A

PROJECT REPORT

ON

Retinal Images Synthesis Using Generative Adversarial Network

SUBMITTED TO

SHIVAJI UNIVERSITY, KOLHAPUR

IN THE PARTIAL FULFILLMENT OF REQUIREMENT FOR THE AWARD OF DEGREE BACHELOR OF ENGINEERING IN COMPUTER SCIENCE AND ENGINEERING

SUBMITTED BY

MR. Gulshan Vikas Ahuja	19UCS002
MR. Shrishail Kiran Bhagat	19UCS011
MR. Hemant Sudarshan Danawade	19UCS026
MR. Junedahmad Zakirhusain Chougale	19UCS023
MR. Shubhankar Suhas Kale	19UCS052

UNDER THE GUIDANCE OF

Prof. U. A. Nuli

DKTE

Promoting Excellence in Teaching, Learning & Research

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

DKTE SOCIETY'S TEXTILE AND ENGINEERING INSTITUTE, ICHALKARANJI

2022-2023

D.K.T.E. SOCIETY'S

TEXTILE AND ENGINEERING INSTITUTE, ICHALKARANJI
(AN AUTONOUMOUS INSTITUTE)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

DKTE
Promoting Excellence in
Teaching, Learning & Research

CERTIFICATE

This is to certify that, project work entitled

Retinal Images Synthesis Using Generative Adversarial Network

is a bonafide record of project work carried out in this college by

MR. Gulshan Vikas Ahuja	**19UCS002**
MR. Shrishail Kiran Bhagat	**19UCS011**
MR. Hemant Sudarshan Danawade	**19UCS026**
MR. Junedahmad Zakirhusain Chougale	**19UCS023**
MR. Shubhankar Suhas Kale	**19UCS052**

is in the partial fulfillment of award of degree Bachelor in Engineering in Computer Science & Engineering prescribed by Shivaji University, Kolhapur for the academic year 2022-2023.

Prof. U. A. Nuli

PROF.(DR.) D.V. KODAVADE
(HOD CSE DEPT.)

PROF. L. S.ADMUTHE
(I/C DIRECTOR)

EXAMINER: __________________

____

DECLARATION

We hereby declare that, the project work report entitled Retinal Image Synthesis Using Generative Adversarial Network which is being submitted to D.K.T.E. Society's Textile and Engineering Institute Ichalkaranji, affiliated to Shivaji University,Kolhapur is in partial fulfillment of degree B.Tech.(CSE). It is a bonafide report of the work carried out by us. The material contained in this report has not been submitted to any university or institution for the award of any degree. Further, we declare that we have not violated any of the provisions under Copyright and Piracy / Cyber / IPR Act amended from time to time.

MR.	Gulshan Vikas Ahuja	19UCS002
MR.	Shrishail Kiran Bhagat	19UCS011
MR.	Hemant Sudarshan Danawade	19UCS026
MR.	Junedahmad Zakirhusain Chougale	19UCS023
MR.	Shubhankar Suhas Kale	19UCS052

ACKNOWLEDGEMENT

With great pleasure we wish to express our deep sense of gratitude to Prof. U. A. Nuli for his valuable guidance, support and encouragement in completion of this project report.

Also, we would like to take opportunity to thank our head of department Dr. D. V. Kodavade for his co-operation in preparing this project report.

We feel gratified to record our cordial thanks to other staff members of Computer Science and Engineering Department for their support, help and assistance which they extended as and when required.

Thank you,

MR.	Gulshan Vikas Ahuja	19UCS002
MR.	Shrishail Kiran Bhagat	19UCS011
MR.	Hemant Sudarshan Danawade	19UCS026
MR.	Junedahmad Zakirhusain Chougale	19UCS023
MR.	Shubhankar Suhas Kale	19UCS052

ABSTRACT

Retinal image synthesis using Generative Adversarial Networks (GANs) is a promising technique for generating synthetic images that can be used to augment limited data sets for medical diagnosis, training of deep learning models, and image analysis. The GAN architecture comprises a generator network that generates synthetic images from random noise and a discriminator network that discriminates between real and synthetic images. The generator learns to produce images that are similar to the real images by fooling the discriminator. In retinal image synthesis, GANs have shown great potential in generating realistic images of the retina, optic nerve, and blood vessels. The generated images have been used to augment limited data sets, thereby improving the accuracy of deep learning models for medical diagnosis. Presenting an overview of retinal image synthesis using GANs, while discussing the challenges in generating high-quality synthetic images and the various techniques used to address them, such as the use of loss functions and data augmentation. It also describes the different applications of retinal image synthesis using GANs, including improving the accuracy of deep learning models for medical diagnosis, generating high-quality images for training purposes, and aiding in image analysis.

Overall, retinal image synthesis using GANs has shown great promise in the field of medical imaging and has the potential to revolutionize the way we diagnose and treat retinal diseases. With further advancements in GAN technology and more comprehensive datasets, we can expect to see even more impressive results in the future.

INDEX

1. INTRODUCTION

The retina, which lines the back of the eyeball and is located at the eye's interior surface, is a key component of vision. Several layers of cells, including photoreceptor cells, bipolar cells, ganglion cells, and different interneurons, make up the retina. For the purpose of sensing light, photoreceptor cells with the names rods and cones are used. The retina's main job is to transform light into electrical impulses that may be sent to the brain. In order to get to the retina after entering the eye, light must first pass through the cornea, lens, and other components. The retina's photoreceptor cells recognise light and transform it into electrical impulses. At the centre of the retina is a small area called the optic disc, where the optic nerve exits the eye. The retina's electrical signals are transported via the optic nerve to the brain for visual processing. Macular degeneration, diabetic retinopathy, retinal detachment, and retinitis pigmentosa are just a few of the conditions and diseases that can have an impact on the retina. Medical treatment is necessary for these diseases since they can cause eyesight loss.

The study of medical image analysis now includes retinal image synthesis as a key area of study. The creation of synthetic retinal images has received a lot of attention as a result of developments in deep learning methods, particularly Generative Adversarial Networks (GANs). These artificial images have enormous potential for a variety of tasks, such as deep learning model training, extending small datasets, and enabling the creation and testing of novel algorithms for retinal image interpretation. These artificial images have enormous promise for a range of uses, including deep learning model training, enhancing datasets, and simplifying the creation and testing of novel algorithms for retinal image interpretation.

Several ophthalmic illnesses, including age-related macular degeneration, glaucoma, and diabetic retinopathy, can only be diagnosed, and monitored with the help of retinal imaging. However, due to privacy issues, expensive prices, and a dearth of ground truth annotations, acquiring large-scale, varied, and labelled retinal image datasets can be difficult. Deep learning models, which largely rely on enormous volumes of labelled data, are sometimes hindered by this data shortage.

One of the most promising and popular ML approaches for radiology and disease identification in general is deep learning. Naturally, picture recognition is a good fit for deep learning algorithms since diagnostic imaging dominates clinical diagnosis. The continuing advancement of computing power and storage technology, the dropping cost of hardware, the labor crisis in the healthcare industry, and the amount of medical data to train models in certain fields in this field are some of the factors driving the usage of DL in medical diagnosis. However, in order to develop a highly accurate model for diagnosis in the case of retinal image analysis, there must be sufficient data.

For a variety of ophthalmic and neurological conditions, retinal imaging is a crucial diagnostic tool. However, assembling annotated retinal images of high quality can be difficult and time-consuming. There are also time restrictions and a finite number of experts. Therefore, it has become more crucial in recent years to develop strategies for retinal image synthesis.

Generative Adversarial Networks (GAN) utilization is one potential method for retinal picture creation. GANs are deep learning models that use training data from two networks—a generator network and a discriminator network—to create synthetic images that closely resemble genuine ones.

The goal of this project is to produce the datasets needed for retinal image diagnosis. These datasets don't actually contain any images; instead, the GAN (Generative Adversarial Network) artificially generates them. Deep learning-based generative models called Generative Adversarial Networks, or GANs, are used. The concepts found in the data set are trained into a generator. In order to distinguish between real images and fraudulent ones, a discriminator is set up to recognize the real ones. It then rejects the fake ideas and sends the real ones back to the generator to retrain it to produce realistic images.

Why Retinal Image Synthesis using GAN?

Building models that are good for analysis is facilitated by deep learning techniques. The potential for them to further medical and ophthalmic applications is therefore enormous. To train and validate DL models for use, however, there is a requirement for a wide variety and a significant amount of data. In order to obtain the necessary dataset, a method is therefore required; it can either be obtained from hospitals or generated intentionally. As it cannot be taken from hospitals due to issues like scarcity and privacy; so, the only option left is to create it artificially.

In this situation, GANs (Generative Adversarial Networks) are useful for the synthesis of images. GAN creates new data with the same properties as the training data after learning from a batch of training data. By doing this, it will be possible to create realistic retinal images for use.

1.1 Objectives:

1. To design a model which is capable of synthesizing new vessel networks and corresponding eye fundus image using GAN using relatively small sample size.
2. To design a model which can generate distinct images while preserving its structural integrity.

3. To design a model whose image generated should be close to real image with same structure.

1.2 Scope:

1. Proposed model works effectively with relatively small sample size.
2. Proposed model is useful in segmentation of medical images, specifically for retinal images.
3. The model can used for synthesizing diverse retinal images (for other purpose, the model needs to be retrained again as per requirement). The retinal image dataset generated can be useful for training other models (This can helpful in improvement of computer-aided techniques in ophthalmology).

1.3 Limitation:

1. To Train the model requires different time for different hardware resources used.
2. Model needs to be retrained for different image datasets.
3. Limited amount of data set.
4. Size of image data set needed for this model is 256x256 pixels.

1.4 **Timeline of Project:**

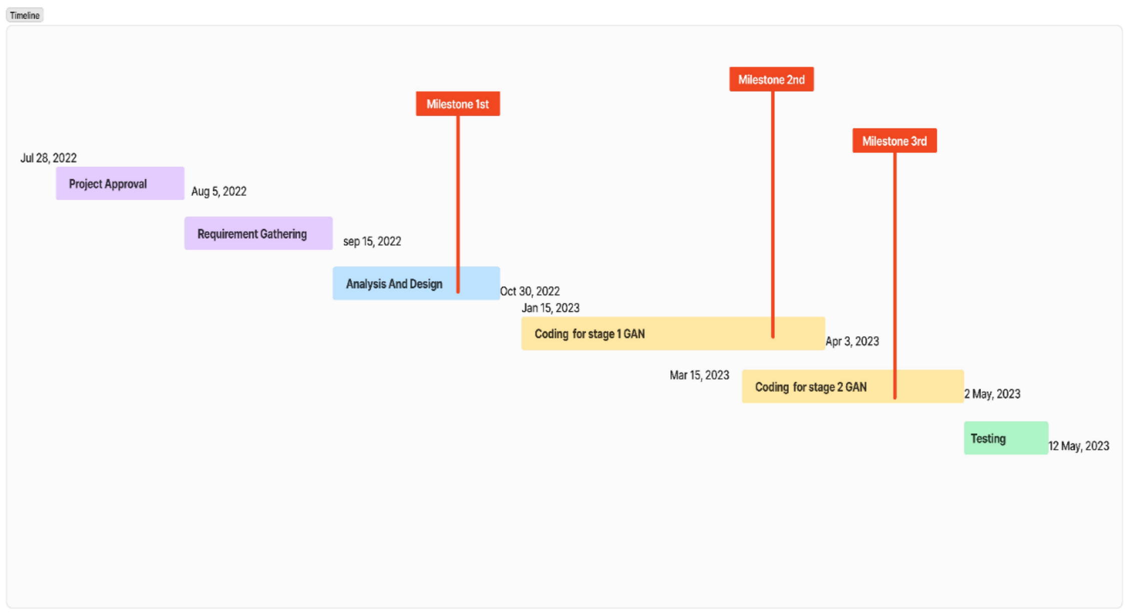


Fig. 1.4.0: Timeline of the project

Task Name	Duration	Start Date	End date
Domain Selection	2 days	28-7-2022	29-7-2022
Domain Finalization	2 days	29-7-2022	30-7-2022
Selection of Problem Statement	3 days	31-7-2022	2-8-2022
Finalization of Problem Statement	2 days	3-8-2022	5-8-2022
Study on Research Paper	25 days	6-8-2022	30-8-2022
Requirement Analysis	10 days	31-8-2022	9-9-2022
System Requirement	5 days	10-9-2022	15-9-2022
System Architecture	44 days	16-9-2022	30-10-2022

Stage 1 implementation	78 days	15-1-2023	3-3-2023
Stage 2 implementation	48 days	15-3-2023	2-5-2023
Testing	10 days	3-5-2023	12-5-2023

Table 1.4.0 – Timeline table

1.5 Project Cost:

Components	Name	Pricing
Graphics Card	Nvidia Tesla K40	50730
Processor	Intel i5 11 gen	13410
RAM	Corsair 16 GB	4895
Total		69035

Table 1.5.0: Hardware Cost

Software Cost:

Line of code: To develop the system 2140 lines of codes are required.

KLOC: KLOC is the estimated size of the software product indicates in Kilo Lines of Code.

KLOC = LOC / 1000

= 2140 / 1000

= 2.14

Effort: The effort is only a function of the number of lines of code and some constants evaluated according to the different software systems.

$E=aL^b$

Where L is KLOC (Kilo Line of Code)

$E=2.4*2.14K^{1.05} = 5.33$

Duration Estimation: Once the effort estimation is obtained, we can estimate the duration of the project. Assuming a standard productivity rate of 2 person per calendar month, the duration can be calculated as:

Duration (in months) = Effort / Productivity
= 5.33 / 2
= 2.66 months

2. RELATED WORK

These days, a lot of organizations, businesses, and governments around the world collect the vast amounts of data that are generated. They analyze these data statistically to get insights and further plan their operations, or they use them to improve their policies and goods. Even having an enormous amount of data, the healthcare industry is prohibited from sharing it with the general public, which can impede the research that requires such data to advance technology. With the use of synthetic data, even if they were scalar numbers, researchers from all over the world have moved data from the private domain to the public domain. This particular concept can also be investigated for the artificial image generation caused by GAN. This innovative method of estimating generative models through an adversarial process, in which two models—a generative model and a discriminative model—are trained simultaneously, has been put forth by Ian J. Goodfellow et al. [1]. The drawback in this case is that D and G must be perfectly synchronized during training. Efficiency can be increased by coming up with improved strategies for coordinating G and D.

For the community of signal processors, Antonia Creswell et al. [2] have presented an overview of GANs, outlining several approaches for developing and training GANs as well as practical uses. A variety of novel architectural features and training techniques that are used with the GAN framework have been supplied by Tim Salimans et al. [3]. Here, unstable training of GAN is addressed and have proposed to increase the stability and to improve quality of generated results. M. Krithika et al. [4] have reviewed different GAN architectures in medical image analysis. Different applications like image synthesis, segmentation, reconstruction, classification, etc., were discussed in this review paper. It was found that many training steps were required to reduce loss function, which makes balancing the generator and discriminator in single training difficult.

For supervised image analysis there is need of image data along with their corresponding ground-truths, which is costly as well as time consuming. Therefore, it is both cost-efficient and effective to use GAN for this. Before GANs may be widely used for retinal imaging, Valentina Bellemo et al. [5] have described the potential benefits and restrictions that need to be addressed. Recently, there has been significant advancement in the synthesis of medical pictures using the adversarial learning theory in general. Although GANs have been extensively used in the field of retinal fundus pictures for segmentation reasons, they have received relatively less attention for synthesis and super-resolution applications. It was well recognised that irregular breaks or abnormalities in synthetic vascular networks frequently make them unsuitable for clinical use.

P. Costa et al. [6] have proposed a purely data-driven approach. Here, model is trained on pairs of real vessel networks and their corresponding retinal fundus images. The drawback is that model dependant on the availability of a pre-existing vessel network in order to generate new one. Squared loss function is used. This is removed in [7] by building an auto-encoder that learns to generate realistic retinal vessel trees. . Pedro Costa et al. [7] have proposed implementing an adversarial autoencoder for the task of retinal network synthesis. The generated vessel trees are used as an intermediate stage for generating colour retinal images with the help of GAN. Both models require the optimization of differentiable loss functions, which allows to train them jointly. However, the generated synthetic vessel networks often exhibit abnormal interruptions and unusual width variation along the same vessel. An attempt is made to address this issue in this paper.

John Guibas et al. [8] proposed a novel, two-stage pipeline for generating synthetic medical images from a pair of generative adversarial networks, tested in practice on retinal fundi images. DCGAN is used in stage-1 with cross-entropy loss function, while CGAN is used in stage-2 for generation of photorealistic medical image. Authors used the universal F1-score to compare the similarity between synthetic and ground-truth images. For pipeline to be executed on a variety of medical images, it must have access to private research data.

Martin Arjovsky et al. [9] proposed a new GAN i.e Wasserstein-gan. This architecture uses the Wasserstein loss proposed by the authors to train the GAN, as opposed to the cross-entropy loss function which was mostly used. The proposed architecture helps us against common problem in GAN like mode collapse and provide more stability than DCGAN. As we know performance of GAN takes a heavy hit due to limited dataset. So, Shengyu Zhao et al. [10] proposed Differentiable Augmentation (DiffAugment) against such problem. This method performs various types of differentiable augmentations on both real and fake dataset, which in turn helps in improving the data efficiency of the GANs.

By using Wasserstein loss in the GAN architecture to provide more stability for the model. As we are training the GAN model on relatively small dataset, the DiffAugment method can be used to tackle the problem. In the previous paper, the optical disc in the fundus image was mostly present in the right side of the image. We can improve on this issue by performing simple data augmentation like flipping on the dataset. So, the trained model can give us more diverse images.

3. Requirement Analysis

3.1 Requirement Gathering:

In order to successfully develop a retinal image synthesis tool using GANs, it is important to gather the necessary requirements. The following is a list of potential requirements for such a project:

1. Data sources: The first requirement is to identify and gather retinal image datasets that will be used for training and testing the GAN model. It is essential to have a large and diverse dataset that accurately represents the target population and contains a range of different retinal pathologies. But we do not have large amount of data set. Datasets used for retinal synthesis are DRIVE (Digital Retinal Images for Vessel Extraction) and HRF (High Resolution Fundus)

2. Image pre-processing: Retinal images often contain various artifacts and image quality issues, such as poor contrast, uneven illumination, and motion blur. The pre-processing step involves addressing these issues and enhancing the image quality, which will improve the accuracy and reliability of the generated images.

3. GAN architecture: The choice of GAN architecture is crucial in determining the quality of the generated images. It is important to explore different generator and discriminator network architectures, as well as various hyperparameters and optimization techniques.

4. GAN Training: The system should train a GAN model using the acquired retinal images dataset to learn the underlying distribution and generate realistic retinal images. This process involves training both the generator and discriminator networks.

5. Evaluation metrics: To measure the performance of the GAN model, it is important to define and implement appropriate evaluation metrics. Common metrics include structural similarity index (SSIM), and the Fréchet inception distance (FID).

6. Output Formats: The system should support various output formats for the generated retinal images, such as JPEG, PNG, or DICOM, to facilitate compatibility and further analysis or sharing

7. Accuracy: The generated retinal images should closely resemble real retinal images, both visually and in terms of features and characteristics relevant to the specific domain. The system should strive for high fidelity and avoid generating unrealistic or distorted images.

8. Stability: The system should exhibit stable and consistent behavior across multiple runs and different hardware environments. It should be able to handle interruptions, errors, or system failures gracefully, recovering without data loss or corruption.

