Scene Labeling with Convolutional Neural Networks

Zeming Lin and Jack Lanchantin
University of Virginia

1. Introduction

Convolutional neural networks have been very successful in solving many problems in the domain of computer vision in a general and scalable way. The first computer vision systems were based on hand-crafted features by experts, and did not work nearly as well as some would have hoped, despite being very powerful on tasks such as feature selection and people detection. Inherently, image processing is a very large-scale task, due to the variations on the number of images in existence. A natural extension of machine learning algorithms includes learning the features along with classifiers. This allows us to scale up the number and variation of images we input into a system, as long as manual labels exist.

The next natural step is to automatically learn representations of data, inspired mainly by how the human vision system seems to work. Our brains tend to have clusters of neurons that are very good at identifying small features, which cascade up to allow us to identify concepts. This is the paradigm adopted by deep learning, by identifying layers of representations and depending on a large repository of data to create a good classifier.

The goal of this project is to use convolutional neural networks to label the class type of individual pixels in an image, and produce a full resolution output image which consists of each labeled pixel of an image. We will discuss the previous work in this area, give an overview to our approach, show our results, and finally discuss the strengths and limitations of of our work.

2. Previous Work

Researchers are able to take a data driven approach to the problem of scene labeling as a result of the work from a group from Stanford University released what is known as the Stanford Background Dataset [3]. This dataset contains 715 images chosen from existing public datasets which consist of outdoor scenes, have approximately 320×240 pixels, contain at least one foreground object, and have the horizon position within the image. Each image has an associated text file which contains a matrix of elements which represent the class label at each pixel in the image. Each pixel contains 1 of 9 total classes (e.g. ‘sky’, ‘road’, ‘mountain’) which were labeled by humans through Amazon’s mechanical turk.

Our approach is mainly inspired by the paper Recurrent Convolutional Network for Scene Labeling [1]. The author uses a 2-layer neural network to label scene pixels by feeding it image patches. Some previous results obtained above 80% class accuracy, at the disadvantage of a very long testing time, with over 60 seconds per image. Previous neural network approaches manage to get near 80% accuracy with a testing time of sub one-second. The author suggests adding recurrence to the 2-layer neural network to improve testing times to the order of one second while preserving state-of-the-art per-pixel accuracy.

Additionally, recent work by Google [2] shown large improvements in the ImageNet competition by using relatively small convolution kernel sizes and deep networks. Recent research has also shown the power of dropout networks

for regularization and rectifier linear units (ReLU) in image processing. We wish to try a deep model, using dropout and ReLU layers in the scene labeling problem and test how these new paradigms compare to the previous state of the art.

3. Design and Implementation

3.1 Model Architecture

In our model, we implement a multi-layer convolutional neural network which accepts an RGB image as input, and outputs a plane pixels which each contain one of 9 different labels. In order to automatically extract good features for labeling the pixels, our network learns a series of filter planes, which each act as a feature map, at each hidden layer in the network.

Choosing the ideal number of hidden layers and feature maps in our network was not clear, but we chose a hybrid architecture of [1] and [2], by taking the overall approach of [1] and extending it to include recent advances from [2] in deep convolutional networks. Our final implementation used 3 hidden layers which each contained 64 feature maps which were to be learned by the network. We used 4 2×2 pooling modules, and 3 4×4 convolutional kernels. A diagram of our model can be seen in figure 1.

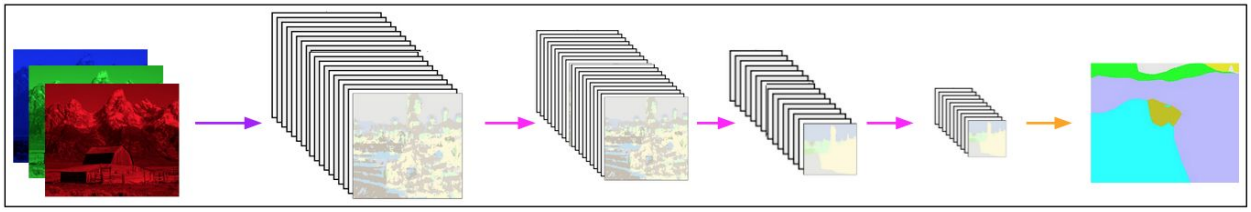

Figure 1 - Model Overview: The R,G, and B planes act as the 3 input planes to the network, which then get fed through 3 different convolutional hidden layers (which extract 64 feature map planes at each layer), and the output is a 9 plane layer which represent the probability of each of the 9 classes at each pixel, which we can then create our output label image. The pink arrows represent convolutional modules which include convolution, max pooling, and a nonlinear transfer function.

