Machine Learning : Class Project 2 - Road Segmentation

Youssef M. Attia
EPFL, Lausanne

Torgeman M. Tarak
EPFL, Lausanne

Léo Dupont
EPFL, Lausanne

Abstract

Computer vision has always been a subject of interest for computer scientists around the world; and with the Deep Learning revolution that began in the 2000s, more and better applications were made possible. Among one of these applications is image segmentation, and this is what we discuss in this paper.

1 Introduction

Image segmentation is a problem in computer vision where, for a given image, we wish to produce a partitioning of an image into several pixel segments, where each pixel in given segment are similar in the original image with respect to some feature. This output is often called a mask.
The problem we treat here is a binary case where we wish to determine in a satellite Google Map image whether a given pixel corresponds to a road or a foreground (house, park, lake, etc..). In the computed mask, either the pixel is white (road), or black.
In this paper we will explain how we built our model, the different methodologies we tried and the data manipulation we did.

2 Data treatment

Given data

For this problem, we were given an initial dataset of 100 training images along with their groundtruth mask, as well as 50 images we were to test our model on. The training images were 400x400 RGB images, and the test images were 608x608 RGB images. As such, when translated to Numpy arrays to work on, these give, respectively (400,400,3) and (608,608,3) arrays. The groundtruth images are grayscale images of a single channel.

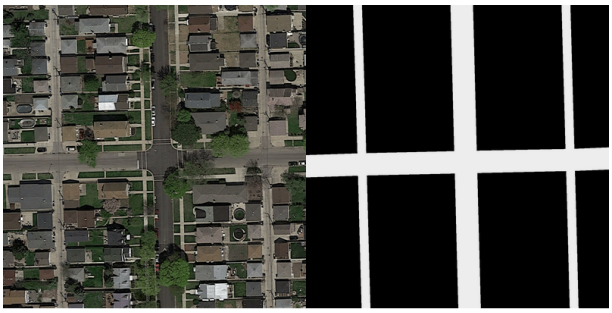

Figure 1: Example of an image and its corresponding groundtruth

Data pre-processing

As is often the case when working with image data in Machine Learning, the pre-processing of the data consisted of dividing every pixel value in the images by 255. This way, we restrict the pixel values between 0 and 1, instead of being between 0 and 255 (it is often easier to work with smaller numbers in Deep Learning). In addition to that, for the masks, we also force each pixel to have a value of 0 or 1 (nothing in between) since there are only 2 classes they can belong to.

3 Building the model

Metric to maximize

In this particular problem, the metric we try to maximize is the **F1_score**. It is defined as follow:

$$F_1 = \frac{2}{\mathrm{recall}^{-1} + \mathrm{precision}^{-1}} = 2 \cdot \frac{\mathrm{precision} \cdot \mathrm{recall}}{\mathrm{precision} + \mathrm{recall}} = \frac{\mathrm{tp}}{\mathrm{tp} + \frac{1}{2}(\mathrm{fp} + \mathrm{fn})}$$

As such, we're looking for a high percentage of true positives (white pixels) to be classified as such, making entirely black masks which, despite giving a high accuracy, will result in a catastrophic F1_score.

External Library

In order to improve our workflow, we used the Tensorflow library which provides many machine

learning methods and also allows us to train our models on a GPU, which drastically shortens training duration.
Be weary however that we ran our programs through Google Colab as it proposes a free offer to use quite a powerful GPU for a few hours per day. As such, unless one has these resources, execution time can be quite high.

First CNN try

Our first idea to approach this problem was to use a concept seen in class : a Convolutional Neural Network. Since such a model cannot do something much more sophisticated than classification, we had to think of a way to apply that to road segmentation. We could not simply classify every pixel individually as only groups of them have an actual meaning here. Therefore, we decided to divide every image in patches of 16x16 pixels, and give each of those a single label based on the average of the pixels of the corresponding mask patch (if this average was more than 0.25, then the label was 1, otherwise it was 0).

Before thinking about the exact topology of our CNN, we decided to do some simple data augmentation, meaning mirroring each image from left to right and applying a 90° rotation (tf.image.rot90, tf being tensorflow).