3.2**Requirement Specification:**

Requirement ID	**Requirement Description**	**Priority**	**Type**
RS-01	Data sources: The first requirement is to identify and gather retinal image datasets that will be used for training and testing the GAN model.	High	Functional
RS-02	Pre-processing: The system should pre-process the acquired retinal images to normalize their sizes, orientations, and colour channels to ensure consistent inputs for the GAN.	High	Functional
RS-03	GAN architecture: The choice of GAN architecture is crucial in determining the quality of the generated images	High	Funtional
RS-04	GAN Training: The system should train a GAN model using the acquired retinal images dataset to learn the underlying distribution and generate realistic retinal images.	High	Functional
RS-05	Evaluation metrics: To measure the performance of the GAN model, it is important to define and implement appropriate evaluation metrics.	High	Functional
RS-06	Output Formats: The system should	High	Functional

	support various output formats for the generated retinal images, such as JPEG, PNG		
RS-07	Accuracy: The generated retinal images should closely resemble real retinal images, both visually and in terms of features and characteristics relevant to the specific domain. The system should strive for high fidelity and avoid generating unrealistic or distorted images	Medium	Non functional
RS-08	Stability: The system should exhibit stable and consistent behavior across multiple runs and different hardware environments.	Medium	Non functional

Table 3.2.0- Requirement Specification

4. Methodology

4.1 Architecture Diagram:

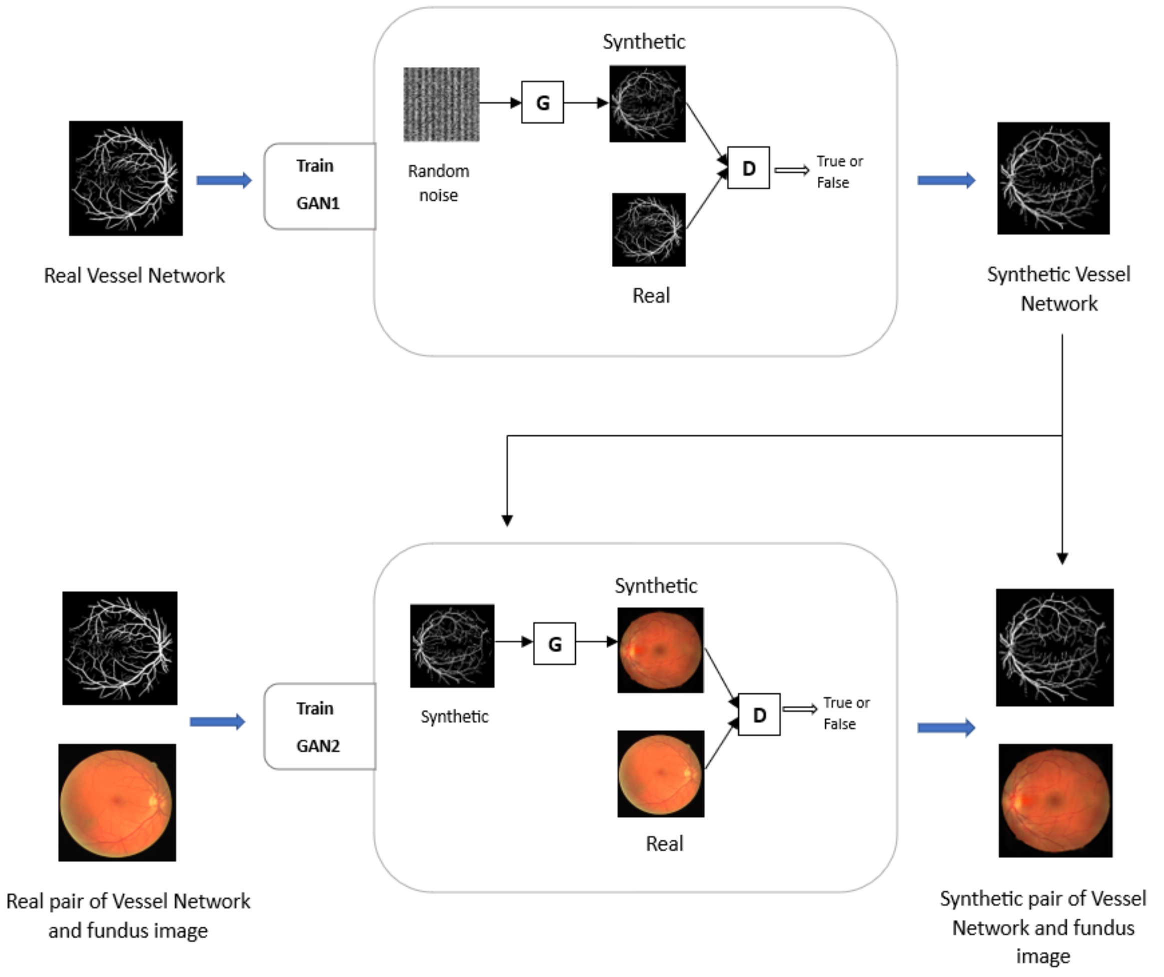


Fig.4.1.0 – Architecture Diagram

4.2Components:

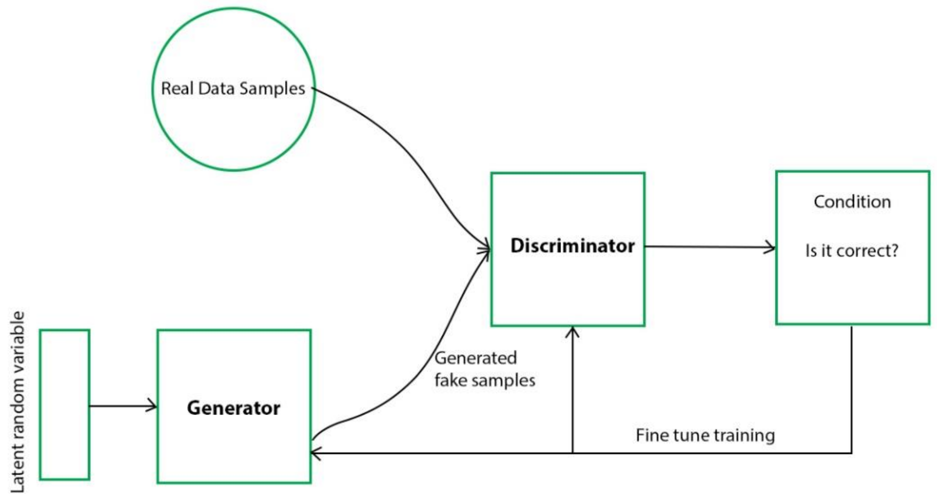


Fig.4.1.1 – GAN (Generative Adversarial Network)

4.1.1 Generator (G):

The generator is a crucial component of a Generative Adversarial Network (GAN) that takes random noise as input and aims to generate synthetic data samples that resemble the real data from the training set. The primary role of the generator is to learn the underlying patterns and structures of the training data and generate new samples that are difficult for the discriminator to distinguish from real data. Here are the key aspects of the generator:

1. Input: The generator typically takes random noise as input, often sampled from a uniform or normal distribution. The size of the input noise vector can vary depending on the specific GAN architecture and the nature of the dataset.

2. Upsampling Layers: The generator network uses upsampling layers (also known as deconvolutional layers or transposed convolutions) to transform the input noise into a higher-dimensional representation that resembles the original data. These layers increase the spatial dimensions of the input and help capture more complex patterns.

3. Activation Functions: Non-linear activation functions like ReLU (Rectified Linear Unit) or LeakyReLU (Leaky Rectified Linear Unit) are commonly used between the layers of the generator. These functions introduce non-linearity and help the network learn complex mappings between the input noise and the desired output.

4. Output Layer: The last layer of the generator produces the generated samples. The activation function in the output layer depends on the nature of the generated data. For example, sigmoid or tanh activations are often used when generating images to ensure that the pixel values fall within a specific range (e.g., [0, 1] or [-1, 1]).

Training:

During the training process of a GAN, the generator is trained to generate synthetic samples that are convincing enough to fool the discriminator. The adversarial training between the generator and the discriminator helps the generator improve its ability to produce samples that are indistinguishable from real data. The generator's training involves the following steps:

1. Forward Pass: The generator takes random noise as input and generates synthetic samples.

2. Discriminator Feedback: The generated samples are then fed into the discriminator, along with real samples from the training set. The discriminator provides feedback on the authenticity of the generated samples without revealing their true source.

3. Backpropagation: The generator's parameters are updated using backpropagation and gradient descent, based on the feedback from the discriminator. The objective is to improve the generator's ability to produce samples that the discriminator classifies as real.

4. Adversarial Training: The generator is trained in an adversarial manner, iteratively competing with the discriminator to generate more realistic samples. This process continues until the generator becomes skilled at generating samples that are difficult for the discriminator to distinguish from real data.

4.1.2 Discriminator (D):

The discriminator is a critical component of a Generative Adversarial Network (GAN) that aims to distinguish between real data samples from the training set and synthetic samples generated by the generator. The primary role of the discriminator is to learn to classify input data as either real or fake, providing feedback to the generator to improve its ability to generate realistic samples. Here are the key aspects of the discriminator:

1. Input: The discriminator takes as input either real data samples from the training set or synthetic samples generated by the generator. The size and shape of the input depend on the specific GAN architecture and the nature of the dataset.

2. Convolutional Layers: The discriminator network is typically constructed using convolutional layers, which are effective in capturing spatial features and patterns in images or other structured data. The number and configuration of these layers may vary depending on the complexity of the dataset and the desired discriminative capacity.

3. Activation Functions: Non-linear activation functions such as ReLU (Rectified Linear Unit) or LeakyReLU (Leaky Rectified Linear Unit) are typically applied after each convolutional layer. These activation

functions introduce non-linearity and help the discriminator learn complex decision boundaries.

4. Fully Connected Layers: Towards the end of the discriminator architecture, fully connected layers are often employed to aggregate the features learned from the convolutional layers. These layers connect all the neurons from the previous layer to the next, ultimately providing a discriminative decision.

5. Output Layer: The final layer of the discriminator produces a single scalar value representing the probability that the input sample is real or fake. In binary classification settings, a sigmoid activation function is usually applied to ensure the output falls within the range [0, 1].

Training:

During the training process of a GAN, the discriminator is trained to classify input samples as real or fake accurately. The adversarial training between the generator and the discriminator helps the discriminator improve its discriminative capabilities. The training of the discriminator involves the following steps:

1. Forward Pass: The discriminator takes a data sample as input, either real or generated.

2. Prediction: The discriminator produces a probability score indicating the likelihood of the input being real or fake. For real samples, the target label is 1 (indicating real), while for generated samples, the target label is 0 (indicating fake).

3. Loss Computation: The discriminator's loss is computed based on the discrepancy between the predicted probabilities and the target labels. Various loss functions can be used, such as binary cross-entropy or adversarial loss, to guide the discriminator's training.

4. Backpropagation: The gradients of the loss with respect to the discriminator's parameters are computed using backpropagation, and the

weights are updated using an optimization algorithm such as stochastic gradient descent (SGD) or Adam.

5. Adversarial Training: The discriminator is trained iteratively alongside the generator. The generator generates synthetic samples, and both real and fake samples are fed into the discriminator to provide feedback on their authenticity. This adversarial training process aims to improve the discriminator's ability to distinguish between real and fake samples accurately.

4.1.3 Stage 1 GAN:

In stage 1 GAN, the model is used to generate the vessel networks. It is based on the Wasserstein-gan architecture, built using TensorFlow. This method provides more training stability than the DCGAN architecture. It helps in reducing the mode collapse, which will help us getting diverse results for use. A noise vector is given as input to the generator, which then passes through multiple layers to generate a synthetic image. This synthetic image along with the real image sample are used to train the critic (similar to discriminator).

Here, the Wasserstein discriminator loss is used to train the critic in this stage:

$$\text{Loss} = D(x) - D(G(z))$$

The Wasserstein generator loss is used to train the generator in this stage:

$$\text{Loss} = D(G(z))$$

Here, $D(x)$ is the critic’s output for real data, $G(z)$ is the generator’s output when given random noise z and $D(G(z))$ is the critic’s output for the fake/synthetic data.

DCGAN architecture:

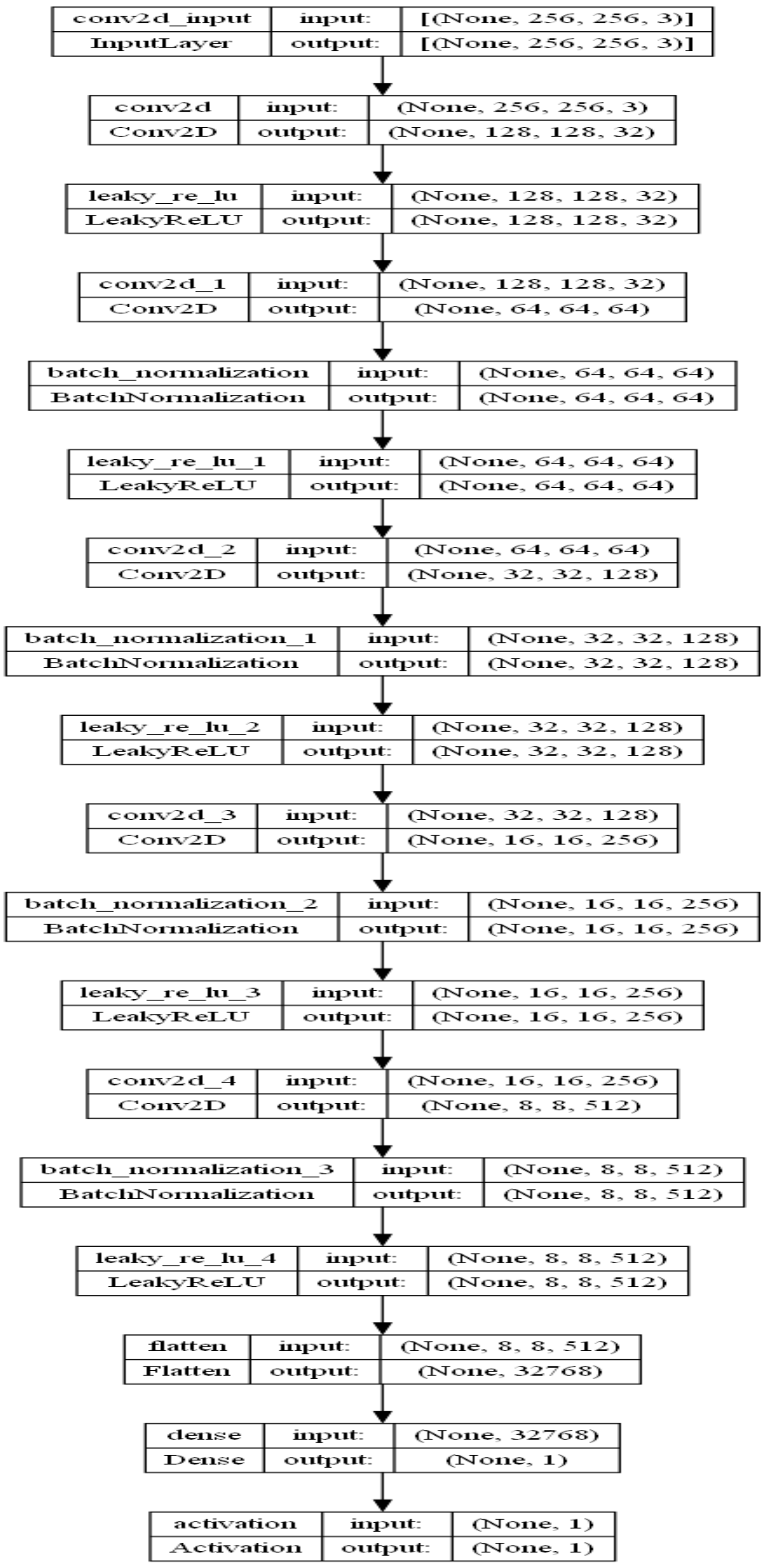

conv2d_input | input: | [(None, 256, 256, 3)]
InputLayer | output: | [(None, 256, 256, 3)]
conv2d | input: | (None, 256, 256, 3)
Conv2D | output: | (None, 128, 128, 32)
leaky_re_lu | input: | (None, 128, 128, 32)
LeakyReLU | output: | (None, 128, 128, 32)
conv2d_1 | input: | (None, 128, 128, 32)
Conv2D | output: | (None, 64, 64, 64)
batch_normalization | input: | (None, 64, 64, 64)
BatchNormalization | output: | (None, 64, 64, 64)
leaky_re_lu_1 | input: | (None, 64, 64, 64)
LeakyReLU | output: | (None, 64, 64, 64)
conv2d_2 | input: | (None, 64, 64, 64)
Conv2D | output: | (None, 32, 32, 128)
batch_normalization_1 | input: | (None, 32, 32, 128)
BatchNormalization | output: | (None, 32, 32, 128)
leaky_re_lu_2 | input: | (None, 32, 32, 128)
LeakyReLU | output: | (None, 32, 32, 128)
conv2d_3 | input: | (None, 32, 32, 128)
Conv2D | output: | (None, 16, 16, 256)
batch_normalization_2 | input: | (None, 16, 16, 256)
BatchNormalization | output: | (None, 16, 16, 256)
leaky_re_lu_3 | input: | (None, 16, 16, 256)
LeakyReLU | output: | (None, 16, 16, 256)
conv2d_4 | input: | (None, 16, 16, 256)
Conv2D | output: | (None, 8, 8, 512)
batch_normalization_3 | input: | (None, 8, 8, 512)
BatchNormalization | output: | (None, 8, 8, 512)
leaky_re_lu_4 | input: | (None, 8, 8, 512)
LeakyReLU | output: | (None, 8, 8, 512)
flatten | input: | (None, 8, 8, 512)
Flatten | output: | (None, 32768)
dense | input: | (None, 32768)
Dense | output: | (None, 1)
activation | input: | (None, 1)
Activation | output: | (None, 1)

