Airbus Ship Detection

Dilemma Project Report

Submitted by:

Ramy Fetteha	19015649
Basel Ahmed	19015513

Contents

1 Problem Statement

The background is that shipping traffic is rapidly increasing. The likelihood of maritime offenses such environmentally disastrous ship accidents, piracy, illegal fishing, drug trafficking, and illicit cargo movement rises as the number of ships increases. This has forced numerous institutions, including national government authorities, insurance firms, and environmental protection agencies, to keep a closer eye on the open seas.
Airbus is eager to issue a challenge to Kagglers to create a model that can swiftly identify all ships in satellite photos.
A data set is given containing multiple satellite pictures and the challenge is to segment/identify the ships in the images.
The data set is as follows:

1. 193K training images.
2. 15.6 testing images.
3. train_ship_segmentations.csv file containing the run-length encoded (RLE) pixels of each ship in all the training images.

We will now venture through our journey to develop a working model that has a decent accuracy.

2 Progress

The project took us a span of 3 weeks where we stumbled upon many challenges and got to learn many new things.
The model was decided to be a semantic segmentation model with U-Net as a baseline model.

2.1 Week 1

Week 1 was probably the hardest week as it took us sometime to try to understand how the data will be handled.
Here is what was done in that week:

1. RLE encoding/decoding.
2. Cleaning the data.
3. Getting introduced to generator functions in python.
4. Building the baseline model.
5. Training the model.

2.1.1 Run-Length Encoding

"Run–length encoding (RLE) is a simple form of lossless data compression that runs on sequences with the same value occurring many consecutive times. It encodes the sequence to store only a single value and its count."
The train_segmentation.csv Fig.1 contains:

1. Image ID.
2. Encoded pixels.

000194a2d.jpg	360486 1 361252 4 362019 5 362785 8 363552 10 364321 10 365090 9 365858 10 366627 10 367396 9 368165...
000194a2d.jpg	51834 9 52602 9 53370 9 54138 9 54906 9 55674 7 56442 7 57210 7 57978 7 58746 7 59514 7 60282 7 6105...
000194a2d.jpg	198320 10 199088 10 199856 10 200624 10 201392 10 202160 10 202928 10 203696 10 204464 10 205232 10 ...
000194a2d.jpg	55683 1 56451 1 57219 1 57987 1 58755 1 59523 1 60291 1

Figure 1: train_segmentation.csv

Every entry in the .csv corresponds to a ship in the image. So multiple entries could have the same image ID.
We will proceed to explain the main RLE decoding functions used to generate our labels.

2.1.1.1 RLE Decode

Input: mask$_r le, shape$
Output: img
// Split the mask run-length encoding
s ← mask$_r le.split()$;
// Extract the start positions and lengths of runs
starts, lengths ← [np.asarray(x, dtype=int) **for** x **in** (s[0:][::2], s[1:][::2])];
// Adjust the start positions
starts − = 1;
// Calculate the end positions
ends ← starts + lengths;
// Create an array of zeros to represent the image
img ← np.zeros(shape[0]*shape[1], dtype=np.uint8);
// Set the corresponding elements to ones based on the runs
for *lo, hi* ***in*** *zip(starts, ends)* **do**
  img[lo:hi] ← 1;
end
// Reshape the image to align with the RLE direction
return *img.reshape(shape).T*;

2.1.1.2 Masks as Images

Input: in$_m ask_l ist$
Output: image
// Create an empty array to store all masks
all$_m asks$← np.zeros((768, 768), dtype=np.int16);
// Iterate through each mask in the input mask list
for *mask* ***in*** *in_mask_list* **do**
  // Check if the mask is a run-length encoded string
  if *isinstance(mask, str)* **then**
    // Decode the mask and add it to the accumulated masks
    all_masks += rle_decode(mask);
  end
end
// Expand the dimensions of the accumulated masks to represent channels
return *np.expand_dims(all_masks, -1)*;

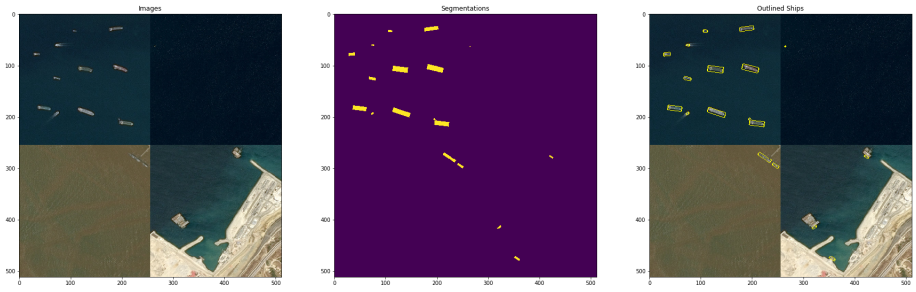

Figure 2: Montage of Decoded Output

2.1.2 Cleaning The Data

The data contained some corrupted images and images that were too small to analyze. So we proceed with filtering any entry with size less than 50 KB as done in this submission.

