Performance Analysis for Projection-Correction Methods in Motion Deblurring Problems

Sara Casadio, Enrico Ferraiolo, Giovanni Maria Savoca

Alma Mater Studiorum - University of Bologna
Master's Degree in Computer Science

June 4, 2025

- The project analyzes the performance of two **Projection-Correction** algorithms for reconstructing medical images affected by **motion blur**.
- The studied algorithms are:
  - **Diffusion Posterior Sampling (DPS)**
  - **Regularization by Denoising with Diffusion (RED-Diff)**
- Both methods are based on **pre-trained diffusion models**.
- Objective: evaluate the effectiveness of these methods in recovering degraded images.

Approach to the Problem

- **Objective**: Analyze the performance of *Projection-Correction* methods **DPS** and **RED-Diff** for motion blur removal on medical images
- **Phase 1**: Dataset preprocessing (128x128)
- **Phase 2**: Data augmentation to increase dataset diversity
- **Phase 3**: Training a DDIM diffusion model on medical data
- **Phase 4**: Simulation of motion blur and its removal
- **Phase 5**: Implementation and comparison of *Projection-Correction* methods: **DPS** and **RED-Diff**
- **Phase 6**: Quantitative evaluation of performance using metrics such as **PSNR** and **SSIM**

Dataset

- We use the "Mayo Clinic CT Dataset" of low-dose CT scans, available via the link provided in this report.
- It contains a total of 6,400 2D slices in PNG format, extracted from 20 different patients.
- The images are organized into:
  - `raw_data/train/`: 5,120 slices for training (80% of the dataset)
  - `raw_data/test/`: 1,280 slices for testing (20% of the dataset)

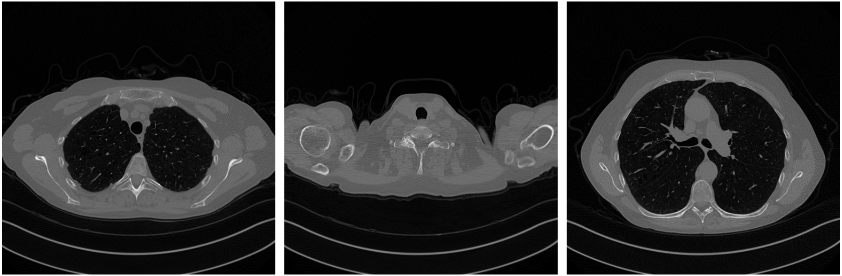

Figure: Examples of CT slices from the Mayo Clinic dataset

Conversion Pipeline

Before applying augmentations, each image is converted using:

1. **Grayscale**: single channel via `transforms.Grayscale(num_output_channels=1)`
2. **Resize**: to 128×128 pixels using bicubic interpolation
3. **Normalization**: values scaled to $[-1, 1]$ using mean 0.5 and std 0.5

```
base_transform = transforms.Compose([
    transforms.Grayscale(1),
    transforms.Resize((128,128), interpolation=Image.BICUBIC),
    transforms.ToTensor(),
    transforms.Normalize([0.5], [0.5]),
])
```

Data Augmentation: Types

For each clean image, we apply the following transformations:

- **Fixed rotations**: $\pm 5^\circ$ via `rotate_fixed()`
- **Horizontal flip**: `horizontal_flip()`
- **Gaussian noise**: mean 0, std 10 via `add_gaussian_noise()`
- **Salt-and-pepper noise**: probability 2% via `add_salt_pepper()`
- **Brightness adjustment**: factor 1.2 via `change_brightness()`
- **Contrast adjustment**: factor 1.3 via `change_contrast()`

Repository Organization

- `raw_data/`: directory containing CT slice images (`train/`, `test/`)
- `checkpoints/`: saved model weights (`*.pth`)
- `scripts/`: main scripts for training and evaluation
- `utils.py`: module with utility functions (dataset, model, checkpoint I/O)
- `notebooks/`: exploratory and prototyping notebooks
- `result/`: output images, plots, and metrics
- `report/`: report materials (`media/`, `capitoli/`)

