Severstal Steel Defects Detection using U-Net Image Segmentation Architecture

Project Proposal

Image segmentation constitutes a special case of computer vision projects. They build upon the encoding task associated with object detection and classification, and add a decoding element to build an image equal to the original size. This involves convolution and max pooling for the encoding task, and deconvolution and concatenation for the decoding task. An image pixel mask is then used as a label set to train the network.

The Severstal steel defects detection task is to localize and classify surface defects on a steel sheet. Flat sheet steel production is quite delicate. Throughout the processes of heating and rolling, drying and cutting, several machines are in contact with the steel, and hence introduce a variety of defects on the surface. Severstal uses images from high frequency cameras to power a defect detection algorithm, and they are looking to improve industrial algorithms using deep learning techniques. While this dataset is a part of a Kaggle challenge, I intend to add my exploratory data analysis and own approaches to complete that challenge. The task is to train the model and evaluate on the Dice coefficient, which is the pixel-wise agreement between a predicted segmentation and its corresponding ground truth. The training set contains 12568 images of 256 x 1600 pixels each. The labels contain the run-length encoded (RLE) defect masks. There are 1801 test images used for the prediction of the four defects.

The metadata contains:

1. Sample submission file
2. List of filenames in the train dataset with their classified masks
3. Train images
4. Test images

The general methodology in solving the task is:

1. Perform exploratory data analysis on the data to identify trends and study the defect patterns
2. Identify compute resources to handle image data
3. Import OpenCV, keras libraries and modules
4. Set up the model architecture for a commonly-employed image segmentation model
5. Fit model, test on validation data, run for few epochs, evaluate the model fit
6. Predict masks for test images

The deliverables for this project would include the code, a paper and a blog.

Introduction

The data for the steel sheets defect image segmentation project was obtained from Severstal and Kaggle repositories (Kaggle Inc, 2019). Identifying different types of steel sheet defects is critical to improving Severstal's automation, increasing efficiency, and maintaining high quality in their production. The company is taking major steps towards combining newer technologies with steel production. As such, deep Learning techniques are currently being employed to characterize and correctly identify defects in images taken by high frequency cameras.

Description of Dataset

The provided dataset contains train and test images with an area of 256 x 1600 pixels each, and the labels have the pixel values where the defects are segmented. They have been encoded in a run-length encoding (RLE) style, and need to be first transformed into an area of 256 x 1600 pixels, before being fitted on the images. For example, '1 3 10 5' implies pixels 1,2,3,10,11,12,13,14 are to be included in the mask. The metric checks that the pairs are sorted, positive, and the decoded pixel values are not duplicated. The pixels are numbered from top to bottom, then left to right: 1 is pixel (1,1), 2 is pixel (2,1), etc.

The Sorensen-Dice coefficient is used as the metric of evaluation: it is used to gauge the similarity of two samples. It is commonly used in image segmentation, and can be used to compare the pixel-wise agreement between a predicted segmentation and its corresponding ground truth. The formula is given by the following, where X is the predicted set of pixels and Y is the ground truth:

$$\frac{2 * |X \cap Y|}{|X| + |Y|}$$

The analysis of this project was possible with the help of xhlulu's boilerplate code (xhlulu, 2019). The author graciously agreed to share their work on public Kaggle kernels, and I was able to fork off of their work and introduce many of my own concepts.

The images are represented as follows (Figure 1). The left image shows the original image as converted using the CV2 library, and the right image shows the defects masked on to the original image. The right images have been transformed using the encoded pixels as explained below. Viridis color scale has been employed to showcase the four defects by different colors: 1 = green, 2 = light blue, 3 = dark blue, 4 = yellow.

