Machine Learning Engineer Nanodegree

Capstone Project

Mark Helms
September 15, 2017

Project Overview

Satellite and aerial imagery around the globe is readily available through the use of self-sourced means, such as airplanes and drones, and through databases such as those used by Google Earth. Not all roads and trails are represented on the corresponding maps, however, or their representation may be incomplete. Having a more complete representation of roads and trails could be helpful for entities such as businesses, governments, militaries, hikers, tourists, and others. In poorer areas, where surveying equipment or services may be considered a high expense, it's possible that databases of roads could be automatically built using the available satellite imagery and image processing software that identifies roads and trails. This project is of personal interest in that I would like to explore the world of image segmentation – one way of discovering and outlining objects within images, as opposed to classification of the image as a whole.

Some machine-learning based research has already been performed on this task or tasks similar in nature to this one. Mnih and Hinton performed the most similar work by analyzing large contextual areas of high-resolution urban imagery using neural networks that were trained on “massive” amounts of data to detect roads [4]. They pre-processed their data using unsupervised Principal Component Analysis, pre-trained their neural network using Restricted Boltzmann Machines (an alternative method of identifying features which can be used as weights in the first layer of a deep convolutional neural network), and further refined their results using another neural network to perform supervised training on each patch of output from the primary neural network.

In research focused on classifying parts of neural tissue cells, Tschopp used a convolutional neural network to perform pixel-wise classification on brain cell images in order to identify areas defined as "foreground" versus those defined as "background" [5]. Not only are cell walls successfully categorized, but more intricate patterns as well. His technique incorporated a “strided-kernel” method introduced by Hongsheng Li [3], whereby through special stepping of the kernel during convolutions, the equivalent result of a simple sliding window can be produced in a manner that reuses as many of the convolutional products as possible which would otherwise be repeated, thereby greatly saving computation.

Liang-Chieh Chen, Papandreou, et. al. combine deep convolutional neural networks with conditional random fields, and apply a "hole" (atrous) algorithm for efficiency. [1] Chen and Papandreou's work produces fascinatingly accurate silhouettes of classified objects, with reduced false blotches and cleaned-up outlines of otherwise coarse results.

Problem Statement

The problem attempted in this project will be the identification of roads in satellite images downloaded from Google Maps of the Washington, DC suburbs. This type of problem is known as a semantic image segmentation problem, or alternatively, as a pixel-wise image classification problem.

Each n x n image will have each pixel classified as either “road” or “not-road,” based on the surrounding pixels (the image context of that pixel). The proper solution, of course, assigns these classifications properly to the image, as judged by a metric presented under the “Metrics” section below called “Intersection over Union.”

Evaluation Metric

Success is measured by the intersection-over-union (IOU) value, also known as the Jaccard index, which is defined as the area (measured in pixels) in common between the predicted and true roads, divided by the total area of the predicted and true roads. IOU measures how well the predicted values match up with the ground truth values, while at the same time penalizing bad predictions that don't match up. The result is a number between 0 and 1, with 0 being no overlap between prediction and truth and 1 being complete overlap thereof.

This can be formulated as:

$$IOU = TP / (TP + FP)$$

The IOU metric is appropriate to this task, because the generated product will be a roadmap which should match up to the true roadmap. IOU simultaneously penalizes predicted roads where none exist while rewarding a high percentage of overlap. This can be considered as combining two other popular metrics, completeness and correctness, which could be analyzed individually in order to analyze the trend of mistakes made by the algorithm. However, these can also be easily observed and understood by graphically comparing the prediction versus the true labeling.

Data Exploration

Satellite imagery and known road maps are used as the feature vectors and ground truth labels, respectively. To keep processing time low, a total of 436 imagery-roadmap pairs are used. Each imagery image is 512x512 pixels in size and recorded in the RGB format. This provides a total of $436x512x512x3 \sim= 343x10^6$ data points, although the images are each later reshaped to 256x256x3 in a preprocessing step, resulting in $436x256x256x3 \sim= 85x10^6$ data points.

An example satellite image and corresponding road map are shown as Figures 1a and 1b. Figure 1a is a satellite image of a suburban area near Washington, DC with all labels omitted. Figure 1b is a simplified road map of that same area and operates as the ground truth for the task. The maps were sourced using Google Map's stylized maps [2] with the python code in Appendix A - Generating the Maps. All other map elements besides roads, such as labels, parks, points of interest, and buildings, are omitted.

As can be seen in Figures 1a and 1b, there are a few areas which are likely to give trouble. The cul-de-sacs in the satellite imagery are not fully represented in the roadmap image. Vegetation and shadow coverage of the road is apparent in a couple points, especially part of the "D"-shaped road in the lower right.

Nearly all images contain at least some road. Many images contain parking lots. Types of roads included vary and include small, country roads, wide, suburban streets and multi-lane highways. Correspondingly, building density varies from wooded areas to that of city-like.

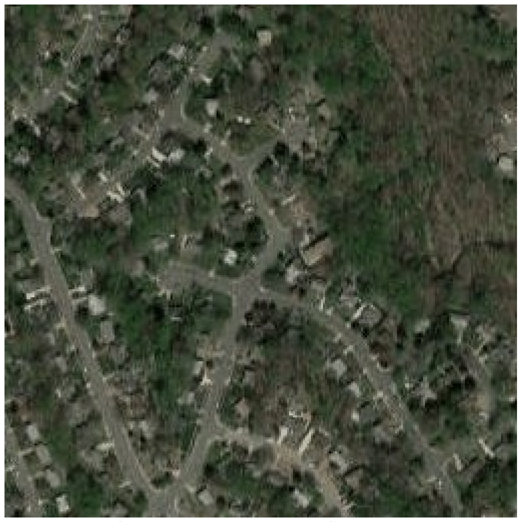
Figure 1a. A sat image.

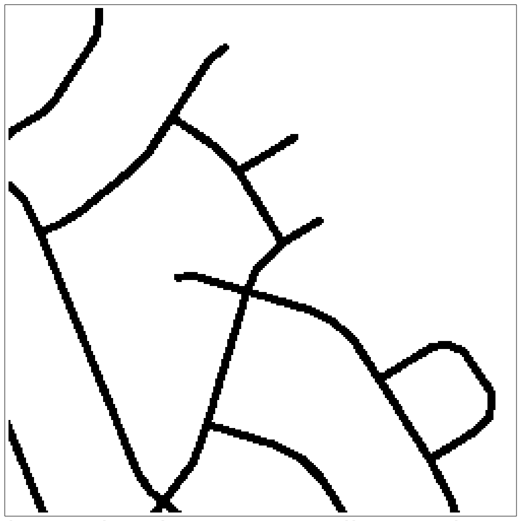
Figure 1b. The corresponding roadmap.

An important parameter which was calculated was the average percentage of each image that is covered by roads, which is about 10%. This is an important metric which can be used for sample weighting.

Part of the original dataset (64 images out of 500) was culled due to its being from a separate data collection. The images removed contained a large number of shadows, whereas the retained images are fairly clear. An example of a shadowy image can be seen in Figure 2a, and a clearer image can be seen in Figure 2b. This was done to better control the quality of the data.

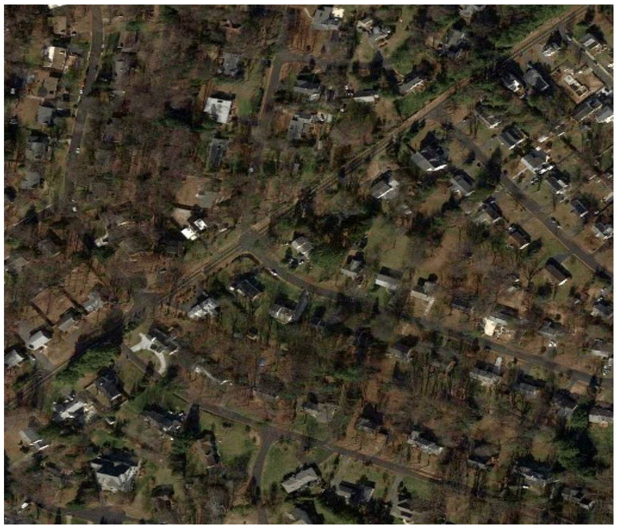
Figure 2a. A shadowy sat image.

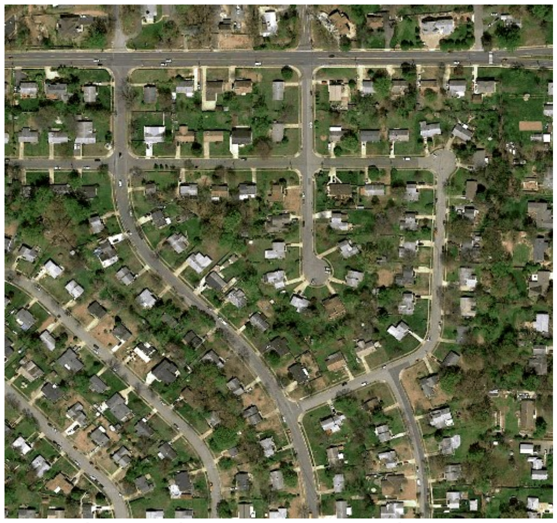
Figure 2b. A clearer sat image.

The biggest hurdles are roads which are completely hidden by vegetation, roads which are extremely skinny, private roads which are unmarked, and parking lots, which resemble roads.

Picking a zoom level and image size were important when obtaining the dataset. The higher the zoom level, the more can be seen, but the blurrier it becomes. One thing that isn't totally clear in the Google Maps API documentation is that there is also a scale option that allows for a user to retrieve an image at twice the otherwise maximum dimensions. Images were retrieved with scale=2 and then resized by a factor of 0.5 in order to obtain very crisp looking features. This appears to be possible because of the blur caused by Google's compression, which is presumably performed independently for each scale.

Algorithms and Techniques

Preprocessing

The first techniques used on the data have to do with preprocessing. They include a reshape method to decrease the processing load and effectively remove image blur. Secondly, the images are adjusted with a brightness and contrast filter so that features stand out more. Both removing the blur and adjusting contrast and brightness should allow the filters to more easily form around the sharper features created.

The next step in preprocessing is to perform Principal Component Analysis (PCA). This is a technique that can be used to reduce the number of features per sample by finding axes along dimensions of highest variance, onto which the data are projected. This can be a great way to decrease computational load, however it comes at a performance cost.

Finally, each component that is returned by PCA is standardized so that the mean of the result is 0 and the standard deviation is 1.

It was decided to not pad the data, in an attempt to achieve insight into the performance of each approach under optimal conditions. When data is padded, either with zeros or using mirroring, it does not reflect true data. Padding can be helpful when attempting to reproduce an entire image, however that was not the top priority of this investigation.

Preparing the Batch Process

An important step in preparing the data is to split it into training, validation and testing sets. Additionally, shuffling the training samples is important and a proper framework for doing so makes the process of dealing with the data much easier. This was accomplished by writing iterators in python to produce the samples in a controlled fashion. A random seed was used to make sure that any shuffling that was performed was repeated in exactly the same way every time, so that results are reproducible.

Benchmark and Main Processing

For the main processing steps, there are three approaches taken in addition to the benchmark model. The benchmark model is a Naïve Bayes classifier which is trained on all the samples in a pixel-by-pixel fashion, using a patch size of 48x48. The main approaches in this report use the convolutional neural network approach, which is commonly used in classification and image processing problems, due to its ability to learn features within images and recognize patterns of those features.

