# Machine Learning Engineer Nanodegree

## Capstone Project: Icebergs vs Ships

Antoine Diffloth
November 6, 2019

## Part 1: Definition

### Project Overview

The objective of my capstone project is to beat the median public leaderboard score for the Statoil/C-CORE Iceberg Classifier Challenge on Kaggle[1].

The shipping industry is a key enabler of the modern global economy. It moved 10.7 billion tons of goods in 2017 and is expected to grow 3.8% annually between 2018 and 2023[2].

In some parts of the world's oceans, icebergs can disrupt commercial vessels by blocking shipping lanes and threatening physical damage to ships. Beyond the obvious collision risk, icebergs can cause delays and reroutes which are very costly to the shipping operators. As the climate warms and arctic ice shrinks, new trade routes will open up[3], but the risk of drifting icebergs will increase. Shipping companies will need better ways of tracking these building-sized chunks of ice.

The shipping industry uses various methods to detect and track icebergs, including airplane-borne and shore-based monitoring. Some parts of the ocean, however, are too distant from land to allow this, and satellite-based monitoring is the only option. The data generated from these satellites is not as easy to interpret as the optical images we're used to seeing but could prove very valuable in detecting problematic icebergs.

As a parent of young children, I feel a personal motivation to leave the planet to future generations in as good a condition as we found it, if not better. I believe that all research, no matter how small, that improves our understanding and raises awareness of climate related topics will improve the odds that we don't ruin this

[1] www.kaggle.com/c/statoil-iceberg-classifier-challenge/overview
[2] United Nations Conference on Trade and Development, *Review of Maritime Transport 2018,* unctad.org/en/PublicationsLibrary/rmt2018_en.pdf
[3] Patel, J. (May 3, 2017) *As Arctic Ice Vanishes, New Shipping Routes Open,* www.nytimes.com/interactive/2017/05/03/science/earth/arctic-shipping.html

planet for our children and our children's children. Although this Kaggle challenge is not directly related to climate change, it is related to our ability to measure changes to our planet from space, particularly related to icebergs and the warming planet.

## Problem Statement

This challenge is a binary classification problem which consists of identifying whether a given satellite image contains an iceberg or a ship. We will be presented with a collection of images that are labeled as containing an iceberg or not containing an iceberg. We will train a machine learning model on this data, then use the trained model to classify a different, unlabeled collection of images based on whether we think it contains an iceberg.

If this approach proves successful, shipping companies could more easily protect their fleet from the threat of drifting icebergs. Scientists may also use similar approaches to measure the breakup of polar icecaps due to climate change.

## Datasets and Inputs

The data sets are provided by Kaggle and can be downloaded here: https://www.kaggle.com/c/7380/download-all.

The data consists of 75x75 pixel images of the radar backscatter patterns from the Sentinel-1 satellite. There are two bands of data provided, one for the horizontal polarization and one for the vertical polarization. In addition to the backscatter data, the angle of incidence of the radar beam to the object is provided.

There are 1604 images in the training data set and 8424 images in the test data set. The data is provided in json format.

The training data has an additional feature "is_iceberg" which is 0 if the image is a ship and 1 if it is an iceberg. This is the dependent (target) variable of the study.

## Solution Statement

The solution I'm proposing is to train a convolutional neural network to learn the difference between ships and icebergs. CNNs are a commonly used in computer vision and image classification tasks. The NN basis of CNNs allow them to model highly complex and non-linear relationships. The convolutions allow the model to look at the data surrounding each pixel, which enables the identification of more and more complex structures within the image.

Although we're dealing with radar backscatter images rather than visible light images, CNNs seem like they are still an appropriate solution. There may be some additional processing required to adapt radar data to optical image manipulation tools.

## Evaluation Metrics

In order to compare to the Kaggle leaderboard, I will use log loss as the primary evaluation metric.

Since this is a classification problem, I will also calculate precision, recall and F1 score in order to understand the kinds of errors that my model is making. I think looking at recall is more relevant than precision in this case since correctly identifying all the icebergs (avoiding false negatives) in the test set is a more important goal than incorrectly identifying a ship as an iceberg. In other words, we will seek to minimize Type 2 errors.