Fig. 4.1.3.0 Discriminator

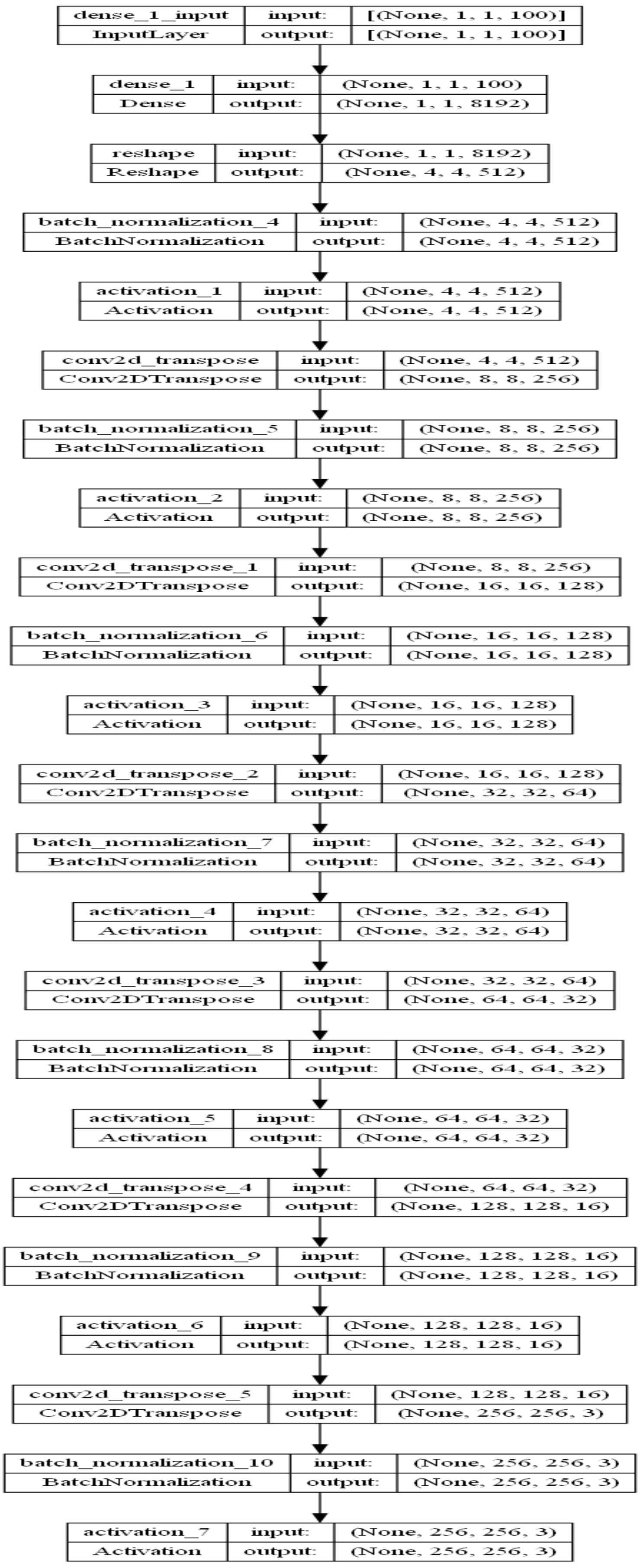

dense_1_input | input: | [(None, 1, 1, 100)]
InputLayer | output: | [(None, 1, 1, 100)]
dense_1 | input: | (None, 1, 1, 100)
Dense | output: | (None, 1, 1, 8192)
reshape | input: | (None, 1, 1, 8192)
Reshape | output: | (None, 4, 4, 512)
batch_normalization_4 | input: | (None, 4, 4, 512)
BatchNormalization | output: | (None, 4, 4, 512)
activation_1 | input: | (None, 4, 4, 512)
Activation | output: | (None, 4, 4, 512)
conv2d_transpose | input: | (None, 4, 4, 512)
Conv2DTranspose | output: | (None, 8, 8, 256)
batch_normalization_5 | input: | (None, 8, 8, 256)
BatchNormalization | output: | (None, 8, 8, 256)
activation_2 | input: | (None, 8, 8, 256)
Activation | output: | (None, 8, 8, 256)
conv2d_transpose_1 | input: | (None, 8, 8, 256)
Conv2DTranspose | output: | (None, 16, 16, 128)
batch_normalization_6 | input: | (None, 16, 16, 128)
BatchNormalization | output: | (None, 16, 16, 128)
activation_3 | input: | (None, 16, 16, 128)
Activation | output: | (None, 16, 16, 128)
conv2d_transpose_2 | input: | (None, 16, 16, 128)
Conv2DTranspose | output: | (None, 32, 32, 64)
batch_normalization_7 | input: | (None, 32, 32, 64)
BatchNormalization | output: | (None, 32, 32, 64)
activation_4 | input: | (None, 32, 32, 64)
Activation | output: | (None, 32, 32, 64)
conv2d_transpose_3 | input: | (None, 32, 32, 64)
Conv2DTranspose | output: | (None, 64, 64, 32)
batch_normalization_8 | input: | (None, 64, 64, 32)
BatchNormalization | output: | (None, 64, 64, 32)
activation_5 | input: | (None, 64, 64, 32)
Activation | output: | (None, 64, 64, 32)
conv2d_transpose_4 | input: | (None, 64, 64, 32)
Conv2DTranspose | output: | (None, 128, 128, 16)
batch_normalization_9 | input: | (None, 128, 128, 16)
BatchNormalization | output: | (None, 128, 128, 16)
activation_6 | input: | (None, 128, 128, 16)
Activation | output: | (None, 128, 128, 16)
conv2d_transpose_5 | input: | (None, 128, 128, 16)
Conv2DTranspose | output: | (None, 256, 256, 3)
batch_normalization_10 | input: | (None, 256, 256, 3)
BatchNormalization | output: | (None, 256, 256, 3)
activation_7 | input: | (None, 256, 256, 3)
Activation | output: | (None, 256, 256, 3)

Fig. 4.1.3.1 Generator

Wasserstein GAN architecture:

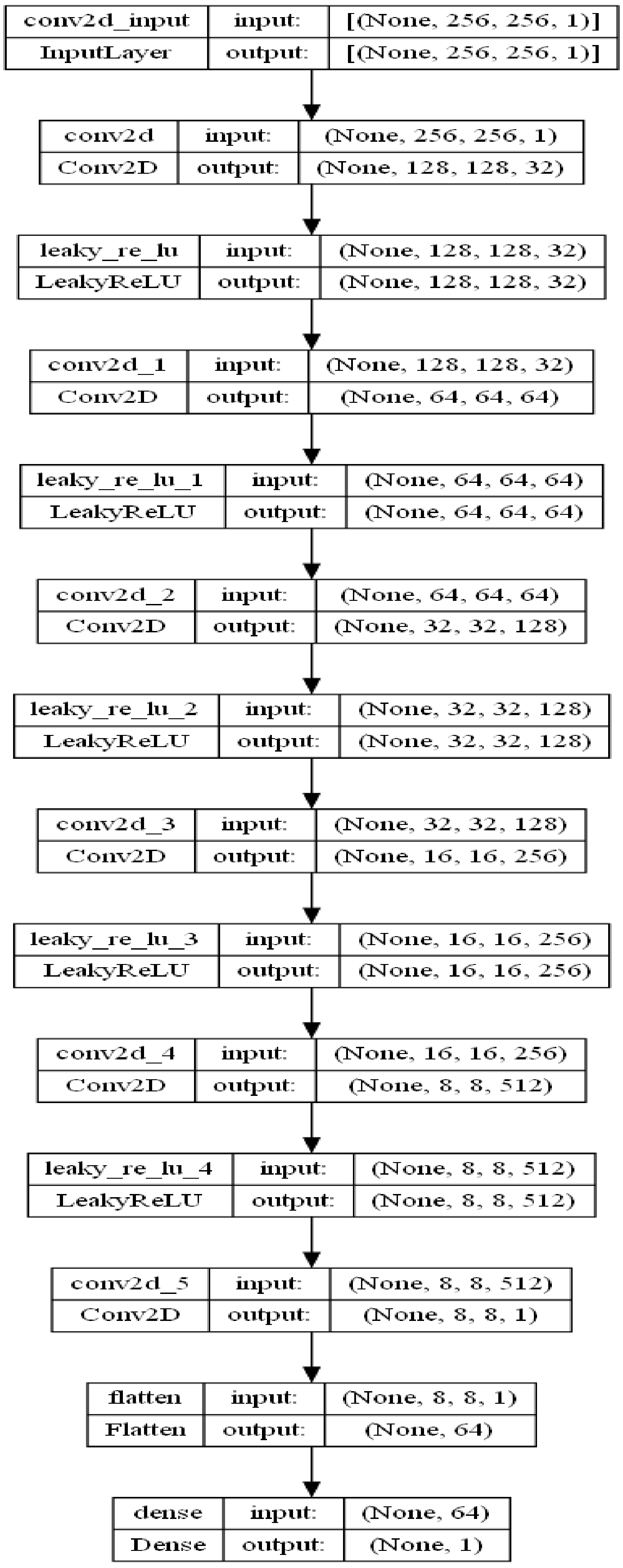

conv2d_input | input: | [(None, 256, 256, 1)]
InputLayer | output: | [(None, 256, 256, 1)]
conv2d | input: | (None, 256, 256, 1)
Conv2D | output: | (None, 128, 128, 32)
leaky_re_lu | input: | (None, 128, 128, 32)
LeakyReLU | output: | (None, 128, 128, 32)
conv2d_1 | input: | (None, 128, 128, 32)
Conv2D | output: | (None, 64, 64, 64)
leaky_re_lu_1 | input: | (None, 64, 64, 64)
LeakyReLU | output: | (None, 64, 64, 64)
conv2d_2 | input: | (None, 64, 64, 64)
Conv2D | output: | (None, 32, 32, 128)
leaky_re_lu_2 | input: | (None, 32, 32, 128)
LeakyReLU | output: | (None, 32, 32, 128)
conv2d_3 | input: | (None, 32, 32, 128)
Conv2D | output: | (None, 16, 16, 256)
leaky_re_lu_3 | input: | (None, 16, 16, 256)
LeakyReLU | output: | (None, 16, 16, 256)
conv2d_4 | input: | (None, 16, 16, 256)
Conv2D | output: | (None, 8, 8, 512)
leaky_re_lu_4 | input: | (None, 8, 8, 512)
LeakyReLU | output: | (None, 8, 8, 512)
conv2d_5 | input: | (None, 8, 8, 512)
Conv2D | output: | (None, 8, 8, 1)
flatten | input: | (None, 8, 8, 1)
Flatten | output: | (None, 64)
dense | input: | (None, 64)
Dense | output: | (None, 1)

Fig. 4.1.3.2 Discriminator

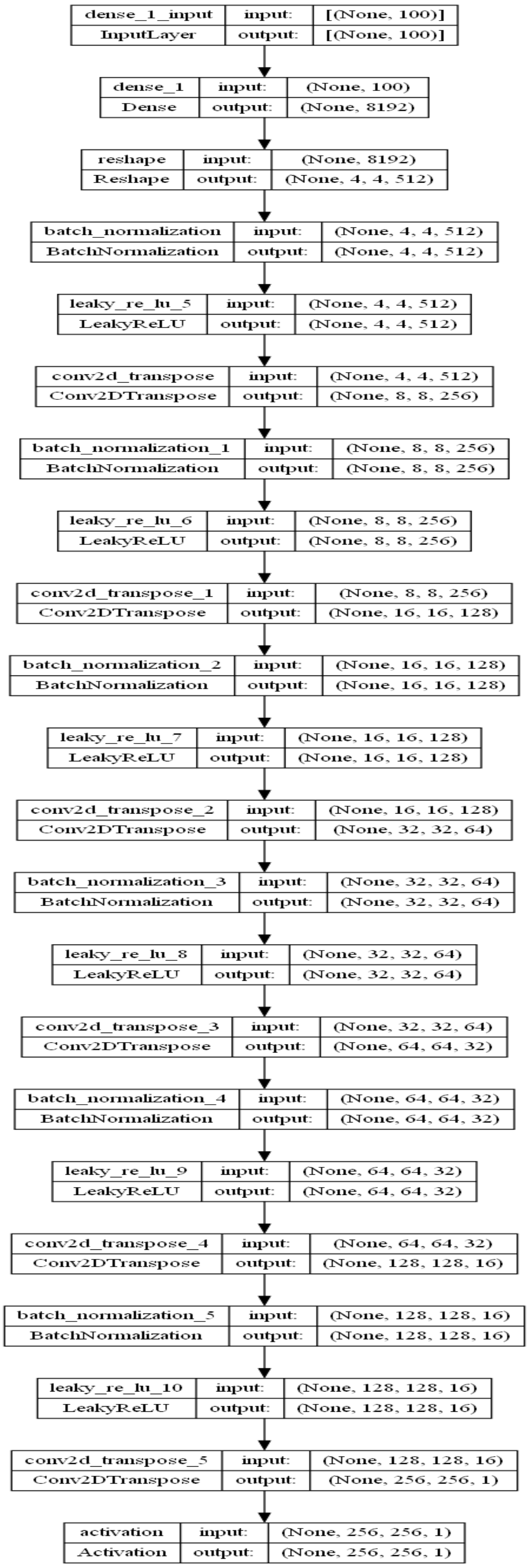

dense_1_input | input: | [(None, 100)]
InputLayer | output: | [(None, 100)]
dense_1 | input: | (None, 100)
Dense | output: | (None, 8192)
reshape | input: | (None, 8192)
Reshape | output: | (None, 4, 4, 512)
batch_normalization | input: | (None, 4, 4, 512)
BatchNormalization | output: | (None, 4, 4, 512)
leaky_re_lu_5 | input: | (None, 4, 4, 512)
LeakyReLU | output: | (None, 4, 4, 512)
conv2d_transpose | input: | (None, 4, 4, 512)
Conv2DTranspose | output: | (None, 8, 8, 256)
batch_normalization_1 | input: | (None, 8, 8, 256)
BatchNormalization | output: | (None, 8, 8, 256)
leaky_re_lu_6 | input: | (None, 8, 8, 256)
LeakyReLU | output: | (None, 8, 8, 256)
conv2d_transpose_1 | input: | (None, 8, 8, 256)
Conv2DTranspose | output: | (None, 16, 16, 128)
batch_normalization_2 | input: | (None, 16, 16, 128)
BatchNormalization | output: | (None, 16, 16, 128)
leaky_re_lu_7 | input: | (None, 16, 16, 128)
LeakyReLU | output: | (None, 16, 16, 128)
conv2d_transpose_2 | input: | (None, 16, 16, 128)
Conv2DTranspose | output: | (None, 32, 32, 64)
batch_normalization_3 | input: | (None, 32, 32, 64)
BatchNormalization | output: | (None, 32, 32, 64)
leaky_re_lu_8 | input: | (None, 32, 32, 64)
LeakyReLU | output: | (None, 32, 32, 64)
conv2d_transpose_3 | input: | (None, 32, 32, 64)
Conv2DTranspose | output: | (None, 64, 64, 32)
batch_normalization_4 | input: | (None, 64, 64, 32)
BatchNormalization | output: | (None, 64, 64, 32)
leaky_re_lu_9 | input: | (None, 64, 64, 32)
LeakyReLU | output: | (None, 64, 64, 32)
conv2d_transpose_4 | input: | (None, 64, 64, 32)
Conv2DTranspose | output: | (None, 128, 128, 16)
batch_normalization_5 | input: | (None, 128, 128, 16)
BatchNormalization | output: | (None, 128, 128, 16)
leaky_re_lu_10 | input: | (None, 128, 128, 16)
LeakyReLU | output: | (None, 128, 128, 16)
conv2d_transpose_5 | input: | (None, 128, 128, 16)
Conv2DTranspose | output: | (None, 256, 256, 1)
activation | input: | (None, 256, 256, 1)
Activation | output: | (None, 256, 256, 1)

Fig. 4.1.3.3 Generator

4.1.4 Stage 2 GAN:

In stage 2 GAN, the model generates corresponding eye fundus image from the synthetic vessel network image from previous stage. It basically performs image-to-image translation from vessel network to the corresponding eye fundus. Pix2pix GAN is used here for the translation.

The stage 2 GAN is trained with the pair of real vessel network image and corresponding eye fundus image in order to find the mapping between the two classes of images. The generator is given vessel network image as an input to translate into corresponding eye fundus image. The discriminator is trained with the real pairs of images as well as the fake pairs, to help it classify between real and fake image.

Pix2Pix GAN architecture:

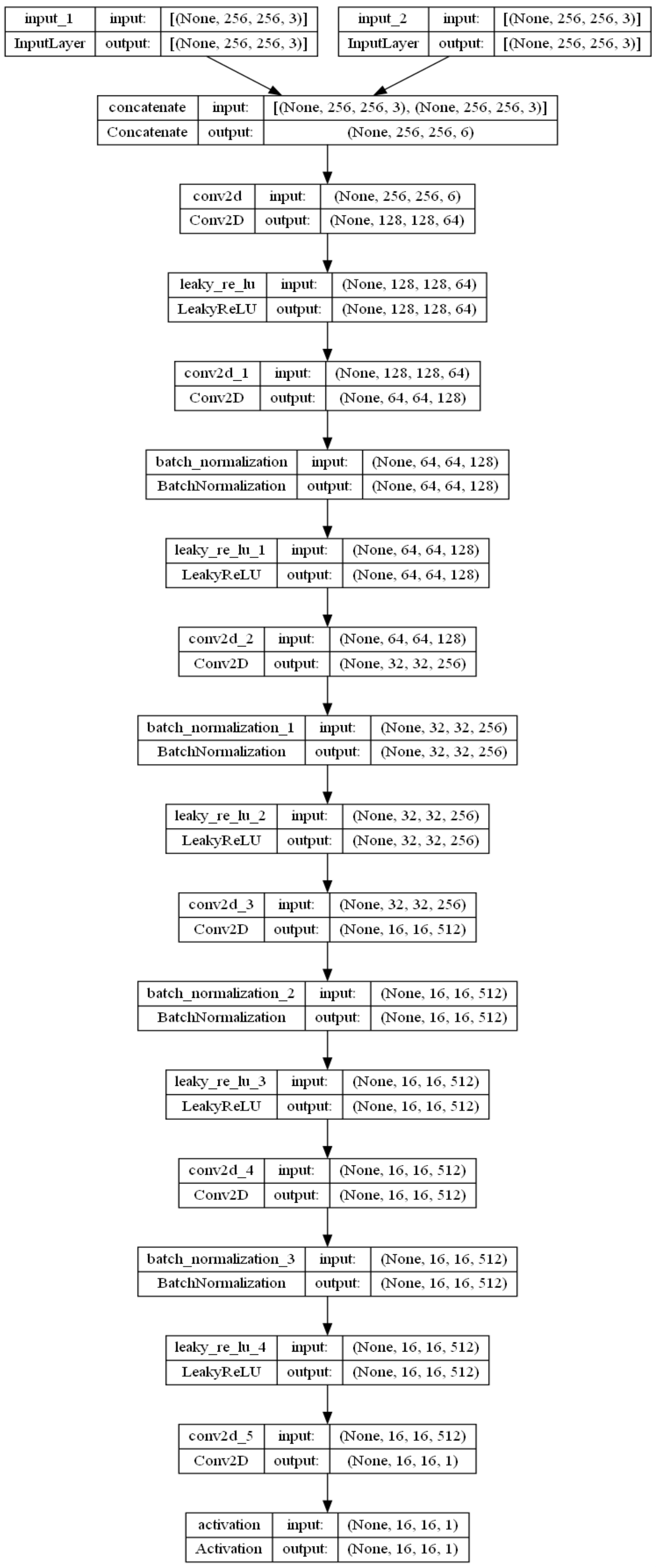
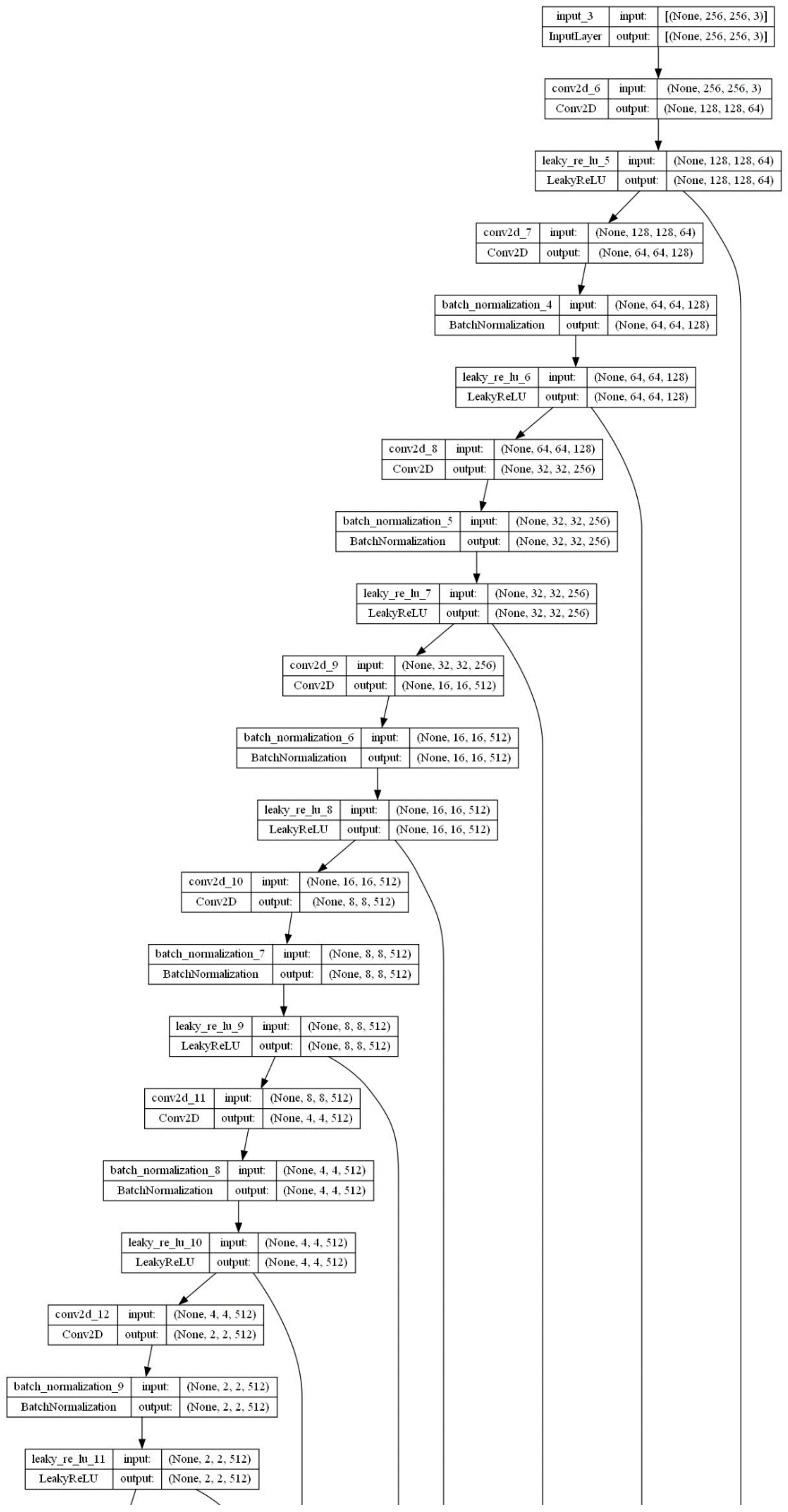
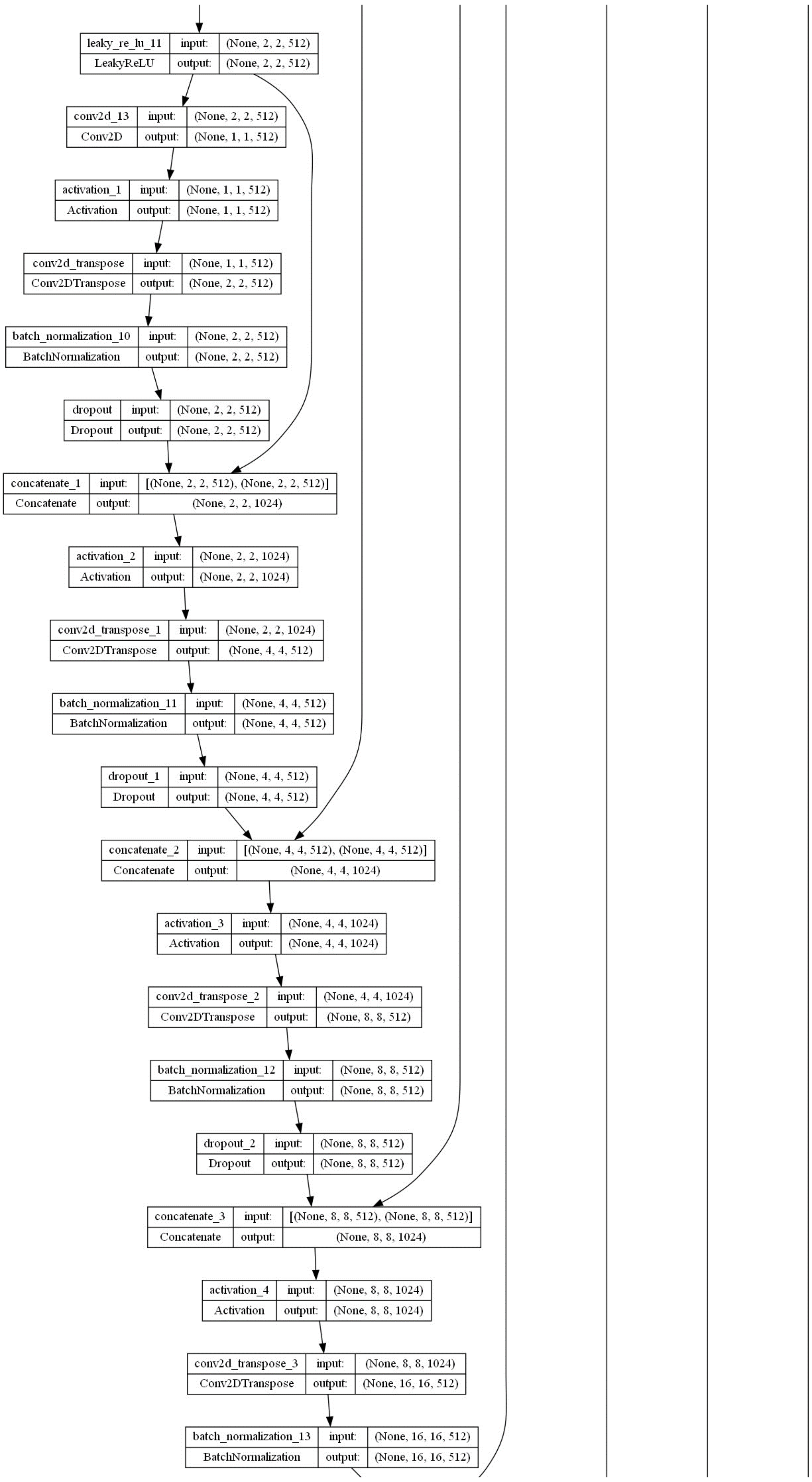
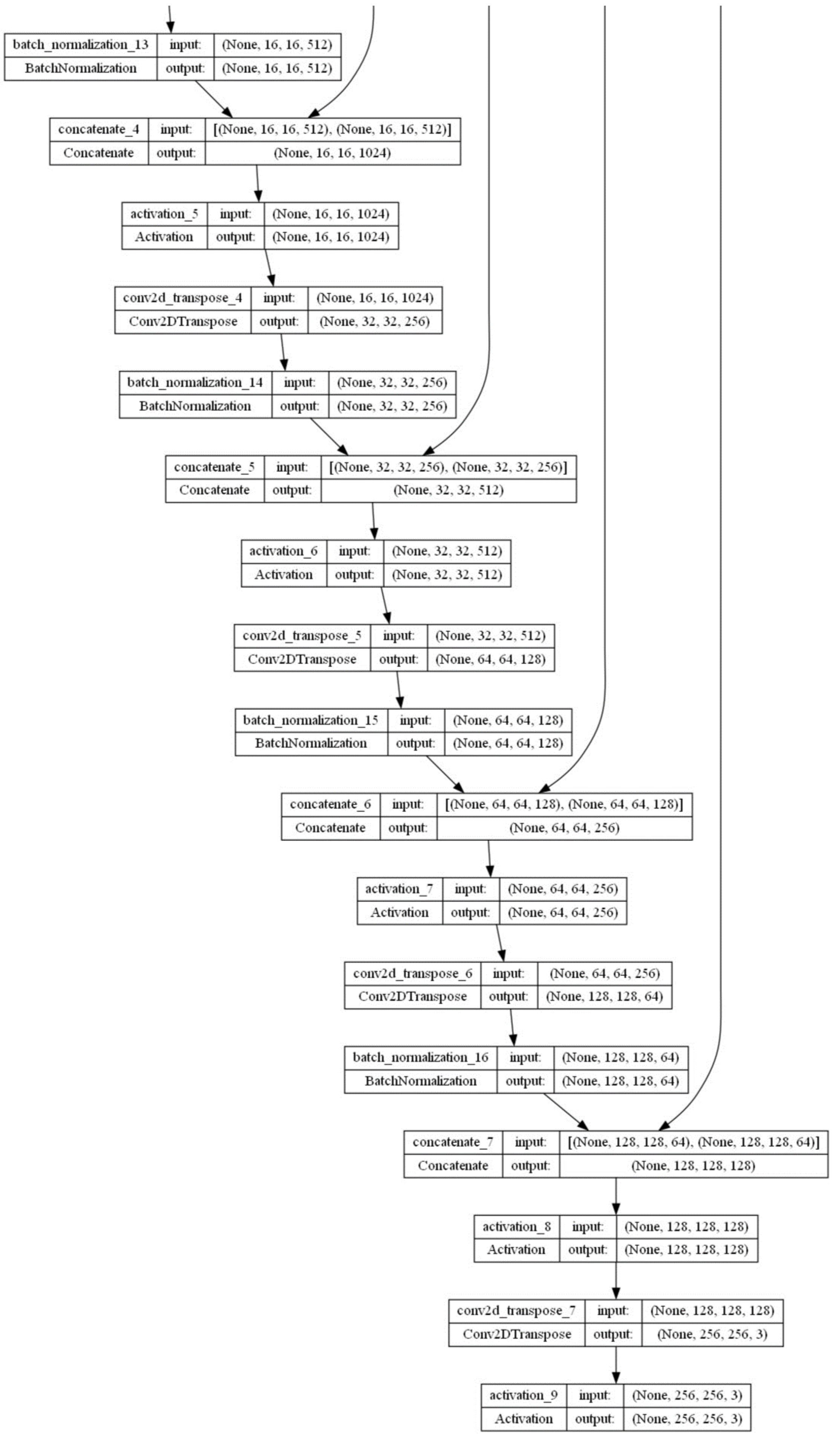

input_1 | InputLayer | input: [(None, 256, 256, 3)] | output: [(None, 256, 256, 3)]
input_2 | InputLayer | input: [(None, 256, 256, 3)] | output: [(None, 256, 256, 3)]
concatenate | Concatenate | input: [(None, 256, 256, 3), (None, 256, 256, 3)] | output: (None, 256, 256, 6)
conv2d | Conv2D | input: (None, 256, 256, 6) | output: (None, 128, 128, 64)
leaky_re_lu | LeakyReLU | input: (None, 128, 128, 64) | output: (None, 128, 128, 64)
conv2d_1 | Conv2D | input: (None, 128, 128, 64) | output: (None, 64, 64, 128)
batch_normalization | BatchNormalization | input: (None, 64, 64, 128) | output: (None, 64, 64, 128)
leaky_re_lu_1 | LeakyReLU | input: (None, 64, 64, 128) | output: (None, 64, 64, 128)
conv2d_2 | Conv2D | input: (None, 64, 64, 128) | output: (None, 32, 32, 256)
batch_normalization_1 | BatchNormalization | input: (None, 32, 32, 256) | output: (None, 32, 32, 256)
leaky_re_lu_2 | LeakyReLU | input: (None, 32, 32, 256) | output: (None, 32, 32, 256)
conv2d_3 | Conv2D | input: (None, 32, 32, 256) | output: (None, 16, 16, 512)
batch_normalization_2 | BatchNormalization | input: (None, 16, 16, 512) | output: (None, 16, 16, 512)
leaky_re_lu_3 | LeakyReLU | input: (None, 16, 16, 512) | output: (None, 16, 16, 512)
conv2d_4 | Conv2D | input: (None, 16, 16, 512) | output: (None, 16, 16, 512)
batch_normalization_3 | BatchNormalization | input: (None, 16, 16, 512) | output: (None, 16, 16, 512)
leaky_re_lu_4 | LeakyReLU | input: (None, 16, 16, 512) | output: (None, 16, 16, 512)
conv2d_5 | Conv2D | input: (None, 16, 16, 512) | output: (None, 16, 16, 1)
activation | Activation | input: (None, 16, 16, 1) | output: (None, 16, 16, 1)

Fig. 4.1.4.0 Pix2pix Discriminator

PIX2PIX Generator


input_3 | input: | [(None, 256, 256, 3)]
InputLayer | output: | [(None, 256, 256, 3)]
conv2d_6 | input: | (None, 256, 256, 3)
Conv2D | output: | (None, 128, 128, 64)
leaky_re_lu_5 | input: | (None, 128, 128, 64)
LeakyReLU | output: | (None, 128, 128, 64)
conv2d_7 | input: | (None, 128, 128, 64)
Conv2D | output: | (None, 64, 64, 128)
batch_normalization_4 | input: | (None, 64, 64, 128)
BatchNormalization | output: | (None, 64, 64, 128)
leaky_re_lu_6 | input: | (None, 64, 64, 128)
LeakyReLU | output: | (None, 64, 64, 128)
conv2d_8 | input: | (None, 64, 64, 128)
Conv2D | output: | (None, 32, 32, 256)
batch_normalization_5 | input: | (None, 32, 32, 256)
BatchNormalization | output: | (None, 32, 32, 256)
leaky_re_lu_7 | input: | (None, 32, 32, 256)
LeakyReLU | output: | (None, 32, 32, 256)
conv2d_9 | input: | (None, 32, 32, 256)
Conv2D | output: | (None, 16, 16, 512)
batch_normalization_6 | input: | (None, 16, 16, 512)
BatchNormalization | output: | (None, 16, 16, 512)
leaky_re_lu_8 | input: | (None, 16, 16, 512)
LeakyReLU | output: | (None, 16, 16, 512)
conv2d_10 | input: | (None, 16, 16, 512)
Conv2D | output: | (None, 8, 8, 512)
batch_normalization_7 | input: | (None, 8, 8, 512)
BatchNormalization | output: | (None, 8, 8, 512)
leaky_re_lu_9 | input: | (None, 8, 8, 512)
LeakyReLU | output: | (None, 8, 8, 512)
conv2d_11 | input: | (None, 8, 8, 512)
Conv2D | output: | (None, 4, 4, 512)
batch_normalization_8 | input: | (None, 4, 4, 512)
BatchNormalization | output: | (None, 4, 4, 512)
leaky_re_lu_10 | input: | (None, 4, 4, 512)
LeakyReLU | output: | (None, 4, 4, 512)
conv2d_12 | input: | (None, 4, 4, 512)
Conv2D | output: | (None, 2, 2, 512)
batch_normalization_9 | input: | (None, 2, 2, 512)
BatchNormalization | output: | (None, 2, 2, 512)
leaky_re_lu_11 | input: | (None, 2, 2, 512)
LeakyReLU | output: | (None, 2, 2, 512)


leaky_re_lu_11 | LeakyReLU | input: (None, 2, 2, 512) | output: (None, 2, 2, 512)
conv2d_13 | Conv2D | input: (None, 2, 2, 512) | output: (None, 1, 1, 512)
activation_1 | Activation | input: (None, 1, 1, 512) | output: (None, 1, 1, 512)
conv2d_transpose | Conv2DTranspose | input: (None, 1, 1, 512) | output: (None, 2, 2, 512)
batch_normalization_10 | BatchNormalization | input: (None, 2, 2, 512) | output: (None, 2, 2, 512)
dropout | Dropout | input: (None, 2, 2, 512) | output: (None, 2, 2, 512)
concatenate_1 | Concatenate | input: [(None, 2, 2, 512), (None, 2, 2, 512)] | output: (None, 2, 2, 1024)
activation_2 | Activation | input: (None, 2, 2, 1024) | output: (None, 2, 2, 1024)
conv2d_transpose_1 | Conv2DTranspose | input: (None, 2, 2, 1024) | output: (None, 4, 4, 512)
batch_normalization_11 | BatchNormalization | input: (None, 4, 4, 512) | output: (None, 4, 4, 512)
dropout_1 | Dropout | input: (None, 4, 4, 512) | output: (None, 4, 4, 512)
concatenate_2 | Concatenate | input: [(None, 4, 4, 512), (None, 4, 4, 512)] | output: (None, 4, 4, 1024)
activation_3 | Activation | input: (None, 4, 4, 1024) | output: (None, 4, 4, 1024)
conv2d_transpose_2 | Conv2DTranspose | input: (None, 4, 4, 1024) | output: (None, 8, 8, 512)
batch_normalization_12 | BatchNormalization | input: (None, 8, 8, 512) | output: (None, 8, 8, 512)
dropout_2 | Dropout | input: (None, 8, 8, 512) | output: (None, 8, 8, 512)
concatenate_3 | Concatenate | input: [(None, 8, 8, 512), (None, 8, 8, 512)] | output: (None, 8, 8, 1024)
activation_4 | Activation | input: (None, 8, 8, 1024) | output: (None, 8, 8, 1024)
conv2d_transpose_3 | Conv2DTranspose | input: (None, 8, 8, 1024) | output: (None, 16, 16, 512)
batch_normalization_13 | BatchNormalization | input: (None, 16, 16, 512) | output: (None, 16, 16, 512)


batch_normalization_13 | BatchNormalization | input: (None, 16, 16, 512) | output: (None, 16, 16, 512)
concatenate_4 | Concatenate | input: [(None, 16, 16, 512), (None, 16, 16, 512)] | output: (None, 16, 16, 1024)
activation_5 | Activation | input: (None, 16, 16, 1024) | output: (None, 16, 16, 1024)
conv2d_transpose_4 | Conv2DTranspose | input: (None, 16, 16, 1024) | output: (None, 32, 32, 256)
batch_normalization_14 | BatchNormalization | input: (None, 32, 32, 256) | output: (None, 32, 32, 256)
concatenate_5 | Concatenate | input: [(None, 32, 32, 256), (None, 32, 32, 256)] | output: (None, 32, 32, 512)
activation_6 | Activation | input: (None, 32, 32, 512) | output: (None, 32, 32, 512)
conv2d_transpose_5 | Conv2DTranspose | input: (None, 32, 32, 512) | output: (None, 64, 64, 128)
batch_normalization_15 | BatchNormalization | input: (None, 64, 64, 128) | output: (None, 64, 64, 128)
concatenate_6 | Concatenate | input: [(None, 64, 64, 128), (None, 64, 64, 128)] | output: (None, 64, 64, 256)
activation_7 | Activation | input: (None, 64, 64, 256) | output: (None, 64, 64, 256)
conv2d_transpose_6 | Conv2DTranspose | input: (None, 64, 64, 256) | output: (None, 128, 128, 64)
batch_normalization_16 | BatchNormalization | input: (None, 128, 128, 64) | output: (None, 128, 128, 64)
concatenate_7 | Concatenate | input: [(None, 128, 128, 64), (None, 128, 128, 64)] | output: (None, 128, 128, 128)
activation_8 | Activation | input: (None, 128, 128, 128) | output: (None, 128, 128, 128)
conv2d_transpose_7 | Conv2DTranspose | input: (None, 128, 128, 128) | output: (None, 256, 256, 3)
activation_9 | Activation | input: (None, 256, 256, 3) | output: (None, 256, 256, 3)

4.3 Algorithm

Stage 1:

1. Start
2. Load the training images which present in the dataset (retinal vessel network)
3. Normalize the images to (-1,1)
4. Create the batches of images set
5. Create the Discriminator
6. Create the Generator
7. Define the loss function and optimizer
8. Write the Training Steps
9. Train the GAN
10. End

Stage 2:

1. Start
2. Load the paired vessel network and corresponding fundus images for training
3. Normalize the images to (-1,1)
4. Create the Discriminator
5. Create the Generator
6. Create GAN model
7. Compile the models with loss function and optimizers
8. Write the training steps
9. Train the GAN
10. End

5. Implementation

5.1. Environment Setting:

5.1.1 Environmental setting for running the model

Name	version
Python	3.9.16
Keras	2.6.0
Keras-preprocessing	1.1.2
Tensorflow	2.6.0
Tensorflow - gpu	2.6.0
Matplotlib	3.7.0
Matplotlib-inline	0.1.6
numpy	1.23.5
Pillow	9.4.0
ipyplot	1.1.1
ipykernel	6.19.2

Table 5.1.0 – Environmental setting

5.1.2 For training the Stage 1:

DCGAN:

Optimizer used for generator and discriminator is Adam. Learning rate is set to 0.0002.

