Road segmentation: extracting roads from satellite images

Mariia Eremina, Qiming Sun, Ulysse Widmer
mariia.eremina@epfl.ch, qiming.sun@epfl.ch, ulysse.widmer@epfl.ch

EPFL, Switzerland

***Abstract*—In this paper, we explore the road segmentation problem. The goal is to define which pixels are part of a road and which aren't in satellite images. For our model, we implement a type of convolutional neural network called a *U-Net*[2], initially developed for biomedical image segmentation. We will use soft dice loss as a loss function and F1 score as the main metric. The dataset[1] we use consists of satellite images from Google Maps.**

I. Introduction

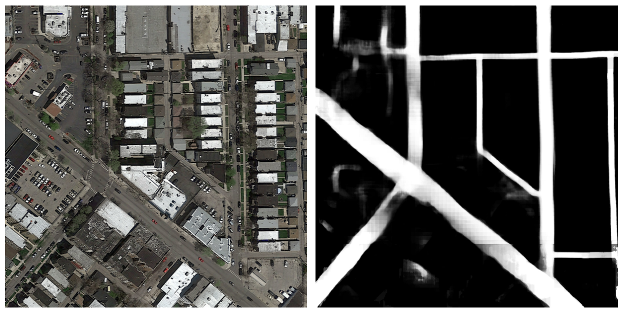

Fig. 1. Road segmentation example: satellite image on the left and prediction result mask on the right

As shown in figure 1, road segmentation consists of labeling the pixels of an image with either "road" (represented as white) or "background" (represented as black). We start with colored satellite images and try to output bitmaps of labels that can be displayed as gray-scale images of the roads. This apparently simple task is quite important to achieve correctly with automated processes, because of the many applications it can have. Not only can it help complete maps applications and guiding systems, but it could also help detect road modifications in real-time when a disaster or an explosion occurs for example. In the following section, we will present how we approached and prepared the data from Google Maps and we will detail our U-Net architecture. We will then display our results obtained with the said model, using the F1-score as a metric and we will end up by discussing some of the weaknesses that can be found in our model.

[1]You can find the dataset on https://www.aicrowd.com/challenges/epfl-ml-road-segmentation

II. Methodology

A. Dataset and challenges

The dataset for training contains 100 squared colored satellite images of 400x400 pixels, and their corresponding gray-scale ground-truth images, similar to figure 1. As we can see in the prediction of figure 1, getting accurate boundaries for roads is actually not an easy task. For example, trees, vehicles, or building shadows may obstruct the continuity of a road border, and in parking areas, we would want the model to distinguish between roads and parking slots. The test set on the other contains 50 colored images of size 608x608 pixels.

B. Data augmentation

The first challenge is to palliate the small number of training data points, which gets even smaller after splitting the data for validation. For this, we used data augmentation by using rotations on the original images. We used 16 random rotations at each batch with an 80/20 split for validation, thus going from 80 samples to $80 \cdot 16 = 1280$ samples available per batch for training.

C. U-Net architecture

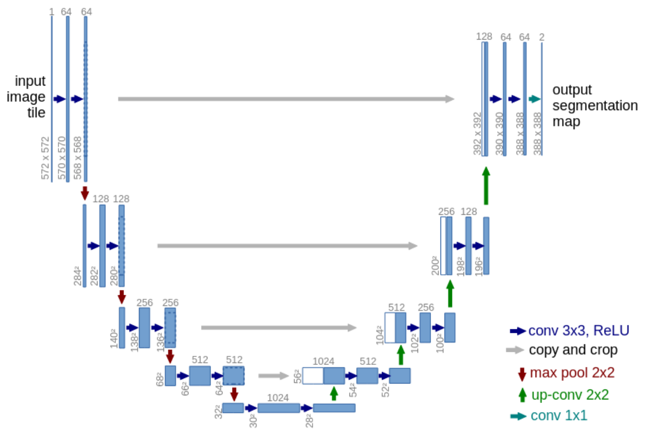

Fig. 2. U-Net architecture based on biomedical imaging[2].

We chose to select a U-Net architecture because it was developed specifically for segmentation, whereas a more traditional CNN would be better suited for full image classification.

We inspired ourselves heavily from the original paper introducing U-Nets, *U-Net: Convolutional Networks for Biomedical Image Segmentation*[2]. Our architecture also consists of two 3x3 convolutions followed by a 2x2 max pooling operation for each encoder block but deviates a little bit for the decoder part, where we use a 3x3 deconvolution followed by two 3x3 convolutions for each block.