# Part 2: Analysis

## Data exploration

The data is presented in two json files, one for training and one for test. The training data set has 1604 rows and test has 8424 rows. Each row of data has:

- **id**: a unique id for this row
- **band_1**: a 5625-element list of floats, representing the 75 by 75-pixel image of the horizontal polarization
- **band_2**: same as band_1, but for the vertical polarization
- **inc_angle**: a float that represents the incidence angle of the radar beam to the target for this image
- **is_iceberg**: Boolean that indicates whether this is an iceberg (1) or a ship (0), only exists in the test data set

**Missing data:** The only feature that has any missing data is inc_angle in the training data set. Of 1604 records in train, 133 have missing inc_angle data. I didn't want to drop these records since the train set is already pretty small. The alternative would have been to impute this data, but the way I ended up using inc_angle did not require it. More on this topic later.

**Class balance:** In the training data set, 47% of the observations are icebergs. This is close enough to half that I won't take any measures to rebalance the classes.

**Inc_angle distribution:** Initial examination of the distribution of inc_angle values did not reveal anything of interest. In fact, I did not even include this data in my early modeling. After hitting a wall with tuning of the CNN, I read a Kaggle forum post from the first-place team[4]. They indicated that there were some strong patterns in the

[4] https://www.kaggle.com/c/statoil-iceberg-classifier-challenge/discussion/48241

distribution of inc_angle if one used very small bins. I ended up incorporating some of this insight into my final model. See the Refinement section for further discussion.

## Exploratory visualization

Figure 1 is a visualization of an example image, in this case an iceberg. The iceberg appears as the area in the middle of the plot with the higher values. The surrounding area is open ocean and is very noisy.

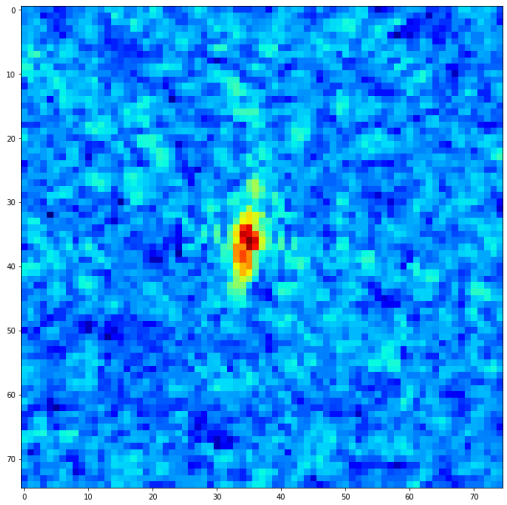



Figure 1: Visualization of band_1 data of a sample iceberg image.

I happen to be an avid amateur photographer, so seeing an image like this makes me immediately want to apply a de-noising filter. The hope is that clearing up the noisy background might help the ML algorithm detect and classify the image more consistently.

Figure 2 shows a few different approaches to reducing noise in the image. The image in the upper left is the original. The image in the lower right has the smoothest open ocean but loses a lot of detail in the actual object.

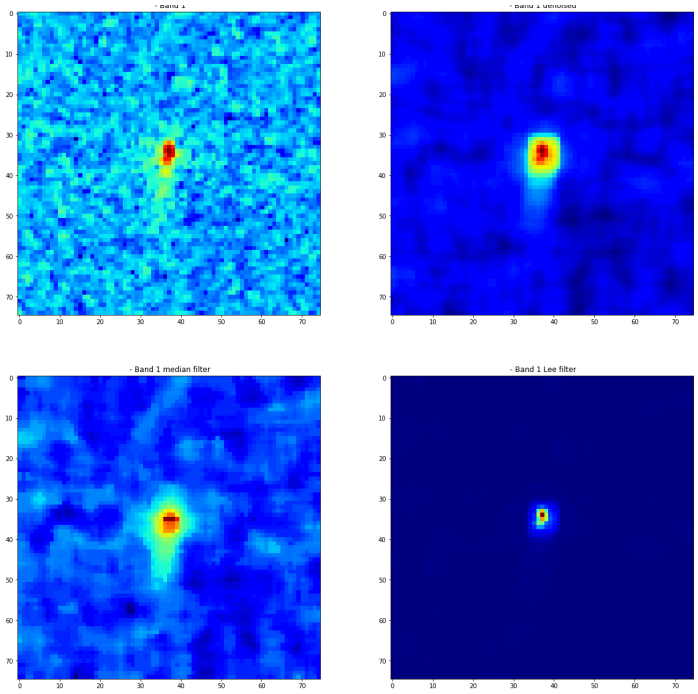



Figure 2: Different denoising techniques.