Training Pipeline

- **Objective**: Train a denoising diffusion model (DDIM U-Net) on grayscale images
- **Main Components**:
  1. Data Augmentation
  2. DataLoader
  3. Model Compilation
  4. Training loop with mixed-precision

Data Augmentation

- **Base Dataset**: Dataset Mayo
  - Grayscale $\rightarrow$ 1 channel
  - Resize images to 128 $\times$ 128
- **Augmentations** (8 types):
  - *None*: no transformation
  - `Rotation` $\pm 5^\circ$ (rotation + centering)
  - `Flip` horizontal
  - `Gaussian noise` (mean=0, std=10)
  - `Salt and pepper noise` (prob=2%)
  - `Brightness` (factor=1.2)
  - `Contrast` (factor=1.3)

Schedulers for Diffusion

- **DDPMScheduler** for training diffusion process
  - `Timesteps` 1000
- **DDIMScheduler** for sampling
  - `Timesteps` 1000

Compiling the Model

- **Why**: optimize the model for better performance
- **Usage**:

```
model = torch.compile(model)
```

- **Benefits**: improved batch throughput

Mixed-Precision with AMP

- **GradScaler amd autocast**:
  - `GradScaler` for scaling gradients
  - `autocast` for automatic mixed precision
- Reduce memory usage and speed up training

Training Loop

1. Loss function: `MSE`
2. Start the training `model.train()`
3. For each epoch:
    - Move images to GPU (if available)
    - Generate noise and timesteps
    - Compute noise prediction on the input data
    - Prediction + MSE loss
    - Optimization + `scheduler.step()`
4. Save validation samples to visualize the model performance during training
5. Compute and log average losses
6. Save model weights each epoch

Checkpointing

- **Validation**:
  - `model.eval()` to set the model to evaluation mode
  - MSE loss on validation set
- **Checkpoint**:
  - Save the model weights to a `.pth` file
  - Update loss, PSNR and SSIM history in `history.txt`
- Monitor train vs validation loss over epochs aswell as PSNR and SSIM between the generated and original images
  - For each epoch sample 10 images from the validation set and compute the metrics

Epoch Validation

- **Metrics**:
  - PSNR: Peak Signal-to-Noise Ratio
  - SSIM: Structural Similarity Index
- **Sample Generation**:
  - Pure noise sampling using DDIM scheduler
  - Validation reconstruction:
    - Add noise to clean validation images
    - Model predicts and removes the noise
- **Quality Assessment**:
  - PSNR range: 20-40 dB (higher = better reconstruction)
  - SSIM range: 0-1 (closer to 1 = better similarity)
  - Average metrics computed over 5-10 validation samples

Monitoring Produced Samples

- **Pure Noise Sampling**:
  - Tests model's ability to generate realistic images
  - Uses DDIM scheduler for iterative denoising
  - Saves generated images as `generated_epoch_{epoch}.png`
- **Validation Reconstruction**:
  - Adds noise to clean validation images
  - Model predicts and removes the noise
  - Direct assessment of denoising performance
- **History Tracking**:
  - All metrics saved to `history.txt`
  - Enables trend analysis and model comparison

Plots

- **Loss Monitoring**:
  - Training vs Validation Loss curves over epochs
  - MSE loss
- **Quality Metrics Visualization**:
  - PSNR trends with average values
  - SSIM trends with average values
  - Both metrics computed on validation reconstructions
  - Useful to track model performance

- **Comprehensive Monitoring**:
  - Three-panel subplot: Loss, PSNR, SSIM (as shown in the Figure 2)
  - Data read from `history.txt` file
  - Enables performance trend analysis

