UCLouvain

École polytechnique de Louvain

Change Detection in Satellite Imagery using Deep Learning

Authors: **Héloïse Baudhuin, Antoine Lambot**
Supervisor: **Pierre Schaus**
Readers: **Siegfried Nijssen, Guillaume Derval**
Academic year 2019–2020
Master [120] in Computer Science and Engineering

ABSTRACT

Change detection in satellite imagery is an excellent way to monitor the evolution of geographical areas. Although object detection in that field is already well-developed, change detection by semantic segmentation isn't, but is a more direct way of seeing the changes in the infrastructure of a region over time. With two picture of a region at different times, we can build a segmentation mask outlining the contours of new buildings.

This thesis aims to develop an architecture that performs change detection in buildings in urban areas. After building a dataset, we first study the performances of the UNet architecture. There is a loss of localisation precision with this model. To remedy to this we study two modular UNets with different depths and a UNet++, both evolutions of the UNet. The performances of 4 different models are compared. Our results show that even with a small architecture of a UNet++, we obtain segmentation masks that allow for an efficient detection of new buildings. The F1 score obtained on the test set is of 42% and visibly neater contours on the masks. Lead for further improvement of the best-performing model are included.

The implementation of this project can be found on the LamboiseNet GitHub repository : `https://github.com/hbaudhuin/LamboiseNet`[2].

Contents

Chapter 1

Introduction

In the last years, Deep Learning architectures have outperformed the state of the art in computer vision tasks [12]. Although Convolutional Neural Networks were developed a few decades ago [15], their performances were held back by the scarcity of large datasets and the limited computational power available. The first real breakthrough by Krizhevsky et al. [12] was due to their capacity to train a larger network and use a big dataset. This paved the way to larger networks and improved performances.

The typical Convolutional Neural Networks application is for classification but for many computer vision tasks, such as satellite imagery analysis, localization is key. Instead of being only one class label, the desired output should have a class label for each pixel of the input. This is called image segmentation.

Aerospacelab is a Belgian startup interested in the analysis of satellite imagery. Based on two images of the same place taken at different times, they wanted to be able to know if anything had changed. Our thesis focuses on the creation of a Convolutional Neural Network capable of detecting changes in buildings and returning a segmentation mask with the shape of the new buildings. Two challenges are surrounding this: first, creating a dataset; secondly, finding an architecture that will be efficient even with the smaller size of our handmade dataset.

When it comes to change detection in satellite imagery to produce a segmentation mask, publications are scarce. It has been done but with imagery with a much lower resolution [10]. Instead of building upon previous work on satellite imagery segmentation, we focused on image segmentation in general. One

of the best known networks to perform image segmentation is the UNet designed by Olaf Ronneberger et al[18] in 2015 for biomedical imagery segmentation. Their goal was to create a CNN capable of semantic segmentation on a small dataset. This architecture was based on the fully connected network but extended it to work with very few training images and to yield more precise segmentation masks. They did so mainly by adding skip pathways to propagate context along the network. With this network, they obtained an architecture capable of localisation, and robust for a small dataset.

With our small dataset, we started by training a UNet and analyzing the results. Although the masks were promising and the F1 score on the test set was around 0.4, the main imperfections were around the edges of the new buildings detected. This indicated that although the localisation was good, small scale details were lost. We then switched to a newer variation of the UNet, the UNet++ [27] specifically designed to target this problem and two modular UNet's, shallower versions of the UNet. Both obtained comparable or higher F1 scores on the test set but the UNet++ brought clear improvements visible on the output masks.

In this thesis, we will first start by introducing the context with a general problem definition. Then will follow a more theoretical overview of the inner workings of CNN architectures in the third chapter. In the fourth chapter, we will present work and publications related to our problem. A formal problem definition will follow in chapter five. The sixth chapter will focus on our dataset and the seventh will shortly touch on computational requirements. The key chapter, "Choice of architecture", will present the different architectures used and compare the results obtained from them. The chapter "Code" will give short explanations regarding the structure of our implementation and the role of each file. The tenth chapter will shortly present ideas for future improvements and the eleventh chapter will summarize the thesis with a final conclusion.

Chapter 2

Context

This section introduces the client, AerospaceLab, and defines the problem and the initial interest of the client in our solution.

We started this thesis in collaboration with AerospaceLab but around the month of November, they realized they hadn't enough time to continue collaborating. We then continued the thesis without their input but in the same direction that they had given us initially.

2.1 AerospaceLab

AerospaceLab is a Belgian startup located in Mont-Saint-Guibert. They were founded by Benoît Deper, an ex-NASA employee. Their goal is to revolution space intelligence. They have two core businesses.

The first one is to develop tools ranging from surveying to monitoring using satellite imagery. They also build their own dataset to be incorporated in their clients dataset.

The second is the building of versatile small satellites. Their satellites will be equipped with a variety of sensors collecting high resolution optical data multiple times per day on selected target areas. They are currently analyzing and tasking the data that they will retrieve with machine learning techniques. In April 2019, AerospaceLab announced it would try to launch 5 satellites within the year.

2.2 Client needs

The client first description was very open. They wanted to discover changes in satellite imagery but were open to different change types because AerospaceLab themselves had not implemented the detection of all types and were interested by a variety of change detection type. Different subjects of changes were discussed such as forest, petroleum platforms, farms, etc. After discussion with our promoter and with AerospaceLab, it was decided this thesis would focus on changes in houses and/or buildings.

2.3 General problem definition

Our model takes two images of the same area at different times as input and needs to output a result that shows if and where buildings were built.

Figure 2.1: First image taken on an earlier time

Figure 2.2: Second image with a new house

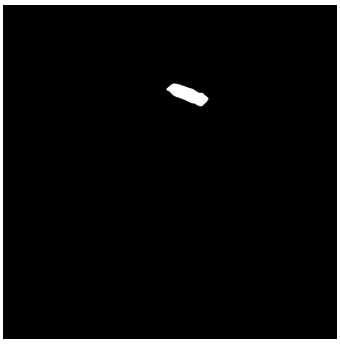

Figure 2.3: Mask showing the new house

The output is a segmentation mask. This mask has the same resolution as the input images. This means that if the mask is superposed on the second image, the shape of the new house is exactly represented on the mask.

The justification behind the design choices will be detailed later on.

2.4 Deep learning for big data

Deep learning is used for this task rather than other computer vision techniques. The first reason behind that is big data. What is meant by big data is that nowadays extremely large amount of data can be accessed and in turn can only be processed by certain techniques.

Deep neural networks are one of such techniques. It is capable, from huge amount of data, of drawing conclusion and pattern that could not be found otherwise. This has been proven to be particularly true with problems dealing with imagery [12].

Deep leaning distinguishes itself from other supervised ML techniques like SVM or single layer perceptron because its accuracy reaches a plateau at higher performances for the same amount of data.

A second reason is that compared to traditional Computer vision techniques, deep learning allows to achieve greater accuracy in image classification [12] and semantic segmentation [18] which are what this thesis focuses on .

The third reason is that deep learning networks are trained rather than programmed. This means expert analysis isn't indispensable to exploit our data. Our architecture will also be less domain-specific than traditional computer vision algorithms.

Chapter 3

Deep learning for Computer Vision

This section describes the core principles of Convolutional Neural Networks (CNN).

3.1 Convolutional Neural Networks

These last years, AI has grown significantly in bridging the gap between the capabilities of humans and machines. Both researchers and enthusiasts have worked on numerous aspects of the field. One of those fields in Computer Vision.[12]

The goal for this field is to enable machines to perceive the world the same way as humans do, and from there to be capable of using the knowledge they acquired in other tasks such as NLP, Image Analysis, Media Recognition, etc. The Computer Vision task has evolved primarily over a single Neural Net architecture type: the Convolutional Neural Network (CNN). This section aims at giving a short introduction to this architecture and obviously not a full course. It only present the most important aspect in the context of this thesis.

A CNN is a Deep Learning algorithm that takes an image as input in the format of 3 RGB matrices[1][24]. It assigns importance to different features of the input image. In earlier computer vision algorithms, the filters applied to an image

[1]This is a standard input. A CNN can take multiple images as input. The distinction is that it doesn't flatten the matrix to take a very long input vector like other Neural Networks.

were hand coded to detect specific types of objects or aspects. A CNN is capable of learning its own filter on its own and even detect aspects that are undetectable to the human eye.

3.1.1 Why CNNs over Feed-Forward Neural Nets

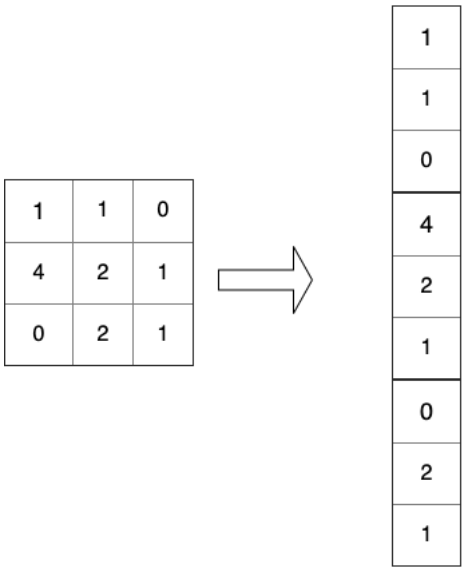

Figure 3.1: Flattening a matrix into a vector

An image is a matrix of pixel values. Why not flatten the matrix and simply feed it to a standard Feed-Forward Network ? Actually this technique performs very poorly.

If the matrix is flattened, all spatial features are lost. For example, if the goal is to detect a line on an image, the human eye checks if pixels next to each other form a line of same colour. If the image is flattened, it becomes much more complicated !

But also imagine that the input is of size $3 \times 256 \times 256$, which means that the input is made of 196608 values, and the first layer has 1024 neurons. In a fully connected Network all neurons are connected together and it would need 1024×196608 so 2.013.265.992 parameters for the first layer ! Computationally, using a Feed-Forward net is impossible.

A CNN turns towards filters. These are smaller weight matrix applied over all the input image. The same filters can be used everywhere on an image

because they have stationary properties. In other words, if a filter is capable of detecting a line in the bottom of the image, it can detect a line everywhere else in the picture. CNNs have locally connected neurons and not fully connected ones. A CNN is thus capable of using the spatial and stationary dependencies in an image with filters and be much lighter than standard Feed-Forward Neural Net.

3.1.2 Convolutional layer

Feature extraction with a convolutional kernel

A convolutional kernel is a series of filters. A filter is a matrix of weights that is applied on a subset of the input pixel values. It produces the result coming from matrix multiplication of the corresponding terms from the filter and the input pixels. The result is then summed up. The filter is then slid across the entire activation volume resulting in a feature map, in other words a filtered version of the image.

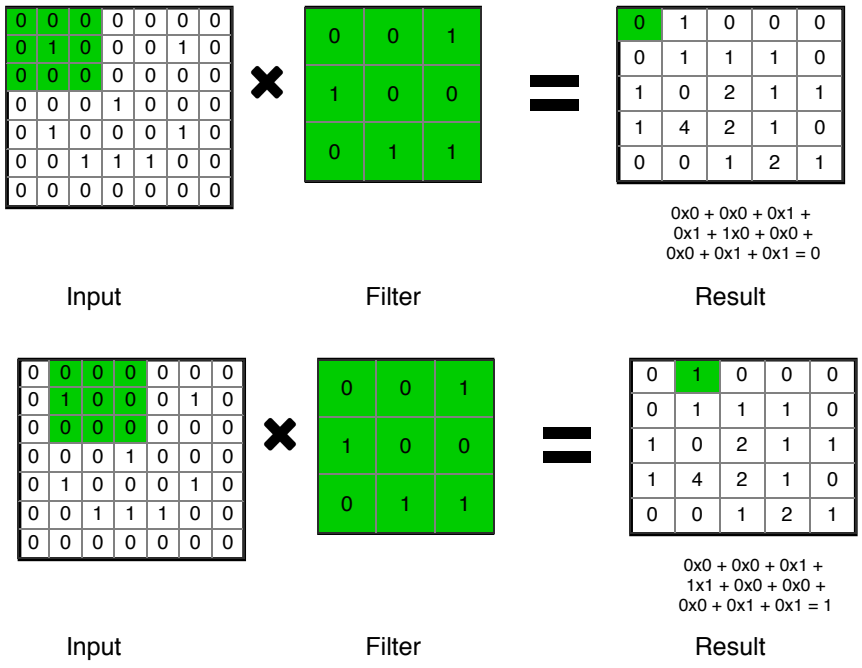

Figure 3.2: First two steps of a convolution

The parameters of a convolutional filter are :

- The padding : The simple application results in a feature map that is smaller in size compared to the input volume. To keep the dimensions identical or higher, the technique is to "pad" the image with pixels along the borders. An example of padding of 1 allowing the output to be of the same size as the input is shown in Figure 3.3.

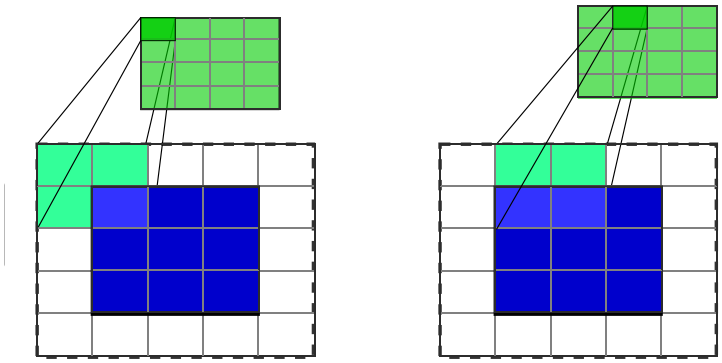

Figure 3.3: Example of padding of 1

- The stride : The stride is the number of pixel a filter slides to the left before applying the filter on a new subset of pixels. In the example shown in figure Figure 3.4, the stride is of two. The filter is sliding two pixels to the right. The output is also smaller because less values are are computed.

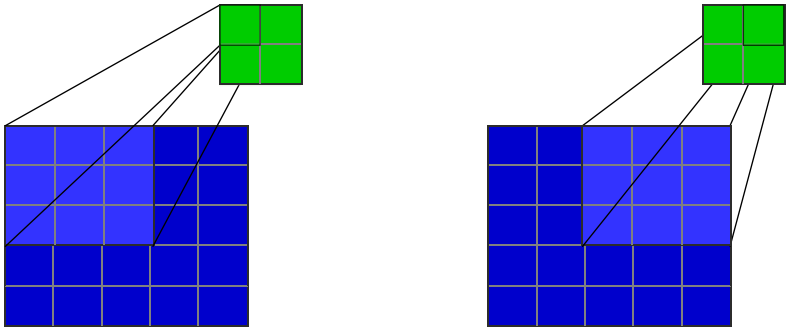

Figure 3.4: First and second step of a convolution with a stride of 2

- The number of filters : In a traditional convolutional layers, there are often much more than 3 filters. 16 or even 64 filters are common.
- The size of the filters.

