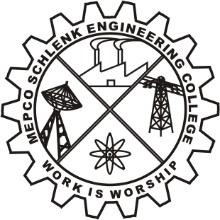
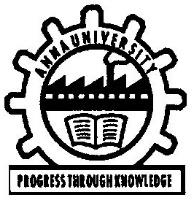
**AI-DRIVEN FRAMEWORK FOR SECURE TRANSMISSION OF**

**MEDICAL IMAGES**



### MINI PROJECT REPORT

***Submitted by***

### DHARSHINI R (9517202109014)

### SANDHIYA DEVI B (9517202109046)

### SHRI SUBIKSHA S T (9517202109048)

***in***

#### 19AD651 –DEEP LEARNING LABORATORY

**DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE**

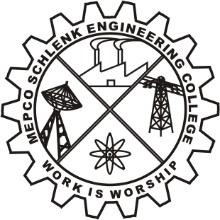
**MEPCO SCHLENK ENGINEERING COLLEGE**

**SIVAKASI**

**APRIL 2024**

**MEPCO SCHLENK ENGINEERING COLLEGE, SIVAKASI AUTONOMOUS**

**DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE**



**BONAFIDE CERTIFICATE**

This is to certify that it is the bonafide work of **DHARSHINI R (9517202109014), SANDHIYA DEVI B (9517202109046), SHRI SUBIKSHA S**

**T(9517202109048)** for the mini project titled **“AI-DRIVEN FRAMEWORK FOR SECURE TRANSMISSION OF MEDICAL IMAGES”** in 19AD651 –**DEEP LEARNING LABORATORY** during the sixth semester December 2023 –April 2024 under my supervision.

**SIGNATURE SIGNATURE**

#### Mrs.L.Prasika,M.E,(Ph.D) Dr. J. Angela Jennifa Sujana, M.E.,Ph.D

**Asst. Professor, Professor &Head,**

AI&DSDepartment, AI&DSDepartment,

Mepco Schlenk Engg.College,Sivakasi Mepco Schlenk Engg. College,Sivakasi.

# ABSTRACT

The need for medical image encryption is increasingly pronounced in today's digital healthcare landscape, driven by the imperative to safeguard the privacy and confidentiality of patients' medical imaging data. Medical images contain sensitive information about patients' conditions, diagnoses, and treatments, making them highly valuable and vulnerable targets for unauthorized access or breaches. Therefore, robust encryption methods are essential to protect these images from unauthorized viewing, tampering, or exploitation.In response to this need, this article introduces a novel deep learning-based approach called DeepKeyGen for private key generation in medical image encryption. DeepKeyGen leverages the power of Generative Adversarial Networks (GANs) as the learning network to generate the private key. GANs are well-suited for this task as they excel in learning complex data distributions and generating realistic data samples. One of the key innovations of DeepKeyGen is the concept of the transformation domain, which represents the desired "style" of the private key to be generated. By incorporating this transformation domain into the learning process, DeepKeyGen can effectively guide the network to learn the mapping relationship between the initial image and the private key. This ensures that the generated private keys possess the desired characteristics and security properties.The primary goal of DeepKeyGen is to learn this mapping relationship in such a way that the generated private keys exhibit high randomness, unpredictability, and cryptographic strength. These properties are crucial for ensuring the security of the encryption scheme and protecting the confidentiality of medical images.To evaluate the effectiveness and security of DeepKeyGen, extensive experiments are conducted using three diverse datasets: the Montgomery County chest X-ray dataset, the Ultrasonic Brachial Plexus dataset, and the BraTS18 dataset. The evaluation findings demonstrate that DeepKeyGen can achieve a high level of security in generating private keys, making it suitable for use in medical image encryption applications.Furthermore, a comprehensive security analysis is performed to assess DeepKeyGen's resilience against various known attacks and vulnerabilities. The results of the security analysis confirm the robustness and effectiveness of the proposed key generation network in ensuring the confidentiality and integrity of medical images.Overall, DeepKeyGen represents a significant advancement in the field of medical image encryption, offering a powerful and versatile solution for generating private keys with high security guarantees. Its effectiveness, scalability, and applicability make it a promising tool for enhancing privacy protection in healthcare settings and mitigating the risks associated with unauthorized access to medical imaging data.

# TABLE OF CONTENTS

|  |  |  |  |
| --- | --- | --- | --- |
| **CHAPTERNO** | **TITLE** | | **PAGENO** |
|  | **ABSTRACT** | |  |
| 1 | **INTRODUCTION** | |  |
|  | 1.1 | Overview |  |
|  | 1.2 | Problem Statement |  |
| 2 | **SYSTEM REQUIREMENTS** | |  |
| 3 | **SYSTEM DESIGN** | |  |
|  | 3.1 | Model Architecture |  |
|  | 3.2 | Dataset |  |
|  | 3.3 | Modules Description |  |
|  | 3.4 | Models Description |  |
| 4 | **IMPLEMENTATION** | |  |
|  | 4.1 | Example Explanation |  |
|  | 4.2 | Source Code |  |
| 5 | **RESULTS** | |  |
| 6 | **CONCLUSION** | |  |
| 7 | **REFERENCES** | |  |

**ACKNOWLEDGEMENT**

First and foremost we praise and thank **“The Almighty”,** the lord of all creations, who by his abundant grace has sustained us and helped us to work on this project successfully.

We really find unique pleasure and immense gratitude in thanking our respected management members**,** who is the backbone of our college.

A deep bouquet of thanks to respected Principal **Dr.S.Arivazhagan M.E.,Ph.D.,** for having provided the facilities required for our mini project.

We sincerely thank our Head of the Department **Dr. J. Angela Jennifa SujanaM.E.,Ph.D.,** Associate Professor(SG) & Head, Department of Artificial Intelligence and Data Science, for her guidance and support throughout the mini project .

We also thank our guide **Mrs.L.Prasika M.Tech,** **(Ph.D)** Assistant Professor, Department of Artificial Intelligence and Data Science for their valuable guidance and it is great privilege to express our gratitude to them.

We extremely thank our project coordinator **Mrs.M Priyadharshini M.Tech,** Assistant Professor of Artificial Intelligence and Data Science, who inspired us and supported us throughout the mini project.

We extend our heartfelt thanks and profound gratitude to all the faculty members of Artificial Intelligence and Data Science department for their kind help during our mini project work.

We also thank our parents and our friends who had been providing us with constant support during the course of the mini project work.

**CHAPTER 1**

**INTRODUCTION**

* 1. **OVERVIEW**

### The increasing use of medical imaging in diagnosis and treatment plans necessitates the protection of sensitive patient information within these images. Leakage of medical images can have severe privacy implications for patients and legal ramifications for hospitals. Cryptographic techniques, like stream ciphers, are employed to secure medical images during storage and transmission. However, designing a secure stream cipher generator to produce random and unpredictable sequences is a challenge. Existing approaches often involve manually designed private key generators, which can be time-consuming, expensive, and require adjustments to achieve the desired security level.

### This project proposes DeepKeyGen, a novel deep learning-based method for private key generation, specifically designed for medical image encryption. DeepKeyGen leverages the concept of image-to-image translation using Generative Adversarial Networks (GANs).Stream ciphers are popular for medical image encryption due to their speed, low error expansion, and better synchronization compared to block ciphers. However, designing a secure stream cipher generator is a challenge. Traditional methods rely on manually designed generators, like linear feedback shift registers (LFSRs) or chaotic systems. These methods require manual adjustments during the design and implementation phases to achieve a specific security level. This process is time-consuming, expensive, and may not always achieve the optimal performance.

### DeepKeyGen addresses the limitations of traditional methods by proposing a deep learning approach to private key generation. It utilizes GANs, a type of deep learning architecture consisting of two neural networks: a generator and a discriminator. The generator's goal is to produce data (in this case, private keys) that resemble the target data distribution (the desired key style). The discriminator aims to distinguish between real data (actual private keys) and the data generated by the generator. Through an iterative training process, the generator and discriminator compete, with the generator improving its ability to create realistic private keys, and the discriminator becoming better at identifying forgeries.

