Projet S8 Veesual

Marc Sun, Mehdi Dahmani, Quentin Perrier, Yahya Janboubi

June 2022

Contents

1 Abstract

The project is about generating missing parts of the skin of a skinless mannequin wearing deformed clothes. We created a dataset using Deepfashion dataset but we wanted to have the capability to expand the dataset with our own data and not data taken from open-source datasets. So, we explored algorithms that performs semantic segmentation as it was the missing piece. For the skin generation model, we went with a simple architecture at the beginning to have some first results. Then, we move on to investigate adversarial networks as they usually perform well for this type of task. We looked into DCGAN but we quickly switch to pix2pix CGAN. We slightly modified the model so that it can takes multiple images as input and we performed many ablation tests to see what parameters gives us the best results. We also introduced some evaluation metrics in order to evaluate the quality of the images generated by the different models.

2 Introduction

At Veesual, they are working on the future of the Fashion e-commerce experience, and they are starting by launching a virtual try-on experience for fashion brand e-commerce sites. Each shopper can choose a model in which he/she identify and add any look on it. It's a new way of browsing an e-commerce site, more inclusive and more representative of the diversity of society.

In the world of Virtual Try-On in 2d, we are limited to the information that is visible on the input images in 2d, that is to say that for example, if we try to place a short-sleeved t-shirt on a photo of a model wearing a long-sleeved t-shirt, it is unclear what the model's arms look like. You have to generate them. Moreover, it is very difficult to deform a garment so that it perfectly matches the pose and morphology of the model. The results can often create inconsistencies in the final rendering, such as skin protrusions, an arm or a leg that is a little off the garment. Moreover, brands do not wish to see the rendering of their garment decrease. One of the ways to simplify the process is to deform the garment as best as possible, then re-generate the visible parts of the mannequin's skin from the garment as well as the ends of it (head, hands, feet).

The goal of this project is to make a model that will be able to generate missing parts of skin in order to reconstruct a complete image of a mannequin wearing an already deformed set of clothes. The input data will therefore be:

- the deformed clothes
- the head and feet of the mannequin
- the openpose keypoints of the mannequin
- the segmentation of the skin (optionnal)

The objective is to create a dataset of paired images of mannequins, train the model from it and evaluate the model.

In order to create dataset, we used Deepfashion dataset and did some preprocessing. Furthermore, we also wanted to have more data but the bottleneck was to get the semantic segmentation map of the mannequin in order to remove the skin. So, we explored some algorithms that do this specific task.

For the model, we went from simple architecture (FCNN) to more complex ones (DCGAN, CGAN).

3 Related work

Conditional GANs. To regularize image generation, several recent studies have extended GANs to conditional settings. For example, class or attribute labels [9], texts [10] or images [4] are used as conditional information. This added information allows for a generator to generate a specific image that is conditioned on it. Among them, our CGAN model is based on the CGAN Pix2Pix model [6] with a "U-Net"-based architecture generator and a convolutional "PatchGAN" classifier for the discriminator. We decided to go with this model as a base because we are working with paired images. However, we are conditioning on multiple images and we restrict our generative model to only modify a specific region of the inputs.

4 Dataset

For this project, we used Deepfashion, a public large-scale fashion database. In particular, we downloaded this dataset : DeepFashion: In-shop Clothes Retrieval [8]. We have chosen this dataset because it has many poses and clothes types. For clothes types, we have upper-body clothes, lower-body clothes and full-body clothes. The different poses are frontal view, side view, back view, zoom-out view, zoom-in view and stand-alone view. Moreover, we also have access to the semantic segmentation map of some images.

4.1 Semantic Segmentation

We kept only the images with a semantic segmentation map and with a frontal view pose. Then we classified each image as full body or upper body using their semantic segmentation map. The semantic segmentation map label each pixel in one of these categories using a specific color: top, skirt, leggings, dress, outer, pants, bag, neckwear, headwear, eyeglass, belt, footwear, hair, skin, face. We ended up with 1107 full body images and 5907 upper body images with 512x512 resolution. We have also 256x256 resolution images but we didn't use them as the quality was too low.

Figure 1: Full body and Upper body fashion mannequins

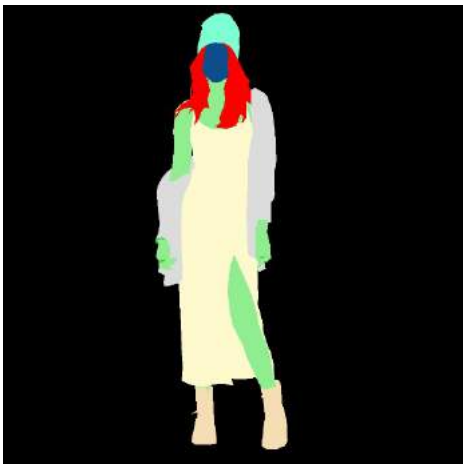


Figure 2: Semantic Segmentation map

For each of these images, we used their semantic segmentation map in order to create what will be the inputs and the output of our models.
Inputs :

• apparel_head_foot.jpg : we removed the skin from the fashion mannequin and the background

• segmentation_skin.jpg : we kept the segmentation of the skin only

Output :

• body.jpg : we removed the background

We decided to remove the background because after removing the skin from the fashion mannequin, we could see clearly the outline of the skin. This is an information that we do not necessarily want to give to our model.

Figure 3: Inputs: apparel_head_foot.jpg and segmentation_skin.jpg

Figure 4: Output: body.jpg

4.2 Openpose keypoints

In order to generate a specific pose of our fashion mannequin, we need to have as input some keypoints. For this purpose, we decided to use Openpose library [11]. By using this library, we were able to extract 25 keypoints (Nose, Neck, RShoulder, LShoulder ...) and generate this image which will be one of the inputs of our model.

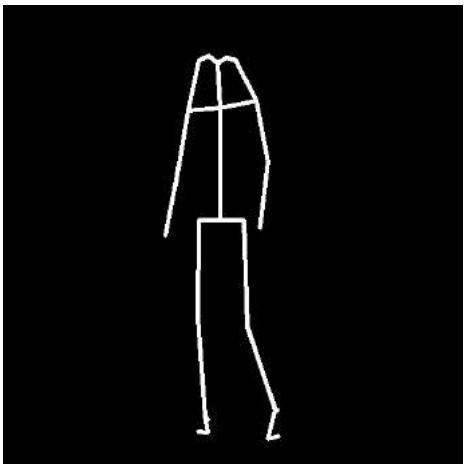


Figure 5: Openpose keypoints image

5 Methods

5.1 Semantic Segmentation map

We look into different methods to get the semantic segmentation map of an image. This will be useful when we want to create a dataset using our own images and not using open-souce images.

