A Project Report

on

A FUSION APPROACH TO INFRARED AND VISIBLE IMAGES

submitted in partial fulfillment of the requirements for the award of the degree

of

BACHELOR OF TECHNOLOGY

in

COMPUTER SICENCE AND ENGINEERING

by

17WH1A0595	**Ms. RAHELA MAHENAZ**
17WH1A0590	**Ms. A. SRAVYA**
18WH5A0515	**Ms. P. DIVYA REKHA**

under the esteemed guidance of

Ms. G. Nagamani
Assistant Professor

Department of Computer Science and Engineering
BVRIT HYDERABAD
College of Engineering for Women
(NBA Accredited – EEE, ECE, CSE and IT)
(Approved by AICTE, New Delhi and Affiliated to JNTUH, Hyderabad)
Bachupally, Hyderabad – 500090

June, 2021

DECLARATION

We hereby declare that the work presented in this project entitled "**A FUSION APPROACH TO INFRARED AND VISIBLE IMAGES**" submitted towards completion of Project Work in IV year of B.Tech., CSE at 'BVRIT HYDERABAD College of Engineering for Women', Hyderabad is an authentic record of our original work carried out under the guidance of Ms. G. Nagamani, Assistant Professor, Department of CSE.

Sign. with date:

Ms. RAHELA MAHENAZ

(17WH1A0595)

Sign. with date:

Ms. A. SRAVYA

(17WH1A0590)

Sign. with date:

Ms. P. DIVYA REKHA

(18WH5A0515)

BVRIT HYDERABAD
College of Engineering for Women
(NBA Accredited – EEE, ECE, CSE and IT)
(Approved by AICTE, New Delhi and Affiliated to JNTUH, Hyderabad)
Bachupally, Hyderabad – 500090

Department of Computer Science and Engineering

Certificate

This is to certify that the Project Work report on "**A FUSION APPROACH TO INFRARED AND VISIBLE IMAGES**" is a bonafide work carried out by Ms. RAHELA MAHENAZ (17WH1A0595), Ms. A. SRAVYA (17WH1A0590), Ms. P. DIVYA REKHA (18WH5A0515) in the partial fulfillment for the award of B.Tech. degree in **Computer Science and Engineering, BVRIT HYDERABAD College of Engineering for Women, Bachupally, Hyderabad**, affiliated to Jawaharlal Nehru Technological University Hyderabad, Hyderabad under my guidance and supervision.

The results embodied in the project work have not been submitted to any other University or Institute for the award of any degree or diploma.

Head of the Department	**Internal Guide**
Dr. K. Srinivasa Reddy	**Ms. G. Nagamani**
Professor and HoD,	**Assistant Professor**
Department of CSE	

External Examiner

ACKNOWLEDGEMENT

We would like to express our sincere thanks to **Dr. K V N Sunitha, Principal**, **BVRIT HYDERABAD College of Engineering for Women**, for providing the working facilities in the college.

Our sincere thanks and gratitude to our **Dr. K. Srinivasa Reddy, Head**, Department of CSE, **BVRIT HYDERABAD College of Engineering for Women** for all the timely support and valuable suggestions during the period of our project.

We are extremely thankful and indebted to our internal guide, **Ms. G. Nagamani, Assistant Professor**, Department of CSE**, BVRIT HYDERABAD College of Engineering for Women** for his constant guidance, encouragement and moral support throughout the project.

Finally, we would also like to thank our Project Coordinator, all the faculty and staff of **CSE** Department who helped us directly or indirectly, parents and friends for their cooperation in completing the project work.

Ms. RAHELA MAHENAZ

(17WH1A0595)

Ms. A. SRAVYA

(17WH1A0590)

Ms. P. DIVYA REKHA

(18WH5A0515)

Contents

ABSTRACT

The infrared and visible image fusion task is an important problem in image processing field. It attempts to extract salient features from source images, and these features are integrated into a single image by appropriate fusion method. It is a novel deep learning architecture for infrared and visible images fusion problems. In contrast to conventional convolutional networks, our encoding network is combined with convolutional layers, a fusion layer, and dense block in which the output of each layer is connected to every other layer. We attempt to use this architecture to get more useful features from source images in the encoding process, and two fusion layers (fusion strategies) are designed to fuse these features. We use encoding network to extract image features and the fused image is obtained by decoding network. The encoding network is constructed by convolutional layer and dense block in which the output of each layer is used as the input of next layer. Finally, the fused image will be reconstructed by fusion strategy and decoding network which includes four CNN layers.

LIST OF FIGURES

1. INTRODUCTION

The infrared and visible image fusion task is an important problem in image processing field. It attempts to extract salient features from source images then these features are integrated into a single image by appropriate fusion method. For decades, these fusion methods achieve extraordinary fusion performance and are widely used in many applications, like video surveillance and military applications.

1.1 Objectives

The task is to propose a novel deep learning architecture which is constructed by encoding network and decoding network. We use encoding network to extract image features and the fused image is obtained by decoding network. The encoding network is constructed by convolutional layer and dense block in which the output of each layer is used as the input of next layer. So in our deep learning architecture, the results of each layer in encoding network are utilized to construct feature maps.

1.2 Methodology

To fuse infrared and visible images a large collection of the images is required. The images are downloaded from the Mendeley database Powerline Image Dataset. In this section the methodology followed is discussed in detail.

1.2.1 Dataset

The dataset for the experiment is downloaded from the Mendeley database Powerline Image Dataset which contains different Infrared-IR and Visible Light-VL images and their labels. It contains a collection of images taken and the images were captured from 21 different regions all over Turkey at different seasonal days. Due to varying background behavior, varying temperatures and weather conditions, and varying lighting conditions, the achieved positive set contains several difficult scenes where low contrast causes close-to invisibility for power lines. The original video resolutions were 576x325 for IR and full HD for VL, however, the captured frames were scaled down to smaller sizes and the effect of resizing was tested for various image sizes. An image size of 128x128 is sufficient for consistently accurate power line recognition.

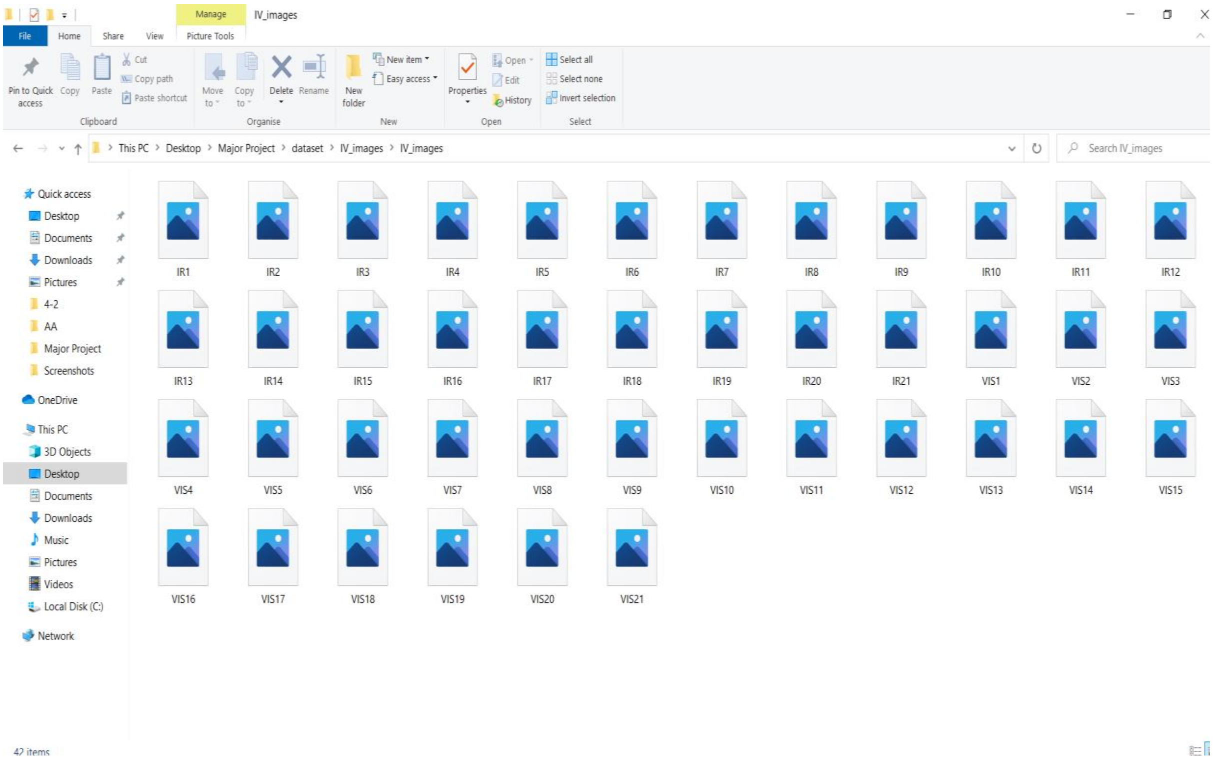


Fig 1.2.1: Dataset

Introduction

In traditional CNN based network, with the increase of network depth, a degradation problem has been exposed and the information which is extracted by middle layers is not used thoroughly. To address the degradation problem, we introduced a deep residual learning framework. To further improve the information flow between layers, we proposed a novel architecture with dense block in which direct connections from any layer to all the subsequent layers are used.

Training the network

In training phase, we just consider encoder and decoder networks (fusion layer is discarded), in which we attempt to train our encoder and decoder networks to reconstruct the input image. After the encoder and decoder weights are fixed, we use adaptive fusion strategy to fuse the deep features which are obtained by encoder.

The detailed framework of our network work in training phase is shown in Fig.2.2.1, and the architecture of our network is outlined in Table I

TABLE I

THE ARCHITECTURE OF TRAINING PROCESS. ***Conv*** DENOTES THE CONVOLUTIONAL BLOCK(CONVOLUTIONAL LAYER + ACTIVATION); ***Dense*** DENOTES THE DENSE BLOCK

	Layer	Size	Stride	Channel (input)	Channel (output)	Activation
Encoder	Conv(C1)	3	1	1	16	ReLu
	Dense					
Decoder	Conv(C2)	3	1	64	64	ReLu
	Conv(C3)	3	1	64	32	ReLu
	Conv(C4)	3	1	32	16	ReLu
	Conv(C5)	3	1	16	1	
Dense (dense block)	Conv(DC1)	3	1	16	16	ReLu
	Conv(DC2)	3	1	32	16	ReLu
	Conv(DC3)	3	1	48	16	ReLu

The apparent advantage of this training strategy is that, we can design appropriate fusion layer for specific fusion tasks. Also, It leaves more space for further development of fusion layer.

1.2.2 The proposed model

The proposed model is a novel deep learning architecture which is constructed by encoding network and decoding network. We use encoding network to extract image features and the fused image is obtained by decoding network. The encoding network is constructed by convolutional layer and dense block in which the output of each layer is used as the input of next layer. So in our deep learning architecture, the results of each layer in encoding network are utilized to construct feature maps. Finally, the fused image will be reconstructed by fusion strategy and decoding network which includes four CNN layers.

