Detection and Segmentation of Defects on Metal Surfaces using Mask-RCNN

Module: VISION SYSTEMS

Term: Year 2, Semester 1

Lecturer: Dr. Tan Jen Hong

Students:

Full Name	Student ID
Santhiyapillai Rajeevan Paul	A0195399E
Ani Sivaram Porkalangad Ayyappan	A0195404E
Ritesh Munjal	A0195304H

Submission Date: 10 May 2020

Table of Contents

1. PROBLEM STATEMENT

Based on the learning we had in this Semester, we knew that the vision system techniques could be very effective to resolve real world problems. As a team we started to look for projects which can be helpful for our companies. After consolidation of all the projects, we found that 'item' surface detection is one painful process in Schlumberger product quality inspection and the vision systems machine learning model could be an effective solution of the problem.

Automatic detection of defects in metal castings is a challenging task, owing to the rare occurrence and variation in appearance of defects. However, automating the process can lead to significant increases in final product quality. Convolutional neural networks (CNNs) have shown outstanding performance in both image classification and localization tasks. Currently, the process of detecting the defect in metal surfaces in Schlumberger is by taking pictures under the microscope manually and rejecting if the defects are visible. All defects were detected at a magnification of x10.5 at the eyepiece of the stereo microscope and images taken at x0.7 zoom on the stereo microscope. It is quite cumbersome to manually detect the defects as there are quite a lot of 'items' assembled in the product line every day. This is where our solution becomes useful to our company Schlumberger, to automate the process.

In this project, a system is proposed for the identification of casting defects in manufactured part images, based on the mask region-based CNN architecture. The proposed defect detection system simultaneously performs defect detection and segmentation on input images, making it suitable for a range of defect detection tasks beyond the parts for which it is originally designed for. Transfer learning is leveraged to reduce the training data demands and increase the prediction accuracy of the trained model. More specifically, the model is first trained with two large openly available image datasets before fine-tuning on a relatively small (about 500 images) of the specific item photo dataset. The accuracy of the trained model is quite good and is fast enough to be proposed for quality inspection in Schlumberger. Several iterations are conducted on the models to explore how transfer learning and its fine-tuning influence the performance of the trained system.

2. DATASET

The dataset was created by taking images of the defected part extensively and consolidating the labelled images under different classes.

There are 3 main classes of defects which are indicated by the filename of the photos:

- Blowhole
- Cavity
- Crack

A blowhole refers to a cavity caught inside the die cast product. Die casting is a manufacturing process in which molten metal is forced into the mould cavity of a die and then solidifies. During this process, blowholes can occur which are caused by air bubbles that are entrapped in the melt and are not discharged during solidification.

Generally, when a blowhole occurs, impregnation treatment can fill up or close off unwanted or unnecessary cavities. However, this treatment can lead to future problems with pressure leaks or fluid leaks, in critical parts and pressure parts.

A cavity may be explained as an open hole, which can be caused due to several reasons such as insufficient drying, poor venting etc. Similarly, in high strained areas, cracks may develop often caused due to material thinning or the cast conditions.

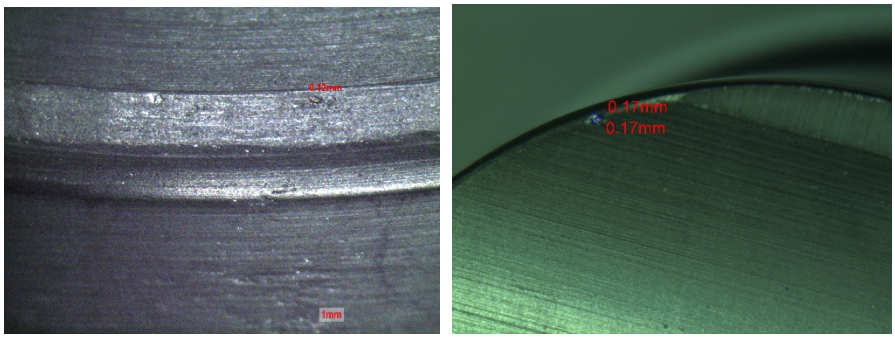


Fig 1: Images for Cavity Defect

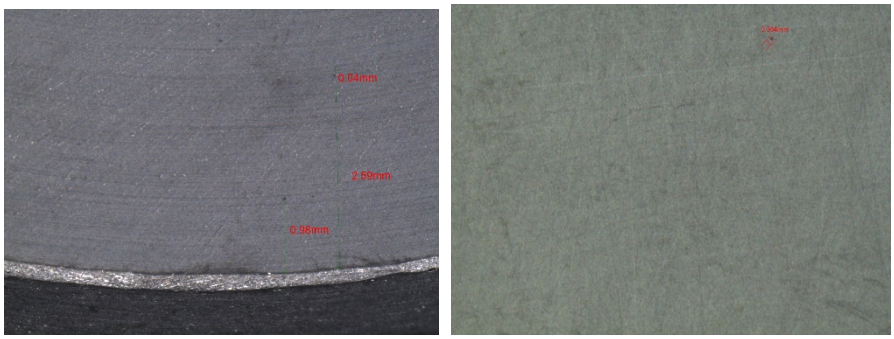


Fig 2: Images for Blow Hole

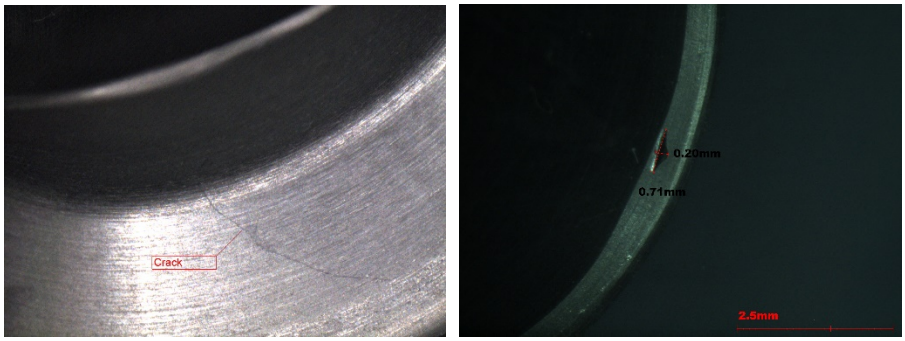


Fig 3: Images for Crack

As we can see in the above pictures, they are of varying intensities and the defects are very small to identify. The number of pictures we could get from Schlumberger were also not many (around 200 altogether from all 3 classes) so there were few challenges we identified to resolve this problem using machine learning.

1. **Challenge:** Number of images for training

The number of images we had to resolve the problem statement was not enough, so we investigated getting images from similar problems. We got more images, around 300 from Magnetic panel defect dataset from MIT to complement the original dataset. So, total we got about 500 images for our training.

Other way we overcome this challenge was by using transfer learning. We used Mask RCNN model pretrained on Severstal Steel Defect Detection dataset[7]. We used this pre-trained as our base model because it had defects which closely resemble the defects, we have in our problem statement.

