Software Engineering Department
Braude College

Capstone Project Phase B – 61999

InpaintGAN - Generate image inpainting with adversarial edge learning

23-1-R-13

Students:

Tal Yehoshua 314774860 talyeho@gmail.com

Moshe Peretz 318263977 Moshe.Peretz318@gmail.com

Supervisors:

Dr. Renata Avros

Prof. Zeev Volkovich

TABLE OF CONTENT

ABSTRACT

This paper presents a novel approach to the image inpainting problem using a Generative Adversarial Network (GAN) architecture. In this paper, various edge generator modules have been proposed and examined to achieve the highest precision and recall in correlation to the lowest number of iterations possible.

The method proposed in this paper achieved a precision of 28%, recall of 25% and, feature matching loss of 25% while keeping the number of iterations at bare minimum of $175{,}000$. In contrast, EdgeConnect [1] achieved precision of 27%, recall of 25% and, feature matching loss of 45%.

Our model can be useful for various image editing tasks such as image completion, object removal and image restoration, where fine details are crucial. Our approach addresses the limitation of current state-of-the-art models in producing fine detailed images and opens new possibilities for image inpainting applications.

The book is aimed at researchers, practitioners, and students in the fields of computer vision, image processing, and deep learning who are interested in learning about the latest advancements in image inpainting and how to improve current methods.

Our source code will be available at: https://github.com/MoshPe/InpaintGAN

1. INTRODUCTION

Image inpainting is the process of filling in the missing areas of an image. There are numerous cutting-edge approaches to improving image inpainting. Many of these approaches, however, fail to reconstruct finely detailed images. Many methods and techniques are being proposed to implement image inpainting, however, they failed to repaint reasonable image parts as they are inconsistent to the real image, over-smooth and/or blurry. Moreover, some of the methods require users to manually batch composite images with missing regions themselves.

In phase A of the final project, there was a discussion about the image inpainting being the process of filling in the missing areas of an image. This paper proposed a method called "InpaintGAN" to optimize the generation of the Edge Generation. The edge generator model was improved by adding 3 concurrent max-pooling layers in the middle block in addition to multi-layer ResNet blocks.

By improving the edge generator model, many existing state-of-the-art image inpainting models can be improved with the cause of having more reliable data to use during the inpainting process.

The proposed model can be utilized in various applications, such as restoring or editing photographs, eliminating blemishes or objects from images, and completing missing or damaged parts of an image, catering to professionals such as photographers, graphic designers, and image editing software.

The proposed method include:

Edge generator that processes the mask, the grey scale of the incomplete image, and the edge map of the incomplete image into a predicted edge map with the missing regions. Afterwards, image inpainting is performed using the predicted edge map and the incomplete image.

2. BACKGROUND AND RELATED WORK

2.1. Enhanced gated convolution

Gated convolution

Gated convolution is an image inpainting variation of standard convolutional neural networks (CNNs). It includes a gating mechanism that amplifies relevant traits while reducing irrelevant ones. The gate functions as a switch, altering neuron activation values based on their importance for inpainting. Gated convolution assists the network in focusing on critical details, capturing fine-grained data, and producing realistic inpainted images.

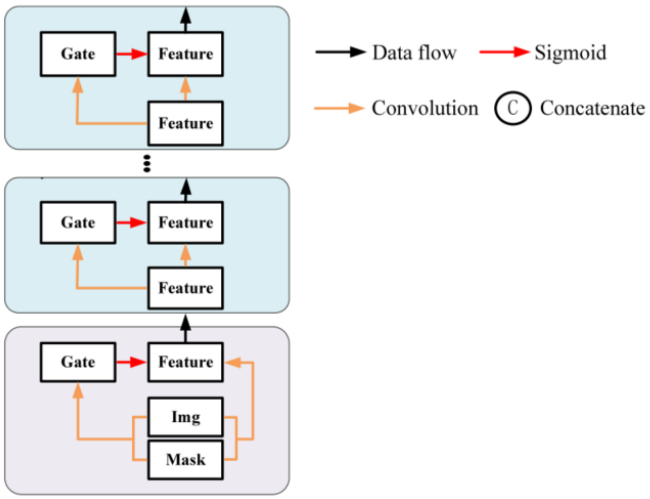


Figure 1: Gated convolution architecture.
Source: [9]

Enhanced gated convolution

Enhanced gated convolution is a more advanced variant of gated convolution that employs extra techniques to improve image inpainting results. The mask information is utilized to improve the gating mechanism in the enhanced gated convolution process. Each layer's gating values are transferred to the next layer, maintaining the gating information of shallow layers. The gating information is improved further by incorporating the current layer's high-dimensional features, the user-specified mask, and the previous layer's gating values.

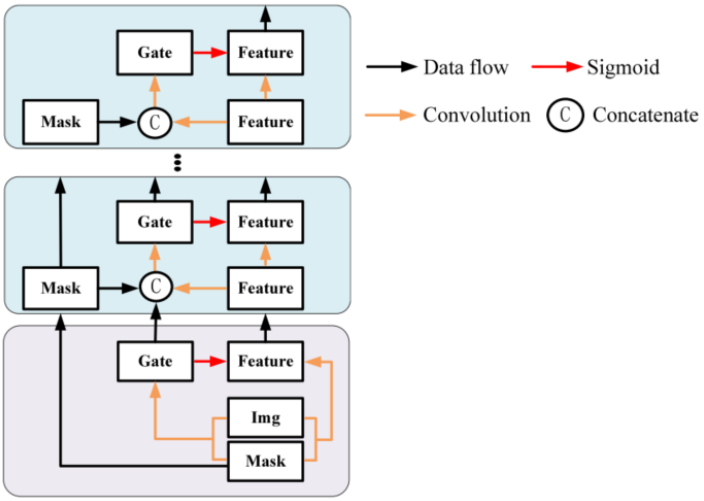


Figure 2: Enhanced gated convolution architecture.
Source: [9]

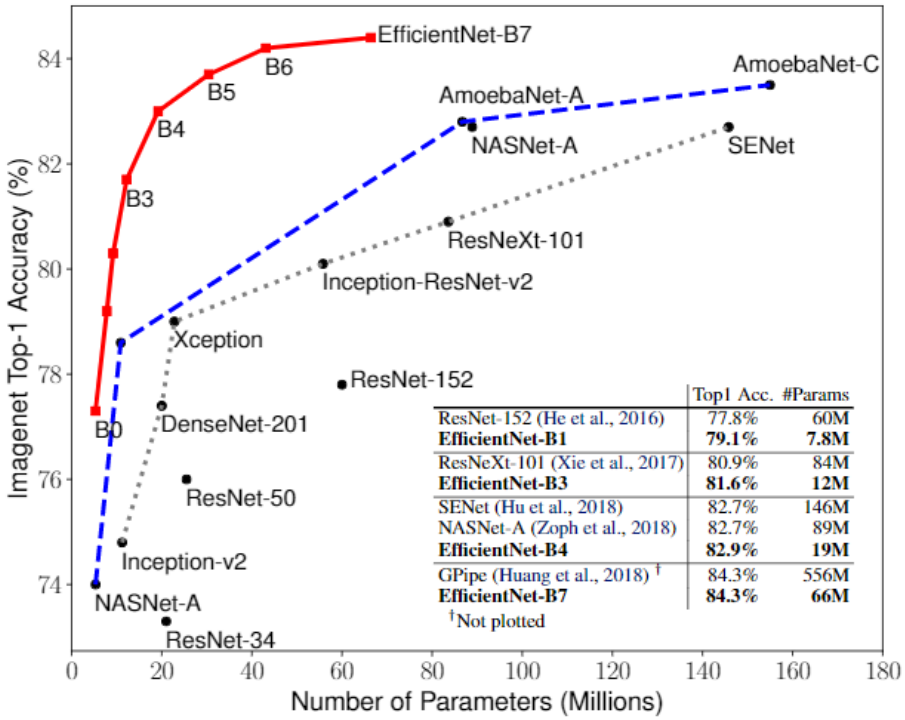
$$
\begin{aligned}
&F_0, Gate_0 = G_{input}([Img, Mask]) \\
&F_i, Gate_i = G_i([F_{i-1}, Gate_{i-1}, Mask]), \\
&Where\ i \in \{1,2,\dots,k\} \\
&and\ k\ denoted\ as\ the\ number\ of\ Enhanced\ Gated\ Convolution\ units \\
&Gate_i = Conv2d\big(\big[\sigma\big(Conv2d(F_{i-1})\big), Gate_{i-1}, Mask\big]\big) \\
&F_i = \phi\big(Conv2d(F_{i-1})\big) \odot \sigma(\mathrm{Gate_i})
\end{aligned}
\tag{1}
$$

2.2. EfficientNet

An alternative to using ResNet with DilatedConv is being researched, inspired by EdgeConnect [1]. When compared to neural networks with simple layers, using ResNet greatly enhanced the performance of neural networks with more layers. Clearly, the difference is massive in networks with 34 layers, where ResNet-34 has a much lower error% than plain-34. Furthermore, the error percentage for plain-18 and ResNet-18 is clearly similar. The more layers there are, the more vanishing gradients there are. To prevent this, residual blocks have been inserted to help increase performance by allowing to develop more complicated and accurate models and to help lessen the vanishing gradient problem. As a result, to scale ResNet, you add more layers. EfficientNet.

As a result, to scale ResNet, you add more layers. EfficientNet manages network scaling and does not require any skip connections. EfficientNet-B1 is 7.6x smaller and 5.7x faster than ResNet-152 .

As a result, EfficientNet is going to be used to provide more consistent and visually appealing outputs than state-of-the-art models. The EfficientNet-B7 achieves ImageNet top-1 accuracy of 84.3%.



	Top1 Acc.	#Params
ResNet-152 (He et al., 2016)	77.8%	60M
EfficientNet-B1	**79.1%**	**7.8M**
ResNeXt-101 (Xie et al., 2017)	80.9%	84M
EfficientNet-B3	**81.6%**	**12M**
SENet (Hu et al., 2018)	82.7%	146M
NASNet-A (Zoph et al., 2018)	82.7%	89M
EfficientNet-B4	**82.9%**	**19M**
GPipe (Huang et al., 2018) †	84.3%	556M
EfficientNet-B7	**84.3%**	**66M**

†Not plotted

Figure 3: comparison between the performance of EfficientNet and other state-of-the-art models.
Source: [2], page 1

There are several approaches to scale up ConvNets. The most popular method is to scale up models by image resolution, however this method produces increasingly better outcomes while increasing the number of parameters.

Compound Scaling Method was developed by Mingxing Tan and Quoc V. Le. Their method scales width, depth, and resolution uniformly using a set of fixed scaling hyperparameters. As the size of the input image grows, the network requires more layers to increase the receptive fields and more channels to catch finer detailed patterns on the larger image.

Unlike all recent ConvNet designs, which often seek the greatest Fi architecture, each layer in ConvNet may be specified as $F_i(X_i) = Y_i$ where Y_i is the output tensor (describing physical properties, just like scalars and vectors), X_i is the input tensor, and F_i is the operator with tensor shape $< H_i, W_i, C_i >$ And each ConvNet can be represented by a list of composite layers.

The most common scaling of the layer architecture is to extend one or more attributes without changing the layer F_i predefined in the basic network.

$$\max_{d,w,r} \ Accuracy\big(\mathcal{N}(d,w,r)\big)$$
$$s.t. \ \ \mathcal{N}(d,w,r) = \bigcup_{i=1\ldots s} \hat{\mathcal{F}}_i^{d\cdot\hat{L}_i}\big(\mathcal{X}_{\langle r\cdot\hat{H}_i,\ r\cdot\hat{W}_i,\ w\cdot\hat{C}_i\rangle}\big) \qquad (2)$$
$$Memory(\mathcal{N}) \leq target\ memory$$
$$FLOPS(\mathcal{N}) \leq target\ flops$$

Source[2]. Page[3]

Where w, d, r are coefficients for scaling network width, depth and resolution.

2.3. Multi-Step Structure Image Inpainting Model with Attention Mechanism

The work presents an improved model for image inpainting with a focus on improving the edge generator. The proposed model employs a two-stage strategy to recover the structure and geometry of damaged images, which is essential for subsequent texture and color filling. A multi-step structure inpainting model is created to improve the effectiveness of first-stage restoration and to address structural reconstruction concerns. During the coarse inpainting stage, the occluded region is divided into areas, and the inpainting order is determined using a priority calculation. The first-stage network completes the inpainting process step by step, prioritizing the restoration of the simpler components at each level. The difficulty of inpainting is reduced by dividing the occlusion area into several fills, resulting in improved overall inpainting quality. A structural attention mechanism is also included to boost structural reconstruction reliability. As inputs, this attention approach uses grayscale and Local Binary Pattern (LBP) images of the damaged image to patch up the gaps with six additional layers of gated convolution. The resulting feature information is sent into the decoder at each scale, increasing the feature representation for later stages. Finally, the fine-stage network feeds the first-stage reconstructed contour image and the original image into an Unet design for color filling, providing the final restored image.

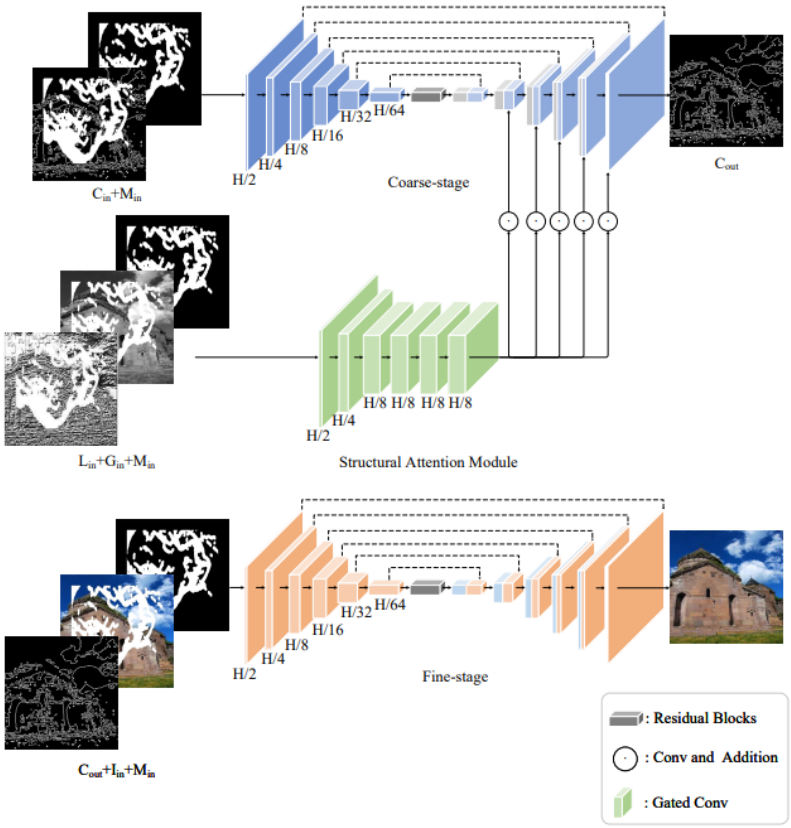


Figure 4: complete model. [3]

