DeepShipNet: Neural Network Architectures for Detecting Ships in Satellite Images

Albrecht Wigand
University of California, San Diego
albrecht.wigand@tum.de
U08106674

Jeffrey Wang
University of California, San Diego
jewang@eng.ucsd.edu
A11708879

Samuel Thornton
University of California, San Diego
sjthornt@eng.ucsd.edu
A53243732

Siddhant Jain
University of California, San Diego
s9jain@eng.ucsd.edu
A53251799

Yan Sun
University of California, San Diego
yas108@eng.ucsd.edu
A53240727

Abstract

For the purposes of this project, we decided to enter a Kaggle data science competition called the "Airbus Ship Detection Challenge"[1]. The task of this challenge is to create a model that can detect ships in images as quickly and accurately as possible. To accomplish this, we decided to train a neural network based on the U-Net architecture to detect ships in satellite images and draw a segmentation mask representing where in the images the ships appear. With this method, we were able to achieve a competition score of 0.80006, which is 6% below the score of the winning Kaggle entry of 0.85448[2].

1 Introduction and Motivation

While we chose this Kaggle competition solely for the purposes of this project, there is a big motivation for this particular problem to be solved, which is why it is on the Kaggle website as a public data science competition in the first place. The competition was sponsored by the company Airbus as they try to improve their ability to detect ships on the open sea. They claim that since shipping traffic is growing at a rapid pace, there is a greater chance of infractions at sea like environmentally devastating ship accidents, piracy, illegal fishing, drug trafficking, and illegal cargo movement. Because of this, Airbus wants to create a comprehensive maritime monitoring service to help to support the maritime industry to increase knowledge, anticipate threats, trigger alerts, and improve efficiency at sea [1].

The desired outputs of the model are segmentation masks which represent the areas in an image in which one or multiple ships appear. The evaluation metric for this competition is the F2 Score at different Intersection over Union (IoU) thresholds, specifically the mean taken over the individual average F2 Scores of each image in the test data set. There is also an evaluation metric for speed which is just the inference time on over 40 image chips (typical size of a full satellite image). There is a significant incentive for participants in the competition since there is 45,000 dollars in prize money split between the top 3 performers in the accuracy metric and a 15,000 dollar prize for the top performer in the speed metric. We did not enter this competition expecting to win any of the prize

money, but rather to get experience applying the machine leaning topics we have learned in class to a real world problem.

2 Description of Method

For the ship segmentation competition, we knew we would need a network that can preform segmentation label predictions on sets of images, but saw that there are a number of different neural network architectures that are used for this purpose [3]. So we decided to try out a few different models initially and then focus our time in tweaking the one that performed best in the baselines tests. In this section, we will discuss the different models we tried out.

2.1 Architecture

Several different architectures were implemented for the segmentation task in this project. The detailed description is available in the following sections.

2.1.1 U-Net

U-Net is a Convolutional Neural Network architecture that has had a lot of success on bio-medical image segmentation. We are using it for the ship segmentation in the satellite images in this project. Initially, the original U-Net architecture [4] was implemented for the ship segmentation task, as mentioned in publication is shown in Figure 1. It could be observed that the shape of this architecture looks like the letter 'U' which lead it to be named as such. One of this architecture's important features is combining high level features with low level features by down-pooling first and up-pooling later. Finally, 1x1 convolution is utilized for the ultimate prediction of each pixel for segmentation.

Additionally, in order to make images in the training set feed into the model successfully, some parameters were changed compared to original architecture. The detailed configuration is shown in Table 4, which is at the end of this report.

2.1.2 U-Net with ResNet-152 Encoder

Since the U-Net architecture consists of two separable stages, the encoding (down-pooling) and decoding (up-pooling) stages, it lends itself to modifications. For example, we modified a ResNet-152 [5] network to replace the encoding stage of our U-Net.

As outlined in class and in the original paper [5], ResNets allow for deeper models to be trained by utilizing shortcut connections between blocks of layers. This allows each successive block to learn the difference between its input and the underlying distribution (the "residual") rather than trying to learn the entire distribution again.

2.1.3 U-Net with Residual Connections

We also trained a U-Net with a ResNet-style encoder with shortcut connections as described in [6]. This structure was proposed for the sea land segmentation task and we thought would transfer well to our task with some modifications. The input images for the network are from the Kaggle dataset with three channels (RGB) and a CSV file with training data pixel wise (specifying the start and end of a segmentation mask for each line of the image).