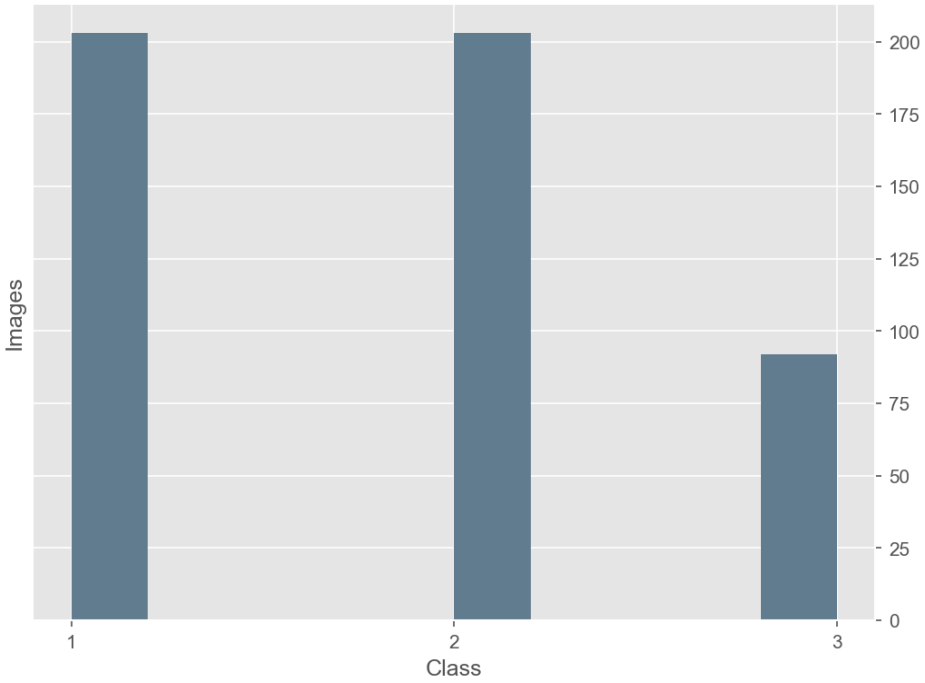


Fig 4: Distribution of dataset in classes

2. **Challenge:** Original images from Schlumberger already have markings which indicate size of defect and its scale. This causes our model to wrongly train.

For the second issue with the images, we used the tool called InPaint to manually remove all the markings on top of the images.

3. **Challenge**: For individual training images, masks had to be drawn which was time consuming.

3. DATA PRE-PROCESSING

When we had enough images to start with, we manually went through the folder structure to identify if there were images without any defects in the dataset. Such non-defect images were removed as part of data cleaning.

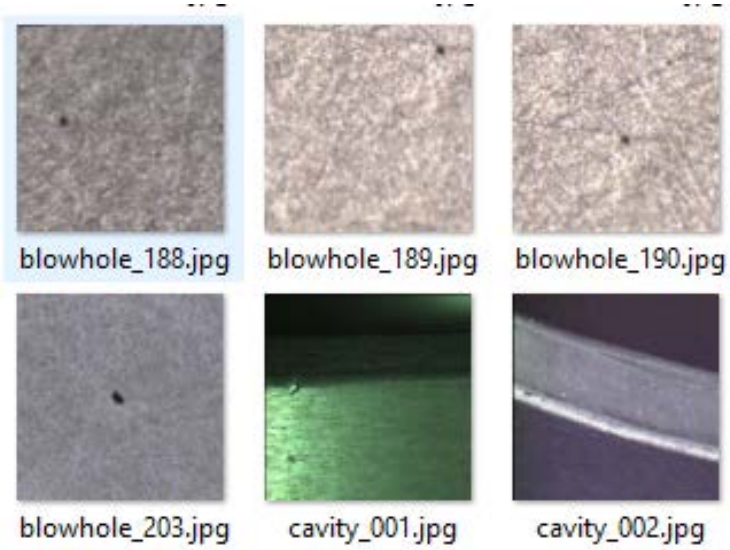


Fig 4: Images for training

Data pre-processing was done in the following order to feed quality data into the model for learning.

1. Drop duplicate images;
2. Remove images without any defects;
3. Image resize to have all images of same size before it is fed into the model;
4. Creation of appropriate mask of the defects in the training images; and
5. Creation of RLE from image mask

The dataset before augmentation had 3 classes with a total of 500 images (200 + 200 + 100) and was thus ready to go through the first iteration of training.

3.1 Image Resizing

This is one of the image pre-processing techniques used. The images were manually resized for images from the company Schlumberger. This is because some of the images were not in the 512 x 512 format.

For the images from MIT dataset, we used a python script for resizing. Using OpenCV, scaling, padding and cropping were done on the images to create a dimension of 512x512. This is because Mask-RCNN require all input images to be in 512 x 512 format.

```
for filename in os.listdir(img_rd_fldr):
        print(filename)
        ##---- Image resize ----##
        # Read image
        img = cv2.imread(os.path.join(img_rd_fldr, filename))
        #print('img:', img.shape)
        # resize image
        (image, window, scale, padding, crop) = utils.resize_image
        #print('rz img:', image.shape)
        # Write resized image
        cv2.imwrite(os.path.join(img_wrt_fldr, filename), image)
```

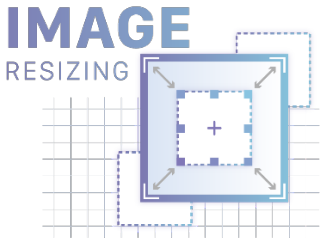


Fig 5: Image resizing

3.2 InPaint

We had to remove the markings on top of the images as some markings were wrongly identified as defects for some images. Inpaint is a software tool that we used to remove captions from the images supplied by Schlumberger.

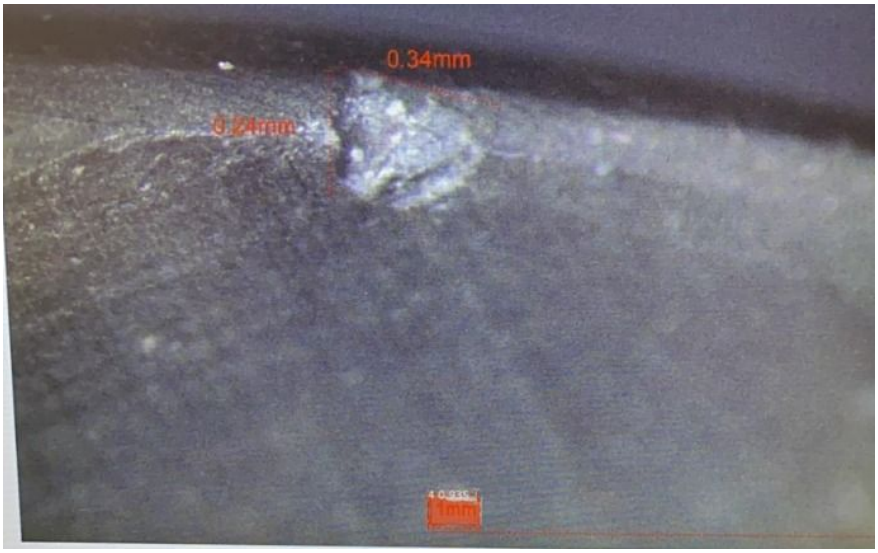


Fig 6: Wrong defect marking by Mask-RCNN (red patch towards bottom of image) due to the presence of red colour letters in source image that specifies measurements.

Source image from Schlumberger	After removal of markings on source image

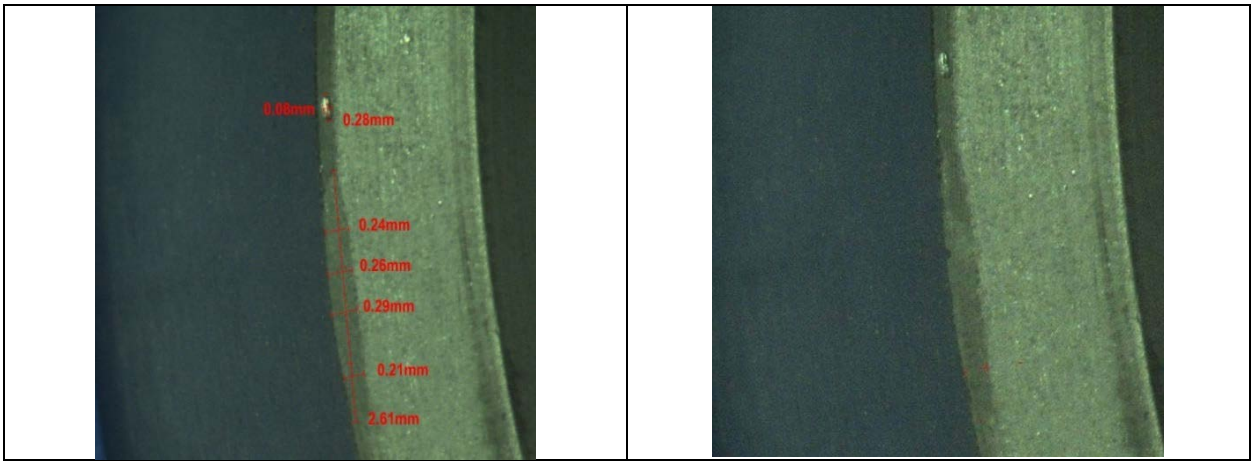


Fig 7: Image cleaning by removal of pre-existing defect measurements in red fonts

Subsequently, we used the same tool to draw masks over the cleaned images from Schlumberger as well as the image dataset from MIT. This was necessary to build a labelled dataset for training. This was a heavy lifting task when we had to spent 4 hours every day for one week to complete.

