National College of Ireland

Configuration Manual

MSc Research Project
Data Analytics

Akansha Bansal
Student ID: X18182615

School of Computing
National College of Ireland

Supervisor: Dr. Catherine Mulwa

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Akansha Bansal		
Student ID:	X18182615		
Programme:	Data Analytics	**Year:**	2020
Module:	MSc Research Project		
Supervisor:	Dr. Catherine Mulwa		
Submission Due Date:	17/08/2020		
Project Title:	Identification and Classification of Defects in Steel Sheets Using Deep Learning Models		
Word Count:	..... **Page Count**.....		

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: .....

Date: .....

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Identification and Classification of Defects in Steel Sheets Using Deep Learning Models

Akansha Bansal
X18182615

1 Introduction

The aim of this manual is to provide walkthrough for any user to setup the configuration in their personal machine to produce desire outcomes. The documents provide the whole process to setup the environment which includes hardware as well as software specifications. This manual includes snapshot of code, visualizations of exploratory data analysis and evaluations of each model.
The rest of the technical report is as follows: Chapter 2 illustrates the Environment Configuration, Chapter 3 discusses about the source of data collection, Chapter 4 discusses about how the data is imported into Google Collaboratory, Chapter 5 illustrates for exploratory data analysis, Chapter 6 , 7, 8 and 9 discusses about the implementation and evaluation of Xception, U-Net, Mask RCNN and UNet++ respectively.

2 Environment

This section discusses about the complete environment requires for ICT solution. This includes hardware and software configurations, python libraries and packages, and setup of Google Collaboratory.

2.1 Hardware and Software Configuration

The figure (1) shows the hardware specification require to implement the ICT solution. The figure displayed the system has 64-bit operating system on Microsoft Windows 10 with 1.80 GHz process and 8 GB Ram used.

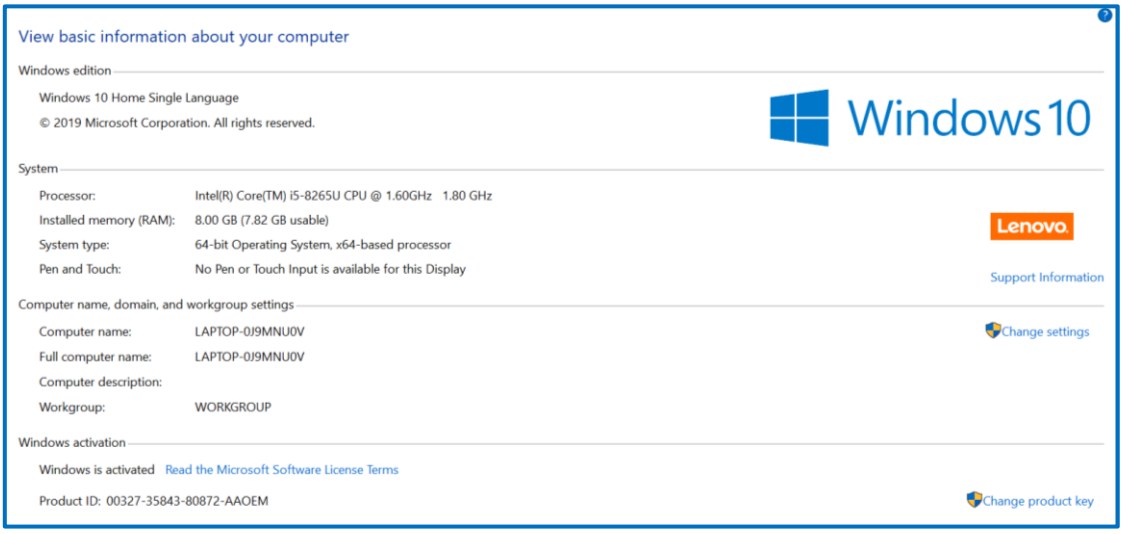

Figure 1: Hardware Configuration

The other part of the section discusses about the software configurations require. For its implementation Google Collaboratory is used. It is cloud based Jupyter notebook used for training the large dataset for deep learning model on GPU.

3 Data Collection

The data has been downloaded from Kaggle. The link of the data set is https://www.kaggle.com/c/severstal-steel-defect-detection/data. This data set contains files such as train.csv, train.csv and12,568 images of steel sheet.

