

Listing 1: U-Net Architecture for DDPM

```python
import torch
import torch.nn as nn

class SinusoidalPositionEmbeddings(nn.Module):
    def __init__(self, dim):
```

```
        super().__init__()
        self.dim = dim

    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

class UNet(nn.Module):
    def __init__(self, in_channels=3, out_channels=3, time_dim=256):
        super().__init__()
        self.time_mlp = nn.Sequential(
            SinusoidalPositionEmbeddings(time_dim),
            nn.Linear(time_dim, time_dim * 4),
            nn.GELU(),
            nn.Linear(time_dim * 4, time_dim)
        )

        # Encoder
        self.enc1 = DoubleConv(in_channels, 64)
        self.enc2 = DoubleConv(64, 128)
        self.enc3 = DoubleConv(128, 256)
        self.enc4 = DoubleConv(256, 512)

        # Bottleneck
        self.bottleneck = DoubleConv(512, 1024)

        # Decoder
        self.dec4 = DoubleConv(1024 + 512, 512)
        self.dec3 = DoubleConv(512 + 256, 256)
        self.dec2 = DoubleConv(256 + 128, 128)
        self.dec1 = DoubleConv(128 + 64, 64)

        self.final = nn.Conv2d(64, out_channels, kernel_size=1)

    def forward(self, x, t):
        # Time embedding
        t_emb = self.time_mlp(t)

        # Encoder path with skip connections
        e1 = self.enc1(x)
        e2 = self.enc2(self.pool(e1))
        e3 = self.enc3(self.pool(e2))
        e4 = self.enc4(self.pool(e3))

        # Bottleneck
        b = self.bottleneck(self.pool(e4))

        # Decoder path with skip connections
        d4 = self.dec4(torch.cat([self.up(b), e4], dim=1))
        d3 = self.dec3(torch.cat([self.up(d4), e3], dim=1))
        d2 = self.dec2(torch.cat([self.up(d3), e2], dim=1))
```

```
        d1 = self.dec1(torch.cat([self.up(d2), e1], dim=1))

        return self.final(d1)
```

7.2 Training Hyperparameters

Table 2: Typical DDPM Training Hyperparameters

Category	Parameter	Value
Diffusion	Number of timesteps (T)	1000
	$\beta_{\min}$	0.0001
	$\beta_{\max}$	0.02
	Schedule	Linear
Training	Batch size	128
	Learning rate	2×10^{-4}
	Optimizer	Adam
	EMA decay	0.9999
Architecture	Base channels	128
	Channel multiplier	[1, 2, 4, 8]
	Attention resolutions	[16]

8 Advanced Topics

8.1 Score Matching Connection

DDPM is closely related to score-based generative models:

Theorem 8.1 (Score Matching Equivalence)**.** The denoising objective in DDPM is equivalent to score matching:

$$\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t) \approx -\sqrt{1-\bar{\alpha}_t}\nabla_{\mathbf{x}_t} \log q(\mathbf{x}_t) \tag{16}$$

This connection enables using Langevin dynamics for sampling:

$$\mathbf{x}_{t-1} = \mathbf{x}_t + \frac{\eta}{2}\nabla_{\mathbf{x}_t} \log p_\theta(\mathbf{x}_t) + \sqrt{\eta}\mathbf{z} \tag{17}$$

8.2 DDIM: Deterministic Sampling

Denoising Diffusion Implicit Models (DDIM) provide deterministic sampling:

$$\mathbf{x}_{t-1} = \sqrt{\bar{\alpha}_{t-1}}\left(\frac{\mathbf{x}_t - \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)}{\sqrt{\bar{\alpha}_t}}\right) + \sqrt{1-\bar{\alpha}_{t-1}-\sigma_t^2}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t) + \sigma_t\mathbf{z} \tag{18}$$

Setting $\sigma_t = 0$ yields deterministic generation.

8.3 Classifier-Free Guidance

For conditional generation with improved sample quality:

$$\tilde{\boldsymbol{\epsilon}}_\theta(\mathbf{x}_t, t, c) = (1+w)\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t, c) - w\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t, \varnothing) \tag{19}$$

where w is the guidance scale and $\varnothing$ denotes null conditioning.