This network, like U-Net, is symmetrical. The path on the left side consists of repeated DownBlocks which are connected to the corresponding UpBlocks on the right side. These connections are shown in the figure 2. They are called u-connections since they concatenate the feature maps of the DownBlock to that of the corresponding UpBlock. Besides the u-connection, there is another kind of short connections between the successive DownBlocks. They are shown with purple lines and called the Plus connections or the Plus layer. The Plus layer is an optimized structure. It can solve the problem of the loss error increasing as the network gets deeper. The Plus layer also avoids the training step converge on the local optimal solution and thus guarantees that the deep networks achieves good performance on complex image segmentation tasks.

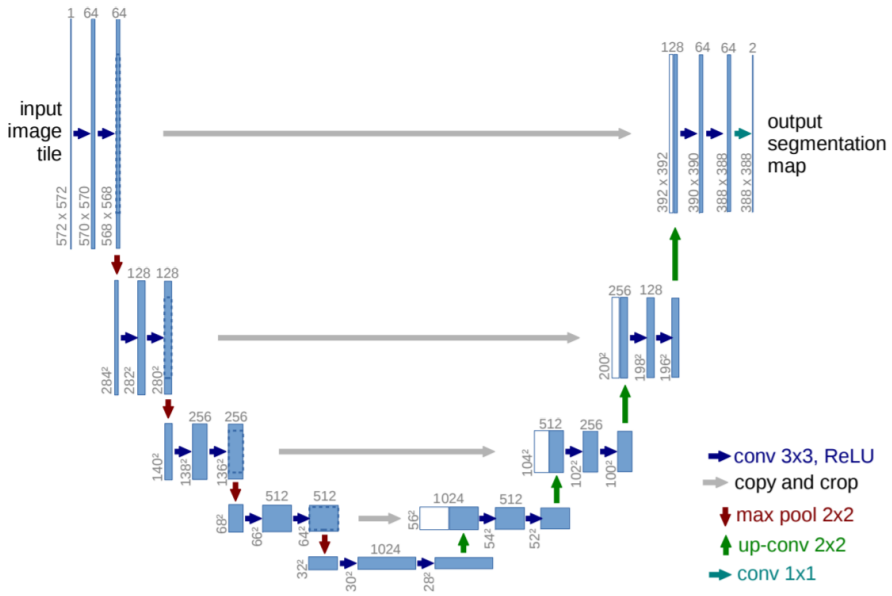

Figure 1: U-Net Original Architecture.

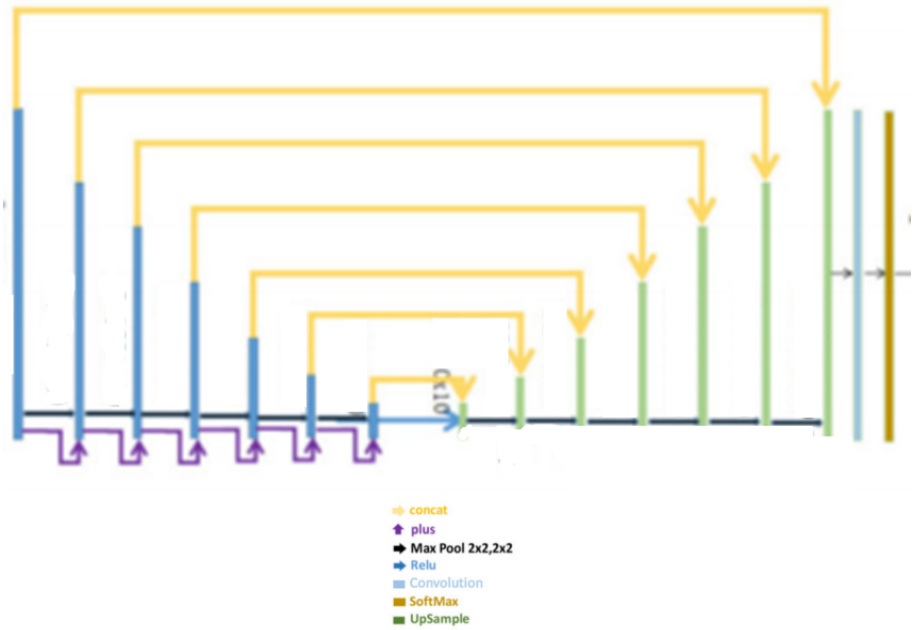

Figure 2: Modified U-Net with Residual Connections.

2.1.4 DenseNet-169

Finally, in an attempt to cut down on false positives, we trained a DenseNet-169 [7] to classify images as "containing ship(s)" or "not containing ship(s)". This would allow us to pass only the images with ships to the segmentation network, which we could then train on data that consisted only of images with ships. This would increase the efficiency of our training process for the segmenting network, allowing us to heavily augment the training data.

The DenseNet is another form of network much like the Residual Connections as described in the previous section, this network also utilizes skip connections or short-cuts to jump over some layers.

