A project report on

"Image Inpainting Using Deep Learning"

Submitted in partial fulfillment for the award of the degree of

Bachelor of Technology

in

Electronics & Communication Engineering

by

Jyotirmoy Nath (201902031020)

Bittu Mishra (201902033036)

John Doley (201902031008)

Under the supervision of

Mr. Haradhan Chel

(Assistant Professor, Department of ECE)

Department of Electronics & Communication Engineering

केन्द्रीय प्रौद्योगिकी संस्थान कोकराझार

CENTRAL INSTITUTE OF TECHNOLOGY, KOKRAJHAR

(Deemed to be University under MoE, Govt. of India)

2022-23

DECLARATION

We hereby declare that the project work entitled **"Image Inpainting Using Deep Learning"** is authenticated work carried out by us under the guidance of **Mr. Haradhan Chel** for the partial fulfillment of the award for B-Tech in Electronics & Communication Engineering and this work has not been submitted elsewhere for similar purpose except to the Department of Electronics & Communication Engineering at Central Institute of Technology, Kokrajhar.

Date: Jyotirmoy Nath(201902031020)

Place: Kokrajhar Bittu Mishra(201902033036)

John Doley(201902031008)

DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING

CENTRAL INSTITUTE OF TECHNOLOGY

KOKRAJHAR - 783370, INDIA

CERTIFICATE of SUBMISSION of PROJECT REPORT

This is to certify that the project report entitled **"Image Inpainting Using Deep Learning"** submitted by **Jyotirmoy Nath (201902031020), Bittu Mishra (201902033036) and John Doley (201902031008)** to **Central Institute of Technology, Kokrajhar** towards partial fulfillment of requirements for the award of the degree of Bachelor of Technology in **Electronics & Communication Engineering**.

HEAD OF THE DEPARTMENT
Mr. Haradhan Chel
Assistant Professor
Department of Electronics &
Communication Engineering

DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING

CENTRAL INSTITUTE OF TECHNOLOGY

KOKRAJHAR - 783370, INDIA

CERTIFICATE FOR APPROVAL

This is to certify that the project report entitled **"Image Inpainting Using Deep Learning"** submitted by **Jyotirmoy Nath (201902031020), Bittu Mishra (201902033036) and John Doley (201902031008)** to **Central Institute of Technology, Kokrajhar** towards partial fulfillment of requirements for the award of the degree of Bachelor of Technology in **Electronics & Communication Engineering** is a record of bonafide work carried out by him under my supervision and guidance during the session 2022-23.

SUPERVISOR
Mr. Haradhan Chel
Assistant Professor
Department of Electronics &
Communication Engineering

Abstract

Image inpainting is a technique to fix the defective regions of an old and damaged image. It has been an important domain in the research of image processing and computer vision for many years. But addressing this problem is challenging due to the loss of information in the defective regions. Based on the approaches and adopted techniques, existing methodologies are divided into two categories: a) traditional methods and b) deep learning-based methods. Furthermore, a list of methods for each category is available that address various types of visual distortions. This project performs a survey of the existing deep learning-based image inpainting techniques. We have developed a shallow UNET for performing the inpainting of the images of a publicly available dataset. Results depict that UNET is an effective deep learning architecture that performs inpainting for images with low and moderate distortion.

Keywords: Image Inpainting, Deep Learning

Acknowledgement

We would like to express our deepest gratitude to our guide, **Mr. Haradhan Chel**, Assistant Professor, Department of Electronics and Communication for his valuable guidance, consistent encouragement, personal caring, and timely help and for providing us with an excellent atmosphere for doing our research. Despite the busy schedule, he has extended his cheerful and cordial support to us, without which we could not have completed our project work.

We would also like to thank all teaching and non-teaching staff of the Electronics and Communication Department, Central Institute of Technology, Kokrajhar for their constant support and encouragement given to us.

Last but not least it is our great pleasure to acknowledge the wishes of our friends and well-wishers, both in academic and non-academic spheres.

Contents

List of Figures

List of Tables

Chapter 1

Intro

Image inpainting is the process of reconstructing damaged/missing parts of an image. It has numerous uses in computational photography, image-based rendering, and picture editing. Early works [1], [2] attempted to solve the problem using ideas similar to texture synthesis [3], [4], i.e. by simply copying background patches into holes starting from low-quality-resolution to high quality-resolution. It's important to note that these techniques work well for inpainting backgrounds in pictures but fall short in situations where:

1. It is possible that the neighboring areas do not contain enough relevant information, such as pixels, to complete the absent portions.
2. The missing regions require the inpainting system to infer the properties of the would-be-present objects.

With the development of deep learning technologies, the use of artificial neural network models has become more popular to solve this problem. These techniques are capable of carrying out the entire image inpainting procedure automatically without the need for people. Image inpainting can be used for restoring missing pixels of an image. Traditional image restoration methods are designed to fill missing pixel values in a way that is similar to the neighboring pixel values. These methods usually give poor results in cases with large missing areas.

Modern methods are usually trained with millions of images from thousands of different labels using supervised machine learning. For example, a generative adversarial

FIGURE 1.1: Example of Image Inpainting on Images. Missing Regions are shown in white. The left is the Input image and the right is the Output Image

network, which can generate new data, is frequently used in such studies. A neural network alone cannot understand the places that need to be filled in a given input image. The output image is generated by passing through various network layers.[5]

1.1 Motivation

For consumers, the study of image inpainting has always been a difficult task. With the rising demand for taking pictures, lots of effort has been made in building efficient tools for users.

1. Image inpainting is needed to put together a damaged or incomplete image. Suppose for instance that some of your favorite old photographs of you as a child with your grandparents have become ruined for various reasons. In this case, Image inpainting can be very much helpful.

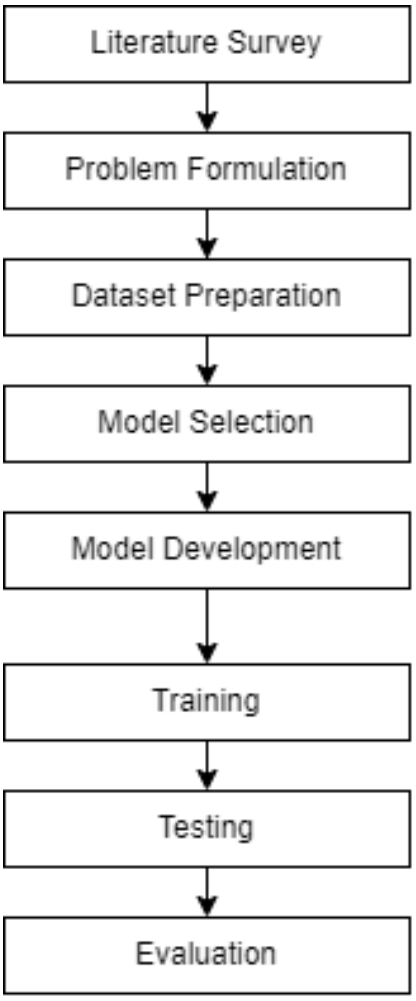
2. Inpainting is useful in the case where museums might not have the budget to appoint a skilled artist to restore deteriorated paintings.

1.2 Objective

This project aims to use advanced techniques to fill the missing parts of an image using deep learning methods. These methods are popular now a days as they perform better than traditional approaches in complex situations. Training in deep learning methods requires huge datasets which is made possible by collecting and labeling a vast number of images. Traditional methods use surrounding pixel values to fill in missing pixels, which often leads to unsatisfactory results when there are large missing areas. In contrast, modern deep learning methods are trained with millions of images and labels using supervised machine learning to produce better results.

1.3 Project Planning

When planning a project, it is essential to have a well-defined project plan that outlines the project goals, timelines, deliverables, and resources required for the project. It involves defining project goals, gathering data, selecting an appropriate algorithm, setting project timelines, allocating resources, testing, evaluating, documenting, and sharing the project plan and results. The steps we have considered when planning the project are shown below in the figure:



Chapter 2

Literature Survey

Recent research works for image inpainting can be divided into two approaches. The first one includes the traditional methods with low-level features. The second one contains modern deep learning-based methods i.e., training convolutional neural networks to predict pixels for the missing parts.

1. C. Ballester et al., traditional patch-based techniques rely on variational algorithms to disseminate data from the background regions. However, these methods are limited in their ability to handle situations where the surrounding areas lack the necessary information, such as pixels, to replace the missing portions. [6][7] [3][4]

2. Simakov et al., suggest using a bidirectional patch similarity-based method to improve the representation of nonstationary visual data for re-targeting. This technique involves analyzing visual data patches in both forward and backward directions to create a more precise representation of moving images, which can then be utilized to adjust the size or shape of the images. [8]

3. R. Kohler et al. have explored the use of initial deep learning approaches as a viable solution for image inpainting. They have employed convolutional neural networks to denoise and restore small regions of images.

4. Zhou et al. [9], aimed to fill the desired area by creating textures with patch-based algorithms for the Image inpainting method.

2.1 Problem Statement

1. Traditional methods for image inpainting, such as interpolation or texture synthesis, often produce unsatisfactory results.
2. Deep learning-based approaches have shown promising results in image inpainting, with the ability to generate realistic and high-quality reconstructions.
3. Traditional inpainting methods often struggle with handling large datasets due to their computational and memory limitations.

To overcome those problems we used modern deep learning-based methods.

Chapter 3

Inpainting Methods

The techniques utilised for image inpainting are very diverse. These techniques all have mathematical underpinnings. The methods can be categorized into groups based on the algorithms that they employ to produce the desired results. The first of these techniques, known as the traditional technique, tries to fill in the missing portion of the main title graphic using only the components already present in the image. Modern approaches or methods based on deep learning, make up the second category of these methods.

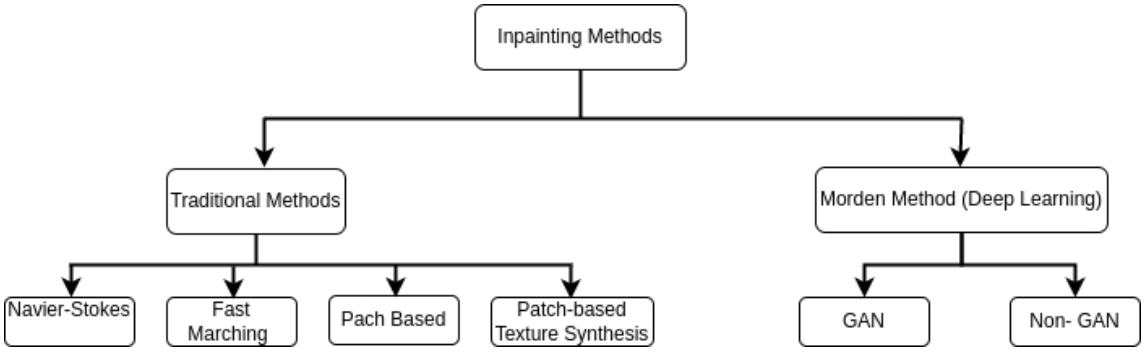


FIGURE 3.1: Inpainting Methods

3.1 Traditional Methods

Without deep learning, the field of computer vision is vast. Object identification was still possible before Single Shot Detectors (SSD) were invented, however, it was

far from being as precise as it is now. Similarly, there are a few traditional computer vision methods for performing picture inpainting.

3.1.1 Navier Stokes Method

Fundamentally, Navier-Stokes equations are partial differential equations that describe how a viscous, incompressible fluid flows. These approaches are commonly based on the notion of energy, momentum and mass conservation. These equations are utilised over a wide range of applications across numerous industries including the modelling of ocean currents, weather patterns and even the construction of vehicles like cars and aeroplanes.
The algorithm utilised in this study moves from the known region to the unknown region by following the picture's edges. The term "isophotes" refers to the lines produced by pixels of the same intensity. The gradient vector's direction in the boundaries was maintained throughout this operation. The colour information is determined to lower the minimum variance when the isophotes are collected, and the inpainting is done by filling in the unknown pixels.[10]

They integrate ideas from partial differential equations and fluid mechanics in their [11] work. It is based on the idea that an image's edges should naturally be continuous. After obtaining the isophotes, colour information is calculated to lower the minimum variance and the inpainting procedure is done by filling in the unidentified pixels.

3.1.2 Fast Searching Methods

Ahmet created an inpainting method based on the rapid searching method in this paper, which was published in 2004 [12]. A filling procedure is created that moves incrementally inward from the boundary pixels of an area where the pixel information is absent. He suggested the following:

1. Take a normalised weighted sum of pixels from a neighbourhood of the missing pixels to approximate the missing pixels. When a collection of pixels is inpainted the boundary that defines this neighbourhood is updated.

