Detection of M-A Islands in Bainite Micrographs Using Convolutional Neural Networks

(Group 3) Karthik Shankar[a], Karthi Hari Krishnan[b], Hiran N[c], Priyansh Godha[d]

Abstract

Microstructure analysis of welds and other steels has always been time-consuming and labor-intensive. The weld must have good physical properties and specific composition, but these tend to be inconsistent. There is a demand in the industry to automate it as much as possible, speeding up this process many times. We tried to address this problem via a Machine learning based approach. For this purpose, we attempted to solve a very special case of segmenting Martensite-Austenite (MA) islands on bainitic steels using a Mask Region-based convolutional neural network (Mask-R CNN).

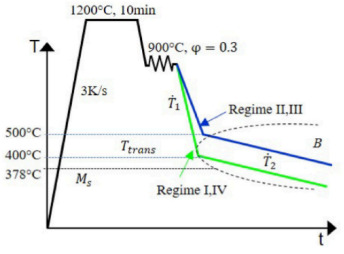
1. Introduction

Bainitie is a non-equilibrium phase of Iron-Carbon alloy system formed by continuous cooling of Austenite at appropriate cooling rates. Excellent combination of mechanical properties makes bainitic steels very attractive for commercial application. It's very essential to study the microstructure before employing a material for any application. Microstructures are material structures seen at the micro level and they tell us a lot about its properties by revealing defects, impurities, grains, and grain boundaries. For instance, volume percentage of MA islands is directly correlated to impact toughness of the material.

A conventional metallographic approach to analyze the microstructure requires several steps of surface preparation - embedding, grinding, polishing, and etching. The etched sample is put under a Scanning electron microscope (SEM), that uses electrons instead of light to form an image. An expert takes micrographs from the thoroughly prepared surface. Finally, the micrograph is analyzed and described by the materials scientist. However this step is very time consuming and has poor reproducible results and there is a need for a more robust and reliable method to do the same. We tried to address this problem via a Machine learning based approach. For this purpose, we attempted to solve a very special case of segmenting Martensite-Austenite (MA) islands on bainitic steels using a Mask Region-based convolutional neural network (Mask R-CNN).

2. Methods

Before starting with the ML model, it is necessary to understand the underlying materials science to it. In bainitic steels with different processing routes ie., cooling regimes we get different morphologies of MA islands.



Cooling strategy	$\dot{T}_1$ [K/s]	$\dot{T}_2$ [K/s]	T_{trans} [°C]
Regime I	5	0.3	400
Regime II	2	1.0	500
Regime III	2	0.3	500
Regime IV	5	0.1	400

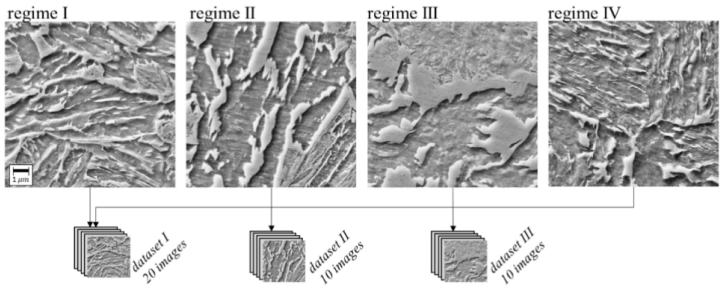


Fig 2.1 Different cooling regimes and the resultant morphologies [1]

Classifying different morphological features would require different ML treatment. So we categorized the microstructure into following categories.
Class 1- Elongated islands (Regime I and IV)
Class 3- Blocky islands (Regime III)
Class 2 -Mixed of bothe (Regime II)
Now that with that domain knowledge acquired the problem can be split into smaller sub-problems:

1. Detecting instances (MA- islands) and forming a bounding box. (was done using an unsupervised learning technique, K-Means)
2. Classification of the instance into one of the two morphologies blocky and elongated. (was done using a CNN framework)
3. Training separate Mask R-CNN models for each class.