One of the main issues in large neural networks is that they tend to overfit the training set. Since we have so many parameters that we are trying to learn, there is a good chance that the parameters will be perfectly tuned to the examples in the training set. To overcome this issue, we use the idea of "dropout" [4], which essentially "drops out" certain nodes in the network with probability *p* so that no individual node ends up overfitting a particular image.

We also use rectifier linear units (ReLUs) rather than traditional tanh or sigmoid functions as the transfer functions in the network. ReLUs speed up the train/test time dramatically, as shown in [2].

3.2 Shifted Label Planes

Since our model seeks to label each individual pixel, the naive implementation would be to feed an individual pixel and its surrounding pixels (a chosen patch size) into the network so that the convolutional layers can learn features from the surrounding pixels of the target. This can be seen at the top of figure 2. However, feeding in each individual pixel's patch from an entire image can be very computationally expensive if we have a large image. Our solution is to feed in the entire image and automatically generate the patch size based on the convolutional and

pooling layers, where our model then outputs a large number of labeled outputs given the input image at once. This approach is shown at the bottom of figure 2. In this technique, we end up generating a low resolution output of labeled pixels due to the effects of the pooling layer, as shown in figure 3.

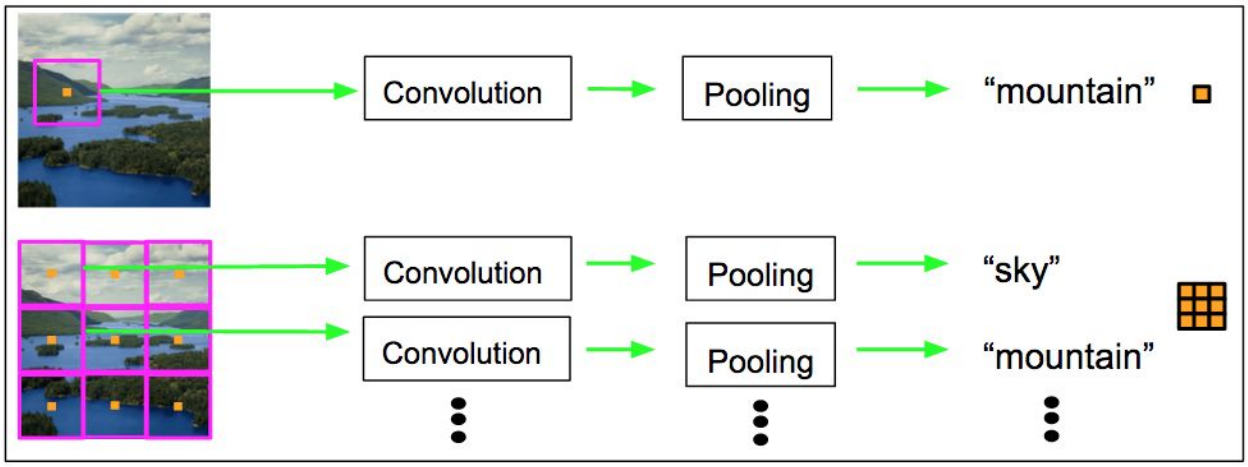

Figure 2 - Labeling many pixels at once: To label an individual pixel, we would input a patch size (shown in pink) around the target pixel (shown in orange), which then outputs a label based on the features from the network. In the basic approach (shown in the top half of the figure), we simply feed in one patch to the network. However, we can take advantage of the network structure and feed in the entire image (shown in the bottom half), which produces a number of labeled pixels.

