Practical Report

Pose Estimation + Novel View Synthesis

Ludwig-Maximilians-Universität München

Jimmy Tan & Amin Dziri

Munich, July 24th, 2025

Supervised by Ulrich Prestel

Contents

Chapter 1

Introduction

1.1 Scalable Interpolant Transformers (SiT)

The architecture we utilized in this practical is SiT [MGA+24]. It extends the Diffusion Transformer architecture by leveraging an interpolant framework.

Traditionally, diffusion models add Gaussian noise step by step to data x_0 over time t to form a noisy latent x_t in the forward process. The reverse process learns to denoise x_t back to x_0. However, the forward and reverse process are stochastic.

SiT, in contrast, adopts a deterministic interpolant path from Flow Matching [LHH+24]. The general idea is that we have a noise distribution p and target distribution q, and we want to map those two distributions from one to the other. According to this framework, we first design a path before training to define how to interpolate between two distributions over time t. After that, during training, we use the trained model to predict the velocity field known to generate the path p_t. After training, to generate the novel target sample x_1, we integrate the estimated velocity field from $t = 0$ to $t = 1$, where x_0 is a novel noise sample.

Let's look at the general formula of the interpolation part because it is important for the understanding of our implementation later. The formula is:

$$x(t) = \alpha(t)x_1 + \sigma(t)\epsilon \tag{1.1}$$

where $\alpha(1) = \sigma(0) = 1$ and $\alpha(0) = \sigma(1) = 0$. The idea is very simple. We have the endpoints x_0 and x_1 where x_0 is the noise endpoint and x_1 is the target endpoint. When $t = 0$, we only have the noise and when $t = 1$ we have only

the pure data latent. This formula just defines the interpolation as the linear combination of the noise and data latent over time t. Because of this, with fixed endpoints, x_t is completely deterministic, which is a very useful property for our pose-estimation practical!

Chapter 2

Training Overview

2.1 Dataset

First, we have a dataset with 30k data points generated by NeRF [MST+20] which has some interesting properties. The data is generated with an object placed at the origin, which in our case is the chair, then the camera is randomly placed at the shell of a sphere with radius 3-5 which is evenly distributed. For each data points, it contains the camera pose, which is a 4×4 matrix, the image of the object view, as well as the focal length, which is not important for our case.

2.2 Convert Data to Latent

With the dataset, we then convert the images into latents with a pre-trained variational autoencoders with the following code:

```
x_latent = vae.encode(x).latent_dist.sample().mul_
   (0.18215)
```

Each point in the latent space is our data latent endpoint x_1. The rescaling factor in the end is just to make sure that the latent variance is around 1. Refer to this paper [RBL+22] if you want to look into this rescaling factor in more detail.

2.3 Sampling and Loss Calculation

During training, we randomly sample the 0 mean gaussian noises $\epsilon \sim \mathcal{N}(0, 1)$ and shift the mean to the poses P by simply adding them together to create a noise endpoint x_0.

$$x_0 = P + \epsilon \tag{2.1}$$

Additionally, we also sample the time t.

Then, we determine the ground truth velocity from the $(t, x0, x1)$ triple. We can do that because as mentioned previously the path is already pre designed before training. With the trained model in every step, we then predict the velocity using the trained model by using only the sampled t and the corresponding x_t.

The loss can then be simply calculated as the MSE loss of the ground truth and predicted velocity.

Chapter 3

Implementation

3.1 Class Conditioning

The original code of SiT is intended for the ImageNet dataset which has 1000 classes. They implemented class conditioning during training by randomly picking a value from 0 - 999, with each integer representing each class.

```
# Labels to condition the model with (feel free to change
    ):
ys = torch.randint(1000, size=(local_batch_size,), device
    =device)
```

We have changed it in our code to take the constant value of 0 because we are only dealing with one class.

```
# Labels to condition the model with (feel free to change
    ):
ys = torch.zeros(local_batch_size, dtype=torch.long,
    device=device)
```

3.2 Introduce Noisy Pose

In the following code snippet, you can see how we introduced the noisy pose as our noise endpoint:

```
def sample(self, x1, noise=None):
    """Add commentMore actions
    Sampling x0 & t based on shape of x1 (if needed).
    Allows optional external noise input.

    Args:
        x1 - data point; [batch, *dim]
        noise - custom noise tensor (optional)
    """
    x0 = noise if noise is not None else th.randn_like(x1
       )
    t0, t1 = self.check_interval(self.train_eps, self.
       sample_eps)
    t = th.rand((x1.shape[0],)) * (t1 - t0) + t0
    t = t.to(x1)
    return t, x0, x1
```

The variable "noise" comes from our dataset class with poses already added to the 0 mean gaussian noise as shown in Equation 2.1.