Loss function used for generator and discriminator is binary cross entropy.

Wasserstein GAN:

Optimizer used for generator and discriminator is Adam. Learning rate is set to 0.0002.

Loss function used for generator and discriminator is Wasserstein loss

5.1.3 For training STAGE 2:

Optimizer used for generator and discriminator is Adam. Learning rate is set to 0.0002.

Loss function used for generator and discriminator is binary cross entropy, MAE.

5.2. Experimentation

STAGE 1:

In experimentation of Stage-1 in GAN the DRIVE (Digital Retinal Images for Vessel Extraction) and HRF (High Resolution Fundus) image datasets are used. DRIVE dataset contains 20 annotated (vessel networks) images, while HRF dataset contains 45 annotated images which are used for synthetic generation vessel network images. The image is resized to 256x256 pixels. The optical disc present in images is mostly present in right side in DRIVE dataset i.e., 16 out of 20 images have optical disc at right, so due to this the generated images will mostly generate vessel images having optical discs at right side. In HRF dataset the optical disc is on right side for 25 times out of 45. So, we have flipped the images horizontally as well as vertically(mirror) for both original and horizontally flipped images. This will help in minimizing the issue as well as somewhat increase the dataset size from 65 to 260.

First train the discriminator using real images and then with fake images both done separately. After training Discriminator, train the Generator network. These both networks were trained using the train_on_batch() method. The training was done for 1000 epochs, with saving samples at every four epochs.

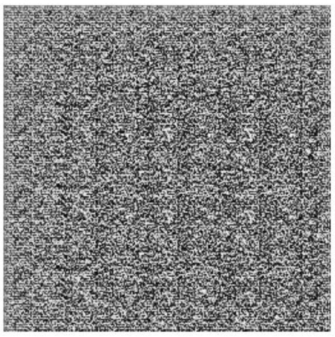


fig 5.2.0- 4th

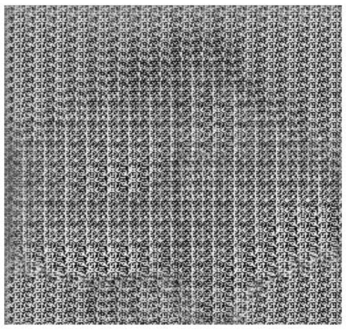


fig 5.2.1- 1000th

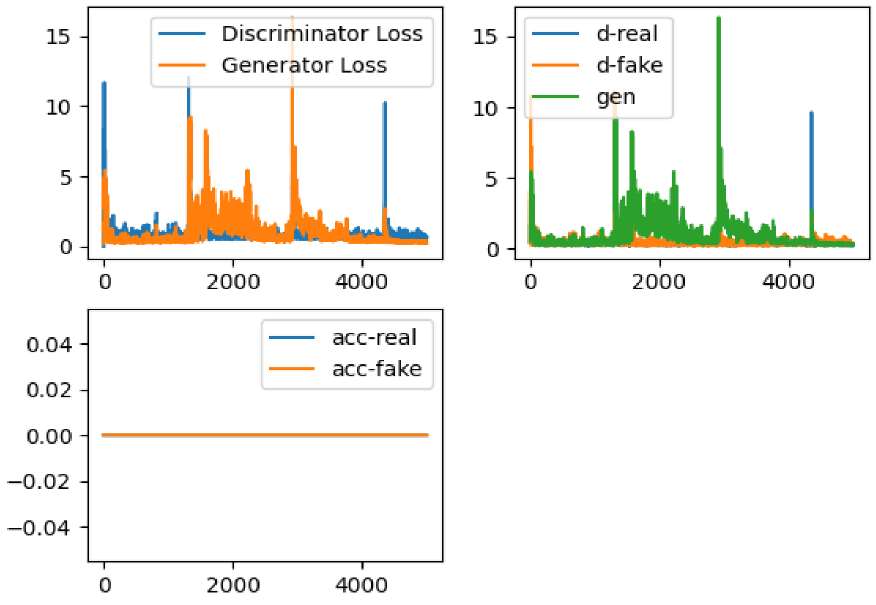


Fig 5.2.2 – Loss

As by seeing the images in fig 5.2.0 and fig 5.2.1, conclusion is that there was not much change in the quality of image generates at 4th epoch and image generated at 1000th epoch.

As by observing the loss plot in fig 5.2.3 and the images generated, we can say that the model has been failed to converge. Ideally, the discriminator loss for real and fake images is about the same at or around 0.5, and loss for the generator is slightly higher between 0.5 and 2.0. But the discriminator loss for real images was observed at 0.324 and for fake images at 0.332, while the generator loss was observed at 0.282. So, we can conclude that this GAN model did not perform as expected.

The Wasserstein GAN is said to provide better stability [] than the simple DCGAN architecture. So, for the next GAN model we use Wasserstein GAN. There is not any gradient clipping in the DCGAN, but it is done in Wasserstein GAN for Critic network (or Discriminator). RMSprop optimizer function is used with learning rate of 0.00005. The loss function used is Wasserstein generator loss. This gives us our Generator network. Create the GAN model using these Discriminator and Generator Networks. First train the critic using real images and then with fake images both done separately. After training Discriminator, train the Generator network. These both networks were trained using the train_on_batch() method where discriminator was updated 5 times more than generator. The training was done for 720 epochs.

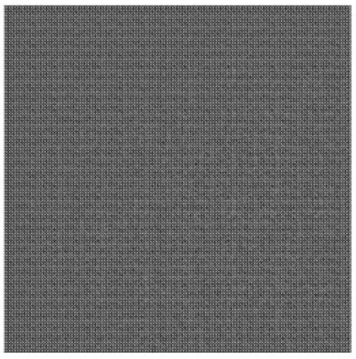


fig 5.2.3- 1th epoch

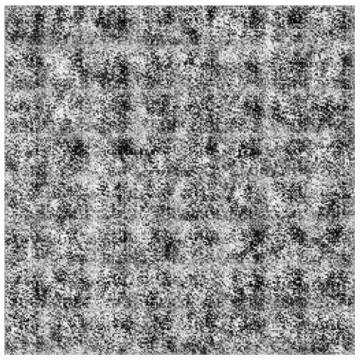


fig 5.2.4- 720th epoch

As by seeing the images in fig 5.2.3 and fig 5.2.3, conclusion is that there may appear a change in image but was not much change in the quality of image generated at 1th epoch and image generated at 720th epoch.

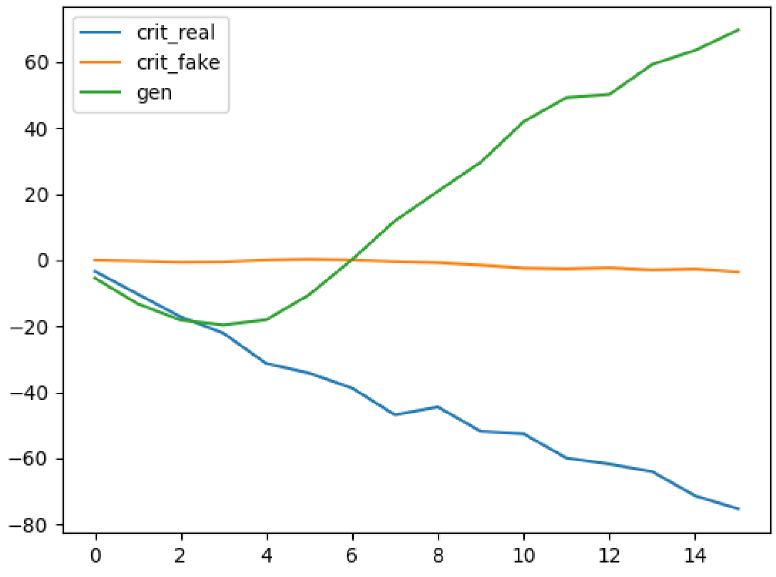


fig 5.2.5- Loss Plot

As by observing the loss plot in fig 5.2.5 and the images generated, conclusion is that the model has been failed to converge. From the fig 5.2.5 conclusion is that the critic loss for real images was observed at approx. 70 and for fake images at approx. 0, while the generator loss was observed at -80. For the lower Wasserstein loss for generated images means better image quality.

From the both GAN models' observation is that model is failing to converge. There are reasons like wrong architecture, low training data or incorrect data, etc due to which model fails to converge. It normally needs 50000 training images to train a high-quality GAN, but we are training the model using only 260 images. So, we apply Differentiable Augmentation (DiffAugment) [10] for increasing the data efficiency of the GAN. DiffAugment helps in adopting differentiable augmentation for the generated samples, thereby stabilizes training and to get a better convergence.

We kept the same DCGAN architecture we previously used, the only change we made was we applied the DiffAugment method on real and fake images before sending them to discriminator for training using train_on_batch method. The training for this model was done for 12500 epochs.

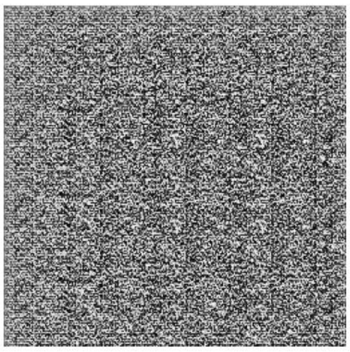


fig 5.2.6- 4th epoch

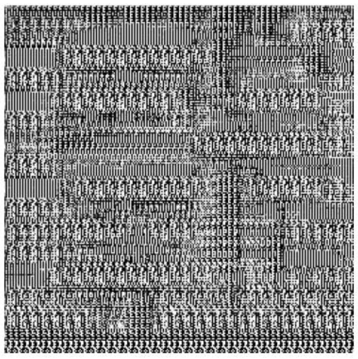


fig 5.2.7- 2500th epoch

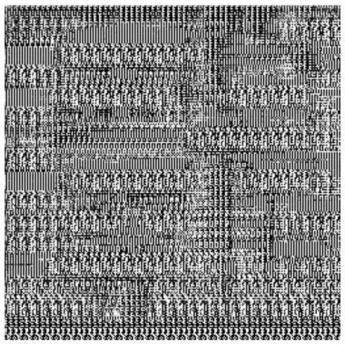


fig 5.2.8- 12500th epoch

As by seeing the images in fig 5.2.6, fig 5.2.7 and fig 5.2.8, conclusion is that there may appear a change in image but was not much change in the quality of image generated at 1th epoch and image generated at 12500th epoch. The image somewhat changed at 500th epoch but the same image was generated consistently till 12500th epoch. Not getting any minute change we stopped training this model.

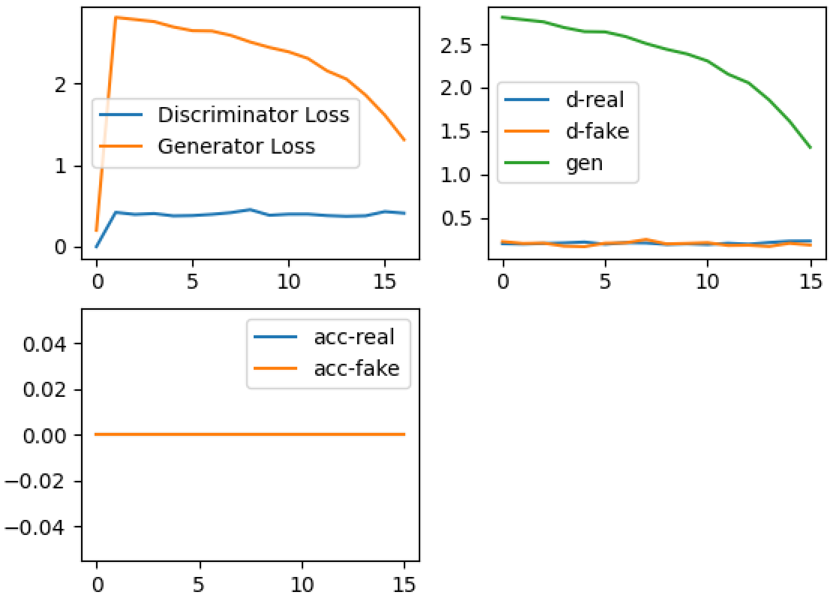


fig 5.2.9epoch 1 to epoch 500

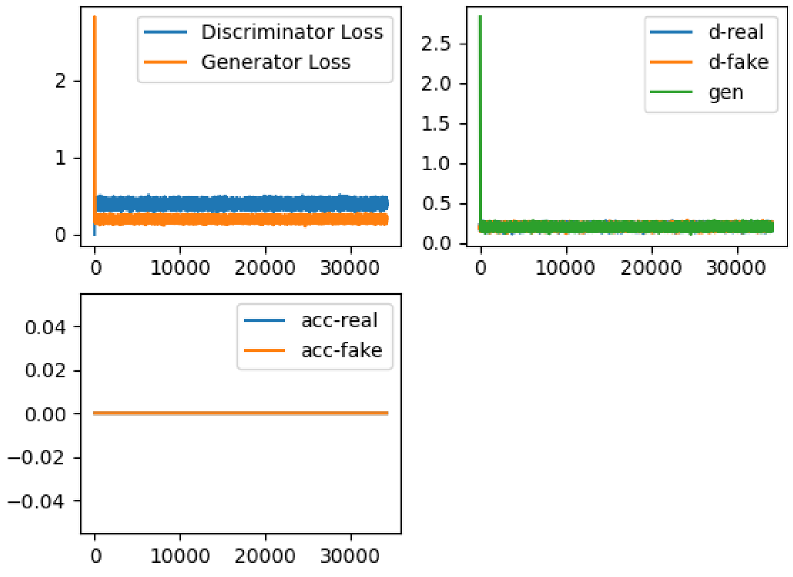


fig 5.2.10- epoch 501 to epoch

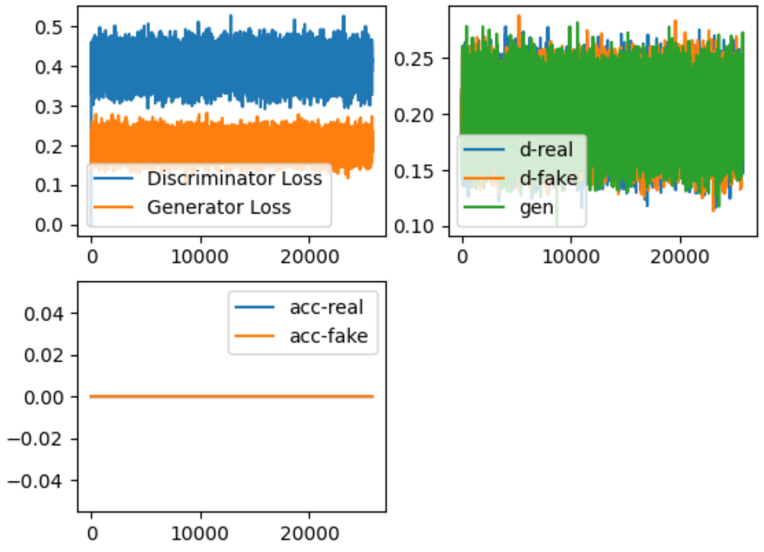


fig 5.2.11- epoch 4501 to epoch 8000

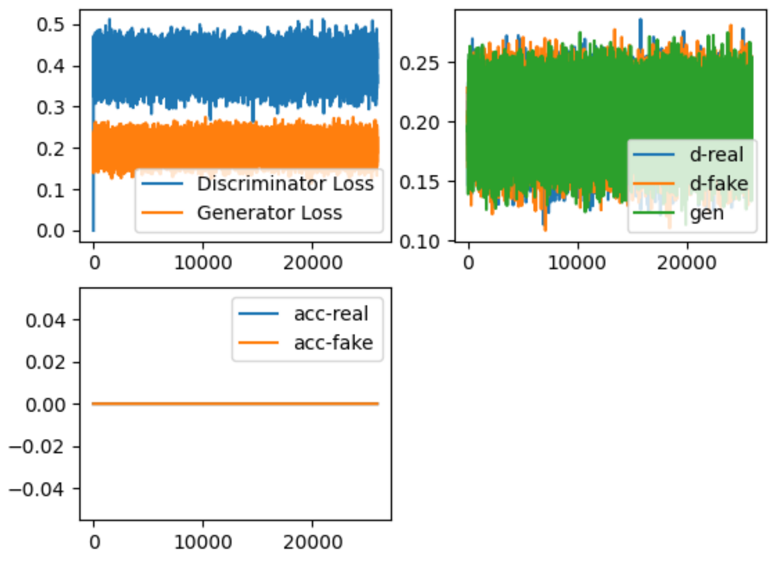


fig 5.2.12- epoch 8001 to epoch 9500epoch

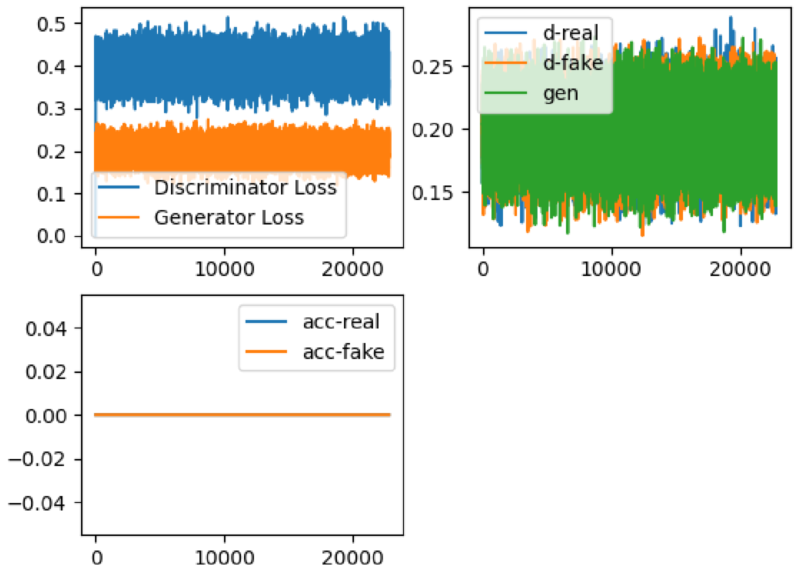


fig 5.2.13- epoch 9501 to epoch 11000

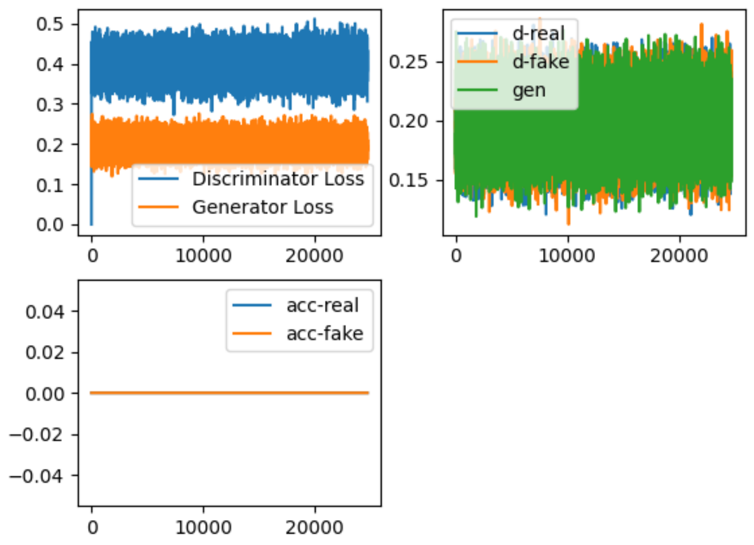


fig 5.2.14- epoch 11001 to epoch 12500epoch

As training was not possible for such large epoch, it was done in 6 batches. As shown in fig 5.2.9, fig 5.2.10, fig 5.2.11, fig 5.2.12, fig 5.2.13, fig 5.2.14 that the loss graph is similar from 500th epoch till 12500th epoch. Conclusion is that this model was also failed to converged and stopped training.

