Supplementary - Wish You Were Here: Context-Aware Human Generation

A. Additional individual component replacement samples

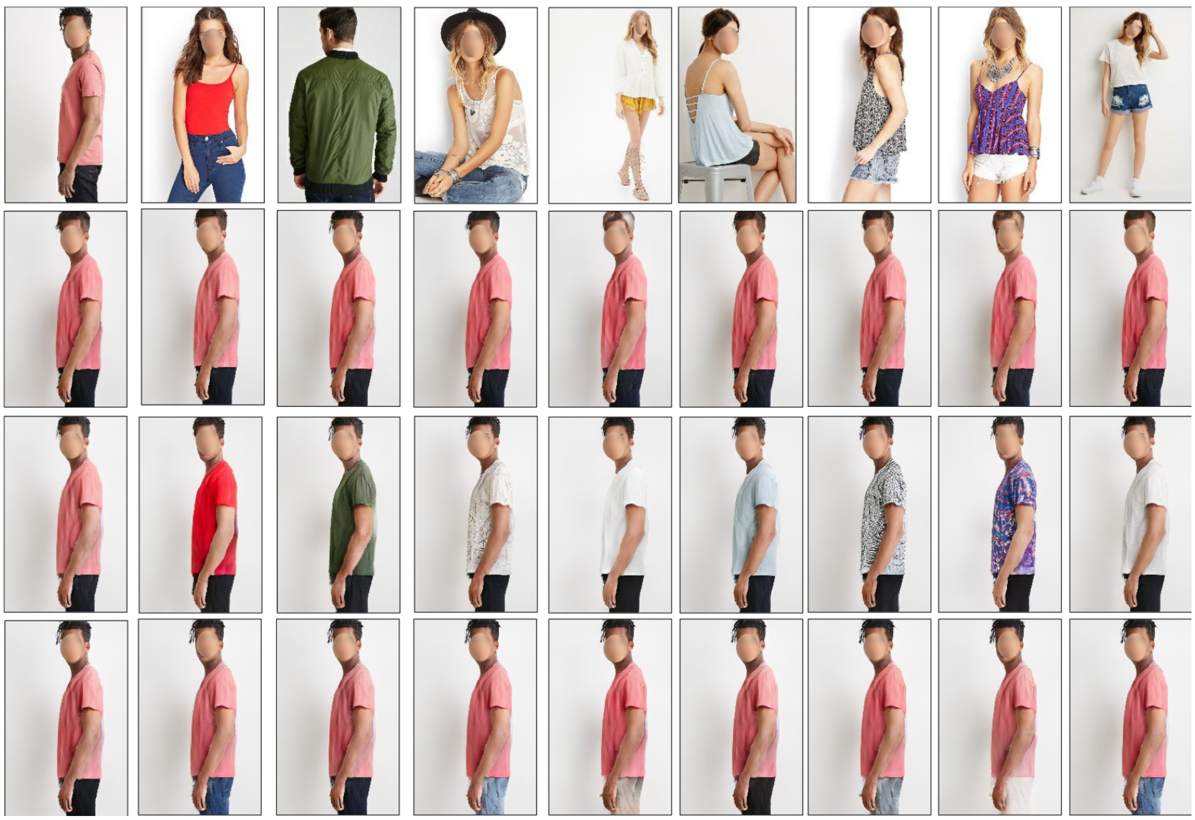

Figure 1: Replacing the hair, shirt, and pants (DeepFashion). For each target y (row 1), the hair (row 2), shirt (row 3), and pants (row 4), are replaced for the semantic map s of the upper-left person. The EGN and FRN are not used.