4 Data Extraction

This position provides an overview of the Python code which is written as a part of the implementation of ICT solution.

4.1 Importing the Libraries

Different packages of python libraries were imported in Google Collaboratory.

```
# some basic imports
import pandas as pd
import numpy as np
import os
import cv2
# visualization
import matplotlib.pyplot as plt
# plotly offline imports
from plotly.offline import download_plotlyjs, init_notebook_mode, iplot
from plotly import subplots
import plotly.express as px
import plotly.figure_factory as ff
from plotly.graph_objs import *
from plotly.graph_objs.layout import Margin, YAxis, XAxis
configure_plotly_browser_state()
init_notebook_mode(connected=False)
# frequent pattern mining
from mlxtend import frequent_patterns
# path where all the training images are
```

Figure 2: Installed packages of python for exploratory data analysis

4.2 File Upload and Reading the dataset

Data files were mounted on Google Drive as shown in figure 3 and first few row of the train.csv were printed (refer Figure 4).

```
!unzip train.csv.zip

unzip:  cannot find or open train.csv.zip, train.csv.zip.zip or train.csv.zip.ZIP.

from google.colab import drive
drive.mount('/gdrive')

Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redirect_uri=urn%3aietf%3awg%3aoauth%3

Enter your authorization code:
··········
Mounted at /gdrive

!cp "/gdrive/My Drive/train_images.zip" train.zip
# !mkdir train_images

# !cp train.zip train_images/train.zip
# !rm -r train_images
!unzip train.zip
```

Figure 3: Data file uploaded on Google Collaboratory

	ImageId	ClassId	EncodedPixels	ClassId_EncodedPixels
0	0002cc93b.jpg	1	29102 12 29346 24 29602 24 29858 24 30114 24 3...	(1, 29102 12 29346 24 29602 24 29858 24 30114 ...
1	0007a71bf.jpg	3	18661 28 18863 82 19091 110 19347 110 19603 11...	(3, 18661 28 18863 82 19091 110 19347 110 1960...
2	000a4bcdd.jpg	1	37607 3 37858 8 38108 14 38359 20 38610 25 388...	(1, 37607 3 37858 8 38108 14 38359 20 38610 25...
3	000f6bf48.jpg	4	131973 1 132228 4 132483 6 132738 8 132993 11 ...	(4, 131973 1 132228 4 132483 6 132738 8 132993...
4	0014fce06.jpg	3	229501 11 229741 33 229981 55 230221 77 230468...	(3, 229501 11 229741 33 229981 55 230221 77 23...

Figure 4: First five rows of the train.csv file

5. Exploratory Data Analysis

Exploratory data analysis has been done to understand relation between the variables. The below figure 5 counts the pixels of the defected area in steel sheet by using the concept of running length encoding.

```
def rle_to_mask(rle_string, height, width):


    rows, cols = height, width

    if rle_string == -1:
        return np.zeros((height, width))
    else:
        rle_numbers = [int(num_string) for num_string in rle_string.split(' ')]
        rle_pairs = np.array(rle_numbers).reshape(-1,2)
        img = np.zeros(rows*cols, dtype=np.uint8)
        for index, length in rle_pairs:
            index -= 1
            img[index:index+length] = 255
        img = img.reshape(cols,rows)
        img = img.T
        return img
```

Figure 5: Running length encoding

'viz_two class_from_path' visualizes the image with four classes of defects.

```
# visualize steel image with four classes of defects in seperate columns
def viz_two_class_from_path(img_path, img_id, encoded_masks):


    img = cv2.imread(os.path.join(img_path, img_id))
    fig, ax = plt.subplots(nrows=1, ncols=2, sharey=True, figsize=(20,10))
    cmaps = ["Reds", "Blues", "Greens", "Purples"]
    axid = 0
    for idx, encoded_mask in enumerate(encoded_masks):
        class_id = idx + 1
        if encoded_mask == -1:
            pass
        else:
            mask_decoded = rle_to_mask(encoded_mask, 256, 1600)
            ax[axid].get_xaxis().set_ticks([])
            ax[axid].get_yaxis().set_ticks([])
            ax[axid].text(0.25, 0.25, 'Image Id: %s - Class Id: %s' % (img_id, class_id), fontsize=12)
            ax[axid].imshow(img)
            ax[axid].imshow(mask_decoded, alpha=0.15, cmap=cmaps[idx])
            axid += 1
```