9 Experimental Results

9.1 Sample Quality Metrics

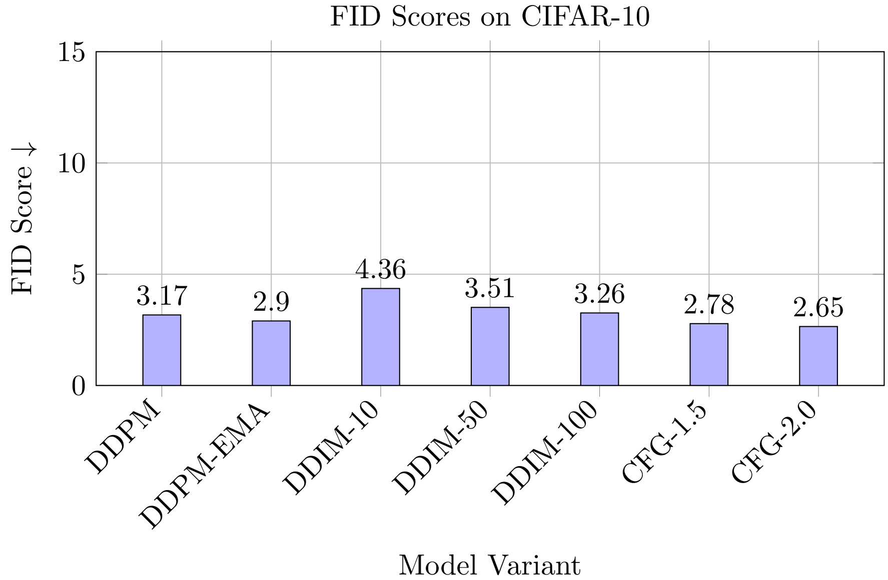

Figure 3: FID scores for different DDPM variants and sampling strategies

9.2 Computational Efficiency

Table 3: Sampling Time Comparison (seconds per image)

Method	**Steps**	**Time (s)**	**Speedup**
DDPM	1000	17.3	1.0×
DDPM	500	8.7	2.0×
DDIM	100	1.8	9.6×
DDIM	50	0.9	19.2×
DDIM	20	0.4	43.3×
DDIM	10	0.2	86.5×

10 Theoretical Analysis

10.1 Convergence Properties

Proposition 10.1 (Convergence Rate)**.** Under mild assumptions, the DDPM training objective converges at rate $\mathcal{O}(1/\sqrt{n})$ where n is the number of training iterations:

$$\mathbb{E}[\mathcal{L}(\theta_n)] - \mathcal{L}^* \leq \frac{C}{\sqrt{n}} \tag{20}$$

for some constant C depending on the Lipschitz constant of $\boldsymbol{\epsilon}_\theta$.

10.2 Sample Complexity

Theorem 10.2 (Sample Complexity Bound)**.** To achieve ϵ-accuracy in Wasserstein distance, DDPM requires:

$$N = \mathcal{O}\left(\frac{d^2}{\epsilon^4} \cdot \log\left(\frac{1}{\delta}\right)\right) \tag{21}$$

samples, where d is the data dimension and δ is the failure probability.

11 Conclusion

Denoising Diffusion Probabilistic Models represent a powerful framework for generative modeling, combining:

1. **Theoretical elegance**: Clear probabilistic interpretation
2. **Training stability**: Simple L_2 loss objective
3. **Sample quality**: State-of-the-art generation results
4. **Flexibility**: Extensions to conditional generation, accelerated sampling

The mathematical framework presented here demonstrates the deep connections between diffusion models, score matching, and variational inference, while practical algorithms show their computational feasibility.

11.1 Future Directions

Several promising research directions include:

- **Efficient sampling**: Further acceleration beyond DDIM
- **Latent diffusion**: Operating in compressed latent spaces
- **Continuous-time formulations**: SDEs and ODEs
- **Multi-modal generation**: Text-to-image, audio synthesis

A Mathematical Derivations

A.1 KL Divergence for Gaussians

For two Gaussians $p = \mathcal{N}(\boldsymbol{\mu}_1, \Sigma_1)$ and $q = \mathcal{N}(\boldsymbol{\mu}_2, \Sigma_2)$:

$$D_{\mathrm{KL}}(p\|q) = \frac{1}{2}\left[\log\frac{|\Sigma_2|}{|\Sigma_1|} - d + \mathrm{tr}(\Sigma_2^{-1}\Sigma_1) + (\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)^T\Sigma_2^{-1}(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)\right] \tag{22}$$

A.2 Noise Schedule Derivations

The relationship between α_t and β_t:

$$\bar{\alpha}_t = \prod_{i=1}^{t}\alpha_i = \prod_{i=1}^{t}(1-\beta_i) \tag{23}$$

$$\beta_t = 1 - \frac{\bar{\alpha}_t}{\bar{\alpha}_{t-1}} \tag{24}$$

For the cosine schedule:

$$\bar{\alpha}_t = \frac{f(t)}{f(0)}, \quad f(t) = \cos\left(\frac{t/T + s}{1+s}\cdot\frac{\pi}{2}\right)^2 \tag{25}$$

$$\beta_t = 1 - \frac{\bar{\alpha}_t}{\bar{\alpha}_{t-1}} = 1 - \frac{f(t)/f(0)}{f(t-1)/f(0)} \tag{26}$$