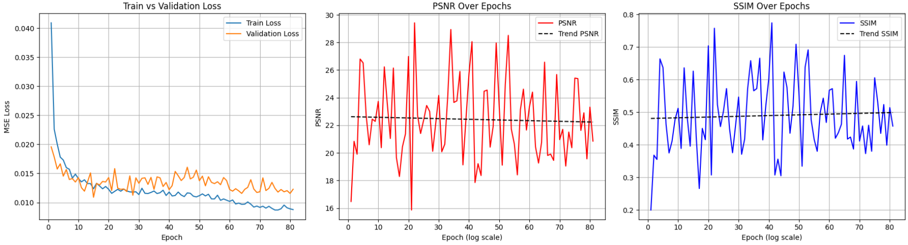

Figure: Training Loss, PSNR, and SSIM trends over epochs

Loading Checkpoints

- **Checkpoint Structure**:
  - Model state dictionary
  - Optimizer state dictionary
  - Current epoch number for resuming training
  - Naming: `ddim_unet_epoch81.pth`
  - `load_checkpoint()` utility function

DPS: Diffusion Posterior Sampling

- **DPS** is a method for solving inverse problems using pre-trained diffusion models
- Key idea: combine data fidelity with diffusion model prior during reverse sampling
- Modifies the standard DDIM reverse process to incorporate measurement consistency

Algorithm Overview:

1. Start with noisy initialization $x_T \sim \mathcal{N}(0, I)$
2. For each timestep t: predict x_0 using UNet
3. Apply posterior correction: $x_0^{post} = x_0^{pred} + \gamma_t \cdot K^T(y - Kx_0^{pred})$
4. Continue DDIM step with corrected x_0^{post}

Posterior Correction Weight:

$$\gamma_t = \frac{\sigma_{prior}^2}{\sigma_y^2 + \sigma_{prior}^2}$$

where $\sigma_{prior}^2 = 1 - \alpha_t$ and σ_y is measurement noise

Implementation of DPS

Algorithm Steps:

1. Initialize x_t with random noise
2. Predict noise $\epsilon_\theta(x_t, t)$ using UNet
3. Compute $x_0^{pred} = \frac{x_t - \sqrt{1-\alpha_t}\epsilon_\theta}{\sqrt{\alpha_t}}$
4. Apply DPS correction with gradient step
5. Update to next timestep using DDIM

RED-Diff: Regularization by Denoising Diffusion

- **RED-Diff** is an optimization-based method for solving inverse problems using diffusion models
- Key idea: combine data fidelity loss with regularization from denoising diffusion priors
- Uses gradient-based optimization to reconstruct images by minimizing combined objective

Algorithm Overview:

1. Initialize reconstruction μ from adjoint operation: $\mu = K^T(y)$
2. For each timestep t: sample noise and create noisy version x_t
3. Compute data fidelity loss: $\mathcal{L}_{obs} = \frac{1}{2\sigma_y^2}\|K(\mu) - y\|^2$
4. Compute regularization loss using diffusion model guidance
5. Update μ using gradient descent on combined loss

Combined Objective:

$$\mathcal{L} = \mathcal{L}_{obs} + \lambda \cdot w_t \cdot \mathcal{L}_{reg}$$

where w_t is a time-dependent weighting strategy

Algorithm Steps:

1. Initialize $\mu \leftarrow K^T(y)$ with gradient enabled
2. Sample noise $\epsilon \sim \mathcal{N}(0, I)$ and create $x_t = \sqrt{\alpha_t}\mu + \sqrt{1 - \alpha_t}\epsilon$
3. Predict noise $\epsilon_\theta(x_t, t)$ using UNet
4. Compute regularization loss: $\mathcal{L}_{reg} = w_t \cdot \|\epsilon_\theta - \epsilon\|^2$
5. Update μ using Adam optimizer on total loss
6. Repeat for all timesteps in reverse order

Weighting Strategies: linear, sqrt, square, log, clip, const

Sottosezione 2.1

Conclusions

Thank you for your attention