Apart from that, we have also changed the loss calculation accordingly to use the noisy pose, if provided, as the noise endpoint for the loss calculation.

```
def training_losses(
    self,
    model,
    x1,
    model_kwargs=None
):
    """
    Loss for training the score model.

    Args:
        model: backbone model; could be score, noise, or
           velocity
        x1: datapoint
        model_kwargs: additional arguments for the model
    """
    if model_kwargs is None:
        model_kwargs = {}
```

```
    # If provided, use the custom noise instead of
       standard Gaussian
    noise = model_kwargs.get('noise', None)
    t, x0, x1 = self.sample(x1, noise=noise)
    t, xt, ut = self.path_sampler.plan(t, x0, x1)
```

3.3 Use of bfloat16 (bf16) Precision for Training Optimization

To enhance training efficiency and resource utilization, this project leverages bfloat16 (bf16) precision via PyTorch's automatic mixed precision (AMP) system. The relevant implementation appears in the training loop:

```
with autocast("cuda", dtype=torch.bfloat16):
    loss_dict = transport.training_losses(model, x_latent
       , model_kwargs)
    loss = loss_dict["loss"].mean()
```

This autocast context ensures that operations within its scope are executed in bfloat16 precision where safe, while preserving full precision for operations sensitive to numerical stability (such as loss calculations and certain normalization layers).

3.4 Poses as Starting Distribution

The flow matching technique allows us to choose our starting distribution individually. Instead of going from pure noise to the image distribution, we incorporate the poses into the noise, which lets us transition from the pose distribution to the image distribution. However, based on our training image size of $128 \times 128 \times 3$, the model expects a sample drawn from the starting distribution to have the shape $16 \times 16 \times 4$, while our poses have the shape 4×4. Therefore, they need to be further preprocessed to match the required shape.

Before reshaping the poses, the values of the rotation matrix and the translation vector are standardized. Let $\mathbf{P} \in \mathbb{R}^{4\times4}$ be a randomly selected pose matrix. Then, the upper-left 3×3 block of $\mathbf{P}$, denoted $\mathbf{P}_{:3,\ :3}$, is the rotation

matrix, and the upper-right 3×1 vector, denoted $\mathbf{P}_{:3,\ 3}$, is the translation vector. The mean and standard deviation are computed over both the rotation matrix $\mathbf{P}_{:3,\ :3}$ and the translation vector $\mathbf{P}_{:3,\ 3}$ over the entire training dataset, resulting in $\mu_{\mathbf{R}}$ and $\mu_{\mathbf{t}}$, which denote the mean of the rotation matrix and the translation vector respectively, and $\sigma_{\mathbf{R}}$ and $\sigma_{\mathbf{t}}$, which denote the standard deviation of the rotation matrix and translation vector respectively. After calculating the means and standard deviations, the pose values are standardized by subtracting the mean and dividing by the standard deviation as seen in equations 3.1.

$$\mathbf{R}_{\text{std}} = \frac{\mathbf{R} - \mu_{\mathbf{R}}}{\sigma_{\mathbf{R}}} \quad \text{and} \quad \mathbf{t}_{\text{std}} = \frac{\mathbf{t} - \mu_{\mathbf{t}}}{\sigma_{\mathbf{t}}} \tag{3.1}$$

After standardizing the rotation matrix and translation vector respectively, the shape is adjusted to match the required $4 \times 16 \times 16$ shape. Each row is transformed into a 16×16 matrix, where each element of the 1×4 row completely fills one quarter of the expanded matrix. This is repeated for each row, resulting in four 16×16 matrices which are then concatenated along dimension 0. An example process of how one row is expanded into a 16×16 matrix is visualized in Figure 3.1.