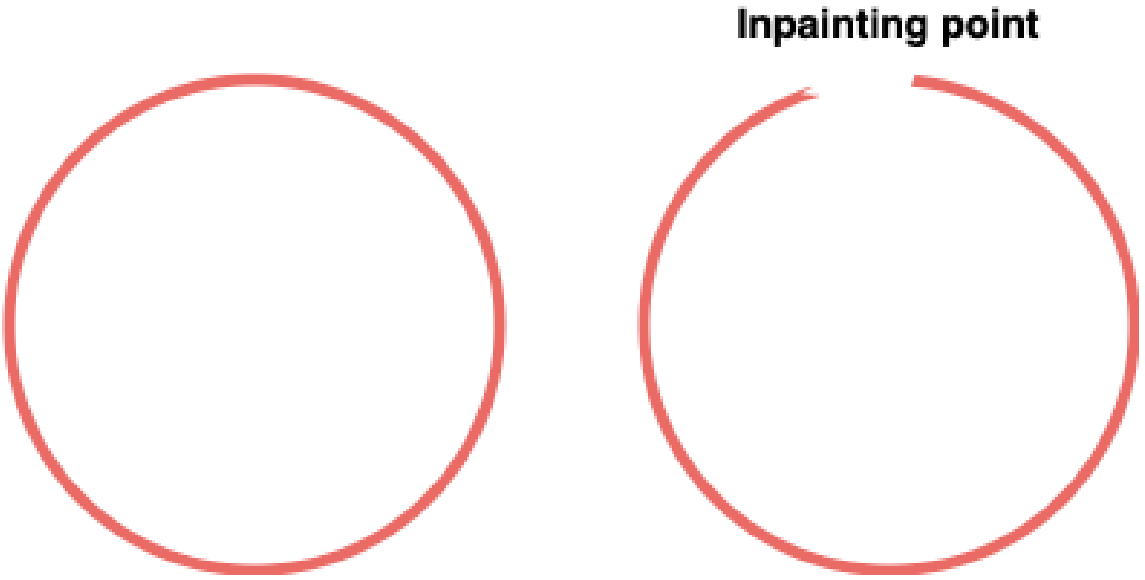


FIGURE 3.2: Edges in an Image are assumed to be Continuous in Nature

2. The gradients of the nearby pixels are utilised to estimate the colour of the pixels.

To fill in missing pixels in an image, we can use a method called weighted summation. We only need to look at a small group of pixels near the missing ones to do this. The weights we use for the summation are based on the border and how close the pixels are to each other.

To apply this method, we can use a technique called fast searching. This involves moving from one pixel to the nearest neighboring pixel and computing all the pixels in between. We keep doing this until we have filled in the whole area with the missing pixels.

3.1.3 Patch-based Methods

Patch-based picture inpainting techniques are used to fill in damaged areas of an image by using undamaged areas. The goal is to find the most similar patches. To put this strategy into practice, an algorithm is needed to move the acquired regions to the areas that need to be filled. Another method is to choose an algorithm that compares regions in different parts of the image to the regions that need to be created.[7].

Patch-based techniques work well, but they assume that the missing data in the image is present elsewhere. These approaches also require a lot of processing power since they often involve searching and comparing

3.1.4 Patch-based Texture Synthesis

Texture techniques are also considered to be one of the early techniques used to solve the image inpainting problem. The notation of patches was first introduced with these methods. Texture synthesis methods randomly utilize the pixel data of neighbouring areas to complete the missing portions of images. This involves selecting analogous pixels from the same image that share similar neighbourhood features.

Zhou et al.[5], aimed to utilize patch-based algorithms in their image inpainting approach to generate textures that can fill in the targeted regions.

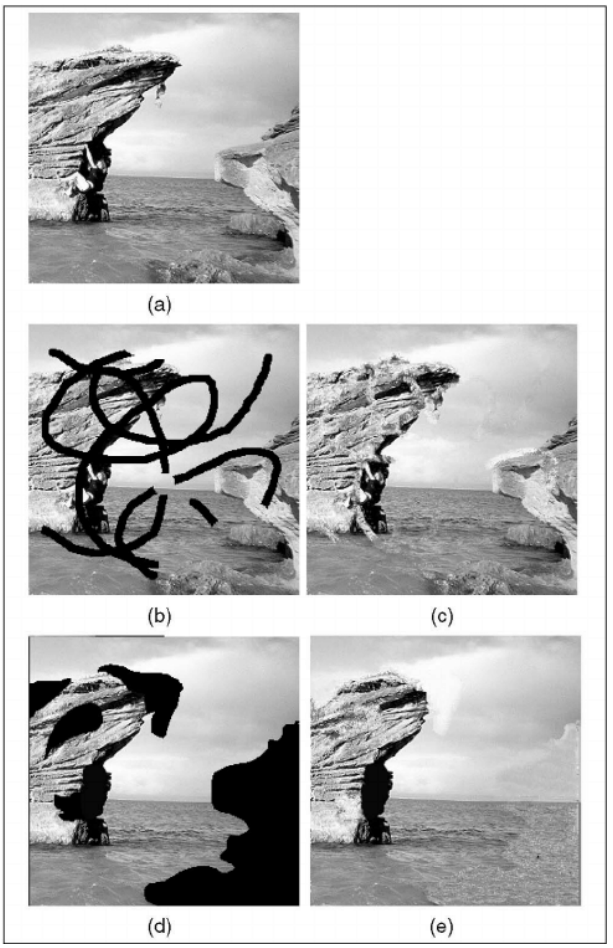


FIGURE 3.3: Inpainted Image with Patch Based Texture Synthesis

3.2 Modern Methods -Deep Learning Based

In this method, an image's missing pieces are predicted using a neural network that has been trained to make predictions that are both aesthetically and semantically coherent. The issue of image inpainting has been solved using a far more stunning and accurate method using deep learning algorithms. Convolution Neural Networks (CNNs), for example, have been modified for picture inpainting. The plan was to use a training set of images to teach CNNs how to fill in or complete missing regions or portions in images. The original method for dealing with image inpainting involved training a special model to fill in a missing spot in a given place across images.

Inpainting techniques that are based on deep learning can be divided into two categories[13] :

1. Non-GAN-Based Systems
2. GAN-Based Systems

3.2.1 Non-GAN-Based Systems

The crucial issue of image inpainting has been solved using several algorithms.
In (Xie, Xu, and Chen 2012), a method made for photos with lesser resolution is provided. The authors used a pre-trained denoising Auto-Encoder (DA) model with a sparse approach. Stacked Sparse Denoising Auto-Encoders (SSDA) is the name of the model. The overlapping image patches are treated as data items by the blind SSDA method. The corrupted image patches, also known as noisy image patches, for i = 1, 2,..., N are given to the model together with the ground truth patches during training.

Further, in Non GAN Based Systems there are two approaches:

1. Vanilla CNN based image inpainting
2. Partial convolution based approach.

3.2.2 Vanilla Convolutional Autoencoder

Autoencoder-based feedforward neural networks produce outputs that are identical to their inputs. These networks first reduce the dimensions of the input and then use this condensed representation to generate the output. The resulting code, also referred to as the latent-space representation, serves as a compressed and summarized version of the input.
Encoder, code, and decoder are the three main parts of an autoencoder. The input is compressed by the encoder, which also creates a code. The decoder then reconstructs the input exclusively using the code.

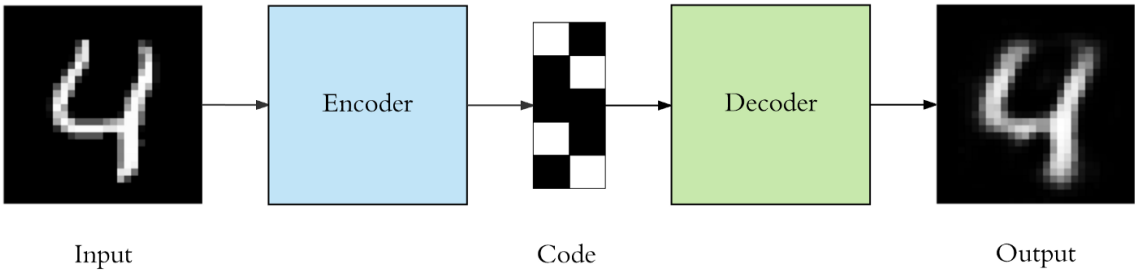


FIGURE 3.4: Autoencoder

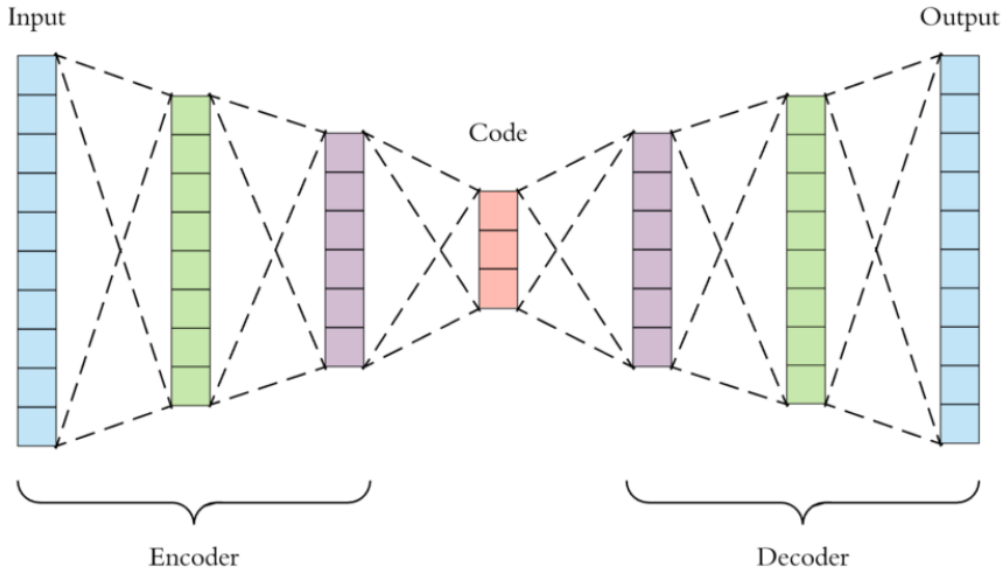


FIGURE 3.5: Simple Autoencoder Architecture

A vanilla autoencoder is taught to reconstruct the input, i.e $g(f(x)) = x$ via a CNN-based manner. The situation is not unique though. If the autoencoder is not carefully trained, it has been observed that it tends to memorise the data and not

learn any relevant prominent features. Instead of putting a cap on the encoder and decoder's capacity, regularised autoencoders are employed. The loss function that is typically utilised encourages the model to learn more than just how to copy the input. These additional characteristics may include the representation's sparsity, resistance to noise, or responsiveness to missing input. From data examples, autoencoders automatically learn. It indicates that it is simple to train customised versions of the algorithm that will work well with a particular kind of input and that it does not involve any new engineering, only the use of existing techniques..
The Vanilla autoencoder is a three-layer net or a neural network with one hidden layer, in its most basic configuration. It uses loss functions like the Adam optimizer and the mean squared error for reconstructing the input, and also the input and output are identical.

3.2.3 Partial Convolution Based

Partial convolution-based image inpainting methods are cutting-edge technique in computer vision that restores the missing or damaged portions of an image. The technique is based on the use of a partial convolution operator, which is a modification of the traditional convolution operator used in image processing.

The partial convolution operator takes the valid pixels in the input image and the pixels that have been damaged or removed. It only convolves the valid pixels in the input image, avoiding the artifacts in the restored image. The operator ensures that the reconstructed image is visually consistent with the original image.

The partial convolution-based image inpainting methods have several advantages over traditional image inpainting techniques. They can produce visually consistent and high-quality restorations of images with complex textures and structures. They are also able to handle images with arbitrary shapes and sizes, making them useful in a variety of applications. One of the benefits of partial convolution-based image inpainting methods is their ability to handle large missing or damaged portions of an image. Traditional image inpainting techniques often introduce artifacts or produce blurry images when large portions of an image are missing, but partial convolution-based methods are able to produce high-quality restorations even for large missing portions.

If the input has some correct pixels, then a mask with multiple partial convolution layers will ultimately have all ones.

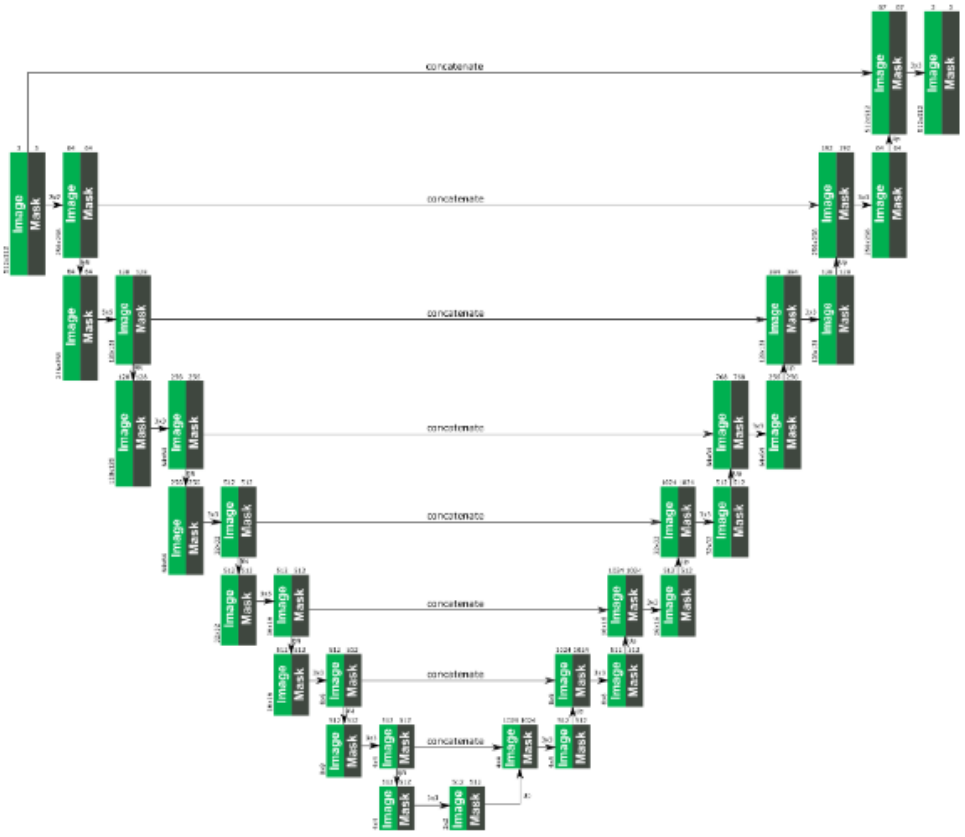


FIGURE 3.6: Use of Partial convolution in an Autoencoder Decoder Architecture

3.3 GAN Based Systems

Convolutional Neural Networks (CNNs) have improved image inpainting, but they still have some big issues. When training a CNN, the goal is to minimize an objective loss function. This function is typically calculated using the Euclidean distance between the recovered pixels and the ground truth pixels. However, using this measure often results in blurry images because averaging all the outputs to minimize the Euclidean distance leads to fuzzy results.