As described in our previous section, the residual connections will only skip a single layer; there are variations which have an additional weight matrix to learn the skip weights which are referred to as HighwayNets. In our case the DenseNets have several parallel skip connections as shown in the table in the appendix. These skip connections greatly reduce the problem of vanishing gradients and allowing us to go deeper. Hence we can go deeper with a significantly larger number of layers (169) as compared to standard networks which would easily run into problems like gradient vanishing much earlier. The implemented code was based on the prior work of the Kaggle User Kevin Mader[8].

2.2 Algorithms

We used the Adam optimizer for training our models for this project. Binary entropy loss is utilized to evaluate the segmentation prediction for each pixel in this project. As with most neural networks, the training process was accomplished using stochastic gradient decent.

2.3 Equations

Since the model we ended up settling on is a U-Net model, we will give the equations used in training that type of architecture. The energy function for U-Net is computed by a pixel-wise soft-max over the final feature map combined with the cross entropy loss function:

$$p_k(\mathbf{x}) = \exp(a_k(\mathbf{x}))/\left(\sum_{k'=1}^{K}\exp(a_{k'}(\mathbf{x}))\right)$$

Where $a_k(x)$ denotes the activation in feature channel k at the pixel position $x \in \Omega$ with $\Omega \subset Z^2$. K is the number of classes and $p_k(x)$ is the approximated maximum-function. The cross entropy then penalizes at each position the derivation of $p_{l(x)}(x)$ from 1 using:

$$E = \sum_{\mathbf{x}\in\Omega} w(\mathbf{x})\log(p_{\ell(\mathbf{x})}(\mathbf{x}))$$

Where $l : \Omega \rightarrow \{\ 1, ..., K\ \}$ is the true label of each pixel and $w : \Omega \rightarrow \Re$ is the weight map that was introduced to give some pixels more importance in training.

There is a performance evaluation metric we are using specifically for this project because it is the one required by the competition, which is the F2 Score at different intersection over union (IoU) thresholds. The IoU of a proposed set of object pixels and a set of true object pixels is calculated as the equation:

$$IoU(A, B) = \frac{A \cap B}{A \cup B}$$

The metric sweeps over a range of IoU thresholds, at each point calculating an F2 Score. The threshold values range from 0.5 to 0.95 with a step size of 0.05. In other words, at a threshold of 0.5, a predicted object is considered a "hit" if its intersection over union with a ground truth object is greater than 0.5.

At each threshold value t, the F2 Score value is calculated based on the number of true positives (TP), false negatives (FN), and false positives (FP) resulting from comparing the predicted object to all ground truth objects. The following equation is equivalent to F2 Score when β is set to 2:

$$F_\beta(t) = \frac{(1+\beta^2)\cdot TP(t)}{(1+\beta^2)\cdot TP(t) + \beta^2 \cdot FN(t) + FP(t)}$$

A true positive is counted when a single predicted object matches a ground truth object with an IoU above the threshold. A false positive indicates a predicted object had no associated ground truth

object. A false negative indicates a ground truth object had no associated predicted object. The average F2 Score of a single image is then calculated as the mean of the above F2 Score values at each IoU threshold:

$$\frac{1}{|thresholds|}\sum_{t} F_2(t)$$

Lastly, the score returned by the competition metric is the mean taken over the individual average F2 scores of each image in the test data set [9].

3 Implementation Details

Our networks, mask encoding-decoding, and evaluation metrics were coded in Python 3. We used the Keras (with TensorFlow backend) framework to create and train the networks.

4 Experimental Setting

Once we settled on the modified U-Net architecture, we were then ready to train our network on the full training data set and see how well it performed on the testing data set. In this section, we will discuss the data set itself as well as the environment and parameters of the training.

4.1 Dataset

The Kaggle competition associated with this project provided a data set specifically for this problem, so we did not need to find or create one on our own. The data set contains 15,606 test images (2.17 Gigabyte) and 192,556 training images (27.00 Gigabyte). The images in the data set are mostly of the open ocean, but there are some images that include land as well. One thing to note about the data set is that many of the images contain no ships. The statistics for number of ships in images of the training set can be seen below.

In the training images, the distribution of images that have ships and do not have ships is shown in Figure 3. It could be observed that there are more non-ship images than images that have ships included. Because of this, 100000 non-ship images are removed in consideration of data balance and training efficiency.