Figure 6: Visualization of four typed of defects

The below snapshot of figure 7 creates the pie-chart for pixel defect and six of the defect masks.

```
configure_plotly_browser_state()
init_notebook_mode(connected=False)

# Create subplots: use 'domain' type for Pie subplot
fig = subplots.make_subplots(rows=1, cols=2, specs=[[{'type':'domain'}, {'type':'domain'}]])

fig.add_trace(Pie(labels=class_ids, values=mask_count_per_class, name="Mask Count"), 1, 1)
fig.add_trace(Pie(labels=class_ids, values=pixel_sum_per_class, name="Pixel Count"), 1, 2)
# Use `hole` to create a donut-like pie chart
fig.update_traces(hole=.4, hoverinfo="label+percent+name")

fig.update_layout(
    title_text="Steel Defect Mask & Pixel Count",
    # Add annotations in the center of the donut pies.
    annotations=[dict(text='Mask', x=0.18, y=0.5, font_size=20, showarrow=False),
                 dict(text='Pixel', x=0.80, y=0.5, font_size=20, showarrow=False)])
fig.show()
```

Figure 7: Mask size per defect class

Figure 8 prints the segments based on defect type in steel sheet

```
def count_segments(mask):

    # if the mask is empty return zero
    if mask.sum() == 0:
        return 0
    else:
        # use open cv and threshold mechanism to calculate contours
        _, threshold = cv2.threshold(mask, 240, 255, cv2.THRESH_BINARY)
        _, contours = cv2.findContours(threshold, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)

        # get the segment count
        for c in contours:
            segments_count = len(c)

        return segments_count
```

Figure 8: Segments count based on defect type

The below figure 9 shows the implementation of frequent pattern algorithm which calculates how frequently the particular defect has occurred and also the ocuurence of defect in pairs.

```
[ ]  # create a list of dict with count of each fault class
     class_per_image_list = []
     for r in class_per_image.iteritems():
         class_count = {1:0,2:0,3:0,4:0}
         # go over each class and
         for image_class in r[1]:
             class_count[image_class] = 1
         class_per_image_list.append(class_count)

[ ]  # do FP calculation with all image
     class_per_image_df = pd.DataFrame(class_per_image_list)
     class_fp_df = frequent_patterns.apriori(class_per_image_df.astype(bool) , use_colnames=True, min_support=0.001)
     class_fp_df = class_fp_df.sort_values(by=['support'])

[ ]  # subset to images with at least one mask
     class_per_fault_image_df = class_per_image_df[(class_per_image_df.T != 0).any()]
     class_fp_faulty_df = frequent_patterns.apriori(class_per_image_df.astype(bool), use_colnames=True, min_support=0.001)
     class_fp_faulty_df = class_fp_faulty_df.sort_values(by=['support'])
```

Figure 9: Frequent pattern function

6. Implementation and Evaluation of Xception Model

Below figure 10 shows the required libraries of python were imported for the implementation of Xception model.

```
import segmentation_models
print(segmentation_models.__version__)

from sklearn.metrics import classification_report, multilabel_confusion_matrix

import keras
from keras.preprocessing.image import ImageDataGenerator
from keras import backend as K
from keras.layers import GlobalAveragePooling2D, Dense, Conv2D, BatchNormalization, Dropout
from keras.models import Model, load_model
import tensorflow as tf
from tensorflow.python.keras.callbacks import TensorBoard
from keras.callbacks import ModelCheckpoint
from sklearn.model_selection import train_test_split
import seaborn
import segmentation_models as sm
from segmentation_models import Unet
from segmentation_models import get_preprocessing
```

Figure 10: Installed libraries of python for the implementation of Xception model