Feature map creation with an activation function

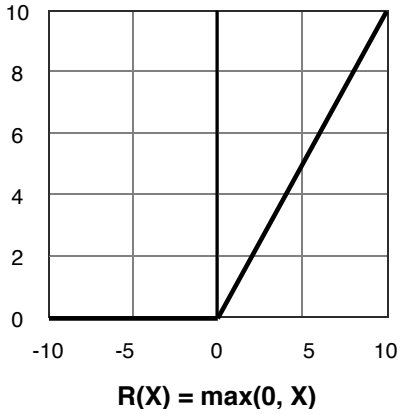

Figure 3.5: ReLU activation function.

An activation function is a function taking the output of the convolution in the feature map and indicates whether it should send a signal to the next layer. This means that less processing is required and that only the important information is forwarded.

The activation function that is used traditionally is the ReLU shown in figure 3.5.[5] For any input under the value of zero, the ReLU outputs a zero. It allows to generalise the learning and diminish the noise within the network.

3.1.3 Up sampling layers

The transposed convolutional layer or more commonly called the "up sampling" layer, is used to up sample from low-resolution to high-resolution.[21] In other words, the goal is to decompress the abstract representation into something larger. There are different methods of up sampling relying on interpolation but transposed convolutions have the advantage that the network can learn to up sample optimally for his problem.

In a standard convolutional layer, the sum of the element-wise multiplication between the input and the kernel matrix is computed. This is a many-to-one operation. A transposed convolution does the opposite. It associates one value with many.

Let's start by describing the basic idea behind this layer: suppose a kernel,

represented in figure 3.6, of size 3×3 and an input of size 4×4. The kernel can be represented as in figure 3.7 in a convolutional matrix. A convolutional matrix is nothing else than a matrix which demonstrates all position of the kernel on the original input. Each row is only a rearranged kernel with zero padding at different places. A matrix product between the input matrix and this matrix outputs the same result as a convolutional operation between the input and the kernel.

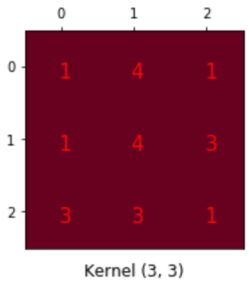

Figure 3.6: Convolutional kernel from [21]

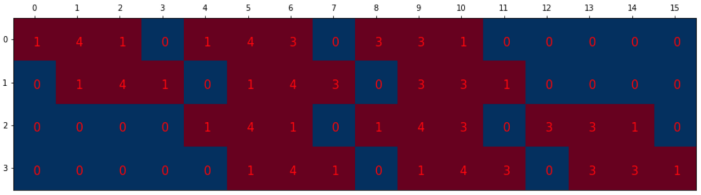

Figure 3.7: Same convolutional kernel 4×16 from [21]

The input matrix of 4×4 is then flattened in a vector of 16×1. A multiplication between the convolutional matrix (4×16) and the input vector (16×1) outputs an output of size 4×1 that can be rearranged as a 2×2 matrix. This would simply be performing a standard convolution. This is shown in figure 3.8. But what the convolutional matrix can be transposed. The operation becomes a multiplication of a 16×4 matrix with an input of size 4×1. The output becomes of size 16×1 which can be rearranged into a 4×4 matrix. This successfully up samples the input matrix. This is shown in figure 3.9.

As such, the transposed convolution is not a simple transposition of a convolutional matrix. The key focus is that the association between the input, turned into a convolutional matrix, and the output is handled backwards. It is also not a convolution in itself but it emulates the transposed convolution with a convolution.

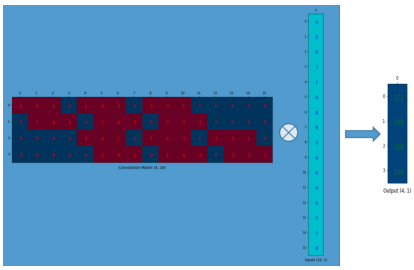

Figure 3.8: Standard convolution. The output dimensions are smaller than the input. Image from [21].

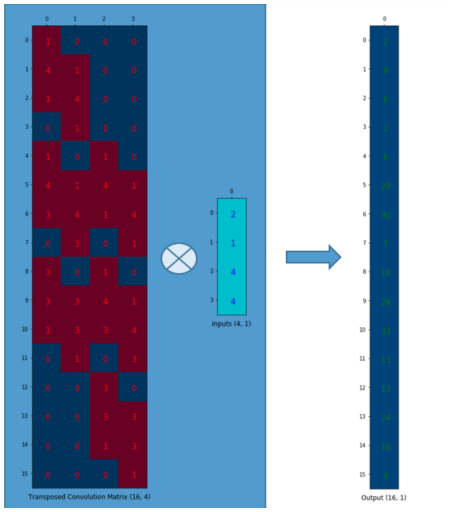

Figure 3.9: Transposed convolution. The size of the output is larger than the size of the input. Image from [21].

The output of this layer then goes through an activation function like a standard output of a convolutional layer.

3.1.4 Pooling layers

Much like a convolutional layer, a pooling layer reduces the dimensions of the convolved features.[4] There are two reasons for wanting to reduce the dimensions : smaller dimensions means less computations and it is useful for extracting dominant features.

There are two types of pooling : max pooling and average pooling. Max pooling takes the maximum value of the filter and average pooling averages all values.

By down sampling, the models understands better what is in the image but looses information about where it is.

Figure 3.10 shows the results of two types of pooling.

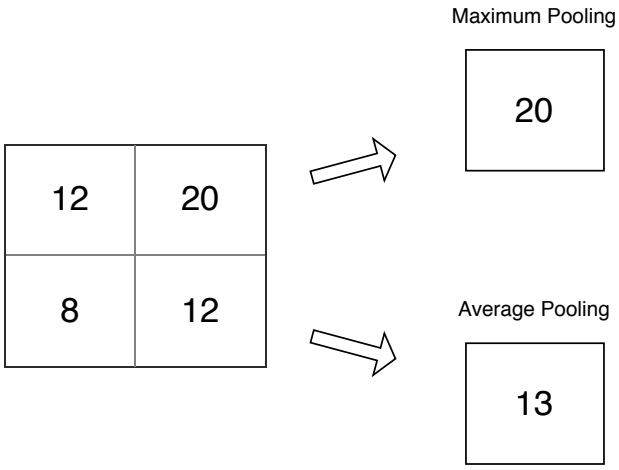

Figure 3.10: Two type of pooling

3.1.5 Batch normalisation

Batch normalisation is usually applied after each convolutional layer, before the activation function [6]. It normalises the output of the convolutional layer. The intuition behind it is that if the input is normalised, why shouldn't the outputs in the hidden layers be too ?

Batch normalisation reduces the shift of the hidden unit values which helps with generalisation. It also allows each layer to learn more independently of the other layers. This allows the use of a higher learning rate because it guarantees that there will be no dramatic shifts in the output of the hidden units.

Batch normalisation normalises the output of the convolutional layer by subtracting the batch mean and dividing by the batch standard deviation. However, this means that the weight of the next layer are not optimal anymore. A gradient descent pass would undo the normalisation because it would lower the loss. To remedy to this, batch normalisation adds two trainable parameter to each layer. One will multiply the input and another will be added. The gradient descent will focus on finding the best value of these two values instead of modifying all the weights.

3.1.6 Final layer

Once the feature map are all learned and the input is brewed down to its key features, it is time to add a layer that decides what to do with the information. There are different types of granularity in the understanding of an image.[14]

Image classification

In image classification, the input is an image and the CNN is expected to output a label corresponding to the main object in the image. Considering it only outputs one label, it is considered that there only is one main object.

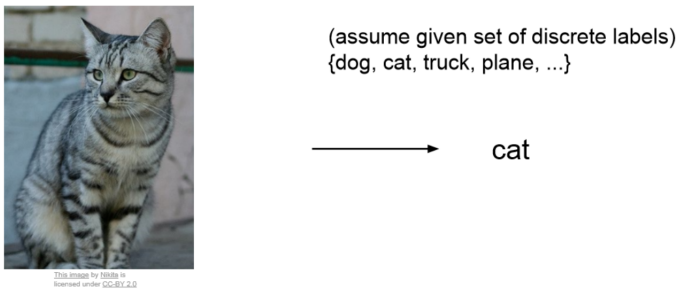

Figure 3.11: Image classification. Image from [14]

A variation is image classification and localisation. The label is output with a bounding box around the object detected.

Object detection

Object detection takes image classification to the next level. The CNN now detects multiple objects and surrounds them with bounding boxes.

Semantic segmentation

With semantic segmentation, the CNN labels each pixel of the input image with a class label. This is referred to as dense prediction.

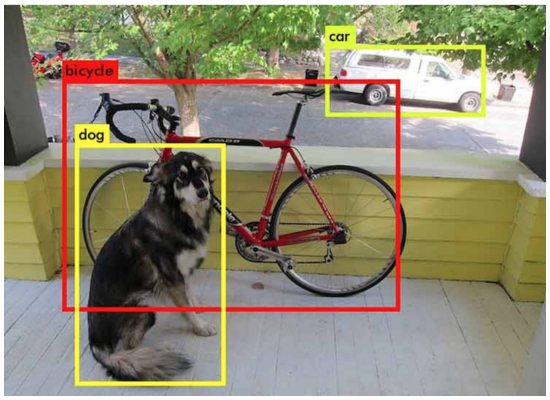

Figure 3.12: Object detection with YOLO. Image from [14]

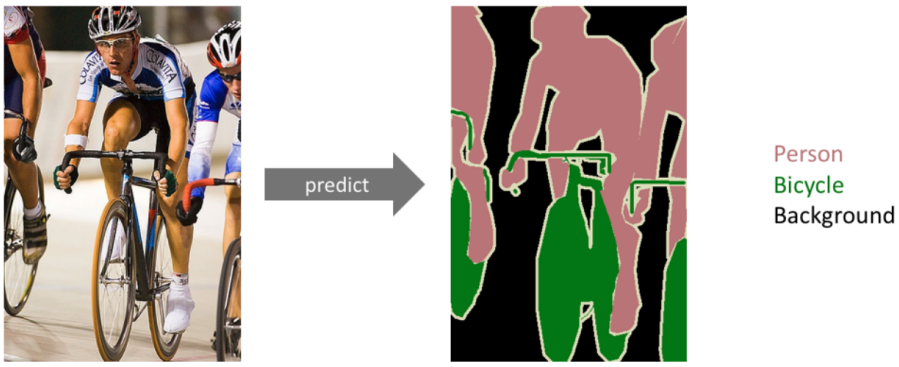

Figure 3.13: Image segmentation. Image from [14]

Unlike the two other methods above, the output is not a label or bounding box coordinates. The output is an image, usually of the same size and resolution as the input where each pixel is labeled. It is a pixel level classification.

3.2 Typical architecture of a CNN

As showed in figure 3.14, a typical CNN takes an image as input, applies a series of convolution followed by activation function and the pooling layers[19]. This is the feature learning phase. Afterwards, a classification phase is connected. That part is optional and specific to the task of the CNN. The key part of a CNN is the feature learning part.

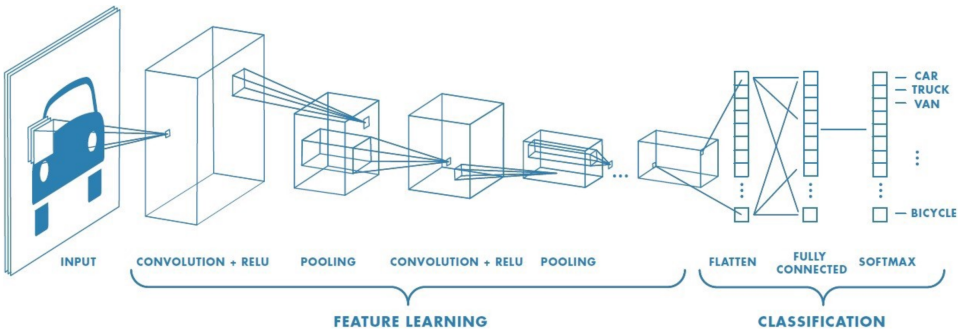

Figure 3.14: A typical CNN architecture for image classification. Image from [19].

3.3 Transfer Learning

When developing a Deep Learning model, instead of building a whole new architecture and training it from scratch, reusing an existing and already trained architecture and model that does a similar task is possible. Transfer Learning[22] is the idea of using knowledge acquired for a task to solve another one.

To apply Transfer Learning in Deep Learning, you would take an existing architecture, in its entirety or just a part of it, and build your own architecture on top of it, typically by adding layers at end of the model. Then, instead of initialising the layer weights at random for all layers, you can use the weights of the already trained model, for the layers you borrowed from it.

For CNN, examples of Transfer Learning would be using a simple Face Detection CNN to build a more complicated Facial Recognition CNN, or in this thesis's case, for satellite images, using a Building Detector to develop our Change Detector in buildings. These tasks are likely to transfer well to the new model because the deeper layers will need to learn the same features.

There are two main reasons Transfer Learning is often used in Deep Learning : the need for data and the training time. Training a model from scratch requires a lot of data and a lot of runtime. If 90% of the layers are already trained, and only some fine-tuning is necessary, a model will need a lot less data and time to complete the training.

But despite all these benefits, we chose not to use Transfer Learning for our model, due to the singular architecture of our model and the absence of a model pre-trained on satellite images that could have been easily adapted to our

architecture. Fortunately, we still managed to train our model from scratch in a reasonable time and to get good results.

3.4 Reusing vs building an architecture

We are using the PyTorch library[16] which allows to build a Neural Net layer by layer. We could have built our own architecture but chose to reuse a pre-existing one as a basis.

Building a network requires to have a complete understanding of the subtleties of each parameter of each layer, which we do not have. Our theoretical background allows us to understand the mechanisms used in networks, but inventing new ones is a different question altogether. Moreover, building a network that would have outperformed pre-existing ones would have required a lot more time (and expertise) than was available. It is a better idea to trust experts of the field and choose fitting architecture to our problem to fine tune.

Chapter 4

Related work

This section presents the two most relevant papers for this thesis that were explored before designing and implementing the model detailed in further chapters. It finishes with a more general conclusion based on these articles but also other not detailed here ([3],[9],[17], [20]). This articles have been useful to to understand the different directions we could take and what kind of results could be expected.

4.1 Change Detection in Synthetic Aperture Radar Images Based on Deep Neural Networks [10] (Maoguo Gong 2015)

4.1.1 Summary overview

This paper describes an early approach in change detection in images (here multi-temporal SAR images) using Deep Neural Networks. The paper quickly mentions the progress made in Deep Learning during the recent years (at the time) using CNNs, but convolutional layers are not used in their network, they use a Deep RBM model instead. In later papers about Change Detection in multi-temporal remote sensing images, the author switches to CNNs, showing that CNNs are probably more fit to the task.

The author explains in great detail the problem statement, the design choices and the training process. Some of the content is related to SAR images,

and thus not directly useful for our problem, but besides that the methodology, a part of the architecture, and the evaluation techniques can easily be reused for our problem.

This is also the oldest appearance of the "Early Fusion" architecture, where the images are concatenated before being input into the network, instead of generating a Difference Image.

