**Identifying defects in Steel Images**

Binary Classification, Multilabel Classification and Image Segmentation

A popular Kaggle competition published in 2019

This is a popular competition hosted on Kaggle in 2019. Here is the [*link*](https://www.kaggle.com/c/severstal-steel-defect-detection)to the competition. Please visit this link to read more about the competition.

**Overview**

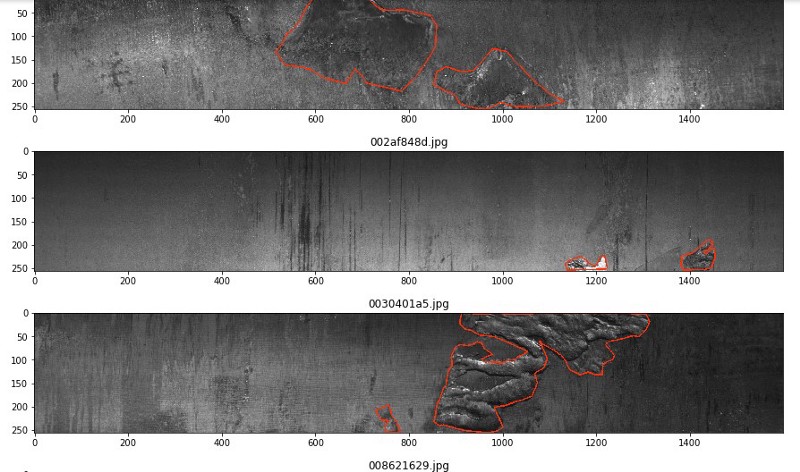
Severstal is leading the charge in efficient steel mining and production. The production process of flat sheet steel is delicate. From heating and rolling, to drying and cutting, several machines touch flat steel by the time it’s ready to ship. Today, Severstal uses images from high frequency cameras to power a defect detection algorithm. The goal of this case study is to localize and classify surface defects on a steel sheet.

Surface quality is the essential parameter for steel sheets. In the steel industry, manual defect inspection is a tedious assignment. Consequently, it is difficult to guarantee the surety of a flawless steel surface. To meet user requirements, vision-based automatic steel surface investigation strategies have been proven to be exceptionally powerful and prevalent solutions over the past two decades

**Problem Statement**

The objective of this competition is to identify the area of defect for each image and each defect type. There are 4 types of defect that an image can have. There will also be non defective images. For example, if an image has no defect then the model should output as [‘’,’’,’’,’’] to indicate no defect. For an image having defect 2 it could be [‘’,’1 23 67 5’,’’,’’] to indicate defect 2 and so on.

Few examples of images with defects with area of defect highlighted is as below:



**Data Source**

***train\_images.Zip***: Zip file containing all train images(12568 unique)  
***test\_images.zip***: Zip file containing all test images(1801 unique)  
***train.csv***: containing Imageid and Encoded Pixels columns of only defective images  
***submission.csv:*** file format for the final submission containing Imageid and Encoded Pixels columns

**Data Description**

The `train.csv` data set provided by SevarSteel contains the following features

a. ImageId — A unique identifier for the image.Example:002cc93b.jpg

b. ClassId — The class of the defect from [1,2,3,4]

c. Encoded Pixels — The pixels which have the defect. Example: 9102 12 29346 24 29602 24 29858 24 30114 24

In this competition we need to predict the location and type of defects found in steel manufacturing. Each image may have no defects, a defect of a single class, or defects of multiple classes. For each image one must segment defects of each class (ClassId = [1, 2, 3, 4]). The segment for each defect class will be encoded into a single row, even if there are several non-contiguous defect locations on an image.

Understanding the output submission - In order to reduce the submission file size, the metric uses run-length encoding on the pixel values. Instead of submitting an exhaustive list of indices for your segmentation, one will submit pairs of values that contain a start position and a run length. E.g. ‘1 3’ implies starting at pixel 1 and running a total of 3 pixels (1,2,3).

**Data Understanding**

1. The images are given in a train folder. This folder contains all the images which are defective as well as which are not defective.
2. An image can have following types of defect

