Novel View Synthesis + Pose Estimation

Presenter: Jimmy Tan & Amin Dziri

SiT - Introduction

- ➡️ Scalable Interpolant Transformers
  - 👉🏻 Extends the Diffusion Transformer (DiT)
- ➡️ Leverages an interpolant framework

SiT - Background

Traditionally, diffusion models:

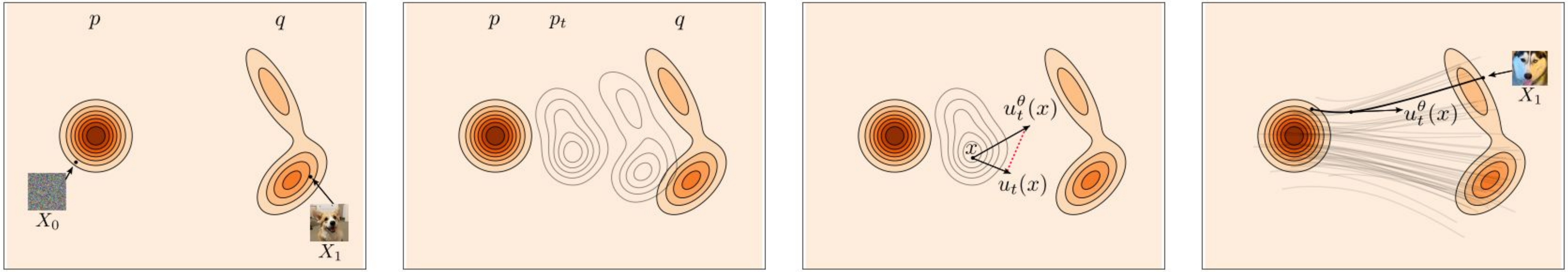
▶️Add gaussian noise step by step to data x_0 over time $t \in [0,T]$ to form a noisy latent x_t in the forward process

▶️The reverse process learns to denoise x_t back to x_0

▶️The forward and reverse processes are stochastic

SiT - Background

What's different about SiT?: ▶️Adopts Flow Matching Framework



(a) Data. (b) Path design. (c) Training. (d) Sampling.

Flow Matching Guide and Code: https://arxiv.org/pdf/2412.06264

SiT - Background

Interpolation:

Time-dependant parameters that define how to interpolate between x_0 and x over t ∈ [0, 1]

$$x(t) = \alpha(t)x_1 + \sigma(t)\epsilon$$

- Endpoints (x_0 = noise, x_1 = original data latent)
- x(t) = Intermediate point at time t along the velocity field

!!NOTE:

- $\alpha(1) = \sigma(0) = 1$
- $\alpha(0) = \sigma(1) = 0$

SiT - Background

Interpolation:

Time-dependant parameters that define how to interpolate between x_0 and x over t ∈ [0, 1]

$$x(t) = \alpha(t)x_1 + \sigma(t)\epsilon$$

- Endpoints (x_0 = noise, x_1 = original data latent)
- x(t) = Intermediate point at time t along the velocity field

!!NOTE:

- $\alpha(1) = \sigma(0) = 1$
- $\alpha(0) = \sigma(1) = 0$

With fixed endpoints, x(t) is completely deterministic. ➡️ Very useful for our pose-estimation practical!

SiT - Training Overview

1. Dataset

➡️Generated using NeRF, size 30k

➡️Properties:

- Object centered at origin, camera placed at the shell of a sphere with radius 3-5 (evenly distributed)
- Each data point: (**camera_pose**, **object_view**, focal_length)

SiT - Training Overview

2. Data → Latent

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

Rescales so latent variance ≈ 1

Rombach, R., Blattmann, A., Lorenz, D., Esser, P., & Ommer, B. (2021). High-Resolution Image Synthesis with Latent Diffusion Models (arXiv:2112.10752). arXiv.

SiT - Training Overview

3. Sampling and loss calculation

➡️Sample gaussian, add to poses -> Noise endpoint x_0

➡️Sample t

➡️Determine the ground truth velocity from (t, x_0, x_1)

➡️Predict velocity using the trained model using only (xt, t)

➡️Compute the MSE loss

LUDWIG-
MAXIMILIANS-
UNIVERSITÄT
MÜNCHEN

SiT - Implementation: Class Conditioning

Original Code:

- Intended for ImageNet dataset which contains 1000 classes. Class conditioning by choosing random integer from 0-999

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

Our Code:

- Changed to constant conditioning because we are only dealing with 1 class:

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

SiT - Implementation: Introduce Noisy Pose

Instead of 0 mean gaussian noise, we shift the mean to the poses :

- Parameter ‘noise’ = reshaped poses + gaussian noise N(0,1)

```
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
```