The first approach in addition to the benchmark is a pixel-by-pixel classification algorithm using a convolutional neural network (CNN) that examines a patch at a time. Each of these patches varies in size, depending on the size of the filters, whether or not max pooling is used, and the number of layers involved, although it was attempted to make the patch size equivalent to that used by the Naïve Bayes classifier. Generally, the bigger the patch, the better. This method is the slowest of all three and is indeed extremely slow, even on a top-of-the-line modern desktop GPU.

The next technique highlights exactly how slow and wasteful the first technique is. This better technique makes use of what will be referred to as atrous convolutional neural networks (ACNN). Atrous convolution is also known as dilated convolution. The ACNNs produce the identical result as their CNN equivalent, but without repeating calculations. When the CNN attempts to classify 2 pixels which are side-by-side, it will classify them one at a time and so the filter will pass over much of the same data twice. This worsens as the patch size grows. ACNNs are formulated to eliminate this problem. This is accomplished by dilating the kernel after convolutions and max pooling layers and predicting many pixels at the same time. One problem encountered when using ACNNs was the lack of a “dilation rate” option for the two-dimensional max pooling layer in Keras. A fix is discussed later in this report. This technique is much faster than the first, but it is not the fastest.

The final approach used is a U-Net architecture, which is indeed the fastest and least memory intensive of all the approaches. A U-Net architecture uses stages of convolutional filters and max-pooling which encode the data, and then uses stages of “upconvolution” to restore the image to its original size. Connections are made between the encoding and decoding stages in order to help guide the training process. This technique was not found to be as accurate as a patch-wise classification.

After an optimal main processing method is discovered, it is trained further with more detailed adjustments to maximize its final classification performance.

Post-Processing

There were two originally intended approaches for performing post-processing on the data. The first was the use of a Conditional Random Field (CRF). This technique attempts to use the structure of the input data in order to refine classification outlines. The second technique intended for use was the application of a pair-wise conditional generative adversarial network (CGAN). The CGAN was supposed to take an image that was close to correct, but perhaps a bit patchy and clean it up.

Benchmark

As stated above, the benchmark algorithm used is a Naïve Bayes classifier, operating on a patch-wise basis to classify each pixel one at a time. This gives the classifier the maximum number of training samples, as opposed to classifying one image at a time, and should result in the best performance. Figures 3a-c below illustrate the performance achieved.

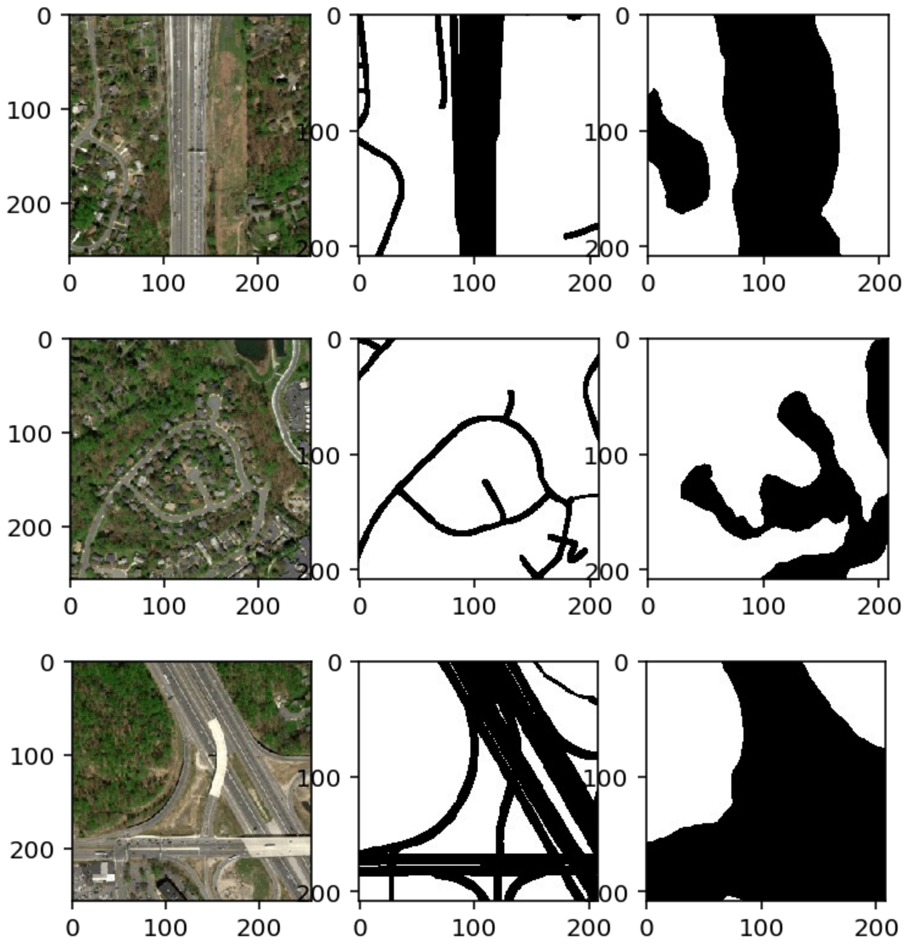

Figure 3a,b,c. Example input images, True roadmaps, Naive Bayes Predicted Roadmaps.

The average IoU for the NB classifier for the test dataset was calculated as: 0.188. A histogram of IoU performance is graphed in Figure 4. Most test images have a prediction with an IoU less than 0.35.

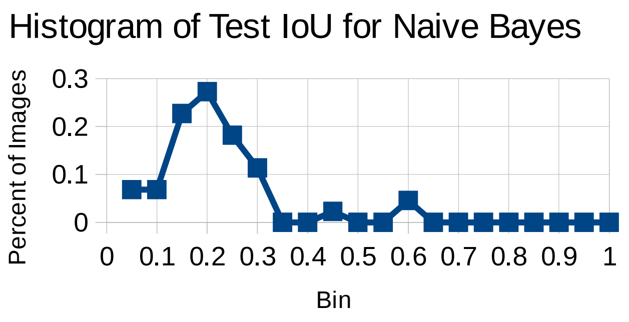

Figure 4. IoU for the NB Classifier.

Data Preprocessing

A prelimary step for preprocessing is organizing and removing outliers. In the original dataset of 500 image/roadmap pairs, there existed 64 images which appeared to be from a separate, lower quality imagery collection and they were removed. The differences in imagery can be seen above in Figures 2a and 2b.

The next step in preprocessing was to enhance the data for feature clarity. This was performed by using the ImageEnhance python module to adjust contrast, brightness, color and sharpness. An example of an original image and the enhanced version of that image are seen in Figures 5a and 5b. Adjusting the color balance gives the image less of a greyish look. The sharpness filter helps undo some of the blur caused by compression on Google's side. The contrast adjustment adds to the sharpness effect. Finally, a brightness effect is intended to allow easier human viewing.

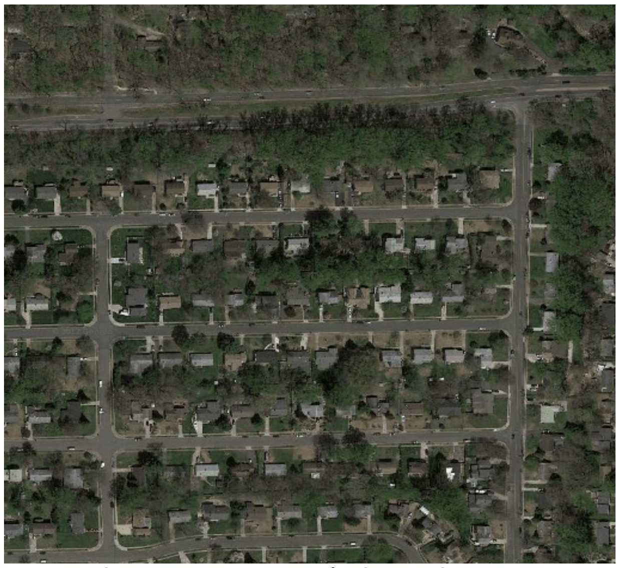
Figure 5a. Example input image.

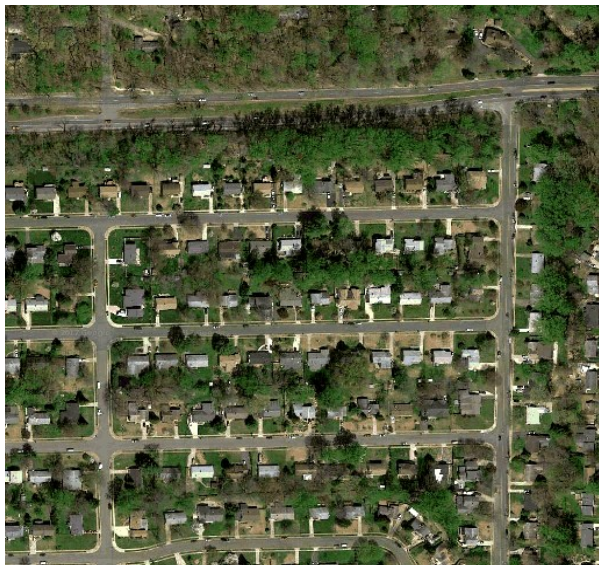
Figure 5b. NB Output.

After enhancing the image, the images are resized from 512x512 to 256x256. This is done to decrease processing load while at the same time further improving the sharpness of the image by effectively decreasing blur. There was some concern about losing image detail, but that was dealt with during data download by adjusting the zoom level so that the coverage area was equivalent.

An attempted step after this was to decrease the number of features for processing by using Principal Component Analysis (PCA). As mentioned above, PCA will attempt to find axes of maximum variance so that the most information can be contained in the least number of components. Any time information is removed that may be valuable, as is the case with all of the RGB data, performance of the classifier will suffer. This was the case as well in this project, despite the PCA being able to provide an explained variance ratio of over 95% with just a single component.

The data was split into separate training, validation and testing sets. The split was 80% training, 10% validation and 10% testing. This means there were 348 training image-roadmap pairs, 44 validation pairs and 44 testing pairs. A random seed was used to ensure the same split would occur each time the project is run.

Implementation

The machine used to train the networks is a home-built PC and the most relevant piece of hardware to the problem at hand is the GPU – an Nvidia GTX1080 Ti with 11 GB of memory.

The software was written using Python 3.6 in both the command-line/script and Jupyter Notebook environments. Tensorflow was first used to build and test networks, but a switch was made to use Keras due its many features which made the bookkeeping aspect of the project much easier, in pretty much every aspect: facilitating training, validation, testing, record keeping, progress monitoring, and readability of code. The NumPy and SKLearn modules were used to prepare data, perform calculations and perform the Naive Bayes classification.

Helper classes were written, which helped managed data, sampling, batch-creation, neural networks, and results visualization. Well-written helper classes were invaluable:

- The sampling management class performed the training/validation/test split.
- The grab_samples class was written to be a single source of patch-wise single-pixel, patch-wise multiple-pixel and whole image data. It was written as an iterator for ease of use and contained options of flattening data, shuffling samples or randomly rotating and flipping whole images.
- The grab_batches class was written to be a single source of batched data for all types of samples generated. It was also written as an iterator, which made it a perfect match for the Keras fit_generator and eval_generator functions.
- Visualization functions were written to accept any type of sampling – whether single-pixel patches, flattened data or whole images. This was necessary to assist in error-checking and ensuring that the datasets and samples were being produced correctly, as well as of course visualizing the roadmaps produced by the networks.
- Network builder and trainer classes were written so that many different styles of networks could be used with minimal specification. A python dictionary describing the networks in general was written and interpreted by the network class which in turn ran the proper methods in sequence for creating a network using Keras. These classes were also used to train the networks and record the weights after each epoch. It's possible to interrupt the training process and resume due to the class finding and loading the most highest epoch weights available.

