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 main neural net designs to a baseline model obtained from our implementation of a basic CNN.**

I. Introduction

This project aims at creating a model capable of classifying satellite image regions as road or non-road, i.e. background. We are given a set of 100 satellite images to work on. This set has to be augmented and split, so that we can use it to train and validate our model. The implemented training method is a convolutional neural network. The report also details various implementations, both successful and unsuccessful, made in terms of pre-processing, neural net design and post-processing.

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

In the following we will go over a range of different implemented pre-processing methods, neural net designs, and post-processing methods. We will detail which methods proved useful and in which context.

A. Data Pre-Processing

We experimented with various methods in order to enhance and modify our given data set. Let us examine these methods and sum up which of them proved useful, and which did not.

[TRANSFORMATIONS] A 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 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.

[EDGES] 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.

[DISTANCE] One additional information we can 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, both methods *EDGES* and *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 neural net will automatically learn useful features. *TRANSFORMATIONS* proved extremely useful, they are necessary in order to ensure that we have sufficient data to train our model on.

B. *Evolution of Our Neural Network*

Let us now look at several CNN designs we experimented with, both successful and unsuccessful. Before we dive into the detailed neural net designs we developed, let us list a few general ideas we experimented with.

[PATCHES] One first such idea is to train our model on small images that are subsets of the given images, we call them patches. 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. This method is quite useful. In fact, to limit parameter count, it is good to work on smaller inputs (and outputs). We have used this method in essentially every model we implemented, with varying patch sizes, going from 16×16 up to over 200×200.

[SEQUENTIAL CNNs] The idea here is to use a combination of two or more convolutional neural networks. In our case, we tried using two CNNs one after the other. The second neural net takes as input the predictions made by the model resulting from the first CNN. In this design, the second net mainly acts as some kind of image filter that tries to fix the mistakes made by the first one. It will fill missing gaps in road predictions and remove some unwanted noise.

We will now detail more precise design decisions, the ideas behind those designs, and their effect on model performance.

[PATCH SIZE] As already mentioned, we divide all of our images (both training and test images) into patches of smaller size. The choice of patch size depends on the implemented neural net design. If we want to use a large number of batches, we need to reduce the patch size. In our case, most implementations use a patch size of 16×16 pixels. This coincides with the evaluation criteria. Some of our implementations classify an image pixel by pixel, some classify patch by patch. We classify a patch as road if more than half of its pixels are classified as road. In general, using patches of side 16, we found that our models' predictions were better evaluated when predicting patch by patch than when predicting pixel by pixel. Our final and best performing model however, works of patches of size 256×256 and predicts every pixel of that patch.

[PATCH EXTENSIONS] Let us consider an implementation using patches of side 16 pixels, and classifying patch by patch. In this case, we notice that patches containing an object that overlays a road are often ill-classified. To counteract this issue, we would like the classifier to take into account the pixels that surround the patch it is working on. That is, when classifying a patch of side 16, we give the classifier a larger input, i.e. a patch of side 24. In order to do this, we add a border of a few pixels to every patch. This border consists of the pixels surrounding the patch in the original image. Note that we have to add a border to the original image as well, of the same size than that of the borders we add to patches. We use reflection padding to create this border. That way, we can add borders even to patches lying at the rim of our original image. Using this method we can significantly improve our prediction accuracies, as shown in figure 2.

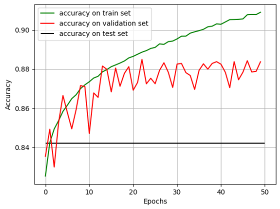

Figure 1: Accuracies of our VGG-16-like model (with patches of size 16×16, without patch extension)

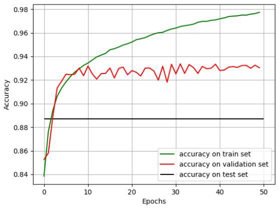

Figure 2: Accuracies of our VGG-16-like model with patch extension (border of 16px added to every patch)

[VGG-16-LIKE] One CNN design we implemented and used is inspired by a design known as VGG-16 [1]. This neural net essentially consists of 3 repetitions of a succession of multiple convolution layers and a max-pooling layer, and ends with two fully connected layers. The convolution layers (Conv2D in Keras) have a kernel size of (3, 3) with a stride of (1, 1). Our model takes a patch of size 16×16 as an input, and outputs a single value, between 0 and 1. We then apply a threshold on that output to characterize the patch as road or background.

Unfortunately, the main issue of this design is that it is difficult for the model to determine the patch's classification due to the latter's small size. In order to get appreciably good results using this design we need to apply *PATCH EXTENSION*. Figures 1 and 2 show the different accuracies obtained when using this model, both with and without patch extension.