### 

### 

### Fig.1.1.1: The Encryption-Decryption Process of Medical Image

### DeepKeyGen utilizes two image domains,

### Source Domain: This domain can include any images with a consistent distribution. An initial image from this domain acts as the "seed" for generating the private key.

### Transformation Domain: This domain represents the desired "style" of the private key, such as a chaotic private key with a certain level of security.

### During training, the generator transforms images from the source domain to the transformation domain. The output of the generator is considered the private key. The discriminator attempts to differentiate between the generated keys and actual keys from the transformation domain. The inherent randomness within deep learning models, due to factors like a large number of parameters and complex network structures, contributes to the generation of unique private keys even under identical training conditions. This characteristic makes DeepKeyGen similar to a one-time pad encryption system, where the key is used only once for maximum security.

### An additional advantage of DeepKeyGen is its ability to train using unlabeled and unpaired images, overcoming limitations associated with data availability issues commonly faced in GAN training.

### Applications and Security Analysis

### Figure 1 depicts a potential application scenario for DeepKeyGen in a healthcare setting. Upon receiving a medical image, a key generation server utilizes DeepKeyGen to generate a private key. This key is then employed with an encryption algorithm to encrypt the medical image, resulting in a ciphertext (encrypted image). Both the ciphertext and the private key are transferred securely and stored within a Picture Archiving and Communication Systems (PACS) server. Authorized healthcare professionals can retrieve the encrypted image and corresponding key from the PACS server for decryption using a decryption algorithm. The decrypted image (original medical image) is then securely sent to the healthcare professional's workstation for viewing.

### Several Security Advantages

### High Randomness and Large Key Space: The randomness inherent in deep learning models contributes to the generation of highly random private keys. Additionally, the vast parameter space of deep learning models translates to a large key space, making it computationally infeasible to guess the key.

### Sensitivity to Changes: Even minor modifications to the training process or seed image can significantly alter the generated private key, enhancing security.

### Resistance to Attacks: DeepKeyGen is claimed to be resilient against various known attacks, even if an attacker possesses knowledge of the entire private key generation process.

### Evaluations

### The project is evaluated using benchmark datasets encompassing various medical image modalities (X-ray, Ultrasound, and MRI) to assess the effectiveness of DeepKeyGen. The findings reportedly demonstrate that DeepKeyGen generates private keys with high randomness and security levels. Additionally, the method efficiently encrypts medical images of various modalities, showcasing its versatility.

* 1. **PROBLEM STATEMENT**

To develop a system for safe transmission of medical images in a Health Expert System. Medical images contain sensitive and private information about the patients and their leakage can have potential privacy implications for patients. Hence the medical images needed to be encrypted during their transmission to safeguard the privacy of the patients. Medical Images considered in our problem are X-ray, MRI and CT scan images.

The problem statement states images when X-Ray, MRI and CT scan images are transmitted from the data acquisition device to doctor's workstation through a channel, there might be chance of vulnerabilities for attacks like data breach, man in the middle attack, tampering of the data and other attacks. Since medical images, such as X-rays, MRIs, CT scans, and ultrasounds, often contain highly sensitive and private information about patients, including their medical conditions, diagnoses, and potentially identifiable personal details, the leakage or unauthorized access to these images can pose significant privacy risks and have serious implications for patients, including potential identity theft, discrimination, or unauthorized disclosure of medical history. Also the tampering of data leads the doctor to wrongly predict the patient's disease, and leading him wrongly assist the patient privacy. Hence it is essential to encrypt medical images during their transmission over networks.

**CHAPTER 2**

**SYSTEM REQUIREMENTS**

## HARDWARE REQUIREMENTS

* Computer with Windows11 Operating System

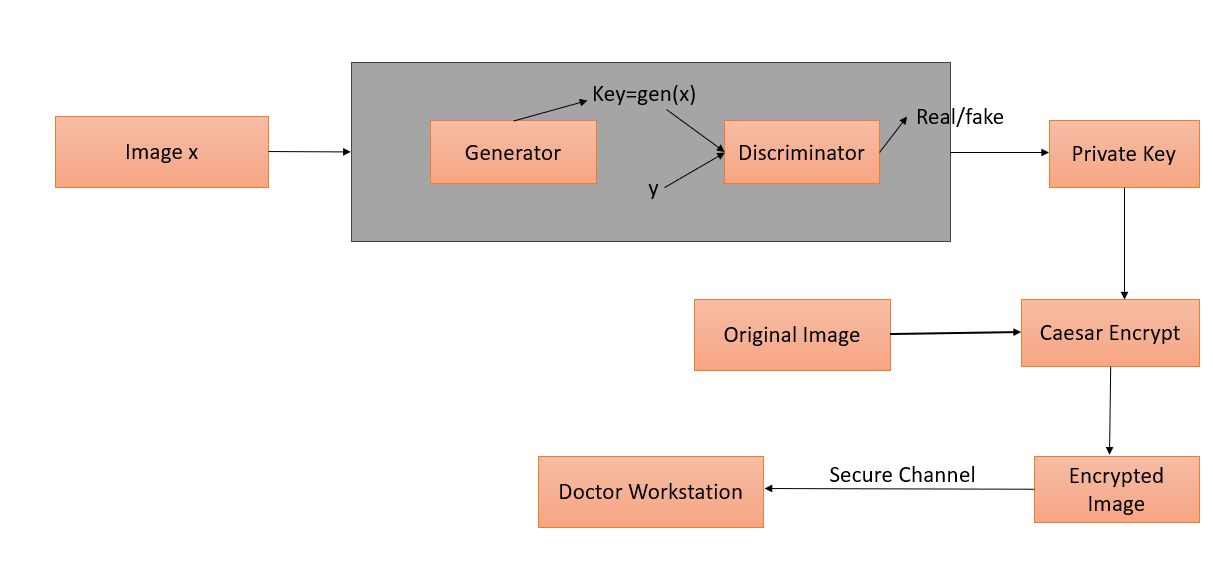
## SOFTWARE REQUIREMENTS

* VS code Version 1.83.1
* PACKAGES: NumPy1.20.3, Matplotlib3.8.0, pyTorch 2.2.2, Nibabel

**CHAPTER 3**

**SYSTEM DESIGN**

**3.1 MODEL ARCHITECTURE**

****

### Fig.3.1.1: Model Architecture

In order to mitigate the risks stated in the problem statement, when the images are transmitted from the medical image acquisition device to the doctor's workstation, the images need to be encrypted securely. Encrypting the image includes a private key that is more secure, random, not vulnerable to attacks, and most probably hard to guess. So in this solution, a Generative Adversarial Network (GAN) is utilized to generate a private key for encrypting and decrypting medical images. The goal of the solution is to learn the mapping relationship of how to transfer the initial image to the private key. The generator network G transforms the initial image from the source domain into the transformation domain, producing the private key.

The architecture of the generator network consists of three down sample layers to extract features from the initial image, six residual blocks to construct the content and diverse features of the image, followed by two transposed convolutional layers to extract the low-level features, and an ordinary convolutional layer to convert the extracted low-level features into an image. The generated private key holds the same security characteristics as the encrypted performance in the transformation domain. Instance normalization is employed in each convolutional layer to enhance the quality of the generated image and accelerate model convergence while preventing gradient explosion.

The loss function of the generator network is designed to mislead the discriminator network into believing that the generated key belongs to the transformation domain. The discriminator network, on the other hand, determines whether the generated image belongs to the transformation domain. It consists of five convolutional layers, four convolutional layers to extract the useful features followed by one convolutional layer to give the output as whether the image is fake or real, and aims to maximize the classification accuracy between real and fake images. The loss function of the discriminator network ensures the discriminative capability of distinguishing between real and generated images. In the adversarial system of GAN, both the loss function of the generator and the loss function of the discriminator contribute to achieving a balance where the discriminator's accuracy approaches 50%. This balance indicates that the generated key closely resembles those from the transformation domain, making it indistinguishable for the discriminator. This generated key is also an image that is more random and can be used to achieve high security, and algorithms like the Caesar algorithm can be adopted as an encryption and decryption algorithm. Then, the image can be transmitted securely.

