8-2022

DEEP LEARNING EDGE DETECTION IN IMAGE INPAINTING

Zheng Zheng

Follow this and additional works at: https://scholarworks.lib.csusb.edu/etd

Part of the Artificial Intelligence and Robotics Commons, and the Software Engineering Commons

Recommended Citation
https://scholarworks.lib.csusb.edu/etd/1536

This Project is brought to you for free and open access by the Office of Graduate Studies at CSUSB ScholarWorks. It has been accepted for inclusion in Electronic Theses, Projects, and Dissertations by an authorized administrator of CSUSB ScholarWorks. For more information, please contact scholarworks@csusb.edu.
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
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

ABSTRACT .......................................................................................................................... iii

ACKNOWLEDGEMENTS ...................................................................................................... v

LIST OF TABLES .................................................................................................................. viii

LIST OF FIGURES ................................................................................................................ ix

LIST OF EQUATIONS .......................................................................................................... xi

CHAPTER ONE: INTRODUCTION

Background ............................................................................................................................. 1

Objectives .............................................................................................................................. 2

CHAPTER TWO: EDGECONNECT ......................................................................................... 5

Runtime Environment ......................................................................................................... 5

Program ................................................................................................................................ 6

Edge Generator .................................................................................................................... 6

Image Completion Network .................................................................................................. 8

Model training ....................................................................................................................... 10

Edge Model Training ........................................................................................................... 10

Inpaint Model Training ....................................................................................................... 11

Edge-Inpaint Training ......................................................................................................... 11

Model Testing ....................................................................................................................... 12

Evaluation ............................................................................................................................. 13

Summary ............................................................................................................................... 13

CHAPTER THREE: EXPERIMENT ....................................................................................... 15

Preparation Work ................................................................................................................ 15
Preprocessing........................................................................................................ 15
Dataset.................................................................................................................. 16
HED......................................................................................................................... 17
Structure ............................................................................................................... 17
Loss Function ...................................................................................................... 18
Comparison ......................................................................................................... 19
Edge Model Training .......................................................................................... 19
Inpaint Model Training ....................................................................................... 26
Edge-Inpaint Training ......................................................................................... 32
Evaluation ............................................................................................................ 38

CHAPTER FOUR: CONCLUSION AND FUTURE WORK ........................................ 44
Improved EdgeConnect ....................................................................................... 44
Future Work ......................................................................................................... 44

APPENDIX A: CODE ............................................................................................ 46

APPENDIX B: OUTPUT SAMPLE .......................................................................... 121
Output Sample (Canny) ...................................................................................... 121
Output Sample (HED) ......................................................................................... 126

REFERENCES ...................................................................................................... 130
LIST OF TABLES

Table 1. Package List ................................................................. 5
Table 2. The Metrics Score of Canny Edge Detection and HED (cat) ............... 40
Table 3. The Metrics Score of Canny Edge Detection and HED (Places2) ........ 41
LIST OF FIGURES