4.1.2 Relevance to our work

The problem is similar to ours, so it is interesting to see what kind of issues the authors ran into, and how they solved it, even if, for some part, the solution is somewhat outdated.

More generally, we can already see what kind of approach we can take for our change detection problem.

4.2 Urban change detection for multispectral earth observation using CNN

4.2.1 Summary overview

Published in October 2018, this paper[7] explores the two ways to input two pictures of the same place but at different times. They first explore the "Early Fusion" (EF) architecture which consists of concatenating the two image pairs as the first step of the network. The input is then processed as a single patch. The last layer of the CNN is a softmax layer with two outputs associated with the classes of change and no change.

The second architecture is the "Siamese" (Siam) one. The two images are processed in parallel by two branches of four convolutional layers with shared weights and the outputs are concatenated. They then use 2 fully connected layers to obtain output values as before.

For their experiment, they varied the number of channels between 3 and 13 and measured the accuracy, the change accuracy and non-change accuracy. In

conclusion, the EF has a better accuracy in a large majority of cases and the addition of color channels leads generally to an improvement in classification but not in a linear way.

4.2.2 Relevance to our work

This paper focuses on a very similar problem than the one we have, albeit at much lower resolution. Their comparison between EF and Siam type of architecture is applicable for us too, and their findings about the number of channels improving the accuracy is consistent with many other papers cited in the introduction of this chapter which tells us that we should focus on using them appropriately.

4.3 Conclusion

This section went over interesting research papers in the field of change detection.

The most commonly used CNN are ResNets, UNets, and Inception. For every problem, the CNN was tailored to fit it best. We have based our tests on these architectures and have finally chosen a UNet.

After discussion with our supervisor and with AeroSpaceLab, we realised that the problem we are facing can be approached either by doing change detection, or object detection in two images followed by a comparison. Depending on the direction we choose, we will have to fine tune pre-existing object detection models or we will have to create something more or less from scratch in change detection. Starting from there we researched both fields.

A first conclusion we have is that high resolution satellite imagery change detection is a mostly unexplored field. Although people have done it before, they applied it on lower resolution imagery and on a very small dataset. Other papers([9],[17], [20]), not described here because redundant to the ones presented above, used a dataset of 20 images or datasets covering a very small town at best. This may be explained by the lack of publicly available and labeled multi-temporal datasets.

Secondly, we can already conclude that for change detection the **Early Fusion** approach is the most promising one. We presented a paper [7] that reached

the conclusion that it led to better results and we didn't find any paper using a Siamese approach.

Chapter 5

Problem definition

This section explains the choices made to formalize the problem and clearly define in which direction this thesis would go for the problem solving and implementation.

5.1 Change detection vs object detection

For this task, an important point was to decide what the neural network should output. There were two main directions to go :

- Build a network that produces a segmentation mask to detect houses and buildings on a single image. Then repeat the process on a second image of the same location, at a different time. The change detection mask is obtained by taking the differences between the two produced masks. That amounts to build a network specialised in object detection.
- Build a network that take the two images together and processes then at the same time. It directly outputs a mask with the changes found. This means that the network specializes in change detection.

Change detection with neural networks is a very developed genre already. With the rise of algorithms like YOLO and Resnet[11], very fast and very accurate algorithms capable of drawing bounding boxes around all the found objects are accessible to the public.

Naturally, other have been interested in building detection in satellite imagery. For example, the SpaceNet challenge [8] is a challenge run every two years or so that gives out large datasets of different cities and the winning Neural Net is the one capable of accurately identifying the most categories of objects (houses, cars, roads illustrated in figure 5.1, etc.). So, many different Neural Nets have been developed on exactly the types of images we are going to use.

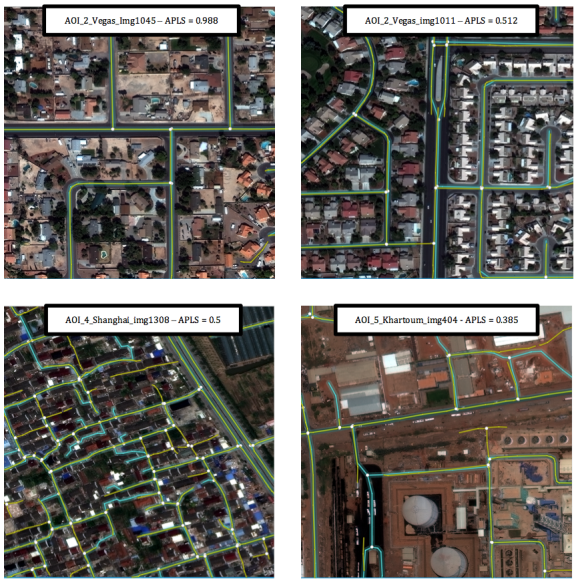

Figure 5.1: Detection of roads in the SpaceNet Challenge in 4 different cities

Change detection in satellite imagery on the other hand is a relatively unexplored branch. The closest match found was a paper describing change detection in satellite imagery with images of a resolution of 20m/pixel[7].

Based on this knowledge, we decided to focus on change detection. It is as much capable as object detection to fulfill our objective while adding the challenge of diving into a relatively unexplored area of the field.

5.2 Dividing the problem

This thesis can be divided in different steps.

The first step aims to build the dataset and enrich it using data augmentation techniques.

The second step aims to build an initial model based on an already existing architecture. This model will then take two images as input and produce a mask with the changes made to the buildings.

The third step will be to customize an architecture tailored to this problem by using the knowledge acquired during the first two steps.

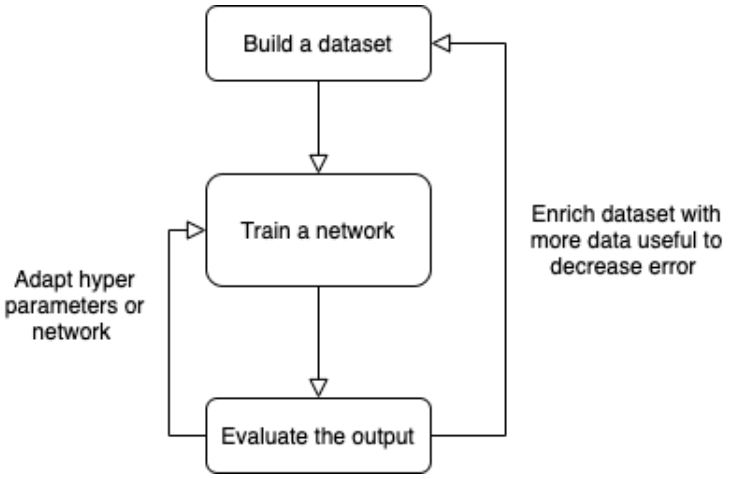

Figure 5.2: Schematization of our workflow

As shown on 5.2, the workflow has three interlinked steps. Any change in the dataset will require a recalculation of the model to compare with other architectures. But on the other hand, from the model results, we will draw conclusion on whether images need to be added to the dataset and what should be on them to correct mistakes from the models.

5.3 Problem formalization

The input are pairs of RGB image of resolution 650×650, thus matrices of size $6 \times 650 \times 650$. The model outputs are mask of the same resolution but with only one channel, containing only 1's where building changes were detected, and 0's otherwise, so $1 \times 650 \times 650$ matrices.

The goal of our model is to be able to, with a given input, produce an output that is the most similar possible with the ground truth mask. To measure the similarity between the produced mask and the ground truth, the Cross Entropy Loss Function is used. The lower the loss, the more our model's outputs are similar

to the ground truth. The problem becomes an optimization problem where the loss is minimized.

5.3.1 Cross Entropy Loss Function

Formally, the Cross Entropy Loss for any number of classes (>= 2) is defined as follows :

$$loss(x, class) = -log(\frac{exp(x[class])}{\sum_j exp(x[j])})$$

Where x is a vector containing the output score for each class, and $class$ is the correct class (ground truth) for the output.

We used a weighted version of the Cross Entropy Loss, defined as following :

$$loss(x, class) = weigth[class](-log(\frac{exp(x[class])}{\sum_j exp(x[j])}))$$

Where weight is a vector of length equal to the number of classes. This can be used to give more importance to a class or another, this is useful when dealing with unbalanced classes like in our case.

In our case, since we only have two classes, and that output values can only be between 0 and 1, it is easy to compute $x[1]$ from $x[0]$ with $x[1] = 1 - x[0]$.

Implementation

We use the Cross Entropy Loss implementation provided by PyTorch in *torch.nn.CrossEntropyLoss*[1] rather than implementing it ourselves. The reasons we choose this loss function is that is simple to use, easy to compute, widely used in Deep Learning in general and in segmentation problems similar to ours, and also for the simple reason that our model works with this function.

5.3.2 Optimization problem formulation

This loss function is applied on all the dataset, pixel-wise comparing output masks to ground truth masks, for each instance of the dataset, taking the arithmetic mean

[1] https://pytorch.org/docs/stable/nn.html#crossentropyloss

of everything. Since this is a minimization problem, the formula becomes :

$$min \sum_{d \in dataset} \sum_{p \in pixels} weigth[class](-log(\frac{exp(x_{d,p}[class])}{\sum_j exp(x_{d,p}[j])}))$$

Where d is an entry of the dataset, and p a pixel in the masks. $x_{d,p}$ is the vector containing the class scores for the pixel p of the mask generated by the entry d of the dataset.

Chapter 6

Dataset

This chapter introduces the process of building a dataset for our problem.

Initially (September 2019), it was planned that AerospaceLab would provide us with a dataset around November. Even before receiving a well built dataset, we needed a minimal one to develop and test our first version of the model. Since it turned out later that AerospaceLab wouldn't be involved with our master's thesis anymore, we had to compose a proper dataset on our own, because the minimal one wasn't large enough to obtain representative results.

6.1 Building our own dataset

For our task, 3 images are required for each single input entry : a first satellite image of a place at a certain time, an image of the same place taken later in time and a mask indicating the areas with changes in buildings. These three images will be respectively referred to as "before", "after" and "mask".

When talking about "changes", we consider changes to be one of the following :

- Entire new building on a previously unoccupied terrain
- Unfinished building on a previously unoccupied terrain
- New part of a building

- Demolished building
- New building where an older building used to be, but was demolished

The following are not considered changes that should be detected by our model :

- Removed or new trees
- Any building that was partially hidden by a tree and is now entirely visible
- Change in the color of crops in a field
- General changes in vegetation
- Change in ground occupation : road, pavement, sand or dirt where grass or dirt used to be
- Changed area near a new building that is not a building itself
- Swimming pools or larger bodies of water
- Change in the roof of a building
- Cars and vehicles
- Buildings too small to be clearly discernible
- Shadows and change in shadows

Figure 6.1: The "before", "after" and "mask" images constituting a single instance of our dataset.

Unfortunately, there was no readily available labeled dataset that would fit this requirement. We had to generate one ourselves. However, object detection

dataset that specialised on buildings were more common, and we tought that having already labeled buildings would help in our task.

The SpaceNet challenges were object detection challenges on satellite images. In 2018, one of these challenges focused on detecting building and roads in various conditions and geographical positions. The datasets used for the challenge are still publicly available. One of them focuses on a city near Paris, with labeled buildings.

The SpaceNet Paris dataset was segmented in 650 by 650 pixels images, at a resolution around 50cm/pixel. We have extracted 20 of them. We then needed to get the corresponding images at a different date. To this intent, the Desktop version of Google Earth Pro[1] was used. Google Earth Pro has a tool shows multiple versions of the same place, taken at different times, coming from different datasets. We tried to find good quality datasets with as much changes as possible.

Then we imported and aligned both pictures on an image editing software. Finally, we looked for changes in between the "before" and "after" images and drew the change mask "by hand".

Some months later, when our model was well developed, we were in the need of a cleaner dataset, with more images and changes on them. We decided to discard the SpaceNet dataset and use Google Earth Pro for both before and after images, because the image quality of the SpaceNet dataset was not optimal. Using Google Earth would also allow to get already aligned images, making the whole process of constituting the dataset much faster. We looked for places rich in changes, exported them with a resolution around 50cm/pixel, and generated the change mask by hand on Photoshop. 32 instances were created this way, meaning that our dataset includes a total of 52 instances. No artificial changes nor any color correction was applied to the new instances, since we want the data to be the most faithful to what you could find yourself.

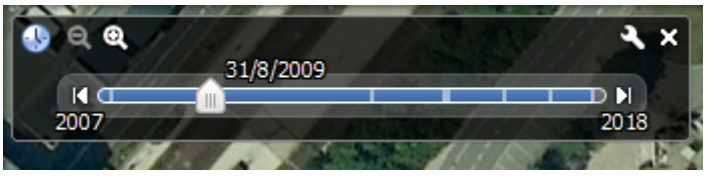

Figure 6.2: A convenient slider to go back in time, in Google Earth Pro

[1]https://www.google.com/earth/versions/#earth-pro

Google Earth Pro uses a collection of datasets, meaning that our images come from datasets provided by Landsat/Copernicus, Maxar Technologies, Aerodata International Surveys, The GeoInformation Group InterAtlas, and other possible sources.

In the DATA files, instances generated from the SpaceNet dataset are called *Paris_1* to *Paris_20*, and the instances generated using Google Earth are called *Earth_1* to *Earth_32*.

6.2 Data augmentation

6.2.1 What is data augmentation ?

Image augmentation is a technique to artificially create new images that are variations of existing images. It is used to expand a dataset. This technique is used to create new and different images from the existing dataset representing probable, realistic-looking variations.

The new images are created by applying a varying amount of transformations to an image. For example rotations, cropping, shearing,.. but also blurring, hue modifications, etc...

6.2.2 Why is it necessary ?

A CNN or any Neural Net for that matter is only as good as its dataset. A larger dataset avoids overfitting and help generalising. A standard dataset for image classification for example can have between 50.000 and 100.000 images.

A CNN more specifically can recognise objects in different sizes, orientations or hues. More images can be added, based on the ones already present in the dataset. This will help create a richer dataset.

In our case data augmentation is used to present cases that are absolutely necessary for generalisation. Because the dataset building our dataset by ourselves, we only use pictures that are clear and not cloudy because we can't see the right contours of new houses otherwise. However, in real time application there is no guarantee that there won't be some clouds or some blurriness on the images. In

the same way, there is no guarantee that the images will always be taken with the same illumination. Therefore, we added blurriness to images and hue changes so the network learns to deal with them.

6.2.3 Data augmentation techniques

Flipping

Images are vertically or horizontally flipped. This modification is simultaneously applied to the "before", "after" and "mask" images obviously to keep the superposition.

Figure 6.3: Original image

Figure 6.4: Vertically flipped image

Figure 6.5: Horizontally flipped image

Shearing

Images are sheared from a 0.15 to 10.0 degrees angle. The result can become unrealistic for greater angles.

Figure 6.6: Original image

Figure 6.7: Sheared image

Gaussian blur

Blur is added on the entire image. This modification is not applied to the mask. This is used to help the network generalise to blurrier pictures due to either picture quality or poor weather conditions.