C. *Post-Processing*

After observing that many of the predictions made by our designed models contained a fair amount of noise, we decided to use some post-processing to get clearer results. The observed noise mostly consists of single patches wrongly classified as road, lying in between patches correctly classified as background. We also notice some patches surrounded by road-patches that are classified as background.

To subvert this issue, we apply a convolution filter to the computed prediction. The convolution looks at blocks of patches of size 3×3 patches, and determines if the central patch should be marked as road or not, depending on the 8 patches surrounding it. It is a convolution that assumes that most roads are straight lines, either horizontally, vertically or diagonally across the 9 considered patches. The effect of this filter is shown in figure 4. A patch surrounded by road patches should be classified as a road, and similarly a patch surrounded by background patches should be classified as background.

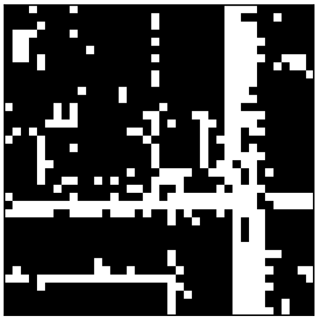

Figure 3: Prediction without post-processing

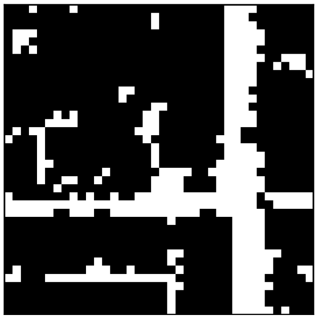

Figure 4: Prediction with post-processing

D. 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.

We thus evaluate our models on their validation and test accuracy (accuracies on validation and test sets respectively). The validation and test accuracies are obtained by splitting the data set into 3 parts, the first, major, part is used for training, the second for validating the model and the last is used to obtain the test accuracy. The used ratios are:

- Training set : 80% of the data set
- Validation set : 10% of the data set
- Testing set : 10% of the data set

We randomize our data before splitting. The training is run for enough epochs, so as to allow the validation accuracy to be maximized for the implementation we are testing. The validation set is used to avoid over-fitting during the training phase. The test set on the other hand is used to evaluate our model's performance (accuracy).

IV. Final Design

Our final design is based on a variant of a U-Net model by Peter Hönigschmid [2]. Our model takes as input an image patch of dimensions (256, 256, 3). It outputs a prediction for every pixel, i.e. the output has the shape (256, 256). Every entry of the output is a single value, between 0 and 1. We apply a threshold of 0.25 on that output to characterize every pixel as road or background. Since the model contains many layers and parameters, we will not detail it here. A complete representation of the model can be seen in `model_plot.png`, appended to the report.

We augmented the basis of Peter Hönigschmid [2] using ideas from a paper by Shaofeng Caiet et al. [3]. The extracted and applied techniques consist in the following:

After each convolution in our initial U-Net model, we perform a batch normalization, followed by a Relu activation function and a two dimensional spatial dropout, in that order. Let us detail the batch normalization and spatial dropout.

Batch normalization (BN) is an useful technique for normalizing the inputs of a layer. As stated by Jason Brownlee [4], it reduces the number of epochs required to train the model correctly. Furthermore, it is adequate to use BN since we use a small batch size in our model, due to memory restrictions.

Two dimensional spatial dropout (*SpatialDropout2D* in Keras) consists of dropping random channels. In our case, we prefer the latter layer to a classical *Dropout* layer, since a *Dropout* layer drops random individual pixels. In our case, neighboring pixels are often highly correlated, which corresponds to the official Keras documentation's [5] use case. A well detailed StackExchange topic [6] motivates this decision further. The spatial dropout percentage we use is 15%.

Our final model is trained on the extended data set, as explained in section III-A (*TRANSFORMATIONS*) . We do not apply the described post-processing method (section III-C) to our final model. Furthermore, since we use a large patch size, no patch extension is necessary.

V. Results

Let us now expose the results obtained using our final model. We trained our model using the Colab infrastructure provided by Google. The training was done on an NVidia Tesla V100-SXM2-16GB GPU and required 12GB of memory. The running time to obtain the following results was about 5 hours. Table I shows a comparison of the achieved performances of our final model and those of two of our previous models. The used evaluation metrics are the following:

- The accuracy on the test set achieved when evaluating the model on our own, local test set (cf. data splitting in section III-D). Denoted as *Accuracy (Local)*.

