OUTLINE

- Introduction
- Related work
- Dataset
- Baseline
- Main approach
- Evaluation matric
- Results & analysis

Introduction

Why text to image?

Text to _____ ?

Intuitive? Interesting?

After taking the lesson, the question "text to ?" got imprinted in our mind. Therefore, when we are told to choose a topic for our final project, we decided to focus on this, and find the field we want to work on through two features: **Intuitive** and **Interesting**.

Intuitive means that the users can understand what our project is about easily.
Interesting is that we wish the application can attract people to try on it.
After discussing and brainstorming, we decided to work on text to image.

Introduction

Why dressing?

What do we care ?

What people of all ages care?

After deciding the field, we want our topic to be useful and attractive to all ages.
Therefore, we started thinking about what we ourselves need daily.

After listing things like how to set environment, how to make a presentation, how to get dressed, and a lot more, we find dressing the most common need for people.

Related Work

HUMANGAN

- a VAE-based framework
- images generated by sampling from the learned distribution

OURS

- texture-aware codebook
- sampler with mixture-of-experts
- feedforward index prediction network for the hierarchical sampling
- allows for controllable human generation by giving texts describing the desired attributes

Datasets

- Downloaded from DeepFashion-MultiModal and extra datasets which we downloaded from another website and we annotated it MANUALLY
- Each image includes densepose and human parsing and attributes for both clothes shapes and fabric
- Contain 10425 train images and 1159 test images
- Size of the dataset: 750x1101