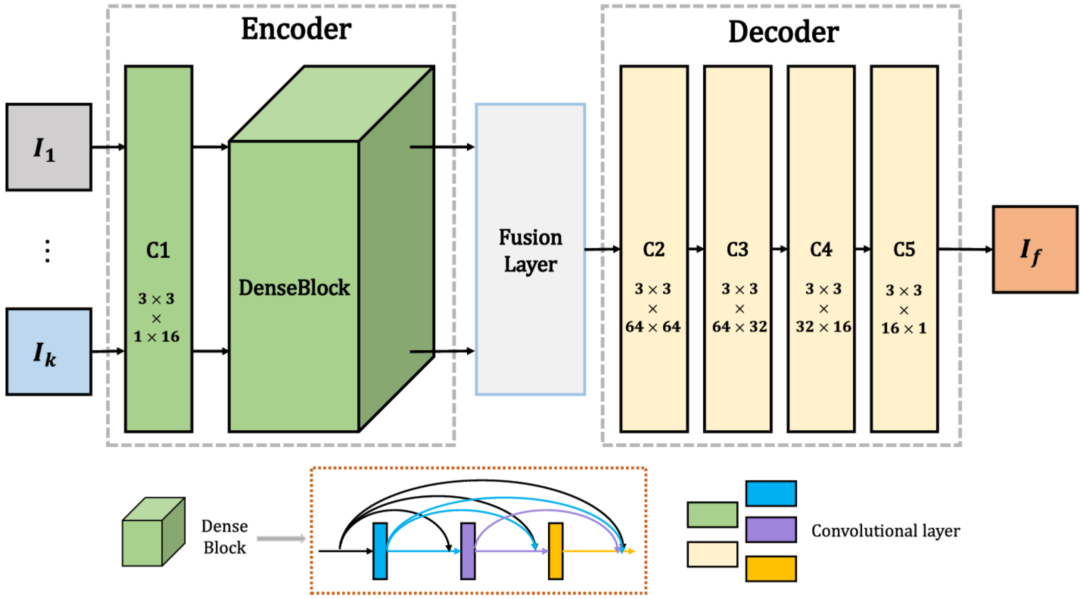


Fig 1.2.2: The architecture of proposed method

A novel deep learning architecture based on CNN layers and dense block. In our network, we use infrared and visible image pairs as inputs for our method. And in dense block, their feature maps which are obtained by each layer in encoding network are cascaded as the input of the next layer.

We use the method Multi-temporal remote sensing image registration using deep convolutional features, to pre-process input images if they are not registered. Our network architecture has three parts: encoder, fusion layer, and decoder. The architecture of the proposed network is shown in Fig.1.2.2.

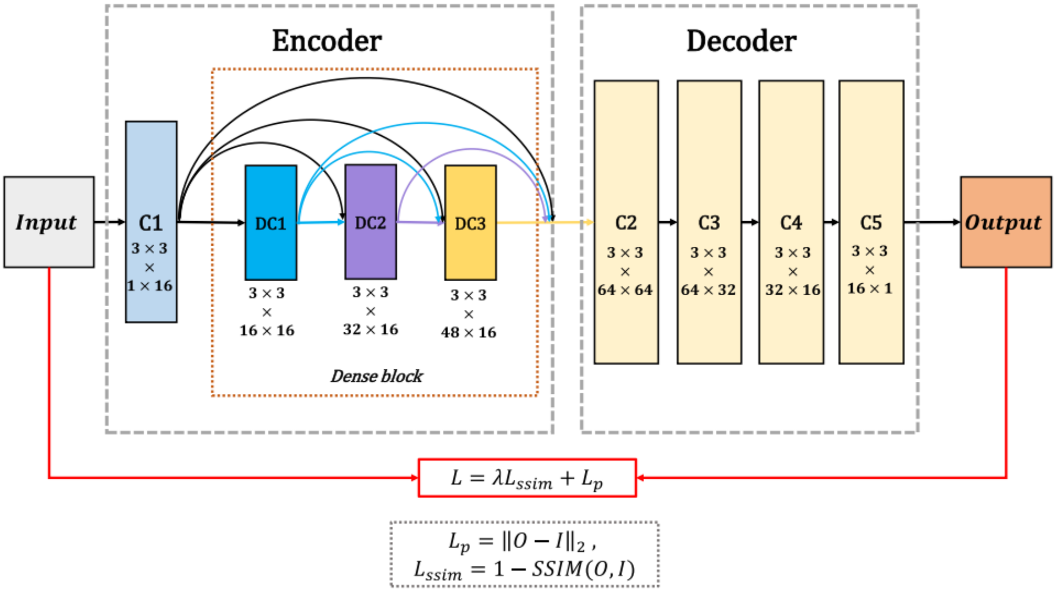


Fig 1.2.2.1: The framework of training process

In Fig.1.2.2.1 and Table I, C1 is convolution layer in encoder network which contains 3 × 3 filters. DC1, DC2 and DC3 are convolution layers in dense block and the output of each layer is connected to every other layer by cascade operation. The decoder consists of C2, C3, C4 and C5, which will be utilized to reconstruct the input image. In order to reconstruct the input image more precisely, we minimize the loss function L to train our encoder and decoder,

$$\mathbf{L = \lambda Lssim + L\ p}$$

which is a weighted combination of pixel loss Lp and structural similarity (SSIM) loss Lssim with the weight λ

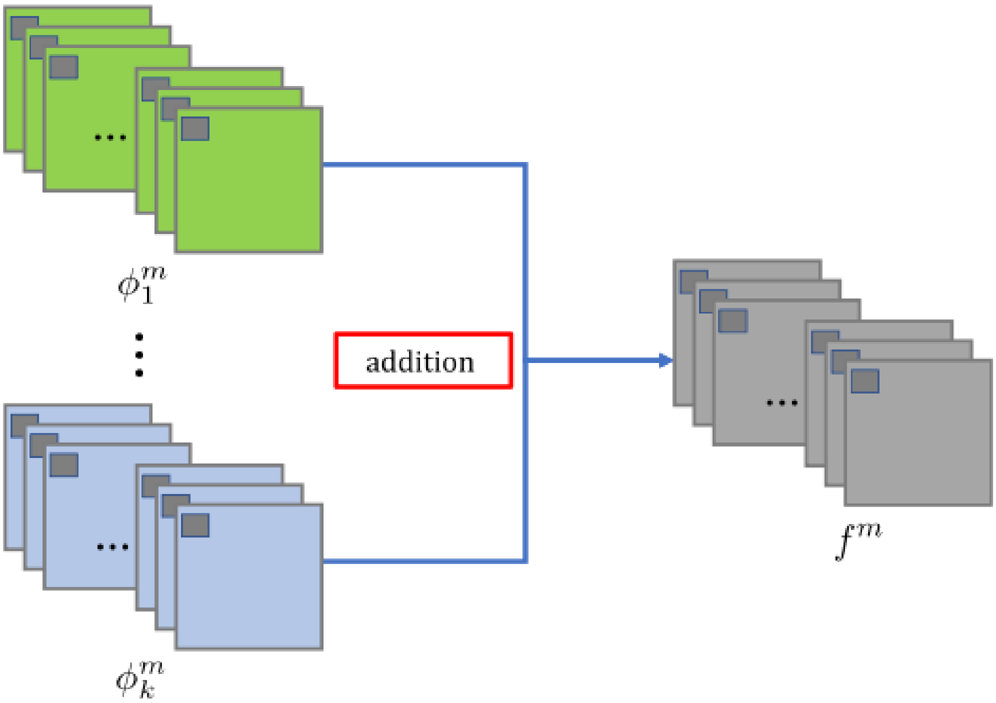


Fig 1.2.2.2: The procedure of addition strategy

The pixel loss Lp is calculated as,

$$\mathbf{L\ p = ||O - I||2}$$

where O and I indicate the output and input images, respectively. It is the Euclidean distance between the output O and the input I.

The SSIM loss Lssim is obtained by,

$$\mathbf{Lssim = 1 - SSIM(O, I)}$$

where SSIM(·) represents the structural similarity operation and it denotes the structural similarity of two images. Because there are three orders of magnitude difference between pixel loss and SSIM loss, in training phase, so the λ is set as 1, 10, 100 and 1000, respectively.

The aim of training phase is to train an auto encoder network (encoder, decoder) which has better feature extraction and reconstruction ability. Due to the training data of infrared and visible images is insufficient, we use gray scale images of MS-COCO to train our model.

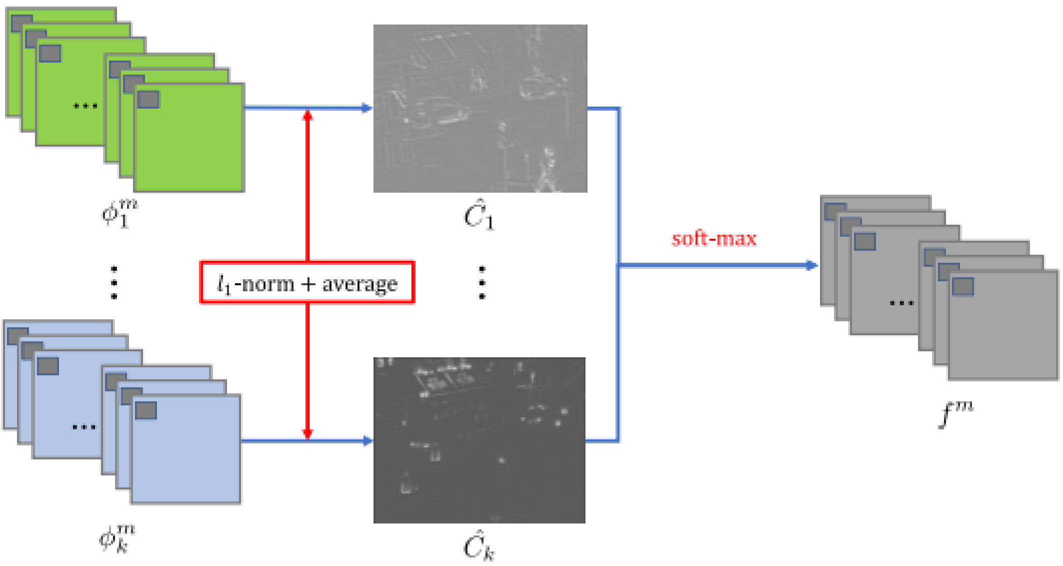


Fig 1.2.2.3 The diagram of l1-norm and soft-max strategy.

Fusion Layer (Strategy)

Addition Strategy: The addition fusion strategy procedure is shown in Fig.1.2.2.2. As shown in Fig, once encoder and decoder networks are fixed, in testing phase, two input images are fed into encoder, respectively. We choose two fusion strategies (addition strategy and l1-norm strategy) to combine salient feature maps which are obtained by encoder. In our network, $m \in \{1, 2, \cdots, M\}$, $M = 64$ represents the number of feature maps. $k \geq 2$ indicates the index of feature maps which are obtained from input images.

l1-Norm Strategy: The performance of this strategy is shown in fig 1.2.2.3. But this operation is a very rough fusion strategy for salient feature selection. We applied a new strategy which is based on l1-norm and soft-max operation into our network.

1.3 Organization of Project

The encoder contains two parts (C1 and DenseBlock) which are utilized to extract deep features. The first layer (C1) contains 3 × 3 filters to extract rough features and the dense block (DenseBlock) contains three convolutional layers (each layer's output is cascaded as the input of the next layer) which also contain 3 × 3 filters. And in our network, the reflection mode is used to pad input images. For each convolutional layer in encoding network, the input channel number of feature maps is 16. The architecture of encoder has two advantages. First, the filter size and stride of convolutional operation are 3×3 and 1, respectively. With this strategy, the input image can be any size. Second, dense block architecture can preserve deep features as much as possible in encoding network and this operation can make sure that all the salient features are used in fusion strategy.