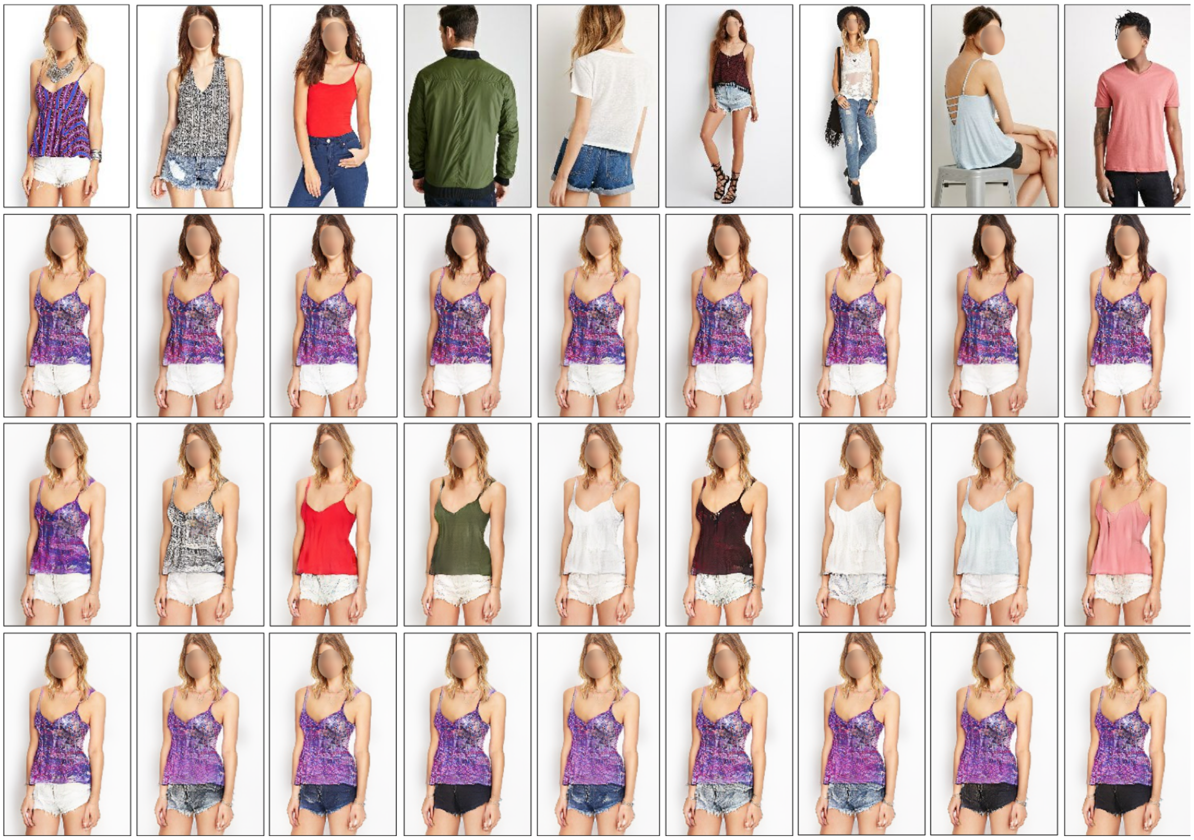

Figure 2: Replacing the hair, shirt, and pants (DeepFashion). For each target y (row 1), the hair (row 2), shirt (row 3), and pants (row 4), are replaced for the semantic map s of the upper-left person. The EGN and FRN are not used.