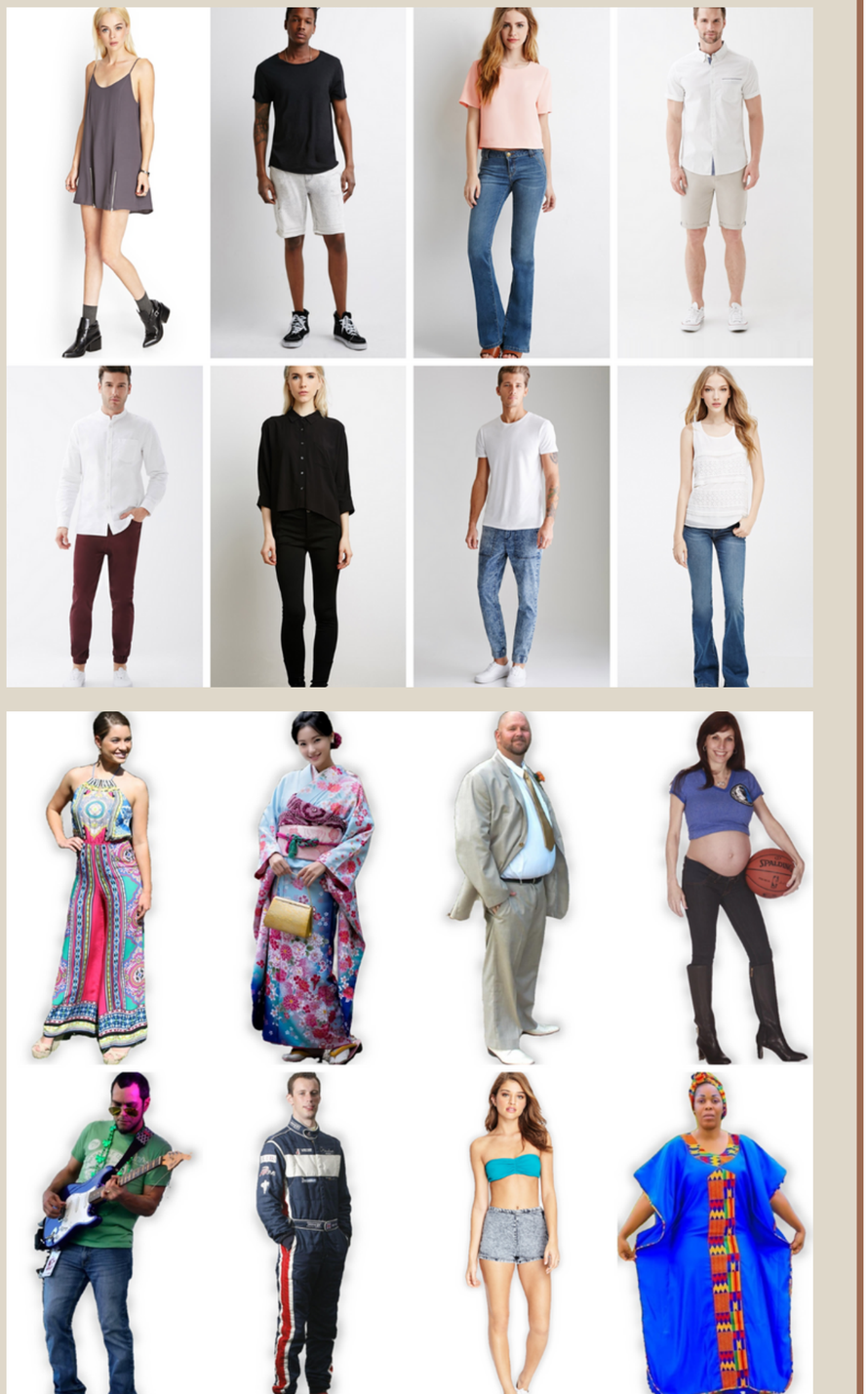
Datasets

Original Datasets

Contain normal clothing like T-shirt,and pants.

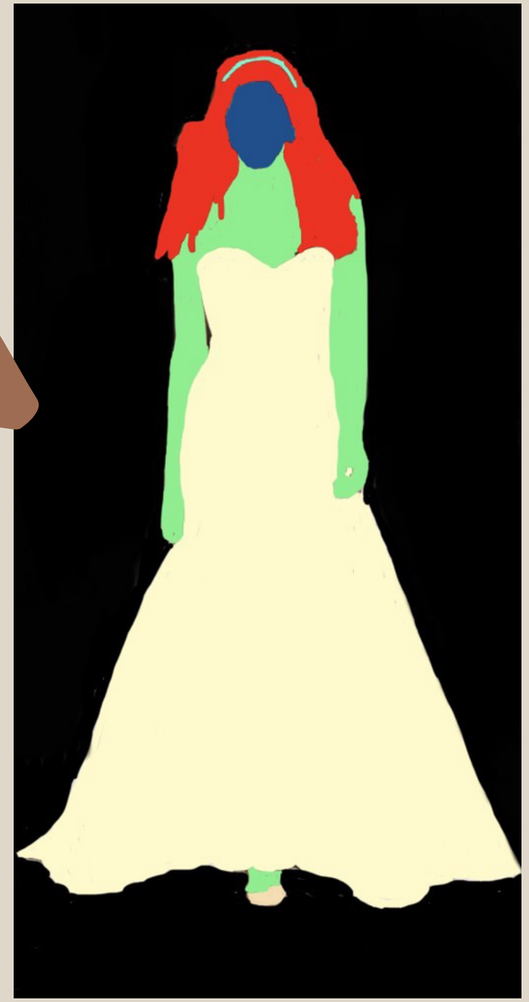
Extra Dataset

We made it more diversed (o^^o) !!