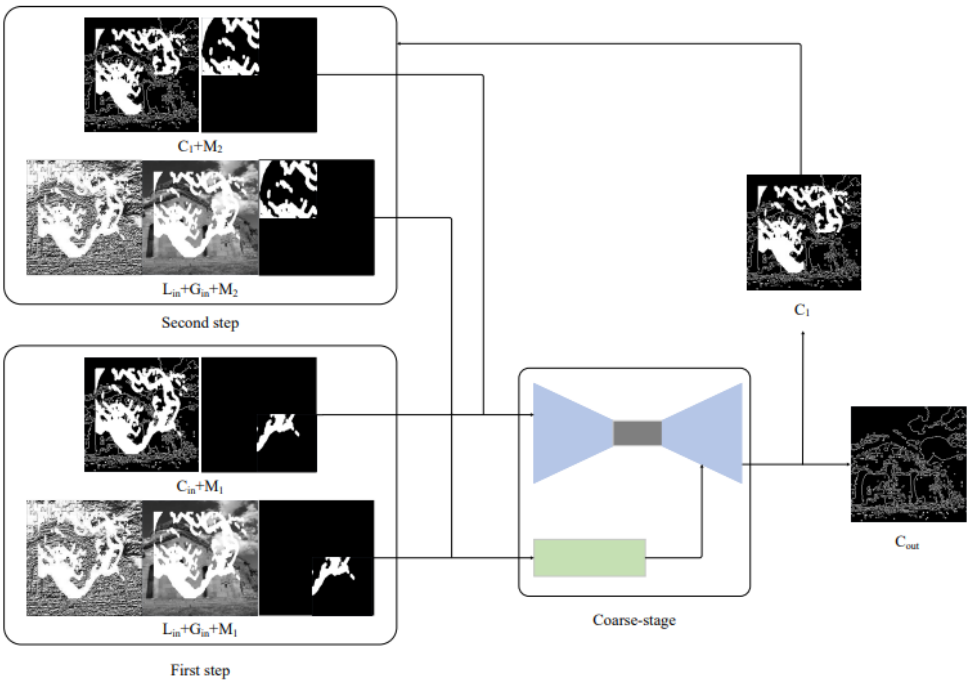


Figure 5: multi-step structure model flowchart. [3]

2.4. Spatial variant reconstruction loss

Prioritizing pixel-level reconstruction loss is crucial in the image inpainting technique. Because missing pixels around hole margins have stronger links with known pixels than those towards the center, the reconstruction loss is determined using a spatial variant reconstruction loss. This approach assigns a higher weight to the finished pixel closest to the boundary, effectively driving the damaged area near the filling boundary to match its neighboring undamaged pixels.

$$
\begin{aligned}
&Q_{weight}^{i} \\
&= \left(g * \tilde{Q}^{i}\right) \odot Mask, Where \\
&\odot \text{ is Hadamard product operator, g is Gaussian filtering} \\
&the\ convolution\ size\ g\ is\ 64 \\
&\qquad \times 64, and\ its\ standard\ deviation\ is\ 40. \\
&\tilde{Q}^{i} = 1 - Mask + Q_{weight}^{i-1} \\
&Q_{weight}^{0} = 0 \\
&Q_{weight} = Q_{weight}^{7}
\end{aligned}
\tag{3}
$$

The Spatial variant reconstruction loss is defined as follows:

$$
Loss_{rec} = \left\| \left(I_{gt} - G([Img, Mask]; \theta)\right) \odot Q_{weight} \right\| \tag{4}
$$

Were I_{gt} is the ground truth image, $G([Img, Mask]; \theta)$ is an intermediate result generated by the generator and θ denotes learnable parameters.

2.5. Multi-scale Edge Completion

Multi-scale edge completion [4] is an image processing approach that fills in missing or damaged edges in an image at several sizes. It's frequently used as a pre-processing step for computer vision applications like picture inpainting, object detection, and segmentation.

The purpose of edge completeness is to rebuild missing or partial edge information using the image's existing context. This can be accomplished by examining the surrounding edge structure and estimating and inferring the missing pieces.

The multi-scale component of edge completion entails taking into account edges at various levels of complexity or granularity. Images have edges at all scales, from fine features to global architecture. The edge completion technique can obtain a more thorough grasp of the image's structure and increase the accuracy of the completed edges by examining edges at several scales.

The process of multi-scale edge completion typically involves the following steps:

1. Edge Extraction: The first stage is to extract image edges using edge detection methods such as Canny, Sobel, or Laplacian of Gaussian. This stage determines the image's existing edges.
2. Scale Decomposition: Using techniques such as Gaussian pyramids or wavelet transforms, the image is decomposed into several scales or levels. Each scale denotes a different level of detail, with larger scales representing global structures and finer ones representing local details.
3. Edge Completion at Each Scale: At each scale, edge completion is performed independently. The system identifies missing or damaged edges and attempts to approximate their appearance based on the available context and surrounding edge information.
4. Edge Completion Integration: The finished edges from each scale are merged to provide the final edge completion result. This integration may entail merging completed edges from different scales, weighing them based on relevance or trustworthiness, and providing a smooth transition across scales.

Edge completion methods that incorporate multi-scale analysis can efficiently handle a variety of circumstances, such as missing edges owing to occlusion, noise, or image deterioration. Combining edge completion with subsequent image processing tasks, such as inpainting, can result in more accurate and visually pleasant outcomes by retaining the image's structural coherence and consistency.

3. Dataset

The dataset that was used in the modules training and testing is Human Faces [5].

A collection of 7.2k images with a mix of all common creeds, age groups and profiles in an attempt to create a unbiased dataset with a few GAN generated images as well to aid the functionality of differentiating between real and generated faces.

4. MODULES

In the modules, the image size that was used is $224 \times 224 \times 3$

4.1. EffU-Net

The first attempt for the Edge Generator implementation is using EfficientNet model and U-Net architecture. The model is implemented as described in Table 1 and 2. Different problems emerged, indicating that the model lacked the capability to inpaint effectively. The utilization of EfficientNet, originally developed for image classification purposes, proved inadequate for the task at hand.

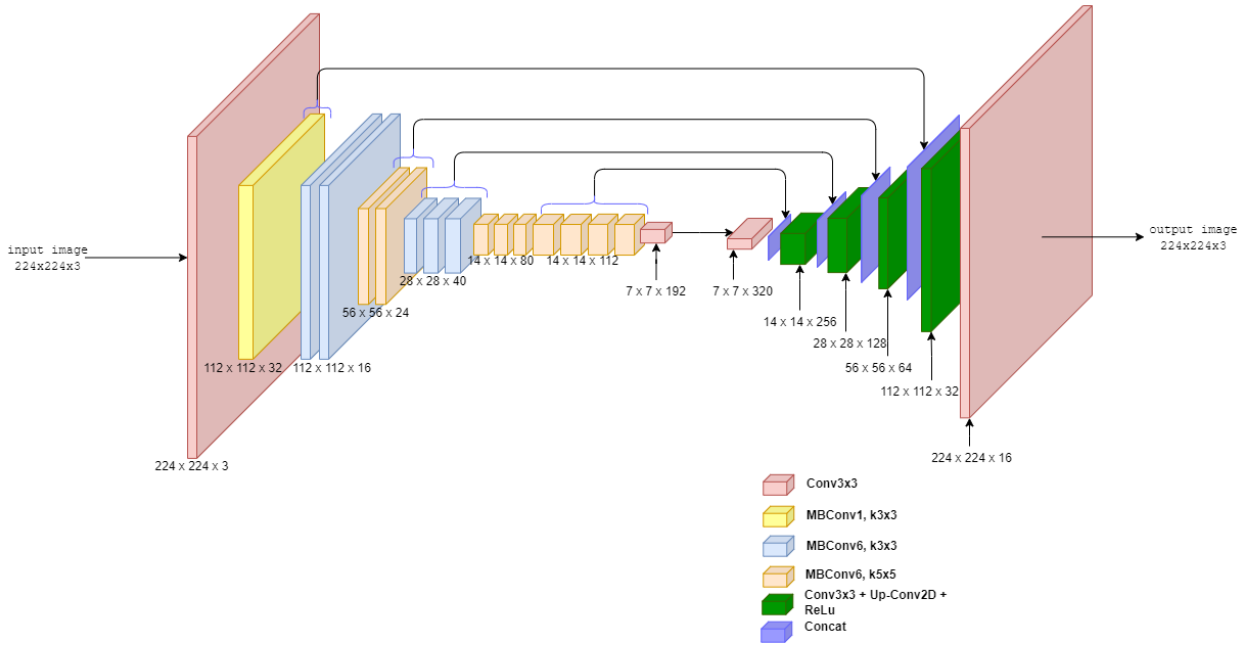


Figure 1: EffU-Net model

Hyperparameters

- $Learning\ rate - 0.001$
- $\frac{Discriminator}{Generator} learning\ rate\ ratio\ - 0.1$
- $Adam\ optimizer\ \beta_1|\beta_2 - 0.0|0.9\ (For\ learning\ rate\ decay)$
- $Batch\ size - 8$
- $Input\ size - 256$
- $Sigma\ (for\ Canny\ Edge\ Detector) - 1.9$
- $Max\ iterations - 2 \cdot 10^6$
- $L_1\ loss\ weight - 1$
- $FM\ loss\ weight - 10$
- $Style\ loss\ weight - 250$
- $Content\ loss\ weight - 0.1$
- $Inpaint\ adverserial\ loss\ weight - 0.1$
- $GAN\ loss - Binary\ cross\ entropy$
- $Edge\ threshold = 0.5$

$\boldsymbol{Stage\ i}$	$\boldsymbol{Operator\ \widehat{F}_i}$	$\boldsymbol{Resolution\ \widehat{H}_i \times \widehat{W}_i}$	$\boldsymbol{\#Channels\ \widehat{C}_i}$	$\boldsymbol{\#Layers\ \widehat{L}_i}$
1	$Conv3 \times 3$	224×224	32	1
2	$MBConv1, k3 \times 3$	112×112	16	1
3	$MBConv6, k3 \times 3$	112×112	24	2
4	$MBConv6, k5 \times 5$	56×56	40	2
5	$MBConv6, k3 \times 3$	28×28	80	3
6	$MBConv6, k5 \times 5$	14×14	112	3
7	$MBConv6, k5 \times 5$	14×14	192	4
8	$Conv3 \times 3$	7×7	320	1
9	$Conv3 \times 3 + Up - conv2D + ReLu$	7×7	256	1

Table 1: EffU-Net encoder

$\boldsymbol{Stage\ i}$	$\boldsymbol{Operator\ \widehat{F}_i}$	$\boldsymbol{Resolution\ \widehat{H}_i \times \widehat{W}_i}$	$\boldsymbol{\#Channels\ \widehat{C}_i}$	$\boldsymbol{\#Layers\ \widehat{L}_i}$
1	$Conv3 \times 3 + Up - conv2D + ReLu$	14×14	128	1
2	$Conv3 \times 3 + Up - conv2D + ReLu$	28×28	64	1
3	$Conv3 \times 3 + Up - conv2D + ReLu$	56×56	32	1
4	$Conv3 \times 3 + Up - conv2D + ReLu$	112×112	16	1
5	$Conv3 \times 3$	224×224	3	1

Table 2: EffU-Net decoder

Results

The recall of the model was observed to be unsatisfactory, while the precision exhibited satisfactory results. This indicates that the model accurately generated certain edges, but failed to generate all the necessary edges. The initial step of the generator involved downsampling the image to a size of $7 \times 7 \times 320$, leading to a loss of features and data. Moreover, the downsampling process employed a 5×5 CNN kernel size, which extracted fewer features compared to a 3×3 kernel size. Additionally, it is believed that the chosen loss function for the model was not suitable.

The Feature Match loss (Figure 3) was calculated as follow:

$real\_features \leftarrow discriminator(ground\_true\_image)$
$fake\_features \leftarrow discriminator(generated\_image)$
$\boldsymbol{for}\ each\ feature\ \boldsymbol{in}\ features$
    $fm\ loss \leftarrow fm\ loss + L_1 loss(real\_feature, fake\_feature)$