Figure 6.8: Original image

Figure 6.9: Blurred image

Hue and saturation

Hue and saturation are modified by a random value on each color channel. This helps the network to be capable of working with the different lighting in which the images can be taken.

Figure 6.10: Original image

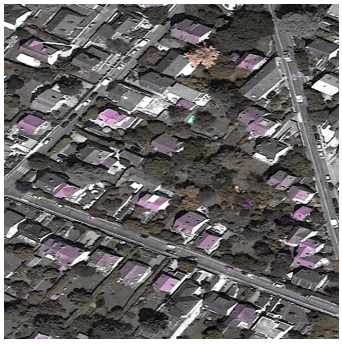

Figure 6.11: Modified image

Rotation

Images are rotated from 0 to 10 degrees. Higher values delete too much area.

Figure 6.12: Original image

Figure 6.13: Rotated image

Gaussian noise

Different noise values can be added, sampled from Gaussian distributions element wise to the two images.

Figure 6.14: Original image

Figure 6.15: Image with noise

Cropping

Random cropping can be applied on the borders of the image.

Figure 6.16: Original image

Figure 6.17: Cropped image

Contrast

Contrast can be changed to either darken or lighten up a picture. This closely resembles the lighting of a sunny or cloudy day.

Figure 6.18: Original image

Figure 6.19: Contrasted image

Scaling

Scaling the image, zooming on a part of it.

Figure 6.20: Original image

Figure 6.21: Scaled image

6.2.4 Augmenting method

Just applying one augmenter is not very efficient and having ten times the same pictures with only a small difference will lead to overfitting. A function was created to apply the changes with certain probabilities. There are thus two variables for changes, the strength and the probability to be applied. Some changes are also too strong to be put together so we constraint them to be applied in a mutually exclusive way.

6.3 Issues with the dataset

By building our own masks and dataset, we are certain that instances are correctly and consistently classified. Although this is a good advantage for our model, there are still issues with our dataset.

The main issue is that our dataset is small. A regular dataset contains thousand of images. This ensures the robustness of the model. Although our architecture, as explained in the later section detailing it, is designed to be robust despite a small-sized dataset, this is even less than what is normally used to train a CNN.

Another issue is that our dataset isn't very diversified. The images all come from a very urbanized area around Paris and in Belgium. This means that the type of building and roofs are mostly the same. The colours, the textures, the structures are similar. For example, the exterior of Paris is quite green and beige but in other countries, earth is more red and the flora is brown. Roof types and housing arrangements are also very different in other countries.

As our model is, it is only for that type of environment. Thankfully it is found in other European countries, such as the Netherlands, etc. that have a similar architectural style and vegetation. We made this choice because we knew we didn't have the time to build an extensive dataset that would cover multiple parts of the world. It would be an interesting idea for improvement.

Chapter 7

Computational power

This short chapter discusses the hardware requirements for training a deep learning model. As it is common knowledge, deep learning requires a lot of computational power to avoid long training periods.

7.1 Deep learning computational requirements

One of the reasons deep learning has reemerged in this last decade is the rise of computational power. With faster computations, the convolutions done in neural networks become suddenly much faster and training time goes from impossible to manageable. Armed with this, the field has yielded incredible results and models in the last years with performances never reached before.

With all these possibilities, new techniques were explored to make training more efficient and faster. One example is AlexNet[13], a CNN divided so that it can run in parallel on two GPU's. This allows to fit a model on two cheaper GPU's rather than a very expensive single one, and to a further extent, split models that would be too big to fit in the memory of the current best GPU's. The best GPU on the market would be the Nvidia Titan RTX, built with Deep Learning and scalability in mind, it offers 576 Tensor Cores, units designed to speed up AI computations, and 24 GB of memory, with the possibility of coupling two of these GPU's together with a high bandwidth link, effectively doubling these numbers. A single Titan RTX costs around 2500 $.

A common misconception is that the reason why deep learning is so slow

on a normal CPU is the operations performed when training a neural network, mainly matrix and vector operations. GPU have the capacity to handle these operations massively in parallel which speeds up the process whereas a CPU will have a much harder time. This is due to the architecture of both. A GPU is made out of many simple core that run processes in parallel and a CPU has many complex cores that run processes sequentially with threads.

While the above is true, the main reason for the GPU performance for training neural networks is memory bandwidth and not necessarily parallelism. CPU's are optimized for latency and GPU's are optimized with regards to bandwidth. In other words, CPU's are great for task that require small amount of memory to be picked up fast and GPU's are better for task that require a lot of data with a higher latency. The best GPU's have a up to 750G/s memory bandwidth whereas the CPU's only go up to 50G/s. Applications requiring high data throughput will benefit from running on a GPU rather than a CPU.

When it comes to the latency issue, GPU's hide it under thread parallelism. Multiple requests are sent in parallel, and while your computer might have to wait before the first one finishes, it will virtually never have to wait for the others.

For all these reasons, training a Neural Network on a GPU can be 15 to 30 times faster than on a CPU, making GPU's pretty vital for Deep Learning.

Last June, AerospaceLab informed us that we would be able to train our model on their GPU's. We haven't heard from then since October. We also considered the computers of the faculty but they didn't have the required GPUs and we couldn't use them four the hours or days that would be necessary to train our model.

Our personal laptops where obviously not fitted with GPU's, or at least GPU's fit for Deep Learning, which made them unsuitable for this work. Even with a good CPU, the smaller size of our RAM would have been a bottleneck. Fortunately, we have a desktop personal computer that proved sufficient for this task.

7.2 System used

All the training was done on Antoine Lambot's personal computer, which was enough to meet our requirements for this task. Its specifications are the following :

- CPU : AMD Ryzen 7 1700 Eight-Core Processor, overclocked at 3.2 GHz ;
- GPU : NVIDIA Geforce GTX 1060 6GB, 1280 CUDA cores, 4004 MHz memory clock, 6144 MB GDDR5 memory size ;
- RAM : Corsair 2 $\times$ 8 GB DDR4, 3200 MHz memory clock.

From a Deep Learning point of view, it could be considered rather low-end, but it was not built with Deep Learning in mind. From a general point of view it is a mid-upper range, having good performances, vastly outperforming any laptop, while remaining rather inexpensive.

Chapter 8

Choice of architecture

This chapter presents the architectures chosen and the result that have been obtained with them. A comparison between each of them and the influence of the hyper-parameters follows.

8.1 UNet

8.1.1 Architecture

The UNet was developed initially for medical image segmentation. As seen on figure 8.1, it has two branches. The first branch is also called the "encoder". The second upcoming branch is called the "decoder" branch. They are connected to each other by skip connections.

Layer types

There are three different layer types used in various combinations in a UNet. They are *convolutional layers*, *upsampling layers* and *MaxPool layers*. The inner workings of those are detailed in the chapter about CNNs. In the UNet and all the architectures derived from it, the convolutional layers have a kernel of size 3, a padding set to output a feature map of the same size of the input and a stride of 1. The MaxPool layer is set to performs a maximum pooling operations such that the

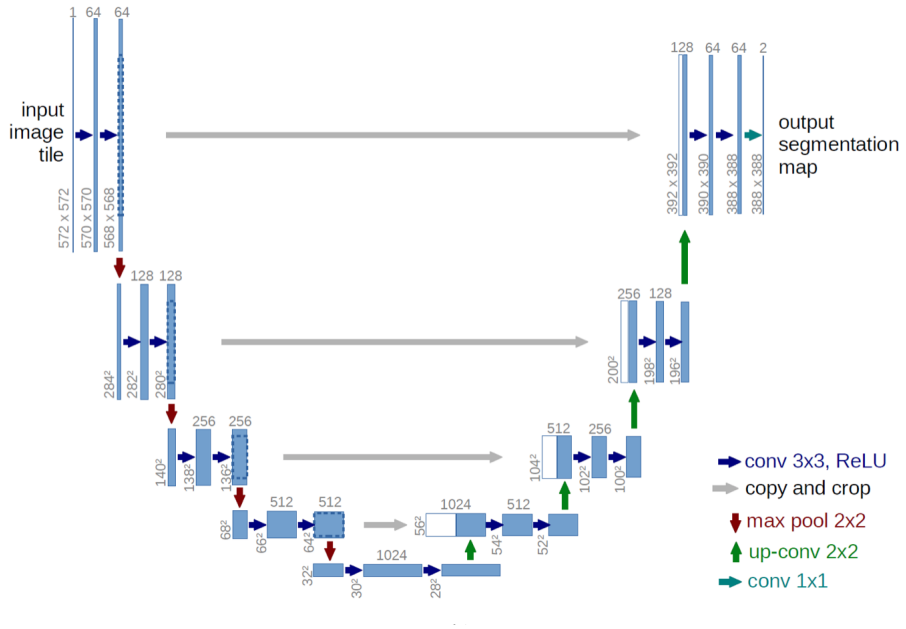

Figure 8.1: The UNet architecture for an input of size 572x572. Each blue box is a feature map with the number channels in gray above. The resolution of the image goes down to 32x32. The different arrow represent different operations explained in the legend.

output is half the size of the input. In the UNet, the upsampling layer has a kernel of size 3 and a stride of 2. This means it doubles the size of the input.

Encoder branch

The encoder branch is used to extract the information about the context or in other words "what" there is in the image. It is made of four "blocks".

Each block follows the pattern :

$$Conv2D \rightarrow BN \rightarrow ReLU \rightarrow Conv2D \rightarrow BN \rightarrow ReLU \rightarrow MaxPool$$

A block takes an input of $n\_channels \times n\_pixels \times n\_pixels$. It applies two times a sequence of a convolution with a kernel of 3 followed by batch normalisation and a ReLU activation function. After the convolutions, the output is of size $(2 \times n\_channels) \times n\_pixels \times n\_pixels$. Then a MaxPool layer is applied reducing the size of the input to $(2 \times n\_channels) \times (n\_pixels \times 0.5) \times$

(n_ pixels × 0.5). If the number of pixels isn't perfectly divisible by 2, a padding of 1 pixel is applied.

This path increases the depth of the image and shrinks its size.

Bottleneck

The bottleneck is the bottom most part. It follows the pattern :

$$Conv2D \rightarrow BN \rightarrow ReLU \rightarrow Cond2D \rightarrow BN \rightarrow ReLU$$

There is no max pooling here because the size of the image has been enough shrunk to extract all the information about "what" there is in the image.

Decoder branch

The second branch, the decoder branch, extracts information about the localization or the "where" part of the analysis.

It follows the pattern:

$$Concatenating \rightarrow Conv2D \rightarrow BN \rightarrow ReLU \rightarrow Cond2D \rightarrow BN$$

$$\rightarrow ReLU \rightarrow TransposedConv$$

First, there is a step of concatenation between the input of the lower layer and the input of the skip connection. Then 2 convolutional layers, each followed by a ReLu and then by an upsampling layer.

A block takes and input of *n_channels* × *n_pixels* × *n_ pixels*. It applies two times a sequence of a convolution with a kernel of 3 followed by batch normalisation and a ReLU activation function. After the convolutions, the output is of size *(0.5* × *n_channels)* × *n_pixels* × *n_ pixels*. Then an up sampling layer is applied making the size of the input to *(0.5* × *n_channels)* × *(n_pixels* × *2)* × *(n_ pixels* × *2)*. Note that with the convolutional layers here reduce the number of channels whereas in the encoder branch they increased the number of channels.

Skip connections

Skip connections connect the output of an encoder block and the input of the decoder branch. These skip connections bring back the information about the localization into the operations because that information was lost when the input was downsampled.

Output layer

As stated before, the output of the UNet is not a label or information about bounding boxes. It is a pixel-wise labeled image in the same resolution than the original image.

The output layer is a convolutional layer with a kernel of 3 and a sigmoid activation function. This constraints the output between 0 and 1. A UNet outputs a feature map for every class. Each feature map is the size of the picture and has for every pixel the probability of belonging to that class.

In our case, we have two feature maps as output because we have two classes, "building built" and "no building built". An inquisitive look to our code would notice that we only use one feature map in the loss computation and the output print. This is because in this case, the classes are mutually exclusive and complementary. If we had three classes a zero probability of belonging to class 1 wouldn't be equivalent to a 100 percent probability of belonging to the class 2 anymore. Although we don't use it, the architecture of the UNet is such that it has to output a feature map per class and it wont output only one feature map.

Why the UNet ?

There are multiple reasons this architecture has been chosen as the basis architecture for our network.

- It is very good for image segmentation thanks to its U-shaped architecture. Although it is a couple years old, it is still used in competitions.
- Its output is in the same resolution as the input. Because we are trying to obtain masks that are as precise as can be, that is important.

- The authors of the UNet claimed in their paper presenting the network that it is designed to build good prediction models even with a smaller dataset but using excessive data augmentation.
- On the contrary of the ResNet or other nets, it doesn't have a very heavy architecture and its training time is shorter.

8.2 Modular UNet

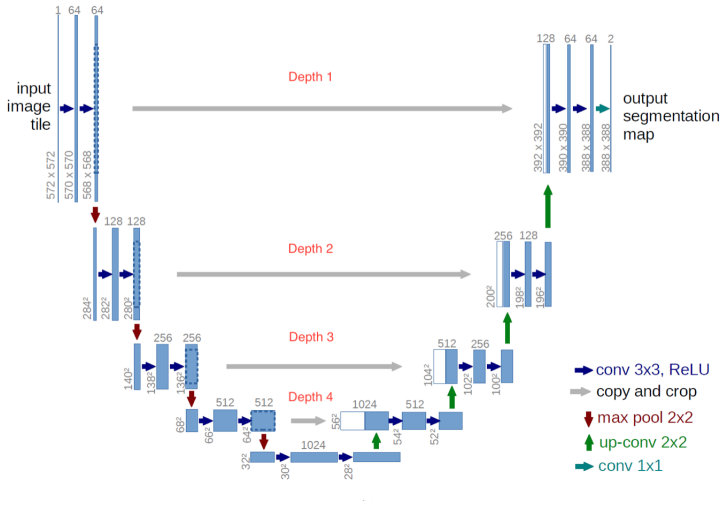

Figure 8.2: A Unet with depth 4. Depths are labeled in red.

What is called the modular UNet is an implementation of the UNet with a variable depth. Layers are added to the bottom. In other words layers with a higher number of output channels and a lower resolution are added. Both follow the descending trends of the initial UNet.

For a layer at depth i :

$$n - input - channels = 2^{(i-1)} \times 64$$

The downscaling layer takes n-input-channels in input and output twice that many channels but with a resolution divided by four. The upscaling layers takes 4 times n-input-channels as input and outputs n-input-channels with a resolution 4 times higher. The reason why it takes 4 times that number as input is that

it receives an input of the size of *resolution* × *(2×n-input-channels)* from the upscaling layer beneath it and *resolution* × *()* from the skip connection. There is thus *resolution* × *(4×n-input-channels)* in total.
The intuition behind this is that the initial model is not capable of fitting our problem well enough. This can come from the fact that there weren't enough parameters.