So the proposed solution in brief is when the patient's MRI or CT scan or X-Ray is acquired from the medical image acquisition device, then it is passed to an AI model that utilizes a GAN network to generate a private key, then the plaintext image is encrypted using the private key with simple algorithms like the Caesar encryption algorithm, then the encrypted ciphertext image is transmitted via a secure channel along with the private key to the doctor's workstation, then can be decrypted using the private key. Since the private key generated is more secure, this simple encryption algorithm, Caesar, can itself give better results and a higher security level. The complexity of the bitwise XOR algorithm is low, as it involves performing a simple bitwise operation on individual bits of data. It has a time complexity of O(n), where n is the number of bits in the input data. Hence, the algorithm is efficient to use for encryption and decryption that does not involve more complexity in the proposed solution and also ensures the secure transmission of medical images. Further the Encryption - Decryption process and loss function of the networks are explained briefly.

**Encryption and Decryption Process**

The encryption and decryption system based on DeepKeyGen combines a stream cipher key generated by DeepKeyGen with Caesar Encryption Decryption algorithm. During encryption, an unencrypted image, plaintext is encrypted using the generated private key and the Caesar Encryption Decryption algorithm, resulting in an encrypted image (ciphertext). Decryption reverses this process.

* Caesar encryption shifts each pixel value of the image tensor by the encryption key modulo 256.

E(x, k) = (x + k) mod 256

* Caesar decryption reverses the encryption by shifting each pixel value back by the decryption key modulo 256.

D(y, k) = (y - k) mod 256

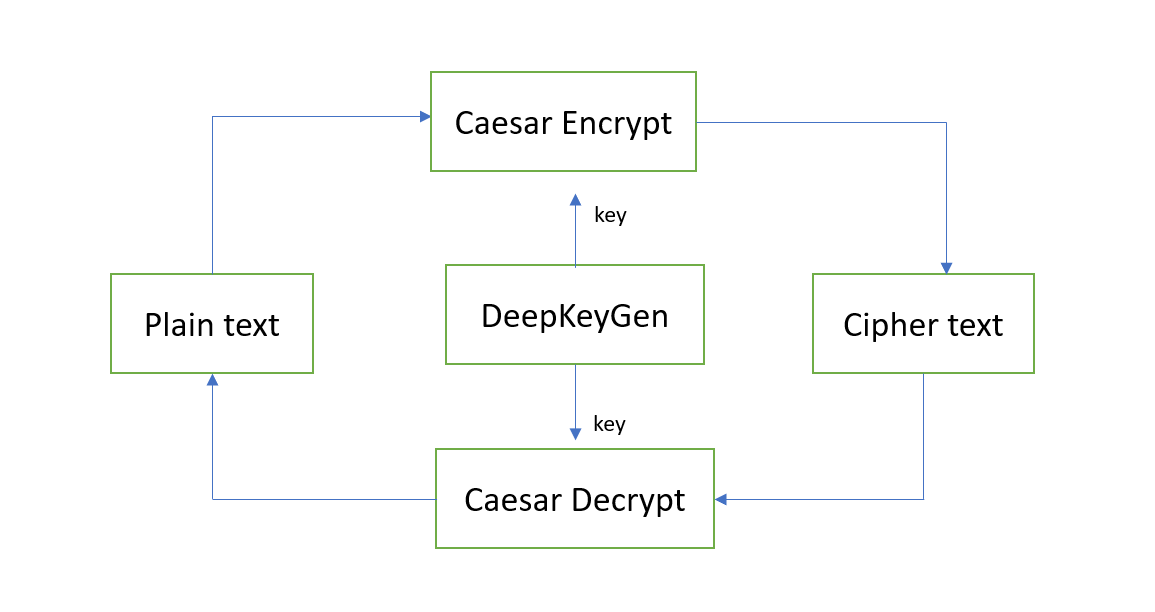


Fig.3.1.2: DeepKeyGen- based encryption and decryption system.

**DeepKeyGen Architecture**

DeepKeyGen consists of two main components:

* a generator (G)
* a discriminator (D)

The generator network G generates the private key from the initial image, while the discriminator network D distinguishes between the generated private key and real data from the transformation domain.

**Generator Network (G)**

The generator network G transforms the initial image from the source domain to the transformation domain to generate the private key. It comprises three down sample layers, six residual blocks, two transposed convolutional layers, and a convolutional layer. These layers extract features from the initial image, construct content and diverse features, and revert them to low-level dimensions to generate the private key. The loss function of the generator network G is

**LG =min G (Ex∼pdata(x) log(1 − D(G(x)))**

In the above equation, G denotes the generator network, D represents the discriminator network, and x represents the initial image. The loss function LG can be understood as the key generated by the generator “mis leading” the discriminator to the maximum extent. It means that the generated key is getting close to the transformation domain, and the discriminator believes that the generated key comes from the transformation domain.

**Discriminator Network (D)**

The discriminator network D determines whether the generated private key belongs to the transformation domain. It consists of five convolutional layers that extract useful features from the input image and output a 1-D result representing real or fake data. The loss function of the discriminator is

**LD =max D (Ey∼pdata(y) log(D(y)) +Ex∼pdata(x) log(1 − D(G(x))))**

In the above equation, G represents the generator network, D represents the discriminator network, x represents the initial image from the source domain, and y represents the data from the transformation domain. The loss function LD can be understood as the maximization of the classification accuracy of the discriminator.

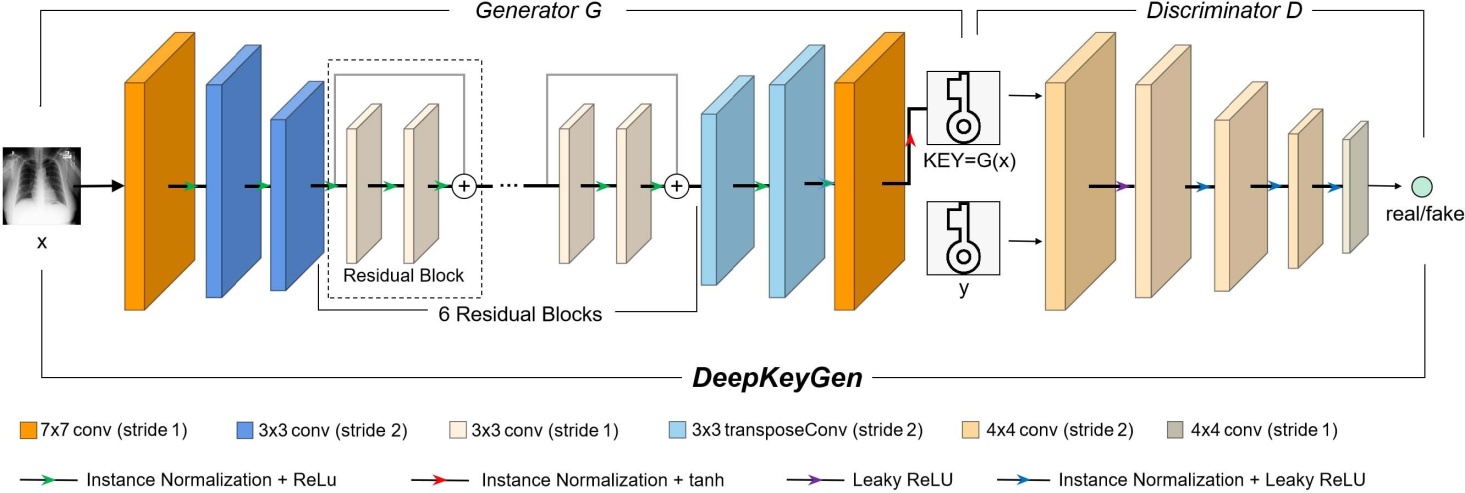


Fig. 3.1.3 Proposed DeepKeyGen architecture

**Image Private Key**

The private key generated by DeepKeyGen is a type of stream cipher and is represented as an image. Each image consists of a sequence of pixels, with each pixel containing the pixel value, horizontal and vertical coordinate values, and RGB color channel information. This unique structure enhances the security level of the key.

