Visual Recognition of Ships Using CNN

By Christopher Broll, Shilpa Rajbhandari, and Zachary Stein

Introduction:

We are using convolutional neural networks and caffe to create a program that can carry out visual recognition of ships at sea. Taking satellite images, sometimes with fog and clouds, we will check the accuracy and loss values as the program analyzes them.

After acquiring the data, we took a subset of it, reduced the pixel values, applied histogram equalization and uploaded the data and the various code files to Google Cloud Platform (GCP). Once we ran them, we were able to look at how the accuracy and loss values changed with each iteration. Based on this, we made some changes to our code and the values that we chose in order to get better results. We were also able to see when our program started to overfit the data, and determined the cutoff for the number of iterations that we should use.

Description of the Data Set:

We acquired our data set from Kaggle (**https://www.kaggle.com/c/airbus-ship-detection**). The data consists of a series of satellite images of the ocean. In some of the photos, there are ships, in others only empty sea or ocean. Some of the photos also have clouds or haze, which add a complicating factor.

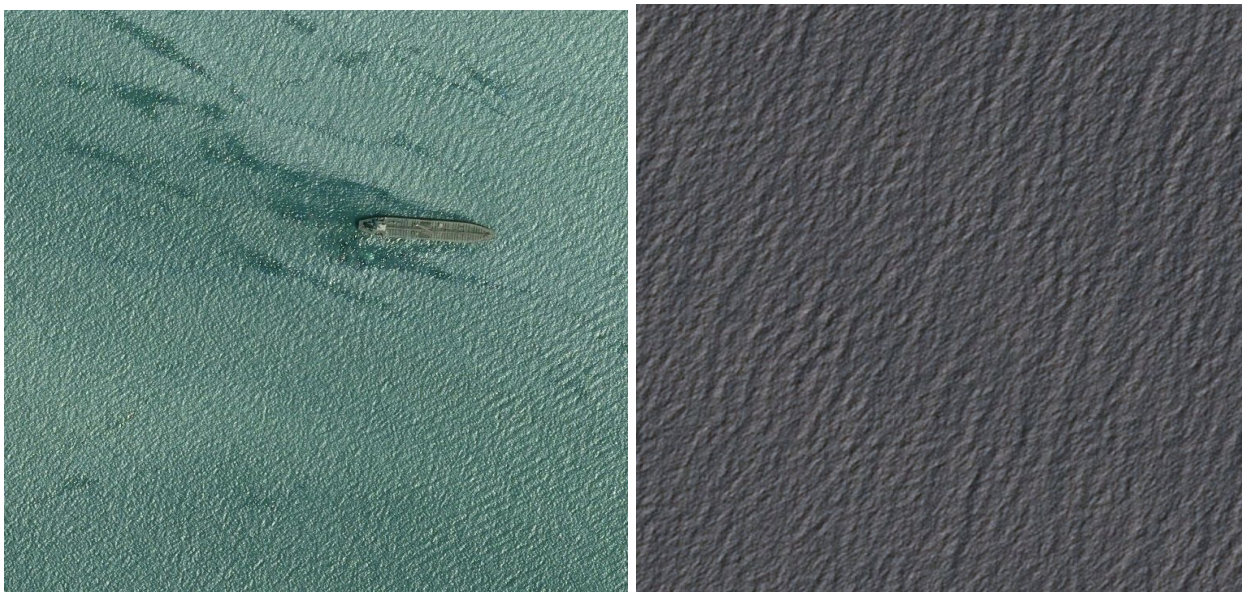

Figure 1 - Example image with a ship, and example image without a ship.

The dataset had 192,556 images. We thought that this was unnecessarily large for our analysis, so we pared it down to 50,000 images for training and 10,000 images for validation.

Description of the Algorithm:
There were multiple files of code that can be divided into three categories:

1. Pre-processing of the data.
2. Processing the data.
3. Support for the file that processes the data.

The Python file preprocess.py was used to create the subset of images from the original dataset. These images are the ones that will be divided into the training and validation images, and further divided into subdirectories 00000 and 00001, no ships and ships, respectively, for creating the lmdb files. After this, the images were uploaded to GCP, along with all of the other programs that we used, and we subsequently ran the create_lmdb.py file to begin training with Caffe.