Now, we kept the same Wasserstein GAN architecture we previously used, the only change we made was we applied the DiffAugment method on real and fake images before sending them to discriminator for training using train_on_batch method. The training for this model was done for 2100 epochs

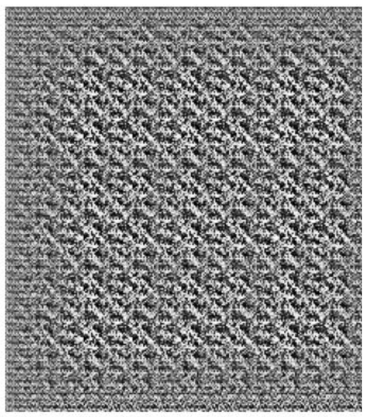


fig 5.2.15- fig p – epoch 100

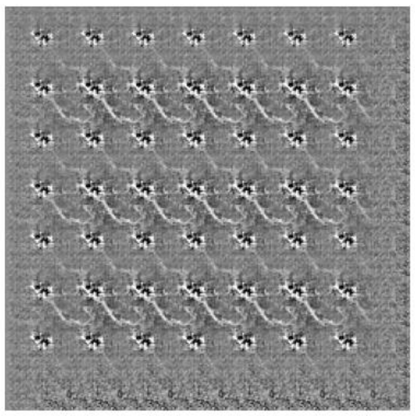


fig 5.2.16- epoch 2100 epoch

As by seeing the images in fig 5.2.15 and fig 5.2.16, conclusion is that there may appear a change in image but was not much change in the quality of image generated at 100th epoch and image generated at 2100th epoch.

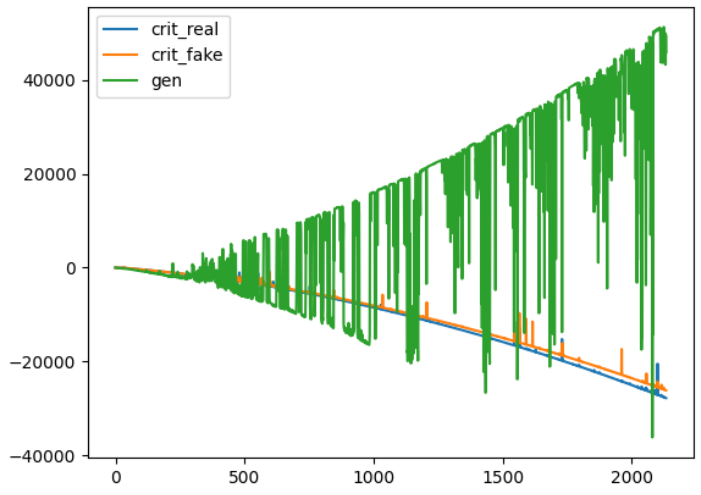


fig 5.2.17- loss plot

By observing the fig 5.2.17 we get that loss of model was behaving erratically, the generator loss was going between approx. 45000 to -40000 and critic loss for real as well as fake images was going to approx. -25000. This is not ideal. So model was stopped training.

As we were not getting expected results from the all the models trained previously, we tried to use a different coding approach to train the GAN. So, we decided to use GradientTape method instead of simply train_on_batch method. The architecture of DCGAN was kept same as starting model but there were some changes. In discriminator we kept all starting layers same but removed las Sigmoid activation function from it. The generator architecture is same, but in both generator and discriminator the optimizer and loss function are applied separately, unlike previous where it was compiled with discriminator and generator networks. Training steps/loop are defined where steps to train the model are given. This model was trained for about 5250 epoches.

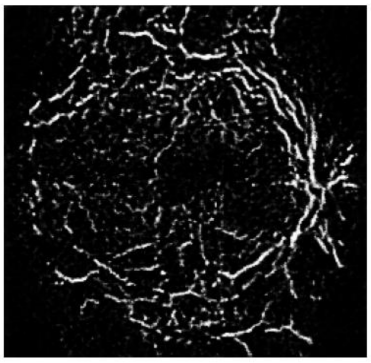


fig 5.2.18- epoch 50

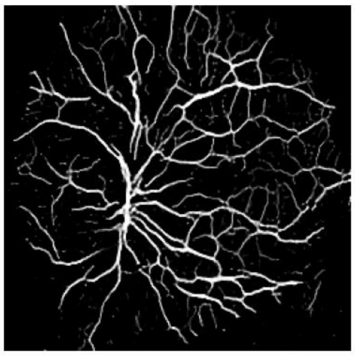


fig 5.2.19- epoch 5250

This model showed much more promise than the previous ones. The generated image at only 50th epoch had started to show the vessel like structures as seen in fig 5.2.18. The image was much more refined at 5250th epoch where it started to resemble the real image.The discriminator loss at 5250 epoch was 0.332 and generator loss was at 3.338. Though this model

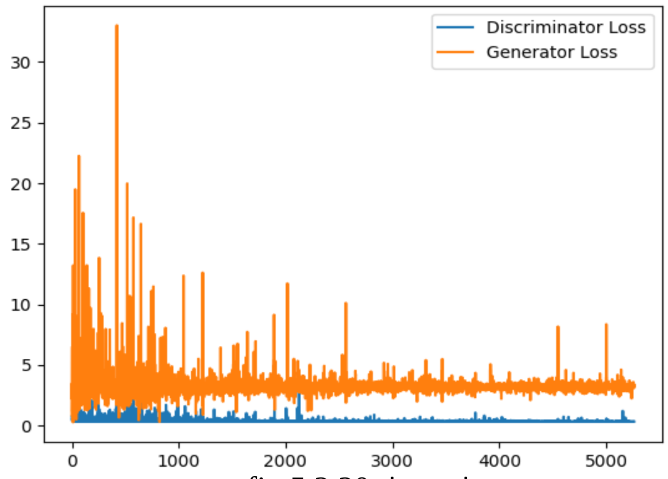


fig 5.2.20- loss plot

generated realistic image it generated same image as shown in fig 5.2.20 continuously i.e., mode collapse has occurred. So, for improving it we decided to use DiffAugment method for this model and create new model to train. The only change we made was we applied the DiffAugment method on real and fake images before sending them to discriminator for training. This model was trained for 3300 epochs.

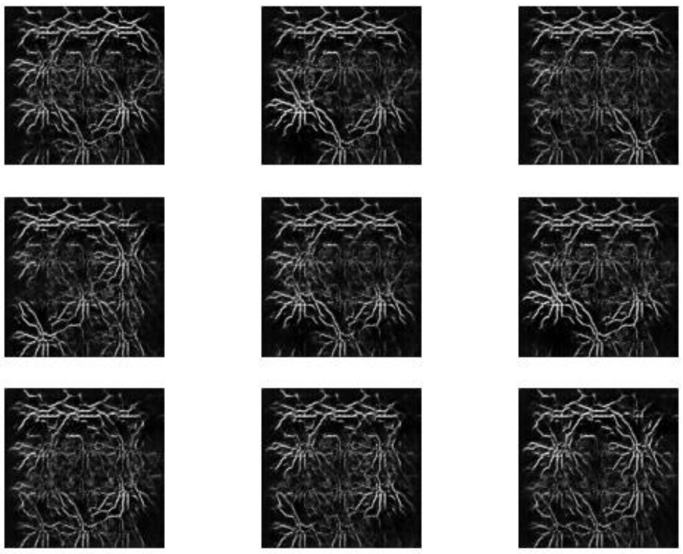


fig 5.2.21- epoch 50

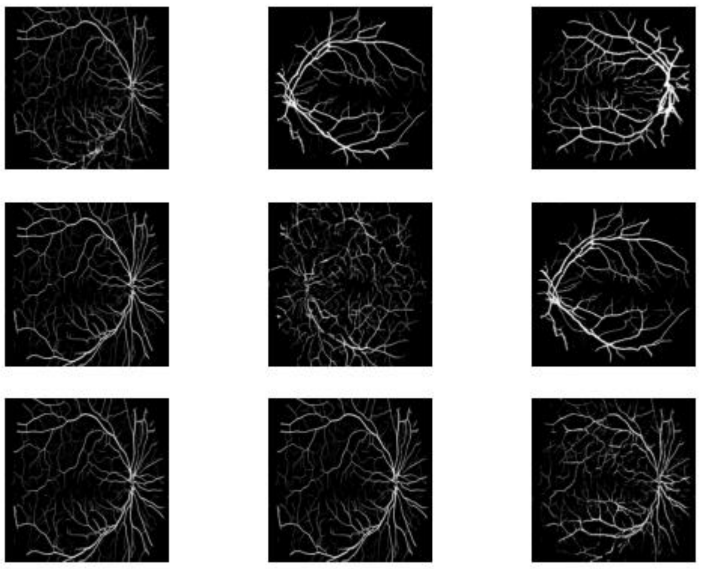


fig 5.2.22- epoch 3300

Fig 5.2.23 shows some realistic images but there are still similar images present. Thus, mode collapse is present here also. The discriminator loss of model at epoch 3300 is 0.415 and generator loss is 3.131.

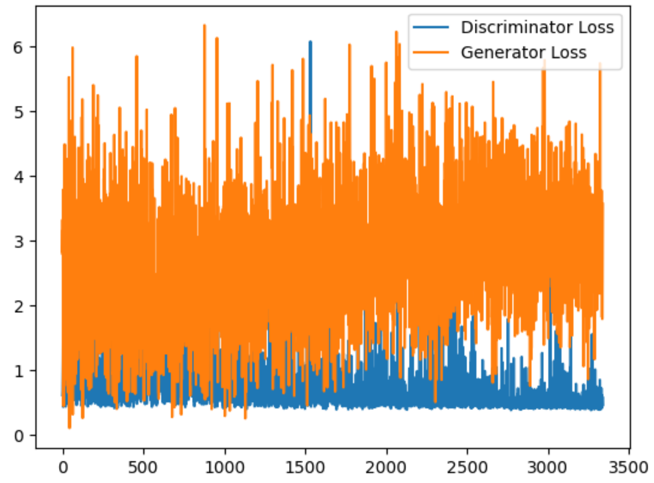


fig 5.2.23- loss plot

So, we decided to train the Wasserstein GAN but using the GradientTape method with DiffAugment method. The architecture was kept same as Wasserstein GAN but, in both generator, and discriminator the optimizer and loss function are applied separately, unlike previous where it was compiled with discriminator and generator networks, also the BatchNormaliztion Layer was removed from the discriminator. Training steps/loop are defined where steps to train the model are given. The model was trained for 3750 epochs.

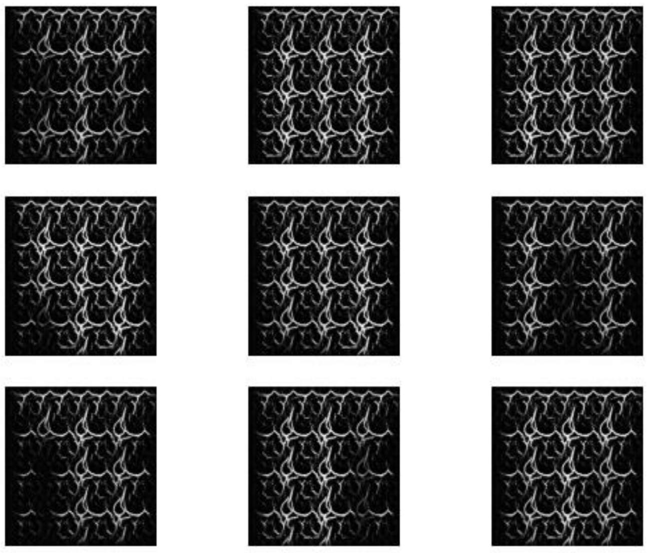


fig 5.2.24- epoch 25

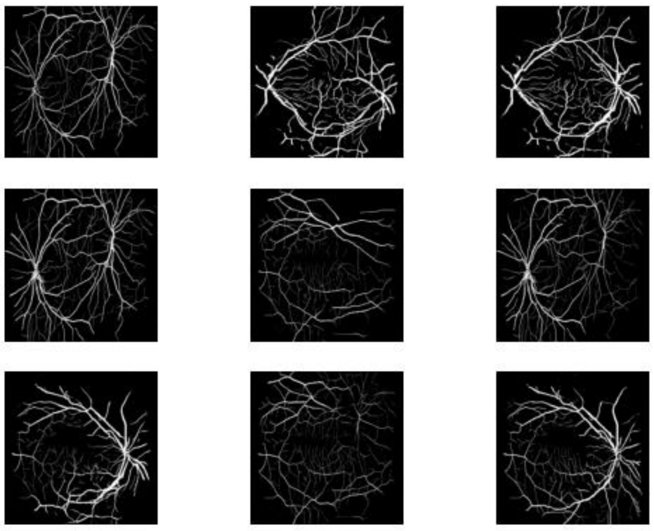


fig 5.2.25- epoch 3750

The mode collapse problem was addressed more here but it was not eliminated as seen in fig 5.2.26.

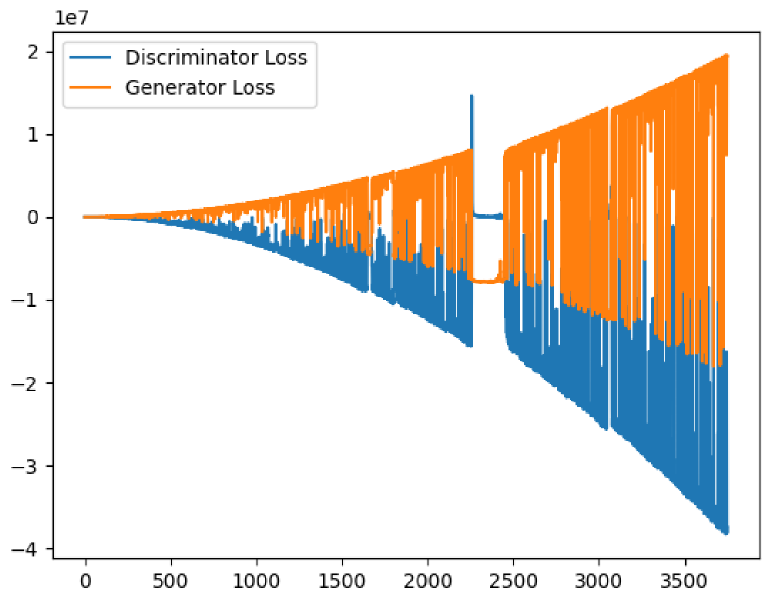


fig 5.2.26- loss plot

We then used the Wasserstein model only but applied gradient penalty to it. In this new model the architecture was kept the same as previous one, but optimizer was changed from RMSprop to Adam optimizer with learning rate of 0.0002.

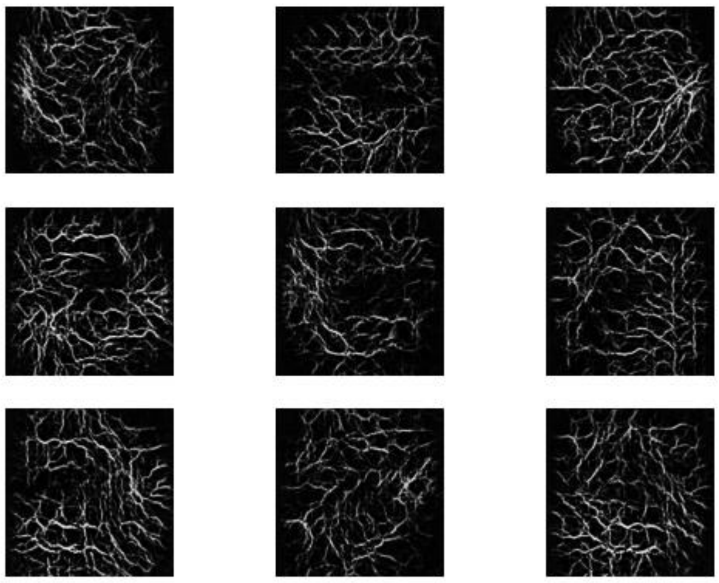


fig 5.2.27- epoch 25

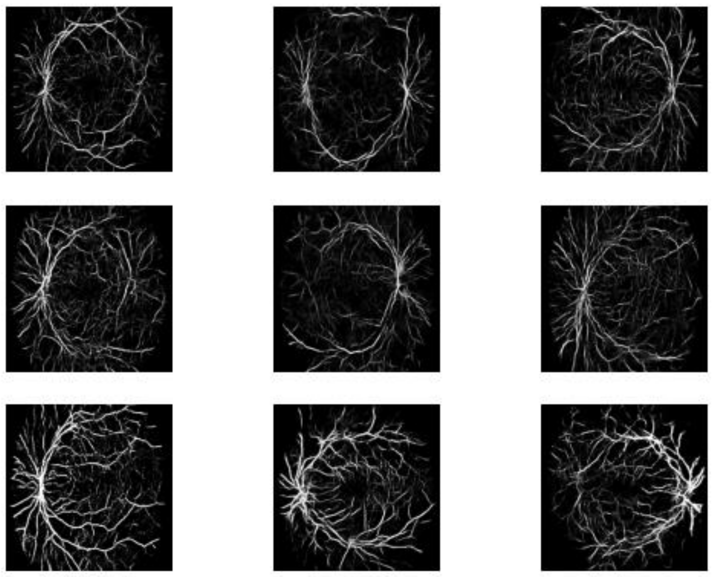


fig 5.2.28- epoch 4800

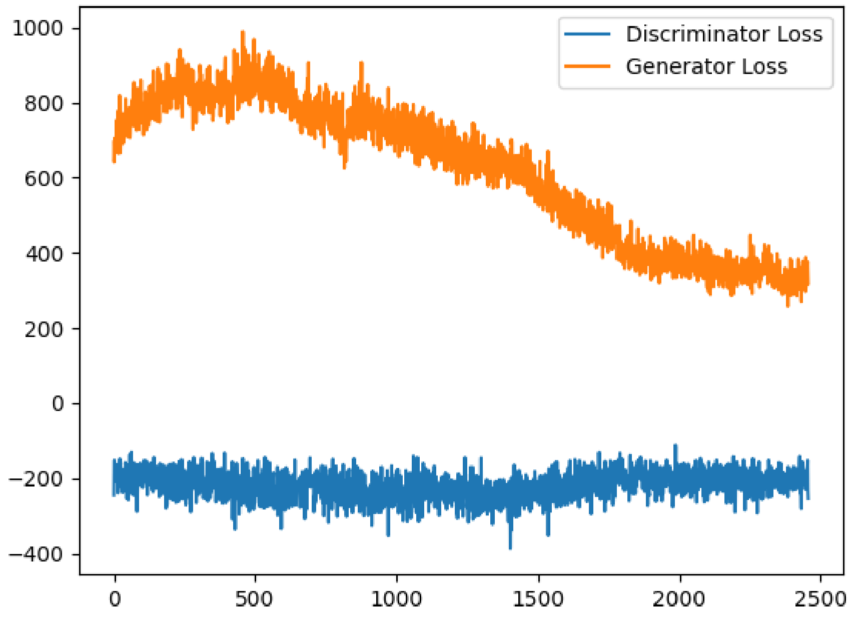


fig 5.2.29- loss plot till 2500

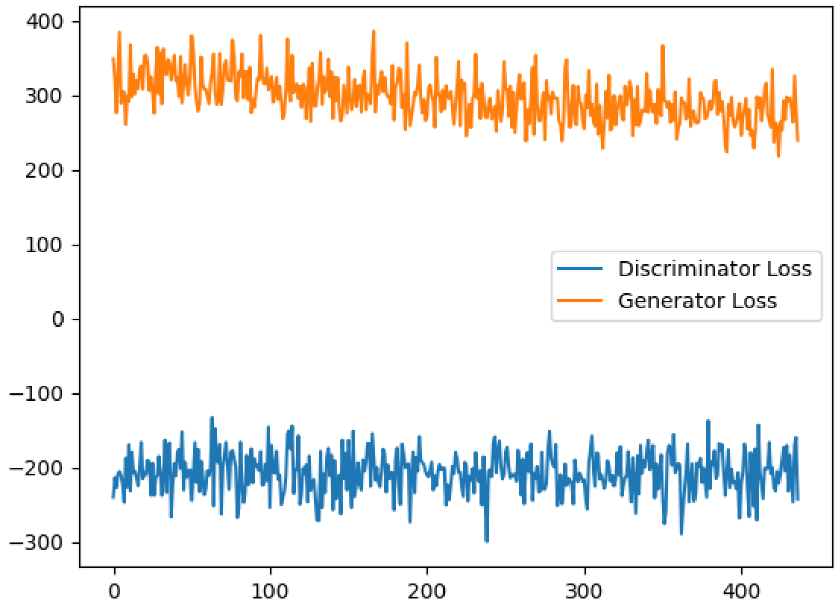


fig 5.2.30- loss plot till 4800epoch

This model was trained in 2 batches the loss plot of these batches is given in fig 5.2.29 and fig 5.2.30. The mode collapse problem was removed as seen in fig b2. But to make the image more refine we decided to apply DiffAugment method to this model and train it.