Figure 1. Edgeconnect Samples................................................................. 2
Figure 2. Edgeconnect Network Structure.................................................. 6
Figure 3. HED Network Structure. ............................................................. 17
Figure 4. Edge Model Training Sample (Canny, Cat)................................... 20
Figure 5. Edge Model Training Sample (HED, Cat)................................. 21
Figure 6. Precision of Canny and HED During the Edge Model Training (Cat) .. 22
Figure 7. Recall of Canny and HED During the Edge Model Training (Cat) ..... 23
Figure 8. Edge Model Training Sample (Canny, Places2).......................... 24
Figure 9. Edge Model Training Sample (HED, Places2)............................ 25
Figure 10. Precision of Canny and HED During the Edge Model Training (Places2) ......................................................................................................... 26
Figure 11. Recall of Canny and HED During the Edge Model Training (Places2) ......................................................................................................... 26
Figure 12. Inpaint Model Training Sample (Canny, Cat).............................. 27
Figure 13. Inpaint Model Training Sample (HED, Cat)............................... 28
Figure 14. Inpaint Model Training Sample (Canny, Places2).................... 29
Figure 15. Inpaint Model Training Sample (HED, Places2)....................... 30
Figure 16. PSNR of Canny and HED During the Inpaint Model Training (Places2) ........................................................................................................... 31
Figure 17. MAE of Canny and HED During the Inpaint Model Training (Places2) ........................................................................................................... 31
Figure 18. Edge-Inpaint Training Sample (Canny, Cat) ................................................. 33
Figure 19. Edge-Inpaint Training Sample (HED, Cat) ..................................................... 35
Figure 20. Edge-Inpaint Training Sample (Canny, Places2) ......................................... 36
Figure 21. Edge-Inpaint Training Sample (HED, Places2) ............................................ 37
Figure 22. PSNR of Canny and HED During the Edge Inpaint Model Training
(Places2) ....................................................................................................................... 38
Figure 23. MAE of Canny and HED During the Edge Inpaint Model Training
(Places2) ....................................................................................................................... 38
Figure 24. Evaluation of Edgeconnect (Canny, Cat) ....................................................... 39
Figure 25. Evaluation of Edgeconnect (HED, Cat) ......................................................... 39
Figure 26: Evaluation of Edgeconnect (Places2) 1 ......................................................... 40
Figure 27: Evaluation of Edgeconnect (Places2) 2 ......................................................... 40
Figure 28: Evaluation of Edgeconnect (Places2) 3 ......................................................... 41
LIST OF EQUATIONS

<table>
<thead>
<tr>
<th>Equation</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>18</td>
</tr>
<tr>
<td>10</td>
<td>18</td>
</tr>
<tr>
<td>11</td>
<td>18</td>
</tr>
<tr>
<td>12</td>
<td>42</td>
</tr>
<tr>
<td>13</td>
<td>42</td>
</tr>
<tr>
<td>14</td>
<td>42</td>
</tr>
<tr>
<td>15</td>
<td>43</td>
</tr>
</tbody>
</table>
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 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>
<thead>
<tr>
<th>site-packages</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>matplotlib</td>
<td>2.2.5</td>
</tr>
<tr>
<td>numpy</td>
<td>1.21.5</td>
</tr>
<tr>
<td>opencv-python</td>
<td>3.4.17.63</td>
</tr>
<tr>
<td>Pillow</td>
<td>6.2.1</td>
</tr>
<tr>
<td>PyYAML</td>
<td>5.4.1</td>
</tr>
<tr>
<td>scikit-image</td>
<td>0.14.5</td>
</tr>
<tr>
<td>scipy</td>
<td>1.2.3</td>
</tr>
<tr>
<td>pytorch</td>
<td>1.0.0</td>
</tr>
<tr>
<td>torchvision</td>
<td>0.2.1</td>
</tr>
</tbody>
</table>

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.

⊙ is hadamard product, for two matrices A and B of the same dimension $m \times n$, 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 ($\bar{I}_{gray}$) and edge map with missing regions ($\bar{C}_{gt}$) and mark it with a wavy line on the letter:

\[
\bar{I}_{gray} = I_{gray} \odot (1 - M)
\]

\[
\bar{C}_{gt} = \bar{C}_{gt} \odot (1 - M)
\]

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

$\bar{I}_{gt} = I_{gt} \odot (1 - M)$, $\bar{I}_{gt}$ 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:

\[
C_{pred} = G_1(\bar{I}_{gray}, \bar{C}_{gt}, M)
\]

The following loss function is constructed to train this adversarial network to obtain the edge generator [1]:
\[ \mathcal{L}_{adv,1} = E_{(c_{gt}, l_{gray})} \log[D_1(C_{gt}, l_{gray})] + E_{l_{gray}} \log[1 - D_1(C_{pred}, l_{grey})] \]

Equation 1.