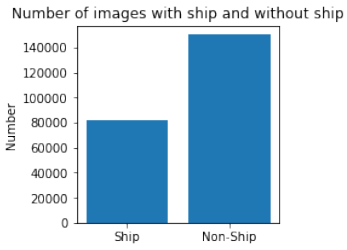

Figure 3: Ship vs Non-Ship Images Counting

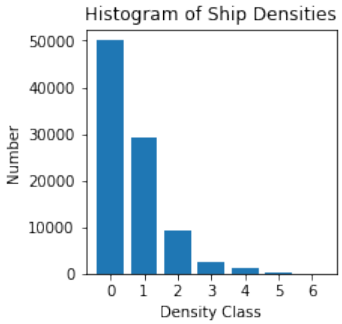

Figure 4: Histogram of Ship Densities

In Figure 4, the percentage of pixels covered by ships is calculated as ship densities, for which the density value is divided into 7 classes. In Figure 4, the higher value of a density class indicates a higher percentage of pixels in the image that are labeled as ships in the training set. As for the data feeding procedure, a specific data pipeline is utilized to make sure that the probability of each data sample being used is equal, including the images that have ships and the images that do not have ships.

4.1.1 Training Parameters

Compared to the architecture mentioned in the original paper, the first several layers of this architecture are adjusted in order to feed images in the given dataset in the correct size. As for the architecture mentioned in Table 4, the total number of parameters is 1954265, including 1950761 trainable parameters and 3504 non-trainable parameters.

4.2 Hardware Used

We originally were training our model on the UCSD Data Science & Machine Learning Platform (DSMLP), but we were running out of GPU memory with this platform and so switched to Amazon Web Services (AWS) using their Elastic Compute Cloud (EC2). We used the p2 instance on the cloud with the xlarge model. This model has a Intel Xeon E5-2686 v4 Processor with a NVIDIA K80 GPU. More information about this model and the other available p2 models can been seen in the figure below (Figure 4.2).

Model	GPUs	vCPU	Mem (GiB)	GPU Memory (GiB)	Network Performance
p2.xlarge	1	4	61	12	High
p2.8xlarge	8	32	488	96	10 Gigabit
p2.16xlarge	16	64	732	192	25 Gigabit

Figure 5: AWS Platform Configuration [10]

5 Results

5.1 Prediction Samples

Several sample prediction figures combined with original images and ground truth labels are shown in Figure 6. The images in the left column are the original images from the dataset, the images in the middle are the ground truth labels provided in the training set, and images on the right side are the predictions made by the trained model. The analysis of the results is available in Section 6.

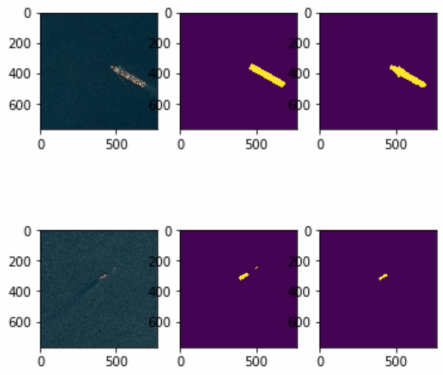

Figure 6: U-net Successful Prediction Samples

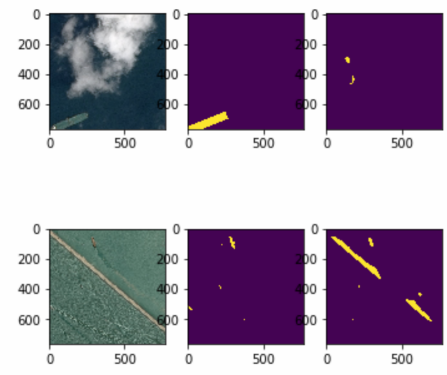

Figure 7: U-Net Bad prediction Samples

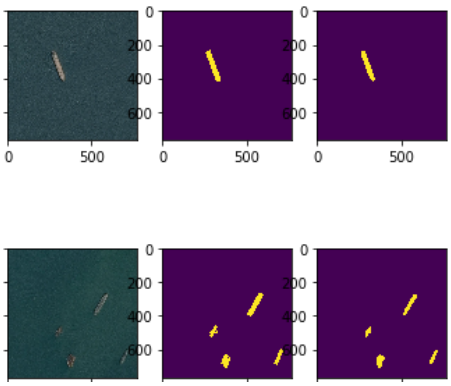

Figure 8: U-Net with Res Connect Good prediction Samples

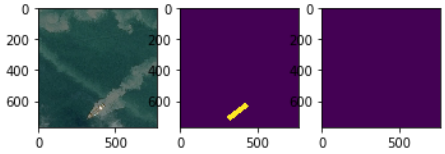

Figure 9: U-Net with Res Connect Bad prediction Samples

5.2 Model Comparison

After tuning hyperparameters and selecting the optimal configuration for each model, comparisons can be made for these models. For this, all the models are reset to train from the beginning initialization and the training time is fixed at 3 hours. After being trained for the same length of time, the optimal decision threshold is searched for each model individually and then the prediction result for testing set is submitted to Kaggle for the evaluation. The evaluation result is shown in Table 1.