5.1.1 Method 1 (Opencv)

We tried other methods to segment the body. One of them was using python's library Open CV with the machine learning model Haar Cascade. But what is open CV and how does it work? Open CV is an open-source library for computer vision and image processing. In this project, we have used it extract the face from a picture. Here is an example of what we can obtain:

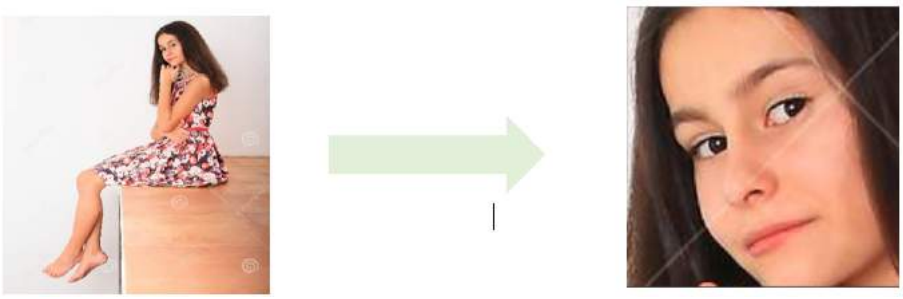


Figure 6: Head segmentation

To be more precise, we will be giving further explanations: to produce such results, the model uses a cascade of function which were trained from a lot of positive and negative images. Each function is obtained from features that come out of the picture. For example, a feature could be : the area around the eyes is often darker than the region of the nose and cheeks. To do so, the program does the sum of pixels on selected area. It then tries to deduce a large range of features and we select only the one with the least error.

Then, if a picture respects a minimum amount of those features, it is classified as face.

Haar Cascade can be use to identify many object. We have used it to extract other part of the body such as the feet or the legs. Here is an example of the accuracy of the algorithm:

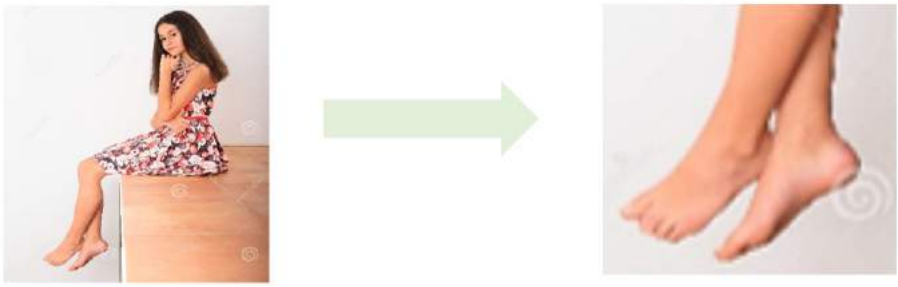


Figure 7: Foot segmentation

The accuracy is great but when the mannequin is wearing shoes, things tend to get a little more complicated. So, we tried the following process: • Using a neural network to detect if there are shoes on the picture • If so, we use the same network to segment the shoes • If no shoes, we use the previous algorithm This seemed like a good solution to the shoes problem. Unfortunately, the accuracy was not as good as we hoped.

5.1.2 Method 2 (TF)

Besides the shoes issue, the segmentation that was obtain with open CV was not exactly what we were looking for: we wanted to obtain something closer to this:

Figure 8: BodyPix

To do so, we try using another library which is BodyPix. BodyPix is based on TensorFlow which is a free and open-source software library for machine learning and artificial intelligence. BodyPix can classify the pixels of an image into two categories: 1) pixels that represent a person and 2) pixels that represent background. It can further classify pixels representing a person into any one of twenty-four body parts.

The first step of the algorithm is to create a mask to crop the person from the background. Then, it uses it own neural network to segmentate each pixel of the crop image. The result from BodyPix algorithm is much closer to what we are looking for.

Here is an example of what we can obtain :

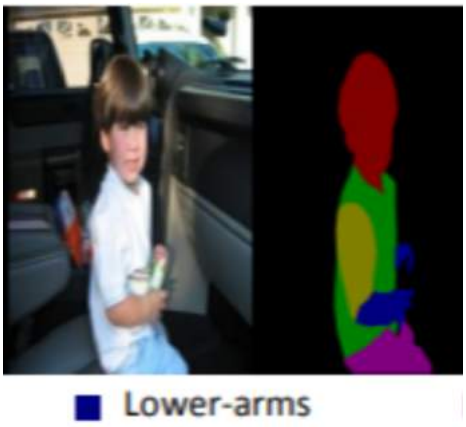


Figure 9: Body Pix example

A few more details about bodypix. You can set a segmentation threshold between 0 and 1 corresponding to how confident the computer should be about this pixel belonging to this part of the body. This way you can adapt the accuracy of the results to your needs. Body pix converts the original picture into a two-dimensional image with float values between 0 and 1 at each pixel indicating the probability that the person exists in that pixel. If the value is above the threshold, then this pixel is considered as part of the body and set to one. Else, it is considered as the background and set to zero. This way you can create a tighter crop around a person but may result in some pixels being that are part of a person being excluded from the returned part segmentation. Once the body is cropped from the background, the program moves on to the 23-part segmentation. The picture is converted in a matrix and each pixel now has a value between 0 and 23 to say which part of the body it represents. Cropping and segmentation are done by the same neural network Mobile Net. When going through the Mobile Net network to segment, the model builds an additional twenty-four channel output tensor P where twenty-four is the number of body parts. Each channel encodes the probability of a body part being present or not. Now to predict the body part, we use the following formula:

$$body\_part\_id = \underset{i \in I}{\operatorname{argmax}}(P(u, v, i))$$
$$where\ I = \{0, 1, ..., 23\}$$

Figure 10: Formula for segmentation

One final part about how to increase the speed and the accuracy: The model size and output stride have the biggest effects on performance and accuracy — they can either be set to values that make BodyPix run faster but less accurately

or more accurately but slower. - Model size is the larger of the layer. The bigger the number, the larger the layers. This will increase accuracy, but speed will decrease. - Output stride is set when doing the segmentation. The higher the value, the higher the speed. But it will decrease accuracy as well. - Size of the image also has a role to play in the speed of the program

5.2 Skin generation

We look into models that do skin generation from simple model to more complex ones.

5.2.1 Simple model

The first idea that came to our minds is to try to generate only some parts of the body. We choose the arms since they appear in the segmentation. To do that, we took a picture of a person and we removed the arms from it, and then try to generate a picture of the same person with full body (including removed arms).