⬅️Set noise endpoint to the noisy pose

SiT - Implementation: Introduce Noisy Pose

Instead of 0 mean gaussian noise, we shift the mean to the poses :

- Parameter ‘noise’ = reshaped poses + gaussian noise N(0,1)

```
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)
```

⬅️Set noise endpoint to the noisy pose according for noise calculation as well

Adding the poses

Previously:

- x_0 ~ $N(0, 1)$
- $x_1 \sim p_{data}$
- x_0 sample of shape 4 x 32 x 32 or (4 x image_length/8 x image_length/8)

Adding the poses

Pose

r_1	r_2	r_3	t_1
r_4	r_5	r_6	t_2
r_7	r_8	r_9	t_3
0	0	0	1

Adding the poses:

Pose

r_1	r_2	r_3	t_1
r_4	r_5	r_6	t_2
r_7	r_8	r_9	t_3
0	0	0	1

μ_{rot}

σ_{rot}

$$\mathrm{pose}_{ij} = \frac{\mathrm{pose}_{ij} - \mu_{\mathrm{rot}}}{\sigma_{\mathrm{rot}}} \quad \text{for } i, j \in \{0, 1, 2\}$$

μ_{tran}

σ_{tran}

$$\mathrm{pose}_{i3} = \frac{\mathrm{pose}_{i3} - \mu_{\mathrm{tran}}}{\sigma_{\mathrm{tran}}} \quad \text{for } i \in \{0, 1, 2\}$$

Adding the poses

Pose

r_1^*	r_2^*	r_3^*	t_1^*
r_4^*	r_5^*	r_6^*	t_2^*
r_7^*	r_8^*	r_9^*	t_3^*
0	0	0	1

4x4

One row will be one channel!

r_1^*	r_1^*	r_1^*	r_1^*	r_1^*	r_1^*	r_1^*	r_1^*	r_2^*	r_2^*	r_2^*	r_2^*	r_2^*	r_2^*	r_2^*	r_2^*
r_1^*	r_1^*	r_1^*	r_1^*	r_1^*	r_1^*	r_1^*	r_1^*	r_2^*	r_2^*	r_2^*	r_2^*	r_2^*	r_2^*	r_2^*	r_2^*
r_1^*	r_1^*	r_1^*	r_1^*	r_1^*	r_1^*	r_1^*	r_1^*	r_2^*	r_2^*	r_2^*	r_2^*	r_2^*	r_2^*	r_2^*	r_2^*
r_1^*	r_1^*	r_1^*	r_1^*	r_1^*	r_1^*	r_1^*	r_1^*	r_2^*	r_2^*	r_2^*	r_2^*	r_2^*	r_2^*	r_2^*	r_2^*
r_1^*	r_1^*	r_1^*	r_1^*	r_1^*	r_1^*	r_1^*	r_1^*	r_2^*	r_2^*	r_2^*	r_2^*	r_2^*	r_2^*	r_2^*	r_2^*
r_1^*	r_1^*	r_1^*	r_1^*	r_1^*	r_1^*	r_1^*	r_1^*	r_2^*	r_2^*	r_2^*	r_2^*	r_2^*	r_2^*	r_2^*	r_2^*
r_1^*	r_1^*	r_1^*	r_1^*	r_1^*	r_1^*	r_1^*	r_1^*	r_2^*	r_2^*	r_2^*	r_2^*	r_2^*	r_2^*	r_2^*	r_2^*
r_1^*	r_1^*	r_1^*	r_1^*	r_1^*	r_1^*	r_1^*	r_1^*	r_2^*	r_2^*	r_2^*	r_2^*	r_2^*	r_2^*	r_2^*	r_2^*
r_3^*	r_3^*	r_3^*	r_3^*	r_3^*	r_3^*	r_3^*	r_3^*	t_1^*	t_1^*	t_1^*	t_1^*	t_1^*	t_1^*	t_1^*	t_1^*
r_3^*	r_3^*	r_3^*	r_3^*	r_3^*	r_3^*	r_3^*	r_3^*	t_1^*	t_1^*	t_1^*	t_1^*	t_1^*	t_1^*	t_1^*	t_1^*
r_3^*	r_3^*	r_3^*	r_3^*	r_3^*	r_3^*	r_3^*	r_3^*	t_1^*	t_1^*	t_1^*	t_1^*	t_1^*	t_1^*	t_1^*	t_1^*
r_3^*	r_3^*	r_3^*	r_3^*	r_3^*	r_3^*	r_3^*	r_3^*	t_1^*	t_1^*	t_1^*	t_1^*	t_1^*	t_1^*	t_1^*	t_1^*
r_3^*	r_3^*	r_3^*	r_3^*	r_3^*	r_3^*	r_3^*	r_3^*	t_1^*	t_1^*	t_1^*	t_1^*	t_1^*	t_1^*	t_1^*	t_1^*
r_3^*	r_3^*	r_3^*	r_3^*	r_3^*	r_3^*	r_3^*	r_3^*	t_1^*	t_1^*	t_1^*	t_1^*	t_1^*	t_1^*	t_1^*	t_1^*
r_3^*	r_3^*	r_3^*	r_3^*	r_3^*	r_3^*	r_3^*	r_3^*	t_1^*	t_1^*	t_1^*	t_1^*	t_1^*	t_1^*	t_1^*	t_1^*
r_3^*	r_3^*	r_3^*	r_3^*	r_3^*	r_3^*	r_3^*	r_3^*	t_1^*	t_1^*	t_1^*	t_1^*	t_1^*	t_1^*	t_1^*	t_1^*