2. THEORETICAL ANALYSIS OF THE PROPOSED PROJECT

2.1 Requirements Gathering

2.1.1 Software Requirements

Programming Language : Python 3.6

Graphical User Interface: Tkinter

Dataset : Powerline Image Dataset

Packages :Opencv-python,Opencv-contrib-python,Tensorflow, Numpy, Pandas, Matplotlib, Scikit-learn, imutils, pillow

Operating System : Windows 10

Tool : Pycharm

2.1.2 Hardware Requirements

Processor : Intel Core i3

CPU Speed : 2.30 GHz

Memory : 2 GB (RAM)

2.2 Technologies Description

Python

Python is an interpreted high-level programming language for general-purpose programming, created by Guido van Rossum and first released in 1991. Python has a design philosophy that emphasizes code readability, notably using significant whitespace.

Python features a dynamic type system and automatic memory management. It supports multiple programming paradigms, including object-oriented, imperative, functional and procedural, and has a large and comprehensive standard library.

- Python is Interpreted − Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
- Python is Interactive − you can actually sit at a Python prompt and interact with the interpreter directly to write your programs.

Python also acknowledges that speed of development is important. Readable and terse code is part of this, and so is access to powerful constructs that avoid tedious repetition of code. Maintainability also ties into this may be an all but useless metric, but it does say something about how much code you have to scan, read and/or understand to troubleshoot problems or tweak behaviors. This speed of development, the ease with which a programmer of other languages can pick up basic Python skills and the huge standard library is a key to another area where Python excels. All its tools have been quick to implement, saved a lot of time, and several of them have later been patched and updated by people with no Python background - without breaking.

Tkinter

Tkinter is the most commonly used library for developing GUI (Graphical User Interface) in Python. It is a standard Python interface to the Tk GUI toolkit shipped with Python. As Tk and Tkinter are available on most of the unix platforms as well as on the Windows system, developing GUI applications with Tkinter becomes the fastest and easiest.

Most of the time, tkinter is all you really need, but a number of additional modules are available as well. The Tk interface is located in a binary module named _tkinter. This module contains the low-level interface to Tk, and should never be used directly by application programmers. It is usually a shared library (or DLL), but might in some cases be statically linked with the Python interpreter.

In addition to the Tk interface module, Tkinter includes a number of Python modules, tkinter.constants being one of the most important.

Powerline Image Dataset

The dataset for the experiment is downloaded from the Mendeley database Powerline Image Dataset which contains different Infrared-IR and Visible Light-VL images and their labels. It contains a collection of images taken and the images were captured from 21 different regions all over Turkey at different seasonal days. Due to varying background behavior, varying temperatures and weather conditions, and varying lighting conditions, the achieved positive set contains several difficult scenes where low contrast causes close-to invisibility for power lines. The original video resolutions were 576x325 for IR and full HD for VL, however, the captured frames were scaled down to smaller sizes and the effect of resizing was tested for various image sizes. An image size of 128x128 is sufficient for consistently accurate power line recognition.

OpenCV-Python

OpenCV-Python is a library of Python bindings designed to solve computer vision problems. OpenCV-Python makes use of **Numpy**, which is a highly optimized library for numerical operations with MATLAB-style syntax. All the OpenCV array structures are converted to and from Numpy arrays. This also makes it easier to integrate with other libraries that use Numpy such as SciPy and Matplotlib.

Tensorflow

TensorFlow is a free and open-source software library for dataflow and differentiable programming across a range of tasks. It is a symbolic math library, and is also used for machine learning applications such as neural networks. It is used for both research and production at Google.

TensorFlow was developed by the Google Brain team for internal Google use. It was released under the Apache 2.0 open-source license on November 9, 2015.

Numpy

Numpy is a general-purpose array-processing package. It provides a high-performance multidimensional array object, and tools for working with these arrays.

It is the fundamental package for scientific computing with Python. It contains various features including these important ones:

- A powerful N-dimensional array object
- Sophisticated (broadcasting) functions
- Tools for integrating C/C++ and Fortran code
- Useful linear algebra, Fourier transform, and random number capabilities

Besides its obvious scientific uses, Numpy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined using Numpy which allows Numpy to seamlessly and speedily integrate with a wide variety of databases.

Pandas

Pandas is an open-source Python Library providing high-performance data manipulation and analysis tool using its powerful data structures. Python was majorly used for data munging and preparation. It had very little contribution towards data analysis. Pandas solved this problem. Using Pandas, we can accomplish five typical steps in the processing and analysis of data, regardless of the origin of data load, prepare, manipulate, model, and analyze. Python with Pandas is used in a wide range of fields including academic and commercial domains including finance, economics, Statistics, analytics, etc.

Matplotlib

Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and IPython shells, the Jupyter Notebook, web application servers, and four graphical user interface toolkits. Matplotlib tries to make easy things easy and hard things possible. You can generate plots, histograms, power spectra, bar charts, error charts, scatter plots, etc., with just a few lines of code. For examples, see the sample plots and thumbnail gallery.

For simple plotting the pyplot module provides a MATLAB-like interface, particularly when combined with IPython. For the power user, you have full control of line styles, font properties, axes properties, etc, via an object oriented interface or via a set of functions familiar to MATLAB users.

Scikit – learn

Scikit-learn provides a range of supervised and unsupervised learning algorithms via a consistent interface in Python. It is licensed under a permissive simplified BSD license and is distributed under many Linux distributions, encouraging academic and commercial use. The library is built upon the SciPy (Scientific Python) that must be installed before you can use scikit-learn. This stack that includes:

- **NumPy**: Base n-dimensional array package
- **SciPy**: Fundamental library for scientific computing
- **Matplotlib**: Comprehensive 2D/3D plotting
- **IPython**: Enhanced interactive console
- **Sympy**: Symbolic mathematics
- **Pandas**: Data structures and analysis
- Extensions or modules for SciPy care conventionally named SciKits. As such, the module provides learning algorithms and is named scikit-learn.

Imutils

Imutils are a series of convenience functions to make basic image processing functions such as translation, rotation, resizing, skeletonization, and displaying Matplotlib images easier with OpenCV and both **Python** 2.7 and **Python** 3

Pillow

Pillow is the friendly PIL fork by Alex Clark and Contributors. PIL is the python imaging Library adds images processing capabilities to your python interpreter.
The core image library is designed for fast access to data stored in a few basic pixel formats. It should provide a solid foundation for a general image processing tool.

Pycharm

PyCharm is an integrated development environment (IDE) used in computer programming, specifically for the Python language. It is developed by the Czech company JetBrains (formerly known as IntelliJ). It provides code analysis, a graphical debugger, an integrated unit tester, integration with version control systems (VCSes), and supports web development with Django as well as data science with Anaconda.
PyCharm is cross-platform, with Windows, macOS and Linux versions. PyCharm provides smart code completion, code inspections, on-the-fly error highlighting and quick-fixes, along with automated code refactorings and rich navigation capabilities.

PyCharm's huge collection of tools available out of the box includes an integrated debugger and test runner, a Python profiler, a built-in terminal, integration with major VCS and built-in database tools, remote development capabilities with remote interpreters, an integrated ssh terminal, and integration with Docker and Vagrant.
PyCharm's huge collection of tools available out of the box includes an integrated debugger and test runner; a Python profiler; a built-in terminal; integration with major VCS and built-in database tools; remote development capabilities with remote interpreters; an integrated ssh terminal; and integration with Docker and Vagrant.
PyCharm provides extensive options for debugging your Python/Django and JavaScript code:

- Set breakpoints right inside the editor and define hit conditions
- Inspect context-relevant local variables and user-defined watches, including arrays and complex objects, and edit values on the fly

3. DESIGN

3.1 Introduction

Software design sits at the technical kernel of the software engineering process and is applied regardless of the development paradigm and area of application. Design is the first step in the development phase for any engineered product or system. The designer's goal is to produce a model or representation of an entity that will later be built. Beginning, once system requirement have been specified and analyzed, system design is the first of the three technical activities -design, code and test that is required to build and verify software.

The importance can be stated with a single word "Quality". Design is the place where quality is fostered in software development. Design provides us with representations of software that can assess for quality. Design is the only way that we can accurately translate a customer's view into a finished software product or system. Software design serves as a foundation for all the software engineering steps that follow. Without a strong design we risk building an unstable system – one that will be difficult to test, one whose quality cannot be assessed until the last stage.

During design, progressive refinement of data structure, program structure, and procedural details are developed reviewed and documented. System design can be viewed from either technical or project management perspective. From the technical point of view, design is comprised of four activities – architectural design, data structure design, interface design and procedural design.

3.2 Architecture Diagram

Web applications are by nature distributed applications, meaning that they are programs that run on more than one computer and communicate through network or server. Specifically, web applications are accessed with a web browser and are popular because of the ease of using the browser as a user client. For the enterprise, software on potentially thousands of client computers is a key reason for their popularity. Web applications are used for web mail, online retail sales, discussion boards, weblogs, online banking, and more. One web application can be accessed and used by millions of people.

Like desktop applications, web applications are made up of many parts and often contain mini programs and some of which have user interfaces. In addition, web applications frequently require an additional markup or scripting language, such as HTML, CSS, or JavaScript programming language. Also, many applications use only the Python programming language, which is ideal because of its versatility.

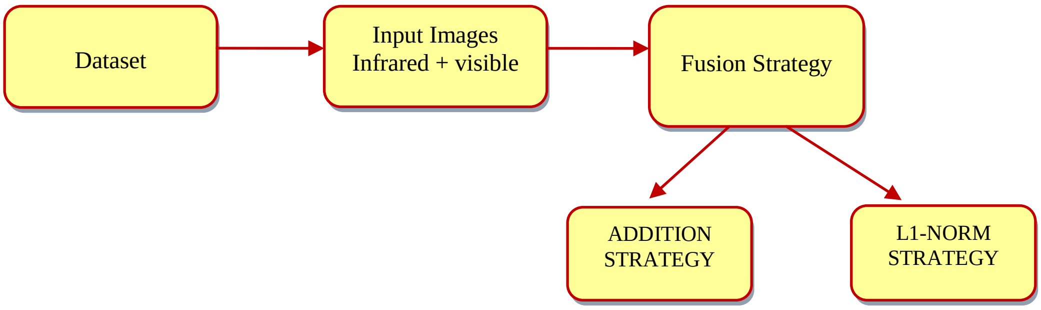


Fig 3.2: Architecture Diagram

3.3 UML Diagrams

3.3.1 Use Case Diagram

To model a system, the most important aspect is to capture the dynamic behavior. Dynamic behavior means the behavior of the system when it is running/operating.

Only static behavior is not sufficient to model a system rather dynamic behavior is more important than static behavior. In UML, there are five diagrams available to model the dynamic nature and use case diagram is one of them. Now as we have to discuss that the use case diagram is dynamic in nature, there should be some internal or external factors for making the interaction.

These internal and external agents are known as actors. Use case diagrams consist of actors, use cases and their relationships. The diagram is used to model the system/subsystem of an application. A single use case diagram captures a particular functionality of a system.

Hence to model the entire system, a number of use case diagrams are used.