Additionally, we down-sample our empty images (images with no ships) as they

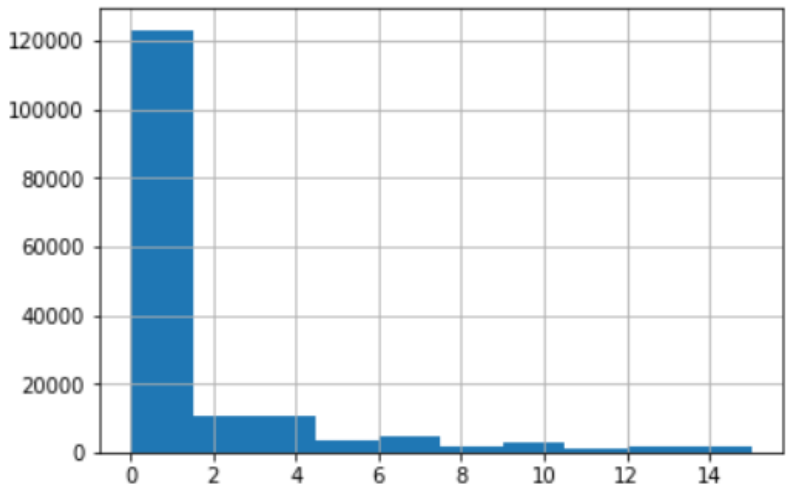

Figure 3: Imbalanced Data

make the data set imbalanced as it appear in fig.3 so down sample the empty images and make our group size equal to 1600(number of ships in one image) which appears in fig.4.

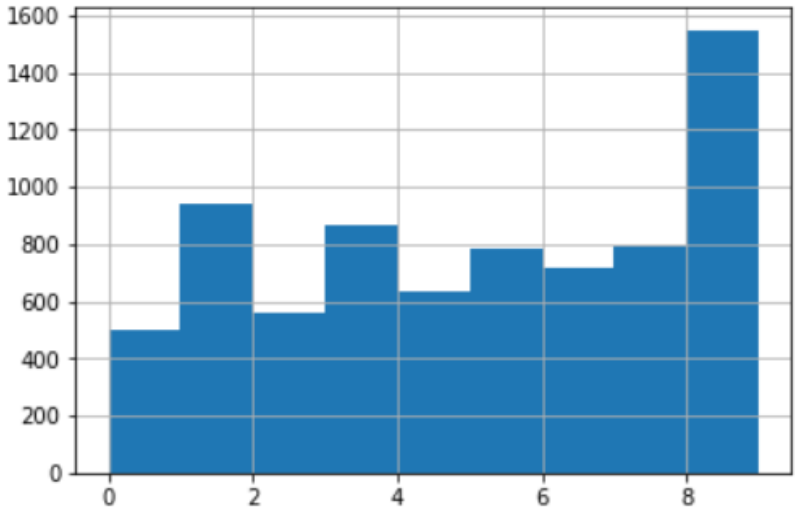

Figure 4: Data After Cleaning & Down-Sampling

2.1.3 Generator Functions

Since trying to load the image and its mask in memory ends up in crashing the kernel. So we use the python generator function to generate our training data.
We return two stacks of size equal to our batch size. One stacks contains original images and the other stack contains their corresponding RLE decoded masks.

2.1.4 Building The Model

We initially used the baseline U-Net model which appears in fig.5 with the original filter sizes which resulted in around 31 million learn-able parameters.

2.1.5 Training The Model

In our first week, we tried to train the model but the model wouldn't learn. Our metrics were Jaccard Index (IoU) and Dice Coefficient. For the loss, we used Cross Entropy.

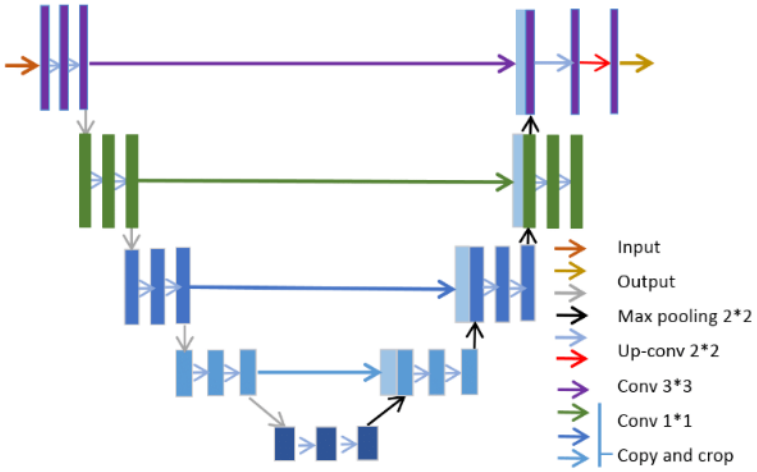

Figure 5: U-Net Baseline Model

2.2 Week 2

Our main focus for this week was to fix the problem with the baseline model by:

1. Decreasing the Training Data Size.
2. Lowering the Number of Filters of Our U-Net.
3. Hyper-parameter Tuning.
4. Image Scaling.
5. Data Augmentation.
6. Changing the Step Size and Number of Epochs.

2.2.1 Improving Baseline Model

2.2.1.1 Decreasing the Training Data Size

We initially had around 11k images after removing the empty pictures. We tried halving the size to 5k and reran the training, but the initial validation error increased and it never improved.

2.2.1.2 Lowering the Number of Filters of Our U-Net

We initially followed the main architecture of the U-Net with the same amount of filters. This resulted in a model with around 31 million parameters which is hard to train with the available resources. Instead, we divided all filters by 8 which lowered

our number of parameters to around 500K, but that resulted in an awful binary accuracy and a high initial validation loss without any improvement.

2.2.1.3 Hyper-parameter Tuning

We decided to change the learning rate of Adam Optimizer. Decreasing the learning rate didn't help as much and just made everything slower without much improvement. Increasing the learning rate wasn't helpful either.