The main code file that we used for analysis is train_airbus.py. This code is based on the use of Caffe and a Convolutional Neural Network. It runs through all of the images and creates a binary output based on whether or not its evaluation determines if a ship is present. It then compares this to the actual results, displaying the accuracy and loss values.

There were a variety of things that we tried before settling on our current set-up. We started off by resizing the images to 80x80 pixels and checking the results, which came off well. We then tried resizing to 32x32 pixels, but the accuracy results became worse. We also tried resizing the images to a size larger than 80x80 pixels, but there were problems when we tried to run it, so we just stayed with 80x80 for all of our tests and final analysis.

There are then three main programs that are used to assist with the analysis:

1. Our training algorithm is create_lmdb.py. This program resizes the images, and then divides images into training and validation test sets. Our original images were 768x768 pixels in size, but we tried multiple different sizes. The one that we settled on was 80x80 pixels. This program was the one placed on Dr. Amir Jafari's GitHub page.
2. airbus_solver.protext sets the various parameters for the iterations: The number of iterations, the learning rate, etc. This program is called by the airbus_train_text.prototxt file.

test_iter	test_interval	base_lr	momentum	Weight_decay	lr_policy
100	500	0.01	0.9	0.0005	inv
gamma	**power**	**display**	**max_iter**	**solver_mode**	
0.0001	0.75	100	100	GPU	

3. airbus_train_text.protext lists the various layers that will be used by the program. This program is called by the airbus_solver.prototxt file.

Convolution 1	num_output	kernel_size	stride
	60	5	1
Convolution 2	num_output	kernel_size	stride
	120	5	1
Pooling 1	kernel_size	stride	
	5	5	
Pooling 2	kernel_size	stride	
	5	5	
For inner product	num_output		
	1000		

Once all of these files are created and set-up, we just need to run create_lmdb.py to properly set-up the data, upload the photos to GCP, and then run train_airbus.py to do the analysis.

Experimental Set-Up:
For training and testing, we did 50,000 images for training and 10,000 images for testing. For every 100 iterations we used 100 images to validate the results. We used 64 iterations for the training set and 100 iterations for the testing set. We based this on what we had done with the mnist dataset in class. For this reason, we kept the proportion of training/test images the same, and chose a test dataset size that is divisible by 100.

For preventing overfitting, we added a layer called loss_val. This keeps track of the loss values for each iteration, and is used to make a graph so that the results can be visualized. When the results start to decline and then rises again, the nadir can be used to identify the point where overfitting occurs.

For setting-up the size of the parameters in airbus_train_test.prototxt, we choose it based on either experience or what was best for the data. For the parameters in the convolution layers, we used the ones that were used in mnist from class. We saw no reason to change this, and kept it with what we had used before that worked. For the pooling, we did choose the parameters, setting kernel_size and stride at 5. This is because the pixel size was 80x80, making it evenly divisible by 5.

Results:
The results that were produced were for the images after they were rendered into a size of 32x32 pixels. We used the following feature maps and kernels for the analysis:

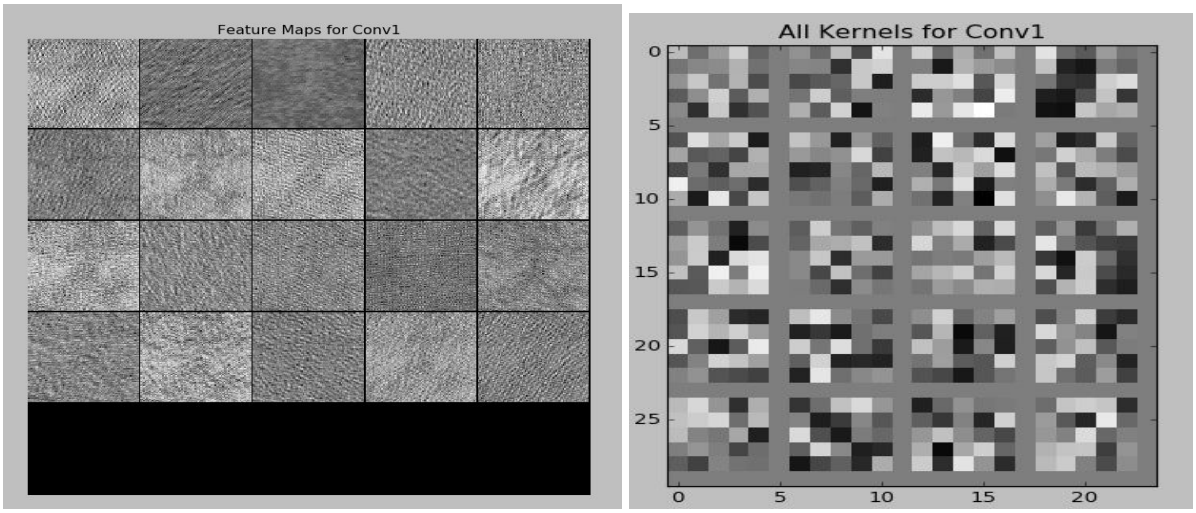

Figure 2 - Feature maps (Left) and Kernel (Right) for 32x32 pixel images.

Then, we started with 20,000 iterations for our program and checked the results. This was done for images that were 32x32 pixels and images that were 80x80 pixels.

```
Iteration 19600 testing... accuracy: 0.26
I1209 12:00:26.419978  2359 solver.cpp:239] Iteration 19700 (17.7861 iter/s, 5.62236s/100 iters), loss = 0.00213971
I1209 12:00:26.420035  2359 solver.cpp:258]     Train net output #0: loss = 0.00213971 (* 1 = 0.00213971 loss)
I1209 12:00:26.420044  2359 sgd_solver.cpp:112] Iteration 19700, lr = 0.00442011
Iteration 19700 testing... accuracy: 0.28
I1209 12:00:32.067144  2359 solver.cpp:239] Iteration 19800 (17.7084 iter/s, 5.64703s/100 iters), loss = 0.00301393
I1209 12:00:32.067215  2359 solver.cpp:258]     Train net output #0: loss = 0.00301393 (* 1 = 0.00301393 loss)
I1209 12:00:32.067225  2359 sgd_solver.cpp:112] Iteration 19800, lr = 0.00440898
Iteration 19800 testing... accuracy: 0.32
I1209 12:00:37.713999  2359 solver.cpp:239] Iteration 19900 (17.7093 iter/s, 5.64674s/100 iters), loss = 0.00429516
I1209 12:00:37.714054  2359 solver.cpp:258]     Train net output #0: loss = 0.00429516 (* 1 = 0.00429516 loss)
I1209 12:00:37.714062  2359 sgd_solver.cpp:112] Iteration 19900, lr = 0.00439791
Iteration 19900 testing... accuracy: 0.29
```

```
Iteration 19600 testing... accuracy: 0.45
I1209 15:09:25.506752  3137 solver.cpp:239] Iteration 19700 (24.0533 iter/s, 4.15743s/100 iters), loss = 0.00296536
I1209 15:09:25.506834  3137 solver.cpp:258]     Train net output #0: loss = 0.00296536 (* 1 = 0.00296536 loss)
I1209 15:09:25.506856  3137 sgd_solver.cpp:112] Iteration 19700, lr = 0.00442011
Iteration 19700 testing... accuracy: 0.51
I1209 15:09:29.656904  3137 solver.cpp:239] Iteration 19800 (24.0957 iter/s, 4.15011s/100 iters), loss = 0.00844194
I1209 15:09:29.656966  3137 solver.cpp:258]     Train net output #0: loss = 0.00844194 (* 1 = 0.00844194 loss)
I1209 15:09:29.656987  3137 sgd_solver.cpp:112] Iteration 19800, lr = 0.00440898
Iteration 19800 testing... accuracy: 0.54
I1209 15:09:33.805871  3137 solver.cpp:239] Iteration 19900 (24.1026 iter/s, 4.14894s/100 iters), loss = 0.024471
I1209 15:09:33.806030  3137 solver.cpp:258]     Train net output #0: loss = 0.024471 (* 1 = 0.024471 loss)
I1209 15:09:33.806040  3137 sgd_solver.cpp:112] Iteration 19900, lr = 0.00439791
Iteration 19900 testing... accuracy: 0.48
```

Figure 3 - Accuracy for images that are 32x32 pixels (Top) and 80x80 pixels (Bottom).