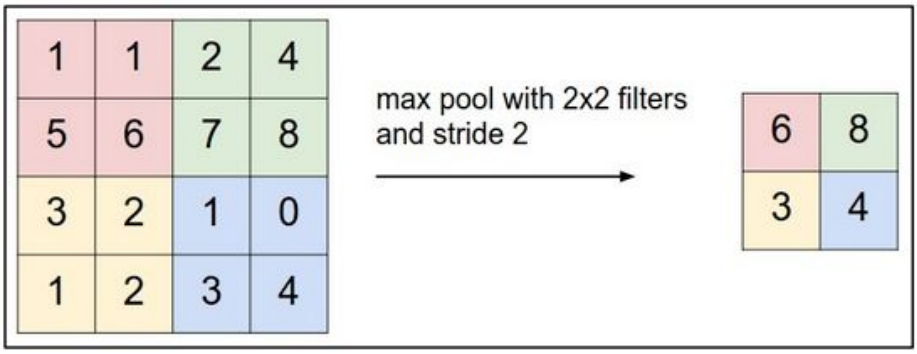

Figure 3 - Resolution from Pooling: The output image after pooling is of lower resolution than the input because pooling only keeps 1 pixel from the patch of the pool size.

We then upsample this output to create a full resolution plane of labeled pixels. This was what we did in our original implementation of the design. However, this solution is not ideal since we do not generate the actual labels for many pixels, we simply interpolate the non-labeled pixels, and thus we end up with less accurately labeled pixels. In order to get around this, [1] proposed the method of generating a series of shifted images which results in a series of outputs which contain different pixel locations from the input image. By then merging the shifted outputs together, we can create a full resolution output image. An example of this shift and merge method is shown in figure 4.

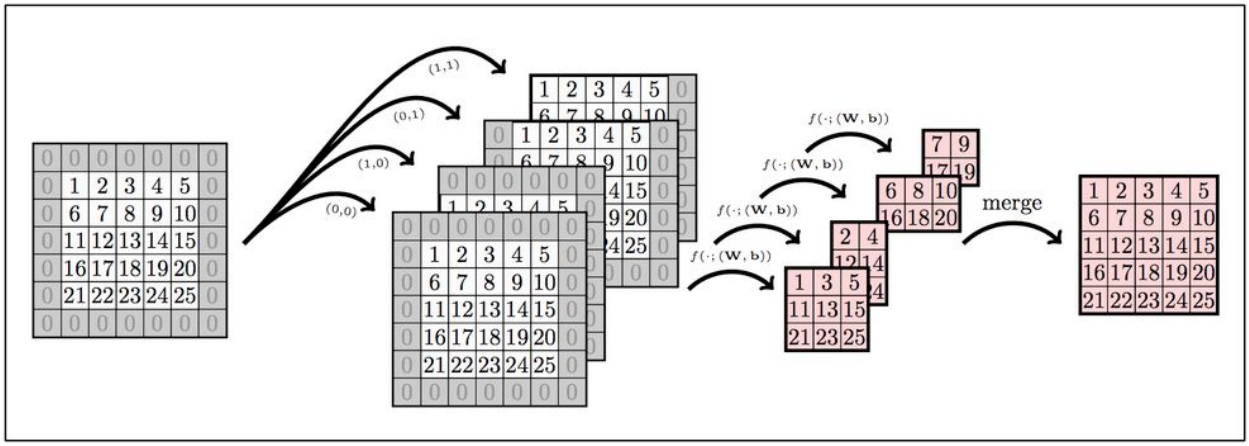

Figure 4 - Shift and Merge Technique: Convolutional networks output downscaled label planes due to pooling layers. In order to output a full resolution image, we feed shifted versions of the input image input our network (as shown in the second 'column' of the figure), which produce labels of different pixels in the image (shown in red in 'column' 3 of the figure), which can then be merged into a full res labeled output. In this example the network is assumed to have a single 2×2 pooling layer. Pixels represented by 0 are padding for the pooling.