Figure 3 shows a 3D projection of the same image. Again, notice the amount of noise in the open ocean.

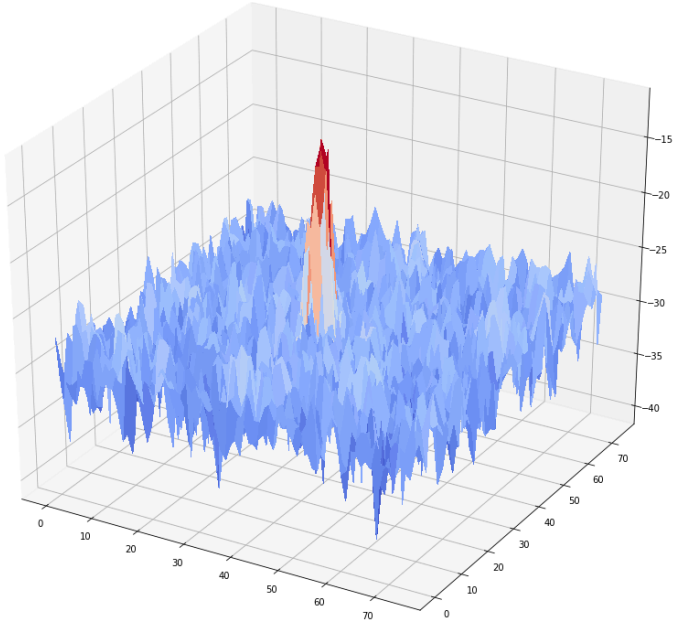



Figure 3: Iceberg image, 3D projection

## Algorithms and techniques

I chose **convolutional neural networks** as the technique to solve this problem. CNNs are the standard choice for use in image classification problems. In a CNN, a filter is passed over an image like a sliding window. The weights in the filter interact with the pixels in the image, creating a new set of outputs. These outputs, or activation maps, start to detect features in an image such as vertical lines, circles, etc. The hierarchical nature of the convolutional layers allow them to detect successively larger and more complex features, eventually combining to identify entire objects in an image, such as a cat, a dog, or in this case an iceberg.

I chose the **Keras** frontend package, with a **Tensorflow** backend, which is the default starting with TF version 2.0. Keras is very well documented and has a large support community, which makes it very easy to use and troubleshoot.

A popular technique that is employed with CNNs is **transfer learning**. This consists of taking a neural network that has a successful architecture and has already been trained on a large set of images, freezing most of the weights in the network, replacing the last few layers of the model and training the combined model on the new data. The pre-trained layers act as feature detectors which feed into a custom classifier. Since only the classifier needs to be trained, the number of parameters is much lower, and the training time is greatly reduced. For this project, I tried the VGG16 and ResNet50 pre-trained models.

To help with evaluating the performance of my model I used **TensorBoard**, which is a visualization toolkit for TensorFlow. The feature I found most useful was the ability to log and plot the accuracy and loss scores at each epoch of each training run. This made it very easy to see if/when a model started to overfit and compare one run to another to see the effects of hyperparameter tuning.

Since there are only 1604 rows in the training data set, I used **data augmentation** to create more training data for my algorithms. The idea behind data augmentation of images is an image of a particular object might be taken from different perspectives, angles and levels of zoom, but are still valid pictures of that object and should be classified as such. Data augmentation takes a given set of labeled images and applies transformations to those images before they are fed to the neural network. While the transformations must be constrained to certain bounds, randomly varying the transformations can generate a theoretically infinite training set.

Another technique I employed was **early stopping**, whereby the validation metric is monitored from epoch to epoch and training is stopped if the metric fails to improve after a given number of epochs. The best model is saved after each epoch and this is the model that is ultimately used. By monitoring the validation metric rather than the training metric, early stopping can prevent over training.

I used **k-fold cross validation** to evaluate my model against different splits of the data. This technique provides a measure of a model's ability to generalize, without "wasting" any training data or introducing data leakage problems.

