

École polytechnique de Louvain

Detection and Tracking of Railways in Satellite Images using Deep Learning

Authors: **Adrien LAUWERS, Jonathan SAMELSON**
Supervisor: **Pierre SCHAUS**
Readers: **Guillaume DERVAL, Mathieu PATINY**
Academic year 2019–2020
Master [120] in Computer Science

Abstract

Having an accurate and up-to-date network is essential for any transport service. This is particularly important in case of railway simulations where every distance has to be respected. At the moment, creating such a network needs human intervention and can take several days to reproduce a single railway line. Our work focuses on making the job easier by detecting the railroads in satellite images, using deep learning. In the recent years, deep neural networks have prevailed in image recognition tasks. This has been made possible thanks to the exponential exploding amount of data, the rise of computer power and the arrival of new algorithms. At the same time, the quality of satellite images has improved, allowing us to get enough details to enable such detections in most parts of the world. After the comparison of several deep convolutional neural network architectures, we build strong models which classify satellite images to detect the presence of railroads. We then use this underlying task to follow a railway line in order to reproduce its route as precisely as possible.

Acknowledgements

We wish to express our gratitude to Pr. Pierre Schaus, our supervisor, for his useful advice, constructive remarks and availability. His instant feedback was highly appreciated and allowed us to progress quickly and efficiently in this thesis.

We would like to acknowledge the staff of Transurb, Mr. Mathieu Patiny and Mr. Jérémy Piotte for their guidance. Their technical knowledge helped us to head this work in the right direction.

We want to thank Guillaume Derval for accepting to read our work and to attend our oral defense.

Finally, we are also grateful to our families and friends for their unfailing support not only during this semester but also throughout our years of study. In particular, we wish to thank Martin Techy for his careful reading, Margot Sartiaux for her encouragements and Joël Samelson for his ideas and suggestions.

Contents

1	Introduction	1
2	Context	3
2.1	Client Description	3
2.2	Client Needs	4
2.3	Problem Definition	5
2.4	Problem Formalization	7
3	Problem Solving	9
3.1	Big Data	9
3.2	Classification Problem	10
3.3	Dividing the Problem	11
4	Dataset	13
4.1	Finding a Suitable Dataset	13
4.2	Building our Own Dataset	15
4.3	Hiding Issues	17
5	Computational Power	20
5.1	Accessibility of Deep Learning	20
5.2	Google Colab	21
6	Deep Learning for Image Classification	22
6.1	Convolutional Neural Network	22
6.1.1	Needs of CNN	22
6.1.2	Layers in CNN	24
6.1.3	Typical Architecture	26
6.2	Comparison between Architectures	27
6.2.1	Use of Pre-defined Architectures	27
6.2.2	Results	28
6.3	Description of Used Architectures	31

6.3.1	Xception	31
6.3.2	MobileNetV2	33
6.4	Heatmap	34
6.4.1	Description of Heatmaps	34
6.4.2	Computation of the 2D Score Grid	35
6.4.3	Results	37
6.4.4	Heatmap for Object Localization	39
6.5	Training Phase	40
6.5.1	Xception	40
6.5.2	MobileNetV2	40
7	Area Detection	42
7.1	Neighboring Tiles	42
7.2	Neighboring Tiles Given an Azimuth	45
7.3	Results	46
7.4	Limitations	47
8	Rail Tracking	48
8.1	Main Method	48
8.1.1	Detection of the Railway on a Tile	51
8.1.2	Speed Factor	52
8.1.3	Perpendicular Tracking	53
8.1.4	Wrong prediction and Hidden Rail	54
8.1.5	Halting Cases	56
8.2	Estimation of the Next Point	56
8.2.1	Weighted Average Angle	56
8.2.2	Quadratic Lagrange Interpolating Polynomials	58
8.3	Checkpoints	62
8.4	Comparison Between Estimation Methods	63
8.4.1	Intermediate Results	65
8.5	Cleaning the Points	66
8.5.1	Naive Approach	67
8.5.2	Ramer-Douglas-Peucker Algorithm	67
8.5.3	Intermediate Interpolation	68
8.5.4	Comparison of the Cleaning Methods	69
9	Integration into the TET	72
10	Future Improvements	73
10.1	Image Quality	73
10.2	Additional Data	75

10.3 Altitude	75
10.4 Multiple Classes Classification	76
11 Conclusion	77
Bibliography	79
A Results of Area Detection	83
B Comparison of Rail Tracking Methods	89
List of Figures	91

Chapter 1

Introduction

Since the advent of computing, there is a constant interest in the automation of repetitive tasks, and more recently machine learning has played an important role. With the rise of machine learning, new algorithms have emerged and have brought new approaches to unexplored challenges. Deep learning is a good illustration of this emergence, it takes advantage of the large amount of data to solve problems of various fields with an incredible accuracy. It includes areas such as audio recognition, bioinformatics, natural language processing and many more.

This master thesis is foremost a research work exploiting cutting-edge technologies to solve a problem of image recognition. This field is prevailed by deep learning algorithms thanks to the method of convolutional neural networks (CNN). AlexNet is the first model that popularized convolutional networks when winning the ILSVRC 2012 challenge. Its success kicked off a renaissance in the deep learning community and many CNN models have seen the light of the day over the past few years.

Our work is incorporated within the context of railway simulations, in partnership with Transurb, a company of land transport-oriented services which has been delivering rail simulators for the past 15 years. It is essential for any transport service to have an accurate and up-to-date network, and particularly important in the context of railway simulations, where every distance has to be respected. Transurb clients can model a rail network for their simulators to train future drivers. At the moment, it requires repetitive human efforts and can take several days to reproduce a single railway line. To ease this task, we classify satellite images with a CNN to reproduce the lines accurately without human intervention.

In this document, we first define the client needs and formalize our task as an optimization problem. We will give more details on the way to solve the problem and on the creation of our dataset. Then, we get to the heart of the matter and tackle convolutional neural networks. We will bring theoretical notions necessary to understand the global working of this topic and details the architectures that are most suitable for this task. Afterwards, we explain how we make use of our trained models to build an algorithm capable of tracking a desired railway. We will give some heuristics that are suitable within the scope of the rail sector and combine them to obtain an efficient solution. Final chapters will be dedicated to the results in relation to our optimization problem and will list possible future enhancements.

Chapter 2

Context

This section introduces our client, defines the problem and describes the genuine added value of this project for our client, Transurb¹.