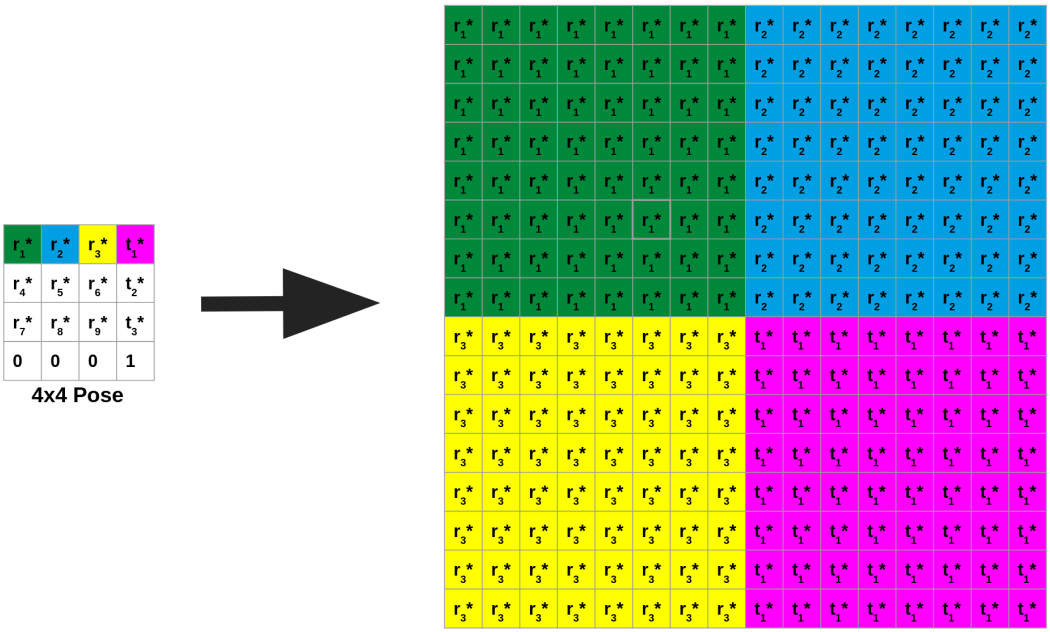

Figure 3.1: Expansion of one 1×4 row into a 16×16 matrix. This is repeated for each of the four rows, resulting in a $4 \times 16 \times 16$ matrix.

After successfully expanding the poses, noise in the shape of $4 \times 16 \times 16$

is sampled from a Gaussian distribution with mean 0 and variance 1. The $4 \times 16 \times 16$ pose matrix is then added to said Gaussian noise.

3.5 Flow Reversal

What was also implemented is the reversal of the SiT to estimate the pose given a scene. To reverse the learned flow, we apply the Euler backward method, as shown in Equation 3.2.

$$x_{t-h} = x_t - h \cdot v_\theta(x_t, t) \tag{3.2}$$

This update subtracts the current velocity field $v_\theta(x_t, t)$ from the interpolant x_t, at each step that is taken.

This method is implemented in the function `sample_backwards`, which iterates over the number of steps and applies the Euler backward method at each step.

```
def sample_backwards(self, x, model, **model_kwargs):
    device = x.device if isinstance(x, th.Tensor) else x
       [0].device

    def flow(t, x):
        t_batch = th.full((x.size(0),), t, device=device)
        ones = th.ones(x.size(0), device=device)
        return self.drift(x, ones - t_batch, model, **
           model_kwargs)

    t = self.t.to(device)
    x_out = x

    for i in range(len(t) - 1):
        t_start = t[i]
        t_end = t[i + 1]
        h = (t_end - t_start).item()
        x_out = x_out - h * flow(t_end.item(), x_out)

    return x_out
```

If we want to estimate the pose given a scene during inference, we first need to specify the backwards sampler that utilizes the above-specified function `sample_backwards`.

```
sample_fn = sampler.sample_ode_backwards(
    sampling_method=ODE_sampling_method,
```

```
    atol=atol,
    rtol=rtol,
    num_steps=num_sampling_steps
)
```

Now that we aim to estimate poses, the input to the model becomes the scene. Initially, the scene image has the shape $3 \times 128 \times 128$ and thus needs to be transformed to $4 \times 16 \times 16$ to fit into the SiT. For this, we utilize the variational autoencoder to encode the image into a latent representation with the required shape.

```
posterior = vae.encode(image)[0]
image = posterior.sample()
image = image * 0.18215
```

Using this encoded image, we can provide it as input to the SiT, which then predicts an estimated pose by applying the reverse sampling process.

