TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

PULCHOWK CAMPUS

A

PROJECT REPORT

ON

IMAGE INPAINTING USING ENCODER-DECODER NETWORK WITH PARTIAL CONVOLUTION

SUBMITTED BY:

ANISH SAPKOTA (076BCT008)

KUSHAL SUBEDI (076BCT031)

NABIN KHANAL (076BCT036)

SUBMITTED TO:

DEPARTMENT OF ELECTRONICS & COMPUTER ENGINEERING

April, 2024

Page of Approval

TRIBHUVAN UNIVERSIY
INSTITUTE OF ENGINEERING
PULCHOWK CAMPUS
DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING

The undersigned certifies that they have read and recommended to the Institute of Engineering for acceptance of a project report entitled **"Image Inpainting Using Encoder-Decoder Network With Partial Convolution"** submitted by **Anish Sapkota, Kushal Subedi and Nabin Khanal** in partial fulfillment of the requirements for the Bachelor's degree in Electronics & Computer Engineering.

.....
Supervisor
Mrs. Anku Jaiswal
Assistant Professor, IC Chair
Department of Electronics and Computer Engineering,
Pulchowk Campus, IOE, TU.

.....
External examiner
Mr. Deepen Chapagain
Country Director
Logpoint, Nepal

. . . . . .
Data of Approval

Copyright

Acknowledgments

We would like to express our sincere appreciation to the Department of Electronics and Computer Engineering, Pulchowk Campus, for granting us the opportunity to work on this proposal. The support and resources provided by the department have been invaluable in successfully completing this proposal.

We would also like to extend our heartfelt gratitude to Asst.Prof. Anku Jaiswal, our esteemed supervisor, for her exceptional guidance, insightful suggestions, and continuous encouragement throughout this journey. Her expertise and mentorship have played a crucial role in shaping this project and deepening our understanding of the subject matter.

Additionally, We would like to acknowledge all the teachers in the department who have shared their knowledge and expertise, laying a solid foundation for our academic growth. Their dedication to teaching and commitment to excellence have significantly contributed to our understanding of Electronics and Computer Engineering.

Lastly, We are grateful to our friends and colleagues who have supported us throughout this endeavor. Their encouragement, constructive feedback, and collaborative spirit have enriched our learning experience and made this proposal a collective effort.

Abstract

Image inpainting is a challenging task in computer vision, aiming to fill in missing or corrupted regions in an image seamlessly and coherently. Traditional convolutional neural networks (CNNs) often struggle to inpaint such regions without introducing artifacts or blurriness. To address this issue, partial convolution has emerged as a promising technique for image inpainting. In partial convolution, each pixel of the convolution kernel is associated with a weight determined by a binary mask, where valid pixels have a weight of 1 and missing or corrupted pixels have a weight of 0. During the convolution process, only the valid pixels contribute to the output, and the convolution values are normalized by the sum of the valid weights. This technique allows the network to focus on reliable information while effectively inpainting the missing regions. In this report, we explore the abstraction of image inpainting using partial convolution. We discuss the input requirements, the partial convolution operation, feature extraction, encoding and decoding processes, reconstruction of the missing regions, and the optimization strategy.

Contents

List of Figures

List of Tables

List of Abbreviations

CNN	Convolutional Neural Network
GAN	Generative Adversial Network
PSNR	Peak Signal-To-Noise Ratio
SSIM	Structural Similarity Index
CAE	Convolutional Autoencoder
MSE	Mean Square Error
FMM	Fast Marching Methods
PDE	Partial Differential Equations
FMM	Fast Marching Methods
ADAM	Adaptive Moment Estimation

1. Introduction

1.1 Background

Image inpainting is a fundamental task in computer vision that involves filling in missing or corrupted regions of an image to restore its visual completeness. It finds applications in various domains, such as image restoration, object removal, and video editing.
Traditional approaches for image inpainting often rely on handcrafted algorithms or simple interpolation methods, which may lead to unrealistic or blurry results. In recent years, deep learning techniques, particularly convolutional neural networks (CNNs), have shown remarkable success in addressing this challenge.

Figure 1.1: Example of image Inpainting

1.2 Problem Statement

Despite the advancements in deep learning-based image inpainting, there still exist limitations in producing high-quality results. Traditional convolutions used in CNNs tend to treat missing or corrupted regions equally, leading to inaccurate or unnatural inpainting. Furthermore, these methods struggle to preserve fine details and handle complex image structures. To overcome these limitations, there is a need to explore alternative convolutional operations that can effectively inpaint missing regions while preserving image coherence and realism.

1.3 Objective

To fill in missing or corrupted parts of an image in a visually plausible and coherent manner by automatically generating content that seamlessly blends with the surrounding areas, making it difficult to discern the inpainted regions from the original image.

1.4 Scope

This project aims to reconstruct the human image by filling in the corrupted regions using the deep learning technology. The scope of the project includes the following:

1. Data Collection: Gather the image set of people which includes diverse facial poses, expressions and lighting conditions.
2. Data Preprocessing: Normalize the image pixels and resize the image to make it suitable for feeding the model. Also, generate the corresponding masked image.
3. Model Development: Develop a predictive model based on U-Net Architecture. Train the model using face images and corresponding masked image to capture the facial pattern.
4. Model Validation and Fine-tuning: Validate the developed model using independent datasets and ground-truth data. Assess the accuracy, robustness, and generalizability of the model across different facial poses, expressions and lighting conditions. Employ suitable performance metrics for model evaluation.

2. Literature Review

2.1 Related work

The field of image inpainting has witnessed significant advancements in recent years, driven by the application of deep learning techniques. Various approaches have been proposed to tackle the challenge of effectively filling in missing or corrupted regions in images. In this section, we provide an overview of some notable works in the field of image inpainting.

- **"Context Encoders: Feature Learning by Inpainting" by Pathak et al. (2016):** This seminal work introduced the concept of using generative adversarial networks (GANs) for image inpainting. The authors proposed a context encoder that learns to inpaint missing regions by training a convolutional neural network (CNN) with an adversarial loss. The network takes the surrounding context as input and generates plausible predictions for the missing regions. [1]
- **"Image Inpainting via Generative Multi-column Convolutional Neural Networks" by Wang et al. (2018):** This work proposed a multi-column convolutional neural network for image inpainting. The network consists of multiple branches, each responsible for capturing different spatial scales of the image. By integrating features from multiple columns, the model produces visually coherent and semantically meaningful inpainted images. [2]
- **"EdgeConnect: Generative Image Inpainting with Adversarial Edge Learning" by Nazeri et al. (2019):** The EdgeConnect model introduced an edge-aware inpainting approach by leveraging the edges of the image to guide the inpainting process. It incorporates an edge generator network that focuses on reconstructing the missing edges, followed by a completion network that generates the filled-in regions using both the edge information and the surrounding context. [3]
- **"DeepFillv2: Free-Form Image Inpainting with Gated Convolution" by Yu et al. (2018):** DeepFillv2 proposed a free-form image inpainting method that allows users to specify the desired inpainting region through arbitrary irregular masks. The model employs a gated convolutional network that adaptively propagates information from the known context to fill in the missing regions. It achieved

state-of-the-art performance in handling irregular and large inpainting areas. [4]

- **"Image Inpainting for Irregular Holes Using Partial Convolutions":** The paper "Image Inpainting for Irregular Holes Using Partial Convolutions" presents a novel approach to address the challenge of inpainting irregularly shaped holes in images. The authors of this paper propose a technique based on partial convolutions, which adaptively combine information from both the observed and missing parts of the image. By modulating the convolutional filters using a mask that reflects the validity of surrounding pixels, the proposed method effectively prevents artifacts and blurring in the inpainted regions. [5]

2.2 Related theory

Image inpainting is a technique used in image processing and computer vision to fill in missing or corrupted parts of an image with plausible content. It involves automatically generating pixels or regions that seamlessly blend with the surrounding areas, creating a visually coherent and realistic appearance. The inpainting process aims to restore or complete the missing information in an image based on the surrounding context and the available visual cues. It can be applied to various tasks, such as image restoration, object removal, completion of occluded regions, visual effects, data augmentation, or privacy protection. The goal of image inpainting is to produce visually convincing results that are indistinguishable from the original image to the human eye.

Some traditional image inpainting methods are:

- Patch-based Inpainting: This method breaks down the inpainting process into patches, where patches from known regions of the image are used to fill in the missing regions. Similar patches are searched in the known areas and then used to estimate the missing content. Patch-based inpainting can be done using algorithms like exemplar-based inpainting, where patches are copied from similar regions, or texture synthesis, where patches are generated based on the surrounding texture. [6]

- Fast Marching Methods: Fast Marching Methods (FMM) use the concept of level set evolution to propagate information from known regions to unknown regions. By defining a speed function that guides the propagation, FMM can estimate the missing content based on the available information. FMM is particularly effective for filling in small, well-defined regions. [7]

- Diffusion-based Methods: Diffusion-based inpainting methods use the concept of diffusion equations to propagate information into the missing regions. The diffusion

process helps in blending the known and unknown regions smoothly, leading to visually coherent results. Examples of diffusion-based methods include the Anisotropic Diffusion and the Heat Equation methods. [8]

- Total Variation-based Methods: Total Variation (TV) inpainting methods aim to minimize the total variation of the image while preserving the known regions. By minimizing the variation, these methods encourage piecewise smoothness in the inpainted regions. TV inpainting is effective in preserving sharp edges and textures. [9]
- Inpainting with Partial Differential Equations (PDEs): PDE-based methods model the inpainting problem as a partial differential equation and solve it iteratively. They incorporate various techniques such as variational models, level set methods, or reaction-diffusion equations to estimate the missing content based on the image's gradient or texture information. [10]

Some fundamentals of Deep Learning for Image Inpainting are:

1. Convolutional neural network: IA Convolutional Neural Network (CNN) is a type of deep learning model designed specifically for analyzing and processing structured data. They are based on supervised learning method which consists of multiple layers of convolutional and pooling operations, which help capture local patterns and extract meaningful features from the input image. These features are then used to reconstruct the restored image.
   In a Convolutional Neural Network (CNN), the image is first passed through a convolutional layer. The convolution operation involves applying weighted filters to the input image to extract local features. The resulting convolved output is then fed into a pooling layer, which reduces the spatial dimensions of the features while preserving important information. This process of convolution and pooling is repeated to obtain a series of two-dimensional tensors.
   Before entering the fully connected layer, the two-dimensional tensors are flattened into a one-dimensional form. This transformation ensures compatibility with the fully connected layer, which performs the final classification task. In the convolutional layers, 3x3 submatrices are commonly used as convolutional filters to extract features, and the output from the pooling layer corresponds to the input of the fully connected layer.
2. Autoencoder Network[11]: Autoencoders are unsupervised neural networks that learn to efficiently represent input data. They consist of an encoder that reduces the dimensionality of the input and a decoder that reconstructs the original data.

By mapping the input to a lower-dimensional representation and then decoding it back, autoencoders can perform tasks such as dimensionality reduction and data compression.

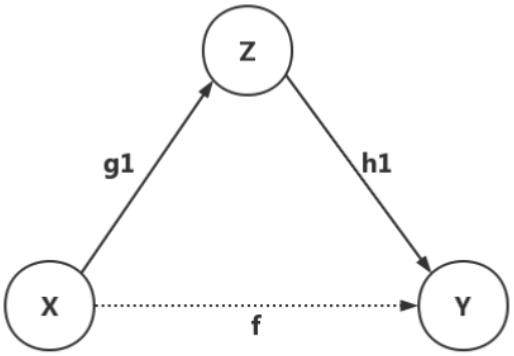


Figure 2.1: The working principle of the autoencoder.

The goal of the autoencoder neural network is to learn to map f: X $\rightarrow$ Y by unsupervised learning, network f consists of two parts the g1: Encoder Network and the h1: Decoder Network, g1 reduces the dimension of high-dimension input X code to low-dimension hidden variable z and realizes the dimension reduction of data. h1 is regarded as the process of data decoding, and the encoded Z input is decoded into the high dimension Y, which realizes the elevation of data dimension. Encoder and decoder together complete the process of data X encoding and decoding, the whole network is called a autoencoder.

Image inpainting using partial convolution is a deep learning-based approach that was introduced in the paper "Image Inpainting for Irregular Holes Using Partial Convolutions" by Liu et al. (2018). This method leverages the concept of partial convolutions to effectively fill in missing regions in an image.

The key idea behind partial convolutions is that the convolution operation is performed differently for known and unknown regions. In traditional convolutions, all pixels in the receptive field contribute to the output, but in partial convolutions, only the pixels in the known region are considered.

Here's a brief overview of the theory behind image inpainting using partial convolutions:

1. Masked Convolution: In partial convolutions, a binary mask is created to indicate the known and unknown regions of the image. The mask is typically obtained by setting pixels in the known region to 1 and pixels in the unknown region to 0. The input image is then multiplied element-wise with the mask, effectively masking out

the unknown regions.

2. Convolution and Normalization: After masking the input image, a convolution operation is performed. However, unlike traditional convolutions, only the pixels in the known region are used for the convolution computation. The convolution result is then divided by the sum of mask values at each location to normalize the output.

3. Handling Unknown Regions: In the masked convolution, the convolution output is only computed for known regions. However, for pixels in the unknown region, the output of the convolution operation is not valid. To address this, the authors introduced a mechanism to estimate the alpha mask, which represents the proportion of valid convolution output at each location. This alpha mask is learned during the training process using additional convolutional layers.

4. Inpainting Process: During the inpainting process, the partial convolutional network takes the input image with holes and the corresponding mask as inputs. The network performs forward propagation, producing an output that fills in the missing regions. The partial convolutions effectively incorporate information from the known regions to generate visually plausible content for the unknown regions.