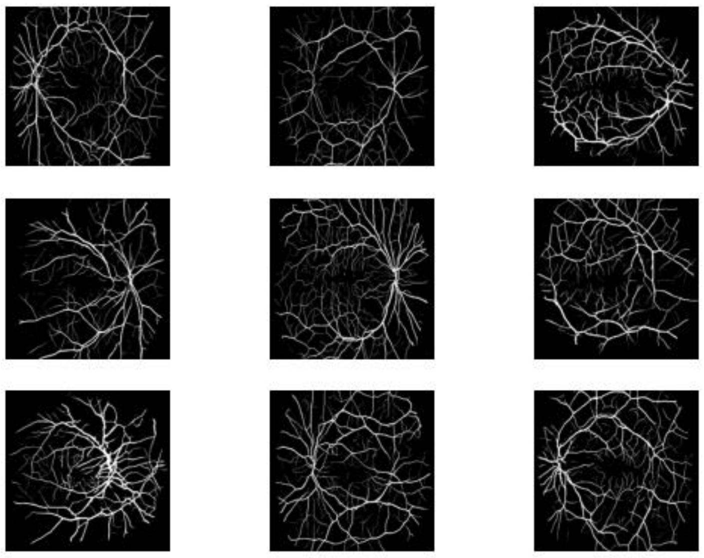


fig 5.2.31- epoch 4200

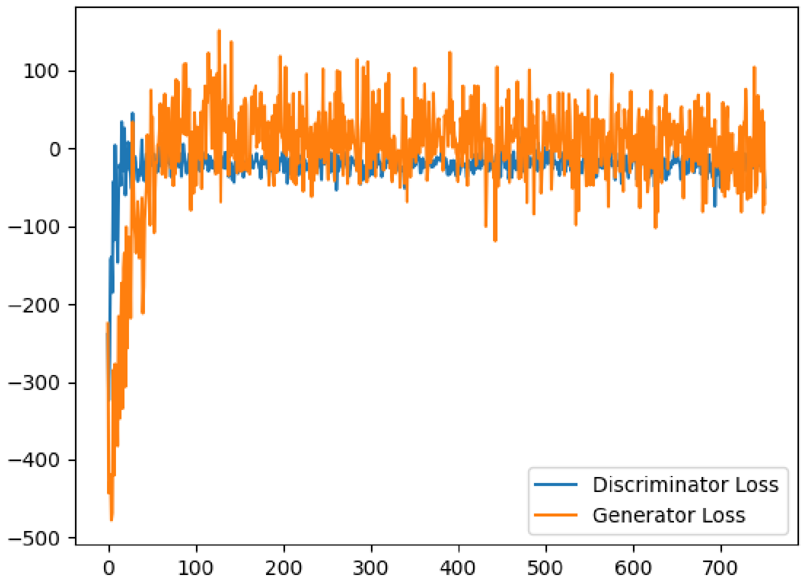


fig 5.2.32- till epoch 700

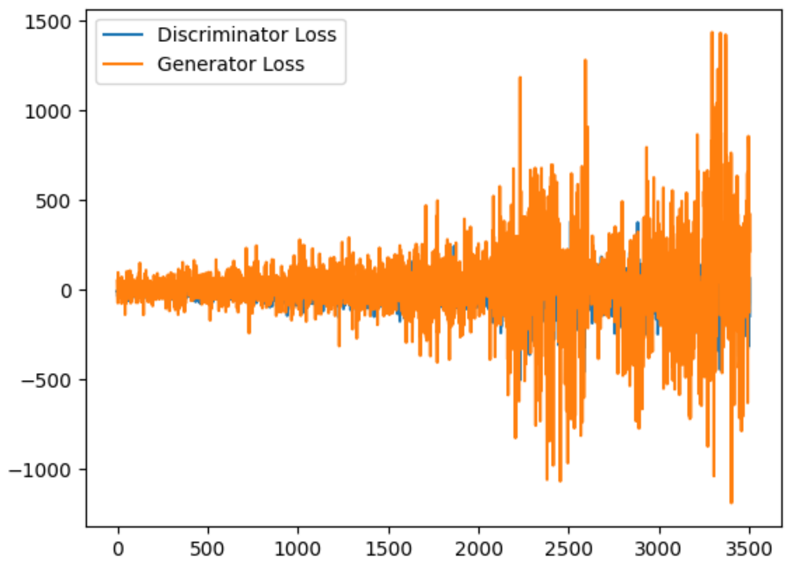


fig 5.2.33- from epoch 701 to epoch 4200

At epoch 4200 discriminator loss is at -67.006 and generator loss -124.066 as seen in fig c3. The images are also more refined than the previous model. The images generated from this model will be used as input for next stage of GAN.

STAGE 2:

At this stage image-to-image translation is performed. Convert the vessel network image to its corresponding eye fundus image. There are 20 vessel network images and 40 fundus images present in the DRIVE dataset. There are 45 vessel network images and 45 corresponding fundus images present in the HRF dataset. To complete this translation first we decided to use cycle GAN. As cycle does not need the paired dataset to do image-to-image translation we gave 260 vessel network images (after flipping) and 340 fundus images after flipping to the GAN for training. Create the GAN model using these Discriminator and Generator Networks. These both networks were trained using the train_on_batch() method. The model was trained for 6500 epochs.

Fig 5.2.34 shows image generated by generator1 which translates images from vessel network to fundus images and fig 5.2.35 shows the image generated by generator2 which translates images from fundus images to vessel network. We need the model of generator1, but as seen in fig 5.2.34 though there is some translation done it is not as expected.

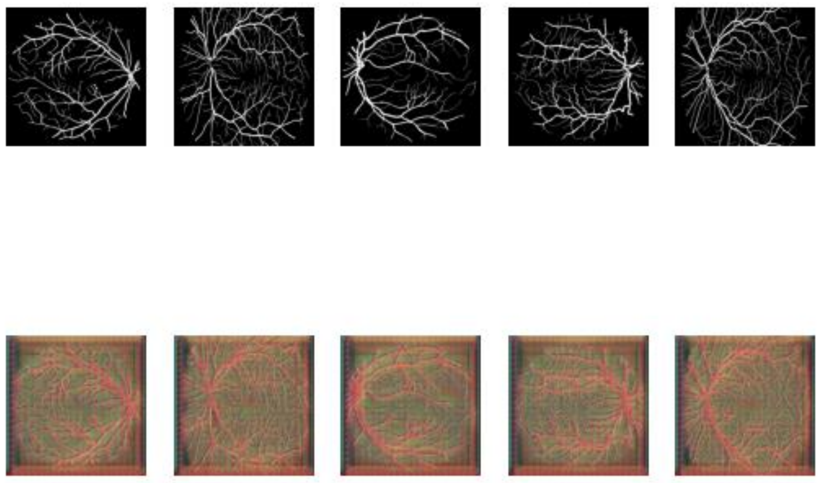


fig 5.2.34- epoch 6500

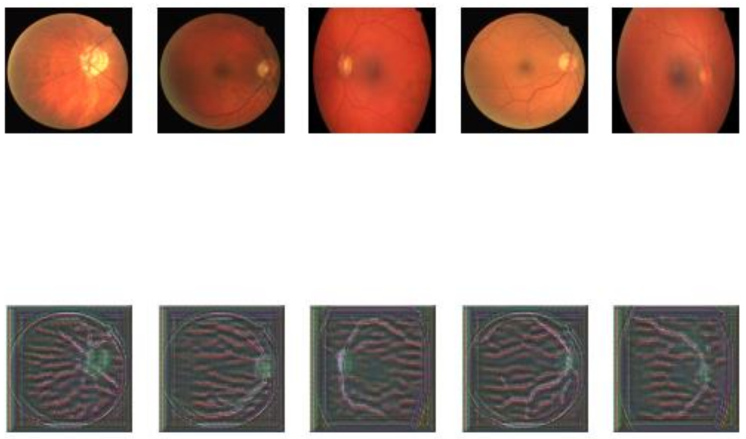


fig 5.2.35- epoch 6500

So, pix2pix GAN is used to perform image-to-image translation. Pix2pix GAN needs the paired image dataset for training the model. So, we used 260 vessel network images (after flipping) and 340 fundus images after flipping to the GAN for training. This creates our Discriminator for GAN. Create the GAN model using these Discriminator and Generator Networks. These both networks were trained using the train_on_batch() method. The model was trained for about 36401 steps.

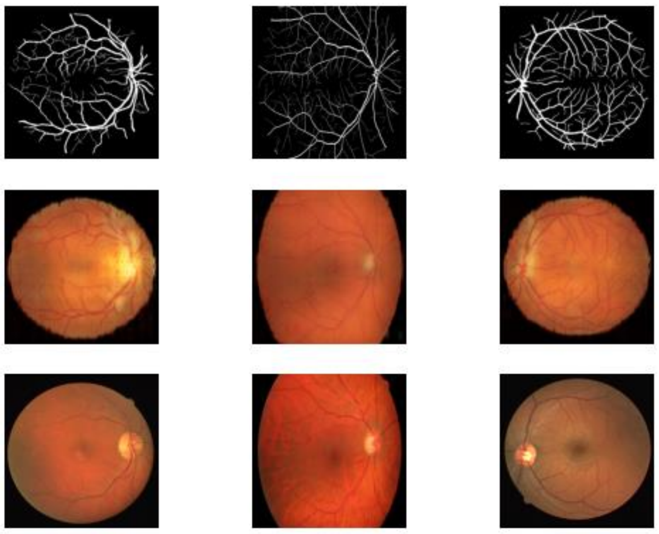


fig 5.2.36- 1300th step

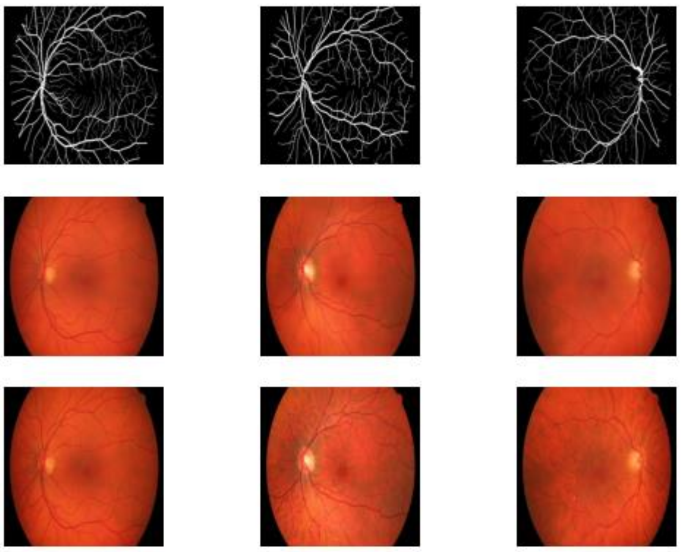


fig 5.2.37- 35100th

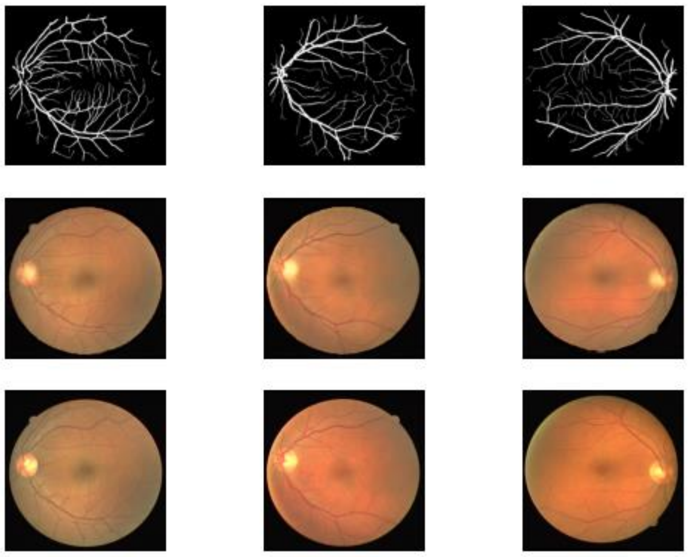


fig 5.2.38- 36400th step

The discriminator loss for real images training is 0.091 and for fake images is 0.082, while the generator loss is 4.180. The model was doing image-to-image translation as expected as seen in fig 5.2.37 and fig 5.2.38. So, this model is be used for stage 2 of the GAN.

6. Testing

6.1 FID

A metric to evaluate the quality of generated images, the Frechet Inception Distance, or FID for short, evolved specifically to evaluate the efficiency of generative adversarial networks. Compared to the current Inception Score, or IS, the suggested score was deemed to be an improvement. Based on how successfully a set of synthetic photos is classified by the top-performing image classification machine Inception v3, the inception score calculates the quality of the set of photographs. How artificial pictures stack up against genuine ones is not represented by the inception score. The purpose of inventing the FID score was to assess synthetic pictures using statistics from a group of synthetic images in comparison to statistics from a group of actual photos from the target domain. Like the inception score, the FID score uses the inception v3 model. Specifically, the coding layer of the model (the last pooling layer prior to the output classification of images) is used to capture computer-vision-specific features of an input image. These activations are calculated for a collection of real and generated images. The activations are summarised as a multivariate Gaussian by calculating the mean and covariance of the images. These statistics are then calculated for the activations across the collection of real and generated images.The distance between these two distributions is then calculated using the Frechet distance, also called the Wasserstein-2 distance. A lower FID indicates better-quality images; conversely, a higher score indicates a lower-quality image and the relationship may be linear.

The FID score is then calculated using the following equation:

d^2 = ||mu_1 – mu_2||^2 + Tr(C_1 + C_2 – 2*sqrt(C_1*C_2))

The feature-wise mean of the real and created pictures is represented by the "mu_1" and "mu_2" variables.

The covariance matrices, or sigma, C_1 and C_2, represent the actual and artificial feature vectors, respectively.

The difference between the two mean vectors is expressed as the sum squared, or ||mu_1 - mu_2||2. The trace linear algebra operation is referred to as Tr.

6.2 SSIM

The term "SSIM" (Structural Similarity Index) refers to a statistic that is frequently used to determine how similar two images are to one another. The Structural Similarity Index determines how similar two provided photos are and returns a result that ranges from -1 to +1. A number of +1 denotes a high degree of similarity or identity between the two provided images, whereas a value of -1 denotes a high degree of contrast between the two images. Frequently, these numbers are changed to fall between [0, 1], where the extremes have the same meaning. This measure takes an image's Luminance, Contrast, and Structure and separates them into their three main components. On the basis of these 3 attributes, a comparison between the two photos is made. The term "luminance" describes an image's general brightness or intensity. Averaging across all of the pixel data yields the luminance value. Contrast is a measurement of the difference in brightness between several areas of a picture. The standard deviation of each pixel's value is taken to determine the contrast. The organisation and layout of elements in a picture are represented by structure. By evaluating the correlation of regions of image in the produced and reference pictures, SSIM determines how structurally similar two images are. It evaluates how effectively the reference picture's structural information and connections are retained in the created image.

SSIM(x, y) = [l(x, y)]^α * [c(x, y)]^β * [s(x, y)]^γ

Where:

x and y represent the two images being compared.

l(x, y) is the luminance comparison term, which measures the similarity of the mean intensities of x and y.

c(x, y) is the contrast comparison term, which captures the similarity of the standard deviations of x and y.

s(x, y) is the structure comparison term, which measures the similarity of the covariance of x and y.

α, β, and γ are weighting factors that adjust the relative importance of each term. These factors are typically positive values, and their sum is usually set to 1.

The computation of SSIM is not a difficult task since SSIM is a built-in method part of Sci-Kit's image library so we can just load it up and compute.

6.3 KL Divergence

An analytical tool for calculating the distance between two probability distributions is the Kullback-Leibler (KL) divergence (also known as the KL) notion. KL Divergence may be utilised in the context of retinal image synthesis using GANs to evaluate the similarity or disagreement between the distributions of produced and real retinal pictures.

The formula for KL Divergence between two probability distributions P and Q is as follows:

$$KL(P \| Q) = \sum(P(x) * \log(P(x) / Q(x)))$$

where each component of the distributions P and Q is represented by a distinct letter, x. The probabilities given to each element x by the distributions P and Q are denoted as P(x) and Q(x), respectively.

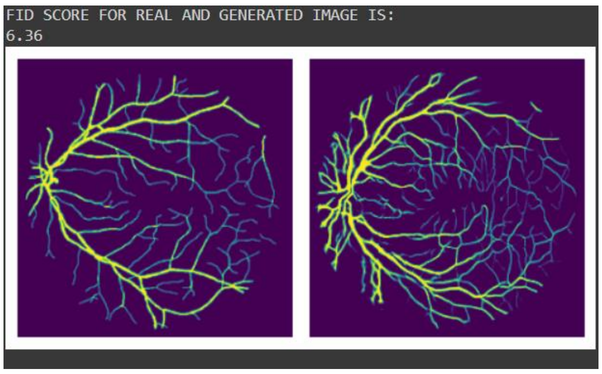
The produced retinal pictures are viewed as one distribution in the context of retinal image synthesis, whilst the genuine retinal images constitute another distribution. The degree to which the produced images closely resemble the distribution of real photos may be determined by calculating the KL Divergence between these two distributions.