Our first model which achieved significant results was the CNN created using the following snippet of code :

```
def build_model():
    model = keras.Sequential()
    model.add(layers.RandomZoom(-0.3))
    model.add(layers.Conv2D(64, (1, 1),
        activation='elu', padding = "same"))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(128, (1, 1),
        activation='elu', padding = "same"))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(256, (1, 1),
        activation='elu', padding = "same"))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(512, (1, 1),
        activation='elu', padding = "same"))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Flatten())
    model.add(layers.Dense(2))
    model.add(layers.Softmax())
    return model
```

This model, using a validation split of 20% and 50 epochs, achieves a f1-score of 0.696 and an accuracy of 0.845. This is not enough for an effective machine learning model, so we needed to understand why we had such results. The main problem of this approach was that we could only label entire 16x16 patches, and not more fine-grain geometry that could be useful for example for diagonal roads. This means that even with a perfect model, it would be impossible to achieve a perfect accuracy. And this also means that this model was trained using patches that originally contained potentially both road and foreground but that were assigned only one of those labels, so it could learn to label some types of foreground as road or road as foreground, which is undesirable. Since we could not rely on a CNN to properly solve our road segmentation problem, we had to explore different, more complex models.

Figure 2: Original strategy: predict the label of a small patch of the image

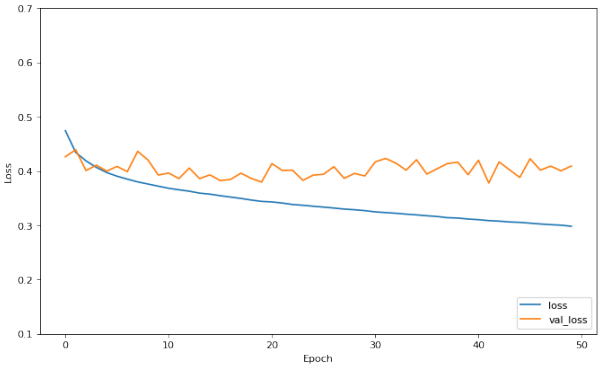

Figure 3: Training and validation loss of the CNN through time

We tried the CNN again afterwards, as explained some sections below

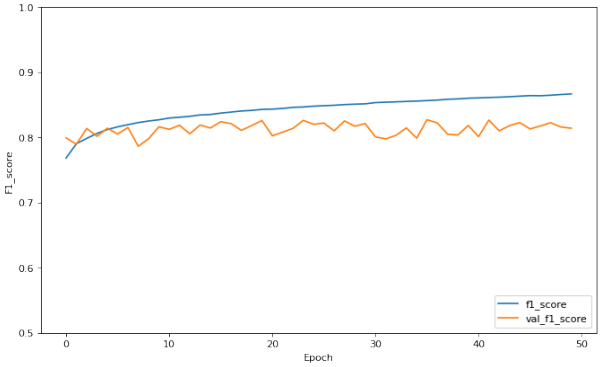

Figure 4: Training and validation F1 score of the CNN through time

U-net

With the limitations of the CNN we implemented observed, we decided to implement another architecture. The main goal was to have a large input size in order to perform an image segmentation, and in order to achieve that we decided to try the U-net architecture, which consists of an encoding path and a decoding path. During the encoding, the spatial information is reduced while feature information is increased. The decoding combines the feature and spatial information through a sequence of up-convolutions and concatenations with high-resolution features from the contracting path as explained in (1).

As we didn't have any prior experience using U-net, we looked online for reference and stumbled upon Harshall Lamba's article (2) which explains in detail how to make a U-net model, and the layers involved.
The model's architecture is available here, as it would be impossible to include the image in this report. We can however discuss the metric we most focused on, the **dice coefficient**.

Dice coefficient

The dice coefficient (DC) is positively correlated to the F1_score, and in the case of boolean data, is quasi-equivalent. In the case of comparing two segmentations of an image, it felt more appropriate to use this metric. The dice coefficient is defined as:

$$DC = \frac{2|X \cap Y|}{|X| + |Y|}$$

In the case of images, it corresponds to the ratio of the number pixels that are the same in the two segmentations, divided by the total number of pixels. The ratio is then multiplied by 2, as there are 2 images. A score of 1 thus means that both images are the same.

We discovered this metric when looking online for good metrics for a segmentation problem. (3)

First U-net strategy

The first (unsuccessful) strategy we followed is the following: considering our train and test images didn't have the same dimensions (respectivelly 400x400 and 608x608), when loading the training data, we resized both the images and groundtruths to be of the same dimensions of the test images.
The reason for that was that we wanted to work on big patches of the images; however, the GCD of 400 and 608 is 16, which isn't big enough. As such, we resized the train images to be 608x608, and used patches of dimensions 304x304. We knew that this reduced the overall quality of the images a bit, but it didn't seem like a problem at the time.
In this case, we used 10% of the training data as a validation set; this first strategy gave us much better results than CNN: an F1/DC of 0.795, and an accuracy of 0.889. However, we quickly ran into a problem: the model was clearly overfitting, despite the regularizations (i.e dropout, L_2 regularization) we introduced. As such, we aimed at introducing more data augmentations, in order to have more training data. However, because of the memory limitations of Google Colab that lets us use 12GB of RAM, we could only introduce another augmentation at a time, and even less on our own machine.
We put that in relation with the resize of the training images we performed, and with the fact that we further divide this image into 4 more patches of dimensions 304x304. As such, we theorized we would have more liberties with loading the images without resizing them, and train the model directly on these.