2.2.1.4 Image Scaling

Our images were originally 768x768 and that proved to be quite troublesome as the computations were quite intense. So we down scaled it by a factor of 3, so we ended up having images and mask with size 256x256. It improved the initial validation loss and overall binary accuracy but the validation loss didn't improve.

2.2.1.5 Data Augmentation

We applied some augmentations to our data and corresponding masks such as rotations, flipping, zooming, shearing, ..etc. Not much improvement was noticed sadly.

2.2.1.6 Changing the Step Size and Number of Epochs

We decided to change the Step size to 9 and the number of epochs to 99 but that was done with the model with low filters so that the kernel doesn't crash. And It improved! But, The validation loss didn't change.

2.2.2 Baseline Fix

2.2.2.1 Fixing The Undersampling function

Due to the data imbalance, we have already done under-sampling to our data, but the first under-sampling function was incorrectly implemented so we reworked that where the data distribution changes as shown in fig.6 and the validation loss started gradually decreasing with each epoch as shown in fig.7
The resulting samples can be seen in fig.8 & 9

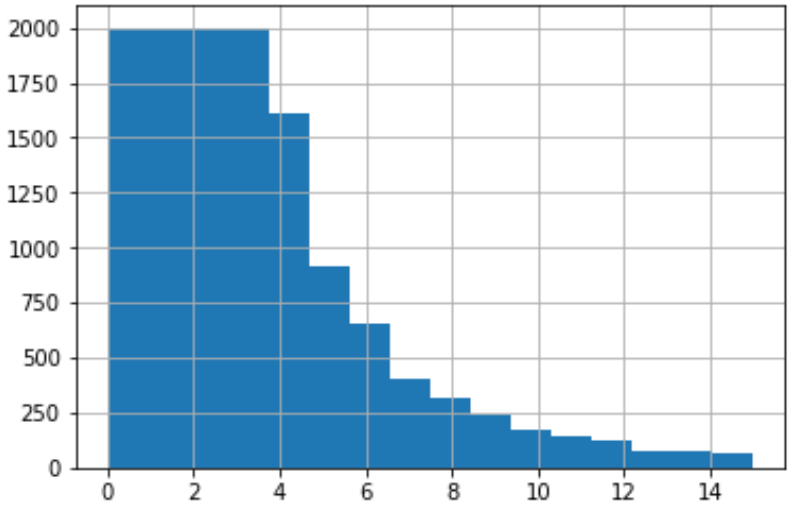

Figure 6: Data After Fixing the Downsampling

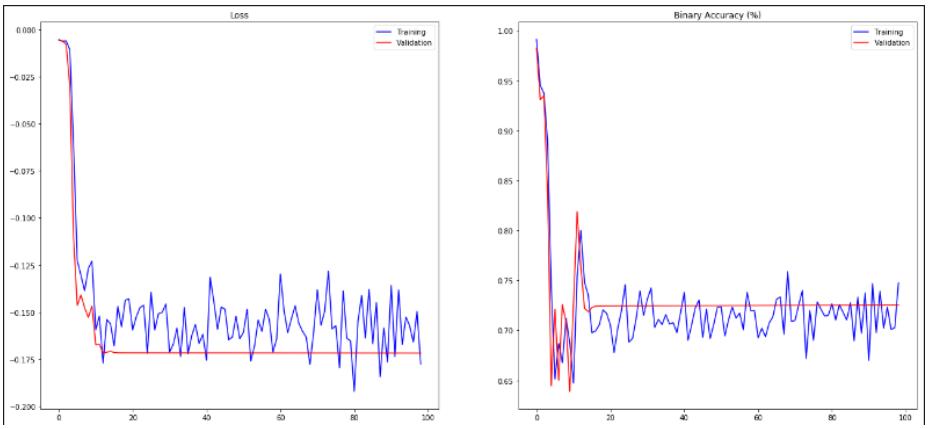

Figure 7: Loss(IoU) and binary accuracy on training and validation data

2.2.2.2 Notes

- We used IoU as our loss metric as it was used by a lot of other submissions and made it easier for us to track whether the model is improving or not.
- F2 score is used by the competition to evaluate the submissions but evaluating it manually would take alot of time so we didn't calculate it.