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

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]:

\[ \min_{G_1} \max_{D_1} \mathcal{L}_{G_1} = \min_{G_1} \max_{D_1} \left( \lambda_{adv,1} \mathcal{L}_{adv,1} + \lambda_{FM} \mathcal{L}_{FM} \right) \]

Equation 3.

\[ \lambda_{adv,1} = 1, \lambda_{FM} = 10 \]

Image Completion Network

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

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

Equation 4.

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

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

\[
L_{\text{adv,2}} = E(I_{\text{gt}}, C_{\text{comp}}) \log[D_2(I_{\text{gt}}, C_{\text{comp}})] + E_{\text{comp}} \log[1 - D_2(I_{\text{pred}}, C_{\text{comp}})]
\]

Equation 5.

\(L_{\text{adv,2}}\) is adversarial loss.

\[
L_{\text{prec}} = E \left[ \sum_i \frac{1}{N_i} \| \phi_{1}^{(i)}(I_{\text{gt}}) - \phi_{1}^{(i)}(I_{\text{pred}}) \|_1 \right]
\]

Equation 6.

\(L_{\text{prec}}\) is perceptual loss, the input images are discriminated using the pre-trained VGG-19 network.

\[
L_{\text{style}} = E_j \left[ \| G_j^\phi(I_{\text{pred}}) - G_j^\phi(I_{\text{gt}}) \|_1 \right]
\]

Equation 7.

\(L_{\text{style}}\) is style loss. The \(G_j^\phi\) in Equation 7. is a Gram Matrix of \(C_j \times C_j\) constructed on the activation function eigenmap \(\phi_j\) [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 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, $402/4 = 100.5 \approx 100$, but $100 \times 4 = 400$, which means 100 doesn’t equal to 100.5.

**Inpaint model training**

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.

**Edge-inpaint training**

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

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.

\( M \) 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, \( \hat{Y} \) is edge map prediction, \( \text{Dist}(\cdot, \cdot) \) 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,

\[
L_{\text{side}}(W, w) = \sum_{m=1}^{M} \alpha_m \ell_{\text{side}}(W, \omega^{(m)})
\]

*Equation 9.*

\[
L_{\text{fuse}}(W, w, h) = \text{Dist}(Y, \hat{Y}_{\text{fuse}})
\]

*Equation 10.*

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

\[
(W, w, h)^* = \text{argmin} \left( L_{\text{side}}(W, w) + L_{\text{fuse}}(W, w, h) \right)
\]

*Equation 11.*
Comparison

The purpose of this experiment 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)
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 detection.
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)](image)
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, Places2)
Figure 9. Edge Model Training Sample (HED, Places2)
Inpaint model training

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)
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.

**Edge-inpaint training**

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.
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.
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)
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:
Figure 24. Evaluation of Edgeconnect (Canny, Cat)

Figure 25. Evaluation of Edgeconnect (HED, Cat)
Table 2. The Metrics Score of Canny Edge Detection and HED (Cat)

<table>
<thead>
<tr>
<th></th>
<th>PSNR</th>
<th>SSIM</th>
<th>MAE</th>
<th>FID</th>
</tr>
</thead>
<tbody>
<tr>
<td>EdgeConnect (Canny)</td>
<td>20.0498</td>
<td>0.7600</td>
<td>0.0536</td>
<td>47.9557</td>
</tr>
<tr>
<td>EdgeConnect (HED)</td>
<td>20.4113</td>
<td>0.7779</td>
<td>0.0594</td>
<td>33.3415</td>
</tr>
<tr>
<td>Improvement</td>
<td>+ 1.8%</td>
<td>+ 2.3%</td>
<td>- 10.8%</td>
<td>+ 30.47%</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>PSNR</th>
<th>SSIM</th>
<th>MAE</th>
<th>FID</th>
</tr>
</thead>
<tbody>
<tr>
<td>EdgeConnect (Canny)</td>
<td>19.8260</td>
<td>0.7239</td>
<td>0.0603</td>
<td>34.2741</td>
</tr>
<tr>
<td>EdgeConnect (HED)</td>
<td>20.4005</td>
<td>0.7497</td>
<td>0.0565</td>
<td>27.0358</td>
</tr>
<tr>
<td>Improvement</td>
<td>+ 2.90%</td>
<td>+ 3.56%</td>
<td>+ 6.3%</td>
<td>+ 21.12%</td>
</tr>
</tbody>
</table>