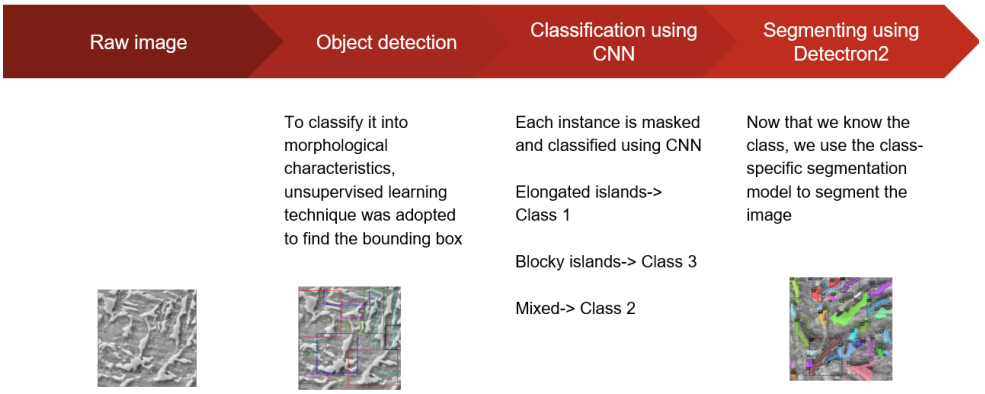


Fig 2.2 The flow of process

3. Results and discussion

3.1 K-means: for bounding boxes

The MA islands in the microstructure image can be clearly distinguished from the other phase by its shade. As a result, we implement the K-Means algorithm for segmentation of the image. This assigns one of K different tags to each pixel. Once this has been done, we use depth-first search on each of the islands to find the bounding boxes, which we can feed into the CNN to classify.

What K-Means does to find these tags is quite simple and can be broken down into a few steps:

1. Pick K random points on the image (marker points)
2. For each pixel in the image, set its value to the nearest marker point

3. For each of the groups, set the marker points to be at the mean of all the points
4. Repeat 2-3 until the marker points do not shift much

After following these steps, it segments the image. However, the 1st step has randomness and can result in incorrect clustering. To eliminate this, we repeat the whole process many times with different marker points in the initial step, until we reach the best looking clustering.

The following image depicts our dataset. We have the microstructure image, where we can clearly see the lighter MA islands. These need to be marked. The dataset has a set of points for each MA island in the image describing its boundary.

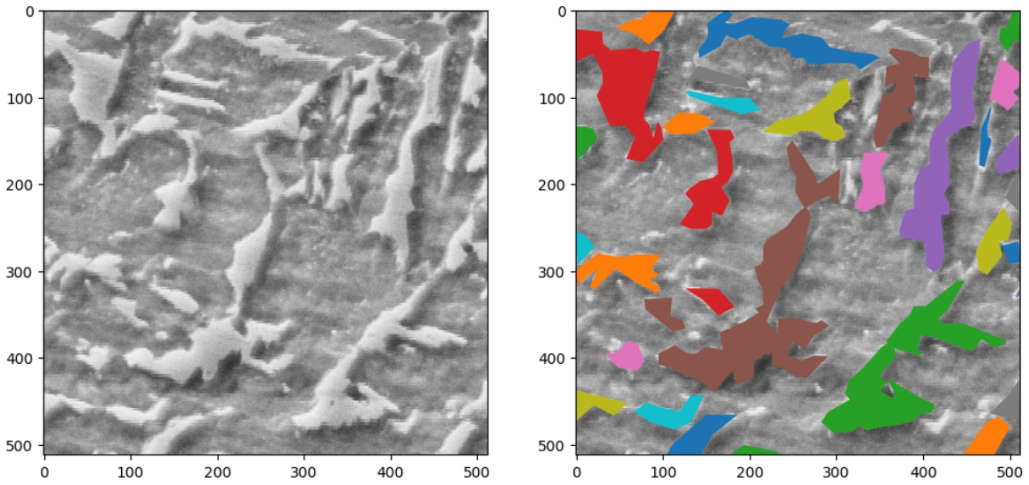


Fig 3.1.1 Dataset labels

Fortunately, this is what K-Means is good at. It easily spots the difference between a lighter MA island and the darker phase. Implementing it gives the following:

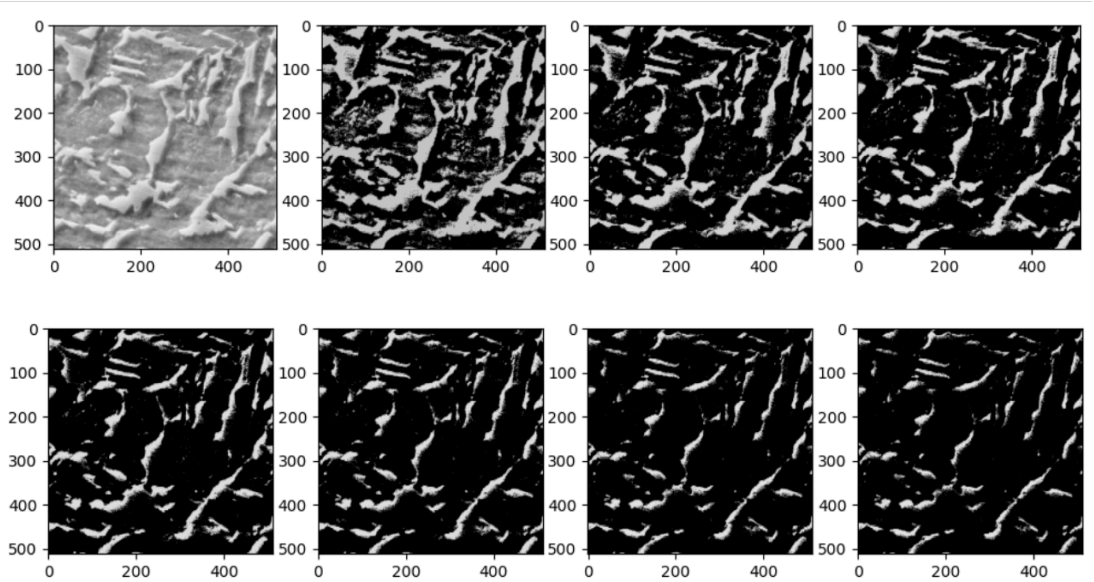


Fig 3.1.2 Performance of K-means

These images in Fig 3.A.2 show how K-Means segments the image for different values of K, ranging from 2 to 8. We can see that lower K values give much coarser MA islands while the higher K values result in only the lightest part of the islands being detected.

How do we ensure a good fit? We check which value of K fits the segmentation given in the dataset best and use it. To further improve accuracy, we can pick the top n elements (n will have to be chosen appropriately) for a certain K.

Now that we have our islands separated, we need to make bounding boxes to feed into the CNN, which will classify the image. We use the depth-first search algorithm, commonly referred to as DFS. DFS systematically explores a graph completely and efficiently. Here's a high level rundown of DFS:

1. Create an array to keep track of visited nodes
2. Choose a starting node
3. Create an empty stack and push the starting node onto the stack
4. Mark the starting node as visited
5. While the stack is not empty, do the following:
    5.1. Pop a node from the stack
    5.2. Process or perform any necessary operations on the popped node
    5.3. Get all the adjacent neighbors of the popped node
    5.4. For each adjacent neighbor, if it has not been visited, do the following:
        5.4.1. Mark the neighbor as visited
        5.4.2. Push the neighbor onto the stack
6. Repeat step 5 until stack is empty

Applying this, we can systematically and efficiently explore all the islands to find their bounding boxes. We ignore very small islands, and draw the bounding boxes on an image:

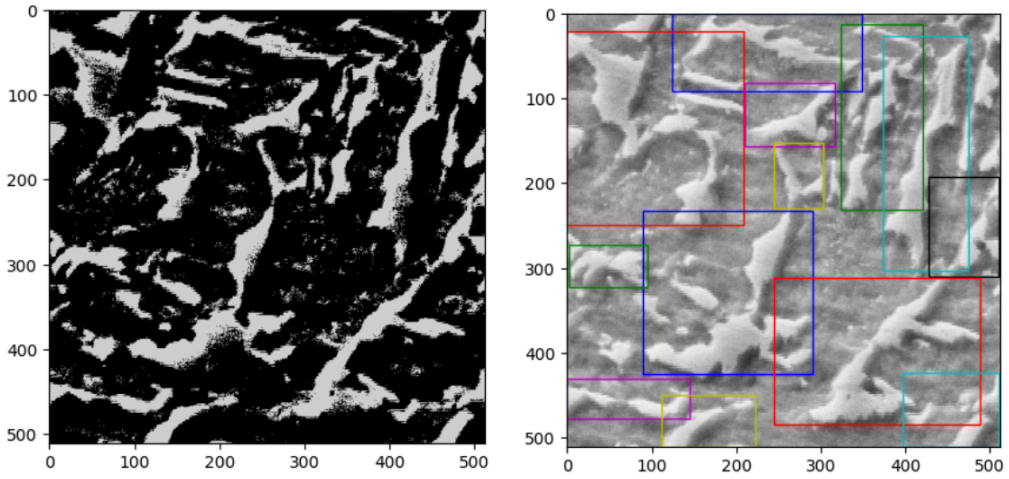


Fig 3.1.3 Bounding box being generated

We implemented this in python, with the help of the numpy and scikit-learn libraries. The upside to the approach we used is that it is very modular and can easily be modified for various use cases. It is also much faster than an RPN (Region Proposal Network) that is often used for this purpose.