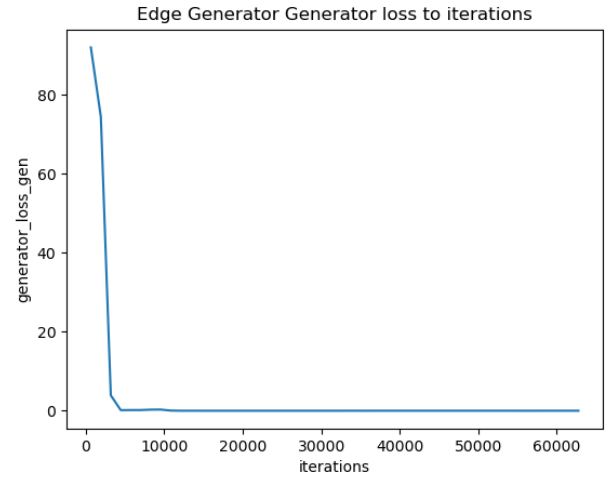


Figure 2: EffU-Net Generator loss

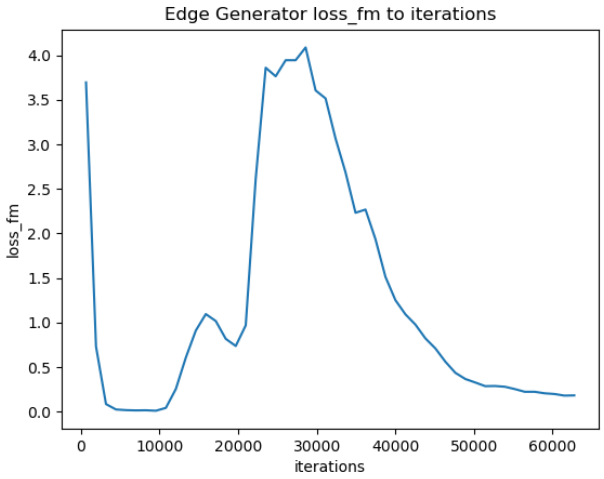


Figure 3: EffU-Net Generator Feature matching loss

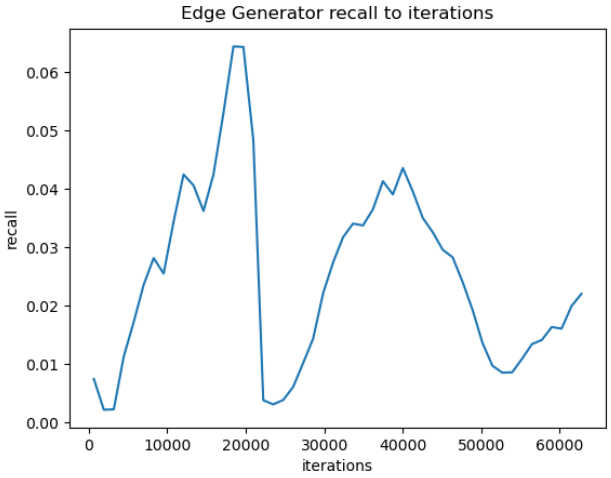


Figure 4: EffU-Net Generator recall

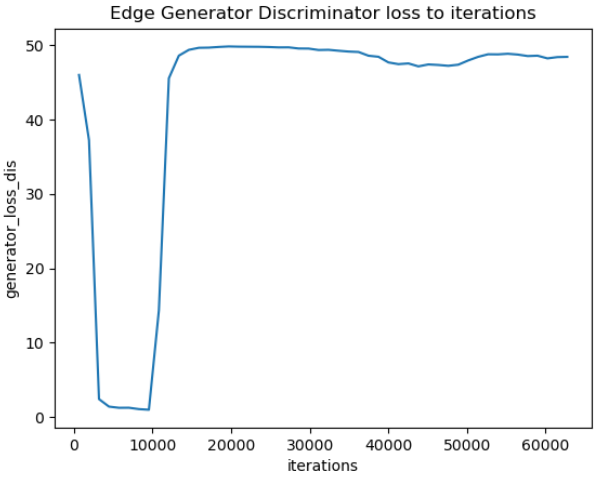


Figure 5: EffU-Net Discriminator loss

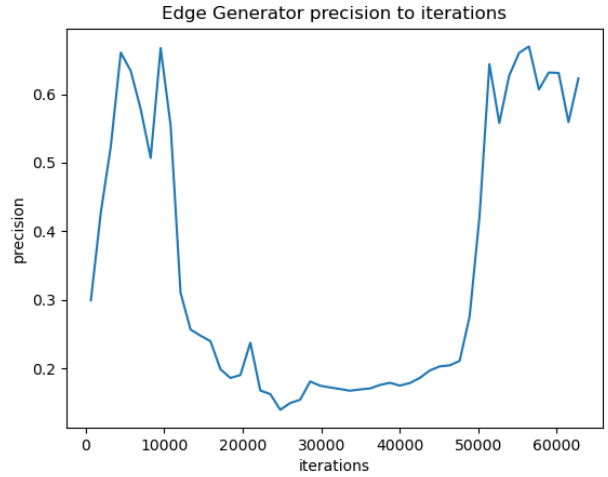


Figure 6: EffU-Net Generator precision

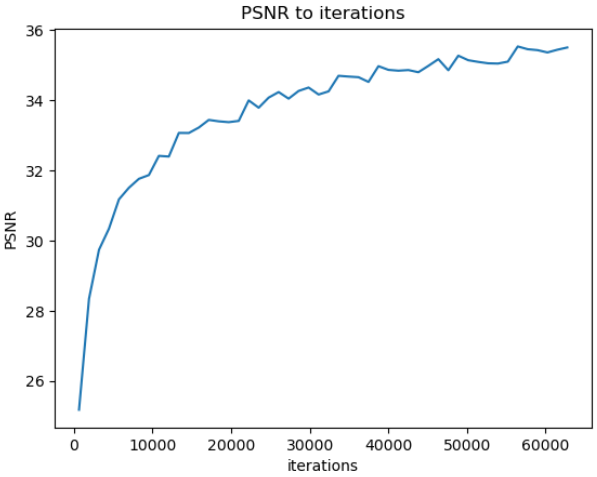


Figure 7: EffU-Net PSNR to iteration

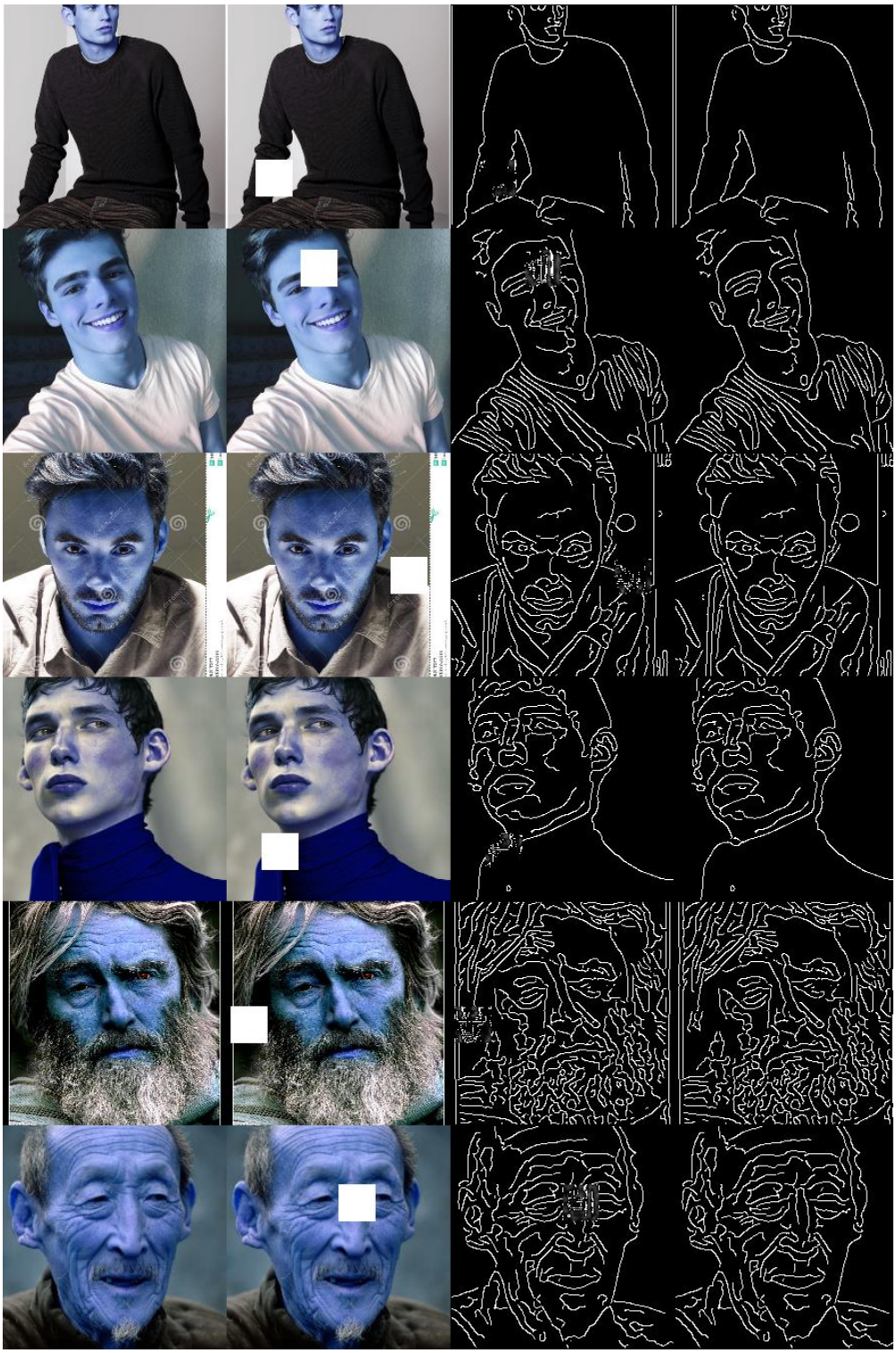


Figure 8: EffU-Net test images – from left to right: ground truth image, ground truth image with the mask, edges image with the generated edges, canny edge detection on the ground truth image.

4.2. Enhanced Gated Convolution model (EGC):

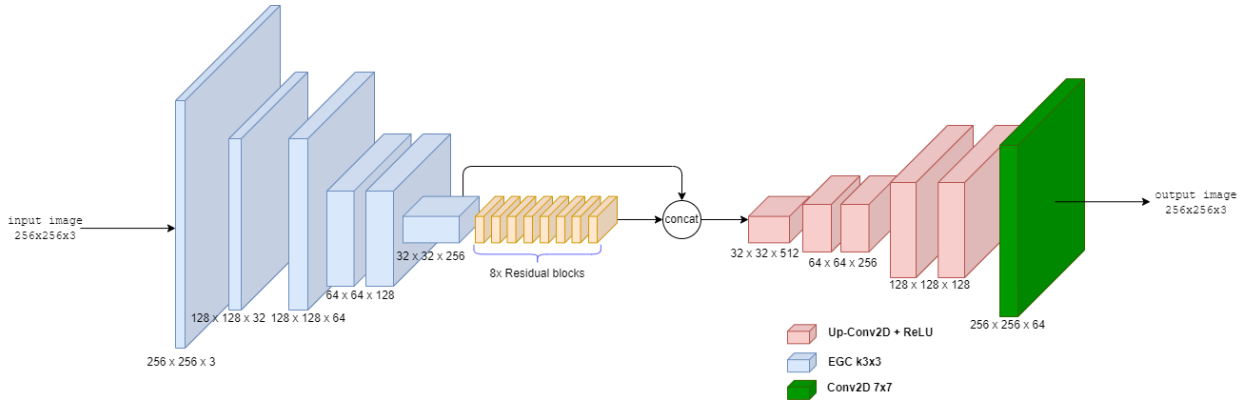


Figure 9: EGC model generator

In the second attempt, Enhanced Gated Convolution was used for the encoder, residual block for the middle section, and regular convolutions for the decoder.

Hyperparameters

- $Learning\ rate - 0.001$
- $\frac{Discriminator}{Generator} learning\ rate\ ratio\ - 0.1$
- $Adam\ optimizer\ \beta_1|\beta_2 - 0.0|0.9\ (For\ learning\ rate\ decay)$
- $Batch\ size - 8$
- $Input\ size - 256$
- $Sigma\ (for\ Canny\ Edge\ Detector) - 1.9$
- $Max\ iterations - 2 \cdot 10^6$
- $L_1\ loss\ weight - 1$
- $FM\ loss\ weight - 10$
- $Style\ loss\ weight - 250$
- $Content\ loss\ weight - 0.1$
- $Inpaint\ adverserial\ loss\ weight - 0.1$
- $GAN\ loss - Binary\ cross\ entropy$
- $Edge\ threshold = 0.5$

$\boldsymbol{Stage\ i}$	$\boldsymbol{Operator: EGC\ \hat{F}_i}$	$\boldsymbol{Resolution\ \hat{H}_i \times \hat{W}_i}$	$\boldsymbol{\#Channels\ \hat{C}_i}$	$\boldsymbol{\#Layers\ \hat{L}_i}$
1	$stride\ 2, k3 \times 3$	256×256	32	1
2	$stride\ 1, k3 \times 3$	128×128	64	1
3	$stride\ 2, k3 \times 3$	128×128	128	1
4	$stride\ 1, k3 \times 3$	64×64	128	2
5	$stride\ 2, k3 \times 3$	64×64 32×32	256	1

Table 3: EGC model encoder

$\boldsymbol{Stage\ i}$	$\boldsymbol{Operator\ \widehat{F}_i}$	$\boldsymbol{Resolution\ \widehat{H}_i \times \widehat{W}_i}$	$\boldsymbol{\#Channels\ \widehat{C}_i}$	$\boldsymbol{\#Layers\ \widehat{L}_i}$
7	$ResidualBlock, k3 \times 3$	32×32	256	8

Table 4: EGC model middle layers - Residual block that composed of 2 convolutions.

$\boldsymbol{Stage\ i}$	$\boldsymbol{Operator\ \widehat{F}_i}$	$\boldsymbol{Resolution\ \widehat{H}_i \times \widehat{W}_i}$	$\boldsymbol{\#Channels\ \widehat{C}_i}$	$\boldsymbol{\#Layers\ \widehat{L}_i}$
1	$Up - conv2D, k4 \times 4$	32×32	256	1
2	$Up - conv2D, k3 \times 3$	64×64	256	1
3	$Up - conv2D, k4 \times 4$	64×64	128	1
4	$Up - conv2D, k3 \times 3$	128×128	128	1
5	$Up - conv2D, k4 \times 4$	128×128	64	1
6	$conv2D, k7 \times 7$	256×256	1	1

Table 5: EGC model decoder

Following the middle section, the output of the encoder is concatenated with the output of the middle section.

Results

The model was performing poorly, the reasons for this may be:

1. Inadequate training duration

2. Inappropriate loss function.

3. Gradient vanishing/exploding

4. Hyperparameter tuning

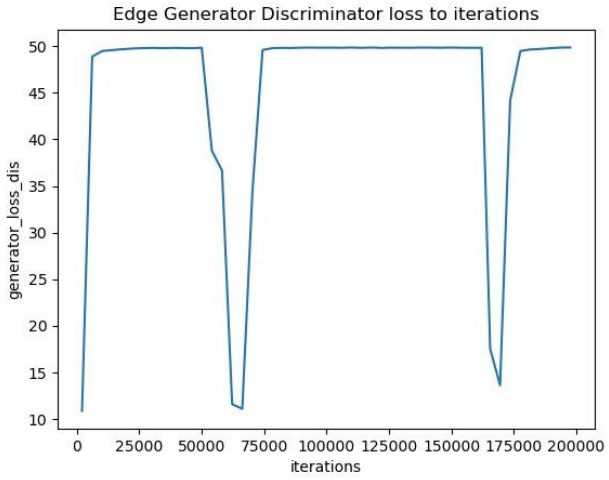


Figure 10: EGC model discriminator loss

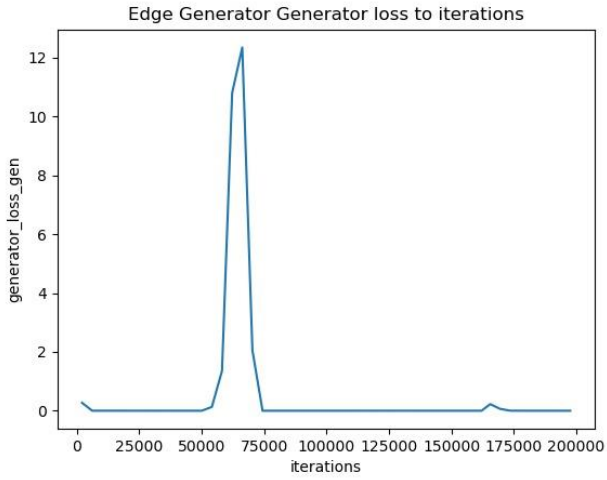


Figure 11: EGC model generator loss

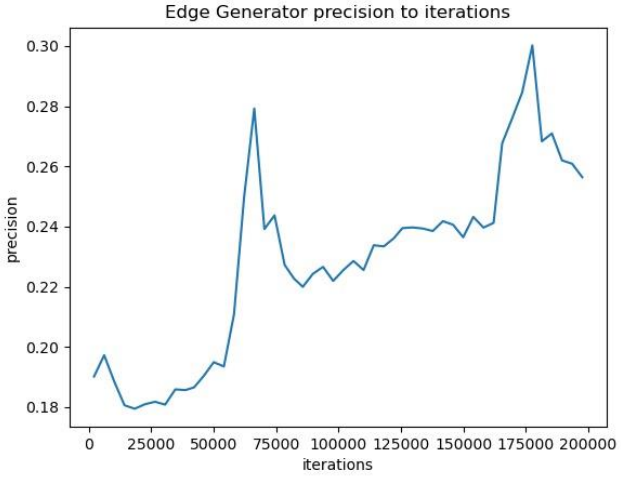


Figure 12: EGC model precision

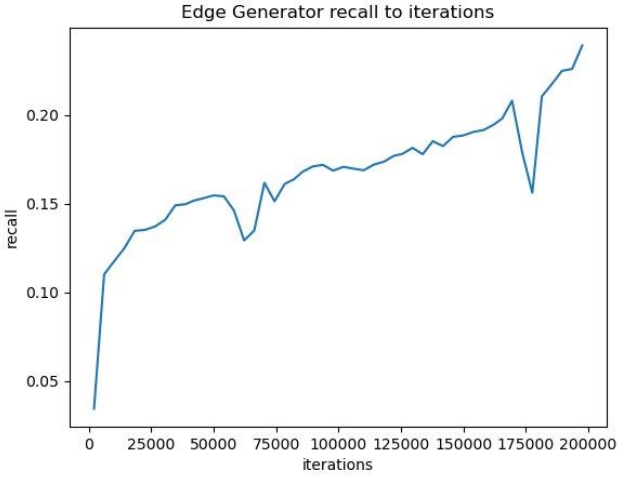


Figure 13: EGC model recall

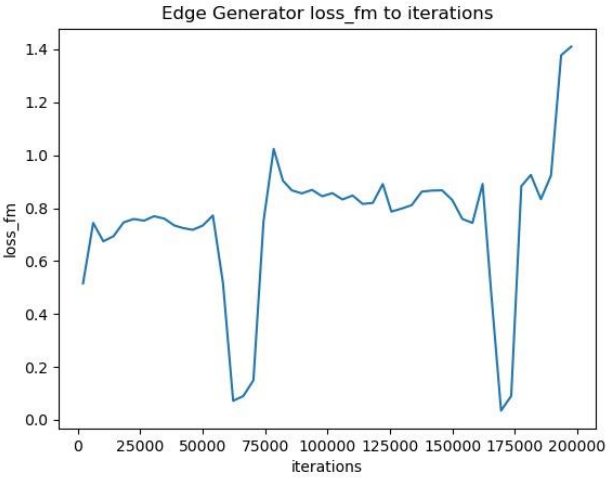


Figure 14: EGC model feature maching loss

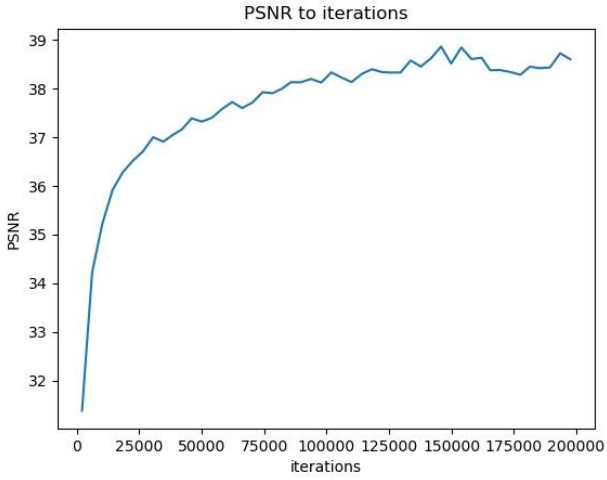


Figure 15: EGC model PSNR

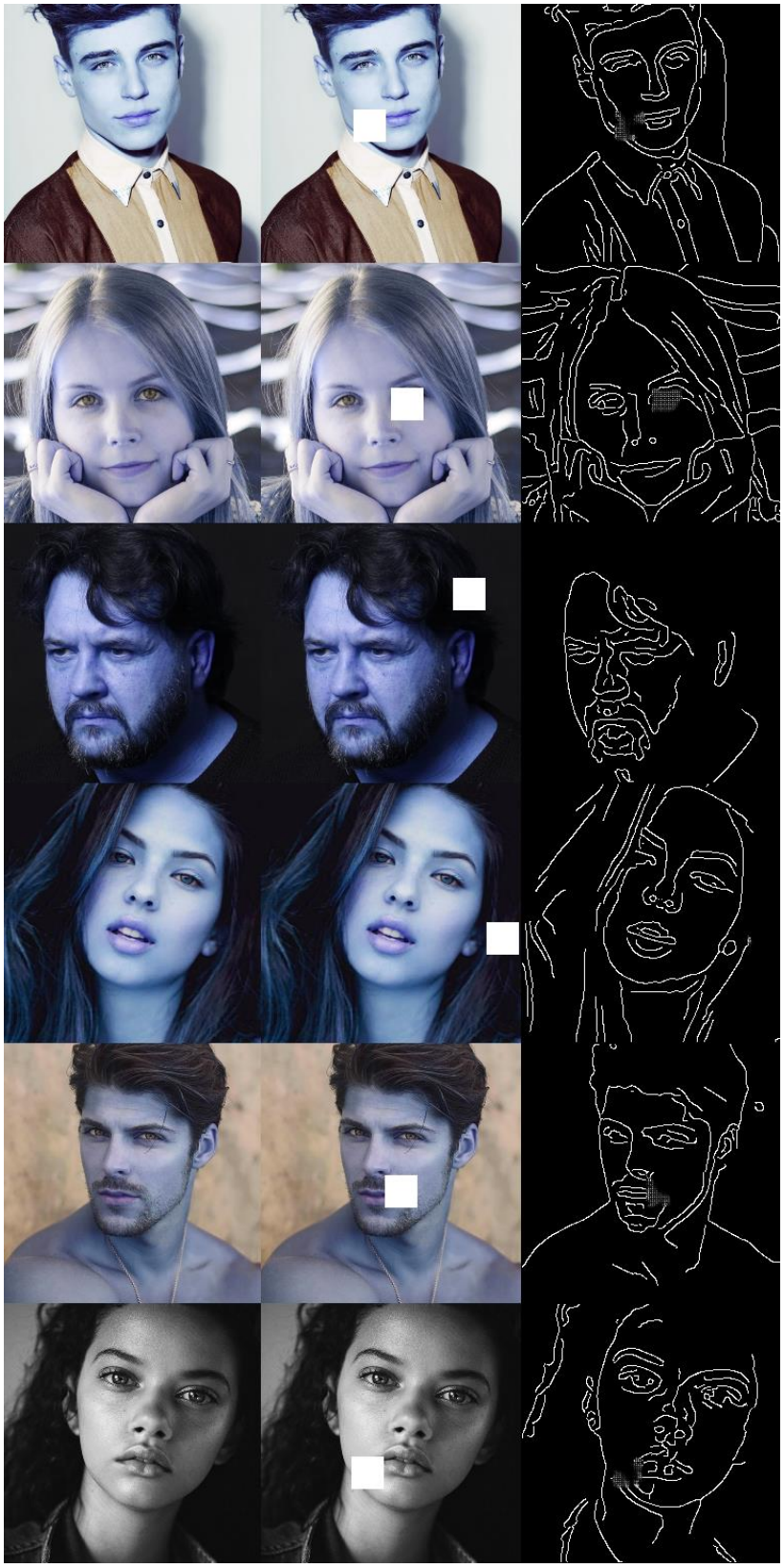


Figure 16: EGC model test images – from left to right: ground truth image, ground truth image with the mask, edges image with the generated edges

4.3. EGC model + Spatial variant reconstruction loss (SVR loss)

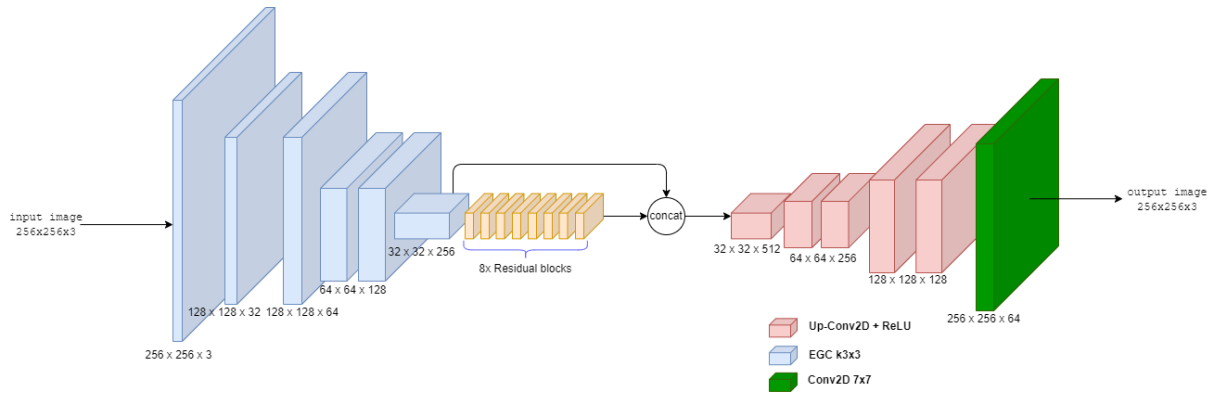


Figure 17: EGC + SVR loss model generator

To see if the loss function play important role in the performance, new loss function was added – SVR loss

Hyperparameters

- $Learning\ rate - 0.001$
- $\frac{Discriminator}{Generator} learning\ rate\ ratio\ - 0.1$
- $Adam\ optimizer\ \beta_1|\beta_2 - 0.0|0.9\ (For\ learning\ rate\ decay)$
- $Batch\ size - 8$
- $Input\ size - 256$
- $Sigma\ (for\ Canny\ Edge\ Detector) - 1.9$
- $Max\ iterations - 2 \cdot 10^6$
- $L_1\ loss\ weight - 1$
- $SVR\ loss\ weight - 1$
- $Style\ loss\ weight - 250$
- $Content\ loss\ weight - 0.1$
- $Inpaint\ adverserial\ loss\ weight - 0.1$
- $GAN\ loss - Binary\ cross\ entropy$
- $Edge\ threshold = 0.5$

$\boldsymbol{Stage\ i}$	$\boldsymbol{Operator: EGC\ \widehat{F}_i}$	$\boldsymbol{Resolution\ \widehat{H}_i \times \widehat{W}_i}$	$\boldsymbol{\#Channels\ \widehat{C}_i}$	$\boldsymbol{\#Layers\ \widehat{L}_i}$
1	$stride\ 2, k3 \times 3$	256×256	32	1
2	$stride\ 1, k3 \times 3$	128×128	64	1
3	$stride\ 2, k3 \times 3$	128×128	128	1
4	$stride\ 1, k3 \times 3$	64×64	128	2
5	$stride\ 2, k3 \times 3$	64×64	256	1
		32×32		

Table 6: EGC + SVD loss model encoder

$\boldsymbol{Stage\ i}$	$\boldsymbol{Operator\ \widehat{F}_i}$	$\boldsymbol{Resolution\ \widehat{H}_i \times \widehat{W}_i}$	$\boldsymbol{\#Channels\ \widehat{C}_i}$	$\boldsymbol{\#Layers\ \widehat{L}_i}$
7	$ResidualBlock, k3 \times 3$	32×32	256	8

Table 7: EGC + SVD loss model middle layers - Residual block that composed of 2 convolutions.

$\boldsymbol{Stage\ i}$	$\boldsymbol{Operator\ \widehat{F}_i}$	$\boldsymbol{Resolution\ \widehat{H}_i \times \widehat{W}_i}$	$\boldsymbol{\#Channels\ \widehat{C}_i}$	$\boldsymbol{\#Layers\ \widehat{L}_i}$
1	$Up - conv2D, k4 \times 4$	32×32	256	1
2	$Up - conv2D, k3 \times 3$	64×64	256	1
3	$Up - conv2D, k4 \times 4$	64×64	128	1
4	$Up - conv2D, k3 \times 3$	128×128	128	1
5	$Up - conv2D, k4 \times 4$	128×128	64	1
6	$conv2D, k7 \times 7$	256×256	1	1

Table 8: EGC + SVD loss model decoder

Results

The model didn’t work at all, due to irrelevance of the loss function. The model calculates the consistency of pixel reconstruction. In an image with 3 channels (RGB) every pixel is related to his neighbors' pixels within a boundary, in contrast to edge map image which there is no correlation between each pixel.

Another reason may be that we didn't implement the loss function correctly.