Model	Public F2 Score	Private F2 Score
U-Net	0.60874	0.79714
Res-U-Net	0.56901	0.75760

Table 1: Model Comparison for 3-hour Training Test

5.3 Ultimate Performance

In order to make the ultimate segmentation performance as good as it can be, larger batch sizes and longer training times are applied to the training process. After training for 5 hours with batch size 16 (selected under the memory limitation), the ultimate (best) performance is in Table 2.

Model	Public F2 Score	Private F2 Score
U-Net	0.60883	0.80006

Table 2: Ultimate Best Performance

5.4 Ship/No-ship Classifier Accuracy

We took the training data and applied a 70-30 training and validation split and implemented Data Augmentation. Although we were not able to train for long, we obtained training and validation accuracy shown in Table 3. This network was trained separately and has to be seen in combination with a training of the U-Net with a training set that excludes images without ships. As these two networks were not combined and published to Kaggle, we did not receive F2 Scores for this approach.

Model	Training Acc.	Validation Acc.
DenseNet-169	0.81	0.76

Table 3: Classifier Accuracy

6 Discussion

As the data shows, we obtained fairly good results with a vanilla U-Net, even though we tried deeper, and more complicated architectures. This speaks perhaps about the size of the dataset we were given, and the limited computing resources at our disposal, but it also speaks for the efficacy of the base U-Net architecture, which is known to be good for limited or small datasets. The results obtained have both good successful prediction and bad prediction results as shown in Figure 6. For the good prediction results, the model made the correct segmentation result that finds most of the pixels for a ship correctly. For the bad prediction results in Figure 7, it is obvious that the classifier is misled by the noise in given image such as the cloud or the bridge captured in the images.

When we tried to make our U-Net deeper by adding a ResNet-152 as the encoder, we were unable to obtain good results. The training never really converged, probably because it was too large (60M parameters compared to the 2M of our original U-Net) to train with the amount of data we had (100K images). Another smaller factor could be how small our batch sizes were (we were only able to fit 2 images in memory), and thus we weren't able to finish training before our resources timed out.

We also tried a U-Net with Residual Connections. The results are in the section above, but surprisingly it performed worse than our baseline U-Net despite having more parameters (about 600K more). Thus, at least for this particular project/data, it seems like ResNet-style encoders are not helpful.

As part of our training we needed to decide on a threshold value above which we would consider a pixel as part of the segmentation mask or not. At first using the arbitrary value of 0.5 lead to poor results as large parts of the masks did not represent any ships; the segmentation was very 'loose' which would give a sub optimal score in the IoU metric (as specified above) that was used for the competition. So to find the optimal threshold criteria we split our training data into a training set and a validation set along the ratio of 99:1 after removing the excess non ship images which meant we had 131,723 training images and around 10,000 validation images from which roughly 463 images were evaluated for a range of thresholds between 0.2-0.9 . In most cases we found the best performing threshold to be between 0.6-0.75 .

The model that obtained the best performance on the Kaggle private leaderboard utilized a larger batch size (16 images) and early stopping, which seemed to help. If we had more resources, we would definitely have tried larger batch sizes, which would probably have helped the training converge faster. The reason why we identify the "best" model based on the private and not the public leaderboard is because the private leaderboard utilizes 88% of the test data whereas the public leaderboard only utilizes 12%.

Another interesting thing to note is that the training data was not perfectly labeled. In Figure 6, we can see that there's actually a ship in the left portion of the image, which was not labeled. Our U-Net correctly identifies and masks the ship, although it does miss the much smaller boat in the middle top of the image.

If we had more time, we would definitely try the deep ResNet encoder architecture (perhaps with a pretrained network) in our U-Net again, and use data augmentation to increase the dataset size to allow for the training to converge more easily.

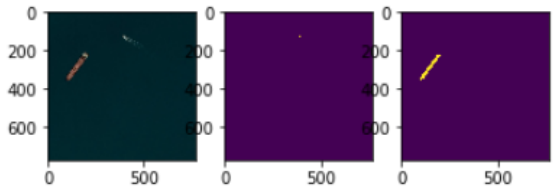

Figure 10: Training Sample (Left), Label (Middle), Model Prediction (Right)

6.1 Conclusion

U-Net and Res-U-Net architectures were implemented for the ship segmentation challenge on Kaggle. Optimal thresholds were determined by scoring the validation set for the inference process. Both good prediction results and bad prediction results occurred for the trained model, during which the bad ones could be misled by the noise inside images. The ultimate best performance of trained model for the test set on Kaggle is 0.60883 F2 score for public leaderboard and 0.80006 F2 score for private leaderboard.

