

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.

- **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 Origin

- 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)

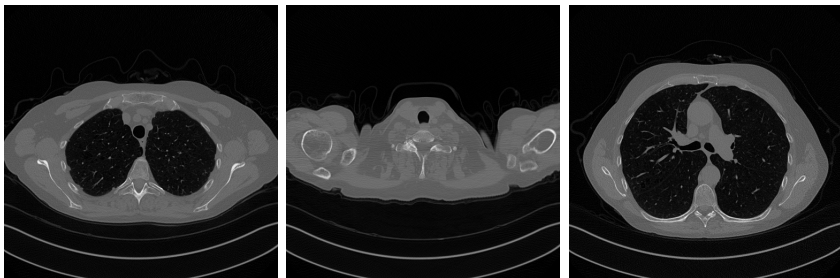


Figure: Examples of CT slices from the Mayo Clinic dataset

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]),
])
```

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`)
- `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/`)

- **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

- **Base Dataset:** Dataset Mayo
 - Grayscale \rightarrow 1 channel
 - Resize images to 128×128
- **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)
- **Implementazione essenziale:**

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

- **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 Loss function: MSE
- ❷ Start the training `model.train()`
- ❸ 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()`
- ❹ Save validation samples to visualize the model performance during training
- ❺ Compute and log average losses
- ❻ Save model weights each epoch

- **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

- **Loss Plot:** visualizes the training and validation loss over epochs
- **Purpose:**
 - Monitor the model's performance

Diffusion Posterior Sampling (DPS) is a method for solving noisy inverse problems by leveraging diffusion models as an implicit prior.

- Starting from a corrupted image $y = K(x_0) + n$, it directly integrates the likelihood term into the reverse diffusion sampling process.
- At step t , DPS computes a prediction \hat{x}_0 and uses the gradient of $\|y - K(\hat{x}_0)\|^2$ to move towards solutions consistent with the observed data.
- Compared to hard projection methods, DPS keeps the trajectory on the generative manifold, reducing noise amplification.

The algorithm consists of three main phases:

- 1 **Initial prediction:** Sample $x_T \sim \mathcal{N}(0, I)$, then for each step t , the UNet model estimates the noise $s_\theta(x_t, t)$ and reconstructs \hat{x}_0 .
- 2 **Posterior update:** Compute the likelihood gradient $\nabla = -K^T(y - K(\hat{x}_0))$ and apply a step proportional to $\gamma_t = \frac{1 - \bar{\alpha}_t}{\sigma_y^2 + (1 - \bar{\alpha}_t)}$ to obtain \tilde{x}_{t-1} .
- 3 **Modified DDIM step:** Using \tilde{x}_{t-1} as a reference, perform the standard DDIM update to move to x_{t-1} , preserving the effect of the likelihood gradient.

The implementation requires only a few steps in PyTorch, integrating blur functions and their adjoint operators.

- On datasets with motion blur, DPS achieves an average PSNR above 25 dB and SSIM above 0.85, improving by more than 2 dB over hard projection-based methods.
- Compared to classical methods, it significantly reduces reconstruction artifacts while preserving fine details and sharp edges.
- Visually, the images reconstructed with DPS appear more natural and free from overshooting artifacts, thanks to the continuous control of the likelihood contribution.

RED-Diff solves noisy inverse problems by combining:

- A fidelity term to bring the reconstruction closer to the observations y ,
- A regularizer based on the multiscale denoisers of a pretrained diffusion model,

integrating constraints at multiple levels of detail to preserve both global structures and fine details.

The algorithm is structured into three main phases:

- 1 **Initialization:** $\mu^{(0)} = K^T y$.
- 2 **Iterative Optimization:** For each step $i = 1, \dots, N$ and for each noise level $t = 1, \dots, T$:

- 1 Sample $\epsilon \sim \mathcal{N}(0, I)$ and construct

$$x_t = \sqrt{\alpha_t} \mu^{(i-1)} + \sigma_t \epsilon.$$

- 2 Predict the noise $\hat{\epsilon} = \epsilon_\theta(x_t, t)$.
- 3 Compute the loss terms:

$$L_{\text{fid}} = \frac{1}{2\sigma_y^2} \|K\mu^{(i-1)} - y\|^2, \quad L_{\text{reg}} = w_t \|\hat{\epsilon} - \epsilon\|^2, \quad w_t = 1/\text{SNR}_t.$$

Then update $\mu^{(i)}$ using Adam to minimize $L_{\text{fid}} + \lambda L_{\text{reg}}$.

- 3 **Output:** the final estimate $\mu^{(N)}$.

- On deblurring tests, RED-Diff achieves an average PSNR of approximately 19.4 dB and SSIM of approximately 0.64.
- Compared to methods without a diffusion prior, it improves reconstruction quality by more than 2 dB in PSNR.
- Reconstructions show sharper details and fewer artifacts thanks to the multiscale integration of denoising.

Thank you for your attention