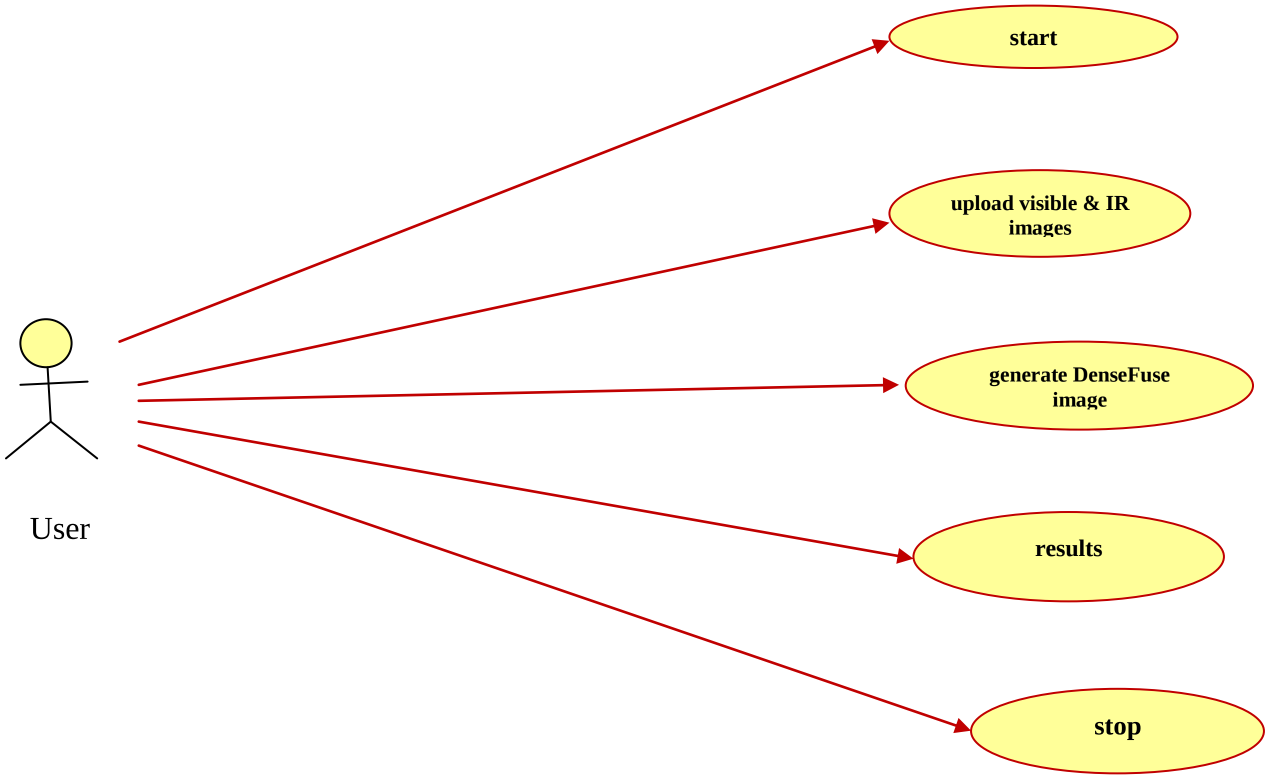


Fig 3.2.1: Use Case Diagram

3.3.2 Sequence Diagram

Sequence Diagrams represent the objects participating the interaction horizontally and time vertically. A Use Case is a kind of behavioral classifier that represents a declaration of an offered behavior. Each use case specifies some behavior, possibly including variants that the subject can perform in collaboration with one or more actors. Use cases define the offered behavior of the subject without reference to its internal structure. These behaviors, involving interactions between the actor and the subject, may result in changes to the state of the subject and communications with its environment. A use case can include possible variations of its basic behavior, including exceptional behavior and error handling.

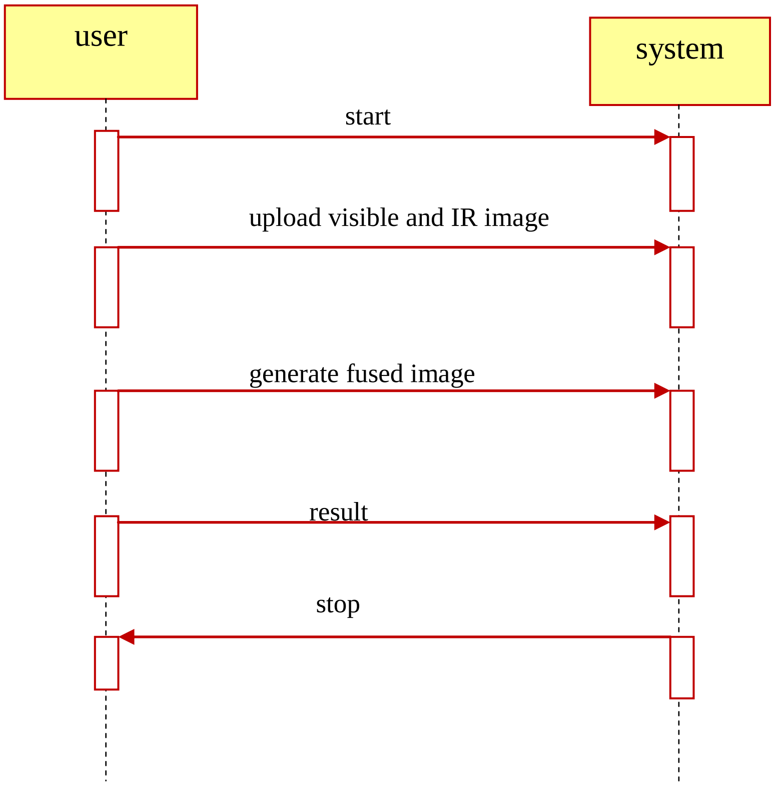


Fig 3.2.2: Sequence Diagram

3.3.3 Activity Diagram

Activity diagrams are graphical representations of workflows of stepwise activities and actions with support for choice, iteration and concurrency. In the Unified Modeling Language, activity diagrams can be used to describe the business and operational step-by-step workflows of components in a system. An activity diagram shows the overall flow of control.

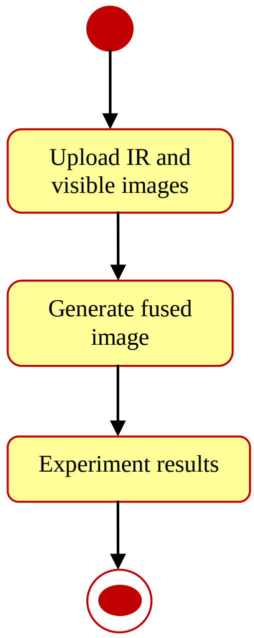


Fig 3.2.3: Activity Diagram

3.3.4 Collaboration Diagram

A collaboration diagram resembles a flowchart that portrays the roles, functionality and behavior of individual objects as well as the overall operation of the system in real time. Objects are shown as rectangles with naming labels inside. These labels are preceded by colons and may be underlined. The relationships between the objects are shown as lines connecting the rectangles. The messages between objects are shown as arrows connecting the relevant rectangles along with labels that define the message sequencing.

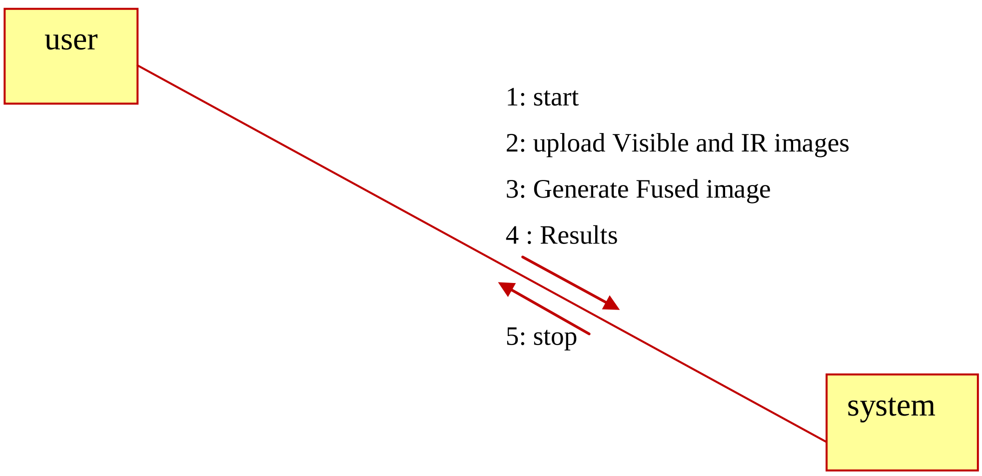


Fig 3.2.4:Collabration Diagram

3.3.5 Class Diagram

The class diagram is the main building block of object-oriented modeling. It is used for general conceptual modeling of the systematic of the application, and for detailed modeling translating the models into programming code. Class diagrams can also be used for data modeling. The classes in a class diagram represent both the main elements, interactions in the application, and the classes to be programmed.

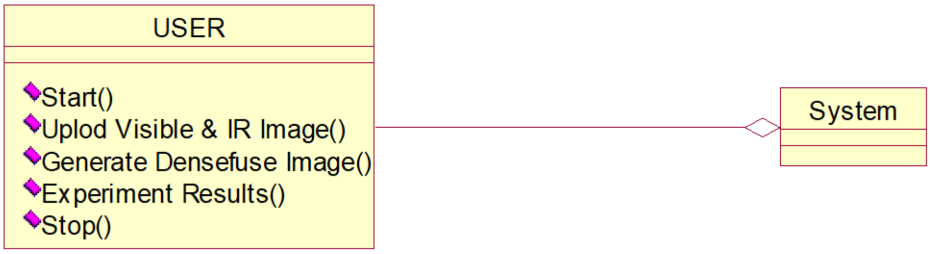


Fig 3.2.5: Class Diagram

4. IMPLEMENTATION

4.1 Coding

DenseFuse.py

```
from __future__ import print_function
from tkinter import *
import tkinter
from tkinter import filedialog
import numpy as np
from tkinter.filedialog import askopenfilename
import pandas as pd
from tkinter import simpledialog
import numpy as np
from train_recons import train_recons
from generate import generate
from utils import list_images
import cv2

main = tkinter.Tk()
main.title("DenseFuse: A Fusion Approach to Infrared and Visible Images")
#designing main screen
main.geometry("800x700")

global filename
SSIM_WEIGHTS = [1, 10, 100, 1000]
MODEL_SAVE_PATHS = [
    './models/densefuse_gray/densefuse_model_bs2_epoch4_all_weight_1e0.ckpt',
    './models/densefuse_gray/densefuse_model_bs2_epoch4_all_weight_1e1.ckpt',
    './models/densefuse_gray/densefuse_model_bs2_epoch4_all_weight_1e2.ckpt',
    './models/densefuse_gray/densefuse_model_bs2_epoch4_all_weight_1e3.ckpt',
]


def upload(): #function to upload
    global filename
    filename = filedialog.askdirectory(initialdir="testImages")
    textarea.insert(END,filename+" loaded\n")

def denseFuse():
    ssim_weight = SSIM_WEIGHTS[0]
    model_path = MODEL_SAVE_PATHS[2]
    infrared = filename + '/IR.png'
    visible = filename + '/VIS.png'
    fusion_type = 'addition'
    output_save_path = 'outputs'
    generate(infrared, visible, model_path, None, ssim_weight, 0, False, False, type =
fusion_type, output_path = output_save_path)
    cv2.imshow("Infrared Image",cv2.imread(infrared))
    cv2.imshow("Visible Image",cv2.imread(visible))
```

```
    cv2.waitKey(0)


def exit():
    global main
    main.destroy()

font = ('times', 16, 'bold')
title = Label(main, text='DenseFuse: A Fusion Approach to Infrared and Visible
Images', justify=LEFT)
title.config(bg='lavender blush', fg='maroon')
title.config(font=font)
title.config(height=3, width=220)
title.place(x=250,y=8)
title.pack()

font1 = ('times', 14, 'bold')
model = Button(main, text="Upload Visible & IR Image", command=upload)
model.place(x=200,y=100)
model.config(font=font1)

uploadimage = Button(main, text="Generate DenseFuse Image",
command=denseFuse)
uploadimage.place(x=200,y=150)
uploadimage.config(font=font1)

exitapp = Button(main, text="Exit", command=exit)
exitapp.place(x=200,y=200)
exitapp.config(font=font1)

font1 = ('times', 12, 'bold')
textarea=Text(main,height=35,width=300)
scroll=Scrollbar(textarea)
textarea.configure(yscrollcommand=scroll.set)
textarea.place(x=10,y=300)
textarea.config(font=font1)

main.config(bg='MistyRose4')
main.mainloop()
```

