8-2022

DEEP LEARNING EDGE DETECTION IN IMAGE INPAINTING

Zheng Zheng

DEEP LEARNING EDGE DETECTION IN IMAGE INPAINTING

A Project

Presented to the

Faculty of

California State University,

San Bernardino

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

by

Zheng Zheng

August 2022

DEEP LEARNING EDGE DETECTION IN IMAGE INPAINTING

A Project

Presented to the

Faculty of

California State University,

San Bernardino

by

Zheng Zheng

August 2022

Approved by:

Haiyan Qiao, Committee Chair

Kerstin Voigt, Committee Member

Yan Zhang, Committee Member

ABSTRACT

In recent years, deep learning has grown rapidly, and it has been creatively implemented for various applications. In 2019, deep learning based EdgeConnect image inpainting algorithm came out and occupied a place in the image inpainting field. Unlike traditional image inpainting methods which mainly read and use the color information of the remaining part of the image to fill the missing regions of the image, EdgeConnect uses the innovative edge-first and color-next approach. It uses an edge detector to generate an edge map of an image with missing regions, then the missing edges are completed by an edge model, finally the completed edge map is recolored by an inpaint model. The result of this algorithm has a significant improvement in the smoothness of the image, compared with conventional image inpainting methods.

In this project, EdgeConnect is improved to become a completely deep learning-based image inpainting method.

This project first implements the EdgeConnect approach. In the implementation, the project selects the optimal training parameters for the three model training phases included EdgeConnect: edge model, inpainting model and joint model, based on the original research paper and the discussions online. Then the EdgeConnect approach is improved by replacing the traditional Canny edge-detection with the deep learning algorithm, Holistically-Nested Edge Detection (HED). With the integration of HED, the accuracy of image inpainting is improved. To compare the performance, the original EdgeConnect and the

modified EdgeConnect are both trained on the same set of data and the results are scored using the image inpainting quality assessment metrics such as PSNR, SSIM, MAE and FID.

The results show that the modified EdgeConnect approach with the integration of HED not only improves the learning performance of edge detection, but also improves the overall quality of the final image inpainting.

The improved EdgeConnect approach proposed and implemented in this project has higher learning efficiency and better image inpainting performance.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Haiyan Qiao, for her guidance of this project and also for her help and advice with my graduate studies. I feel very privileged to have had the opportunity to work on a project in an area of great personal interest under Professor Qiao's supervision.

I am grateful to the committee members Professors Yan Zhang and Kerstin Voigt for their help on my preparation for the MS oral exam and MS project proposal. The class I took with Professor Zhang sparked my interest in deep learning.

I sincerely thank Professors Voigt and Zhang for their suggestions and support in this project.

Finally, I would like to thank my dear family for their love and trust in me over the years. Special thanks go to my friends who had discussions with me on this project.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF EQUATIONS

CHAPTER ONE

INTRODUCTION

Background

The most fundamental function of image inpainting is to fill the missing regions of the image.

The conventional image inpainting algorithm mainly reads the color information of the unmasked parts of the image and then calculates similar information to fill the missing regions. Although this conventional image inpainting method can successfully recolored the missing regions, it usually cannot reconstruct a reasonable image structure, oftentimes the results are too smooth or blurred, and the whole recolored image may deviate far from original image structure so that people may not understand what it was.

EdgeConnect is a new image inpainting method that can better fill the missing regions. The algorithm follows the innovative edge-first and color-next approach. It includes edge generator and image completion network. The edge generator to generate a complete edge map from the image with missing regions, and the image completion network is used to fill the missing regions of image by coloring the edge map.

EdgeConnect attempts to restore the entire image structure based on remaining structure information of the image, and to then restore the entire image based on the restored structure map and the remaining color information of the image. Thus, EdgeConnect method reduces the appearance of unreasonable parts of the restored image [1].

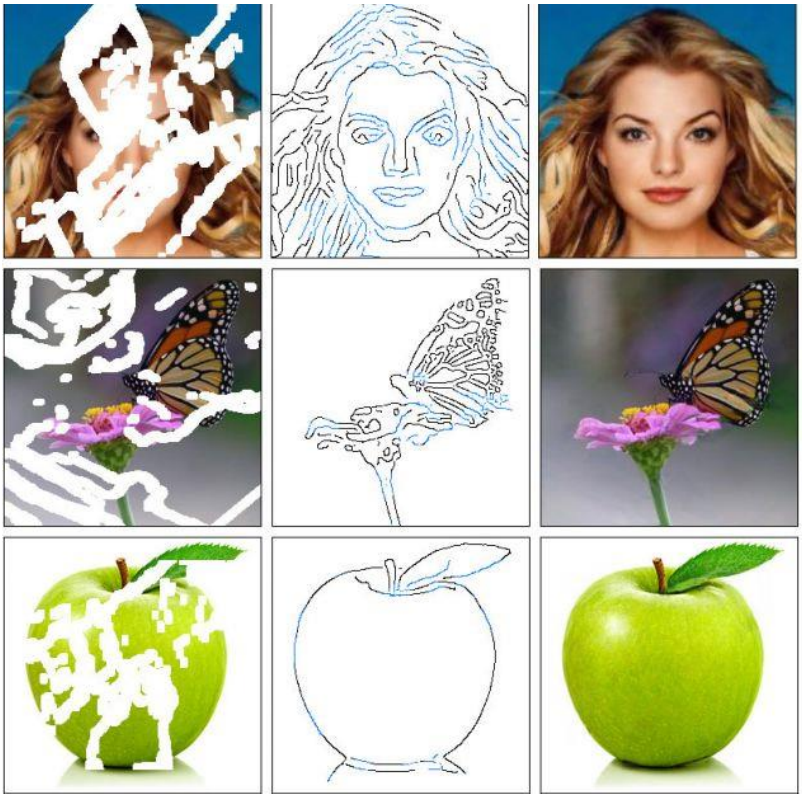


Figure 1 [1]. EdgeConnect Samples

Figure 1 above shows the image inpainting process. The input images on the left images in each row are the masked images where white regions are the missing regions. Each image in the middle column is edge map generated by edge detection and restored by deep learning. The images on the right column in each row are the restored images after filling missing regions by deep learning.

Objectives

The objective of this project is to study deep-learning based EdgeConnect approach and make further improvement of this approach.

The model training of EdgeConnect consists of three parts. The first part is to convert the image into an edge map through edge detection, which is also a part of preprocessing. In the second part, the edge model is trained by the edge map. The third part is to train the inpainting model through the edge map and the ground truth with missing regions (original masked image) and mask.

In the first part of EdgeConnect, the Canny edge detection is used for edge map conversion [1]. In this project, with the intention to improve the accuracy of the whole image inpainting algorithm, the first part is replaced and implemented with deep learning-based edge detection, Holistically-Nested Edge Detection (HED) [2].

Holistically-nested edge detection (HED) is an end-to-end edge detection algorithm that uses “holistically” in name to indicate that the result of edge prediction is based on an image-to-image, end-to-end process; while “nested” emphasizes the process of generating results is the process of training. The algorithm uses a multi-scale approach for feature learning, and the final output of the HED method is far superior to the Canny algorithm [2].

To verify the improvement of deep learning edge detection on image inpainting, comparison experiments are conducted. While ensuring that the experimental conditions are the same, the models are trained separately from scratch to restore a set of images with one model implemented using with for edge detection and another model trained using HED for edge detection. At the end, the image painting results are scored with image inpainting quality metrics to determine whether the modified image inpainting algorithm has been improved.

CHAPTER TWO

EDGECONNECT

Runtime Environment

To reproduce EdgeConnect, the same runtime environment is needed to be set up.

Computer software technology is advancing rapidly, and the latest versions of some software are not compatible for the EdgeConnect project which is only three years old.

In terms of software operating environment, python 3.7 is the most suitable version for the project, The following packages are also used:

Table 1. Package List

site-packges	Version
matplotlib	2.2.5
numpy	1.21.5
opencv-python	3.4.17.63
Pillow	6.2.1
PyYAML	5.4.1
scikit-image	0.14.5
scipy	1.2.3
pytorch	1.0.0
torchvision	0.2.1.

The CUDA 10.2 is adapt to version 1.0 of the pytorch, because the latest CUDA 11 may not allow the torch to recognize the GPU, the same version of pytorch can be adapted to multiple versions of CUDA, so please select the wheel file of pytorch corresponding to the CUDA version to download and install.

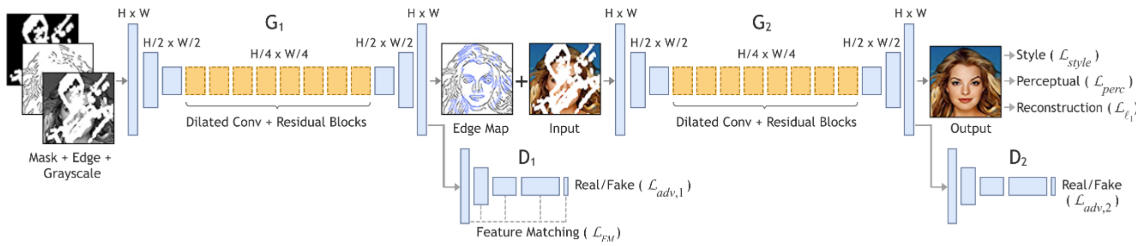


Figure 2[1]. EdgeConnect Network Structure.

Program

EdgeConnect proposed an image inpainting network, which consists of two stages, as shown in Figure 2. G_1 is edge generator and G_2 is image inpainting network.

Two networks are used in both stages as follows:

The generator uses a network architecture which is commonly used for image-to-image translation tasks such as style transfer, super-resolution, etc. [3]. The discriminator uses a 70x70 PatchGAN, which means the discriminant image is divided into 70x70 for discrimination, and the results are averaged [4]. The entire network uses instance normalization, the normalization process simplifies generation by allowing instance-specific contrast information to be removed from content images in tasks such as image stylization [5].

Edge Generator

As can be seen from the left side of Figure 2, in edge generation, mask (M), edge with missing regions ($\tilde{C}_{gt}$) and grayscale with missing regions ($\tilde{I}_{gray}$) are used as inputs, predicted edge map (C_{pred}) will be generated by edge generator, the edge generator G_1 is trained using the standard adversarial loss and the feature matching loss.

I_{gt} is the ground truth, I_{gray} represents the grayscale of the ground truth.

C_{gt} is the edge map of the real image.

M is the mask.

$\odot$ is hadamard product, for two matrices A and B of the same dimension m × n, the Hadamard product $A \odot B$ is a matrix of the same dimension as the operands, with elements given by $(A \odot B)_{ij} = (A)_{ij}(B)_{ij}$ [6].

Deleting the mark regions in ground truth and edge map to generate image with missing regions ($\tilde{I}_{gray}$) and edge map with missing regions ($\tilde{C}_{gt}$) and mark it with a wavy line on the letter:

$\tilde{I}_{gray} = I_{gray} \odot (1 - M)$

$\tilde{C}_{gt} = \tilde{C}_{gt} \odot (1 - M)$

C_{pred} is the prediction result of the Edge Generator.

$\tilde{I}_{gt} = I_{gt} \odot (1 - M)$, $\tilde{I}_{gt}$ is ground truth with missing regions.

I_{pred} is the result of image inpainting.

Predicted edge map generated by generator (G_1) Edge Generator can be expressed as：

$C_{pred} = G_1(\tilde{I}_{gray}, \tilde{C}_{gt}, M)$

The following loss function is constructed to train this adversarial network to obtain the edge generator [1]:

$$\mathcal{L}_{adv,1} = E_{(C_{gt}, I_{gray})} \log\left[D_1\left(C_{gt}, I_{gray}\right)\right] + E_{I_{gray}} \log\left[1 - D_1\ \left(C_{pred}, I_{grey}\right)\right]$$

Equation 1.

$\mathcal{L}_{adv,1}$ is adversarial loss.

$$\mathcal{L}_{FM} = E\left[\sum_{i=1}^{L} \frac{1}{N_i} ||D_1^{(i)}\left(C_{gt}\right) - D_1^{(i)}\left(C_{pred}\right)||_1\right]$$

Equation 2.

$\mathcal{L}_{FM}$ is feature map loss, the input image is discriminated using a pre-trained VGG network, similar to PatchGAN, but since VGG is not trained to extract the contour edges of an image, we cannot use the VGG results directly [4]. We use $\mathcal{L}$ to represent the last convolutional layer of the discriminator. N_i is the activation in the i'th layer of the discriminator.

The edge maps are discriminated using an edge discriminator that combines the adversarial loss with the feature matching loss [1]:

$$\underset{G_1}{min}\underset{D_1}{max}\mathcal{L}_{G_1} = \underset{G_1}{min}\left(\lambda_{adv,1}\underset{D_1}{max}\left(\mathcal{L}_{adv,1}\right) + \lambda_{FM}\mathcal{L}_{FM}\right)$$

Equation 3.

$$\lambda_{adv,1} = 1, \lambda_{Fm} = 10$$

<u>Image Completion Network</u>

As the right side of Figure 2, in image completion network, ground truth with missing regions ($\tilde{I}_{gt}$) and composite edge map (C_{comp}) are used as inputs, predicted result RGB image (I_{pred}) will be generated by inpainting generator, the inpainting generator G_1 is trained using the standard adversarial loss and the feature matching loss.

Predicted result RGB image (I_{pred}) is generated by inpainting generator (G_2) image completion generator can be expressed as [1]:

$$I_{pred} = G_2(\tilde{I}_{gt}, C_{comp})$$

Equation 4.

where $C_{comp} = \tilde{C}_{gt} \odot (1 - M) + C_{pred} \odot M$, which is the combination of the edge of the edge map with missing regions ($\tilde{C}_{gt} \odot （1 - \mathrm{M}）$) and the edge predicted ($C_{pred} \odot M$) by G_1.

The following loss function is constructed to train this adversarial network to obtain the Edge Generator [1].

$$\mathcal{L}_{adv,2} = E_{(I_{gt}, C_{comp})} \log\left[D_2\left(I_{gt}, C_{comp}\right)\right] + E_{C_{comp}} \log\left[1 - D_2\ \left(I_{pred}, C_{comp}\right)\right]$$

Equation 5.

$\mathcal{L}_{adv,2}$ is adversarial loss.

$$\mathcal{L}_{prec} = E\left[\sum_i \frac{1}{N_i} \left\|\phi_1^{(i)}(I_{gt}) - \phi_1^{(i)}(I_{pred})\right\|_1\right]$$

Equation 6.

$\mathcal{L}_{prec}$ is perceptual loss, the input images are discriminated using the pre-trained VGG-19 network.

$$\mathcal{L}_{style} = E_j[\quad ||G_j^{\phi}(I_{pred}) - G_j^{\phi}(I_{gt})||_1]$$

Equation 7.

$\mathcal{L}_{style}$ is style loss. The G_j^{ϕ}in Equation 7. is a Gram Matrix of $C_j \times C_j$ constructed on the activation function eigenmap ϕ_j [7].

The edge maps are discriminated using a map discriminator combining absolute value parametrization (L1 distance l_1), adversarial loss, perceptual loss, and style loss [1].

$$\mathcal{L}_{G_2} = \lambda_{l_1}\mathcal{L}_{l_2} + \lambda_{adv,2}\mathcal{L}_{adv,2} + \lambda_p\mathcal{L}_{perc} + \lambda_s\mathcal{L}_{style}$$

Equation 8.

$$\lambda_{l_1} = 1, \lambda_{adv,2} = \lambda_p = 0.1, \lambda_{style} = 250$$

Model training

A total of two programs are prepared for the experiment, one is the original EdgeConnect, and the other is Improved EdgeConnect, kept the same as that of original EdgeConnect except for the different edge detection used.

Edge model training

The edge model is working for edge generator (G_1) to generate predicted edge map.

To train the edge model, it requires reading the edge map with missing regions, greyscale image and mask as input for training, since edge map with missing regions can be generated by canny edge detection or HED in improved EdgeConnect, so the ground truth and the mask are inputted the program. The program will combine the ground truth and mask into a masked image (image with missing regions) like the left image in Figure 1 to generate an edge map with missing regions by edge detection. The original image validation set to output samples for validation, in order to show the model training results, every 1000 iterations, it will use some images selected from the image validation set and mask validation set as input into the model to generate predicted edge map samples.

The pixels of the image must can be divisible by 4, otherwise it is possible to make the program stop by accident because the pixels before and after the image convolution are different. For example, 402/4 = 100.5 ≈ 100, but 100 * 4 = 400, which means 100 doesn't equal to 100.5.

<u>Inpaint model training</u>

The inpainting model is working for image completion network to generate predicted RGB image. The model will fill in the color of the missing regions of edge map which generated by edge detection.

To train the inpainting model, it is necessary to input masked image (image with missing regions), edge map generated by Canny edge detector or HED and mask, though the edge map of ground truth will be generated from in program.

The model completes the image inpainting by coloring the edge map and then filling the missing regions of the masked image.