2.1 Client Description

Transurb is a Belgian limited liability company of land transport-oriented services, pooling the experience and resources of its shareholders:

- John Cockerill (Cockerill Maintenance & Engineering)
- SNCB (The National Belgian Railway Company)
- STIB (The Brussels Intercommunal Transport Company)

Transurb Simulation has been delivering rail simulators and advanced training solutions across the world for the past 15 years, focusing on European Rail Traffic Management System Levels 1 and 2. They provide three types of simulator:

- The nano simulator, a computer-based solution designed to be installed on any standard PC. The quick and cost-effective solution that offers the highest degree of portability.
- The compact simulator, an advanced training unit providing a high degree of immersion while optimizing the space dedicated to simulators in the learning center. A trade-off between the level of portability and the degree of immersion.

¹Transurb : transurb.com

- The full-cab simulator, it provides the most intense and accurate driving experience to trainees. This complete and top-level solution can be used to teach most topics and operational needs trainees.

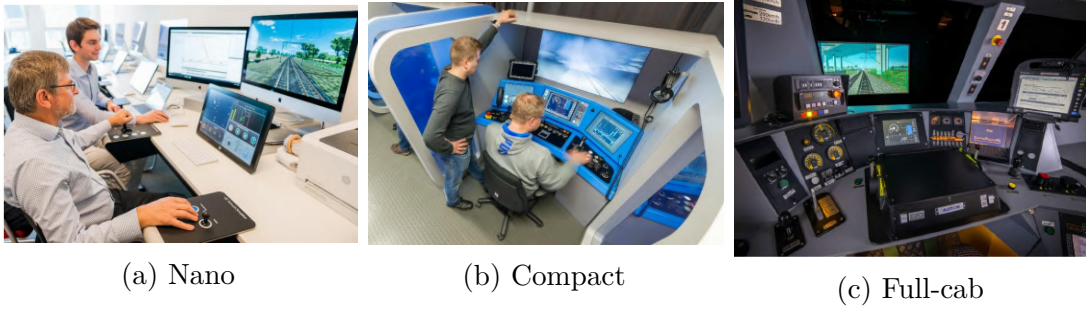


Figure 2.1: Three simulator solutions proposed by Transurb

2.2 Client Needs

The Transurb company has two types of customers:

1. Some of them use the simulation to get familiar with a specific model of train and its kinetic behavior, traction power, braking system, rail interaction. Indeed, the customer can easily learn procedures such as coupling and uncoupling, start and stop, boarding of passengers, etc., allowing the customer to get a better understanding of specific models.
2. Others use it to have a better knowledge of a specific railway line, the simulation can recreate the network and its environment in a way that is faithful to reality. Drivers can thus become accustomed to the actual network with its stations, switches, level crossings, tunnels, bridges, critical portions, etc. All the signals, panels, marker boards and beacons found on the network are modeled to get drivers familiarized with the signaling.

For the second case, the lines, level crossings, tunnels, signals and all other components of the rail line must be provided with accuracy by the instructors to the simulator to make the simulation realistic. The creation of the virtual line is made with one of their software called “Track Editor Tools”.

At the moment, all these pieces of information must be provided manually, taking a lot of time to recreate the network accurately. However, some tasks can be automated, saving considerable time for instructors. Indeed, the creation

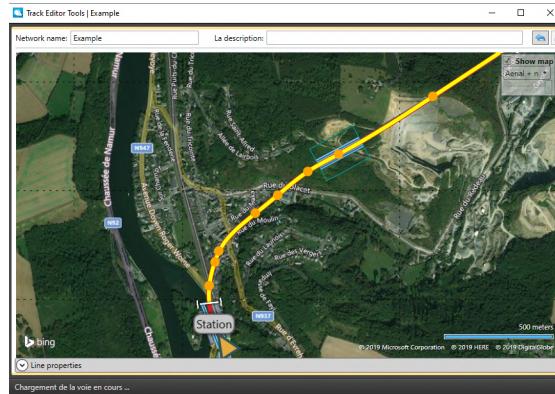


Figure 2.2: Example of a track in the Track Editor Tools

of the rail line itself (without signaling) can take several days. If we build an artificial intelligence capable of recognizing the rails in satellite images, we could then recreate the network and then provide this information to the “Track Editor Tools”, avoiding the instructors to create it from the ground up.

2.3 Problem Definition

The problem is to find the curve of the railway line that connects two given coordinates using satellite images to find the presence of railroads, and output a list of coordinates that describes the path as accurately as possible. This list of coordinates is stored in an XML file, readable by the Track Editor Tools (TET) to produce the desired rail network in this software, as shown in Figure 2.3.

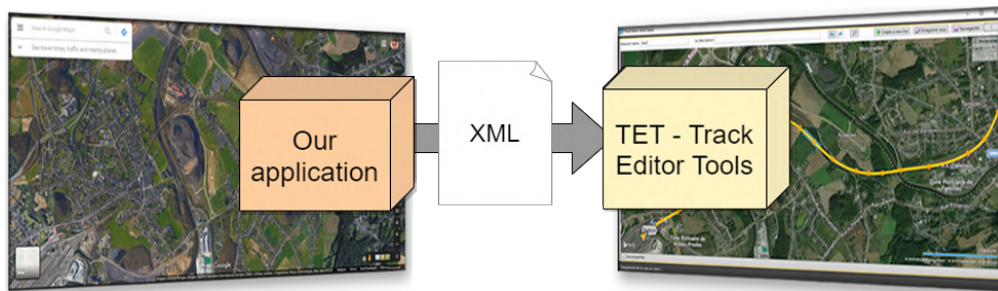


Figure 2.3: Interaction between our application and the TET

The two positions are provided by the user using two geographical coordinates. Satellite images are provided by a satellite image service like Google Map, Bing Map, etc.

A revision of the output of the program must be done to adjust the coordinates of the path that are not exactly on the railway. Consequently, the list of coordinates describing the railway must be as small as possible while maintaining the same curve shape. This allows the end user to review the result faster as there are fewer coordinates to adjust, as shown in Figure 2.4.