5. Training: The partial convolutional network is trained using a combination of reconstruction loss and adversarial loss. The reconstruction loss measures the pixel-wise difference between the network output and the ground truth complete image. The adversarial loss encourages the network to generate inpainted regions that are visually realistic by training a discriminator network to distinguish between the inpainted images and the ground truth images.

By leveraging partial convolutions and training a deep neural network, image inpainting using partial convolution can effectively fill in missing regions of an image with visually coherent content.

3. Methodology

The methodology employed in our project focuses on the image inpainting through the implementation of partial convolution in UNET based autoencoder network. This section outlines the key steps, processes, and tools involved in the development and evaluation of our image inpaiting model.

3.1 Data Collection

Data acquisition is a critical aspect of any image inpainting project, as the performance of the inpainting model heavily relies on the quality and diversity of the dataset used for training. In this study, we leveraged the CelebA dataset due to its widespread use in facial image-related tasks and its comprehensive coverage of facial variations.

CelebA Dataset: The CelebA dataset is a large-scale face attributes dataset that contains over 200,000 celebrity images, each annotated with 40 attribute labels. These attributes include aspects like gender, age, and presence of accessories such as eyeglasses and hats. The dataset encompasses diverse facial poses, expressions, and lighting conditions, making it suitable for training deep learning models for various computer vision tasks, including image inpainting.

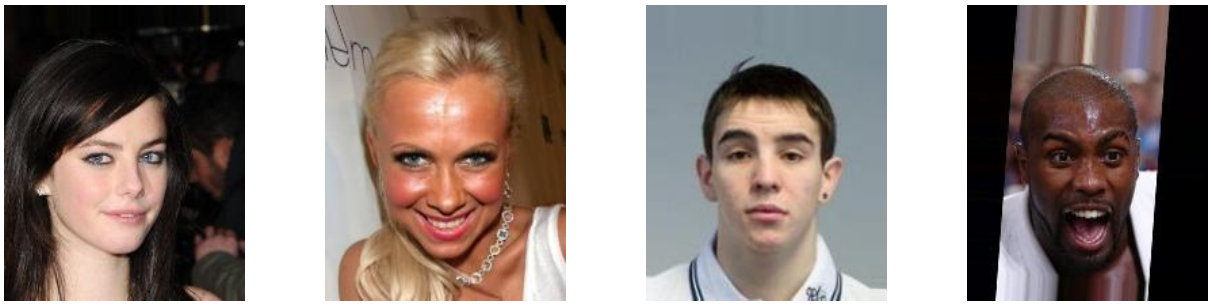


Figure 3.1: Sample Images from CelebA Dataset

3.2 Data Preprocessing

Before utilizing the CelebA dataset for training our inpainting model, several preprocessing steps were performed to ensure data consistency and quality:

1. Normalization: To bring all pixel values within a standardized range, thus preventing any particular feature from dominating the learning process due to differences

in scale, we normalized the image pixel values between 0 and 1 using Pytorch's built-in normalization functions. [12]

2. Resolution Adjustment: To ensure uniformity in input image sizes, we resized all facial images to a predefined resolution 178*178, specifically chosen to balance computational efficiency and inpainting quality.

3. Data Augmentation: To increase the diversity of our training dataset and enhance the robustness of our model, we applied data augmentation techniques such as random rotations, flips, and brightness adjustments.

3.3 Masked Dataset Preparation

The CelebA dataset serves as a valuable resource for training image-related tasks due to its large-scale collection of celebrity images with attribute annotations. However, one limitation of the CelebA dataset for our specific task of image inpainting is the absence of masked images. Image inpainting involves filling in missing or damaged regions of an image, a task that requires both the original images and corresponding masked versions.

To address this limitation, we employed data processing techniques to generate masked images from the original CelebA dataset. This involved applying binary masks to selected regions of the facial images, simulating scenarios where portions of the face are occluded or missing. By creating these masked images, we augmented the dataset with the additional information necessary for training our image inpainting model.

3.3.1 Mask Generation and Application

Before applying masks to the images, we generated mask images corresponding to the regions to be inpainted. These masks were carefully designed to cover specific areas of interest, such as occluded facial features or unwanted objects, while preserving essential image components like facial boundaries and background context.

Mask Application Procedure:
The mask application procedure involved the following steps:

1. Loading Images: We loaded the preprocessed facial images from the CelebA dataset using OpenCV's image loading functionalities.

2. Creating Masks: Using OpenCV, we created binary mask images corresponding to the areas of interest that needed to be inpainted. These masks were generated based on predetermined criteria, such as the location and size of the missing regions.

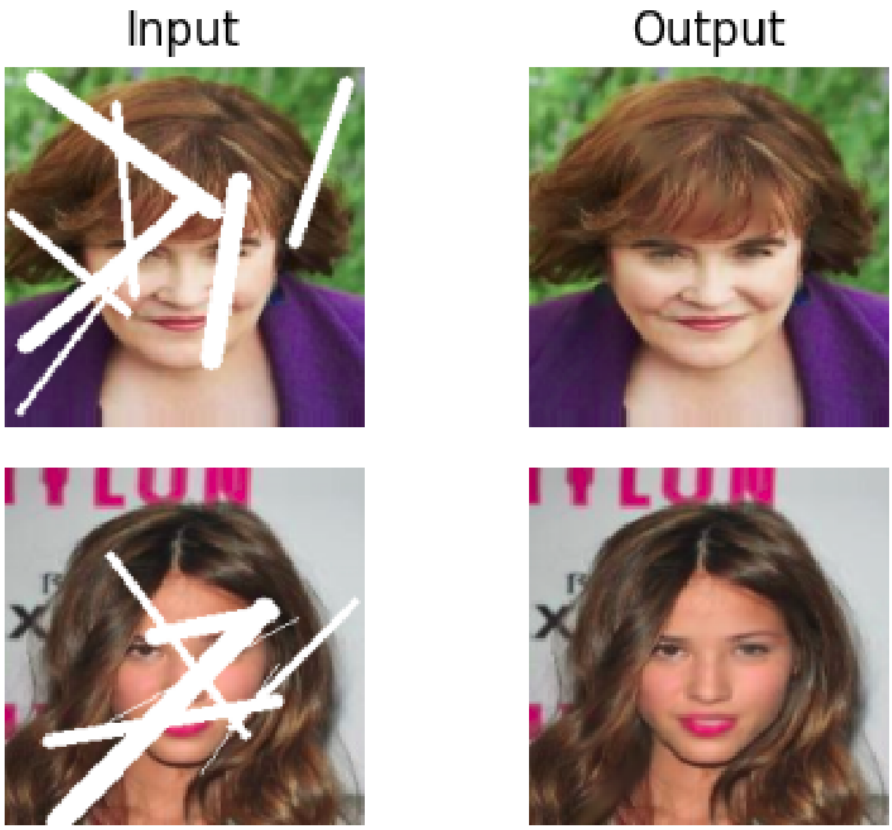


Figure 3.3: Application of binary masks to CelebA dataset to create labeled dataset

Figure 3.2: Randomly generated line masks using OpenCV

3. Applying Masks: With the generated masks prepared, we applied them to the loaded facial images. This process involved overlaying the binary masks onto the original images, effectively masking out the regions to be inpainted while leaving the rest of the image intact.

4. Saving Masked Images: Once the masks were applied, resulting in images with masked regions, we saved these modified images along with their corresponding masks for subsequent use in training the inpainting model.

3.4 Data Split

To facilitate model training, validation, and evaluation, we partitioned the preprocessed CelebA dataset into three subsets:

- Training Set: This subset comprised 80% of the preprocessed images and was used to train the encoder-decoder network with partial convolution.
- Validation Set: Consisting of 10% of the data, this subset was utilized to tune hyperparameters, monitor training progress, and prevent overfitting.
- Test Set: The remaining 10% of the data served as an independent evaluation set to assess the generalization performance of the trained model on unseen data.

3.5 Model Development

3.5.1 Architecture

In our image inpainting task, we adopted the U-Net architecture, a convolutional neural network (CNN) model known for its effectiveness in semantic segmentation and image-to-image translation tasks. U-Net is particularly well-suited for image inpainting due to its ability to capture both local and global image features while preserving spatial information.

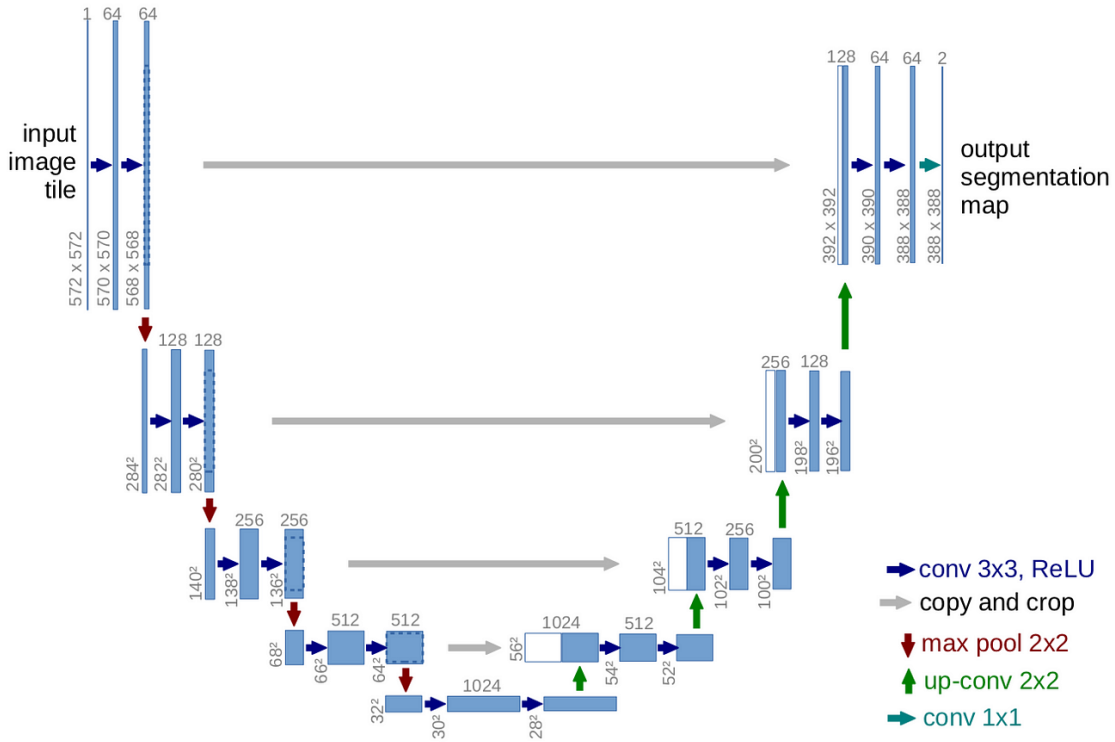
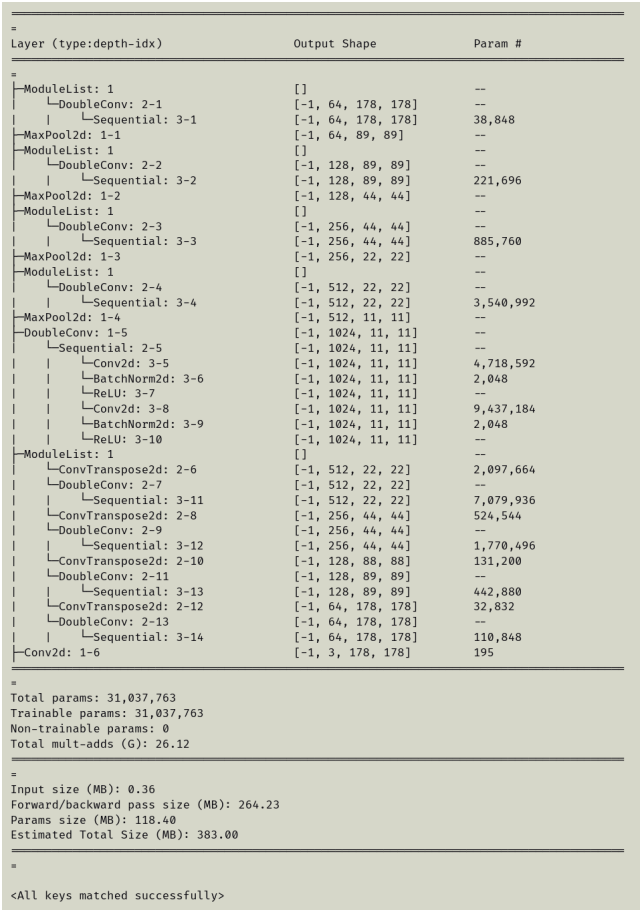


Figure 3.4: UNET Architecture

In the implementation of the UNet architecture for image inpainting, we configured the model with a contracting and expansive path, following the classic U-shaped structure. The contracting path consisted of convolutional layers with max-pooling operations to capture contextual information and extract relevant features through down-sampling. Conversely, the expansive path utilized transposed convolutions for up-sampling, aiding in the precise localization of details.

For the specific task of image inpainting, we adapted the model to handle the reconstruction of missing or damaged regions. During the training phase, we employed a dataset containing both original and partially masked images, where the missing regions served as the ground truth for training. The loss function was tailored to penalize the difference between the inpainted regions and the corresponding ground truth.