## Benchmark Model

I will use the median score of the Kaggle public leaderboard as my benchmark. The Kaggle community has many highly skilled data scientists, but if I aspire to join their ranks, I think beating half of them is a reasonable goal. At the time of this writing, there were 3339 teams on the leaderboard and the 1670$^{th}$ team has a **log loss score of 0.19387**. This is the score I will aim to beat.

# Part 3: Methodology

## Data preprocessing

Many ML models perform better when trained on scaled or normalized data. I don't think the radar data fits a normal distribution, so I chose to run the band_1 and band_2 data through sklearn's **MinMaxScaler**. This scaler linearly remaps all the values in an array to the range (0, 1).

In order to remove the noise in the images, I tried applying **median and Lee filters** to the images before feeding them into the data augmentation generator. For the median filter, I used scipy's median_filter with a window size of 6x6. For the Lee filter, I started with pyradar's lee module[5], but ended up using an implementation by stackoverflow user Alex I[6].

To increase the amount of training data available, I implemented **data augmentation** through an ImageDataGenerator with the following parameters:

- horizontal_flip=True
- vertical_flip=True
- width_shift=0.1
- height_shift=0.1
- zoom_range=0.1
- rotation_range=0.2

These seemed like appropriate transformations for an image taken from above that doesn't really have a sense of up vs down, and delivered significant gains in model performance.

---

[5] https://pyradar-tools.readthedocs.io/en/latest/api/pyradar.filters.html
[6] https://stackoverflow.com/questions/39785970/speckle-lee-filter-in-python

## Implementation

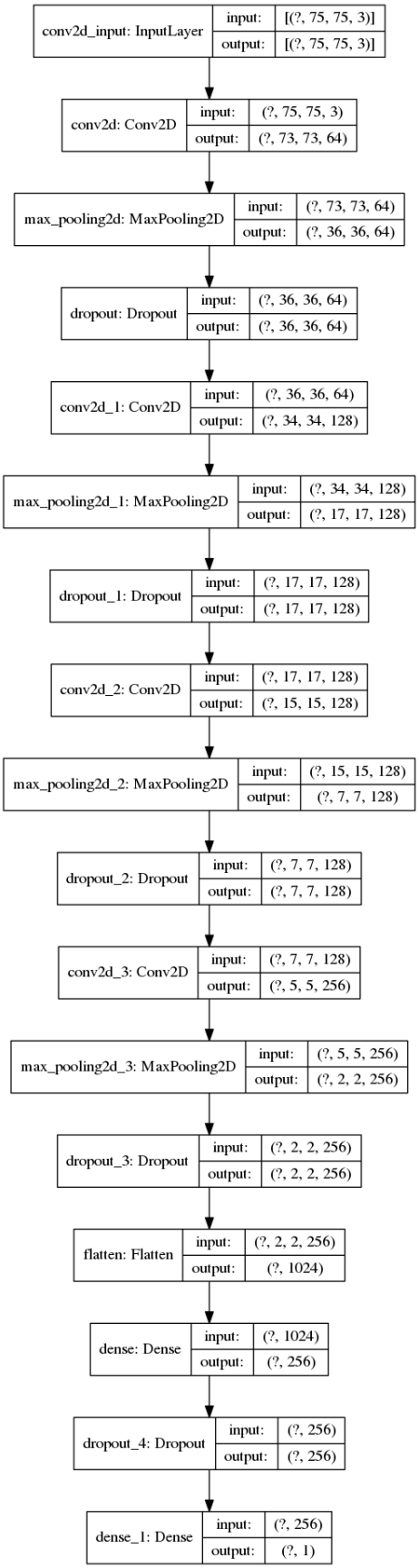
The first step after loading and scaling the data was to create **4-dimensional arrays** of images to feed into the CNN. The second and third dimensions of the array are the x/y grid of the image itself. The fourth dimension of the array is the three bands of data. And the first dimension is simply each observation in the training or test data.

This 4D array of data was then fed into a CNN. The **architecture of the CNN** was a combination of architectures I've used on other projects and the architecture from a public kernel on Kaggle.[7] Some highlights of the architecture:

- Uses 4 blocks of [convolutional layer, pooling layer, dropout]
- Steadily increasing number of filters, starting with 64 and working up to 256
- Kernel size of (3,3) for the Conv2D layers
- Pool size of (2,2) and stride of (2,2) for the MaxPooling2D layers
- Flatten the output and feed it into a 256 node fully connected layer
- Dropout of 0.2 after each MaxPool layer and the final FC layer
- Finally, output to a single neuron with sigmoid activation

See below for a diagram of the neural network.