GANs always have two major networks, each of which plays a certain role and competes with the other

1. Discriminator: D

2. Generator: G

The discriminator D grades the picture according to D(x) $\rightarrow$ [0,1], while the generator G transfers the latent space noise vector z into an image, where: G(z)$\rightarrow$ x. Both the discriminator and generator are parametric functions. The GAN learning process can be compared to a two-player game where the discriminator is in charge of telling bogus images from real ones, or vice versa. The generator is in charge of creating phoney images that mimic actual ones in order to trick the discriminator. Competitive training is combined for the discriminator and generator.

The process of a GAN can be described using the following steps:

1. The generator produces a picture.
2. A variety of photos from the real, authentic training dataset are provided to the discriminator along with this produced image.
3. The discriminator accepts both genuine and fake images and outputs a judgement as a probability, or a number between 0 and 1, where 0 denotes the prediction of a genuine image and 1 denotes the prediction of a fake image.

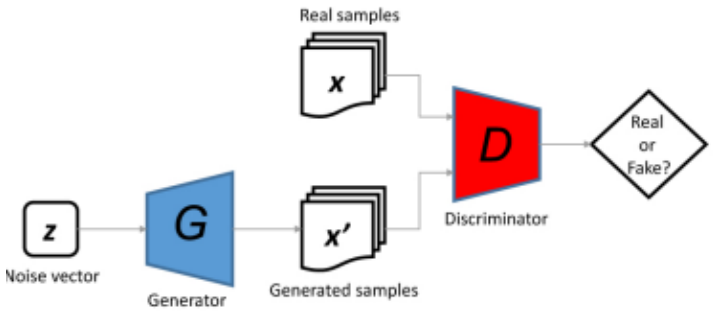


FIGURE 3.7: GAN Structure

Chapter 4

Deep Learning

4.0.1 Introduction

Deep Learning is a rapidly growing field of artificial intelligence that uses neural networks to model and solve complex problems. It has gained widespread attention due to its ability to outperform traditional machine learning methods on a variety of tasks, such as image and speech recognition, natural language processing, and robotics. Deep Learning is a subset of machine learning that utilizes deep neural networks to learn representations of data and extract features through multiple layers of interconnected nodes. These networks are designed to mimic the structure and function of the human brain, enabling them to learn and recognize patterns in large datasets with high accuracy. As such, Deep Learning is a powerful tool that has the potential to revolutionize the way we solve problems and make decisions in many different industries and domains.

4.1 Overview of Deep Learning-Based Neural Network

4.1.1 Neural Network

Although it took a while for neural networks to be effectively implemented, the concept of neural networks, which has been around for more than 70 years, is similar

to the neurons in the human brain. McCulloch and Pitts develop the first neural network, but they lacked the technical resources to run it. At MIT, Clark and Farley were able to operate the first neural network later in 1954.

The fact that neural networks are composed of neurons is the key component that makes them function similarly to neurons seen in the human brain. These neurons Understands how human brain neurons function and how they connect to artificial neural networks (ANN) is a fundamental necessity for understanding how neural networks operate. A human brain contains billions of neurons. These neurons exchange electrical signals with one another. Neurons generate their output in response to the various signal types they receive. It produces an electrical impulse in response only if the value they receive is beyond a predetermined threshold.

Depending on their size, ANNs can have a variety of layers. Each layer's output is transmitted to the following layer by neurons.

Neuron structure is shown below:

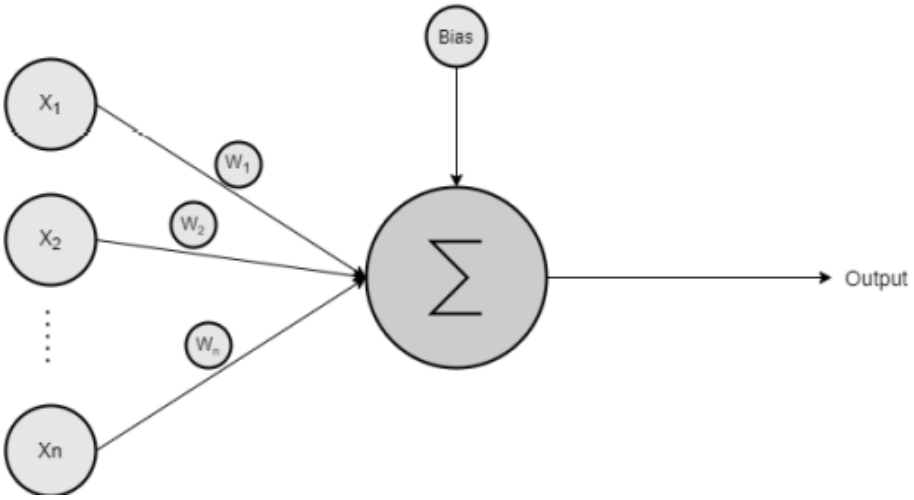


FIGURE 4.1: Neuron

4.1.2 Artifical Neural Network

The term "artificial neural network" refers to biological neural networks that create the framework of the human brain. Artificial neural networks also have neurons that are related to one another at different levels of the network, similar to how neurons in the real brain are coupled to one another. These neurons are referred to as nodes.

Artificial Neural Networks are made up of three layers:

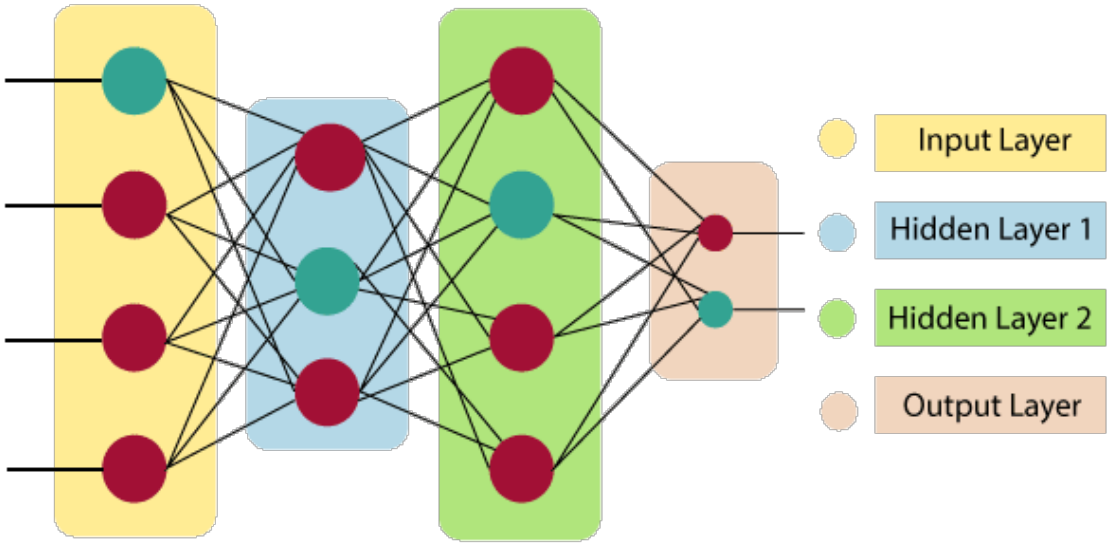


FIGURE 4.2: Basic Architecture of an Artificial Neural Network

Input Layer: As the name implies, it accepts inputs in a variety of programming-provided formats.

Hidden Layer: Between the input and output layers is the hidden layer. It makes all the computations necessary to uncover patterns and buried features.

Output Layer: The input is transformed using the hidden layer, resulting in an output that is conveyed using this layer.

4.1.3 Convolutional Neural Network(CNN)

A Convolutional Neural Network (ConvNet/CNN) is a deep learning algorithm capable of taking an input image, assigning importance (learnable weights and biases) to various aspects/objects in the image, and distinguishing one from the other. When compared to other classification methods, CNN requires substantially less pre-processing. While filters in primitive methods are hand-engineered, CNN can learn these filters/characteristics with adequate training.

There are two main parts of CNN:

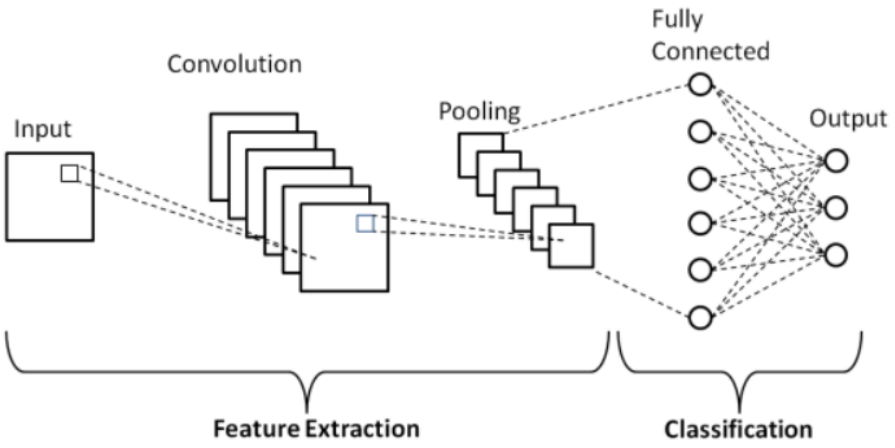


FIGURE 4.3: Basic Architecture of a CNN Network

1. A convolution tool that separates and identifies the image's numerous features for examination in a process known as Feature Extraction.

2. A fully connected layer that uses the convolution process output to forecast the image class based on the features retrieved in preceding stages.

1. Convolutional Layer

This is the first layer that is utilized to extract the different features from the input photos. This layer performs the mathematical action of convolution between the input image and a filter of a specific size MM. The dot product between the filter and the sections of the input image about the size of the filter MM is calculated by sliding the filter across the input image. The result is known as the Feature map, and it contains information about the image such as its corners and edges. This feature map is then supplied to further layers, which learn various features from the input image.

2. Pooling Layer

A Convolutional Layer is usually followed by a Pooling Layer. This layer's major goal is to lower the size of the convolved feature map to reduce computational expenses. This is accomplished by reducing the connections between layers and operating independently on each feature map. There are various sorts of Pooling procedures depending on the approach utilized. The largest piece from the feature map is used in Max Pooling. Average Pooling computes the average of the elements in a predefined

image segment size. Sum Pooling computes the total sum of the components in the predefined section. The Pooling Layer is typically used to connect the Convolutional Layer with the FC Layer.

3. Fully Connected Layer
The Fully Connected (FC) layer, which includes weights and biases as well as neurons, is used to connect neurons from different layers. These layers are often placed before the output layer and constitute the final few layers of a CNN Architecture. The input image from the preceding layers is flattened and supplied to the FC layer in this step. The flattened vector is then sent via a few additional FC layers, where the mathematical function's operations are often performed. At this point, the classification procedure is initiated.

4. Dropout
Overfitting in the training dataset is common when all features are connected to the FC layer. Overfitting happens when a model performs so well on training data that it has a negative influence on the model's performance when applied to new data. To address this issue, a dropout layer is used, in which a few neurons are removed from the neural network during the training process, resulting in a smaller model.

4.1.4 Deep Learning Parameters

In deep learning, parameters are values that are learned during the training process of a neural network. These parameters represent the weights and biases that are used to transform the input data into the output. The values of these parameters determine how well the model performs on a given task. Finding the optimal values for these parameters is the primary objective of the training process, and it is typically achieved through a process called gradient descent. Fine-tuning the parameters of a deep learning model can significantly improve its accuracy and performance on a given task.

4.1.4.1 Optimizers

Optimizers are techniques or approaches that adjust the characteristics of the neural network, such as weights and learning rate, to reduce losses. The following optimizers are well explained below:

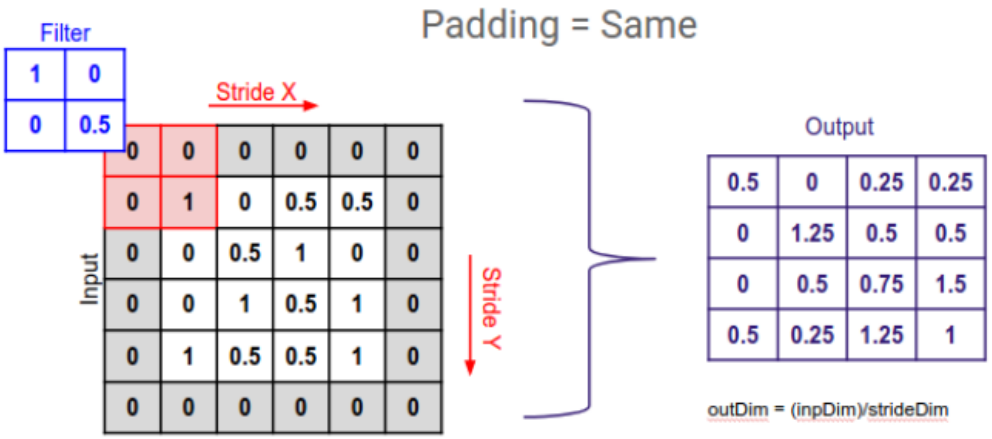
(i) Gradient Descent: Gradient descent is a common optimization technique that is used in linear regression, classification, and neural network backpropagation. It is based on the first-order derivative of a loss function and is used to identify the weights that need to be adjusted to reach the minimum loss. This approach is used in backpropagation to propagate the loss from one layer to the next and update the model's parameters, also known as weights, to minimize the loss.