To enhance the model's performance, we incorporated skip connections between corresponding layers in the contracting and expansive paths. This facilitated the transfer of detailed information, allowing the model to reconstruct high-resolution images more effectively. The model architecture summary for training CelebA dataset is given below:



```
==========
Layer (type:depth-idx)                   Output Shape              Param #
==========
├─ModuleList: 1                          []                        --
│    └─DoubleConv: 2-1                   [-1, 64, 178, 178]        --
│    │    └─Sequential: 3-1              [-1, 64, 178, 178]        38,848
├─MaxPool2d: 1-1                         [-1, 64, 89, 89]          --
├─ModuleList: 1                          []                        --
│    └─DoubleConv: 2-2                   [-1, 128, 89, 89]         --
│    │    └─Sequential: 3-2              [-1, 128, 89, 89]         221,696
├─MaxPool2d: 1-2                         [-1, 128, 44, 44]         --
├─ModuleList: 1                          []                        --
│    └─DoubleConv: 2-3                   [-1, 256, 44, 44]         --
│    │    └─Sequential: 3-3              [-1, 256, 44, 44]         885,760
├─MaxPool2d: 1-3                         [-1, 256, 22, 22]         --
├─ModuleList: 1                          []                        --
│    └─DoubleConv: 2-4                   [-1, 512, 22, 22]         --
│    │    └─Sequential: 3-4              [-1, 512, 22, 22]         3,540,992
├─MaxPool2d: 1-4                         [-1, 512, 11, 11]         --
├─DoubleConv: 1-5                        [-1, 1024, 11, 11]        --
│    └─Sequential: 2-5                   [-1, 1024, 11, 11]        --
│    │    └─Conv2d: 3-5                  [-1, 1024, 11, 11]        4,718,592
│    │    └─BatchNorm2d: 3-6             [-1, 1024, 11, 11]        2,048
│    │    └─ReLU: 3-7                    [-1, 1024, 11, 11]        --
│    │    └─Conv2d: 3-8                  [-1, 1024, 11, 11]        9,437,184
│    │    └─BatchNorm2d: 3-9             [-1, 1024, 11, 11]        2,048
│    │    └─ReLU: 3-10                   [-1, 1024, 11, 11]        --
├─ModuleList: 1                          []                        --
│    └─ConvTranspose2d: 2-6              [-1, 512, 22, 22]         2,097,664
│    └─DoubleConv: 2-7                   [-1, 512, 22, 22]         --
│    │    └─Sequential: 3-11             [-1, 512, 22, 22]         7,079,936
│    └─ConvTranspose2d: 2-8              [-1, 256, 44, 44]         524,544
│    └─DoubleConv: 2-9                   [-1, 256, 44, 44]         --
│    │    └─Sequential: 3-12             [-1, 256, 44, 44]         1,770,496
│    └─ConvTranspose2d: 2-10             [-1, 128, 88, 88]         131,200
│    └─DoubleConv: 2-11                  [-1, 128, 89, 89]         --
│    │    └─Sequential: 3-13             [-1, 128, 89, 89]         442,880
│    └─ConvTranspose2d: 2-12             [-1, 64, 178, 178]        32,832
│    └─DoubleConv: 2-13                  [-1, 64, 178, 178]        --
│    │    └─Sequential: 3-14             [-1, 64, 178, 178]        110,848
├─Conv2d: 1-6                            [-1, 3, 178, 178]         195
==========
Total params: 31,037,763
Trainable params: 31,037,763
Non-trainable params: 0
Total mult-adds (G): 26.12
==========
Input size (MB): 0.36
Forward/backward pass size (MB): 264.23
Params size (MB): 118.40
Estimated Total Size (MB): 383.00
==========

<All keys matched successfully>
```

Figure 3.5: Model architecture summary Used to train CelebA dataset

3.5.2 Partial Convolution

The model uses stacked partial convolution operations and mask updating steps to perform image inpainting. Partial convolution operation and mask update function jointly as the Partial Convolutional Layer.

Let W be the convolution filter weights for the convolution filter and b is the corresponding bias. X are the feature values (pixels values) for the current convolution (sliding) window and M is the corresponding binary mask. The partial convolution at every location is expressed as :

$$x' = \begin{cases} \mathbf{W}^T(\mathbf{X} \odot \mathbf{M})\frac{1}{\text{sum}(\mathbf{M})} + b, & \text{if sum}(\mathbf{M}) > 0 \\ 0, & \text{otherwise} \end{cases}$$

Where $\odot$ denotes element-wise multiplication. As can be seen, output values depend only on the unmasked inputs. The scaling factor 1/sum(M) applies appropriate scaling to

adjust for the varying amount of valid (unmasked) inputs. After each partial convolution operation, the mask has been updated. The unmasking rule is simple: if the convolution was able to condition its output on at least one valid input value, then remove the mask for that location. This is expressed as:

$$m' = \begin{cases} 1, & \text{if sum}(\mathbf{M}) > 0 \\ 0, & \text{otherwise} \end{cases}$$

and can easily be implemented in any deep learning framework as part of the forward pass.

3.5.3 Training

The whole architecture of the model is developed using Pytorch library. The model is trained using following loss functions and optimizers:

- **MSE Loss Function:** In the context of image inpainting, the Mean Squared Error (MSE) loss function is often utilized to quantify the quality of the inpainted images. Image inpainting refers to the process of filling in missing or damaged regions of an image with plausible content.

  The MSE loss function for image inpainting can be represented as follows:

$$\text{MSE} = \frac{1}{n \cdot h \cdot w} \sum_{i=1}^{n} \sum_{j=1}^{h} \sum_{k=1}^{w} (I_{\text{gt}}(i,j,k) - I_{\text{inp}}(i,j,k))^2 \tag{3.1}$$

  Where:

  n is the number of images in the dataset.
  h, w are the height and width of the images, respectively.
  $I_{\text{gt}}(i,j,k)$ is the pixel value of the ground truth image at position (i,j,k).
  $I_{\text{inp}}(i,j,k)$ is the pixel value of the inpainted image at position (i,j,k).

  In the context of image inpainting, $I_{\text{gt}}(i,j,k)$ represents the pixel value that should ideally be present at position (i,j,k), while $I_{\text{inp}}(i,j,k)$ represents the pixel value that was filled in during the inpainting process.

The equation calculates the squared difference between the corresponding pixel values of the ground truth and inpainted images, sums up these squared differences across all images and pixel positions, and then divides by the total number of images (n) and the product of image dimensions ($h \cdot w$) to calculate the average squared difference per pixel.

The MSE loss helps to quantify how closely the inpainted image matches the ground truth image in terms of pixel values. Lower MSE values indicate better inpainting quality, as they reflect smaller differences between the inpainted and ground truth images. However, it's important to note that MSE may not always correlate perfectly with human perception of image quality, and other perceptual loss functions might be preferred in certain cases.

- **Adam Optimizer:** The Adam optimizer combines adaptive learning rates and momentum to efficiently optimize neural network parameters. It maintains two moving average estimates of gradients, m_t and v_t, and adjusts learning rates individually for each parameter. [13]

  The update rules for Adam are as follows:

$$
\begin{aligned}
m_t &= \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t \quad \text{(First Moment Estimate)} \\
v_t &= \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2 \quad \text{(Second Moment Estimate)} \\
\hat{m}_t &= \frac{m_t}{1 - \beta_1^t} \quad \text{(Bias-Corrected First Moment)} \\
\hat{v}_t &= \frac{v_t}{1 - \beta_2^t} \quad \text{(Bias-Corrected Second Moment)} \\
\theta_{t+1} &= \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \cdot \hat{m}_t \quad \text{(Parameter Update)}
\end{aligned}
$$

  Where:

  - m_t and v_t are the first and second moment estimates of the gradient at time step t.
  - g_t is the gradient at time step t.
  - $\hat{m}_t$ and $\hat{v}_t$ are bias-corrected moment estimates.
  - β_1 and β_2 are exponential decay rates for the first and second moments.
  - α is the learning rate.
  - t represents the time step.

– ϵ is a small constant for numerical stability.

Adam's adaptive learning rates and momentum characteristics make it effective for training neural networks across various tasks.

- **Hyperparameters:**

Hyperparameters	
Epochs	20
Batch Size	5
Learning Rate(First 8 epochs)	1e-3
Learning Rate(Last 12 epochs)	1e-4

Table 3.1: Hyperparameters in the training

3.5.4 Training & Validation Losses

The following graphs shows training and validation losses during the process of training:

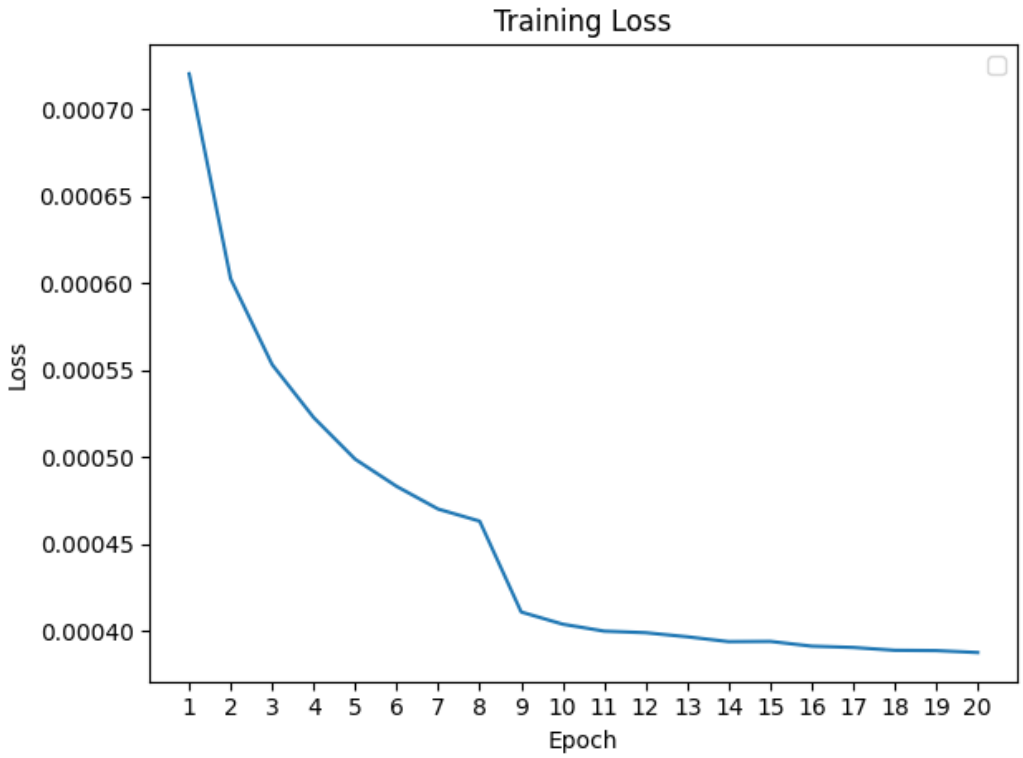


Figure 3.6: Training Loss Curve

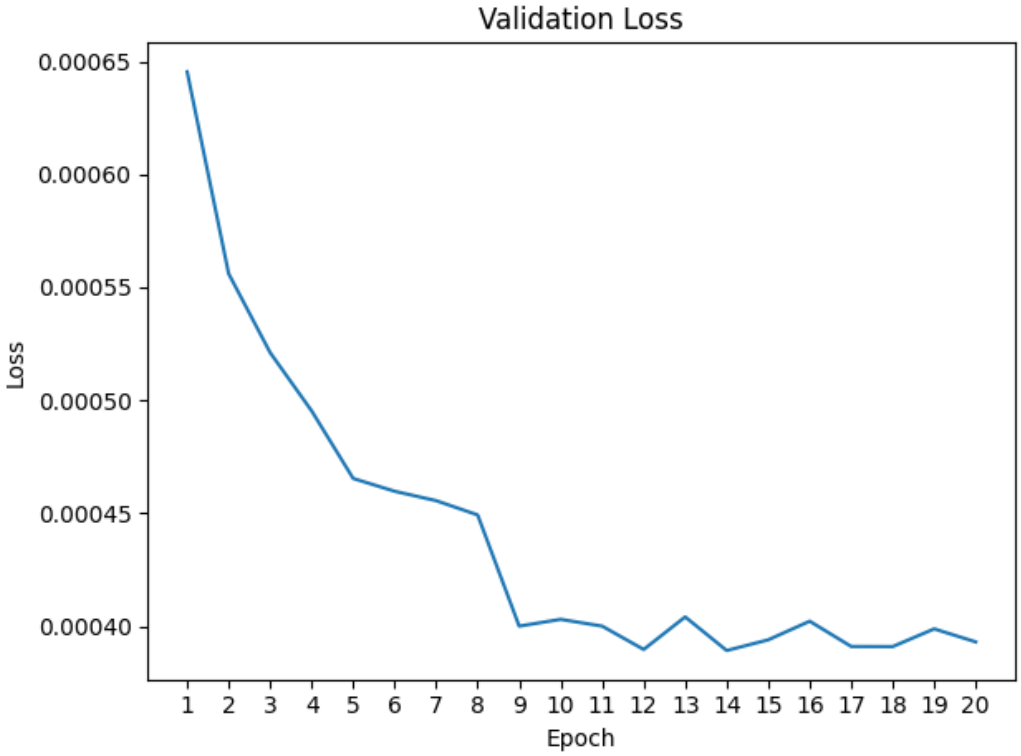


Figure 3.7: Validation Loss Curve

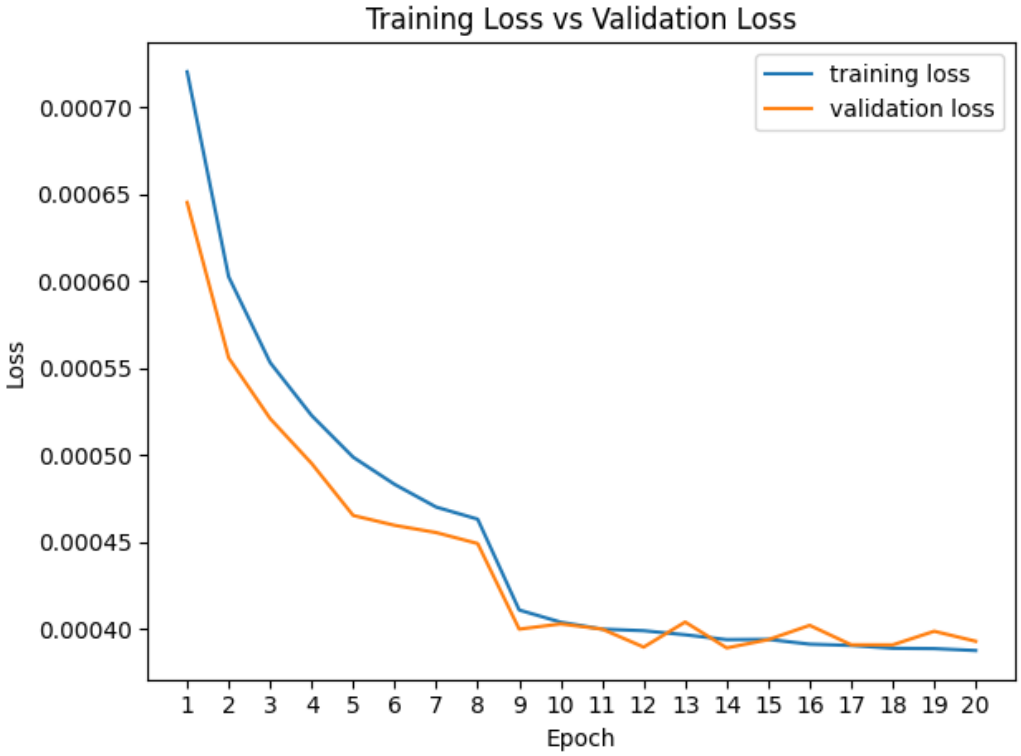


Figure 3.8: Training Vs Validation Loss Curve

3.5.5 Evaluation and Fine-tuning

Once the model was trained, we evaluated its performance using visual inspection and overall loss during the training. We fine-tuned the model by adjusting the hyperparameters or incorporating additional training data to improve the inpainting quality.