<u>Edge-inpaint training</u>

After edge and inpaint models are trained, there is a third training, it replaces the edge map in the inpainting model training with the predicted edge map from the output of the edge model to improve the inpainting model. So masked images, predicted edge map and mask are inputted and G_2 generates predicted RGB image.

The network structure of EdgeConnect inpainting approach is given in Figure 2. The first generator G1 takes the mask, masked edge image and the masked grayscale image as input and gives a predicted edge map. The second generator G2 takes the predicted edge map and the masked RGB image as input and outputs a predicted RGB image [8].

Model testing

The purpose of model testing is to verify the ability of the models' image inpainting through the actual output. In addition to observing the results to check the model training effect, the results are also quantitatively measured using the image inpainting metrics as evaluation.

In this section, the images in the test set need to be pre-masked outside the program, and only the masked images set, and the mask set need to be inputted, and they need to be aligned one to one in their respective folders (same sorting order).

The program will read the masked image and mask, then generate a predicted edge map by the edge generator G_1, and then color the edge map through the edge completion network G_2, finally inpaint the missing regions of masked image by colored edge map. The mask is used to determine what regions of masked image need to be restored.

For now, the images with missing regions in test set are all restored as the result of model testing, the results will be needed in evaluation later.

Finally, the test part is also actually the process of inpainting the image after the models is all trained.

Evaluation

The output set of the model testing and the corresponding ground truth set are used as the input for the evaluation. The two sets of images need to be in one-to-one correspondence and have the same file name, otherwise the program will not detect them. The two sets of data will be compared in terms of Peak Signal-to-Noise Ratio (PSNR), Structural similarity (SSIM), Mean Absolute Error (MAE) and Fréchet inception distance (FID). Through these metrics, we can see the gap by scores between the restored image and the ground truth.

Summary

During the entire EdgeConnect process, the training part is the most important part of the whole project. Although the edge detection only exists as the first step, the edge map generated by the edge detection is used in almost every step of the model training. Therefore, the accuracy of the edge map determines the effect of the edge model and the inpaint model. It is no exaggeration to say that the quality of the edge detector directly affects the quality of image inpainting.

At the same time, the current use of EdgeConnect has some defects, such as the model testing part, the software no longer provides automatic masking function, but requires users to manually batch composite images with missing regions outside the program. If users do not want to use Canny edge detector, then they need to use an additional three folders to store the edge map and edge map with missing regions which are needed to be manually preprocessed with other edge detection outside the EdgeConnect.

In the program test, in most cases, even if some images’ pixels are not a multiple of 4, the program can run normally, but the program always stops running because of one of the images.

CHAPTER THREE

EXPERIMENT

Before experiment, there are some preparations need to be done to make the experiment go smoothly.

Preparation work

Preprocessing

"makimg.py" is wrote and added to the project to generate mask images in batches for the test set, which solved the problem of requiring manual masking in the test part but could not find the script.

"batch_rezise.py" is wrote and added to the project, so that when the number of files in training set, test set and validation set is too large and the pixels of one image causing program stop cannot be found, the images and the masks can be batch preprocessed to 256*256 or any unified specification like 500*500 to avoid program errors.

In order to avoid the need of pre-generating the edge map of HED outside the EdgeConnect, the project provides two solutions, one is to rewrite and add the "hed_processing.py" file to project to generate the edge map in batches outside the EdgeConnect to use with the original EdgeConnect, the second is integrating the HED into EdgeConnect allows the use of the HED in programs.

Dataset

The project has prepared two datasets, the first dataset is one of EdgeConnect used in their paper called Places2 from Massachusetts Institute of Technology, it includes over 400 unique scene categories. such as abbey, badlands, campus, etc. [9].

The other database is downloaded from the web, it includes different breeds of cats in different environments [10][11].

In addition, a mask dataset called Quick Draw Irregular Mask Dataset by Karim Iskakov which is combination of 50 million strokes drawn by human hand. The function of the mask dataset is to cover parts of the image in the original image dataset, thereby forming a lost area on the original image [12].

In each dataset, 48,000 images are selected as the training set, limited by the memory capacity of the graphics card, the batch size is different in different parts of training, and 48000 is just a multiple of 3 batch sizes to ensure that the samples are fully trained. 4,000 as the test set, and 4,000 as the validation set. The training set is used to train the model to improve accuracy, and the validation machine is used to generate image inpainting samples during the training process to view the training effect of the current model and restore the images of the test set through the trained model.

The script "maskimg.py" is used in advance to combine the ground truth and mask into a masked image, which is convenient for the model testing later, ground truth of test set also needed in the evaluation part.

HED

In Improved EdgeConnect, HED has been integrated for edge detection.

Structure

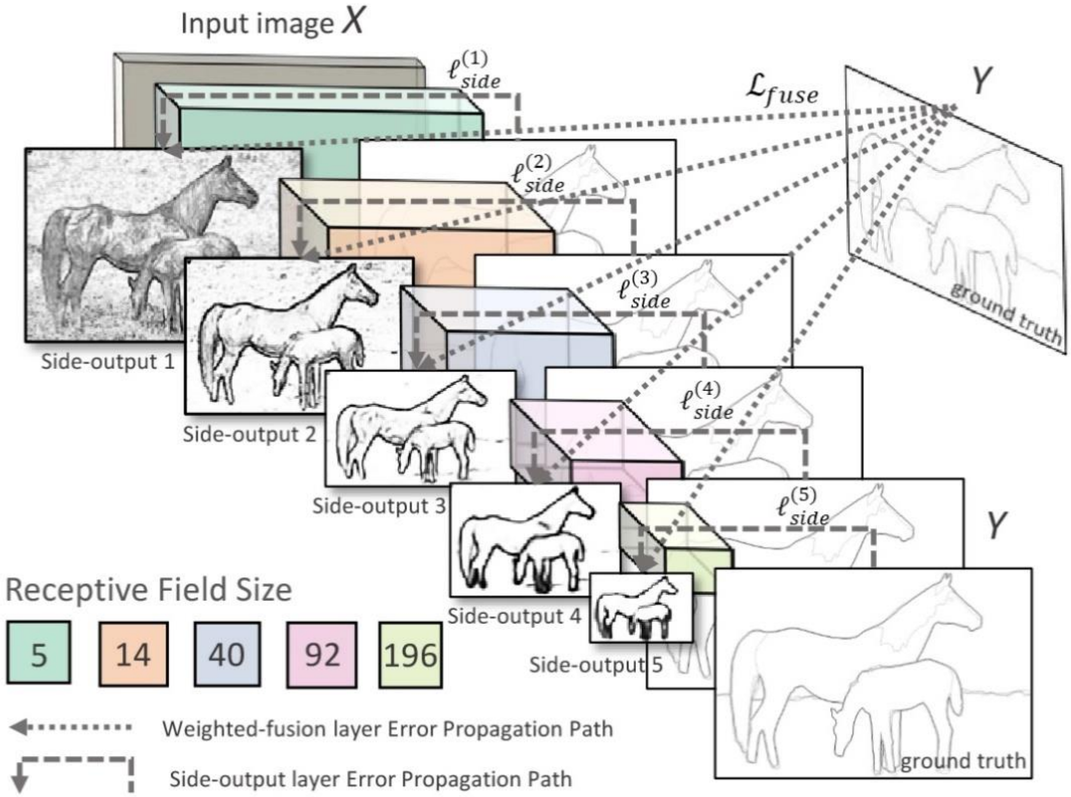


Figure 3 [10]. HED Network Structure.

The HED model consists of five layers of feature extraction architecture, in each layer: layer feature maps are extracted using VGG blocks, layer outputs are computed using layer feature maps, and layer outputs are up-sampled. Finally, the final output of the model is fused with the output of the five layers: the channel dimension is stitched with the output of the five layers 1x1 convolution to fuse the layer outputs [10].

Loss function

Overall, this loss function has two parts: side-output is the prediction result of five different scales in Figure 3, by up-sampling into the original Figure size, and then doing cross-entropy with mask. Because there are five diagrams, the loss is the sum of five. Five graphs fusion out of Y, fusion is the Y and the ground truth of the cross-entropy.

$\boldsymbol{M}$ is number of Side output layers, **W** is the collection of all standard network layer parameters, **w** is the corresponding weights, Index j is over the image spatial dimensions of image X, h is the fusion weight, $\boldsymbol{\hat{Y}}$ is edge map prediction, $\mathbf{Dist}(\cdot,\cdot)$ is the distance between the fused predictions and the ground truth label map, which set as cross-entropy loss.

There is side out loss function and weight-fusion loss function,

$$\mathcal{L}_{side}(\mathbf{W},\mathbf{w}) = \sum_{m=1}^{M} \alpha_m \ell_{side}^{(m)}\left(\mathbf{W},\mathbf{w}^{(\boldsymbol{m})}\right)$$

Equation 9.

$$\mathcal{L}_{fuse}(\mathbf{W},\mathbf{w},\mathbf{h}) = \mathrm{Dist}\left(Y,\hat{Y}_{fuse}\right)$$

Equation 10.

the objective function when training the model is to minimize the sum of the side branch $\mathcal{L}_{side}(W,w)$ and fuse loss $\mathcal{L}_{fuse}(W,w,h)$ [10]:

$$(\mathbf{W},\mathbf{w},\mathbf{h})^* = \operatorname{argmin}\left(\mathcal{L}_{side}(\mathbf{W},\mathbf{w}) + \mathcal{L}_{fuse}(\mathbf{W},\mathbf{w},\mathbf{h})\right)$$

Equation 11.

Comparison

The purpose of this experiments is to carry out the effect of two different edge detectors on image inpainting, so in the experiments, the experiments abandoned the use of the EdgeConnect author's model that has gone through 2,000,000 iterations, and instead trained it myself from 0 iteration. Since the target number of iterations of my model is significantly less than the model of the original author, the effect of the model has a significant worse compared to the original author. Except for the difference in edge detectors, the two sets of models were trained under the same learning rate, number of batches, learning rate, and iterations, etc.

Edge model training

So, for the edge model training, Setting the learning rate at 0.0001 and set the size of batches to 16, while setting the style loss weight at 250 to ensure the best training effect. To ensure that both models have the same training environment, the edge training for both groups will stop at 20 epochs.

Because the edge model training is directly based on the original edge maps generated by the edge detection and predicted edge map generated by G_1 affect the third part of model training, the edge maps have a direct impact on the deep learning.

Figure 4. Edge Model Training Sample (Canny, Cat)

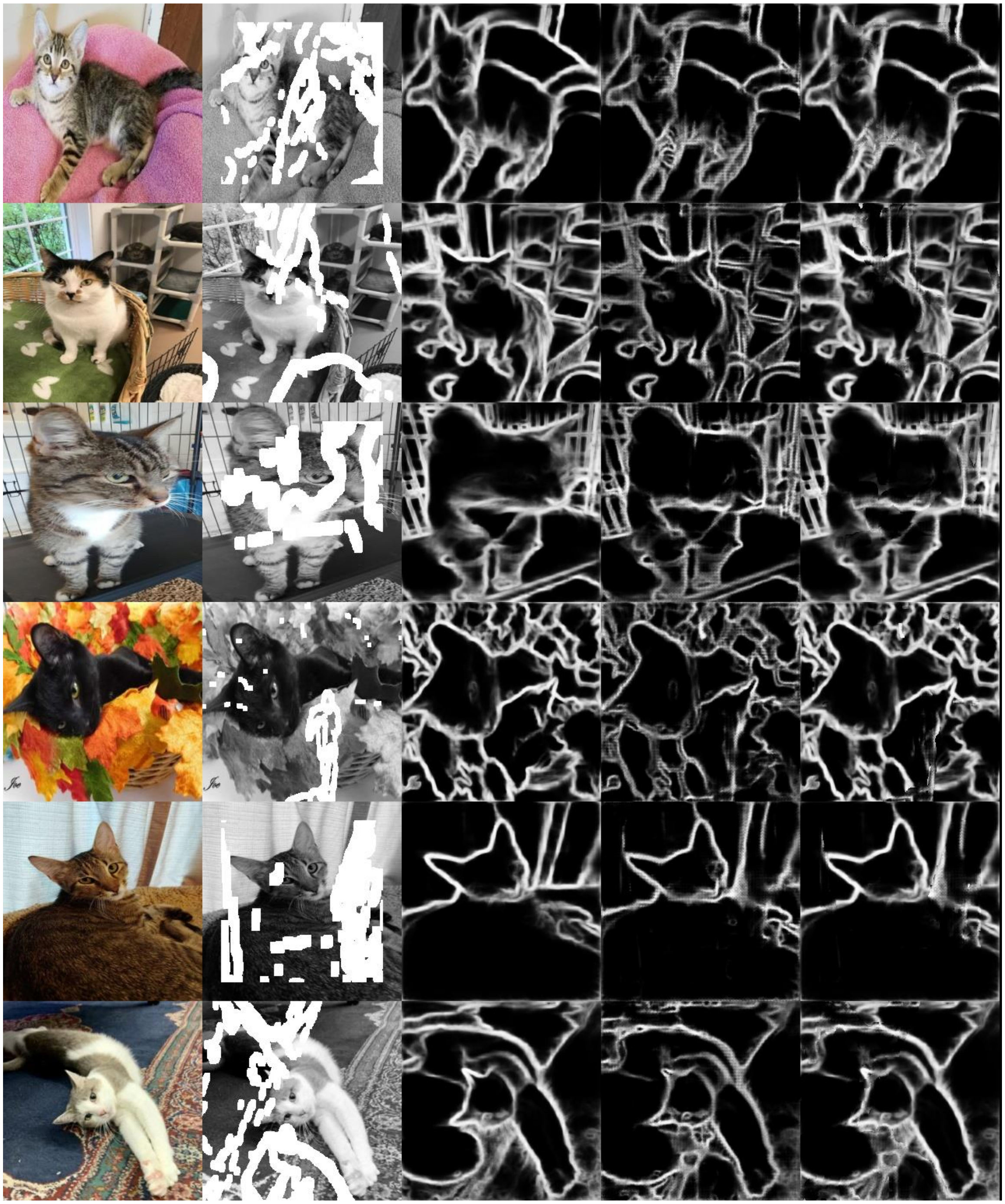


Figure 5. Edge Model Training Sample (HED, Cat)

In Figure 4, the first images in column are the ground truth (original image). The second images in the column are the masked image (also input). The third images in column are the edge map from ground truth by Canny edge generator. The fourth images in column are the actual output of the network. Finally, the fifth images in column are the combination of the third and fourth images in column, the known area is from the third images in column and the masked area is from the fourth images in column.

In Figure 5, the third images in column are the edge map from ground truth by Canny edge generator and the others are same to Figure 4.

The process generates the predicted edge map by the edge model, then use it to fill the missing regions of masked images' edge map and check the precision and recall after comparing the predicted edge map and original edge map. Every 1000 iterations, the program will test the model by validation set, to show the learning result of the model at that time.

As epochs increase, the edge predicted by the edge model will become more and more accurate.

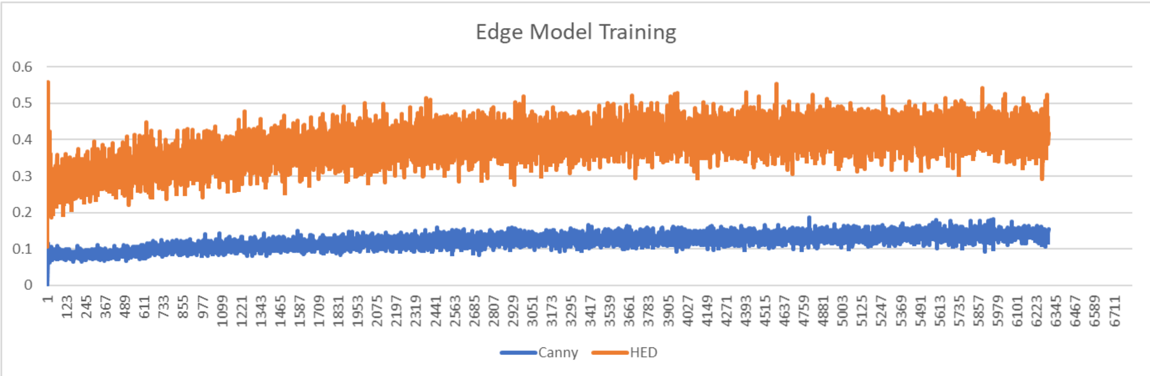


Figure 6. Precision of Canny and HED During the Edge Model Training (Cat)

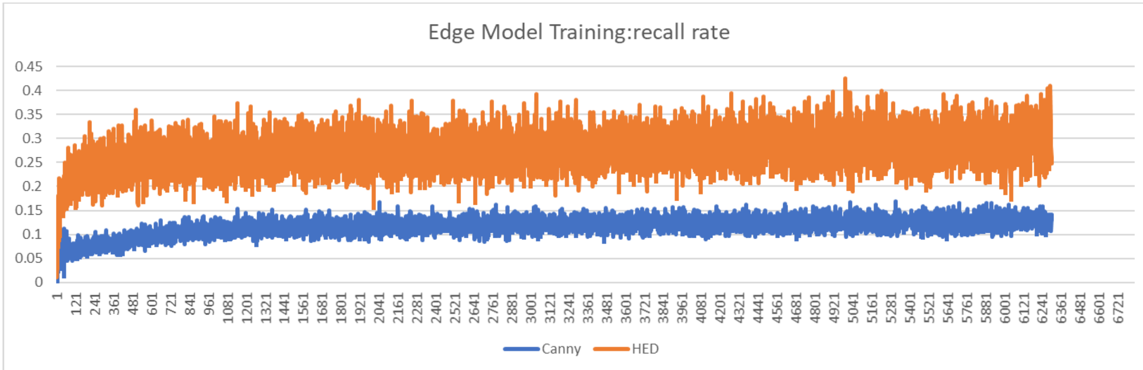


Figure 7. Recall of Canny and HED During the Edge Model Training (Cat)

Precision means the percent of correctly predicted edge lines in all predicted edge lines. Recall means the percent of correctly predicted edge lines in all edge lines needed to be predicted.

