Diffusion Models for Medical Imaging

RISE-MICCAI Summer School

Jorge Cardoso

King's College London

July 15, 2025

Tutorial Overview

What We Will Achieve Today

Part 1: DDPM

- Build a Denoising Diffusion Model
- Generate synthetic hand X-rays
- Understand the noise prediction process
- Visualize the denoising chain

Part 2: LDM

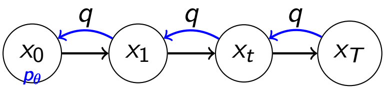
- Implement Latent Diffusion Models
- Train an autoencoder for compression
- Work in compressed latent space
- Generate higher quality images efficiently

Goal: Master diffusion models for medical image generation using MONAI

Outline

What are Diffusion Models?

- Generative models that learn to create data by reversing a gradual noising process
- Based on the paper: "Denoising Diffusion Probabilistic Models" by Ho et al. (2020)
- Key idea:
  - Forward process: Gradually add noise to data
  - Reverse process: Learn to denoise step by step
- Applications: Image synthesis, super-resolution, inpainting



Part 1: DDPM Tutorial Overview

- **Goal**: Train a DDPM to generate synthetic 2D medical images
- **Dataset**: MedNIST "Hand" X-ray images
  - 7,999 training images
  - 64×64 pixel resolution
  - Grayscale medical images
- **Framework**: MONAI's generative module
- **Training time**: 50 minutes for 75 epochs
- **Key components**:
  - U-Net architecture with attention mechanisms
  - DDPM scheduler (1000 timesteps)
  - Diffusion inferer for training and sampling

Architecture Components

1. **DiffusionModelUNet**
   - 2D U-Net backbone
   - Channel progression: 128 $\rightarrow$ 256 $\rightarrow$ 256
   - Attention at levels 2 and 3
   - 256 attention head channels
2. **DDPMScheduler**
   - 1000 timesteps
   - Linear noise schedule
   - Handles β_t scheduling
3. **DiffusionInferer**
   - Manages forward diffusion
   - Handles reverse sampling
   - Integrates model and scheduler
4. **Training Setup**
   - Adam optimizer (lr=2.5e-5)
   - Mixed precision training
   - MSE loss on noise prediction

Data Pipeline

Data Transformations:

```python
train_transforms = transforms.Compose([
    transforms.LoadImaged(keys=["image"]),
    transforms.EnsureChannelFirstd(keys=["image"]),
    transforms.ScaleIntensityRanged(
        keys=["image"],
        a_min=0.0, a_max=255.0,  # Input range
        b_min=0.0, b_max=1.0,    # Output range
        clip=True
    ),
    transforms.RandAffined(
        keys=["image"],
        rotate_range=[(-pi/36, pi/36), (-pi/36, pi/36)],
        translate_range=[(-1, 1), (-1, 1)],
        scale_range=[(-0.05, 0.05), (-0.05, 0.05)],
        spatial_size=[64, 64],
        padding_mode="zeros",
        prob=0.5
    )
])
```

Key Concept 1: Understanding the Training Process

Forward Diffusion Process:

1. Sample random timesteps t
2. Generate random noise ϵ
3. Create noisy images:
   $x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon$
4. Model predicts the noise $\epsilon_\theta(x_t, t)$
5. Loss: $\mathcal{L} = ||\epsilon - \epsilon_\theta(x_t, t)||^2$

Key Insight:

- We're training the model to predict *noise*, not images!
- The model learns the noise distribution at each timestep
- During inference, we iteratively remove predicted noise

Key Concept 2: The Inferer Abstraction


```
# Training: Forward diffusion + noise prediction
with autocast(enabled=True):
    noise = torch.randn_like(images)
    timesteps = torch.randint(0, 1000, (batch_size,))

    # This handles the forward diffusion internally!
    noise_pred = inferer(
        inputs=images,
        diffusion_model=model,
        noise=noise,
        timesteps=timesteps
    )

    loss = F.mse_loss(noise_pred, noise)
```

What happens inside the inferer:

- Applies forward diffusion to create x_t
- Passes x_t and t to the U-Net
- Returns predicted noise

Key Concept 3: Sampling Process

Reverse Diffusion (Sampling):

```
# Start from pure noise
noise = torch.randn((1, 1, 64, 64))
scheduler.set_timesteps(num_inference_steps=1000)

# Iteratively denoise
image = inferer.sample(
    input_noise=noise,
    diffusion_model=model,
    scheduler=scheduler
)
```