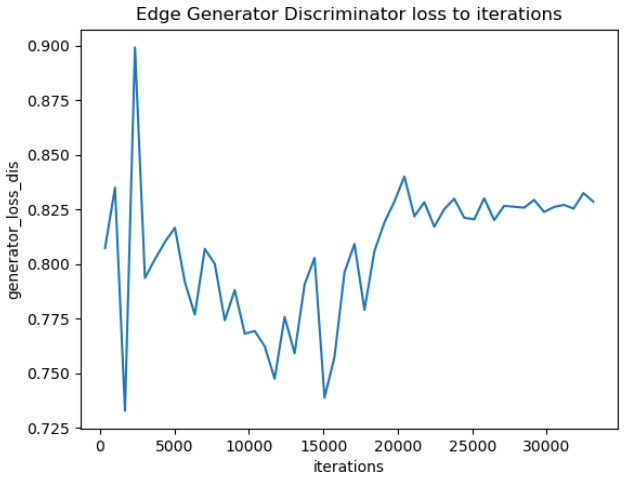


Figure 18: EGC + SVR loss model discriminator loss

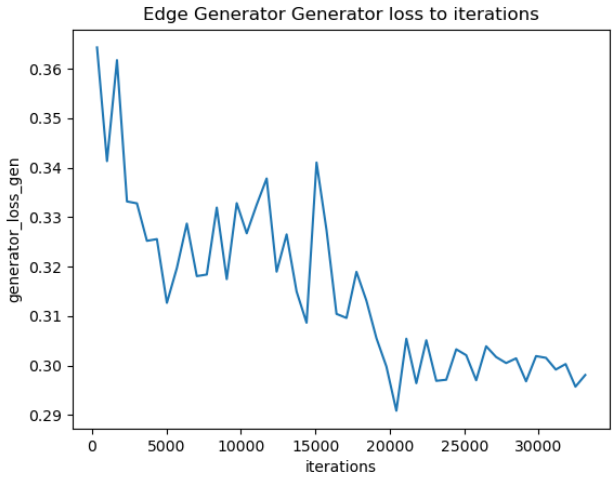


Figure 19: EGC + SVR loss model generator loss

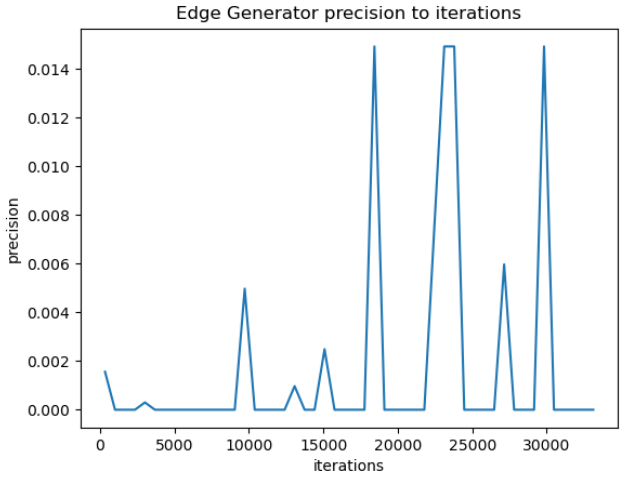


Figure 20: EGC + SVR loss model precision

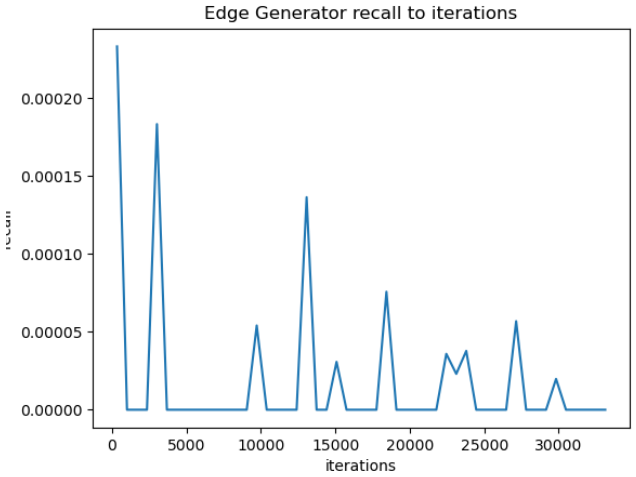


Figure 21: EGC + SVR loss model recall

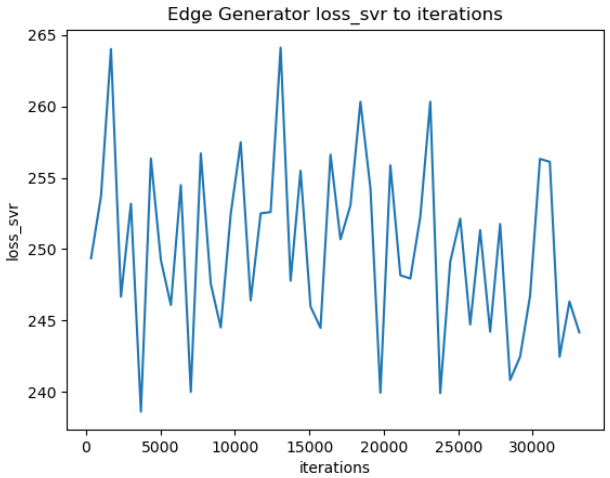


Figure 22: EGC + SVR loss model SVR loss

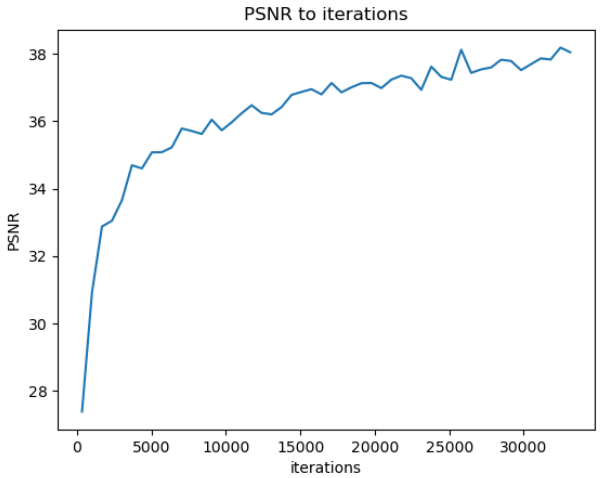


Figure 23: EGC + SVR loss model PSNR

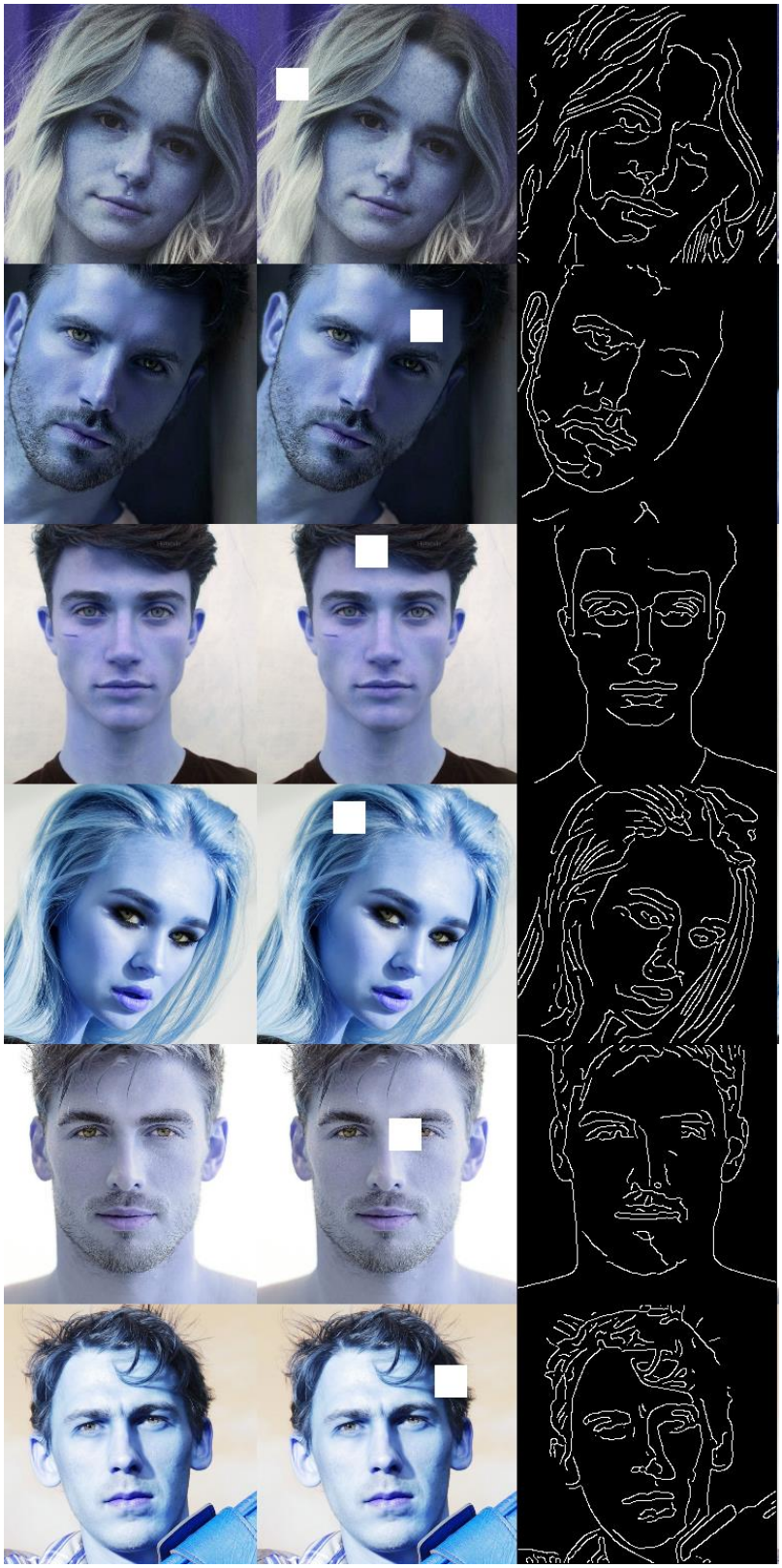


Figure 18: EGC + SVR loss model test images – from left to right: ground truth image, ground truth image with the mask, edges image with the generated edges

5. MODEL

The final model consists of two GAN models, the first of which is the edge generator and the second of which is dedicated to picture inpainting. The edge generator model's generator is identical to the model specified in the final module. The discriminator utilised is the same as that used in EdgeConnect. Similarly, the second GAN model, which is in charge of image inpainting, is the same as the one used in EdgeConnect.

5.1. Multi-scale Edge Completion

To analyse a corrupted picture I with a mask of holes M, Canny edge detector used on I to recover an incomplete edge map E. To gain the benefits of a multi-scale representation, the features pass through two convolution layers before being pooled into many scales, most notably 8, 16, and 32. These scaled features are then individually put into three branches, each with eight Residual Blocks. A deconvolution layer is used to up-sample the lower-scale branch features, which are then merged with the equivalent feature map from the subsequent higher-scale branch to reconstruct the higher-scale features. Finally, three more deconvolution layers are applied to provide the best scale edge completion result.

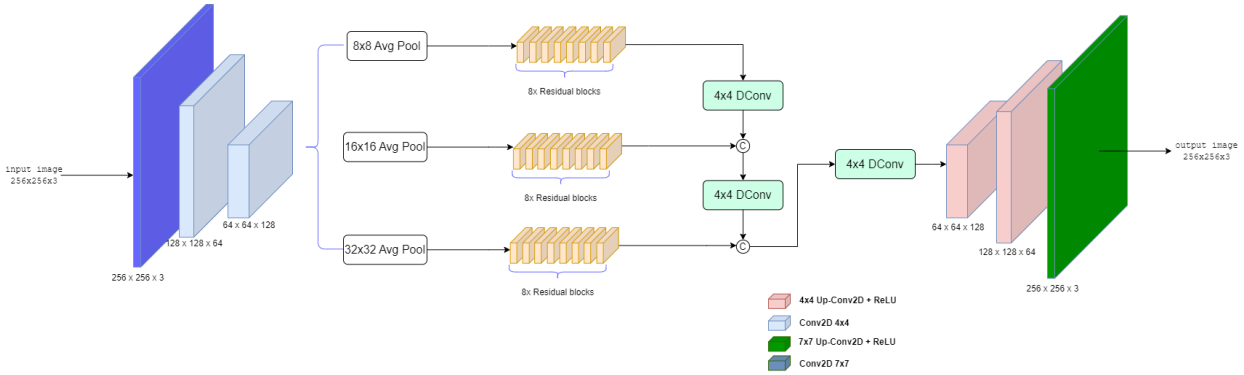


Figure 19: Multi-scale Edge Completion module [4]

The ground truth images are referred to as in the context of the edge generator module I_{gt}. The corresponding edge map is denoted as C_{gt}, while the grayscale version is represented by I_{gray}. For input, a masked grayscale image is used, denoted as $\tilde{I}_{gray}$, which is obtained by element-wise multiplying I_{gray} with the complement of the mask $(1 - M)$. Additionally, the edge map of the ground truth for the masked region, $\tilde{C}_{gt}$, is calculated by element-wise multiplying C_{gt} with the complement of the mask $(1 - M)$. The image mask, denoted as M, serves as a pre-condition, with 1 indicating the missing region and 0 representing the background. The main objective of the generator is to predict the edge map for the masked region.

$$\begin{aligned} \tilde{I}_{gray} &= I_{gray} \odot (1 - M) \\ \tilde{C}_{gt} &= C_{gt} \odot (1 - M) \\ C_{pred} &= G_1(\tilde{I}_{gray}, \tilde{C}_{gt}, M) \end{aligned} \tag{5}$$

The inputs to the discriminator include C_{gt} and C_{pred}, which are conditioned on I_{gray}. The role of the discriminator is to predict the authenticity of an edge map. The network is trained with an objective that combines an adversarial loss and a feature-matching loss:

$$\min_{G_1} \max_{D_1} \mathcal{L}_{G_1} = \min_{G_1} (\lambda_{adv,1} \max_{D_1} (\mathcal{L}_{adv,1}) + \lambda_{FM} \mathcal{L}_{FM}) \quad (6)$$

where $\lambda_{adv,1}$ and λ_{FM} are regularization parameters.

The adversarial loss is defined as:

$$\mathcal{L}_{adv,1} = \mathrm{E}_{C_{gt},I_{gray}}\left[logD_1\left(C_{gt}, I_{gray}\right)\right] + \mathrm{E}_{I_{gray}} \log\left[1 - D_1(C_{pred}, I_{gray})\right] \quad (7)$$

The feature-matching loss, known as LFM, involves comparing the activation maps in the discriminator's intermediate layers. This technique improves the training process's stability by compelling the generator to provide outputs with representations that closely mirror those of real images. The feature matching loss $\mathcal{L}_{FM}$ is defined as:

$$\mathcal{L}_{FM} = \mathrm{E}\left[\sum_{i=1}^{L} \frac{1}{N_i} \left\| D_1^{(i)}(C_{gt}) - D_1^{(i)}(C_{pred}) \right\|\right] \quad (8)$$

In the given equation, L represents the final convolution layer of the discriminator. N_i denotes the number of elements in the i-th activation layer, while $D_1^{(i)}$ represents the activation value in the i-th layer of the discriminator.

5.2. Inpaint generator + Discriminator

The inpaint generator and the discriminator for both GAN models that used in the final model is the same as the modules that was described in the article EdgeConnect. The difference between this model and the one in EdgeConnect is the edge generator model that was optimized.

Using three pooling layers in the center of a neural network architecture has several advantages. They first enable spatial downsampling, which decreases computational complexity and enables the extraction of higher-level abstract information while preserving spatial context. Second, they contribute to translation invariance by summarizing local information and capturing patterns regardless of their actual location. Third, pooling layers aid in the extraction of essential features by emphasizing vital information and eliminating superfluous data. Fourth, they widen the network's receptive field, making it more capable of capturing contextual information. Finally, pooling layers provide hierarchical feature learning, which allows the network to collect features at multiple levels of abstraction. The number and placement of pooling layers may vary depending on the network architecture and job requirements.

5.3. Hyperparameters

- $Learning\ rate - 0.0001$
- $\frac{Discriminator}{Generator} learning\ rate\ ratio\ - 0.1$
- $Adam\ optimizer\ \beta_1|\beta_2 - 0.0|0.9\ (For\ learning\ rate\ decay)$
- $Batch\ size - 8$
- $Input\ size - 256$
- $Sigma\ (for\ Canny\ Edge\ Detector) - 2$
- $Max\ iterations - 2 \cdot 10^6$
- $L_1\ loss\ weight - 1$
- $FM\ loss\ weight - 10$
- $Style\ loss\ weight - 1$
- $Content\ loss\ weight - 1$
- $Inpaint\ adverserial\ loss\ weight - 0.01$
- $GAN\ loss - Binary\ cross\ entropy$
- $Edge\ threshold = 0.5$

Stage $\boldsymbol{i}$	***Operator*** $\boldsymbol{\widehat{F}_i}$	***Resolution*** $\boldsymbol{\widehat{H}_i \times \widehat{W}_i}$	***#Channels*** $\boldsymbol{\widehat{C}_i}$	***#Layers*** $\boldsymbol{\widehat{L}_i}$
1	$stride\ 2, k7 \times 7$	256×256	64	1
2	$stride\ 2, k4 \times 4$	128×128	128	1
3	$stride\ 1, k4 \times 4$	64×64	256	1
		64×64		

Table 9: Multi-scale Edge Completion model encoder

Stage $\boldsymbol{i}$	***Operator*** $\boldsymbol{\widehat{F}_i}$	***Resolution*** $\boldsymbol{\widehat{H}_i \times \widehat{W}_i}$	***#Channels*** $\boldsymbol{\widehat{C}_i}$	***#Layers*** $\boldsymbol{\widehat{L}_i}$
1	$Avg - Pooling, 8 \times 8$	64×64	256	1
	$8\ layers - ResidualBlock, k3 \times 3$	$out: 8 \times 8$		
2	$Avg - Pooling, 16 \times 16$	64×64	256	1
	$8\ layers - ResidualBlock, k3 \times 3$	$out: 16 \times 16$		
3	$Avg - Pooling, 32 \times 32$	64×64	256	1
	$8\ layers - ResidualBlock, k3 \times 3$	$out: 32 \times 32$		

Table 10: Multi-scale Edge Completion model middle layers

Stage $\boldsymbol{i}$	***Operator*** $\boldsymbol{\widehat{F}_i}$	***Resolution*** $\boldsymbol{\widehat{H}_i \times \widehat{W}_i}$	***#Channels*** $\boldsymbol{\widehat{C}_i}$	***#Layers*** $\boldsymbol{\widehat{L}_i}$
1	$Up - conv2D, k4 \times 4$	8×8	256	1
2	$Up - conv2D, k4 \times 4$	16×16	256	1
3	$Up - conv2D, k4 \times 4$	32×32	256	1
4	$Up - conv2D, k4 \times 4$	64×64	128	1
5	$Up - conv2D, k4 \times 4$	128×128	64	1
6	$conv2D, k7 \times 7$	256×256	1	1

Table 11: Multi-scale Edge Completion model decoder

Results

The findings indicate that this model outperformed the recent models. However, it was not possible to train the model beyond 10% of the specified number of iterations mentioned in EdgeConnect.