After the edge model training, the difference between Canny edge detection and HED can be seen from the accuracy and recall of feedback during training. With the same learning rate, the edge restoration level of the edge model learned through the edge map generated by HED higher than Canny's.

The same effect can also be seen from the edge training of the comparative experiment based on another set of Places2 datasets.

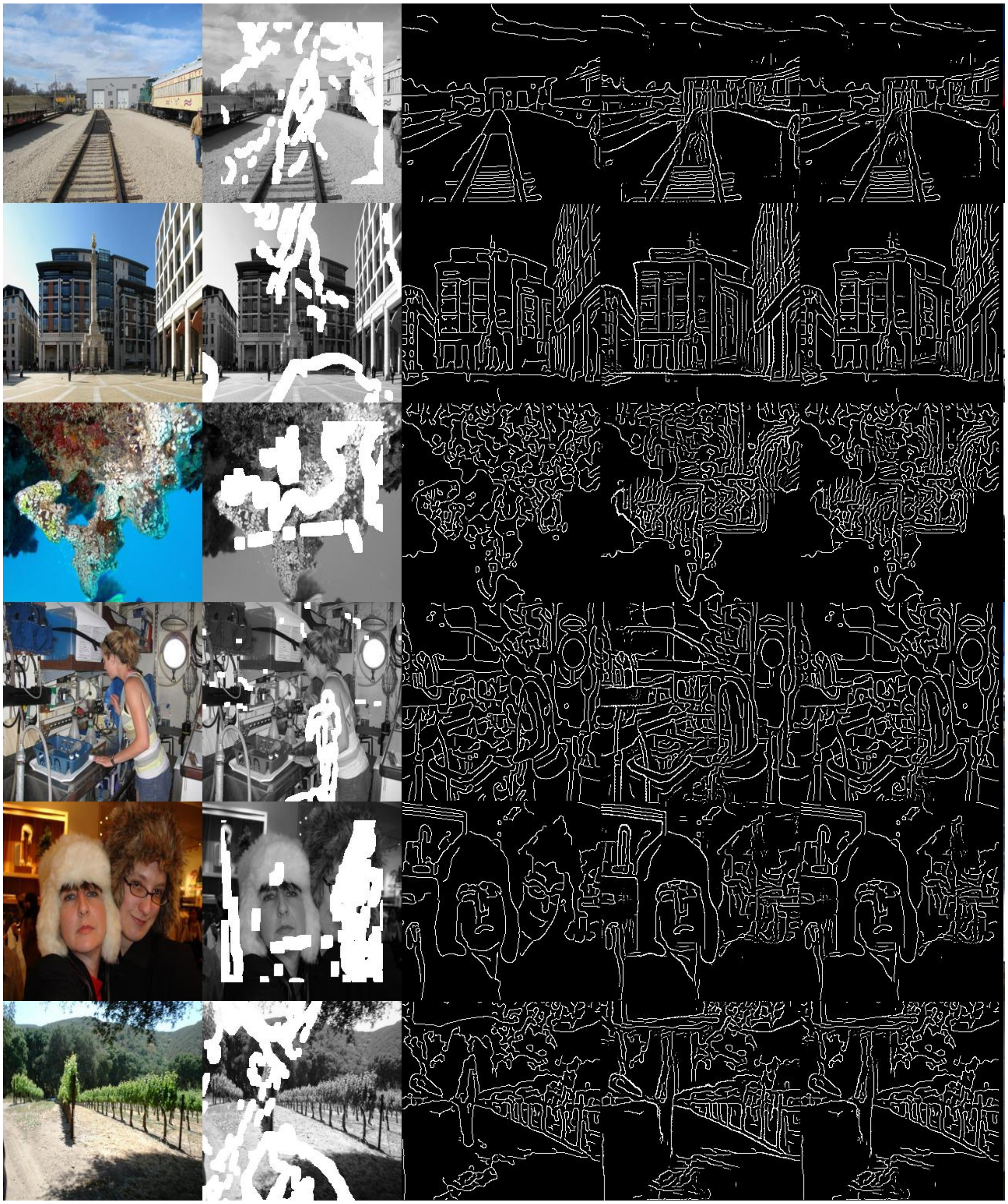


Figure 8. Edge Model Training Sample (Canny, Placese2)

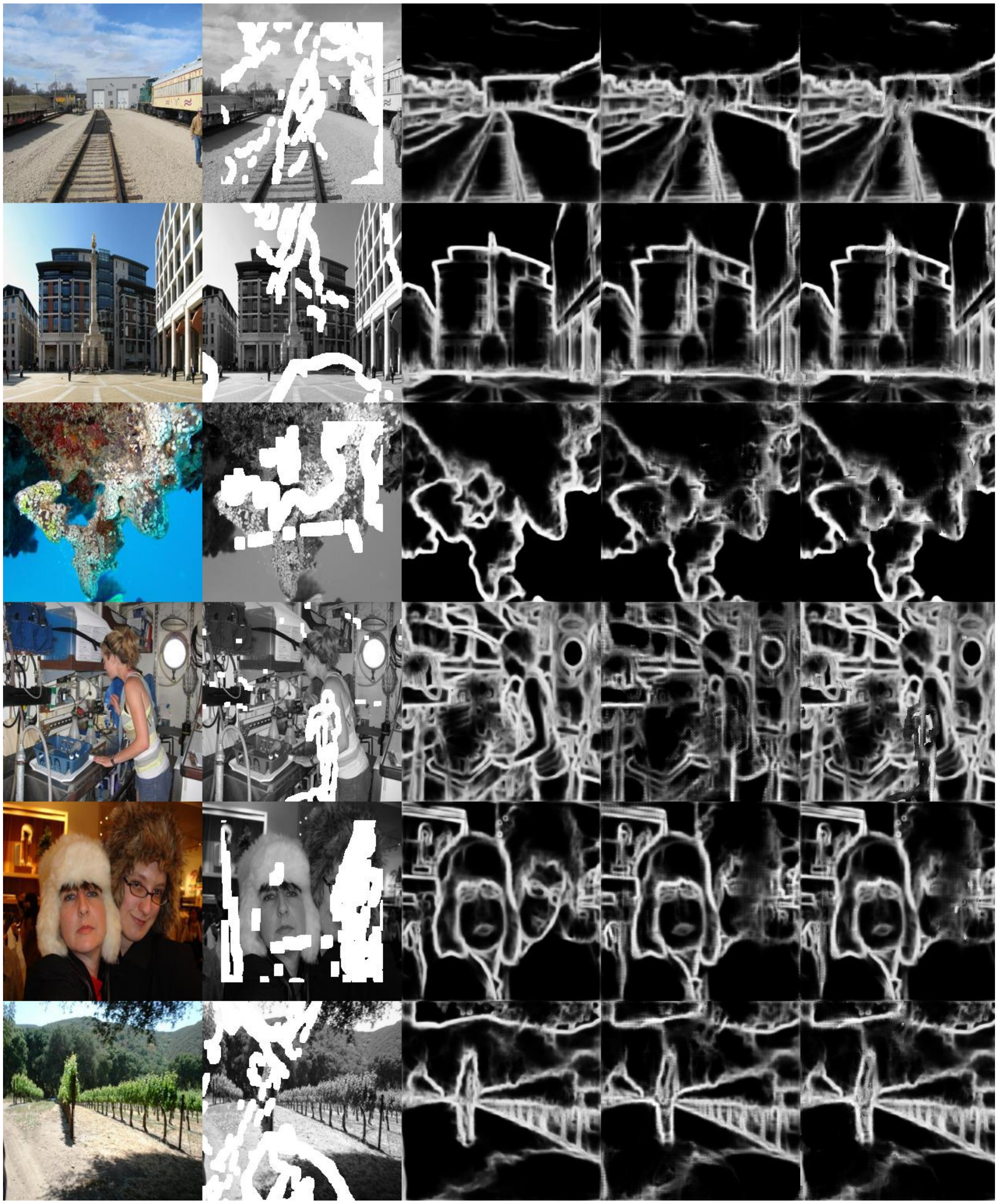


Figure 9. Edge Model Training Sample (HED, Placese2)

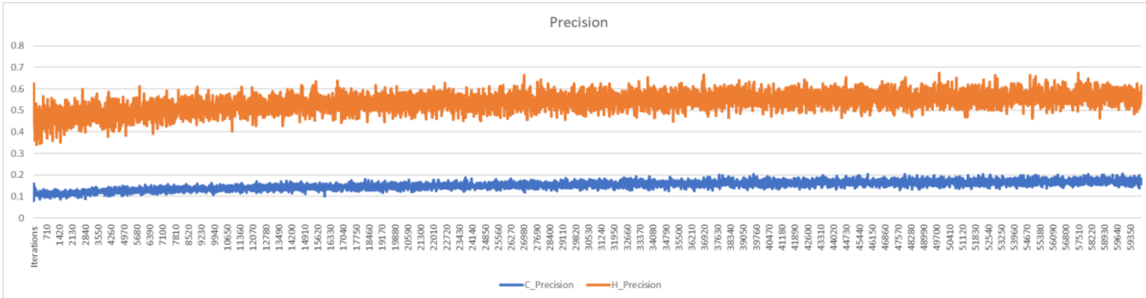


Figure 10. Precision Of Canny and HED During the Edge Model Training (Places2)

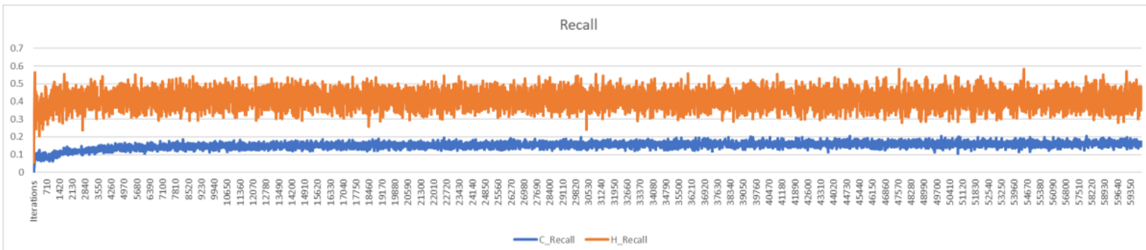


Figure 11. Recall Of Canny and HED During the Edge Model Training (Places2)

<u>Inpaint model training</u>

In the next training of the inpainting model, because the size of the input becomes larger, the GPU memory must be increased to maintain the previous batch size setting or reduce the size of the batch.

Therefore, in this section, other settings remain the same, but the batch size is changed to 8. In the inpaint training, the model still needs the edge map as input and then combines the colors of the ground truth with missing regions and predicted RGB image.

In this training, the output (predicted RGB image) generated by the inpaint model will be closer and closer to the ground truth, so the inpainting effect will be better and better.

Figure 12. Inpaint Model Training Sample (Canny, Cat)

Figure 13. Inpaint Model Training Sample (HED, Cat)

Figure 14. Inpaint Model Training Sample (Canny, Places2)

Figure 15. Inpaint Model Training Sample (HED, Places2)

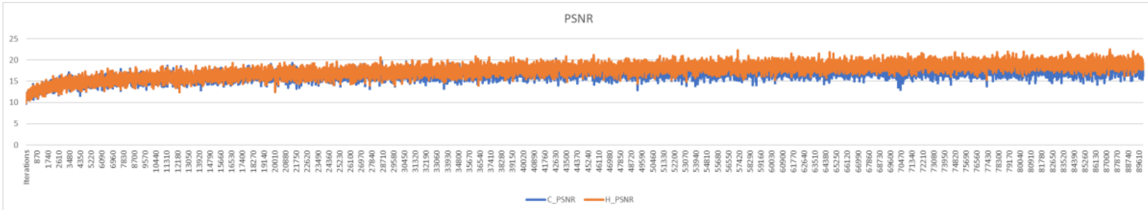


Figure 16. PSNR Of Canny and HED During the Inpaint Model Training (Places2)

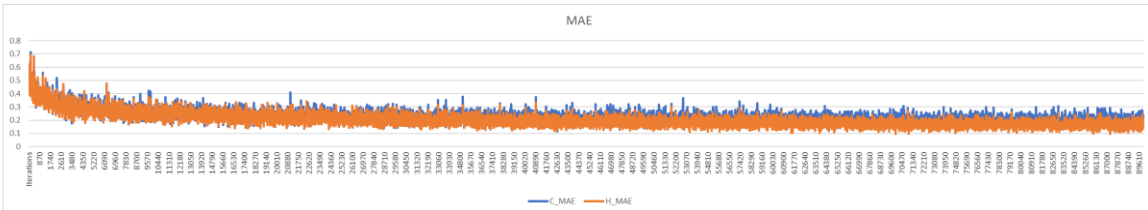


Figure 17. MAE Of Canny and HED During the Inpaint Model Training (Places2)

On each row in Figure 12, starting from the left, first image is the ground truth (original image), second image is the masked image (also input). The third is the edge map from original image by Canny edge detection. The fourth image is the actual output of the network. Finally, the last image is the combination of the second and fourth image: the known area is from the second image and the masked area is from the fourth image.

In Figure 13, the third image on column is the edge map from ground truth by HED.

PSNR is peak signal-to-noise ratio, it is the basis for judging image noise. The smaller the PSNR value, the more noise the image has, which means the more blurred the image is, the worse the level of image restoration is.

MAE means Mean Absolute Error，it is used to reflect the error value between the predicted image and the original image. The smaller the value, the better restoration.

Although their difference is not large, it can be seen that HED's inpaint model is still superior to Canny's. Because in the Figure 16 PSNR chart, the most of blue value is under orange's and also in Figure 17 MAE, the blue is always at orange's upside.

For consistency, both groups of model training were stopped after completing 15 epochs.

<u>Edge-inpaint training</u>

The final edge-inpaint training only backpropagates for inpaint model but use the output of edge model as edge input, this is for G_2 to adapt to the predicted edge map of G_1 as input. Because the training requires the input of both models, the memory requirement is increased again. Currently, the size of batch processing is decreased to 6, and change the learning rate to 0.00001 to help the model converge. This training ends after 10 epochs.

In other words, this third training just replaces the correct edge map with the edge map predicted by the edge model to train the inpainting ability of the inpaint model, which can well adjust the inpaint model to adapt to the edge model, this also explains importance of edge detection for overall image inpainting.

Figure 18. Edge-Inpaint Training Sample (Canny, Cat)

In Figure 18, the first images in column are the ground truth (original image). The second images in column are the masked image (also input). The third images in column are the predicted edge from the edge model (Canny). The fourth images in column are the actual output of the network. Finally, the fifth images in column are the combination of the second and fourth images in column, the known area is from the second images in column and the masked area are from the fourth images in column.

Figure 19. Edge-Inpaint Training Sample (HED, Cat)

In Figure 19, the third image in column is the predicted edge from the edge model (HED).