This shift and merge implementation gives the most accurate results, but the tradeoff is that we have to run many more inputs through the network, which is very time consuming. In figure 4, there are only 4 different shifted versions of the input image that we have to run through the network, but this is due to the fact that there is only one 2×2 pooling layer. The number of shifted images needed to run through the network is proportional to the total number of pooling layers. For example, if we have 4 2×2 max pooling layers, then we have to generate $(2^4)^2 = 256$ different input images to run through the network. The ideal way to handle this issue would be to parallelize the shifted versions of the image, so we could run 256 shifted versions through the network at once, and then do the merging once they are all completed. However, since we did not have the resources to do this, we proposed a training model which randomly selects different scaled versions of the input images to train on so that we don't have to train on every shifted version of the input image.

3.3 Implementation

We trained our model using many different parameters, including the number of hidden layers, the number of units at each layer, the pooling sizes, and the convolution sizes. Since we seeked to run our model on a web server and allow users to submit images to our model to be labeled, we ended up choosing the architecture which was computationally fast at test time. We currently have our model running on a web server (ip address shown in section 6), which is able to label images from a URL link.

4. Results

Overall, our model produces favorable results. We achieve around 70% per pixel accuracy on our training set. Since we were limited with computing resources, we were not able to do proper training using every pixel of the training set. However, our model is able to run quickly (around a minute) with fairly good accuracy on images that users submit to our website.

The following figures show example outputs from our system. The left most image is the input image, the right-most image is the output, and the middle image is the input overlayed with the output. The top 3 images in each figure are the results of our model, while the bottom 3 images are the ground truth results.