(ii) Stochastic Gradient Descent: Gradient Descent is a variation. It attempts to update the model's parameters more regularly. The model parameters are changed after each training example's loss computation. As a result, if the dataset contains 1000 rows, SGD will update the model parameters 1000 times in one dataset cycle, rather than once as in Gradient Descent.[14]
(iii) Adagrad : One of the drawbacks of all the optimizers described is that the learning rate is consistent across all parameters and cycles. The learning rate is altered by this optimizer. It modifies the learning rate for each parameter and time step 't'. This is a second-order optimisation algorithm. It operates on an error function's derivative.

(iv) Adam: Adam (Adaptive Moment Estimation) works with first and second-order momentums. The idea behind the Adam is that we don't want to roll too fast just because we can jump over the minimum; instead, we want to slow down a little for a more cautious search. Adam, like AdaDelta, preserves an exponentially decaying average of past squared gradients as well as an exponentially decaying average of past gradients M(t).

4.1.4.2 Padding

In convolutional neural networks, padding is the process of adding extra pixels to an image when it is processed by the CNN kernel. The amount of padding determines how many pixels are added to the image, and what value those pixels have. For example, If one-pixel padding is used, a border of pixels with a value of 0 will be added to the image.



4.1.4.3 Kernels

The kernel function translates data from its native space to a higher dimensional feature space indirectly. Kernel-based machine learning methods are generally used on data that cannot be separated linearly in its original space. Although kernel methods are among the most elegant aspects of machine learning users face difficulties in defining or selecting an appropriate kernel function with optimised parameter settings for their data.

4.1.4.4 Learning Rate

The learning rate is a tuning parameter in an optimisation algorithm that controls the step size at each iteration as it moves towards a loss function minimum.

The step size or "learning Rate" is the amount by which the weights are updated during training. The learning rate is a customizable hyperparameter used in neural network training that has a modest positive value, often between 0.0 and 1.0.

4.1.4.5 Activation Function

In a neural network, activation functions compute the weighted total of inputs and biases, which are then used to determine whether a neuron can be activated or not. It manipulates the provided data and generates an output for the neural network that includes the data's parameters. In the hidden layers of a linear model, a linear mapping of an input function to output is performed before the final prediction for each label is supplied. The transformation of the input vector x is given by,

$$fx = w^T + b$$

where, x= input, w = weight, and b = bias. The few activation functions are discussed below:

(i) ReLU

Right now, the ReLU is the most widely utilized activation function in the world. Because it is employed practically in most of convolutional neural networks and deep learning algorithms. ReLU has a value between 0 and infinity. Both the function and its derivative are monotonic.

(ii) Softmax

The Softmax function computes the event's probability distribution over 'n' distinct events. This function will, in general, calculate the probabilities of each target class across all potential target classes. The estimated probabilities will later be used to assist identify the target class for the given inputs.

4.1.5 Batch Size

The batch size is a hyperparameter that specifies how many samples must be processed before the internal model parameters are updated. Consider a batch to be a for-loop that iterates across one or more samples and makes predictions. The predictions are compared to the expected output variables after the batch, and an error is calculated.

4.1.5.1 Epochs

An epoch is one loop across the entire training dataset. Training a neural network usually takes more than a few epochs. In other words, if we feed a neural network

training data in varied patterns over more than one epoch, we aim for improved generalisation when presented with new "unseen" input (test data). An epoch is frequently confused with an iteration. The number of iterations required to complete one epoch is the number of batches or steps through partitioned packets of training data.

4.1.5.2 Iteration

The number of batches equals the number of iterations for a single epoch. An iteration is the number of times a batch of data is processed by the algorithm. In the context of neural networks, this refers to the forward and backward passes. As a result, each time a batch of data is passed through the neural network, an iteration is completed.

4.2 ANN VS CNN

Artificial Neural Network	**Convolution Neural Network**
Fully connected network layers are used.	A partially linked layer is used.
Only suitable for small photos	It can be applied to any image.
The number of parameters is really large.	In comparison to ANN, the number of parameters is much lower.
Despite its high cost, the image is inefficient.	Image is very efficient and less expensive than ANN.
Doesn't learn spatial hierarchy i.e., Higher layer of ANN does not combine the lower layer output	Learn Spatial hierarchy of pattern i.e., Higher layers of CNN are formed by combining the lower layer. This helps to identify the pattern more efficiently than ANN.

Chapter 5

Methodology

This chapter presents an overview of the image inpainting project, detailing the design and production choices made during development. The project's working environment, including setup and dataset, is outlined. The chapter covers preprocessing, architecture, training, and testing tasks. A simplified workflow for the model is shown in Figure 5.1.

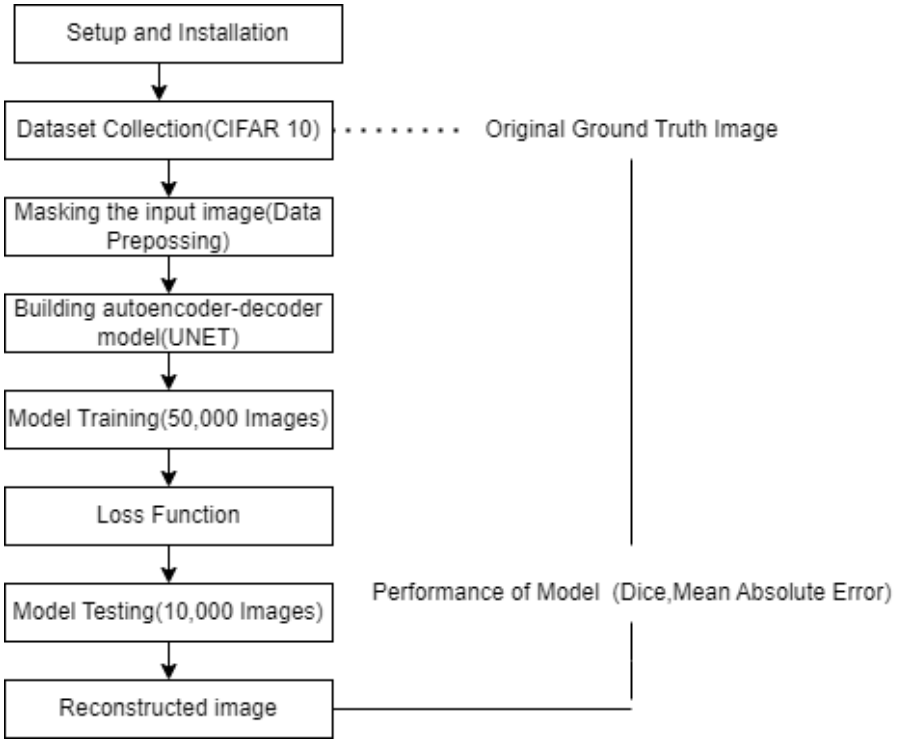


FIGURE 5.1: Working Methodology

5.1 Setup

Python 3 was chosen as the primary programming language for the project due to its user-friendly syntax and widespread use in the deep learning community. The workstation had an Intel i5 processor and 8GB RAM. To enable flexible code snippet execution, Google Colab was utilized, providing open access to the GPU. Python libraries such as Keras, TensorFlow, OpenCV, Numpy, and Matplotlib were employed to handle various data formats in our project.

NumPy: NumPy is a popular library for the Python programming language that enhances its capabilities by enabling support for creating and working with large multi-dimensional arrays and matrices. Additionally, it includes a vast collection of high-level mathematical functions that can be applied to these arrays.

Matplotlib: Matplotlib is a Python library that provides an extensive range of tools for generating static, animated, and interactive visualizations. It is widely used by data visualization professionals and is considered one of the most popular Python packages for this purpose. The library supports the creation of 2D plots from arrays of data and can be utilized on various operating systems.

OpenCV: OpenCV is an extensive open-source library that provides a wide range of functionalities for image processing, computer vision, and machine learning. It enables the identification of different elements, such as objects, faces, and human handwriting in images and videos. By integrating OpenCV with other libraries, such as Numpy, which is a highly optimized numerical operations library, one can significantly expand their capabilities.

TensorFlow: TensorFlow is an open-source library that allows for fast numerical computing, making it a high-performance choice for developing deep learning models. TensorFlow can be used directly, or through wrapper libraries that simplify the development process. The library provides multiple workflows that enable the creation and training of models using Python.

Keras: Keras is an open-source Python library for creating artificial neural networks. It serves as a user-friendly interface for the TensorFlow library and offers a

diverse range of implementations of neural network building blocks, including objectives, layers, activation functions, optimizers, and other tools.

5.2 Dataset Preparation

The data for this experiment was gathered from the well-known dataset **CIFAR10**. (`https://www.kaggle.com/c/cifar-10`). They have our genuine gratitude.

The CIFAR-10 dataset consists of 6000 images per class in 10 classes totaling 60000 32x32 color images. 10000 test photos and 50,000 training images are available. Five training batches and one test batch, each with 10,000 photos, make up the dataset. Exact 1000 randomly chosen photos from each class make up the test batch. The training batches consist of exactly 5000 photos from each class combined. The dataset's classes are listed below, along with 10 randomly chosen photographs from each class:

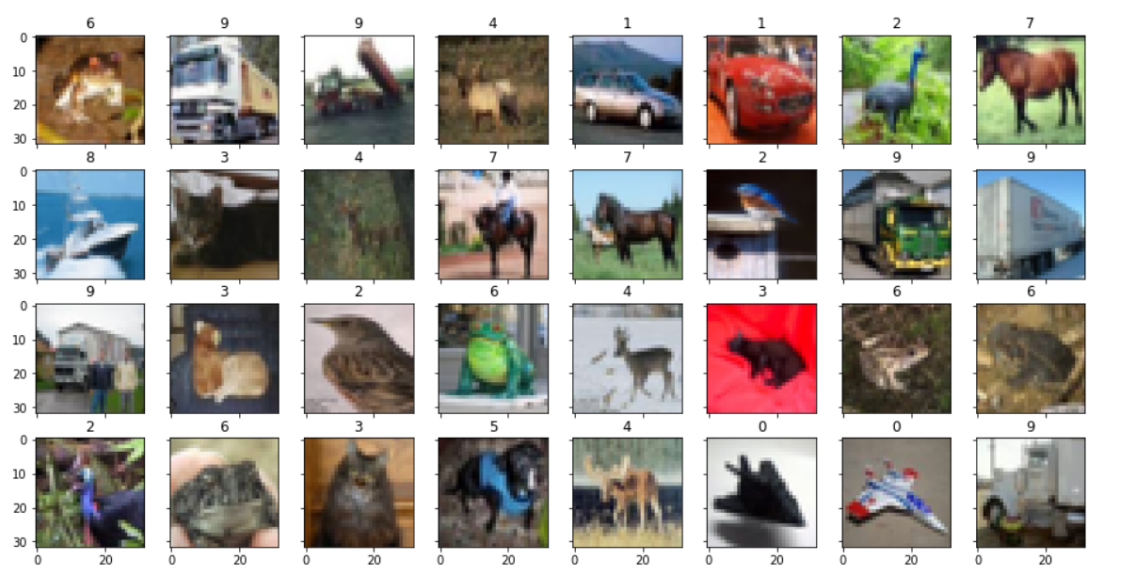


FIGURE 5.2: Sample Images of CIFAR10 Dataset

5.2.1 Data Preprocessing

Data collection is the first step in any deep learning task, followed by data preprocessing to prepare the data for the model. In this project, we artificially degraded our images to generate masked images. For this purpose we have used a common image-processing technique. For training our model, we needed pairs of X and Y (which is the same as X) as it is a self-supervised learning environment. In this case, X represented the original/ground truth images, and "X" represented batches of masked images. The dataset's classes are listed below, and ten randomly selected images from each class are shown:

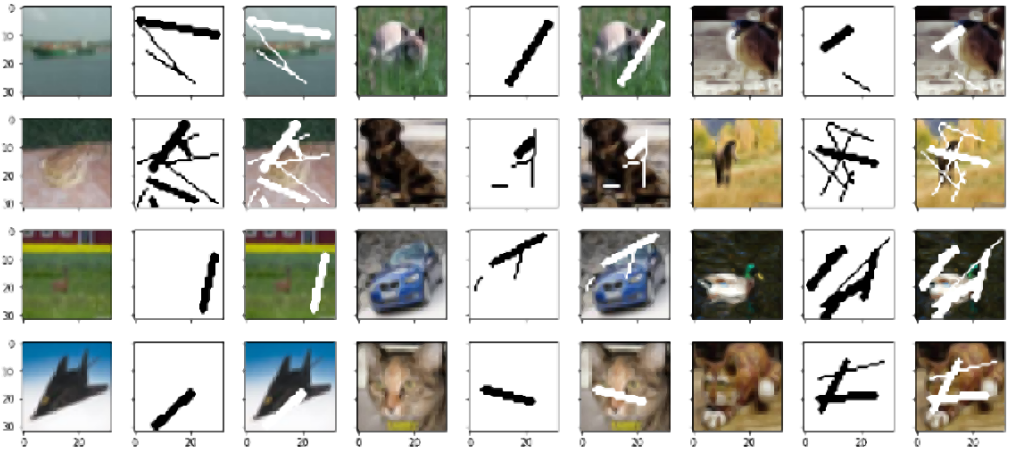


FIGURE 5.3: Original Images, Masks, and Masked Images on CIFAR10

To simplify the masking, we have drawn lines of random length and thickness using OpenCV in the input image, and the original image was considered its labeled image. To make the classification space-invariant, we randomized the position of the lines along with their dimensions.