a. No defect

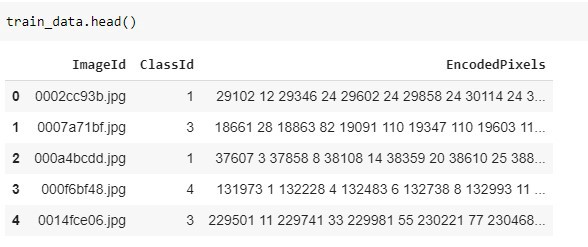
b. One defect

c. Two defects

d. Three defects

e. Four defects

3. For the images which we have defects, we have additional information about the location of the defect as follows:

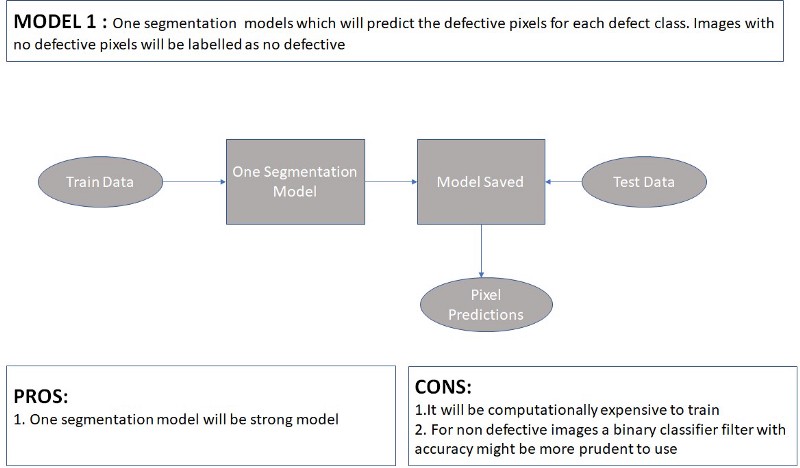
5 rows from train.csv

**Machine Learning Objective**

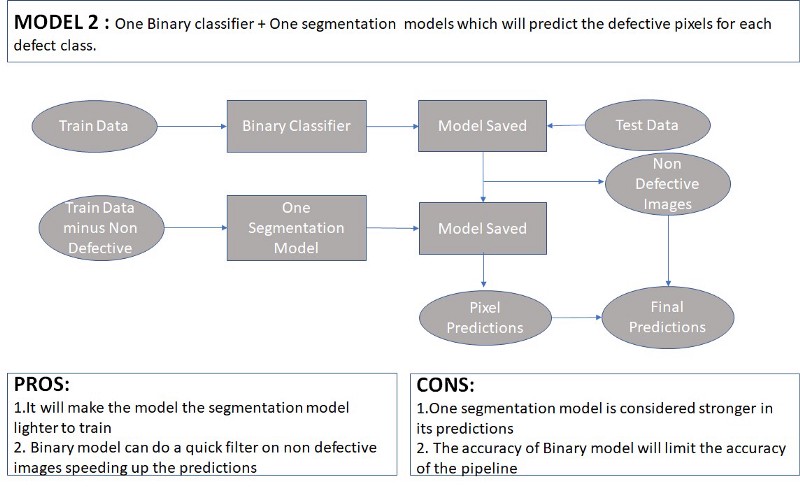
The business objective can be translated to machine learning as a Segmentation problem where we take in an image and output the same image with each pixel categorized into 5 categories [No defect, Defect1,Defect2,Defect3,Defect4]

Lets look into some of the possible architectures that can be explored

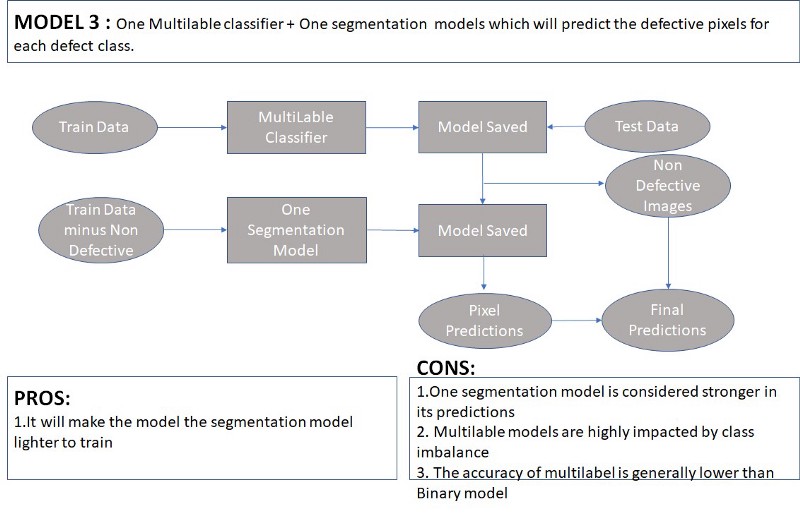
1. Single Segmentation Model



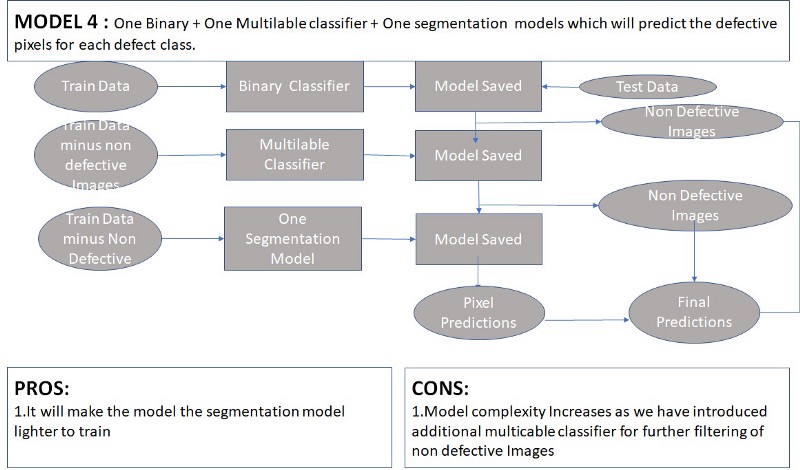
2. Binary Classifier followed by Segmentation Model