Peak Signal-to-Noise Ratio (PSNR)

PSNR is a widely used metric to measure the quality of a reconstructed or processed signal, often used in image processing. It measures the ratio between the maximum possible power of a signal and the power of the noise that affects its fidelity. Higher PSNR values indicate better image quality. [14]

The formula for PSNR is:

$$\text{PSNR} = 10 \cdot \log_{10} \left(\frac{\text{MAX}^2}{\text{MSE}} \right)$$

Where:

- MAX is the maximum possible pixel value of the image.
- MSE (Mean Squared Error) is the average squared difference between the original image and the reconstructed image.

Structural Similarity Index (SSIM)

SSIM is another metric commonly used to assess the similarity between two images. It takes into account luminance, contrast, and structural information. Higher SSIM values indicate greater similarity between the images. [15]

The formula for SSIM is:

$$\text{SSIM}(x, y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)}$$

Where:

- x and y are the input images.
- μ_x and μ_y are the means of x and y respectively.
- σ_x^2 and σ_y^2 are the variances of x and y respectively.
- σ_{xy} is the covariance of x and y.

- C_1 and C_2 are constants to stabilize the division with weak denominator.

Metrics like PSNR [14] and SSIM[15] were implemented and obtained as shown in the table.

Evaluation Metrics			
SSIM	PSNR(dB)	L2 Loss(Training)	L2 Loss(Validation)
0.910	30.456	4e-4	4.2e-4

Table 3.2: Evaluation Metrics

3.6 Web Development & Deployment

The completed project aimed to create a web application that enables users to upload images and apply masks to them, enhancing their editing capabilities. The application provides a user-friendly interface for image manipulation, making it easy for users to edit their photos.

3.6.1 Design and Architecture

Frontend with SvelteKit: SvelteKit was used to design the frontend, providing a responsive and interactive user interface. SvelteKit is a framework for building web applications, known for its simplicity and performance. It allows us to write components using a reactive approach, making it easier to manage the state of our application.

Manual Masking with HTML Canvas: HTML Canvas was used to implement the manual masking process, allowing users to draw masks directly on the images. Canvas provides a drawing API that allows us to create dynamic graphics and animations on the web. By using Canvas, we were able to create a user-friendly interface for users to interact with the images.

Backend with FastAPI and PyTorch on Azure Virtual Machine: FastAPI was used to create a RESTful API for handling image processing requests. FastAPI is a modern web framework for building APIs with Python. It is known for its fast performance and easy-to-use interface. PyTorch, a deep learning framework, was likely used for the actual image processing tasks, such as applying the masks to the images. Running this backend on Azure Virtual Machine provided a scalable and reliable infrastructure for your application.

Nginx as a Reverse Proxy: Nginx was configured as a reverse proxy to route requests from the frontend to the backend. This setup ensured secure and efficient communica-

tion between the two components. Nginx is a high-performance web server and reverse proxy server, commonly used in production environments to improve the performance and reliability of web applications.

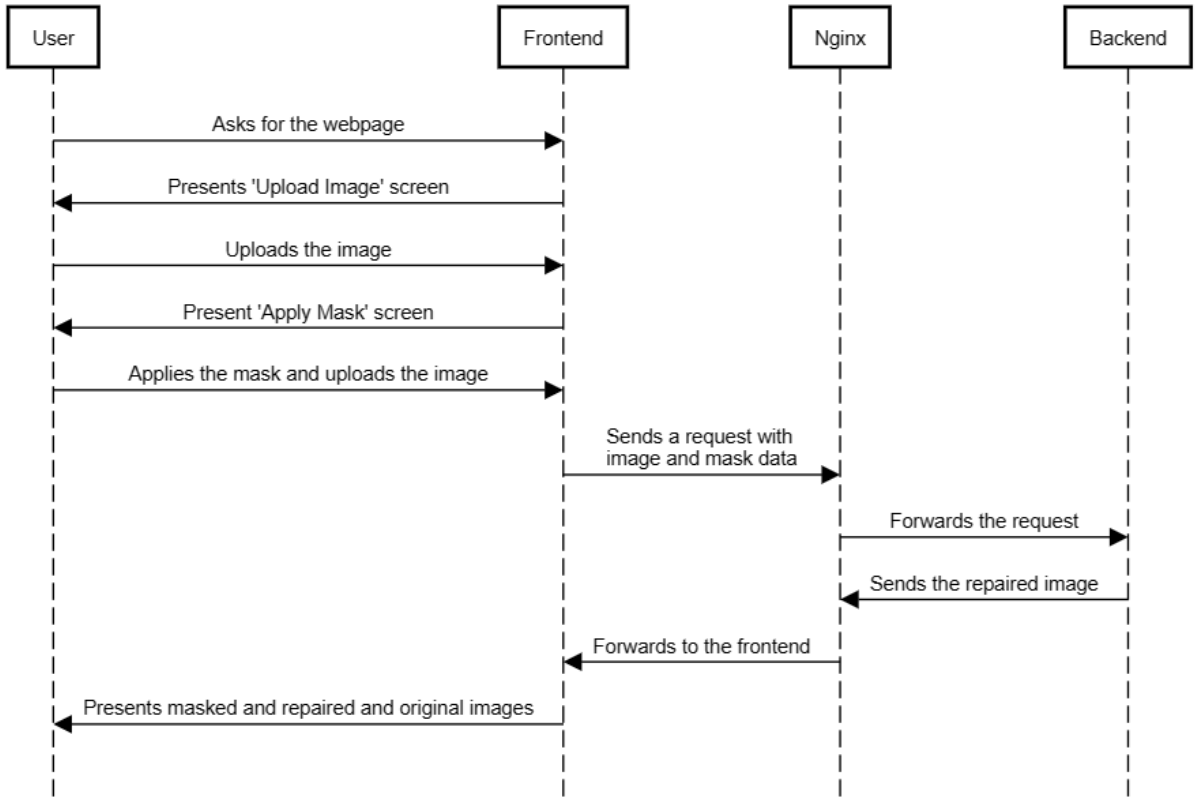


Figure 3.9: Sequence Diagram

3.6.2 Implementation

Frontend Components for Image Uploading, Mask Drawing, and Image Display: We developed frontend components to handle various aspects of the image masking process. This included a component for uploading images, allowing users to select files from their device and upload them to the application. Another component was responsible for displaying the uploaded images and the masks applied to them. Additionally, a component for drawing masks on the images was implemented, providing users with a way to apply masks with precision.

HTML Canvas for Dynamic Masking Interface: The HTML Canvas element was used to create a dynamic masking interface. Canvas provides a drawing surface that allows you to draw graphics, images, and other visual elements dynamically using JavaScript. By using Canvas,we were able to create an interactive interface that allowed users to draw masks directly on the images. This provided a more intuitive and user-friendly experience compared to other methods of applying masks.

Backend with FastAPI and PyTorch: The backend API was implemented using FastAPI, a modern web framework for building APIs with Python. FastAPI was used to create endpoints for handling image processing tasks, such as applying masks to images. PyTorch, a deep learning framework, was used for the actual image processing tasks, as it provides powerful tools for working with images and applying complex transformations.

3.6.3 Hosting

For deployment, we utilized Vercel for hosting the frontend and an Azure Virtual Machine for hosting the backend. Vercel provided a fast and reliable hosting platform for the frontend, ensuring that the application was responsive and accessible to users. The Azure Virtual Machine was chosen for the backend deployment due to its scalability and performance, allowing the application to handle a large number of requests efficiently.

To ensure efficient communication between the frontend and backend, we configured Nginx as a reverse proxy. Nginx was responsible for routing requests from the frontend to the backend, ensuring that the two components could communicate seamlessly. This setup helped in maintaining the overall performance and reliability of the application. Scalabality can be ensured using load balancing.

Overall, the deployment setup using Vercel, Azure Virtual Machine, and Nginx as a reverse proxy ensured that the our project was fast, reliable, and scalable, providing users with a seamless experience.

4. Experimental Setup

The experimental configuration for assessing the auto-encoding UNET model in image inpainting initially involved using Google Colab for preliminary testing. However, due to constraints like memory limitations and computation restrictions within Google Colab, we transitioned to utilizing our own computing resources for training. We opted against shifting to cloud-based computation due to the prohibitive cost associated with using GPUs in the cloud. The experimental setup encompasses the subsequent steps:

4.1 Google Colab

We utilized T4 GPU of the colab for the initial training process. The specification of T4 GPU and other cloud resources in colab is given below:

Specification	**Details**
Architecture	Turing
CUDA Cores	2560
Tensor Cores	Yes
Memory	16GB GDDR6
Memroy Bandwidth	300GB/s
Maximum Power Consumption	70W
Form Factor	Single-slot, low-profile PCIe x8/x16

Table 4.1: Google Colab T4 GPU Specification

4.1.1 Issues with Google Colab

Since the free tier of the google colab was used for the training, following issues were faced during the training in colab:

- Session Timeout: Colab sessions are automatically disconnected after 90 minutes of inactivity.
- Runtime Limit: Colab notebooks can run for a maximum of 12 hours at a time.
- Hardware Sharing: Multiple users may be sharing the underlying hardware resources, so performance can vary.

4.2 Local Device

Due timeout and runtime limits of the Colab's, we were forced to train our model in the local device for **nearly 3 days continuously**. The specifications for the device is given below:

Specification	**Details**
Architecture	Ampere (Samsung 8nm)
CUDA Cores	3584
Tensor Cores	224
Memory	6GB
Memroy Bandwidth	18Gbps/s
Base Clock	1425 MHz
Boost Clock	Up to 1702 MHz (depends on model)
TDP (Typical Power Draw)	170 W

Table 4.2: RTX 3060 GPU Specification

Specification	**Details**
Architecture	TSMC 7nm FinFET
CPU Cores	8
CPU Threads	224
L2 Cache	4MB
L3 Cache	16MB
Memroy Bandwidth	18Gbps/s
Base Clock	3.2 GHz
Boost Clock	Up to 4.2 GHz
TDP (Typical Power Draw)	45 W

Table 4.3: AMD Ryzen 7 5800H CPU Specification

Other details about the device specification:

- **RAM**: 16GB DDR4

4.3 Software Technology Setup

4.3.1 Jupyter Notebook

In our project, we leveraged Jupyter Notebooks as a central tool for conducting our data analysis, model development, and experimentation processes. The interactive nature of Jupyter Notebooks allowed us to seamlessly integrate code execution with narrative text, visualizations, and documentation, facilitating clear and concise communication of our methodologies and findings. We utilized Python code cells within the notebooks to implement and test various algorithms for image inpainting, with libraries such as PyTorch

for model development and OpenCV for preprocessing tasks. The ability to execute code blocks individually and visualize the results in real-time enabled us to iteratively refine our models and techniques, enhancing their effectiveness and performance. Moreover, Jupyter Notebooks served as a collaborative workspace, allowing team members to collaborate remotely, share insights, and provide feedback in real-time. By harnessing the power of Jupyter Notebooks, we streamlined our workflow, fostered collaboration, and achieved significant advancements in our image inpainting project.

4.3.2 OpenCV

In our image inpainting project, OpenCV played a pivotal role in generating masks to delineate the regions requiring inpainting. OpenCV, an open-source computer vision library, provided comprehensive functionality for image manipulation and processing, crucial for mask generation. Specifically, we leveraged OpenCV's capabilities to create binary masks that precisely identified the missing or damaged regions within input images. This involved employing various image processing techniques, such as thresholding, morphological operations, and contour detection, to accurately delineate the boundaries of the regions to be inpainted. Additionally, OpenCV's robust implementation of these operations ensured efficient and accurate mask generation across diverse datasets, contributing to the overall effectiveness and reliability of our image inpainting methodology. By harnessing OpenCV's advanced features for mask generation, we were able to streamline the preprocessing pipeline and facilitate seamless integration with subsequent stages of the inpainting process, ultimately enhancing the quality and fidelity of the inpainted images.