3.3 RLE

RLE (Run-Length Encoding) is a simple yet efficient format for storing binary masks. RLE first divides a vector (or vectorized image) into a series of piecewise constant regions and then for each piece simply stores the length of that piece.

Many common operations on masks can be computed directly using the RLE (without need for decoding). This includes computations such as area, union, intersection, etc. All of these operations are linear in the size of the RLE, in other words they are O(sqrt(n)) where n is the area of the object. Computing these operations on the original mask is O(n). Thus, using the RLE can result in substantial computational savings.

4. WORKING DETAILS OF THE METHOD

We started with looking at different methods (Mask RCNN, Deeplab V3 and YOLACT) as shown below.

No	Item	Link
1	Comparison of R-CNN and DeepLab	http://cs231n.stanford.edu/slides/2018/cs231n_2018_ds06.pdf
2	Performance of 3 models	https://www.youtube.com/watch?v=s8Ui_kV9dhw
3	Papers and releases	https://github.com/hoya012/deep_learning_object_detection
4	Papers and releases	https://awesomeopensource.com/project/mrgloom/awesome-semantic-segmentation
5	Deeplab	https://www.analyticsvidhya.com/blog/2019/02/tutorial-semantic-segmentation-google-deeplab/
6	Deeplab	https://ai.googleblog.com/2018/03/semantic-image-segmentation-with.html
7	Deeplab v3	https://towardsdatascience.com/review-deeplabv3-atrous-convolution-semantic-segmentation-6d818bfd1d74
8	Deeplab and FCN	https://www.learnopencv.com/pytorch-for-beginners-semantic-segmentation-using-torchvision/
9	Deeplab and FCN	https://towardsdatascience.com/review-deeplabv3-atrous-convolution-semantic-segmentation-6d818bfd1d74
10	Deeplab usage	https://www.tensorflow.org/lite/models/segmentation/over
11	Yolact	https://www.reddit.com/r/MachineLearning/comments/cuu7ce/r_yolact_realtime_instance_segmentation/
12	Yolact	https://augmentedstartups.com/yolact-real-time-instance-segmentation/
13	Yolact in realtime	https://neurohive.io/en/news/yolact-new-method-for-instance-segmentation-in-real-time/
14	Yolact Paper with comparison to Mask - RCNN	https://arxiv.org/pdf/1904.02689.pdf
15	Yolact Video	https://www.youtube.com/watch?v=DRFXG2TCRIg
16	Mask RCNN	https://www.analyticsvidhya.com/blog/2019/07/computer-vision-implementing-mask-r-cnn-image-segmentation/
17	Mask RCNN with Keras and Tensorflow	https://www.youtube.com/watch?v=lLM8oAsi32g
18	From CNN to RCNN to Yolo	https://towardsdatascience.com/computer-vision-a-journey-from-cnn-to-mask-r-cnn-and-yolo-1d141eba6e04
19	RCNN using Keras on different dataset	https://machinelearningmastery.com/how-to-perform-object-detection-in-photographs-with-mask-r-cnn-in-keras/
20	RCNN using Keras on different dataset	https://machinelearningmastery.com/how-to-train-an-object-detection-model-with-keras/
21	Masked RCNN	https://arxiv.org/pdf/1703.06870.pdf
22	Mask Scoring RCNN	https://arxiv.org/pdf/1903.00241.pdf

24	Masked RCNN	https://towardsdatascience.com/using-tensorflow-object-detection-to-do-pixel-wise-classification-702bf2605182
25	Masked RCNN	https://www.analyticsvidhya.com/blog/2019/07/computer-vision-implementing-mask-r-cnn-image-segmentation/
26	Masked RCNN	https://towardsdatascience.com/computer-vision-instance-segmentation-with-mask-r-cnn-7983502fcad1
27	Interesting application of Mask RCNN	https://arxiv.org/ftp/arxiv/papers/1907/1907.08884.pdf
28	Indian Driving Dataset	http://idd.insaan.iiit.ac.in/
29	Mask RCNN and Deeplab performance	https://cloud.google.com/blog/products/ai-machine-learning/whats-in-an-image-fast-accurate-image-segmentation-with-cloud-tpus
30	Deeplearning Review	https://www.researchgate.net/publication/316450908_A_Review_on_Deep_Learning_Techniques_Applied_to_Semantic_Segmentation
31	Latest Paper	https://towardsdatascience.com/eagleview-super-high-resolution-image-segmentation-with-deeplabv3-mask-rcnn-using-keras-arcgis-9be08caac42c
33	Extracting people and putting in different background	https://arxiv.org/ftp/arxiv/papers/1907/1907.08884.pdf

Table 1: Architecture Comparisons

Analysis:

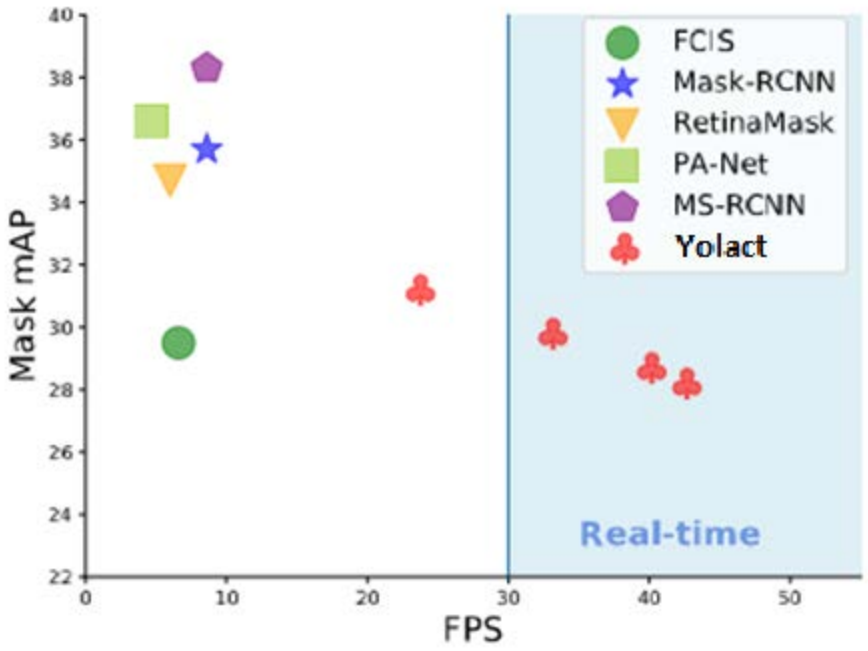


Fig 8: Comparison of Mask-RCNN with Yolact

As shown in Fig6, Mask RCNN performs better for less frames per second or image segmentation. Considering our problem we are trying to solve we selected MaskRCNN as our method as it has better performance on the instance segmentation, can detect multiple instances in the same image and provides effective masking on the Region Of Interest (ROI) with probability score.

Faster R-CNN is a popular framework for object detection, and Mask R-CNN extends it with instance segmentation, among other things. Instance segmentation is the task of identifying object outlines at the pixel level. Compared to similar computer vision tasks, it's one of the hardest possible vision tasks.