3. Multilabel Classifier followed by Segmentation Model

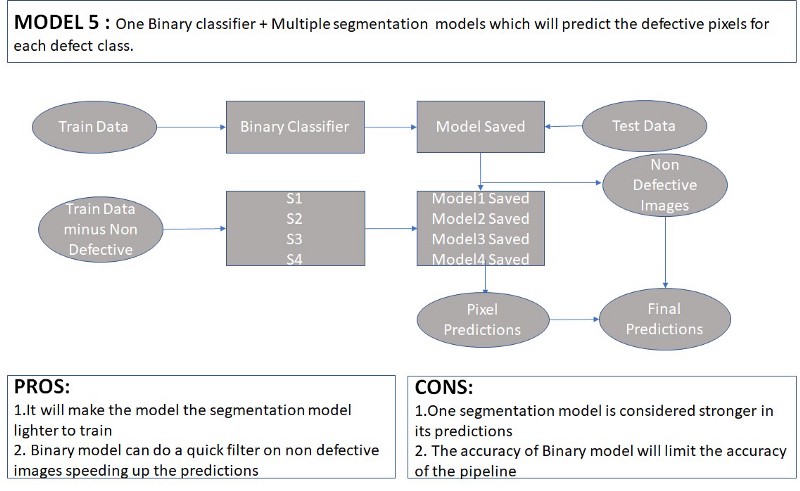


4. Binary Classifier followed by Multilabel Classifier followed by Segmentation model

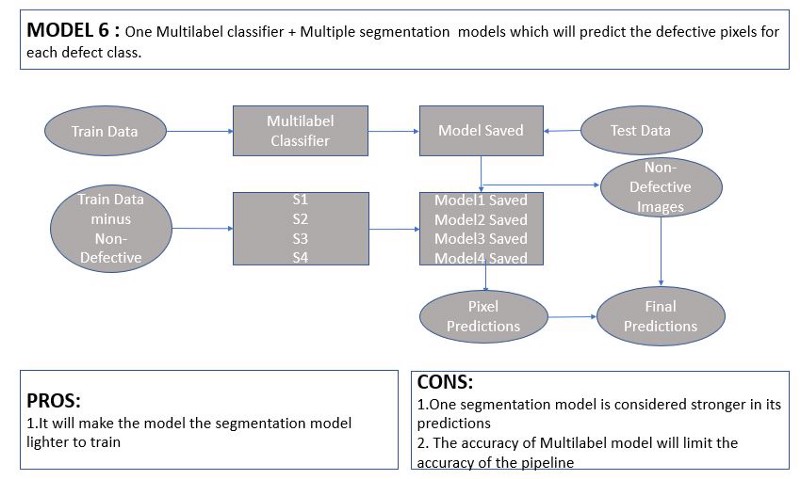


**We will be using Model 4 as our strategy in this case study**

5. One Binary Classifier followed by Multiple Segmentation Model



6. One Multilabel Classifier followed by Multiple Segmentation Model



**Metrics**

1. The metric for evaluation is dice metric which tries to measure how well the predicted defect area matches the original defect area.

2. The formula is given by:2∗|X∩Y|/|X|+|Y| where X is the predicted set of pixels and Y is the ground truth. The Dice coefficient is defined to be 1 when both X and Y are empty

3. We need to measure the average dice coef for each image and each defect

**Exploratory Data Analysis**

Images having no defects = 5902

The dataset had information only about defective images. Non defective images were found by fining all image names which were not mentioned in the defective file.

Images having defects = 6666

No of defects = 7095



**Defective versus Non defective the class is almost balanced with 5902 and 7095 data points respectively**

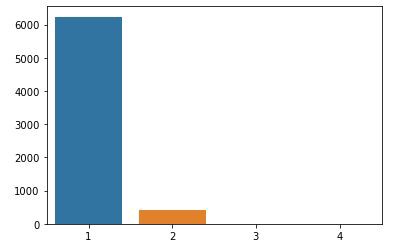
Image defect distribution is as follows:

Images with one defect = 6239

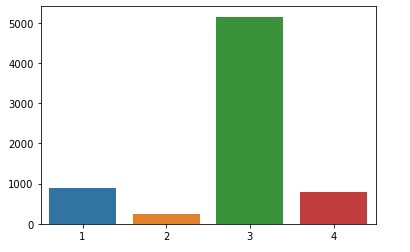
Images with two defects = 425

Images with three defects = 2

Images with four defects = 0



There are 4 categories of defect and the distribution of defect type is as follows:

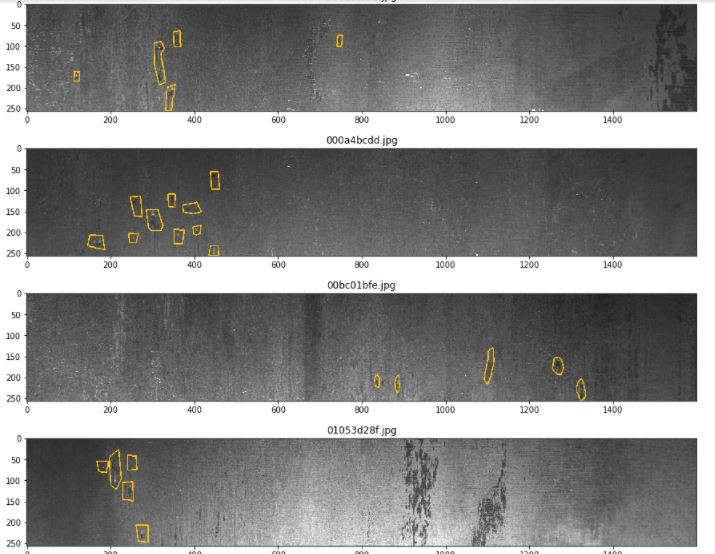
Distribution of defect type

As we can see Defect\_Class\_3 is the highest and this is a heavily imbalanced dataset within defective class

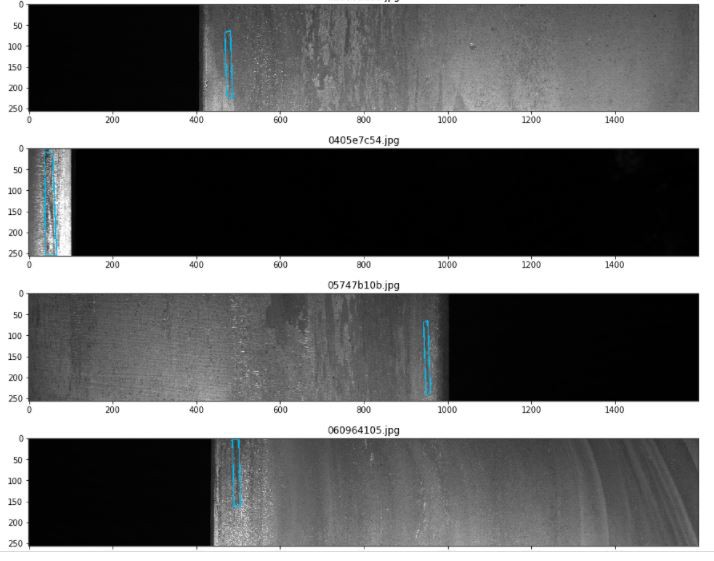
**Visualizing each defect type**

Let us see some images of each class. This visualization can be done easily by masking given encoded pixels on the train data images, you can refer for code [here](https://www.kaggle.com/go1dfish/clear-mask-visualization-and-simple-eda).