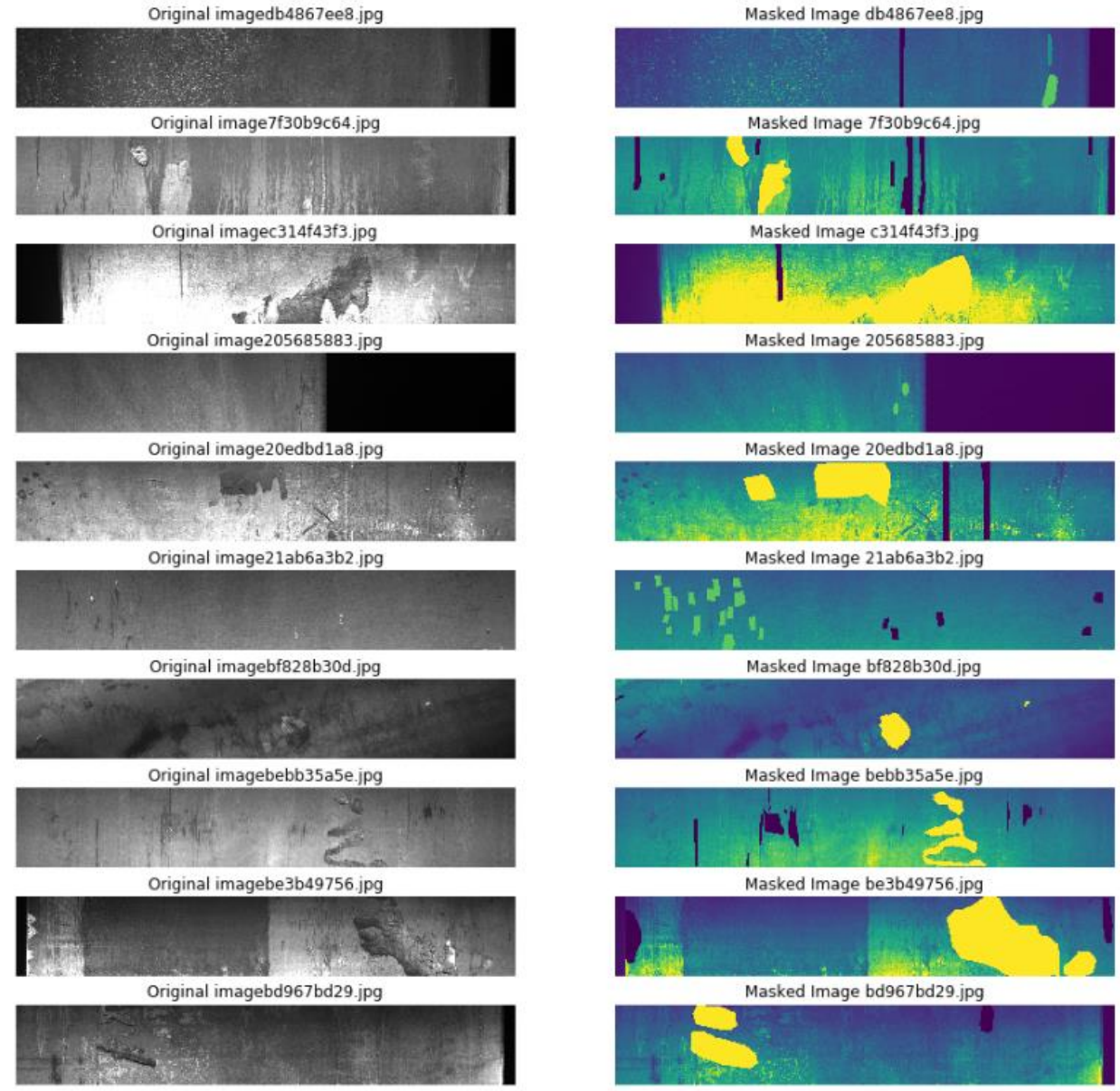

Figure 1: Original and Masked Image Subset of the 12568 Steel Images

Data Preprocessing and Wrangling

The preprocessed data had columns of ImageId_ClassId and EncodedPixels. The first step involved splitting the ImageId and ClassId from each other, and in classifying whether the image has a mask of defects. After wrangling, the data looked like the following table (Table 1).

Table 1: Train data after separating ImageID and ClassID

	ImageId_ClassId	EncodedPixels	ImageId	ClassId	hasMask
0	0002cc93b.jpg_1	29102 12 29346 24 29602 24 29858 24 30114 24 3...	0002cc93b.jpg	1	True
1	0002cc93b.jpg_2	NaN	0002cc93b.jpg	2	False
2	0002cc93b.jpg_3	NaN	0002cc93b.jpg	3	False
3	0002cc93b.jpg_4	NaN	0002cc93b.jpg	4	False
4	00031f466.jpg_1	NaN	00031f466.jpg	1	False

As portrayed in the table, each image has a separate mask for the four types of defects, and these masks will be used as labels for the image set.

To convert the values and images from pixels to images and vice-versa, the OpenCV Python library was used (OpenCV, 2019). Functions that built masks from RLEs and those that built RLEs from masks were defined. For example, *build_masks* function was used to first create a layer of zeros, and then call the *rle2mask* function. The *rle2mask* function was used to convert "10 4 30 2" to 10, 11, 12, 13, 30, 31, and assign them a value of 1. This function is a mask encoding function. Similarly, mask decoding function *mask2rle* was also defined for converting mask pixel data to RLE data.

Exploratory Data Analysis

The four types of defects were worth looking into, and their summary statistics are shown below in Figure 2. Defect 3 is quite common, showing up in about 40% of all images.