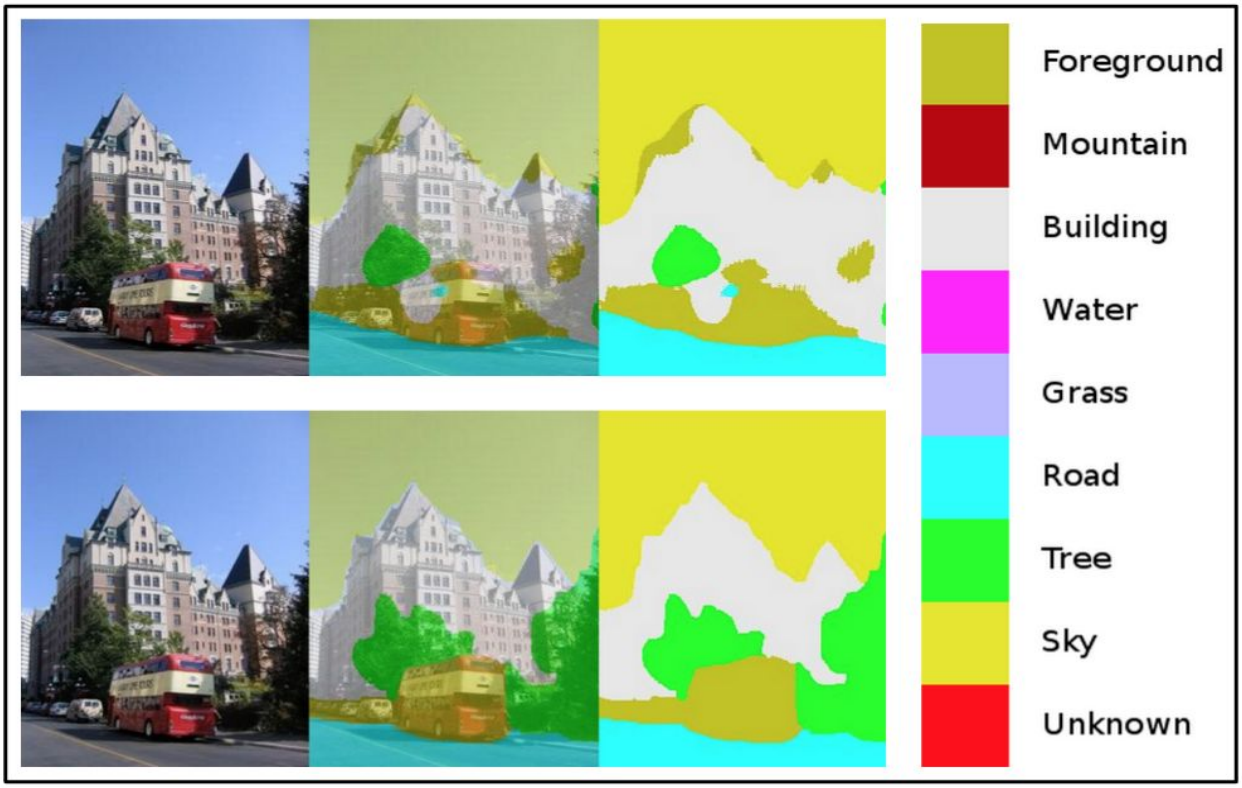

Figure 5

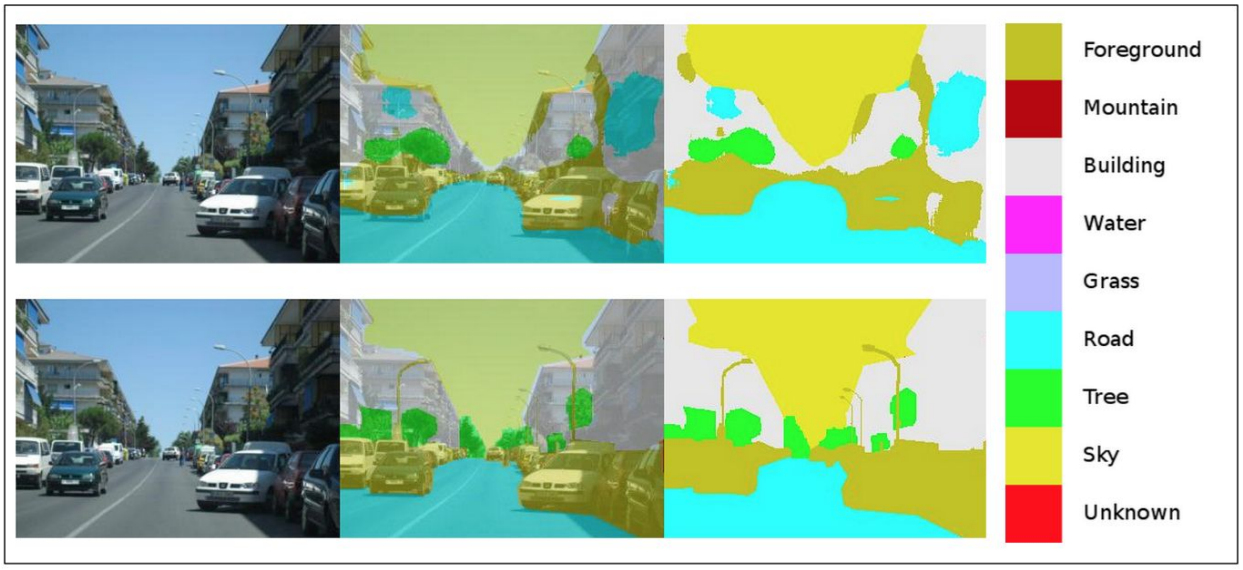

Figure 6

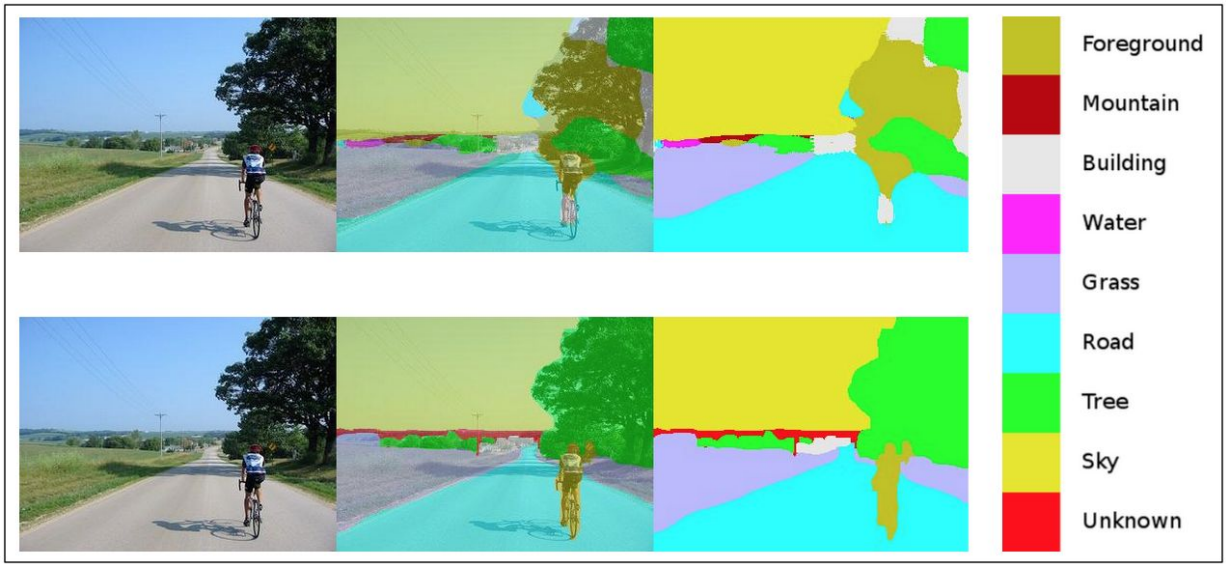

Figure 7

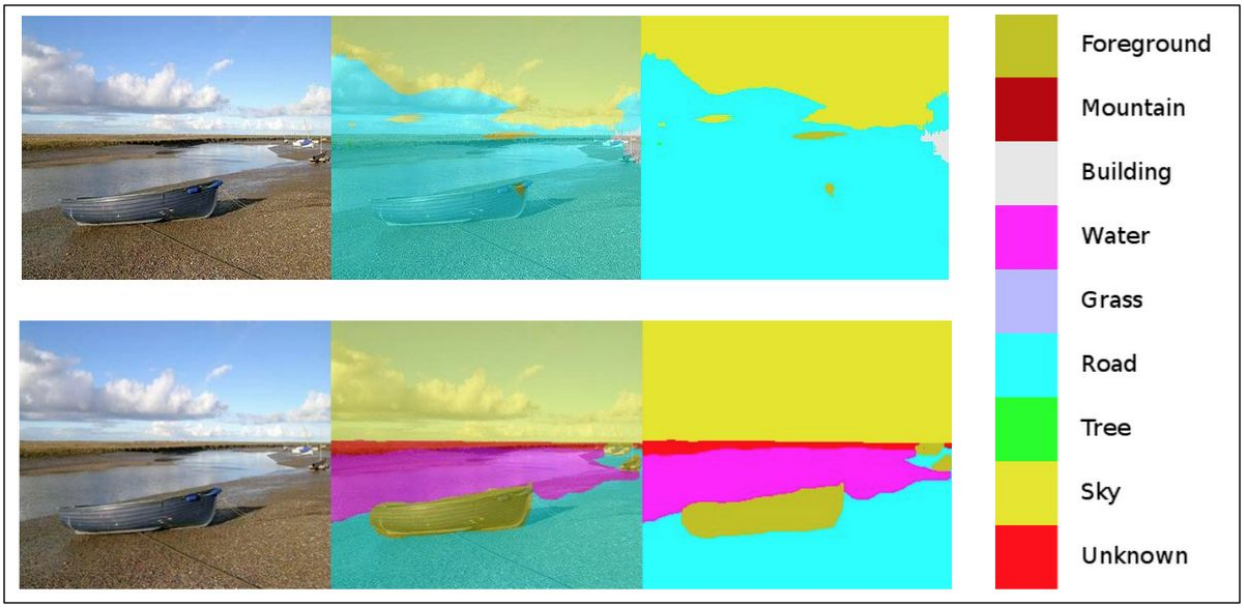

Figure 8

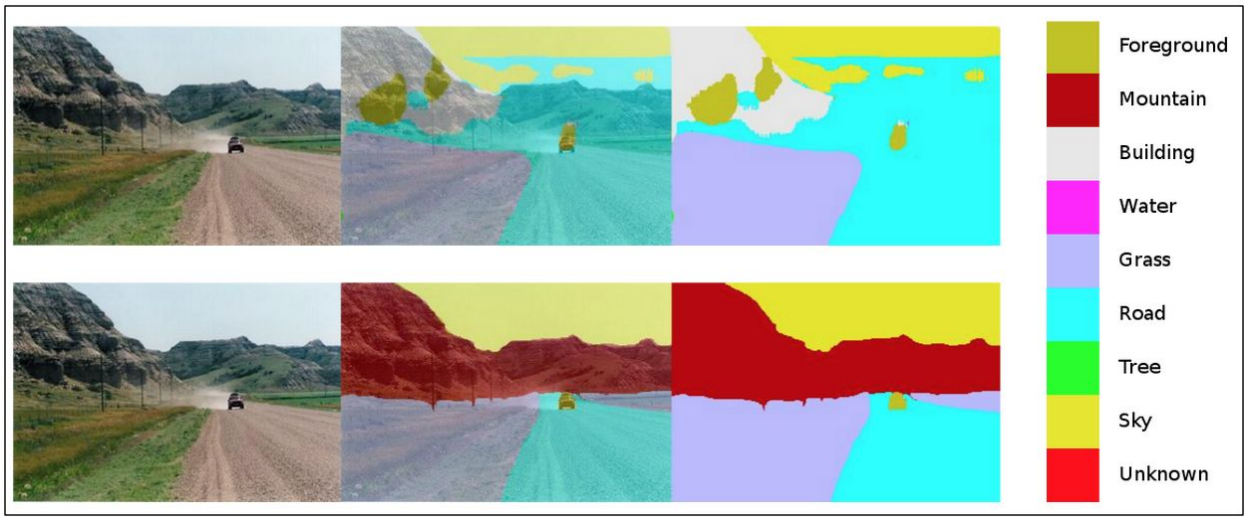

Figure 9

Figures 5,6 and 7 are examples that work fairly well using our model. Figures 8 and 9 have many mislabeled pixels. We attribute many of these labeled pixels to our training approach; since we train on a number of randomly selected pixels from each image, the number of mountain and water training pixels are much smaller than road pixels (due to the fact that there is part of a road in many of the training images), so our model does not learn good features for mountains or water and thus when a pixel from these classes is tested, the features for the correct classes are not triggered in the network.

5. Discussion and Further Work

The biggest advantage of convolutional neural networks is their generalizability. The inputs are the raw image pixels while the outputs are just labels. Therefore, any other problem which could be formulated as labeling each pixel could be easily put into this model in almost a plug-and-play fashion. The only extra work the modeler needs to do is to find more optimal parameters and gather enough labeled data instances.

