

# 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 19, 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.

- **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 on medical data
- **Phase 4:** Simulation of motion blur and its removal to test the quality of the model
- **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**

- We use the "Mayo Clinic CT Dataset" of low-dose CT scans.

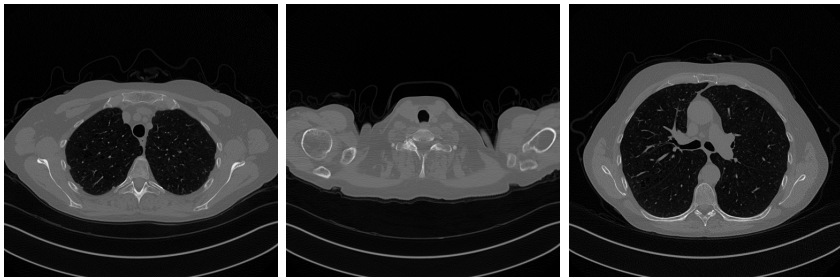


Figure 1: Examples of CT slices from the Mayo Clinic dataset

Before applying augmentations, each image is converted using:

- 1 **Grayscale**: single channel via `transforms.Grayscale(num_output_channels=1)`
- 2 **Resize**: to  $128 \times 128$  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]),
])
```

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()`

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

- **Model Type:** UNet2DModel from HuggingFace Diffusers library
- **Task:** Denoising diffusion probabilistic model for grayscale image generation
- **Input/Output:**
  - Input channels: 1
  - Output channels: 1
  - Sample size:  $128 \times 128$  pixels



- **Block Configuration:**

- Layers per block: 2
- Block output channels: (64, 128, 256)
- Dropout rate: 0.1

- **Downsampling Path:**

- DownBlock2D  $\rightarrow$  DownBlock2D  $\rightarrow$  AttnDownBlock2D
- Progressive feature extraction with attention in the deepest layer

- **Upsampling Path:**

- AttnUpBlock2D  $\rightarrow$  UpBlock2D  $\rightarrow$  UpBlock2D
- Symmetric architecture with attention mechanism

- **Training Scheduler:** DDPM Scheduler
  - Number of timesteps: 1000
  - Used for forward diffusion process during training
  - Adds noise progressively over 1000 steps
- **Inference Scheduler:** DDIM Scheduler
  - Number of timesteps: 1000
  - Deterministic sampling process
  - Used for image generation and inverse problems
  - Shares beta schedule with DDPM scheduler

- **Optimizer:** Adam
  - Learning rate:  $1 \times 10^{-4}$
  - Weight decay:  $1 \times 10^{-5}$
- **Loss Function:** Mean Squared Error *MSE*
  - Compares predicted noise with actual noise
  - Standard objective for diffusion models
- **Performance Optimizations:**
  - Model compilation with `torch.compile`
  - Mixed precision training with GradScaler
  - Cosine annealing learning rate scheduler

- **Total Parameters:** 15.722.625
- **Key Features:**
  - Attention mechanisms in deepest layers for better feature learning
  - Symmetric U-Net design for optimal information flow
  - Dropout regularization to prevent overfitting
  - Grayscale-optimized with single channel processing

- **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

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