encoder.py

```
import tensorflow as tf
from tensorflow.python import pywrap_tensorflow

WEIGHT_INIT_STDDEV = 0.1
DENSE_layers = 3
DECAY = .9
EPSILON = 1e-8

class Encoder(object):
    def __init__(self, model_pre_path):
        self.weight_vars = []
        self.model_pre_path = model_pre_path

        with tf.variable_scope('encoder'):
            self.weight_vars.append(self._create_variables(1, 16, 3, scope='conv1_1'))

            self.weight_vars.append(self._create_variables(16, 16, 3,
scope='dense_block_conv1'))
            self.weight_vars.append(self._create_variables(32, 16, 3,
scope='dense_block_conv2'))
            self.weight_vars.append(self._create_variables(48, 16, 3,
scope='dense_block_conv3'))

            # self.weight_vars.append(self._create_variables(64, 32, 3, scope='conv1_2'))

    def _create_variables(self, input_filters, output_filters, kernel_size, scope):
        shape = [kernel_size, kernel_size, input_filters, output_filters]
        if self.model_pre_path:
            reader = pywrap_tensorflow.NewCheckpointReader(self.model_pre_path)
            with tf.variable_scope(scope):
                kernel = tf.Variable(reader.get_tensor('encoder/' + scope + '/kernel'),
name='kernel')
                bias = tf.Variable(reader.get_tensor('encoder/' + scope + '/bias'),
name='bias')
        else:
            with tf.variable_scope(scope):
                kernel = tf.Variable(tf.truncated_normal(shape,
stddev=WEIGHT_INIT_STDDEV), name='kernel')
                bias = tf.Variable(tf.zeros([output_filters]), name='bias')
        return (kernel, bias)

    def encode(self, image):
        dense_indices = (1, 2, 3)
        final_layer_idx = len(self.weight_vars) - 1

        out = image
        for i in range(len(self.weight_vars)):
            kernel, bias = self.weight_vars[i]
```

```
        # if i == final_layer_idx:
        #     out = transition_block(out, kernel, bias)
        # el
        if i in dense_indices:
            out = conv2d_dense(out, kernel, bias, use_relu=True)
        else:
            out = conv2d(out, kernel, bias, use_relu=True)


    return out


def conv2d(x, kernel, bias, use_relu=True):
    # padding image with reflection mode
    x_padded = tf.pad(x, [[0, 0], [1, 1], [1, 1], [0, 0]], mode='REFLECT')

    # conv and add bias
    # num_maps = x_padded.shape[3]
    # out = __batch_normalize(x_padded, num_maps)
    # out = tf.nn.relu(out)
    out = tf.nn.conv2d(x_padded, kernel, strides=[1, 1, 1, 1], padding='VALID')
    out = tf.nn.bias_add(out, bias)
    out = tf.nn.relu(out)

    return out


def conv2d_dense(x, kernel, bias, use_relu=True):
    # padding image with reflection mode
    x_padded = tf.pad(x, [[0, 0], [1, 1], [1, 1], [0, 0]], mode='REFLECT')

    # conv and add bias
    # num_maps = x_padded.shape[3]
    # out = __batch_normalize(x_padded, num_maps)
    # out = tf.nn.relu(out)
    out = tf.nn.conv2d(x_padded, kernel, strides=[1, 1, 1, 1], padding='VALID')
    out = tf.nn.bias_add(out, bias)
    out = tf.nn.relu(out)
    # concatenate
    out = tf.concat([out, x], 3)

    return out


def transition_block(x, kernel, bias):

    num_maps = x.shape[3]
    out = __batch_normalize(x, num_maps)
    out = tf.nn.relu(out)
    out = conv2d(out, kernel, bias, use_relu=False)
```

```
    return out


def __batch_normalize(inputs, num_maps, is_training=True):
    # Trainable variables for scaling and offsetting our inputs
    # scale = tf.Variable(tf.ones([num_maps], dtype=tf.float32))
    # offset = tf.Variable(tf.zeros([num_maps], dtype=tf.float32))

    # Mean and variances related to our current batch
    batch_mean, batch_var = tf.nn.moments(inputs, [0, 1, 2])

    # # Create an optimizer to maintain a 'moving average'
    # ema = tf.train.ExponentialMovingAverage(decay=DECAY)
    #
    # def ema_retrieve():
    #     return ema.average(batch_mean), ema.average(batch_var)
    #
    # # If the net is being trained, update the average every training step
    # def ema_update():
    #     ema_apply = ema.apply([batch_mean, batch_var])
    #
    #     # Make sure to compute the new means and variances prior to returning their
values
    #     with tf.control_dependencies([ema_apply]):
    #         return tf.identity(batch_mean), tf.identity(batch_var)
    #
    # # Retrieve the means and variances and apply the BN transformation
    # mean, var = tf.cond(tf.equal(is_training, True), ema_update, ema_retrieve)
    bn_inputs = tf.nn.batch_normalization(inputs, batch_mean, batch_var, None, None,
EPSILON)

    return bn_inputs
```

decoder.py

```
import tensorflow as tf
from tensorflow.python import pywrap_tensorflow

WEIGHT_INIT_STDDEV = 0.1


class Decoder(object):

    def __init__(self, model_pre_path):
        self.weight_vars = []
        self.model_pre_path = model_pre_path
```

```
        with tf.variable_scope('decoder'):

            self.weight_vars.append(self._create_variables(64, 64, 3, scope='conv2_1'))
            self.weight_vars.append(self._create_variables(64, 32, 3, scope='conv2_2'))
            self.weight_vars.append(self._create_variables(32, 16, 3, scope='conv2_3'))
            self.weight_vars.append(self._create_variables(16, 1 , 3, scope='conv2_4'))

    def _create_variables(self, input_filters, output_filters, kernel_size, scope):

        if self.model_pre_path:
            reader = pywrap_tensorflow.NewCheckpointReader(self.model_pre_path)
            with tf.variable_scope(scope):
                kernel = tf.Variable(reader.get_tensor('decoder/' + scope + '/kernel'),
name='kernel')
                bias = tf.Variable(reader.get_tensor('decoder/' + scope + '/bias'),
name='bias')
        else:
            with tf.variable_scope(scope):
                shape = [kernel_size, kernel_size, input_filters, output_filters]
                kernel = tf.Variable(tf.truncated_normal(shape,
stddev=WEIGHT_INIT_STDDEV), name='kernel')
                bias = tf.Variable(tf.zeros([output_filters]), name='bias')
        return (kernel, bias)

    def decode(self, image):
        final_layer_idx  = len(self.weight_vars) - 1

        out = image
        for i in range(len(self.weight_vars)):
            kernel, bias = self.weight_vars[i]

            if i == final_layer_idx:
                out = conv2d(out, kernel, bias, use_relu=False)
            else:
                out = conv2d(out, kernel, bias)
            # print('decoder ', i)
            # print('decoder out:', out.shape)
        return out


def conv2d(x, kernel, bias, use_relu=True):
    # padding image with reflection mode
    x_padded = tf.pad(x, [[0, 0], [1, 1], [1, 1], [0, 0]], mode='REFLECT')

    # conv and add bias
    out = tf.nn.conv2d(x_padded, kernel, strides=[1, 1, 1, 1], padding='VALID')
    out = tf.nn.bias_add(out, bias)

    if use_relu:
```

```
        out = tf.nn.relu(out)

    return out
```