4.3.3 Pytorch

PyTorch served as the cornerstone for model development in our image inpainting project, providing a powerful framework for building, training, and evaluating deep learning models. PyTorch's rich set of functionalities enabled us to construct and fine-tune complex neural network architectures tailored to the specific requirements of image inpainting. Leveraging PyTorch's dynamic computational graph, we seamlessly defined and executed intricate network structures, facilitating experimentation and rapid prototyping. Furthermore, PyTorch's autograd mechanism efficiently computed gradients during training, automating the backpropagation process and enabling seamless optimization of model parameters. The framework's GPU acceleration capabilities, integrated seamlessly with CUDA, expedited training processes, allowing for faster convergence and scalability to handle large-scale datasets. PyTorch's modular design and extensive library support also facilitated the implementation of state-of-the-art inpainting techniques, enhancing

the robustness and adaptability of our models. By harnessing PyTorch's versatile features for model development, we achieved significant advancements in image inpainting performance, underscoring its pivotal role in driving innovation within our project.

5. System design

5.1 Model Pipeline Design

The system architecture for our image inpainting model revolves around an UNET framework as shown in Fig 5.1. This architecture allows the model to effectively handle missing or corrupted regions while preserving the contextual information necessary for generating realistic and coherent inpainted images.

5.1.1 Data Processor

We start by creating a custom class specifically designed to handle large amounts of images. This class prepares the data for use with PyTorch's dataset and dataloader functionalities. The custom class takes care of splitting the data into training, validation, and testing sets. Built-in PyTorch transformations are applied within the class to resize and normalize the images. Finally, the prepared datasets are fed into dataloaders. Dataloaders enable features like batching, shuffling, and parallel processing of the data for efficient training.

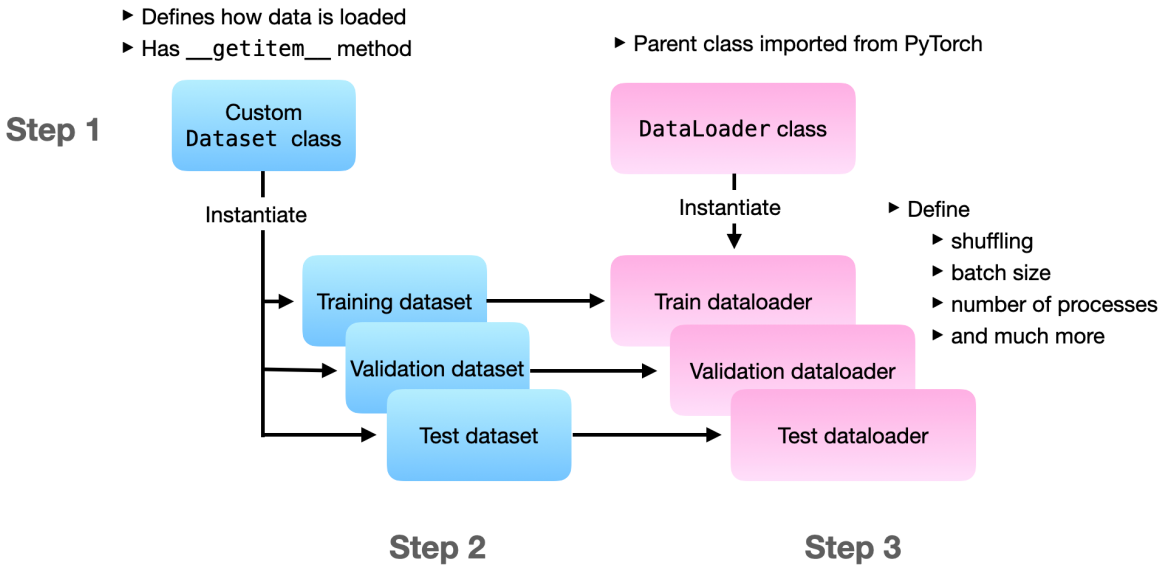


Figure 5.2: Dataset and Dataloader Schematic View

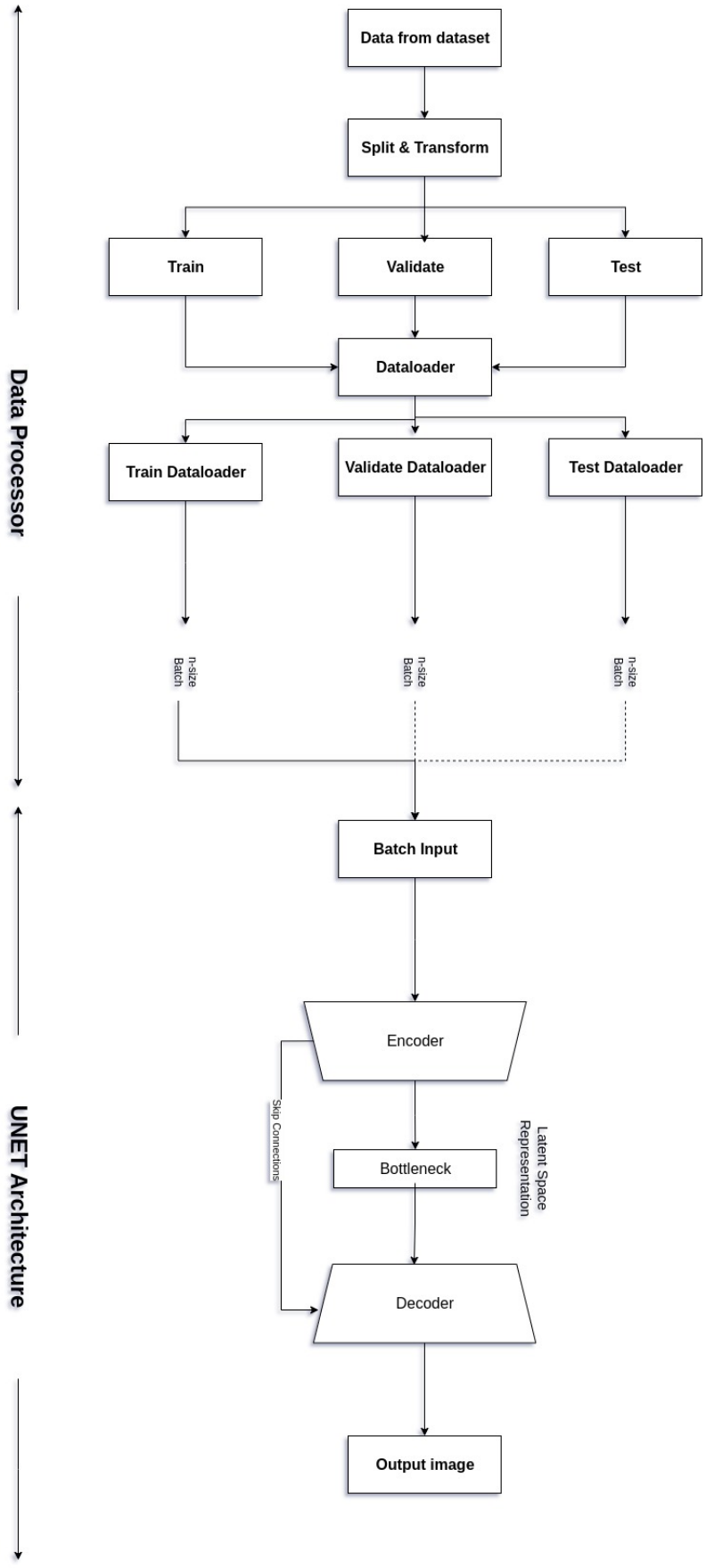


Figure 5.1: System Design

5.1.2 UNET Model

U-Net, adapted for image inpainting tasks, is a convolutional neural network architecture specifically tailored to reconstruct missing or damaged regions within images. In this context, U-Net's encoder captures the image's context and features, while its decoder reconstructs missing regions by upsampling and integrating spatial details. Skip connections between encoder and decoder layers facilitate the flow of both high-level semantic information and fine-grained spatial details, crucial for accurate inpainting. This architecture excels in preserving image structure and texture, making it well-suited for tasks requiring seamless completion of missing image regions, such as removing objects or repairing damaged images.

Encoder Network

The encoder is responsible for capturing contextual information and extracting relevant features from the input image. It comprises a series of convolutional layers with max-pooling operations, systematically downsampling the spatial dimensions of the input. This process helps the network to learn hierarchical features and global context by focusing on larger receptive fields.[16][17]

Skip Connections

Skip connections in the U-Net architecture serve as vital conduits for the seamless integration of high-level semantic information from the encoder with low-level spatial details from the decoder. By directly connecting corresponding layers between the encoder and decoder, skip connections facilitate the preservation of fine-grained spatial features, mitigating the information bottleneck that often arises during the downsampling process. This preservation of spatial details enhances the model's ability to accurately reconstruct output images, particularly in tasks such as image segmentation and image inpainting. Furthermore, skip connections promote stable gradient flow during training, contributing to faster convergence and improved optimization of network parameters. Overall, skip connections play a crucial role in enhancing the effectiveness and robustness of U-Net, enabling it to produce high-quality output images across a wide range of image processing tasks.

Decoder Network

Conversely, the decoder is designed to reconstruct the spatial details and localize information with precision. It involves transposed convolutions (also known as up-sampling or deconvolutions) to gradually increase the spatial resolution of the encoded features. Skip

connections connect corresponding layers between the encoder and decoder, facilitating the transfer of high-resolution details. This enables the network to generate accurate pixel-wise predictions and reconstruct the input image at its original resolution.

Together, the encoder and decoder components of the UNet architecture create a U-shaped network that excels in tasks such as image segmentation and inpainting by effectively capturing both global context and fine-grained details.[16]

5.2 Overall Project Design

Overall project consists of User, Application Component and Model Component. User is responsible for interacting with the Application. Application sits between the User and the Model and acts as the intermediary between the two. Model performs the actual image transformation using the UNET.

1. User navigates to the web page here he/she can select and upload the image.
2. Application validates the uploaded file (image). If the file is valid, use is presented with the option to create a mask.
3. User applies the mask to the image with the provided brush tool and sends it to the application.
4. Application transforms the image into 178 X 178 pixels and feeds the transformed image to the model.
5. Model performs the image reconstruction using UNET and sends back the image to application.
6. Application encodes the image in Base64 format and sends it back to the user.
7. The final reconstructed image is rendered in the user's browser along with the original uploaded image.

5.2.1 Activity Diagram

Activity Diagram presents what activities are performed and how those activities flow within the system.

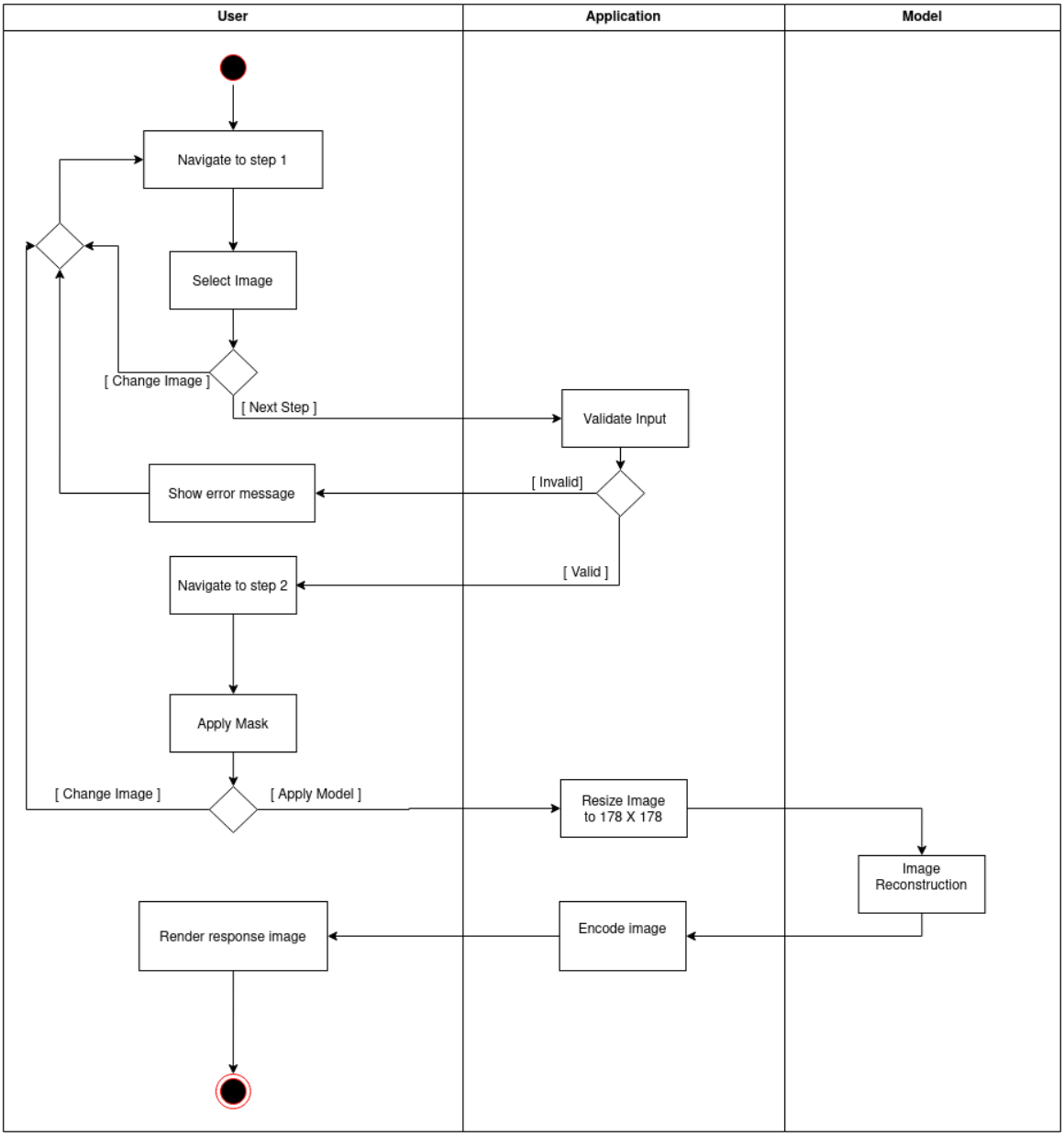


Figure 5.3: Activity Diagram

5.2.2 Sequence Diagram

Sequence Diagram focuses on how the objects within the system interact with each other over time.