8.3 UNet++

8.3.1 Architecture

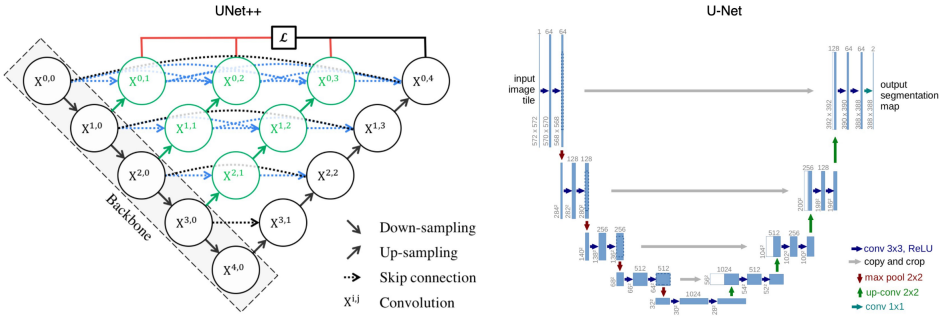

Figure 8.3: UNet++ next to the UNet

The UNet++ [27] published in 2018. It has 3 major differences with the initial UNet.

- It has convolutional layers on the skip pathways. This bridges the semantic gap between the encoding branch and the decoding branch.
- It has dense skip connections on the skip pathways, which improves the gradient flow.
- It has deep supervision on the skip pathways (in red on figure 8.3). However, for reasons explained later, we have chosen not to use this feature.

The UNet++ starts like the UNet with an encoder branch or backbone and follows with a decoder branch. The redesigned skip pathways connect the two branches.

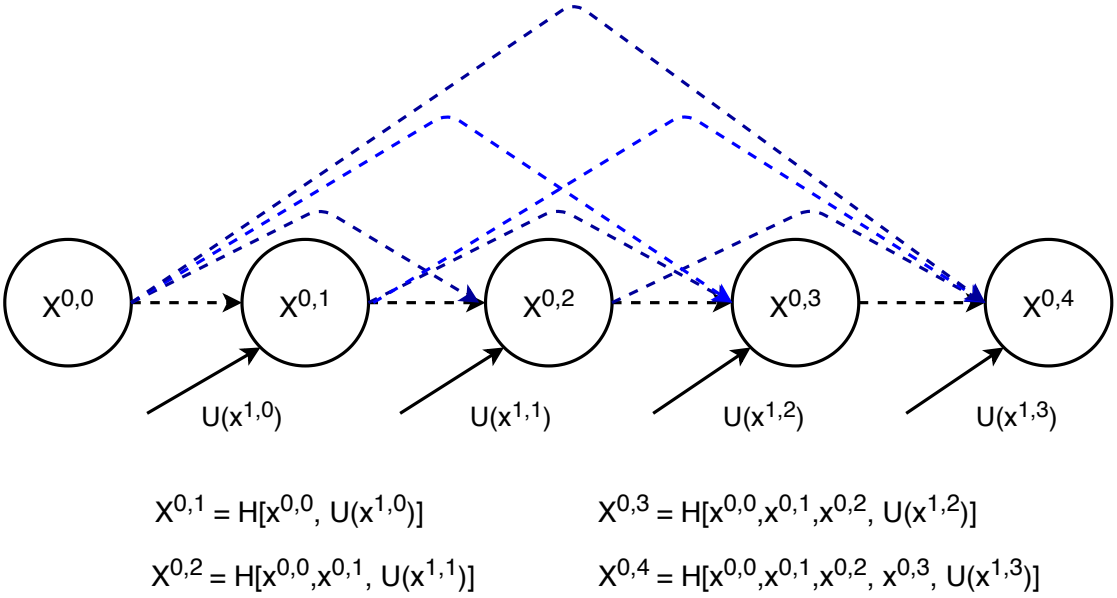

Figure 8.4: Re-designed skip pathways

The figure 8.4 shows this in more details. H() being a convolutional layer and U() an upsampling layer. The brackets denote a concatenation. The idea of dense pathways comes form the DenseNet architecture originally.

We had to modify slightly the architecture because the original models requires more than 10 GB of memory and the GPU we are using has a memory of 6 GB. We had to reduce the size of filters and suppress a series of layers. We deleted the layers $X^{4,0}, X^{3,1}, X^{2,2}$ and $X^{0,4}$. As will be shown further, a lighter UNet++ is sufficient to obtain good results.

We did not use the deep supervision because we are operating with a lower number of parameter and the deep supervision is intricate to implement and would not result in a sufficient time gain to be worth it.

8.4 Training

In order for these models to be comparable, they first needed to be trained. Since the architectures of the model are so different, we had to train each one of them from scratch, as Transfer Learning was not applicable : we couldn't find pre-trained models with a similar architecture that were trained on satellite images.

8.4.1 Training Procedure

Initially, we tried to establish an efficient training procedure applicable on all models, where we decide for example that we train the model X for Y1 epochs with learning rate Z1, and then Y2 epochs with learning rate Z2, etc... but due to the randomness of the model initialisation and of our data augmentation method, it was too hard to find a good method that would work 100% of the time, even for a single model.

Instead, we "hand-trained" the model : after training it for a certain number of epochs, we manually checked the metrics (loss functions, F1-score, ROC curve) but also the output masks to evaluate whether we could continue the training with a lowered learning rate, or whether we should keep the same learning rate for 5 or 10 additional epochs. We stopped the training process when the model converged and reached its top accuracy.

An initial idea we discarded was to fix a small learning rate and train a model until convergence without changing it. The problem is that the difference between the starting learning (0.1) and the fine-tuning learning rate (0.0001) is huge. Training a model with this protocol would have been much slower to the point that it would have taken weeks to train a single model.

Training a model of course still takes time but fortunately, due to the size of the dataset, the number of model parameters, and clever choices of techniques for the model (such as ReLU activation function, Adam optimizer and Batch Normalization), training the model from scratch, where the model parameters are initialised randomly, takes a reasonable amount of time. A model can be fully trained in a couple of hours. Note that, depending on the model and due to the randomness of the process, the number of epochs to reach convergence is not always the same.

8.4.2 Testing different thresholds

Our change detection problem is a classification problem where the classes are "change" or "no change", but our Neural Network outputs a "continuous" value between 0 and 1 for each pixel. To solve this problem, we simply use a threshold : a value between 0 and 1 such that all pixels with smaller values will be considered as "no change", and with greater values will be considered as "change". But there is still the question of the optimal threshold...

The optimal threshold is defined as the threshold value that will maximise the F1-score of an output mask. If the threshold is too low, too much areas will be detected as changes, and the Precision will be poorer, on the opposite a too high threshold will be bad for the Recall.

The optimal value can change from an epoch to the next, so it cannot just be fixed once and for all. To determine the optimal threshold at a given epoch, the F1-Score of the training set is computed with different threshold values between 0 and 1, and the one giving the best results is kept. When computing metrics on the test set, the optimal threshold found on the train set can be used. It is also possible to find the best threshold value for the test set, in the same way that we did for the train set, and compare them : we observed that they are very close to each other most of the time, meaning that the best threshold found on the train set can safely be used on the test set.

8.4.3 Choice of hyperparameters and other variables

To compare the models, we trained them using the "Earth" dataset, using the images 1 to 22 for as training set, and images 23 to 32 as test set. We used data augmentation on the training set only, to generate two new images from each reference image, using random vertical, horizontal flips, and hue/saturation changes. The data augmentation is redone at each epoch, to ensure greater variety in the training data. Unless stated otherwise, no data augmentation is performed on the test data.

For all models, the architectures described in the above section were used, without them changing during training. For the loss function, the CrossEntropyLoss from torch.nn was used, with weights set to $[1, 8]$. Weights are used to specify the importance of each class when computing the loss. [1, 8] means that errors on class 0 (background, no changes) are 8 times less important than errors on class 1 (changes). The need for these weights comes from the class imbalance in the training set. On average, the images have much more background than changes in them. If a smaller ratio is used, like 1/1, the model will poorly detect change areas in images. If the ratio is too high, like 10/1, background detection will start to worsen, bounds between the two classes will become less precise and false positive will be detected in the form of small erratic spots. By testing different ratios, we have determined that the range between 5/1 to 8/1 gives the best results.

8.4.4 Time repartition in an epoch

The following graph shows the time needed by each part of our program during an epoch.

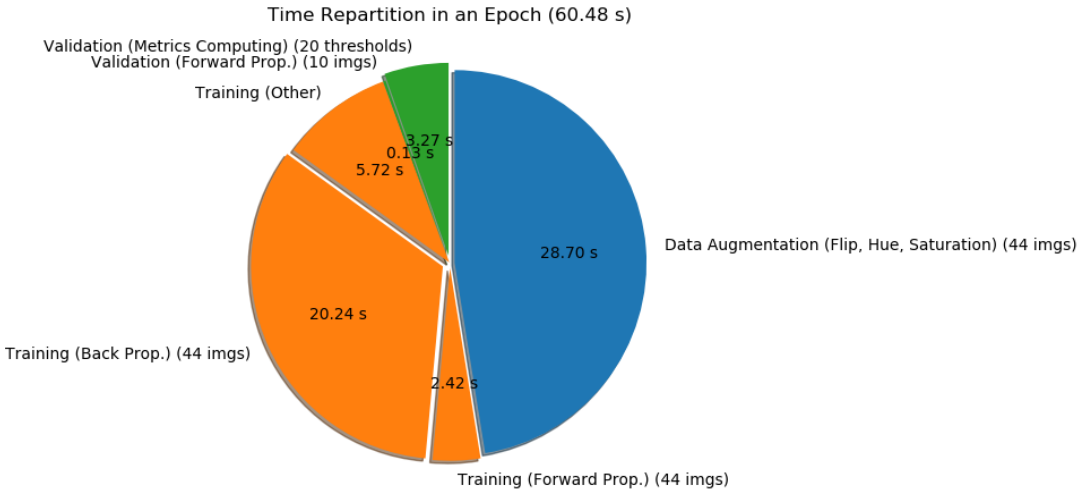

Figure 8.5: Time repartition in an Epoch

We can see that the most time-demanding tasks are the Data Augmentation, to generate 44 images, and the Training itself (Forward, Backward propagation, and various manipulations) on these 44 images. Validation is much faster and linearly depends on the number of images in the test set, and the number of thresholds we compute the metrics on. Another observation we can make is that Forward Propagation on one (pair of) image(s), is slower during Training (55ms) than Validation (13ms). The reason is that during training forward propagation, gradients need to be computed and stored for each layer, but not during validation.

Regarding Data Augmentation, we can see that performing it at each epoch has a huge impact on training time, practically doubling it. But one thing to keep in mind is that the Training can be done on the GPU, while Data Augmentation can be done on the CPU of your system. The current state of our implementation is not optimal regarding multithreading and parallelisation, since everything is single threaded on the CPU side, and that there is no parallelisation between the CPU and the GPU : during forward or back propagation on the GPU, our CPU is idle, and vice-versa.

Thus, if we properly used CPU/GPU parallelisation, we would cut the epoch time in half, the GPU would be under almost constant usage, without having

to wait too much for the Data Augmentation to be finished by the CPU. Also if all CPU cores available were exploited instead of using single threading, the Data Augmentation could potentially be made 8 times faster (on our system).

8.5 Model comparison

We are going to compare the 4 models with each other on different aspects : differences in training, performance on some metrics, and quality of the output masks.

8.5.1 Summary of architecture differences

We are going first to recapitulate what are the differences in architectures between the 4 compared models.

There are two main approaches, the Basic UNet one, and the UNet++ one, with more interconnection and layers added on the skip connections pathways. The Modular UNet versions follows the Basic UNet architectures, but have a shallower "depths" of 3 and 2. The Basic UNet is equivalent to a Modular UNet of depth 4.

8.5.2 Memory usage and number of parameters

All these models were designed to fully use the 6GB of GPU Memory available on our system, so their total memory usage are just under that value. Memory usage is tied to the architecture design. The input size has an inversely proportional impact on the number of parameters available, but other factors also weigh in. During training, the model has to keep in memory the output of each internal layer, and the associated gradients. If you have more layers or more interconnections between the layers, these internal representations will require more memory, on top of the memory used to store the trainable parameters themselves.

The resulting difference in number of parameters can be quite drastic. The way we can compare these number is using *torchsummary*, a tool that provides a detailed estimation of our model memory usage, or simply looking at the size of our weights save files.

The results are the following :

- BasicUNet : $\sim$ 35 millions parameters, for a save file of 131 MB ;
- ModularUNet with a depth of 3 : $\sim$ 8.7 millions parameters, for a save file of 32.7 MB ;
- ModularUNet with a depth of 2 : $\sim$ 2.1 millions parameters, for a save file of 7.93 MB ;
- LightUNet++ : $\sim$ 3.5 millions parameters, for a save file of 13.0 MB.

The reason why "deeper" UNets can have more parameters (more convolutional filters) in total is that, at greater depths, the resolution of the internal representation decreases, meaning you can have more filters for the same memory usage.

8.5.3 Learning curves

The following graphs show the evolution of different metrics during the training of each model (the training procedure was described in the previous section). The train and test loss are computed using the CrossEntropyLoss function from torch.nn. The AUC is Area Under Curve of the Receiver Operating Characteristic, the ROC itself being plotted using 20 thresholds between 0 and 1.

In addition, for each threshold, the Recall, Precision and F1 Score are computed. The confusion matrix is computed pixel-wise on the ground truth masks and generated masks. The displayed F1-Score is the highest F1-Score computed over all thresholds. The reasons we compute the F1 Score in addition to the ROC AUC, is that the ROC AUC is comparable to a measure of the Precision of the model, but not its Recall, and we considered that this metric could not be ignored, because the AUC alone may not make the difference between a good model and a model that misses a lot of changes that should have been detected.

Both the AUC and F1-Score have values between 0 and 1, the higher the better. For readability reasons, we chose to plot 1-AUC and 1-F1, so that all four curves decrease when improving.

The ROC AUC and F1-Score displayed are test set metrics only, they are not computed on the train set.

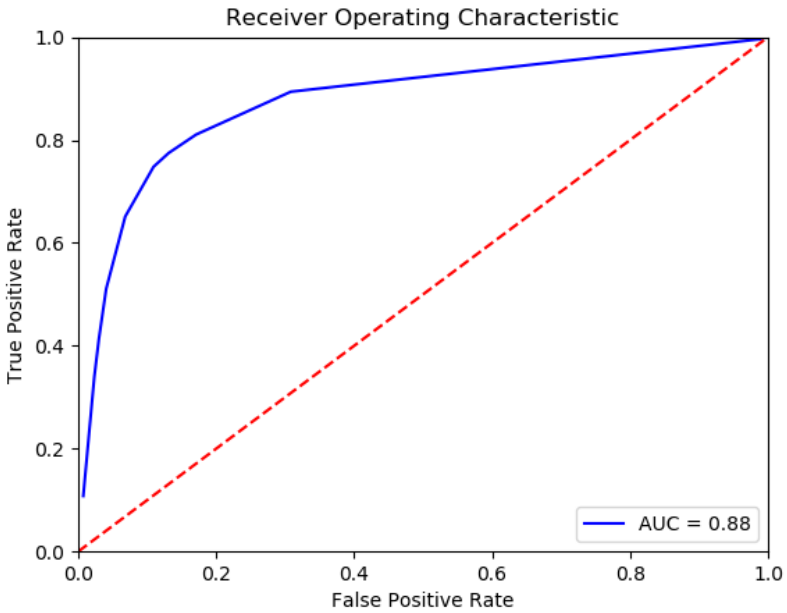

Figure 8.6: Typical ROC curve showing how the True Positive Rate evolves with the False Positive Rate for different threshold values

The observations we can make from these curves are multiple.