ClassId	num_defects
1	897.0
2	247.0
3	5150.0
4	801.0

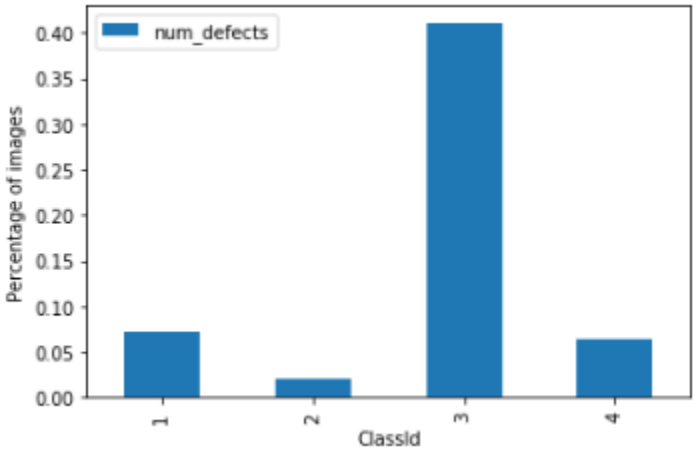

Figure 2: Absolute and Relative Number of Defects in the Dataset

Studying the defects more deeply, the area they covered per image has been visualized in their log histograms. The following figure (Figure 3) shows the number of images that have zero defects in each of the four defect categories, and the distribution of the area covered by the four defects (calculated as total number of pixels of defect / total number of pixels).

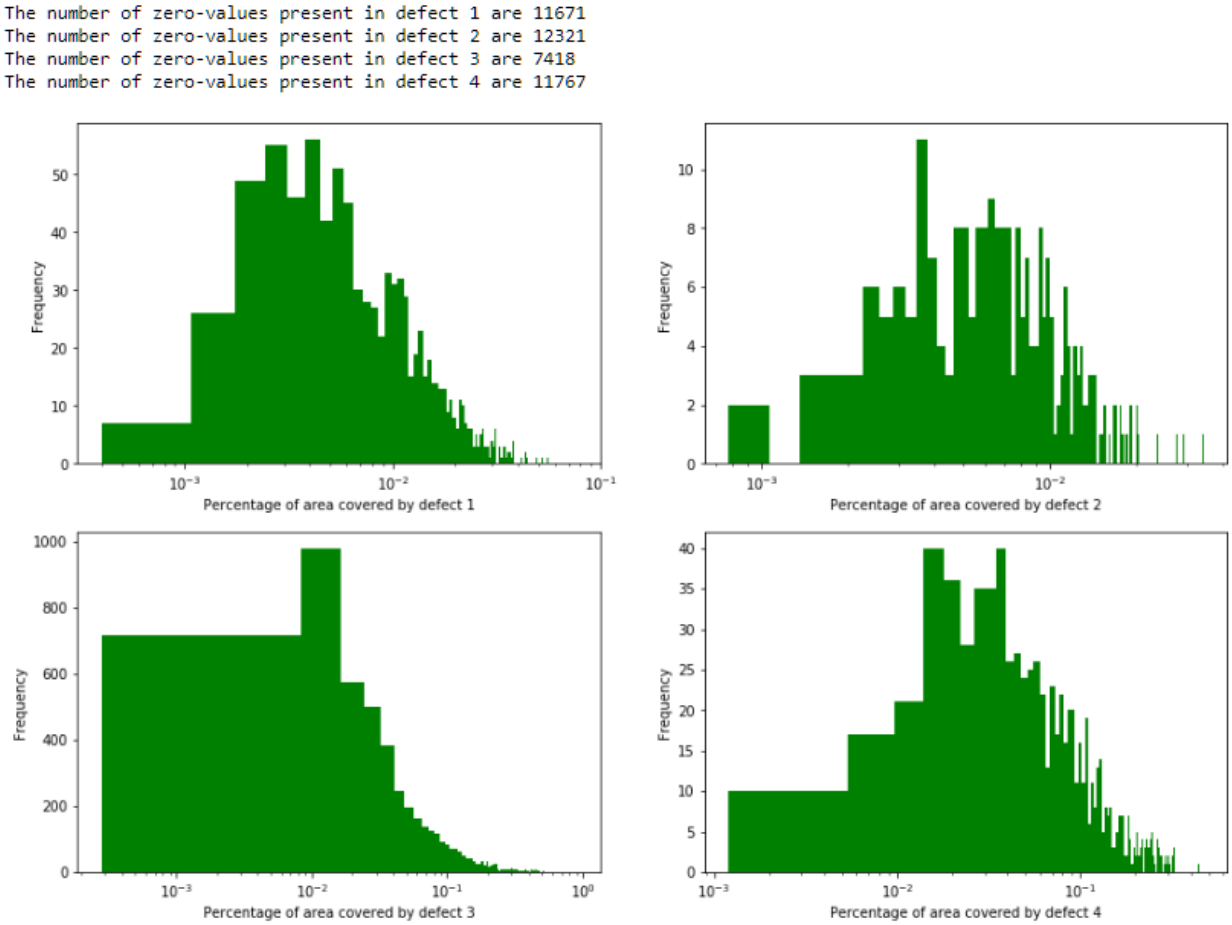

Figure 3: Total Areas covered by each of the 4 Defects on Log Histogram Distributions

The areas covered by both defects 3 and 4 occupy major portions of the image (between 0.01 and 0.1% of the total area), whereas defects 1 and 2 occupy tinier portions. The following table shows the descriptive statistics (Table 2) for the four defects. Again, defect 3 shows the highest values for means.