generate.py

```
# Use a trained DenseFuse Net to generate fused images

import tensorflow as tf
import numpy as np
from datetime import datetime

from fusion_l1norm import L1_norm
from densefuse_net import DenseFuseNet
from utils import get_images, save_images, get_train_images, get_test_image_rgb


def generate(infrared_path, visible_path, model_path, model_pre_path, ssim_weight,
index, IS_VIDEO, IS_RGB, type='addition', output_path=None):

  if IS_VIDEO:
    print('video_addition')
    _handler_video(infrared_path, visible_path, model_path, model_pre_path,
ssim_weight, output_path=output_path)
  else:
    if IS_RGB:
      print('RGB - addition')
      _handler_rgb(infrared_path, visible_path, model_path, model_pre_path,
ssim_weight, index,
            output_path=output_path)

      print('RGB - l1')
      _handler_rgb_l1(infrared_path, visible_path, model_path, model_pre_path,
ssim_weight, index,
              output_path=output_path)
    else:
      if type == 'addition':
        print('addition')
        _handler(infrared_path, visible_path, model_path, model_pre_path,
ssim_weight, index, output_path=output_path)
      elif type == 'l1':
        print('l1')
        _handler_l1(infrared_path, visible_path, model_path, model_pre_path,
ssim_weight, index, output_path=output_path)


def _handler(ir_path, vis_path, model_path, model_pre_path, ssim_weight, index,
```

```
output_path=None):
    ir_img = get_train_images(ir_path, flag=False)
    vis_img = get_train_images(vis_path, flag=False)
    # ir_img = get_train_images_rgb(ir_path, flag=False)
    # vis_img = get_train_images_rgb(vis_path, flag=False)
    dimension = ir_img.shape

    ir_img = ir_img.reshape([1, dimension[0], dimension[1], dimension[2]])
    vis_img = vis_img.reshape([1, dimension[0], dimension[1], dimension[2]])

    ir_img = np.transpose(ir_img, (0, 2, 1, 3))
    vis_img = np.transpose(vis_img, (0, 2, 1, 3))

    print('img shape final:', ir_img.shape)

    with tf.Graph().as_default(), tf.Session() as sess:
        infrared_field = tf.placeholder(
            tf.float32, shape=ir_img.shape, name='content')
        visible_field = tf.placeholder(
            tf.float32, shape=ir_img.shape, name='style')

        dfn = DenseFuseNet(model_pre_path)

        output_image = dfn.transform_addition(infrared_field, visible_field)
        # restore the trained model and run the style transferring
        saver = tf.train.Saver()
        saver.restore(sess, model_path)

        output = sess.run(output_image, feed_dict={infrared_field: ir_img, visible_field:
vis_img})

        save_images(ir_path, output, output_path,
                    prefix='fused' + str(index),
suffix='_densefuse_addition_'+str(ssim_weight))


def _handler_l1(ir_path, vis_path, model_path, model_pre_path, ssim_weight, index,
output_path=None):
    ir_img = get_train_images(ir_path, flag=False)
    vis_img = get_train_images(vis_path, flag=False)
    dimension = ir_img.shape

    ir_img = ir_img.reshape([1, dimension[0], dimension[1], dimension[2]])
    vis_img = vis_img.reshape([1, dimension[0], dimension[1], dimension[2]])

    ir_img = np.transpose(ir_img, (0, 2, 1, 3))
    vis_img = np.transpose(vis_img, (0, 2, 1, 3))

    print('img shape final:', ir_img.shape)
```

```
    with tf.Graph().as_default(), tf.Session() as sess:

        # build the dataflow graph
        infrared_field = tf.placeholder(
            tf.float32, shape=ir_img.shape, name='content')
        visible_field = tf.placeholder(
            tf.float32, shape=ir_img.shape, name='style')

        dfn = DenseFuseNet(model_pre_path)

        enc_ir = dfn.transform_encoder(infrared_field)
        enc_vis = dfn.transform_encoder(visible_field)

        target = tf.placeholder(
             tf.float32, shape=enc_ir.shape, name='target')

        output_image = dfn.transform_decoder(target)

        # restore the trained model and run the style transferring
        saver = tf.train.Saver()
        saver.restore(sess, model_path)

        enc_ir_temp, enc_vis_temp = sess.run([enc_ir, enc_vis],
feed_dict={infrared_field: ir_img, visible_field: vis_img})
        feature = L1_norm(enc_ir_temp, enc_vis_temp)

        output = sess.run(output_image, feed_dict={target: feature})
        save_images(ir_path, output, output_path,
                    prefix='fused' + str(index),
suffix='_densefuse_l1norm_'+str(ssim_weight))


def _handler_video(ir_path, vis_path, model_path, model_pre_path, ssim_weight,
output_path=None):
    infrared = ir_path[0]
    img = get_train_images(infrared, flag=False)
    img = img.reshape([1, img.shape[0], img.shape[1], img.shape[2]])
    img = np.transpose(img, (0, 2, 1, 3))
    print('img shape final:', img.shape)
    num_imgs = len(ir_path)

    with tf.Graph().as_default(), tf.Session() as sess:
        # build the dataflow graph
        infrared_field = tf.placeholder(
            tf.float32, shape=img.shape, name='content')
        visible_field = tf.placeholder(
            tf.float32, shape=img.shape, name='style')

        dfn = DenseFuseNet(model_pre_path)
```

```
    output_image = dfn.transform_addition(infrared_field, visible_field)

    # restore the trained model and run the style transferring
    saver = tf.train.Saver()
    saver.restore(sess, model_path)

    ##################GET
IMAGES#############################################################
######################
    start_time = datetime.now()
    for i in range(num_imgs):
      print('image number:', i)
      infrared = ir_path[i]
      visible = vis_path[i]

      ir_img = get_train_images(infrared, flag=False)
      vis_img = get_train_images(visible, flag=False)
      dimension = ir_img.shape

      ir_img = ir_img.reshape([1, dimension[0], dimension[1], dimension[2]])
      vis_img = vis_img.reshape([1, dimension[0], dimension[1], dimension[2]])

      ir_img = np.transpose(ir_img, (0, 2, 1, 3))
      vis_img = np.transpose(vis_img, (0, 2, 1, 3))

      ################FEED########################################
      output = sess.run(output_image, feed_dict={infrared_field: ir_img,
visible_field: vis_img})
      save_images(infrared, output, output_path,
              prefix='fused' + str(i), suffix='_addition_' + str(ssim_weight))

#####################################################################
#################################
    elapsed_time = datetime.now() - start_time
    print('Dense block video==> elapsed time: %s' % (elapsed_time))


def _handler_rgb(ir_path, vis_path, model_path, model_pre_path, ssim_weight,
index, output_path=None):
  # ir_img = get_train_images(ir_path, flag=False)
  # vis_img = get_train_images(vis_path, flag=False)
  ir_img = get_test_image_rgb(ir_path, flag=False)
  vis_img = get_test_image_rgb(vis_path, flag=False)
  dimension = ir_img.shape

  ir_img = ir_img.reshape([1, dimension[0], dimension[1], dimension[2]])
  vis_img = vis_img.reshape([1, dimension[0], dimension[1], dimension[2]])

  #ir_img = np.transpose(ir_img, (0, 2, 1, 3))
  #vis_img = np.transpose(vis_img, (0, 2, 1, 3))
```

```
    ir_img1 = ir_img[:, :, :, 0]
    ir_img1 = ir_img1.reshape([1, dimension[0], dimension[1], 1])
    ir_img2 = ir_img[:, :, :, 1]
    ir_img2 = ir_img2.reshape([1, dimension[0], dimension[1], 1])
    ir_img3 = ir_img[:, :, :, 2]
    ir_img3 = ir_img3.reshape([1, dimension[0], dimension[1], 1])

    vis_img1 = vis_img[:, :, :, 0]
    vis_img1 = vis_img1.reshape([1, dimension[0], dimension[1], 1])
    vis_img2 = vis_img[:, :, :, 1]
    vis_img2 = vis_img2.reshape([1, dimension[0], dimension[1], 1])
    vis_img3 = vis_img[:, :, :, 2]
    vis_img3 = vis_img3.reshape([1, dimension[0], dimension[1], 1])

    print('img shape final:', ir_img1.shape)

    with tf.Graph().as_default(), tf.Session() as sess:
        infrared_field = tf.placeholder(
            tf.float32, shape=ir_img1.shape, name='content')
        visible_field = tf.placeholder(
            tf.float32, shape=ir_img1.shape, name='style')

        dfn = DenseFuseNet(model_pre_path)

        output_image = dfn.transform_addition(infrared_field, visible_field)
        # restore the trained model and run the style transferring
        saver = tf.train.Saver()
        saver.restore(sess, model_path)

        output1 = sess.run(output_image, feed_dict={infrared_field: ir_img1,
visible_field: vis_img1})
        output2 = sess.run(output_image, feed_dict={infrared_field: ir_img2,
visible_field: vis_img2})
        output3 = sess.run(output_image, feed_dict={infrared_field: ir_img3,
visible_field: vis_img3})

        output1 = output1.reshape([1, dimension[0], dimension[1]])
        output2 = output2.reshape([1, dimension[0], dimension[1]])
        output3 = output3.reshape([1, dimension[0], dimension[1]])

        output = np.stack((output1, output2, output3), axis=-1)
        #output = np.transpose(output, (0, 2, 1, 3))
        save_images(ir_path, output, output_path,
                    prefix='fused' + str(index),
suffix='_densefuse_addition_'+str(ssim_weight))


def _handler_rgb_l1(ir_path, vis_path, model_path, model_pre_path, ssim_weight,
index, output_path=None):
```

```
    # ir_img = get_train_images(ir_path, flag=False)
    # vis_img = get_train_images(vis_path, flag=False)
    ir_img = get_test_image_rgb(ir_path, flag=False)
    vis_img = get_test_image_rgb(vis_path, flag=False)
    dimension = ir_img.shape

    ir_img = ir_img.reshape([1, dimension[0], dimension[1], dimension[2]])
    vis_img = vis_img.reshape([1, dimension[0], dimension[1], dimension[2]])

    #ir_img = np.transpose(ir_img, (0, 2, 1, 3))
    #vis_img = np.transpose(vis_img, (0, 2, 1, 3))

    ir_img1 = ir_img[:, :, :, 0]
    ir_img1 = ir_img1.reshape([1, dimension[0], dimension[1], 1])
    ir_img2 = ir_img[:, :, :, 1]
    ir_img2 = ir_img2.reshape([1, dimension[0], dimension[1], 1])
    ir_img3 = ir_img[:, :, :, 2]
    ir_img3 = ir_img3.reshape([1, dimension[0], dimension[1], 1])

    vis_img1 = vis_img[:, :, :, 0]
    vis_img1 = vis_img1.reshape([1, dimension[0], dimension[1], 1])
    vis_img2 = vis_img[:, :, :, 1]
    vis_img2 = vis_img2.reshape([1, dimension[0], dimension[1], 1])
    vis_img3 = vis_img[:, :, :, 2]
    vis_img3 = vis_img3.reshape([1, dimension[0], dimension[1], 1])

    print('img shape final:', ir_img1.shape)

    with tf.Graph().as_default(), tf.Session() as sess:
      infrared_field = tf.placeholder(
        tf.float32, shape=ir_img1.shape, name='content')
      visible_field = tf.placeholder(
        tf.float32, shape=ir_img1.shape, name='style')

      dfn = DenseFuseNet(model_pre_path)

      enc_ir = dfn.transform_encoder(infrared_field)
      enc_vis = dfn.transform_encoder(visible_field)

      target = tf.placeholder(
        tf.float32, shape=enc_ir.shape, name='target')

      output_image = dfn.transform_decoder(target)

      # restore the trained model and run the style transferring
      saver = tf.train.Saver()
      saver.restore(sess, model_path)

      enc_ir_temp, enc_vis_temp = sess.run([enc_ir, enc_vis],
  feed_dict={infrared_field: ir_img1, visible_field: vis_img1})
```

```
    feature = L1_norm(enc_ir_temp, enc_vis_temp)
    output1 = sess.run(output_image, feed_dict={target: feature})

    enc_ir_temp, enc_vis_temp = sess.run([enc_ir, enc_vis],
feed_dict={infrared_field: ir_img2, visible_field: vis_img2})
    feature = L1_norm(enc_ir_temp, enc_vis_temp)
    output2 = sess.run(output_image, feed_dict={target: feature})

    enc_ir_temp, enc_vis_temp = sess.run([enc_ir, enc_vis],
feed_dict={infrared_field: ir_img3, visible_field: vis_img3})
    feature = L1_norm(enc_ir_temp, enc_vis_temp)
    output3 = sess.run(output_image, feed_dict={target: feature})

    output1 = output1.reshape([1, dimension[0], dimension[1]])
    output2 = output2.reshape([1, dimension[0], dimension[1]])
    output3 = output3.reshape([1, dimension[0], dimension[1]])

    output = np.stack((output1, output2, output3), axis=-1)
    #output = np.transpose(output, (0, 2, 1, 3))
    save_images(ir_path, output, output_path,
                prefix='fused' + str(index),
suffix='_densefuse_l1norm_'+str(ssim_weight))
```

fusion_addition.py

```
# Additioin

def Strategy(content, style):
    # return tf.reduce_sum(content, style)
    return content+style
```

fusion_l1norm.py

```
import tensorflow as tf
import numpy as np

def L1_norm(source_en_a, source_en_b):
    result = []
    narry_a = source_en_a
    narry_b = source_en_b

    dimension = source_en_a.shape

    # caculate L1-norm
    temp_abs_a = tf.abs(narry_a)
    temp_abs_b = tf.abs(narry_b)
    _l1_a = tf.reduce_sum(temp_abs_a,3)
    _l1_b = tf.reduce_sum(temp_abs_b,3)
```

```
    _l1_a = tf.reduce_sum(_l1_a, 0)
    _l1_b = tf.reduce_sum(_l1_b, 0)
    l1_a = _l1_a.eval()
    l1_b = _l1_b.eval()

    # caculate the map for source images
    mask_value = l1_a + l1_b

    mask_sign_a = l1_a/mask_value
    mask_sign_b = l1_b/mask_value

    array_MASK_a = mask_sign_a
    array_MASK_b = mask_sign_b

    for i in range(dimension[3]):
        temp_matrix = array_MASK_a*narry_a[0,:,:,i] +
array_MASK_b*narry_b[0,:,:,i]
        result.append(temp_matrix)

    result = np.stack(result, axis=-1)

    resule_tf = np.reshape(result, (dimension[0], dimension[1], dimension[2],
dimension[3]))

    return resule_tf
```