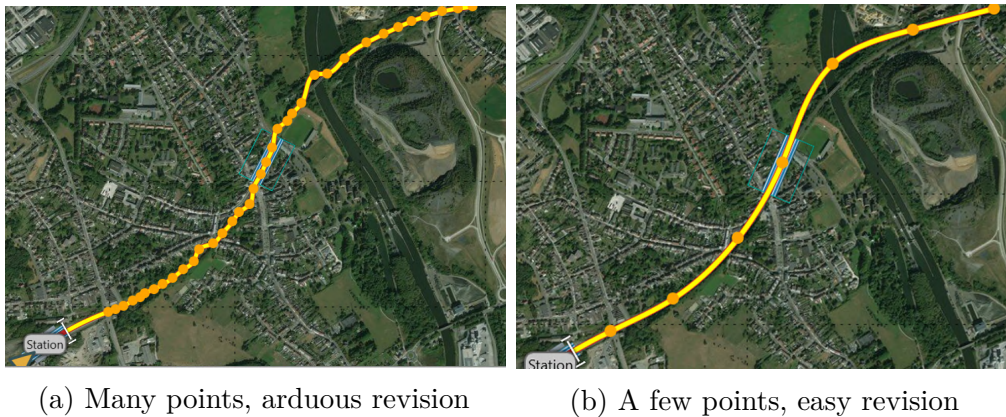


Figure 2.4: Illustration that highlights the importance of keeping only a few coordinates

Another point to mention is that our task is to create the curve of the railway, also called the torsion axis by the company Transurb. In the TET, the user has a library of modules of a railway topology at its disposal as illustrated in Figure 2.5. These modules can be assembled one after the other along the torsion axis to model the number of lines, the switches and other rail components along the generated torsion axis.

Figure 2.6 illustrates an example of the modeling of a real network in the Track Editor Tools. The green line represents the torsion axis that must be found, in red the five modules added to this torsion axis by the user.

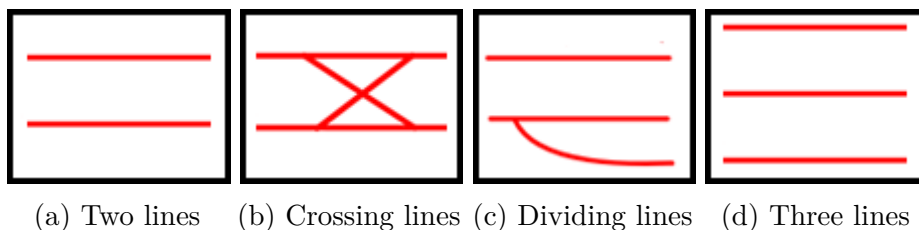


Figure 2.5: Example of modules of railway topology

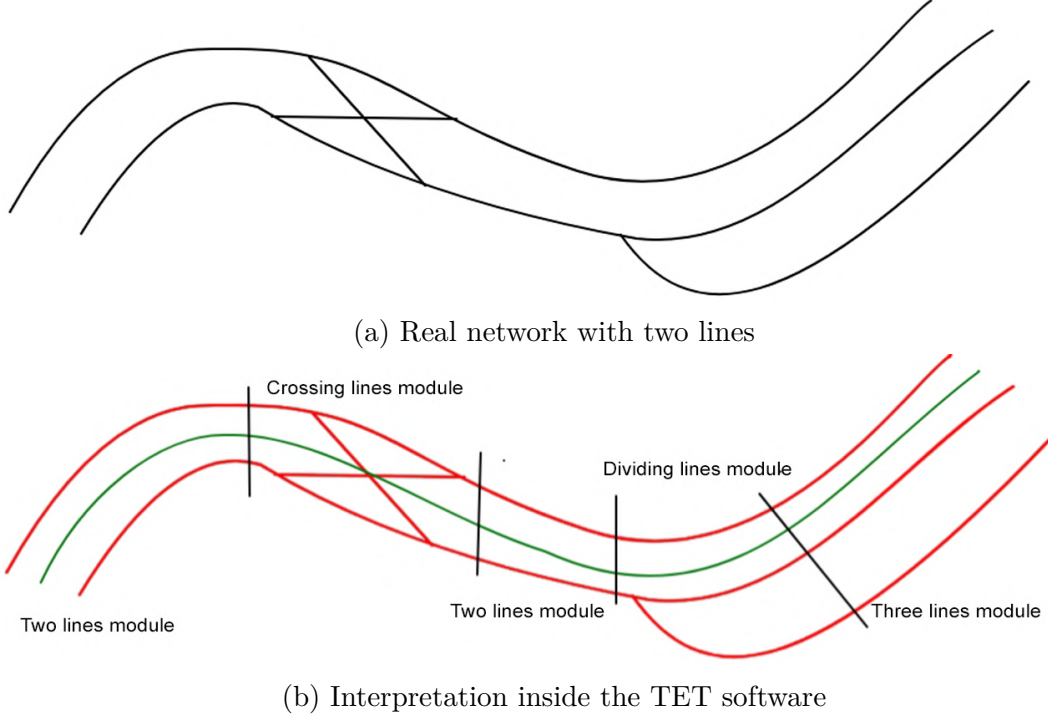


Figure 2.6: Modeling the network into the TET

2.4 Problem Formalization

In this section, we formalize the problem and explain how we can measure the performance of our solution.

Given $GT = \{x_1^g, \dots, x_i^g, \dots, x_n^g\}$ a list of coordinates defining the ground-truth railway path, where x^g is defined by a latitude and a longitude. Let $curve_A$ be the spline that passes through the list of coordinates A. The aim is to find $E = \{x_1^e, \dots, x_j^e, \dots, x_m^e\}$, a list of coordinates such that $curve_E$ gives the most accurate estimation of $curve_{GT}$.

To evaluate the accuracy of $curve_E$, let us define $ABC(curve_{GT}, curve_E)$, a metric that gives the Area Between the Curves $curve_{GT}$ and $curve_E$. We also want to minimize $|E|$, the number of points in E , in order to avoid redundant points along the path and make faster adjustments to the estimation. The problem is thus formalized as the following optimization problem:

$$\text{minimize} \quad ABC(curve_{GT}, curve_E) + \lambda \cdot |E|$$

The lambda factor is not part of the solution we provide. It is a factor to balance the weight of the number of estimated points with the ABC, which have different orders of magnitude. Giving more weight to $|E|$ would result in fewer points whereas giving more weight to the ABC would give a more accurate result. This trade-off between speed and accuracy is left to the final user because it depends on the user preference and on the complexity of the railway line.

The Area Between the Curves is simply the sum of the yellow areas shown in Figure 2.7. The green path being the ground-truth path (GT), the red path being the estimation (E). A spline is described by a huge number of points connected by straight lines, which altogether form a curve. It can be computed by intermediate interpolation given an initial list of points. The intersection between two splines forms polygons with a huge number of sides. Each of the yellow area is a polygon, and the area of these polygons can be easily computed. We can then sum the area of these polygons to get the ABC.