Table 2: Descriptive Statistics for the 4 Defects

	num_total_masks	defect_0_percentage	defect_1_percentage	defect_2_percentage	defect_3_percentage
count	12568.000000	12568.000000	12568.000000	12568.000000	12568.000000
mean	0.564529	0.000760	0.000162	0.025507	0.005349
std	0.560757	0.003605	0.001342	0.066735	0.027432
min	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.000000	0.000000	0.000000	0.000000	0.000000
50%	1.000000	0.000000	0.000000	0.000000	0.000000
75%	1.000000	0.000000	0.000000	0.020638	0.000000
max	3.000000	0.076423	0.034236	0.899023	0.470654

Convolutional Neural Networks

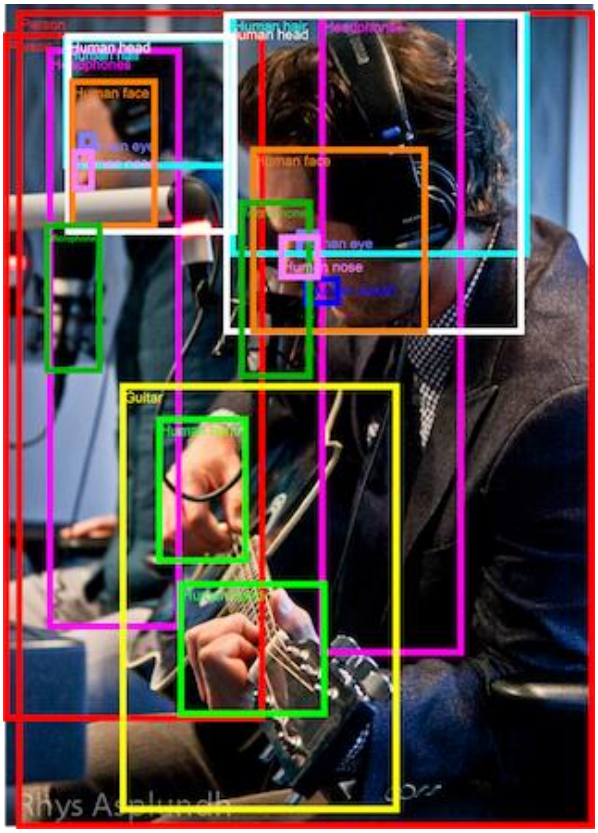

Figure 4: Bounding boxes and object identification example for CNN deep learning problems (Image source: Google Open Images V5 (Google Inc., 2019))

Object detection, classification and image segmentation tasks require a particular branch of deep learning called Convolutional Neural Networks (CNNs). Just like neural networks, the architecture uses a deep multi-layered multi-nodal system, which gets trained when an input (an image in our case) gets analyzed against a labelled dataset (object bounding boxes and classes in the case of object detection and classification). As such, an input in the form of a tensor with a shape (image height, image width, image depth in 3 colors) is added to the model. Then through two main operations, convolution and max pooling, the image is broken down into its base dimensions. Some excellent resources can be found online (Saha, 2018) (Skymind.ai, 2019).

In summary, convolution involves moving a smaller 3x3 or 5x5 square, called a kernel or filter across the entire image, in regular steps called strides. With the rectified linear unit (ReLU) averaging, each kernel has a particular 0 and 1 pattern. As it moves across image, each pixel value performs a dot product with this kernel, and the final value is stored in another composite layer. Running another iteration creates a

second convolution layer. Padding helps maintain the original dimensions of the image as we progress through the layers.

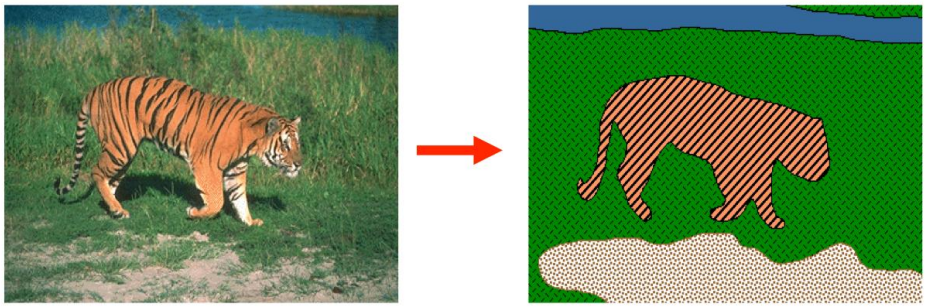

Figure 5: Image segmentation task, which requires encoding-decoding neural networks (Image source: Stanford AI Introduction to Computer Vision (Amy, 2019))

Max pooling involves running a 2x2 square across the entire layer, and taking the max value of the four pixel values to propagate into the next layer. Because of this operation, the original image size is halved. A series of convolutions and max pooling operations creates a final product as a vector of numbers. This method of reducing an image into a vector is called encoding. This vector is then transformed using a softmax distribution, and compared against a classification object or a bounding box and trained. A second image is run through the same model, and compared with the pretrained model weights and the actual object or bounding box. Based on distance from the ground truth, the weights are backpropagated and changed to reflect the two images. Many images need to be sent in to this auto-updating model to accurately classify images and bounding boxes. The softmax distribution is a normalization technique where the input values are reduced to between 0 and 1.