4.1 Mask RCNN

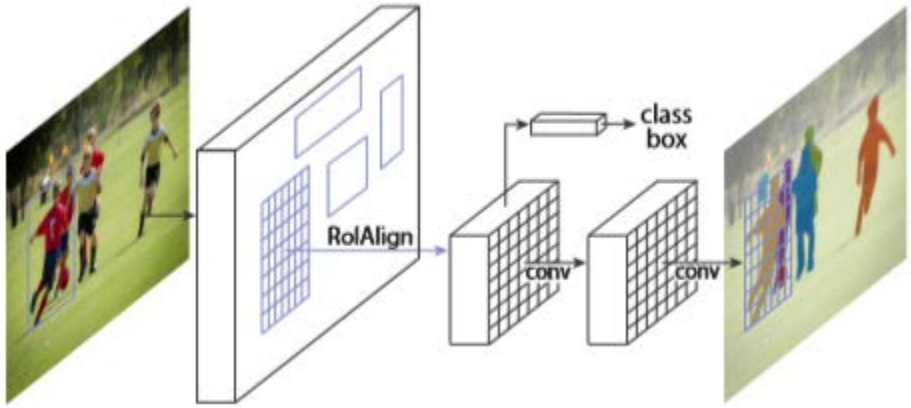


Fig 9: Mask-RCNN framework for instance segmentation

Mask R-CNN adopts the two-stage procedure, with region proposal Network (RPN) as the first stage where it proposes candidate object bounding boxes. In the second stage, in parallel to predicting the class and box offset, Mask R-CNN also outputs a binary mask for each RoI. Our approach follows the spirit of Fast R-CNN that applies bounding-box classification and regression in parallel (which turned out to largely simplify the multi-stage pipeline of original R-CNN). Formally, multi-task loss on each sampled RoI as L = Lcls + Lbox + Lmask is defined during training.

Where – Lcls is classification loss

Lbox is bounding-box loss

The mask branch has a Km2dimensional output for each RoI, which encodes K binary masks of resolution m x m, one for each of the K classes. To this we apply a per-pixel sigmoid, and define Lmask as the average binary cross-entropy loss.

Faster R-CNN has two outputs

- For each candidate object, a class label and a bounding-box offset

Mask R-CNN has three outputs

- For each candidate object, a class label and a bounding-box offset
- Third output is the object mask

4.2 Mask RCNN Architecture

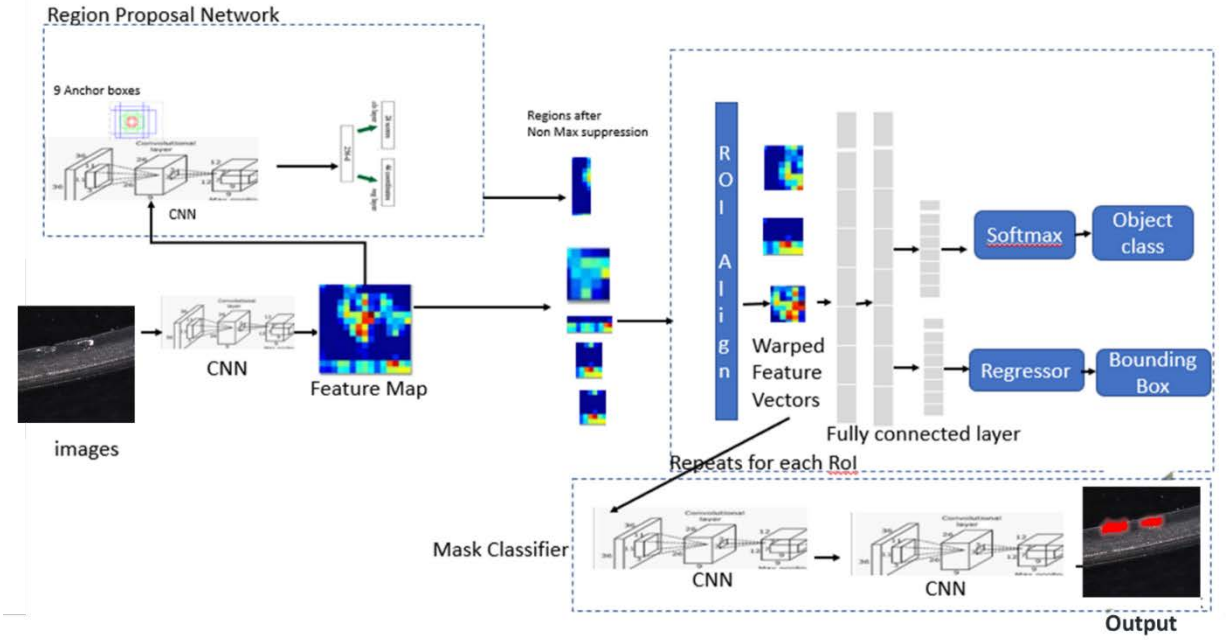


Fig 10: Architecture of Mask RCNN

- Image is run through the CNN to generate the feature maps.
- Region Proposal Network (RPN) uses a CNN to generate the multiple Region of Interest(RoI) using a lightweight binary classifier. It does this using 9 anchors boxes over the image. The classifier returns object/no-object scores. Non Max suppression is applied to Anchors with high objectness score
- The RoI Align network outputs multiple bounding boxes rather than a single definite one and warp them into a fixed dimension.

- Warped features are then fed into fully connected layers to make classification using softmax and boundary box prediction is further refined using the regression model

- Warped features are also fed into Mask classifier, which consists of two CNN's to output a binary mask for each RoI. Mask Classifier allows the network to generate masks for every class without competition among classes.

4.3 Anchor Boxes

Mask R-CNN uses anchor boxes to detect multiple objects, objects of different scales, and overlapping objects in an image. This improves the speed and efficiency for object detection. Anchor boxes are a set of predefined bounding boxes of a certain height and width. These boxes are defined to capture the scale and aspect ratio of specific object classes you want to detect.

To predict multiple objects or multiple instances of objects in an image, Mask R-CNN makes thousands of predictions. Final object detection is done by removing anchor boxes that belong to the background class and the remaining ones are filtered by their confidence score. We find the anchor boxes with IoU greater than 0.5. Anchor boxes with the greatest confidence score are selected using Non-Max suppression explained below

4.4 Intersection Over Union

IoU computes intersection over the union of the two bounding boxes, the bounding box for the ground truth and the bounding box for the predicted box by algorithm. When IoU is 1 that would imply that predicted and the ground-truth bounding boxes overlap perfectly. To detect an Object once in an image, Non-Max suppression considers all bounding boxes with IoU >0.5.

4.5 Non-Max Suppression

Non-Max Suppression will remove all bounding boxes where IoU is less than or equal to 0.5. It will pick the bounding box with the highest value for IoU and suppress the other bounding boxes for identifying the same object.

5. CODE STRUCTURE

5.1 Code Architecture

Following is the architecture for this project:

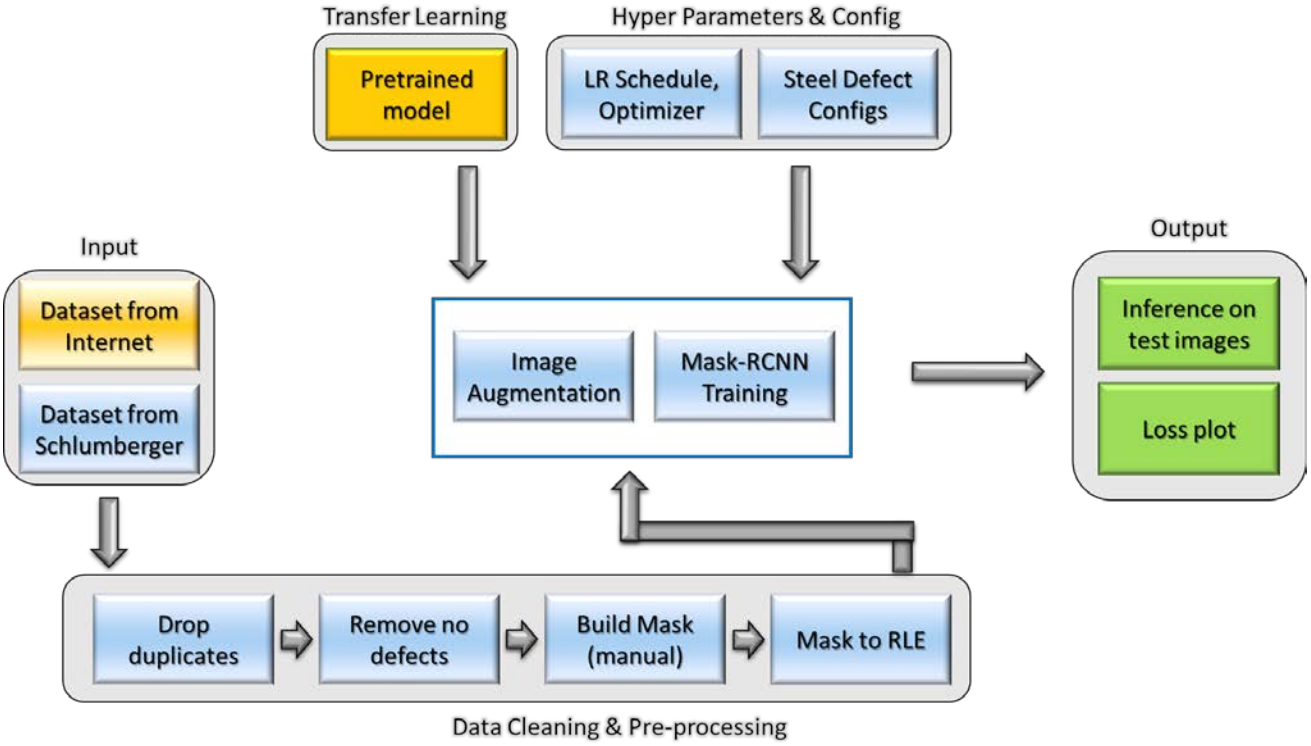


Fig 11: Code Architecture

The technical architecture above shows that we are leveraging on transfer learning by using an existing pre-trained Mask-RCNN model on COCO dataset. The hyperparameters are adjusted for each iteration described in Section 7, introducing accuracy increase and fine tuning of the model. Since our source dataset doesn't contain much images, we had to add some from the internet for similar steel panel defects. The data cleaning and preparation pipeline is also shown in the architecture. All the blue shaded boxes are done in-house, orange colored are sourced from internet and green shaded is the output which can be used by the sponsor company.

5.2 Dynamic Learning Rate

Choosing the learning rate is challenging as a value too small may result in a long training process that could get stuck, whereas a value too large may result in learning a sub-optimal set of weights too fast or an unstable training process.

In our project, our models were not improving beyond a certain point when we introduced dynamic learning rate to support the ADAM optimizer. Learning rate schedules seek to adjust the learning rate during training by reducing it according to a pre-defined schedule.

```
def lrSchedule(epoch):
    lr  = 1e-4

    if epoch > 100:
        lr  *= 0.5e-3
    elif epoch > 75:
        lr  *= 1e-3
    elif epoch > 50:
        lr  *= 1e-2
    elif epoch > 25:
        lr  *= 1e-1

    print('Learning rate: ', lr)
    return lr
```

Here, we have defined learning rate to be different for different number of epochs.

This lrSchedule function is called by ADAM optimizer for updated learning rates to improve overall training loss.

An lrSchdule function as shown here is passed as an argument for callbacks to return the updated learning rates.

5.3 Optimizer

The Mask-RCNN comes with default SGD as the optimizer. However, we modified the MaskRCNN master code to support ADAM optimizer. This was to support the dynamic learning rates to find individual learning rates for each parameter. Adam can be looked at as a combination of RMSprop and Stochastic Gradient Descent with momentum. It uses the squared gradients to scale the learning rate like RMSprop and it takes advantage of momentum by using moving average of the gradient instead of gradient itself like SGD with momentum.

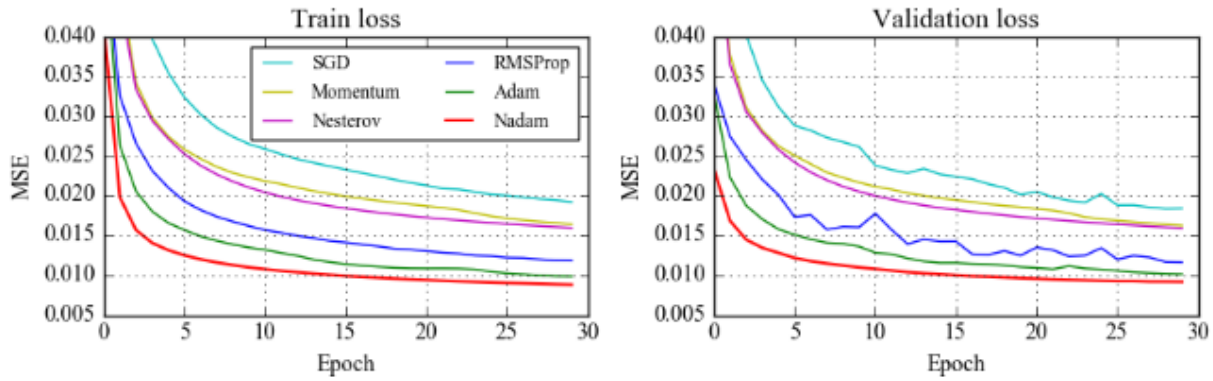


Fig 12: Loss difference between Adam & other optimizers

5.4 Image Augmentation

Deep learning neural networks need large amount of training data to achieve good performance. To build a powerful image classifier using very little training data, image augmentation is usually required to boost the performance of deep networks. Image augmentation artificially creates training images through different ways of processing or combination of multiple processing, such as random rotation, shifts, shear and flips, etc.

```
imgaug_set = seq = iaa.Sequential([iaa.Fliplr(0.3), # horizontal flips
                                   sometimes(iaa.Crop(percent=(0, 0.1))), # random crops
                                   sometimes(iaa.Affine(
                                       scale={"x": (0.9, 1.1), "y": (0.9, 1.1)},
                                       translate_percent={"x": (-0.1, 0.1), "y": (-0.1, 0.1)},
                                       rotate=(-30, 30)
                                   )),
                                   iaa.Grayscale(alpha=(0.0, 0.7))],
                                  random_order=True)
```

We used imgaug library for this project. We had to experiment with different values on image augmentation to improve the accuracy on training dataset. Here we are using a 30% flip on images, and a random crop of 10%. Affine transformations such as scaling between 90% to 110%, translation and rotation (-30 to +30 degrees) are also done randomly. One of the additional APIs from this library we used here is to convert to greyscale in the range 0 to 70%, as some images from Schlumberger were colored.

5.5 Weight Decay Regularization

Weight regularization provides an approach to reduce the overfitting of a deep learning neural network model on the training data and improve the performance of the model on new data, such as the holdout test set. We have used a weight decay value of 0.0001 for our model.