Chapter 4

Experiments

4.1 Experiment Setup

To measure the performance of our trained SiT, we evaluate it on both the initial novel view synthesis task and the additional pose estimation task. For novel view synthesis, we employ several metrics that compare the predicted scene to the ground truth: the Peak Signal-to-Noise Ratio (PSNR), the Mean Absolute Error (MAE), and the Learned Perceptual Image Patch Similarity (LPIPS). For pose estimation, we utilize two metrics: the translation error (e_T), defined as the Euclidean distance between the ground truth and predicted translation vectors, and the rotation error (e_R), which measures the angular difference between the ground truth and predicted rotation matrices.

As mentioned in Chapter 2.1, the SiT was trained on data evenly sampled with a radius between 3 and 5. We evaluate its performance not only on new data within this radius range but also on data sampled from a broader radius range of 1–7. This allows us to see how the model performs outside of its training distribution.

We first evaluate the model using the specified evaluation metrics and then conduct additional qualitative tests. First, we randomly select two pose–image pairs and interpolate the pose as shown in Equation 4.1.

$$\text{pose}(t) = t \cdot \text{pose}_a + (1 - t) \cdot \text{pose}_b \tag{4.1}$$

We then generate the scene at each interpolation step. Furthermore, we evaluate the pose estimation on randomly sampled data within a ball around the chair object. To make the results more interpretable, we also estimate poses on turntable data and analyze whether the model can reproduce the circular

motion pattern. Finally, as a last experiment, we first predict a scene given a noisy pose. In a second step, we use this predicted scene to estimate the pose. Lastly, using this estimated pose, we generate the scene again, resulting in a scene generation loop.

4.2 Quantitative Results

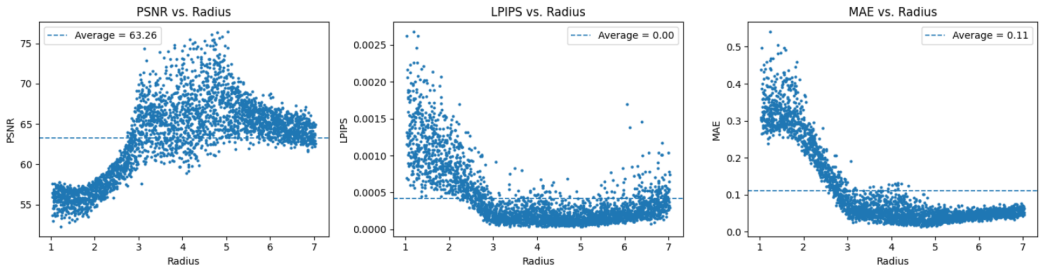

Figure 4.1: Evaluation of the novel view generation on the test dataset.

radius range	e_R	e_T
$r \in [1, 7]$	58.02	5.35
$r \in [3, 5]$	16.85	0.78

Figure 4.2: Evaluation of the pose estimation on the test dataset, for two different radii-ranges.

4.3 Qualitative Results

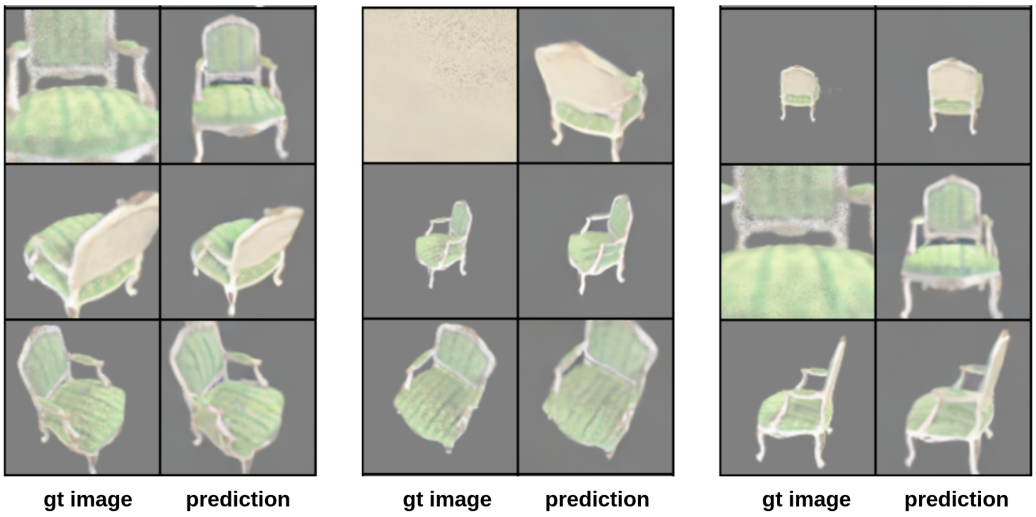

Figure 4.3: Examples of ground truth scenes and predicted scenes given the pose.