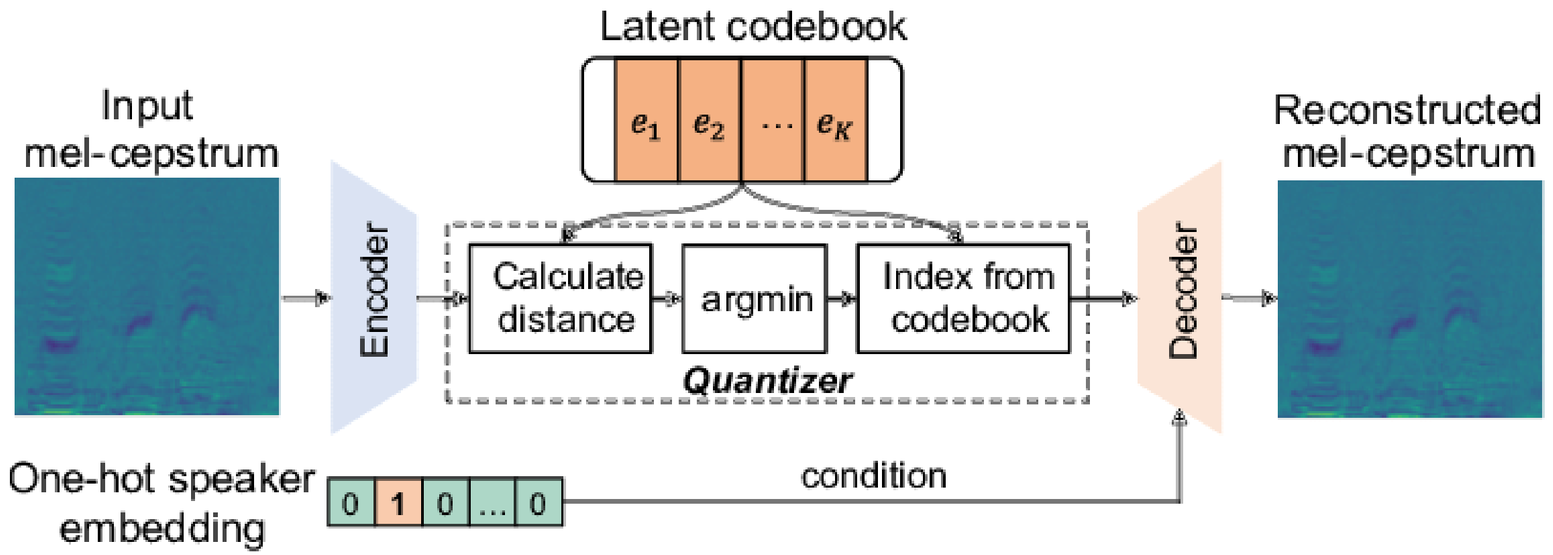
Datasets

For the segmentation dataset, we drew the colors by hand. The colors could be recognized and each of them matches to one annotation



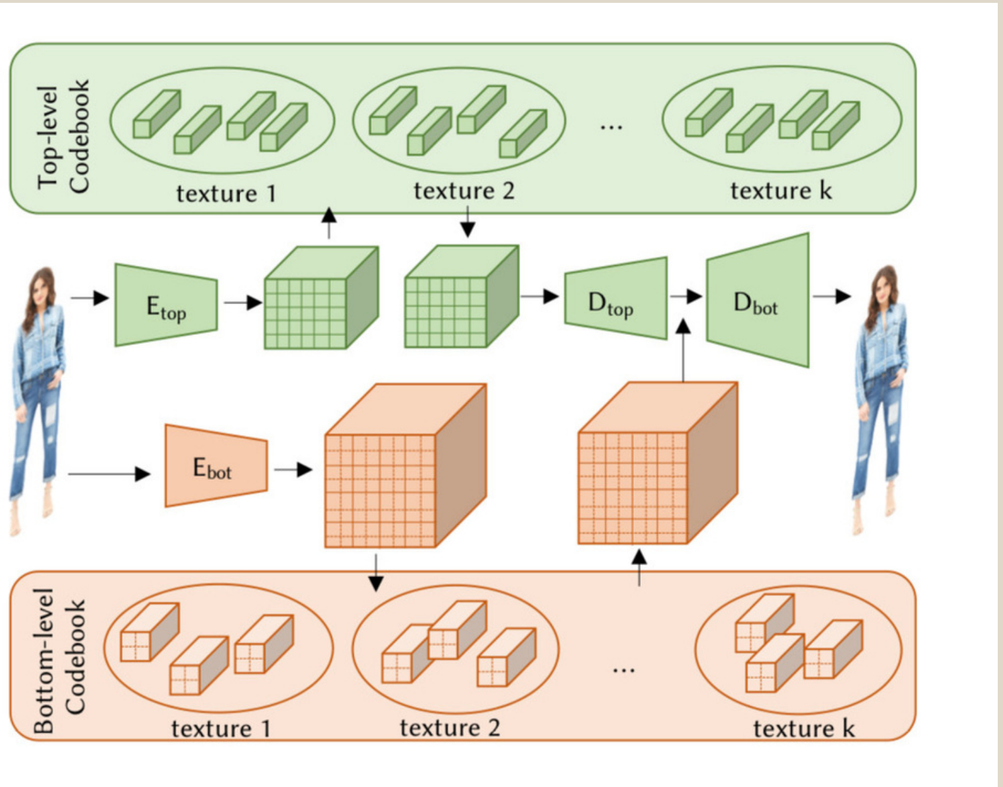
```
0: 'background' (0, 0, 0)
1: 'top' (255, 250, 250)
2: 'outer' (220, 220, 220)
3: 'skirt' (250, 235, 215)
4: 'dress' (255, 250, 205)
5: 'pants' (211, 211, 211)
6: 'leggings' (70, 130, 180)

7: 'headwear' (127, 255, 212)
8: 'eyeglass' (0, 100, 0)
9: 'neckwear' (50, 205, 50)
10: 'belt' (255, 255, 0)
11: 'footwear' (245, 222, 179)
12: 'bag' (255, 140, 0)
```