ssim_loss_function.py

```
import tensorflow as tf
import numpy as np

def _tf_fspecial_gauss(size, sigma):
    """Function to mimic the 'fspecial' gaussian MATLAB function
    """
    x_data, y_data = np.mgrid[-size//2 + 1:size//2 + 1, -size//2 + 1:size//2 + 1]

    x_data = np.expand_dims(x_data, axis=-1)
    x_data = np.expand_dims(x_data, axis=-1)

    y_data = np.expand_dims(y_data, axis=-1)
    y_data = np.expand_dims(y_data, axis=-1)

    x = tf.constant(x_data, dtype=tf.float32)
    y = tf.constant(y_data, dtype=tf.float32)

    g = tf.exp(-((x**2 + y**2)/(2.0*sigma**2)))
    return g / tf.reduce_sum(g)
```

```
def SSIM_LOSS(img1, img2, size=11, sigma=1.5):
    window = _tf_fspecial_gauss(size, sigma) # window shape [size, size]
    K1 = 0.01
    K2 = 0.03
    L = 1  # depth of image (255 in case the image has a differnt scale)
    C1 = (K1*L)**2
    C2 = (K2*L)**2
    mu1 = tf.nn.conv2d(img1, window, strides=[1,1,1,1], padding='VALID')
    mu2 = tf.nn.conv2d(img2, window, strides=[1,1,1,1],padding='VALID')
    mu1_sq = mu1*mu1
    mu2_sq = mu2*mu2
    mu1_mu2 = mu1*mu2
    sigma1_sq = tf.nn.conv2d(img1*img1, window,
strides=[1,1,1,1],padding='VALID') - mu1_sq
    sigma2_sq = tf.nn.conv2d(img2*img2, window,
strides=[1,1,1,1],padding='VALID') - mu2_sq
    sigma12 = tf.nn.conv2d(img1*img2, window, strides=[1,1,1,1],padding='VALID')
- mu1_mu2

    value = (2.0*sigma12 + C2)/(sigma1_sq + sigma2_sq + C2)
    value = tf.reduce_mean(value)
    return value
```

train_recons.py

```
# Train the DenseFuse Net

from __future__ import print_function

import scipy.io as scio
import numpy as np
import tensorflow as tf

from ssim_loss_function import SSIM_LOSS
from densefuse_net import DenseFuseNet
from utils import get_train_images, get_train_images_rgb

STYLE_LAYERS = ('relu1_1', 'relu2_1', 'relu3_1', 'relu4_1')

HEIGHT = 256
WIDTH = 256
CHANNELS = 1 # gray scale, default

LEARNING_RATE = 1e-4
EPSILON = 1e-5
```

```
def train_recons(original_imgs_path, validatioin_imgs_path, save_path, model_pre_path, ssim_weight, EPOCHES_set, BATCH_SIZE, IS_Validation, debug=False, logging_period=1):
    if debug:
        from datetime import datetime
        start_time = datetime.now()
    EPOCHS = EPOCHES_set
    print("EPOCHES   : ", EPOCHS)
    print("BATCH_SIZE: ", BATCH_SIZE)

    num_val = len(validatioin_imgs_path)
    num_imgs = len(original_imgs_path)
    # num_imgs = 100
    original_imgs_path = original_imgs_path[:num_imgs]
    mod = num_imgs % BATCH_SIZE

    print('Train images number %d.\n' % num_imgs)
    print('Train images samples %s.\n' % str(num_imgs / BATCH_SIZE))

    if mod > 0:
        print('Train set has been trimmed %d samples...\n' % mod)
        original_imgs_path = original_imgs_path[:-mod]

    # get the traing image shape
    INPUT_SHAPE_OR = (BATCH_SIZE, HEIGHT, WIDTH, CHANNELS)

    # create the graph
    with tf.Graph().as_default(), tf.Session() as sess:
        original = tf.placeholder(tf.float32, shape=INPUT_SHAPE_OR, name='original')
        source = original

        print('source  :', source.shape)
        print('original:', original.shape)

        # create the deepfuse net (encoder and decoder)
        dfn = DenseFuseNet(model_pre_path)
        generated_img = dfn.transform_recons(source)
        print('generate:', generated_img.shape)

        ssim_loss_value = SSIM_LOSS(original, generated_img)
        pixel_loss = tf.reduce_sum(tf.square(original - generated_img))
        pixel_loss = pixel_loss/(BATCH_SIZE*HEIGHT*WIDTH)
        ssim_loss = 1 - ssim_loss_value

        loss = ssim_weight*ssim_loss + pixel_loss
        train_op = tf.train.AdamOptimizer(LEARNING_RATE).minimize(loss)

        sess.run(tf.global_variables_initializer())

        # saver = tf.train.Saver()
```

```
        saver = tf.train.Saver(keep_checkpoint_every_n_hours=1)

        # ** Start Training **
        step = 0
        count_loss = 0
        n_batches = int(len(original_imgs_path) // BATCH_SIZE)
        val_batches = int(len(validatioin_imgs_path) // BATCH_SIZE)

        if debug:
            elapsed_time = datetime.now() - start_time
            print('\nElapsed time for preprocessing before actually train the model: %s' %
elapsed_time)
            print('Now begin to train the model...\n')
            start_time = datetime.now()

        Loss_all = [i for i in range(EPOCHS * n_batches)]
        Loss_ssim = [i for i in range(EPOCHS * n_batches)]
        Loss_pixel = [i for i in range(EPOCHS * n_batches)]
        Val_ssim_data = [i for i in range(EPOCHS * n_batches)]
        Val_pixel_data = [i for i in range(EPOCHS * n_batches)]
        for epoch in range(EPOCHS):

            np.random.shuffle(original_imgs_path)

            for batch in range(n_batches):
                # retrive a batch of content and style images

                original_path =
original_imgs_path[batch*BATCH_SIZE:(batch*BATCH_SIZE + BATCH_SIZE)]
                ### read gray scale images
                original_batch = get_train_images(original_path, crop_height=HEIGHT,
crop_width=WIDTH, flag=False)
                ### read RGB images
                # original_batch = get_train_images_rgb(original_path,
crop_height=HEIGHT, crop_width=WIDTH, flag=False)
                original_batch = original_batch.transpose((3, 0, 1, 2))

                # print('original_batch shape final:', original_batch.shape)

                # run the training step
                sess.run(train_op, feed_dict={original: original_batch})
                step += 1
                if debug:
                    is_last_step = (epoch == EPOCHS - 1) and (batch == n_batches - 1)

                    if is_last_step or step % logging_period == 0:
                        elapsed_time = datetime.now() - start_time
                        _ssim_loss, _loss, _p_loss = sess.run([ssim_loss, loss, pixel_loss],
feed_dict={original: original_batch})
                        Loss_all[count_loss] = _loss
```

```
                Loss_ssim[count_loss] = _ssim_loss
                Loss_pixel[count_loss] = _p_loss
                print('epoch: %d/%d, step: %d,  total loss: %s, elapsed time: %s' %
(epoch, EPOCHS, step, _loss, elapsed_time))
                print('p_loss: %s, ssim_loss: %s ,w_ssim_loss: %s ' % (_p_loss,
_ssim_loss, ssim_weight * _ssim_loss))

                # IS_Validation = True;
                # Calculating the accuracy rate for 1000 images, every 100 steps
                if IS_Validation:
                    val_ssim_acc = 0
                    val_pixel_acc = 0
                    np.random.shuffle(validatioin_imgs_path)
                    val_start_time = datetime.now()
                    for v in range(val_batches):
                        val_original_path = validatioin_imgs_path[v * BATCH_SIZE:(v
* BATCH_SIZE + BATCH_SIZE)]
                        val_original_batch = get_train_images(val_original_path,
crop_height=HEIGHT, crop_width=WIDTH,flag=False)
                        val_original_batch = val_original_batch.reshape([BATCH_SIZE,
256, 256, 1])
                        val_ssim, val_pixel = sess.run([ssim_loss, pixel_loss],
feed_dict={original: val_original_batch})
                        val_ssim_acc = val_ssim_acc + (1 - val_ssim)
                        val_pixel_acc = val_pixel_acc + val_pixel
                    Val_ssim_data[count_loss] = val_ssim_acc/val_batches
                    Val_pixel_data[count_loss] = val_pixel_acc / val_batches
                    val_es_time = datetime.now() - val_start_time
                    print('validation value, SSIM: %s, Pixel: %s, elapsed time: %s' %
(val_ssim_acc/val_batches, val_pixel_acc / val_batches, val_es_time))
                    print('------------------------------------------------------------------------
---')
                count_loss += 1


    # ** Done Training & Save the model **
    saver.save(sess, save_path)

    loss_data = Loss_all[:count_loss]

scio.savemat('./models/loss/DeepDenseLossData'+str(ssim_weight)+'.mat',{'loss':loss
_data})

    loss_ssim_data = Loss_ssim[:count_loss]
    scio.savemat('./models/loss/DeepDenseLossSSIMData'+str(ssim_weight)+'.mat',
{'loss_ssim': loss_ssim_data})

    loss_pixel_data = Loss_pixel[:count_loss]
    scio.savemat('./models/loss/DeepDenseLossPixelData.mat'+str(ssim_weight)+'',
{'loss_pixel': loss_pixel_data})
```

```
        # IS_Validation = True;
        if IS_Validation:
            validation_ssim_data = Val_ssim_data[:count_loss]
            scio.savemat('./models/val/Validation_ssim_Data.mat' + str(ssim_weight) + '',
{'val_ssim': validation_ssim_data})
            validation_pixel_data = Val_pixel_data[:count_loss]
            scio.savemat('./models/val/Validation_pixel_Data.mat' + str(ssim_weight) + '',
{'val_pixel': validation_pixel_data})


        if debug:
            elapsed_time = datetime.now() - start_time
            print('Done training! Elapsed time: %s' % elapsed_time)
            print('Model is saved to: %s' % save_path)
```

utils.py

```
# Utility

import numpy as np

from os import listdir, mkdir, sep
from os.path import join, exists, splitext
#from scipy.misc import imread, imsave, imresize
import skimage
import skimage.io
import skimage.transform
import tensorflow as tf
from PIL import Image
from functools import reduce
import cv2

def list_images(directory):
    images = []
    dir = listdir(directory)
    dir.sort()
    for file in dir:
        name = file.lower()
        if name.endswith('.png'):
            images.append(join(directory, file))
        elif name.endswith('.jpg'):
            images.append(join(directory, file))
        elif name.endswith('.jpeg'):
            images.append(join(directory, file))
        elif name.endswith('.bmp'):
            images.append(join(directory, file))
    return images
```

```
# read images
def get_image(path, height=256, width=256, set_mode='L'):
    print(path)
    image = cv2.imread(path,0)
    if height is not None and width is not None:
        image = cv2.resize(image, (height, width), cv2.INTER_NEAREST)
    return image


def get_train_images(paths, resize_len=512, crop_height=256, crop_width=256,
flag=True):
    if isinstance(paths, str):
        paths = [paths]

    images = []
    for path in paths:
        image = get_image(path, height=crop_height, width=crop_width, set_mode='L')

        if flag:
            image = np.stack(image, axis=0)
            image = np.stack((image, image, image), axis=-1)
        else:
            image = np.stack(image, axis=0)
            image = image.reshape([crop_height, crop_width, 1])
        images.append(image)
    images = np.stack(images, axis=-1)
    return images


def get_train_images_rgb(paths, crop_height=256, crop_width=256, flag=False):
    if isinstance(paths, str):
        paths = [paths]

    images = []
    for path in paths:
        image = get_image(path, height=crop_height, width=crop_width,
set_mode='RGB')
        image = np.stack(image, axis=0)
        images.append(image)
    images = np.stack(images, axis=-1)
    return images


def get_test_image_rgb(path, resize_len=512, crop_height=256, crop_width=256, flag
= True):
    # image = imread(path, mode='L')
    image = imread(path, mode='RGB')
    return image
```

```
def get_images_test(path, mod_type='L', height=None, width=None):

    image = imread(path, mode=mod_type)
    if height is not None and width is not None:
        image = imresize(image, [height, width], interp='nearest')

    if mod_type=='L':
        d = image.shape
        image = np.reshape(image, [d[0], d[1], 1])

    return image


def get_images(paths, height=None, width=None):
    if isinstance(paths, str):
        paths = [paths]

    images = []
    for path in paths:
        image = imread(path, mode='RGB')

        if height is not None and width is not None:
            image = imresize(image, [height, width], interp='nearest')

        images.append(image)

    images = np.stack(images, axis=0)
    print('images shape gen:', images.shape)
    return images


def save_images(paths, datas, save_path, prefix=None, suffix=None):
    if isinstance(paths, str):
        paths = [paths]

    t1 = len(paths)
    t2 = len(datas)
    assert(len(paths) == len(datas))

    if not exists(save_path):
        mkdir(save_path)

    if prefix is None:
        prefix = ''
    if suffix is None:
        suffix = ''

    for i, path in enumerate(paths):
```

```
        data = datas[i]
        # print('data ==>>\n', data)
        if data.shape[2] == 1:
            data = data.reshape([data.shape[0], data.shape[1]])
        # print('data reshape==>>\n', data)

        name, ext = splitext(path)
        name = name.split(sep)[-1]

        path = join(save_path, prefix + suffix + ext)
        print('data path==>>', path)


        # new_im = Image.fromarray(data)
        # new_im.show()
        data = data/255
        data = cv2.rotate(data, cv2.cv2.ROTATE_90_CLOCKWISE)
        cv2.imshow("Fuse_Image.png",data)
        #imsave(path, data)

def get_l2_norm_loss(diffs):
    shape = diffs.get_shape().as_list()
    size = reduce(lambda x, y: x * y, shape) ** 2
    sum_of_squared_diffs = tf.reduce_sum(tf.square(diffs))
    return sum_of_squared_diffs / size
```