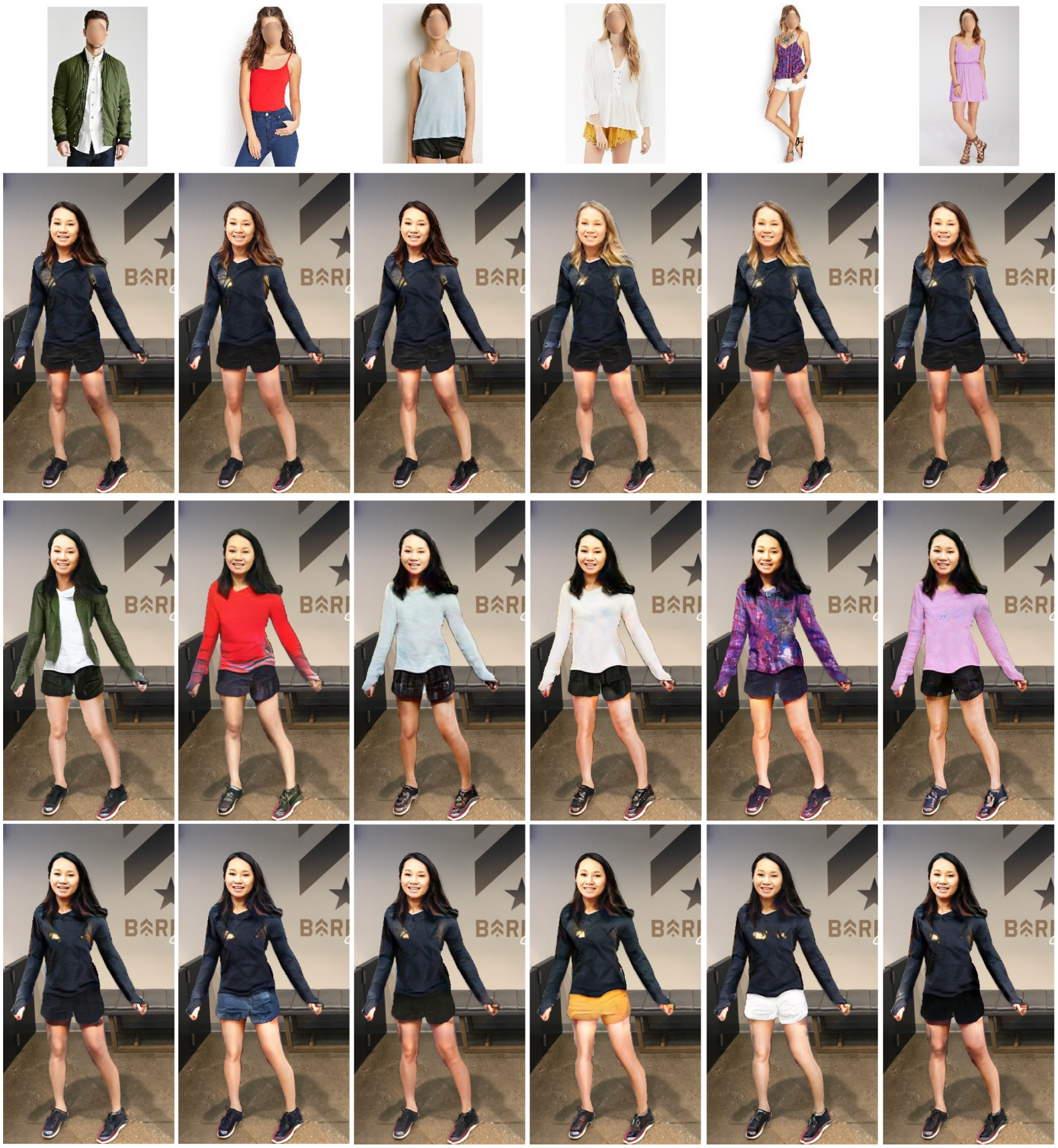

Figure 3: Replacing the hair, shirt, and pants for high-resolution unconstrained images. For each target y (row 1), the hair (row 2), shirt (row 3), and pants (row 4), are replaced using a chosen semantic map s.