Metrics and Loss Functions

As mentioned above, four metrics were attempted to be used in the project as loss functions. Firstly, the softmax cross-entropy loss along with class-weighting was used for the CNN networks. When the CNN networks were converted to ACNN networks, the loss metric changed in order to reflect the loss of class-weighting and the new required method of training on an entire image at a time (the CNN could take patches shuffled across the entire dataset), due to it's requirement to work on contiguous data in order to achieve the corresponding efficiency boost. The output layer was converted to a sparse-categorical one and the loss function was custom written. Both intersection-over-union and the related Sorensen-Dice coefficients were programmed, until it was noticed in [10] that the Tversky Index is a generalization of both of these, and as such is the only one that will be shown here.

The Tversky Index can be calculated as [11]:

$$S(X,Y) = \frac{|X \cap Y|}{|X \cap Y| + \alpha|X - Y| + \beta|Y - X|}$$

where X and Y are the true and predicted roadmaps and alpha and beta help control penalizations. Alpha lets you control penalization for false negatives, whereas beta lets you control for false positives. If alpha=beta=1, then the Tversky index is equivalent to IoU. If alpha=beta=0.5, then the Tversky index is equivalent to the Sorensen-Dice coefficient. In the end, the Sorensen-Dice version of this equation was used successfully as a loss function for training, however the intersection-over-union equation is used to provide a final metric. To use any of these metrics as a loss, it is multiplied by -1.

CNN Implementation

The CNNs were built using the above mentioned class. An example of this is as such: a network would be specified layer-by-layer in python dictionary form, such as:

```
self.conv_nets["example_convnet"] = [
  {'type':"c2d",'feat_out':48,
                  'k_sz':[12,12],'k_str':[4,4], 'mp_sz':[3,3],'mp_str':[2,2]},
  {'type':"c2d",'feat_out':96,
                  'k_sz':[4,4],'k_str':[1,1], 'mp_sz':[2,2],'mp_str':[2,2]},
  {'type':"c2d",'feat_out':128,
                  'k_sz':[3,3],'k_str':[1,1], 'mp_sz':[1, 1],'mp_str':[1,1]},
  {'type':"flatten"},
  {'type':"fcn",'feat_out':1024},
  {'type':"dropout"}
]
```

The network builder class would then loop over each layer, building it in Keras based upon the type of layer and the options specified. In this example, there are three 2D convolutional layers, followed by a flatten layer, a fully connected and a dropout layer. Implied after this is the appropriate conversion to a two layer output of road/not-road with a softmax activation layer.

Because the CNNs operated on one patch at a time, which allowed for extreme shuffling, they were able to use a simple optimizer. The shuffling was performed by shuffling patches across all images by using an indexing method to keep track of everything. The loss used was softmax categorical cross entropy loss. This loss is appropriate in this case because there are two classifications (road/not-road) which are mutually exclusive and discrete and because class weights are allowed. The optimizer used was the Keras Adam optimizer with a default learning rate of 0.001 and no decay. Dropout layers are generally used on the last two layers with a universal dropout rate of 0.5

ACNN Implementation

The ACNNs were built in a very similar fashion as the CNNs. In fact, the same network definitions were reused and reinterpreted in the appropriate fashion. Details of the conversion from CNN to ACNN can be found in [1] and [3]. The general idea is that every time down-sampling occurs, whether through a 2D convolution or maxpooling layer with a kernel stride greater than 1, an adjustment is made to the dilation rate of the following layers. The stride of all layers of an ACNN is however 1 in exchange, which means extra processing at the beginning layer occurs. The biggest benefit is achieved at lower layers in the network, as dilation increases and fewer steps are necessary to process all of the data.

A complication arose when training ACNNs which was that the 2D maxpooling layer in Keras did not support a dilation rate, even though there existed a Tensorflow 2D maxpooling layer that did. Because of that, a custom Keras layer was written and is included as Appendix B. The custom layer was not very difficult to write, as is almost identical to the original maxpooling layer. Without this layer, it was impossible to make an equivalent network to the CNNs, however with this in place, it was possible to perfectly recreate the CNN in ACNN form.

Another complications arose when training the ACNNs, due to the fact that they must be trained on one continuous image, as opposed to the CNNs which can be fed entirely independent patches each time. The smaller amount of roads in many images resulted in the networks degenerating to always guessing either “road” or “no-road.” To combat this, the learning rate was reduced from 1e-3 to 1e-5. It was further adjusted up to 5e-5, which was approximately the highest learning rate possible without degenerating.

One more complication arose, which was that class weights are not supported for 2D classifications. This meant that a custom loss function had to be written using the Keras backend. The chosen loss function was at first the Intersection over Union function, however this was changed to the Sorensen-Dice coefficient, which is related to the Intersection over Union metric, and finally the loss function was changed to the Tversky Index, which is a generalization of both of these methods. These are explained below in Metrics. In order to facilitate an easier calculation of this data, specifically a method that didn't require using an argmax function, the classification was converted to a sparse_categorical one, wherein a single integer column is used in place of two one-hot encoded output columns. So for instance, instead of classifying road as [0,1] and not-road as [1,0], they can be classified as 1 and 0, respectively.

CNN/ACNN Network Design

The CNN networks classify a single pixel at a time and the ACNN networks take that network and eliminate repeated calculations when classifying blocks of contiguous pixels in an image. In that sense, the proper approach for this project was to prove the general functionality of the CNN, but switch to the ACNN as quickly as possible in order to save testing time greatly. For this project, each epoch for

CNN data could take up to 1 hr, whereas an epoch for the equivalent ACNN may take only 30 seconds - about a 120X speedup! The ACNNs could accept batches of more than one image, however it was determined that a single image at a time produced the best training.

The CNN and ACNN networks were designed in three stages. The first network attempts were loose copies of the networks from the Tschopp and Mnih papers, in order to prove the architecture of the code and test the performance of these networks on the data. The second network attempts were based on a grid-search approach to finding the best number of convolutional layers and kernel sizes. The third approach was to use a PCA-based approach to identify the optimal kernel sizes and number of stages. This approach was concocted by the author and has not been thoroughly vetted, but appeared to produce the best results of all the networks on this set of data.

The Tschopp and Mnih Networks

The Tschopp network was used for medical image segmentation, whereby borders of neural tissue are identified. The Mnih network was used to identify roads in a large dataset of Massachusetts roads. Like many CNN designs for image data, these networks start with a series of 2D convolutions and max-pooling layers, followed by a flattening and a series of Dense/Fully-Connected layers. They can be seen in Tables 1 and 2.

Layer	Features Out	Kernel Size	Kernel Stride
2D Conv	48	7x7	1x1
Max Pool	-	2x2	2x2
2D Conv	128	5x5	1x1
Max Pool	-	2x2	2x2
2D Conv	192	3x3	1x1
Max Pool	-	2x2	2x2
Dense	1024	10x10	1x1
Dense	512	-	-
Dense	2	-	-
Softmax	-	-	-

Table 1. The Tschopp Network for neural tissue border segmentation. [5]

Layer	Features Out	Kernel Size	Kernel Stride
2D Conv	64	12x12	4x4
Max Pool	-	3x3	1x1
2D Conv	112	4x4	1x1
2D Conv	80	3x3	1x1
Flatten	-	-	-
Dense	4096	-	-
Softmax	-	-	-

Table 2. The Mnih Network for road segmentation on a Massachusetts dataset. [4]

Looking back now upon the design of the Tschopp network, it appears it has a fundamental feature size of 7x7 (the first kernel size) and assumes these features have widths/heights of 4 (the first max-pool stride*size). Further layers imply that combinations of these features up to a patch size of 100 are important. To determine the patch size, a calculation can be made in reverse from a single pixel output layer up to the input to the first layer.

For this calculation, kernel sizes in reverse require a (k_size – 1) increase in patch size. Striding in reverse has the effect of a multiplication. Because all kernel sizes and strides have heights that match the widths, this calculation only needs to be performed once.

For the Tschopp network:

Patch Size 1D = (7-1) + 2*((5-1) + (2-1) + 2 *((3-1) + (2-1) + 2 * ((9-1) + 1))) = 100

Patch Size 2D = <u>100x100</u>

For the Mnih Network:

Patch Size 1D = 4 * ((12-1) + 3 * ((4-1) + (3-1))) = 104

Patch Size 2D = <u>104x104</u>

The graphs in Figures 6 and 7 show the results from the Tschopp and Mnih networks. Their performance is very good. The first thing to notice is how the test curve is above the validation curve. This implies that the test data likely has more in common with the test data than the validation data does. For the training curve to be above the others is expected, as the network is learning about what's inside that specific dataset. The Mnih network appears to maintain much better generality when training, because its training performance does not quickly out-perform the validation curve as it does in the Tschopp network. This could be due to the larger kernel sizes in the first layer of the Tschopp network. It also appears that the Mnih convnet could improve more over time, however the Tschopp network looks like it may have nearly reached maximum performance.

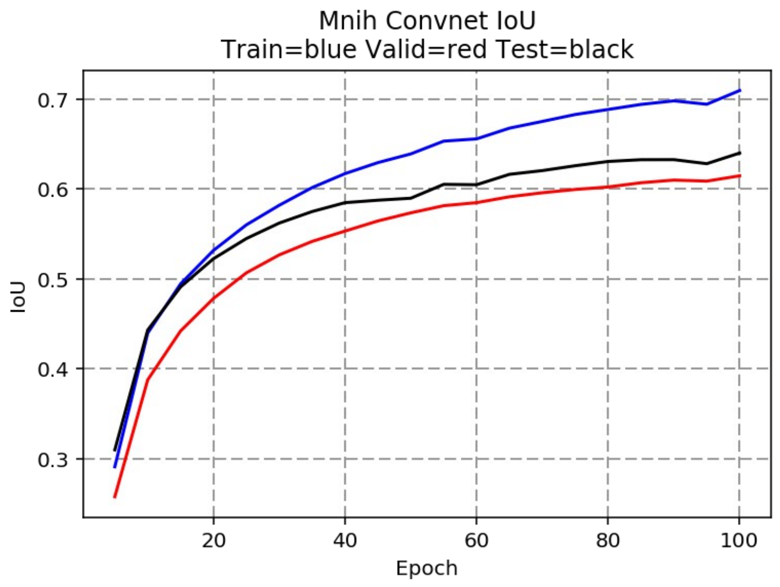

Figure 6. Mnih Convnet Train, Validation, and Test IoU –
Max Validation IoU = 0.615

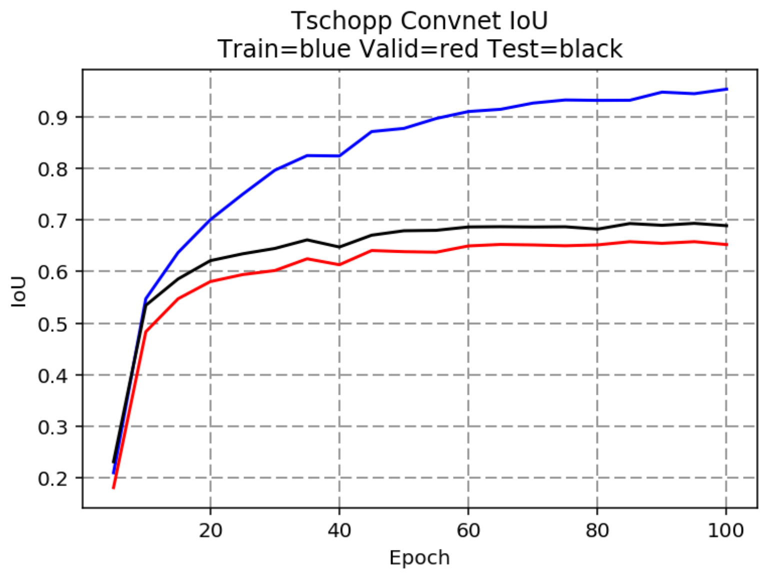

Figure 7. Tschopp Convnet Train, Validation, and Test IoU –
Max Validation IoU= 0.658

The Grid-Search Networks

The grid search networks are similar in design to the Tschopp and Mnih networks and progress from a single convolutional layer to a three-tier convolutional layer. For each layer architecture, a small, medium and large network was built, where these terms refer to the kernel sizes and number of features. Additionally, all of these networks were run with maxpooling disabled. The networks can be seen in Tables 3-11. Each layer except the final one uses ReLU activation.