Baseline VQVAE



Baseline IMPLEMENTATION

- Is used to generate human images from the synthesized human parsing masks
- A hierarchical texture-aware codebook is built to store multi-scale neural representations for each type of texture
- The codebook indices are then generated using a diffusion-based transformer sampler with a mixture of experts
- To speed up the sampling process and refine the sampled images from the coarse level, a feed-forward codebook index prediction network is employed



Baseline

```python
class Encoder(nn.Module):

    def __init__(self,
                 ch,
                 num_res_blocks,
                 attn_resolutions,
                 in_channels,
                 resolution,
                 z_channels,
                 ch_mult=(1, 2, 4, 8),
                 dropout=0.0,
                 resamp_with_conv=True,
                 double_z=True):
        super().__init__()
        self.ch = ch
        self.temb_ch = 0
        self.num_resolutions = len(ch_mult)
        self.num_res_blocks = num_res_blocks
        self.resolution = resolution
        self.in_channels = in_channels
```

```python
class Decoder(nn.Module):

    def __init__(self,
                 in_channels,
                 resolution,
                 z_channels,
                 ch,
                 out_ch,
                 num_res_blocks,
                 attn_resolutions,
                 ch_mult=(1, 2, 4, 8),
                 dropout=0.0,
                 resamp_with_conv=True,
                 give_pre_end=False):
        super().__init__()
        self.ch = ch
        self.temb_ch = 0
        self.num_resolutions = len(ch_mult)
        self.num_res_blocks = num_res_blocks
        self.resolution = resolution
        self.in_channels = in_channels
        self.give_pre_end = give_pre_end
```

```python
class Discriminator(nn.Module):

    def __init__(self, nc, ndf, n_layers=3):
        super().__init__()

        layers = [
            nn.Conv2d(nc, ndf, kernel_size=4, stride=2, padding=1),
            nn.LeakyReLU(0.2, True)
        ]
        ndf_mult = 1
        ndf_mult_prev = 1
        for n in range(1,
                       n_layers):  # gradually increase the number of filters
            ndf_mult_prev = ndf_mult
            ndf_mult = min(2**n, 8)
            layers += [
                nn.Conv2d(
                    ndf * ndf_mult_prev,
                    ndf * ndf_mult,
                    kernel_size=4,
                    stride=2,
                    padding=1,
                    bias=False),
                nn.BatchNorm2d(ndf * ndf_mult),
                nn.LeakyReLU(0.2, True)
            ]
```

Baseline

```
557  class ResnetBlock(nn.Module):
558
559      def __init__(self,
560                   *,
561                   in_channels,
562                   out_channels=None,
563                   conv_shortcut=False,
564                   dropout,
565                   temb_channels=512):
566          super().__init__()
567          self.in_channels = in_channels
568          out_channels = in_channels if out_channels is None else out_channels
569          self.out_channels = out_channels
570          self.use_conv_shortcut = conv_shortcut
```