6.2 Future Work

We are proud of what we were able to accomplish for this project in the time allotted, but there is more that can be done to improve our solution. The Kaggle competition that this project is based on ended about a month before this report was due so we were able to see what some of the top performing groups did in their solutions. One thing that many of these groups did was use a stacked network, where the first part of the network is for classification for weather the image has ship(s) in it or not followed by a segmentation network. The segmentation network seems to produce a large number of false positives so by having a classifier before so that we are feeding in all or mostly all images that actually contain ships and assigning an empty segmentation mask for the images classified to not have ships should greatly increase the performance. We trained a DenseNet network to help us do the ship/no-ship classification, but didn't have the time to integrate it with our segmentation network.

Additionally, performing data augmentation to increase the size of the training set could be beneficial as well. We also had some concerns with our process possibly over fitting to the training data so splitting the training data into a validation set to check for early stopping could help as well.

Another area to consider would be transfer learning. Since we ran into issues training a deep ResNet-encoder, we would look into using a ResNet pre-trained on ImageNet [11] as the encoder, perhaps even freezing some earlier layers if it helped training converge. Another (and perhaps more novel) way of doing transfer learning would be by applying image augmentations to our training set with augmentation policies obtained using AutoAugment [12] on ImageNet.

Finally, because we're trying these different architectures, we could always ensemble the networks together to try and obtain a higher score.

References

[1] Kaggle Inc. *Airbus Ship Detection Challenge - Description*. 2018. URL: https://www.kaggle.com/c/airbus-ship-detection#description (visited on 12/12/2018).

[2] Kaggle. *Airbus Ship Detection Challenge: Leaderboard*. 2018. URL: https://www.kaggle.com/c/airbus-ship-detection/leaderboard (visited on 12/14/2018).

[3] Kaggle. *U-Net Starter Trial*. https://www.kaggle.com/kotarojp/first-step-for-submission-u-net-tta/. [Kaggle Competition]. 2018.

[4] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation". In: *CoRR* abs/1505.04597 (2015). arXiv: 1505.04597. URL: http://arxiv.org/abs/1505.04597.

[5] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[6] R. Li et al. "DeepUNet: A Deep Fully Convolutional Network for Pixel-Level Sea-Land Segmentation". In: *JSTARS* 10.1109/JSTARS.2018.2833382 (2018). URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8370071&isnumber=8556520.

[7] Gao Huang et al. "Densely Connected Convolutional Networks". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2017), pp. 2261–2269.

[8] Kevin Mader. *Transfer Learning for Boat or No-Boat*. 2018. URL: https://www.kaggle.com/kmader/transfer-learning-for-boat-or-no-boat (visited on 12/10/2018).

[9] Kaggle Inc. *Airbus Ship Detection Challenge - Evaluation*. 2018. URL: https://www.kaggle.com/c/airbus-ship-detection#evaluation (visited on 12/12/2018).

[10] Amazon Web Services Inc. *Amazon EC2 Instance Types*. 2018. URL: https://aws.amazon.com/ec2/instance-types/ (visited on 12/12/2018).

[11] J. Deng et al. "ImageNet: A large-scale hierarchical image database". In: (June 2009), pp. 248–255. ISSN: 1063-6919. DOI: 10.1109/CVPR.2009.5206848.

[12] Ekin Dogus Cubuk et al. "AutoAugment: Learning Augmentation Policies from Data". In: *CoRR* abs/1805.09501 (2018). arXiv: 1805.09501. URL: http://arxiv.org/abs/1805.09501.

7 Appendix

7.1 Git Repository: Readme

DeepShipNet: Exploring neural network architectures for locating ships in images

This code is part of the final project of the University of California San Diego Fall Semester 2018 course *Machine Learning for Image Processing*. During the project, the team participated in the Kaggle Competition *Airbus Ship Detection*[1] using the same code.
Read the following instructions if you would like to run the code yourself.

Executable Files

For a quick demo, run the Jupyter Notebook `Demo.ipynb` .
For a full training on the dataset, run the Jupyter Notebook `train.ipynb` .