Before using more complex deep learning models, we opted for a simple neural network and we trained it using a small dataset, and then we used it to generate the desired pictures.

5.2.1.1 Dataset

To be as simpler as possible, and to make the task of removing and generating arms easier, we kept the only a small portion of the Deepfashion dataset. We only used the pictures of men wearing shirt polos. Our dataset contains 81 pictures.

5.2.1.2 Preprocessing

In the preprocessing part, we focused on removing the arms from the pictures. For that, we used the segmentation that come with the Deepfashion dataset, then we selected the pixels forming the arms and changed their color the the background's color.

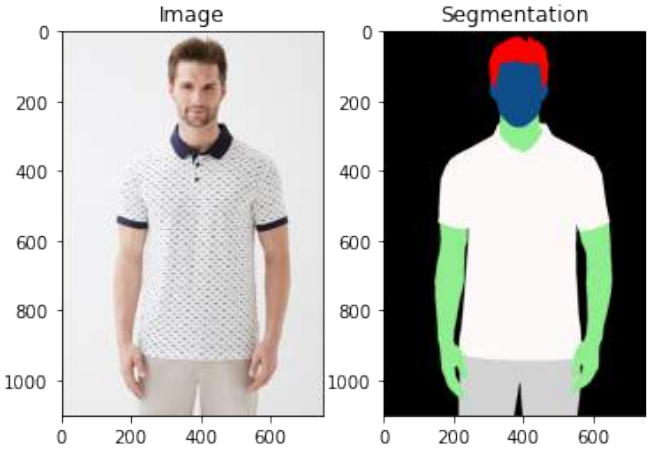


Figure 11: The original picture and its segmentation.

Figure 12: The result of preprocessing.

5.2.1.3 The network architecture

With this simple neural network, we mainly aim to regenerate the removed arms of the person. So the model takes the picture of the person without arms as input, which is a $1101 \times 750 \times 3$ picture, we have then a 2477250-d vector as an input for the neural network. The output of this network is going to be a 2477250-d vector as well, showing the predicted image with regenerated arms. For the rest of the network, we tried different architectures that we are going to discuss in the next section.

5.2.1.4 Training

In the training part, and since our model is simple and easy to train, we had the possibility to try many architectures with different number of hidden layers and different number of neurons in each layer. The optimal architecture in terms of results and execution time is as follows : one hidden layer with 20 neurons, followed by a *Relu* activation function. The loss function used in training is the *Mean squared error (MSE)*. This loss function compares the pictures pixelwise. The learning rate is $l_r = 10^{-2}$, the optimizer is *Adam* and we trained the model on 20 epochs.

5.2.1.5 Results

After the training is finished,the simple neural network succeeded to generate pictures from the input given. An example is given in the following figure :

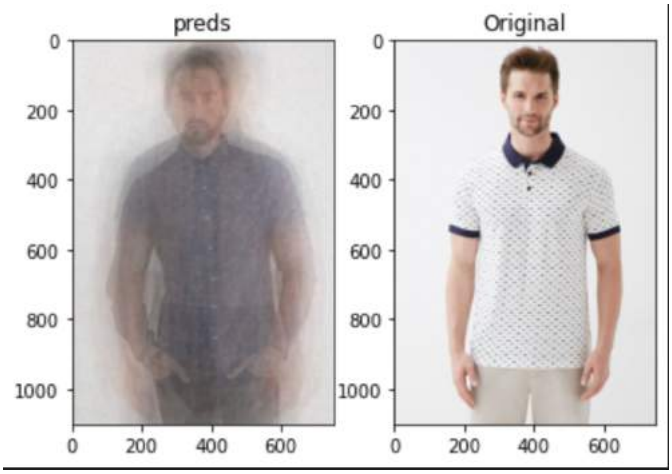


Figure 13: Example of results.

We can see that the trained model has generated blurry images that has the shape of a person with arms. The generated picture is clearly a superposition of many images of the dataset used in training. From this point, we can conclude that the simple models will fail in regenerating arms. We start then to consider using some deep neural networks, such as DCGAN and Pix2Pix that we discuss in the next sections.

5.2.2 DCGAN

GANs are generative models that learn a mapping from random noise vector z to output image y, $G : z \rightarrow y$. In contrast, conditional GANs learn a mapping from observed image x and random noise vector z, to y, $G : x, z \rightarrow y$. The generator G is trained to produce outputs that cannot be distinguished

from "real" images by an adversarially trained discriminator, $D : x, y \rightarrow t$, which is trained to do as well as possible at detecting the generator's "fakes". DCGAN is an extension of the GAN that we have described, but it uses convolutional and convolutional-transpose layers in both the discriminator part and the generator one. The discriminator has a convolution layers while the generator has a convolution-transpose layers. Batch norm layers and a relu activation are used for both. The input of the Generator is a latent vector called z, that follows a standard normal distribution and the output of it is the new generated image. This image is also the input of the Discriminator that should predict if this image is a fake or a real one.

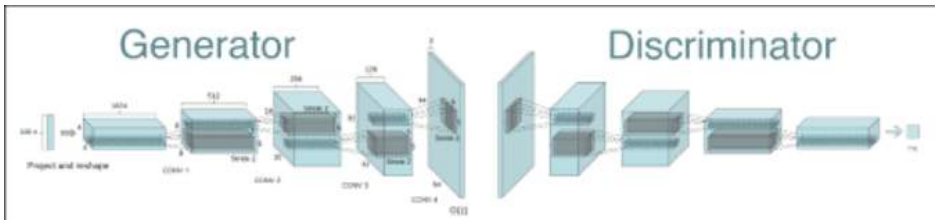


Figure 14: Convolutional and Convolutional-transpose layers

The idea behind the DCGAN is that the Discriminator tries to maximize the probability that it classifies correctly fakes and reals, but also the Generator wants to minimize the probability that the discriminator will predict its outputs fake. That's why we have this sort of loss functions:

$$\min_G \max_D V(D,G) = E_{x \sim p_{\text{data}}\,(x)}[\log D(x)] + E_{z \sim p_z(z)}[\log(1 - D(G(z)))]$$

5.2.2.1 The DCGAN architecture

The architecture that we used is almost the same the architecture in the paper DCGAN, we just added some additionnal layers to fit the dimension of our input images which is 512×512. The architecture of the DCGAN can be found on the paper [3]. For the additional layers, they are all of the same structure of DCGAN original layers, a $Conn2d$ followed by a $BatchNorm2d$ and finally a $LeakyReLU$. And for every layer added in the generator, we added its transpose in the discriminator in order to respect the architecture of a GAN.