[7] https://www.kaggle.com/henokanh/cnn-batchnormalization-0-1646


conv2d_input: InputLayer
input: [(?, 75, 75, 3)]
output: [(?, 75, 75, 3)]
conv2d: Conv2D
input: (?, 75, 75, 3)
output: (?, 73, 73, 64)
max_pooling2d: MaxPooling2D
input: (?, 73, 73, 64)
output: (?, 36, 36, 64)
dropout: Dropout
input: (?, 36, 36, 64)
output: (?, 36, 36, 64)
conv2d_1: Conv2D
input: (?, 36, 36, 64)
output: (?, 34, 34, 128)
max_pooling2d_1: MaxPooling2D
input: (?, 34, 34, 128)
output: (?, 17, 17, 128)
dropout_1: Dropout
input: (?, 17, 17, 128)
output: (?, 17, 17, 128)
conv2d_2: Conv2D
input: (?, 17, 17, 128)
output: (?, 15, 15, 128)
max_pooling2d_2: MaxPooling2D
input: (?, 15, 15, 128)
output: (?, 7, 7, 128)
dropout_2: Dropout
input: (?, 7, 7, 128)
output: (?, 7, 7, 128)
conv2d_3: Conv2D
input: (?, 7, 7, 128)
output: (?, 5, 5, 256)
max_pooling2d_3: MaxPooling2D
input: (?, 5, 5, 256)
output: (?, 2, 2, 256)
dropout_3: Dropout
input: (?, 2, 2, 256)
output: (?, 2, 2, 256)
flatten: Flatten
input: (?, 2, 2, 256)
output: (?, 1024)
dense: Dense
input: (?, 1024)
output: (?, 256)
dropout_4: Dropout
input: (?, 256)
output: (?, 256)
dense_1: Dense
input: (?, 256)
output: (?, 1)

To **compile** the model, I used the binary_crossentropy loss function, which is standard for a binary classification problem. For the optimizer, I used Adam, with default settings. I experimented with different settings of the learning rate, including a variable learning rate callback, but didn't see much improvement, so ultimately stuck with the default lr of 0.001.

The approach I used for **transfer learning** was to initialize the model with the ResNet50 or VGG16 architectures, with the pre-trained ImageNet weights, but without the final classifier layer, freeze the network, then add my own fully connected and output layers. Finally, I would proceed with training as usual.

As I was tuning my CNN, I struggled with how to combine **early stopping and k-fold cross validation**. While both concepts made sense separately, at first it wasn't clear what to do if a model stopped at different times for different splits of data. The goal of k-fold CV is to measure generalizability of a model, but if we allow early stopping, aren't we potentially over-fitting to a particular subset of data? The approach I settled on was to perform my development and initial tuning with early stopping on a randomly generated split of the training data using sklearn's train_test_split. This allowed me to iterate quickly. Once I had a promising model, I used k-fold cross validation to confirm the performance of the model, without early stopping. Then I would go back and do another round of development with early stopping and repeat.

## Refinement

When I was first setting up the structure of the project, I had some problems with the network **failing to converge**. The log loss metric would start very high (8+) and would remain high for the duration of training. Running the data through the MinMaxScaler solved this problem.

Applying **noise reduction** filters made the images nicer to look at but didn't substantially improve the performance of the model.

Of all of the data pre-processing steps I took, **data augmentation** by far had the most positive impact. This alone took my log loss score from the 0.4-0.5 range to the 0.2-0.3 range.

Tuning the **CNN architecture** was mildly helpful. Following Andrej Karpathy's advice of "bigger is usually better"[8] didn't really help in this case. Some gains were made on the training data but they didn't carry over to the test data.

Surprisingly, **transfer learning** didn't help much. I suspect the reason is that these radar images are not very similar to the ImageNet images the pre-trained networks were trained on. The radar images are much simpler yet more similar to each other.

[8] https://www.youtube.com/playlist?list=PLkt2uSq6rBVctENoVBg1TpCC7OQi31AlC

They aren't made up of the kinds of features the pre-trained networks are good at detecting.