**Key Generation Process**

The key generation process involves training the DeepKeyGen network to learn the mapping relationship from the source domain to the transformation domain. Before training, the network parameters are randomly initialized. The training process involves forward propagation to generate the original private key, calculation of the total loss to measure the difference between the generated key and the target key, backward propagation to update network parameters, and iteration until convergence.

Overall, the proposed method utilizes deep learning techniques to generate private keys for encryption, with safeguards against potential imitation learning attacks to ensure the security of the encryption system.

**3.2 DATASET**

The BraTS (Multimodal Brain Tumor Segmentation) dataset is a widely used benchmark dataset in the field of medical image analysis, specifically focusing on brain tumor segmentation. It provides researchers with a diverse collection of multi-modal magnetic resonance imaging (MRI) scans, along with corresponding expert annotations delineating tumor regions.

Key features of the BraTS dataset includes,

**Multi-Modal Imaging:** The dataset consists of MRI scans acquired using multiple imaging sequences, including T1-weighted, T1-weighted with contrast enhancement (T1ce), T2-weighted, and Fluid Attenuated Inversion Recovery (FLAIR) sequences. Each modality provides complementary information about brain tissue and tumor characteristics.

**Expert Annotations:** Expert annotations are provided for each MRI scan, delineating various tumor regions such as the tumor core (including necrotic and non-enhancing tumor regions), peritumoral edema, and enhancing tumor regions. These annotations serve as ground truth labels for training and evaluating tumor segmentation algorithms.

**Diverse Pathologies:** The BraTS dataset encompasses a wide range of brain tumor pathologies, including gliomas of different grades (e.g., low-grade gliomas, high-grade gliomas) and tumor types (e.g., glioblastoma multiforme). This diversity enables researchers to evaluate segmentation algorithms across different tumor types and grades.

**Large-Scale and Continuously Updated:** The BraTS dataset is continuously updated with new data and annotations, ensuring its relevance and usefulness for ongoing research in brain tumor segmentation and related fields. The dataset has grown over the years in terms of both the number of subjects and the quality of annotations.

**Challenges and Competitions:** The BraTS dataset has been the focus of various challenges and competitions, where researchers develop and evaluate state-of-the-art algorithms for brain tumor segmentation. These challenges foster collaboration and innovation within the medical imaging research community.

Overall, the BraTS dataset plays a crucial role in advancing research in brain tumor segmentation and understanding the underlying pathology of brain tumors. Its multi-modal nature, expert annotations, and diversity of pathologies make it a valuable resource for developing and benchmarking segmentation algorithms aimed at improving diagnosis and treatment planning for patients with brain tumors.

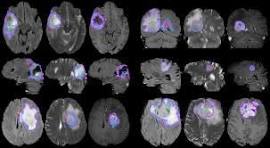


Fig. 3.2.1 Sample Images of BraTS Dataset

**3.3 MODULES DESCRIPTION**

### NUMPYMODULE

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O and much more.

### MATPLOTLIB MODULE

Matplotlib is a library in Python that enables users to generate visualizations like histograms, scatterplots, bar charts, pie charts and much more. It provides data visualizations that are typically more aesthetic and statistically sophisticated.

**PYTORCH MODULE**

The PyTorch framework offers essential functionalities for deep learning. It includes tensor operations, automatic differentiation for gradient computation, neural network building blocks, optimization algorithms, data utilities, and device management. This module provides a versatile and efficient platform for developing and training neural networks, enabling seamless computation on both CPUs and GPUs while simplifying model construction and optimization.

**NIBABEL MODULE**

Nibabel is a Python library designed for reading and writing neuroimaging data in various formats, including NIfTI and ANALYZE. It provides a flexible and intuitive interface for working with volumetric medical imaging data, allowing users to access and manipulate voxel-wise information efficiently. Nibabel supports both 3D and 4D imaging data, along with associated metadata such as affine transformation matrices and voxel dimensions. Its simple yet powerful API enables tasks such as loading, saving, visualization, and preprocessing of neuroimaging datasets, making it a valuable tool for researchers and practitioners in the field of neuroimaging. Nibabel's open-source nature and extensive documentation further contribute to its popularity and usability within the neuroimaging community.

**3.4 MODELS DESCRIPTION**

**GAN**

Generative Adversarial Networks (GANs) are a class of deep learning models composed of two neural networks, the generator and the discriminator, trained simultaneously through a min-max game. The generator learns to produce synthetic data samples, such as images or text, that resemble real data, while the discriminator learns to distinguish between real and fake data. Through iterative training, the generator improves its ability to generate realistic samples by fooling the discriminator, while the discriminator improves its ability to differentiate between real and fake samples. This adversarial training process leads to the generation of increasingly realistic data samples. GANs have demonstrated remarkable success in various tasks such as image generation, style transfer, and data augmentation, making them a powerful tool for generating high-quality synthetic data and pushing the boundaries of generative modeling.

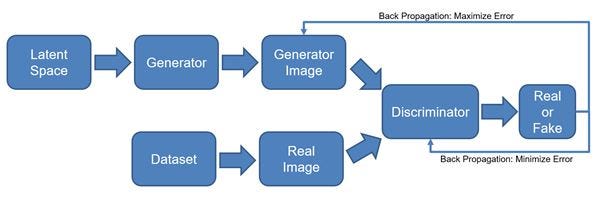


Fig 3.4.1: GAN Architecture

**CHAPTER 4**

**IMPLEMENTATION**

**4.1 Example Explanation**

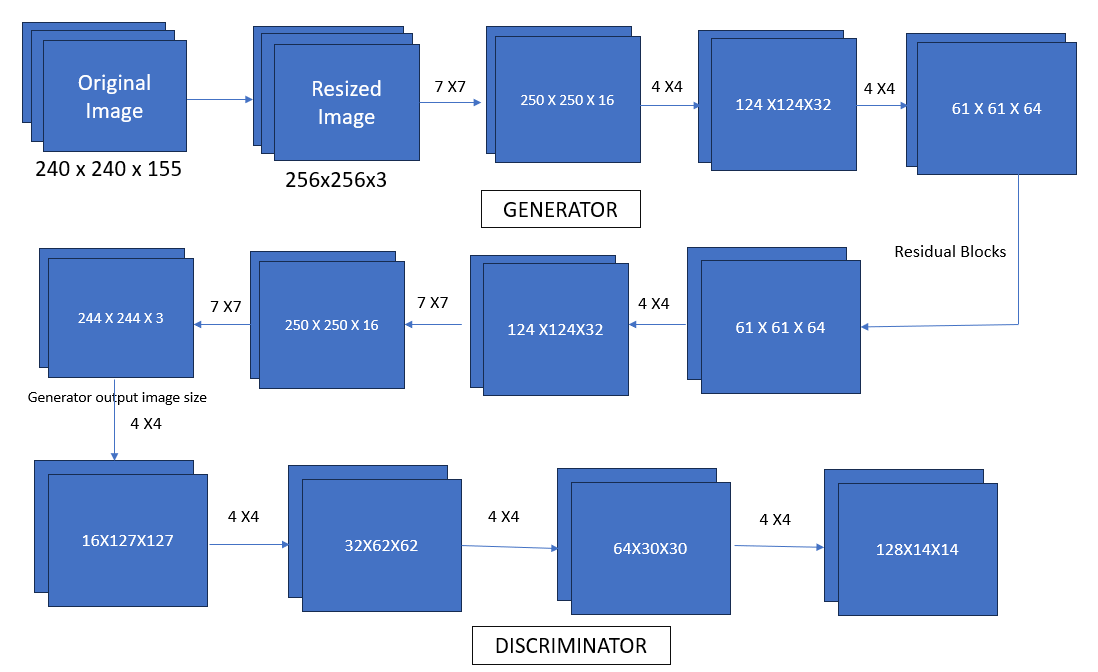
****

Fig 4.1.1. Image transformation

The original image size is 240X240X155 and then it is resized to 256X256X3 then it is convoluted into 250X250X16 with an 7X7 convolution kernel then it is convoluted into 124X124X32 with an 4X4 convolution kernel then it is convoluted into 61X61X64 with an 4X4 convolution kernel then it is convoluted into 61X61X64 by residual block then it is convoluted into 124X124X32 with an 4X4 convolution kernel then it is convoluted into 250X250X16 with an 7X7 convolution kernel then it is convoluted into 244X244X3 with an 7X7 convolution kernel. Then it is passed to the discriminator network and it has four convolution layers, all layers having kernel size of 4X4. The image transformation is as follows, 127X127X16, 62X62X32, 30X30X64, 14X14X128. This is used to predict its real or fake by the discriminator.