```
model = torch.compile(model)
```
- **Benefits:** improved batch throughput

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

- ❶ Start the training `model.train()`
- ❷ 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()`
- ❸ Save samples to visualize the model performance during training
- ❹ Compute and log average losses
- ❺ Save model weights each epoch



- **Validation:**

- `model.eval()` to set the model to evaluation mode

- **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

- **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

- **Pure Noise Sampling:**

- Tests model's ability to generate images from noise that are similar to the training data
- Uses DDIM scheduler for iterative denoising
- Saves generated images as `epoch_{epoch}.png`

- **Validation Reconstruction:**

- Adds noise to clean validation images
- Model predicts and removes the noise
- Direct assessment of denoising performance
- Saves reconstructed images as `reconstructed_{epoch}.png`

- **History Tracking:**

- All metrics saved to `history.txt`
- Enables trend analysis and model comparison

- **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

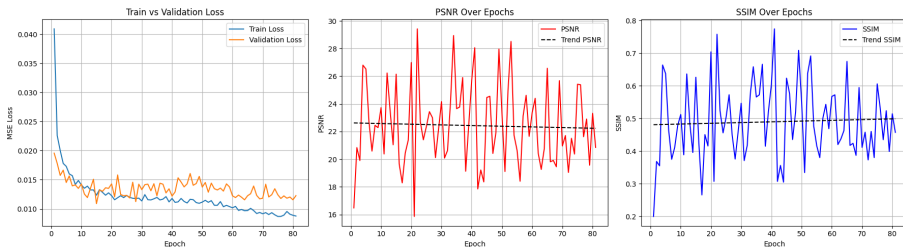


Figure 2: Training Loss, PSNR, and SSIM trends over epochs

# Generated Samples from Pure Noise

- Samples generated from pure noise using the trained model
- Visualized to assess the model's generative capabilities
- Useful for understanding the model's learned features
- Figure 3 shows 10 generated samples to assess the model's performance

# Generated Samples Visualization

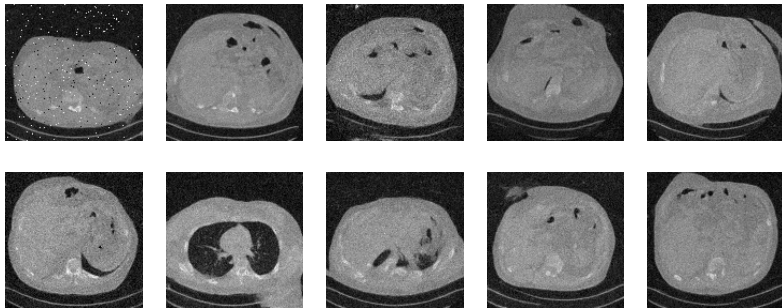


Figure 3: 10 generated samples from pure noise using the trained model at epoch 81.

- **Checkpoint Structure:**

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



- **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  $\sigma_y$  is measurement noise

## 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** 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  $\mu$  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  $\mu$  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  $\mu$  using Adam optimizer on total loss
- 6 Repeat for all timesteps in reverse order

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

- **Library:** IPPy
- **Degradation Type:** Motion Blur
- **Implementation:** Linear operator approach
- **Purpose:** Create realistic inverse problems for model evaluation

- **Operator:** `operators.Blurring`
- **Parameters:**
  - Image shape:  $(128 \times 128)$  pixels
  - Kernel type: "motion"
  - Motion angle:  $45^\circ$
  - Kernel sizes tested:  $[5, 7, 9, 11, 13, 15]$  pixels
- **Criteria:**
  - 5 images per kernel size for statistical reliability
- **Mathematical Model:**

$$y = K(x) + n \quad (1)$$

where  $K$  is the blur operator,  $x$  is the clean image, and  $n$  is noise



- **Dataset:** Validation set from the before-mentioned dataset
- **Degradation:** Motion blur with varying kernel sizes