5.2.2.2 Training Problems and Possible Solutions

After defining the modified DCGAN model , we ran the training on the large dataset above (section 4), on our personal computers. First, the model took infinite time to finish. We thought that it was due to the large dataset we had , so we tried to use only 100 sample from the initial dataset, and that to get at least a result. This time we got a RAM error, that's because PYTORCH store all the trining images in RAM before fitting the model. In addition to that, the

huge number of parameters that we had in our model, makes it more difficult for the RAM to store. The trivial solution to this model is to have enough RAM, hardware or cloud, which cannot be possible for us (16Gb of RAM that we used). A second possible solution was to change the model.

5.2.3 CGAN Pix2Pix based model

The CGAN is also a particular case of the GAN. The output can be a scalar value in the range $[0, 1]$ as well as a tensor of size 30×30 with values in the range $[0, 1]$. In our case, we do not add the noise vector z as we want to produce deterministic outputs. Furthermore, we consider x as a concatenation of n images which gives us a input of size $(512, 512, 3 * n)$ and t a tensor of size 30×30 with values in the range $[0, 1]$.

5.2.3.1 Generator

We use a U-net generator conditioned on multiples images. The U-net architecture is great for our task because is has an encoder decoder architecture with skip connections. The skip connections are useful because there is a great deal of low-level information shared between the inputs and output, and it would be desirable to send this information directly across the net. It will help recovering all information lost during the downsampling of the input at Encoder. During backpropagation, it even helps improve the gradient flow by avoiding the vanishing gradient issue. (See Section 8.1 for more details)

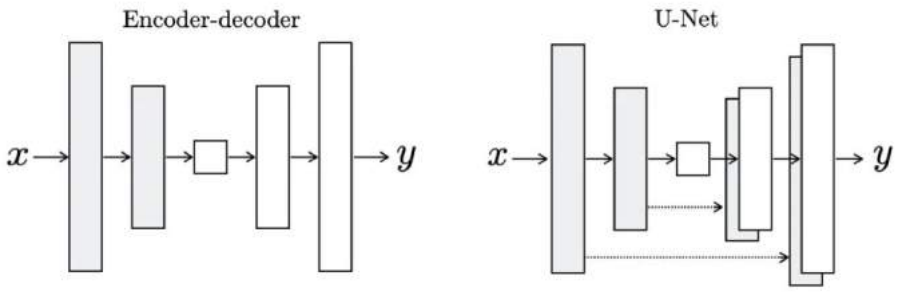


Figure 15: U-net Generator

5.2.3.2 Discriminator

The discriminator used is called PatchGan. The PatchGan discriminator has the same goal as any other GAN discriminator, i.e., to classify an input as real (sampled from the dataset) or fake (produced by generator). Its architecture differs a bit though, mainly in terms of how the input is regressed at the final layer. It outputs a tensor of values (30×30) in the range $[0, 1]$ instead of a scalar value in the range [0, 1], as seen in previous GAN architectures. Unlike

the traditional GAN model that uses a CNN with a single output to classify images, PatchGAN tries to classify if each NxN patch in an image is real or fake, rather than considering the entire image at one go. This way, it only penalizes structure at the scale of patches. Another feature of this discriminator is that it is conditioned on the input images. Hence, the input is a concatenated version of the real or fake image and the input images. (See Section 8.1 for more details)

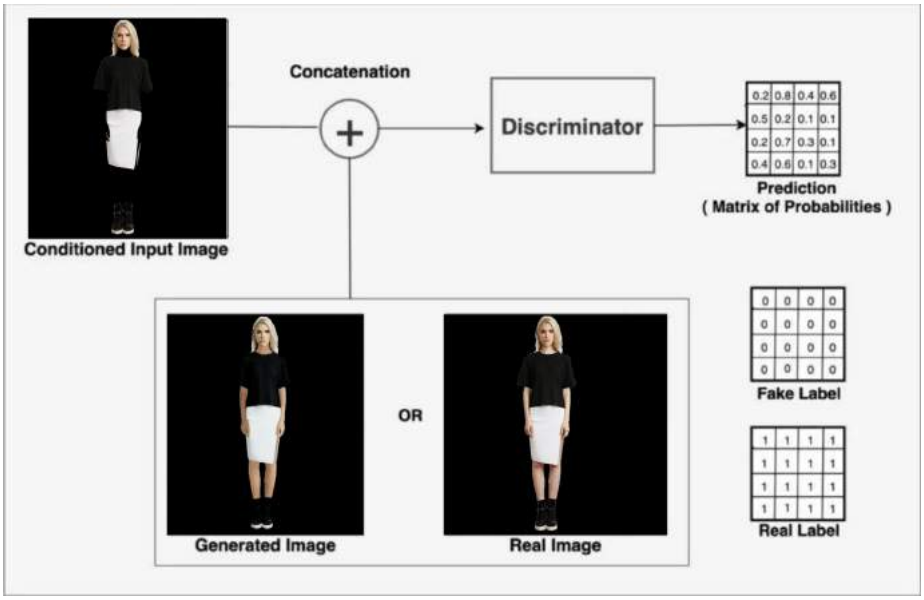


Figure 16: PatchGan Discriminator

5.2.3.3 Objective

The objective of a conditional GAN can be expressed as :

$$L_{cGAN}(G, D) = E_{x,y}[logD(x, y)] + E_x[log(1 - D(x, G(x)))]$$

where G tries to minimize this objective against an adversarial D that tries to maximize it, i.e. $G^* = argmin_G max_D L_{cGAN}(G, D)$.

To test the importance of conditioning the discriminator, we also compare to an unconditional variant in which the discriminator does not observe x:

$$L_{cGAN}(G, D) = E_y[logD(y)] + E_x[log(1 - D(G(x)))]$$

We also add an L1 distance so that the generator is tasked to not only fool the discriminator but also to be near the ground truth output. We prefer using L1 loss over L2 as L1 encourage less blurring.

$$L_{L1}(G) = E_{x,y}[\|y - G(x)\|_1]$$

Our final objective is $G^* = argmin_G max_D \lambda_{adv} L_{cGAN}(G, D) + \lambda_{rec} L_{L1}(G)$
With more details, here are the generator loss and the discriminator loss :

$$L_{cGAN}(G) = -\lambda_{adv} E_x[log(D(x, G(x)))] + \lambda_{rec} L_{L1}(G)$$

$$L_{cGAN}(D) = -\frac{E_{x,y}[logD(x, y)] + E_x[log(1 - D(x, G(x)))])}{2}$$

with a factor 2 for the discriminator loss in order to slow down the rate at which the discriminator learns relative to the generator. With this setup, rather than training G to minimize $log(1 - D(x, G(x))$, we instead train to maximize $logD(x, G(x))$.