Densefuse_net.py

```
# DenseFuse Network
# Encoder -> Addition/L1-norm -> Decoder

import tensorflow as tf

from encoder import Encoder
from decoder import Decoder
from fusion_addition import Strategy

class DenseFuseNet(object):

    def __init__(self, model_pre_path):
        self.encoder = Encoder(model_pre_path)
        self.decoder = Decoder(model_pre_path)

    def transform_addition(self, img1, img2):
        # encode image
        enc_1 = self.encoder.encode(img1)
        enc_2 = self.encoder.encode(img2)
        target_features = Strategy(enc_1, enc_2)
        # target_features = enc_c
```

```
        self.target_features = target_features
        print('target_features:', target_features.shape)
        # decode target features back to image
        generated_img = self.decoder.decode(target_features)
        return generated_img

    def transform_recons(self, img):
        # encode image
        enc = self.encoder.encode(img)
        target_features = enc
        self.target_features = target_features
        generated_img = self.decoder.decode(target_features)
        return generated_img


    def transform_encoder(self, img):
        # encode image
        enc = self.encoder.encode(img)
        return enc

    def transform_decoder(self, feature):
        # decode image
        generated_img = self.decoder.decode(feature)
        return generated_img
```

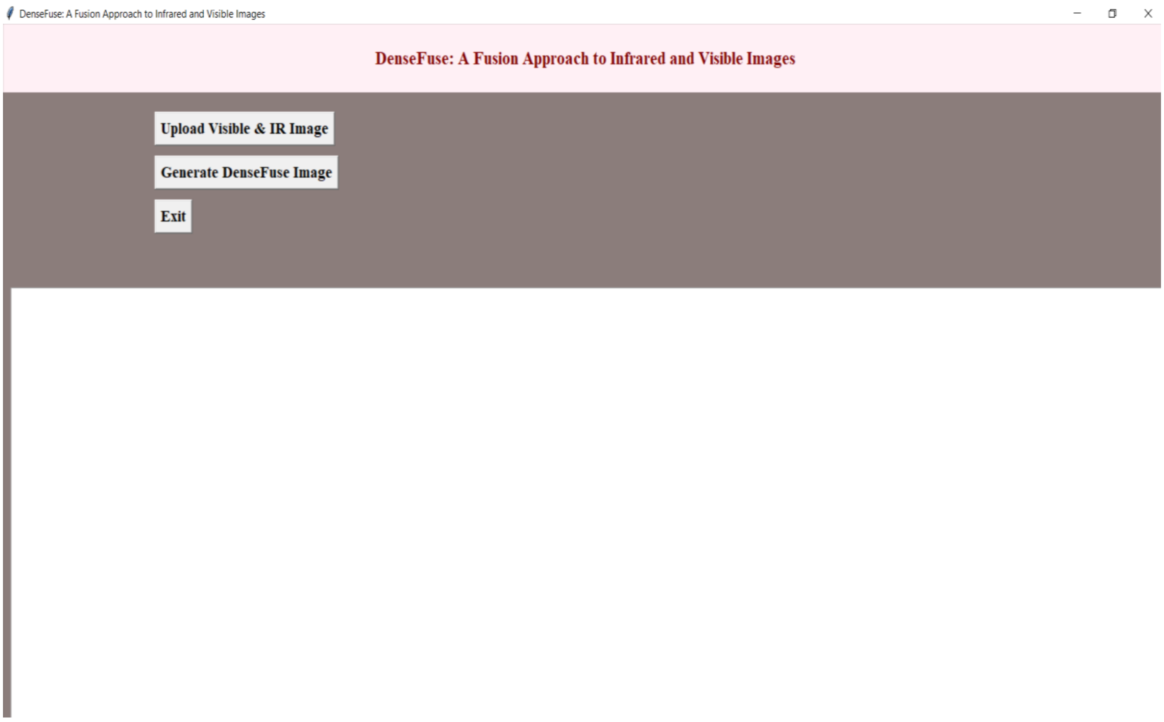
4.2 TEST CASES

Both input images Infrared and visible are source images and both are kept in a folder to upload it as an input.

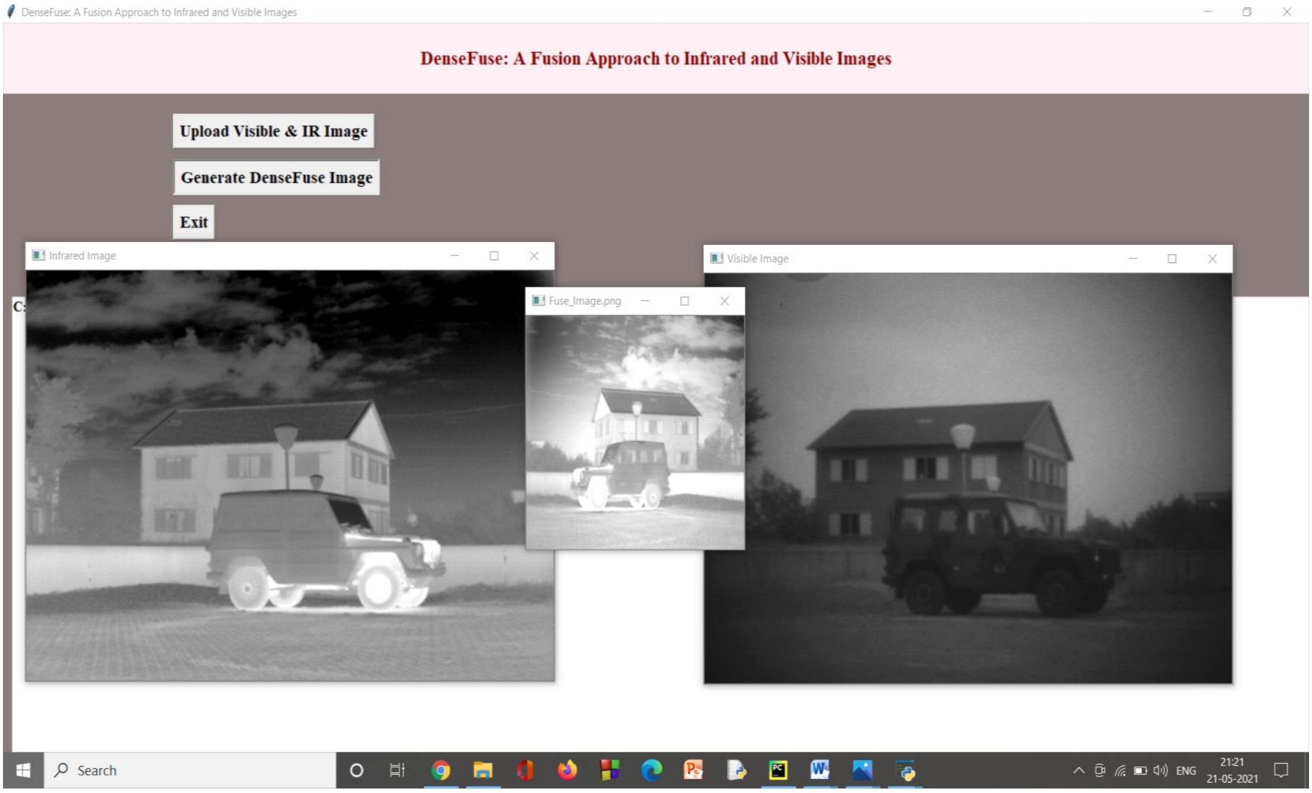
Test Case ID	Test Scenario	Expected Result	Actual Result	Pass/Fail
TC01	check whether the application is working	output page should be opened	As Expected	Pass
TC02	check whether the upload option is working	Images should be accessible	As Expected	Pass
TC03	check whether generate option is working	Images should be generated	As Expected	Pass
TC04	check whether exit option is working	should exit from the page	As Expected	Pass
TC05	check whether image is getting uploaded	images should get uploaded	As Expected	Pass
TC06	check whether getting the correct result	should get correct result	As Expected	Pass

Fig 4.3: Test cases

4.3 INPUT SCREENSHOTS



4.4 OUTPUT SCREENSHOTS



5. CONCLUSION AND FUTURE SCOPE

Our network has three parts: encoder, fusion layer and decoder. Firstly, the source images (infrared and visible images) are utilized as the input of encoder. And the features maps are obtained by CNN layer and dense block, which are fused by fusion strategy (addition and l1-norm). After fusion layer, the feature maps are integrated into one feature map which contains all salient features from source images. Finally, the fused image is reconstructed by decoder network. We use both subjective and objective quality metrics to evaluate our fusion method.

The future enhancement of this application is

The proposed method deals with IR and visible image fusion problem, however, it is general and can be also applied to other image processing problems such as super-resolution. In the proposed method, as others cannot distinguish between IR target and brightness of visible images. We want to focus on this aspect as future work.

6. REFERENCES

[1] Hui Li and Xiao-Jun Wu, DenseFuse: A Fusion Approach to Infrared and Visible Images, IEEE Transactions on image processing, Vol 28, No : 5, May 2019

[2] S. Li, X. Kang, L. Fang, J. Hu, and H. Yin, “Pixel-level image fusion: A survey of the state of the art”, Inf Fusion, vol. 33, pp. 100–112, Jan 2017.

[3] L. Wang, B. Li, and L.-F. Tian, “EGGDD: An explicit dependency model for multi-modal medical image fusion in shift-invariant shearlet transform domain,” Inf. Fusion, vol. 19, pp. 29–37, Sep 2014.

[4] D. P. Bavirisetti and R. Dhuli, “Two-scale image fusion of visible and infrared images using saliency detection,” Infr. Phys. Technol., vol. 76, pp. 52–64, May 2016

[5] Y. Liu, X. Chen, H. Peng, and Z. Wang, “Multi-focus image fusion with a deep convolutional neural network,” Inf. Fusion, vol. 36 pp. 191–207, Jul 2017.