Road Segmentation using Convolutional Neural Networks

Perret, Thibaud
thibaud.perret@epfl.ch

Feng, Paul
paul.feng@epfl.ch

Madillo, Raphaël
raphael.madillo@epfl.ch

***Abstract*—This paper goes around convolutional neural networks used for feature detection in images, namely segmenting roads on satellite pictures, in the context of a Kaggle competition. We talk about the augmentation of input data and give a description of the methods used to improve our neural network architecture . We ended up with a F1-score of 0.9018 on Kaggle.**

I. Introduction

Image segmentation has come a long way in the last decade or so. While traditional image algorithms for edge detection and feature detection still prove to be efficient, newer techniques are being introduced with the rise of Machine Learning and Deep Learning. In particular, the use of convolutional neural networks is becoming the reference standard for such tasks. We will show how we used those networks to do image segmentation to detect roads in the images. We used a dataset of 100 satellite pictures (1) with their corresponding groundtruth images; black and white binary images, white representing the roads (2).

Figure 1. Satellite image

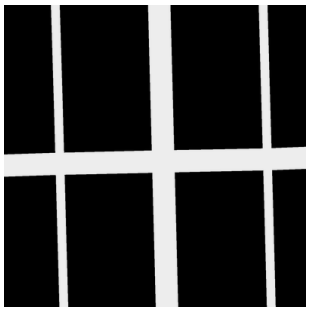

Figure 2. Groundtruth image

II. Data augmentation

Machine learning performance is deeply related to the amount of data we are able to collect. Usually, the more data we feed into the system, the better the results are. This is especially true when using neural networks in order to avoid overfitting. Neural networks are described in the next section. In our case, the data we were given comprised 100 images which is a relatively low number to perform a training. To generate more data, we simply flipped the image horizontally (we could have flipped it vertically also but we figured we had enough data with one flip), and we rotated each result in every direction (i.e. 0°, 90°, 180° and 270°). Thus, one image from the initial dataset generates 7 alternative forms. We ended up with 800 satellite images and groundtruth. What is interesting though, is that the machine consider no similarity between these images as it just reads a different set of number that have nothing in common. This means it really learns new way of detecting roads from the same data.

III. Neural networks

A. Basic neural networks

A neural network is a tool to learn from input which try to reproduce what it learns on new data. It is composed of an input layer, several hidden layers and one output layer. The input could be whatever data as long as it is possible to transform it as vectors of numbers. Each layer is composed of nodes, where weights and biases are associated to them. The machine then uses theses weights and biases to apply computations on the input vector, combining all those numbers, producing new numbers that are used as input of another layer. The weights and biases are combined and are passed through an activation function (a simple non linear function to remap the data).
The learning part resides in the back propagation algorithm. Thanks to the chain rule property that computes the product of an upstream gradient and a local gradient, the network is able to correct the weights and biases of each layer, and this way, the whole network gets modeled after the data used.

B. Convolutional neural networks

Neural networks with many layers tend to get really large easily. If each node from a layer of K_1 nodes is connected to each node of another layer of K_2 nodes, we have $K_1 \cdot K_2$ weights and K_2 biases. An image which is $N \times N$ pixels can be represented as a vector of N^2, which means the input layer has this many nodes. Thus the number of parameters in the network can really explode if we have many layers densely connected. Convolutional neural networks comes to solve this problem in specific cases. When we work with images, the structure is so that pixels that are neighbors are probably related, i.e. they are not put in random. However, pixels that are far away in the picture may not be related. The convolutional neural network puts links from neighbor pixels to a single node. This means we deeply reduce the number of parameters to optimize.