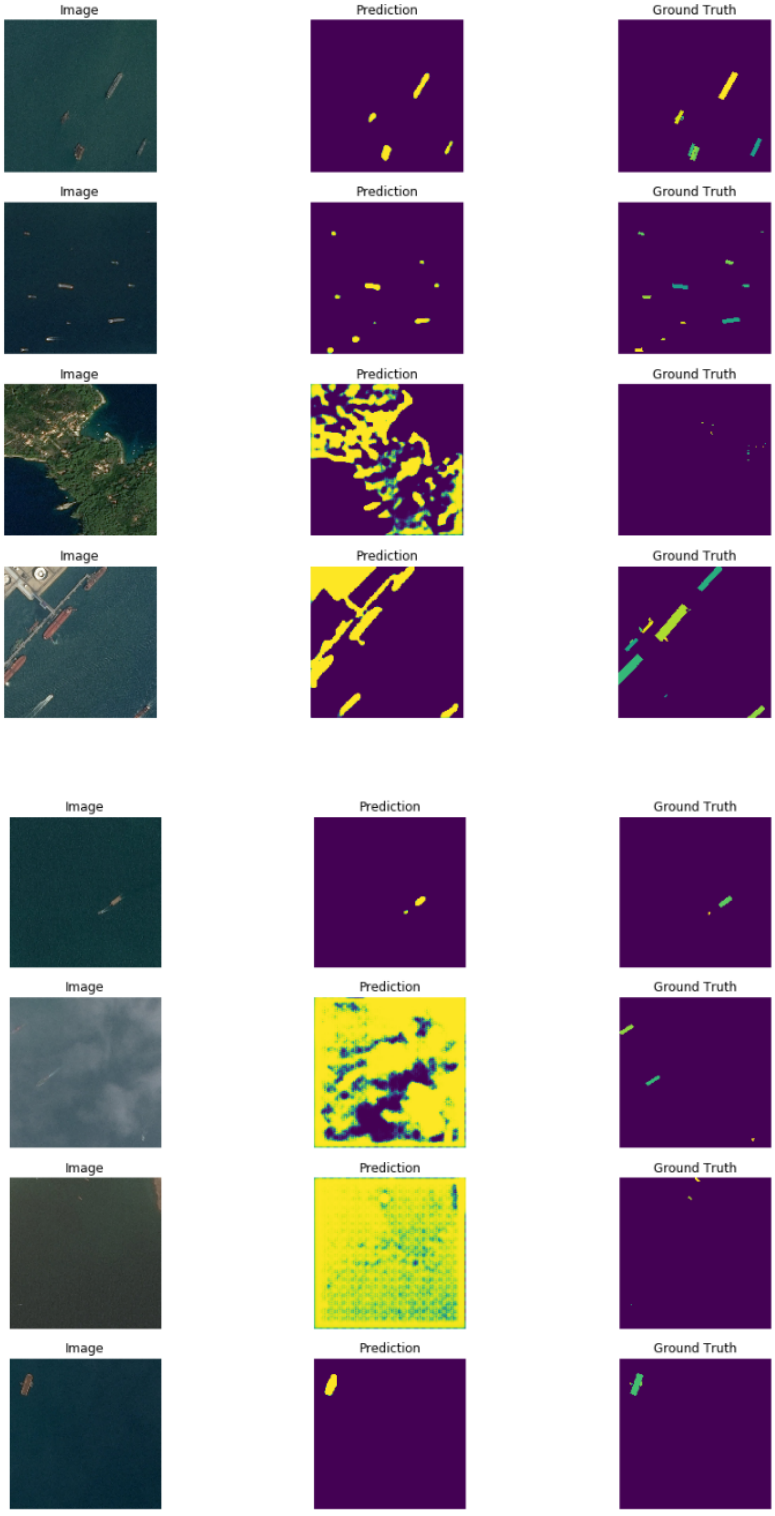

Figure 8: Model predictions on validation data

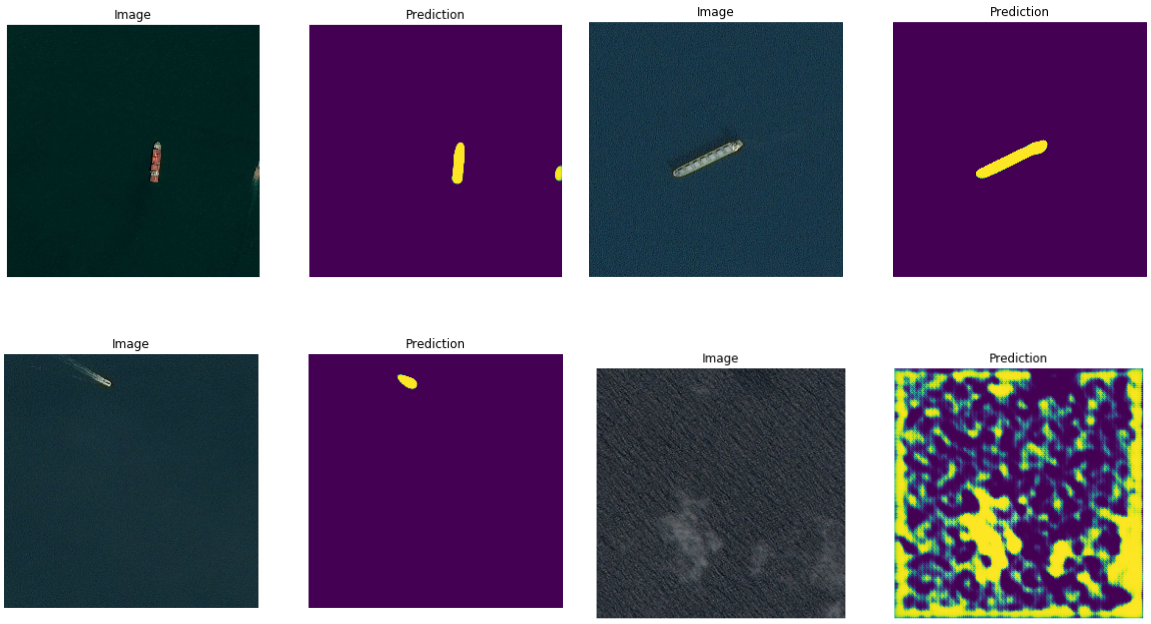

Figure 9: Model predictions on test data

2.2.3 U-net using pre-trained encoder (ResNet50, VGG19)

2.2.3.1 Using ResNet50

- We observed that ResNet50 layers cannot be fine tuned as memory crashes because of the large number of parameters to be tuned in addition to training the U-net decoder.
- We tried to freeze ResNet50 layers and only train U-net decoder but there is no training happens because the validation loss from the first epoch is not decreasing.