Adding the poses

Then:

- Using each row as a channel leads to a 4x16x16 matrix
- We sample the noise from a standard gaussian noise ~ $N(\mu=0, \sigma^2=1)$
- We simply add the pose to the noise (pose + noise)

Flow reversal

For our forward flow we have our interpolant: $x_t = tx_1 + (1-t)x_0$

We walk from t= 0 (noise + pose) to t= 1 (image)

Using Euler we can calculate: $x_{t+h} = x_t + h \cdot v_\theta(x_t, t)$

We can now simply take the reverse of euler: $x_{t-h} = x_t - h \cdot v_\theta(x_t, t)$

Flow reversal

```python
def sample(self, x, model, **model_kwargs):

    device = x[0].device if isinstance(x, tuple) else x.device

    def flow(t, x):
        #This reshapes our t into a batch of t's
        t = th.ones(x[0].size(0)).to(device) * t if isinstance(x, tuple) else th.ones(x.size(0)).to(device) * t
        model_output = self.drift(x, t, model, **model_kwargs)
        return model_output

    t = self.t.to(device)
    atol = [self.atol] * len(x) if isinstance(x, tuple) else [self.atol]
    rtol = [self.rtol] * len(x) if isinstance(x, tuple) else [self.rtol]

    samples = odeint(
        flow,
        x,
        t,
        method=self.sampler_type,
        atol=atol,
        rtol=rtol
    )
    return samples
```

Flow reversal

```
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
```

Flow reversal: Inference time

First, specify new backward sampling function

```
sample_fn = sampler.sample_ode_backwards(
    sampling_method=ODE_sampling_method,
    atol=atol,
    rtol=rtol,
    num_steps=num_sampling_steps,
    reverse=False
)
```

Get latent representation of image

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

This latent representation is given as our x_0

Evaluation: Metrics

$$\mathrm{MAE}(I_{\mathrm{gt}}, I_{\mathrm{pred}})$$

$$e_R$$

$$\mathrm{PSNR}(I_{\mathrm{gt}}, I_{\mathrm{pred}})$$

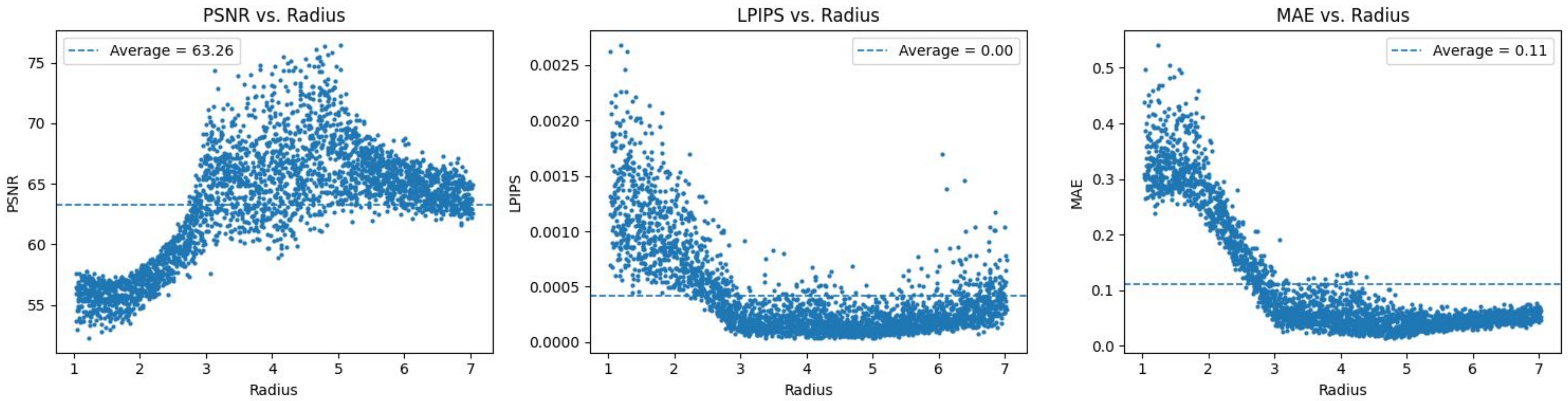
$$e_t$$

$$\mathrm{LPIPS}(I_{\mathrm{gt}}, I_{\mathrm{pred}})$$

Evaluation: Novel View Synthesis

Radius of train data in [3;5]