Figure 20. Edge-Inpaint Training Sample (Canny, Places2)

Figure 21. Edge-Inpaint Training Sample (HED, Places2)

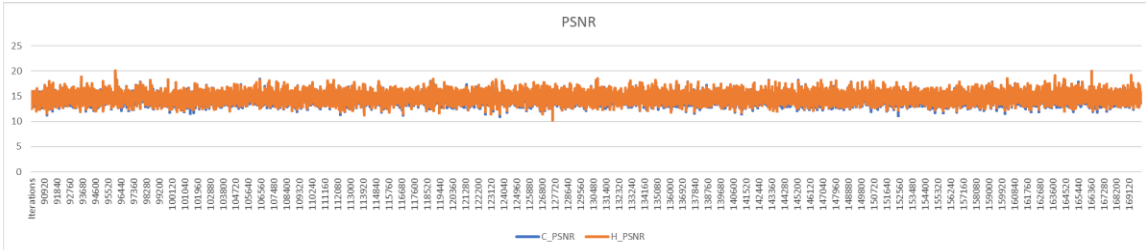


Figure 22. PSNR Of Canny and HED During the Edge Inpaint Model Training (Places2)

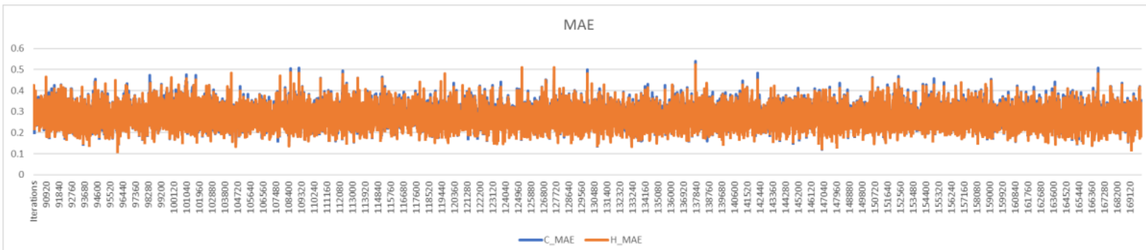
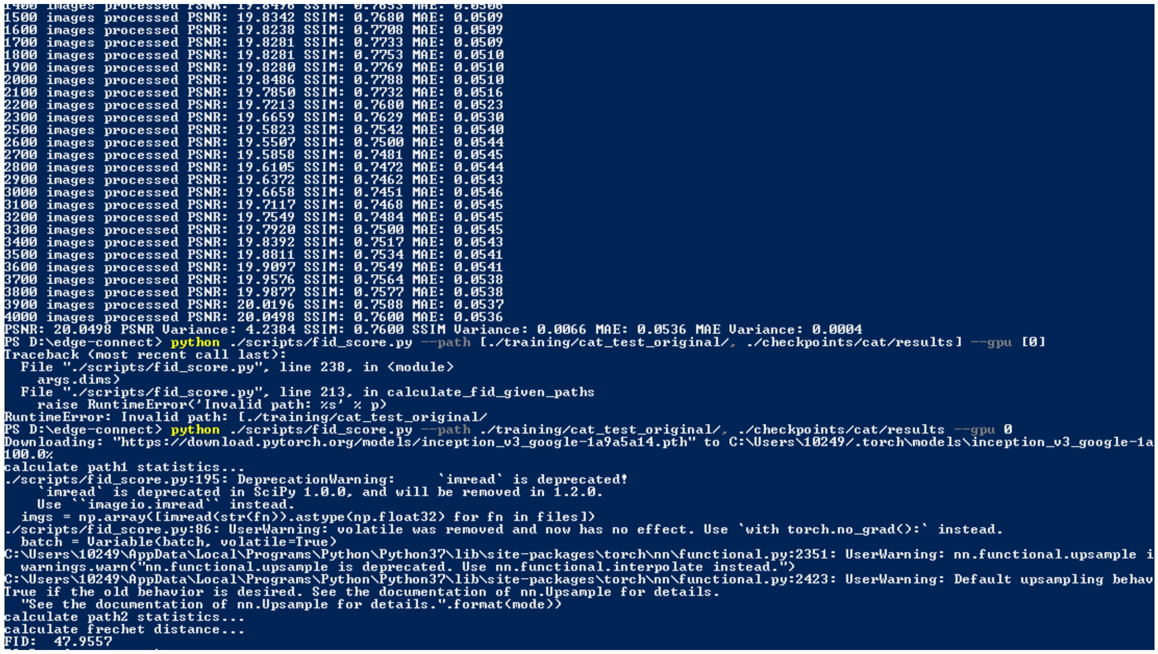


Figure 23. MAE Of Canny and HED During the Edge Inpaint Model Training (Places2)

The trend of edge-inpaint mode is similar to inpaint mode, most HED scores are better than Canny's.

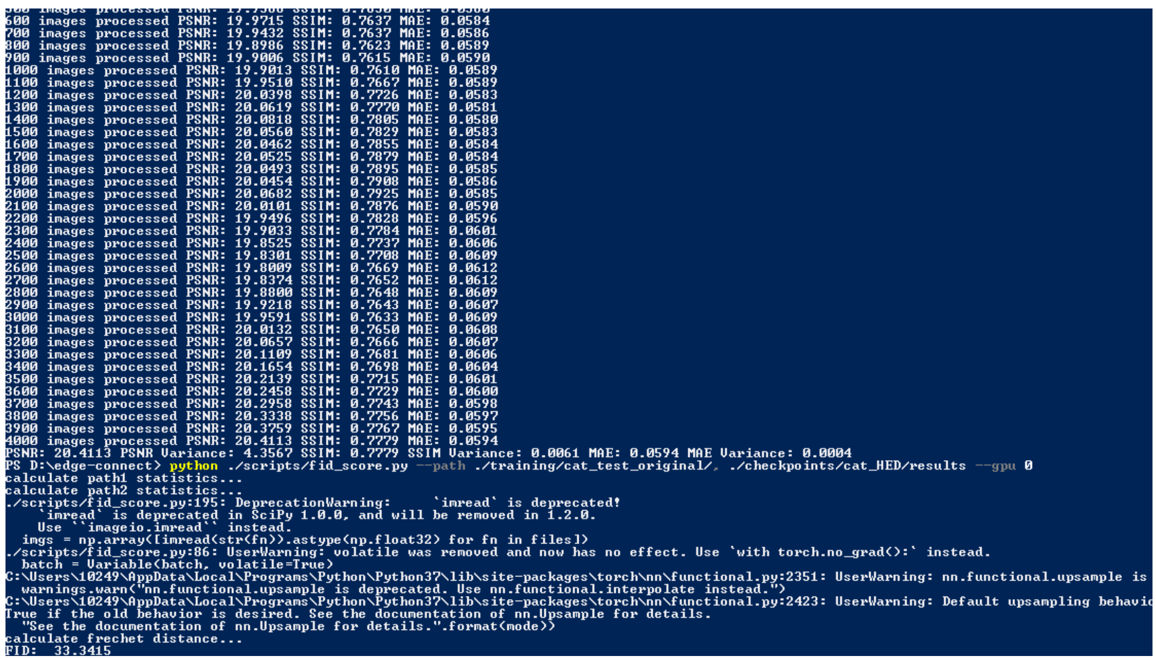
Evaluation

After training the model, put the test set with masked image into "test.py" for image inpainting, and then put the results and the ground truth into "metrics.py" and "fid-score.py" for scoring, finally obtain the average value of the inpainting degree of test set images for the models trained based on two sets of different edge detections:



```
1500 images processed PSNR: 19.8342 SSIM: 0.7680 MAE: 0.0509
1600 images processed PSNR: 19.8238 SSIM: 0.7708 MAE: 0.0509
1700 images processed PSNR: 19.8281 SSIM: 0.7733 MAE: 0.0509
1800 images processed PSNR: 19.8281 SSIM: 0.7753 MAE: 0.0510
1900 images processed PSNR: 19.8280 SSIM: 0.7769 MAE: 0.0510
2000 images processed PSNR: 19.8486 SSIM: 0.7788 MAE: 0.0510
2100 images processed PSNR: 19.7850 SSIM: 0.7732 MAE: 0.0516
2200 images processed PSNR: 19.7213 SSIM: 0.7680 MAE: 0.0523
2300 images processed PSNR: 19.6659 SSIM: 0.7629 MAE: 0.0530
2500 images processed PSNR: 19.5823 SSIM: 0.7542 MAE: 0.0540
2600 images processed PSNR: 19.5507 SSIM: 0.7500 MAE: 0.0544
2700 images processed PSNR: 19.5858 SSIM: 0.7481 MAE: 0.0545
2800 images processed PSNR: 19.6105 SSIM: 0.7472 MAE: 0.0544
2900 images processed PSNR: 19.6372 SSIM: 0.7462 MAE: 0.0543
3000 images processed PSNR: 19.6658 SSIM: 0.7451 MAE: 0.0546
3100 images processed PSNR: 19.7117 SSIM: 0.7468 MAE: 0.0545
3200 images processed PSNR: 19.7549 SSIM: 0.7484 MAE: 0.0545
3300 images processed PSNR: 19.7920 SSIM: 0.7500 MAE: 0.0545
3400 images processed PSNR: 19.8392 SSIM: 0.7517 MAE: 0.0543
3500 images processed PSNR: 19.8811 SSIM: 0.7534 MAE: 0.0541
3600 images processed PSNR: 19.9097 SSIM: 0.7549 MAE: 0.0541
3700 images processed PSNR: 19.9576 SSIM: 0.7564 MAE: 0.0538
3800 images processed PSNR: 19.9877 SSIM: 0.7577 MAE: 0.0538
3900 images processed PSNR: 20.0196 SSIM: 0.7588 MAE: 0.0537
4000 images processed PSNR: 20.0498 SSIM: 0.7600 MAE: 0.0536
PSNR: 20.0498 PSNR Variance: 4.2384 SSIM: 0.7600 SSIM Variance: 0.0066 MAE: 0.0536 MAE Variance: 0.0004
PS D:\edge-connect> python ./scripts/fid_score.py --path [./training/cat_test_original/, ./checkpoints/cat/results] --gpu [0]
Traceback (most recent call last):
  File "./scripts/fid_score.py", line 238, in <module>
    args.dims)
  File "./scripts/fid_score.py", line 213, in calculate_fid_given_paths
    raise RuntimeError('Invalid path: %s' % p)
RuntimeError: Invalid path: [./training/cat_test_original/
PS D:\edge-connect> python ./scripts/fid_score.py --path ./training/cat_test_original/, ./checkpoints/cat/results --gpu 0
Downloading: "https://download.pytorch.org/models/inception_v3_google-1a9a5a14.pth" to C:\Users\10249/.torch\models\inception_v3_google-1a
100.0%
calculate path1 statistics...
./scripts/fid_score.py:195: DeprecationWarning:     `imread` is deprecated!
    `imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
    Use ``imageio.imread`` instead.
  imgs = np.array([imread(str(fn)).astype(np.float32) for fn in files])
./scripts/fid_score.py:86: UserWarning: volatile was removed and now has no effect. Use `with torch.no_grad():` instead.
  batch = Variable(batch, volatile=True)
C:\Users\10249\AppData\Local\Programs\Python\Python37\lib\site-packages\torch\nn\functional.py:2351: UserWarning: nn.functional.upsample i
  warnings.warn("nn.functional.upsample is deprecated. Use nn.functional.interpolate instead.")
C:\Users\10249\AppData\Local\Programs\Python\Python37\lib\site-packages\torch\nn\functional.py:2423: UserWarning: Default upsampling behav
True if the old behavior is desired. See the documentation of nn.Upsample for details.
  "See the documentation of nn.Upsample for details.".format(mode))
calculate path2 statistics...
calculate frechet distance...
FID:  47.9557
```

Figure 24. Evaluation of Edgeconnect (Canny, Cat)



```
600 images processed PSNR: 19.9715 SSIM: 0.7637 MAE: 0.0584
700 images processed PSNR: 19.9432 SSIM: 0.7637 MAE: 0.0586
800 images processed PSNR: 19.8986 SSIM: 0.7623 MAE: 0.0589
900 images processed PSNR: 19.9006 SSIM: 0.7615 MAE: 0.0590
1000 images processed PSNR: 19.9013 SSIM: 0.7610 MAE: 0.0589
1100 images processed PSNR: 19.9510 SSIM: 0.7667 MAE: 0.0589
1200 images processed PSNR: 20.0398 SSIM: 0.7726 MAE: 0.0583
1300 images processed PSNR: 20.0619 SSIM: 0.7770 MAE: 0.0581
1400 images processed PSNR: 20.0818 SSIM: 0.7805 MAE: 0.0580
1500 images processed PSNR: 20.0560 SSIM: 0.7829 MAE: 0.0583
1600 images processed PSNR: 20.0462 SSIM: 0.7855 MAE: 0.0584
1700 images processed PSNR: 20.0525 SSIM: 0.7879 MAE: 0.0584
1800 images processed PSNR: 20.0493 SSIM: 0.7895 MAE: 0.0585
1900 images processed PSNR: 20.0454 SSIM: 0.7908 MAE: 0.0586
2000 images processed PSNR: 20.0682 SSIM: 0.7925 MAE: 0.0585
2100 images processed PSNR: 20.0101 SSIM: 0.7876 MAE: 0.0590
2200 images processed PSNR: 19.9496 SSIM: 0.7828 MAE: 0.0596
2300 images processed PSNR: 19.9033 SSIM: 0.7784 MAE: 0.0601
2400 images processed PSNR: 19.8525 SSIM: 0.7737 MAE: 0.0606
2500 images processed PSNR: 19.8301 SSIM: 0.7708 MAE: 0.0609
2600 images processed PSNR: 19.8009 SSIM: 0.7669 MAE: 0.0612
2700 images processed PSNR: 19.8374 SSIM: 0.7652 MAE: 0.0612
2800 images processed PSNR: 19.8800 SSIM: 0.7648 MAE: 0.0609
2900 images processed PSNR: 19.9218 SSIM: 0.7643 MAE: 0.0607
3000 images processed PSNR: 19.9591 SSIM: 0.7633 MAE: 0.0609
3100 images processed PSNR: 20.0132 SSIM: 0.7650 MAE: 0.0608
3200 images processed PSNR: 20.0657 SSIM: 0.7666 MAE: 0.0607
3300 images processed PSNR: 20.1109 SSIM: 0.7681 MAE: 0.0606
3400 images processed PSNR: 20.1654 SSIM: 0.7698 MAE: 0.0604
3500 images processed PSNR: 20.2139 SSIM: 0.7715 MAE: 0.0601
3600 images processed PSNR: 20.2458 SSIM: 0.7729 MAE: 0.0600
3700 images processed PSNR: 20.2958 SSIM: 0.7743 MAE: 0.0598
3800 images processed PSNR: 20.3338 SSIM: 0.7756 MAE: 0.0597
3900 images processed PSNR: 20.3759 SSIM: 0.7767 MAE: 0.0595
4000 images processed PSNR: 20.4113 SSIM: 0.7779 MAE: 0.0594
PSNR: 20.4113 PSNR Variance: 4.3567 SSIM: 0.7779 SSIM Variance: 0.0061 MAE: 0.0594 MAE Variance: 0.0004
PS D:\edge-connect> python ./scripts/fid_score.py --path ./training/cat_test_original/, ./checkpoints/cat_HED/results --gpu 0
calculate path1 statistics...
calculate path2 statistics...
./scripts/fid_score.py:195: DeprecationWarning:     `imread` is deprecated!
    `imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
    Use ``imageio.imread`` instead.
  imgs = np.array([imread(str(fn)).astype(np.float32) for fn in files])
./scripts/fid_score.py:86: UserWarning: volatile was removed and now has no effect. Use `with torch.no_grad():` instead.
  batch = Variable(batch, volatile=True)
C:\Users\10249\AppData\Local\Programs\Python\Python37\lib\site-packages\torch\nn\functional.py:2351: UserWarning: nn.functional.upsample is
  warnings.warn("nn.functional.upsample is deprecated. Use nn.functional.interpolate instead.")
C:\Users\10249\AppData\Local\Programs\Python\Python37\lib\site-packages\torch\nn\functional.py:2423: UserWarning: Default upsampling behavio
True if the old behavior is desired. See the documentation of nn.Upsample for details.
  "See the documentation of nn.Upsample for details.".format(mode))
calculate frechet distance...
FID:  33.3415
```

Figure 25. Evaluation of Edgeconnect (HED, Cat)

Table 2. The Metrics Score of Canny Edge Detection and HED (Cat)

	PSNR	SSIM	MAE	FID
EdgeConnect (Canny)	20.0498	0.7600	0.0536	47.9557
EdgeConnect (HED)	20.4113	0.7779	0.0594	33.3415
Improvement	+ 1.8%	+ 2.3%	- 10.8%	+ 30.47%

The "+" sign represents the improvement in performance, and the "-" sign represents the decline in performance. Red numbers are better performance scores.