Layer	Feat	K Size	K Stride
2D Conv	32	4x4	1x1
Max Pool	-	2x2	2x2
Flatten	-	-	-
Dense	1024	-	-
Dense	256	-	-
Dense	2	-	-
Softmax	-	-	-

Table 3. Convnet 1x2 Small.

Layer	Feat	K Size	K Stride
2D Conv	64	8x8	1x1
Max Pool	-	2x2	2x2
Flatten	-	-	-
Dense	2048	-	-
Dense	512	-	-
Dense	2	-	-
Softmax	-	-	-

Table 4. Convnet 1x2 Medium.

Layer	Feat	K Size	K Stride
2D Conv	128	16x16	1x1
Max Pool	-	2x2	2x2
Flatten	-	-	-
Dense	4096	-	-
Dense	1024	-	-
Dense	2	-	-
Softmax	-	-	-

Table 5. Convnet 1x2 Large.

Layer	Feat	K Size	K Stride
2D Conv	32	4x4	1x1
Max Pool	-	2x2	2x2
2D Conv	64	4x4	1x1
Max Pool	-	2x2	2x2
Flatten	-	-	-
Dense	1024	-	-
Dense	256	-	-
Dense	2	-	-
Softmax	-	-	-

Table 6. Convnet 2x2 Small.

Layer	Feat	K Size	K Stride
2D Conv	64	8x8	1x1
Max Pool	-	2x2	2x2
2D Conv	128	8x8	1x1
Max Pool	-	2x2	2x2
Flatten	-	-	-
Dense	2048	-	-
Dense	512	-	-
Dense	2	-	-
Softmax	-	-	-

Table 7. Convnet 2x2 Medium.

Layer	Feat	K Size	K Stride
2D Conv	128	16x16	1x1
Max Pool	-	2x2	2x2
2D Conv	256	8x8	1x1
Max Pool	-	2x2	2x2
Flatten	-	-	-
Dense	4096	-	-
Dense	1024	-	-
Dense	2	-	-
Softmax	-	-	-

Table 8. Convnet 2x2 Large.

Layer	Feat	K Size	K Stride
2D Conv	32	4x4	1x1
Max Pool	-	2x2	2x2
2D Conv	64	4x4	1x1
Max Pool	-	2x2	2x2
2D Conv	128	4x4	1x1
Max Pool	-	2x2	2x2
Flatten	-	-	-
Dense	1024	-	-
Dense	256	-	-
Dense	2	-	-
Softmax	-	-	-

Table 9. Convnet 3x2 Small.

Layer	Feat	K Size	K Stride
2D Conv	64	8x8	1x1
Max Pool	-	2x2	2x2
2D Conv	128	8x8	1x1
Max Pool	-	2x2	2x2
2D Conv	256	8x8	1x1
Max Pool	-	2x2	2x2
Flatten	-	-	-
Dense	2048	-	-
Dense	512	-	-
Dense	2	-	-
Softmax	-	-	-

Table 10. Convnet 3x2 Medium.

Layer	Feat	K Size	K Stride
2D Conv	128	16x16	1x1
Max Pool	-	2x2	2x2
2D Conv	256	8x8	1x1
Max Pool	-	2x2	2x2
2D Conv	512	8x8	1x1
Max Pool	-	2x2	2x2
Flatten	-	-	-
Dense	4096	-	-
Dense	1024	-	-
Dense	2	-	-
Softmax	-	-	-

Table 11. Convnet 3x2 Large.

The graph in Figure 8 show the results from the Grid-Search networks. This graph differs from the Mnih and Tschopp graphs in that it just plots the maximum validation IoU across network designs. Note how the difference in performance improves for each network architecture as the number of features increases. Also note how adding a layer to the architecture always increases validation performance, given the same number of features in the other layers.

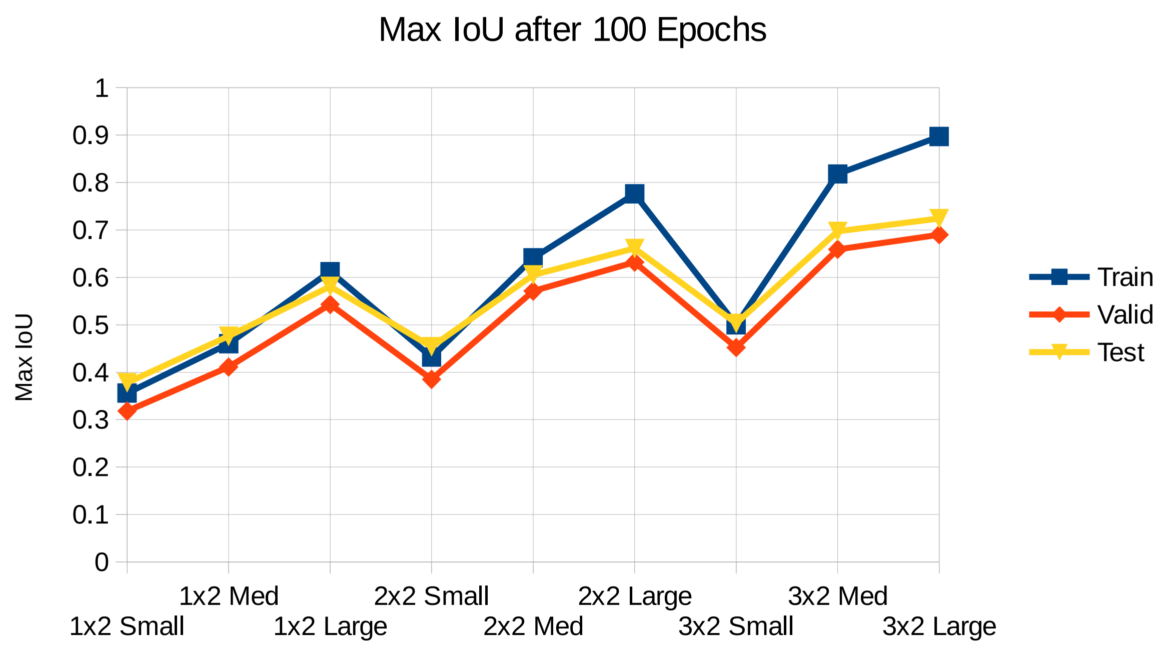

Figure 8. Maximum Validation IoU for each Grid-Search network.

As the network achieves greater than 0.5 IoU, separation between the training and the validation and testing curves begins to increase. Just as in the Mnih and Tschopp network results, the testing data performance is greater than the validation performance, which is likely due to it being more similar to the training data than the validation data is. This graph indicates that a 3x2 network with features somewhere between those in the Medium and Large networks could offer a good solution.

U-Net Implementation

The U-Net architecture is designed such that the entire roadmap is predicted, as opposed to a context-bordered subset, such as is the case in the unpadded CNN and ACNN implementations. Because the entire image is predicted at once, the U-Net architecture is the fastest implementation, however it's also likely to be the least accurate, as it is predicted roads along the borders, especially in the corners of the image, where only a quarter of the otherwise available information is present.

The U-Net architecture makes use of "skip connections" - connections across the network from encoder layers to decoder layers of equal dimensions. The Keras Sequential() model for network building cannot be used in this case - instead the Keras functional API must be used. This required a slight rewrite to the networking building code and the ability to specify which layers should connect with which other layers.

U-Net Network Design

The U-Net architecture used is taken from the Pix2Pix paper [6] and features a series of 2D convolutional layers with a stride of 2 until the network reaches a 1x1 non-feature dimension. Each convolutional layer is followed by a batch normalization layer. The number of features increases as the primary dimensions shrink. This is considered the encoding phase. The decoding phase is the opposite – 2D upconvolution layers are applied in series, with their outputs merged with the appropriate output layers from the encoder.

Three versions of the original Pix2Pix U-Net were tested, where the number of features were adjusted to explore performance on the dataset. The original architecture is seen in Tables 12 – the other two architectures are identical except the features out are divided by either 2, 4 or 8. All of the layers use a kernel size of 4 and a stride of 2. Each layer except the final one uses ReLU activation. The final layer's activation function was changed from the tanh for a GAN implementation to a sigmoid.

The performance for the Pix2Pix networks is shown in Figure 9. Relative to the other networks, the performance is low and will not be further pursued. However, this network could be very useful in cases where only approximate labeling is required in a very short amount of time.

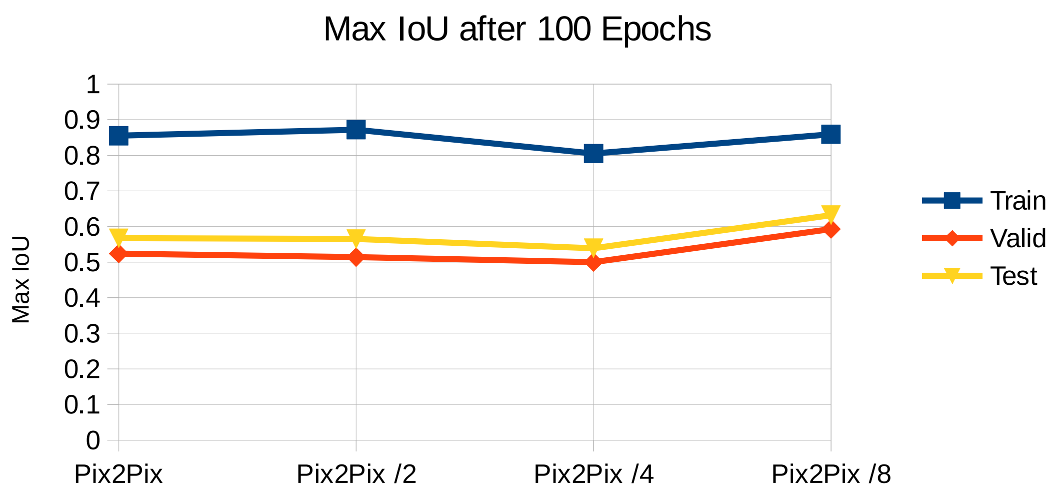

Figure 9. Maximum Validation IoU for each Pix2Pix network.