Figure 2.8 shows the ABC when the polygons are described with very few points. The use of splines gives a more realistic representation of the real rail curve, eases the creation of the ground truth (as much fewer points are needed to describe the real curve) and is closer to what is used in the Track Editor Tools.

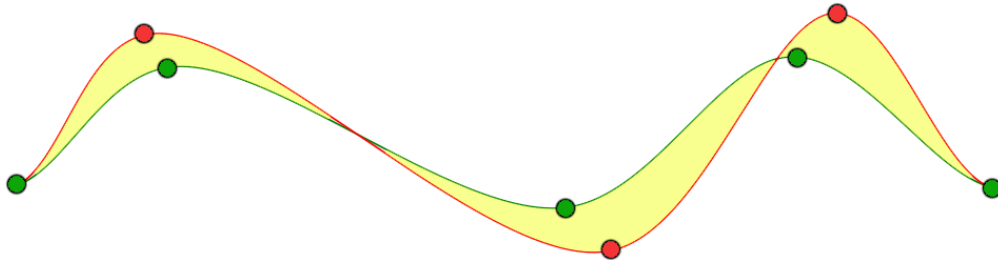


Figure 2.7: Illustration of the Area Between the Curves

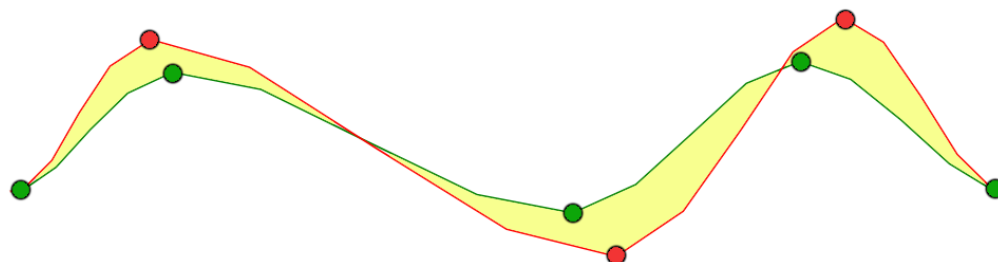


Figure 2.8: Illustration of the ABC, with much fewer points

Chapter 3

Problem Solving

3.1 Big Data

Nowadays, the two words “big data” are on everyone’s lips, but only few people know what it is exactly. The definition of big data is “very large sets of data that are produced by people using the internet, and that can only be stored, understood, and used with the help of special tools and methods”[1].

Deep neural network is a method that takes advantage of this huge amount of training data in order to understand it and to draw a pattern that traditional machine learning methods could not find.

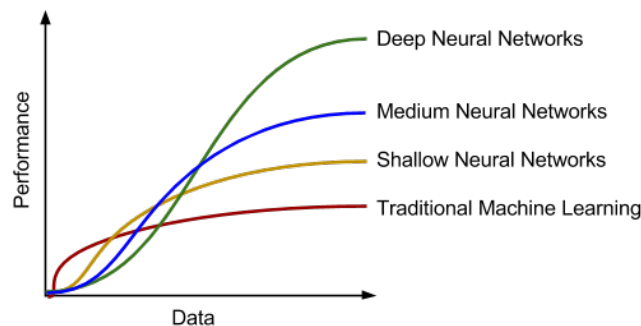


Figure 3.1: Performance of machine learning methods as a function of the amount of training data [2]

Of course it does not mean that traditional learning methods such as logistic regression or SVM will not still perform well, it only means that even if we give them more training data, they will usually reach a plateau in their performance [3].

For some tasks, however, we do not have this amount of data, simply because they might be harder to obtain or produce.

Nevertheless, in our case, railways are everywhere on the planet and in the recent years, we have obtained more and more detailed images of the Earth allowing us to take advantage of this so-called big data to obtain a strong model with high performance.

3.2 Classification Problem

The recent progress in deep learning shows us new algorithms capable of identifying different objects, people, animals and they are even capable of localizing it in the space with incredible accuracy. These kinds of algorithms work generally by adding a box surrounding the object, and labeling them thanks to the general shape of the object. As an example, the YOLO [4] algorithm uses bounding boxes to localize objects in an image, as the Figure 3.2 shows.

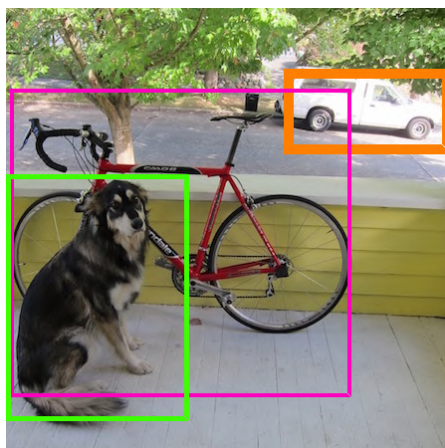


Figure 3.2: Object detection using the YOLO algorithm

However, in case of railroads in satellite images, it is not possible to get the shape of a railway, because no matter the zoom level, the rail usually crosses the image by coming from a border to another. Moreover, even if we could add a box on part of the network, it would not capture the exact path that it describes.

Thus, our idea was to cut the region of interest by little pieces, more generally called “tile”. Those little squares cannot be too small as they need to capture part of the context that is next to the rails. Nevertheless, they cannot be too large

in order to give the best accuracy of the rails position, as we are considering the center of the tile to be the position of the rails.

Indeed, in the context of big data, it is impossible to output the exact path of the railway for every single tile. A possible solution would have been to do mask segmentation, meaning that all our training set would be pixel-wise labeled as shown in 3.3. With mask segmentation we would know each pixel that is part of the rail for each tile. However, this solution was too laborious to be done manually according to the amount of training examples we had.



Figure 3.3: Mask segmentation, manually labeled

Instead, we choose to separate our dataset in two classes: the label is 1 if the image contains a rail and 0 if it does not, leading to a classification problem where our model must predict small tiles of 256×256 pixels as "rail" or "non-rail".

3.3 Dividing the Problem

Thereby, we can separate our work in two different problems. The first is the classification problem mentioned above, where the program must be able to classify small tiles as "rail" or "non-rail", this problem is detailed in chapters 6 and 7.

The second part of the problem is to follow a specific railroad from an origin to a destination by minimizing the number of classified tiles to reach this destination. Indeed, we cannot classify all tiles of a specific region (defined by a rectangle in the map for instance) to find the railway as it would take too much time. Also, we could

find multiple railroads in the region which do not belong to the desired railway line (see more details in section 7.4). Consequently, we must design an algorithm that can track the desired railway while minimizing the number of tiles to download and classify, this algorithm that we call “Rail Tracking” is detailed in chapter 8.