The ROC AUC metric is not very representative of a "good" model, as it always stays near 90%. A model with high Precision (From the detected changes, what proportion is really a change in the ground truth ?) but with poor Recall (From all ground truth changes, what proportion are we able to detect ?) will still have a high AUC. We can take the Modular UNet with depth 2 as example. It has the highest AUC, but a poor F1-Score.

The F1-Score seems to be the most interesting metric to summarise models performances. The reasons we chose it in the first place is that it is supposed to provide meaningful insights on the model performance even with a high class imbalanced data, by taking the Precision and Recall into account. It is also more interpretable than the Cross Entropy Loss.

From all models following a Basic/Modular UNet architecture, performance increases in deeper architectures. The approach of increasing the number of layers seems to be better than increasing the number of filters per layer. Also, shallower architectures will train faster, but yield poorer results.

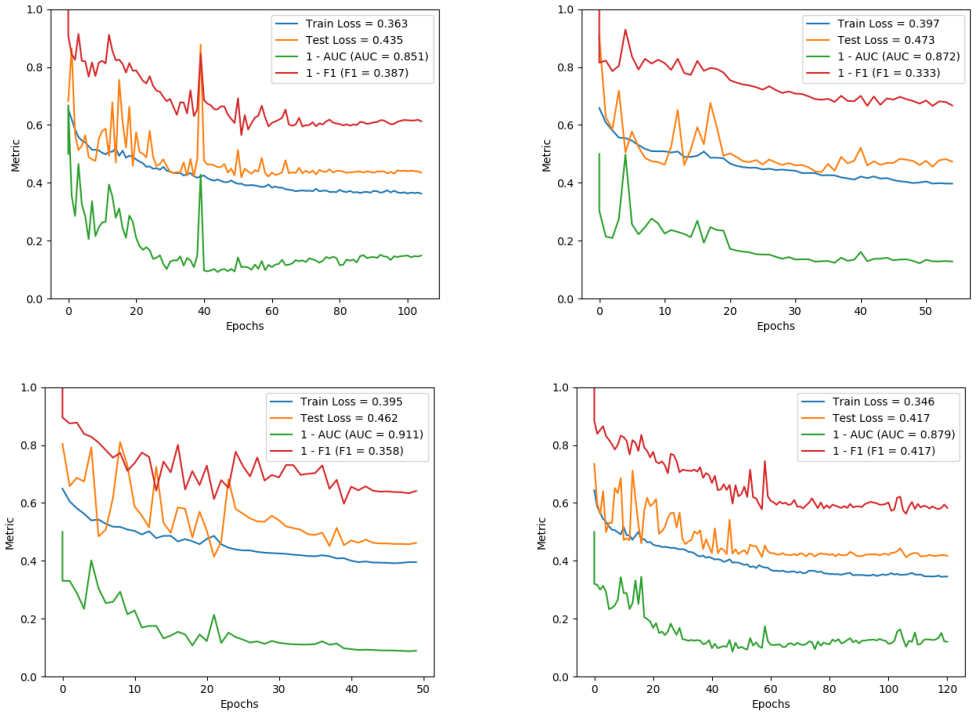

Figure 8.7: Training curves
Top Left: Basic UNet
Top Right: Modular UNet (Depth=3)
Bottom Left: Modular UNet (Depth=2)
Bottom Right: Light UNet++

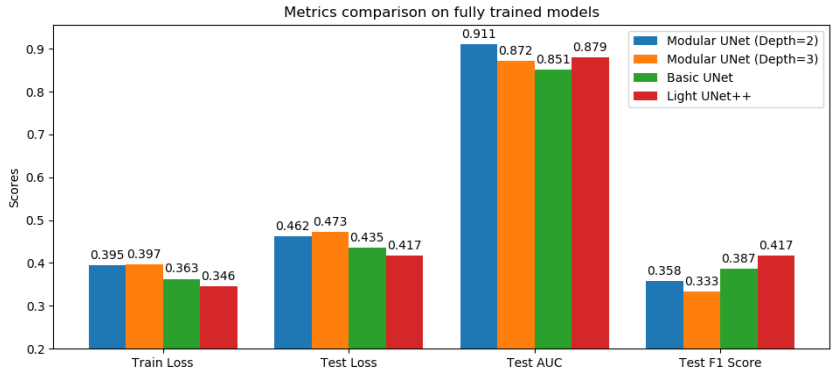

Figure 8.8: Metrics comparison on fully trained models

The best architecture within the ones we tested is the Light UNet++, largely surpassing the Basic UNet. The main advantage of the Light UNet++ compared to the Basic UNet is supposed to be a better prediction on the edges of detected zones, and it is what we see here too. The reader should remember that the Basic UNet has around 10 times more parameters (10 times more convolutional filters, ...) than the Light UNet++, it is the reason we call it "Light". But despite this significant difference, the UNet++ architecture shows its superiority.

The distance between the Train and Test Loss curves is relatively small, and does not increase during later stages of training. This indicates that our model does not suffer from much overfitting. Of course, even with data augmentation, our dataset is rather small, and the train set might not be able to cover some cases of changes in the test set, so our Test Loss is still greater than our Train Loss.

8.5.4 Output masks

To really appreciate what the models are capable of, we can directly look at the produced masks.

The example above is taken from the training set. The first two images on the top are the input of the model ("Before" and "After" images). Next to them is the "Ground Truth" image. The first row of blue images are the corresponding outputs for each model. The output masks are supposed to replicate the ground truth mask, the white areas indicating changes detected, and the blue or black areas no changes. The second row of blue images are enlargements of the same part of the image above, to make it easier to see differences at a smaller scale. The reader should not hesitate to zoom even further to appreciate subtler changes.

First, with a general glance, we can see that all models manage to produce an output relatively similar to the ground truth. There are some areas where the models predict changes where there shouldn't be any (false positives), where roads or concrete paveways have been built. Starting from an empty field, it is hard for our models to tell if what was built is a house, a road, or a ground-level concrete slab, even on training set images.

If we now take a closer look at the boundaries of the areas, we can understand why the Light UNet++ architecture might have better performances than the other models. From up close, we can see that the boundaries on the Light UNet++ model are smoother, more consistent than on the Basic UNet model, where numerous small irregularities are present on the borders. When looking on

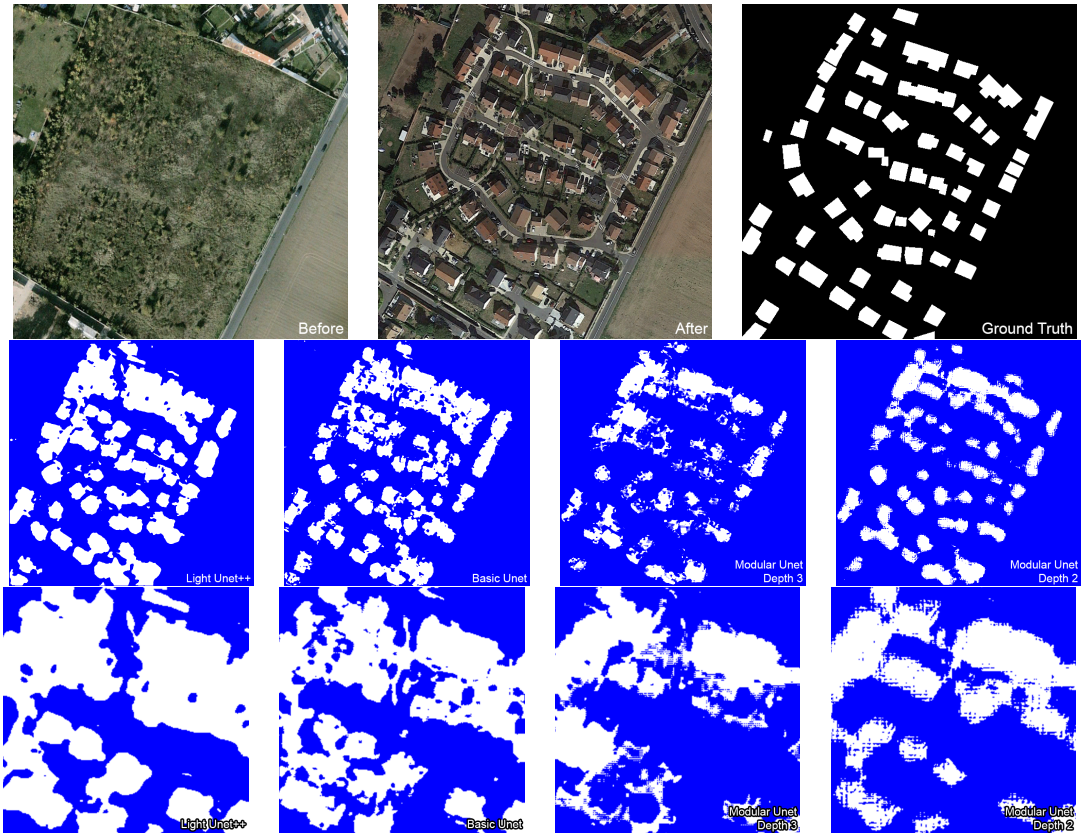

Figure 8.9: Example of input data and the corresponding outputs masks on fully trained models

the mask output by the Modular UNet of Depth 3, we can see that in addition to these irregularities, there are some kind of visual artefacts, horizontal bars on some areas. This kind of artifact is even more visible and larger with the Modular UNet of Depth 2.

If we recall how a UNet architecture is supposed to work, we can note that boundaries are supposed to be handled in the "Upper" layers, while deciding if larger areas are changes is handled on "Deeper" layers. The Modular UNet of Depth 2 has less "deep" layers, but compensates by having more filters on the "upper" layers. This means that more filters are dedicated to borders, and thus the boundaries between areas should be better on shallower models : more defined and clean, without visual artefacts. However, we can see that this is not the case. At the same time, we can observe that boundaries are the cleanest on the Light UNet++ model, which has fewer filters than all the other models.

The explanation for the artefacts on the shallower models could be caused

by a high number of parameters, inducing some kind of overfitting. Another explanation could be that models also need to be able to "understand" larger structures to draw clear boundaries, a task shallower models struggle to do.

This is another good illustration of why choosing a good architecture is more important than just increasing the number of parameters.

8.6 Generated masks

In this section, we will look at the masks generated by the Light UNet++ model, that we consider the best results our model can produce. This version of Light UNet++ has been trained using all the data augmentation methods described in section 6.3. All the masks presented contains values between 0 and 1, but we have added colors to make them more readable. A black background indicates a ground truth mask, a blue indicates a mask generated by an instance of the training set, and red an instance from the test set.

Masks are presented in pairs of columns. On the left column (blue or red) the generated output is shown, and the right columns (black) shows the ground truth.