Name	Layer	Features Out	Kernel Size	Kernel Stride	Merge With
Encoder 1	2D Conv	64	4x4	2x2	-
	Batch Norm	-	-	-	-
Encoder 2	2D Conv	128	4x4	2x2	-
	Batch Norm	-	-	-	-
Encoder 3	2D Conv	256	4x4	2x2	-
	Batch Norm	-	-	-	-
Encoder 4	2D Conv	512	4x4	2x2	-
	Batch Norm	-	-	-	-
Encoder 5	2D Conv	512	4x4	2x2	-
	Batch Norm	-	-	-	-
Encoder 6	2D Conv	512	4x4	2x2	-
	Batch Norm	-	-	-	-
Encoder 7	2D Conv	512	4x4	2x2	-
	Batch Norm	-	-	-	-
Encoder 8	2D Conv	512	4x4	2x2	-
	Batch Norm	-	-	-	-
Decoder 1	2D Conv	512	4x4	2x2	-
	Batch Norm	-	-	-	Encoder 7
Decoder 2	2D Conv	512	4x4	2x2	-
	Batch Norm	-	-	-	Encoder 6
Decoder 3	2D Conv	512	4x4	2x2	-
	Batch Norm	-	-	-	Encoder 5
Decoder 4	2D Conv	512	4x4	2x2	-
	Batch Norm	-	-	-	Encoder 4
Decoder 5	2D Conv	256	4x4	2x2	-
	Batch Norm	-	-	-	Encoder 3
Decoder 6	2D Conv	128	4x4	2x2	-
	Batch Norm	-	-	-	Encoder 2
Decoder 7	2D Conv	64	4x4	2x2	-
	Batch Norm	-	-	-	Encoder 1
Final Out	2D Conv	1	4x4	2x2	-
	Sigmoid	-	-	-	-

Table 12. Pix2Pix U-Net

PCA-based Networks

The term PCA-based is used here to mean networks that have been designed using results from PCA analysis of the data. To perform this analysis, the dataset was first converted to a single component using a pre-processing PCA. This was done to speed up the analysis and to ensure the data could be flattened appropriately for the block-wise PCA analysis. Next, kernel sizes were chosen ranging from 8 to 48 in steps of 8 and also 4 to 16 in steps of 2. The dataset was partitioned into blocks of the kernel size, flattened, and stacked together in a matrix. PCA was performed with number of components equal to the square of the kernel size. The explained variance ratios were cumulatively summed and graphed as shown in Figures 10 and 11. Note that the x-axis is the ratio of feature count to the maximum features available.

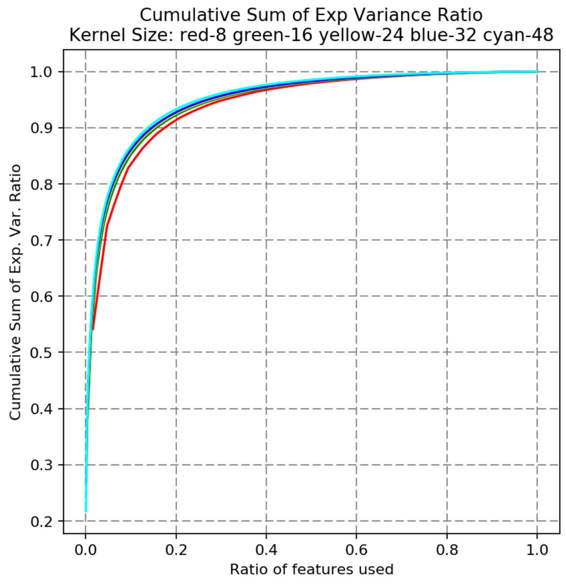

Figure 10. Cumulative Sums of Explained Variance Ratio for various Kernel Sizes

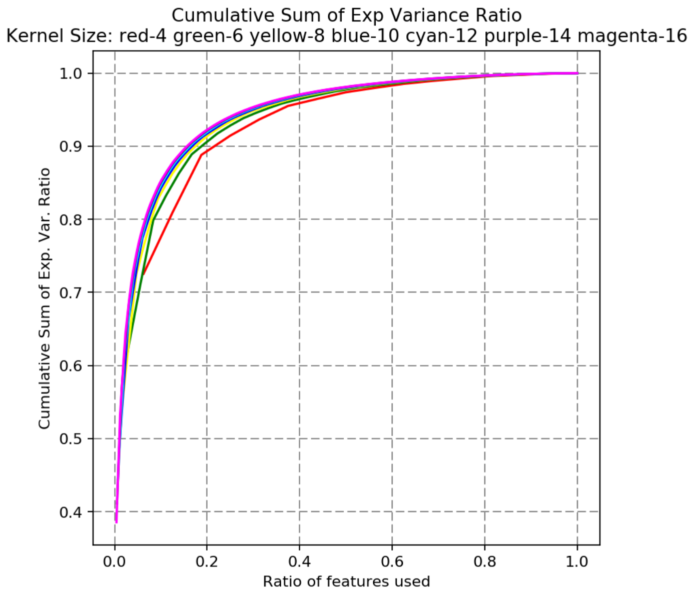

Figure 11. Cumulative Sums of Explained Variance Ratio for various Kernel Sizes

The graphs show that no particular kernel size is much better than another at recreating the data, which suggests that there should be no kernel size better than another for analyzing the data with the neural network.

The next step in PCA analysis then is to try to determine how well a PCA trained on the training data can recreate the validation data. The noise in dB for recreations of the validation data after PCA is trained with various kernel sizes is shown in Figure 12.

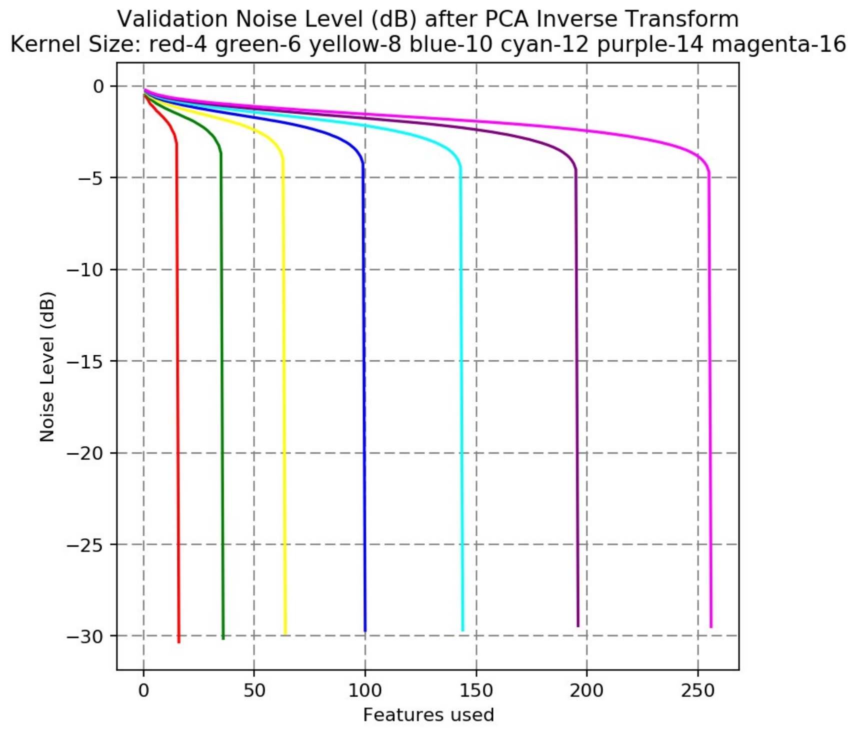

Figure 12. Validation Noise Level after PCA Inverse Transform.

The most critical observation is that no matter which kernel size you choose, the number of features really matters! In this case, despite the components being arranged in order of most explained variance first, having every single feature represented, including the last one, makes a huge difference.

In order to generalize the network, however, it's important to choose a smaller kernel size. What is also important is that there appears to be a minimum ratio of feature space to the square of the kernel size that should be observed when building the networks, depending on the accuracy of representation desired. Because the neural network uses more than one layer, this ratio should be considered an upper-bound. According to Figures 6 and 7, the ratio is about 0.25 for 90% fit and 0.5 for 95% fit. So, for instance a 16x16 kernel was used with 256 components, but only 128 are necessary to produce a 95%-accurate representation. However, according to Figure 8 the full 256 components should be used. Because the data analyzed by the network has all three colors, this should be increased roughly by a factor of 3 at least. As stated, this should be considered an upper bound, because the network will be able to formulate for example stripes in later stages from a single line in the first stage. This should reduce the number of required features greatly. Therefore, the number of features used in the first layer will be calculated as features = kernel_size * kernel_size * 3 (for rgb) / 2 (multi-layer factor).

There are three PCA-based networks designed for this project. They are shown in Table 13-15. Note that they all employ a similar architecture to the 3x2 Convnet above with an additional fully connected layer added, whereby 3 convolutional-maxpooling layers are followed by 3 dense layers, which is then followed by another dense layer as the output. ReLUs are used as activation functions after each convolution or dense layer.

Layer	Features Out	Kernel Size	Kernel Stride
2D Conv	150	10x10	1x1
Max Pool	-	2x2	2x2
2D Conv	120	4x4	1x1
Max Pool	-	2x2	2x2
2D Conv	192	4x4	1x1
Max Pool	-	2x2	2x2
Flatten	-	-	-
Dense	2800	-	-
Dense	280	-	-
Dense	140	-	-
Softmax	-	-	-

Table 13. PCA-Based Network 1

Layer	Features Out	Kernel Size	Kernel Stride
2D Conv	96	8x8	1x1
Max Pool	-	2x2	2x2
2D Conv	346	6x6	1x1
Max Pool	-	2x2	2x2
2D Conv	554	6x6	1x1
Max Pool	-	2x2	2x2
Flatten	-	-	-
Dense	2800	-	-
Dense	1400	-	-
Dense	140	-	-
Softmax	-	-	-

Table 13. PCA-Based Network 2

Layer	Features Out	Kernel Size	Kernel Stride
2D Conv	54	6x6	1x1
Max Pool	-	2x2	2x2
2D Conv	346	6x6	1x1
Max Pool	-	2x2	2x2
2D Conv	554	6x6	1x1
Max Pool	-	2x2	2x2
Flatten	-	-	-
Dense	840	-	-
Dense	420	-	-
Dense	210	-	-
Softmax	-	-	-

Table 13. PCA-Based Network 3

The following graph in Figure 13 shows the results from the PCA-Based networks.