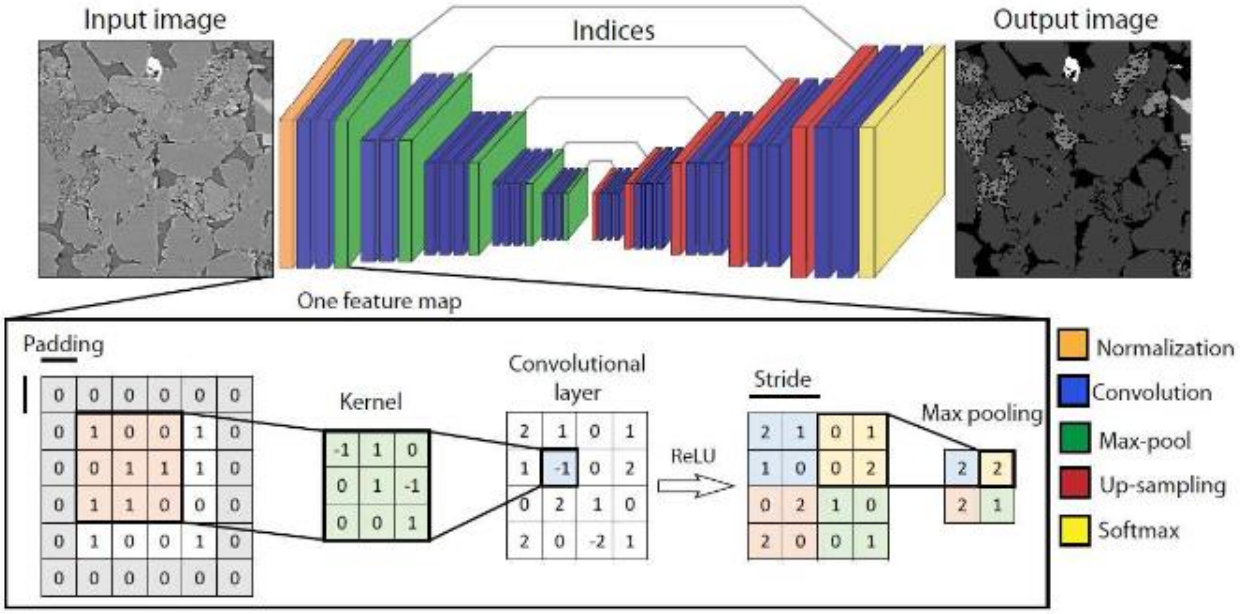

Figure 6: Image segmentation describing the convolution and max-pool operations. (image source: (Sadegh Karimpouli, 2019))

In our particular example of image segmentation, we are using the U-Net Image segmentation model, developed by (Olaf Ronneberger, 2015). The architecture has been taken from the paper, and is shown below. The architecture analyzes an encoder-decoder system. Here the encoded vector is further expanded out into the original image size using upsampling techniques and concatenation with previous transformed data. This process is called decoding.