The closer the distributions match, the lower the KL Divergence value, suggesting that the produced images are more statistically comparable to the genuine images. A higher KL Divergence, on the other hand, denotes a greater difference between the distributions, suggesting that the produced pictures are different.

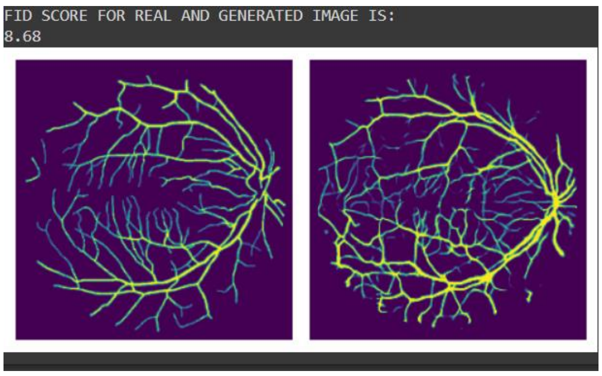
7. Performance Analysis

SSIM, KL Divergence and FID are the three metrics used to see the performance of the images.

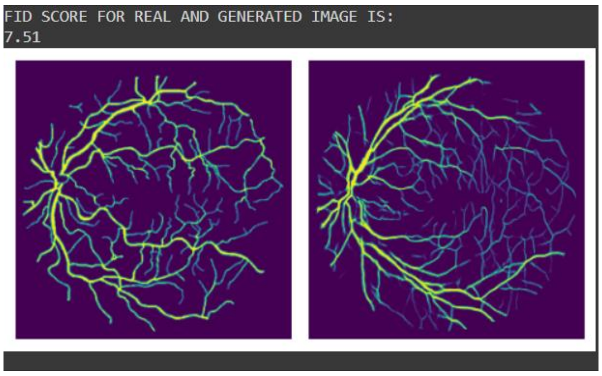
The FID value of the images are : 7.93 ± 1.26



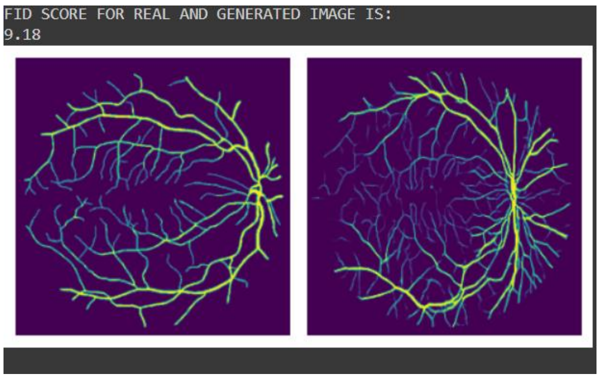
```
FID SCORE FOR REAL AND GENERATED IMAGE IS:
6.36
```



```
FID SCORE FOR REAL AND GENERATED IMAGE IS:
8.68
```

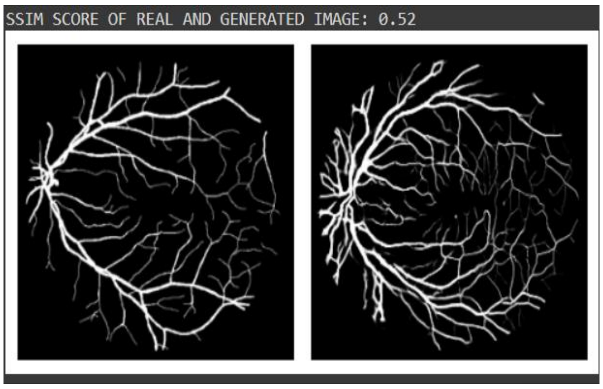


```
FID SCORE FOR REAL AND GENERATED IMAGE IS:
7.51
```

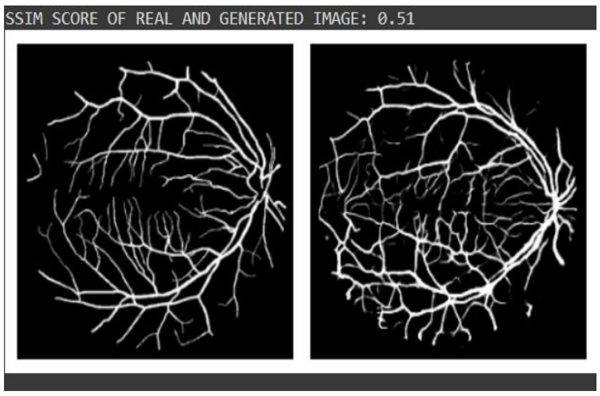


```
FID SCORE FOR REAL AND GENERATED IMAGE IS:
9.18
```

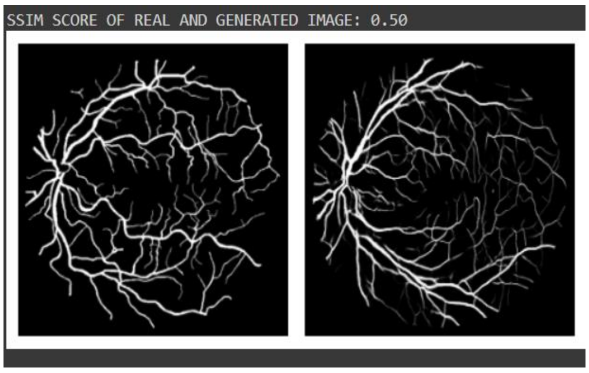
The SSIM value of the images are : 0.5125 ± 0.0096



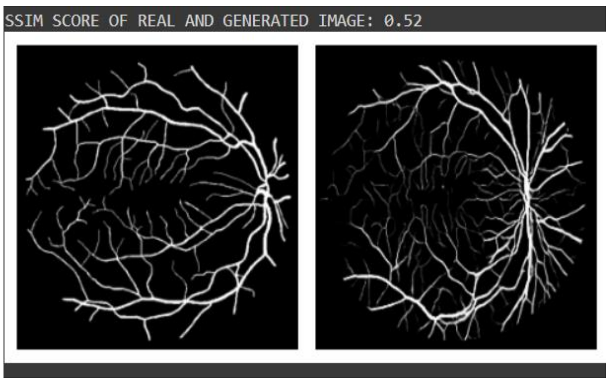
```
SSIM SCORE OF REAL AND GENERATED IMAGE: 0.52
```



```
SSIM SCORE OF REAL AND GENERATED IMAGE: 0.51
```

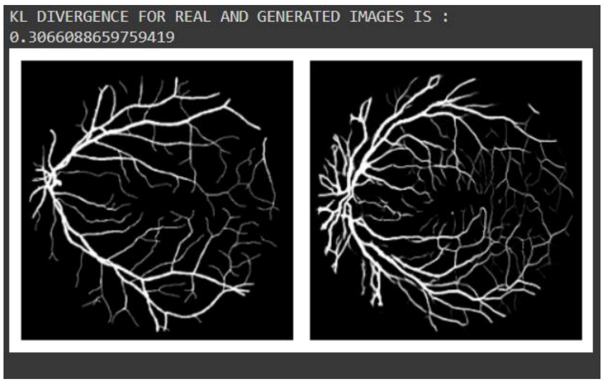


```
SSIM SCORE OF REAL AND GENERATED IMAGE: 0.50
```

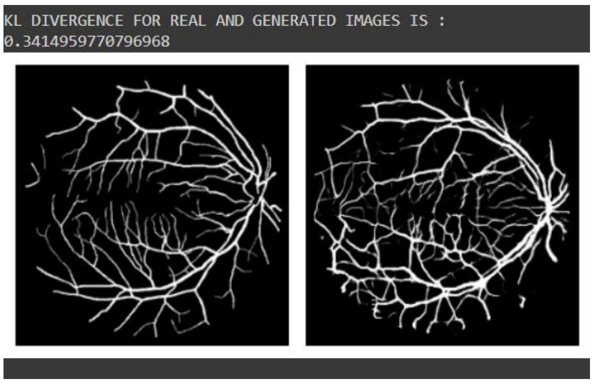


```
SSIM SCORE OF REAL AND GENERATED IMAGE: 0.52
```

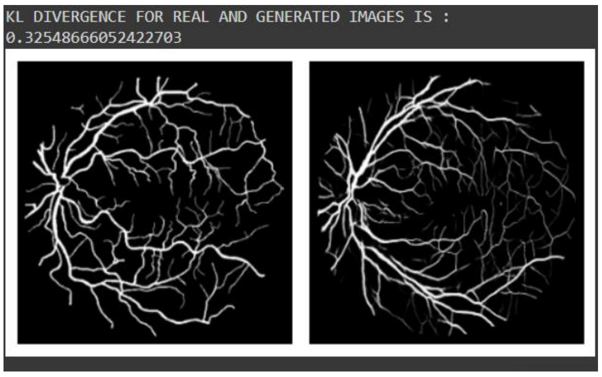
The KL Divergence value of the images are : 0.3233 ± 0.0002



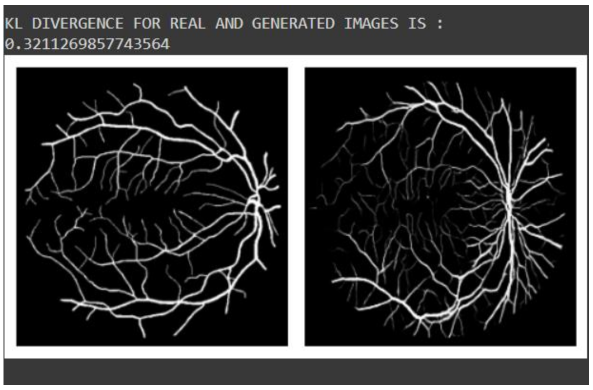
```
KL DIVERGENCE FOR REAL AND GENERATED IMAGES IS :
0.3066088659759419
```



```
KL DIVERGENCE FOR REAL AND GENERATED IMAGES IS :
0.3414959770796968
```



```
KL DIVERGENCE FOR REAL AND GENERATED IMAGES IS :
0.32548666052422703
```



```
KL DIVERGENCE FOR REAL AND GENERATED IMAGES IS :
0.3211269857743564
```

The checking the above given values of the three metrics we can conclude that our images are different from the source dataset. This tells us that our model gives us diverse retinal images.

8. Future Scope

Research in the rapidly developing field of retinal image synthesis has the potential to completely alter how various eye disorders are identified and treated. Retinal image synthesis is now much more accurate because to advances in deep learning algorithms, making it an exciting new field of study.

Here are some potential future directions for retinal image synthesis:

Improved accuracy: While the current state-of-the-art models have shown remarkable accuracy in generating synthetic retinal images, there is still room for improvement. Researchers are exploring novel approaches to enhance the accuracy of synthesized images.

Real-time retinal imaging: With the development of high-performance computing technologies, it is becoming feasible to generate synthetic retinal images in real-time. This could be especially useful in situations where time is of the essence, such as emergency room visits or telemedicine consultations.

Automated diagnosis: Synthesized retinal images could also be used to train deep learning algorithms to accurately diagnose and classify various eye diseases. This could potentially lead to faster and more accurate diagnosis, which is crucial for managing conditions like diabetic retinopathy and macular degeneration.

Overall, the future of retinal image synthesis looks promising, and continued research in this field could have significant implications for the diagnosis and treatment of various eye diseases.

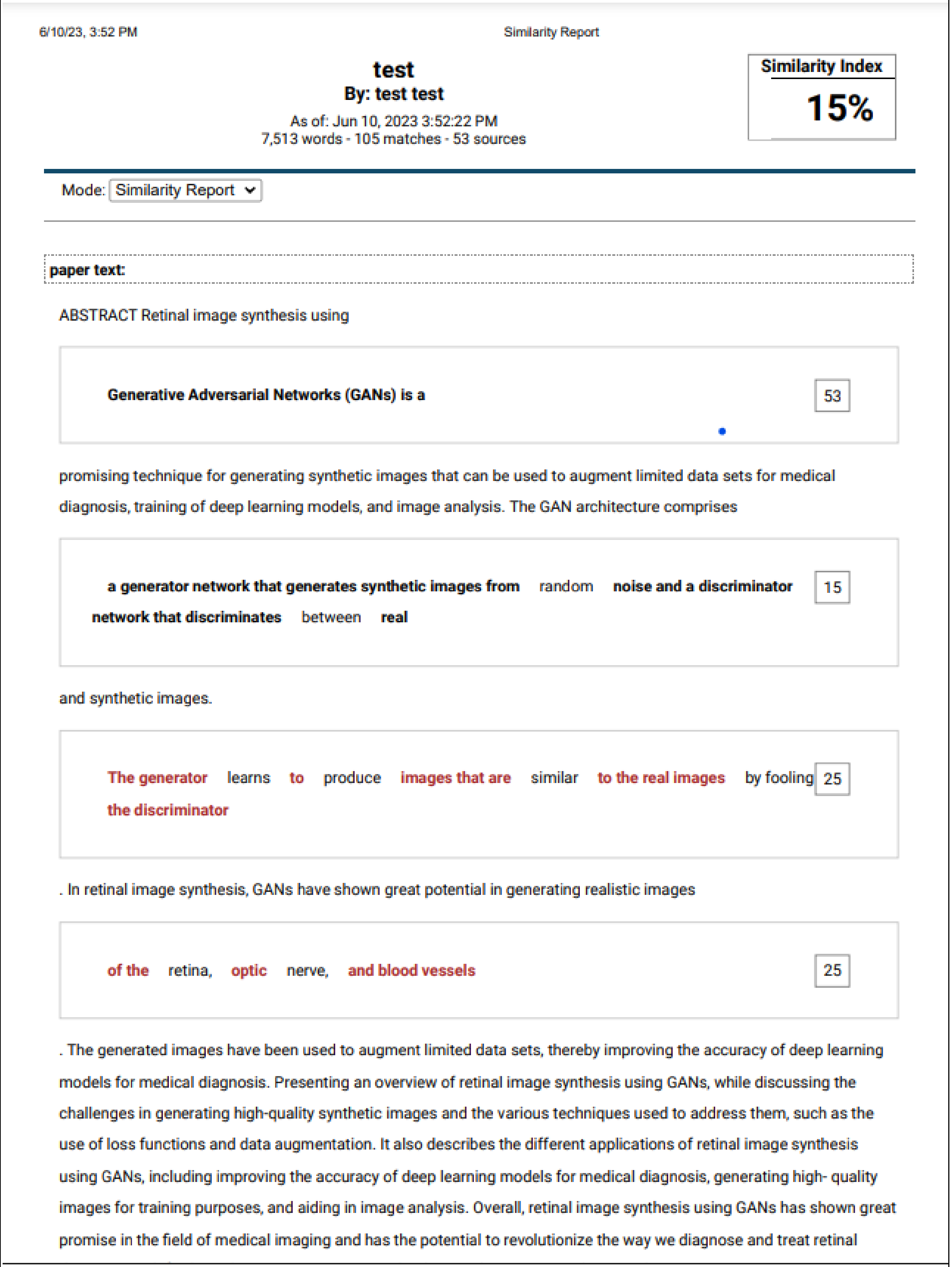
9. Applications

1. Data Augmentation for Disease Detection: Retinal image synthesis can be used to generate new synthetic images of retinas with varying degrees of disease. This can be used to augment the training dataset of deep learning-based disease detection algorithms, enabling them to learn the different features and patterns associated with various stages of the disease.

2. Training Medical Professionals: Retinal image synthesis can also be used to train medical students and professionals in the diagnosis of various retinal diseases. Synthetic retinal images of different disease stages can be used to demonstrate the changes and symptoms associated with each stage.

3. Automated Diagnosis: Retinal image synthesis can be used to create artificially augmented datasets for automated diagnosis systems. This can assist researchers to improve upon the accuracy of machine-learning models for diagnosis of retinal diseases.

10.Plagiarism Report



6/10/23, 3:52 PM Similarity Report

test
By: test test
As of: Jun 10, 2023 3:52:22 PM
7,513 words - 105 matches - 53 sources

Similarity Index
15%

Mode: Similarity Report

paper text:

ABSTRACT Retinal image synthesis using

Generative Adversarial Networks (GANs) is a 53

promising technique for generating synthetic images that can be used to augment limited data sets for medical diagnosis, training of deep learning models, and image analysis. The GAN architecture comprises

a generator network that generates synthetic images from random **noise and a discriminator network that discriminates** between **real** 15

and synthetic images.

The generator learns **to** produce **images that are** similar **to the real images** by fooling **the discriminator** 25

. In retinal image synthesis, GANs have shown great potential in generating realistic images

of the retina, **optic** nerve, **and blood vessels** 25

. The generated images have been used to augment limited data sets, thereby improving the accuracy of deep learning models for medical diagnosis. Presenting an overview of retinal image synthesis using GANs, while discussing the challenges in generating high-quality synthetic images and the various techniques used to address them, such as the use of loss functions and data augmentation. It also describes the different applications of retinal image synthesis using GANs, including improving the accuracy of deep learning models for medical diagnosis, generating high- quality images for training purposes, and aiding in image analysis. Overall, retinal image synthesis using GANs has shown great promise in the field of medical imaging and has the potential to revolutionize the way we diagnose and treat retinal

11.References

[1] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley,Sherjil Ozair, Aaron Courville, Yoshua Bengio, "Generative Adversarial Nets", Advances in Neural Information Processing Systems (NIPS), DOI: 10.1145/3422622, June 2014.

[2] Antonia Creswell, Tom White, Vincent Dumoulin, Kai Arulkumaran, Biswa Sengupta, Anil Bharath, "Generative Adversarial Networks: An Overview", IEEE Signal Processing Magazine, vol 35, issue 1, pp 53-65, Jan 2018.

[3] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, Xi Chen, "Improved Techniques for Training GANs", DOI: arXiv:1606.03498, Jun 2016.

[4] M.Krithika alias AnbuDevi, Dr.K.Suganthi, "Review of Medical Image Synthesis using GAN Techniques", ITM Web of Conferences, vol. 37, article no. 01005, Mar. 2021.

[5] Valentina Bellemo, Philippe Burlina, Liu Yong, Tien Yin Wong, Daniel Shu Wei Ting, "Generative Adversarial Networks (GANs) for Retinal Fundus Image Synthesis", Computer Vision – ACCV 2018 Workshops book, pp 289-302, DOI: 10.1007/978-3-030-21074-8_24, Jun 2019.

[6] Pedro Costa, Adrian Galdran, Mania Ines Meyer, Michael David Abramoff, Meindert Niemeijier, Ana Maria Mendonca, Aurelio Campilho, "Towards Adversarial Retinal Image Synthesis", DOI: arXiv:1701.08974, Jan 2017.

[7] Pedro Costa, Adrian Galdran, Maria Ines Meyer, Meindert Niemeijer, Michael Abràmoff, Ana Maria Mendonça, and Aurélio Campilho, "End-to-End Adversarial Retinal Image Synthesis", IEEE Transactions on Medical Imaging, vol. 37, no. 3, pp. 781-791, Mar 2018.

[8] John T. Guibas, Tejpal S. Virdi, Peter S. Li, "Synthetic Medical Images from Dual Generative Adversarial Networks", DOI: arXiv:1709.01872v3, Jan 2018.

[9] Martin Arjovsky, Soumith Chintala2, and Leon Bottou, "Wasserstein GAN", DOI: arXiv:1701.07875v3, Dec 2017.

[10] Shengyu Zhao, Zhijian Liu, Ji Lin, Jun-Yan Zhu and Song Han, “Differentiable Augmentation for Data-Efficient GAN Training”, DOI: arXiv:2006.10738v4, Dec 2020.