D. Hyperparameters

Our architecture allows tweaking many hyperparameters: the number of blocks for the encoder and decoder, the input size, the dropout rate, etc. We've chosen to focus on a subset of them to keep it achievable in terms of training time. To avoid an exponential explosion of parameter combinations, we trained parameters pairwise on a small number of epochs that still allow seeing significant differences in learning rates. We started by iteration over different input sizes and block numbers, then we added dropout variations, followed by rotation angle variations for data augmentation, and finally, we iterated over different numbers of batches per epoch. Detailed results are explained in the next section.

E. Loss function

For the loss function, we implemented soft dice loss and binary cross-entropy, and we ended up using exclusively soft dice loss as it appeared to perform better, because of the imbalance between background pixels and road pixels. We didn't do hyperparameter optimization with the loss functions because of a lack of time and computing capabilities, but it is one possible path to explore to improve our model.

F. Implementation details

To implement our U-Net architecture, we use the Python library `tensorflow`, specifically its `keras` API. This gave us useful tools to implement the encoder and decoder layer and access readily available activation functions. We used a ReLU activation function. We implemented the encoder and decoder using the `keras` API, and we implemented all the model API ourselves, as well as the loss functions and the various utility functions such as data augmentation.

G. Training

We tried to train our model on one of the EPFL clusters (izar), but in the end, one of our home computers with an NVIDIA RTX 2070 appeared to be faster than what was available on the cluster, maybe because there were too much requests on the cluster at this time of the semester. The training took between 20 seconds up to 120 seconds per epoch, depending on the parameters.

III. Results

A. Hyperparameters optimization

We ran several phases of training to optimize our hyperparameters. We started by comparing variations in the input size and the number of blocks in the encoder and decoder. We iterated over input sizes 32, 64, and 128, and the number of blocks 3,4,5,6 and got the following results:

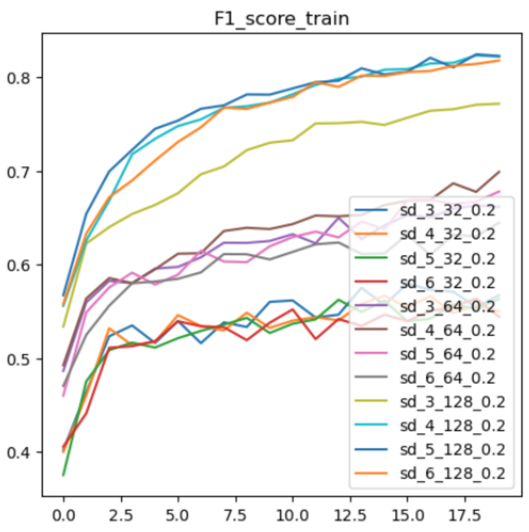

Fig. 3. F1 score for training. sd_X_Y_0.2 where X is the number of blocks and Y is the input size. (Dropout of 0.2)

As we can see, the variation in the number of blocks has not a significant impact, except at input size 128 with 3 blocks. On the contrary, the variation of input size has a great impact on the learning rate, with a higher input size being better.
For the second iteration, we added a higher input size and varied the dropout rate:

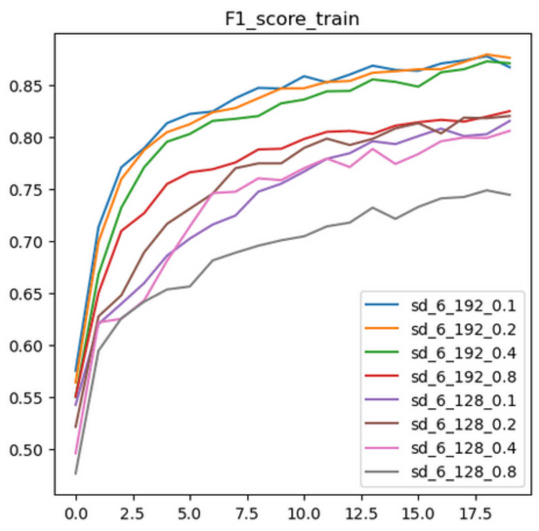

Fig. 4. F1 score for training. sd_6_X_Y, where X is the input size and Y is the dropout rate.

Here we can see that further augmenting the input size improves the learning rate, but augmenting the dropout rate above 0.4 diminished it, while changes don't have much impact between 0.1 and 0.4.
Again we further augmented the input size and tried to modify the angle amplitude for the rotations of the data augmentation, but we got similar results, only the input size had a significant

impact.

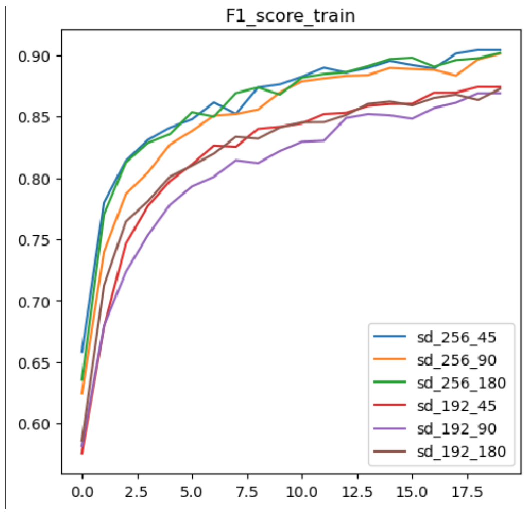

Fig. 5. F1 score for training. sd_X_Y, where X is the input size and Y is the image rotation amplitude for data augmentation.