6 Experiments

6.1 Skin generation using Pix2Pix CGAN model

6.1.1 Parameters

The model have many parameters that needs to be set. We trained the model on the 80% of the 1107 full body images and the rest was used for the testing. Many of the suggested values comes from those two papers.[4][6]

Training parameters

- We can set the number of epoch with *num_epochs*. We decided to go with 120 epochs as we saw that the generator loss was stabilizing at the end of the training. Furthermore, we had to have a decent amount of training time but still being able to do many tests. One run took around 2h00 (Tesla P100 16GB).

Discriminator and Generator parameters

- Inputs : we can choose which images we want our generator and discriminator to be conditioned on. We have to choose the inputs from the list ["apparel_head_foot","line","segmentation_skin"] with at least "apparel_head_foot".

- When we have in our inputs the segmentation of the skin (*segmentation_skin*), we can decide to only keep what the generator has generated on the skin region by setting the variable *mask* to True

- We can choose to condition or not the discriminator on the inputs by setting *add_discriminator_input_data* to $True$ or $False$.

- We apply the Adam solver on both the generator and the discriminator. We suggest to set a higher learning rate for generator (10 times) to that of adversarial discriminator in order to let the generator learn faster. For our different tests, we decided to set $l_{gen} = 2 * 10^{-4}$ and $l_{disc} = 2 * 10^{-5}$, and momentum parameters $\beta_1 = 0.5$, $\beta_2 = 0.999$.

- We can choose to use or not the adversarial loss or the l1 loss of the generator by setting the booleans *adv_loss* and *l1_loss*

Hardware dependant parameters

- We use minibatch SGD of 32 for the training (train_batch_size). The batch size depends on the hardware you use. We found that with a 32 batch size, we were able to fully use the GPU of Google Colab Pro.

- We have written a wrapper on pytorch Dataset class in order to load efficiently our batch of images with pytorch Dataloader class. With pytorch dataloader, we can specify the number of workers in order to allow multi-processing. After some tests, we decided to set *num_workers* to 0 because the loading was already fast enough with only the main processor for a *batch_size* of 32. The GPU was fully used and enabling multi-processing only slowed down the loading.

In our program, we require the user to give as input a dictionary with specific fields. Here is an example:

```
{
  "num_epochs": 120,
  "dataset_folder_path":"drive/MyDrive/ProjetS8/data/512/full_body/*",
  "img_size":(512,512),
  "train_batch_size":32,
  "test_batch_size":20,
  "learning_rate_generator":2e-4,
  "learning_rate_discriminator":2e-5,
  "betas":(0.5, 0.999),
  "path_results":"drive/MyDrive/ProjetS8/results/512/full_body",
  "num_workers":0,
  "input_name_list":["apparel_head_foot","line","segmentation_skin"],
  "add_discriminator_input_data":True,
  "adv_loss":True,
  "l1_loss":True,
  "mask":True,
}
```

6.1.2 Evaluation Metrics

Evaluating the quality of generated images is an difficult problem and is essential in this project. By having evaluation metrics, we can compare totally different models with each other. Moreover, we have a large number of images, so we can't evaluate the models by looking at every output. Here, we use a couple of metrics in order to evaluate the similarity of our output with the target. Some metrics are based on absolute error , other look into the structure of the image.

Mean square error (MSE)[1] $MSE = \frac{\sum_{p \in pixels_G} (p_{target} - p)^2}{Card(pixels_G)}$

MSE can be used to calculate the similarity of two images. In fact, it is can also be used as a loss function for the generator. We use it to quantify the difference in the values of each of the corresponding pixels between the sample and the reference image. MSE value is in range [0,+inf[with 0 being the best score

Peak Signal-to-Noise Ratio(PSNR)[12][1]) $PSNR = 20 * log(\frac{max(pixels)}{\sqrt{MSE}})$.

PSNR is an expression for the ratio between the maximum possible value (pixels) of an image and the power of distorting noise that affects the quality of

its representation. PSNR value is in range [0,+inf] with 20-40 considered to be a good score.

Structural Similarity Index(SSIM)[12][1] SSIM is a perception-based model that considers image degradation as perceived change in structural information, while also incorporating important perceptual phenomena, including both luminance masking and contrast masking terms. The difference with other techniques such as MSE or PSNR is that these approaches estimate absolute errors. Structural information is the idea that the pixels have strong interdependencies especially when they are spatially close. SSIM value is in range [0,1] with 1 being the best score

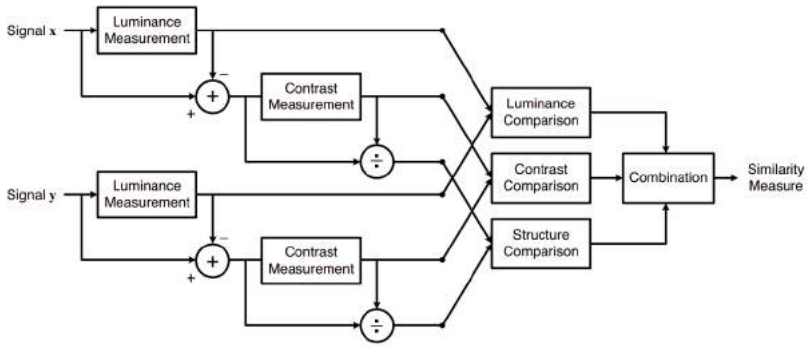


Figure 17: The Structural Similarity Measurement System

Frechet Inception Distance (FID)[5][2] $FID(x,g) = \|\mu_x - \mu_g\|_2^2 + Tr(\Sigma_x + \Sigma_g - 2(\Sigma_x^{\frac{1}{2}}\Sigma_g\Sigma_x^{\frac{1}{2}})^{\frac{1}{2}})$

FID measures the distance between the Inception-v3 activation distributions for generated and real samples. Rather than directly comparing images pixel by pixel (for example, as done by the MSE), the FID compares the mean and standard deviation of one of the deeper layers in CNN Inception v3. These layers are closer to output nodes that correspond to real-world objects such as a specific breed of dog or an airplane, and further from the shallow layers near the input image. As a result, they tend to mimic human perception of similarity in images. The FID metric is the current standard metric for assessing the quality of GANs as of 2020. It has been used to measure the quality of many recent GANs.

The calculation can be divided into three parts:

- We use the Inception network to extract 2048-dimensional activations from the pool3 layer for real and generated samples respectively.