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

*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 \( \mu_x \) is the average of \( x \); \( \mu_y \) is the average of \( y \); \( \sigma_x^2 \) is the variance of \( x \); \( \sigma_y^2 \) is the variance of \( y \); \( \sigma_{xy} \) is the covariance of \( x \) and \( y \).

\[ c_1 = (k_1L)^2, \quad c_2 = (k_2L)^2 \] variables to stabilize the division with weak denominator.

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

\( k_1 = 0.01 \) 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:

\[ MAE = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i|, \quad MAE \in [0, +\infty) \]

*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]:

\[
FID = ||\mu - \mu_w||^2 + tr \left( \Sigma + \Sigma_w - 2 \left( \Sigma^{1/2} \Sigma_w \Sigma^{1/2} \right)^{1/2} \right)
\]

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, larger 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
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 visible devices
    os.environ['CUDA_VISIBLE_DEVICES'] = ','.join(str(e) for e in config.GPU)
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)
# build the model and initialize

model = EdgeConnect(config)

model.load()

# model training

if config.MODE == 1:
    config.print()
    print("start training...
")
    model.train()

# model test

elif config.MODE == 2:
    print("start testing...
")
    model.test()

# eval mode

else:
    print("start eval...
")
    model.eval()
```python
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 doesn't exist
if not os.path.exists(args.path):
    os.makedirs(args.path)

# copy config template if doesn'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
```
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()
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()
```python
    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.\n')
    return

while(keep_training):
    epoch += 1
    print('Training 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('_')])

# 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('
start eval...
')
    self.eval()

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

print('
End 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)
self.edge_model.eval()
self.inpaint_model.eval()

progbar = Progbar(total=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()

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('End 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
' % ' '.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()
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('saving %s...\n' % self.name)
    torch.save(
        {'iteration': self.iteration,
        })
class EdgeModel(BaseModel):
    def __init__(self, config):
        super(EdgeModel, self).__init__(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

```python
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

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

```python
return outputs, gen_loss, dis_loss, logs
```

```python
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()
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] = %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)
)
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)')
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(’Propagating 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 $X_1 \sim N(\mu_1, C_1)$
    and $X_2 \sim N(\mu_2, C_2)$ is
    
    $d^2 = ||\mu_1 - \mu_2||^2 + \text{Tr}(C_1 + C_2 - 2\sqrt{C_1C_2}).$
    
    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
             representative data set.
-- sigma1: The covariance matrix over activations for generated samples.
-- sigma2: The covariance matrix over activations, precalculated on an
       representative data set.

Returns:
-- : The Frechet Distance.

```python
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):
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))
# Required Libraries

```python
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

```python
ap = argparse.ArgumentParser()
```

```python
ap.add_argument("-i", "--image",
                required=True,
                help="Path to folder")
```

```python
ap.add_argument("-e", "--mask",
                required=True,
                help="Path to folder")
```

```python
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")
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']

fist(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
```

110
# 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

```python
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
```

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

```python
def load_item(self, index):
    size = self.input_size
```
# load image

```python
img = imread(self.data[index])
```

# gray to rgb

```python
if len(img.shape) < 3:
    img = gray2rgb(img)
```

# resize/crop if needed

```python
if size != 0:
    img = self.resize(img, size, size)
```

# create grayscale image

```python
img_gray = rgb2gray(img)
```

# load mask

```python
mask = self.load_mask(img, index)
```

# load edge

```python
edge = self.load_edge(img_gray, img, index, mask)
```

# augment data

```python
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
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
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
OUTPUT SAMPLE (CANNY)
OUTPUT SAMPLE (HED)
REFERENCES