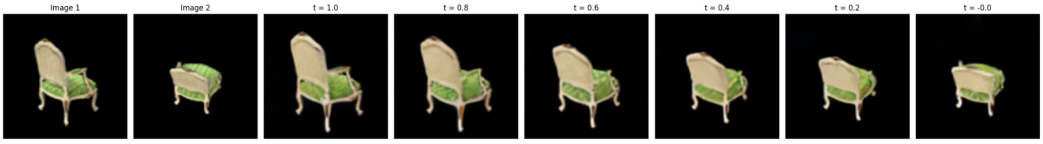

Figure 4.4: Example of the generated scenes given the interpolated poses.

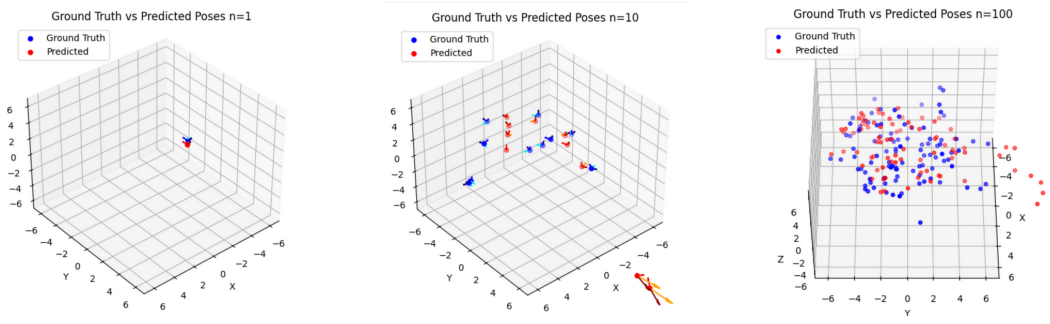

Figure 4.5: Estimated poses for n=1, n=10 and n=100.

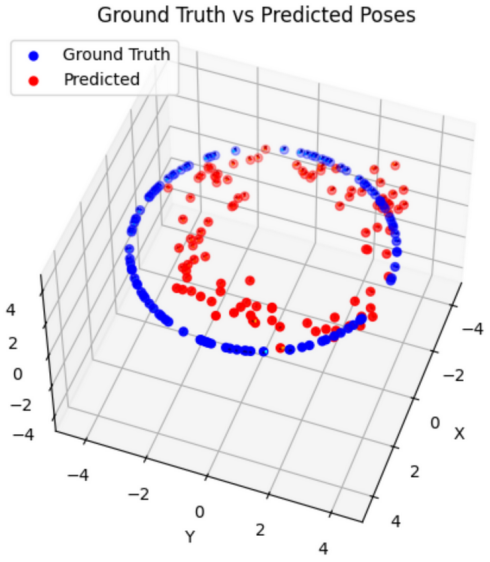

Figure 4.6: Estimated poses of turntable data.

