Hubble Bubble: Satellite Image Road Prediction

Elliot Smith and Santiago Tellez

COMP 540

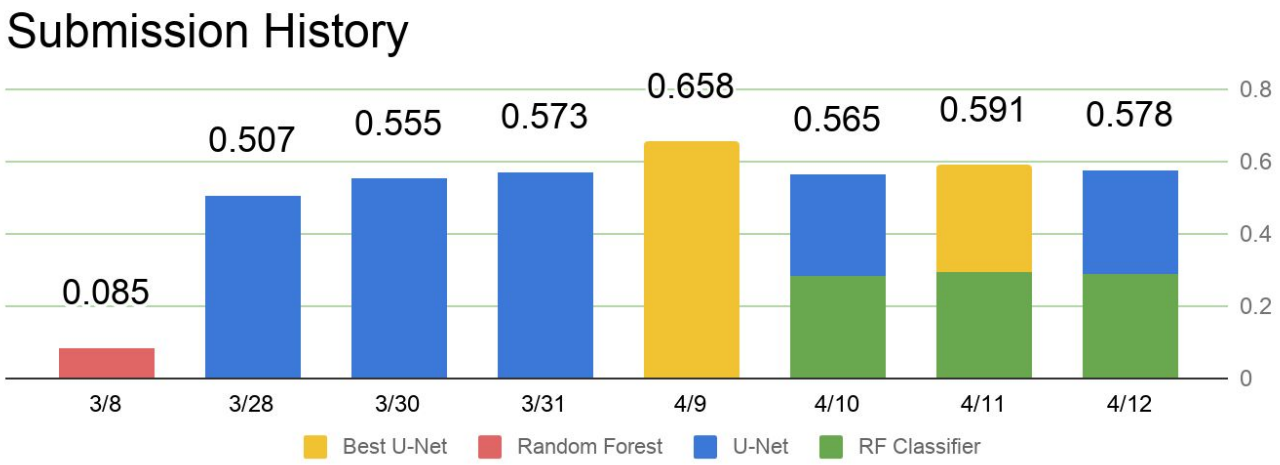
Introduction

Generally, the goal for this project is to detect roads in satellite images. Our training data includes the satellite images along with masks indicating which pixels are deemed to be roads.

Our strategy for the satellite image road segmentation task involved two attempts at finding good mask predictions.

1. The first strategy was to take the data, unfiltered and with no processing, and to fit a U-Net to the images to find roads. It ended up producing our best result.
2. The second strategy was to combine 2-3 models in order to predict smooth masks. For the first model, we tried to classify whether an image contained any roads. The motivation for this was due to the large amount of completely black masks in our training set. Perhaps the U-Net, when trained only on images containing roads, performs much better than when it tries to learn to discriminate between the blank and non-blank masks. The

second model is a U-Net (same architecture as in the first strategy). For the last model we have not finalized the method we will implement, but it could involve some sort smoothing to the predicted masks. This could be training a second U-Net on the predicted masks, or perhaps applying a filter to the masks to fill in missing pixels in roads.

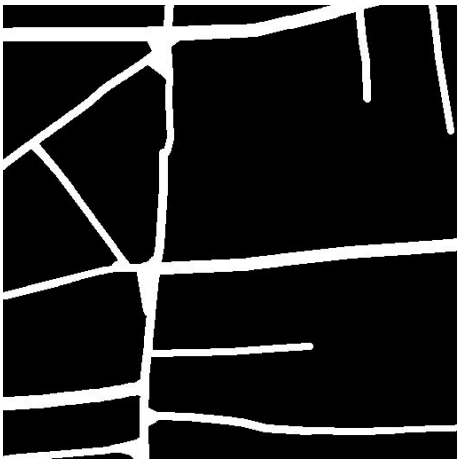


Before we get in to our modeling process, above is a chart showing our submissions over time. A lot of the submissions involved trial and error to see if changes could add some points to our dice score. The red indicates our random forest pixel-by-pixel model submission, the blue indicates U-Net submissions, the yellow is our best U-Net architecture, and the green indicates submissions using the random forest classifier which we will describe later on in the report. The best dice score we achieved was 0.67 on April 9th.