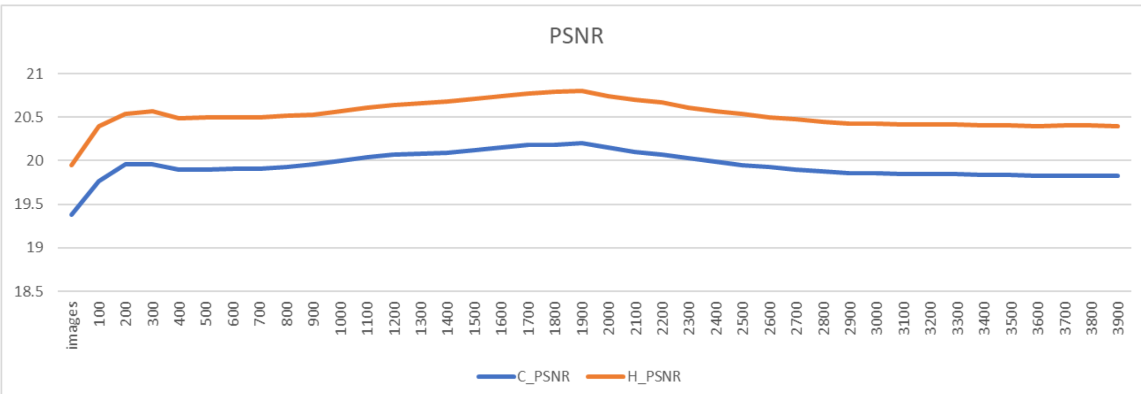


Figure 26: Evaluation of EdgeConnect (Places2) 1

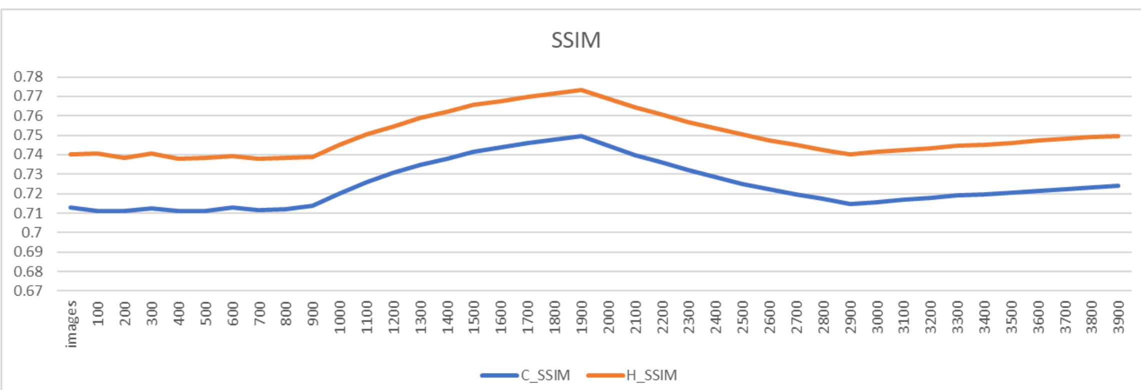


Figure 27: Evaluation of EdgeConnect (Places2) 2

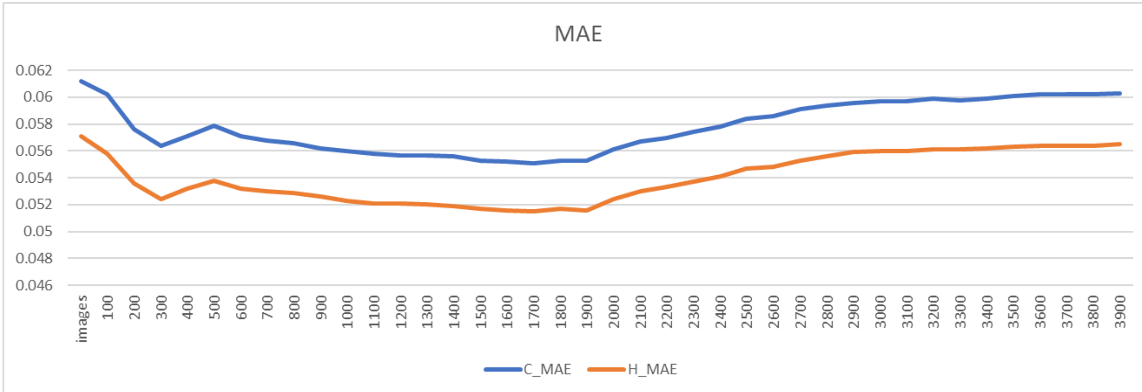


Figure 28: Evaluation of EdgeConnect (Places2) 3

Table 3. The Metrics Score of Canny Edge Detection and HED (Places2)

	PSNR	SSIM	MAE	FID
EdgeConnect (Canny)	19.8260	0.7239	0.0603	34.2741
EdgeConnect (HED)	20.4005	0.7497	0.0565	27.0358
Improvement	+ 2.90%	+ 3.56%	+ 6.3%	+ 21.12%

The "+" sign represents the improvement in performance, and the "-" sign represents the decline in performance. Red numbers are better performance scores

The term peak signal-to-noise ratio (PSNR) is most used to measure the quality of reconstruction of lossy compression codecs (e.g., for image compression). The signal in this case is the original data, and the noise is the error introduced by compression. Typical values for the PSNR in lossy image and video compression are between 30 and 50 dB, provided the bit depth is 8 bits [13] High PSNR means good image quality and less ERROR introduced to the image [14].

$$PSNR = 10log_{10}\left(\frac{(L-1)^2}{MSE}\right) = 20log_{10}\left(\frac{L-1}{RMSE}\right)$$

Equation 12.

The structural similarity index measure (SSIM) measures image similarity in terms of brightness, contrast, and structure, respectively. The value range of SSIM is [0, 1], the larger the value, the smaller the image distortion [15].

$$SSIM\ (x,y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)}$$

Equation 13.

Where μ_x is the average of x; μ_y is the average of y; σ_x^2 is the variance of x; σ_y^2 is the variance of y; σ_{xy} is the covariance of x and y.

$c_1 = (k_1 L)^2, c_2 = (k_2 L)^2$ variables to stabilize the division with weak denominator.

L is the dynamic range of the pixel-values (typically this is $2^{\#bits\ per\ pixel}$ – 1).

$k_1 = 0.01$ and $k_{2=0.03}$ by default.

The mean absolute error (MAE) is used to measure the mean absolute error between the predicted value and the true value. The smaller the MAE, the better the model [16]. It is defined as follows:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y_i}|, MAE\epsilon[0, +\infty)$$

Equation 14.

The Fréchet Inception Distance score (FID) is a measure of calculating the distance between the real image and the feature vector of the generated image, the smaller the index value, the more similar the generated image is to the real image, it can be computed from the mean and the covariance of the activations when the synthesized and real images are fed into the Inception network as [17]:

$$FID = ||\mu - \mu_w||_2^2 + tr\left(\Sigma + \Sigma_w - 2\left(\Sigma^{\frac{1}{2}}\Sigma_w\Sigma^{\frac{1}{2}}\right)^{\frac{1}{2}}\right)$$

Equation 15.

As can be seen from the Table 2, EdgeConnect (HED) is better than EdgeConnect (Canny) in three of the four matrices, and the difference in MAE is only 0.0058, which is not a big difference.

In Table 3, in PSNR, lager on is better, SSIM larger on better, MAE smaller one better, FID, Smaller one better, so, the EdgeConnect with HED is better than Canny's in all four metrics. Therefore, replacing Canny edge detection with HED has a considerable improvement in image inpainting.

CHAPTER FOUR

CONCLUSION AND FUTURE WORK

Improved EdgeConnect

The original EdgeConnect uses Canny edge detection to generate edge maps by default, but it can be seen from the above comparative experiments that a better edge detection can significantly improve the image inpainting algorithm and results. In the project, HED is integrated into EdgeConnect, which improves the effect of edge model and inpainting model and thus makes the effect of image inpainting better.

During the implementation, the HED batch program is added to project, which is outside the EdgeConnect to generate edge maps in batches, and then the training set, test set, and validation set folders for the third-party edge detection reserved by the original author are used to train the edge and inpainting models.

The improved EdgeConnect allows the choice of edge detection: either Canny or HED edge detection.

Therefore, compared with the original EdgeConnect, little has changed in the way the program is used, but the image inpainting quality has been greatly improved. The implementation makes it easier for performance comparison. It also allows integration with other edge detection methods in the future.

Future Work

The following regions can the considered for future work.

1) Increasing the training time and the number of training set allows the model to be better trained to improve the accuracy of image inpainting.
2) Developing a better method to estimate the degree of convergence, alternating Model 2 and Model 3 with regular training might improve the effect of the inpaint model.
3) Using Canny and HED to train alternately in the improved EdgeConnect, integrate the results to see if it can help get better result.
4) The occasional problem that the image resolution is not consistent before and after convolution can be solved in program, for example, by numerical conversion in program.
5) Since HED also uses deep learning, we can improve the accuracy of image inpainting by improving the accuracy of edge detection.
6) Fragmentary functions outside the main program, such as adding masks to images, benchmark, etc., can be integrated into the main program for further automation.

APPENDIX A

CODE

MAIN.PY

```
import os

import cv2

import random

import numpy as np

import torch

import argparse

from shutil import copyfile

from src.config import Config

from src.edge_connect import EdgeConnect



def main(mode=None):

    r"""starts the model


    Args:

        mode (int): 1: train, 2: test, 3: eval, reads from config file if not specified

    """


    config = load_config(mode)



    # cuda visble devices

    os.environ['CUDA_VISIBLE_DEVICES'] = ','.join(str(e) for e in config.GPU)
```

```python
print( os.environ['CUDA_VISIBLE_DEVICES'])



# init device
if torch.cuda.is_available():
    config.DEVICE = torch.device("cuda")
    torch.backends.cudnn.benchmark = True   # cudnn auto-tuner
    print("using GPU")
else:
    config.DEVICE = torch.device("cpu")
    print("using CPU")




# set cv2 running threads to 1 (prevents deadlocks with pytorch dataloader)
cv2.setNumThreads(0)



# initialize random seed
torch.manual_seed(config.SEED)
torch.cuda.manual_seed_all(config.SEED)
np.random.seed(config.SEED)
random.seed(config.SEED)
```

```python
# build the model and initialize
model = EdgeConnect(config)
model.load()


# model training
if config.MODE == 1:
    config.print()
    print('\nstart training...\n')
    model.train()

# model test
elif config.MODE == 2:
    print('\nstart testing...\n')
    model.test()

# eval mode
else:
    print('\nstart eval...\n')
    model.eval()
```

```
def load_config(mode=None):
    r"""loads model config

    Args:
        mode (int): 1: train, 2: test, 3: eval, reads from config file if not specified
    """

    parser = argparse.ArgumentParser()
    parser.add_argument('--path', '--checkpoints', type=str, default='./checkpoints',
help='model checkpoints path (default: ./checkpoints)')
    parser.add_argument('--model', type=int, choices=[1, 2, 3, 4], help='1: edge model, 2:
inpaint model, 3: edge-inpaint model, 4: joint model')

    # test mode
    if mode == 2:
        parser.add_argument('--input', type=str, help='path to the input images directory or an
input image')
        parser.add_argument('--mask', type=str, help='path to the masks directory or a mask
file')
        parser.add_argument('--edge', type=str, help='path to the edges directory or an edge
file')
        parser.add_argument('--output', type=str, help='path to the output directory')

    args = parser.parse_args()
```

```python
config_path = os.path.join(args.path, 'config.yml')

# create checkpoints path if does't exist
if not os.path.exists(args.path):
    os.makedirs(args.path)

# copy config template if does't exist
if not os.path.exists(config_path):
    copyfile('./config.yml.example', config_path)

# load config file
config = Config(config_path)

# train mode
if mode == 1:
    config.MODE = 1
    if args.model:
        config.MODEL = args.model

# test mode
elif mode == 2:
    config.MODE = 2
    config.MODEL = args.model if args.model is not None else 3
    config.INPUT_SIZE = 0
```

```python
        if args.input is not None:
            config.TEST_FLIST = args.input

        if args.mask is not None:
            config.TEST_MASK_FLIST = args.mask

        if args.edge is not None:
            config.TEST_EDGE_FLIST = args.edge

        if args.output is not None:
            config.RESULTS = args.output

    # eval mode
    elif mode == 3:
        config.MODE = 3
        config.MODEL = args.model if args.model is not None else 3

    return config


if __name__ == "__main__":
    main()
```