This leads us to two orthogonal problems. Indeed, the choice of the classification model does not affect the implementation of the rail tracking algorithm and vice versa. Figure 3.4 describes our work flow. The transitions follow the possible ways to improve the solution.

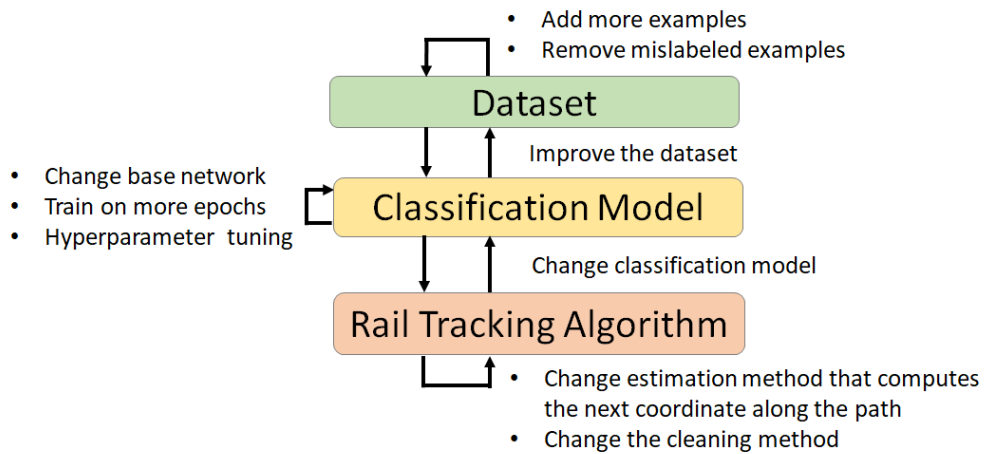


Figure 3.4: Work flow

Chapter 4

Dataset

This chapter introduces the important elements to be considered to constitute our dataset as well as how we create it.

4.1 Finding a Suitable Dataset

For this task, we need satellite images as well as labels to describe the presence of rails on them. Images are easy to obtain from satellite image providers and deep learning competitions, but finding a labeled dataset suitable for our classification problem is impossible to find. Often, we could find datasets of good quality but they did not contain enough railways, as they are often built for object detection competitions (such as detecting airplanes, boats or cars). Also, the zoom level was very variable from one image to another.

Also, the size of each example must be the same so that our model gets uniform input data. This is why we generate and use tiles of size 256 by 256 pixels, which corresponds to a size where we get enough context without the center being too far from the rails when they are decentered.

The zoom level determines how much of the world is visible on the map [5]. Thus, it defines a scale (on the screen) that determines the level of detail we get in a single image. It is a common measure used by maps services. Thereby, an image with zoom level 0 shows the whole world while the zoom level 10 usually shows a metropolitan area [6].

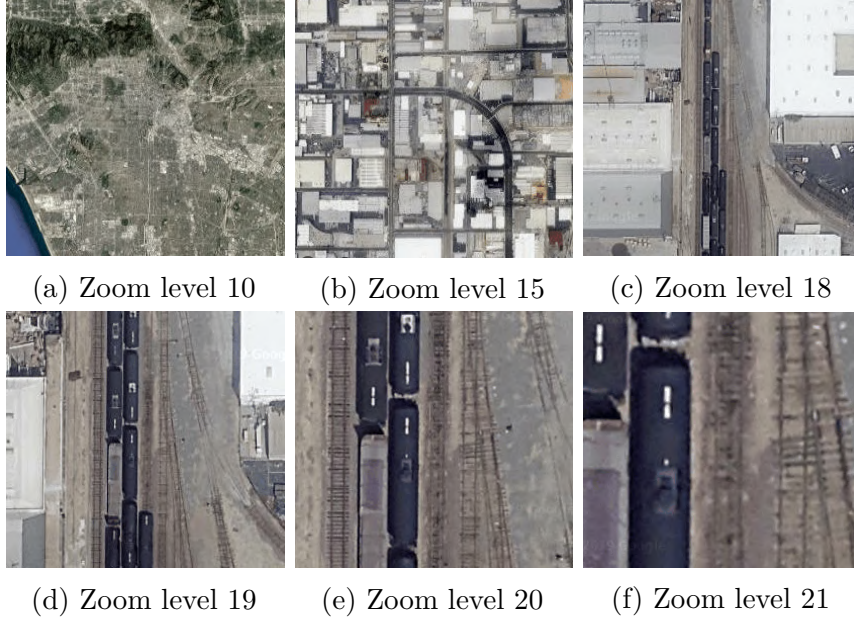


Figure 4.1: Tiles showing the different zoom levels provided by Google Maps

For our task, we need as many details as possible in order to feed our neural networks with the components that make a human capable of recognizing a railway (the railroad ties, catenaries, etc.). Moreover, it will allow the center of the tiles to be as close as possible from the actual position of the railways. Therefore, we choose the zoom level 20, which is a good compromise between the level of detail and the availability worldwide.

All satellite image providers do not offer the same zoom level range in the different regions of the world. They are better in the developed countries than in the developing countries, and they are often better in the cities than in the countryside. Nevertheless, according to Transurb needs, our artificial intelligence should be able to recognize railways worldwide, not only where good quality is provided. However, it is difficult if even a human eye cannot say whether there is a railway or an unmade road. Figure 4.2 shows the quality provided in different countries.