2.2.3.2 Using VGG19

VGG19 model as seen in fig.10 seems to be more relevant as its architecture is similar to the vanilla encoder in vanilla U-net architecture with respect to the number of filters used in different conv layers.

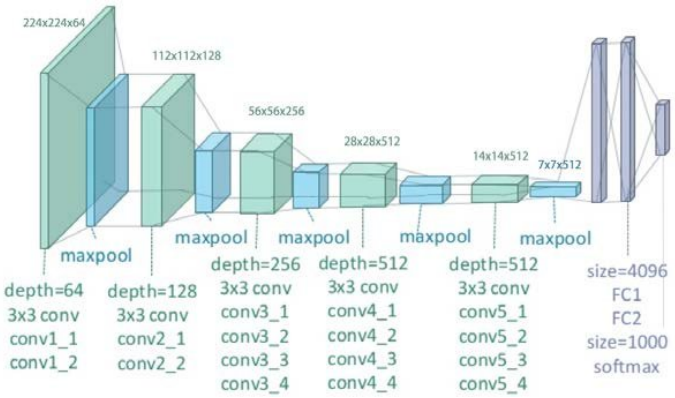

Figure 10: VGG19 architecture

2.2.3.3 Trials on VGG19 (11) (12) (13)

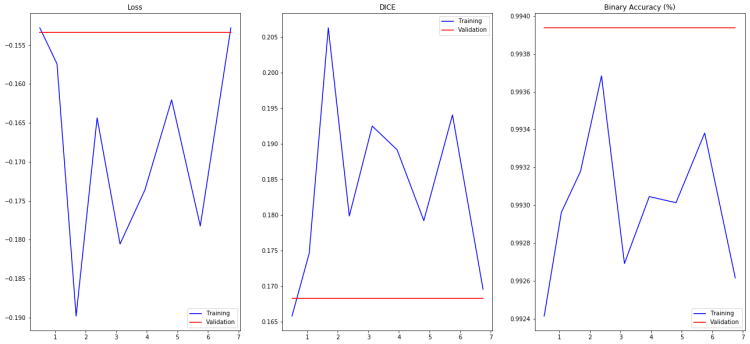

Figure 11: Using augmented images as Input to the model and freezing VGG19 layers and using number of filters in the U-net decoder like in vanilla U-net.

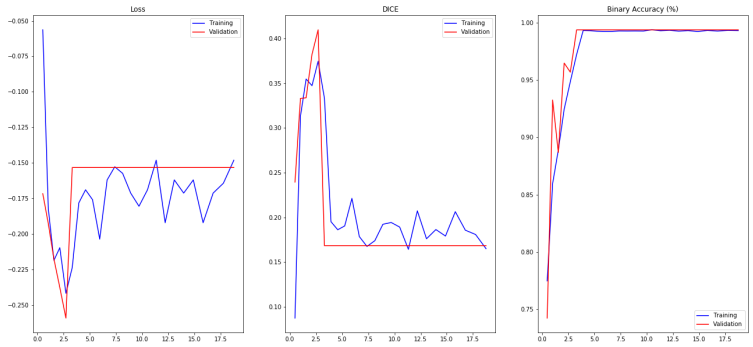

Figure 12: Using augmented images as Input to the model without freezing VGG19 layers and using number of filters in the U-net decoder like in vanilla U-net.

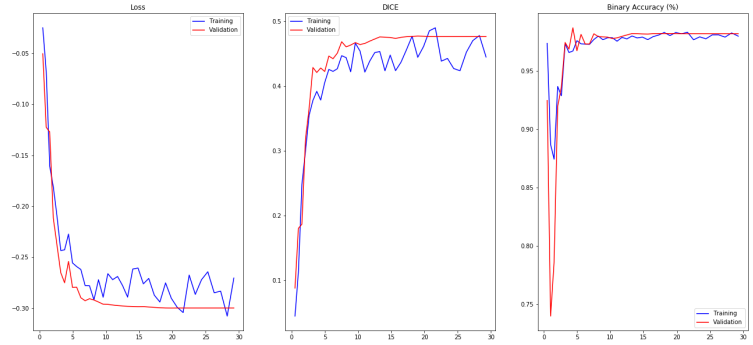

Figure 13: Using augmented images as Input to the model without freezing VGG19 layers and decreasing the no of filters in the U-net decoder compared to that of the vanilla U-net architecture

2.2.3.4 Comparing prediction on random samples of validation data with ground truth (14)

2.2.3.5 Showing prediction on random samples of test data (15)

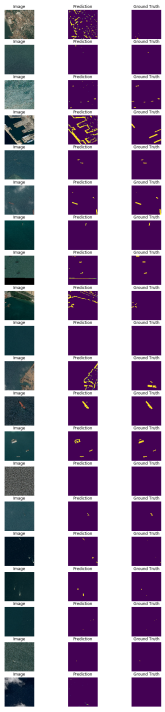

Figure 14: Model predictions vs ground truth on validation data

2.3 Week 3

In week 3, our main goal was to try to improve the baseline model and to over-fit our model to check whether the data or the model were causing the problems.
So we proceed to do the following:

1. Rerunning the model.
2. Checking if the model itself is flawed by testing on another data set.
3. Decreasing our training sample size.
4. Increasing Batch Size.
5. Increasing the learning rate.