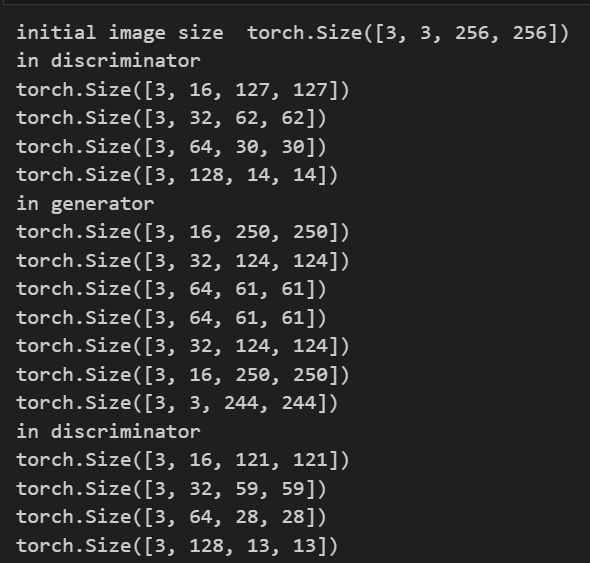
****

Fig 4.1.2 Image transformation sizes after every convolution

**4.2 Source Code**

import os

import nibabel as nib

import matplotlib.pyplot as plt

import torch

import os

import numpy as np

import nibabel as nib

from torch.utils.data import Dataset, DataLoader

from torchvision.transforms import Resize, ToTensor

img\_size=240

batch\_size=3

def load\_and\_preprocess\_nifti(file\_path):

nifti\_data = nib.load(file\_path)

nifti\_array = nifti\_data.get\_fdata()

nifti\_tensor = torch.from\_numpy(nifti\_array)

return nifti\_tensor

brats18\_dataset\_dir = r"D:\documents\datasets\brats18\MICCAI\_BraTS\_2018\_Data\_Training\HGG\Brats18\_2013\_2\_1"

file\_names = os.listdir(brats18\_dataset\_dir)

file\_paths = [os.path.join(brats18\_dataset\_dir, file\_name) for file\_name in file\_names]

class BRATS18Dataset(Dataset):

def \_\_init\_\_(self, file\_paths):

self.file\_paths = file\_paths

def \_\_len\_\_(self):

return len(self.file\_paths)

def \_\_getitem\_\_(self, idx):

file\_path = self.file\_paths[idx]

image = load\_and\_preprocess\_nifti(file\_path)

return image

brats18\_dataset = BRATS18Dataset(file\_paths)

brats18\_dataloader = DataLoader(brats18\_dataset, batch\_size=batch\_size, shuffle=True)

import torch

import matplotlib.pyplot as plt

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import DataLoader

from torchvision.transforms import Resize

from torchvision.utils import save\_image

class ResidualBlock(nn.Module):

def \_\_init\_\_(self, in\_channels):

super(ResidualBlock, self).\_\_init\_\_()

self.conv1 = nn.Conv2d(in\_channels, in\_channels, kernel\_size=3, stride=1, padding=1)

self.conv2 = nn.Conv2d(in\_channels, in\_channels, kernel\_size=3, stride=1, padding=1)

self.instancenorm = nn.InstanceNorm2d(in\_channels)

self.relu = nn.ReLU(inplace=True)

def forward(self, x):

residual = x

#print("Input tensor shape:", x.shape)

out = self.relu(self.instancenorm(self.conv1(x)))

#print("Intermediate tensor shape:", out.shape)

out = self.instancenorm(self.conv2(out))

#print("Output tensor shape:", out.shape)

out += residual

return out

class Generator(nn.Module):

def \_\_init\_\_(self, in\_channels, out\_channels):

super(Generator, self).\_\_init\_\_()

self.conv1 = nn.Conv2d(in\_channels, 16, kernel\_size=7, stride=1)

self.norm1 = nn.InstanceNorm2d(16)

self.relu = nn.ReLU(inplace=True)

# Downsampling

self.conv2 = nn.Conv2d(16, 32, kernel\_size=4, stride=2)

self.norm2 = nn.InstanceNorm2d(32)

self.conv3 = nn.Conv2d(32, 64, kernel\_size=4, stride=2)

self.norm3 = nn.InstanceNorm2d(64)

# Residual blocks

self.residual\_blocks = nn.Sequential(\*[ResidualBlock(64) for \_ in range(6)])

# Upsampling

self.deconv1 = nn.ConvTranspose2d(64, 32, kernel\_size=4, stride=2)

self.norm4 = nn.InstanceNorm2d(32)

self.deconv2 = nn.ConvTranspose2d(32, 16, kernel\_size=4, stride=2)

self.norm5 = nn.InstanceNorm2d(16)

# Output layer

self.conv4 = nn.Conv2d(16, out\_channels, kernel\_size=7, stride=1)

self.norm6 = nn.InstanceNorm2d(out\_channels)

def forward(self, x):

out = self.relu(self.norm1(self.conv1(x)))

#print("in gen")

#print(out.shape)

out = self.relu(self.norm2(self.conv2(out)))

#print(out.shape)

out = self.relu(self.norm3(self.conv3(out)))

#print(out.shape)

out = self.residual\_blocks(out)

#print(out.shape)

out = self.relu(self.norm4(self.deconv1(out)))

#print(out.shape)

out = self.relu(self.norm5(self.deconv2(out)))

#print(out.shape)

out = torch.tanh(self.norm6(self.conv4(out)))

#print(out.shape)

return out

class Discriminator(nn.Module):

def \_\_init\_\_(self, in\_channels):

super(Discriminator, self).\_\_init\_\_()

self.conv1 = nn.Conv2d(in\_channels, 16, kernel\_size=4, stride=2)

self.leaky\_relu1 = nn.LeakyReLU(0.2, inplace=True)

self.conv2 = nn.Conv2d(16, 32, kernel\_size=4, stride=2)

self.norm1 = nn.InstanceNorm2d(32)

self.leaky\_relu2 = nn.LeakyReLU(0.2, inplace=True)

self.conv3 = nn.Conv2d(32, 64, kernel\_size=4, stride=2, )

self.norm2 = nn.InstanceNorm2d(64)

self.leaky\_relu3 = nn.LeakyReLU(0.2, inplace=True)

self.conv4 = nn.Conv2d(64, 128, kernel\_size=4, stride=2)

self.norm3 = nn.InstanceNorm2d(128)

self.leaky\_relu4 = nn.LeakyReLU(0.2, inplace=True)

self.conv5 = nn.Conv2d(128, 128, kernel\_size=4, stride=1, )

def forward(self, x):

out = self.leaky\_relu1(self.conv1(x))

#print("in dis")

#print(out.shape)

out = self.leaky\_relu2(self.norm1(self.conv2(out)))

#print(out.shape)

out = self.leaky\_relu3(self.norm2(self.conv3(out)))