EDGE_CONNECT.PY

```
import os
from pickle import GLOBAL
import numpy as np
import torch
from torch.utils.data import DataLoader
from .dataset import Dataset, CropLayer
from .models import EdgeModel, InpaintingModel
from .utils import Progbar, create_dir, stitch_images, imsave
from .metrics import PSNR, EdgeAccuracy
import cv2
import time




class EdgeConnect():
    def __init__(self, config):
        self.config = config

        if config.MODEL == 1:
            model_name = 'edge'
        elif config.MODEL == 2:
            model_name = 'inpaint'
```

```
        elif config.MODEL == 3:
            model_name = 'edge_inpaint'
        elif config.MODEL == 4:
            model_name = 'joint'

        self.debug = False
        self.model_name = model_name
        self.edge_model = EdgeModel(config).to(config.DEVICE)
        self.inpaint_model = InpaintingModel(config).to(config.DEVICE)

        self.psnr = PSNR(255.0).to(config.DEVICE)
        self.edgeacc = EdgeAccuracy(config.EDGE_THRESHOLD).to(config.DEVICE)

        # test mode
        if self.config.MODE == 2:
            self.test_dataset = Dataset(config, config.TEST_FLIST, config.TEST_EDGE_FLIST,
config.TEST_MASK_FLIST, augment=False, training=False)
        else:
            self.train_dataset = Dataset(config, config.TRAIN_FLIST, config.TRAIN_EDGE_FLIST,
config.TRAIN_MASK_FLIST, augment=True, training=True)
```

```
        self.val_dataset = Dataset(config, config.VAL_FLIST, config.VAL_EDGE_FLIST,
config.VAL_MASK_FLIST, augment=False, training=True)
        self.sample_iterator = self.val_dataset.create_iterator(config.SAMPLE_SIZE)

    self.samples_path = os.path.join(config.PATH, 'samples')
    self.results_path = os.path.join(config.PATH, 'results')

    if config.RESULTS is not None:
        self.results_path = os.path.join(config.RESULTS)

    if config.DEBUG is not None and config.DEBUG != 0:
        self.debug = True

    self.log_file = os.path.join(config.PATH, 'log_' + model_name + '.dat')

  def load(self):
    if self.config.MODEL == 1:
        self.edge_model.load()

    elif self.config.MODEL == 2:
        self.inpaint_model.load()

    else:
        self.edge_model.load()
```

```
        self.inpaint_model.load()

    def save(self):
        if self.config.MODEL == 1:
            self.edge_model.save()

        elif self.config.MODEL == 2 or self.config.MODEL == 3:
            self.inpaint_model.save()

        else:
            self.edge_model.save()
            self.inpaint_model.save()

    def train(self):
        train_loader = DataLoader(
            dataset=self.train_dataset,
            batch_size=self.config.BATCH_SIZE,
            num_workers=4,
            drop_last=True,
            shuffle=True
        )

        epoch = 0
        keep_training = True
```

```
model = self.config.MODEL

max_iteration = int(float((self.config.MAX_ITERS)))

total = len(self.train_dataset)


if total == 0:

    print('No training data was provided! Check \'TRAIN_FLIST\' value in the configuration file.')

    return


while(keep_training):

    epoch += 1

    print('\n\nTraining epoch: %d' % epoch)


    progbar = Progbar(total, width=20, stateful_metrics=['epoch', 'iter'])


    for items in train_loader:


        self.edge_model.train()

        self.inpaint_model.train()


        images, images_gray, edges, masks = self.cuda(*items)


        # edge model

        if model == 1:

            # train
```

```
        outputs, gen_loss, dis_loss, logs = self.edge_model.process(images_gray, edges, masks)

        # metrics
        precision, recall = self.edgeacc(edges * masks, outputs * masks)
        logs.append(('precision', precision.item()))
        logs.append(('recall', recall.item()))

        # backward
        self.edge_model.backward(gen_loss, dis_loss)
        iteration = self.edge_model.iteration


    # inpaint model
    elif model == 2:
        # train
        outputs, gen_loss, dis_loss, logs = self.inpaint_model.process(images, edges, masks)
        outputs_merged = (outputs * masks) + (images * (1 - masks))

        # metrics
        psnr = self.psnr(self.postprocess(images), self.postprocess(outputs_merged))
        mae = (torch.sum(torch.abs(images - outputs_merged)) / torch.sum(images)).float()
        logs.append(('psnr', psnr.item()))
        logs.append(('mae', mae.item()))
```

```
        # backward
        self.inpaint_model.backward(gen_loss, dis_loss)
        iteration = self.inpaint_model.iteration



    # inpaint with edge model
    elif model == 3:
        # train
        if True or np.random.binomial(1, 0.5) > 0:
            outputs = self.edge_model(images_gray, edges, masks)
            outputs = outputs * masks + edges * (1 - masks)
        else:
            outputs = edges

        outputs, gen_loss, dis_loss, logs = self.inpaint_model.process(images, outputs.detach(),
masks)
        outputs_merged = (outputs * masks) + (images * (1 - masks))

        # metrics
        psnr = self.psnr(self.postprocess(images), self.postprocess(outputs_merged))
        mae = (torch.sum(torch.abs(images - outputs_merged)) / torch.sum(images)).float()
        logs.append(('psnr', psnr.item()))
        logs.append(('mae', mae.item()))
```

```
        # backward
        self.inpaint_model.backward(gen_loss, dis_loss)
        iteration = self.inpaint_model.iteration


    # joint model
    else:
        # train
        e_outputs, e_gen_loss, e_dis_loss, e_logs = self.edge_model.process(images_gray, edges,
masks)
        e_outputs = e_outputs * masks + edges * (1 - masks)
        i_outputs, i_gen_loss, i_dis_loss, i_logs = self.inpaint_model.process(images, e_outputs,
masks)
        outputs_merged = (i_outputs * masks) + (images * (1 - masks))

        # metrics
        psnr = self.psnr(self.postprocess(images), self.postprocess(outputs_merged))
        mae = (torch.sum(torch.abs(images - outputs_merged)) / torch.sum(images)).float()
        precision, recall = self.edgeacc(edges * masks, e_outputs * masks)
        e_logs.append(('pre', precision.item()))
        e_logs.append(('rec', recall.item()))
        i_logs.append(('psnr', psnr.item()))
        i_logs.append(('mae', mae.item()))
        logs = e_logs + i_logs
```

```
        # backward
        self.inpaint_model.backward(i_gen_loss, i_dis_loss)
        self.edge_model.backward(e_gen_loss, e_dis_loss)
        iteration = self.inpaint_model.iteration


    if iteration >= max_iteration:
        keep_training = False
        break

    logs = [
        ("epoch", epoch),
        ("iter", iteration),
    ] + logs

    progbar.add(len(images), values=logs if self.config.VERBOSE else [x for x in logs if not
x[0].startswith('l_')])

    # log model at checkpoints
    if self.config.LOG_INTERVAL and iteration % self.config.LOG_INTERVAL == 0:
        self.log(logs)

    # sample model at checkpoints
```

```
            if self.config.SAMPLE_INTERVAL and iteration % self.config.SAMPLE_INTERVAL == 0:
                self.sample()

            # evaluate model at checkpoints
            if self.config.EVAL_INTERVAL and iteration % self.config.EVAL_INTERVAL == 0:
                print('\nstart eval...\n')
                self.eval()

            # save model at checkpoints
            if self.config.SAVE_INTERVAL and iteration % self.config.SAVE_INTERVAL == 0:
                self.save()

    print('\nEnd training....')

def eval(self):
    val_loader = DataLoader(
        dataset=self.val_dataset,
        batch_size=self.config.BATCH_SIZE,
        drop_last=True,
        shuffle=True
    )

    model = self.config.MODEL
    total = len(self.val_dataset)
```

```python
self.edge_model.eval()
self.inpaint_model.eval()

progbar = Progbar(total, width=20, stateful_metrics=['it'])
iteration = 0

for items in val_loader:

    iteration += 1
    images, images_gray, edges, masks = self.cuda(*items)

    # edge model
    if model == 1:
        # eval
        outputs, gen_loss, dis_loss, logs = self.edge_model.process(images_gray, edges, masks)

        # metrics
        precision, recall = self.edgeacc(edges * masks, outputs * masks)
        logs.append(('precision', precision.item()))
        logs.append(('recall', recall.item()))


    # inpaint model
```

```
elif model == 2:
    # eval
    outputs, gen_loss, dis_loss, logs = self.inpaint_model.process(images, edges, masks)
    outputs_merged = (outputs * masks) + (images * (1 - masks))

    # metrics
    psnr = self.psnr(self.postprocess(images), self.postprocess(outputs_merged))
    mae = (torch.sum(torch.abs(images - outputs_merged)) / torch.sum(images)).float()
    logs.append(('psnr', psnr.item()))
    logs.append(('mae', mae.item()))


# inpaint with edge model
elif model == 3:
    # eval
    outputs = self.edge_model(images_gray, edges, masks)
    outputs = outputs * masks + edges * (1 - masks)

    outputs, gen_loss, dis_loss, logs = self.inpaint_model.process(images, outputs.detach(),
masks)
    outputs_merged = (outputs * masks) + (images * (1 - masks))

    # metrics
    psnr = self.psnr(self.postprocess(images), self.postprocess(outputs_merged))
```

```
            mae = (torch.sum(torch.abs(images - outputs_merged)) / torch.sum(images)).float()
            logs.append(('psnr', psnr.item()))
            logs.append(('mae', mae.item()))


        # joint model
        else:
            # eval
            e_outputs, e_gen_loss, e_dis_loss, e_logs = self.edge_model.process(images_gray, edges,
masks)
            e_outputs = e_outputs * masks + edges * (1 - masks)
            i_outputs, i_gen_loss, i_dis_loss, i_logs = self.inpaint_model.process(images, e_outputs, masks)
            outputs_merged = (i_outputs * masks) + (images * (1 - masks))

            # metrics
            psnr = self.psnr(self.postprocess(images), self.postprocess(outputs_merged))
            mae = (torch.sum(torch.abs(images - outputs_merged)) / torch.sum(images)).float()
            precision, recall = self.edgeacc(edges * masks, e_outputs * masks)
            e_logs.append(('pre', precision.item()))
            e_logs.append(('rec', recall.item()))
            i_logs.append(('psnr', psnr.item()))
            i_logs.append(('mae', mae.item()))
            logs = e_logs + i_logs
```

```
            logs = [("it", iteration), ] + logs
            progbar.add(len(images), values=logs)

    def test(self):
        self.edge_model.eval()
        self.inpaint_model.eval()

        model = self.config.MODEL
        create_dir(self.results_path)

        test_loader = DataLoader(
            dataset=self.test_dataset,
            batch_size=1,
        )

        index = 0
        for items in test_loader:

            name = self.test_dataset.load_name(index)
            images, images_gray, edges, masks = self.cuda(*items)
            index += 1

            # edge model
```

```
if model == 1:
    outputs = self.edge_model(images_gray, edges, masks)
    outputs_merged = (outputs * masks) + (edges * (1 - masks))

# inpaint model
elif model == 2:
    outputs = self.inpaint_model(images, edges, masks)
    outputs_merged = (outputs * masks) + (images * (1 - masks))

# inpaint with edge model / joint model
else:
    edges = self.edge_model(images_gray, edges, masks).detach()
    outputs = self.inpaint_model(images, edges, masks)
    outputs_merged = (outputs * masks) + (images * (1 - masks))

output = self.postprocess(outputs_merged)[0]
path = os.path.join(self.results_path, name)
print(index, name)

imsave(output, path)

if self.debug:
    edges = self.postprocess(1 - edges)[0]
    masked = self.postprocess(images * (1 - masks) + masks)[0]
```

```
            fname, fext = name.split('.')

            imsave(edges, os.path.join(self.results_path, fname + '_edge.' + fext))
            imsave(masked, os.path.join(self.results_path, fname + '_masked.' + fext))

        print('\nEnd test....')

    def sample(self, it=None):
        # do not sample when validation set is empty
        if len(self.val_dataset) == 0:
            return

        self.edge_model.eval()
        self.inpaint_model.eval()

        model = self.config.MODEL
        items = next(self.sample_iterator)
        images, images_gray, edges, masks = self.cuda(*items)

        # edge model
        if model == 1:
            iteration = self.edge_model.iteration
            inputs = (images_gray * (1 - masks)) + masks
            outputs = self.edge_model(images_gray, edges, masks)
```

```
    outputs_merged = (outputs * masks) + (edges * (1 - masks))

# inpaint model
elif model == 2:
    iteration = self.inpaint_model.iteration
    inputs = (images * (1 - masks)) + masks
    outputs = self.inpaint_model(images, edges, masks)
    outputs_merged = (outputs * masks) + (images * (1 - masks))

# inpaint with edge model / joint model
else:
    iteration = self.inpaint_model.iteration
    inputs = (images * (1 - masks)) + masks
    outputs = self.edge_model(images_gray, edges, masks).detach()
    edges = (outputs * masks + edges * (1 - masks)).detach()
    outputs = self.inpaint_model(images, edges, masks)
    outputs_merged = (outputs * masks) + (images * (1 - masks))

if it is not None:
    iteration = it

image_per_row = 2
if self.config.SAMPLE_SIZE <= 6:
    image_per_row = 1
```

```
    images = stitch_images(
        self.postprocess(images),
        self.postprocess(inputs),
        self.postprocess(edges),
        self.postprocess(outputs),
        self.postprocess(outputs_merged),
        img_per_row = image_per_row
    )


    path = os.path.join(self.samples_path, self.model_name)
    name = os.path.join(path, str(iteration).zfill(5) + ".png")
    create_dir(path)
    print('\nsaving sample ' + name)
    images.save(name)

def log(self, logs):
    with open(self.log_file, 'a') as f:
        f.write('%s\n' % ' '.join([str(item[1]) for item in logs]))

def cuda(self, *args):
    return (item.to(self.config.DEVICE) for item in args)
```

```
def postprocess(self, img):
    # [0, 1] => [0, 255]
    img = img * 255.0
    img = img.permute(0, 2, 3, 1)
    return img.int()
```

MODELS.PY

```python
import os
import torch
import torch.nn as nn
import torch.optim as optim
from .networks import InpaintGenerator, EdgeGenerator, Discriminator
from .loss import AdversarialLoss, PerceptualLoss, StyleLoss


class BaseModel(nn.Module):
    def __init__(self, name, config):
        super(BaseModel, self).__init__()

        self.name = name
        self.config = config
        self.iteration = 0

        self.gen_weights_path = os.path.join(config.PATH, name + '_gen.pth')
        self.dis_weights_path = os.path.join(config.PATH, name + '_dis.pth')

    def load(self):
        if os.path.exists(self.gen_weights_path):
            print('Loading %s generator...' % self.name)
```

```
        if torch.cuda.is_available():
            data = torch.load(self.gen_weights_path)
        else:
            data = torch.load(self.gen_weights_path, map_location=lambda storage, loc:
storage)

        self.generator.load_state_dict(data['generator'])
        self.iteration = data['iteration']

    # load discriminator only when training
    if self.config.MODE == 1 and os.path.exists(self.dis_weights_path):
        print('Loading %s discriminator...' % self.name)

        if torch.cuda.is_available():
            data = torch.load(self.dis_weights_path)
        else:
            data = torch.load(self.dis_weights_path, map_location=lambda storage, loc: storage)

        self.discriminator.load_state_dict(data['discriminator'])

def save(self):
    print('\nsaving %s...\n' % self.name)
    torch.save({
        'iteration': self.iteration,
```

```python
            'generator': self.generator.state_dict()
        }, self.gen_weights_path)

        torch.save({
            'discriminator': self.discriminator.state_dict()
        }, self.dis_weights_path)


class EdgeModel(BaseModel):
    def __init__(self, config):
        super(EdgeModel, self).__init__('EdgeModel', config)

        # generator input: [grayscale(1) + edge(1) + mask(1)]
        # discriminator input: (grayscale(1) + edge(1))
        generator = EdgeGenerator(use_spectral_norm=True)
        discriminator = Discriminator(in_channels=2, use_sigmoid=config.GAN_LOSS != 'hinge')
        if len(config.GPU) > 1:
            generator = nn.DataParallel(generator, config.GPU)
            discriminator = nn.DataParallel(discriminator, config.GPU)
        l1_loss = nn.L1Loss()
        adversarial_loss = AdversarialLoss(type=config.GAN_LOSS)

        self.add_module('generator', generator)
        self.add_module('discriminator', discriminator)
```

```
    self.add_module('l1_loss', l1_loss)
    self.add_module('adversarial_loss', adversarial_loss)

    self.gen_optimizer = optim.Adam(
        params=generator.parameters(),
        lr=float(config.LR),
        betas=(config.BETA1, config.BETA2)
    )

    self.dis_optimizer = optim.Adam(
        params=discriminator.parameters(),
        lr=float(config.LR) * float(config.D2G_LR),
        betas=(config.BETA1, config.BETA2)
    )

def process(self, images, edges, masks):
    self.iteration += 1


    # zero optimizers
    self.gen_optimizer.zero_grad()
    self.dis_optimizer.zero_grad()
```

```
        # process outputs
        outputs = self(images, edges, masks)
        gen_loss = 0
        dis_loss = 0


        # discriminator loss
        dis_input_real = torch.cat((images, edges), dim=1)
        dis_input_fake = torch.cat((images, outputs.detach()), dim=1)
        dis_real, dis_real_feat = self.discriminator(dis_input_real)        # in: (grayscale(1) + edge(1))
        dis_fake, dis_fake_feat = self.discriminator(dis_input_fake)        # in: (grayscale(1) +
edge(1))
        dis_real_loss = self.adversarial_loss(dis_real, True, True)
        dis_fake_loss = self.adversarial_loss(dis_fake, False, True)
        dis_loss += (dis_real_loss + dis_fake_loss) / 2


        # generator adversarial loss
        gen_input_fake = torch.cat((images, outputs), dim=1)
        gen_fake, gen_fake_feat = self.discriminator(gen_input_fake)        # in: (grayscale(1) +
edge(1))
        gen_gan_loss = self.adversarial_loss(gen_fake, True, False)
        gen_loss += gen_gan_loss
```

```
        # generator feature matching loss
        gen_fm_loss = 0
        for i in range(len(dis_real_feat)):
            gen_fm_loss += self.l1_loss(gen_fake_feat[i], dis_real_feat[i].detach())
        gen_fm_loss = gen_fm_loss * self.config.FM_LOSS_WEIGHT
        gen_loss += gen_fm_loss


        # create logs
        logs = [
            ("l_d1", dis_loss.item()),
            ("l_g1", gen_gan_loss.item()),
            ("l_fm", gen_fm_loss.item()),
        ]

        return outputs, gen_loss, dis_loss, logs

    def forward(self, images, edges, masks):
        edges_masked = (edges * (1 - masks))
        images_masked = (images * (1 - masks)) + masks
        inputs = torch.cat((images_masked, edges_masked, masks), dim=1)
```

```
        outputs = self.generator(inputs)                                    # in: [grayscale(1) + edge(1) +
mask(1)]
        return outputs

    def backward(self, gen_loss=None, dis_loss=None):
        if dis_loss is not None:
            dis_loss.backward()
        self.dis_optimizer.step()

        if gen_loss is not None:
            gen_loss.backward()
        self.gen_optimizer.step()


class InpaintingModel(BaseModel):
    def __init__(self, config):
        super(InpaintingModel, self).__init__('InpaintingModel', config)

        # generator input: [rgb(3) + edge(1)]
        # discriminator input: [rgb(3)]
        generator = InpaintGenerator()
        discriminator = Discriminator(in_channels=3, use_sigmoid=config.GAN_LOSS != 'hinge')
        if len(config.GPU) > 1:
            generator = nn.DataParallel(generator, config.GPU)
```

```
    discriminator = nn.DataParallel(discriminator , config.GPU)

l1_loss = nn.L1Loss()
perceptual_loss = PerceptualLoss()
style_loss = StyleLoss()
adversarial_loss = AdversarialLoss(type=config.GAN_LOSS)

self.add_module('generator', generator)
self.add_module('discriminator', discriminator)

self.add_module('l1_loss', l1_loss)
self.add_module('perceptual_loss', perceptual_loss)
self.add_module('style_loss', style_loss)
self.add_module('adversarial_loss', adversarial_loss)

self.gen_optimizer = optim.Adam(
    params=generator.parameters(),
    lr=float(config.LR),
    betas=(config.BETA1, config.BETA2)
)

self.dis_optimizer = optim.Adam(
    params=discriminator.parameters(),
    lr=float(config.LR) * float(config.D2G_LR),
```

```
            betas=(config.BETA1, config.BETA2)
        )

    def process(self, images, edges, masks):
        self.iteration += 1

        # zero optimizers
        self.gen_optimizer.zero_grad()
        self.dis_optimizer.zero_grad()


        # process outputs
        outputs = self(images, edges, masks)
        gen_loss = 0
        dis_loss = 0


        # discriminator loss
        dis_input_real = images
        dis_input_fake = outputs.detach()
        dis_real, _ = self.discriminator(dis_input_real)                    # in: [rgb(3)]
        dis_fake, _ = self.discriminator(dis_input_fake)                    # in: [rgb(3)]
        dis_real_loss = self.adversarial_loss(dis_real, True, True)
        dis_fake_loss = self.adversarial_loss(dis_fake, False, True)
```

```
            dis_loss += (dis_real_loss + dis_fake_loss) / 2


            # generator adversarial loss
            gen_input_fake = outputs
            gen_fake, _ = self.discriminator(gen_input_fake)                  # in: [rgb(3)]
            gen_gan_loss = self.adversarial_loss(gen_fake, True, False) *
self.config.INPAINT_ADV_LOSS_WEIGHT
            gen_loss += gen_gan_loss


            # generator l1 loss
            gen_l1_loss = self.l1_loss(outputs, images) * self.config.L1_LOSS_WEIGHT /
torch.mean(masks)
            gen_loss += gen_l1_loss


            # generator perceptual loss
            gen_content_loss = self.perceptual_loss(outputs, images)
            gen_content_loss = gen_content_loss * self.config.CONTENT_LOSS_WEIGHT
            gen_loss += gen_content_loss


            # generator style loss
```

```
        gen_style_loss = self.style_loss(outputs * masks, images * masks)
        gen_style_loss = gen_style_loss * self.config.STYLE_LOSS_WEIGHT
        gen_loss += gen_style_loss


        # create logs
        logs = [
            ("l_d2", dis_loss.item()),
            ("l_g2", gen_gan_loss.item()),
            ("l_l1", gen_l1_loss.item()),
            ("l_per", gen_content_loss.item()),
            ("l_sty", gen_style_loss.item()),
        ]

        return outputs, gen_loss, dis_loss, logs

    def forward(self, images, edges, masks):
        images_masked = (images * (1 - masks).float()) + masks
        inputs = torch.cat((images_masked, edges), dim=1)
        outputs = self.generator(inputs)                                    # in: [rgb(3) + edge(1)]
        return outputs

    def backward(self, gen_loss=None, dis_loss=None):
        dis_loss.backward()
```

```
self.dis_optimizer.step()


gen_loss.backward()
self.gen_optimizer.step()
```