After going through quite a few rounds of tuning and testing, my best log loss score was 0.19652, just short of my target of 0.19387. No matter what I tried, I would only get small incremental improvements. I needed a breakthrough. Seeking inspiration, I looked at a post from the contest winners that explained their approach. They indicated that there was **leakage in the inc_angle data**[9].

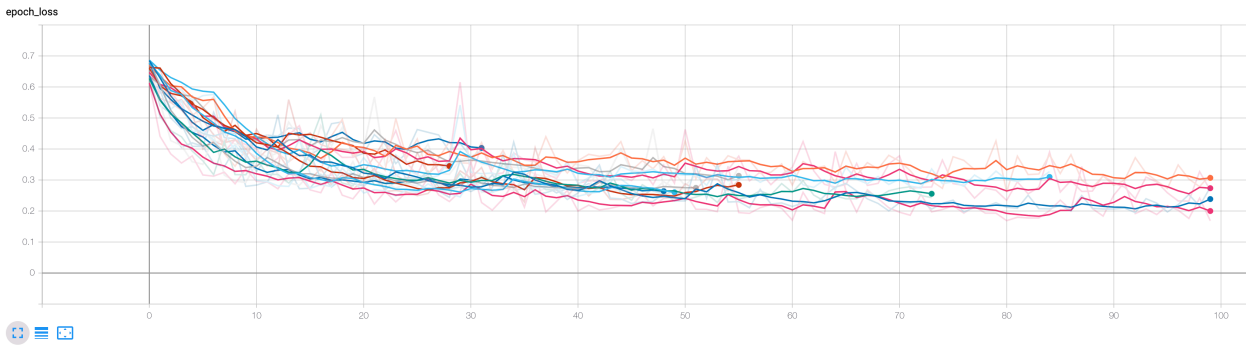
There were certain narrow ranges of inc_angle that contained only icebergs in the training data. For example, there are 27 images with inc_angles between 36.1 and 36.15, all of which are icebergs. This pattern seemed to carry over to the test data. I was able to use this insight to visually identify bands of likely icebergs. I ensembled my best CNN output with some heuristically labeled points by picking the max of the two outputs to get my best performing Kaggle submission.

# Part 4: Results

## Model evaluation

Figure 4 shows the Tensorboard output for about a dozen training runs. The x-axis is the training epoch number and the y-axis is the log loss score on the validation data. There are few things to note here.

- First, all the models converge, and the better ones end up in the 0.2 to 0.3 log loss range. This was not the case before I rescaled the data.
- Second, and more importantly, the shape of the curves suggest we are not over-training. If the loss scores against validation data decreased during early epochs, but then started to increase during later epochs, we would suspect over-fitting to the training data. This is not happening here, so the models should generalize well to the test data.




[9] https://www.kaggle.com/c/statoil-iceberg-classifier-challenge/discussion/48241

Figure 4: log loss score over time for multiple training runs.

k-fold CV shows stable results across different folds of the training data. With k=5, a typical run had log losses ranging between 0.19 and 0.31. This indicates the model should generalize well to unseen images.

## Justification

Figure 5 shows my final submission on Kaggle with a 0.19186 public leaderboard score, beating my benchmark of 0.19387. This benchmark was somewhat arbitrarily chosen, but it does mean that my model beat over half of the participants on this Kaggle competition.

| Submission and Description | Private Score | Public Score | Use for Final Score |
|---|---|---|---|
| inc_angle8.csv<br>2 days ago by Antoine Diffloth<br>1110_1054 | 0.22195 | 0.19186 | ☐ |

Figure 5: Final submission showing log loss score of 0.19186

My model's area under the ROC curve was 0.96, which is quite high. The plot of the ROC curve is in Figure 6.