- Then we model the data distribution for these features using a multivariate Gaussian distribution $\mathcal{N}(\mu_x, \Sigma_x)$ and $\mathcal{N}(\mu_g, \Sigma_g)$,

- Finally Wasserstein-2 distance is calculated for the mean and covariance of real images(x) and generated images(g).

FID value is in range [0,+inf[with 0 being the best score.

6.1.3 Analysis of the results on Pix2Pix based model

We did many tests by varying many parameters of our models. The results we obtained can be compared thanks to the evaluation metrics we have set (section 6.1.2). In the following paragraphs, we do ablation tests to see the influence of some parameters. You can find the exact parameters that was used for the different tests at the end of the report and some sample images generated by our models (see section 8.1).

6.1.3.1 Inputs

We have 3 images that can be put as inputs of our models: apparel_head_foot, line and segmentation_skin. We do the following tests with :

- T1 : ["$apparel\_head\_foot$"]
- T2 : ["$apparel\_head\_foot$","$line$"]
- T3 : ["$apparel\_head\_foot$","$line$","$segmentation\_skin$"]

From the results, we can see that adding the openpose keypoints as well as the segmentation of the skin increase the quality of our output.We see that the value of the metrics of T2 is better than T1. On the same note, T3 values are better than T2 ones. We see the utility of the openpose keypoints in the sample images (see section 8.1). With T1 configuration, we were not able to generate correctly the pose of the model but with T2 configuration, the result is better.

TEST	MSE	SSIM	PNSR	FID
1 input	0.027	0.90	17.9	163
2 inputs	0.017	0.90	20.1	158
3 inputs	0.006	0.94	26.1	155

6.1.3.2 L1 loss and adversarial loss

We do the following tests with :

- T1 : L1 loss
- T2 : Adversarial loss
- T3 : L1 loss + Adversarial loss

From the results, we can see that using the adversarial loss reduce the quality of our output. The results are really bad with only the adversarial loss and adding the adversarial loss to the l1 loss disturb the generator.

TEST	MSE	SSIM	PNSR	FID
L1	0.010	0.95	20.6	155
Adv	0.067	0.60	11.8	338
L1+Adv	0.011	0.94	20.1	158

6.1.3.3 Mask

Until now, generator was modifying the whole image (without mask), whereas it should be more efficient if we only let him modify the region where the skin

is (with mask). To see the influence of the mask, we do the following tests:

- T1 : without mask
- T2 : with mask

From the results, we can see that allowing the generator to only generate on skin region increase the qualify of the output by a lot.

TEST	MSE	SSIM	PNSR	FID
without mask	0.0030	0.96	26.1	155
with mask	0.0023	0.98	27.4	153

6.1.3.4 Condition on the discriminator

We do the following tests with :

- T1 : not conditioned
- T2 : conditioned

From the results, we can see that conditioning the discriminator do not change the the quality of the output.This is expected as we saw that the adversarial loss do not improve the quality of the image.

TEST	MSE	SSIM	PNSR	FID
not conditioned	0.0119	0.94	20.1	158
conditioned	0.0118	0.94	20.1	158

7 Conclusion

The results in this report suggest that conditional adversarial networks are not are working for generating missing skin. However, there are still many perspectives that need to be explored:

- In order to gather more data, we can code a scrapper to collect mannequin images from fashion websites. We also need to code an algorithm that filter theses images based on their poses and their clothes types. We can also use open-souce dataset and apply our semantic segmentation algorithm.
- We haven't explored data augmentation. The most common data augmentation techniques used for images are Position augmentation(Scaling,Cropping, Flipping, Padding, Rotation, Translation, Affine transformation) and Color augmentation (Brightness, Contrast, Saturation, Hue) .
- We did not used all the images that was at our disposal (upper-body model).
- Instead of using the skin segmentation as the region that the generator can modify, we can forbid the generator to touch the head, the feet and the apparel.
- We can also do ablation tests with the size of the patchGan.
- We can test other architectures for the generator.
- We can add a new neural network with the task of increasing the qualify of the image.
- We can try other approaches for image inpainting. [7]

-We can look at the influence of the noise as it was part of the basic theory of GANs (as an input of our generator or using dropout layers)

- We can do an ablation test with WGan + L1 and L1

References

[1] @andrewekhalel. *All image quality metrics you need in one package.* URL: https://github.com/andrewekhalel/sewar.

[2] @vict0rsch. *A simple wrapper around @mseitzer's great pytorch-fid work to compute Fréchet Inception Distance in-memory from batches of images, using PyTorch.* URL: https://github.com/vict0rsch/pytorch-fid-wrapper.

[3] Soumith Chintala Alec Radford Luke Metz. *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. arXiv preprint arXiv:1511.06434, 2016.*

[4] E. Denton, S. Chintala, A. Szlam, and R. Fergus. Deep. *Generative image models using a Laplacian pyramid of adversarial networks. In NIPS, 2015.*

[5] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. *GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium.* URL: https://arxiv.org/abs/1706.08500.

[6] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. *Image-to-Image Translation with Conditional Adversarial Networks.*

[7] Chu-Tak Li. *10 Papers You Must Read for Deep Image Inpainting.* URL: https://towardsdatascience.com/10-papers-you-must-read-for-deep-image-inpainting-2e41c589ced0.

[8] Ziwei Liu, Ping Luo, Shi Qiu, Xiaogang Wang, and Xiaoou Tang. *DeepFashion: In-shop Clothes Retrieval.* June 2016. URL: http://mmlab.ie.cuhk.edu.hk/projects/DeepFashion/InShopRetrieval.html.

[9] M. Mirza and S. Osindero. *Conditional generative adversarial nets. arXiv preprint arXiv:1411.1784, 2014.*

[10] S. Reed, Z. Akata, X. Yan, L. Logeswaran, B. Schiele, and H. Lee. *Generative adversarial text to image synthesis. In ICML, 2016.*

[11] Github repository. *OpenPose: Real-time multi-person keypoint detection library for body, face, hands, and foot estimation.* URL: https://github.com/CMU-Perceptual-Computing-Lab/openpose.

[12] Zhou Wang, Alan Conrad Bovik, and Hamid Rahim Sheikhand Eero P. Simoncelli. *Image Quality Assessment: From Error Visibility to Structural Similarity.* URL: https://www.cns.nyu.edu/pub/eero/wang03-reprint.pdf.