'dataGenerator' function is used to resize the image and also use to outputs the array of images to the model.

```
class test_DataGenerator_3(keras.utils.Sequence):
    def __init__(self, df, batch_size = 1, image_path = '/content/image_data/',
                 preprocess=None, info={}):
        super().__init__()
        self.df = df
        self.batch_size = batch_size
        self.preprocess = preprocess
        self.info = info
        self.data_path = image_path
        self.on_epoch_end()
    def __len__(self):
        return int(np.floor(len(self.df) / self.batch_size))
    def on_epoch_end(self):
        self.indexes = np.arange(len(self.df))
    def __getitem__(self, index):

        X = np.empty((self.batch_size,256,800,3),dtype=np.float32)
        indexes = self.indexes[index*self.batch_size:(index+1)*self.batch_size]
        for i,f in enumerate(self.df['ImageId'].iloc[indexes]):
            self.info[index*self.batch_size+i]=f
            X[i,] = Image.open(self.data_path + f).resize((800,256))
        if self.preprocess!=None: X = self.preprocess(X)
        return X
```

Figure 11: DataGenerator function

‘pred_classification’ function is used to input data frame of image id and return the prediction of classification model.

```
def pred_classification(X):

    X = X.reset_index().drop('index',axis=1)
    data_generator = ImageDataGenerator(rescale=1./255).flow_from_dataframe(dataframe=X, directory='/content/image_data/',
                                                                            x_col="ImageId", class_mode = None,
                                                                            target_size=(256,512), batch_size=1, shuffle=False)

    data_preds_binary = model_binary.predict_generator(data_generator,verbose=0)
    data_preds_multi_label = model_multi.predict_generator(data_generator,verbose=0)
    data_classification = pd.DataFrame(data_preds_multi_label, columns = ['defect_1','defect_2','defect_3','defect_4'])
    data_classification['hasDefect'] = data_preds_binary
    data_classification['ImageId'] = X['ImageId']
    return data_classification[['ImageId','hasDefect','defect_1','defect_2','defect_3','defect_4']]
```

Figure 12: pred_classification function

‘pred_segmentation’ function is used to input data frame of image id and return the prediction of segmentation model.

```
def pred_segmentation(X):

    X = X.reset_index().drop('index',axis=1)
    preprocess = get_preprocessing('efficientnetb1')
    tmp=[]
    loop_num = 50
    for j in range((len(X)//loop_num)+1):
        test_dataf = X[loop_num*j:loop_num*j+loop_num]
        test_batches =  test_DataGenerator_3(test_dataf,preprocess=preprocess)
        test_preds_1 = model_segment_1.predict_generator(test_batches,verbose=0)
        test_preds_2 = model_segment_2.predict_generator(test_batches,verbose=0)
        test_preds_3 = model_segment_3.predict_generator(test_batches,verbose=0)
        test_preds_4 = model_segment_4.predict_generator(test_batches,verbose=0)
        for i in range(len(test_preds_1)):
            ep1 = mask2rle(np.array((Image.fromarray((test_preds_1[i][:,:,0])>=0.5)).resize((1600,256))).astype(int))
            ep2 = mask2rle(np.array((Image.fromarray((test_preds_2[i][:,:,0])>=0.5)).resize((1600,256))).astype(int))
            ep3 = mask2rle(np.array((Image.fromarray((test_preds_3[i][:,:,0])>=0.5)).resize((1600,256))).astype(int))
            ep4 = mask2rle(np.array((Image.fromarray((test_preds_4[i][:,:,0])>=0.5)).resize((1600,256))).astype(int))
            tmp.append([test_dataf.ImageId.iloc[i],ep1,ep2,ep3,ep4])

    return pd.DataFrame(tmp,columns=['ImageId','EncodedPixels_1','EncodedPixels_2','EncodedPixels_3','EncodedPixels_4'])
```

Figure 13: pred_segmentation function

The snapshot of python code is use to produce the confusion matrix of test dataset.

```
def plt_confusion(d):
    '''
    Input: Confusion_matrix
    Output: Seaborn heatmap visualization (OneVersusRest mode of confusion matrices)
    '''
    fig, ax = plt.subplots(1,5,figsize=(20,4))
    for i in range(4):
        seaborn.heatmap(d[i],annot=True,fmt='d',cmap="Reds", ax = ax[i])
        ax[i].set_ylabel('Actual')
        ax[i].set_xlabel('Predicted')
        ax[i].set_title(f'Defect_{i+1} confusion matrix')
    seaborn.heatmap(d[4],annot=True,fmt='d',cmap="Reds", ax = ax[4])
    ax[4].set_ylabel('Actual')
    ax[4].set_xlabel('Predicted')
    ax[4].set_title('No defect confusion matrix')
    plt.show()
```

Figure 14: Function for confusion matrix

The below snapshot shows the confusion matrix for four types of defects.