At this point, we hit a ceiling with input size, having memory issues with higher sizes. We then tried to iterate over different numbers of batches per epoch and tried to remove the dropout rate:

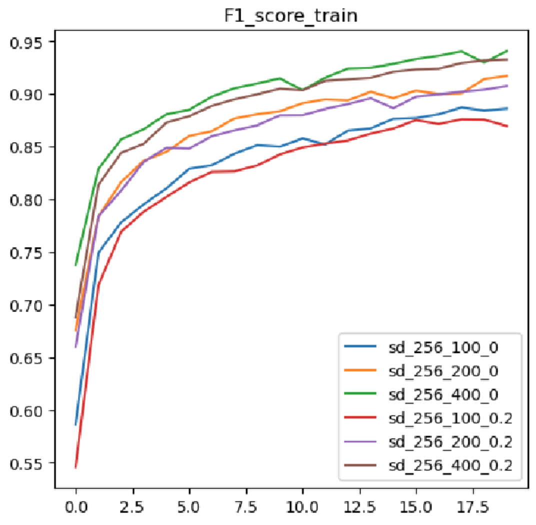

Fig. 6. F1 score for training. sd_256_X_Y, where X is the batch per epoch number and Y is the dropout rate.

We found interesting results: removing dropout entirely improved slightly performance, and increasing the number of batches per epoch also had a noticeable impact. We didn't go higher than 400 batches per epoch because the training time that what going out of hand.

B. Final training iteration

During the process of optimizing our parameters, we submitted 3 close predictions on the final test set, all trained over 1000 epochs:

	input size	dropout	batches per epoch	F1 score	accuracy
1	128	0.2	200	0.885	0.940
2	256	0.2	200	0.895	0.945
3	256	0	400	0.895	0.945

TABLE I
TEST SET PREDICTIONS RESULTS

Those results show that our incremental optimization was indeed useful. We can see that parameters 2 and 3 yielded the same final result, even tho 3 showed a better learning rate. This shows that the increase in batches per epoch had an effect of overfitting.

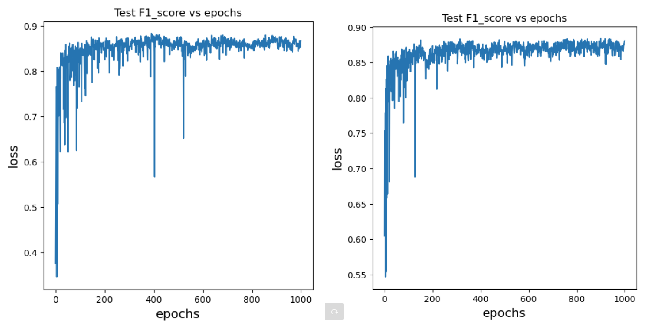

Fig. 7. F1 score for validation. 2 on the left and 3 on the right.

Here we can see that even tho the learning rate of 3 was noticeably better, this time the validation score doesn't follow this trend, it rather stays similar.

IV. Discussion

A. Methodology limitations

Our methodology can be improved on multiple fronts. As already explained, one possible improvement is to optimize our hyperparameters further, with more iterations and ideally more computation power. An other obvious improvement in our method can be to extend the data augmentation process to get more training samples. We thought about adding noise to the images in addition to rotation for example. More work can also be put in finding a better loss function for segmentation, for example a combination of dice loss and cross entropy has shown to be promising in other contexts[1]. Finally, we could have also tried to modify the activation function.

V. Summary

Although our implementation is far from perfect, we can see that we managed to get very promising results using a U-Net architecture, which suggests that with more effort, it would be fairly doable to get even better results. We showed that not all hyperparameters where equally relevant to optimize our U-Net architecture and that the input size of the model was the main factor but also a computational limit.

References

[1] Adrian Galdran, Gustavo Carneiro, and Miguel Ángel González Ballester. *On the Optimal Combination of Cross-Entropy and Soft Dice Losses for Lesion Segmentation with Out-of-Distribution Robustness*. 2022. DOI: 10.48550/ARXIV.2209.06078. URL: https://arxiv.org/abs/2209.06078.

[2] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. “U-Net: Convolutional Networks for Biomedical Image Segmentation”. In: (2015). DOI: 10.48550/ARXIV.1505.04597. URL: https://arxiv.org/abs/1505.04597.