8 Annex

8.1 Pix2Pix architecture

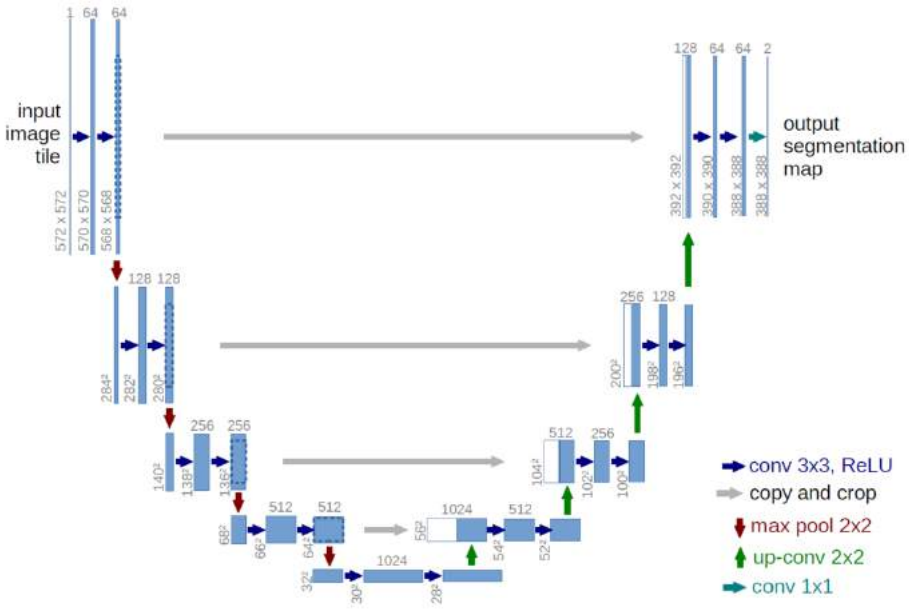
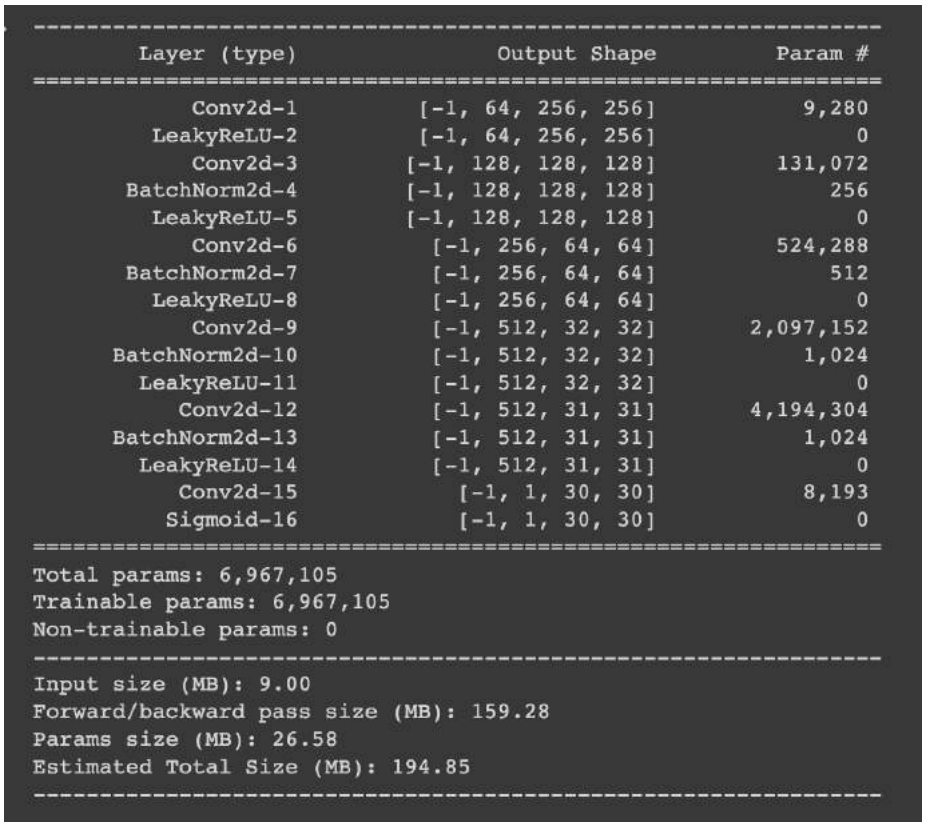


Figure 18: U-net visual architecture



Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 256, 256]	9,280
LeakyReLU-2	[-1, 64, 256, 256]	0
Conv2d-3	[-1, 128, 128, 128]	131,072
BatchNorm2d-4	[-1, 128, 128, 128]	256
LeakyReLU-5	[-1, 128, 128, 128]	0
Conv2d-6	[-1, 256, 64, 64]	524,288
BatchNorm2d-7	[-1, 256, 64, 64]	512
LeakyReLU-8	[-1, 256, 64, 64]	0
Conv2d-9	[-1, 512, 32, 32]	2,097,152
BatchNorm2d-10	[-1, 512, 32, 32]	1,024
LeakyReLU-11	[-1, 512, 32, 32]	0
Conv2d-12	[-1, 512, 31, 31]	4,194,304
BatchNorm2d-13	[-1, 512, 31, 31]	1,024
LeakyReLU-14	[-1, 512, 31, 31]	0
Conv2d-15	[-1, 1, 30, 30]	8,193
Sigmoid-16	[-1, 1, 30, 30]	0

Total params: 6,967,105
Trainable params: 6,967,105
Non-trainable params: 0

Input size (MB): 9.00
Forward/backward pass size (MB): 159.28
Params size (MB): 26.58
Estimated Total Size (MB): 194.85

Figure 19: PatchGan architecture

8.2 Analysis of the results

Here are the different configuration file that were used for the different ablation tests we did on section 6.1.3. We also added some sample images generated by our best model:

8.2.1 Inputs

T1 : 1 input

```
config_file = {
      "num_epochs": 120,
      "dataset_folder_path":"drive/MyDrive/ProjetS8/data/512/full_body/*",
      "img_size":(512,512),
      "train_batch_size":32,
      "test_batch_size":20,
      "learning_rate_generator":2e-4,
      "learning_rate_discriminator":2e-5,
```

```
        "betas":(0.5, 0.999),
        "path_results":"drive/MyDrive/ProjetS8/results/512/full_body",
        "num_workers":0,
        "input_name_list":["apparel_head_foot"],
        "add_discriminator_input_data":True,
        "adv_loss":True,
        "l1_loss":True,
        "mask":False,
}
```