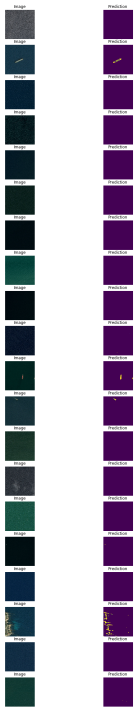

Figure 15: Model predictions on test data

2.3.1 Improving Baseline Model

2.3.1.1 Rerunning The Model

While it might seem strange, but retraining the model yielded better results Fig.16 17 18 than week 2 results.
Note: It was concluded that the cause of that behavior was that we didn't have a constant seed so every run might have been different than the other.

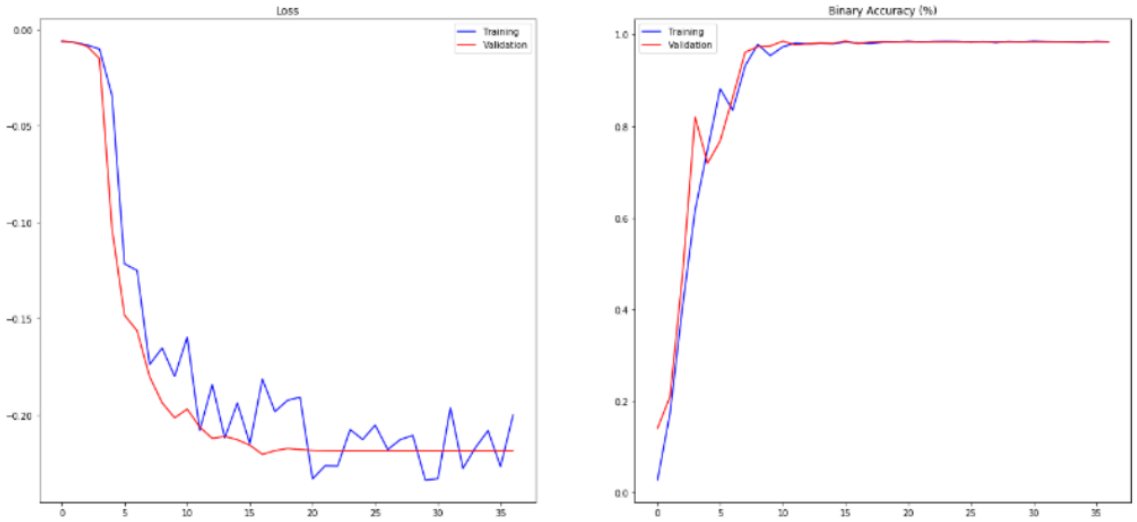

Figure 16: Metrics of Baseline

2.3.2 Checking the Correctness of the Model

We tested our model architecture (Baseline with decreased filter size) on a Brain Segmentation Data set that can be found here.

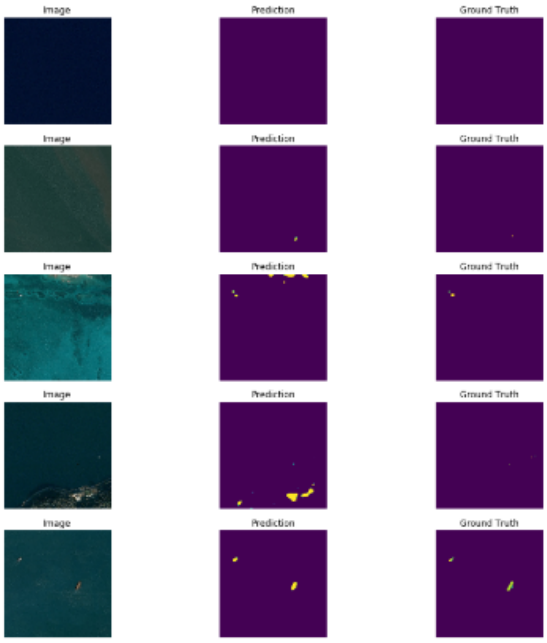
Image
Prediction
Ground Truth
Image
Prediction
Ground Truth
Image
Prediction
Ground Truth
Image
Prediction
Ground Truth
Image
Prediction
Ground Truth

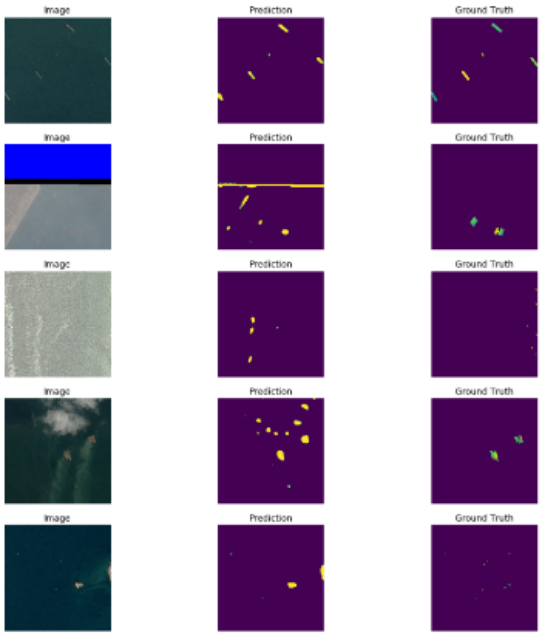
Image
Prediction
Ground Truth
Image
Prediction
Ground Truth
Image
Prediction
Ground Truth
Image
Prediction
Ground Truth
Image
Prediction
Ground Truth