However, it does have room for improvement. Some of the main points:

1. We need a better criterion to select the islands. As of now, we just ignore the small ones, whose areas are below a certain threshold. A better approach would be to eliminate the islands based on a percentage area measure.
2. DFS is inherently sequential, so cannot be sped up through parallel processing. We can use the breadth-first search (BFS) algorithm instead, which does have an implementation with parallel processing and will also achieve the same thing.

3. Implementing this in a faster language like C++ will significantly speed up the program.

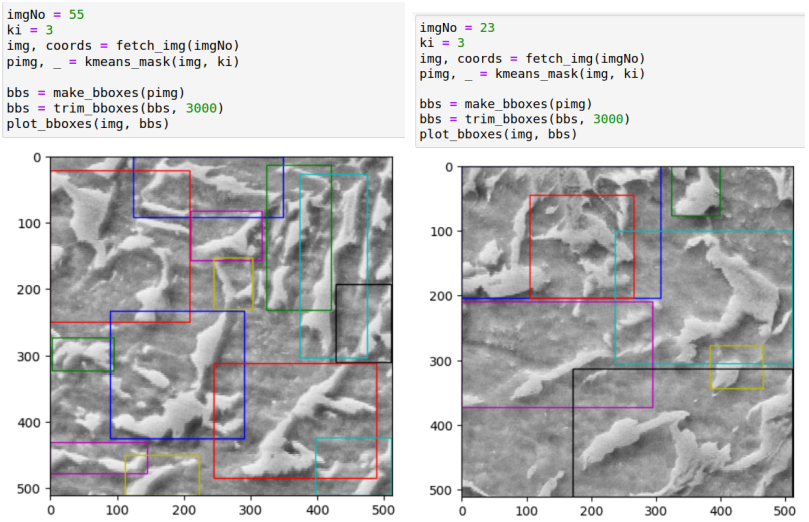


Fig 3.1.3 Execution on different images on the dataset

3.2. CNN for classification

In this section, we are going to see how we made our CNN model. These are the steps we followed:

1. Getting Data.
2. Formatting data to create a new usable dataset.
3. Making a baseline model, to use for comparison in future.
4. Developing better models.

3.2 a. Data extraction

We retrieved [3] two files from the mentioned paper, i) Folder containing all images, ii) JSON file with coordinates of martensite islands.

3.2 b. Creating usable dataset

Using these two files we separated the martensitic islands by wrapping them by rectangles and then masking each of them on a 512 x 512 blank image.