Radius of evaluation data in [1; 7]

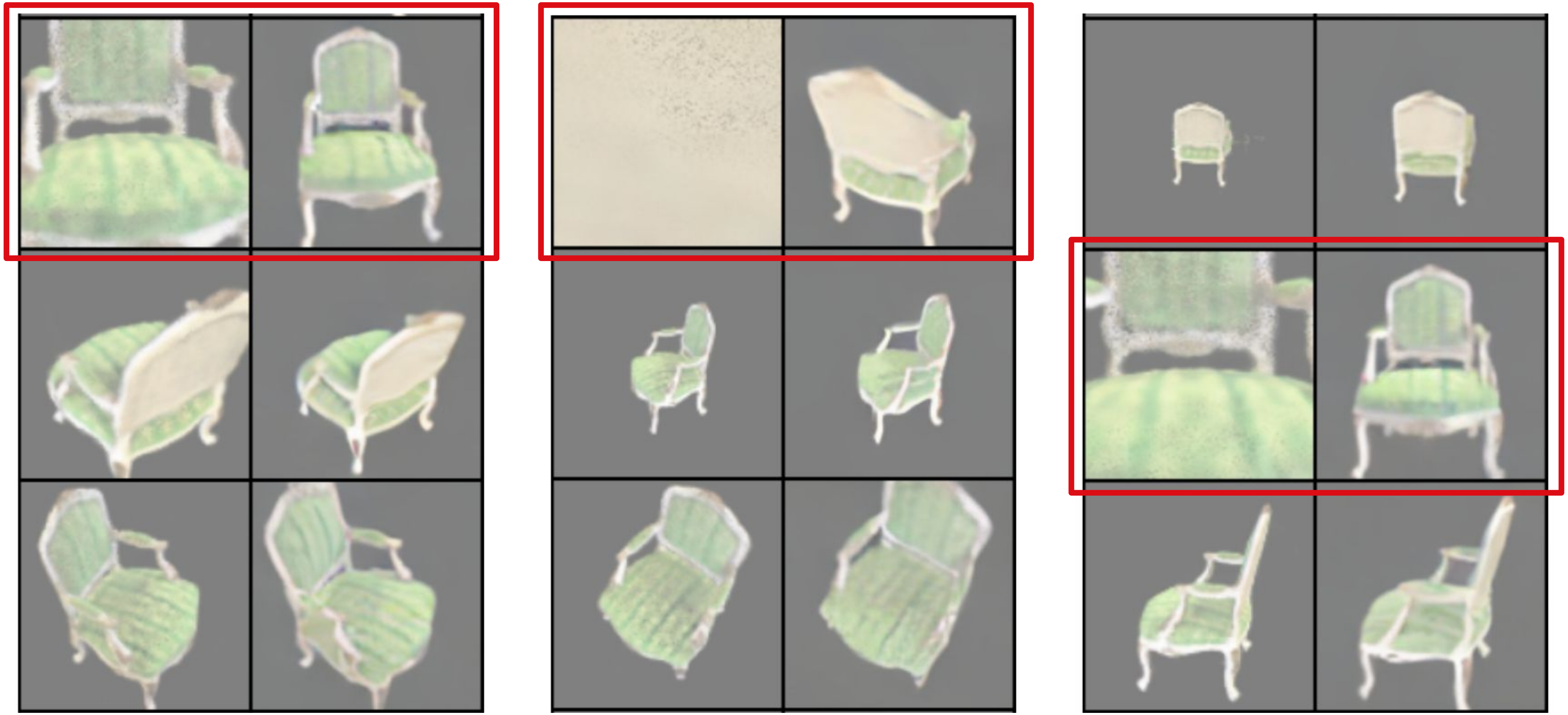


Evaluation: Pose Estimation

Table 1: Rotation error and translation error evaluation of the pose estimation

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

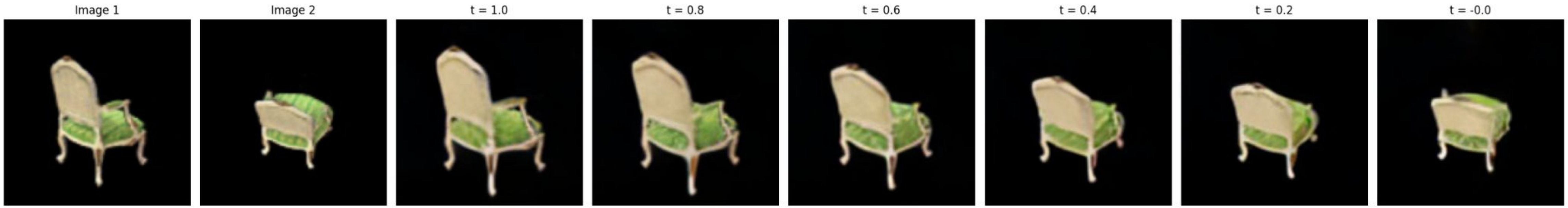
Evaluation: Novel View Synthesis examples



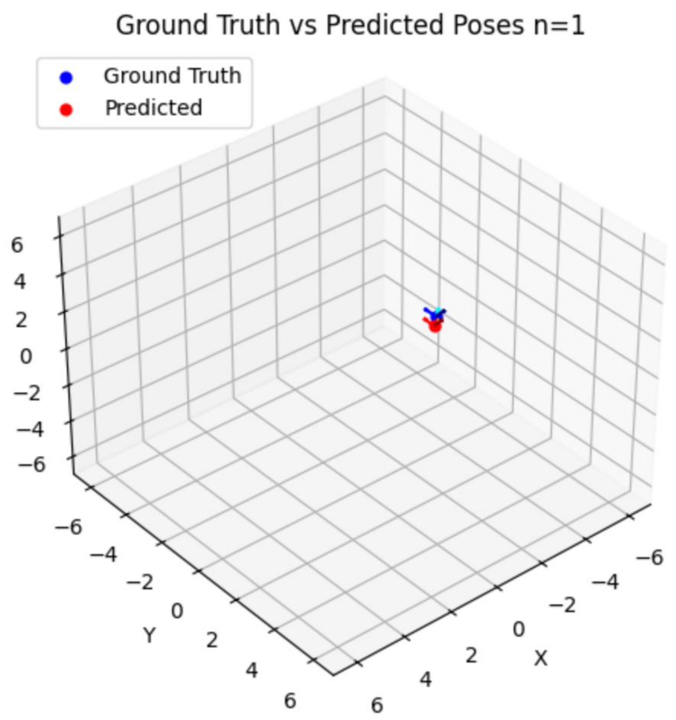
gt image prediction gt image prediction gt image prediction