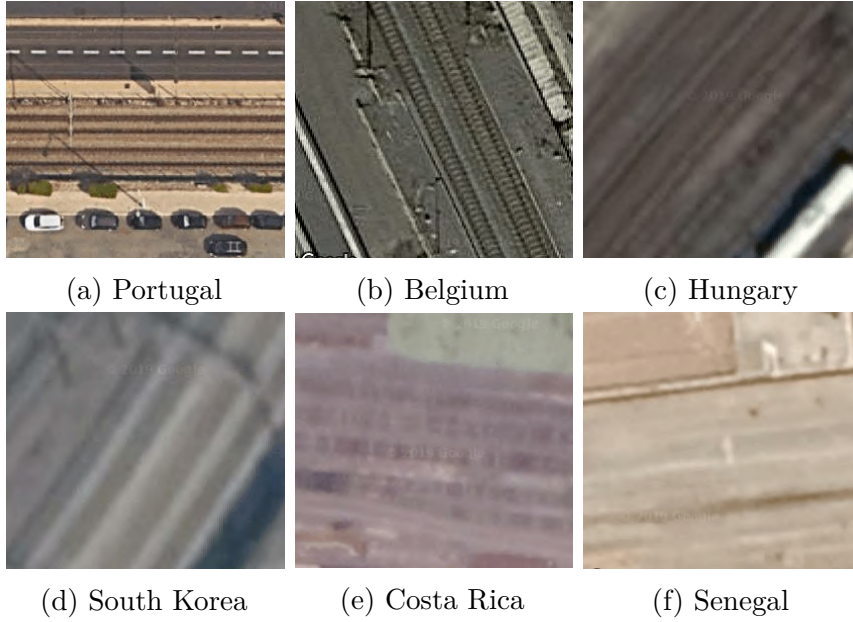


Figure 4.2: Tiles showing the quality provided by Google Maps in different countries

For this reason, we choose to focus our work on image quality such as the one we can obtain in Portugal, Belgium and in some regions of Hungary. We expect the image quality of the other countries to be improved over the next few years, allowing our artificial intelligence to do the same job there.

The recent deep-learning results have shown the capability of exceeding human performance in some fields. Thus, we can expect to do well on this task which is quite basic for a human. By outputting a certainty percentage, the threshold could even be configured to deal with blurry tiles. However, it is hard to believe that it will be able to follow a railway line, because it requires a very high accuracy that it will not have all along the way.

4.2 Building our Own Dataset

Those reasons urged us to create our own dataset with the following criteria:

- Size of 256 by 256 pixels
- Zoom level of 20
- Labeled "rail" $([1, 0])$ or "non-rail" $([0, 1])$

- Good quality allowing a human eye to recognize rails
- The proportion of rails in the image should be sufficiently high

As explained previously, manually labeling each tile would be too laborious, so this process must be automated.

OpenStreetMap ¹ can be used as a labeling tool for creating the dataset. Indeed, it can provide the position of the rail network in a specific area. The idea is then to extract the coordinates of these rails and generate tiles of satellite images of those specific positions. We can get this kind of data easily from OpenStreetMap thanks to a web service called “Overpass-turbo” ². With simple queries, we can get the rail positions in cities, regions or even countries. Thereby, we have selected some places all over the world to get rails with different aspects.

This can seem a bit paradoxical. If OpenStreetMap references the rail positions, we could use them directly to recreate the line in the simulator. After all, map providers use them to create their own route on their plan view. After some investigations, we noticed that OpenStreetMap does not have data for every regions of the world. Sometimes only parts of the network is known. This is probably because it is added by contributors, just like Wikipedia. Other services are proprietary or often less complete than this open source service. Nevertheless, OpenStreetMap is sufficiently accurate to retrieve some positions of rails and build a dataset.

Using the plan view instead of satellite imagery would require to be service dependent, as the railway lines are displayed differently from one application to another. Although we will use a service to get the satellite images, they all display the Earth as it is.

Several services for generating tiles at a specific position can be used. However, each of these services does not offer the same quality and resolution. As an example, Figure 4.3 shows the quality difference between the tiles at the same coordinates, provided by Google Maps, Bing Maps and Mapquest.

¹OpenStreetMap : openstreetmap.org

²Overpass-turbo : overpass-turbo.eu/



(a) Google Maps

(b) Bing Maps

(c) Mapquest

Figure 4.3: Tiles showing the quality provided by different satellite image services

As you can see, there is a huge quality difference between these three images. Tiles from Bing Maps do not provide a good quality and zoom level to recognize the rails, it could easily be confused with a road. Mapquest provides a decent quality but does not provide a sufficient zoom level to give an accurate rail position. The quality, the maximum zoom level and resolution of the satellite images of each service depends on the location, but Google Maps seems to be globally the best choice.

Google Maps provides an API to retrieve the tiles of satellite images, called “Maps Static API”. We simply have to give the coordinates of the desired tile, its zoom level, resolution and the map type (satellite in our case). This service could be completely interchangeable as we have all the required parameters and we do not have any other dependency with Google.

Finally, to build our database from the ground up, we generate images labeled "rail" by passing the selected rail coordinates to the Maps Static API. Conversely, to generate the images labeled "non-rail", we select random coordinate around a specific position and we check that this coordinate is sufficiently far from the closest rail. With this approach, we generated about 60.000 tiles, out of which 10% of them were taken for the validation set and 10% for the test set.

4.3 Hiding Issues

Although aerial images were taken at times that favor recognition (favorable weather conditions), this did not prevent us from finding places where the railroads are hidden. Sometimes because of clouds, but also because the pictures were captured

when the sun was not at its zenith, creating shadows on the tracks (see Figure 4.4). This makes rail recognition difficult or sometimes impossible. Another recurrent issue is the case of the tunnels. Obviously, the quality improvement of the satellite image provided by Google will not solve the problem as the track will always be hidden.