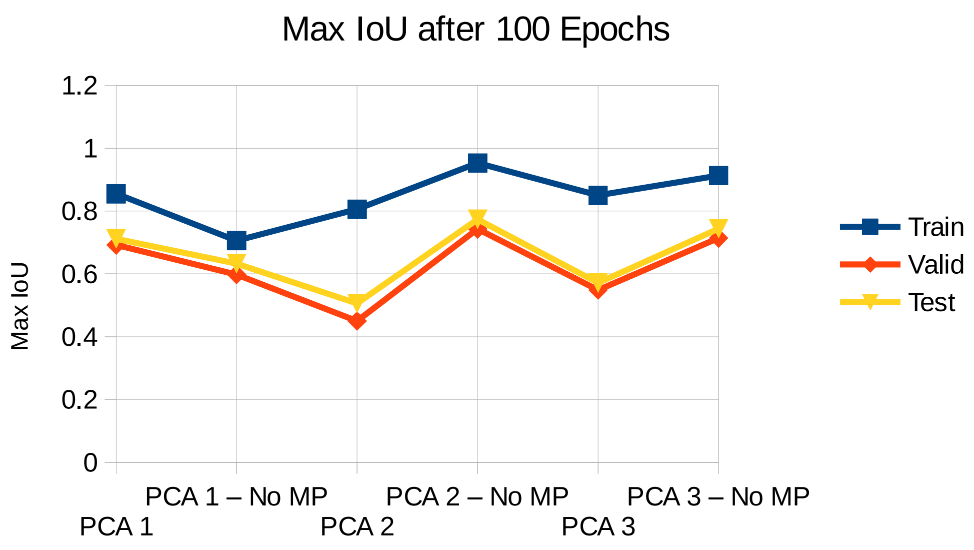

Figure 13. Maximum Validation IoU for each PCA-Based network.

The networks were all trained with max-pooling layers enabled and disabled. An interesting note is that PCA-Network 2 did not converge with max-pooling enabled when it was fed a single image at a time; two images per batch were required.

The best performing network was PCA2- No MP, which is the PCA-Network 2, with max-pooling disabled.

Refinement

The best network identified so far, based on highest validation IoU is the PCA-Based Network 2 without maxpooling layers. The refinement steps considered at this point are batch normalization, learning rate decay and continued training for more epochs.

Batch Normalization was added to the input of any convolutional layers in order to improve performance, as proposed in [8]. The results of this run are seen in Figure 14. Because this performance is not improved, it was removed from the design of the algorithm. An interesting effect of batch normalization seems to be that the test performance now closely matches the validation performance, allow they both perform worse than without the batch normalization.

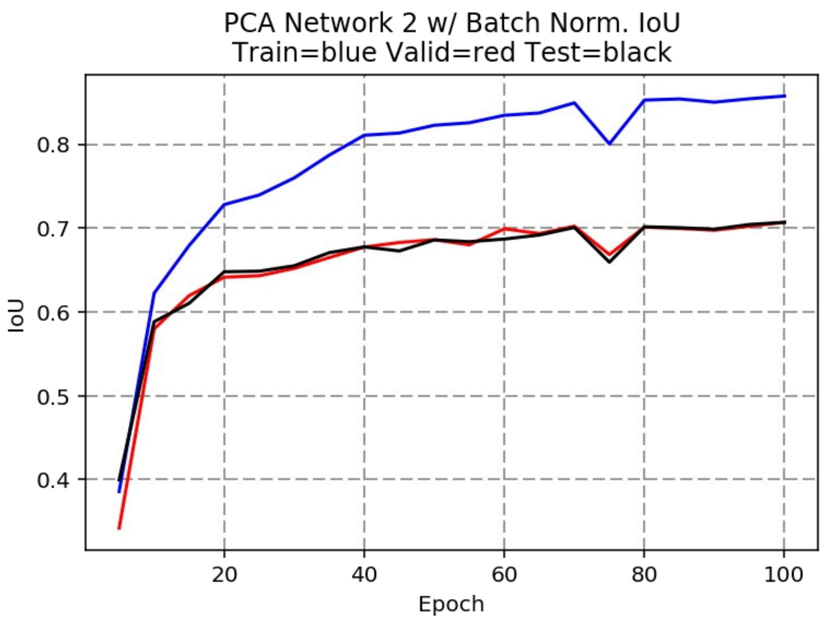

Figure 14. Batch Normalization Train, Validation, and Test IoU –
Max Validation IoU= 0.707

Learning rate decay is not being considered, due to the smoothness of the curve, which indicates that the network is not significantly suffering from back-and-forth corrections.

The network was trained further and its extended graph is shown in Figure 15. _*The final maximum validation IoU is 0.743 and the maximum final test output is 0.774.*_ This performance is very strong compared to the other networks, including the Mnih and Tschopp networks, which were unable to exceed an IoU of 0.7 on the test data within 100 epochs (this network achieved that benchmark prior at approximately 25 epochs).

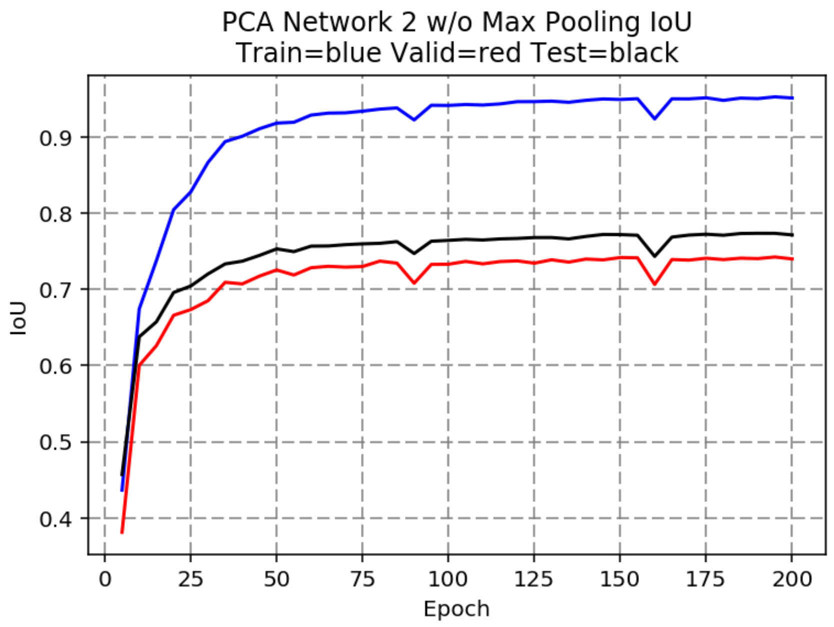

Figure 15. Best Model Convnet Train, Validation, and Test IoU –
Max Validation IoU= 0.743, Max Test IoU = 0.774

Example satellite images and their corresponding roadmaps as generated by the network are shown in Figures 17a,b,c. While not perfect, the predictions are of high quality, especially relative to the Naive Bayes classification algorithm. As can be seen from the histogram in Figure 16, the most frequent IoU is around 0.9, which is outstanding.

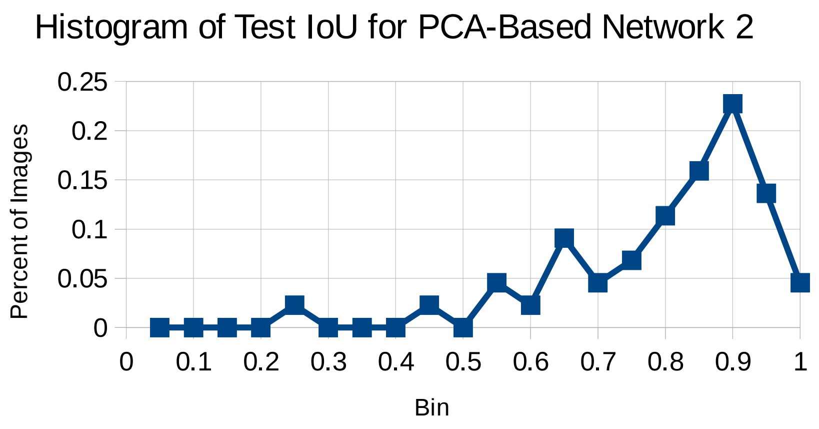

Figure 16. IoU for the PCA-Based Network 2.

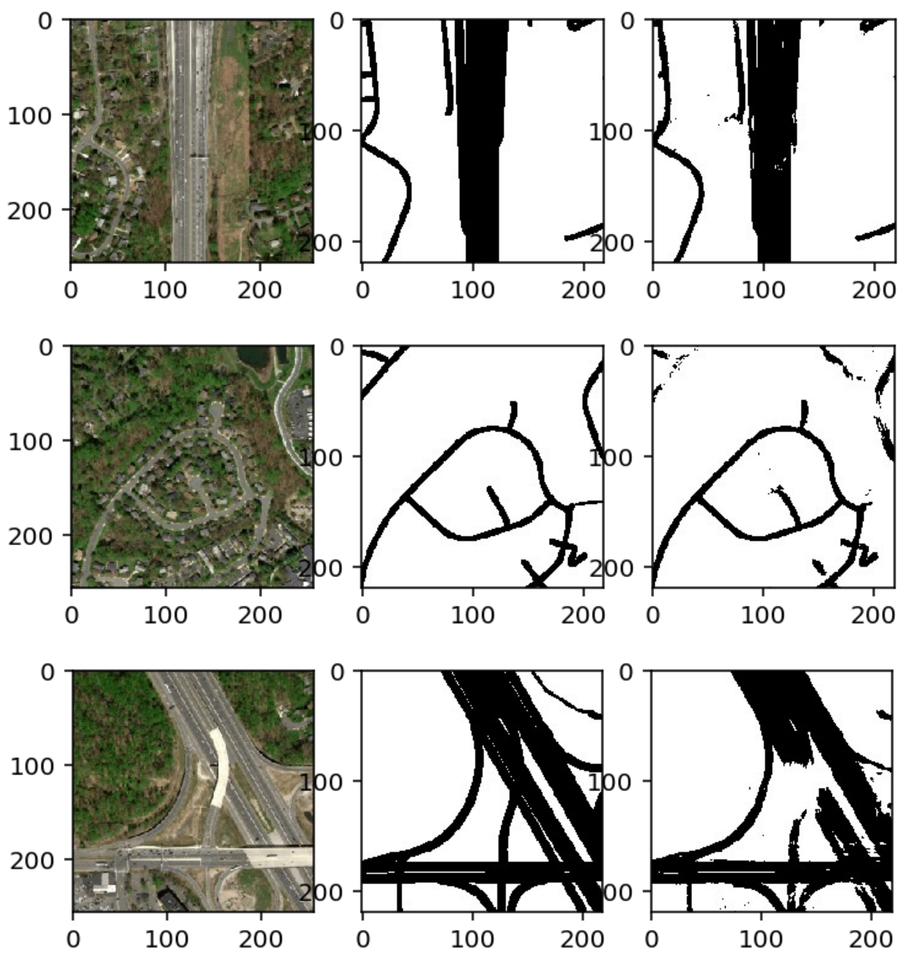

Figure 17a,b,c. Example input images, True roadmaps, Best Network Predicted Roadmaps.

Post-Processing Refinement

Post-processing was considered for this project, however a proper solution was not discovered. The reason for this is likely the nature of shortcomings for the networks. What was originally expected to be an issue were faded or spotty roads, along with spots around the image. However, the main issues were seemingly inescapable problems, such as a very narrow road that is completely covered by foliage in the satellite imagery and therefore not recognized by the network. These types of missed labeling are deemed to be acceptable for the purposes of this project, which has a scope of identifying roads from satellite imagery.

Conditional Random Fields – this was briefly tested, but it was found because the CRF algorithm depends on pixel intensity comparisons between neighboring pixels, it acted more to remove the roads than clean them up. This can be seen in Figure 18. The code used to create this implementation is taken from the pydensecrf module, which is a python wrapper on the original Krähenbühl and Koltun code written in C++ [12].