Model	Accuracy (Local)	Accuracy (AiCrowd)	F1 Score (AiCrowd)
VGG-16-like	84.2%	87.4%	76.8%
VGG-16-like with patch extension	88.9%	88.9%	78.0%
Final model (U-Net-like)	92.7%	94.3%	89.4%

Table I: Accuracies and F1 Scores of our final model compared to two of our previous models.

- The accuracy achieved when evaluating the model on the AiCrowd platform. Denoted as *Accuracy (AiCrowd)*.
- The F1 Score achieved when evaluating the model on the AiCrowd platform. Denoted as *F1 Score (AiCrowd)*.

The models in question are those described in previous sections. Note that *VGG-16-like with patch extension* is identical to the *VGG-16-like* model, except that is uses patch extension, with borders of size 16 pixels. All models are trained on the same, extended data set (i.e. a data set augmented using *TRANSFORMATIONS*, cf. section III-A). They are locally evaluated and trained as described in section III-D.

Figure 5 shows the training evolution of our model. The locally achieved test accuracy is of 92.7%. In the AiCrowd competition, we attained an accuracy of 94.3% and a corresponding F1 score of 89.4%. Figure 6 shows a test image with the corresponding predictions, made by our model, highlighted in red.

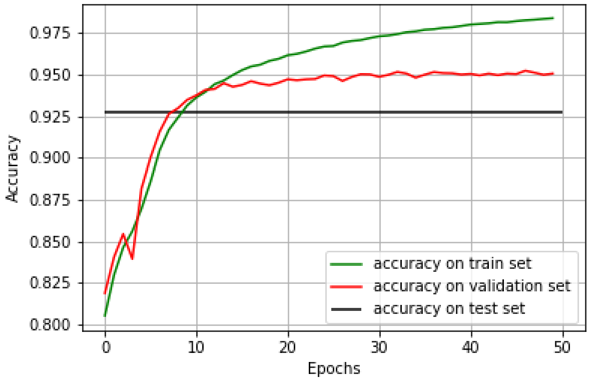

Figure 5: Accuracies of our final model w.r.t. the number of epochs the model was run for.

VI. Conclusion

Our final model improves in accuracy by about 8.5% compared to our baseline model (VGG-16-like). Due to time restrictions we were unfortunately unable to enhance our final model any further than we did. In fact, we experimented with many different models and CNN designs throughout the project's development. Sadly, we only found out about the U-Net-like model quite late in the development process. We would have liked to play further with our final model. Undoubtedly we could have improved its performance, using for example some of the methods mentioned in section III.

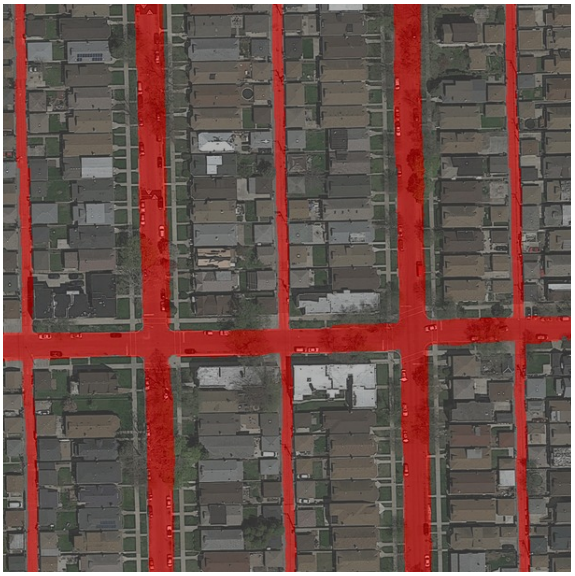

Figure 6: A training image overlayed with the predictions made by our model, in red (road prediction).

References

[1] M. ul Hassan, "Vgg16 – convolutional network for classification and detection," *Neurohive*, 2018. [Online]. Available: https://neurohive.io/en/popular-networks/vgg16/

[2] P. Hönigschmid, "U-net, dropout, augmentation, stratification," *Kaggle*, 2018. [Online]. Available: https://www.kaggle.com/phoenigs/u-net-dropout-augmentation-stratification

[3] W. W. G. C. M. Shaofeng Cai, Yao Shu and B. C. Ooi, "Efficient and effective dropout for deepconvolutional neural networks," 2020. [Online]. Available: https://arxiv.org/pdf/1904.03392.pdf

[4] J. Brownlee, "A gentle introduction to batch normalization for deep neural networks," *Machine Learning Mastery*, 2019. [Online]. Available: https://machinelearningmastery.com/batch-normalization-for-training-of-deep-neural-networks/

[5] Google. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/layers/SpatialDropout2D

[6] StackExchange. [Online]. Available: https://stats.stackexchange.com/questions/282282/how-is-spatial-dropout-in-2d-implemented