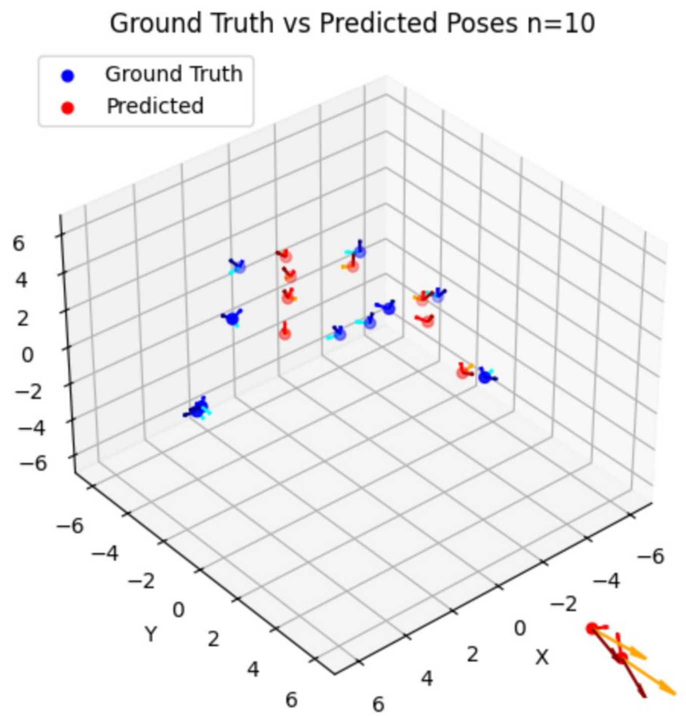
Evaluation: Pose Interpolation

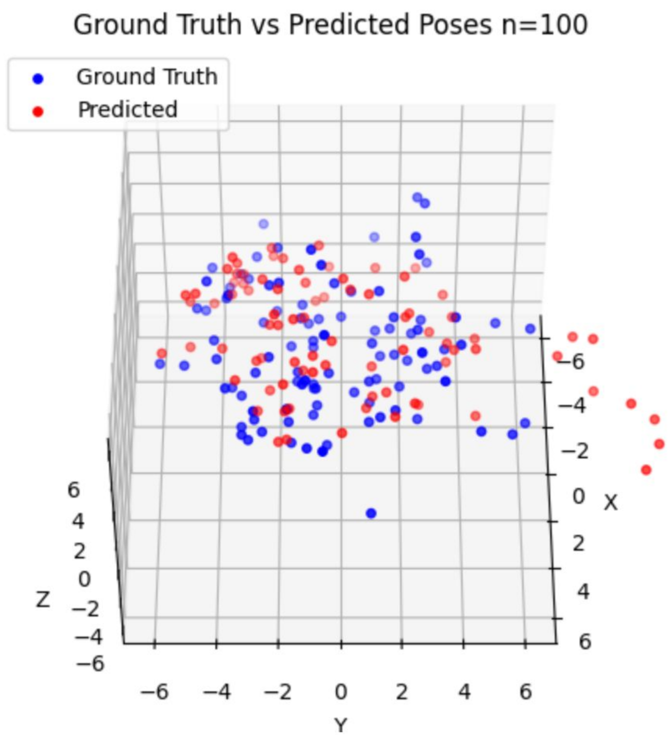
1. Randomly sample two **noisy_pose - image** pairs
2. Generate new starting poses as interpolations:
   t * noisy_pose_1 + (1 - t) * noisy_pose_2



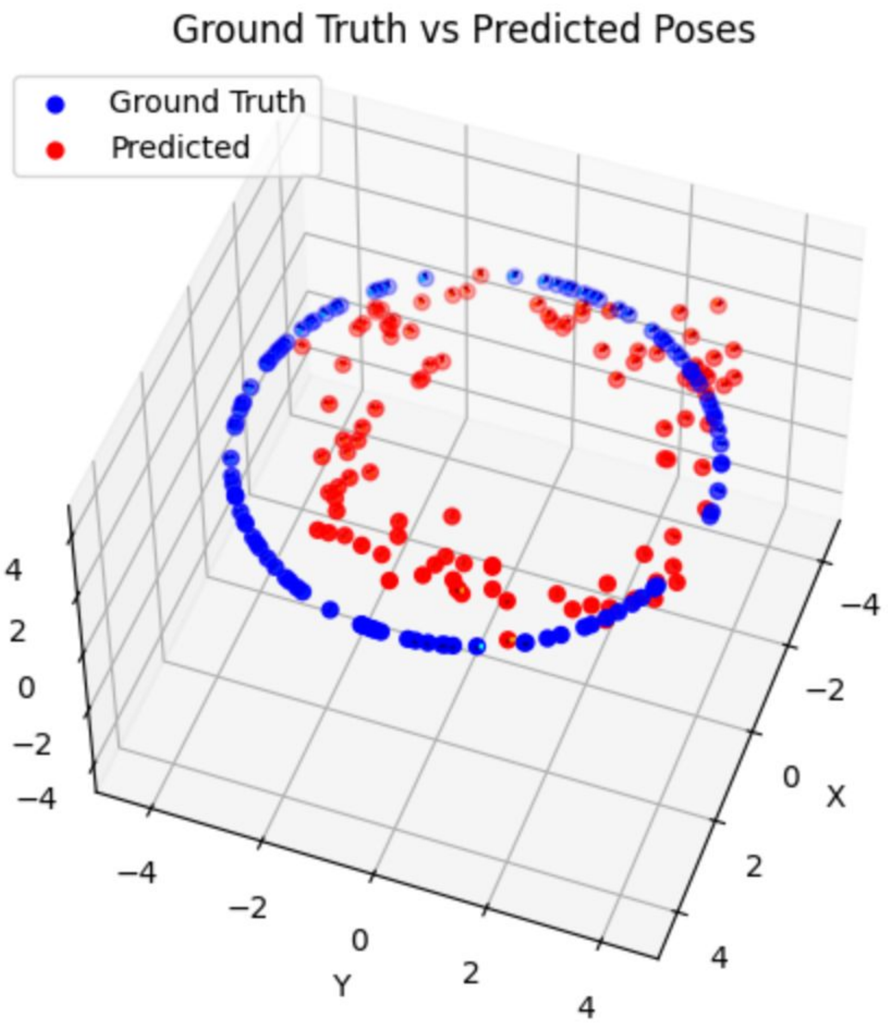
Evaluation: Pose Estimation Examples







Evaluation: Pose Estimation on Turntable data



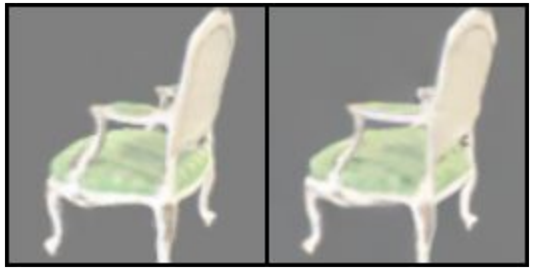
Evaluation: View synthesis - pose estimation loop