Exploratory Analysis

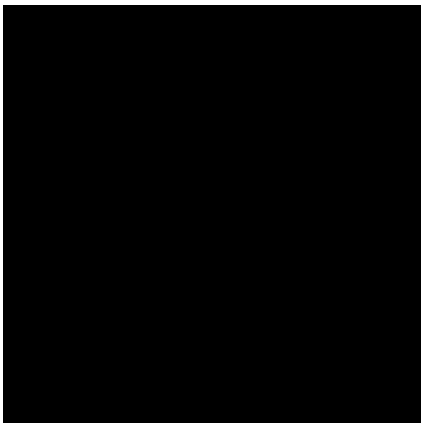
The images have varying landscapes, color palettes, and types of roads. One of the things we noticed early on is the difficulty in detecting roads even manually. A lot of the labeling of

roads seemed somewhat arbitrary. The fundamental rule for why what would appear to be a road ends up being labeled as a road or not still remains unclear to the eye test. For example, take this image below:



The mask seems to clearly catch the most obvious roads in the image. However, the less obvious roads appear to be arbitrarily labeled. In the middle-right of the image there is a horizontal road with a smaller road stemming down from it. While to us this may obviously seem to be a street, the mask does not have it labeled.

Another case of ambiguous road labeling is shown in the pictures below:



To me, it seems that there is a clear road cutting from the top left through the collection of buildings at the bottom. One of the challenges we faced was determining what constitutes a road, and adapting our modeling process to reflect the generating function.

At a high level, the problem is this: we don't know how the roads were labeled, but if DeepGlobe.org is posing the challenge then we can perhaps assume the roads were drawn by hand. The goal then becomes not specifically to label pixels which to us appear as roads, but rather to label roads on unlabeled images according to the rules the manual road labeling adhered to. It would be very useful if along with the data we were provided some general rules describing how the roads were labeled. Since we do not have this information, then we must train our models with limited prior knowledge about what a road should look like and see if the methods we implement can "learn" what the human road labelers were thinking when drawing the masks.

Road vs. Non-Road Image Classification

The purpose of the first model in our pipeline is to classify satellite images as not having any roads or having some roads. The underlying motivation for this first model is that the U-Net model will better fit a road prediction task if it is trained on images containing roads rather than forcing it to handling the road vs. non-road discrimination problem at the same time.

Random Forest

The first attempt at classifying the images involved using a random forest classifier and summary statistic-based feature engineering. The mean, median, variance, minimum, maximum, range, 25th quartile and 75th quartile was computed for the image's RGB values. Then, a random forest

classifier was fit to classify each image as having roads or not having roads. The chart below shows the results on the training data:

	No Roads	Roads
Predicted No Roads	1583	63
Predicted Roads	255	8996

Convolutional Network

For the next attempt, we tried fitting a convolutional network to our images. Unfortunately we were unable to get better predictions than the random forest classifier.

	No Roads	Roads
Predicted No Roads	1	14
Predicted Roads	1365	6793

K Nearest Neighbors

Since the convolutional network didn't work, we tried a simpler method.

	K = 2		K = 10		K = 25	
Prediction	No Roads	Roads	No Roads	Roads	No Roads	Roads
No Roads	1838	4719	1570	5201	1425	4545
Roads	0	4339	268	3858	413	4514

The KNN classifier performed very poorly and it was unable to distinguish between images with and without roads. The best model we fit was the random forest using the engineered features. The idea is to now train a road segmentation model using only the images deemed to have roads

by the random forest classifier. Unfortunately, we didn't develop a good classifier until the very last week so the models discussed below do not use this method. For future work, training a model on the road images would be a great step to try out. For now, what we did was submit blank masks for the images deemed not to have roads by the classifier and then submitting masks using the model trained on all the images, not just the "have roads" images. The dice score did not improve significantly, and the best dice score we achieved did not involve the road vs. non-roads classifier.

Road Segmentation

Naive Approach: Random Forest

Before we knew about neural networks, we tried fitting a random forest to the image pixels. The idea was to generate pixel-by-pixel features and then fit a random forest to a dataframe with each observation being a pixel. We quickly realized that this method would be very difficult to construct with the amount of data we were dealing with. Just one image produced 256,000 pixels.

The features we used for this model involved taking averages for the RGB values within each "neighborhood" of the image. For example, our images were 512 by 512, so one set of possible features could be represented as:

140.5	138
142.2	141.1

Where the 512 by 512 image is divided into four different 256 by 256 squares. The RGB values for each square are averaged, and every pixel in the square would receive those averages as a feature. This process is repeated for neighborhoods of 128 by 128 grids, 64 by 64, and so on.