What happens during sampling:

- Start with $x_T \sim \mathcal{N}(0, I)$
- For $t = T, T-1, ..., 1$:
  - Predict noise: $\epsilon_\theta(x_t, t)$
  - Apply denoising step: $x_{t-1} = \frac{1}{\sqrt{\alpha_t}}(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta)$
  - Add controlled noise (except at $t = 0$)
- Result: Generated image x_0

Key Concept 4: Memory and Computation

Challenges:

- Each training step requires:
  - Random timestep sampling
  - Noise generation
  - U-Net forward pass
- Memory considerations:
  - Batch size = 128 (adjust based on GPU)
  - Mixed precision training (GradScaler)
  - Persistent workers for data loading

Optimization strategies used:

- `autocast` for automatic mixed precision
- `CacheDataset` to avoid repeated transforms
- `persistent_workers=True` in DataLoader
- `optimizer.zero_grad(set_to_none=True)`

Training Results

Training Progress:

- 75 epochs total
- Validation every 5 epochs
- Progressive quality improvement
- MSE loss on noise prediction

Sampling Visualization:

- Shows denoising process
- 1000 steps from noise to image
- Intermediate steps saved every 100 timesteps

Expected Outcomes:

- Generated hand X-rays
- Similar style to training data
- Novel, unique samples
- Quality improves with training

Visualization includes:

- Learning curves (train/val loss)
- Sample generations during training
- Full denoising chain visualization

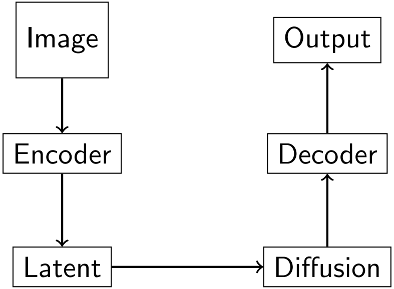
Part 2: Latent Diffusion Models

Motivation:

- DDPMs work in pixel space $\rightarrow$ computationally expensive
- Medical images often high-resolution
- Solution: Work in compressed latent space

Key Innovation:

- Train autoencoder to compress images
- Run diffusion in latent space
- Decode back to image space
- Much more efficient!



LDM Architecture Overview

Two-Stage Training Process:

1. **Stage 1: Autoencoder Training**
   - AutoencoderKL with KL-regularization
   - Compression: $64 \times 64 \times 1 \rightarrow 16 \times 16 \times 3$ (latent)
   - Losses: Reconstruction + Perceptual + Adversarial + KL
   - Training time: 55 minutes
2. **Stage 2: Diffusion Model Training**
   - U-Net operates on latent representations
   - Smaller spatial dimensions $\rightarrow$ faster training
   - Same DDPM principles apply
   - Training time: 80 minutes

Autoencoder Architecture


```
autoencoderkl = AutoencoderKL(
    spatial_dims=2,
    in_channels=1,
    out_channels=1,
    num_channels=(128, 128, 256),
    latent_channels=3,
    num_res_blocks=2,
    attention_levels=(False, False, False),
    with_encoder_nonlocal_attn=False,
    with_decoder_nonlocal_attn=False,
)
```

Key Features:

- Variational autoencoder with KL regularization
- $4\times$ spatial compression ($64\times64 \rightarrow 16\times16$)
- 3 latent channels for richer representation
- No attention layers (faster training)

Autoencoder Loss Functions

Combined Loss Function:

$$\mathcal{L}_{AE} = \mathcal{L}_{recon} + \lambda_{KL}\mathcal{L}_{KL} + \lambda_{perc}\mathcal{L}_{perc} + \lambda_{adv}\mathcal{L}_{adv}$$

- **Reconstruction Loss** ($\mathcal{L}_{recon}$): L1 distance between input and output
- **KL Loss** ($\mathcal{L}_{KL}$): Regularizes latent distribution

$$\mathcal{L}_{KL} = \frac{1}{2}\sum[\mu^2 + \sigma^2 - \log(\sigma^2) - 1]$$

- **Perceptual Loss** ($\mathcal{L}_{perc}$): Uses AlexNet features (weight: 0.001)
- **Adversarial Loss** ($\mathcal{L}_{adv}$): PatchGAN discriminator (weight: 0.01)

Note: Adversarial loss starts after 10 epochs warm-up

Latent Diffusion Model

Diffusion U-Net for Latent Space:

- Input/Output: 3-channel latent representations
- Architecture: Similar to DDPM but smaller spatial size
- Channel progression: $128 \rightarrow 256 \rightarrow 512$
- Attention at levels 2 and 3 (256, 512 head channels)

Key Difference from DDPM:

- Works on $16 \times 16 \times 3$ latents instead of $64 \times 64 \times 1$ images
- $16\times$ reduction in spatial dimensions
- Significantly faster training and sampling

Important: Scaling Factor

Why Scaling Factor Matters:

- Latent space may not have unit variance
- Diffusion assumes Gaussian with unit variance
- Mismatch can hurt performance

```python
# Calculate scaling factor
with torch.no_grad():
    z = autoencoderkl.encode_stage_2_inputs(images)

scale_factor = 1 / torch.std(z)
print(f"Scaling factor: {scale_factor}")

# Use in inferer
inferer = LatentDiffusionInferer(
    scheduler,
    scale_factor=scale_factor
)
```

Note: This normalizes the latent space to match diffusion assumptions

LDM Training Process

```python
# Encode images to latent space
z_mu, z_sigma = autoencoderkl.encode(images)
z = autoencoderkl.sampling(z_mu, z_sigma)

# Standard diffusion training in latent space
noise = torch.randn_like(z)
timesteps = torch.randint(0, 1000, (batch_size,))

# Inferer handles scaling automatically
noise_pred = inferer(
    inputs=images,
    diffusion_model=unet,
    noise=noise,
    timesteps=timesteps,
    autoencoder_model=autoencoderkl
)

loss = F.mse_loss(noise_pred, noise)
```

LDM Sampling


```
# Start with noise in LATENT space
z = torch.randn((1, 3, 16, 16))

# Sample using both models
scheduler.set_timesteps(num_inference_steps=1000)
decoded = inferer.sample(
    input_noise=z,
    diffusion_model=unet,
    scheduler=scheduler,
    autoencoder_model=autoencoderkl
)
```

Sampling Process:

1. Generate random latent noise ($16\times16\times3$)
2. Run reverse diffusion in latent space
3. Decode final latent to image space
4. Result: $64\times64\times1$ generated image

DDPM vs LDM Comparison

Aspect	**DDPM**	**LDM**
Working Space	Pixel ($64\times64\times1$)	Latent ($16\times16\times3$)
Model Parameters	Smaller U-Net	Larger U-Net + Autoencoder
Training Time	50 min	135 min total
Sampling Speed	Slower	$16\times$ faster
Memory Usage	Higher	Lower (per diffusion step)
Image Quality	Good	Better (perceptual loss)
Scalability	Limited by resolution	Excellent

Expected Results

Autoencoder (Stage 1):

- Reconstruction quality improves over epochs
- Latent space becomes well-regularized
- Adversarial loss stabilizes after warm-up

Diffusion Model (Stage 2):

- Similar loss curves to DDPM
- Faster convergence due to smaller space
- Generated samples show:
  - Sharp details from perceptual loss
  - Consistent anatomy
  - Diverse poses and orientations

Visualization: Complete denoising chain from latent noise to final image

Tips for Both Models

General Best Practices:

- Start with provided hyperparameters
- Monitor GPU memory usage
- Use mixed precision training
- Save checkpoints regularly

DDPM Specific:

- Batch size affects training stability
- Consider DDIM for faster sampling
- Experiment with noise schedules

LDM Specific:

- Train autoencoder to convergence first
- Check reconstruction quality before Stage 2
- Scaling factor is crucial for performance
- Balance autoencoder losses carefully

Common Issues and Solutions

Memory Issues:

- Reduce batch size
- Use gradient accumulation
- Enable AMP (automatic mixed precision)

Poor Generation Quality:

- Train longer
- Check data preprocessing
- Verify scaling factor (LDM)
- Adjust learning rates

Training Instability:

- Lower learning rate
- Gradient clipping
- Check for NaN losses
- Ensure proper data normalization

Questions and Resources

Resources:

- DDPM paper: arxiv.org/abs/2006.11239
- LDM paper: arxiv.org/abs/2112.10752
- MONAI Generative: github.com/Project-MONAI/GenerativeModels
- Tutorial notebooks: Available in MONAI tutorials repository

Further Reading:

- Medical diffusion review: Pinaya et al. (2022)
- DDIM for faster sampling
- Classifier-free guidance
- 3D medical image generation

Questions?