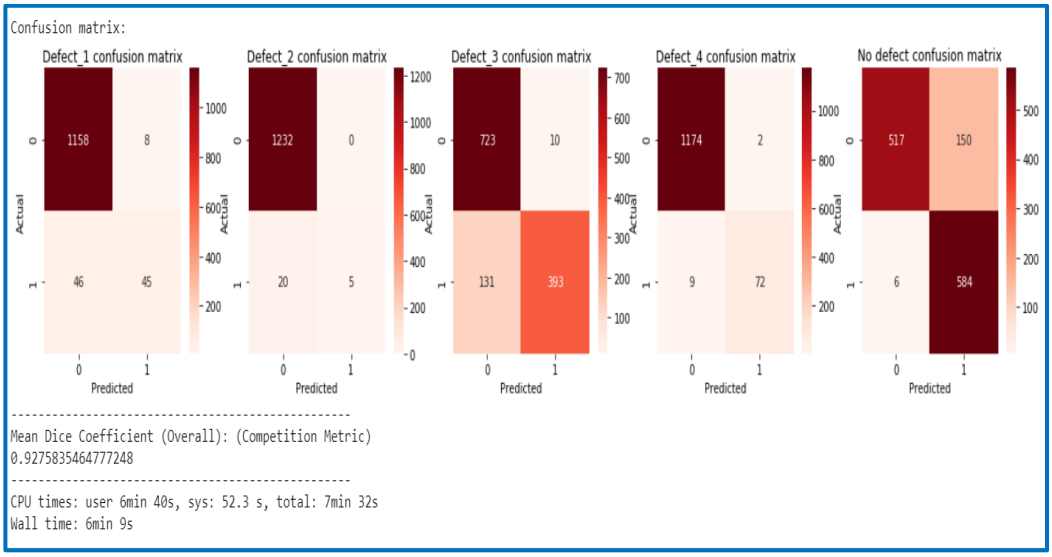

Figure 15: Confusion matrix for four types of defects

7. Implementation and Evaluation of U-Net Model

The 'build_masks' function is used to build mask over the defects on steel sheet.

```
def build_masks(rles, input_shape):

    depth = len(rles)    #4
    height, width = input_shape
    masks = np.zeros((height, width, depth))

    for i, rle in enumerate(rles):
        """Looping for each of the 4 types of defects"""
        if type(rle) is str:
            masks[:, :, i] = rle2mask(rle, (width,height))
        else:
            masks[:,:,i] = np.nan

    return masks

def build_rles(masks):
    width, height, depth = masks.shape

    rles = [mask2rle(masks[:, :, i])
            for i in range(depth)]

    return rles
```

Figure 16: Function to create mask over defect

The 'dice_coef' function is use to flatten the image into arrays and calculates its dice coefficient.

```python
def dice_coef(y_true, y_pred, smooth=1):
    y_true_f = K.flatten(y_true)
    y_pred_f = K.flatten(y_pred)
    intersection = K.sum(y_true_f * y_pred_f)
    return (2. * intersection + smooth) / (K.sum(y_true_f) + K.sum(y_pred_f) + smooth)

def dice_loss(y_true, y_pred):
    return 1.0 - dice_coef(y_true, y_pred)

def bce_dice_loss(y_true, y_pred):
    return binary_crossentropy(y_true, y_pred) + dice_loss(y_true, y_pred)

def wbce_dice_loss(y_true, y_pred):
    return weighted_bce()(y_true, y_pred) + dice_loss(y_true, y_pred)

def weighted_bce(weight = 0.6):

    def convert_2_logits(y_pred):
        y_pred = tf.clip_by_value(y_pred, K.epsilon(), 1 - K.epsilon())
        return tf.log(y_pred / (1-y_pred))

    def weighted_binary_crossentropy(y_true, y_pred):
        y_pred = convert_2_logits(y_pred)
        loss = tf.nn.weighted_cross_entropy_with_logits(logits = y_pred, targets = y_true, pos_weight = weight)
        return loss

    return weighted_binary_crossentropy
```