- **Methods Compared:**
  - DPS (Diffusion Posterior Sampling)
  - RED-Diff (Regularization by Denoising)
- **Evaluation Metrics:**
  - PSNR (Peak Signal-to-Noise Ratio) in dB
  - SSIM (Structural Similarity Index)

- **For each test image:**

- 1 Load ground truth image  $x_{gt}$
- 2 Apply motion blur:  $y = K(x_{gt})$
- 3 Reconstruct using DPS:  $x_{dps} = \text{DPS}(y, K)$
- 4 Reconstruct using RED-Diff:  $x_{red} = \text{RED-Diff}(y, K)$
- 5 Compute metrics:  $\text{PSNR}(x_{gt}, x_{rec})$ , and  $\text{SSIM}(x_{gt}, x_{rec})$

Figure 4 shows the PSNR values for both methods across different kernel sizes.

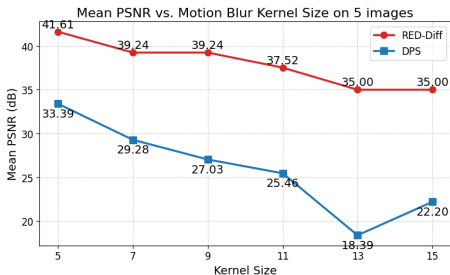


Figure 4: PSNR values for DPS and RED-Diff across different kernel sizes.

Figure 5 illustrates the SSIM values for both methods across different kernel sizes.

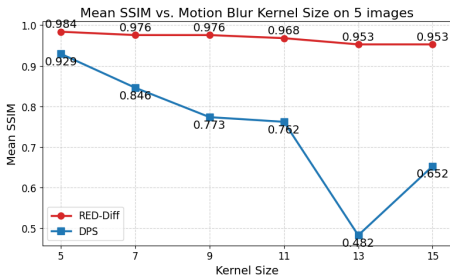


Figure 5: SSIM values for DPS and RED-Diff across different kernel sizes.

- **Trend Analysis:**

- Both methods show performance degradation with larger kernels, even though the RED-Diff method generally outperforms DPS.
- PSNR and SSIM correlate with blur severity

- **Qualitative Assessment:**

- Side-by-side comparisons: Original → Blurred → Reconstructed
- Visual quality correlation with quantitative metrics
- Edge preservation and artifact analysis

- **Key Findings:**

- RED-Diff better preserves fine details
- RED-Diff shows less artifacts compared to DPS
- RED-Diff keeps consistent performances across different kernel sizes
- DPS is more sensitive to kernel size variations, as shown in the PSNR and SSIM results

- **Visual Results:** To better understand the performance of both methods, we will show visual results for both DPS and RED-Diff a visual comparison had been conducted and reported on kernel size equal to 7 in the next slides.

# Visual Results Summary - DPS

Figure 6 presents visual comparisons of the original, blurred, and reconstructed images for both methods for kernel size 7 using DPS.

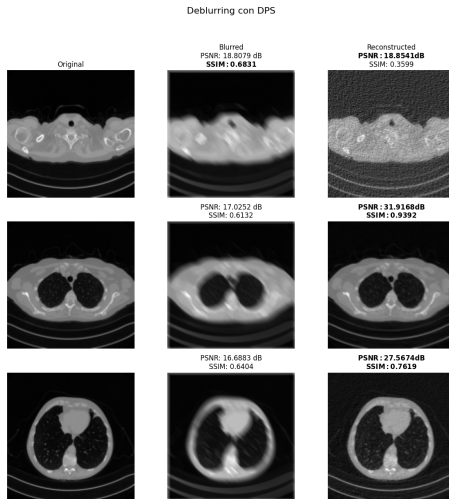


Figure 6: Visual results for DPS.

# Visual Results Summary - RED-Diff

Figure 7 presents visual comparisons of the original, blurred, and reconstructed images for both methods for kernel size 7 using RED-Diff.

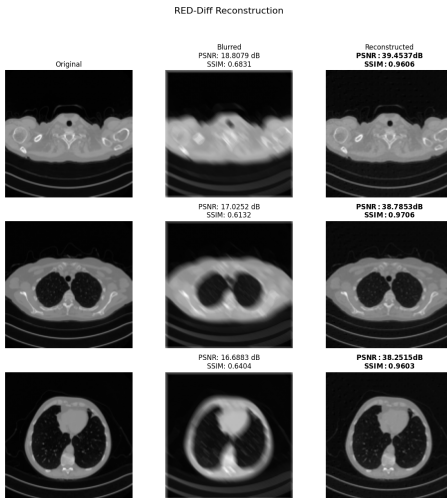


Figure 7: Visual results for RED-Diff.



- We explored training at a larger target size of  $256 \times 256$  pixels.
- Due to hardware and Colab limits, full  $256 \times 256$  training proved very slow.
- We expect comparable results at  $256 \times 256$  because:
  - Model architecture and training pipeline remain the same.
- If  $256 \times 256$  runs underperform, we can still approach  $128 \times 128$ -level results by:
  - Leveraging our robust data augmentation to enrich the larger-scale inputs.
- The implementation is flexible, so once faster hardware or longer runtimes are available, we can re-run full  $256 \times 256$  experiments with minimal changes.

Thank you for your attention