#print(out.shape)

out = self.leaky\_relu4(self.norm3(self.conv4(out)))

#print(out.shape)

out = self.conv5(out)

return out

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

lr = 0.0002

batch\_size = 64

num\_epochs = 100

img\_size = 240

latent\_dim = 3

transform = Resize((256, 256))

generator = Generator(in\_channels=latent\_dim, out\_channels=3).to(device)

discriminator = Discriminator(in\_channels=3).to(device)

epoch=0

criterion = nn.BCEWithLogitsLoss()

optim\_G = optim.Adam(generator.parameters(), lr=lr, betas=(0.5, 0.999))

optim\_D = optim.Adam(discriminator.parameters(), lr=lr, betas=(0.5, 0.999))

generatorloss=[]

discriminatorloss=[]

while epoch < 100:

for batch\_idx, real\_images in enumerate(brats18\_dataloader):

real\_images = real\_images.to(device)

real\_images = real\_images.to(torch.float32)

#print(real\_images.shape)

final\_tensor = real\_images.permute(0, 3, 1, 2)

final\_tensor = transform(final\_tensor)

#print("1",final\_tensor.shape)

reduced\_img\_data = torch.mean(final\_tensor, dim=1, keepdim=True)

#print(reduced\_img\_data.shape)

img\_tensor\_3\_channels = reduced\_img\_data.repeat(1, 3, 1, 1)

#print("check",img\_tensor\_3\_channels.shape)

# Train discriminator

optim\_D.zero\_grad()

# Train with real images

real\_outputs = discriminator(img\_tensor\_3\_channels)

d\_loss\_real = criterion(real\_outputs, torch.ones\_like(real\_outputs))

d\_loss\_real.backward()

# Train with fake images

fake\_images = generator(img\_tensor\_3\_channels)

fake\_outputs = discriminator(fake\_images)

d\_loss\_fake = criterion(fake\_outputs, torch.zeros\_like(fake\_outputs))

d\_loss\_fake.backward()

d\_loss = d\_loss\_real + d\_loss\_fake

optim\_D.step()

# Train generator

optim\_G.zero\_grad()

z = torch.randn\_like(img\_tensor\_3\_channels)

#print("z",z.shape)

gen\_images = generator(z)

gen\_outputs = discriminator(gen\_images)

g\_loss = criterion(gen\_outputs, torch.ones\_like(gen\_outputs))

g\_loss.backward()

optim\_G.step()

if batch\_idx % 100 == 0:

print(

f"Epoch [{epoch}/{num\_epochs}], "

f"Generator Loss: {g\_loss.item():.4f}, Discriminator Loss: {d\_loss.item():.4f}"

)

generatorloss.append(g\_loss.item())

discriminatorloss.append(d\_loss.item())

epoch += 1

with torch.no\_grad():

z = torch.randn\_like(img\_tensor\_3\_channels).to(device)

gen\_images = generator(z)

print(gen\_images.shape)

save\_image(gen\_images, f"encryption\_key.png", normalize=True)

encryption\_key=torch.tensor(gen\_images)

encryption\_key

def load\_nifti\_file(file\_path):

nifti\_data = nib.load(file\_path)

nifti\_array = nifti\_data.get\_fdata()

nifti\_tensor = torch.from\_numpy(nifti\_array)

return nifti\_tensor

def visualize\_nifti(nifti\_tensor):

plt.figure(figsize=(12, 6))

slice\_w = min(25, nifti\_tensor.shape[2] // 2)

middle\_slice = nifti\_tensor[:, :, nifti\_tensor.shape[2] // 2]

plt.imshow(middle\_slice, cmap='gray')

plt.title('Original Image')

plt.axis('off')

plt.show()

original\_image =r"D:\documents\datasets\brats18\MICCAI\_BraTS\_2018\_Data\_Validation\Brats18\_CBICA\_ALA\_1"

for file\_name in os.listdir(original\_image):

if file\_name.endswith(".nii") or file\_name.endswith(".nii.gz"):

file\_path = os.path.join(original\_image, file\_name)

print("Loading and visualizing:", file\_path)

original\_tensor = load\_nifti\_file(file\_path)

visualize\_nifti(original\_tensor)

print(original\_tensor.shape)

break

original\_tensor=original\_tensor.permute(2, 1, 0)

print(original\_tensor.shape)

img = original\_tensor.repeat(3, 1, 1, 1)

img.shape

import torch

import torch.nn.functional as F

def resize\_tensor(input\_tensor, target\_size):

resized\_tensor = F.interpolate(input\_tensor, size=(target\_size[2], target\_size[3]), mode='bilinear', align\_corners=False)

if resized\_tensor.shape[2] < target\_size[2]:

pad\_size = (target\_size[2] - resized\_tensor.shape[2]) // 2

resized\_tensor = F.pad(resized\_tensor, (pad\_size, pad\_size, pad\_size, pad\_size))

elif resized\_tensor.shape[2] > target\_size[2]:

crop\_size = (resized\_tensor.shape[2] - target\_size[2]) // 2

resized\_tensor = resized\_tensor[:, :, crop\_size:crop\_size+target\_size[2], :]

if resized\_tensor.shape[3] < target\_size[3]:

pad\_size = (target\_size[3] - resized\_tensor.shape[3]) // 2

resized\_tensor = F.pad(resized\_tensor, (pad\_size, pad\_size, pad\_size, pad\_size))

elif resized\_tensor.shape[3] > target\_size[3]:

crop\_size = (resized\_tensor.shape[3] - target\_size[3]) // 2

resized\_tensor = resized\_tensor[:, :, :, crop\_size:crop\_size+target\_size[3]]

return resized\_tensor

target\_size=(2,3,244,244)

resized\_tensor = resize\_tensor(img, target\_size)

print("Resized tensor shape:", resized\_tensor.shape)

import torch.nn.functional as F

def reduce\_channels\_and\_width(input\_tensor, target\_channels, target\_width):

reduced\_channels\_tensor = input\_tensor[:target\_channels]

reduced\_width\_tensor = F.interpolate(reduced\_channels\_tensor, size=(input\_tensor.shape[2], target\_width), mode='bilinear', align\_corners=False)

return reduced\_width\_tensor

resize\_tensor=resized\_tensor.permute(0, 2, 3,1)

target\_channels = 2

target\_width = 3

reduced\_tensor = reduce\_channels\_and\_width(resize\_tensor, target\_channels, target\_width)

print("Reduced tensor shape:", reduced\_tensor.shape)

reduced\_tensor=reduced\_tensor.permute(0,3,1,2)

print("Reduced tensor shape:", reduced\_tensor.shape)

reduced\_tensor.shape

encryption\_key.shape

import torch

def caesar\_encrypt\_image\_tensor(image\_tensor, key\_tensor):

assert image\_tensor.size() == key\_tensor.size(), "Image tensor and key tensor must have the same size"

encrypted\_tensor = (image\_tensor + key\_tensor) % 256 # Ensure values are in the range [0, 255]

return encrypted\_tensor

def caesar\_decrypt\_image\_tensor(encrypted\_tensor, key\_tensor):

assert encrypted\_tensor.size() == key\_tensor.size(), "Encrypted tensor and key tensor must have the same size"

decrypted\_tensor = (encrypted\_tensor - key\_tensor) % 256 # Ensure values are in the range [0, 255]

return decrypted\_tensor

encrypted\_image\_tensor = caesar\_encrypt\_image\_tensor(reduced\_tensor, encryption\_key)

decrypted\_image\_tensor = caesar\_decrypt\_image\_tensor(encrypted\_image\_tensor, encryption\_key)

encrypted\_image\_tensor.shape

import torch

encrypted\_img = encrypted\_image\_tensor.permute(1, 2, 3, 0)

encrypted\_img = encrypted\_img.contiguous().view(3, 244, -1)

encrypted\_img.shape

import torchvision.transforms as transforms

new\_height = 244

new\_width = 244

resize\_transform = transforms.Resize((new\_height, new\_width))