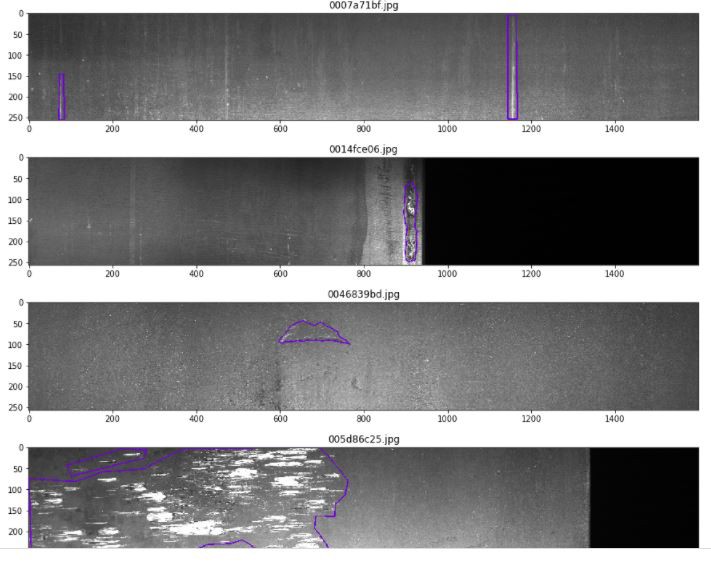
**Defect 1**



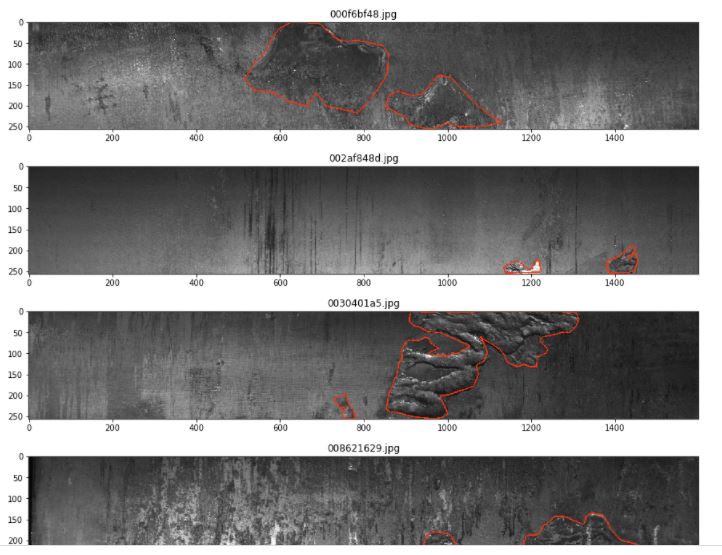
**Defect 2**



**Defect 3**



**Defect 4**



**EDA Conclusions**

1. Defective versus Non defective the class is almost balanced with 5902 and 7095 data points respectively
2. There are 4 Defect classes that needs to be predicted for Defective images.
3. These classed are highly imbalanced with distribution of Images across defects as follows({Defect\_3: 5150, Defect\_1: 897, Defect\_4: 801, Defect\_2: 247})
4. Most images have just 1 defect with others having 2 defects. Images with 3 or 4 defects are rare. {Images with 1 defect : 6239, Images with 2 defect: 425, Images with 3 defects: 2,Images with 4 defects: 0}
5. All images across train and test are of the same size

**Model Training**

We will start from Binary Classification model. This will be followed by Multilabel Classification model and Segmentation model.

**Binary Classification**

**Data Splitting**

The data we have is not time series so we can employ random data split strategy.

X\_train\_data, X\_test\_data= train\_test\_split(train\_data, test\_size=0.1,stratify=train\_data['no\_of\_defects'] ,random\_state=42)

X\_train\_data, X\_val\_data= train\_test\_split(X\_train\_data, test\_size=0.2,stratify=X\_train\_data['no\_of\_defects'],random\_state=4)

X\_train = X\_train\_data['ImageId']  
y\_train = X\_train\_data['has\_defect']  
X\_val = X\_val\_data['ImageId']  
y\_val = X\_val\_data['has\_defect']  
X\_test = X\_test\_data['ImageId']  
y\_test = X\_test\_data['has\_defect']

We are using Tensorflow 2.0 to train the model hence dataset has to be converted to tensors.

train\_dataset = tf.data.Dataset.from\_tensor\_slices((X\_train, y\_train))  
val\_dataset = tf.data.Dataset.from\_tensor\_slices((X\_val, y\_val))  
test\_dataset = tf.data.Dataset.from\_tensor\_slices((X\_test, y\_test))

**We need to create Functions to Read Images, Augment Images and Resize Image so that they can be fed to machine learning models**

#https://cs230.stanford.edu/blog/datapipeline/  
#https://www.tensorflow.org/guide/data

base\_path ='train\_images/'