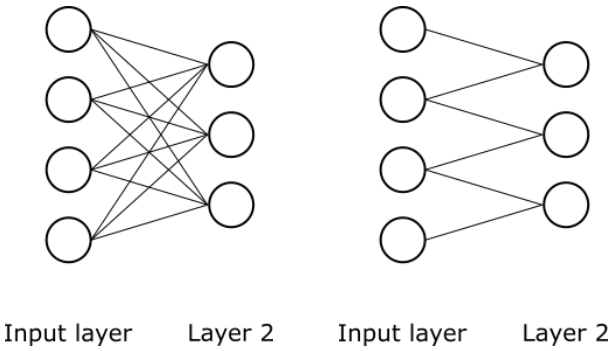

Figure 3. Dense and lighter networks

To do an even better job, the weights are shared between every link, which means that for every neighborhood, weights are associated to position within that neighborhood, and the weights are the same for every neighborhood in the image. The biases can be also set to 0. Thus, we get even less parameters. For example, we will later use pictures of 72×72 pixels as input, so let's take this size as our input. If the second layer we use has the same size, a fully-connected network would get $72^2 \cdot 72^2 + 72^2 = 26879040$ parameters. Now if we use a convolution layer instead with a neighborhood defined as a 3×3 square around a position, we would reduce the number of parameters to $9 + 1 = 10$.
The fact that we use neighborhoods, and that the weights are the same means that we can represents these weights as a filter that we use for a *convolution* on the image.

IV. Inputs and outputs

A. A naive input

The trivial way we could have used for the segmentation would be a *per-pixel classification*. Using this technique, the network tries to predict whether each pixel is representing the road or the background, based only on this single pixel. This way, the machine can only learn about the color of the pixel. The classification is thus simple and efficient, but it does not give that good of a score.

B. A time-saving approach

The technique above can be augmented to a *per-patch classification*, where the classification is done on patches of pixel, so that we do not have to put every pixel as the input, but only a predetermined number of patches. A patch is a small piece of the image (16 x 16) centered at a certain pixel of the image. From those patches, it is possible to extract features such as the mean or the variance. All the images are divided into a set of patches. It's a patch-level classification. This method is prefered because of its efficiency and simplicity. The prediction is of course less precise but for this particular classification it proves to be adequat and the time saved can be used at better ends.

C. The sliding window

When we want to see whether a position is road or not, it is interesting to know a context around it. A case where this might indeed be useful is when trees are obstructing the asphalt. If we only learn from color of the position, either the machine will learn than only different shades of gray are road and it will predict that a tree is not road, or the green color from the tree is going to falsify the color for the road and then a tree will be predicted as road even if it is not above a road. In both cases, this is not the expected outcome.
We used the concept of a *sliding window*. The window represents the context around the position that we want to classify. For example we define a window of 72×72 pixels around a given position, and feed this sub-image as input in our network. We then have a heavier input, as we have a certain number of windows (that may even overlap) for each position we are interested in. However, the context gives us valuable information which leads to better outcomes.

D. Output labels

As said above, the groundtruth image is separated into patches. Doing so boils down to the same thing as using a more pixelated version of itself. That is, for each position where we want the resulting classification for the patch, we get the pixels around the position with a given patch size, and we compute the mean of those pixels. If the mean is ≥ 0.5, then we say that this patch is road, otherwise, the patch is background. Rather than returning a pixel o a certain color, we simply return a label, being 1 for road or 0 for background.

E. Construction of the training data

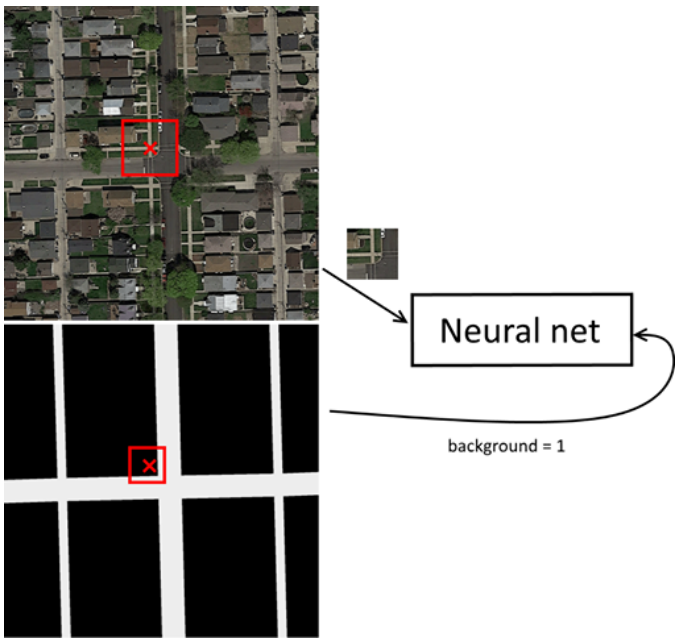

Figure 4. Construction of the input and output training data

We then depend on three things, namely the image size, the window size and the patch size. Knowing the image size and the patch size, we can know how many patches we will be able to get from the image (everywhere where a patch fits in the window), and the position of these patches. With these