resized\_encrypted\_img = resize\_transform(encrypted\_img)

resized\_encrypted\_img.shape

resized\_encrypted\_img=resized\_encrypted\_img.permute(1,2,0)

resized\_encrypted\_img.shape

plt.figure(figsize=(12, 6))

slice\_w = min(25, resized\_encrypted\_img.shape[2] // 2)

middle\_slice = resized\_encrypted\_img[:, :, resized\_encrypted\_img.shape[2] // 2]

plt.imshow(middle\_slice, cmap='gray')

plt.title('Encrypted Image')

plt.axis('off')

plt.show()

decrypted\_image\_tensor.shape

import torch

decrypted\_img = decrypted\_image\_tensor.permute(1, 2, 3, 0)

decrypted\_img = decrypted\_img.contiguous().view(3, 244, -1)

decrypted\_img.shape

import torchvision.transforms as transforms

new\_height = 244

new\_width = 244

resize\_transform = transforms.Resize((new\_height, new\_width))

resized\_decrypted\_img = resize\_transform(decrypted\_img)

resized\_decrypted\_img.shape

resized\_decrypted\_img=resized\_decrypted\_img.permute(1,2,0)

resized\_decrypted\_img.shape

import matplotlib.pyplot as plt

plt.figure(figsize=(12, 6))

slice\_w = min(25, resized\_decrypted\_img.shape[2] // 2)

middle\_slice = resized\_decrypted\_img[:, :, resized\_decrypted\_img.shape[2] // 2]

plt.imshow(middle\_slice, cmap='gray')

plt.title('Decrypted Image')

plt.axis('off')

plt.show()

plt.plot(discriminatorloss,label="Discriminator loss")

plt.plot(generatorloss,label="Generator loss")

plt.xlabel('epochs')

plt.ylabel('loss')

plt.legend()

plt.show()

import numpy as np

import matplotlib.pyplot as plt

image\_array = reduced\_tensor.numpy()

flat\_image = image\_array.flatten()

plt.figure(figsize=(10, 6))

plt.hist(flat\_image, bins=256, range=(0, 255), color='b', alpha=0.7)

plt.title('Histogram of Image')

plt.xlabel('Pixel Intensity')

plt.ylabel('Frequency')

plt.show()

import numpy as np

import matplotlib.pyplot as plt

image\_array = resized\_encrypted\_img.numpy()

flat\_image = image\_array.flatten()

plt.figure(figsize=(10, 6))

plt.hist(flat\_image, bins=256, range=(0, 255), color='b', alpha=0.7)

plt.title('Histogram of Image')

plt.xlabel('Pixel Intensity')

plt.ylabel('Frequency')

plt.show()

**CHAPTER 5**

**RESULTS**

****

Fig 5.1: Encrypted Image

This output represents the image encrypted using a key generated by DeepKeyGen, ensuring secure data transmission and storage through robust encryption. The encrypted image underscores DeepKeyGen's capability in providing secure encryption solutions, safeguarding sensitive data against unauthorized access and ensuring privacy in communication channels.

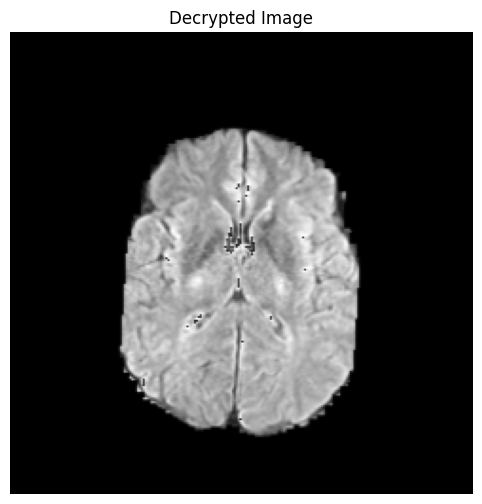
****

Fig 5.2: Decrypted Image

This decrypted image underscores DeepKeyGen's effectiveness in securely decrypting data encrypted with keys it generates, highlighting its role in maintaining data integrity and privacy. This decrypted image, uniquely deciphered using the key exclusively generated by DeepKeyGen, underscores its role in ensuring data security, as other keys are unable to decrypt the encrypted content.

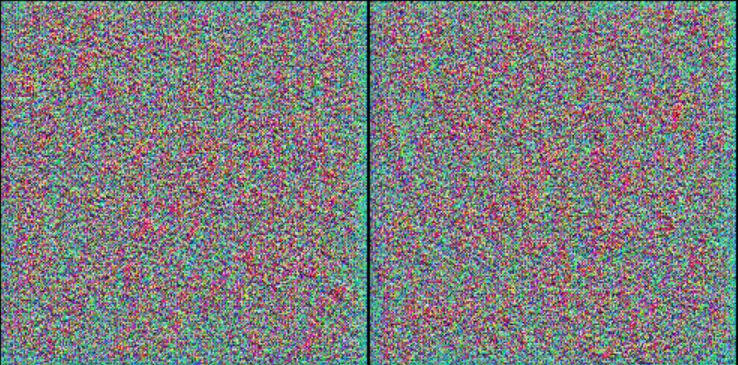
****

Fig 5.3: Key 1 generated by GAN

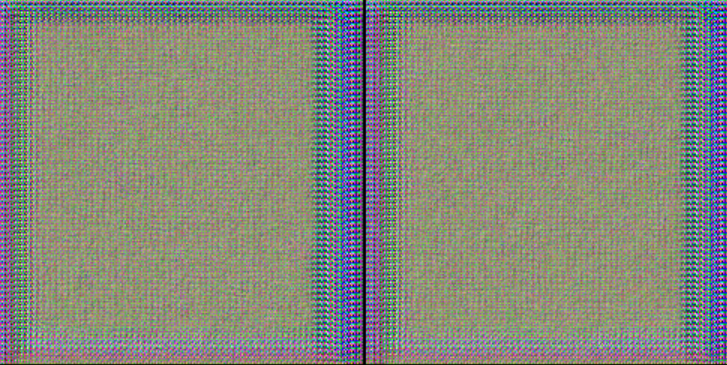
****

Fig 5.4: Key 2 generated by GAN

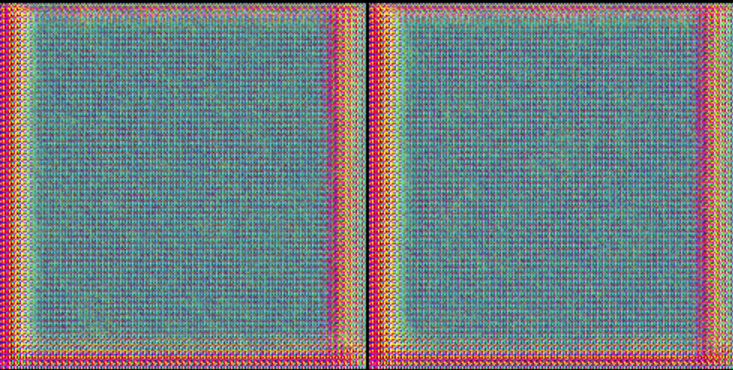
****

Fig 5.5: Key 3 generated by GAN

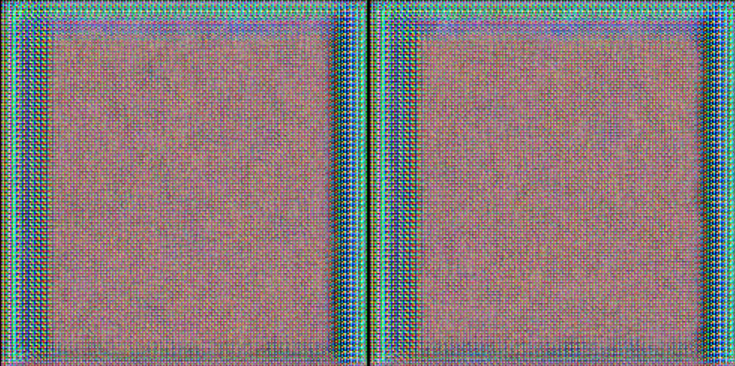
****

Fig 5.6: Key 4 generated by GAN

****

Fig 5.7: Key 5 generated by GAN

The figures above provide visual representations of five distinct private keys generated by DeepKeyGen under consistent experimental settings. Through individual training runs, each key undergoes a unique generation process, amplifying randomness and thereby fortifying security measures. This variability in key generation not only complicates pattern prediction but also ensures heightened resilience against potential attacks, underscoring the robustness and reliability of DeepKeyGen in safeguarding sensitive data.