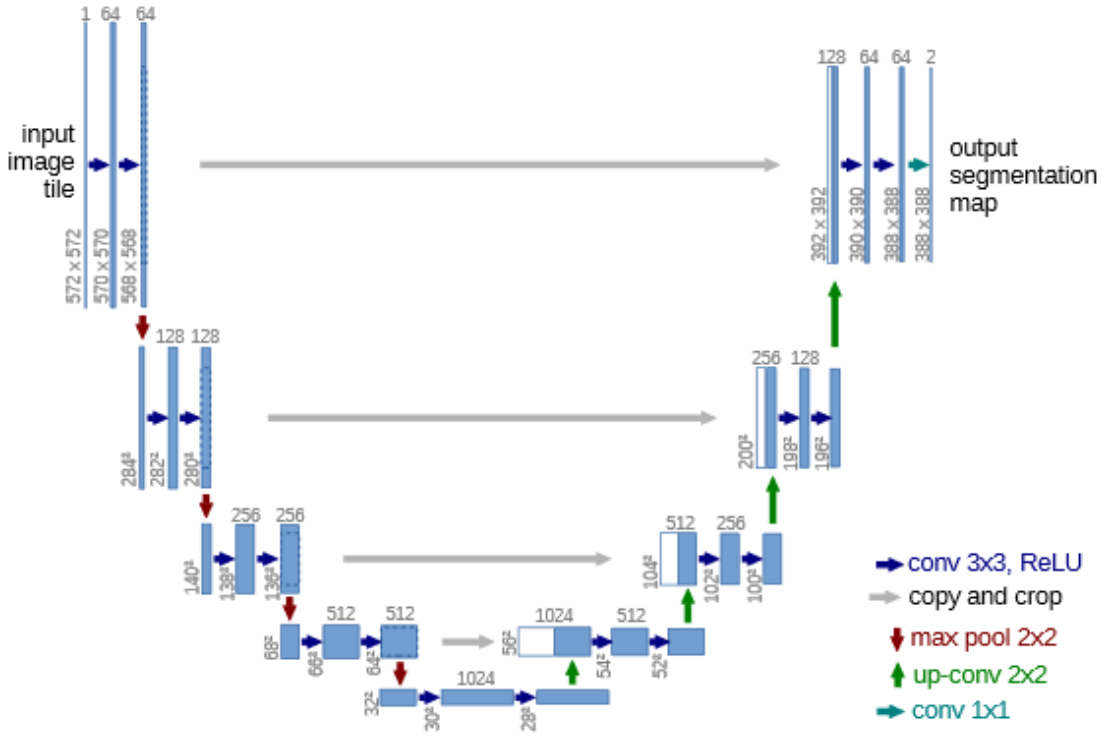

Figure 7: U-Net architecture, for a 572 x 572 pixel image. The arrows represent convolution or max-pooling operations. The number of channels or nodes, are denoted at the top of the boxes. The blue boxes represent multi-channel feature maps, and each white box represents copied feature maps. Source: Ronneberger et. al. (Olaf Ronneberger, 2015)

While there are many other architectures that are applicable in different situations, we will decide to use only the U-Net model due to limited compute resources.

Utility Functions

The dice coefficient as described above was coded as a function *dice_coef(y_true, y_pred).* Two other functions *bce_dice_loss* and *dice_loss* are defined for calculating the log-loss of the model.

To be able to load the large number of files in batches, a custom data generator class with the *keras.utils.Sequence* parameter was instantiated. The *__len__* method returns the number of batches per epoch, and *__get_item__(index)* returns the images (X) and the masks (y) associated with the indices of the training and the validation data. The *on_epoch_end()* function is used to trigger a shuffle of the indices at the end of an epoch, to create a more robust model.

A size of 32 images was randomly chosen to sequentially feed the data in batches.

Model Architecture

The model architecture is as follows:

```
def build_model(input_shape):
    inputs = Input(input_shape)

    c1 = Conv2D(8, (3, 3), activation='elu', padding='same') (inputs)
    c1 = Conv2D(8, (3, 3), activation='elu', padding='same') (c1)
    p1 = MaxPooling2D((2, 2)) (c1)

    c2 = Conv2D(16, (3, 3), activation='elu', padding='same') (p1)
    c2 = Conv2D(16, (3, 3), activation='elu', padding='same') (c2)
    p2 = MaxPooling2D((2, 2)) (c2)

    c3 = Conv2D(32, (3, 3), activation='elu', padding='same') (p2)
    c3 = Conv2D(32, (3, 3), activation='elu', padding='same') (c3)
    p3 = MaxPooling2D((2, 2)) (c3)

    c4 = Conv2D(64, (3, 3), activation='elu', padding='same') (p3)
    c4 = Conv2D(64, (3, 3), activation='elu', padding='same') (c4)
    p4 = MaxPooling2D(pool_size=(2, 2)) (c4)

    c5 = Conv2D(64, (3, 3), activation='elu', padding='same') (p4)
    c5 = Conv2D(64, (3, 3), activation='elu', padding='same') (c5)
    p5 = MaxPooling2D(pool_size=(2, 2)) (c5)

    c55 = Conv2D(128, (3, 3), activation='elu', padding='same') (p5)
    c55 = Conv2D(128, (3, 3), activation='elu', padding='same') (c55)

    u6 = Conv2DTranspose(64, (2, 2), strides=(2, 2), padding='same') (c55)
    u6 = concatenate([u6, c5])
    c6 = Conv2D(64, (3, 3), activation='elu', padding='same') (u6)
    c6 = Conv2D(64, (3, 3), activation='elu', padding='same') (c6)

    u71 = Conv2DTranspose(32, (2, 2), strides=(2, 2), padding='same') (c6)
    u71 = concatenate([u71, c4])
    c71 = Conv2D(32, (3, 3), activation='elu', padding='same') (u71)
    c61 = Conv2D(32, (3, 3), activation='elu', padding='same') (c71)

    u7 = Conv2DTranspose(32, (2, 2), strides=(2, 2), padding='same') (c61)
    u7 = concatenate([u7, c3])
    c7 = Conv2D(32, (3, 3), activation='elu', padding='same') (u7)
    c7 = Conv2D(32, (3, 3), activation='elu', padding='same') (c7)

    u8 = Conv2DTranspose(16, (2, 2), strides=(2, 2), padding='same') (c7)
    u8 = concatenate([u8, c2])
    c8 = Conv2D(16, (3, 3), activation='elu', padding='same') (u8)
    c8 = Conv2D(16, (3, 3), activation='elu', padding='same') (c8)

    u9 = Conv2DTranspose(8, (2, 2), strides=(2, 2), padding='same') (c8)
    u9 = concatenate([u9, c1], axis=3)
```

```
    c9 = Conv2D(8, (3, 3), activation='elu', padding='same') (u9)
    c9 = Conv2D(8, (3, 3), activation='elu', padding='same') (c9)

    outputs = Conv2D(4, (1, 1), activation='sigmoid') (c9)

    model = Model(inputs=[inputs], outputs=[outputs])
    adam = Adam(learning_rate=0.001)
    model.compile(optimizer=adam, loss=bce_dice_loss, metrics=[dice_coef])

    return model
```

The Adam optimizer was used for compiling at a learning rate of 0.001. The current model takes in images with at least one type of defect, as a low accuracy was being registered when images with no defects were also being added. As a result, the current model has improved metrics.