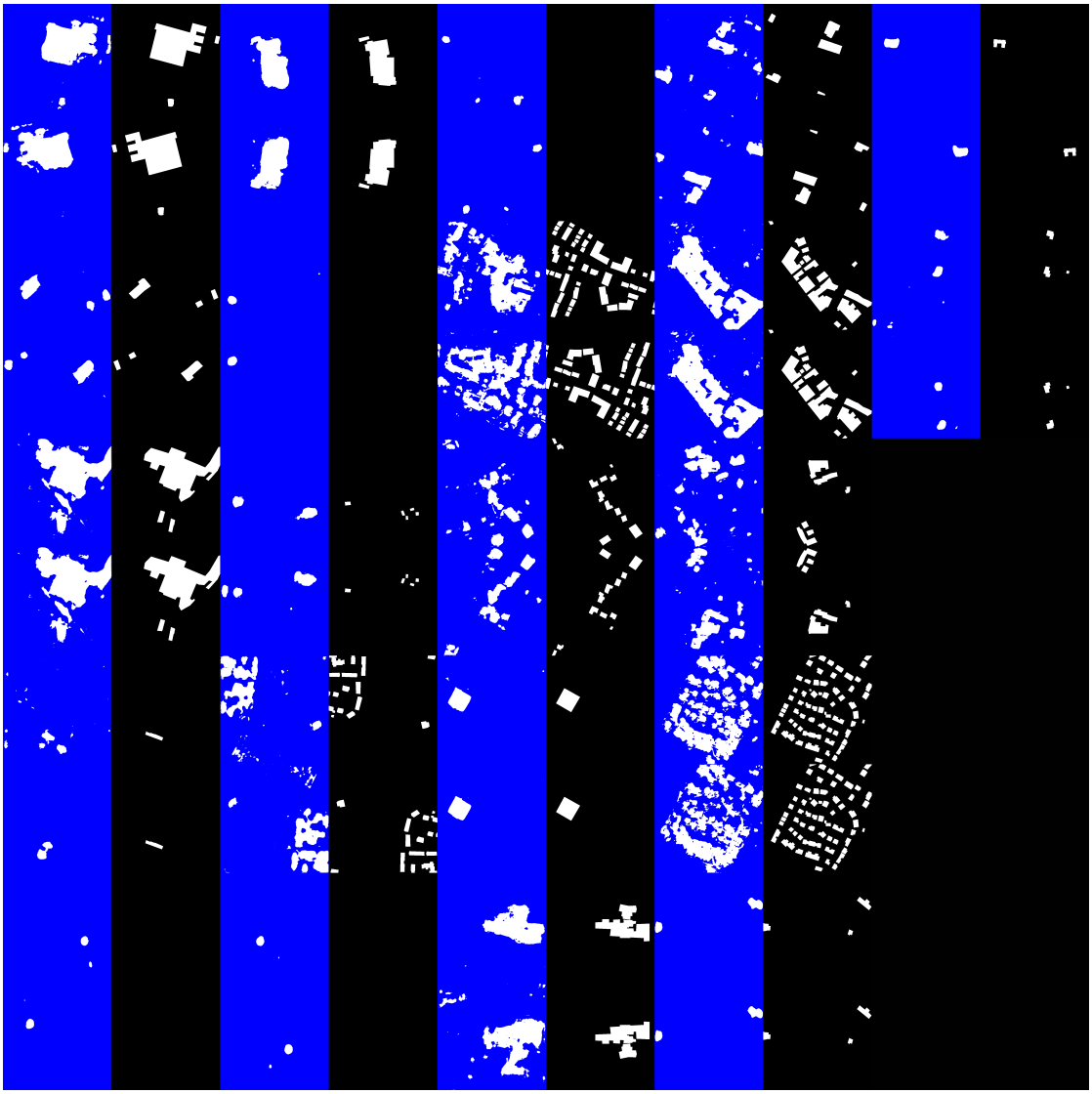

Figure 8.10: Masks generated on the train set, do not hesitate to zoom

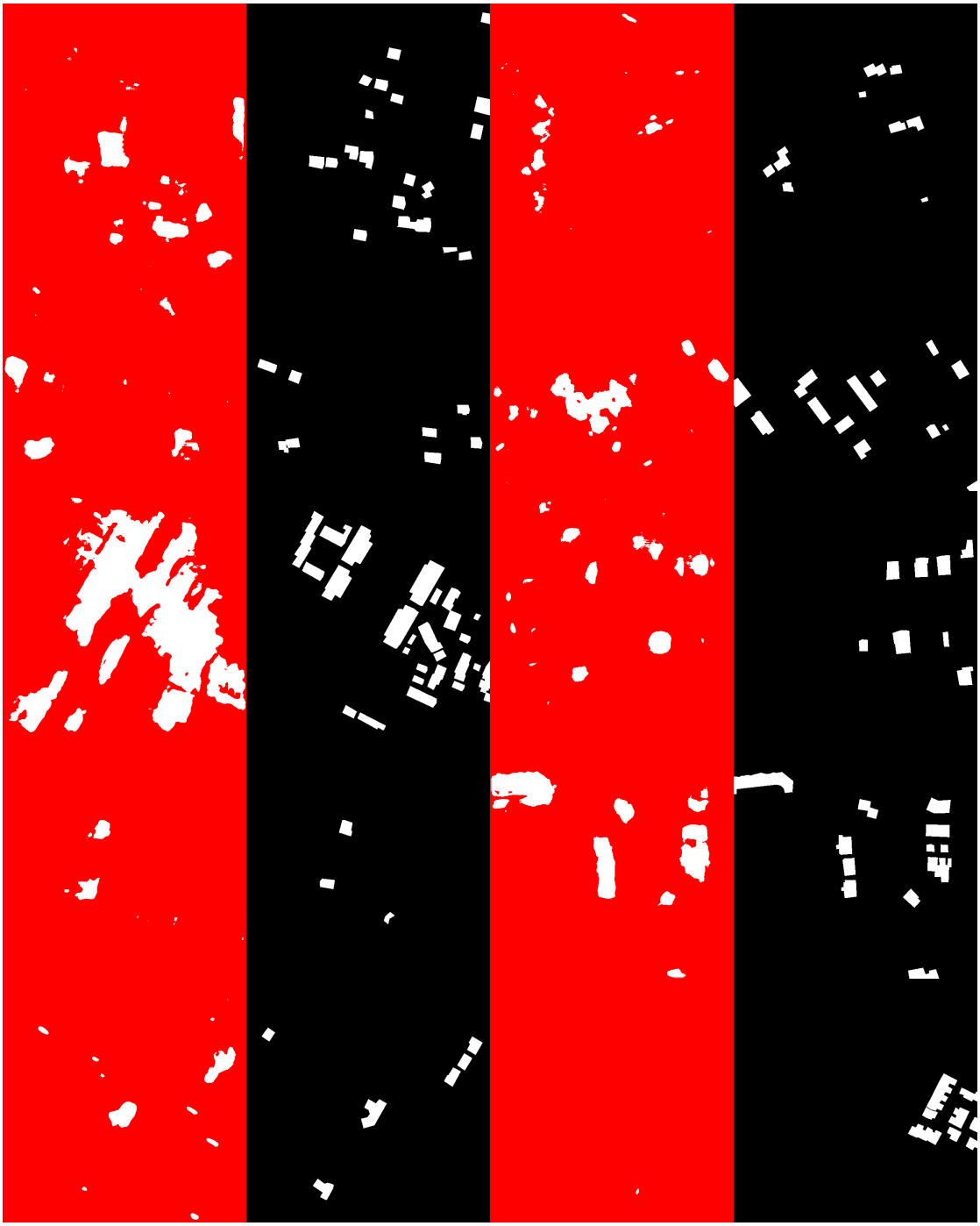

Figure 8.11: Masks generated on the test set, do not hesitate to zoom

We are now going to take a closer look at these results and try to understand them better, finding what the model can and cannot learn.

8.6.1 False positives

False positives are defined as areas detected as changes by the model, but that are not valid changes on the ground truth. There is two examples containing false positives below.

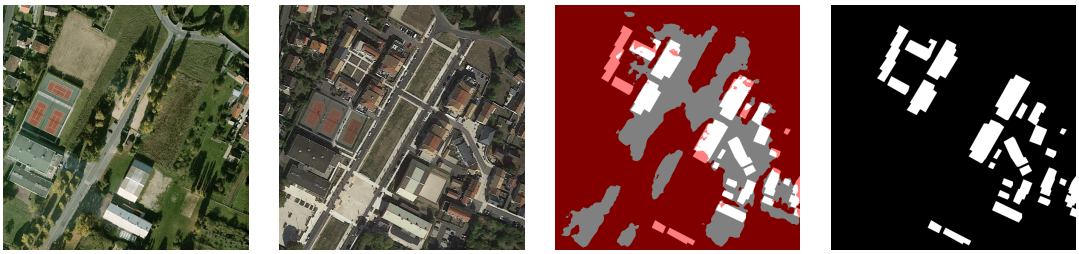

Figure 8.12: Before, After, Generated Mask and Ground Truth Mask

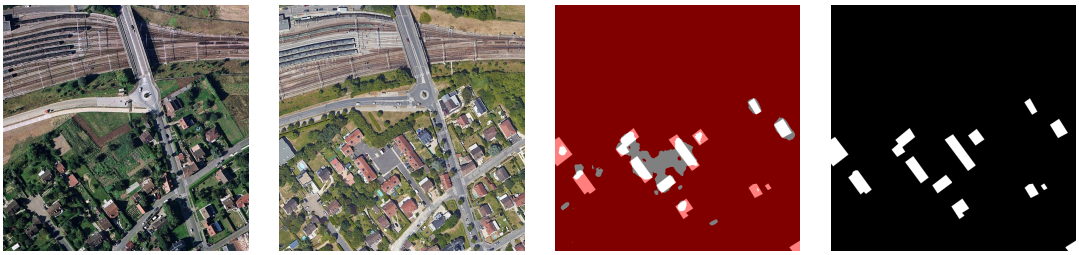

Figure 8.13: Before, After, Generated Mask and Ground Truth Mask

We observed that false positives were generally due to changes that were actual changes, but not changes in the buildings. For example, roads, concrete or pavements on the ground, where there was previously trees or an empty field. These changes should not be detected, but in some cases it is hard for the model to tell the difference between them and new buildings. We had also one case of "false positive" where in reality our model successfully detected a change that we missed when constituting the dataset.

8.6.2 False negatives

False negatives are defined as areas where the model does not detect changes that should have been detected.

Figure 8.14: Before, After, Generated Mask and Ground Truth Mask

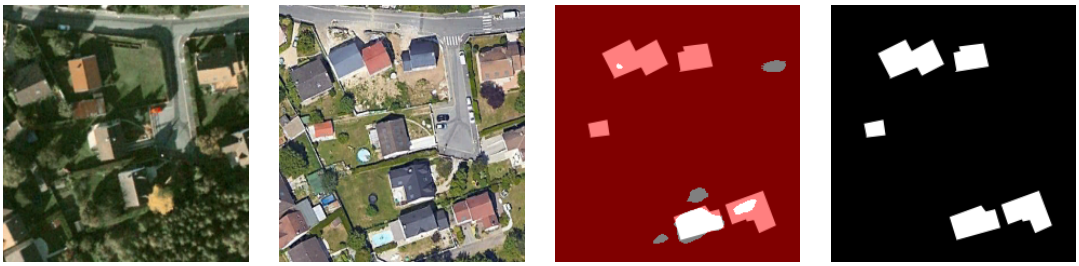

Figure 8.15: Before, After, Generated Mask and Ground Truth Mask

False negatives are harder to explain than false positives. Sometimes a tree or hedge casting an unlucky shadow will look similar enough to house in the eyes of the model, so that he does not detect it. Sometimes, like in the first example, it is because a building was demolished and replaced by another that will be vaguely similar in aspect. Small and individual houses can also be harder to detect than larger groups of houses.

8.6.3 General observations

In terms of performances on the masks, both the detection of changes and the accuracy of the boundaries are heavily influenced by the quality of the images. Some older images can be blurry with a strong orange tint. Images taken when the sun was low show huge shadows, and when the sun was high have none. Images with long shadows, or pairs of images with very different shadows in terms of length and angle are harder to understand for our model. The alignment between the "before" and "after" images is also not always perfect, sometimes there can be a 3 or 4 pixels offset (2 meters in real life), and while this does not seem like a lot, the difference in mask quality is noticeable. Also, due to the size of the dataset, some cases of the test set are not covered by any instance of the train set, in terms of shapes and colors, even with Data Augmentation.

Even still, we think that our current results with a small dataset (22 pairs of original images were used for training the model in this section) are promising, and that we could get far more interesting results with a larger and more diverse dataset. The LightUNet++ architecture seems to be fit for the job, even with a low number of parameters available.

8.7 Impact of Data Augmentation

A quality of the general UNet architecture is that it works well with heavily augmented images. In this section we explore how data augmentation improves our model. The architecture we use in this section is the Light UNet++.

8.7.1 Our dataset

The dataset of building changes we composed is relatively small (52 pairs of images in total) and only covers restricted geographic areas. Most of our images come from the outskirts of Louvain-La-Neuve in Belgium and Paris in France, both semi-rural areas. The quality of the picture and the circumstances in which they were taken differ from one image to the other. For the "after" pictures, finding good quality images was easier, since there are many recent datasets to pick from, but for the "before" pictures, it was much harder to find good quality satellite imagery from 10 or 15 years ago.

We have several examples where the position of the sun is not the same between the "before" and "after" images, meaning that buildings will cast shadows with different directions and lengths. The position of the satellite also plays a part, since it is rarely perfectly vertical. Depending on the angle, you can see a different side of the house. Colors and tints also vary. Different satellites might produce images with different tints, and colors might change depending on the illumination.

All these changes are not related to the construction of buildings and should not be detected, but the model needs to learn that they should not be considered. Since our dataset is small and lacks diversity, our model might have a hard time performing on slightly different images (tint, satellite angle, sun angle,...). Data Augmentation is there to introduce some artificial diversity.

8.7.2 Results and comparison

We will compare 4 cases for the same model. First, we train the model from scratch without any Train Set Data Augmentation, then test it without and with Test Set Data Augmentation. Then, we train again the model from scratch, but with Train Set Data Augmentation this time, and we test it with and without Test Set Data Augmentation. The results are shown in the following figures.

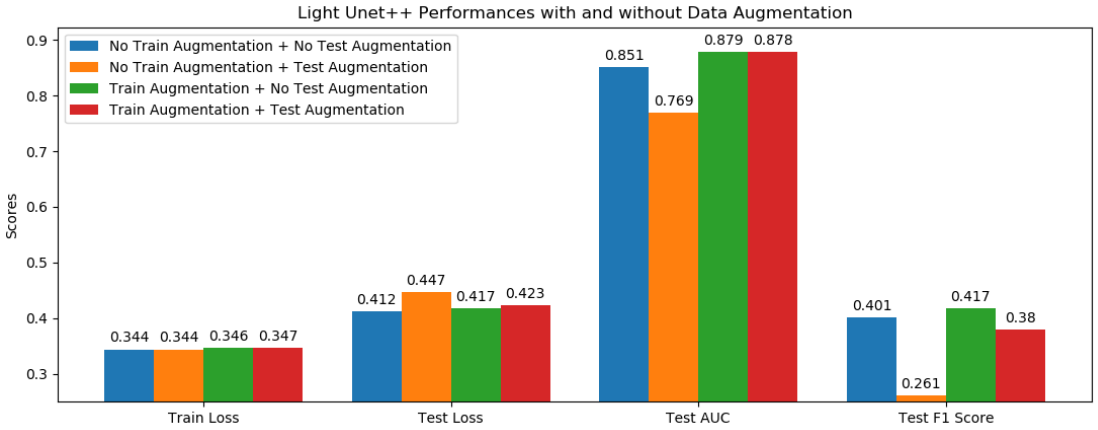

Figure 8.16: Impact of Data Augmentation

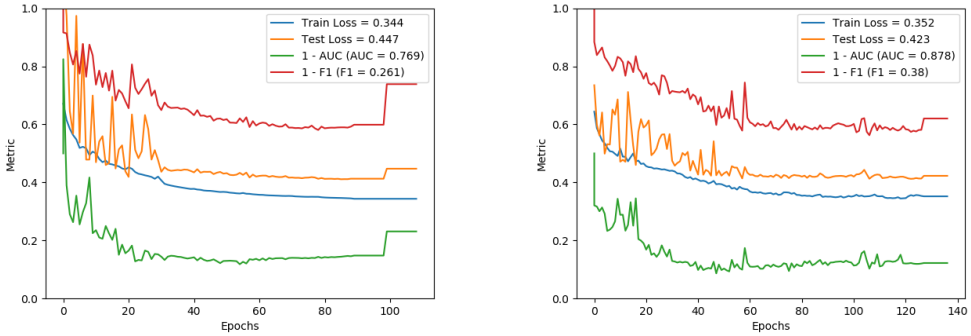

Figure 8.17: Light UNet++ trained without (left) and with (right) data augmentation

If we look at the learning curves, we can see that we first completely train the model in both cases. Then we apply the same, non-random, Data Augmentation (vertical and horizontal flips, small hue and saturation shifts) on the Test set, to see how the model reacts to these changes. We let the model for 10 epochs with a very small learning rate to create a flat line in the end, purely for readability reasons. We can directly see the decrease in Test Loss, ROC AUC and F1-Score for the model trained without Data Augmentation, while the change is less remarkable for the other model. The precise values are reported in Figure 8.16.

In terms of performances on the un-Augmented Test Set, both models are pretty similar. A slightly higher F1-Score for the model trained with Data Augmentation might indicate that it allowed the model to generalize better on some cases not originally present in the Train Set, but present in the Test Set.

We believe that the reason why there is a slight drop in performances for the model trained with Augmentation when subjected to the Augmented Test Set, is caused by our implementation of Data Augmentation. When this experiment was run, the Data Augmentation implementation was such that the random changes had smaller probabilities, meaning that the pictures were "less augmented", closer to the original Train Set.

Another observation we can make is that the training curves are smoother for the un-Augmented model than the other, and that it is trained in less epochs. It can be explained by the fact that Data Augmentation makes the Train Set harder to learn by artificially increasing its size. And since the augmented Dataset is regenerated at each epoch, it is different from an epoch to the other, adding some jittering in the metrics.

8.8 ReLU vs Sigmoid activation function in the output layer

This section will detail one choice of implementation we had to make : the activation function used in the final layer of the network.

The output of our network consists of two "masks", but since we only have two classes, we can only use the first mask as result for the class "0", the background, and "invert" it to get the result for class "1", the detected changes.