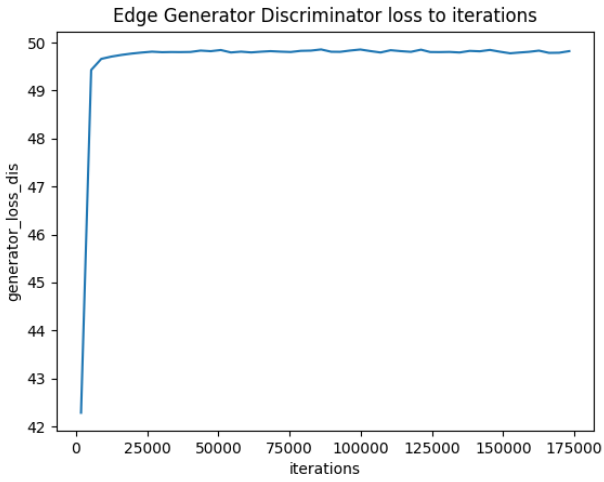


Figure 20: MEC loss model discriminator loss

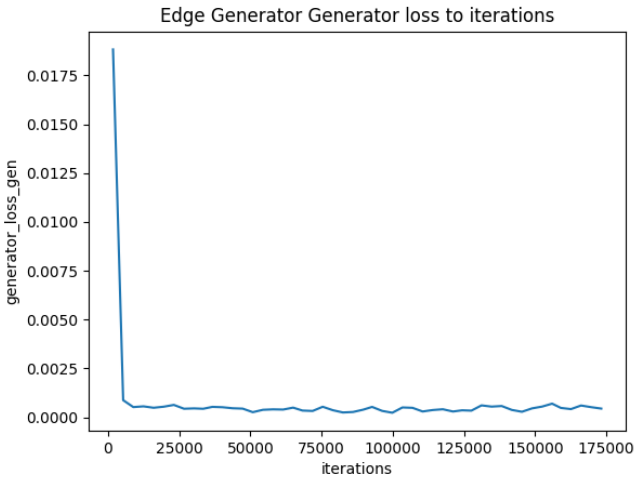


Figure 21: MEC loss model generator loss

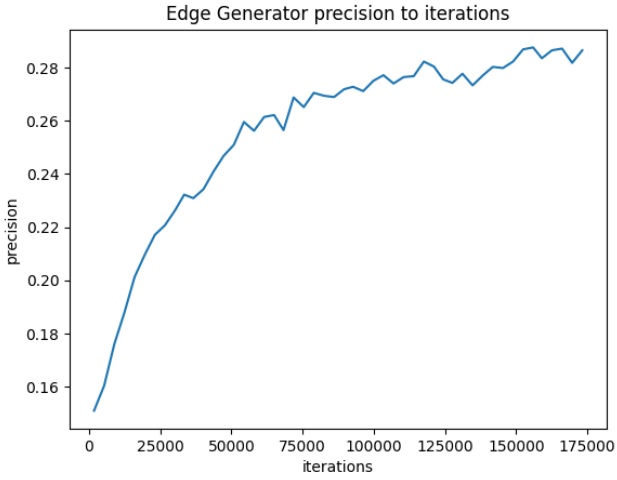


Figure 22: MEC loss model precision

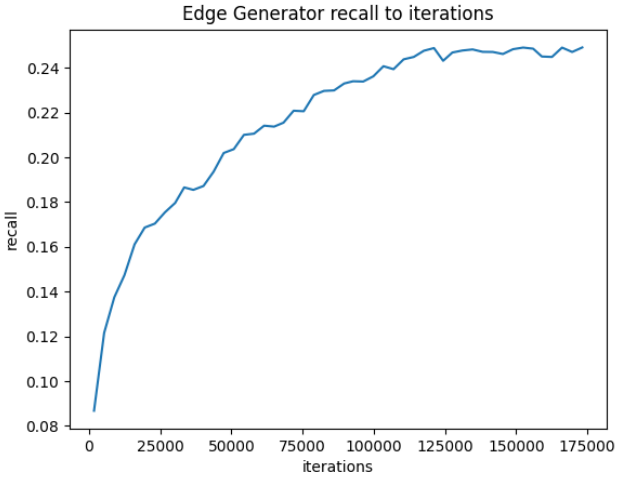


Figure 23: MEC loss model recall

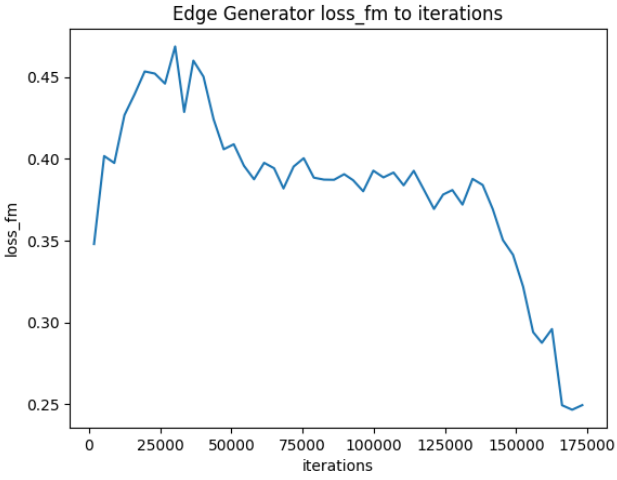


Figure 24: MEC loss model feature matching loss

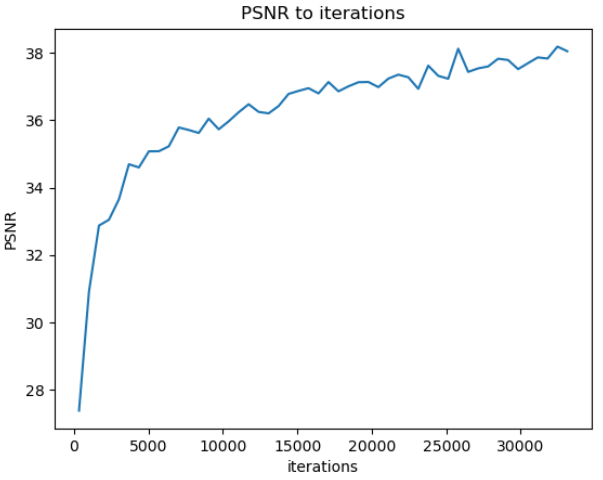


Figure 25: MEC loss model PSNR

Figure 26: MEC model results

6. State-of-the-art comparison

	$FM\ loss$	$L_1\ loss$	$Presicion$	$Recall$	$Perceptual\ loss$	$Style\ loss$	$PSNR$
$EdgeConnect$	0.4099	0.6827	0.2755	0.2387	$\mathbf{0.0309}$	0.0149	38.6038
$InpaintGAN$	$\mathbf{0.2487}$	$\mathbf{0.5707}$	$\mathbf{0.2821}$	$\mathbf{0.2581}$	0.2099	$\mathbf{0.000636}$	$\mathbf{38.954}$

Table 12: Statistics parameters of EdgeConnect and InpaintGAN

7. Research process

To begin the research, numerous methodologies, models, and architectures such as Encoder-Decoder, U-Net, Attention module, and so on must be investigated. To put the various models produced to the test, they had to be implemented and trained, which took a long time. To address this constraint, each model was trained for only 5%-10% of the recommended number of iterations, and the statistical parameters were examined to determine if they were rising upward. This could lead to missing out on the full potential of any model and drawing inaccurate conclusions. Kaggle[6] was used to test the models. There was limited access to GPU and a platform to post the dataset on Kaggle.

8. Testing the models

The focus of the testing approach was on evaluating the output of each component in the image inpainting models to ensure they functioned as expected. Individual components or functions within the models were examined separately. Specific inputs were supplied, and the related outputs were assessed to ensure that each component performed as expected. This method assisted in identifying disparities or faults at a granular level, allowing for faster debugging and improvement. Interactions and collaborations between the models' various components or modules were also tested. The evaluation of data flow and connectivity between these pieces ensured seamless integration and smooth functioning. Any difficulties that arose as a result of their combination were identified and rectified. To verify the correct integration of the model, it was necessary to check the changes in statistical parameters precision and recall during the run.

8.1. Precision

$$Precision = \frac{true\ positives}{true\ positives + false\ positives} \tag{9}$$

Precision is the proportion of true positive findings among all pixels projected to be positive. True Positives are pixels that were forecasted as positive but were actually negative (incorrectly inpainted). False Positives are pixels that were projected as positive but were actually negative (incorrectly inpainted). Precision is measured on a scale of 0 to 1, with a greater score indicating a higher proportion of right predictions.

8.2. Recall

$$Recall = \frac{ture\ positives}{true\ positives + false\ negatives} \quad (10)$$

Recall, also known as sensitivity or true positive rate, indicates the algorithm's ability to accurately identify and inpaint the image's positive pixels. True Positives are pixels that were accurately anticipated as positive (inpainted correctly), while False Negatives are pixels that were wrongly projected as negative (inpainting missed). The fraction of correctly inpainted pixels out of all pixels that should have been inpainted is measured by recall.

8.3. GUI Testing

Test	Module	Test Description	Expected Result
1	Main window	Click 'Home' button.	Open 'Home' window.
2	Main window	Click 'Train' button.	Open 'Train' window.
3	Main window	Click 'Inference' button.	Open 'Inference' window.
4	Train window -> InpaintGAN configuration	Click Drop-down 'Model to train List' and choose 'Edge Model'.	Option 'Edge Model' should be chosen.
5	Train window -> InpaintGAN configuration	Click Drop-down 'Model to train List' and choose 'Inpaint Model.	Option 'Inpaint Model' should be chosen.
6	Train window -> InpaintGAN configuration	Click Drop-down 'Model to train List' and choose 'Joint Model'.	Option 'Joint Model' should be chosen.
7	Train window -> InpaintGAN configuration	Click Drop-down 'Masking image' and choose 'Random block'.	Option 'Random block' should be chosen.
8	Train window -> InpaintGAN configuration	Click Drop-down 'Edge Detector' and choose 'Canny'.	Option 'Canny' should be chosen.
9	Train window -> Dataset folder path	Click 'Select Dataset Path' and select the dataset folder.	The path should be written below the button
10	Train window -> Dataset folder path	Click 'next' without selecting a folder.	Error message saying 'Dataset path is not defined' should be written in Red below the path.
11	Train window -> Training Process	Click on button 'next'.	Open training configuration window.
12	Train window -> Generator Discriminator configuration	Click Drop-down 'Learning Rate' and choose '0.0001'.	Option '0.0001' should be chosen.

13	Train window -> Generator Discriminator configuration	Click on button 'Train The Model'.	Open active training with timer window.
14	Inference window -> upload images	Click on 'Upload Image'.	Open the file explorer to select the image.
15	Inference window -> upload images	Selected an image. Click on 'Upload Image' and select an image.	New file selected should override the previously selected image
16	Inference window -> upload images	Click on button 'Mask Image'.	Open Inference window -> mask image.
17	Inference window -> mask images	Mask an image using the eraser.	The image erased parts should be white only without any image underneath.
18	Inference window -> mask images	Mask an image using the cursor.	The image should be saved after erasing the parts.
19	Inference window -> mask images	Select a different 'Line width' 7	The cursor drawing line should be to the selected width 7
20	Inference window -> mask images	Click 'Clear Area'	Should clear all drawn white lines.
21	Inference window -> mask images	Click on button 'Fill missing regions'.	Should open 'Filling image missing regions window.
22	Inference window -> Fill missing regions	Click on button 'Save image'.	Should open file explorer to save the image.

Table 13: GUI testing plan

9. Results and conclusions

Given the training time constraints, the final model has the potential to surpass EdgeConnect, as demonstrated by the state-of-the-art comparison. If more resources were available, completing all of the suggested iterations could result in a superior model to EdgeConnect.

When evaluating the other models, elements of the edge generator module were adjusted based on how the model generated the edges. If the model generated edges in the correct place but with poor precision, it suggested a problem with low-level feature extraction in the encoder/decoder. This is due to the fact that low-level feature extraction uses a smaller set of pixels than deep-level feature extraction, resulting in a lower image resolution but a greater number of channels.

In retrospect, the utilization of resources would be the only aspect to change. Renting resources to remove limitations on GPU usage would have likely resulted in a significantly better outcome compared to the constraints faced. Additionally, a clearer understanding of the performance of the other outlined models during their development would have been beneficial.

10. User Documentation

10.1 General Description

The inpaintGAN GUI is intended to provide a comprehensive interface for an inpainting model that this paper proposes. Training section that allows user to configure the model's parameters and select the desired dataset for training. The GUI will present the live training of the model, including all the metrics of the model per iteration. Additionally, the GUI includes an Inference tab, enabling user to utilize a pretrained model for testing purposes. Within the Inference tab, users can upload an image, leverage the GUI's canvas to create a mask by drawing on the image, and them execute the model to fill in the missing regions in the masked image, generating a visually complete output.

10.2 User Instructions

Instructions

Simply clone the project from Github repo, InpaintGAN[7].

```
git clone https://github.com/MoshPe/InpaintGAN.git
cd InpaintGAN
```

Install python dependencies

```
pip install -r requirements.txt
```

Dataset

This paper mode was trained with HumanFaces[5] dataset. To train the model on the dataset please download the data from Kaggle website with your registered user.

⚠ The dataset must contain only images and placed with an English path only! (The model can't recognize any other languages in the path)

Getting Started

Simply run `python main.py` to open Eel home page. See Figure[27].

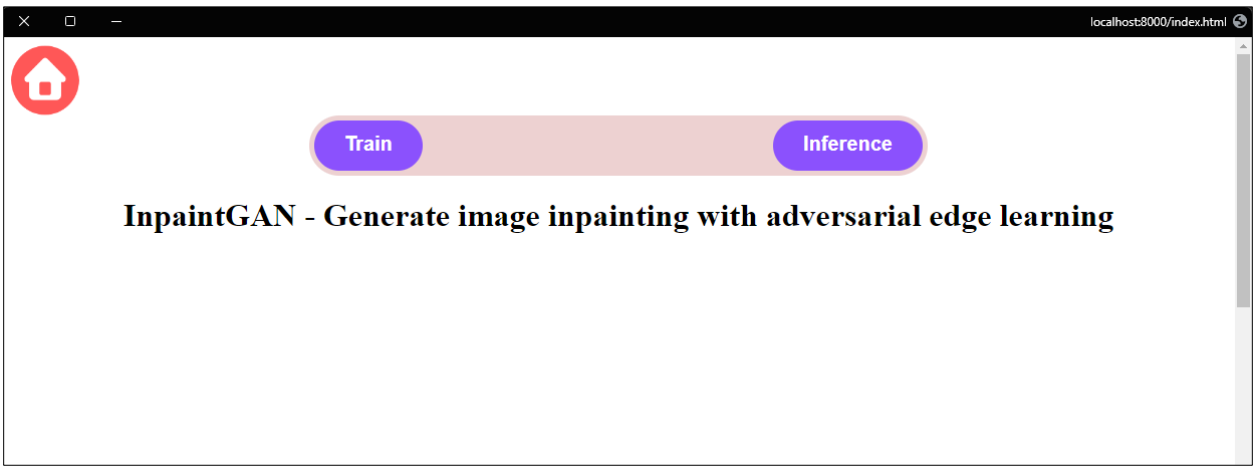


Figure 27: InpaintGAN Home page

On the home page, the user can choose to open either Train or Inference by pressing the buttons. In addition, in any time the user can return to the home page by pressing the Home button at the top left corner of the window. See Figure[28].

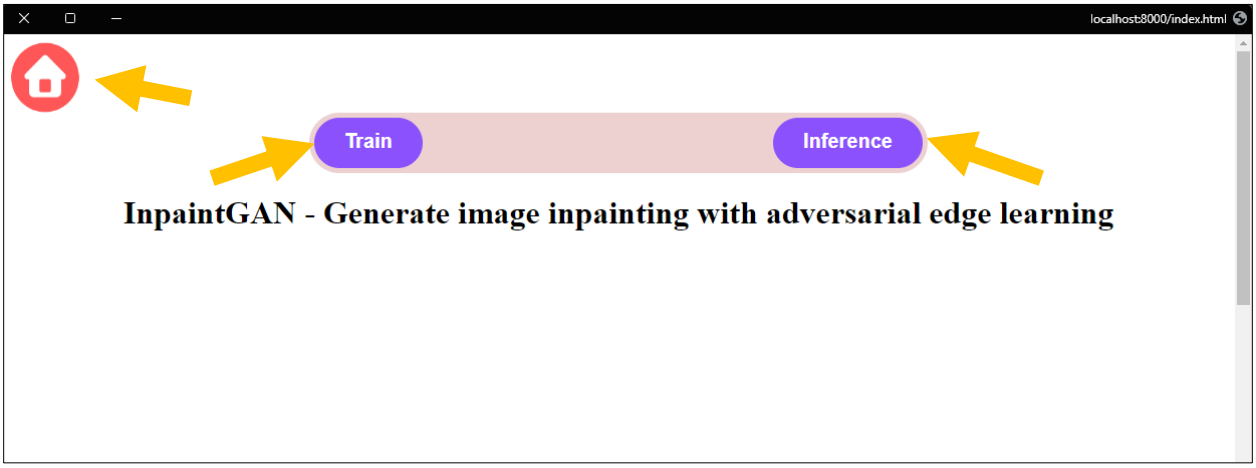


Figure 28: InpaintGAN Home page buttons. 'Home' button, 'Train' button and 'Inference' button.

Training

Pressing "Train" button will open the train window, where the user can configure the training parameters for both generator and discriminator and the dataset folder paths. This is made to make training the model accessible and easy to configure and operate. See figure[29].

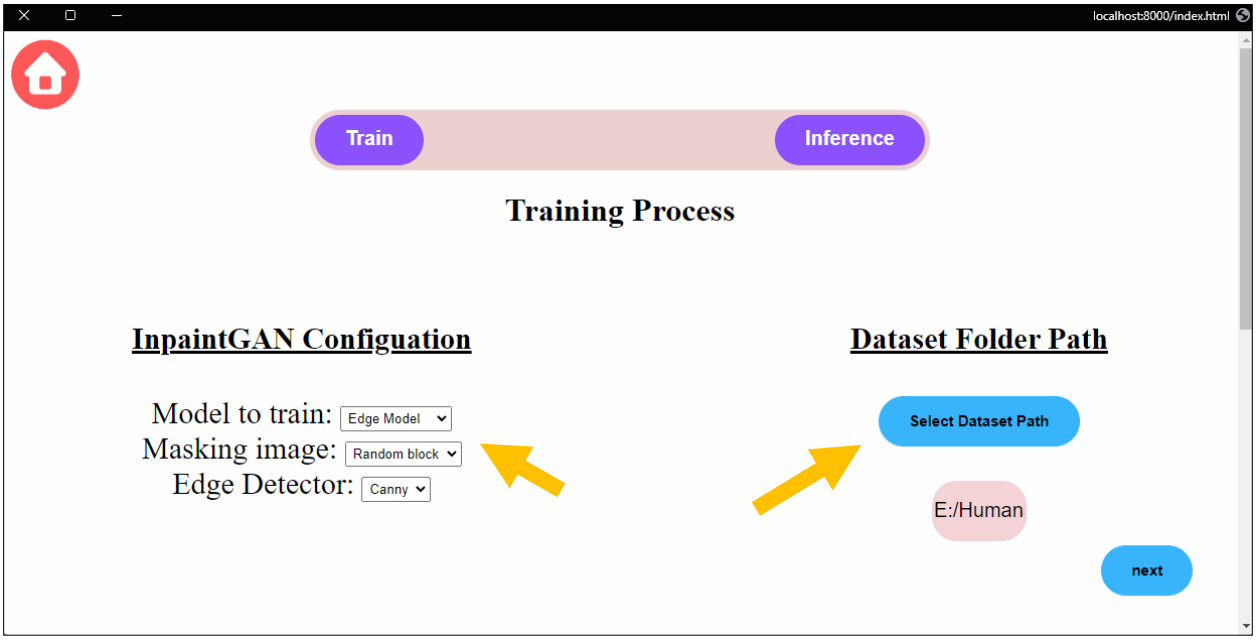


Figure 29: User has to select which model to train and providing the dataset path

In the configuration of generator and discriminator this paper has utilized EdgeConnect[1] GAN configuration. See table[9].