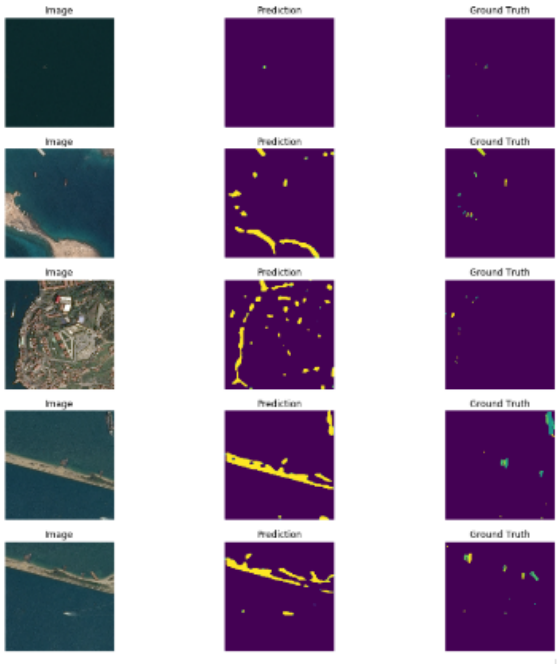

Figure 17: Model predictions on validation data

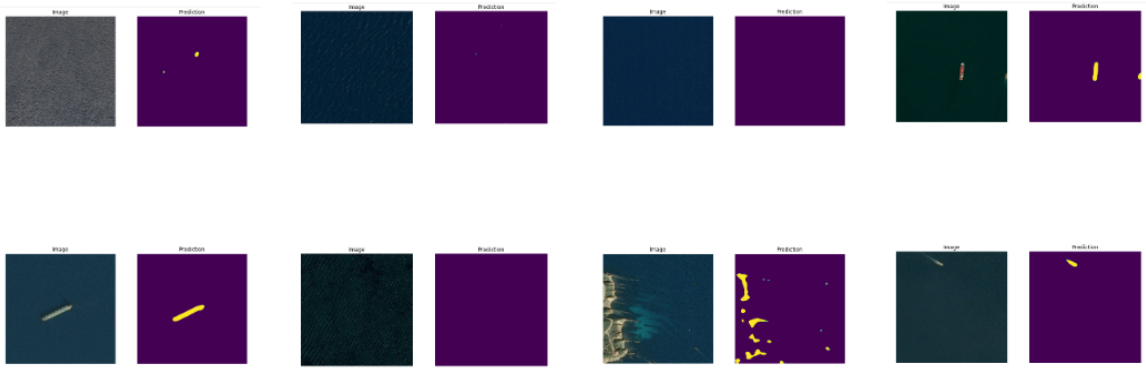

Figure 18: Model predictions on test data

The original model loss (19)(20):

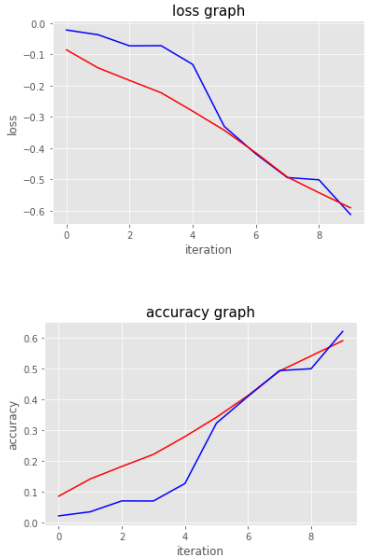

Figure 19: Original Model Metrics

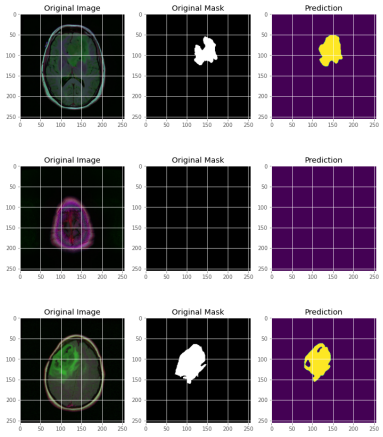

Figure 20: Original Model Prediction

Our model's loss (21)(22):

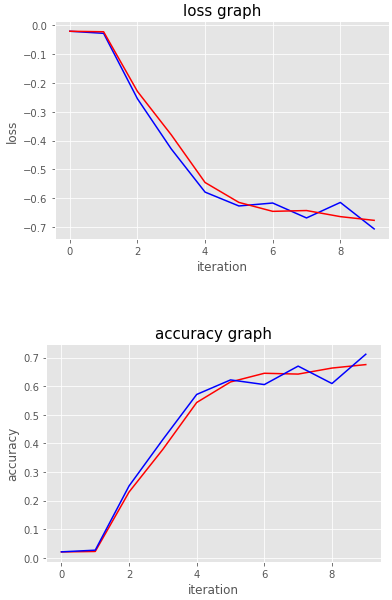

Figure 21: Our Model Metrics

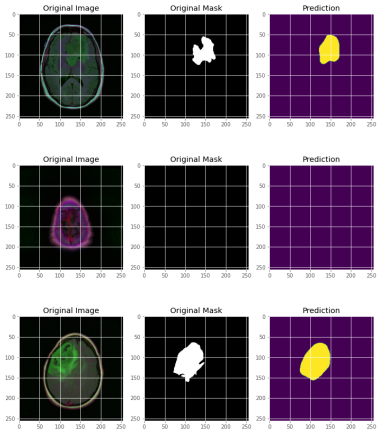

Figure 22: Our Model Prediction

2.3.3 Over-fitting Our Model

We then proceeded to try to over-fit the model as much as possible to check if there are any problems in the data.