METRICS.PY

```
import numpy as np

import argparse

import matplotlib.pyplot as plt


from glob import glob

from ntpath import basename

from scipy.misc import imread

from skimage.measure import compare_ssim

from skimage.measure import compare_psnr

from skimage.color import rgb2gray




def parse_args():

    parser = argparse.ArgumentParser(description='script to compute all statistics')

    parser.add_argument('--data-path', help='Path to ground truth data', type=str)

    parser.add_argument('--output-path', help='Path to output data', type=str)

    parser.add_argument('--debug', default=0, help='Debug', type=int)

    args = parser.parse_args()

    return args




def compare_mae(img_true, img_test):

    img_true = img_true.astype(np.float32)
```

```
    img_test = img_test.astype(np.float32)
    return np.sum(np.abs(img_true - img_test)) / np.sum(img_true + img_test)


args = parse_args()
for arg in vars(args):
    print('[%s] =' % arg, getattr(args, arg))

path_true = args.data_path
path_pred = args.output_path

psnr = []
ssim = []
mae = []
names = []
index = 1

files = list(glob(path_true + '/*.jpg')) + list(glob(path_true + '/*.png'))
for fn in sorted(files):
    name = basename(str(fn))
    names.append(name)

    img_gt = (imread(str(fn)) / 255.0).astype(np.float32)
    img_pred = (imread(path_pred + '/' + basename(str(fn))) / 255.0).astype(np.float32)
```

```
img_gt = rgb2gray(img_gt)
img_pred = rgb2gray(img_pred)

if args.debug != 0:
    plt.subplot('121')
    plt.imshow(img_gt)
    plt.title('Groud truth')
    plt.subplot('122')
    plt.imshow(img_pred)
    plt.title('Output')
    plt.show()

psnr.append(compare_psnr(img_gt, img_pred, data_range=1))
ssim.append(compare_ssim(img_gt, img_pred, data_range=1, win_size=51))
mae.append(compare_mae(img_gt, img_pred))
if np.mod(index, 100) == 0:
    print(
        str(index) + ' images processed',
        "PSNR: %.4f" % round(np.mean(psnr), 4),
        "SSIM: %.4f" % round(np.mean(ssim), 4),
        "MAE: %.4f" % round(np.mean(mae), 4),
    )
index += 1
```

```
np.savez(args.output_path + '/metrics.npz', psnr=psnr, ssim=ssim, mae=mae, names=names)

print(

    "PSNR: %.4f" % round(np.mean(psnr), 4),

    "PSNR Variance: %.4f" % round(np.var(psnr), 4),

    "SSIM: %.4f" % round(np.mean(ssim), 4),

    "SSIM Variance: %.4f" % round(np.var(ssim), 4),

    "MAE: %.4f" % round(np.mean(mae), 4),

    "MAE Variance: %.4f" % round(np.var(mae), 4)

)
```

FID_SCORE.PY

```
import os
import pathlib
from argparse import ArgumentParser, ArgumentDefaultsHelpFormatter

import torch
import numpy as np
from scipy.misc import imread
from scipy import linalg
from torch.autograd import Variable
from torch.nn.functional import adaptive_avg_pool2d

from inception import InceptionV3



parser = ArgumentParser(formatter_class=ArgumentDefaultsHelpFormatter)
parser.add_argument('--path', type=str, nargs=2, help=('Path to the generated images or
to .npz statistic files'))
parser.add_argument('--batch-size', type=int, default=64, help='Batch size to use')
parser.add_argument('--dims', type=int, default=2048,
choices=list(InceptionV3.BLOCK_INDEX_BY_DIM), help=('Dimensionality of Inception features to use. By
default, uses pool3 features'))
parser.add_argument('-c', '--gpu', default='', type=str, help='GPU to use (leave blank for CPU
only)')
```

```python
def get_activations(images, model, batch_size=64, dims=2048,
                    cuda=False, verbose=False):
    """Calculates the activations of the pool_3 layer for all images.
    Params:
    -- images      : Numpy array of dimension (n_images, 3, hi, wi). The values
                     must lie between 0 and 1.
    -- model       : Instance of inception model
    -- batch_size  : the images numpy array is split into batches with
                     batch size batch_size. A reasonable batch size depends
                     on the hardware.
    -- dims        : Dimensionality of features returned by Inception
    -- cuda        : If set to True, use GPU
    -- verbose     : If set to True and parameter out_step is given, the number
                     of calculated batches is reported.
    Returns:
    -- A numpy array of dimension (num images, dims) that contains the
       activations of the given tensor when feeding inception with the
       query tensor.
    """
    model.eval()

    d0 = images.shape[0]
```

```
if batch_size > d0:
    print(('Warning: batch size is bigger than the data size. '
           'Setting batch size to data size'))
    batch_size = d0

n_batches = d0 // batch_size
n_used_imgs = n_batches * batch_size

pred_arr = np.empty((n_used_imgs, dims))
for i in range(n_batches):
    if verbose:
        print('\rPropagating batch %d/%d' % (i + 1, n_batches),
              end='', flush=True)
    start = i * batch_size
    end = start + batch_size

    batch = torch.from_numpy(images[start:end]).type(torch.FloatTensor)
    batch = Variable(batch, volatile=True)
    if cuda:
        batch = batch.cuda()

    pred = model(batch)[0]

    # If model output is not scalar, apply global spatial average pooling.
```

```
        # This happens if you choose a dimensionality not equal 2048.
        if pred.shape[2] != 1 or pred.shape[3] != 1:
            pred = adaptive_avg_pool2d(pred, output_size=(1, 1))

        pred_arr[start:end] = pred.cpu().data.numpy().reshape(batch_size, -1)

    if verbose:
        print(' done')

    return pred_arr


def calculate_frechet_distance(mu1, sigma1, mu2, sigma2, eps=1e-6):
    """Numpy implementation of the Frechet Distance.
    The Frechet distance between two multivariate Gaussians X_1 ~ N(mu_1, C_1)
    and X_2 ~ N(mu_2, C_2) is
            d^2 = ||mu_1 - mu_2||^2 + Tr(C_1 + C_2 - 2*sqrt(C_1*C_2)).
    Stable version by Dougal J. Sutherland.
    Params:
    -- mu1   : Numpy array containing the activations of a layer of the
               inception net (like returned by the function 'get_predictions')
               for generated samples.
    -- mu2   : The sample mean over activations, precalculated on an
               representive data set.
```

```
-- sigma1: The covariance matrix over activations for generated samples.
-- sigma2: The covariance matrix over activations, precalculated on an
        representive data set.
Returns:
--   : The Frechet Distance.
"""

mu1 = np.atleast_1d(mu1)
mu2 = np.atleast_1d(mu2)

sigma1 = np.atleast_2d(sigma1)
sigma2 = np.atleast_2d(sigma2)

assert mu1.shape == mu2.shape, \
    'Training and test mean vectors have different lengths'
assert sigma1.shape == sigma2.shape, \
    'Training and test covariances have different dimensions'

diff = mu1 - mu2

# Product might be almost singular
covmean, _ = linalg.sqrtm(sigma1.dot(sigma2), disp=False)
if not np.isfinite(covmean).all():
    msg = ('fid calculation produces singular product; '
```

```
                  'adding %s to diagonal of cov estimates') % eps
        print(msg)
        offset = np.eye(sigma1.shape[0]) * eps
        covmean = linalg.sqrtm((sigma1 + offset).dot(sigma2 + offset))

    # Numerical error might give slight imaginary component
    if np.iscomplexobj(covmean):
        if not np.allclose(np.diagonal(covmean).imag, 0, atol=1e-3):
            m = np.max(np.abs(covmean.imag))
            raise ValueError('Imaginary component {}'.format(m))
        covmean = covmean.real

    tr_covmean = np.trace(covmean)

    return (diff.dot(diff) + np.trace(sigma1) +
            np.trace(sigma2) - 2 * tr_covmean)


def calculate_activation_statistics(images, model, batch_size=64,
                                    dims=2048, cuda=False, verbose=False):
    """Calculation of the statistics used by the FID.
    Params:
    -- images      : Numpy array of dimension (n_images, 3, hi, wi). The values
                     must lie between 0 and 1.
```

```
    -- model       : Instance of inception model
    -- batch_size  : The images numpy array is split into batches with
                     batch size batch_size. A reasonable batch size
                     depends on the hardware.
    -- dims        : Dimensionality of features returned by Inception
    -- cuda        : If set to True, use GPU
    -- verbose     : If set to True and parameter out_step is given, the
                     number of calculated batches is reported.
    Returns:
    -- mu    : The mean over samples of the activations of the pool_3 layer of
               the inception model.
    -- sigma : The covariance matrix of the activations of the pool_3 layer of
               the inception model.
    """
    act = get_activations(images, model, batch_size, dims, cuda, verbose)
    mu = np.mean(act, axis=0)
    sigma = np.cov(act, rowvar=False)
    return mu, sigma


def _compute_statistics_of_path(path, model, batch_size, dims, cuda):
    npz_file = os.path.join(path, 'statistics.npz')
    if os.path.exists(npz_file):
        f = np.load(npz_file)
```

```
        m, s = f['mu'][:], f['sigma'][:]
        f.close()
    else:
        path = pathlib.Path(path)
        files = list(path.glob('*.jpg')) + list(path.glob('*.png'))

        imgs = np.array([imread(str(fn)).astype(np.float32) for fn in files])

        # Bring images to shape (B, 3, H, W)
        imgs = imgs.transpose((0, 3, 1, 2))

        # Rescale images to be between 0 and 1
        imgs /= 255

        m, s = calculate_activation_statistics(imgs, model, batch_size, dims, cuda)
        np.savez(npz_file, mu=m, sigma=s)

    return m, s


def calculate_fid_given_paths(paths, batch_size, cuda, dims):
    """Calculates the FID of two paths"""
    for p in paths:
        if not os.path.exists(p):
```

```python
            raise RuntimeError('Invalid path: %s' % p)

    block_idx = InceptionV3.BLOCK_INDEX_BY_DIM[dims]

    model = InceptionV3([block_idx])
    if cuda:
        model.cuda()

    print('calculate path1 statistics...')
    m1, s1 = _compute_statistics_of_path(paths[0], model, batch_size, dims, cuda)
    print('calculate path2 statistics...')
    m2, s2 = _compute_statistics_of_path(paths[1], model, batch_size, dims, cuda)
    print('calculate frechet distance...')
    fid_value = calculate_frechet_distance(m1, s1, m2, s2)

    return fid_value


if __name__ == '__main__':
    args = parser.parse_args()
    os.environ['CUDA_VISIBLE_DEVICES'] = args.gpu

    fid_value = calculate_fid_given_paths(args.path,
                                          args.batch_size,
```

```
                                  args.gpu != '',
                                  args.dims)
print('FID: ', round(fid_value, 4))
```

MASKIMG.PY

```python
# Required Libraries

import cv2

import numpy as np

from os import listdir

from os.path import isfile, join

from pathlib import Path

import argparse

import numpy


# Argument parsing variable declared

ap = argparse.ArgumentParser()


ap.add_argument("-i", "--image",

                required=True,

                help="Path to folder")

ap.add_argument("-e", "--mask",

                required=True,

                help="Path to folder")


args = vars(ap.parse_args())



# Find all the images in the provided images folder
```

```
mypath1 = args["image"]

mypath2 = args["mask"]

onlyfiles1 = [f for f in listdir(mypath1) if isfile(join(mypath1, f))]

onlyfiles2 = [f for f in listdir(mypath2) if isfile(join(mypath2, f))]

images = numpy.empty(len(onlyfiles1), dtype=object)

masks = numpy.empty(len(onlyfiles2), dtype=object)


# Iterate through every image

# and resize all the images.

for n in range(0, len(onlyfiles1)):


        path1 = join(mypath1, onlyfiles1[n])

        path2 = join(mypath2, onlyfiles2[n])

        images[n] = cv2.imread(join(mypath1, onlyfiles1[n]),

                                        cv2.IMREAD_UNCHANGED)

        masks[n] = cv2.imread(join(mypath2, onlyfiles2[n]),

                                        cv2.IMREAD_UNCHANGED)

        # Load the image in img variable

        img = cv2.imread(path1, 1)

        msk= cv2.imread(path2, 1)

        resize_width = int(256)

        resize_hieght = int(256)

        resized_dimensions = (resize_width, resize_hieght)

        resized_msk = cv2.resize(msk, resized_dimensions, interpolation=cv2.INTER_AREA)
```

```
        # Define a resizing Scale

        # To declare how much to resize

        mask_img = cv2.bitwise_or(resized_msk, img)


        # Create resized image using the calculated dimensions


        # Save the image in Output Folder

        cv2.imwrite(

        'output/' + str(n) + '_resized.png', mask_img)


print("Images masked Successfully")
```

HED_PROCESSING.PY

```python
import cv2 as cv

import os

import numpy as np

import time




# ! [CropLayenr]

class CropLayer(object):

    def __init__(self, params, blobs):

        self.xstart = 0

        self.xend = 0

        self.ystart = 0

        self.yend = 0


    # Our layer receives two inputs. We need to crop the first input blob

    # to match a shape of the second one (keeping batch size and number of channels)

    def getMemoryShapes(self, inputs):

        inputShape, targetShape = inputs[0], inputs[1]

        batchSize, numChannels = inputShape[0], inputShape[1]

        height, width = targetShape[2], targetShape[3]


        # self.ystart = (inputShape[2] - targetShape[2]) / 2

        # self.xstart = (inputShape[3] - targetShape[3]) / 2
```

```
        self.ystart = int((inputShape[2] - targetShape[2]) / 2)
        self.xstart = int((inputShape[3] - targetShape[3]) / 2)

        self.yend = self.ystart + height
        self.xend = self.xstart + width

        return [[batchSize, numChannels, height, width]]

    def forward(self, inputs):
        return [inputs[0][:, :, self.ystart:self.yend, self.xstart:self.xend]]


def hed(net, start_paths, target_paths):
    width = 256
    height = 256
    for start_path_i in range(len(start_paths)):
        s_path = start_paths[start_path_i]
        t_path = target_paths[start_path_i]
        if not os.path.exists(t_path):
            os.makedirs(t_path)
        image_lists = [os.path.join(s_path, i) for i in os.listdir(s_path)]
        size = len(image_lists)
        for img_i, img_path in enumerate(image_lists):
```

```
        if '.jpg' not in img_path.lower() and '.png' not in img_path.lower():
            continue
        if img_i % 10 == 0:
            print(f'{t_path} finish {img_i}/{size}.')
        frame = cv.imread(img_path)

        inp = cv.dnn.blobFromImage(frame, scalefactor=1.0, size=(width, height),
                                   mean=(104.00698793, 116.66876762, 122.67891434),
                                   swapRB=False, crop=False)
        net.setInput(inp)

        out = net.forward()
        out = out[0, 0]
        out = cv.resize(out, (frame.shape[1], frame.shape[0]))
        out = out * 255
        cv.imwrite(os.path.join(t_path, img_path[img_path.rfind('\\')+1:]), out.astype('uint8'))
        time.sleep(0.05)
    return


def flist(paths, outputs):
    ext = {'.JPG', '.JPEG', '.PNG', '.TIF', 'TIFF'}
    for path_i, path in enumerate(paths):
        output = outputs[path_i]
```

```
        images = []
        for root, dirs, files in os.walk(path):
            print('loading ' + root)
            for file in files:
                if os.path.splitext(file)[1].upper() in ext:
                    images.append(os.path.join(root, file))

        images = sorted(images)
        np.savetxt(output, images, fmt='%s')
    return


if __name__ == '__main__':
    # ! [CropLayer]

    # ! [Register]
    cv.dnn_registerLayer('Crop', CropLayer)
    # ! [Register]

    # Load the model.
    prototxt_path = 'deploy.prototxt'
    caffemodel_path = 'hed_pretrained_bsds.caffemodel'
    net = cv.dnn.readNet(cv.samples.findFile(prototxt_path),
cv.samples.findFile(caffemodel_path))
```

```
start_paths = ['training/cat_train', 'training/cat_test_original', 'training/cat_val']
target_paths = ['training/cat_edges_train', 'training/cat_edges_test', 'training/cat_edges_val']
hed(net, start_paths, target_paths)

outputs = ['datasets/cat_edges_train.flist', 'datasets/cat_edges_test.flist',
'datasets/cat_edges_val.flist']
flist(target_paths, outputs)
```