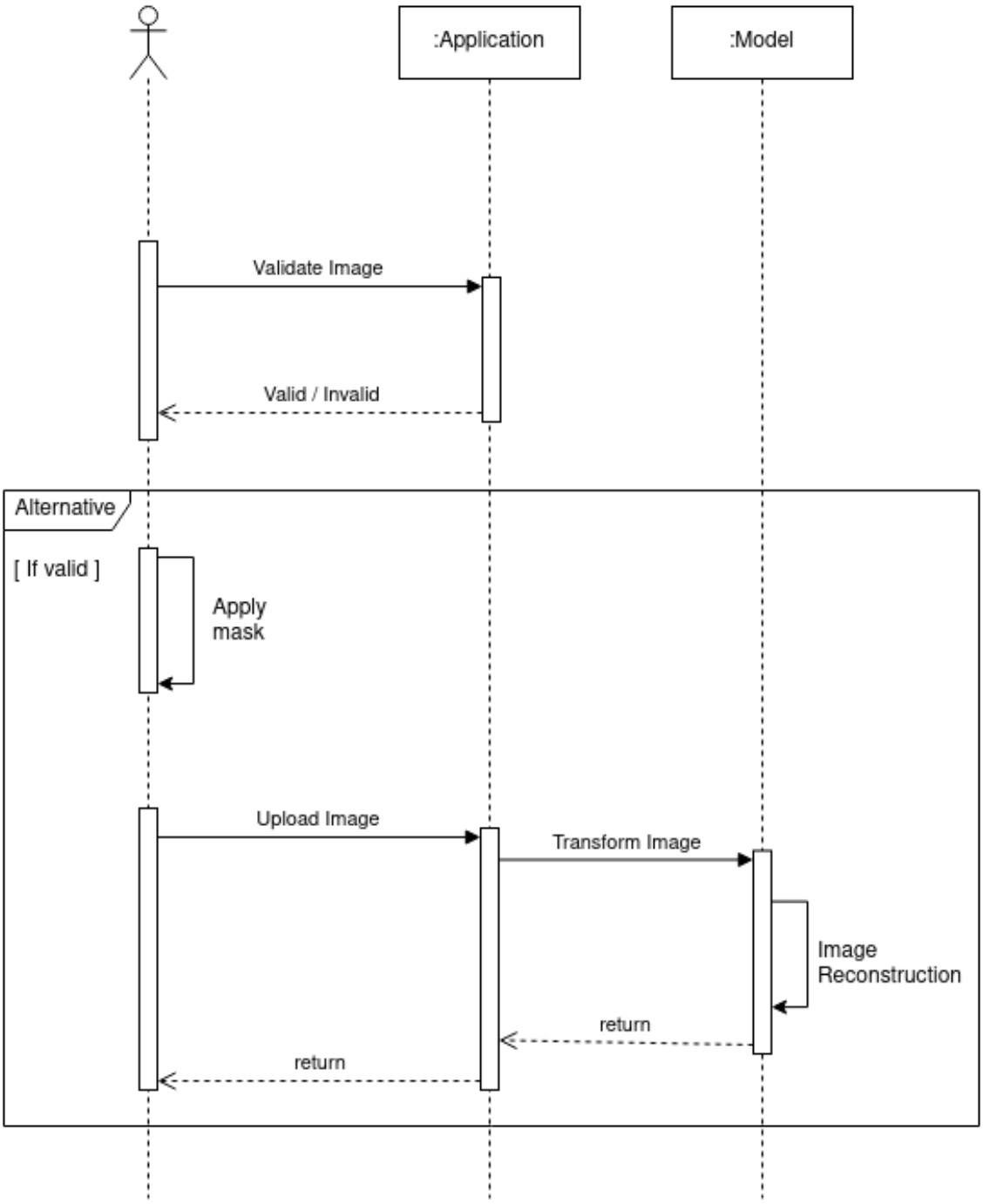


Figure 5.4: Sequence Diagram

6. Results & Discussion

6.1 Phase 1: Training with small subset of data and resources

During the project's initial phase, a preliminary model architecture resembling the UNET architecture was formulated. To assess its effectiveness, the model underwent validation using a subset of the original dataset. Specifically, 50,000 images were selected from the dataset for training, validation, and testing purposes, with data splits allocated at 80%, 10%, and 10%, respectively. Additionally, the applied masks were sparsely distributed, ensuring that most parts of the images remained visible. Upon completion of the training process, the model's performance on test images yielded the following results:

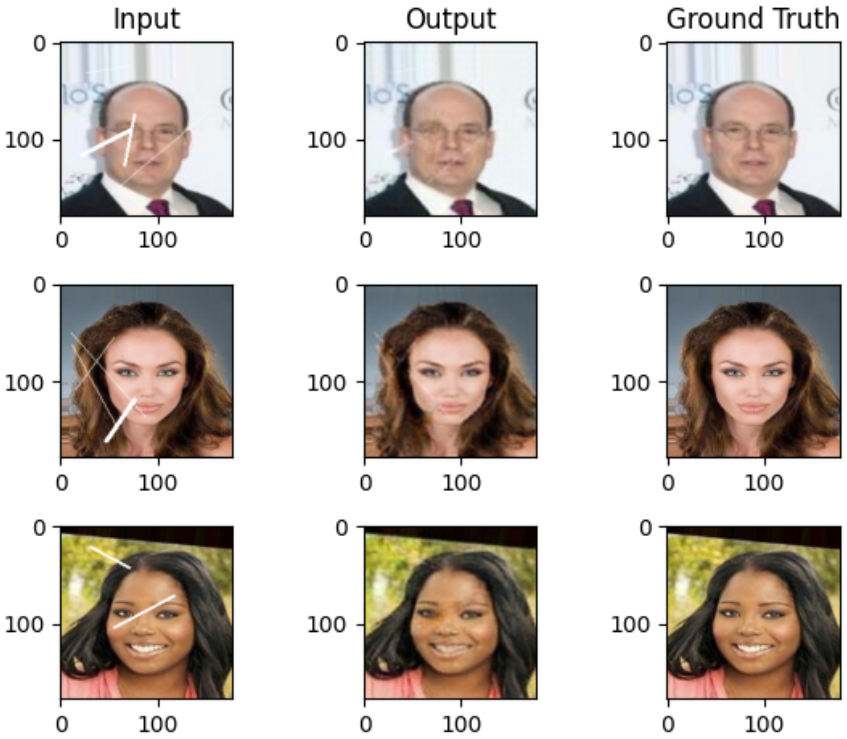


Figure 6.1: Output Results of the model on CelebA dataset - 1

In Figure 6.1 & 6.2, it is evident that while the majority of the line masks have been successfully removed from the images, traces of the lines remain visible. Moreover, the model struggles to accurately reconstruct complex facial features such as eyes and hair,

which were originally masked. Specifically, in the first row of the sample images, the eye region appears significantly distorted, indicating a limitation in the model's ability to accurately inpaint intricate details.

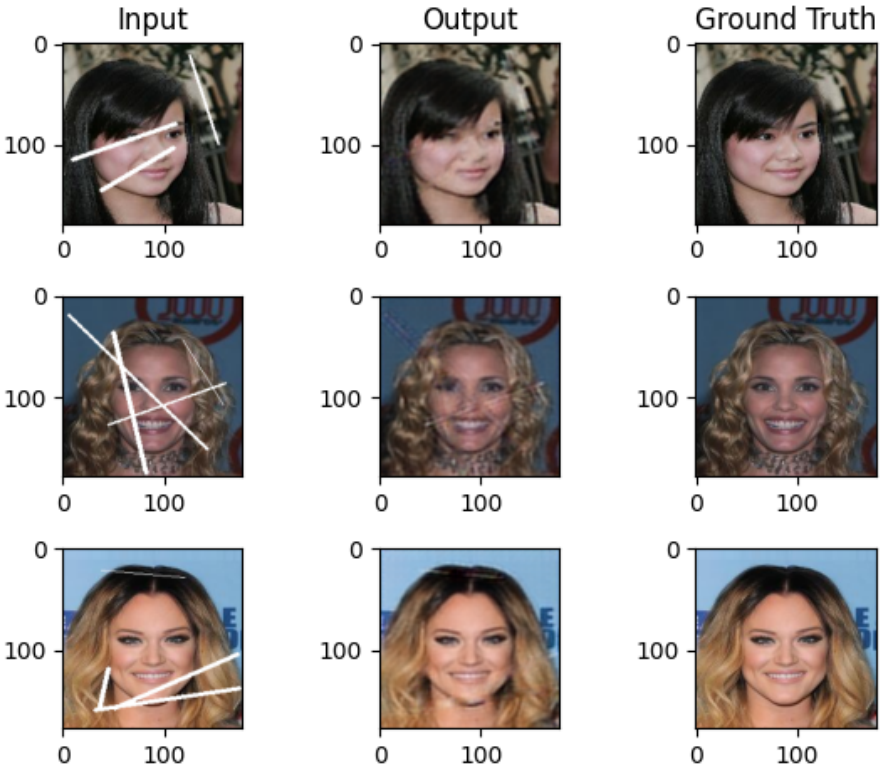


Figure 6.2: Output Results of the model on CelebA dataset - 2

6.2 Phase 2: Training on whole dataset

Following the validation of the initial smaller model, the model was expanded to its full architecture, incorporating additional layers, skip connections, and partial convolution operations to enhance its inpainting capabilities. Furthermore, the entire dataset, comprising over 200,000 images, was utilized for training, validation, and testing, with a split ratio of 8:1:1. The applied masks were also made more complex, featuring larger and thicker lines, as well as an increased number of lines, thereby concealing intricate parts of the images for a more rigorous evaluation of the model's performance.

As observed in the images below Figure 6.3 & 6.4, the output generated by the model has notably improved compared to the previous phase. The inpainted images exhibit reduced information loss, resulting in sharper overall appearance. Furthermore, the issue of residual traces from the masks has been effectively addressed. Notably, intricate facial features such as eyes, hair, nose, and facial contours are generated with greater fidelity, underscoring a significant enhancement in the model's capability. This improvement

reflects the successful refinement and optimization of the model architecture and training process, leading to more accurate and visually appealing inpainted results.

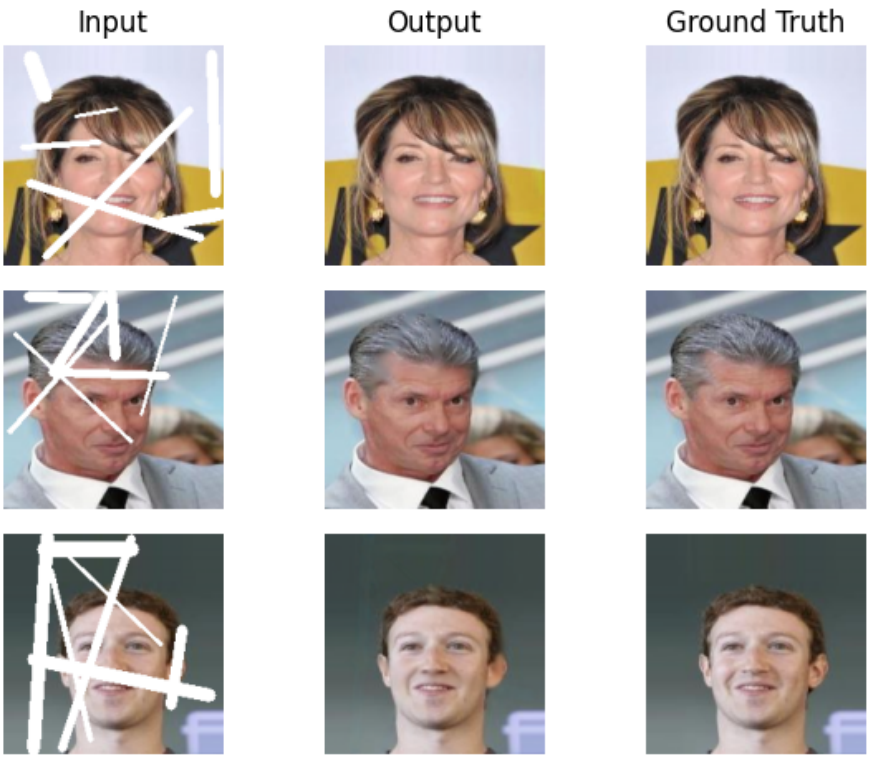


Figure 6.3: Output Results of the model on CelebA dataset - 1

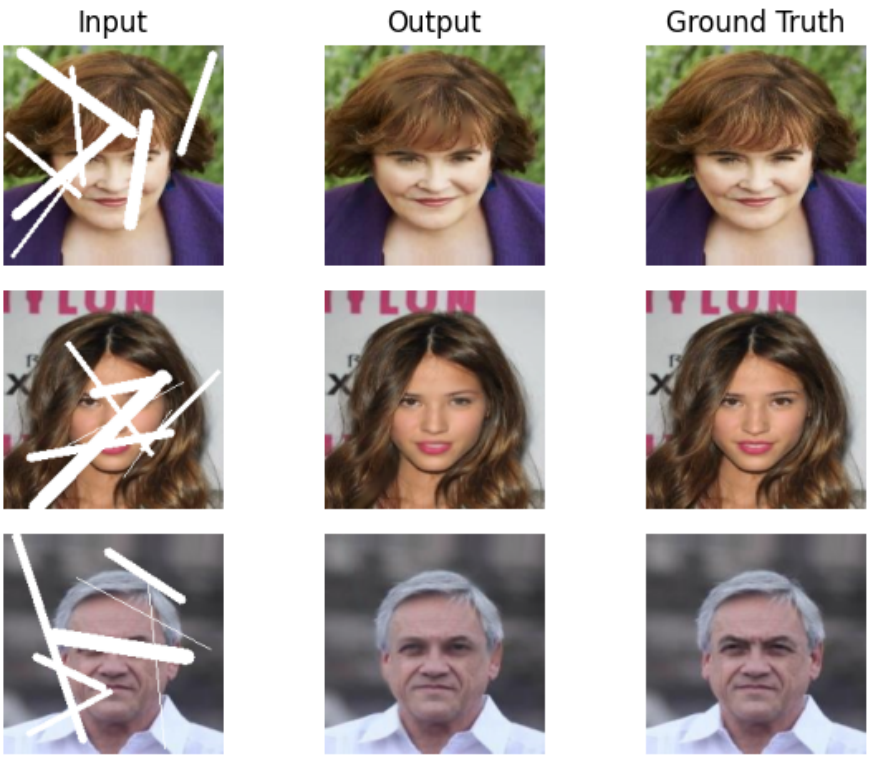


Figure 6.4: Output Results of the model on CelebA dataset - 2

One of the most notable achievements of our model is its ability to generate complex structures such as eyes, as depicted in the figure below. While the generated eye may not perfectly match the original, it showcases the model's remarkable capability to autonomously generate intricate components. This accomplishment highlights the model's capacity to understand and replicate intricate details, contributing to its effectiveness in inpainting tasks. By successfully reconstructing complex parts like eyes, the model demonstrates its potential to produce realistic and contextually appropriate inpainted images, thereby advancing the state-of-the-art in image inpainting technology.