Requirements

The dataset can be downloaded on the kaggle competition website[1].
In the notebooks, the following locations are referenced.
Adapt the code or make sure to recreate the data structure:
csv data for labels: `/datasets/ee285f-public/airbus_ship_detection/`
training data: `/datasets/ee285f-public/airbus_ship_detection/train_v2/`
test data: `/datasets/ee285f-public/airbus_ship_detection/test_v2/`

To run the notebook, you need the following packages installed:

- tensorflow
- keras
- matplotlib
- pandas
- scikit.image
- scikit.learn

ECE 285 Group Project Members

Albrecht Wigand
University of California San Diego
albrecht.wigand@tum.de
U08106674

Jeffrey Wang
University of California San Diego
jewang@eng.ucsd.edu
A11708879

Samuel Thornton
University of California San Diego
sjthornt@eng.ucsd.edu
A53243732

Siddhant Jain
University of California San Diego
sojain@ucsd.edu
A53251799

Yan Sun
University of California San Diego
yas108@eng.ucsd.edu
A53240727

References

[1] - https://www.kaggle.com/c/airbus-ship-detection

7.2 Code

The repository for this project can be found on GitHub following this link: `https://github.com/jsiddhant/ECE285-DeepShipNet`

7.3 U-Net Architecture

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 768, 768, 3)	0	
conv2d_1 (Conv2D)	(None, 768, 768, 8)	224	input_1[0][0]
batch_normalization_1	(None, 768, 768, 8)	32	conv2d_1[0][0]
conv2d_2 (Conv2D)	(None, 768, 768, 8)	584	batch_normalization_1[0][0]
batch_normalization_2	(None, 768, 768, 8)	32	conv2d_2[0][0]
average_pooling2d_1	(None, 128, 128, 8)	0	batch_normalization_2[0][0]
conv2d_3 (Conv2D)	(None, 128, 128, 16)	0	average_pooling2d_1[0][0]
batch_normalization_3	(None, 128, 128, 16)	64	conv2d_3[0][0]
conv2d_4 (Conv2D)	(None, 128, 128, 16)	2320	batch_normalization_3[0][0]
batch_normalization_4	(None, 128, 128, 16)	64	conv2d_4[0][0]
dropout_1 (Dropout)	(None, 128, 128, 16)	0	batch_normalization_4[0][0]
max_pooling2d_1 (MaxPooling2D)	(None, 64, 64, 16)	0	dropout_1[0][0]
conv2d_5 (Conv2D)	(None, 64, 64, 32)	4640	max_pooling2d_1[0][0]
batch_normalization_5	(None, 64, 64, 32)	128	conv2d_5[0][0]
conv2d_6 (Conv2D)	(None, 64, 64, 32)	9248	batch_normalization_5[0][0]
batch_normalization_6	(None, 64, 64, 32)	128	conv2d_6[0][0]
dropout_2 (Dropout)	(None, 64, 64, 32)	0	batch_normalization_6[0][0]
max_pooling2d_2 (MaxPooling2D)	(None, 32, 32, 32)	0	dropout_2[0][0]
conv2d_7 (Conv2D)	(None, 32, 32, 64)	18496	max_pooling2d_2[0][0]
batch_normalization_7	(None, 32, 32, 64)	256	conv2d_7[0][0]
conv2d_8 (Conv2D)	(None, 32, 32, 64)	36928	batch_normalization_7[0][0]
batch_normalization_8	(None, 32, 32, 64)	256	conv2d_8[0][0]
dropout_3 (Dropout)	(None, 32, 32, 64)	0	batch_normalization_8[0][0]
max_pooling2d_3 (MaxPooling2D)	(None, 16, 16, 64)	0	dropout_3[0][0]
conv2d_9 (Conv2D)	(None, 16, 16, 128)	73856	max_pooling2d_3[0][0]
batch_normalization_9	(None, 16, 16, 128)	512	conv2d_9[0][0]
conv2d_10 (Conv2D)	(None, 16, 16, 128)	147584	batch_normalization_9[0][0]
batch_normalization_10	(None, 16, 16, 128)	512	conv2d_10[0][0]
dropout_4 (Dropout)	(None, 16, 16, 128)	0	batch_normalization_10[0][0]
max_pooling2d_4 (MaxPooling2D)	(None, 8, 8, 128)	0	dropout_4[0][0]
conv2d_11 (Conv2D)	(None, 8, 8, 256)	295168	max_pooling2d_4[0][0]
batch_normalization_11	(None, 8, 8, 256)	1024	conv2d_11[0][0]
conv2d_12 (Conv2D)	(None, 8, 8, 256)	590080	batch_normalization_11[0][0]
batch_normalization_12	(None, 8, 8, 256)	1024	conv2d_12[0][0]
up_sampling2d_1 (UpSampling2D)	(None, 16, 16, 256)	0	batch_normalization_12[0][0]
conv2d_13 (Conv2D)	(None, 16, 16, 128)	131200	up_sampling2d_1[0][0]
batch_normalization_13	(None, 16, 16, 128)	512	conv2d_13[0][0]
concatenate_1 (Concatenate)	(None, 16, 16, 256)	0	dropout_4[0][0] batch_normalization_13[0][0]
conv2d_14 (Conv2D)	(None, 16, 16, 128)	295040	concatenate_1[0][0]
batch_normalization_14	(None, 16, 16, 128)	512	conv2d_14[0][0]
conv2d_15 (Conv2D)	(None, 16, 16, 128)	147584	batch_normalization_14[0][0]
batch_normalization_15	(None, 16, 16, 128)	512	conv2d_15[0][0]
up_sampling2d_2 (UpSampling2D)	(None, 32, 32, 128)	0	batch_normalization_15[0][0]
conv2d_16 (Conv2D)	(None, 32, 32, 64)	32832	up_sampling2d_2[0][0]
batch_normalization_16	(None, 32, 32, 64)	256	conv2d_16[0][0]
concatenate_2 (Concatenate)	(None, 32, 32, 128)	0	dropout_3[0][0] batch_normalization_16[0][0]
conv2d_17 (Conv2D)	(None, 32, 32, 64)	73792	concatenate_2[0][0]