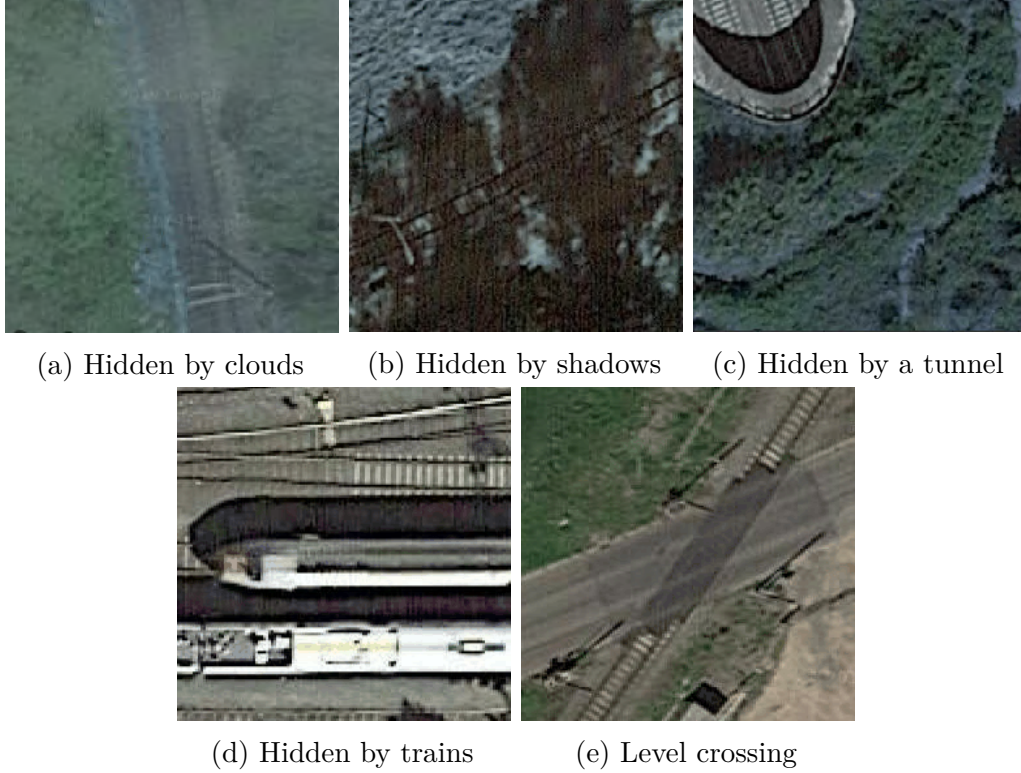


Figure 4.4: Tiles showing hiding issues encountered for the classification

Amongst the images generated in our dataset, some include trains that randomly crossed the path at the moment when the image was taken. This proportion is relatively negligible and does not guarantee that a track will be considered as "rail" by our final model if a train is on top of it. It would have been interesting if our model predicted those tiles as "rail" but we could not include more images containing trains in our dataset because we do not have any way to easily find them on the tracks.

Conversely, the level crossings represent a case where the railways are partly visible, but simply harder to recognize. Luckily, many level crossing coordinates are available in OpenStreetMap database. Therefore, we add to our dataset examples of level crossings, so that our model can learn that they are parts of the railroads.

All those issues prove that a post-selection is required to remove rail tiles that are actually not rails. Since OpenStreetMap does not contain all rail coordinates, the opposite also has to be done because we might have retrieved rail tiles while looking for non-rail tiles.

Having mislabeled examples is a common issue in machine-learning datasets. It only becomes a problem when there is a sufficiently high proportion of the error rate that comes from the mislabeled. For instance, if a model gets 15% of error rate and 0.5% is due to the mislabeled examples, then it is much more interesting to spend time improving the remaining 14.5% instead.

Chapter 5

Computational Power

In this chapter, we explain how we obtain the computational power needed to train our models. As you may know, deep learning requires more computational power than traditional machine learning techniques to avoid long training periods.

5.1 Accessibility of Deep Learning

Besides big data, deep learning has emerged during the last decade thanks to the rise of computer power. Indeed, faster computation enables the training of much bigger neural networks, and results in huge progress of the trained models in terms of performance. Moreover, as more researchers get into deep learning, new algorithms see the light of the day that also increases the speed of training. Thereby, deep learning becomes more and more accessible to everyone who would like to build their own model.

The operations behind the scenes of deep learning mainly consist of simple matrices and vectors operations. Usually, they can be executed in parallel for faster result. This is the kind of procedure that is much easily handled by a GPU than a CPU. The latter has few cores which can run complex processes sequentially with few threads, whereas the former has a large number of simple cores which allow parallel computing through thousands of threads computing at a time.

Furthermore, the large amount of data has to fit into the memory while training a model as well as all the intermediate computations and the parameters. GPUs have dedicated amount of memory, called VRAM, which makes them bandwidth optimized [7].

Obviously, our typical laptops are not suitable for this work. First, they only have an integrated graphics processor and they have only a small amount of RAM. We thus had to find a solution to train our model without it taking ages.

Our first idea was to use computers provided by our university to get more powerful resources. However, they do not have GPUs compatible with CUDA¹, a Nvidia technology allowing the GPU to handle general-purpose computations in the place of the CPU[8]. Furthermore, the training of our models could take several hours or days and we could not monopolize those computers.

5.2 Google Colab

Google Colab platform is a free cloud service offered by Google to encourage machine learning and artificial intelligence research. They provide Jupyter notebooks attached to back-ends that include a Nvidia Tesla K80, 12 GB of dedicated VRAM, 25 GB of RAM and 358 GB of disk storage [9]. This is the best solution to train our models quickly at no cost.

Thanks to that, our 55.000 images of training and validation data can now fit into memory. However, with the intermediate computations, we notice that we could not add more training examples. If we want to increase the size of our dataset, we would need another solution, such as attaching another back-end, but this solution would be very costly.

¹CUDA : developer.nvidia.com/cuda-zone

Chapter 6

Deep Learning for Image Classification

In this chapter, we first describe the rudiments of Convolutional Neural Network (CNN or ConvNet for short). Then, we compare some architectures and detail the ones that we selected for this work based on the obtained results. Finally, we give some insights of the features detected by our models and we explain how they were trained.

6.1 Convolutional Neural Network

This section does not claim to describe the full working of a convolutional neural network, but explains key concepts that are required to understand the deep learning models used in our work. This content is partly inspired by [11, 12, 13, 14, 15, 16, 17, 18, 19].

6.1.1 Needs of CNN

Imagine if we only use a traditional deep-learning model with two fully-connected hidden layers and one output layer of two units (rail and non-rail) as illustrated in Figure 6.1. Each tile of the dataset is a RGB image of dimension $256 \times 256 \times 3$, meaning that the model takes 196.608 input values. Let's say we want the fully-connected layer to have 2048 units. It means that if we connect every neurons together, the number of parameters in the weight matrix would be 196.608×2048 which is about 47 million parameters, just for a single layer. This is impractical knowing that accurate deep-learning models have many more layers.

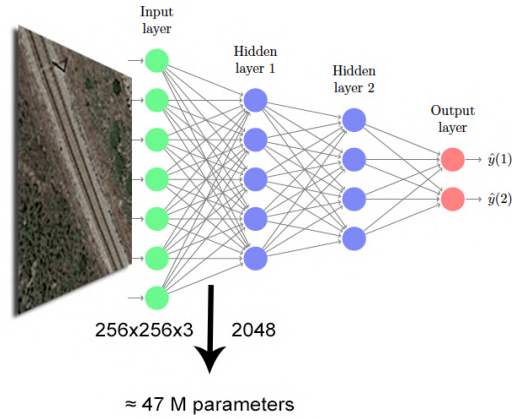


Figure 6.1: A simple deep-learning model

Instead of ordinary neural networks, the ConvNet architectures take advantage of the fact that inputs are images. Natural images have the property of being “stationary”, i.e. the features that we learn on a part of the image can also be applied to other parts of the image. With this property, we can build architectures that are locally connected. Concretely, the neurons of one layer will only be connected to a small region of the previous layer, instead of all neurons as in a fully-connected layer.