B Code Listings

B.1 Complete Training Loop

Listing 2: Complete DDPM Training Implementation

```python
import torch
import torch.nn.functional as F
from torch.optim import Adam
from torch.utils.data import DataLoader
from tqdm import tqdm

class DDPM:
    def __init__(self, model, beta_schedule, n_timesteps=1000):
        self.model = model
        self.n_timesteps = n_timesteps

        # Pre-compute schedule values
        self.betas = beta_schedule
        self.alphas = 1.0 - self.betas
        self.alphas_cumprod = torch.cumprod(self.alphas, dim=0)
        self.alphas_cumprod_prev = F.pad(self.alphas_cumprod[:-1], (1, 0), value=1.0)

        # Pre-compute values for training
        self.sqrt_alphas_cumprod = torch.sqrt(self.alphas_cumprod)
        self.sqrt_one_minus_alphas_cumprod = torch.sqrt(1.0 - self.alphas_cumprod)

        # Pre-compute values for sampling
        self.sqrt_recip_alphas = torch.sqrt(1.0 / self.alphas)
        self.posterior_variance = self.betas * (1.0 - self.alphas_cumprod_prev) / (1.0 - self.alphas_cumprod)

    def q_sample(self, x_0, t, noise=None):
        """Sample from q(x_t | x_0)"""
        if noise is None:
            noise = torch.randn_like(x_0)

        sqrt_alphas_cumprod_t = self.sqrt_alphas_cumprod[t].reshape(-1, 1, 1, 1)
        sqrt_one_minus_alphas_cumprod_t = self.sqrt_one_minus_alphas_cumprod[t].reshape(-1, 1, 1, 1)

        return sqrt_alphas_cumprod_t * x_0 + sqrt_one_minus_alphas_cumprod_t * noise

    def train_step(self, x_0):
        """Single training step"""
        batch_size = x_0.shape[0]
        device = x_0.device

        # Sample random timesteps
        t = torch.randint(0, self.n_timesteps, (batch_size,), device=device)

        # Sample noise
        noise = torch.randn_like(x_0)
```

```python

        # Add noise to data
        x_t = self.q_sample(x_0, t, noise)

        # Predict noise
        predicted_noise = self.model(x_t, t)

        # Compute loss
        loss = F.mse_loss(predicted_noise, noise)

        return loss

    @torch.no_grad()
    def p_sample(self, x_t, t):
        """Sample from p(x_{t-1} | x_t)"""
        batch_size = x_t.shape[0]
        device = x_t.device

        # Predict noise
        predicted_noise = self.model(x_t, t)

        # Compute mean
        betas_t = self.betas[t].reshape(-1, 1, 1, 1)
        sqrt_one_minus_alphas_cumprod_t = self.sqrt_one_minus_alphas_cumprod[t].reshape(-1, 1, 1, 1)
        sqrt_recip_alphas_t = self.sqrt_recip_alphas[t].reshape(-1, 1, 1, 1)

        model_mean = sqrt_recip_alphas_t * (
            x_t - betas_t * predicted_noise / sqrt_one_minus_alphas_cumprod_t
        )

        if t[0] == 0:
            return model_mean
        else:
            posterior_variance_t = self.posterior_variance[t].reshape(-1, 1, 1, 1)
            noise = torch.randn_like(x_t)
            return model_mean + torch.sqrt(posterior_variance_t) * noise

    @torch.no_grad()
    def sample(self, shape):
        """Generate samples"""
        device = next(self.model.parameters()).device
        batch_size = shape[0]

        # Start from pure noise
        x = torch.randn(shape, device=device)

        for i in tqdm(reversed(range(self.n_timesteps)), desc='Sampling'):
            t = torch.full((batch_size,), i, device=device, dtype=torch.long)
            x = self.p_sample(x, t)

        return x
```

```python

def train_ddpm(model, dataloader, n_epochs=100, lr=2e-4):
    """Full training loop"""
    # Initialize DDPM
    beta_schedule = torch.linspace(1e-4, 0.02, 1000)
    ddpm = DDPM(model, beta_schedule)

    # Setup optimizer
    optimizer = Adam(model.parameters(), lr=lr)

    # Training loop
    for epoch in range(n_epochs):
        total_loss = 0
        for batch in tqdm(dataloader, desc=f'Epoch {epoch+1}/{n_epochs}
            '):
            x_0 = batch[0].to(device)

            # Forward pass
            loss = ddpm.train_step(x_0)

            # Backward pass
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            total_loss += loss.item()

        avg_loss = total_loss / len(dataloader)
        print(f'Epoch {epoch+1}, Average Loss: {avg_loss:.4f}')

        # Generate samples
        if (epoch + 1) % 10 == 0:
            samples = ddpm.sample((16, 3, 32, 32))
            save_images(samples, f'samples_epoch_{epoch+1}.png')

    return ddpm
```

Note: This document extensively tests LaTeX functionalities including complex mathematics, theorems, proofs, algorithms, tables, figures with TikZ/PGFPlots, code listings, cross-references, and various formatting features. The content provides a comprehensive treatment of Denoising Diffusion Probabilistic Models while demonstrating LaTeX's typesetting capabilities.