The 80x80 pixel images gave a significantly better accuracy than the 32x32 pixel images. For this reason, we used 80x80 pixel images for the rest of our analysis. We then graphed the accuracy and loss values over the various iterations.

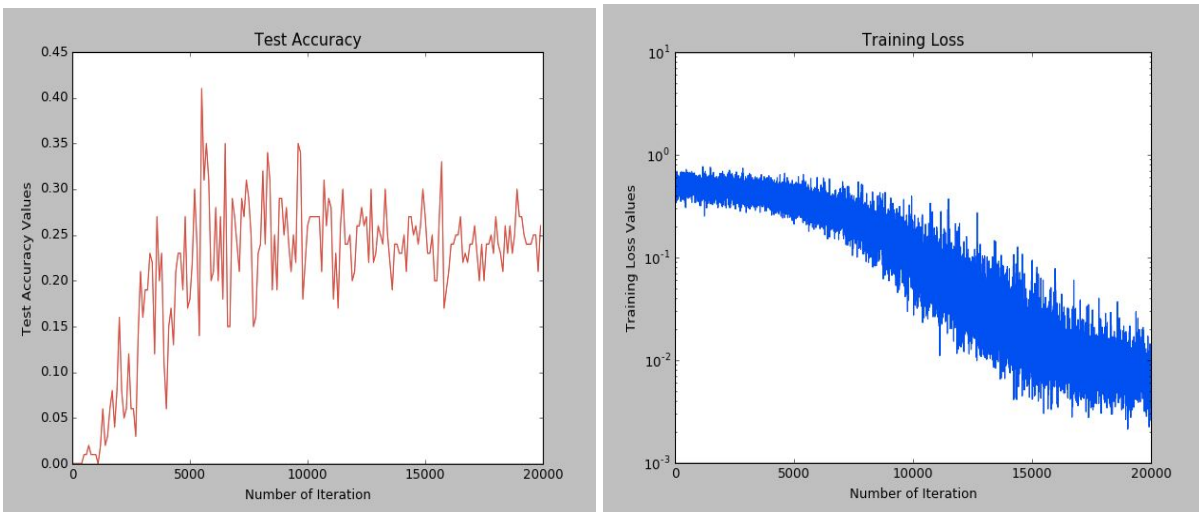

Figure 4 - Accuracy (Left) and loss (Right) for 80x80 pixel images.

After a certain number of iterations, the accuracy value levels off, while the training loss values start to seriously decline. To make sure we were doing the best job for our analysis, we checked for overfitting. This was done by plotting the validation and testing loss over the number of iterations. The validation is the most valuable part. It will decrease and then increase again. We can then use the nadir to identify the point where, if we continue iterations, we will be overfitting.