5.3 Model Selection

This project uses a **Vanilla CNN-based autoencoder model** for image inpainting, which involves filling in missing pixels. Autoencoders are commonly used in addressing issues such as deblurring and artifact removal. The network consists of a decoder that creates the reconstruction, r = g(h), and an encoder that learns a

code to describe the input, h = f(x).
A vanilla CNN-based autoencoder is a type of neural network architecture that is used for image inpainting tasks. It consists of two main parts: an encoder and a decoder. The encoder is a convolutional neural network that learns a compact representation of the input image, while the decoder is also a convolutional neural network that generates the missing parts of the image using the learned representation.

5.3.1 Network architecture

The network architecture we used is based on the U-Net, which uses convolution layers in an encoder-decoder structure to produce recovered images from masked images. The network consists of an encoder, which is a contracting network, and a decoder, which is an expanding network.

5.3.2 UNET:

The network has a U-shaped architecture because it has both a contracting path and an expansive path [15].

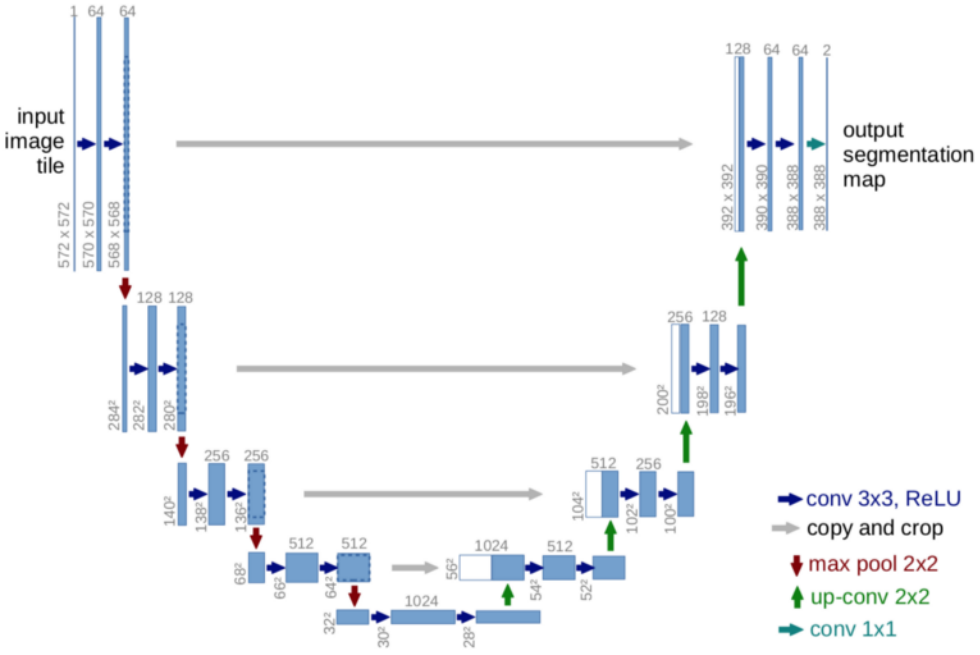


FIGURE 5.4: UNET Architecture

Contracting path/ downsampling

The contracting path is of four blocks and each block consists of:

1. Repeated application of two 3*3 convolutions (unpadded convolutions), each followed by a rectified linear unit (ReLU)
2. 2*2 Max Pooling operation with stride 2

At each downsampling, we double the number of feature channels, starting with 64 feature maps for the first block, 128 for the second, and so on. This contraction path's objective is to obtain the input image's-context to do segmentation.

Expansive path/upsampling

The expansive path is composed of four blocks and each block consists of :

1. Upsampling of the feature map followed by a 2*2 convolution ("up-convolution") that halves the number of feature channels
2. Concatenation of the contracting path's feature map with the similarly cropped version
3. Two 3*3 convolutions, each followed by a ReLU

Advantages

1. To obtain the necessary general information combining localization and context, the U-Net integrates the location information from the downsampling path with the contextual information from the upsampling path.
2. No dense layer, so images of different sizes can be used as input (since the only parameters to learn on convolution layers are the kernel, and the size of the kernel is independent of the size).
3. The use of massive data augmentation is important in domains like biomedical segmentation since the number of annotated samples is usually large.

5.4 Model Implementation

```python
class inpaintingModel:
  '''
  Build UNET like model for image inpaining task.
  '''
  def prepare_model(self, input_size=(32,32,3)):
    inputs = keras.layers.Input(input_size)

    conv1, pool1 = self.__ConvBlock(32, (3,3), (2,2), 'relu', 'same', inputs)
    conv2, pool2 = self.__ConvBlock(64, (3,3), (2,2), 'relu', 'same', pool1)
    conv3, pool3 = self.__ConvBlock(128, (3,3), (2,2), 'relu', 'same', pool2)
    conv4, pool4 = self.__ConvBlock(256, (3,3), (2,2), 'relu', 'same', pool3)

    conv5, up6 = self.__UpConvBlock(512, 256, (3,3), (2,2), (2,2), 'relu', 'same', pool4, conv4)
    conv6, up7 = self.__UpConvBlock(256, 128, (3,3), (2,2), (2,2), 'relu', 'same', up6, conv3)
    conv7, up8 = self.__UpConvBlock(128, 64, (3,3), (2,2), (2,2), 'relu', 'same', up7, conv2)
    conv8, up9 = self.__UpConvBlock(64, 32, (3,3), (2,2), (2,2), 'relu', 'same', up8, conv1)

    conv9 = self.__ConvBlock(32, (3,3), (2,2), 'relu', 'same', up9, False)

    outputs = keras.layers.Conv2D(3, (3, 3), activation='sigmoid', padding='same')(conv9)

    return keras.models.Model(inputs=[inputs], outputs=[outputs])

  def __ConvBlock(self, filters, kernel_size, pool_size, activation, padding, connecting_layer, pool_layer=True):
    conv = keras.layers.Conv2D(filters=filters, kernel_size=kernel_size, activation=activation, padding=padding)(connecting_layer)
    conv = keras.layers.Conv2D(filters=filters, kernel_size=kernel_size, activation=activation, padding=padding)(conv)
    if pool_layer:
      pool = keras.layers.MaxPooling2D(pool_size)(conv)
      return conv, pool
    else:
      return conv

  def __UpConvBlock(self, filters, up_filters, kernel_size, up_kernel, up_stride, activation, padding, connecting_layer, shared_layer):
    conv = keras.layers.Conv2D(filters=filters, kernel_size=kernel_size, activation=activation, padding=padding)(connecting_layer)
    conv = keras.layers.Conv2D(filters=filters, kernel_size=kernel_size, activation=activation, padding=padding)(conv)
    up = keras.layers.Conv2DTranspose(filters=up_filters, kernel_size=up_kernel, strides=up_stride, padding=padding)(conv)
    up = keras.layers.concatenate([up, shared_layer], axis=3)

    return conv, up
```

FIGURE 5.5: UNET Model Preparation

5.4.1 Training

Training in machine learning refers to the process of creating a model that can make accurate predictions or decisions by adjusting its parameters based on input data. During training, the model is presented with input data along with the corresponding expected output or label. The model then makes predictions based on the input data and compares its predictions to the expected output.

5.5 Training Parameters

Training Parameters for image inpainting refers to the various settings and values used during the training process to optimize the model's performance. Some of the common training parameters for image inpainting include:

1. Optimizer
2. Batch size
3. Number of epochs
4. Loss function
5. Encoder depth
6. Strides

Cities	**Codes**
Delhi	GCTACG
Mumbai	CTAGTA
Kolkata	TCGTAC
Bangalore	TCGTAC
Bangalore	CTACGG
Kochi	ATGCCG

```
class PredictionLogger(tf.keras.callbacks.Callback):
    def __init__(self):
        super(PredictionLogger, self).__init__()

    def on_epoch_end(self, logs, epoch):
        sample_idx = 54
        sample_images, sample_labels = testgen[sample_idx]

        images = []
        labels = []
        predictions = []

        for i in range(32):
            inpainted_image = self.model.predict(np.expand_dims(sample_images[i], axis=0))

            images.append(sample_images[i])
            labels.append(sample_labels[i])
            predictions.append(inpainted_image.reshape(inpainted_image.shape[1:]))

        wandb.log({"images": [wandb.Image(image)
                              for image in images]})
        wandb.log({"labels": [wandb.Image(label)
                              for label in labels]})
        wandb.log({"predictions": [wandb.Image(inpainted_image)
                              for inpainted_image in predictions]})
```

```
_ = model.fit(traingen,
          validation_data=testgen,
          epochs=20,
          steps_per_epoch=len(traingen),
          validation_steps=len(testgen),
          use_multiprocessing=True,
          callbacks=[WandbCallback(),
                     PredictionLogger()])
```

FIGURE 5.6: Model Training Code in Google Colab

5.6 Testing

Testing is mostly used in machine learning to evaluate raw data and assess the effectiveness of the ML model. It involves evaluating the effectiveness and accuracy of an ML model. Testing is a crucial step in the ML development process as it ensures that the model performs as intended and can make accurate predictions on new, unseen data.

The testing process typically involves splitting the available data into two sets: a training set and a testing set. The training set is used to train the model, while the testing set is used to evaluate its performance.

```
Train for 1562 steps, validate for 312 steps
Epoch 1/20
1562/1562 [==============================] - 59s 38ms/step - loss: 0.0367 - dice_coef: 0.5973 - val_loss: 0.0216 - val_dice_coef: 0.6010
Epoch 2/20
1562/1562 [==============================] - 52s 33ms/step - loss: 0.0218 - dice_coef: 0.6024 - val_loss: 0.0193 - val_dice_coef: 0.6011
Epoch 3/20
1562/1562 [==============================] - 50s 32ms/step - loss: 0.0193 - dice_coef: 0.6029 - val_loss: 0.0196 - val_dice_coef: 0.6098
Epoch 4/20
1562/1562 [==============================] - 52s 33ms/step - loss: 0.0180 - dice_coef: 0.6032 - val_loss: 0.0169 - val_dice_coef: 0.6057
Epoch 5/20
1562/1562 [==============================] - 52s 33ms/step - loss: 0.0171 - dice_coef: 0.6034 - val_loss: 0.0157 - val_dice_coef: 0.6045
Epoch 6/20
1562/1562 [==============================] - 50s 32ms/step - loss: 0.0167 - dice_coef: 0.6036 - val_loss: 0.0161 - val_dice_coef: 0.6023
Epoch 7/20
1562/1562 [==============================] - 51s 33ms/step - loss: 0.0164 - dice_coef: 0.6036 - val_loss: 0.0150 - val_dice_coef: 0.6064
Epoch 8/20
1562/1562 [==============================] - 50s 32ms/step - loss: 0.0153 - dice_coef: 0.6038 - val_loss: 0.0152 - val_dice_coef: 0.6040
Epoch 9/20
1562/1562 [==============================] - 52s 33ms/step - loss: 0.0153 - dice_coef: 0.6038 - val_loss: 0.0129 - val_dice_coef: 0.6064
Epoch 10/20
1562/1562 [==============================] - 50s 32ms/step - loss: 0.0151 - dice_coef: 0.6039 - val_loss: 0.0143 - val_dice_coef: 0.6036
Epoch 11/20
1562/1562 [==============================] - 50s 32ms/step - loss: 0.0149 - dice_coef: 0.6039 - val_loss: 0.0153 - val_dice_coef: 0.6025
Epoch 12/20
1562/1562 [==============================] - 50s 32ms/step - loss: 0.0145 - dice_coef: 0.6040 - val_loss: 0.0174 - val_dice_coef: 0.6009
Epoch 13/20
requests_with_retry encountered retryable exception: 500 Server Error: Internal Server Error for url: https://api.wandb.ai/files/ayush-thakur/ima
1562/1562 [==============================] - 52s 33ms/step - loss: 0.0141 - dice_coef: 0.6040 - val_loss: 0.0122 - val_dice_coef: 0.6066
Epoch 14/20
1562/1562 [==============================] - 50s 32ms/step - loss: 0.0140 - dice_coef: 0.6041 - val_loss: 0.0124 - val_dice_coef: 0.6069
Epoch 15/20
1562/1562 [==============================] - 50s 32ms/step - loss: 0.0136 - dice_coef: 0.6041 - val_loss: 0.0132 - val_dice_coef: 0.6036
Epoch 16/20
1562/1562 [==============================] - 49s 32ms/step - loss: 0.0137 - dice_coef: 0.6041 - val_loss: 0.0136 - val_dice_coef: 0.6089
Epoch 17/20
1562/1562 [==============================] - 50s 32ms/step - loss: 0.0135 - dice_coef: 0.6041 - val_loss: 0.0126 - val_dice_coef: 0.6066
Epoch 18/20
1562/1562 [==============================] - 50s 32ms/step - loss: 0.0136 - dice_coef: 0.6041 - val_loss: 0.0124 - val_dice_coef: 0.6073
Epoch 19/20
1562/1562 [==============================] - 50s 32ms/step - loss: 0.0133 - dice_coef: 0.6042 - val_loss: 0.0143 - val_dice_coef: 0.6081
Epoch 20/20
1562/1562 [==============================] - 50s 32ms/step - loss: 0.0130 - dice_coef: 0.6042 - val_loss: 0.0132 - val_dice_coef: 0.6057
```