DATASET.PY

```python
import os

import glob

import scipy

import torch

import random

import numpy as np

import torchvision.transforms.functional as F

from torch.utils.data import DataLoader

from PIL import Image

from scipy.misc import imread

from skimage.feature import canny

from skimage.color import rgb2gray, gray2rgb

from .utils import create_mask

import cv2



class CropLayer(object):

    def __init__(self, params, blobs):

        self.xstart = 0

        self.xend = 0

        self.ystart = 0

        self.yend = 0
```

```
    # Our layer receives two inputs. We need to crop the first input blob
    # to match a shape of the second one (keeping batch size and number of channels)
    def getMemoryShapes(self, inputs):
        inputShape, targetShape = inputs[0], inputs[1]
        batchSize, numChannels = inputShape[0], inputShape[1]
        height, width = targetShape[2], targetShape[3]

        # self.ystart = (inputShape[2] - targetShape[2]) / 2
        # self.xstart = (inputShape[3] - targetShape[3]) / 2

        self.ystart = int((inputShape[2] - targetShape[2]) / 2)
        self.xstart = int((inputShape[3] - targetShape[3]) / 2)

        self.yend = self.ystart + height
        self.xend = self.xstart + width

        return [[batchSize, numChannels, height, width]]

    def forward(self, inputs):
        return [inputs[0][:, :, self.ystart:self.yend, self.xstart:self.xend]]


# hed network
```

```
global net_hed
cv2.dnn_registerLayer('Crop', CropLayer)
prototxt_path = 'deploy.prototxt'
caffemodel_path = 'hed_pretrained_bsds.caffemodel'
net_hed = cv2.dnn.readNet(cv2.samples.findFile(prototxt_path),
cv2.samples.findFile(caffemodel_path))
```

```
class Dataset(torch.utils.data.Dataset):
    def __init__(self, config, flist, edge_flist, mask_flist, augment=True, training=True):
        super(Dataset, self).__init__()
        self.augment = augment
        self.training = training
        self.data = self.load_flist(flist)
        self.edge_data = self.load_flist(edge_flist)
        self.mask_data = self.load_flist(mask_flist)
        self.input_size = config.INPUT_SIZE
        self.sigma = config.SIGMA
        self.edge = config.EDGE
        self.mask = config.MASK
        self.nms = config.NMS
```

```
        # in test mode, there's a one-to-one relationship between mask and image
        # masks are loaded non random
        if config.MODE == 2:
            self.mask = 6

    def __len__(self):
        return len(self.data)

    def __getitem__(self, index):
        try:
            item = self.load_item(index)
        except:
            print('loading error: ' + self.data[index])
            item = self.load_item(0)

        return item

    def load_name(self, index):
        name = self.data[index]
        return os.path.basename(name)

    def load_item(self, index):

        size = self.input_size
```

```
# load image
img = imread(self.data[index])

# gray to rgb
if len(img.shape) < 3:
    img = gray2rgb(img)

# resize/crop if needed
if size != 0:
    img = self.resize(img, size, size)

# create grayscale image
img_gray = rgb2gray(img)

# load mask
mask = self.load_mask(img, index)

# load edge
edge = self.load_edge(img_gray, img, index, mask)

# augment data
if self.augment and np.random.binomial(1, 0.5) > 0:
    img = img[:, ::-1, ...]
```

```
            img_gray = img_gray[:, ::-1, ...]
            edge = edge[:, ::-1, ...]
            mask = mask[:, ::-1, ...]

        return self.to_tensor(img), self.to_tensor(img_gray), self.to_tensor(edge), 
self.to_tensor(mask)

    def load_edge(self, img, img_ori, index, mask):
        sigma = self.sigma

        # in test mode images are masked (with masked regions),
        # using 'mask' parameter prevents canny to detect edges for the masked regions
        mask = None if self.training else (1 - mask / 255).astype(np.bool)

        # canny
        if self.edge == 1:
            # no edge
            if sigma == -1:
                return np.zeros(img.shape).astype(np.float)

            # random sigma
            if sigma == 0:
                sigma = random.randint(1, 4)
            return canny(img, sigma=sigma, mask=mask).astype(np.float)
```

```
# external
else:
    imgh, imgw = img.shape[0:2]
    if len(self.edge_data) != 0:
        edge = imread(self.edge_data[index])
    else:
        width = 256
        height = 256
        img_input = cv2.cvtColor(img_ori, cv2.COLOR_RGB2BGR)
        frame = img_input.copy()

        inp = cv2.dnn.blobFromImage(frame, scalefactor=1.0, size=(width, height),
                                    mean=(104.00698793, 116.66876762, 122.67891434),
                                    swapRB=False, crop=False)
        net_hed.setInput(inp)

        out = net_hed.forward()
        out = out[0, 0]
        out = cv2.resize(out, (frame.shape[1], frame.shape[0]))
        edge = out.copy()
    edge = self.resize(edge, imgh, imgw)

    # non-max suppression
```

```python
        if self.nms == 1:
            edge = edge * canny(img, sigma=sigma, mask=mask)

        return edge

    def load_mask(self, img, index):
        imgh, imgw = img.shape[0:2]
        mask_type = self.mask

        # external + random block
        if mask_type == 4:
            mask_type = 1 if np.random.binomial(1, 0.5) == 1 else 3

        # external + random block + half
        elif mask_type == 5:
            mask_type = np.random.randint(1, 4)

        # random block
        if mask_type == 1:
            return create_mask(imgw, imgh, imgw // 2, imgh // 2)

        # half
        if mask_type == 2:
            # randomly choose right or left
```

```
            return create_mask(imgw, imgh, imgw // 2, imgh, 0 if random.random() < 0.5 else imgw // 2, 0)

        # external
        if mask_type == 3:
            mask_index = random.randint(0, len(self.mask_data) - 1)
            mask = imread(self.mask_data[mask_index])
            mask = self.resize(mask, imgh, imgw)
            mask = (mask > 0).astype(np.uint8) * 255       # threshold due to interpolation
            return mask

        # test mode: load mask non random
        if mask_type == 6:
            mask = imread(self.mask_data[index])
            mask = self.resize(mask, imgh, imgw, centerCrop=False)
            mask = rgb2gray(mask)
            mask = (mask > 0).astype(np.uint8) * 255
            return mask

    def to_tensor(self, img):
        img = Image.fromarray(img)
        img_t = F.to_tensor(img).float()
        return img_t
```

```python
def resize(self, img, height, width, centerCrop=True):
    imgh, imgw = img.shape[0:2]

    if centerCrop and imgh != imgw:
        # center crop
        side = np.minimum(imgh, imgw)
        j = (imgh - side) // 2
        i = (imgw - side) // 2
        img = img[j:j + side, i:i + side, ...]

    img = scipy.misc.imresize(img, [height, width])

    return img

def load_flist(self, flist):
    if isinstance(flist, list):
        return flist

    # flist: image file path, image directory path, text file flist path
    if isinstance(flist, str):
        if os.path.isdir(flist):
            flist = list(glob.glob(flist + '/*.jpg')) + list(glob.glob(flist + '/*.png'))
            flist.sort()
            return flist
```

```
        if os.path.isfile(flist):
            try:
                return np.genfromtxt(flist, dtype=np.str, encoding='utf-8')
            except:
                return [flist]

    return []

def create_iterator(self, batch_size):
    while True:
        sample_loader = DataLoader(
            dataset=self,
            batch_size=batch_size,
            drop_last=True
        )

        for item in sample_loader:
            yield item
```

APPENDIX B

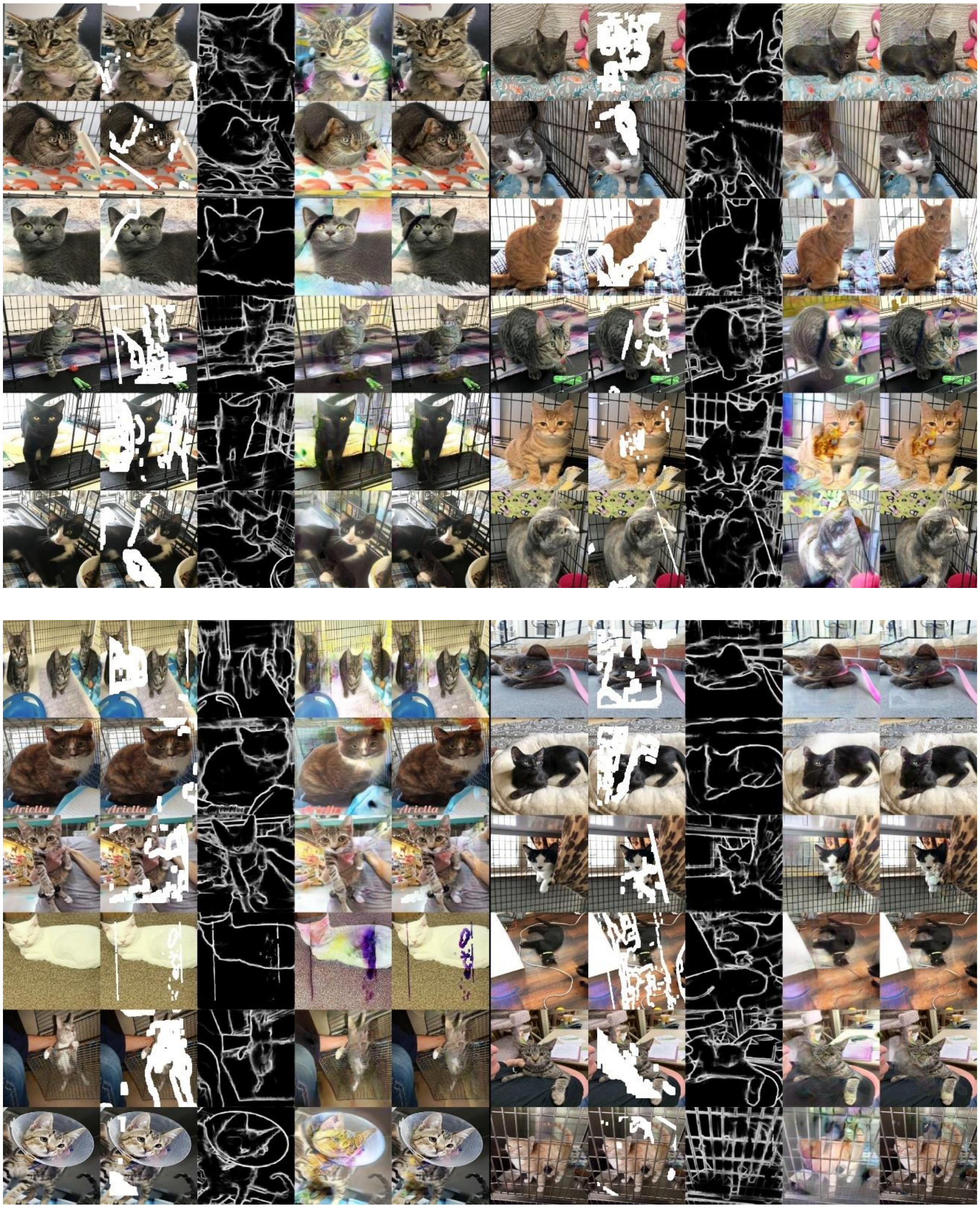
OUTPUT SAMPLE

OUTPUT SAMPLE (CANNY)

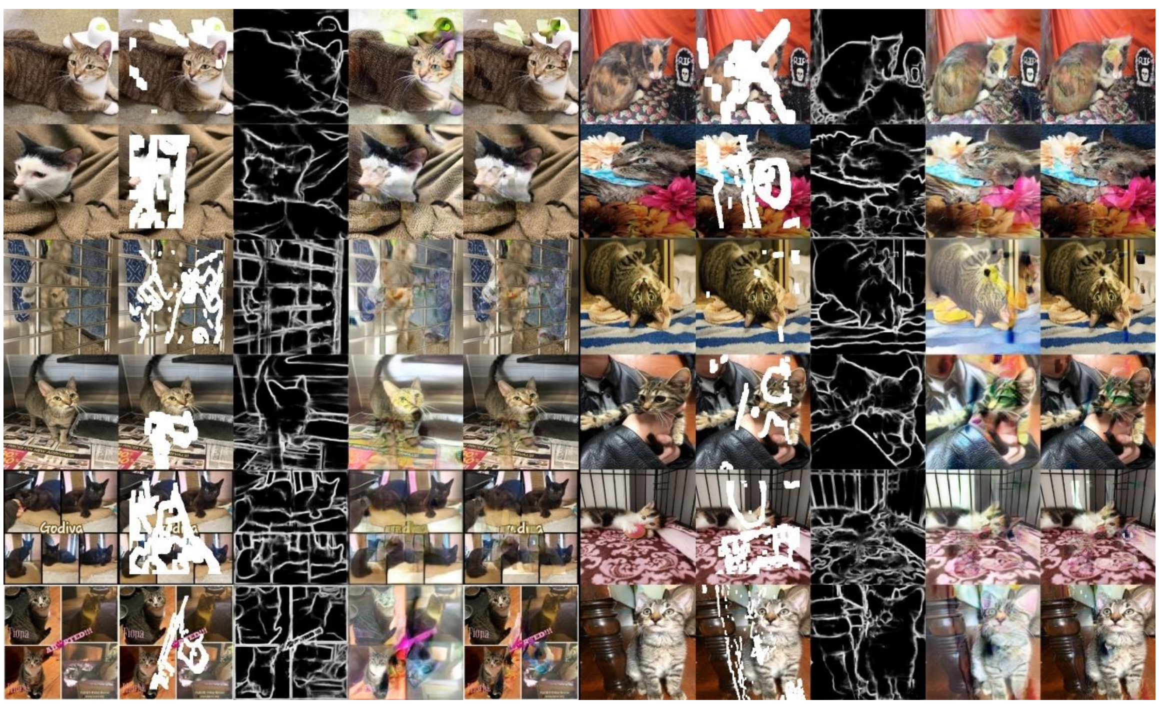

Declawed


ROSY

OUTPUT SAMPLE (HED)


Ariella


Reece


Godiva
Fiona


Pearl
ELLIOTT

REFERENCES

[1] Nazeri, K., Ng, E., Joseph, T., Qureshi, F. Z., and Ebrahimi, M., "EdgeConnect: Generative Image Inpainting with Adversarial Edge Learning", 2019.

[2] Xie, S. and Tu, Z., "Holistically-Nested Edge Detection", 2015.

[3] Johnson, J., Alahi, A., Fei-Fei, L. (2016). Perceptual Losses for Real-Time Style Transfer and Super-Resolution. In: Leibe, B., Matas, J., Sebe, N., Welling, M. (eds) Computer Vision – ECCV 2016. ECCV 2016. Lecture Notes in Computer Science(), vol 9906. Springer, Cham. https://doi.org/10.1007/978-3-319-46475-6_43

[4] Isola, P., Zhu, J.-Y., Zhou, T., and Efros, A. A., "Image-to-Image Translation with Conditional Adversarial Networks", 2016.

[5] Ulyanov, D., Vedaldi, A., and Lempitsky, V., "Instance Normalization: The Missing Ingredient for Fast Stylization", 2016.

[6] E. Million, "The Hadamard product Elizabeth million April 12, 2007 1 introduction and basic results," 2007.

[7] Sajjadi, M. S. M., Schölkopf, B., and Hirsch, M., "EnhanceNet: Single Image Super-Resolution Through Automated Texture Synthesis", 2016.

[8] T. Chu, "“Lines First, Color Next” An Inspirational Deep Image Inpainting Approach", Medium, 2022. [Online]. Available: https://towardsdatascience.com/lines-first-color-next-an-inspirational-deep-image-inpainting-approach-b2d980efb364. [Accessed: 18- Apr- 2022]

[9] B. Zhou, A. Lapedriza, A. Khosla, A. Oliva, and A. Torralba. Places: A 10 million image database for scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2017.

[10] Crawford, C. and Nian., “Cat Dataset”, 2018, https://www.kaggle.com/datasets/crawford/cat-dataset?datasetId=13371&sortBy=dateRun&tab=profile

[11] MA7555., “Cat Breeds Dataset”, 2020, https://www.kaggle.com/datasets/ma7555/cat-breeds-dataset

[12] Iskakov, K., “QD-IMD: Quick Draw Irregular Mask Dataset” , 2018, QD-IMD, https://github.com/karfly/qd-imd

[13] Faragallah O. S. et al., "A Comprehensive Survey Analysis for Present Solutions of Medical Image Fusion and Future Directions," in IEEE Access, vol. 9, pp. 11358-11371, 2021, doi: 10.1109/ACCESS.2020.3048315.

[14] "Python | Peak Signal-to-Noise Ratio (PSNR) - GeeksforGeeks", GeeksforGeeks, 2022. [Online]. Available: https://www.geeksforgeeks.org/python-peak-signal-to-noise-ratio-psnr/. [Accessed: 18- Apr- 2022]

[15] "Structural similarity - Wikipedia", En.wikipedia.org, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Structural_similarity. [Accessed: 18- Apr- 2022]

[16] "Mean absolute error - Wikipedia", En.wikipedia.org, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Mean_absolute_error. [Accessed: 18- Apr- 2022]

[17] "Fréchet inception distance - Wikipedia", *En.wikipedia.org*, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Fr%C3%A9chet_inception_distance. [Accessed: 18- Apr- 2022]