Analysis Table		
-	Training IoU Loss	Validation IoU Loss
1000 Training Sample 23	-0.2283	-0.2217
Batch Size 64 24	-0.1874	-0.1953
Batch Size 64 + LR 0.5 25	-0.0065	-0.00575

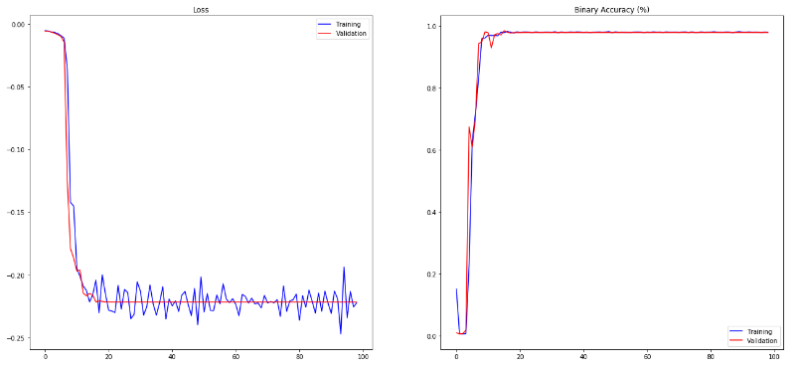

Figure 23: 1000 Training Sample

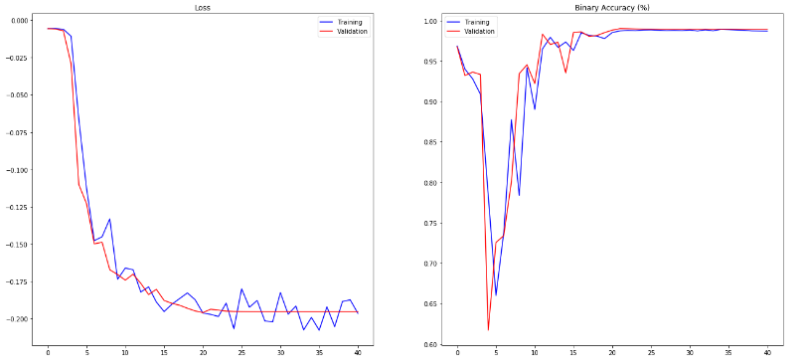

Figure 24: Batch Size 64

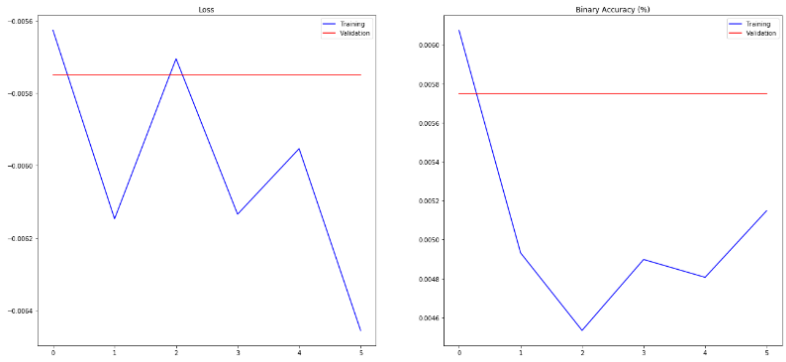

Figure 25: Batch Size 64 + LR 0.5

2.3.4 Improving the pre-trained model

We trained the same pre-trained model 26 27 from last week more than once and it kept giving different results. Training data is not so good so we believe that training on shuffled training set would make the model performance differ every time we train the model

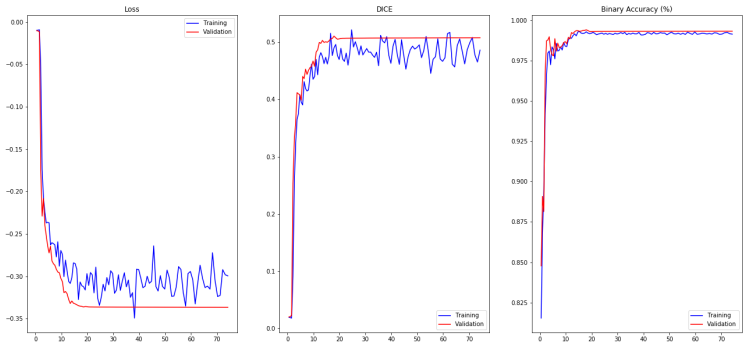

Figure 26: Best Metric results of pre-trained VGG19 U-net model

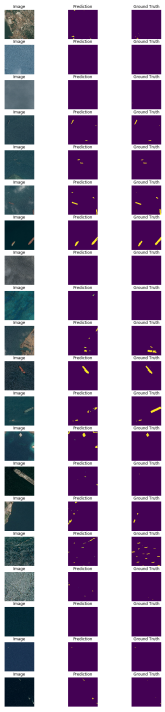

Figure 27: Prediction of best retrained VGG19 U-net model vs Ground Truth

3 Conclusion

It was a long journey and we finally made it.
What we learned from this project:

- Not all data sets are equally clean.
- Segmentation with U-Net Model and learning about other models (FCNN, Segmenter).
- Using transfer learning to train encoder backbones.
- RLE and data pre-processing.

4 Code

- Baseline U-net model
  https://www.kaggle.com/code/baselbyte/mla-project
- U-net with pretrained encoder
  https://www.kaggle.com/code/unstablediffuser/mla-project-pretrained-model

Thank you TAs and Professor Marwan Torki for you efforts and consistent guidance and special thanks to TA. Muhannad for his patience and initiative to always help us.