Due to the feature engineering portion of predicting roads using this method, we had to train the model on a small subset of the images. The random subset consisted of about 20 images worth of pixels drawn randomly from 500 images in the training set. Within this subset, the model was able to achieve a dice coefficient of 0.92 on holdout validation. However, when predicting roads with the random forest on the validation set provided for Kaggle, the model generalizes very poorly.

U-Net

For our final approach, and the one that we selected to perform the road pixel classification, we utilized, as a foundation, the U-Net architecture referenced here (https://www.kaggle.com/keegil/keras-u-net-starter-lb-0-277). The U-Net architecture has a reputation for having great success on image classification problems, and with provided code, we could work with the supplied skeleton and look to tweak certain attributes of the network to refine our results.

The U-Net architecture involves two distinct phases: first, is a contraction phase, which consists of a series of convolution phases, an application of an activation function and a max pooling event. The contraction phases works to reduce down the spatial element of the images while

magnifying the features present. After this, an expansion phase occurs where convolutions continue to occur expanding the image while concatenating the reduced images from the contraction phase.

For our first attempt, we used the provided code “out of the box” and didn’t change anything at all; we used that architecture as is to train our network. Our goal for our first attempt was to get a baseline dice score to beat; in this way we could understand what a “vanilla” solution would achieve and iteratively try to best that number. The pixel dice score that we got was 0.5.

For our second attempt, we wanted to expand the image size from 128x128 to 256x256. With the existing code infrastructure, the images are originally taken from 512x512 and reduced down to 128x128; this takes sixteen pixels and crunches them down into one. Obviously, there is a lot of information lost using this approach, so our first approach was to instead reduce four pixels down to one and see if we had an improvement in our results. We did, and with this attempt achieved a pixel classification dice score of 0.55, a marked improvement.

Our third attempt look to improve upon our previous result by this time feeding the full images to the network and seeing how our dice score improved. We again did not change anything about the structure of the network and this time we got a dice score of 0.57. At this point in our experiment, we had exhausted our ability to provide unadulterated images to our network and instead looked to other ways to improve our network performance.

Our fourth attempt, or rather a series of attempts aimed at experimenting with the activation functions and number of layers we were using to try to improve performance did not yield very promising results; I will list these attempted changes below:

- We tried changing all of of our activation functions from exponential linear units to sigmoid functions. The result of this was that the network was completely unable to learn and was paralyzed. In retrospect, we realized that we needed to scale the output from the sigmoid between 0 and 1, but we did not realize it at that time.
- Then we tried alternating activation functions in each layer between exponential linear units and sigmoid functions. However, we met with the same fate as above for the reason outlined above.
- Our next attempt was to try using a rectified linear unit as the activation function; this attempt was again meant with poor results. It was at this time that we decided to stick with the exponential linear unit and look for other areas to try to optimize.
- Finally, we attempted adding an additional layer to our network at both the beginning and end of the network. Even adding one additional layer completely paralyzed our network and we decided to return to our additional layer count.

Our fifth attempt was the most successful since we decided to use the full images. In this case, we decided to alter the current network infrastructure by doing the following: at each layer, instead of having a convolution step, followed by a drop out, followed by another convolution step and the max pooling, we instead decided to have a convolution step, a drop out, another convolution, then a second drop out, followed by a third convolution. We implemented this

change both during the contraction phase and the concatenation phase. This change yielded great results, as our dice score jumped from 0.57 to 0.66.

For our sixth attempt, now that we identified that there was a notable improvement by expanding each layers drop out and convolution steps, our next prerogative was to add even another drop out and convolution step to each layer. We wanted to determine whether adding yet more complexity to each later would grant us more favorable results. This was however not the case; when we added yet another drop out and convolution step to our network, the network was unable to learn properly and we quickly deduced that this was not the right approach. We reverted back to our original change from our previous attempt.

For attempt seven, we wanted to experiment with the drop out parameter, in other words, with the fraction of neurons that would be turned off at each layer during a given pass. Our first inclination was to reduce the drop out and see how our network would respond. At each layer, we reduced the drop out percentage by 5% (0.05). In this way, less neurons would be dropped out and we may encourage neuron specialization. We inferred that there would not be improvement and we were rewarded with what we expected, our dice score decreased to 0.57.