positions, we get windows of the defined size around each position, that we will use as input for our network. With the positions, we also get a label (from the groundtruth image) for each position. The input-output training data is then a window matched to a label (4). It is important to note that the windows overlap because the window size is larger than the patch size.

F. Image extension

Since the window size is larger than the patch size, the window might get out of bounds for pixels that are near the border. The simple way to solve this is to extend the image to a bigger size using a mirroring technique. Each pixel outside the image takes the value at the pixel inside the image where the positions are reflections of each other regarding the border.

V. Our neural network

A. The model

Our model follows a rather simple structure consisting of a succession of the same layers: 2D Convolution - Max Pooling - Dropout. At the end of the network, we also have this succession: Flatten - Dense - Dropout - Dense This is a structure that is similar to the AlexNet which is one of the first deep neural network with a light architecture compared to its peers. Deep learning structures trend toward lower filter size and increasing number of layers and filters. (Often a power of 2). In fact 3 3x3 filters have the same effective receptive field as one 7x7 filter while having fewer parameters. 3^3 vs 7^2 weights. A VGG [SZ15] like-architecture was also tested (CONV-RELU-CONV-RELU-POOL)-like unit. While it yielded similar results, it was computationnaly more expensive than our current model.

1) 2D Convolution: As mentioned above, the weights are shared and only take into account th neighborhood, which means we can represent the weights as filters. The filter is then used for a 2D convolution on the input window. The stride is kept at one and the padding is maintained the same in order to keep the size of images intact. A single filter produces an activation map with the same dimensions but a depth of 1.

2) Max Pooling: Max pooling is a sample-based discretization process that downsamples an image to a lower resolution form. We take a window of 2×2 pixels that we transform into a single pixel, its value being the max of the values of the pixels in the window. We then get 4 times less pixels. This is a way to reduce computational cost and ovefitting while maintaining similar results in the optimization. It does not introduce any additional parameters.

3) Dropout: The dropout layer is used to put certain weights to 0 with a given probability. This is mainly used to tackle overfitting problems. The max dropout value is 0.5 and is proven to be the best value in general.

4) Flatten: The data we use until the flatten layer is three dimensional. We always talked about 2 dimension before because we use images, but each pixel has 3 channels (red, green and blue), which adds a dimension. The flatten layer is used to squeeze the data to a (large) one dimensional vector.

5) Dense: A dense layer is a fully connected layer where every node of a layer L is connected to every node of the Layer L + 1.
We used 2 different activation function (as specified in the table I): the LeakyReLU and the Sigmoid. The functions have the shape shown in fig 5 and 6.

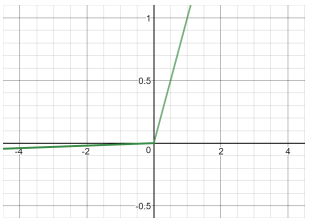

Figure 5. LeakyReLU function

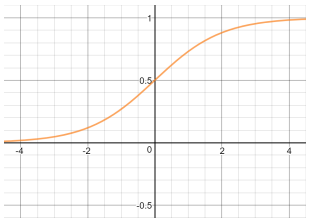

Figure 6. Sigmoid function

The LeakyReLU is a modified version of the ReLU function which is defined as $ReLU(x) = x$ if $x \geq 0, 0$ elsewhere. This is used so that there is no more negative values (they are set to 0). The specificity of the LeakyReLU is that it allows a "leak", as shown in the fig 5 for the negative values, i.e. the negative values are not directly mapped to 0 but to a less negative value.
The sigmoid function is a function which maps the real numbers to the interval $[0, 1]$. The output can be interpreted as a probability, in our case, the probability of the patch being a road.
Here is a summary of our model with the input size of each layer, as well as the number of parameters the layer adds:

Layer	Input size	Number of params
2D Conv.* (32 filters)	(72, 72, 3)	896
Max pooling	(72, 72, 32)	0
Dropout (prob. 0.25)	(36, 36, 32)	0
2D Conv .*(64 filters)	(36, 36, 32)	18496
Max pooling	(36, 36, 64)	0
Dropout (prob. 0.25)	(18, 18, 64)	0
2D Conv.* (128 filters)	(18, 18, 128)	73856
Max pooling	(9, 9, 128)	0
Dropout (prob. 0.25)	(9, 9, 128)	0
Flatten	(9, 9, 128)	0
Dense of 512*	10368	5308928
Dropout (prob. 0.25)	512	0
Dense of 1**	1	513
		Total: 5 402 689

* used with LeakyReLU
** used with sigmoid

Table I
Summary of our CNN

For the optimization, we chose to use the Adam optimizer, a recent technique that proves to be the most efficient . It's based on Adagrad, an adaptive learning rate method. It is

used to normalize the parameter update step.

$$m = \beta_1 * m + (1 - \beta_1) * dx$$
$$v = \beta_2 * v + (1 - \beta_2) * (dx * *2)$$
$$x+ = -LR * m/(\sqrt{(v)} + \epsilon)$$

with m, β_1, β_2 and ϵ being hyperparameters. ϵ ensures a non-zero division.
It is also important to update the learning rate with the callback method when the validation test loss is stale. This helps to reach the minimum of the loss without getting stuck at some stage. $reduceLR = ReduceLROnPlateau(monitor = val\_loss, factor = 0.3, patience = 5, min\_lr = 0.0)$

B. Implementation

To implement our neural network, we used the Keras library. This library is running on the Tensorflow backend, and it is very high-level. The layers are added like `model.add(Dense(2))` or `model.add(Conv2D(32, (3, 3))`, so the implementation in itself was not the biggest challenge of the project.

C. Running

When we use all of the non-augmented data, and we create our windows, we get 62500 windows of 72×72 pixels, each pixel being 3 float values. This large amount of data quickly becomes unmanageable if we limit ourselves with CPUs. Tensorflow has an easy way of working with GPUs after setting up CUDA and CUDNN. We could then run our programs much faster. Our final run of the model described above was done on a NVIDIA GeForce GTX 960M and it ran for a total of 6 hours.

VI. Results

A. Setup

We used 500 images chosen randomly for training, and 50 images for validating. We ran it for 30 epochs (one epoch = one forward pass and one backward pass ofall the training examples), using a batch size of 128. Keras provide us with a report when the program has stopped which tells us this:

	precision	recall	f1-score	support
0	1.00	0.97	0.98	24948
1	0.88	0.99	0.93	5677
total	0.98	0.97	0.97	30625

The validating data is taken from different images than the training, but the images are still similar, meaning the rotation of the roads are always the same from image to image. This means that our F1-score is really good on our validation data (0.97 as mentionned in the table above), but our score on Kaggle is less good (0.90). The data we have to predict on contain roads that are diagonal, and our training data did not have many example to train on that. Thus, the horizontal and vertical roads are pretty well predicted, whereas diagonal one not so much.

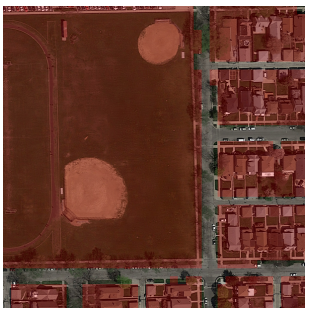

Figure 7. Good example

Figure 8. Bad example

VII. Conclusion

The heart of the project is to find a compromise between the complexity of the convolutional neural net which in theory yields better results and overfitting which has the opposite effect. The prediction score overall increases with the number of epochs and the batch size. Coupled with the fact that the architecture of convolutional neural network is chosen through trial and error processes, it is highly recommanded to use powerful engines hence the use of the GPU that is adapted to numerous and parallel operations.
To go further, inception modules, that are used in the GoogLeNet architecture would have been tested. It's a module that concatenates the activation map of several sized filters from the same input. This allows the model to take advantage of multi-level feature extraction from each input.

References

[SZ15] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large scale image recognition, 2015.