5.6 Code walkthrough

Github Link: https://github.com/rmunjal123/VisionSystems_CA1

To start, setup the environment first (refer to setup.txt file)

Windows, linux:

1. conda create -n y2-s1-ca1 python=3.6 numpy=1.18.1 opencv=3.4.2 matplotlib=3.1.3 tensorflow=1.14.0 tensorflow-gpu=1.14.0 keras=2.3.1 keras-gpu=2.3.1 cudatoolkit=10.0 cudnn=7.6.5 scipy=1.4.1 scikit-learn=0.22.1 pillow=6.1.0 ipython=7.12.0 spyder=4.0.1 imgaug cython pathlib yaml pandas pydot graphviz seaborn

(Assume Nvidia Cuda 10.2 is installed according to your Nvidia driver version (see https://docs.nvidia.com/deploy/cuda-compatibility/index.html#binary-compatibility), and cudnn 7.6.5 installed, get cudnn from https://developer.nvidia.com/cuda-downloads)

2. conda activate y2-s1-ca1

For the code structure, first, import the necessary libraries:

```
import os
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import tensorflow as tf
from tensorflow.python.client import device_lib
from sklearn.model_selection import train_test_split

from mrcnn.utils import Dataset
from mrcnn.config import Config
from mrcnn.model import MaskRCNN

import imgaug.augmenters as iaa
```

Next, we define Constants, Functions, and Classes:

```
# Location of input images
input_img_folder = "dataset/all_images"

# Distint class values (numeric)
class_names = ['Blowhole', 'Cavity', 'Crack']
```

Following are some of the variables we had to go through changing many times to derive our required accuracy. During each change, the training had to run for few hours before we could determine whether the configurations are good enough for our model.

```
# Give the configuration a recognizable name
NAME = "steel_detection"

# We use a GPU with 12GB memory, which can fit two images.
# Adjust down if you use a smaller GPU.
IMAGES_PER_GPU = 1

# Number of classes (including background)
NUM_CLASSES = 1 + 3  # background + steel defects

# Number of training steps per epoch - 100, 64
STEPS_PER_EPOCH = 200

# Skip detections with < 90% confidence
DETECTION_MIN_CONFIDENCE = 0.8 # base 0.7
```

```
# Learning rate and momentum
LEARNING_RATE = 0.001
LEARNING_MOMENTUM = 0.9

# Weight decay regularization
WEIGHT_DECAY = 0.0001

# Number of epochs to run - 100
EPOCH_SIZE = 15
```

The following functions take the dataframe containing

1. file names of our images (which we will append to the directory to find our images)
2. a list of rle for each image (which will be fed to our build_mask() function we also used in the eda section)

And adds images to the dataset with the utils.Dataset's add_image() method

```
def load_dataset(self):

    # add our four classes
    for i in range(1,4): #range(1,5)
        self.add_class(source='', class_id=i, class_name=f'defect_{i}')

    # add the image to our utils.Dataset class
    for index, row in self.dataframe.iterrows():
        file_name = row.ImageId
        file_path = f'{input_img_folder}/{file_name}'

        assert os.path.isfile(file_path), 'File doesn\'t exist.'
        self.add_image(source='',
                       image_id=file_name,
                       path=file_path)
```

```python
def rle_to_mask(lre, shape=(512,512)):
    '''
    params:  rle   - run-length encoding string (pairs of start & length of encoding)
             shape - (width,height) of numpy array to return

    returns: numpy array with dimensions of shape parameter
    '''
    # the incoming string is space-delimited
    runs = np.asarray([int(run) for run in lre.split(' ')])

    # we do the same operation with the even and uneven elements, but this time with a
    runs[1::2] += runs[0::2]
    # pixel numbers start at 1, indexes start at 0
    runs -= 1

    # extract the starting and ending indeces at even and uneven intervals, respective
    run_starts, run_ends = runs[0::2], runs[1::2]

    #print("run:",run_starts, run_ends)

    # build the mask
    h, w = shape
    mask = np.zeros(h*w, dtype=np.uint8)
    for start, end in zip(run_starts, run_ends):
        mask[start:end] = 1

    # transform the numpy array from flat to the original image shape
    return mask.reshape(shape)
```

The following function takes a pair of lists of encodings and labels, and turns them into a 3d numpy array of shape (512, 512, 3)

```python
def build_mask(encodings, labels):

    # initialise an empty numpy array
    mask = np.zeros((512,512,3), dtype=np.uint8)

    # building the masks
    for rle, label in zip(encodings, labels):

        # classes are [1, 2, 3], corresponding indeces are [0, 1, 2]
        index = label - 1

        # fit the mask into the correct layer
        # note we need to transpose the matrix to account for
        # numpy and openCV handling width and height in reverse order
        mask[:,:,index] = rle_to_mask(rle).T

    return mask
```

Next, as found in https://github.com/matterport/Mask_RCNN/blob/master/samples/coco/coco.py , load instance masks for the given image. This function converts the different mask format to one format in the form of a bitmap [height, width, instances]. It returns:

- masks: A bool array of shape [height, width, instance count] with one mask per instance
- class_ids: a 1D array of class IDs of the instance masks

```
    def load_mask(self, image_id):

        # find the image in the dataframe
        row = self.dataframe.iloc[image_id]

        # extract function arguments
        rle = row['EncodedPixels']
        labels = row['ClassId']

        # create our numpy array mask
        mask = build_mask(encodings=rle, labels=labels)

        return mask.astype(np.bool), np.array([1, 2, 3], dtype=np.int32)
```

One of the important methods is to leverage on transfer learning by using a pre-trained model, who trained a Mask-RCNN for localizing and classifying surface defects on steel sheets using dataset from Severstal, a Russian company mainly operating in the steel and mining industry.

```
model.load_weights('pretrained_models/severstal/mask_rcnn_severstal_0100.h5',
                   by_name=True,
                   exclude=['mrcnn_bbox_fc',
                            'mrcnn_class_logits',
                            'mrcnn_mask',
                            'mrcnn_bbox'])
```

Next perform image augmentation to improve accuracy with the limited dataset that we have:

```
imgaug_set = seq = iaa.Sequential([iaa.Fliplr(0.5), # horizontal flips
                                   sometimes(iaa.Crop(percent=(0, 0.1))), # random crops
                                   sometimes(iaa.Affine(
                                        scale={"x": (0.8, 1.2), "y": (0.8, 1.2)},
                                        translate_percent={"x": (-0.1, 0.1), "y": (-0.1, 0.1)},
                                        rotate=(-30, 30)
                                    )),
                                   iaa.Grayscale(alpha=(0.0, 0.7))],
                                  random_order=True)
```

And finally, the training starts:

```
model.train(dataset_train,
            dataset_validate,
            epochs=epoch_size,
            layers='heads',
            augmentation=imgaug_set,
            custom_callbacks=callbacks_list,
            learning_rate=steel_defect_config.LEARNING_RATE)

train_hist = model.keras_model.history
```

The outputs for various iterations and how we fine-tuned the parameters to achieve higher accuracy is described in the next sections.

6. DESIGN OF THE APIs

Let's take a look at the various APIs used in this project.

Sl No	API	Function
Training & Testing APIs		
1	*build_mask(encodings, labels)*	takes a pair of lists of encodings and labels, and turns them into a 3d numpy array of shape (512, 512, 3)
2	*rle_to_mask(lre, shape=(512,512))*	Takes input params: rle - run-length encoding string (pairs of start & length of encoding) and shape - (width,height) of numpy array to return. Returns: numpy array with dimensions of shape parameter.
3	*load_dataset(self)*	Takes: pandas df containing (1) file names of our images and (2) a list of rle for each image. It adds images to the dataset with the utils.Dataset's add_image() method.
4	*load_mask(self, image_id)*	This function converts the different mask format to one format in the form of a bitmap [height, width, instances]
5	*rle_encoding(x)*	x: numpy array of shape (height, width), 1 - mask, 0 - background. Returns run length as list
6	*MaskRCNN()*	Class for creating MaskRCNN model. We used the functions load_weights(), train() & detect() from the model.
Preprocessing (Finding Contours for labelling)		
7	*mask_to_contours(image, mask_layer, color)*	converts a mask to contours using OpenCV and draws it on the image
8	*write_mask_to_csv(path_arr)*	Creates a csv file with filename, image class and the rle value.

Table 2: API design

7. STEPS TO IMPROVE PERFORMANCE

Since the base model is taken from Mask RCNN defect detection for Severstal, it required fine tuning of many parameters and we had to train multiple times to achieve the final model.

7.1 Overfitting Model

In this iteration, we have base model with raw data and training done. Following is our Loss plot for this iteration. We can see that it doesn't converge well, therefore we proceed to the 2nd iteration.