Option	**Description**
LR	learning rate
D2G_LR	discriminator/generator learning rate ratio
BETA1	adam optimizer beta1
BETA2	adam optimizer beta2
BATCH_SIZE	input batch size
INPUT_SIZE	input image size for training.
SIGMA	standard deviation of the Gaussian filter used in Canny edge detector
MAX_ITERS	maximum number of iterations to train the model
EDGE_THRESHOLD	edge detection threshold
L1_LOSS_WEIGHT	l1 loss weight
FM_LOSS_WEIGHT	feature-matching loss weight
STYLE_LOSS_WEIGHT	style loss weight
CONTENT_LOSS_WEIGHT	perceptual loss weight
INPAINT_ADV_LOSS_WEIGHT	adversarial loss weight
GAN_LOSS	**nsgan**: non-saturating gan, **lsgan**: least squares GAN, **hinge**: hinge loss GAN

Table 14: Presents the GAN configuration hyperparameters. Employed from EdgeConnect[1] paper.

Pressing 'Train the model' will configure the InpaintGAN model and run the training. See figure[30].

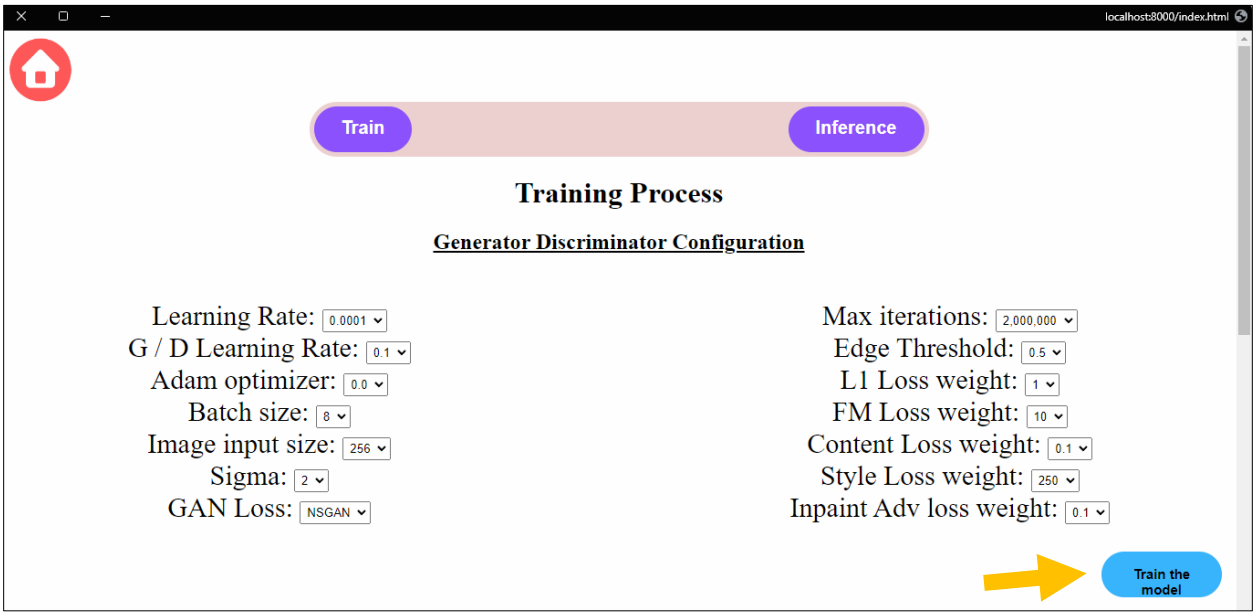


Figure 30: Pressing 'Train the model' button for executing the training process.

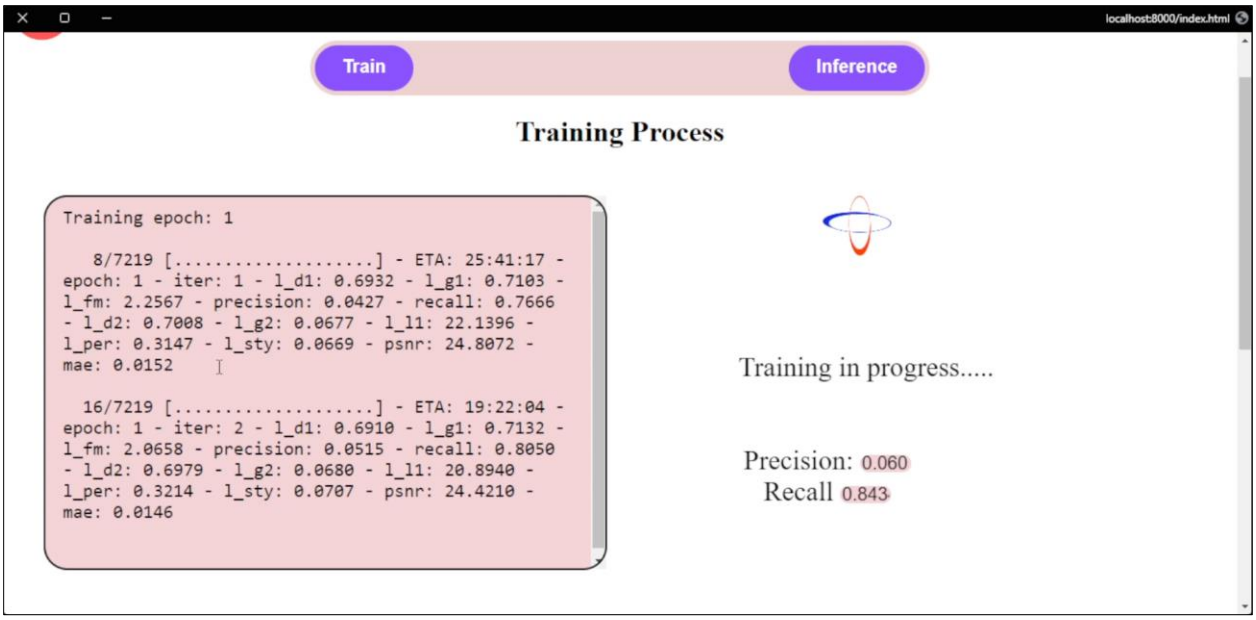


Figure 31: The InpaintGAN training process on the given dataset with metrics displaying

⚠ The model while training cannot be interfered. In order to navigate to the other parts of the application a restart is required or waiting till the end of training.

Inference

Pressing on the 'Inference' button will open the model testing section in which the user upload an image, create a mask on the image using an on board canvas, and utilizing the model weights filling the missing areas of the image. See figure 32.

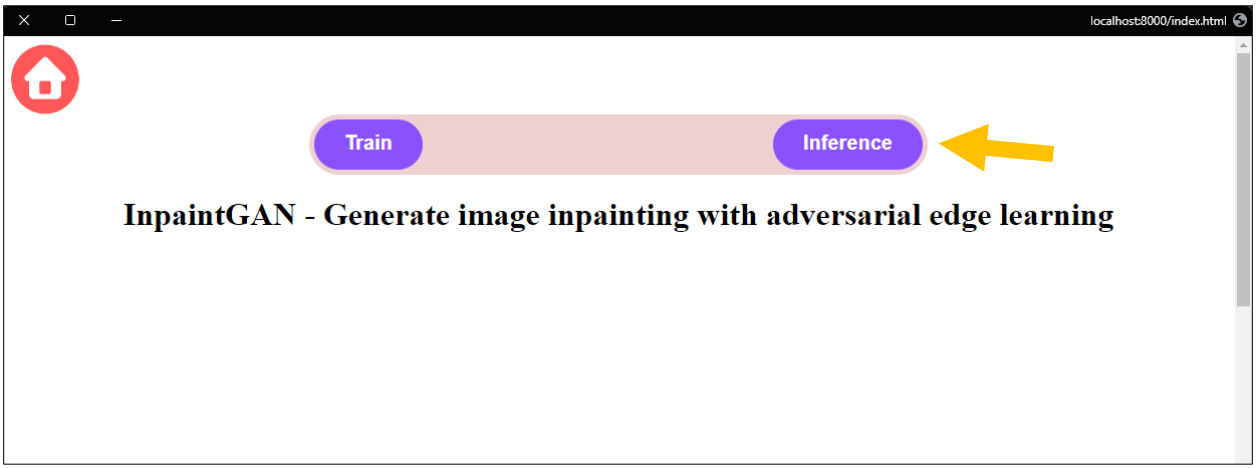


Figure 32: Pressing 'Inference' button in Home Page.

Pressing 'Upload image' will open a file selection window for the user to select an image to upload. Pressing 'Mask Image' will open the Mask Image tab for drawing the mask on the image. See figure[33, 34].

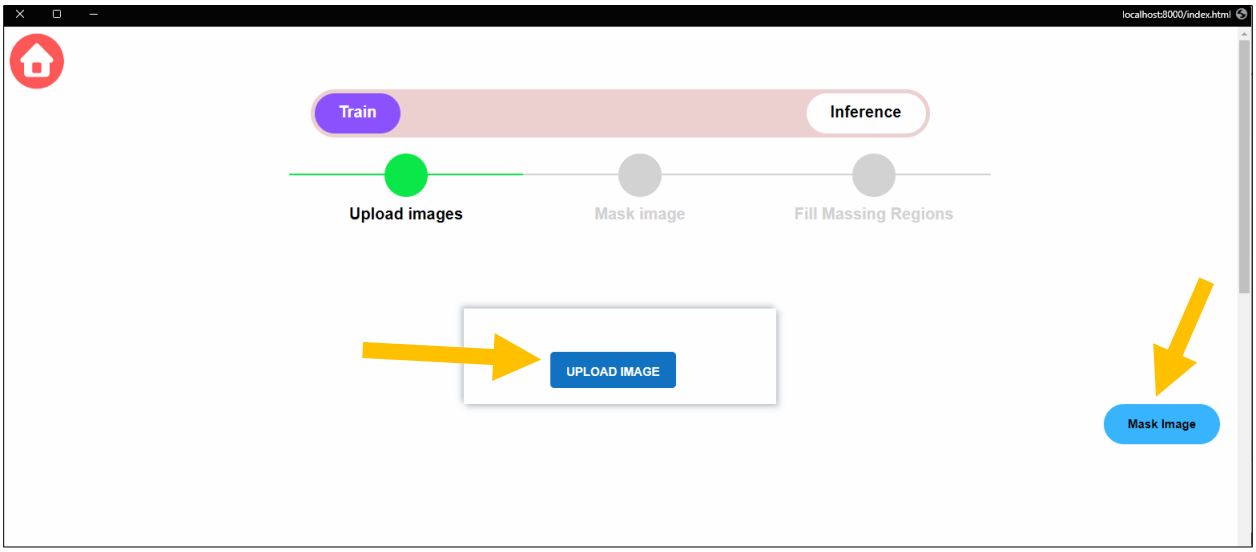


Figure 33: Pressing ‘Upload image’ button will open a file selection window. Pressing ‘Mask Image’ will open the Mask Image tab for creating the mask.

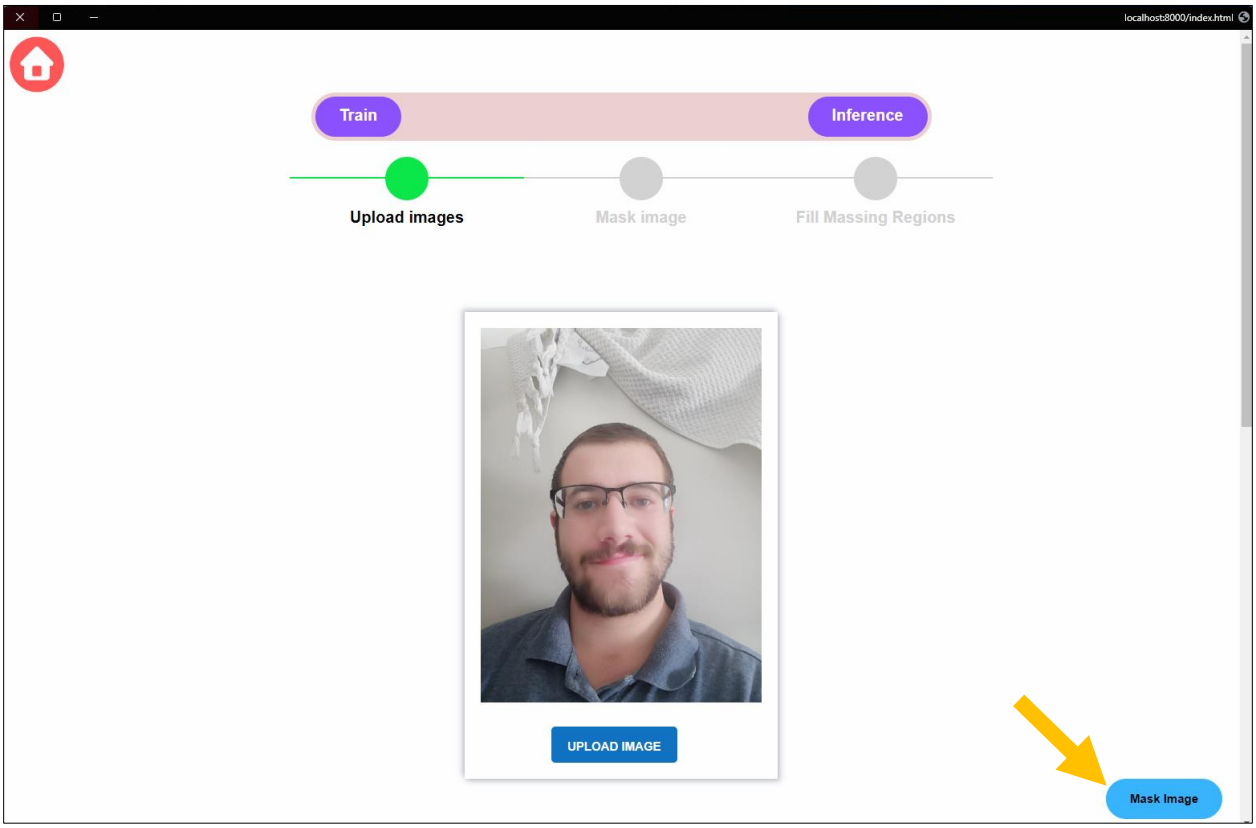


Figure 34: Selection of an image by a user.

In the masking section the user is able to draw a white line, creating a mask upon the selected image. The user can alter the size of drawn line or clear the drawn line. After creating a mask, pressing ‘Fill Missing Regions’ will execute InpaintGAN model in testing mode on the created mask and the selected image. See figure [35].

⚠ The model was trained upon a square mask so different kind of drawing might won't return the expected results.