MODEL RESULTS

The details of the 7 epoch runs are as follows. The loss values show large negative values. This could be the result of the function not averaging the pixel-wise losses, and instead showing the combined values of the pixel losses. The parameter of interest is the dice coefficient though, and it shows a 0.6868 value during its training and validation over 7 epochs. This is quite a good model for a first-time effort.

Epoch 1/7
177/177 [==============================] - 265s 1s/step - loss: -38669029316707909369856.0000 - dice_coef: 0.6955 - val_loss: -69966797910921183232000.0000 - val_dice_coef: 0.6877
Epoch 2/7
177/177 [==============================] - 249s 1s/step - loss: -9289937371560017290854 4.0000 - dice_coef: 0.6893 - val_loss: -111755698853292015616000.0000 - val_dice_coef: 0.6882
Epoch 3/7
177/177 [==============================] - 251s 1s/step - loss: -1302837310174147400171 52.0000 - dice_coef: 0.6897 - val_loss: -1451135730628491277762 56.0000 - val_dice_coef: 0.6880
Epoch 4/7
177/177 [==============================] - 252s 1s/step - loss: -1621595315535856145530 88.0000 - dice_coef: 0.6894 - val_loss: -1757142805301352390983 68.0000 - val_dice_coef: 0.6878
Epoch 5/7
177/177 [==============================] - 250s 1s/step - loss: -1931495023877419119411 20.0000 - dice_coef: 0.6892 - val_loss: -2059157078616579369861 12.0000 - val_dice_coef: 0.6875
Epoch 6/7
177/177 [==============================] - 251s 1s/step - loss: -2234596785720158679203 84.0000 - dice_coef: 0.6887 - val_loss: -2354589070860426726604 80.0000 - val_dice_coef: 0.6871
Epoch 7/7
177/177 [==============================] - 252s 1s/step - loss: -2529897859086160096133 12.0000 - dice_coef: 0.6885 - val_loss: -2641776593482424558551 04.0000 - val_dice_coef: 0.6868

The dice coefficient evaluation graph is shown as follows:

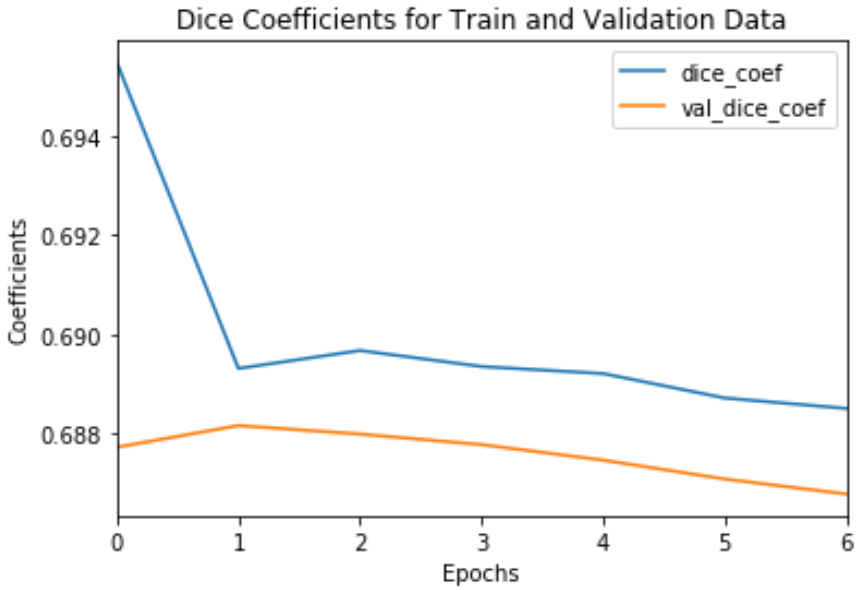

Figure 8: Dice coefficients for the training and the validation datasets, over each epoch

The model (model1.h5) was then run against the test results inputted as a data generator class as shown above. The images were segmented and the pixels were encoded in the RLE format. This dataframe was then imported as an Excel file, and is attached as part of the deliverables.

Future Work

Because image segmentation is an active area of research, various other architectures are available for image segmentation tasks, that could be used to get better fit on the test data. These include base Fully Convolutional Networks (FCNs), ParseNet, Feature Pyramid Network (FPN), Pyramid Scene Parsing Network (PSPNet), Mask R-CNN, DeepLabv3, Path Aggregation Network (PANet), Context Encoding Network (EncNet). Each network represents the particular applications where they can be used, and incorporating these changes can create better masks.