B. Pose-transfer qualitative comparison

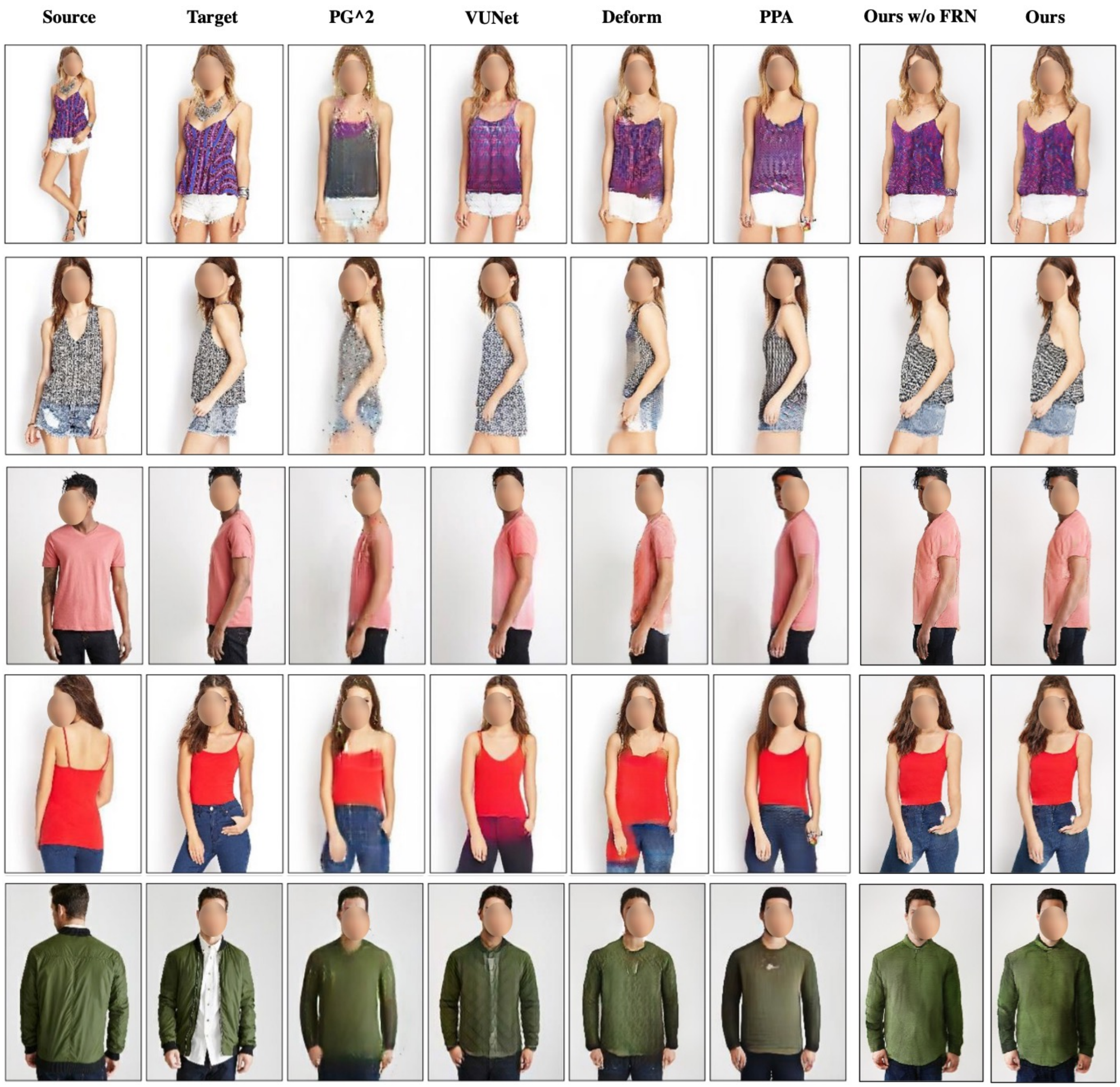

Figure 4: Comparison of our method on the pose-transfer task. Even without the Face Refinement Network, our method provides photorealistic rendered targets.

C. EGN training samples

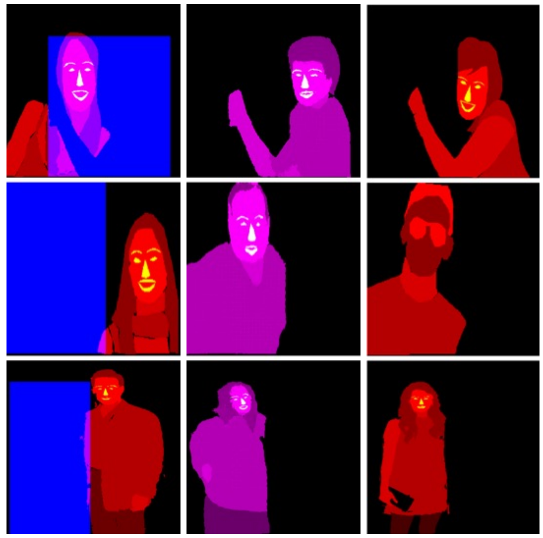

Figure 5: Training the Essence Generation Network. Shown for each row are the (a) input semantic map and bounding box, (b) generated semantic map, (c) ground truth semantic map. The scene essence is captured, while the generated semantic map is not identical to the ground truth.