**#function to read the image and convert into pixel**  
def parse\_function(image\_path,label):  
 image\_string = tf.io.read\_file(base\_path+image\_path)  
 image = tf.image.decode\_jpeg(image\_string, channels=3)

#This will convert to float values in [0, 1]  
 image = tf.image.convert\_image\_dtype(image, tf.float32)  
 print(image)  
 print(label)  
 return image, label

**#function to augment the images**def augment(image, label):  
 image = tf.image.random\_flip\_left\_right(image)  
 image = tf.image.random\_flip\_up\_down(image)  
 image = tf.image.random\_brightness(image, max\_delta = 0.4)  
 image = tf.image.random\_crop(image, size=[224, 1568, 3])  
 image = tf.clip\_by\_value(image, 0.0, 1.0)  
 return image, label

**#function to resize the images**def image\_resize(image,label):  
 resized\_image = tf.image.resize(image, [256, 512])  
 resized\_image = tf.clip\_by\_value(resized\_image, 0.0, 1.0)  
 return resized\_image, label

**Dataset pre processing where image augmentation is performed**

**#https://www.tensorflow.org/tutorials/images/data\_augmentation**BATCH\_SIZE = 8  
augmented\_train\_batches =(train\_dataset  
.cache()  
.shuffle(X\_train.shape[0]//4,reshuffle\_each\_iteration=True)  
# The augmentation is added here.  
.map(parse\_function, num\_parallel\_calls=AUTOTUNE)  
.map(augment, num\_parallel\_calls=AUTOTUNE)  
.map(image\_resize)  
.batch(BATCH\_SIZE)  
.prefetch(1))

val\_batches = (val\_dataset  
.cache()  
.map(parse\_function, num\_parallel\_calls=AUTOTUNE)  
.map(image\_resize)  
.batch(BATCH\_SIZE)  
.prefetch(1))

test\_batches = (test\_dataset  
.cache()  
.map(parse\_function, num\_parallel\_calls=AUTOTUNE)  
.map(image\_resize)  
.batch(1)  
.prefetch(1))

augmented\_test\_batches = (test\_dataset  
.cache()  
# The augmentation is added here.  
.map(parse\_function, num\_parallel\_calls=AUTOTUNE)  
.map(augment, num\_parallel\_calls=AUTOTUNE)  
.map(image\_resize)  
.batch(1)  
.prefetch(1))

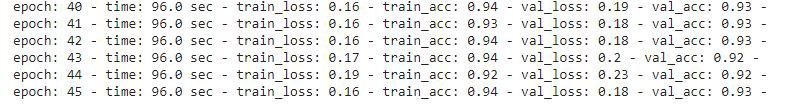
**Machine Learning Binary Classification using InceptionResNetV2 trained on ImageNet dataset as the pre trained model**

# Define the input  
input = Input(shape=(256,512,3))  
# Append the pre trained model  
trained\_model\_output = pre\_trained\_model.output  
# Append a pooling layer  
glb\_pool = GlobalAveragePooling2D()(trained\_model\_output)  
# Define first layer of Dense-BatchNorm-Droput  
layer1 = Dense(1024,activation='relu')(glb\_pool)  
batch\_norm1 = BatchNormalization()(layer1)  
dropout1 = Dropout(0.1)(batch\_norm1)  
# Define second layer of Dense-BatchNorm-Droput  
layer2 = Dense(512,activation='relu')(dropout1)  
batch\_norm2 = BatchNormalization()(layer2)  
dropout2 = Dropout(0.1)(batch\_norm2)  
# Define third layer of Dense-BatchNorm-Droput  
layer3 = Dense(256,activation='relu')(dropout2)  
batch\_norm3 = BatchNormalization()(layer3)  
# dropout3 = Dropout(0.3)(batch\_norm3)  
out = Dense(1,activation='sigmoid')(batch\_norm3)

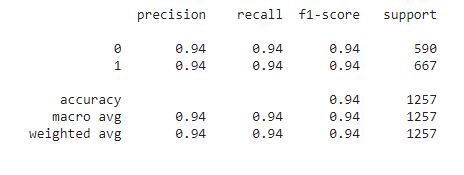
Gradient Tape is used to perform model training

**Results**

After a lot of training we see that loss has been minimized and is stable.



An accuracy of 94% was found on the test dataset.



**Multilabel Classification**

**Data Splitting**

X\_train\_multilable = X\_train\_data['ImageId']

y\_train\_multilable=X\_train\_data[['has\_defect\_1','has\_defect\_2','has\_defect\_3','has\_defect\_4']]

X\_val\_multilable = X\_val\_data['ImageId']

y\_val\_multilable=X\_val\_data[['has\_defect\_1','has\_defect\_2','has\_defect\_3','has\_defect\_4']]

X\_test\_multilable = X\_test\_data['ImageId']

y\_test\_multilable=X\_test\_data[['has\_defect\_1','has\_defect\_2','has\_defect\_3','has\_defect\_4']]

**Creating Tensors**

train\_dataset\_multilable=tf.data.Dataset.from\_tensor\_slices((X\_train\_multilable, y\_train\_multilable))

val\_dataset\_multilable=tf.data.Dataset.from\_tensor\_slices((X\_val\_multilable, y\_val\_multilable))

test\_dataset\_multilable=tf.data.Dataset.from\_tensor\_slices((X\_test\_multilable, y\_test\_multilable))