Figure 17: Dice Coefficient for evaluation of model

Figure 18 and 19 implements DataGenerator function to load the file into batches and returns mask associated with it.

```python
class DataGenerator(keras.utils.Sequence):
    'Generates data for Keras'
    def __init__(self, list_IDs, df, target_df=None, mode='fit',
                 base_path='content/train_images/',
                 batch_size=32, dim=(256, 1600), n_channels=1,
                 n_classes=4, random_state=2019, shuffle=True):
        self.dim = dim
        self.batch_size = batch_size
        self.df = df
        self.mode = mode
        self.base_path = base_path
        self.target_df = target_df
        self.list_IDs = list_IDs
        self.n_channels = n_channels
        self.n_classes = n_classes
        self.shuffle = shuffle
        self.random_state = random_state

        self.on_epoch_end()

    def __len__(self):
        'Denotes the number of batches per epoch'
        return int(np.floor(len(self.list_IDs) / self.batch_size))

    def __getitem__(self, index):
        'Generate one batch of data'
        # Generate indexes of the batch
        indexes = self.indexes[index*self.batch_size:(index+1)*self.batch_size]
```

Figure 18: DataGenerator function

```
        # Generate data
        for i, ID in enumerate(list_IDs_batch):
            im_name = self.df['ImageId'].iloc[ID]
            img_path = f"{self.base_path}/{im_name}"
            img = self.__load_grayscale(img_path)

            # Store samples
            X[i,] = img

        return X

    def __generate_y(self, list_IDs_batch):
        y = np.empty((self.batch_size, *self.dim, self.n_classes), dtype=int)

        for i, ID in enumerate(list_IDs_batch):
            im_name = self.df['ImageId'].iloc[ID]
            image_df = self.target_df[self.target_df['ImageId'] == im_name]

            rles = image_df['EncodedPixels'].values
            masks = build_masks(rles, input_shape=self.dim)

            y[i, ] = masks

        return y

    def __load_grayscale(self, img_path):
        img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
        img = img.astype(np.float32) / 255.
        img = np.expand_dims(img, axis=-1)

        return img
```

Figure 19: Functions to create image into grayscale and RGB format

The model is trained over 7 epochs and dice coefficient is calculated for each epoch.

```
checkpoint = ModelCheckpoint(
    'model1.h5',
    monitor='val_dice_coef',
    verbose=0,
    save_best_only=True,
    save_weights_only=False,
    mode='auto'
)

history = model1.fit_generator(
    train_generator,
    validation_data=val_generator,
    callbacks=[checkpoint],
    use_multiprocessing=False,
    workers=1,
    epochs=7
)
```

```
Epoch 1/7
177/177 [==============================] - 265s 1s/step - loss: -3866902931607909369856.0000 - dice_coef: 0.6955 - val_loss: -6996679791092118323200.0000 - val_dice_coef: 0.6877
Epoch 2/7
177/177 [==============================] - 249s 1s/step - loss: -9289937371560017290854.4000 - dice_coef: 0.6893 - val_loss: -1117556988532920156160000.0000 - val_dice_coef: 0.6882
Epoch 3/7
177/177 [==============================] - 251s 1s/step - loss: -1302837310174147400171520.0000 - dice_coef: 0.6897 - val_loss: -1451135730628491277762560.0000 - val_dice_coef: 0.6880
Epoch 4/7
177/177 [==============================] - 252s 1s/step - loss: -1621595315535856145530880.0000 - dice_coef: 0.6894 - val_loss: -1757142805301352390983680.0000 - val_dice_coef: 0.6878
Epoch 5/7
177/177 [==============================] - 250s 1s/step - loss: -1931495023877419119411200.0000 - dice_coef: 0.6892 - val_loss: -2059157078616579369861120.0000 - val_dice_coef: 0.6875
Epoch 6/7
177/177 [==============================] - 251s 1s/step - loss: -2234596785720158679203840.0000 - dice_coef: 0.6887 - val_loss: -2354589070860426726604800.0000 - val_dice_coef: 0.6871
Epoch 7/7
177/177 [==============================] - 252s 1s/step - loss: -2529897859086160096133120.0000 - dice_coef: 0.6885 - val_loss: -2641776593482424558551040.0000 - val_dice_coef: 0.6868
```