D. DPBS and DPIS (Python) code

```
import numpy as np
import os
import cv2

gt_path = ’path/to/ground_truth_densepose’
gen_path = ’path/to/generated_densepose’

read_gen_by_order = True  # Read generated images by order, else by name
n_DP_MAX_IDX = 24  # DensePose generates I with values 0-24

def get_I_iou(img_gt, img_gen, I_idx):
    I_gt = np.zeros_like(img_gt)
    I_gen = np.zeros_like(img_gen)

    I_gt[img_gt == I_idx] = 1  # binarization of the GT image
    I_gen[img_gen == I_idx] = 1  # binarization of the generated image

    I_iou = get_iou(I_gt, I_gen)
    return I_iou

def get_iou(img_gt, img_gen):
    bin_gt = img_gt.copy()
    bin_gen = img_gen.copy()

    bin_gt[bin_gt > 0] = 1  # binarization of the GT image
    bin_gen[bin_gen > 0] = 1  # binarization of the generated image

    bin_union = bin_gt.copy()
    bin_union[bin_gen == 1] = 1  # union over gt and gen (1 where either is present)

    bin_overlap = bin_gt + bin_gen  # overlap of both
    bin_overlap[bin_overlap != 2] = 0  # overlap will be == 2
    bin_overlap[bin_overlap != 0] = 1  # binarization

    union_sum = np.sum(bin_union)
    if union_sum == 0:  # if neither the generated or GT image are present, mask out
        iou = -1
    else:
        iou = np.sum(bin_overlap) / union_sum

    return iou

def get_stats(metric, masked=False):
    if masked:
        return np.ma.mean(metric), np.ma.std(metric), np.ma.median(metric)
    else:
        return np.mean(metric), np.std(metric), np.median(metric)


gt_list = os.listdir(gt_path)  # get ground-truth file names
gt_list.sort()

gen_list = os.listdir(gen_path)  #  get ground-truth files
gen_list.sort()

n_list = len(gt_list)
n_gen = len(gen_list)
if n_list != n_gen:
    print(’Error. Ground-truth and generated folders do not contain the same number of images’)
    exit(1)
else:
    print(’Computing distance metrics over {} images.’.format(n_list))

DPBSs = np.zeros((n_list))  # DensePose Binary Similarity
```

```
DPISs = np.zeros((n_list))  # DensePose Index Similarity

for img_idx, filename in enumerate(gt_list):
    img_gt = cv2.imread(os.path.join(gt_path, filename), cv2.IMREAD_UNCHANGED)[:, :, 0]  # DP GT image
    if read_gen_by_order:
        img_gen = cv2.imread(os.path.join(gen_path, gen_list[img_idx]), cv2.IMREAD_UNCHANGED)[:, :, 0]
     # DP Generated image read by order
    else:
        img_gen = cv2.imread(os.path.join(gen_path, filename), cv2.IMREAD_UNCHANGED)[:, :, 0]  # DP
    Generated image read by name

    max_idx = max(np.amax(img_gt), np.amax(img_gen))  #  the max index is taken as the max between the
    generated and GT
    if max_idx > n_DP_MAX_IDX:
        print('Error. The maximum index value was {}. Should not be over 24'.format(max_idx))
        exit(1)

    DPBSs[img_idx] = get_iou(img_gt, img_gen)  # get DensePose Binary Similarity
    I_ious = np.zeros((n_DP_MAX_IDX))  # DPIS indices per image
    I_mask = np.ones_like(I_ious, dtype=bool)  # masking for DPIS indices per image
    for I_idx in range(1, max_idx + 1):  # iterated over the indices present
        I_ious[I_idx - 1] = get_I_iou(img_gt, img_gen, I_idx)  # index IoU (per body part)
    I_mask[I_ious != -1] = 0  # do not mask IoUs found
    masked_arr = np.ma.array(I_ious, mask=I_mask)  # masked IoUs

    DPISs[img_idx] = np.ma.mean(masked_arr)  # DensePose Index Similarity is calculated over the
    present indices

    if img_idx % 1000 == 0:
        print('Done with {}/{} images.'.format(img_idx, n_list))
DPBSs_mask = np.zeros_like(DPBSs, dtype=bool)  # masking for DPBS
DPBSs_mask[DPBSs == -1] = 1
masked_DPBSs_arr = np.ma.array(DPBSs, mask=DPBSs_mask)  # masked IoUs

DPBS_mean, DPBS_SD, DPBS_median = get_stats(masked_DPBSs_arr, masked=True)
DPIS_mean, DPIS_SD, DPIS_median = get_stats(DPISs)

print('----------------------------')
print('DPBS')
print('Mean: {}, SD: {}, Median: {}'.format(DPBS_mean, DPBS_SD, DPBS_median))
print('----------------------------')
print('DPIS')
print('Mean: {}, SD: {}, Median: {}'.format(DPIS_mean, DPIS_SD, DPIS_median))
print('----------------------------')
```