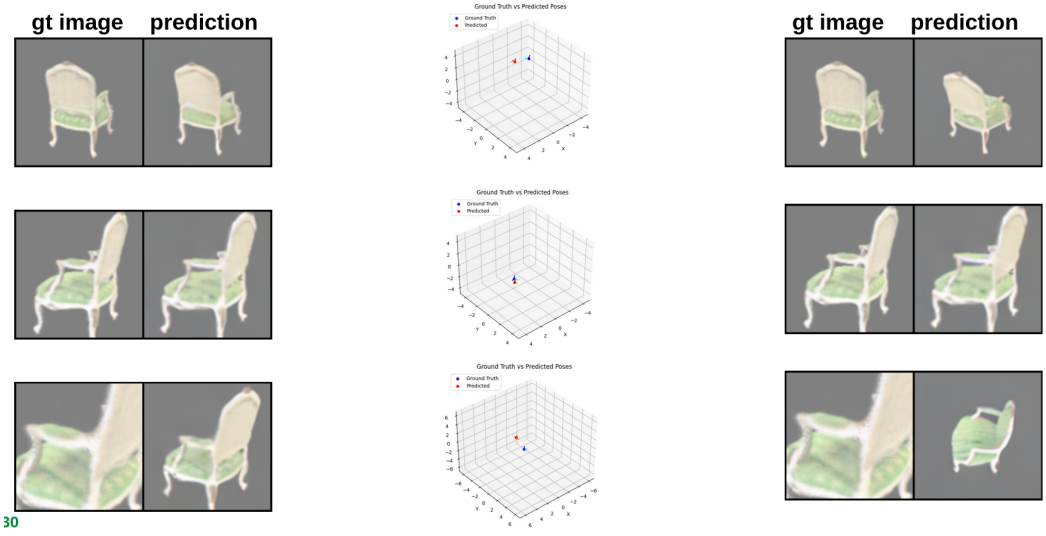

Figure 4.7: Examples of scene synthesis loop.

4.4 Discussion of Results

The model performs well in generating novel views and estimating the poses. The performance in both tasks is higher for samples generated within the range

of 3–5 and decreases outside this range. However, this is to be expected, as the model was trained exclusively on data within this range. Nevertheless, the exceptionally high results on the novel view synthesis task raise some concerns, as an average PSNR of 63.36 and an LPIPS of 0.0004 seem unusually good. One possible explanation is that the model was trained on a large dataset of 30k samples collected within a small radius, meaning that new test data is very similar to the training data and thus leads to a high performance, similar to overfitting. Due to time constraints, we weren't able to further investigate that concern. Additionally, for samples with a very small radius, the model's predictions are significantly inaccurate, as visible in Figure 4.3. This could also explain the presence of outliers in the pose estimation results shown in Figure 4.5.

Chapter 5

Failure Cases

5.1 Artifacts in Generated Image

During the implementation phase, we encountered several issues related to the incorporation of noisy pose information into our training loop. Initially, introducing Gaussian noise ($\sigma = 1.0$) to the pose resulted in predictions that appeared to blend characteristics of images corresponding to nearby poses (refer to Figure 5.1). This led us to hypothesize that the pose latents were overlapping in the latent space.

To mitigate this, we reduced the standard deviation of the noise from 1.0 to 0.2. However, the resulting predictions remained qualitatively similar, suggesting that the overlapping issue persisted (refer to Figure 5.2).