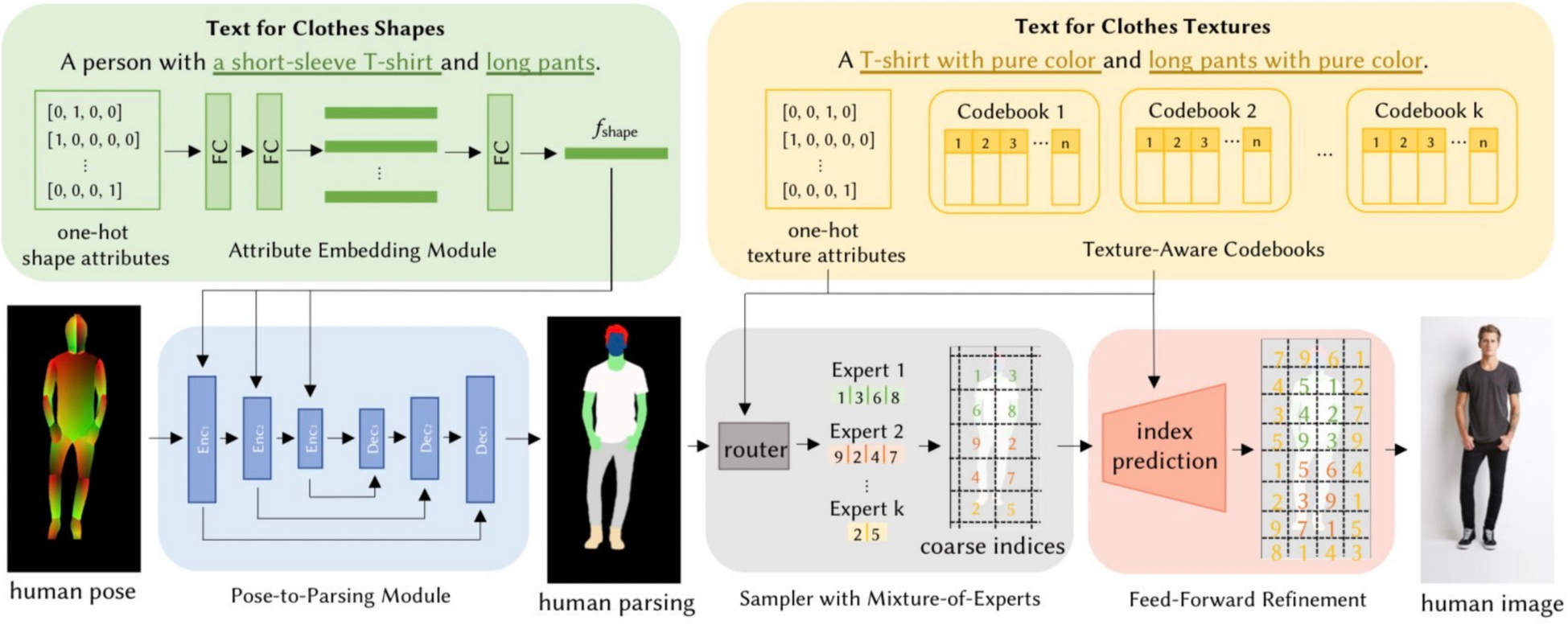
```
623  class AttnBlock(nn.Module):
624
625      def __init__(self, in_channels):
626          super().__init__()
627          self.in_channels = in_channels
628
629          self.norm = Normalize(in_channels)
630          self.q = torch.nn.Conv2d(
631              in_channels, in_channels, kernel_size=1, stride=1, padding=0)
632          self.k = torch.nn.Conv2d(
633              in_channels, in_channels, kernel_size=1, stride=1, padding=0)
634          self.v = torch.nn.Conv2d(
635              in_channels, in_channels, kernel_size=1, stride=1, padding=0)
636          self.proj_out = torch.nn.Conv2d(
637              in_channels, in_channels, kernel_size=1, stride=1, padding=0)
```