**Data Preprocessing**

#https://www.tensorflow.org/tutorials/images/data\_augmentation

BATCH\_SIZE = 8

augmented\_train\_batches\_multilable = (train\_dataset\_multilable  
.cache()  
.shuffle(X\_train.shape[0]//4,reshuffle\_each\_iteration=True)  
# The augmentation is added here.  
.map(parse\_function, num\_parallel\_calls=AUTOTUNE)  
.map(augment, num\_parallel\_calls=AUTOTUNE)  
.map(image\_resize)  
.batch(BATCH\_SIZE)  
.prefetch(1))

val\_batches\_multilable = (val\_dataset\_multilable  
.cache()  
.map(parse\_function, num\_parallel\_calls=AUTOTUNE)  
.map(image\_resize)  
.batch(BATCH\_SIZE)  
.prefetch(1))

test\_batches\_multilable = (test\_dataset\_multilable  
.cache()  
.map(parse\_function, num\_parallel\_calls=AUTOTUNE)  
.map(image\_resize)  
.batch(1)  
.prefetch(1))

augmented\_test\_batches\_multilable = (test\_dataset\_multilable  
.cache()  
# The augmentation is added here.  
.map(parse\_function, num\_parallel\_calls=AUTOTUNE)  
.map(augment, num\_parallel\_calls=AUTOTUNE)  
.map(image\_resize)  
.batch(1)  
.prefetch(1))

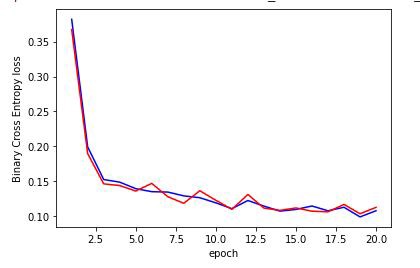
**Machine Learning Multilabel Classification using InceptionResNetV2 trained on ImageNet dataset as the pre trained model**

**# Define the input**  
input = Input(shape=(256,512,3))  
**# Append the pre trained model**  
trained\_model\_output = pre\_trained\_model.output  
**# Append a pooling layer**  
glb\_pool = GlobalAveragePooling2D()(trained\_model\_output)  
**# Define first layer of Dense-BatchNorm-Droput**  
layer1 = Dense(1024,activation='relu')(glb\_pool)  
batch\_norm1 = BatchNormalization()(layer1)  
dropout1 = Dropout(0.0)(batch\_norm1)  
**# Define second layer of Dense-BatchNorm-Droput**  
layer2 = Dense(512,activation='relu')(dropout1)  
batch\_norm2 = BatchNormalization()(layer2)  
dropout2 = Dropout(0.0)(batch\_norm2)  
**# Define third layer of Dense-BatchNorm-Droput**  
layer3 = Dense(128,activation='relu')(dropout2)  
batch\_norm3 = BatchNormalization()(layer3)  
**# dropout3 = Dropout(0.0)(batch\_norm3)**  
out =Dense(4,activation='sigmoid')(batch\_norm3)

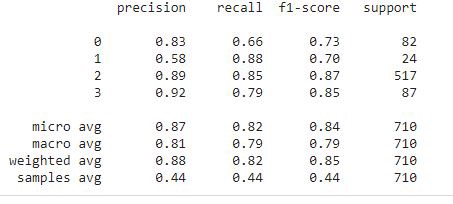
This model is almost similar as the binary classification model except the output layer which has 4 outputs defined by **Dense(4,activation=’sigmoid’)**

**Model Results**

After training for few epochs loss is minimized and is stable



An accuracy of 84% was found on the test dataset

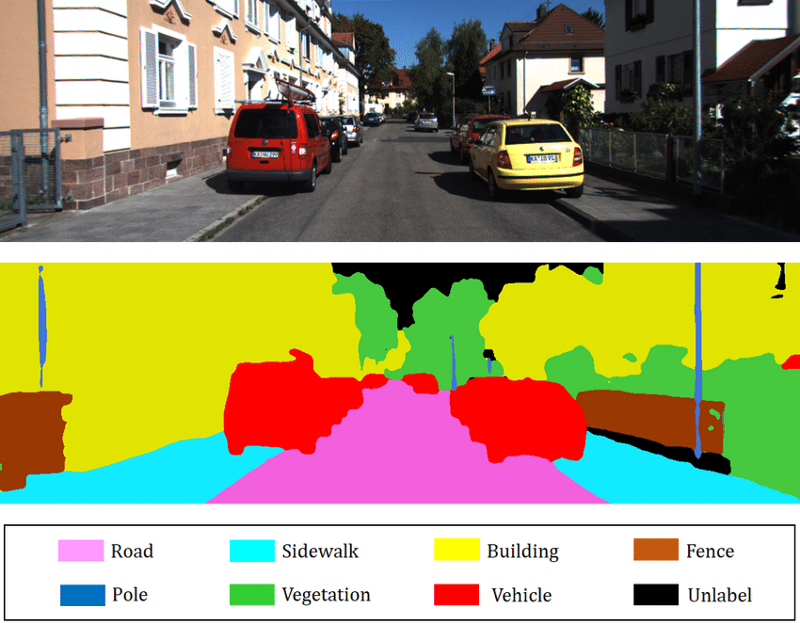


An important technique which often tends to improve the results is called **Test Time Augmentation** which was employed in this case study

You can read more about it [here](https://towardsdatascience.com/test-time-augmentation-tta-and-how-to-perform-it-with-keras-4ac19b67fb4d)

**Binary Segmentation Model**

**Semantic segmentation** achieves fine-grained inference by making predictions inferring labels for every pixel, so that each pixel is labeled with the class of its enclosing object.

<https://www.researchgate.net/figure/Example-of-2D-semantic-segmentation-Top-input-image-Bottom-prediction_fig3_326875064>

A separate segmentation model was built for each defect class.

