Road Segmentation of Satellite Images Using Convolutional Neural Networks

Jean Gillain
sciper no. 331411

David Desboeufs
sciper no. 287441

Mathieu Caboche
sciper no. 282182

***Abstract*—Outlining roads on satellite images can be useful for creating accurate maps of all four corners of the earth. Machine Learning enables us to automate this task. In fact, convolutional neural networks (CNNs) are a suitable tool for image recognition, and thus, for road segmentation as well. In this project we design a CNN and evaluate the resulting model. We compare our various CNN designs to a baseline model obtained from a basic CNN.**

I. Introduction

II. The Google Maps Satellite Images Data Set

The data set we are given to train a model on, consists of 100 satellite images from Google Maps. The pictures look like they were all taken from very similar locations, probably one single city. As such, the given data set is not very diverse. Furthermore, it is quite limited in size. This explains why we need to enlarge our data set. Note that our training set is a fraction of the same Google Maps satellite images. Thus, we expect that our road classifier can be trained to perform well for segmenting roads of that particular city/area or similar cities/areas. Its performance when applied to radically different roads and areas can however not be certified.

In section III-A we describe how the data set is enlarged and how we manage to train our CNN on a substantial amount of images, so as to yield proper results.

III. Models and Methods

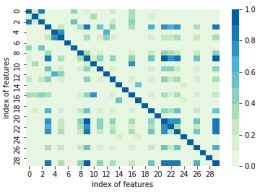

Figure 1: Correlation matrix of all features

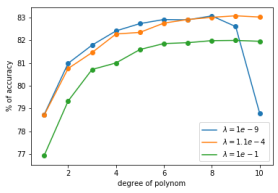

Figure 2: Accuracies obtained for different polynomial extension degrees and λ values

A. Creating a Large Data Set

We use a variety of methods and combine them in order to have a lot of data available to train our CNN.

Firstly, although we only have 100 images at our disposal for training, we actually train on smaller images that are subsets of the given images. For example, we can choose to train on images of size 20×20 pixels. This means that we get 400 training input sets for one single image (the given images have size 400×400). We are effectively partitioning large images into many small images, which has the added benefit of reducing the parameter count of our CNN.

Another common way to enlarge a data set, which we use, is to apply various transformations on the original images. By rotating, cropping and flipping a large scale image, we can obtain many variations of that image which prove very useful for training. Note that in this process we might lose image resolution, for example if we rotate and then crop. However that loss of resolution is effectively a zoom, another transformation. Any additional transformed images can only increase the sturdiness of our trained model.

More specifically, we rotate the training images by multiples of 45 degrees. When necessary, (for example when doing 45 degree rotations) we crop the images so as to avoid the resulting black areas. In cases where cropping is necessary, we lose close to one third of the image's resolution and content. To counterbalance this, we upscale those images. That way, all resulting images share the exact same size. This process leaves us with 800 images after processing the original 100. To further increase the size of our training data set, we mirror every image over its vertical axis. Eventually, we find ourselves with a total of 1600 images to train our model on.

We explored further ways of increasing the amount of input data we can feed to our CNN. One way to attain this, is to give as input the edges found in an image, along with the pixels of that image. We process the gray-scale converted image with a Sobel filter using the Scipy library. This yields a gray-scale image indicating where edges are to be found. We do not extract vectors from the resulting gray-scale picture, but only give the latter's pixels as an additional input to our CNN.

One additional information we extract and feed to our CNN, is what we like to call the *distance* of an image. This *distance* is the norm of the vector we obtain as a result of subtracting from every pixel's RGB value a value we call the *mean road RGB color*. We compute the *mean road RGB color* by averaging all RGB colors that are known to represent a piece of road, i.e. all pixels of our training set that we know to be part of a road (by looking at the ground-truth

images). The *distance* of an image can thus be represented as another image, of identical dimensions. It is an image that represents, at every pixel, how closely the color of the latter resembles the known color of a road. The resulting image can be fed to our CNN as additional data.

However, extracting edges from images and using the image *distance* did not prove useful. In fact, it is not necessary to manually extract features such as those when we train a model using a CNN. A good CNN will automatically learn those features. That is why, do not go into detail about implementations using these features, since the results obtained were not up to our expectations. The implementations can be found in the script `cnn_on_2_extracted_features.py` .

B. Evolution of our CNN

In the following we will discuss several CNN designs we experimented with, both successful and unsuccessful. We will also detail the design of our final CNN, showing which methods we experimented with were useful and included in our final design.

Before we dive into the detailed CNN designs we developed, let us list a few general ideas we experimented with. One first such idea is to use a combination of two or more CNNs. In our case, we tried using two CNNs one after the other. The second CNN takes as input the predictions made by the model resulting from the first CNN. In this design, the second CNN mainly acts as some kind of image filter that tries to fix the mistakes made by the first CNN. In fact, whereas the second CNN nicely fills missing gaps in road prediction for some images, it also creates non existing roads in the resulting predictions.

Another idea would be to use two or more CNNs in parallel, and for example, merge their results using another (following-up) CNN. In this case, we would need to apply those parallel CNNs on different inputs. However, as mentioned in section III-A, using inputs other than the images' RGB values is unnecessary, or at least not very useful in most cases. Thus, and due to the limited effect of using multiple CNNs in a row, we decided that we should stick to designing a single, well-tailored CNN for training a model.

As mentioned in section III-A, we divide all of our images (both training and test images) into patches of smaller size. The effect of patch size Since we cannot compute our model by directly taking whole images as inputs, due to the large number of resulting parameters, we use this image segmentation method in all implementations.

C. Model Validation

To validate our model we first experimented with k-fold cross validation. However, in order to get satisfactory results from the latter method, we would need to train our model multiple times for every implementation we designed. Since for most implementations a single training round can take a few hours, comparing our models using k-fold cross validation is not an option.

IV. Results

Regression Technique	Accuracy (Avg.) - STD
Least Squares GD (and SGD)	68.5% - 0.20%
Least Squares	74.4% - 0.25%
Ridge Regression	74.4% - 0.24%
Logistic Regression	65.7% - 0.29%
Newton Logistic Regression	65.7% - 0.29%
Our Best Regression	81.3% - 0.82%

Table I: Accuracy of Various Models Obtained by Our Regressions

V. Conclusion