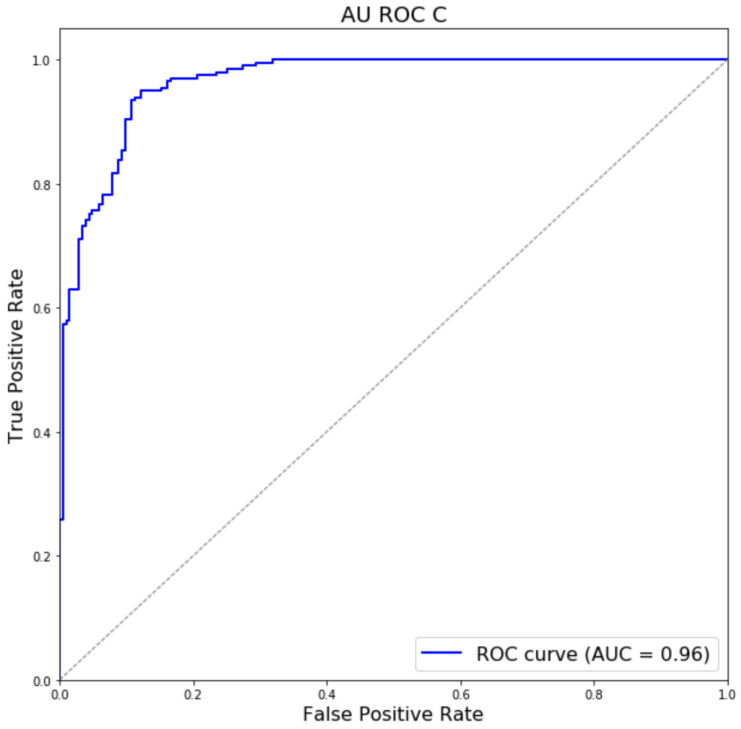



Figure 6: Area under ROC curve

Using a threshold of 0.35 to distinguish icebergs from ships, I get a recall of 0.95 with an F1 score of 0.91. In this case, recall is more relevant than precision, since finding all the icebergs is more important than avoiding false positives. These are very good scores.

Checking the confusion matrix in Figure 7 shows mostly true positives and true negatives, with slightly more false positives than false negatives. Once again, this is a good result since the cost of missing an iceberg is higher than the cost falsely identifying a ship as an iceberg.

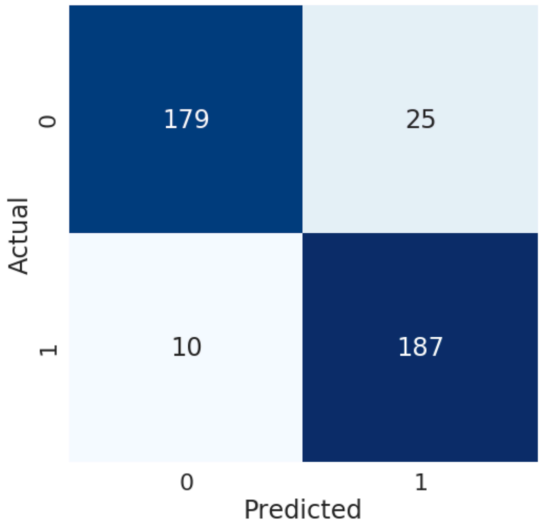



Figure 7: Confusion matrix at threshold of 0.35.

# Part 5: Conclusion

## Free-form visualization

Figure 8 shows the distribution of ships (orange) and icebergs (blue) when examined with very fine granularity. Each bin in the histogram is 0.05 degrees. At this level of zoom, a very striking pattern emerges. There are very narrow bands of inc_angle values that only contain icebergs, and there are wider bands of inc_angle values that contain both icebergs and ships. This exact same banding pattern exists in the test data. I was able to use this pattern to visually inspect the test data and pick out bands of 100% ships.

This insight, which allowed me to finally beat the benchmark, made me consider whether this was cheating. Technically this is data leakage since this pattern was an artifact of the way the data was collected for the competition and not representative of the way data would present in the wild. In the end, I felt it was fair to use this leaky feature since my benchmark was other competitors that also had the insight into the

pattern (there were posts about the leakiness of this feature before the close of the competition).