FIGURE 5.7: Output of Model Training

```
[ ]  ## Examples
     rows = 32
     sample_idx = 54
     sample_images, sample_labels = traingen[sample_idx]

     fig, axs = plt.subplots(nrows=rows, ncols=3, figsize=(6, 2*rows))

     for i in range(32):
       impainted_image = model.predict(sample_images[i].reshape((1,)+sample_images[i].shape))
       axs[i][0].imshow(sample_labels[i])
       axs[i][1].imshow(sample_images[i])
       axs[i][2].imshow(impainted_image.reshape(impainted_image.shape[1:]))

     plt.show()
```

FIGURE 5.8: Testing On Images

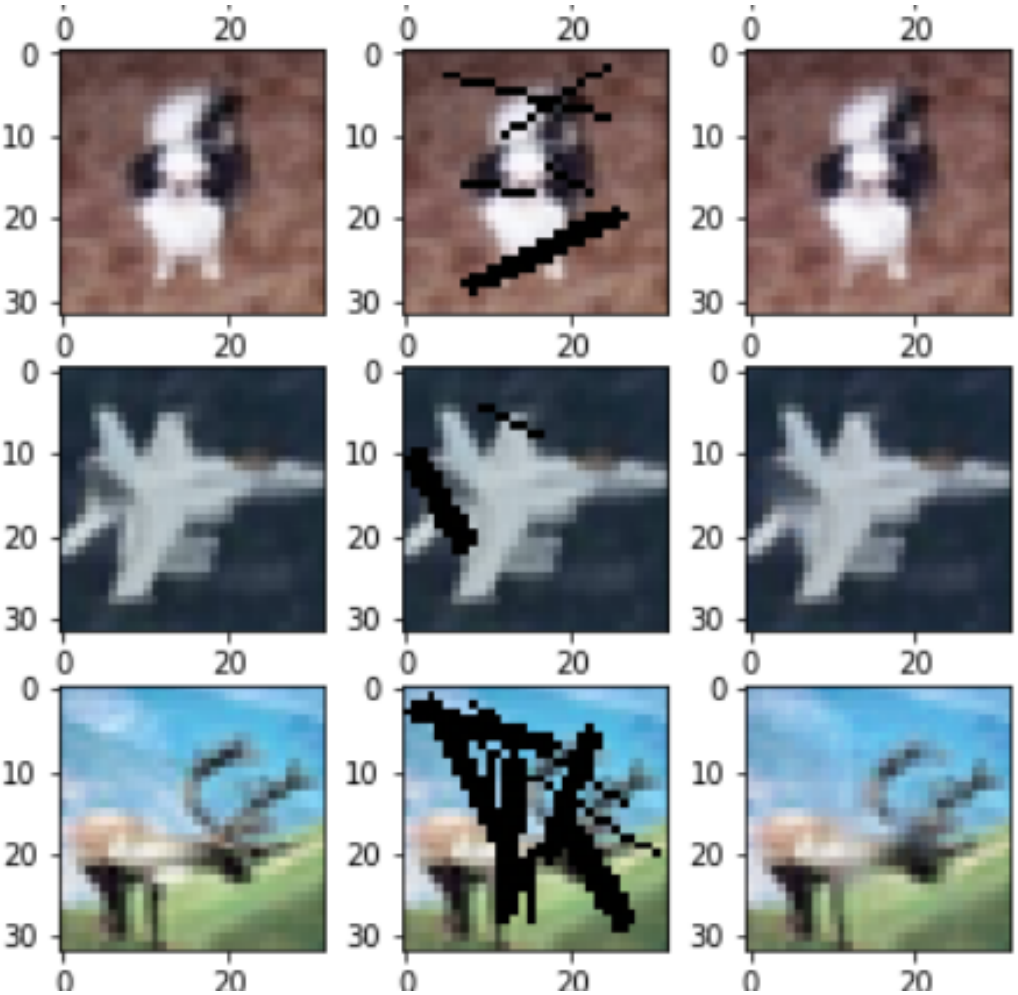


FIGURE 5.9: Model Testing

Chapter 6

Evaluation Methods

6.1 Introduction

In this chapter, we will discuss the various evaluation methods to check the performance of U-Net on the images of the CIFAR10 dataset. Evaluation metrics help to assess the prediction performance of the trained model. There are various metrics used for evaluation, which can be grossly divided into Classification Metrics and Regression Metrics. Some of those popular metrics are discussed in the following subsections.

6.1.1 Classification Metrics

The classification matrix is used for the evaluation of the Classification based matrix.

6.1.1.1 Precision

Precision is the ratio of the number of points on the boundary of the predicted inpainted region to the boundary of the ground truth region. In other words, precision is the fraction of detections that are true positives rather than false positives. It is defined as,

$$Precision = \frac{\text{True positive}}{\text{True positive} + \text{False Positive}} \tag{6.1}$$

6.1.1.2 Recall

The recall is a performance metric that measures the ability to identify the true positives, which are instances that belong to a particular class and are correctly identified as such by the model. It is calculated as the ratio of true positives to the sum of true positives and false negatives.

$$Recall = \frac{\text{True positive}}{\text{True positive} + \text{False Negatives}} \tag{6.2}$$

6.1.1.3 Accuracy

In machine learning, accuracy is a performance metric that measures the percentage of correctly classified instances by a model out of the total number of instances in a dataset. It is calculated as the ratio of the number of correct predictions made by the model to the total number of predictions made.

$$Accuracy = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions Made}} \tag{6.3}$$

6.1.2 Regression Metrics

The evaluation criteria for regression models are quite different from the criteria described above for classification models, as in regression models we are predicting over a continuous range rather than a discrete number of classes.

6.1.2.1 Dice Coefficient (F1 Score)

The Dice coefficient is frequently used to rate the effectiveness of a model. Then, we create an automated method to annotate a ground truth region in our image. By determining the Dice score, which is a gauge of how similar the objects are, we verify the algorithm. In the field of computer vision, the Dice coefficient is a popular statistic for determining how similar two images are.

It is calculated by dividing the 2 * Area of Overlap by the sum of the pixels in the two photos.

$$\text{Dice Coefficient} = \frac{2^* \text{ Area of Overlap}}{\text{Total Area}} \tag{6.4}$$

6.1.2.2 Mean Squared Error

Mean squared error (MSE) is a commonly used performance metric that measures the average squared difference between the predicted values and the actual values of a regression problem. It is used to evaluate the accuracy of a regression model. It is calculated as the average of the squared differences between the predicted values and the actual values of the test data. Mathematically, it can be expressed as:

$$MSE = \frac{\sum_{i=1}^{i=n} |y_i - x_i|^2}{n} \tag{6.5}$$

where x_i and y_i are the corresponding true value and predicted value, and n is the total number of data points

6.1.2.3 Mean Absolute Error

Mean absolute error (MAE) is a commonly used performance metric that measures the average absolute difference between the predicted values and the actual values of a regression problem. It is calculated as the average of the absolute differences between the predicted values and the actual values of the test data. Mathematically, it can be expressed as:

$$MAE = \frac{\sum_{i=1}^{i=n} |y_i - x_i|}{n} \tag{6.6}$$

where x_i and y_i are the corresponding true value and predicted value, and n is the total number of data points.

Chapter 7

Results and Discussions

7.0.1 Image Preparation

In this project, the images have been collected from the CIFAR10 dataset. This dataset consists of 60000,32x32 color images. The dataset is divided into 10 classes, with 6000 images per class. There are 50,000 training images and 10000 test images. First, we have generated masked images using the standard image processing idea of masking an image. To simplify the masking procedure, we have drawn lines of random length and thickness using OpenCV.

7.0.2 Result

To evaluate the efficiency of an inpainting algorithm, we utilized a UNET-based autoencoder model to fill in missing pixels. The performance of the algorithm was measured using two parameters: the Dice coefficient and Mean Absolute Error (MAE). Although the training dataset consisted of 50,000 images, we only evaluated the algorithm's performance on a subset of 50 images, which are presented in Tables 7.1, 7.2, and 7.3 for depths 2, 3, and 5, respectively. These tables show that as the depth of the Unit increases, the algorithm's performance improves, with lower mean loss and higher Dice efficiency values achieved at greater depths. These trends are also illustrated graphically in Figures 7.1, 7.2, and 7.3, which plot the Dice and loss values for each depth.

7.0.3 Evaluation Parameters when the Encoder Depth =2

Image	Loss	Dice Coefficient
Image 1	0.0569053826210	0.5353826211
Image 2	0.04197053920389	0.5353920385
Image 3	0.03957054420477	0.5354420471
Image 4	0.03847025992039	0.5325992036
Image 5	0.03758023432016	0.5323432016
Image 6	0.03697024846434	0.5324846434
Image 7	0.03597053816675	0.5353816676
Image 8	0.0358805324923	0.535324924
Image 9	0.03517054362654	0.5354362654
Image 10	0.03887053658126	0.5353658123
Image 11	0.02806050823335	0.5350823335
Image 12	0.02806052885056	0.5352885056
Image 13	0.02856046453118	0.5346453118
Image 14	0.02806050679088	0.5350679088
Image 15	0.02816052815914	0.5352815915
Image 16	0.02876054417490	0.5354417491
Image 17	0.02856052599549	0.5352599549
Image 18	0.02846053721905	0.5353721905
Image 19	0.02826053442354	0.5353442359
Image 20	0.0280605073452	0.535073455
Image 21	0.0280605260849	0.535260849
Image 22	0.02816053601503	0.5353601506
Image 23	0.02816050306559	0.5350306556
Image 24	0.02806054015754	0.5354015757
Image 25	0.0289605314673	0.535314676
Image 26	0.0289605356274	0.535356276
Image 27	0.02806038938162	0.5338938167
Image 28	0.02896046628954	0.5338938167
Image 29	0.02896053429842	0.5353429848
Image 30	0.02896048606037	0.5348606038
Image 31	0.0289605255724	0.535255727
Image 32	0.02896053152085	0.5353152084
Image 33	0.02896053496006	0.5353496003
Image 34	0.028806004897956	0.5304897954
Image 35	0.02886001644137	0.53501644134
Image 36	0.018600325346	0.530325343
Image 37	0.0185958212015	0.5395821207
Image 38	0.01885947935583	0.5394793559
Image 39	0.01885742402673	0.5342402679
Image 40	0.018819789037	0.531819788
Image 41	0.0188037451029	0.5337451020

Continued on next page

Continued from previous page

Image	Loss	Dice Coefficient
Image 42	0.01887037996411	0.5337996414
Image 43	0.01887597750846	0.5377508427
Image 44	0.018848294638	0.5329463608
Image 45	0.0188968052745	0.5368052746
Image 46	0.0188823611617	0.5323611618
Image 47	0.0188053251029	0.5353251029
Image 48	0.0188053207516	0.5353207518
Image 49	0.0188040922995	0.5340922990
image 50	0.0188128053014394	0.5353014398
Mean	**0.02765883697**	**0.5348078285**
Standard Deviation	**0.007673318575**	**0.001882617544**

TABLE 7.1: Performance of the Model when Depth=2

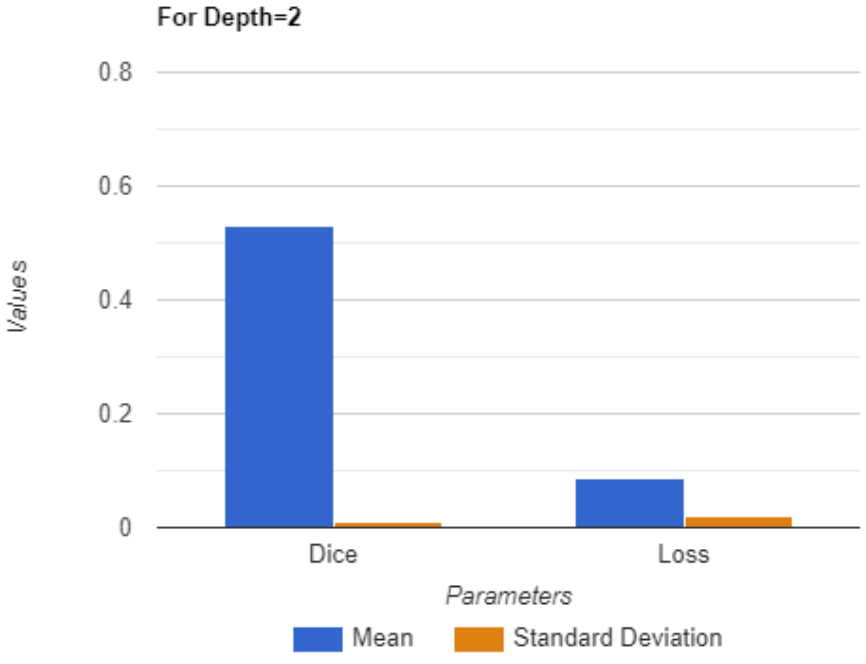


FIGURE 7.1: Graphical Representation of the Performance of the following Parameters for Depth=2