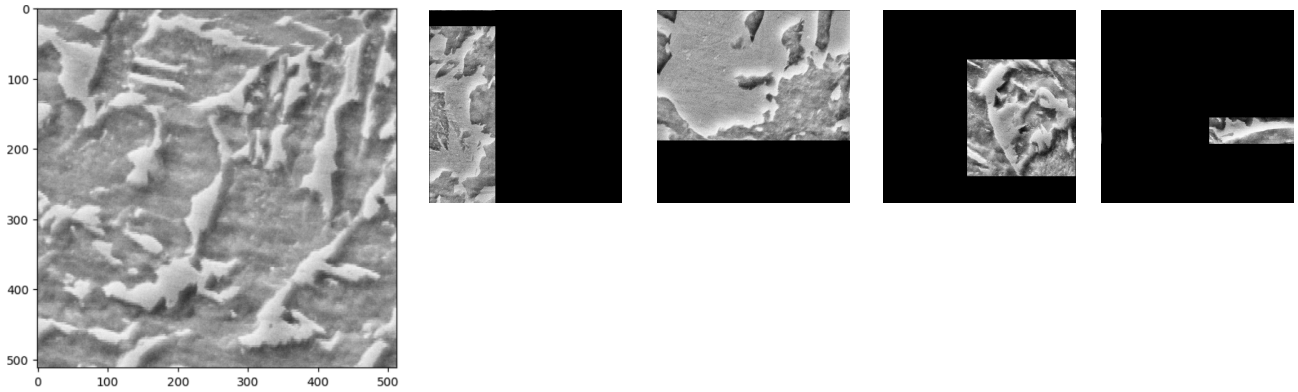


Fig 3.2 .1 Each instance being masked and being split into a separate images

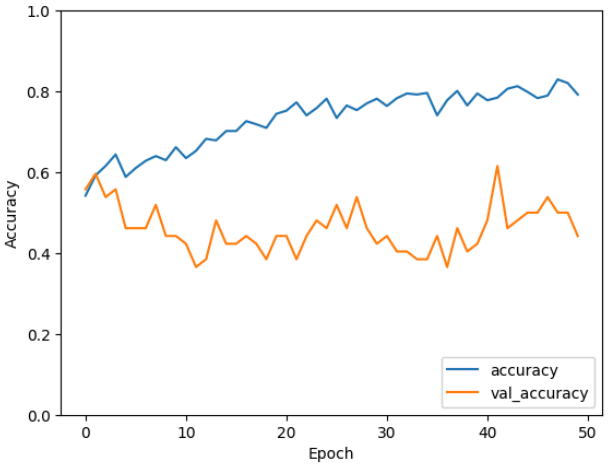
We did this process for both type I and type III martensite islands and then saved them in 2 different folders. Following figure shows the initial data image and final dataset images. After creating these folders, we further bifurcated the images into test and train. Now we have our final dataset containing 2 folders for each martensite type.

3.2 c. Baseline model

For our baseline model we created a CNN model whose structure is as mentioned. For this model, the results obtained could not have been worse. Even after 50 epochs, the model's performance was not improving.

Layer (type)	Output Shape	Param #
conv2d_9 (Conv2D)	(None, 510, 510, 32)	320
max_pooling2d_9 (MaxPooling 2D)	(None, 255, 255, 32)	0
dropout_12 (Dropout)	(None, 255, 255, 32)	0
conv2d_10 (Conv2D)	(None, 253, 253, 64)	18496
max_pooling2d_10 (MaxPoolin g2D)	(None, 126, 126, 64)	0
dropout_13 (Dropout)	(None, 126, 126, 64)	0
conv2d_11 (Conv2D)	(None, 124, 124, 128)	73856
max_pooling2d_11 (MaxPoolin g2D)	(None, 62, 62, 128)	0
dropout_14 (Dropout)	(None, 62, 62, 128)	0
flatten_3 (Flatten)	(None, 492032)	0
dense_6 (Dense)	(None, 256)	125960448
dropout_15 (Dropout)	(None, 256)	0
dense_7 (Dense)	(None, 2)	514

```
Total params: 126,053,634
Trainable params: 126,053,634
Non-trainable params: 0
```



3.2 d. Improving the model

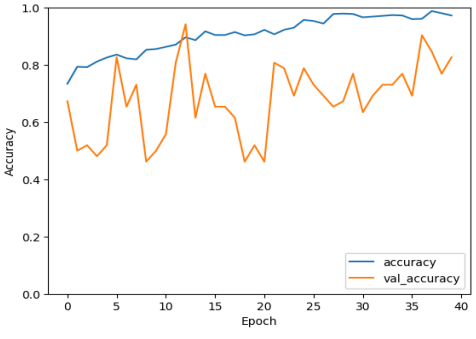
For optimisation of the model, we adopted several techniques mentioned below:

1. Kernel initialization (Normal)
2. Kernel regularization (L2)
3. Dropouts
4. Normalization
5. Optimization
6. Loss: Categorical Crossentropy
7. And several hours of computing time.

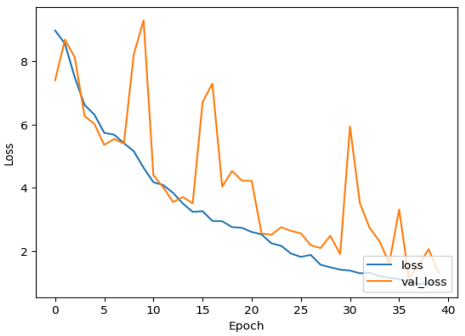
3.2 e. The Final-CNN model

After trying out around 30-40 models, we landed on the model with satisfying results. The structure for the model is given below. Using this model, we achieved an average of 80% accuracy with it reaching to even 95% in some cases (epochs). We are yet to confirm whether these >90% cases are results of the model learning the pattern and learning from the dataset, or just some over-fitting case.