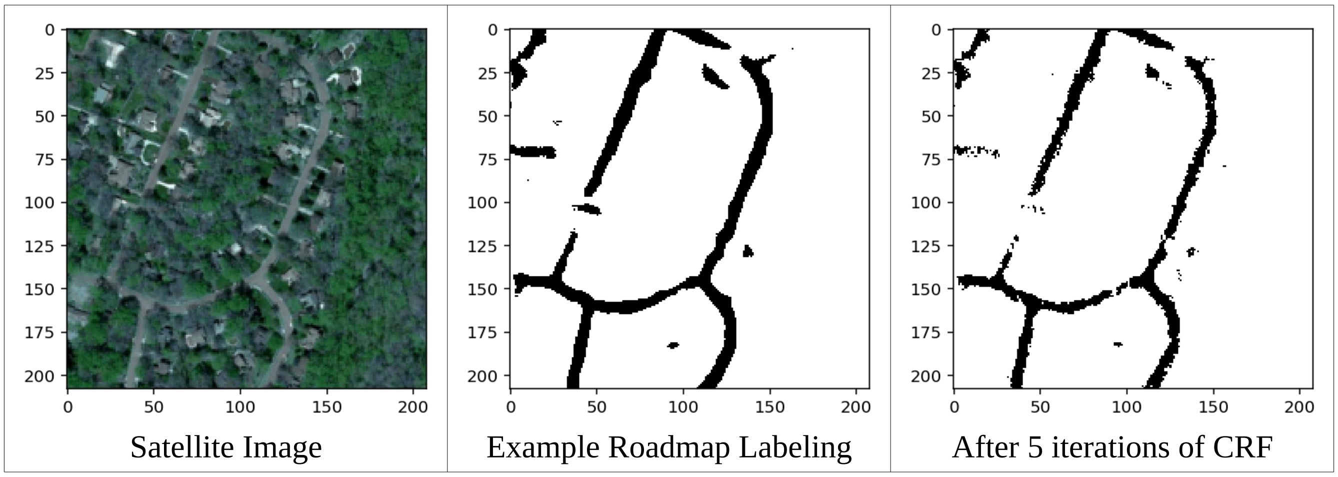

Figure 18. CRF example. Mis-categorized specks remain while correct roadways disappear.

It's possible and likely that more sophisticated structured analysis methods exist, but they seemed out of the scope of this project.

Conditional GAN – this method was considered until it was seen how the U-Net network from the Pix2Pix paper [6] performed poorly in comparison to the other algorithms. Another issue was that after many epochs, the networks were performing much better on the training data than the validation data, which meant that there were few corrections for a CGAN to make. The original idea was to feed in pairs of true roadmaps and predicted roadmaps so that the network could learn to produce truer roadmaps from predicted roadmaps. Some code was written for this, but it was not performed.

Conclusion

The final model selected here, the PCA-Based Network 2, which made use of the ACNN architecture had an overall IoU on the test data of 0.774, outperformed the selected benchmark Naive Bayes model, which had an IoU of 0.188, by a very wide margin. It also outperformed other models developed for similar tasks by Mnih and Tschopp, which were developed under the guidance of professors in a university setting. Its high metric performance is perhaps its top quality.

Second to that is its great speed. It can predict approximately 6 images per second using an un-optimized, Keras-based architecture.

The ACNN architecture was not as fast as the U-Net architecture, but it was much more accurate. Given a satellite image at a comparable resolution of meters-per-pixel of a size greater than the context size required, it should be able to identify visible roads within the contextual borders of that image. It is not able to discover all roads, especially those hidden by foliage.

Reflection

I learned many things in this project – many, if not most, of them being of a practical nature.

When it comes to retrieving and loading data, I learned about Google's poorly-documented scale feature that allows a user to download twice the advertised image size, which allows for much improved resolution.

When it comes to preprocessing data, I learned that you may end up with far less data than you original thought you had, due to fundamental issues with parts of it, which was the case in the original dataset, where approximately 15% of the data had a drastically different look due to lighting and shadow effects. I learned that clarity/quality of the features matters and one way to improve imagery is by resizing to decrease blur caused by compression. I learned that PCA explained variance ratios may be very misleading, because even though 95% of the total dataset variance may be explained, the remaining 5% may have a very big effect on error, depending on the metric used. Therefore, PCA should be used sparingly, or in cases where there is room for error in exchange for speed. Histogram analysis was useful when it came to picking class weights. As far as padding goes, not much was learned because only valid padding was used, however it must be the case that true data is superior to padding with zeros or using mirror algorithms.

For handling and processing data, having a sample generator and batch generator class is extremely valuable and works well with the Keras architecture. It also helps to keep the code readable and compressed. Another major benefit of using Keras was that there are very easy ways to save weights as epochs progress. This allowed me to interrupt a training session of 100 epochs without losing progress. This seems very important to incorporate in any data-/time-intensive machine learning project.

Some observations about the Naive Bayes classifier are that it takes a long time to fit the data, relative to a U-Net architecture, or a low-epoch network training of comparable performance, and its performance in this case was not very good at all, even when classifying individual pixels based on a patch size. It is an example of how not paying attention to structure in the data can hurt performance.

Learning about the Tversky Index was very exciting, because it allows for flexible weighting of FP and FN that someone can use when they want to emphasize the performance of one over the other. It's also great because by using certain values of alpha and beta, it becomes equivalent to the popular Dice Coefficient or Intersection Over Union metrics.

When working with classic CNNs on a pixel-wise task, it was extremely time-consuming. It does have the advantage of being able to randomly sample across the entire dataset and use class weights, which led to easier setup and more easily achieved stability in the training phase. It was good as a proof-of-concept, but I would never start with it again, now that I know the code for an Atrous CNN.

The day that I got the ACNNs working was a great day, because training that took 1 hour per epoch with a classic CNN took around 30-60 seconds with an ACNN. Converting to ACNNs was disheartening at first, because no convergence was occurring, due to the lopsidedness of the data. Once the proper loss function was discovered, this was solved. One trade-off is that it can't randomly sample as well as the CNN, because the samples by definition need to be contiguous, but this problem can be handled by lowering the learning rate so that overly ambitious steps early on can be avoided. Having the tremendous speedup made it easier to experiment with different networks - it still took around 1.5 hours to train a network for 100 epochs.

Writing the custom Keras layer for the ACNN's so that maxpooling with a dilated kernel could be represented seemed daunting at first, but was actually not too difficult. One surprise is that the original MaxPooling2D layer calls tf.nn.pool2d, which does not have a dilation rate option, however another Tensorflow layer, tf.nn.pool, does. I assume this is due to constantly changing code.

I learned that U-Nets are extremely fast extremely fast. Because they had to be built with Keras' functional API, I learned about how that process works, including how to merge outputs from different layers. When I tried experimenting with removing the deepest layer, the algorithm seemed to fall apart, and I wasn't able to figure out why.

I learned a modest amount from my post-processing attempts, although I think what I learned helped me understand the results from the Chen paper [1] a bit better. I understand that there may be more advanced structured learning algorithms that might have helped me connect roads with breaks in them, for instance. One of the modules I experimented slightly with was the PyStruct module [14] and its EdgeFeatureGraphCRF method, however I wasn't able to understand it completely.

The CGAN approach didn't seem helpful, unless there was a lot more training data available, because it seemed like the network learned the training dataset very well and left little room for post-processing improvements, or improvements that would be out of the range of a CGAN to fix – such as roads that are completely hidden by foliage and therefore unlabeled by the network. I was concerned that the CGAN would actually learn about these and add them seemingly at random.

Besides learning exactly what I wanted to learn in this project – how image segmentation works - a big thing I'm taking away from the project is research practice. Being able to benefit from the time, effort, and insights of others is truly invaluable and I'm very grateful for each research paper I read while doing this project – in addition to the Udacity lessons and example notebooks, of course!

References

1. L. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille. “Semantic Image Segmentation with Deep Convolutional Nets and Fully Connected CRFs”. In: ArXiv e-prints (June, 2016). arXiv:1412.7062 [cs.CV].

2. Google. Styled Maps – Google Static Maps API. https://developers.google.com/maps/documentation/static-maps/styling

3. H. Li, R. Zhao and X.Wang. “Highly Efficient Forward and Backward Propagation of Convolutional Neural Networks for Pixelwise Classification”. In: ArXiv e-prints (Dec., 2014). arXiv:1412.4526 [cs.CV].

4. V. Mnih, G. Hinton. "Learning to Detect Roads in High-Resolution Aerial Images." In: CiteSeerX (Sep, 2010) (http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.178.7318)

5. F. Tschopp. "Efficient Convolution Neural Networks for Pixelwise Classification on Heterogeneous Hardware Systems." In: ArXiv e-prints (Sep, 2015). arXiv:1509.03371 [cs.CV].

6. P. Isola, J. Zhu, T. Zhou, A. Efros. “Image-to-Image Translation with Conditional Adversarial Networks.” In: ArXiv e-print (Nov, 2016). arXiv:1611.07004 [cs.CV].

7. V. Badrinarayana, A. Kendall, R. Cipolla. “SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation.” In:ArXiv e-prints (Oct, 2016). arXiv:1511.00561v3 [cs.CV].

8. S. Ioffe, C. Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.” In: ArXiv e-prints (Mar, 2015). arXiv:1502.03167v3 [cs.LG].

9. J. Yang, D. Zhang, A. Frangi, J. Yang. “Two-Dimensional PCA: A New Approach to Appearance-Based Face Representation and Recognition.” Pattern Analysis and Machine Intelligence, IEEE Transactions on. vol. 26, no. 1, pp. 131-137, Jan. 2004.

10. S. Salehi, D. Erdogmus, A. Gholipour. “Tversky loss function for image segmentation using 3D fully convolutional deep networks.” In: ArXiv e-prints (Jun, 2017). arXiv:1706.05721v1[cs.CV].

11. Wikipedia contributors. "Tversky index." *Wikipedia, The Free Encyclopedia*. Wikipedia, The Free Encyclopedia, 5 Mar. 2017. Web.

12. P. Krähenbühl, V. Koltun. “Efficient Inference in Fully Connected CRFs with Gaussian Edge Potentials.” In: ArXiv e-prints (Oct, 2012). arXiv:1210.5644 [cs.CV].

13. PyDenseCRF: https://github.com/cdmh/deeplab-public/tree/master/densecrf

14. PyStruct: https://pystruct.github.io/

Appendix A – Retrieving, “Cleaning,” and Enhancing the Satellite Imagery and Road Maps.