**Data Splitting**

train\_data\_1 = X\_train\_data[X\_train\_data['has\_defect\_1']==1][['ImageId','Defect\_1']]  
train\_data\_2 = X\_train\_data[X\_train\_data['has\_defect\_2']==1][['ImageId','Defect\_2']]  
train\_data\_3 = X\_train\_data[X\_train\_data['has\_defect\_3']==1][['ImageId','Defect\_3']]  
train\_data\_4 = X\_train\_data[X\_train\_data['has\_defect\_4']==1][['ImageId','Defect\_4']]

val\_data\_1 = X\_val\_data[X\_val\_data['has\_defect\_1']==1][['ImageId','Defect\_1']]  
val\_data\_2 = X\_val\_data[X\_val\_data['has\_defect\_2']==1][['ImageId','Defect\_2']]  
val\_data\_3 = X\_val\_data[X\_val\_data['has\_defect\_3']==1][['ImageId','Defect\_3']]  
val\_data\_4 = X\_val\_data[X\_val\_data['has\_defect\_4']==1][['ImageId','Defect\_4']]

test\_data\_1 = X\_test\_data[X\_test\_data['has\_defect\_1']==1][['ImageId','Defect\_1']]  
test\_data\_2 = X\_test\_data[X\_test\_data['has\_defect\_2']==1][['ImageId','Defect\_2']]  
test\_data\_3 = X\_test\_data[X\_test\_data['has\_defect\_3']==1][['ImageId','Defect\_3']]  
test\_data\_4 = X\_test\_data[X\_test\_data['has\_defect\_4']==1][['ImageId','Defect\_4']]

**RLE(Run Length Encoder)**

In order to reduce the submission file size, the metric uses run-length encoding on the pixel values*.* Instead of submitting an exhaustive list of indices for your segmentation, you will submit pairs of values that contain a start position and a run length. E.g. ‘1 3’ implies starting at pixel 1 and running a total of 3 pixels.

**def mask2rle(mask):**  
 mask\_flatten = mask.flatten(order = 'F')  
 # We will create another mask shifted by one digit and compare side by side  
 masks\_flatten\_append0 = np.concatenate([[0], mask\_flatten, [0]])  
 runs=np.where(masks\_flatten\_append0[1:]!=masks\_flatten\_append  
[:-1])[0]+1  
 runs[1::2] -= runs[::2]  
 return ' '.join(str(x) for x in runs)

@tf.function()  
def tf\_function\_mask2rle(input):  
 y = tf.numpy\_function(mask2rle, [input], (tf.string))  
 return y

**def rle2mask(rle,shape=(256,1600)):**  
 mask\_label = np.zeros(shape[0]\*shape[1],dtype=np.float32)  
 if not isinstance(rle, str):  
 rle = rle.decode("utf-8")  
 if rle is not np.nan and rle is not '':  
 masks = rle.split(" ")  
 positions = map(int, masks[0::2])  
 lengths = map(int, masks[1::2])  
 for pos, le in zip(positions,lengths):  
 mask\_label[pos-1:pos+le-1] = 1.  
 # print(mask\_label.reshape(shape[0], shape[1], order='F').shape)

return np.expand\_dims(mask\_label.reshape(shape[0], shape[1], order='F'),-1)

@tf.function()  
def tf\_function\_rle2mask(input):  
 y = tf.numpy\_function(rle2mask, [input], (tf.float32))  
 y = tf.reshape(y,shape=(256,1600,1))  
 return y

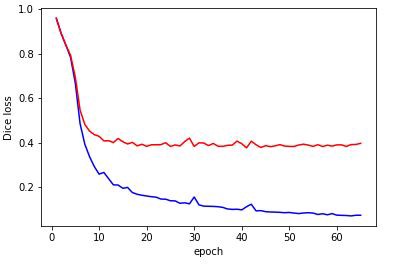
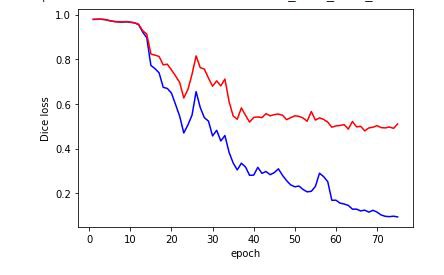
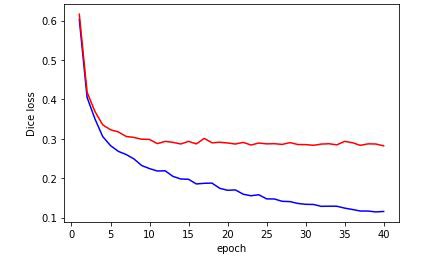
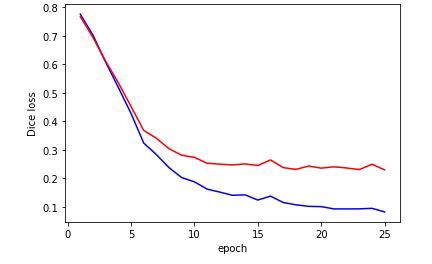
**Importing Segmentation Model**

!pip install segmentation\_models  
import segmentation\_models as sm  
sm.set\_framework('tf.keras')  
from segmentation\_models import Unet

#https://segmentation-models.readthedocs.io/en/latest/tutorial.htmlhttps://github.com/qubvel/segmentation\_models

segmentation\_model\_1 = sm.Unet('efficientnetb1', input\_shape= (256,512,3), classes=1, encoder\_weights='imagenet', activation ='sigmoid')  
segmentation\_model\_1.summary()