Second U-net strategy

At this point, we decided to reload the data but this time keeping the original sizes of the images, thus their quality as well. The decrease in data consumption also allowed us to add many data augmentations, more precisely:

- Added a 180° rotation of the images, using tf.image.rot90 once again;

- Added rotations such that there would be more diagonal roads in the training data (225° and 315°), as we noticed the model sometimes performs poorly on diagonal roads, especially smaller ones
- Adjusting the brightness of the images, using tf.image.stateless_random_brightness;
- Adjusting the contrast of the images, using tf.image.stateless_random_contrast;
- Adjusting the saturation of the images, using tf.image.stateless_random_saturation;
- Adjusting the quality of the images, using tf.image.random_jpeg_quality.

We also removed the images which in our opinion didn't match properly their groundtruth.
After training the model, we had to predict on the test images which didn't have the same dimensions; in order to do this, we first tried to resize down the image (from 608x608 to 400x400), but the decrease in quality was too important to expect good results.
We settled on dividing the test image in four 400x400 patches such that the whole image is covered, and predict the mask of these patches. Obviously, some columns or rows could be common to two different patches; that is fine as the areas in common between two patches will be predicted the same way, and so when putting the mask back together, the result still makes sense.

With this strategy, with 5% of the data used as validation (we made this choice so that the model has more data to learn), the U-net gave us a new high performance: 0.893 of F1 , and 0.941 of accuracy; from this point on, we assumed this is the best model we'll be able to build, and so tried to tune the hyper-parameters (batch size, dropout rate and L2 regularization factor) in search of a potential amelioration.

Batch size	Drop rate	L2 rate	Val split	Acc	F1
8	0.1	10^{-4}	0.05	0.941	0.893
8	0.2	10^{-3}	0.15	0.941	0.893
12	0.2	10^{-4}	0.05	0.940	0.892
4	0.2	10^{-4}	0.05	0.940	0.892
8	0.2	10^{-4}	0.05	0.940	0.892
8	0.2	10^{-4}	0.15	0.939	0.890
8	0.1	10^{-4}	0.05	0.938	0.887

At this point, after adding all these data augmentations, we tried to use the CNN approach again, hoping for better results, and most importantly so that we compare the U-net and the CNN under similar conditions. No better results have been observed despite the additional data augmentations.

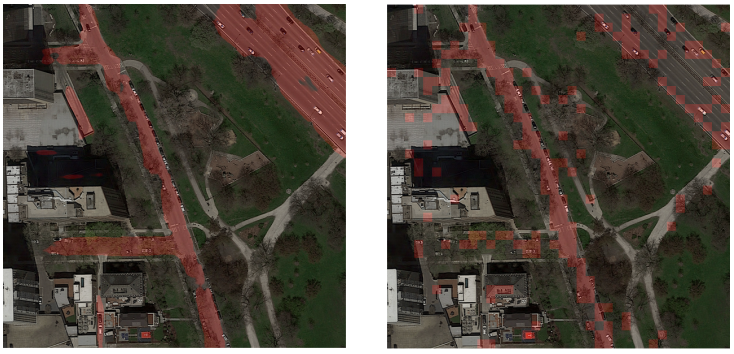

Figure 5: Comparing the models' powers: on the left, the U-net, on the right, the CNN.

4 Conclusion

Based on the evidence presented, we can deduce that The U-net model with a batch size of 8, a Drop rate of 0.1 and a L_2 rate of 10^{-4} gave us the best results, as was shown in the F1 score and the accuracy. In general we saw in these experiments that the U-net architecture outperforms the CNN architecture, which is to be expected as a U-net generally has more layers than the CNN thus more trainable parameters, yielding more complex models. While U-nets are widely used for image segmentation, the Mask R-CNN usually yields better results but are also much more complex, in this architecture objects are classified and localized using a bounding box and semantic segmentation that classifies each pixel into a set of categories. Every region of interest gets a segmentation mask. A class label and a bounding box are produced as the final output.

References

[1] Ronneberger, O., Fisher, P., Brox, T. (2015). U-Net: Convolutional Networks for Biomedical Image Segmentation (pp. 234-241). Springer Verlag.

[2] Harshall, L. (2019). Understanding Semantic Segmentation with UNET. Medium. From https://towardsdatascience.com/understanding-semantic-segmentation-with-unet-

[3] Ekin, T. (2019). Metrics to Evaluate your Semantic Segmentation Model. Medium. From https://towardsdatascience.com/metrics-to-evaluate-your-semantic-segmentation