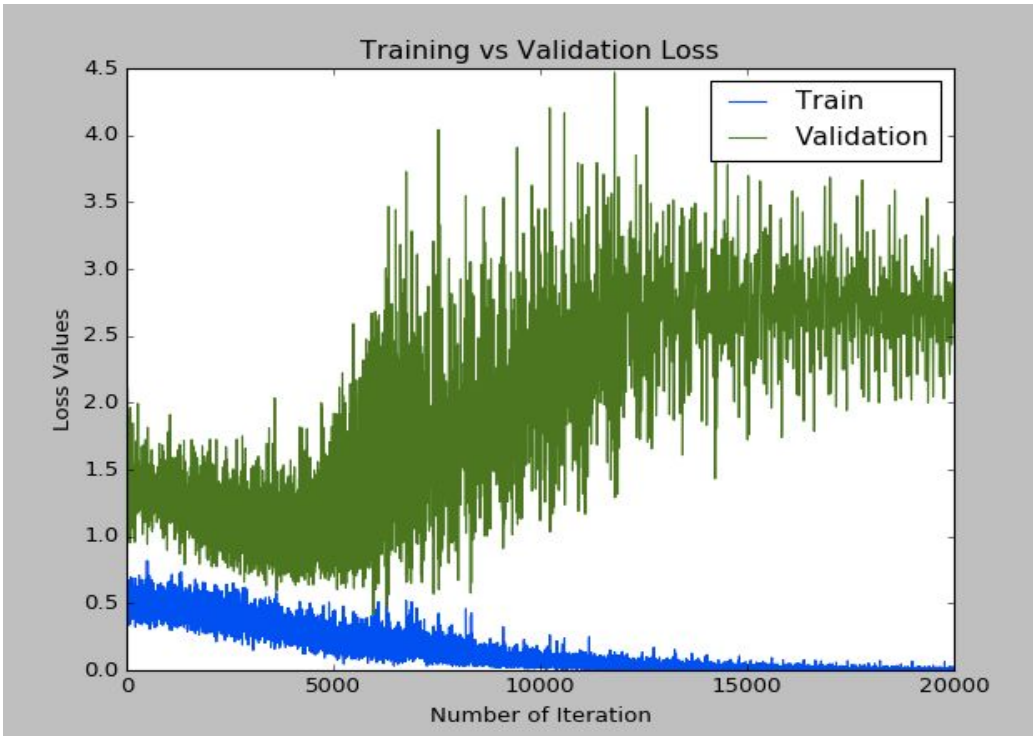

Figure 5 - Validation (Green) and Testing (Blue) Loss

We found that overfitting happened after 5,000 iterations, so we marked this as our cutoff point.

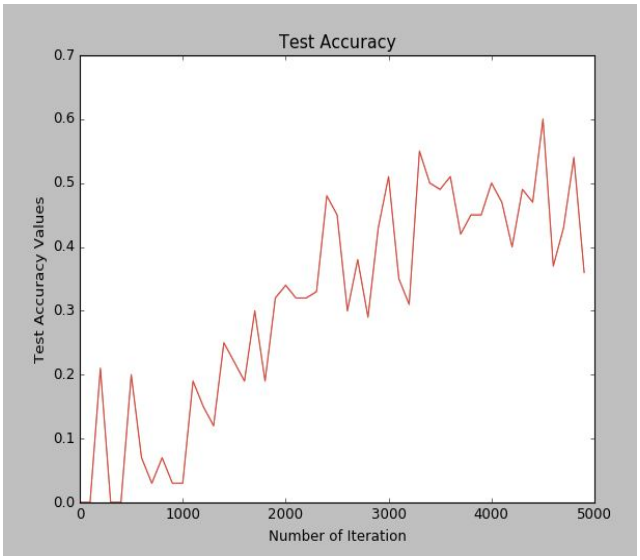

Figure 6 - Early Stopping

Summary and Conclusions:

These results show that the results for our data analysis are best when we convert the images to 80x80 pixels. At his resolution, we get an accuracy rating of around 50%, though once we are past 5,000 iterations we start to get overfitting. This means that we can successfully identify satellite images with ships with an accuracy equal to a coin flip.

Through our work with the code, we have learned that our results are very sensitive to the resolution of an image, and that there is such a thing as too much resolution and too little resolution. We have also found that the number of convolution layers and the parameters we choose for them have an effect on our results.

If we want to improve this, we need to work to find the ideal resolution, layers, and parameters. This is a delicate act, which requires a great deal of trial and error to learn what will work with the images. It might also help to load a larger dataset, as the greater variety of images can help to train the program to better discriminate between oceans and ships, though it will require more time to set-up and run.

References:

Data - https://www.kaggle.com/c/airbus-ship-detection
Code - https://github.com/amir-jafari/Deep-Learning/tree/master/Caffe_