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 510, 510, 32)	320
batch_normalization (Batch Normalization)	(None, 510, 510, 32)	128
max_pooling2d (MaxPooling2D)	(None, 255, 255, 32)	0
dropout (Dropout)	(None, 255, 255, 32)	0



```
 conv2d_1 (Conv2D)           (None, 253, 253, 64)      18496
 batch_normalization_1(Batch  (None, 253, 253, 64)     256
 Normalization)
 max_pooling2d_1 (MaxPooling  (None, 126, 126, 64)     0
 2D)
 dropout_1 (Dropout)         (None, 126, 126, 64)      0
 conv2d_2 (Conv2D)           (None, 124, 124, 128)     73856
 batch_normalization_2(Batch  (None, 124, 124, 128)    512
 Normalization)
 max_pooling2d_2 (MaxPooling  (None, 62, 62, 128)      0
 2D)
 dropout_2 (Dropout)         (None, 62, 62, 128)       0
 conv2d_3 (Conv2D)           (None, 60, 60, 256)       295168
 batch_normalization_3(Batch (None, 60, 60, 256)       1024
 Normalization)
 max_pooling2d_3 (MaxPooling  (None, 30, 30, 256)      0
 2D)
 dropout_3 (Dropout)         (None, 30, 30, 256)       0
 flatten (Flatten)           (None, 230400)            0
 dense (Dense)               (None, 512)               117965312
 dropout_4 (Dropout)         (None, 512)               0
 dense_1 (Dense)             (None, 128)               65664
 dropout_5 (Dropout)         (None, 128)               0
 dense_2 (Dense)             (None, 2)                 258
=================================================================
Total params: 118,420,994
Trainable params: 118,420,034
Non-trainable params: 960
_________________________________________________________________
```



3.2. f. Room for improvement

1. Augmenting the data to increase the volume of data, the model is being trained on.
2. Optimising Neural Network Architecture.
3. Better optimization of hyperparameters.
4. Adjusting the learning rate. (We used Adam rate i.e. 0.001)
5. Training on other quality images either 216x or 1536x

4.3 Mask R-CNN using Detectron2

For the final stage of the segmenting and identifying process, we use Detecron2, a library built by Facebook AI Research. We use this library to implement a Mask RCNN in order to identify and demarcate the boundaries of the M-A islands.

The Mask R-CNN architecture comprises mainly of three parts:

1. The Backbone Network:

```
[04/19 19:14:08 d2.engine.defaults]: Model:
GeneralizedRCNN(
  (backbone): FPN(
    (fpn_lateral2): Conv2d(256, 256, kernel_size=(1, 1), stride=(1, 1))
    (fpn_output2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (fpn_lateral3): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1))
    (fpn_output3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (fpn_lateral4): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1))
    (fpn_output4): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (fpn_lateral5): Conv2d(2048, 256, kernel_size=(1, 1), stride=(1, 1))
    (fpn_output5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (top_block): LastLevelMaxPool()
    (bottom_up): ResNet(
      (stem): BasicStem(
        (conv1): Conv2d(
          3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False
          (norm): FrozenBatchNorm2d(num_features=64, eps=1e-05)
        )
      )
      (res2): Sequential(
        (0): BottleneckBlock(
          (shortcut): Conv2d(
```

Fig 4.3.1: A portion of Detectron2's ResNet backbone implementation

The backbone network is used to convert a raw image into a feature map, in order to achieve faster computation times and improve accuracy by making it easier to find and demarcate the relevant feature, which in our case is the M-A islands.

This is done by using multiple convolution layers, which act as bottlenecks to filter out the important features. Since passing the image through multiple convolutions is computationally intensive, the image is downsampled to a smaller size, which is faster to work on, and then upsampled before outputting the feature map.
This process can cause a lot of data loss, which is solved by using a method called ResNet (Residual Network), which combines the upsampled image with the outputs of previous convolutions, by having direct connections between much older layers and the newer layers that skips a few in between. This preserves most image data.

2. The Region Proposal Network:
   The region proposal network, as the name suggests, proposes possible regions where there may be possible M-A islands. It does this as two parts: a binary object classification, which uses a CNN to detect whether or not there is the required object in the image or a region of the image, and a bounding box regression. The bounding box regression works by placing an anchor at a point in the image, generally the center, and drawing boxes of various sizes around this anchor, proposing the ones that match the closest with actual bounding boxes as regions.
   Together, these give possible regions where M-A islands may be present.

```
(proposal_generator): RPN(
  (rpn_head): StandardRPNHead(
    (conv): Conv2d(
      256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)
      (activation): ReLU()
    )
    (objectness_logits): Conv2d(256, 3, kernel_size=(1, 1), stride=(1, 1))
    (anchor_deltas): Conv2d(256, 12, kernel_size=(1, 1), stride=(1, 1))
  )
  (anchor_generator): DefaultAnchorGenerator(
    (cell_anchors): BufferList()
  )
)
```

Fig 3.3.2: Detecron2's RPN implementation

Before the next step, a step called Regions of Interest (RoI) alignment is performed, which makes all the regions of interest the same image size tensor in order to make operations on them easier.