To further investigate, we removed noise entirely in order to test whether a

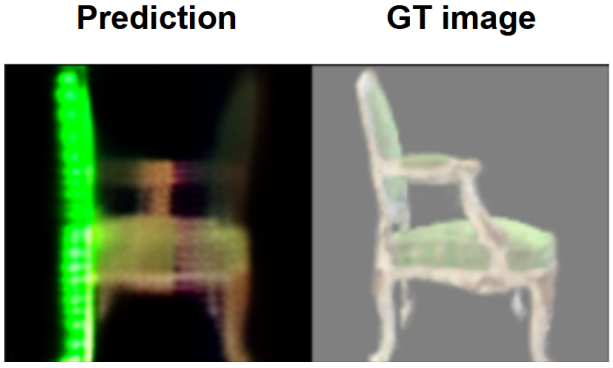

Figure 5.1: Failure Case 1

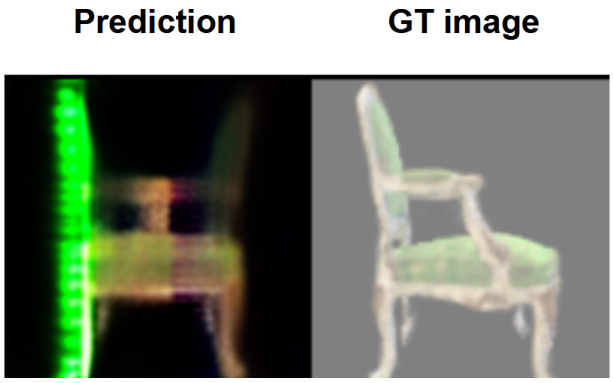

Figure 5.2: Failure Case 2

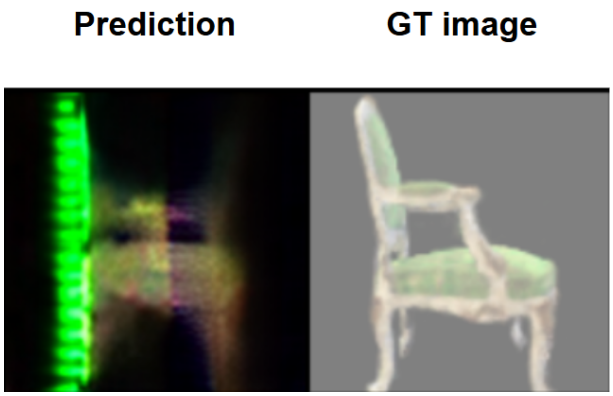

Figure 5.3: Failure Case 3

clean one-to-one mapping between pose and image could be learned. Surprisingly, the output was still similar, albeit noticeably more blurry compared to results with noise (refer to Figure 5.3).

We hypothesized that small variations in pose might be leading to disproportionately large changes in image space, preventing the model from learning well-separated latents. To address this, we magnified the pose signal by a factor of 10, intending to push pose representations further apart in latent space. Unfortunately, this produced the poorest results observed so far (refer to Figure 5.4).

Eventually, we identified the root cause: the random horizontal flip applied during image transformations.

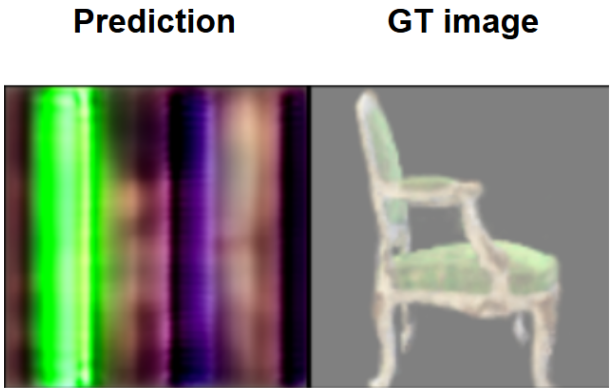

Figure 5.4: Failure Case 4

```
# Setup data:
transform = transforms.Compose([
    transforms.Resize(args.image_size),
    transforms.Lambda(lambda pil_image: center_crop_arr(
       pil_image, args.image_size)),
    # transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    # transforms.Normalize(mean=[0.5, 0.5, 0.5], std
       =[0.5, 0.5, 0.5], inplace=True)
    transforms.Normalize(mean=[0.0, 0.0, 0.0], std=[1.0,
       1.0, 1.0], inplace=True)
])
```

This augmentation inadvertently introduced inconsistency between the pose input and image output, leading to artifacts and degraded performance. After disabling horizontal flipping and reverting to consistent data normalization (zero mean and unit variance), the model's behavior aligned more closely with expectations.

5.2 Pose Estimation Error During Flow Reversal

In addition to the previously discussed issue, we also encountered significant errors during the flow reversal process. Specifically, the estimated pose showed a substantial deviation from the ground truth. Upon investigation, we identi-

fied two primary causes:

- The mean and standard deviation used for pose de-standardization were taken from the test set instead of the training set. This mismatch led to incorrect scaling of the output poses.
- The translation vector and rotation matrix were standardized and de-standardized together, even though they are semantically distinct and should be handled separately.

To address these problems, we implemented the following solutions:

- We ensured that pose destandardization uses the mean and standard deviation computed from the training set, aligning with the statistics used during training.
- We separated the standardization and destandardization procedures for the rotation matrix and translation vector to better capture their individual characteristics.

These corrections successfully resolved the pose estimation discrepancy.

Bibliography

[LHH+24] Yaron Lipman, Marton Havasi, Peter Holderrieth, Neta Shaul, Matt Le, Brian Karrer, Ricky T. Q. Chen, David Lopez-Paz, Heli Ben-Hamu, and Itai Gat. *Flow Matching Guide and Code*, 2024.

[MGA+24] Nanye Ma, Mark Goldstein, Michael S. Albergo, Nicholas M. Boffi, Eric Vanden-Eijnden, and Saining Xie. *SiT: Exploring Flow and Diffusion-based Generative Models with Scalable Interpolant Transformers*, 2024.

[MST+20] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. *NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis*, 2020.

[RBL+22] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. *High-Resolution Image Synthesis with Latent Diffusion Models*, 2022.