Figure 20: T1 : 1 input

T2 : 2 inputs

```
config_file = {
        "num_epochs": 120,
        "dataset_folder_path":"drive/MyDrive/ProjetS8/data/512/full_body/*",
        "img_size":(512,512),
        "train_batch_size":32,
        "test_batch_size":20,
        "learning_rate_generator":2e-4,
        "learning_rate_discriminator":2e-5,
```

```
        "betas":(0.5, 0.999),
        "path_results":"drive/MyDrive/ProjetS8/results/512/full_body",
        "num_workers":0,
        "input_name_list":["apparel_head_foot","line"] ,
        "add_discriminator_input_data":True,
        "adv_loss":True,
        "l1_loss":True,
        "mask":False,
}
```

Figure 21: T2 : 2 inputs

T3: 3 inputs

```
config_file = {
        "num_epochs": 120,
        "dataset_folder_path":"drive/MyDrive/ProjetS8/data/512/full_body/*",
        "img_size":(512,512),
        "train_batch_size":32,
        "test_batch_size":20,
        "learning_rate_generator":2e-4,
        "learning_rate_discriminator":2e-5,
```

```
        "betas":(0.5, 0.999),
        "path_results":"drive/MyDrive/ProjetS8/results/512/full_body",
        "num_workers":0,
        "input_name_list":["apparel_head_foot","line","segmentation\_skin"] ,
        "add_discriminator_input_data":True,
        "adv_loss":True,
        "l1_loss":True,
        "mask":False,
}
```

Figure 22: T3: 3 inputs

8.2.2 L1 loss and adversarial loss

T1 : L1 loss

```
config_file = {
        "num_epochs": 120,
        "dataset_folder_path":"drive/MyDrive/ProjetS8/data/512/full_body/*",
        "img_size":(512,512),
        "train_batch_size":32,
        "test_batch_size":20,
```

```
        "learning_rate_generator":2e-4,
        "learning_rate_discriminator":2e-5,
        "betas":(0.5, 0.999),
        "path_results":"drive/MyDrive/ProjetS8/results/512/full_body",
        "num_workers":0,
        "input_name_list":["apparel_head_foot","line"],
        "add_discriminator_input_data":True,
        "adv_loss":False,
        "l1_loss":True,
        "mask":False,
}
```

Figure 23: T1 : L1 loss

T2 : Adv loss

```
config_file = {
        "num_epochs": 120,
        "dataset_folder_path":"drive/MyDrive/ProjetS8/data/512/full_body/*",
        "img_size":(512,512),
        "train_batch_size":32,
        "test_batch_size":20,
```

```
        "learning_rate_generator":2e-4,
        "learning_rate_discriminator":2e-5,
        "betas":(0.5, 0.999),
        "path_results":"drive/MyDrive/ProjetS8/results/512/full_body",
        "num_workers":0,
        "input_name_list":["apparel_head_foot","line"],
        "add_discriminator_input_data":True,
        "adv_loss":True,
        "l1_loss":False,
        "mask":False,
}
```

Figure 24: T2 : Adv loss

T3: L1 loss + Adv loss

```
config_file = {
        "num_epochs": 120,
        "dataset_folder_path":"drive/MyDrive/ProjetS8/data/512/full_body/*",
        "img_size":(512,512),
        "train_batch_size":32,
        "test_batch_size":20,
```

```
        "learning_rate_generator":2e-4,
        "learning_rate_discriminator":2e-5,
        "betas":(0.5, 0.999),
        "path_results":"drive/MyDrive/ProjetS8/results/512/full_body",
        "num_workers":0,
        "input_name_list":["apparel_head_foot","line"],
        "add_discriminator_input_data":True,
        "adv_loss":True,
        "l1_loss":True,
        "mask":False,
}
```

Figure 25: T3: L1 loss + Adv loss

8.2.3 Mask

T1: without mask

```
config_file = {
        "num_epochs": 120,
        "dataset_folder_path":"drive/MyDrive/ProjetS8/data/512/full_body/*",
        "img_size":(512,512),
```

```
        "train_batch_size":32,
        "test_batch_size":20,
        "learning_rate_generator":2e-4,
        "learning_rate_discriminator":2e-5,
        "betas":(0.5, 0.999),
        "path_results":"drive/MyDrive/ProjetS8/results/512/full_body",
        "num_workers":0,
        "input_name_list":["apparel_head_foot","line","segmentation\_skin"] ,
        "add_discriminator_input_data":True,
        "adv_loss":True,
        "l1_loss":True,
        "mask":False,
}
```

Figure 26: T1: without mask

T2: with mask

```
config_file = {
        "num_epochs": 120,
        "dataset_folder_path":"drive/MyDrive/ProjetS8/data/512/full_body/*",
        "img_size":(512,512),
```

```
        "train_batch_size":32,
        "test_batch_size":20,
        "learning_rate_generator":2e-4,
        "learning_rate_discriminator":2e-5,
        "betas":(0.5, 0.999),
        "path_results":"drive/MyDrive/ProjetS8/results/512/full_body",
        "num_workers":0,
        "input_name_list":["apparel_head_foot","line","segmentation\_skin"] ,
        "add_discriminator_input_data":True,
        "adv_loss":True,
        "l1_loss":True,
        "mask":True,
}
```

Figure 27: T2: with mask

8.2.4 Condition on the generator

T1: not conditioned

```
config_file = {
        "num_epochs": 120,
```

```
        "dataset_folder_path":"drive/MyDrive/ProjetS8/data/512/full_body/*",
        "img_size":(512,512),
        "train_batch_size":32,
        "test_batch_size":20,
        "learning_rate_generator":2e-4,
        "learning_rate_discriminator":2e-5,
        "betas":(0.5, 0.999),
        "path_results":"drive/MyDrive/ProjetS8/results/512/full_body",
        "num_workers":0,
        "input_name_list":["apparel_head_foot","line"],
        "add_discriminator_input_data":False,
        "adv_loss":True,
        "l1_loss":True,
        "mask":False,
}
```

Figure 28: T1: not conditioned

T2: conditioned

```
config_file = {
        "num_epochs": 120,
```

```
        "dataset_folder_path":"drive/MyDrive/ProjetS8/data/512/full_body/*",
        "img_size":(512,512),
        "train_batch_size":32,
        "test_batch_size":20,
        "learning_rate_generator":2e-4,
        "learning_rate_discriminator":2e-5,
        "betas":(0.5, 0.999),
        "path_results":"drive/MyDrive/ProjetS8/results/512/full_body",
        "num_workers":0,
        "input_name_list":["apparel_head_foot","line"],
        "add_discriminator_input_data":True,
        "adv_loss":True,
        "l1_loss":True,
        "mask":False,
}
```

Figure 29: T2: conditioned