3. The Head:
   The head consists of two portions, the object detection head and the mask generation head. The object detection head classifies objects, which is not needed in our case as we only have one, the M-A islands. The mask detection head uses a fully convolutional network (FCN) to find the M-A island location and create a mask over it that demarcates the island. With this, the process is completed.

4.3 a. Implementation

We trained Detectron2's mask-RCNN on an annotated dataset with bounding boxes. The images were split into the three regimes using the K-Means method and DFS, and each regime was considered separately. The images were cropped into 256x256 smaller images and augmented as well. Augmentation included contrast changes, image flipping and brightness changes. This overall doubled the dataset size, from 432 to 864 images.
The images and their JSON annotation and bounding box files were registered using the COCO (Common Objects in Context) format for use by the Detectron2 library for training.

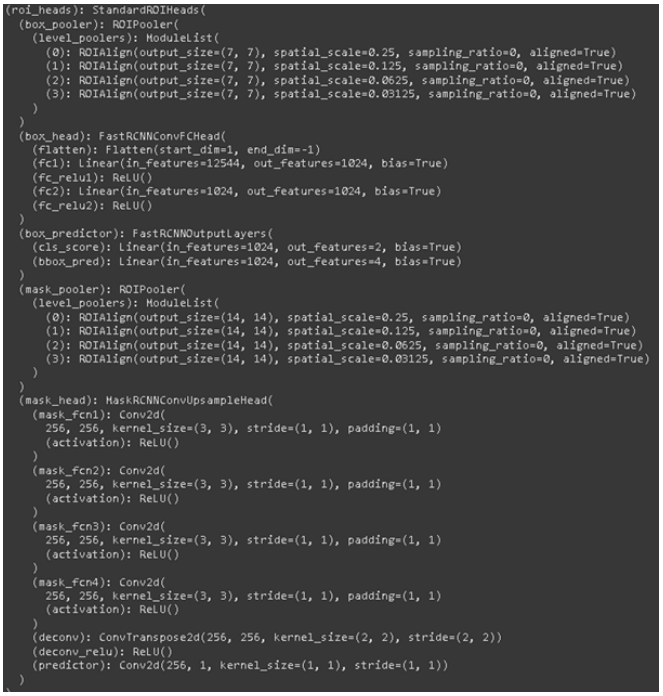


Fig 3.3.3: Detectron2's implementation of the head

```
cfg = get_cfg()
cfg.merge_from_file(model_zoo.get_config_fil
cfg.DATASETS.TRAIN = (tr_name,)
cfg.DATASETS.TEST = (val_name,)
cfg.DATALOADER.NUM_WORKERS = 2
cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint
cfg.SOLVER.IMS_PER_BATCH = 2  # This is the
cfg.SOLVER.BASE_LR = 0.0005  # learning rate
cfg.SOLVER.GAMMA = 0.05 #Learning rate chang
cfg.SOLVER.MAX_ITER = 750  # no of iteration
cfg.SOLVER.STEPS = []        # do not decay
cfg.MODEL.ROI_HEADS.BATCH_SIZE_PER_IMAGE = 1
cfg.MODEL.ROI_HEADS.NUM_CLASSES = 1  # only
#cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = 0.4
cfg.TEST.EVAL_PERIOD = 20
cfg.OUTPUT_DIR="/content/Out"
#cfg.MODEL.DEVICE="cpu"
os.makedirs(cfg.OUTPUT_DIR, exist_ok=True)
trainer = CocoTrainer(cfg)
#trainer = DefaultTrainer(cfg)
trainer.resume_or_load(resume=False)
trainer.train()

#Extracting trained model weigths

cfg.MODEL.WEIGHTS = os.path.join(cfg.OUTPUT_
cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = 0.8
```

Fig 3.C.4: Some modifiable hyperparameters implemented, most left default

For training, several hyperparameters can be used. We considered mainly 4, namely the batch size, the number of iterations , the base learning rate and learning rate decay.

***Batch Size*:** The number of images sampled before updating the model parameters. This was set to 2, which increases computation times. Possibly due to the relatively small dataset size, this does not hurt convergence.
***Iterations*:** The number of passes of one batch size each. This means the model is updated as many times as the number of iterations. After some experimenting, we arrived at 750 as the ideal number of iterations. Beyond this, there is little to no reduction in total loss, and soon even overfitting starts occurring, which is seen in increasing validation loss.
***Base Learning Rate*:** This hyperparameter dictates how big the gradient descent function's steps are initially. This parameter was left at the default, namely, 0.0005.