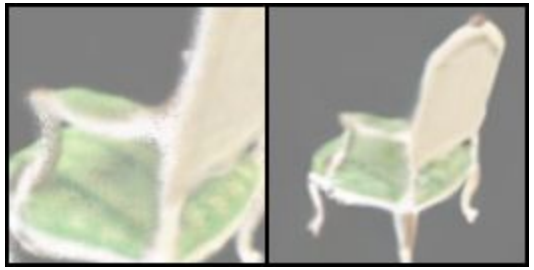
Experiment Setup

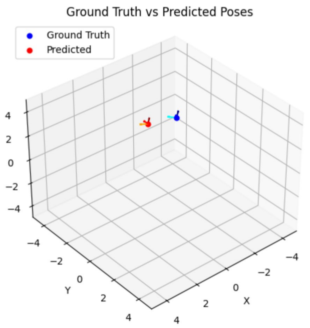
1. **First View synthesis:** Predict scene given noisy pose
2. **Pose Estimation:** Estimate the pose given the new scene
3. **Second View synthesis:** Predict scene again with estimated pose

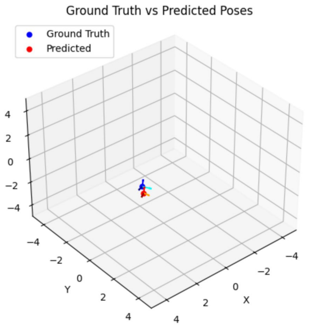
Evaluation: View synthesis - pose estimation loop