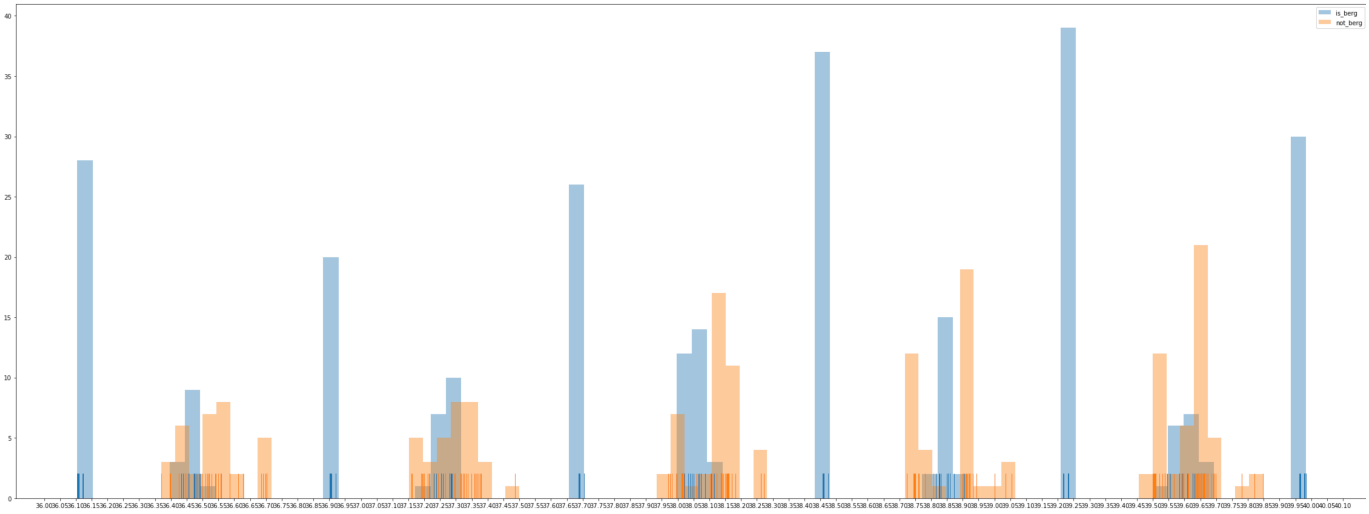



Figure 8: Distribution of ships and icebergs by inc_angle.

## Reflection

Overall, I was pleased with my approach to the problem and the results. I performed EDA to get a feel for the data, did some pre-processing to make the data usable by my model, I picked a reasonable formulation for my problem, fit my model to the data, iteratively tuned it and finally analyzed my results. I learned quite a bit in the process, but was also able to leverage what I learned during the course and on previous project work.

One thing that surprised me was that filtering and de-noising the images did not help the model's performance. My instinct as a photographer was to clean up the images and make them look "cleaner" so that the CNN could better classify them. Perhaps the lack of improvement is evidence that humans and computers see differently – high contrast, low noise images are easier to process for humans but not for machines (at least in this case).

From the beginning of the project, even before discovery of the banding pattern, I knew I had to find a good way to use the inc_angle data. I spent quite a while researching the topic of how to combine co- and cross-polarization data[10], reading research papers[11], and looking at trigonometric ways to transform the radar data using the incidence data[12]. None of that turned out to be as useful as simply exploring

[10] https://earth.esa.int/c/document_library/get_file?folderId=409229&name=DLFE-5566.pdf
[11] https://elib.dlr.de/99079/2/2016_BENTES_Frost_Velotto_Tings_EUSAR_FP.pdf
[12] https://www.degruyter.com/view/j/geo.2016.8.issue-1/geo-2016-0029/geo-2016-0029.xml

the data and visualizing it. Lesson learned for me: invest the time up front to do the EDA.

## Improvement

My approach to tuning the CNN was very “hunt and peck”. I would try one parameter change, evaluate the result and decide to keep or revert the change. I kept a log of the changes and the changing metrics but didn’t follow a disciplined approach to tuning. A more structured approach to hyper-parameter tuning (e.g.: grid search) would help me get the most out of my CNN.

I’d like to revisit transfer learning. The radar images I was trying to classify are quite dissimilar to the ImageNet data set the pre-trained networks were trained on. Rather than freezing the entire network and only training the classifier, a better approach might be to unfreeze a few of the later stage hidden layers to allow the network to figure out the radar image data set.

There is clearly a lot of signal to be exploited in the leaky inc_angle feature. There are two ideas I’d like to explore to further exploit this leakage. First, remove the images from the bands of known icebergs, leaving the bands of mixed icebergs and ships. Removing this “noise” might give the network a better chance of figuring out the characteristics of icebergs vs ships. The second idea I’d like to explore is finding a way to feed the inc_angle data directly into the CNN itself.

Overall, this was a fun project that allowed me to play with various techniques and apply the skills I’ve learned.