Main Approach

We decomposed it into Two stages

- Stage I: Pose to Parsing
- Stage II: Parsing to Human

- Input: Densepose, and texts of shape and texture
- Output: A person wearing clothes (>^ω^<)



Main Approach

STAGE 1: POSE TO PARSING

In the first stage of the project, a module called the **Pose-to-Parsing** Module was developed to generate a human parsing mask. This was done by combining human pose features with text descriptions of clothes' shapes. The text information was transformed into clothes **shape attributes**, which were then embedded and concatenated with the pose features. The encoder in the module spatially modulated the pose features and embedded attributes to produce the desired human parsing mask.

Main Approach

```python
class SampleFromPoseModel(BaseSampleModel):
    """SampleFromPose model.
    """

    def __init__(self, opt):
        super().__init__(opt)
        # pose-to-parsing
        self.shape_attr_embedder = ShapeAttrEmbedding(
            dim=opt['shape_embedder_dim'],
            out_dim=opt['shape_embedder_out_dim'],
            cls_num_list=opt['shape_attr_class_num']).to(self.device)
        self.shape_parsing_encoder = ShapeUNet(
            in_channels=opt['shape_encoder_in_channels']).to(self.device)
        self.shape_parsing_decoder = FCNHead(
            in_channels=opt['shape_fc_in_channels'],
            in_index=opt['shape_fc_in_index'],
            channels=opt['shape_fc_channels'],
            num_convs=opt['shape_fc_num_convs'],
            concat_input=opt['shape_fc_concat_input'],
            dropout_ratio=opt['shape_fc_dropout_ratio'],
            num_classes=opt['shape_fc_num_classes'],
            align_corners=opt['shape_fc_align_corners'],
        ).to(self.device)
        self.load_shape_generation_models()
```

```python
def generate_parsing_map(self):
    with torch.no_grad():
        attr_embedding = self.shape_attr_embedder(self.shape_attr)
        pose_enc = self.shape_parsing_encoder(self.pose, attr_embedding)
        seg_logits = self.shape_parsing_decoder(pose_enc)
    self.segm = seg_logits.argmax(dim=1)
    self.segm = self.segm.unsqueeze(1)
```

Main Approach

STAGE 2: PARSING TO HUMAN

In Stage II, the goal is to generate a human image using the human parsing mask obtained in Stage I. This involves enhancing the mask with diverse textures of clothes based on user-provided texts. A hierarchical texture-aware codebook is used to learn and store neural representations of different textures. The human image generation process includes sampling indices from the codebook at multiple levels. Coarse-level indices are predicted using a mixture-of-experts sampler, while fine-level indices are refined using a feed-forward network. The aim is to create a realistic human image by incorporating a range of textures into the parsing mask.