Fig 13: Loss plot for iteration #1

7.2 Increase Batch Size

In this iteration, we have increased the batch size to 2 (2 images per step) and the steps per epoch was increased from 30 to 100 and number of epochs is 150, with no image augmentation and optimizer.

Once again, it's not converging well though it shows a very small improvement over the previous iteration.

Fig 14: Loss plot for iteration #2

7.3 Introduce Augmentation

Fig 15: Loss plot for iteration #3

Here, we included imgaug library for image augmentation. Following is our Loss plot which shows that our model stopped overfitting. However, the accuracy is still low as the augmentation values we applied were high. Here we used 50% horizontal flips, scaling between 80 to 120% and rotation between -25 to +25 degrees. We didn't utilize the greyscale conversion of imgaug library in this iteration though.

7.4 Optimize Augmentation

In this iteration, we have applied "sometimes" parameter to imgaug to reduce augmentation. The value of sometimes is set at 50% so that augmentation is done only in 50% of the cases. Grayscale conversion was also included as some images from Schlumberger were colored.

Following is our Loss plot, which shows improvement from the previous ones. However, validation loss is lower than training loss. This is not acceptable as normally validation loss should be identical or a bit higher than training loss.

Fig 16: Loss plot for iteration #4

7.5 Introduce ADAM

Next we introduced ADAM as the optimizer instead of the default SGD for Mask-RCNN.

Mask-RCNN code from Facebook doesn't support ADAM out of the box, hence here we are using a unique approach by editing the original APIs and the model to support ADAM.

```
    def compile(self, learning_rate, momentum, optimizer):
        """Gets the model ready for training. Adds losses, regularization, and
        metrics. Then calls the Keras compile() function.
        """

        # Optimizer object
        if optimizer == 'Adam':
            keras.optimizers.Adam(lr=learning_rate, clipnorm=self.config.GRADIENT_CLIP_NORM)
        else:
            optimizer = keras.optimizers.SGD(
                lr=learning_rate, momentum=momentum,
                clipnorm=self.config.GRADIENT_CLIP_NORM)

        print("Optimizer: ", optimizer)
```

```
model.train(dataset_train,
            dataset_validate,
            epochs=epoch_size,
            layers='heads',
            augmentation=imgaug_set,
            custom_callbacks=callbacks_list,
            learning_rate=steel_defect_config.LEARNING_RATE,
            optimizer='Adam')
```

Following is our Loss plot with the above changes. The validation loss was fluctuating high because our learning rate (0.001) for ADAM is too high. Since the values were spiking, we stopped the training at lower epochs to save time.

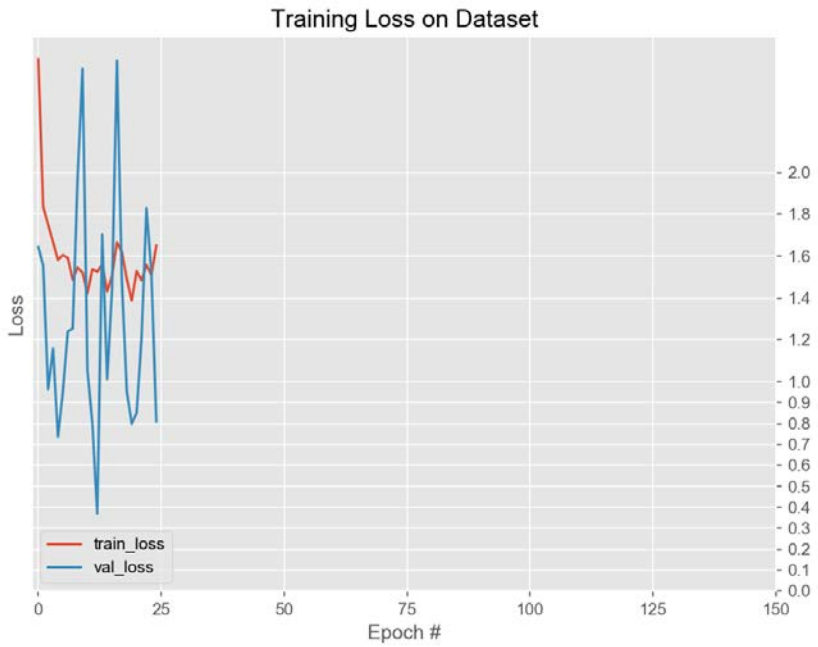


Fig 17: Loss plot for iteration #5

7.6 Introduced Dynamic Learning Rate With Changes For ADAM

Following is our Loss plot with the introduction of dynamic learning rate and code changes for ADAM. The validation loss was fluctuating high because our learning rate (0.001) for ADAM is too high. More details on dynamic learning rate has been discussed in Section 4.9.

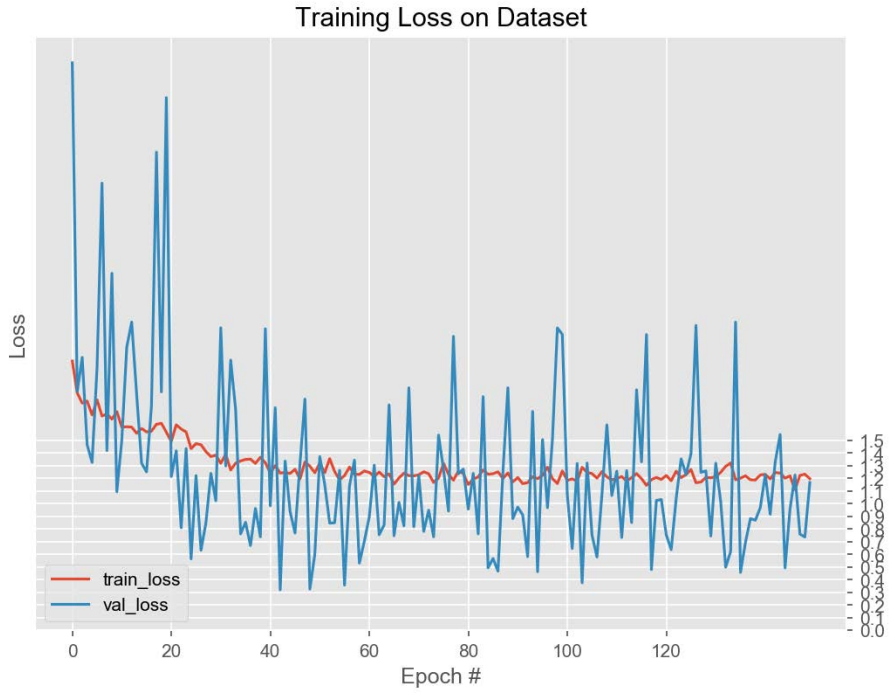


Fig 18: Loss plot for iteration #6

8. RESULTS & INFERENCE

Following are the results of validation on few images, whereby our model was able to accurately detect, segment and classify the images.