Additionally, almost every process in this code is extremely parallelizable and online. The majority of computation within our model sits within convolution modules, which can be heavily optimized in a linear algebra library, or computed on a GPU, since GPUs are specialized for numerical processing such as convolutions. Furthermore, in merging low resolution label planes, we must pass several shifted images into our model at the same time and recombine the images. Each of these passes are disjoint from every other one, so we can take advantage of many cores or a GPU for large pictures, or use mapreduce to further increase parallelism.

Training of this system is also done online, so our system has the advantage of being able to take advantage of user feedback to continually train the model. In fact, we manage over 70% accuracy without even seeing the entire dataset once, and 67% after seeing only 1/16th of the data. Training for several more days will supposedly allow for near 80% accuracy, according to the papers provided.

The main disadvantage of our system is the speed in training. By using the shift and merge technique discussed in section 3.2, our model would take about 2 days to train on our 550 training images. We overcome this by randomly selecting shifted inputs during training which only takes about 4 hours to train, but we do not end up training on

every pixel in the training set. Using a large Likewise, testing on large images is also a slow process, although since in a production system our model holds the advantage of being very easily parallelizable. In our current approach, the "deeper" models (5 hidden layers) get up to ~70% accuracy on test data, but take about 5 minutes to test a 240×320 image. We were thus inclined to use a "shallower" model (3 hidden layers) which achieves ~67% accuracy and takes about 1 minute to test a 240×320 image.

In the future, we would like to parallelize the shift and merge process, train on more pixels, and try deeper models with larger input patches. We would also like to train on other datasets with more classes and datasets with different types of classes than scenery (e.g. medical images).

7. References

[1] Recurrent Convolutional Network for Scene Labeling
[2] Going Deeper with Convolutions
[3] Stanford Background Dataset
[4] Dropout: A Simple Way to Prevent Neural Networks from Overfitting