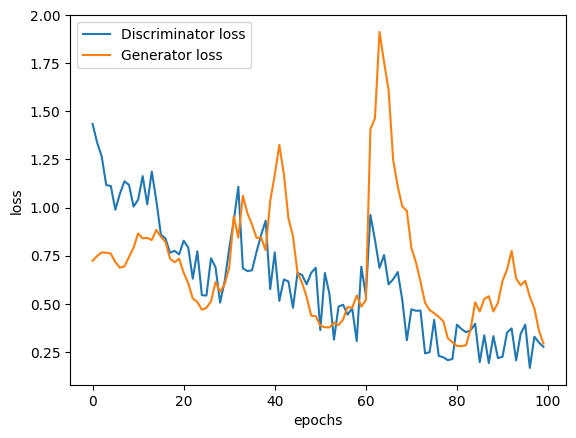
****

Fig 5.8: Discriminator Loss VS Generator Loss

This plot illustrates the convergence of both the generator and discriminator losses, indicating successful training. This convergence suggests that both components, the generator and discriminator, have effectively learned to generate more random keys.

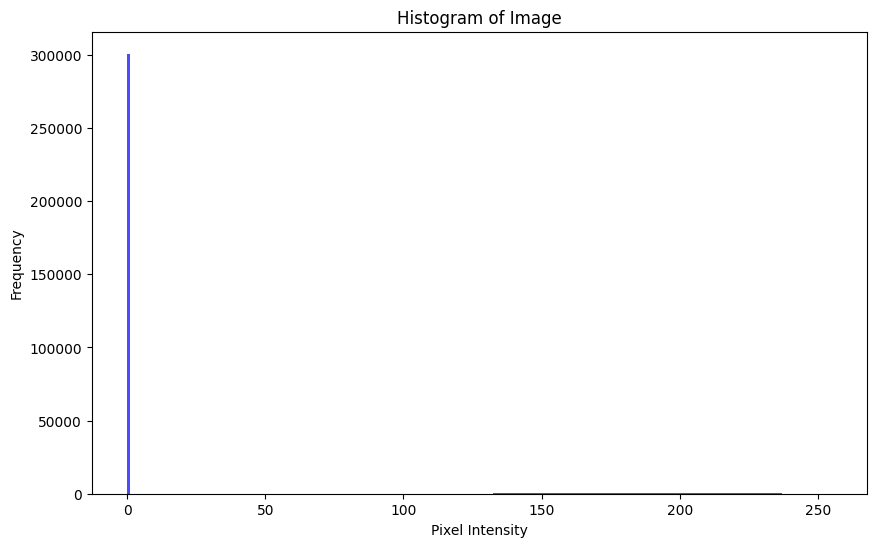
****

Fig 5.9: Pixel intensity of the original image

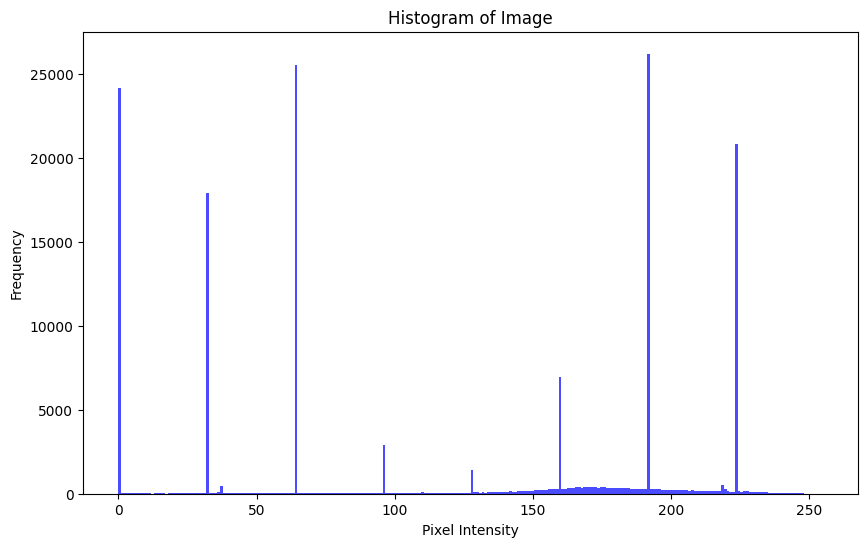
****

Fig 5.10: Pixel intensity of the Encrypted image

Comparing histograms between an original image and its encrypted counterpart offers insights into the encryption process. A uniform distribution in the encrypted image suggests effective randomization, crucial for security. Similar shapes between histograms indicate preservation of statistical characteristics, implying reversible encryption. Shifts or spreads in histograms reveal alterations made during encryption, like addition or multiplication. Missing or reduced histogram parts may signify information loss due to compression or quantization. Unexpected artifacts in histograms could indicate weaknesses in the encryption algorithm. Consistency across multiple image pairs indicates the reliability and predictability of the encryption method. Overall, histogram analysis aids in assessing encryption quality, security, and characteristics, but should be supplemented with other cryptographic evaluations for thorough assessment.

**CHAPTER 6**

**CONCLUSION**

In conclusion, the DeepKeyGen project represents a significant advancement in the field of medical image encryption, leveraging deep learning techniques to address the critical need for secure and efficient protection of patient data in digital healthcare settings. By proposing a novel approach to private key generation using Generative Adversarial Networks (GANs), DeepKeyGen offers a promising solution to the challenges faced by traditional cryptographic methods in medical imaging. Through the development and evaluation of the DeepKeyGen architecture, including the generator and discriminator networks, the project has demonstrated the feasibility and effectiveness of using deep learning for generating secure private keys from medical images. The integration of DeepKeyGen with existing encryption and decryption systems, such as XOR algorithms, provides a seamless and efficient encryption solution for medical imaging data, ensuring confidentiality and integrity during storage and transmission.The evaluation and validation of DeepKeyGen using diverse medical imaging datasets, including the BraTS dataset, have highlighted its ability to generate private keys with high randomness and security levels across various imaging modalities and clinical scenarios. Moreover, the project's exploration of potential vulnerabilities and attack vectors, such as imitation learning attacks, underscores the importance of robust security measures to safeguard against unauthorized access and data breaches.Overall, the DeepKeyGen project contributes to the advancement of encryption technologies in healthcare, offering a practical and scalable solution for protecting sensitive medical imaging data while promoting interoperability and compliance with regulatory requirements. By addressing key challenges in medical image encryption and security, DeepKeyGen paves the way for improved privacy protection, enhanced data security, and better patient care in digital healthcare ecosystems.As future work, further refinements and optimizations of the DeepKeyGen architecture, as well as continued research into advanced encryption techniques and security mechanisms, will be essential to address emerging threats and ensure the long-term effectiveness and resilience of medical image encryption systems. Additionally, collaborations with healthcare providers, industry partners, and regulatory bodies will be vital to integrate DeepKeyGen into clinical practice and promote its adoption as a standard encryption solution in healthcare settings.

**CHAPTER 7**

**REFERENCES**

* DeepKeyGen: A Deep Learning-Based Stream Cipher Generator for Medical Image Encryption and Decryption Yi Ding, Member, IEEE, Fuyuan Tan, ZhenQin Kim-Kwang Raymond Choo, Member, IEEE, Mingsheng Cao, Senior Member, IEEE, and Zhiguang Qin, Member, IEEE, Member, IEEE