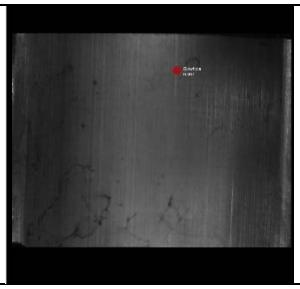
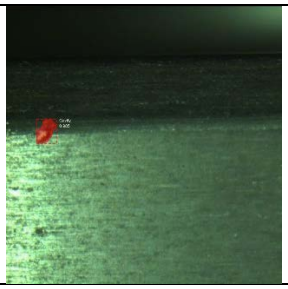
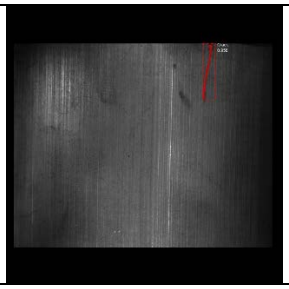
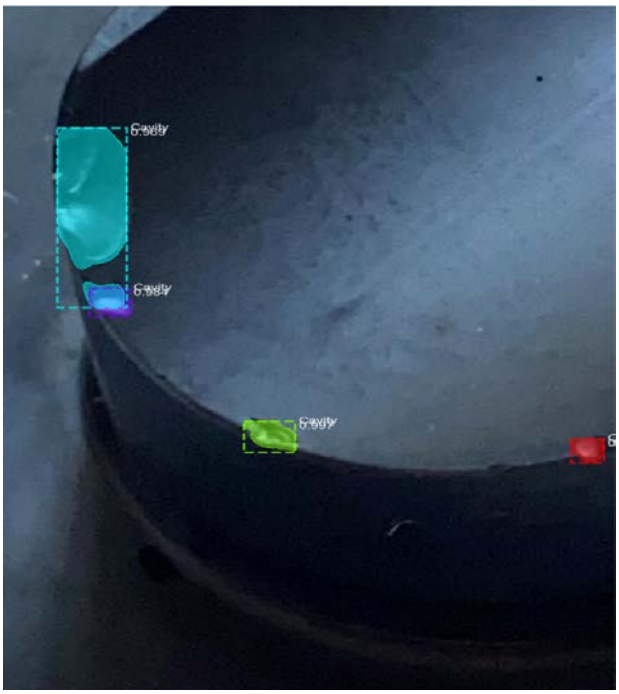
		
Blowhole	Cavity	Crack

Fig 19: Validation on train dataset

We tried the model on images which was never seen by the model previously, and we can see that its accurately detecting the defects as per the customer (Schlumberger) requirements.

Test Image	Result
	

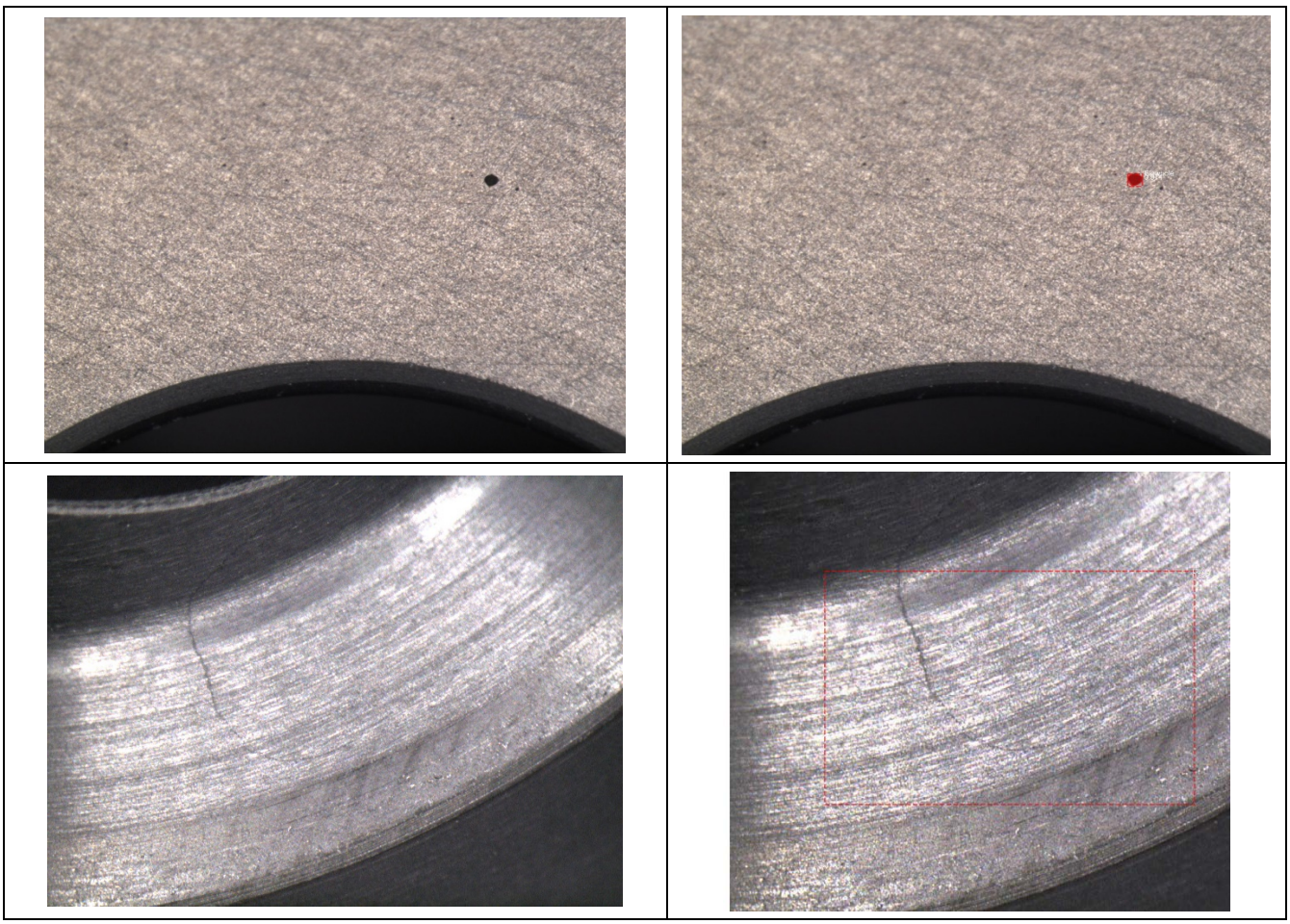


Fig 20: Testing on unseen images

Inference:

As shown in Fig 19, validation passed correctly for all the 3 classes.

As shown in Fig 20, the model is able to detect multiple instances of the same defect which is what we wanted to achieve.

This model is modular and scalable in a way such that by adjusting the confidence level, we can vary the detection and classification of defects. The current confidence level used for segmentation in the above result is 96%.

In the first two cases, the defect is correctly identified. In the 3rd case of crack as shown above, we can see that the model is detecting a crack as a cavity.

9. FINDINGS & CONCLUSION

Mask-RCNN is a proven solution for instance segmentation on images, and we have been able to use it for defect detection and segmentation on metal surfaces for our company's production line.

Few of the findings from this project are:

1. For training dataset, it's important that all images must be of the same size (WxH) which helps to improve the performance of the model.
2. RLE is an effective method to encode the masks into an array of numbers in CSV format to speed up the training process.
3. Transfer learning has become particularly useful since we have less images for training. Based on the number of images in the dataset, we trained not only the lower layers but also few middle layers which helped to improve the performance of the model.
4. Augmentation is quite effective method for small dataset training by helping to prevent overfitting.
5. Increasing the step size per epoch helps to improve the accuracy.
6. ADAM is a better optimizer than SGD because it combines the advantages of two SGD extensions — Root Mean Square Propagation (RMSProp) and Adaptive Gradient Algorithm (AdaGrad) — and computes individual adaptive learning rates for different parameters.
7. For the incorrect detection of crack on few test images, the accuracy can be improved if more images can be obtained for training from Schlumberger.

This was very interesting and quite a learning experience for us to do our 1st image segmentation project. We got introduced to various important concepts of Artificial Intelligence through this Project. Some of them to highlight here are:

1. Transfer Learning
2. Image Semantic Segmentation
3. Instance Segmentation using Deep Learning
4. Real problem solving using Deep Learning

Challenges we faced and overcome during the project like Dataset Collection, Cleaning Images, Mask Creation are the manual tasks which needs equal importance as the machine learning algorithms.

We would be providing this model as a prototype for testing defects in Schlumberger which can be further optimized with more training data and implemented as a solution for identifying defects using phone camera.

Please visit Github Link: https://github.com/rmunjal123/VisionSystems_CA1 for code and model files

10. REFERENCES

[1] YOLACT Real-time Instance Segmentation

[2] Mask R-CNN

[3] Instance Segmentation with Mask R-CNN

[4] Implementing Mask R-CNN for Image Segmentation

[5] Splash of Color: Instance Segmentation with Mask R-CNN and TensorFlow

[6] Building a Mask R-CNN Model for Detecting Car Damage

[7] https://www.kaggle.com/c/severstal-steel-defect-detection