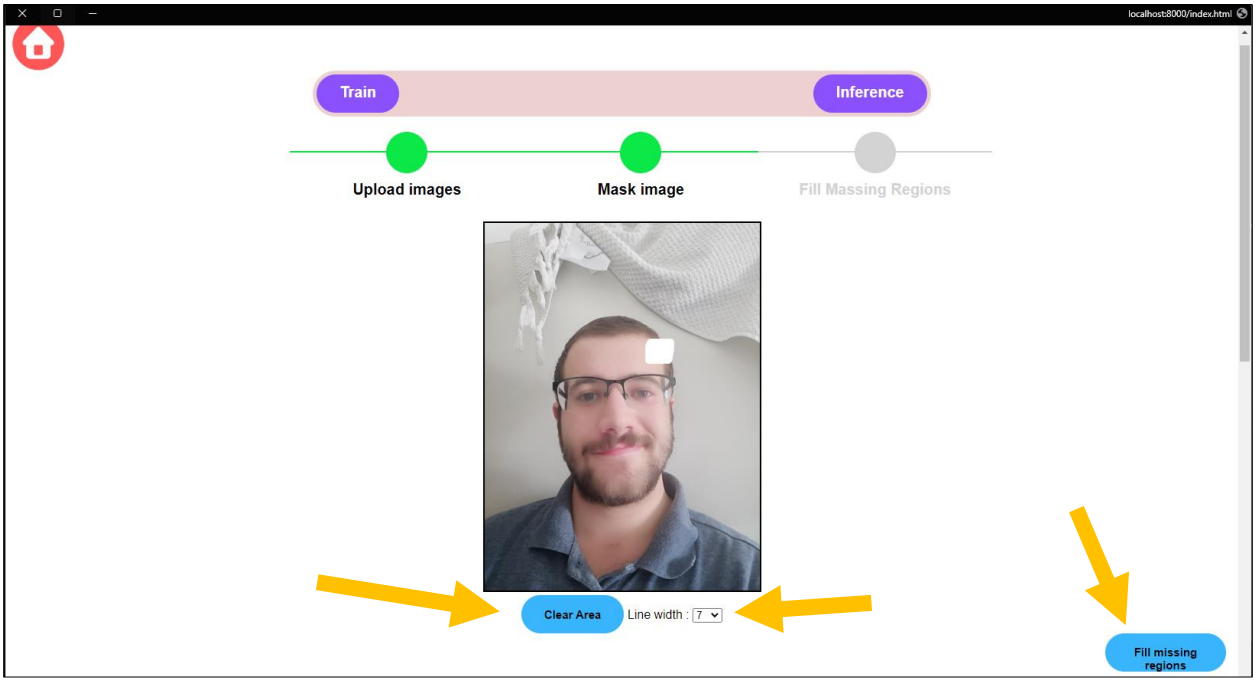
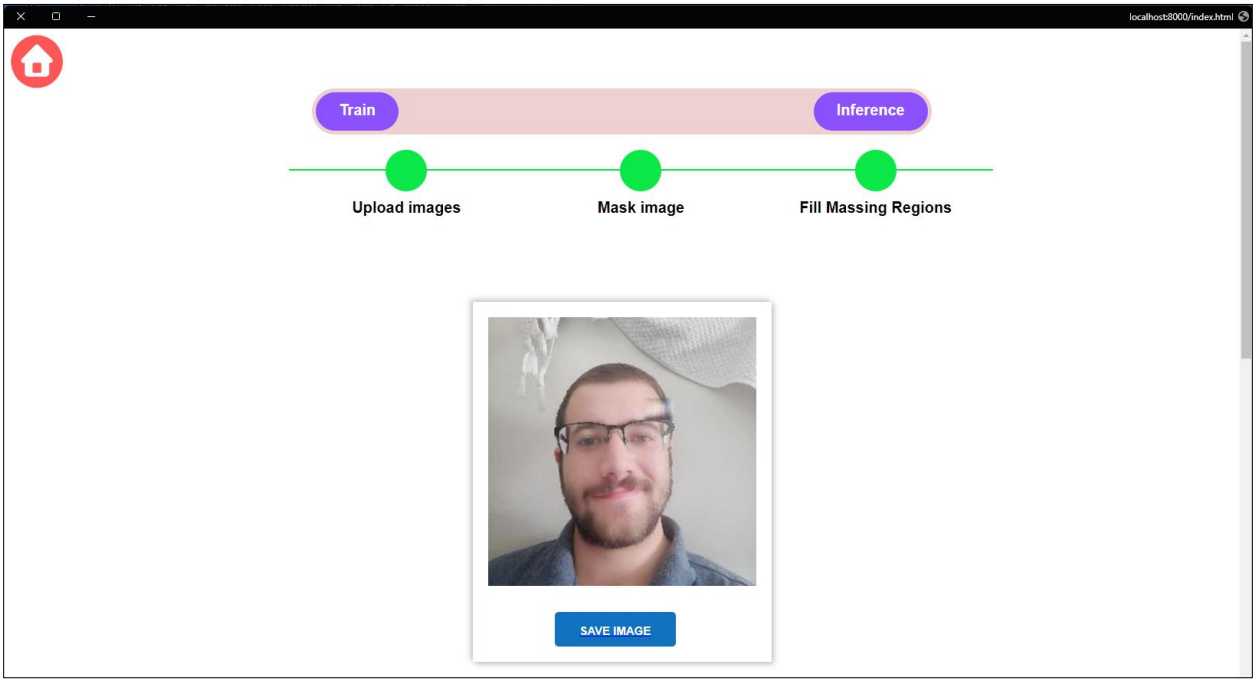


Figure 35: Creating a mask upon the selected image.

The final stage is the completed image generated by InpaintGAN. The user can save the image to local file system by pressing the 'save image' button. See figure [36].



Figure[36] InpaintGAN model fill in the missing regions.

10.3. Maintenance Guide

Use-Case diagram

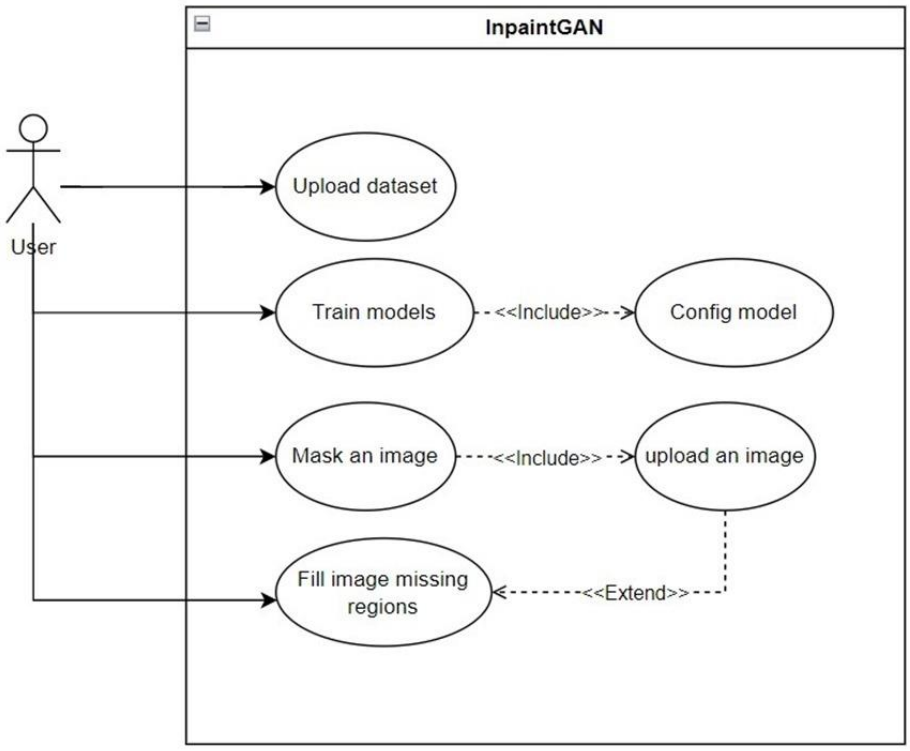


Figure 37: Use-Case diagram

Class diagram

In figure [38] representing the Class diagram of the project and the relations of each class. The class EdgeGenerator and InpaintGenerator contains the model implementations in PyTorch including all the model's layers and executing the model on a certain input. The InpaintModel and EdgeModel classes represents the overall models containing the image processing and executing the EdgeGenerator and InpaintGenerator networks accordingly. In addition, defining the loss functions and calculating the Feature match loss which was explained earlier.

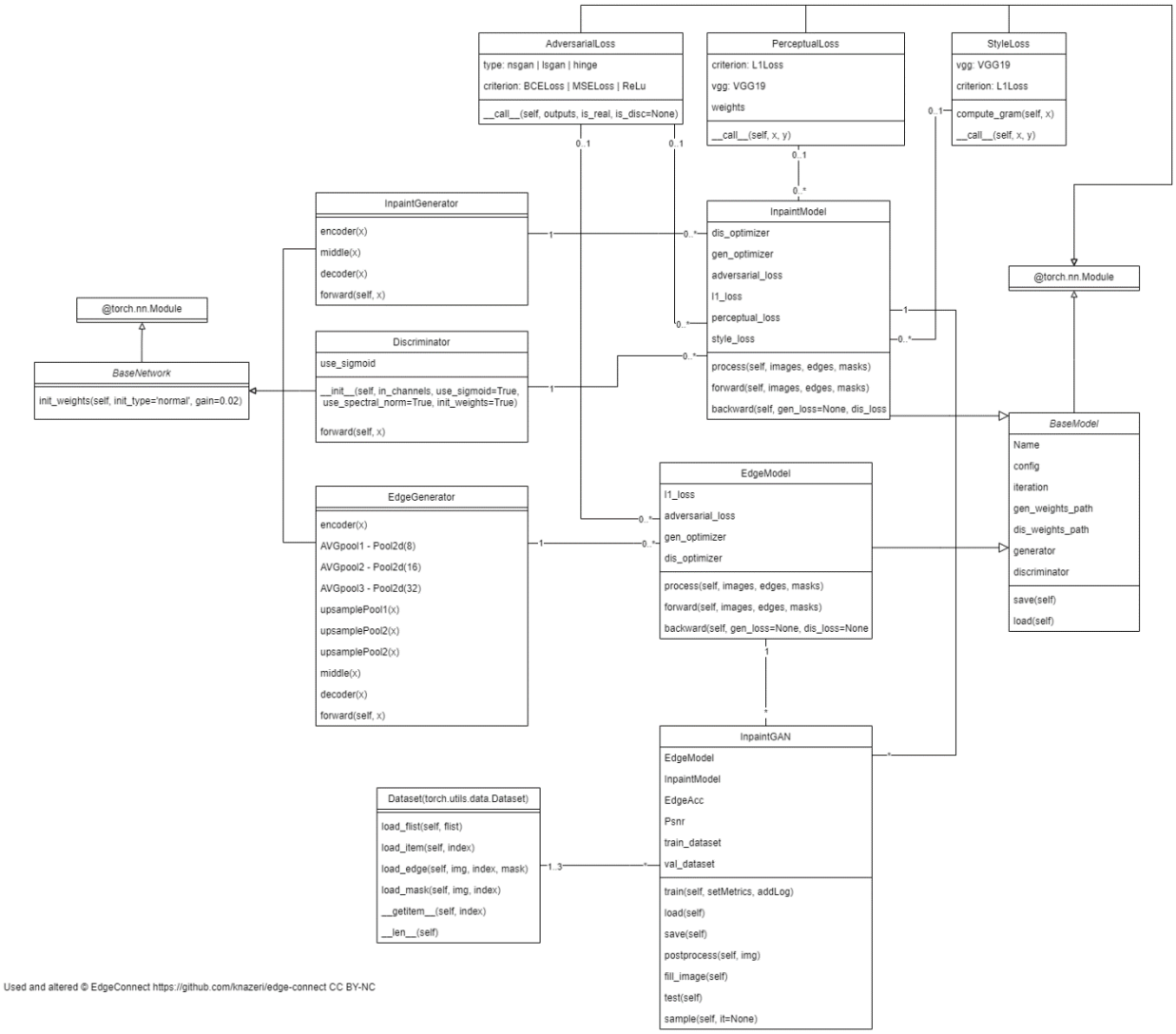


Figure 38: InpaintGAN project Class diagram

Package Diagram

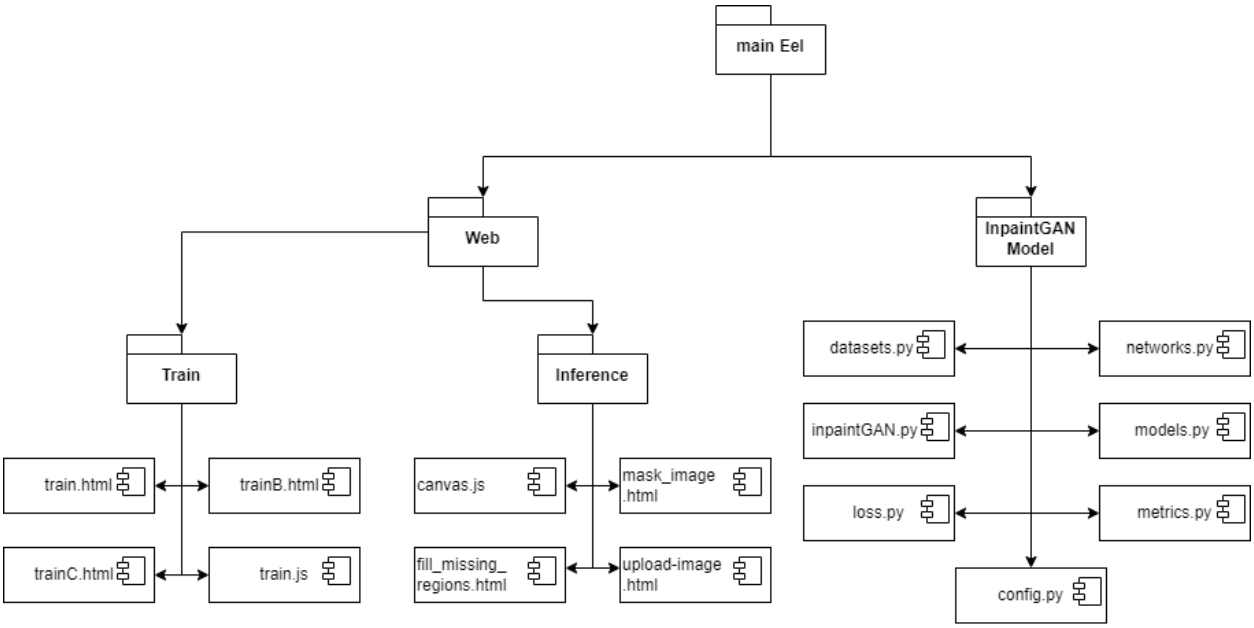


Figure 39

The Eel [8] python package is responsible of linking python files to JavaScript files. The package run the index.html which is the main file for the GUI and exposes python functions.

Train

The project is divided into 2 main packages. See figure 39. The 'Web' package contains the InpaintGAN GUI implementation. The package contains a training package called 'Train' in which is responsible to configure the model from the GUI, linking the dataset path to the model and training the model. The package contains 3 html files corresponding to the 3 stages of the training. See User Instructions 10.2. The 'train.js' file is responsible of collecting all the configurations and calling the exposed python function utilizing Eel package.

Inference

The inference package is responsible of uploading an image from the user in 'upload-image.html', masking the image utilizing HTML canvas for drawing the mask's white lines in 'mask_image.html' and 'canvas.js'. The final stage is running the InpaintGAN model in testing mode utilizing the existing weights to fill the missing regions and present the predicted image to the user.

11. References

[1] Kamyar Nazeri, Eric Ng, Tony Joseph, Faisal Z. Qureshi, Mehran Ebrahimi. EdgeConnect: Generative Image Inpainting with Adversarial Edge Learning

[2] Mingxing Tan, Quoc V.Le. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks

[3] Cai Ran, Xinfu Li, Fang Yang: Multi-Step Structure Image Inpainting Model with Attention Mechanism

[4] Dongsheng Wang, Chaohao Xie, Shaohui Liu, Zhenxing Niu, Wangmeng Zuo: Image Inpainting with Edge-guided Learnable Bidirectional Attention Maps

[5] https://www.kaggle.com/datasets/ashwingupta3012/human-faces

[6] https://www.kaggle.com/

[7] https://github.com/MoshPe/InpaintGAN

[8] https://github.com/python-eel/Eel

[9] Min Wang, Wanglong Lu, Jiankai Lyu, Kaijie Shi, Hanli Zhao: Generative image inpainting with enhanced gated convolution and Transformers