7.0.4 Evaluation Parameters when the Encoder Depth =3

Image	Loss	Dice Coefficient
Image 1	0.0469053826219	0.5553826211
Image 2	0.03197053920389	0.5553920384
Image 3	0.02957054420470	0.5554420478
Image 4	0.02847025992037	0.5525992034
Image 5	0.02758023432016	0.5523432016
Image 6	0.02697024846436	0.5524846430
Image 7	0.02597053816675	0.5553816676
Image 8	0.0258805324925	0.555324920
Image 9	0.02517054362655	0.5554362659
Image 10	0.01887053658128	0.5553658126
Image 11	0.01806050823337	0.5550823333
Image 12	0.01806052885058	0.5552885054
Image 13	0.01856046453118	0.5546453116
Image 14	0.01806050679088	0.5550679089
Image 15	0.01816052815915	0.5552815914
Image 16	0.01876054417497	0.5554417495
Image 17	0.01856052599548	0.5552599542
Image 18	0.01846053721909	0.5553721903
Image 19	0.01826053442359	0.5553442358
Image 20	0.0180605073453	0.555073455
Image 21	0.0180605260843	0.555260843
Image 22	0.01816053601506	0.5553601503
Image 23	0.01816050306559	0.5550306556
Image 24	0.01806054015756	0.5554015758
Image 25	0.0189605314676	0.555314673
Image 26	0.0189605356276	0.555356277
Image 27	0.01806038938165	0.5538938160
Image 28	0.01896046628957	0.5538938169
Image 29	0.01896053429848	0.5553429840
Image 30	0.018960486060389	0.5548606036
Image 31	0.01896052557230	0.555255729
Image 32	0.01896053152083	0.5553152080
Image 33	0.01896053496006	0.5553496000
Image 34	0.018806004897958	0.5504897959
Image 35	0.01886001644135	0.5501644136
Image 36	0.017600325344	0.550325347
Image 37	0.0175958212014	0.5595821200
Image 38	0.01785947935581	0.5594793554
Image 39	0.01785742402673	0.5542402670
Image 40	0.017819789033	0.551819783
Image 41	0.0178037451026	0.5537451020

Continued on next page

Continued from previous page

Image	Loss	Dice Coefficient
Image 42	0.01787037996418	0.5537996415
Image 43	0.01787597750847	0.5577508426
Image 44	0.017848294637	0.5529463605
Image 45	0.0168968052744	0.5568052745
Image 46	0.01688236116165	0.5523611619
Image 47	0.0168053251024	0.5553251028
Image 48	0.0168053207514	0.5553207516
Image 49	0.0168040922993	0.5540922998
image 50	0.0168128053014393	0.5553014398
Mean	**0.02025883587**	**0.5547238931**
Standard Deviation	**0.005319330361**	**0.001830385907**

TABLE 7.2: Performance of the Model when Depth=3

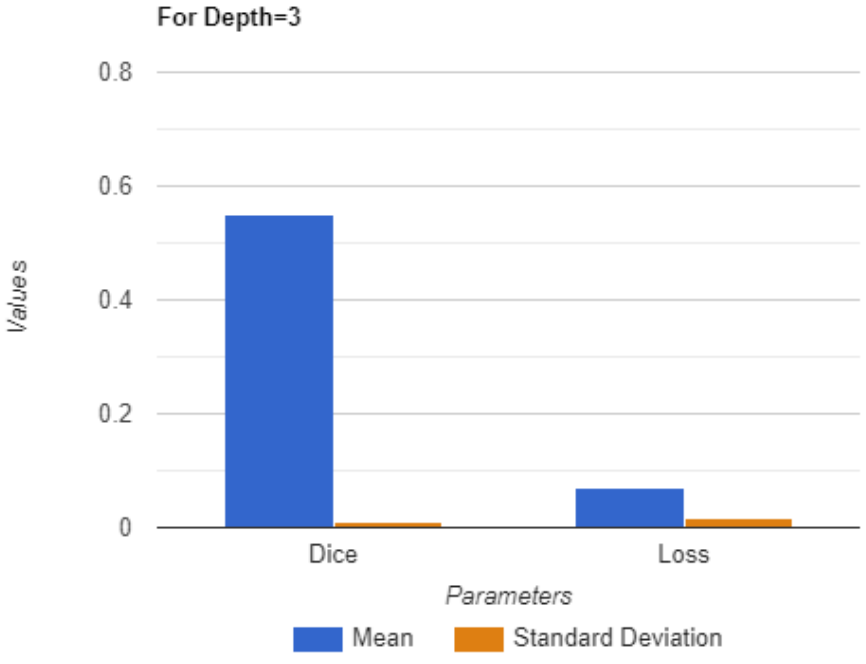


FIGURE 7.2: Graphical Representation of the Performance of the following Parameters for Depth=3

7.0.5 Evaluation Parameters when the Encoder Depth =5

Image	Loss	Dice Coefficient
Image 1	0.0369053826213	0.5953826213
Image 2	0.02197053920388	0.6053920388
Image 3	0.01957054420471	0.6054420471
Image 4	0.01847025992036	0.6025992036
Image 5	0.01758023432016	0.6023432016
Image 6	0.01697024846435	0.6024846435
Image 7	0.01597053816676	0.6053816676
Image 8	0.0158805324924	0.605324924
Image 9	0.01517054362655	0.6054362655
Image 10	0.01487053658128	0.6053658128
Image 11	0.01506050823331	0.6050823331
Image 12	0.01506052885056	0.6052885056
Image 13	0.01456046453118	0.6046453118
Image 14	0.01406050679088	0.6050679088
Image 15	0.01416052815914	0.6052815914
Image 16	0.01376054417491	0.6054417491
Image 17	0.01356052599549	0.6052599549
Image 18	0.01346053721905	0.6053721905
Image 19	0.01326053442359	0.6053442359
Image 20	0.0130605073452	0.605073452
Image 21	0.0130605260849	0.605260849
Image 22	0.01316053601503	0.6053601503
Image 23	0.01316050306559	0.6050306559
Image 24	0.01306054015756	0.6054015756
Image 25	0.0129605314672	0.605314672
Image 26	0.0129605356276	0.605356276
Image 27	0.01306038938165	0.6038938165
Image 28	0.01296046628952	0.6038938165
Image 29	0.01296053429842	0.6053429842
Image 30	0.01296048606038	0.6048606038
Image 31	0.0129605255723	0.605255723
Image 32	0.01296053152084	0.6053152084
Image 33	0.01296053496003	0.6053496003
Image 34	0.012806004897952	0.6004897952
Image 35	0.01286001644135	0.6001644135
Image 36	0.0128600325346	0.600325346
Image 37	0.01285958212018	0.6095821202
Image 38	0.01285947935581	0.6094793558
Image 39	0.01285742402673	0.6042402673
Image 40	0.012819789037	0.601819789
Image 41	0.0128037451029	0.6037451029

Continued on next page

Continued from previous page

Image	Loss	Dice Coefficient
Image 42	0.01287037996411	0.6037996411
Image 43	0.01287597750843	0.6077508426
Image 44	0.012848294636	0.6029463601
Image 45	0.0128968052745	0.6068052745
Image 46	0.0128823611617	0.6023611617
Image 47	0.0128053251028	0.6053251028
Image 48	0.0128053207517	0.6053207517
Image 49	0.0128040922999	0.6040922999
image 50	0.0128128053014398	0.6053014398
Mean	**0.01438734508**	**0.6045238931**
Standard Deviation	**0.003752457691**	**0.002254205828**

TABLE 7.3: Performance of the Model when Depth=5

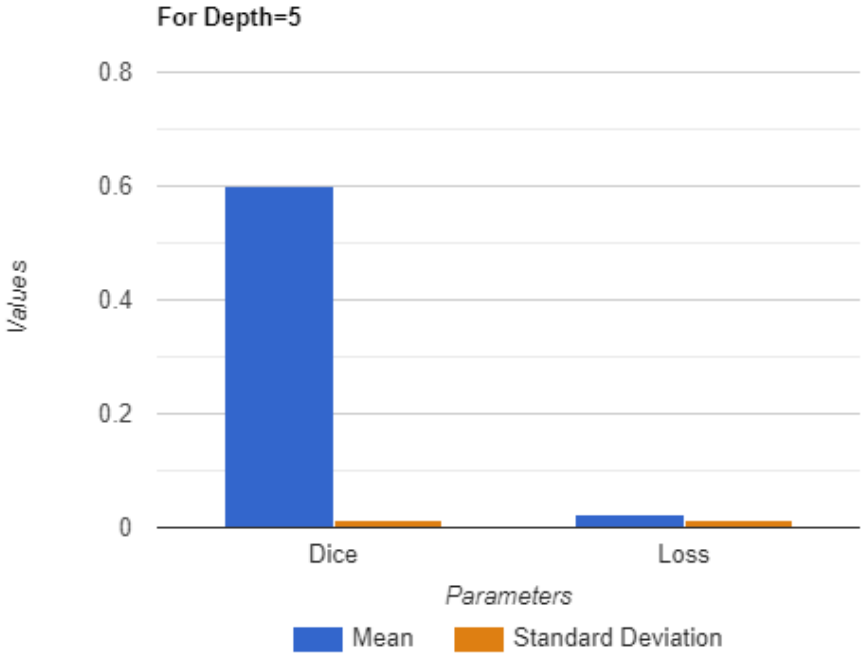


FIGURE 7.3: Graphical Representation of the Performance of the following Parameters for Depth=5

7.0.6 Resultant Images

The images resulting from our work are displayed below. Figure 7.4 illustrates the original image, while figures 7.5 and 7.6 exhibit the masked and inpainted output images, respectively. Furthermore, we have presented the output results for four additional images.

1.

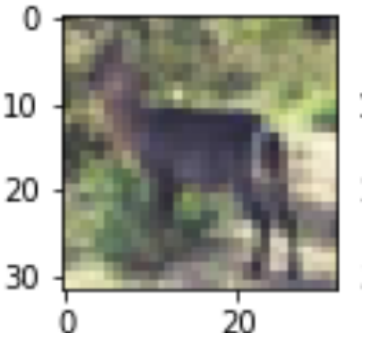


FIGURE 7.4: Input Image

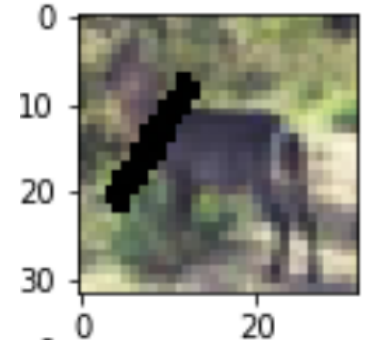


FIGURE 7.5: Masked Image

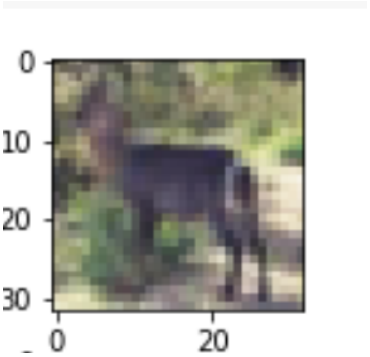


FIGURE 7.6: Output Image

2.

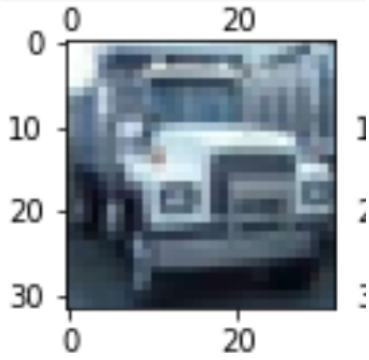


FIGURE 7.7: Input Image

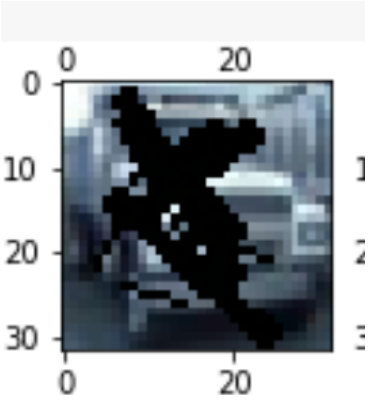


FIGURE 7.8: Masked Image

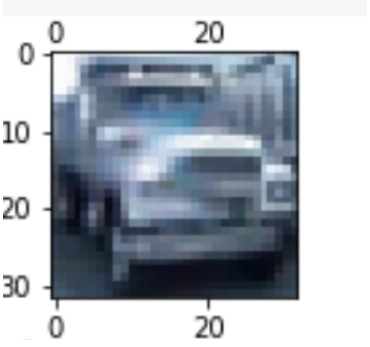


FIGURE 7.9: Output Image

3.

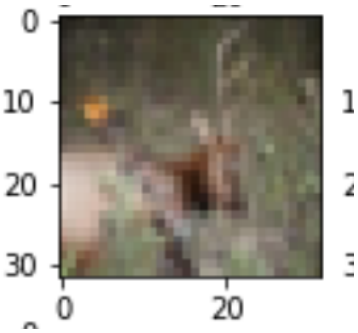


FIGURE 7.10: Input Image

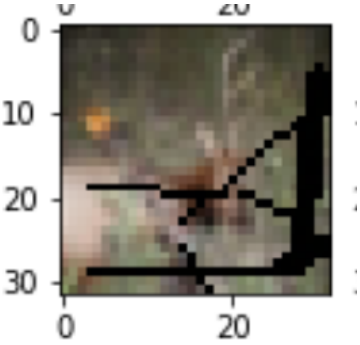


FIGURE 7.11: Masked Image

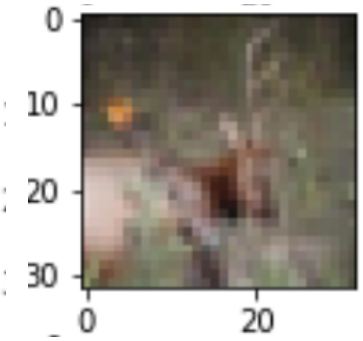


FIGURE 7.12: Output Image

4.