Figure 6.5: Eye Generation

6.3 Interactive UI Results

- First step in image inpainting involves selection of appropriate image. Users have an options to select images from local device by clicking the ui component or dragging the image over the component.

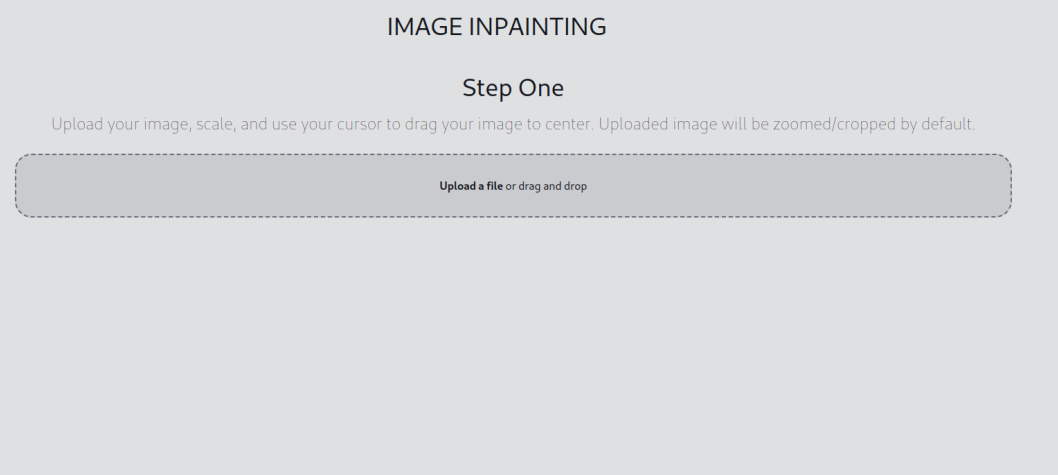


Figure 6.6: Interactive UI Page 1

- User can confirm the selected image or change the image in this page.

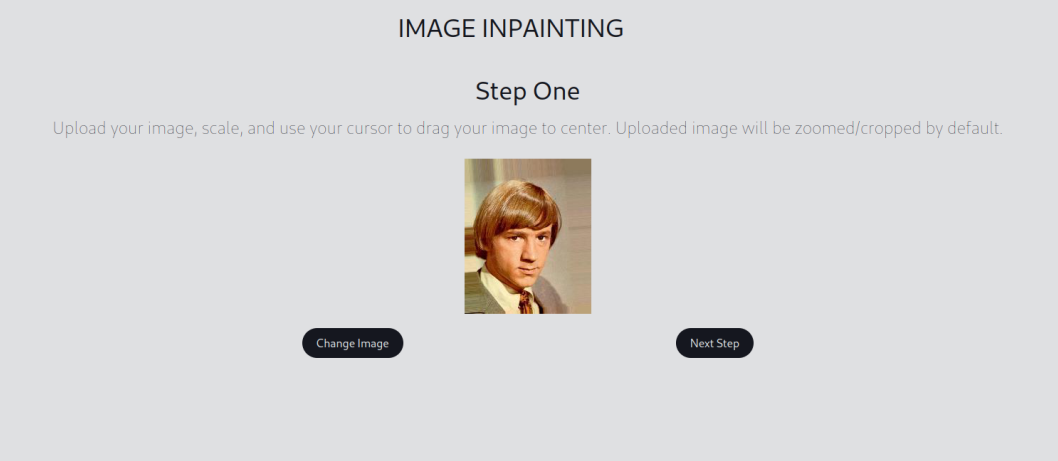


Figure 6.7: Interactive UI Page 2

- This page provides options for applying mask over the image for interactive inpainting experience. The user can change brush size, undo the mask, clear all mask and also toggle mask view. After masking is completed, user should press 'Apply Model' button to proceed to inpainting step.

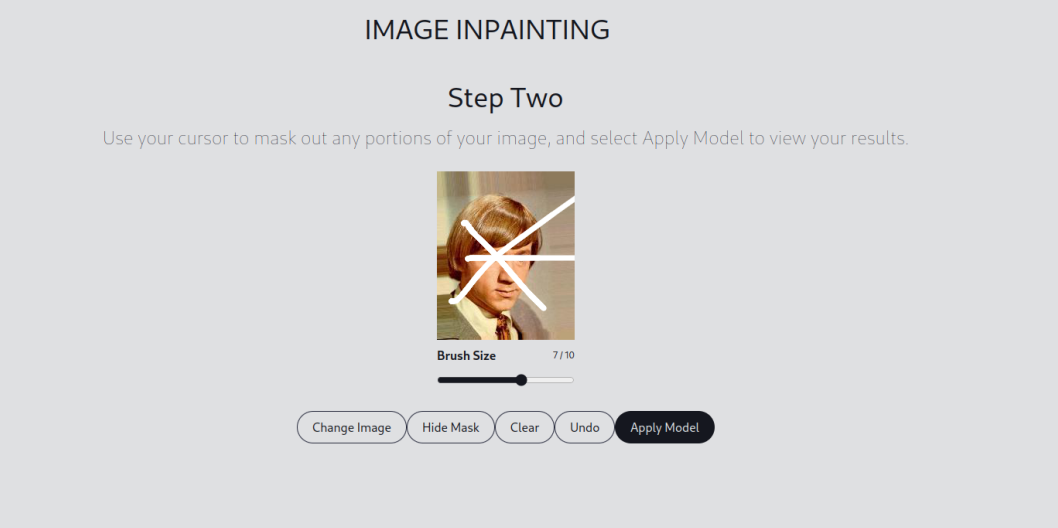


Figure 6.8: Interactive UI Page 3

- This page sends the masked image to the backend server where the model is applied to get the inpainted result. The inpainted result is presented side-by-side to the masked image to analyze the difference. User can also start over by selecting new image for inpainting.

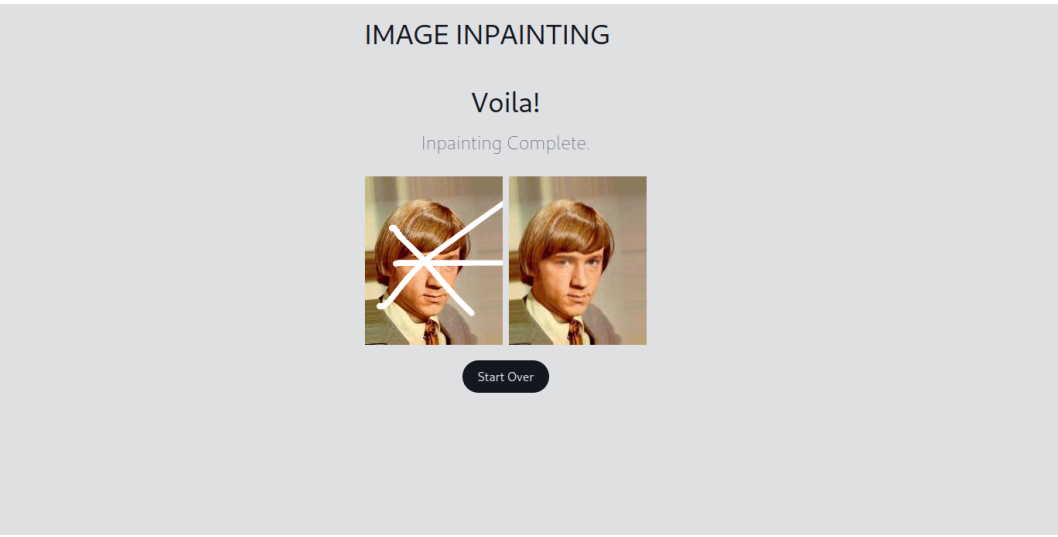


Figure 6.9: Interactive UI Page 4

6.4 Results on real images

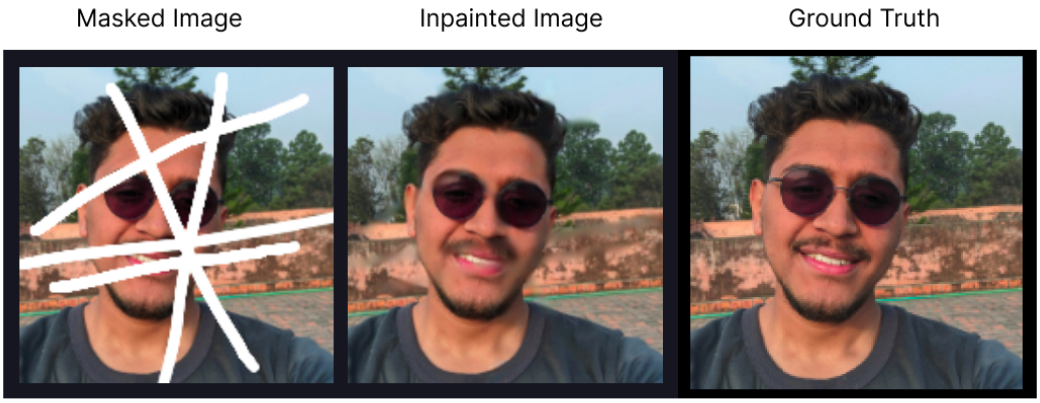


Figure 6.10: Model Output on personal selfie image

The model's performance on personal images of ourselves provides valuable insights into its behavior across various image resolutions. The results depicted in the figure 6.10 indicate that the model effectively reconstructs the masked portions of the images. However, it appears to encounter challenges when inpainting intricate parts, exhibiting signs of struggle in achieving accurate reconstructions. One plausible explanation for this discrepancy could be the divergence between the provided images and the dataset used for training. The model was primarily trained on images with similar facial views and lighting conditions, which may not fully align with the characteristics of the provided images. As a result, discrepancies between the dataset and the provided images may lead to inconsistencies in certain reconstructed parts. This underscores the importance of ensuring that training data closely aligns with the target application scenarios to enhance the model's generalization capabilities and improve its performance across diverse inputs.

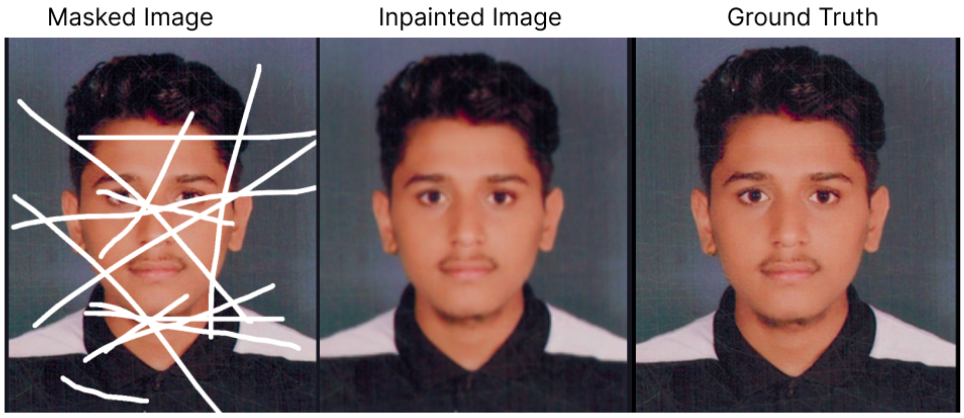


Figure 6.11: Model Output on pp-size image

In the case of the image depicted in Figure 6.11, the similarity between the provided image and those in the dataset contributes to the close resemblance observed in the model output. This alignment allows the model to produce inpainted regions that closely match those seen in the test images of the dataset. However, it's worth noting that the overall sharpness of the output image appears to be reduced compared to the original. Despite this, the reconstruction of intricate parts within the image is notably impressive. This success can be attributed to the model's ability to leverage learned features and patterns from the dataset to accurately inpaint complex regions. While there may be slight compromises in sharpness, the model demonstrates proficiency in reconstructing detailed features, highlighting its effectiveness in capturing and reproducing intricate image components.

7. Conclusions

In conclusion, our investigation into image inpainting using encoder-decoder networks with partial convolution has yielded significant advancements in the realm of computer vision and deep learning. Beginning with an introduction to the problem domain and a review of relevant literature, we established a foundation for our research by exploring existing techniques and methodologies.

Through meticulous data processing utilizing OpenCV, we prepared the dataset and generated masks essential for training our models. Subsequently, we embarked on model development, implementing UNET architecture augmented with partial convolution to enhance the model's inpainting capabilities. Our experimentation led to the refinement of model architectures, leveraging insights from the literature and iterative testing to improve performance.