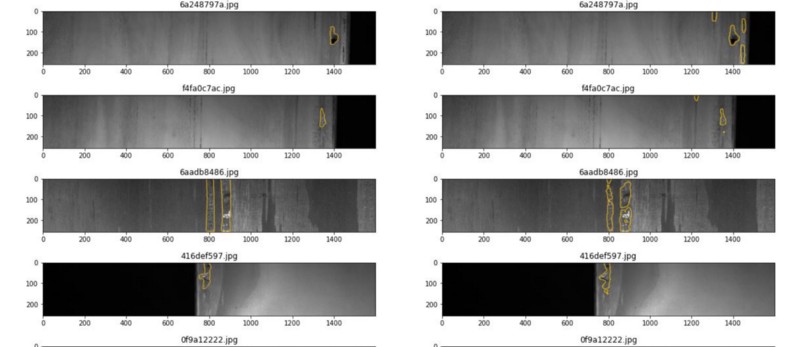
**Results**

**Defect Class 1 train and test loss****Defect Class 2 train and test loss****Defect Class 3 train and test loss****Defect Class 4 train and test loss**

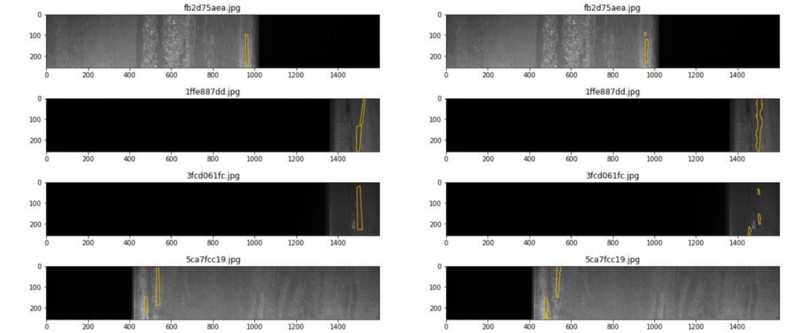
**Lower dice loss indicates better performance and we can see for Defect Class 2 the loss if quiet high.**

Lets see visually some of the predictions. Left side is the actual picture and right side is the predicted picture.

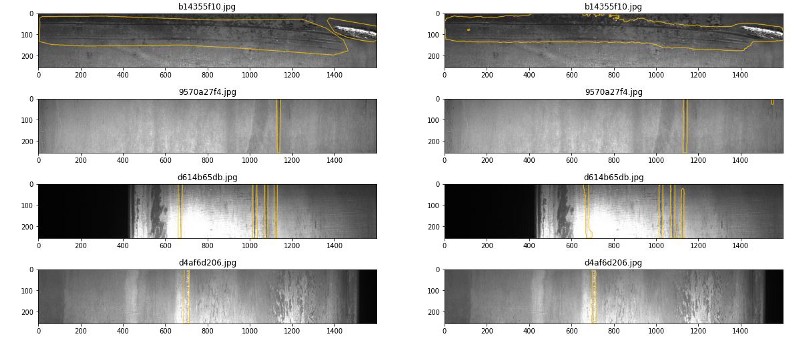
**Defect\_Class\_1**



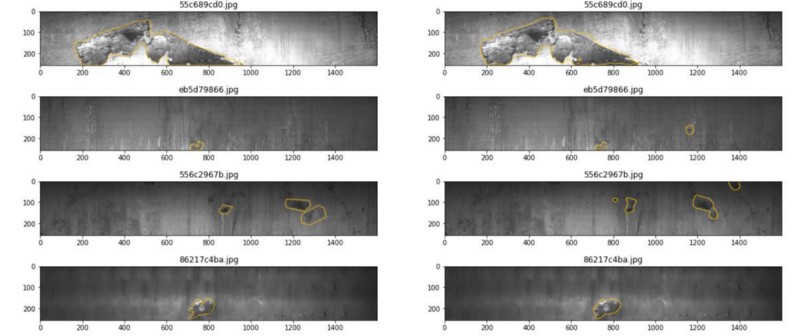
**Defect\_Class\_2**



Defect\_Class\_3



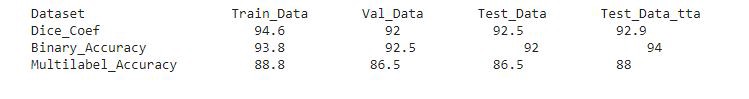
**Defect\_Class\_4**



After all the models are built the final predictions have further massaging as follows

1. For Binary Classification, a threshold of 0.55 was considered
2. For Segmentation model defects greater than a particular area was only considered. For example for Defect\_Class\_1, area\_defect should be > 400. For Defect\_Class\_2, area\_defect should be >500. For Defect\_Class\_3, area\_defect should be >800. For Defect\_Class\_4, area\_defect should be >1900.

The final results were summarized as follows:



**Next Steps**

1. We have experimented with one architecture. It would be good to see how other architectures perform.
2. Data/Image augmentation was experimented but not in entirety and would be make sense to explore and experiment with some advanced image augmentation techniques.

**References**

1. <https://www.kaggle.com/c/severstal-steel-defect-detection/discussion/114309?rvi=1>
2. <https://www.kaggle.com/go1dfish/clear-mask-visualization-and-simple-eda>
3. <https://medium.com/analytics-vidhya/severstal-steel-defect-detection-2d2e836855c2>
4. <https://github.com/qubvel>
5. <https://www.kaggle.com/c/severstal-steel-defect-detection/discussion/114254>
6. Appliedaicourse.com

**Github Repository**

[**reachkasturi - Repositories**  
*reachkasturi has one repository available. Follow their code on GitHub.*github.com](https://github.com/reachkasturi?tab=repositories)

**Contact**

<https://www.linkedin.com/in/kasturi-karuna-b5456564/>