A script was used in each case to retrieve, clean and enhance the satellite imagery and road maps. Retrieving the data involved downloading it using the Google Maps API through the use of URLs. Cleaning the data was simply a stage where the Google Maps logo was removed from the bottom of the images, a necessary step so that the algorithm didn't learn the logo! The final step was enhancing, as described in the preprocessing step. The scripts to accomplish these three steps are shown below:

grab_17_data.py

```
import os.path
from io import BytesIO
from PIL import Image
import urllib
startlat = 38.81
endlat = 38.86
startlon = -77.30
endlon = -77.14
steps = 22.0 # total images = steps * steps
latstep = (endlat - startlat) / steps
lonstep = (endlon - startlon) / steps
size = "512x564"
zoom = 17
scale = 2

lat = startlat
lon = startlon
while lat < endlat:
  while lon < endlon:
    url_satellite = "http://maps.googleapis.com/maps/api/staticmap?center=%s,%s&size=
%s&zoom=%s&scale=%s&sensor=false&maptype=satellite&style=feature:all|element:labels|
visibility:off&format=png8&key=AIzaSyAIatajXC9oQUFMaEZXRbzsgG1V6BGArms" %
(lat,lon,size,zoom,scale)
    url_roadmap = "http://maps.googleapis.com/maps/api/staticmap?center=%s,%s&size=%s&zoom=
%s&scale=%s&sensor=false&maptype=roadmap&style=feature:all|element:labels|
visibility:off&style=feature:administrative|element:all|
visibility:off&style=feature:landscape|element:all|visibility:off&style=feature:poi|
element:all|visibility:off&style=feature:transit|element:all|
visibility:off&style=feature:water|element:all|visibility:off&style=feature:road|
element:geometry|color:white&key=AIzaSyAIatajXC9oQUFMaEZXRbzsgG1V6BGArms" %
(lat,lon,size,zoom,scale)
    file_satellite = "./original_scale2/map_%s_%s_satellite.png" % (lat,lon)
    file_roadmap = "./original_scale2/map_%s_%s_roadmap.png" % (lat,lon)
    #print url_satellite
    print "%s %s" %(lat,lon)
    if not os.path.isfile(file_satellite):
      #print "Grabbing sat image...",url_satellite
      buffer = BytesIO(urllib.urlopen(url_satellite).read())
      #print "Saving..."
      image = Image.open(buffer)
      image.save(file_satellite)
      del image
      del buffer
    else:
        print "File exists, skipping"
    if not os.path.isfile(file_roadmap):
      buffer = BytesIO(urllib.urlopen(url_roadmap).read())
      image = Image.open(buffer)
      image.save(file_roadmap)
      del image
      del buffer
    lon += lonstep
    #break
  lon = startlon
  lat += latstep
  #break
```

clean_17_data.py

```
import os.path
from PIL import Image
startlat = 38.81
endlat = 38.86
startlon = -77.30
endlon = -77.14
steps = 12.0 # total images = steps * steps
latstep = (endlat - startlat) / steps
lonstep = (endlon - startlon) / steps
size = "512x564"
zoom = 17
cropsize = 512

lat = startlat
lon = startlon
while lat < endlat:
  print lat
  while lon < endlon:
    #print "%s %s" %(lat,lon)
    orig_satellite = "./original/map_%s_%s_satellite.png" % (lat,lon)
    orig_roadmap = "./original/map_%s_%s_roadmap.png" % (lat,lon)
    if os.path.isfile(orig_satellite) and os.path.isfile(orig_roadmap):
      print "%s %s" %(lat,lon)
      im_satellite = Image.open(orig_satellite).crop(box=(0,0,cropsize,cropsize))
#.convert(mode="L")
      im_roadmap = 
Image.open(orig_roadmap).crop(box=(0,0,cropsize,cropsize)).convert(mode="1")
      #im_roadmap = Image.eval(im_roadmap,lambda x: 255*(x>1) ).convert(mode="1")

      im_satellite.save("./cleaned/map_%s_%s_satellite_clean.png" % (lat,lon))
      im_roadmap.save("./cleaned/map_%s_%s_roadmap_clean.png" % (lat,lon))
    else:
        print "%s %s NOT FOUND" %(lat,lon)
    lon += (endlon - startlon)/steps
    #break
  lon = startlon
  lat += (endlat - startlat) / steps
  #break
```

enhance_17_data.py

```
import os.path
from PIL import Image, ImageEnhance
startlat = 38.81
endlat = 38.86
startlon = -77.30
endlon = -77.14
steps = 22.0 # total images = steps * steps
latstep = (endlat - startlat) / steps
lonstep = (endlon - startlon) / steps
size = "512x564"
zoom = 17
scale = 2
cropsize = 1024
newsize = 512,512
```

```
lat = startlat
lon = startlon
while lat < endlat:
  print lat
  while lon < endlon:
    #print "%s %s" %(lat,lon)
    orig_satellite = "./original_scale2/map_%s_%s_satellite.png" % (lat,lon)
    orig_roadmap = "./original_scale2/map_%s_%s_roadmap.png" % (lat,lon)
    if os.path.isfile(orig_satellite) and os.path.isfile(orig_roadmap):
      print "%s %s" %(lat,lon)
      im_satellite = Image.open(orig_satellite).crop(box=(0,0,cropsize,cropsize))
      im_roadmap = 
Image.open(orig_roadmap).crop(box=(0,0,cropsize,cropsize)).convert(mode="1")
      im_satellite.thumbnail(newsize, Image.ANTIALIAS)
      im_roadmap.thumbnail(newsize, Image.ANTIALIAS)
      im_satellite.save("./cleaned_scale2/map_%s_%s_satellite_clean.png" % (lat,lon))
      im_roadmap.save("./cleaned_scale2/map_%s_%s_roadmap_clean.png" % (lat,lon))
      im_satellite = im_satellite.convert('RGB')
      adjust_factor = 1.3
      contrast = ImageEnhance.Contrast(im_satellite)
      im_satellite = contrast.enhance(adjust_factor)
      brightness = ImageEnhance.Brightness(im_satellite)
      im_satellite = brightness.enhance(adjust_factor)
      color_balance = ImageEnhance.Color(im_satellite)
      im_satellite = color_balance.enhance(adjust_factor)
      sharpness = ImageEnhance.Sharpness(im_satellite)
      im_satellite = sharpness.enhance(1.0)

      im_satellite.save("./enhanced_scale2/map_%s_%s_satellite_clean.png" % (lat,lon))
      im_roadmap.save("./enhanced_scale2/map_%s_%s_roadmap_clean.png" % (lat,lon))
    else:
        print "%s %s NOT FOUND" %(lat,lon)
    lon += (endlon - startlon)/steps
    #break
  lon = startlon
  lat += (endlat - startlat) / steps
  #break
```

Appendix B – Custom Keras Dilated Pooling2D Layer

At present, the Keras implementation of MaxPooling2D does not support dilated convolution. However, the Tensorflow function tf.nn.pool does. Because this functionality is needed in order to convert CNN's with max-pooling layers to their equivalent ACNNs, it was necessary to write the appropriate Keras layer. This layer and the dependent parts are shown below. The two layers added are AvgPool2D_DR and MaxPool2D_DR. They behave similarly to the official AvgPool2D and MaxPool2D layers, except they also accept as an argument a dilation_rate. No additional error checking was added, since many errors are raised by the backend Tensorflow function. Bigger blocks of comments are removed here for brevity, but can be seen in the included capstone_keras_layers.pl file. All of the code is separate from but very similar to the original layers from which they were spun-off. Important changes are highlighted in ***bold-italics***.

```
import tensorflow as tf

from keras.backend import tensorflow_backend as KBETF
from keras import backend as K
from keras.engine import Layer
from keras.engine import InputSpec
from keras.utils import conv_utils
from keras.legacy import interfaces

def pool2d_DR(x, pool_size, strides=(1, 1),
           padding='valid', data_format=None, dilation_rate=(1,1),
           pool_mode='max'):
    """2D Pooling.
    """
    if data_format is None:
        data_format = KBETF.image_data_format()
    if data_format not in {'channels_first', 'channels_last'}:
        raise ValueError('Unknown data_format ' + str(data_format))

    padding = KBETF._preprocess_padding(padding)
    pool_size = pool_size
    x = KBETF._preprocess_conv2d_input(x, data_format)

    if pool_mode.upper() in ['MAX','AVG']:
        x = tf.nn.pool(input=x, window_shape=pool_size, padding=padding.upper(),
                       dilation_rate=dilation_rate, pooling_type=pool_mode.upper())
    else:
        raise ValueError('Invalid pooling mode:', pool_mode)

    return KBETF._postprocess_conv2d_output(x, data_format)

class _Pooling2D_DR(Layer):
    """Abstract class for different pooling 2D layers.
    """
    def __init__(self, pool_size=(2, 2), strides=None, padding='valid',
                 data_format=None, dilation_rate=(1, 1), **kwargs):
        super().__init__(**kwargs)
        data_format = conv_utils.normalize_data_format(data_format)
        if strides is None:
            strides = pool_size
        self.pool_size = conv_utils.normalize_tuple(pool_size, 2, 'pool_size')
        self.strides = conv_utils.normalize_tuple(strides, 2, 'strides')
        self.padding = conv_utils.normalize_padding(padding)
        self.data_format = conv_utils.normalize_data_format(data_format)
        self.dilation_rate = dilation_rate
        self.input_spec = InputSpec(ndim=4)
    def compute_output_shape(self, input_shape):
        if self.data_format == 'channels_first':
            rows = input_shape[2]
            cols = input_shape[3]
        elif self.data_format == 'channels_last':
            rows = input_shape[1]
            cols = input_shape[2]
        rows = conv_utils.conv_output_length(rows, self.pool_size[0],
                                             self.padding, self.strides[0],
                                             dilation=self.dilation_rate[0])
        cols = conv_utils.conv_output_length(cols, self.pool_size[1],
                                             self.padding, self.strides[1],
                                             dilation=self.dilation_rate[1])
        if self.data_format == 'channels_first':
            return (input_shape[0], input_shape[1], rows, cols)
        elif self.data_format == 'channels_last':
            return (input_shape[0], rows, cols, input_shape[3])

    def _pooling_function(self, inputs, pool_size, strides,
                          padding, data_format):
        raise NotImplementedError

    def call(self, inputs):
        output = self._pooling_function(inputs=inputs,
                                        pool_size=self.pool_size,
                                        strides=self.strides,
                                        padding=self.padding,
                                        data_format=self.data_format,
                                        dilation_rate=self.dilation_rate)
        return output

    def get_config(self):
        config = {'pool_size': self.pool_size,
                  'padding': self.padding,
                  'strides': self.strides,
                  'data_format': self.data_format,
                  'dilation_rate': self.dilation_rate}
        base_config = super().get_config()
        return dict(list(base_config.items()) + list(config.items()))


class MaxPooling2D_DR(_Pooling2D_DR):
    """Max pooling operation for spatial data.
    """
    @interfaces.legacy_pooling2d_support
    def __init__(self, pool_size=(2, 2), strides=None, padding='valid',
                 data_format=None, dilation_rate=(1,1), **kwargs):
        super().__init__(pool_size, strides, padding,
                                           data_format, dilation_rate, **kwargs)

    def _pooling_function(self, inputs, pool_size, strides,
                          padding, data_format, dilation_rate):
        output = pool2d_DR(inputs, pool_size, strides,
                          padding, data_format, dilation_rate,
                          pool_mode='max')
        return output


class AveragePooling2D_DR(_Pooling2D_DR):
    """Average pooling operation for spatial data.
    """
    @interfaces.legacy_pooling2d_support
    def __init__(self, pool_size=(2, 2), strides=None, padding='valid',
                 data_format=None, dilation_rate=(1,1), **kwargs):
        super().__init__(pool_size, strides, padding,
                                            data_format, dilation_rate, **kwargs)

    def _pooling_function(self, inputs, pool_size, strides,
                          padding, data_format, dilation_rate):
        output = K.pool2d_DR(inputs, pool_size, strides,
                          padding, data_format, dilation_rate, pool_mode='avg')
        return output

# Aliases
AvgPool2D_DR = AveragePooling2D_DR
MaxPool2D_DR = MaxPooling2D_DR
```