Table 4 continued from previous page

batch_normalization_17	(None, 32, 32, 64)	256	conv2d_17[0][0]
conv2d_18 (Conv2D)	(None, 32, 32, 64)	36928	batch_normalization_17[0][0]
batch_normalization_18	(None, 32, 32, 64)	256	conv2d_18[0][0]
up_sampling2d_3 (UpSampling2D)	(None, 64, 64, 64)	0	batch_normalization_18[0][0]
conv2d_19 (Conv2D)	(None, 64, 64, 32)	8224	up_sampling2d_3[0][0]
batch_normalization_19	(None, 64, 64, 32)	128	conv2d_19[0][0]
concatenate_3 (Concatenate)	(None, 64, 64, 64)	0	dropout_2[0][0] batch_normalization_19[0][0]
conv2d_20 (Conv2D)	(None, 64, 64, 32)	18464	concatenate_3[0][0]
batch_normalization_20	(None, 64, 64, 32)	128	conv2d_20[0][0]
conv2d_21 (Conv2D)	(None, 64, 64, 32)	9248	batch_normalization_20[0][0]
batch_normalization_21	(None, 64, 64, 32)	128	conv2d_21[0][0]
up_sampling2d_4 (UpSampling2D)	(None, 128, 128, 32)	0	batch_normalization_21[0][0]
conv2d_22 (Conv2D)	(None, 128, 128, 16)	2064	up_sampling2d_4[0][0]
batch_normalization_22	(None, 128, 128, 16)	64	conv2d_22[0][0]
concatenate_4 (Concatenate)	(None, 128, 128, 32)	0	dropout_1[0][0] batch_normalization_22[0][0]
conv2d_23 (Conv2D)	(None, 128, 128, 16)	4624	concatenate_4[0][0]
batch_normalization_23	(None, 128, 128, 16)	64	conv2d_23[0][0]
conv2d_24 (Conv2D)	(None, 128, 128, 16)	2320	batch_normalization_23[0][0]
batch_normalization_24	(None, 128, 128, 16)	64	conv2d_24[0][0]
up_sampling2d_5 (UpSampling2D)	(None, 768, 768, 16)	0	batch_normalization_24[0][0]
concatenate_5 (Concatenate)	(None, 768, 768, 24)	0	up_sampling2d_5[0][0] batch_normalization_25[0][0]
conv2d_25 (Conv2D)	(None, 768, 768, 16)	3472	concatenate_5[0][0]
batch_normalization_25	(None, 768, 768, 16)	64	conv2d_25[0][0]
conv2d_26 (Conv2D)	(None, 768, 768, 8)	1160	batch_normalization_25[0][0]
batch_normalization_26	(None, 768, 768, 8)	32	conv2d_26[0][0]
conv2d_27 (Conv2D)	(None, 768, 768, 1)	9	batch_normalization_26[0][0]

Table 4: U-Net Architecture

Table 5: Classifier Network Architecture

Layer (type)	Output Shape	Param #
Image_RGB_In (InputLayer)	(None, 299, 299, 3)	0
gaussian_noise_2 (GaussianNo	(None, 299, 299, 3)	0
densenet169 (Model)	(None, 9, 9, 1664)	12642880
batch_normalization_2 (Batch	(None, 9, 9, 1664)	6656
spatial_dropout2d_2 (Spatial	(None, 9, 9, 1664)	0
global_max_pooling2d_2 (Glob	(None, 1664)	0
dense_3 (Dense)	(None, 128)	213120
dropout_2 (Dropout)	(None, 128)	0
dense_4 (Dense)	(None, 1)	129