Figure 20: Model trained and validated

8. Implementation and Evaluation of Mask RCNN Model

Required libraries of python were imported for the implementation of Mask RCNN model.

```
# Basics
from glob import glob # finds pathnames
import os # Miscellaneous operating system interfaces
import sys
import random
import timeit
import imp
import gc

import numpy as np
import pandas as pd
from scipy.ndimage import label as scipy_label
from scipy.ndimage import generate_binary_structure
from dices import *

from mrcnn.config import Config
import mrcnn.utils as utils
import mrcnn.model as modellib
import mrcnn.visualize as visualize
from mrcnn.model import log

import matplotlib.pyplot as plt

from steel_dataset import SteelDataset
from steel_config import s
```

Figure 21: Python libraries were imported

Model has been used in inference model to learn about data generation process. Also, the function for dice coefficient is used for evaluation.

```
# create model for inferring
infer_config = SteelConfig()
infer_config.BATCH_SIZE=1
infer_config.IMAGES_PER_GPU=1
infer_config.DETECTION_MIN_CONFIDENCE=.5
infer_config.DETECTION_NMS_THRESHOLD=.5
model_infer = modellib.MaskRCNN(mode="inference", config=infer_config, model_dir=MODEL_DIR)
model_infer.load_weights('logs/steel20191209T0236/mask_rcnn_steel_0091.h5', by_name=True)
```

```
from tqdm import tqdm
# calculate validation set positive and negative indices
pos = 0
neg = 0
for image_id in tqdm(range(500)):
# for image_id in range(3):
    image = dataset_train.load_image(image_id)
    mask, class_ids = dataset_train.load_mask(image_id)
    rr = model_infer.detect([image])
    r = rr[0]
    score = get_score(mask, class_ids, r)
    pos += score[0]
    neg += score[1]
pos /= len(val)
neg /= len(val)
```

Figure 22: Creation of model for inference

9. Implementation and Evaluation of UNet++ Model

Required libraries of python were evaluated for the implementation of UNet++ model.

```
import os
import json
import gc

import albumentations as albu
import cv2
import keras
from keras import backend as K
from keras.engine import Layer, InputSpec
from keras.utils.generic_utils import get_custom_objects
from keras import initializers, constraints, regularizers, layers
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Model, load_model
from keras.layers import Input, Dropout, Conv2D, BatchNormalization, add
from keras.layers.convolutional import Conv2D, Conv2DTranspose
from keras.layers.pooling import MaxPooling2D
from keras import backend as K
from keras.layers import LeakyReLU
from keras.losses import binary_crossentropy
from keras.layers.merge import concatenate, Concatenate, Add
from keras.optimizers import Adam
from keras.callbacks import Callback, ModelCheckpoint
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from tqdm import tqdm
from sklearn.model_selection import train_test_split

import efficientnet.keras as efn
```

Figure 23: Python libraries were imported

'DataGenerator' function is used to generate data into a single batch where each batch denotes per epoch.

```
class DataGenerator(keras.utils.Sequence):
    'Generates data for Keras'
    def __init__(self, list_IDs, df, target_df=None, mode='fit',
                 base_path='../input/severstal-steel-defect-detection/train_images',
                 batch_size=32, dim=(256, 1600), n_channels=3, reshape=None,
                 augment=False, n_classes=4, random_state=2019, shuffle=True):
        self.dim = dim
        self.batch_size = batch_size
        self.df = df
        self.mode = mode
        self.base_path = base_path
        self.target_df = target_df
        self.list_IDs = list_IDs
        self.reshape = reshape
        self.n_channels = n_channels
        self.augment = augment
        self.n_classes = n_classes
        self.shuffle = shuffle
        self.random_state = random_state

        self.on_epoch_end()
        np.random.seed(self.random_state)

    def __len__(self):
        'Denotes the number of batches per epoch'
        return int(np.floor(len(self.list_IDs) / self.batch_size))

    def __getitem__(self, index):
        'Generate one batch of data'
        # Generate indexes of the batch
        indexes = self.indexes[index*self.batch_size:(index+1)*self.batch_size]

        # Find list of IDs
        list_IDs_batch = [self.list_IDs[k] for k in indexes]

        X = self.__generate_X(list_IDs_batch)

        if self.mode == 'fit':
            y = self.__generate_y(list_IDs_batch)

            if self.augment:
                X, y = self.__augment_batch(X, y)

            return X, y

        elif self.mode == 'predict':
            return X

        else:
            raise AttributeError('The mode parameter should be set to "fit" or "predict".')
```