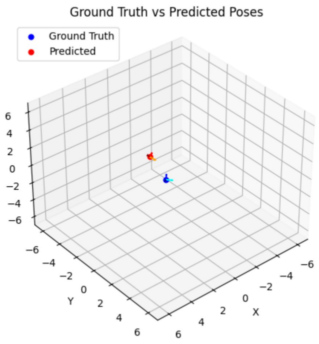
gt image prediction



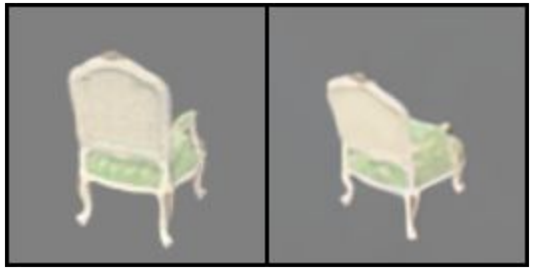


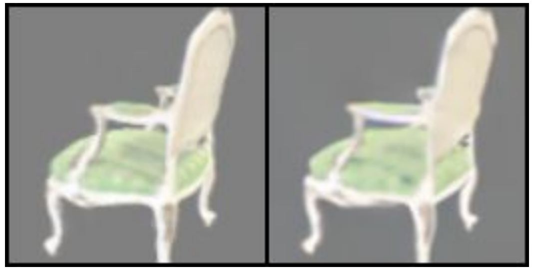


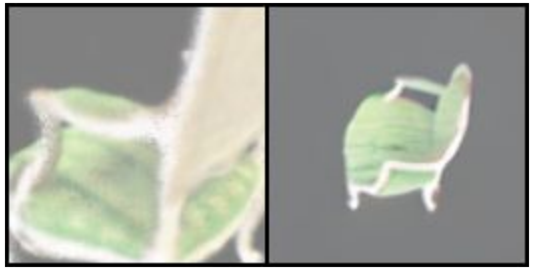




gt image prediction



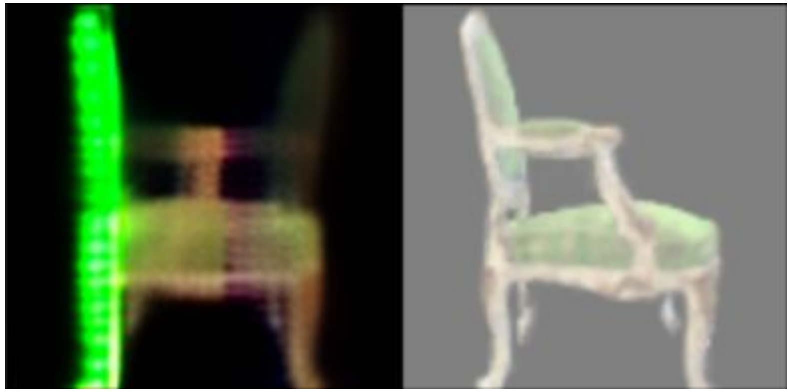




LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

SiT - Implementation: Failure Cases

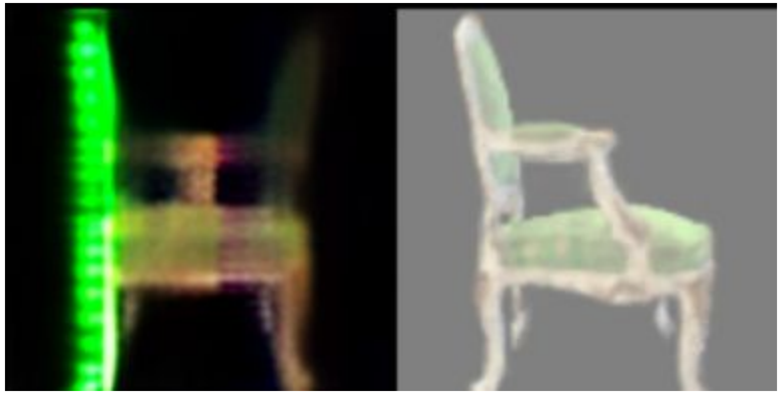
Prediction **GT image**



- Happened when we first implemented noisy pose
- Hypothesis: Overlapped latent between different poses
- Proposed solution: Decrease std of gaussian noise from 1 to 0.2

SiT - Implementation: Failure Cases

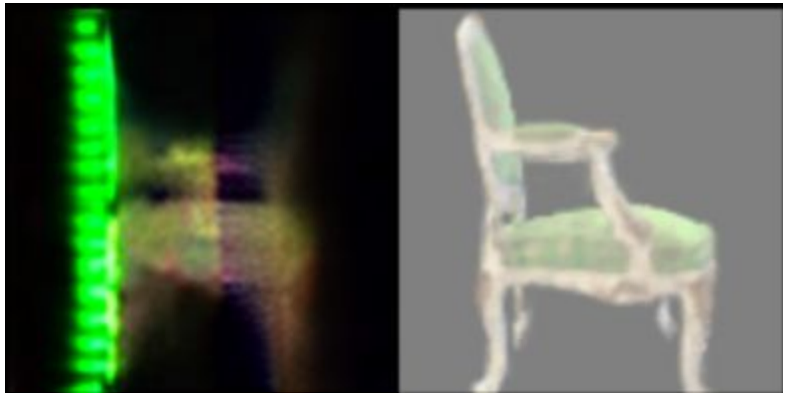
Prediction GT image



- Almost identical to the previous approach (less fuzzy)
- Hypothesis: Overlapped latent between different poses (still)
- Proposed solution: Remove noise entirely (forget generalization, focus on one to one mapping to at least work)

SiT - Implementation: Failure Cases

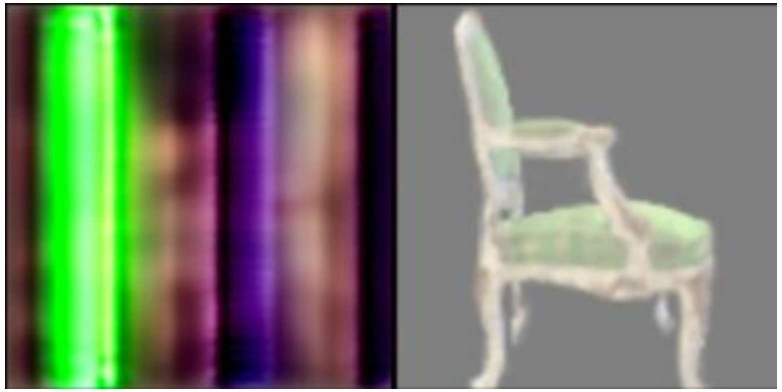
Prediction **GT image**



- Still a blend of images of different poses, more blurry
- Hypothesis: Image changes drastically with small change of pose value, huge noise to signal ratio
- Proposed solution: Amplify pose value by 10

SiT - Implementation: Failure Cases

Prediction **GT image**



- Unintelligible

SiT - Implementation: Failure Cases

💡Culprit & Solution:

```
# 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)
])
```

- transforms.RandomHorizontalFlip()
- transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5], inplace=True)

SiT - Implementation: Failure Cases

Pose de-standardization after flow reversal for pose estimation:

🔎**Initial failure:** Estimated pose has huge different than the ground truth.

🔥**Culprit:**
- Mean and standard deviation of the test set is used instead of those from training set.
- Mean and standard deviation is calculated using rotation AND translation pose value (semantically different).

💡**Solution:**
- Used mean and standard deviation from training set to de-standardize poses
- Separate the standardization and destandardization of the rotation matrix and the translation vector.

Thank You!