First, we used the ReLU activation function in the final layer of the network, meaning that the output "masks" would have values greater than 0, but without bound on the positive values. Since we use the "background" mask, higher values would mean that no change is detected, and values near 0 would indicate detected changes. While most values stay in a reasonable range, smaller than 2, some of them can jump up to several hundreds. This was not really a problem when displaying our first results, values greater than 1 were set to 1 and the masks turned out fine, but to avoid this artificial change, our promoter Pierre Schaus advised us to use the Sigmoid activation function instead.

The advantage of the Sigmoid activation function is that the mask values output by the final layer would fall between 0 and 1, using the network architecture itself instead of an external manipulation. Another advantage over the ReLU activation function is that the network can get "stuck" when all the outputs of a layer are zero. In this case, backpropagation will not be able to train the model

anymore, and the weights of the model would have to be reverted to a previous version. In practice, this never happens because having all the outputs of a layer to be equal to 0 is an extremely rare case. However, it happened to us, when we started the training of the Light UNet++, because this model has less parameters than other UNet models. This situation only happens if the learning rate is quite high, meaning that it can be prevented and can only occur during the very first epochs of the training. If we switch to the Sigmoid activation function, this risk disappears.

However, the ReLU activation also has its advantages. It prevents the problem of vanishing and exploding gradients, allowing faster training. It is also easier to compute, without impacting performances. It is widely recognized as the "best" activation function, and that's why we use it in all the layers of our networks, apart from the final, output one, for the reasons stated above.

To summarize, in all our models, we use the ReLU activation function in all layers except for the output one, where the Sigmoid activation function is used instead.

8.9 Conclusion

We were able to confirm that our implementation choices, based on theoretical arguments, works effectively in practice. We observed the advantage of a thoughtful architecture with fewer parameters confronted to a simpler architecture with significantly more parameters. The Light UNet++ architecture seems to be the best for this change detection in satellite images problem.

Chapter 9

Code

This section presents the code in more details. Starting by the overall structure and then focusing on each file. The implementation is available on the GitHub repository "LamboiseNet"[2].

9.1 Overall structure

The structure of the project is shown in figure 9.1. The first three directories do not contain any code. In the Loss directory are files containing the losses and other metrics from previous runs. This allows us to interrupt learning, look at the loss and tune the hyper parameters accordingly. This also allows to study a long evolution without having to train hours without interruption.

Similarly, the Weights directory is used to save the weights after training the network. Those records allow to reload a pre-trained model to continue training it later or compare it to another. Each saved file contains the learned weights for a single model after training, and can be quite heavy depending on the model (up to 130 MB for the Basic UNet).

The DATA directory contains the dataset. For each instance, there is a directory containing the first picture, the second picture and the mask. The directories are named from *Paris_1* to *Paris_20* and *Earth_1* to *Earth_32*. The three images in these folders are named "before.png", "after.png" and "mask.png".

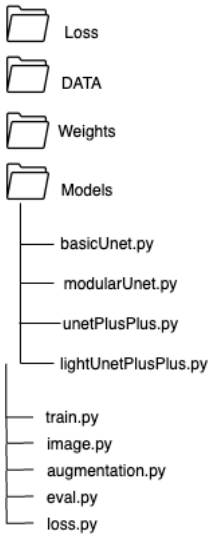

Figure 9.1: Structure of the project

9.2 Models

The models directory contains the files with the 4 models implementations, each models in a different file. For example the basicUnet.py contains the code that initializes a basic UNet model. Each file contains the class defining the model. The basicUnet.py files also has the classes of the building blocks of different layers. For example the Doubleconvolutional layer is a layer found into all different UNets, so it is defined once and called in the other files.

Figure 9.2 shows an example of a definition of a model class, here the basic UNet. It takes the number of input channels and output classes in argument. Firstly, all the different layers with the input and output channels are defined. Secondly, the pipeline of the data through all the layers is fixed.

9.3 train.py

This is the main file of the project, where the main workflow of the project is defined, used to launch the full training process. It is wherethe parameters of the function *train_model*, such as the number epochs, the learning rate, how the dataset are loaded and augmented, if a previous model is reloaded or the training starts from scratch, and if the model has to be saved at the end of the training are

```
class BasicUnet(nn.Module):
    def __init__(self, n_channels, n_classes):
        super(BasicUnet, self).__init__()

        self.name = "UNet"
        self.n_classes = n_classes
        self.n_channels = n_channels

        self.input_layer = DoubleConvolutionLayer(n_channels, 64)
        self.downscaling_layer1 = Downscaling_layer(64, 128)
        self.downscaling_layer2 = Downscaling_layer(128, 256)
        self.downscaling_layer3 = Downscaling_layer(256, 512)
        self.bottleneck = Bottleneck(512, 1024, 512)
        self.upscaling_layer1 = ExpandingLayer(1024, 512, 256)
        self.upscaling_layer2 = ExpandingLayer(512, 256, 128)
        self.upscaling_layer3 = ExpandingLayer(256, 128, 64)
        self.output_layer = FinalLayer(128, 64, n_classes)



    def forward(self, x):
        down1 = self.input_layer(x)
        down2 = self.downscaling_layer1(down1)
        down3 = self.downscaling_layer2(down2)
        down4 = self.downscaling_layer3(down3)
        bottleneck = self.bottleneck(down4)
```

Figure 9.2: Exemple of the definition of a model. Here the basic UNet specified.

9.4 image.py

This file contains a lot of helper functions related to image processing, and addresses tasks like loading the dataset from files, converting them into specific formats, normalising them, but also visualising results and saving generated masks as png files. The Python libraries used to support these operations are PIL and ImageIO. The Data Augmentation section of the code is defined in its own file.

9.5 augmentation.py

All the functions described in the data augmentation part of the dataset chapter are in this file. It contains the main function called during the creation of the dataset that applies augmentations with a certain degree of randomness to the images and mask given in argument and returns the augmented images and corresponding masks.

9.6 eval.py

This file contains an *evaluation* function that launches and handles the workflow of the evaluation phase on the test set.

9.7 loss.py

This files contains multiple helpers functions related to the loss and other metrics, that are called during training and/or evaluation phases. This is where the Cross Entropy Loss, the ROC AUC, F1-Score are computed. Some other unused loss functions are also defined here.

Chapter 10

Improvements

This section points out different leads to improve this project. Some of them were not realised for time reasons and others for purely materialistic reasons.

10.1 Dataset

The dataset could first of all be largely extended. Even for a model specialised in one environment, the dataset is small. Although good results are obtained but a larger dataset would result in a more robust model.

Extension should not only quantity, but also diversity. Looking at the SpaceNet challenge that has a dataset spanning five cities in drastically different styles and the robustness and the performance of the model that were produced by it, our dataset seems small extremely uniform. After the analysis of the result obtained, it can be confidently said that our architecture could be very performing on different environments, if it could benefit from such a large and diverse dataset.

10.2 Model

A very obvious idea would be too try the entirety of the UNet++ architecture with the filter sizes as large as they are in its initial form. This has not been done because of memory shortage on our GPU. To a certain point, more parameters

means more accuracy. However this could just saturate the accuracy and reduce the performance of the model. This can not be investigated without actual testing. One way to implement this increase in number of parameters while keeping the same memory usage, would be to decrease the input image size and change the architecture accordingly, splitting the dataset images in 4, for example.

There are different architectures that can be explored. We have only focused on one type because it was promising and the most adapted to our task. However, there are other variants of UNet that can be explored such as the Res-Unet [23], a combination of a UNet and the ResNet architecture or the attention Res-Unet [25], a variation of Res-Unet using attention modules. We chose not to try too many architecture and instead try to focus on one and fine-tune it.

More specifically, a promising architecture is the ICNet or the Image Cascade Network [26], another image segmentation architecture developed around the same time as UNet++, summer 2018.

Other approaches could be also considered. Change detection could use any object detection approaches, in addition to same-size segmentation masks, we could use larger areas segmentation masks, where instead of trying to determine the class of each pixel, we do it on larger areas, at the expense of clear boundaries. An approach using bounding boxes could also work.

10.3 Implementation

Like mentioned in section *8.4.3 : Time repartition in an epoch*, our model does not exploit multithreading and CPU/GPU parallelism. In our case it is not a huge problem because we can still train our models in a reasonable time frame (in a couple of hours in total), but with a larger model and dataset the time gain would be huge.

Chapter 11

Conclusion

This thesis focused on developing a Neural Network capable of change detection in satellite imagery, by generating a segmentation mask highlighting where new buildings were added. It takes two satellite images of the same place taken at different times as input. The output is a segmentation mask classifying each pixel as either change or no change. The segmentation mask has the same resolution as the input images.

We started by building our own dataset focusing on urbanized areas around Paris and in Belgium. We took pairs of 650×650 pixels pictures and built the corresponding segmentation mask by hand.

The next step was to test the UNet architecture, a popular architecture for image segmentation. It was trained from the ground up and looked at the resulting segmentation masks. The architecture had a F1 score of 38% on the test set. However a closer looks at the output masks showed that the contours of the building were imprecise. This means that the networks extract the correct information but not the localisation. From there we tried two different architectures to remedy to that. Firstly, the modular UNet to remove deeper paths were the context was supposed to be lost. Secondly, the lightUNet++ with the same depth as the UNet but with convolutions along and between the skip pathways.

As shown in the chapter dedicated to the comparison of the architectures, a shallower modular UNet does not resolve the issue of loss of context and brings other issues whereas the lightUNet++ results in a higher F1 score on the test set and visible improvements on the output masks. It has also the advantage to have 10 times less parameters than the starting UNet. We focused on this architecture

and improving its performance and robustness with data augmentation to obtain our final model.

There are leads to improve the model : using a larger and more diverse dataset, trying different image segmentation architectures such as ICNet, or allowing for more trainable parameters by using a GPU with more memory or using smaller input images.

In conclusion, our model is a lighter version of the UNet++ that achieves satisfying image segmentation performances on multi-temporal satellite imagery.

Appendix A

List of figures

1. Figure 3.1, Figure 3.2, Figure 3.3, Figure 3.4, Figure 3.5 and Figure 3.10 are original.

2. Figure 3.6 to Figure 3.9 reproduced from:
   Naoki Shibuya. Up-sampling with transposed convolution, Jan 2019

3. Figure 3.14 reproduced from:
   Sumit Saha. A comprehensive guide to convolutional neural networks, the eli5 way, Dec 2018.

4. Figure 3.11, Figure 3.12 and Figure 3.13 reproduced from:
   Harshall Lamba. Understanding semantic segmentation with unet. *Medium*, Feb 2019

5. Figure 5.1 reproduced from:
   Avanettten. Spacenet: A remote sensing dataset and challenge series. *GroundAI*, Jul 2018

6. Figure 5.2 to Figure 6.21 are original.

7. Figure 8.1 and Figure 8.2 reproduced from:
   reproduced from:
   Naoki Shibuya. Up-sampling with transposed convolution, Jan 2019

8. Figure 8.3 reproduced from:
   Zongwei Zhou, Md Mahfuzur Rahman Siddiquee, Nima Tajbakhsh, and Jianming Liang. Unet++: A nested u-net architecture for medical image segmentation. *CoRR*, abs/1807.10165, 2018

9. Figure 8.4 is original.
10. Figure 8.5 to Figure 8.17 are original.
11. Figure 9.1, Figure 9.2 are original.

Bibliography

[1] Avanettten. Spacenet: A remote sensing dataset and challenge series. *GroundAI*, Jul 2018.

[2] Héloïse Baudhuin and Antoine Lambot. LamboiseNet. `https://github.com/hbaudhuin/LamboiseNet`. Accessed: 13-06-2020.

[3] Derrick Bonafilia, James Gill, Danil Kirsanov, and Jason Sundram. Mapping for humanitarian aid and development with weakly and semi-supervised learning. 2019.

[4] Jason Brownlee. A gentle introduction to pooling layers for convolutional neural networks, Jul 2019.

[5] Chm. Key intuition about alexnet architecture, Jan 2019.

[6] F D. Batch normalization in neural networks, Oct 2017.

[7] Rodrigo Caye Daudt, Bertrand Le Saux, Alexandre Boulch, and Yann Gousseau. Urban change detection for multispectral earth observation using convolutional neural networks. *CoRR*, abs/1810.08468, 2018.

[8] Adam Van Etten, Dave Lindenbaum, and Todd M. Bacastow. Spacenet: A remote sensing dataset and challenge series. *CoRR*, abs/1807.01232, 2018.

[9] Maoguo Gong, Tao Zhan, Puzhao Zhang, and Qiguang Miao. Superpixel-based difference representation learning for change detection in multispectral remote sensing images. *IEEE Transactions on Geoscience and Remote Sensing*, 55(5):2658–2673, 2017.

[10] Maoguo Gong, Jiaojiao Zhao, Jia Liu, Qiguang Miao, and Licheng Jiao. Change detection in synthetic aperture radar images based on deep neural networks. *IEEE transactions on neural networks and learning systems*, 27(1):125–138, 2015.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[13] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[14] Harshall Lamba. Understanding semantic segmentation with unet. *Medium*, Feb 2019.

[15] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[16] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[17] Geesara Prathap and Ilya Afanasyev. Deep learning approach for building detection in satellite multispectral imagery. In *2018 International Conference on Intelligent Systems (IS)*, pages 461–465. IEEE, 2018.

[18] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[19] Sumit Saha. A comprehensive guide to convolutional neural networks, the eli5 way, Dec 2018.

[20] Mayar A Shafaey, Mohammed A-M Salem, Hala Mousher Ebied, Maryam N Al-Berry, and Mohamed F Tolba. Deep learning for satellite image classification. In *International Conference on Advanced Intelligent Systems and Informatics*, pages 383–391. Springer, 2018.

[21] Naoki Shibuya. Up-sampling with transposed convolution, Jan 2019.

[22] Hoo-Chang Shin, Holger R Roth, Mingchen Gao, Le Lu, Ziyue Xu, Isabella Nogues, Jianhua Yao, Daniel Mollura, and Ronald M Summers. Deep convolutional neural networks for computer-aided detection: Cnn architectures, dataset characteristics and transfer learning. *IEEE transactions on medical imaging*, 35(5):1285–1298, 2016.

[23] Xiao Xiao, Shen Lian, Zhiming Luo, and Shaozi Li. Weighted res-unet for high-quality retina vessel segmentation. In *2018 9th International Conference on Information Technology in Medicine and Education (ITME)*, pages 327–331. IEEE, 2018.

[24] LC Yan, YS Bengio, and G Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[25] Zhi-hai XU Yue-ting CHEN Qi LI Yue DONG, Hua-jun FENG. Attention res-unet: an efficient shadow detection algorithm. *Journal of ZheJiang University (Engineering Science)*, 53(2):373, 2019.

[26] Hengshuang Zhao, Xiaojuan Qi, Xiaoyong Shen, Jianping Shi, and Jiaya Jia. Icnet for real-time semantic segmentation on high-resolution images. *CoRR*, abs/1704.08545, 2017.

[27] Zongwei Zhou, Md Mahfuzur Rahman Siddiquee, Nima Tajbakhsh, and Jianming Liang. Unet++: A nested u-net architecture for medical image segmentation. *CoRR*, abs/1807.10165, 2018.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**