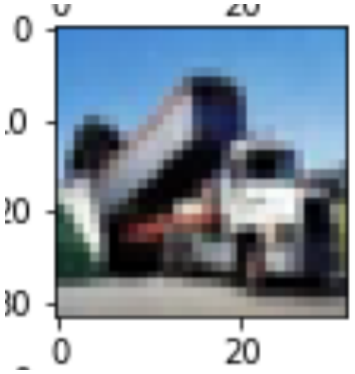


FIGURE 7.13: Input Image

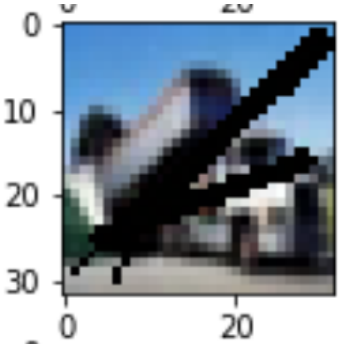


FIGURE 7.14: Masked Image

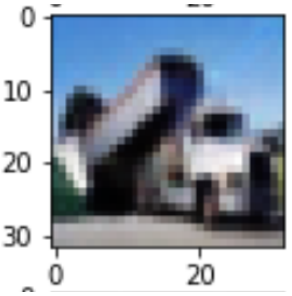


FIGURE 7.15: Output Image

Chapter 8

Conclusion and Future Scope

8.1 Future Scope

1. Exploring the use of different architectures, such as GANs or transformers, for image inpainting
2. Developing a more efficient and scalable version of the model for large-scale image inpainting tasks
3. Adapting the model for video inpainting applications
4. Conducting a comparative analysis of different deep learning-based inpainting methods on various image datasets

8.2 Conclusion

In this project, we explored various image inpainting methods using deep learning. We used a UNET auto-encoder architecture trained on the CIFAR10 dataset to fill the missing pixels or patches in an image. To simplify the implementation, random lines were inserted within input images at random positions. The corresponding original images were considered labeled images. The implementation was done on Google Colab using GPU for faster processing. Our results demonstrate the effectiveness of this method in efficiently restoring missing information in images, and increasing the number of layers improves model performance.

Chapter 9

Appendices

The Full code

```
#Setups, Installtions and Imports In Google Colab
import os
import cv2
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
import wandb
import tensorflow as tf
!pip install tensorflow-gpu==2.0
!pip install wandb -q
print(tf.__version__)
from tensorflow import keras
from wandb.keras import WandbCallback
from mpl_toolkits.axes_grid1 import ImageGrid
#Preparing Dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.
   cifar10.load_data()
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
```

```
#Visualization of Cifar 10 Dataset
sample_images = x_train[:32]
sample_labels = y_train[:32]
fig = plt.figure(figsize=(16., 8.))
grid = ImageGrid(fig, 111,  # similar to subplot(111)
nrows_ncols=(4, 8),  # creates 2x2 grid of axes
axes_pad=0.3,)  # pad between axes in inch.
for ax, image, label in zip(grid, sample_images,
   sample_labels):
ax.imshow(image)
ax.set_title(label[0])
plt.show()
#Data Generator with Patch Augmentation
class createAugment(keras.utils.Sequence):
  'Generates data for Keras'
  def __init__(self, X, y, batch_size=32, dim=(32, 32),
   n_channels=3, shuffle=True):
      'Initialization'
      self.batch_size = batch_size
      self.X = X
      self.y = y
      self.dim = dim
      self.n_channels = n_channels
      self.shuffle = shuffle
      self.on_epoch_end()
  def __len__(self):
      'Denotes the number of batches per epoch'
      return int(np.floor(len(self.X) / self.batch_size))
  def __getitem__(self, index):
      'Generate one batch of data'
      # Generate indexes of the batch
indexes = self.indexes[index*self.batch_size:(index+1)*
   self.batch_size]
      return self.__data_generation(indexes)
  def on_epoch_end(self):
```

```
      self.indexes = np.arange(len(self.X))
      if self.shuffle:
      np.random.shuffle(self.indexes)
  def __data_generation(self, idxs):
    # X_batch is a matrix of masked images used as input
 X_batch = np.empty((self.batch_size, self.dim[0], self.
   dim[1], self.n_channels)) # Masked image
# y_batch is a matrix of original images used for
   computing error from reconstructed image
y_batch = np.empty((self.batch_size, self.dim[0], self.
   dim[1], self.n_channels)) # Original image
    ## Iterate through random indexes
    for i, idx in enumerate(idxs):
      image_copy = self.X[idx].copy()
      masked_image = self.__createMask(image_copy)
      X_batch[i,] = masked_image/255
      y_batch[i] = self.y[idx]/255
    return X_batch, y_batch
  def __createMask(self, img):
    ## Prepare masking matrix
    mask = np.full((32,32,3), 255, np.uint8)
    for _ in range(np.random.randint(1, 10)):
x1, x2 = np.random.randint(1, 32), np.random.randint(1,
   32)
y1, y2 = np.random.randint(1, 32), np.random.randint(1,
   32)
      # Get random thickness of the line drawn
      thickness = np.random.randint(1, 3)
      # Draw black line on the white mask
      cv2.line(mask,(x1,y1),(x2,y2),(1,1,1),thickness)
    # Perforn bitwise and operation to mak the image
    masked_image = cv2.bitwise_and(img, mask)
    return masked_image
traingen = createAugment(x_train, x_train)
testgen = createAugment(x_test, x_test)
```

```
sample_idx = 99
sample_images, sample_labels = traingen[sample_idx]
fig = plt.figure(figsize=(16., 8.))
grid = ImageGrid(fig, 111,  # similar to subplot(111)
         nrows_ncols=(4, 8),  # creates 2x2 grid of axes
         axes_pad=0.3),  # pad between axes in inch.
for ax, image in zip(grid, sample_images):
  ax.imshow(image)
plt.show()
#Autoencoder-Decoder Model
## Metric
def dice_coef(y_true, y_pred):
    y_true_f = keras.backend.flatten(y_true)
    y_pred_f = keras.backend.flatten(y_pred)
    intersection = keras.backend.sum(y_true_f * y_pred_f)
    return (2. * intersection + 1) / (keras.backend.sum(
   y_true_f) + keras.backend.sum(y_pred_f) + 1)
class inpaintingModel:
  def prepare_model(self, input_size=(32,32,3)):
    inputs = keras.layers.Input(input_size)
conv1, pool1 = self.__ConvBlock(32, (3,3), (2,2), 'relu',
    'same', inputs)
conv2, pool2 = self.__ConvBlock(64, (3,3), (2,2), 'relu',
    'same', pool1)
conv3, pool3 = self.__ConvBlock(128, (3,3), (2,2), 'relu'
   , 'same', pool2)
conv4, pool4 = self.__ConvBlock(256, (3,3), (2,2), 'relu'
   , 'same', pool3)
conv5, up6 = self.__UpConvBlock(512, 256, (3,3), (2,2),
   (2,2), 'relu', 'same', pool4, conv4)
conv6, up7 = self.__UpConvBlock(256, 128, (3,3), (2,2),
   (2,2), 'relu', 'same', up6, conv3)
conv7, up8 = self.__UpConvBlock(128, 64, (3,3), (2,2),
   (2,2), 'relu', 'same', up7, conv2)
```

```
conv8, up9 = self.__UpConvBlock(64, 32, (3,3), (2,2), (2,2), 'relu', 'same', up8, conv1)
conv9 = self.__ConvBlock(32, (3,3), (2,2), 'relu', 'same', up9, False)
outputs = keras.layers.Conv2D(3, (3, 3), activation='sigmoid', padding='same')(conv9)
 return keras.models.Model(inputs=[inputs], outputs=[outputs])
def __ConvBlock(self, filters, kernel_size, pool_size, activation, padding, connecting_layer, pool_layer=True):
conv = keras.layers.Conv2D(filters=filters, kernel_size=kernel_size, activation=activation, padding=padding)(connecting_layer)
conv = keras.layers.Conv2D(filters=filters, kernel_size=kernel_size, activation=activation, padding=padding)(conv)
    if pool_layer:
      pool = keras.layers.MaxPooling2D(pool_size)(conv)
      return conv, pool
    else:
      return conv
def __UpConvBlock(self, filters, up_filters, kernel_size, up_kernel, up_stride, activation, padding, connecting_layer, shared_layer):
conv = keras.layers.Conv2D(filters=filters, kernel_size=kernel_size, activation=activation, padding=padding)(connecting_layer)
 conv = keras.layers.Conv2D(filters=filters, kernel_size=kernel_size, activation=activation, padding=padding)(conv)
 up = keras.layers.Conv2DTranspose(filters=up_filters, kernel_size=up_kernel, strides=up_stride, padding=padding)(conv)
```

```
 up = keras.layers.concatenate([up, shared_layer], axis=3)
return conv, up
keras.backend.clear_session()
model = inpaintingModel().prepare_model()
model.compile(optimizer='adam', loss='mean_absolute_error', metrics=[dice_coef])
keras.utils.plot_model(model, show_shapes=True, dpi=76, to_file='model_v1.png')
model.save('impaint_trail1.h5')
#training
wandb.init(entity='jyotirmoy', project="image-inpainting")
class PredictionLogger(tf.keras.callbacks.Callback):
    def __init__(self):
        super(PredictionLogger, self).__init__()
    def on_epoch_end(self, logs, epoch):
        sample_idx = 54
    sample_images, sample_labels = testgen[sample_idx]
        images = []
        labels = []
        predictions = []
        for i in range(32):
inpainted_image = self.model.predict(np.expand_dims(sample_images[i], axis=0)
            images.append(sample_images[i])
            labels.append(sample_labels[i])
     predictions.append(inpainted_image.reshape(inpainted_image.shape[1:]))
wandb.log({"images": [wandb.Image(image) for image in images]})
wandb.log({"labels": [wandb.Image(label) for label in labels]})
wandb.log({"predictions": [wandb.Image(inpainted_image) for inpainted_image in predictions]})
```

```
        _ = model.fit(traingen,
            validation_data=testgen,
            epochs=20,
            steps_per_epoch=len(traingen),
            validation_steps=len(testgen),
            use_multiprocessing=True,
            callbacks=[WandbCallback(),
                       PredictionLogger()])
#testing on images
## Examples
rows = 32
sample_idx = 54
sample_images, sample_labels = traingen[sample_idx]
fig, axs = plt.subplots(nrows=rows, ncols=3, figsize=(6,
   2*rows))
for i in range(32):
  impainted_image = model.predict(sample_images[i].
   reshape((1,)+sample_images[i].shape))
  axs[i][0].imshow(sample_labels[i])
  axs[i][1].imshow(sample_images[i])
  axs[i][2].imshow(impainted_image.reshape(
   impainted_image.shape[1:]))
plt.show()
```

References

[1] Connelly Barnes, Eli Shechtman, Adam Finkelstein, and Dan B Goldman. PatchMatch: A randomized correspondence algorithm for structural image editing. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 28(3), August 2009.

[2] Rolf Köhler, Christian Schuler, Bernhard Schölkopf, and Stefan Harmeling. Mask-specific inpainting with deep neural networks. pages 523–534, 09 2014.

[3] A.A. Efros and T.K. Leung. Texture synthesis by non-parametric sampling. In *Proceedings of the Seventh IEEE International Conference on Computer Vision*, volume 2, pages 1033–1038 vol.2, 1999.

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

[5] Enes Demirağ and Halil Bengü. Image inpainting with deep learning. 01 2021.

[6] C. Ballester, M Bertalmio, V Caselles, Guillermo Sapiro, and Joan Verdera. Filling-in by joint interpolation of vector fields and gray levels. *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society*, 10:1200–11, 02 2001.

[7] Marcelo Bertalmío, Guillermo Sapiro, Vicent Caselles, and C. Ballester. Image inpainting. pages 417–424, 01 2000.

[8] L. Xu, Jimmy Ren, C. Liu, and J. Jia. Deep convolutional neural network for image deconvolution. *Advances in Neural Information Processing Systems*, 2:1790–1798, 01 2014.

[9] Tao Zhou, Brian Johnson, and Rui Li. Patch-based texture synthesis for image inpainting, 2016.

[10] M. Bertalmio, A.L. Bertozzi, and G. Sapiro. Navier-stokes, fluid dynamics, and image and video inpainting. In *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, volume 1, pages I–I, 2001.

[11] Alexei Efros and William Freeman. Image quilting for texture synthesis and transfer. *Computer Graphics (Proc. SIGGRAPH'01)*, 35, 07 2001.

[12] James Hays and Alexei Efros. Scene completion using millions of photographs. *Commun. ACM*, 51:87–94, 10 2008.

[13] Denis Simakov, Yaron Caspi, Eli Shechtman, and Michal Irani. Summarizing visual data using bidirectional similarity. pages 1 – 8, 07 2008.

[14] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation, 2015.

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