To improve upon the existing U-Net model, many other changes can be made. The number of nodes per layer could be increased by a factor of 2, thus increasing the resolution of the trained weights. Another convolution and corresponding deconvolution layer can be added. A train-test split that was more representative of the defect distribution could have definitely improved the model accuracy and Dice coefficient.

Conclusions

Using the current U-Net model, a Dice coefficient of 0.681 was obtained for the train and validation data. This model's weights were then used to determine the defects of the test set. The output submission includes the run-length encoded image masks in the attached submission.xlsx file, and the model file model1.h5. Also included is the code writeup as an iPython notebook.

BIBLIOGRAPHY

Amy, V. a. (2019). *Introduction to Computer Vision: Tutorial 3: Image Segmentation*. Retrieved 08 02, 2019, from https://ai.stanford.edu/~syyeung/cvweb/tutorial3.html

Google Inc. (2019). *Open Images Dataset V5+ Extensions*. (Google Inc.) Retrieved 08 2019, from https://storage.googleapis.com/openimages/web/index.html

Kaggle Inc. (2019). *Severstal: Steel Defect Detection*. Retrieved from Kaggle.com: https://www.kaggle.com/c/severstal-steel-defect-detection

Olaf Ronneberger, P. F. (2015). U-Net: Convolutional Networks for Biomedical Image Segmentation. In H. J. Navab N. (Ed.), *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015* (Vol. 9351). Springer, Cham. doi:https://doi.org/10.1007/978-3-319-24574-4_28

OpenCV. (2019). *OpenCV 4.1.1*. (OSS) Retrieved 08 2019, from https://opencv.org/

Sadegh Karimpouli, P. T. (2019). Segmentation of digital rock images using deep convolutional autoencoder networks. *Computers and Geosciences, 126*(May), 142-150.

Saha, S. (2018, 12 15). *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way*. (Towards Data Science - Medium.com) Retrieved 09 05, 2019, from https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

Skymind.ai. (2019). *A Beginner's Guide to Convolutional Neural Networks (CNNs)*. (Skymind) Retrieved 10 01, 2019, from https://skymind.ai/wiki/convolutional-network

xhlulu. (2019). *Severstal: Simple Keras U-Net Boilerplate*. (Kaggle) Retrieved 08 05, 2019, from https://www.kaggle.com/xhlulu/severstal-simple-keras-u-net-boilerplate

Appendix: Initial Project Ideas

Google 8M Segments Dataset

Human-verified segment annotations aligned with 5-second Youtube video segments describe the contents of this dataset. The Kaggle challenge asks the users to temporally localize events and topics within these video snippets: https://www.kaggle.com/c/youtube8m-2019. The objective of this competition is to predict the Class labels of these Youtube video segments. The metadata looks like this:

- Each video has:
  1. id: unique id for the video, in train set it is a YouTube video id, and in test/validation they are anonymized.
  2. labels: list of labels of that video.
  3. Each frame has rgb: float array of length 1024,
  4. Each frame has audio: float array of length 128

- A subset of the validation set videos are provided with segment-level labels. In addition to id, labels and the frame level features described above, they come with:
  1. segment_start_times: list of segment start times.
  2. segment_end_times: list of segment end times.
  3. segment_labels: list of segment labels.
  4. segment_scores: list of binary values indicating positive or negative corresponding to the segment labels.

The objective is to localize video-level labels to the precise time in the video where the label actually appears. Improved video search, safe content detection, and action moment detection are some of the advantages, among many others.

AVA Actions Dataset Analysis

430 15-minute Youtube video clips are labelled and classified using 80 visual actions, and bounded boxes are created and localized in space and time, resulting in 1.62M action labels with multiple labels per human occurring frequently. The dataset is available here: https://research.google.com/ava/index.html. Also here: https://deepmind.com/research/open-source/open-source-datasets/kinetics/ . The metadata looks like this: In the CSV files, each row describes one video and the columns are organized as follows:

- label - (string) a human-readable name for the action class. Characters used are lowercase letters, spaces, and single quotation ('). (Held out for the test set.)
- youtube_id - (string) the YouTube identifier of the video the segment was extracted from. One may view the selected video at http://youtu.be/${youtube_id}.
- time_start - (integer) the starting time of the action snippet in the video, in seconds.
- time_end - (integer) the ending time of the action snippet in the video, in seconds.
- split - (string) "train", "val", or "test".

Many questions can be posed to the datasets:

1. Creating accurate bounded boxes and correctly identifying the actions

2. Main speaker recognition using audio analysis with video sequences

Amazon Product Reviews (May 1996 - July 2014) for Recommendation Systems

A large crawl of product reviews from Amazon (http://jmcauley.ucsd.edu/data/amazon/index.html) with the following dataset features:

- reviews and ratings
- item-to-item relationships (e.g. "people who bought X also bought Y")
- timestamps
- helpfulness votes
- product image (and CNN features)
- price
- category
- salesRank

This dataset could be used to design recommendation systems on images, similar NLP-based reviews, categories, prices, or timestamps (when the products sell over the season).