In regards to our eight attempt, we wanted to try something very unique to address the problem at hand: we wanted to help our network identify where the roads were by adding extra dimensions that highlighted where there were stark pixel differences in our images. To do this , we used a technique called Canny filters that looks at grey scale images and attempts to highlight

where the sharp contrasts in the pixels are (this should help us identify roads!). To facilitate this, we appended four Canny filtered images to each of our RBG images with varying levels of contrast hyper parameters (basically, how sharp of a contrast shall we consider). While we were very excited about this approach, our results were lacking. Actually, the network did not learn at all after the first few epochs and the network's "patience" tolerance was met and the learning terminated.

Our final attempt for our pixel classification model was to try now implementing a 5% increase in the drop out parameter; this we believed would have a positive affect on our network's discriminative ability because neurons would become less specialized. For this network, our dice score was 0.6 which was not an improvement on our optimal model which received a score of 0.66.

We learned a lot from the process of implementing the U-Net and tweaking the architecture for incremental improvement. The implementation was a very informative process as we had minimal experience working with Keras implementations. In addition, we were also both relatively new to using AWS particularly as it relates to large-scale computing projects. We had to engage in an interesting balance between letting networks train for a period of time to assess whether or not they could be effective and the time cost associated with training a network that ultimately might not do well.

We however ran out of time to attempt some of the components we were considering on implementing. One thing that we would have liked to include was successfully adding more parameters to our network, we felt as though that would have been a great opportunity to improve our network's performance. Unfortunately, each time we added additional layers, our network performance dropped significantly; we never had the time to diagnose why this was happening and so we moved on to other performance-enhancing techniques. Another element that we strongly believe would help make our network a better predictor was the use of the Canny filters, but we never had any material success using them so we dropped them from our final model. Had we had more time to refine our network, we would want to spend a substantial amount of time on the Canny filtering.

Mask Smoothing and Filling in Predictions

Something we noticed from the output of our U-Net was that our mask predictions didn't always look like roads. The predicted pixels in the mask were spotty and discontinuous, unlike roads in real life. For example, take the predicted mask below:

There seems to be one main road in this mask and then two roads branching from it. Regardless of whether or not the road pixels are correct or not, the mask does not resemble a road. If the main road exists, then it should be continuous and connect the left and right sides of the image. Our idea for correcting this issue, although we didn't have time to fully develop it, was to create a model or filter to fill in roads with jumps in the mask. We thought of implementing a simple, non-machine learning based algorithm to smooth out the predictions by taking the average mask value around each pixel and iterating it until the mask had no discontinuities. A more systematic approach would be to take the predicted mask and fit another U-Net algorithm, perhaps helping the original network in filling in roads.

For future work, this is definitely something we would like to explore. It seems that the U-Net does well from a naive standpoint, but needs help linking the gap between a probabilistic pixel-by-pixel prediction and a smooth prediction using prior knowledge about what roads should look like in practice.

Conclusion

We learned a lot of great lessons from this experience. In general, we found it very useful to learn on both the experience and the hard work of our predecessors. The opportunity afforded to us by having the skeleton of the U-Net architecture provided to us allowed us to focus on the small changes that might be optimal to our particular problem as opposed to focusing on engineering a completely new architecture for this particular problem. In addition, we learned a lot about how to break down a problem into module that we can focus on individually. For

example, the idea to stack two models, one that would discriminate between whether or not an image had roads or no roads and then one to identify the road pixels for an image that had roads was a major breakthrough for us. It showed us that we can develop individual pieces that are specialized at certain parts of the pipeline and then combine for our final solution.

There were many areas that we could have improved on, though we had to learn these lessons through experience. One issue we had was that too early in the process we became very focused on trying to build the solutions ourselves instead of turning to existing packages that already solved our particular needs; this led to a lot of time spent that had no impact our final result. In addition, we found that at the end we were trying to experiment and do a lot of network training very late in the process. It is not that we started late, but had we made certain design decisions early (for example: to use the U-Net architecture), we would have had much more time to train more and experiment with ways to improve our network accuracy.

Another area of improvement that we would have liked to include but did not was ensembling our results. As we know, convolutional neural networks can be heavily biased by starting location. To combat that, it would have been a great idea to, for example, train several U-Nets, get the predictions from these different networks and then for each predicted pixel take a mean, median, minimum or maximum value. In this way, we can protect against potentially adverse starting locations for our network and use this ensembling approach to “vote” on a winning prediction.

References

- https://en.wikipedia.org/wiki/U-Net
- https://www.kaggle.com/abhmul/python-image-generator-tutorial
- https://www.kaggle.com/keegil/keras-u-net-starter-lb-0-277
- https://spark-in.me/post/unet-adventures-part-one-getting-acquainted-with-unet