Learning Rate Decay: Learning rate for a set of iterations is given as the product of the learning rate for the previous set of iterations and the learning rate decay. This parameter was set to 0.05, and updated every 20 iterations.

For the base learning rate and learning rate decay, there seemed to be little difference in results even if they were set to a value a few magnitudes of order different. Convergence became an issue only if the learning rate exceeded approximately 0.05. Our hypothesis as to why this was happening is that either the dataset size being small made the learning rate have a negligible effect, or the gradient descent function was finding the minima quite fast, due to which changing the hyperparameter value was not doing much.

4.3 b. Results

Evaluation was performed using the test/validation dataset. The evaluation proved that augmentation was quite effective, with an average precision in the non-augmented dataset being around 30, while the average precision in the augmented dataset achieved a significantly better 52. If we consider all M-A islands with an above 50% confidence as accurate, which will often be a reasonable assumption, the average precision is much higher at around 81.

```
[04/19 19:42:31 d2.evaluation.coco_evaluation]: Evaluation results for segm:
|   AP   |  AP50  |  AP75  |  APs   |  APm   |  APl   |
|:------:|:------:|:------:|:------:|:------:|:------:|
| 51.429 | 81.143 | 55.515 | 27.660 | 71.653 | 82.580 |
```

Fig 3.3.5: Precision values for evaluation on test dataset

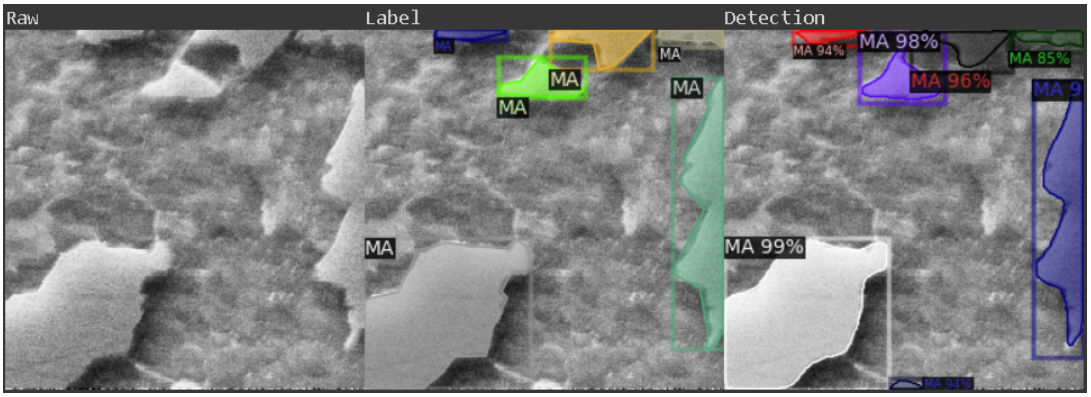


Fig 3.3.6: Side by side comparison of the raw image, original annotated image and output using Detectron2

4. Conclusion

We hereby have successfully segmented the MA islands present in bainitic steels. With our model one can segment MA islands, find the volume percentage of the same and get an insight into its mechanical properties. Even though we have trained it for this particular problem, one can use the same framework for segmenting any simple microstructure by training the model on a different dataset.

5. Future Work

Our project was concerned only with a special category of steel and that too with only two phases which is a single-class segmentation problem from a ML point of view. And we stopped at the 3rd stage of the ML process - training.

However we would like to take this opportunity to develop a more general framework to detect a lot more classes on complex steel micrographs. So that we could finally reach the 4th level - applying the ML model thus made into segmenting different phases present for an actual industrial application.

6. Author's contribution

[a]Implementing K-Means for generating bounding boxes (MM21B035)

[b]Training RCNN model for segmentation using Detectron2 (MM21B034)

[c]Literature Review & Training RCNN model for segmentation using Detectron2 (MM21B030)

[d]Training and implementing CNN for classification (MM21B052)

7. References

[1] Ackermann, Marc & İren, Deniz & Wesselmecking, Sebastian & Shetty, Deekshith & Krupp, Ulrich. (2022). Automated segmentation of martensite-austenite islands in bainitic steel. Materials Characterization. 191. 112091. 10.1016/j.matchar.2022.112091.

[2] Dataset: https://figshare.com/articles/dataset/Image_data_and_labels/19232523

[3] Detectron 2 documentation: https://detectron2.readthedocs.io/en/latest/

[4] TensorFlow documentation: https://www.tensorflow.org/api_docs

[5] scikit-learn documentation (K-Means): https://scikitlearn.org/stable/modules/generated/sklearn.cluster.KMeans.html