Main Approach

```python
def sample_and_refine(self, save_dir=None, img_name=None):
    # sample 32x16 features indices
    sampled_top_indices_list = self.sample_fn(
        temp=1, sample_steps=self.sample_steps)

    for sample_idx in range(self.batch_size):
        sample_indices = [
            sampled_indices_cur[sample_idx:sample_idx + 1]
            for sampled_indices_cur in sampled_top_indices_list
        ]
        top_quant = self.top_quantize.get_codebook_entry(
            sample_indices, self.texture_mask[sample_idx:sample_idx + 1],
            (sample_indices[0].size(0), self.shape[0], self.shape[1],
             self.opt["top_z_channels"]))

        top_quant = self.top_post_quant_conv(top_quant)

        bot_indices_list = self.bot_index_prediction(
            top_quant, self.texture_mask[sample_idx:sample_idx + 1])

        quant_bot = self.bot_quantize.get_codebook_entry(
            bot_indices_list, self.texture_mask[sample_idx:sample_idx + 1],
            (bot_indices_list[0].size(0), bot_indices_list[0].size(1),
             bot_indices_list[0].size(2),
             self.opt["bot_z_channels"]))  #.permute(0, 3, 1, 2)
        quant_bot = self.bot_post_quant_conv(quant_bot)
        bot_dec_res = self.bot_decoder_res(quant_bot)
```

```python
dec = self.decoder(top_quant, bot_h=bot_dec_res)

dec = ((dec + 1) / 2)
dec = dec.clamp_(0, 1)
if save_dir is None and img_name is None:
    return dec
else:
    save_image(
        dec,
        f'{save_dir}/{img_name[sample_idx]}',
        nrow=1,
        padding=4)
```

```python
def generate_texture_map(self):
    upper_cls = [1., 4.]
    lower_cls = [3., 5., 21.]
    outer_cls = [2.]

    mask_batch = []
    for idx in range(self.batch_size):
        mask = torch.zeros_like(self.segm[idx])
        upper_fused_attr = self.upper_fused_attr[idx]
        lower_fused_attr = self.lower_fused_attr[idx]
        outer_fused_attr = self.outer_fused_attr[idx]
        if upper_fused_attr != 17:
            for cls in upper_cls:
                mask[self.segm[idx] == cls] = upper_fused_attr + 1

        if lower_fused_attr != 17:
            for cls in lower_cls:
                mask[self.segm[idx] == cls] = lower_fused_attr + 1

        if outer_fused_attr != 17:
            for cls in outer_cls:
                mask[self.segm[idx] == cls] = outer_fused_attr + 1

        mask_batch.append(mask)
    self.texture_mask = torch.stack(mask_batch, dim=0).to(torch.float32)
```

EXPERIMENT

Result and Analysis

- Experiment I: Different Sample Steps
- Experiment II: Different trained models, one with original dataset, another with extended dataset

Evaluation Matric DIFFERENT SAMPLE STEPS

Sample Steps	50	100	150	200	250	300
Result						

We found that when sample steps is around to 250, it has the best result.

Evaluation Matric DIFFERENT MODELS

Original trained model					
Extended trained model					

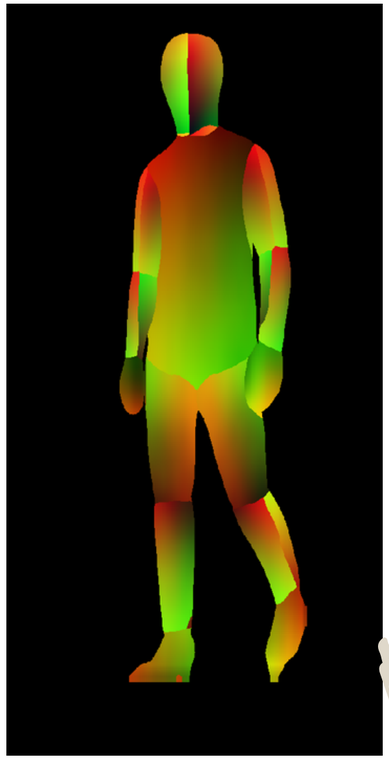
The below one is more diversity

Evaluation Matric DIFFFERENT TYPES OF CLOTHING

Short-pants	Long-pants	short-skirts	long-skirts	Short-jacket	Long-jacket
 (5,3.25)	 (4.75,3.75)	 (4,2.25)	 (1,1.25)	 (2.5,2.5)	 (3.75,4)
 (5,3.5)	 (5,4.25)	 (4.75,4.25)	 (3,3)	 (1.25,1.5)	 (3.25,3.25)
 (3.75,3)	 (4.25,3.5)	 (3.75,3)	 (2.5,2.25)	 (2,2.25)	 (2.75,2.5)

We give rates to each images by **completeness** and **preference** ,range from 1~5. The numbers below the images are the average rates given by our 4 members.

Pants perform better than skirts and jackets in completeness.



DISCUSSION AND ANALYSIS

Rseult and Analysis

We found that not every denseposes can generate the clothing with good quantility
Example: This densepose doesn't have a good quantility of generating long-pants

LIMITATION

Rsult and Analysis

We found that if the legs of the pose are not joined, the generated image would have broken skirt. The reason might be the bias caused by the dataset.

APPLICATION

Result and Analysis

We applyed the result images to 3D images, giving us more information about dressing