The results obtained from our trained models showcased promising outcomes, demonstrating the efficacy of our approach in accurately reconstructing missing regions within images. These results were achieved through an experimental setup that leveraged both cloud-based computing resources, such as Google Colab, and local devices, balancing computational efficiency and flexibility.

Looking beyond the technical aspects, the application of our project holds immense potential in various domains. From restoring damaged photographs to aiding in medical image analysis and facilitating content creation, the capabilities of our inpainting models extend to diverse fields. By seamlessly reconstructing missing regions while preserving image context and structure, our approach contributes to enhancing visual content quality and usability.

In summary, our project represents a significant step forward in image inpainting research, with implications spanning digital photography, medical imaging, and beyond. As we continue to refine our methodologies and explore new avenues for application, we remain committed to advancing the state-of-the-art in image inpainting and contributing to the broader landscape of computer vision and artificial intelligence.

8. Limitations and Future enhancement

8.1 Limitations

Despite the successes achieved in our image inpainting project, several limitations merit consideration. Some of the limitations of our project are:

- Dataset Diversity: The performance of our models may be limited by the diversity and quality of the training data, potentially impacting model generalization.
- Computational Constraints: Limited computational resources may have constrained the complexity and scale of our models, affecting their performance and scalability.
- Contextual Complexity: The effectiveness of our inpainting techniques may vary depending on the complexity of the inpainted regions and the surrounding context, leading to suboptimal results in certain scenarios.

8.2 Future Enhancements

To address these limitations and further enhance the efficacy and applicability of our image inpainting methods, several avenues for future research and development can be explored.

- Dataset Expansion: Diversifying and expanding the training dataset to encompass a broader range of image variations and contexts could improve model generalization and robustness.
- Advanced Data Augmentation: Leveraging advanced data augmentation techniques, such as domain randomization and adversarial training, could enhance model performance in handling diverse inpainting scenarios.
- Novel Model Architectures: Exploring novel model architectures, optimization strategies, and loss functions tailored specifically for inpainting tasks could yield further improvements in reconstruction accuracy and visual quality.
- Integration of Contextual Information: Integrating additional contextual information, such as semantic segmentation masks or depth maps, into the inpainting pro-

cess could enhance the model's ability to generate contextually consistent inpainted images.

References

[1] Deepak Pathak, Philipp Krahenbuhl, Jeff Donahue, Trevor Darrell, and Alexei A. Efros. Context encoders: Feature learning by inpainting, 2016.

[2] Yi Wang, Xin Tao, Xiaojuan Qi, Xiaoyong Shen, and Jiaya Jia. Image inpainting via generative multi-column convolutional neural networks, 2018.

[3] Kamyar Nazeri, Eric Ng, Tony Joseph, Faisal Z. Qureshi, and Mehran Ebrahimi. Edgeconnect: Generative image inpainting with adversarial edge learning, 2019.

[4] Jiahui Yu, Zhe Lin, Jimei Yang, Xiaohui Shen, Xin Lu, and Thomas Huang. Free-form image inpainting with gated convolution, 2019.

[5] Guilin Liu, Fitsum A. Reda, Kevin J. Shih, Ting-Chun Wang, Andrew Tao, and Bryan Catanzaro. Image inpainting for irregular holes using partial convolutions, 2018.

[6] Ugur Demir and Gozde Unal. Patch-based image inpainting with generative adversarial networks, 2018.

[7] Alexandru Telea. An image inpainting technique based on the fast marching method. *Journal of Graphics Tools*, 9, 01 2004.

[8] Andreas Lugmayr, Martin Danelljan, Andres Romero, Fisher Yu, Radu Timofte, and Luc Van Gool. Repaint: Inpainting using denoising diffusion probabilistic models, 2022.

[9] S Padmavathi. Hierarchical approach for total variation digital image inpainting. *International Journal of Computer Science, Engineering and Applications*, 2(3):173–182, June 2012.

[10] Carola-Bibiane Schönlieb. *Partial Differential Equation Methods for Image Inpainting*. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, 2015.

[11] Diederik P. Kingma and Max Welling. An introduction to variational autoencoders. *CoRR*, abs/1906.02691, 2019.

[12] Tao Yu, Zongyu Guo, Xin Jin, Shilin Wu, Zhibo Chen, Weiping Li, Zhizheng Zhang, and Sen Liu. Region normalization for image inpainting, 2023.

[13] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[14] Fernando A. Fardo, Victor H. Conforto, Francisco C. de Oliveira, and Paulo S. Rodrigues. A formal evaluation of psnr as quality measurement parameter for image segmentation algorithms, 2016.

[15] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

[16] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015.

[17] Deepak Pathak, Philipp Krähenbühl, Jeff Donahue, Trevor Darrell, and Alexei A. Efros. Context encoders: Feature learning by inpainting. *CoRR*, abs/1604.07379, 2016.

Appendices

```python
def gen_line_mask(size: tuple, thickness_range: tuple = (1, 3), bg_color=0, patch_color: tuple = (255, 255, 255)):
    h, w, _ = size
    mask = np.full(size, bg_color, np.uint8)

    for _ in range(7):
        # get random x locations to start line
        x1, y1 = np.random.randint(1, w-1), np.random.randint(1, h-1)
        # get random y locations to start line
        x2, y2 = np.random.randint(1, w-1), np.random.randint(1, h-1)
        # get random thickness of the line drawn
        thickness = np.random.randint(
            min(1, thickness_range[0]), thickness_range[1])
        # draw black line on hte white mask
        cv2.line(mask, (x1, y1), (x2, y2), patch_color, thickness)
    return mask
```

Figure 8.1: Mask Generation Code

```python
import torch
from skimage.metrics import structural_similarity as ssim
from skimage.metrics import peak_signal_noise_ratio as psnr
import numpy as np

def evaluate_model(image1_tensor, image2_tensor):
    # Convert tensors to numpy arrays
    img1 = image1_tensor.cpu().detach().numpy()
    img2 = image2_tensor.cpu().detach().numpy()

    # Normalize image values to [0, 1]
    img1 = img1.astype(np.float32) / 255.0
    img2 = img2.astype(np.float32) / 255.0

    # Calculate SSIM
    ssim_value, _ = ssim(img1, img2, full=True)

    # Calculate PSNR
    psnr_value = psnr(img1, img2)

    return ssim_value, psnr_value
```

Figure 8.2: Evalucation Metrics Code

```python
class PartialConv(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, stride=1,
                 padding=0, dilation=1, groups=1, bias=True):
        super().__init__()
        self.input_conv = nn.Conv2d(in_channels, out_channels, kernel_size,
                                    stride, padding, dilation, groups, bias)
        self.mask_conv = nn.Conv2d(in_channels, out_channels, kernel_size,
                                   stride, padding, dilation, groups, False)
        self.input_conv.apply(weights_init('kaiming'))

        torch.nn.init.constant_(self.mask_conv.weight, 1.0)

        # mask is not updated
        for param in self.mask_conv.parameters():
            param.requires_grad = False

    def forward(self, input, mask):
        # http://masc.cs.gmu.edu/wiki/partialconv
        # C(X) = W^T * X + b, C(0) = b, D(M) = 1 * M + 0 = sum(M)
        # W^T* (M .* X) / sum(M) + b = [C(M .* X) – C(0)] / D(M) + C(0)

        output = self.input_conv(input * mask)
        if self.input_conv.bias is not None:
            output_bias = self.input_conv.bias.view(1, -1, 1, 1).expand_as(
                output)
        else:
            output_bias = torch.zeros_like(output)

        with torch.no_grad():
            output_mask = self.mask_conv(mask)

        no_update_holes = output_mask == 0
        mask_sum = output_mask.masked_fill_(no_update_holes, 1.0)

        output_pre = (output - output_bias) / mask_sum + output_bias
        output = output_pre.masked_fill_(no_update_holes, 0.0)

        new_mask = torch.ones_like(output)
        new_mask = new_mask.masked_fill_(no_update_holes, 0.0)

        return output, new_mask

```

Figure 8.3: Partial Convolution Code

```python
class CelebDatasetFast(Dataset):
    def __init__(self, split, transform, total=200000):
        self.root_dir = "./dataset"
        self.partition_frame = pd.read_csv("./list_eval_partition.csv")
        self.transform = transform
        self.train = split == 'train'
        self.test = split == 'test'
        self.val = split == 'val'
        self.total= total
        self.train_frame = self.partition_frame.loc[self.partition_frame['partition'] == 0]
        self.val_frame = self.partition_frame.loc[self.partition_frame['partition'] == 1]
        self.test_frame = self.partition_frame.loc[self.partition_frame['partition'] == 2]

    def __len__(self):
        if self.train:
            return len(self.train_frame)
        elif self.val:
            return len(self.val_frame)
        elif self.test:
            return len(self.test_frame)

    def __getitem__(self, index):

        mask = torch.from_numpy(gen_line_mask((178, 178, 3), (7, 12))).permute((2,0,1))/255

        if self.train:
            img_name = os.path.join(self.root_dir, self.train_frame.iloc[index, 0]) # type: ignore
            image = Image.open(img_name)
        elif self.val:
            img_name = os.path.join(self.root_dir, self.val_frame.iloc[index, 0]) # type: ignore
            image = Image.open(img_name)
        elif self.test:
            img_name = os.path.join(self.root_dir, self.test_frame.iloc[index, 0]) # type: ignore
            image = Image.open(img_name)
        img = self.transform(image)
        return (mask,torch.maximum(mask,img)),img
```

Figure 8.4: Custom Dataset Class for Data Processing

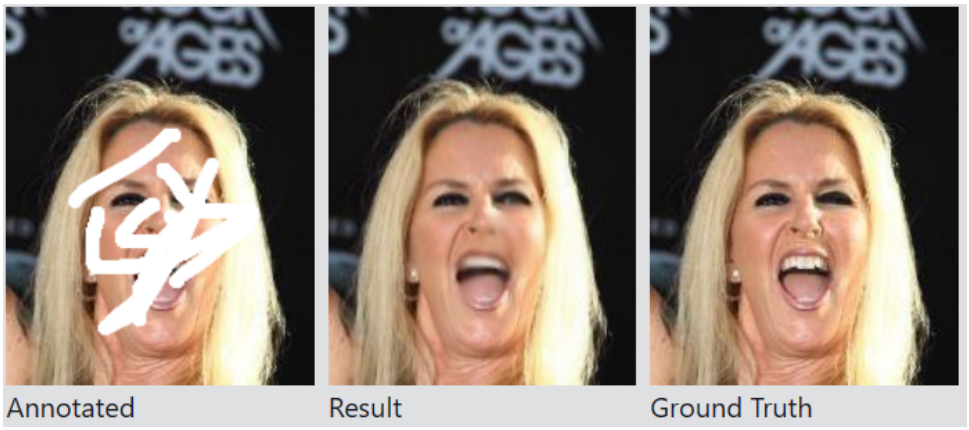


Figure 8.5: Image Inpainting Example 1

Figure 8.6: Image Inpainting Example 2

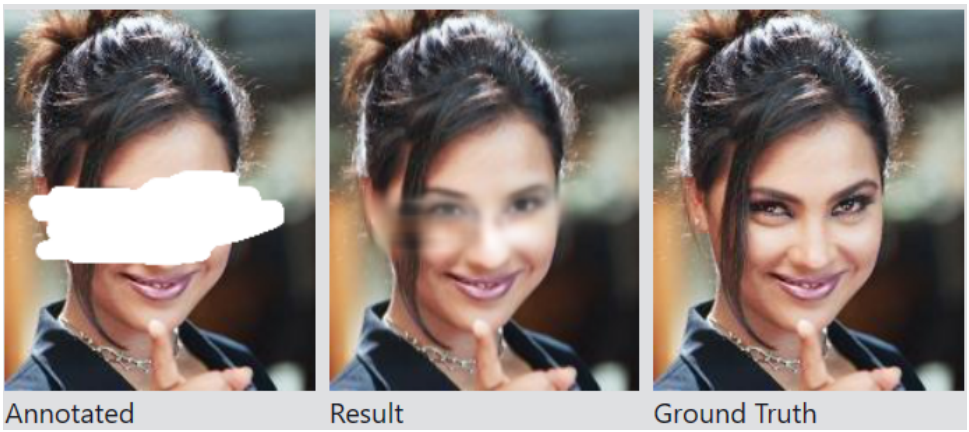


Figure 8.7: Image Inpainting Example 3

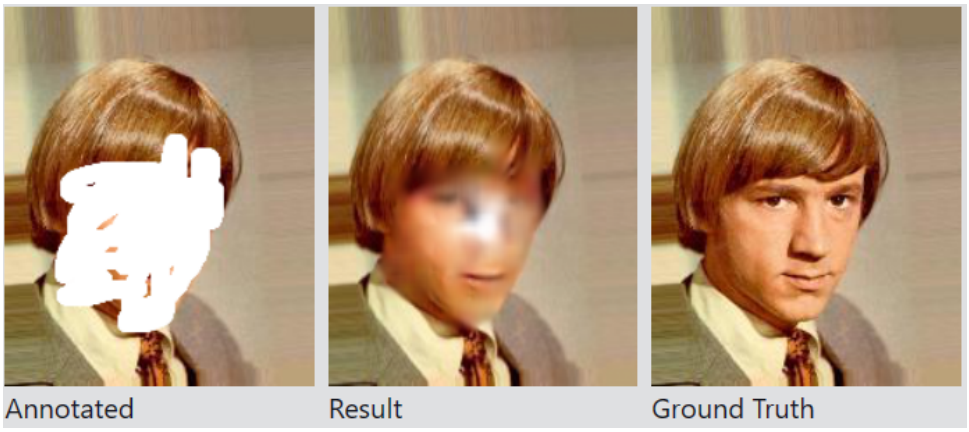


Figure 8.8: Image Inpainting Example 4

Figure 8.9: Image Inpainting Example 5