Figure 24: 'DataGenerator' function

‘EfficientUNet’ function is a model builder

```
def EfficientUNet(input_shape):
    backbone = efn.EfficientNetB4(
        weights=None,
        include_top=False,
        input_shape=input_shape
    )

    backbone.load_weights(('../input/efficientnet-keras-weights-b0b5/'
                           'efficientnet-b4_imagenet_1000_notop.h5'))

    # Skipping block 4f and block 6h since they have the same output dim as 5f and 7b
    x00 = backbone.input  # (256, 512, 3)
    x10 = backbone.get_layer('stem_activation').output  # (128, 256, 4)
    x20 = backbone.get_layer('block2d_add').output  # (64, 128, 32)
    x30 = backbone.get_layer('block3d_add').output  # (32, 64, 56)
    x40 = backbone.get_layer('block5f_add').output  # (16, 32, 160)
    x50 = backbone.get_layer('block7b_add').output  # (8, 16, 448)

    x01 = H([x00, U(x10)], 'X01')
    x11 = H([x10, U(x20)], 'X11')
    x21 = H([x20, U(x30)], 'X21')
    x31 = H([x30, U(x40)], 'X31')
    x41 = H([x40, U(x50)], 'X41')

    x02 = H([x00, x01, U(x11)], 'X02')
    x12 = H([x11, U(x21)], 'X12')
    x22 = H([x21, U(x31)], 'X22')
    x32 = H([x31, U(x41)], 'X32')

    x03 = H([x00, x01, x02, U(x12)], 'X03')
    x13 = H([x12, U(x22)], 'X13')
    x23 = H([x22, U(x32)], 'X23')

    x04 = H([x00, x01, x02, x03, U(x13)], 'X04')
    x14 = H([x13, U(x23)], 'X14')

    x05 = H([x00, x01, x02, x03, x04, U(x14)], 'X05')

    x_out = Concatenate(name='bridge')([x01, x02, x03, x04, x05])
    x_out = Conv2D(4, (3,3), padding="same", name='final_output', activation="sigmoid")(x_out)

    return Model(inputs=x00, outputs=x_out)
```

Figure 25: EfficientUNet function

Model has been trained for the batch size of 8

```
BATCH_SIZE = 8

train_idx, val_idx = train_test_split(
    non_missing_train_idx.index,  # NOTICE DIFFERENCE
    random_state=2019,
    test_size=0.2
)

train_generator = DataGenerator(
    train_idx,
    reshape=(256, 512),
    df=mask_count_df,
    target_df=train_df,
    augment=True,
    batch_size=BATCH_SIZE,
    n_classes=4
)

val_generator = DataGenerator(
    val_idx,
    reshape=(256, 512),
    df=mask_count_df,
    target_df=train_df,
    augment=False,
    batch_size=BATCH_SIZE,
    n_classes=4
)
```

Figure 26: Creating the batches for image data

Weights are designed to train the model for the purpose of making predictions.

```
checkpoint = ModelCheckpoint(
    'model.h5',
    monitor='val_loss',
    verbose=0,
    save_best_only=True,
    save_weights_only=False,
    mode='auto'
)

history = model.fit_generator(
    train_generator,
    validation_data=val_generator,
    callbacks=[checkpoint],
    verbose=1,
    epochs=30
)
```

Figure 27: Weights were assigned to the model

Model has been evaluated using Dice Coefficient

```
with open('history.json', 'w') as f:
    json.dump(history.history, f)

history_df = pd.DataFrame(history.history)
history_df[['loss', 'val_loss']].plot()
history_df[['dice_coef', 'val_dice_coef']].plot()
```

Figure 28: Dice Coefficient