In CNN, neurons are arranged in three dimensions: width, height and depth to deal with 3D activation volumes. As illustrated in Figure 6.2, the 3D input volume of our model is $256 \times 256 \times 3$. Indeed, our input images are as wide as high and contain three color channels. Finally, we end up with an output volume that is a $2 \times 1 \times 1$ vector containing the class probabilities.

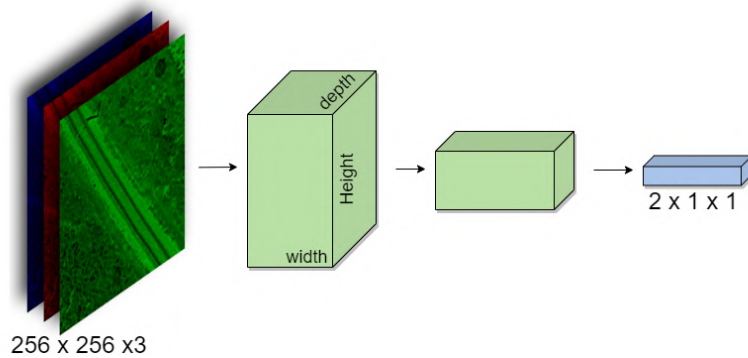


Figure 6.2: CNN adapted to this task

6.1.2 Layers in CNN

As we have just seen, one layer transforms a 3D activation volume into another. ConvNet typically uses two main types of layers in addition to fully-connected layers, each of them has different usage: convolutional layers are used for feature extraction while pooling layer is used for dimensionality reduction. The next subsections describe how these layers work.

Convolutional Layer

Convolutional layers are divided into two parts: feature extraction through convolutional filters and feature map enhancement with an activation function.

- **Feature Extraction with Convolutional Filters**

A convolutional layer first extracts features from these images through the application of sliding convolutional filters. Hundreds of convolutional filters are applied to the activation volume, each of them resulting in a feature map: a filtered version of the image. If n filters of size $(f_h \times f_w \times d)$ are applied on an activation volume of size $(h \times w \times d)$, it outputs a volume dimension of size $((h - f_h + 1) \times (w - f_w + 1) \times n)$ as illustrated in Figure 6.3 (with only one filter). The output activation volume is then composed of n feature maps also called “channels”. [13]

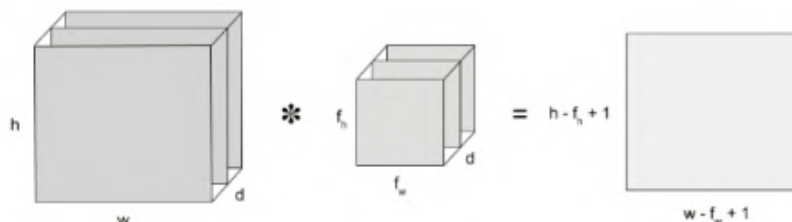


Figure 6.3: Convolution of an RGB image with a filter [13]

Figure 6.4 shows an example of the application of a convolutional filter of size $(3 \times 3 \times 1)$ on a small image of size $(5 \times 5 \times 1)$. Figure 6.4a shows the first step of the convolution. The weight of the filter is in red and its current position in orange. The result is the product-wise operation of the filter and the pixel values of the image where it is applied. The filter is then shifted one cell at a time from left to right and from top to bottom until the image is fully traversed, as shown in Figure 6.4b.

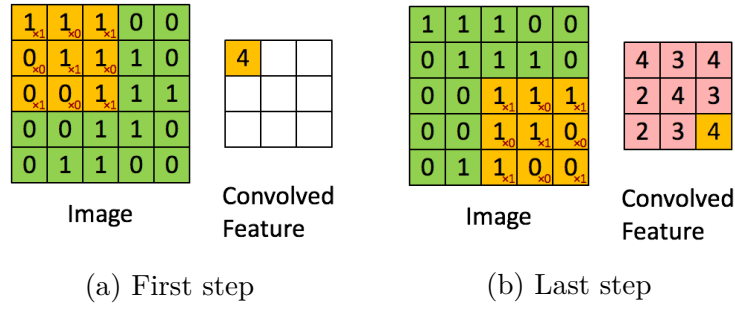


Figure 6.4: Steps of the application of a convolutional filter [13]

The values of the filters are the parameters of the convolutional layer, these values are learned during the training phase and updated at every training iteration, just like in a traditional deep learning network. The different options (called hyperparameters) to control a convolutional layer are:

- The stride: it is the number of pixels shifted at each step of the application of the filter. When the stride is 1, we move 1 pixel at a time as shown in the previous figure. When the stride is 2, we move 2 pixels at a time and so on.
- The padding: it adds empty pixels around the image so that the output of the convolutional layer has the same dimension as its input. On the previous example, no padding was used, leading to a size reduction of the output volume.
- The number of filters to be applied on the activation volume.
- The height and width of the filter, the depth depends on the number of channels in the activation volume.

• Feature Map Enhancement with activation function

The activation function decides whether a given neuron from the output of the feature extraction sends a signal to the subsequent layers. This enables the model to reduce the amount of processing required for the CNN while maintaining neurons that are useful as they contain meaningful aspects.

For instance, the ReLU function is a commonly used activation function, it replaces all the negative activation values in each feature map by 0 (see Figure 6.5). It also reduces noise within the network, the dangers of overfitting and incorrect learning correlations between inputs and the outputs.

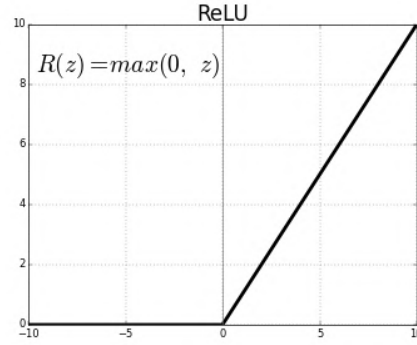


Figure 6.5: The ReLU activation function [10]

Pooling layer

The goal of a pooling layer is to reduce the feature maps size without the loss of information, saving time and resources at training time. To do that, the layer proceeds like with a convolutional filter, except that instead of making an element-wise product with a filter, it takes the maximum (max pooling), the average (average pooling) or the sum (sum pooling) of the pixel values. Thence, there are no parameters learned during the training phase. The three types of pooling layers are illustrated in Figure 6.6.

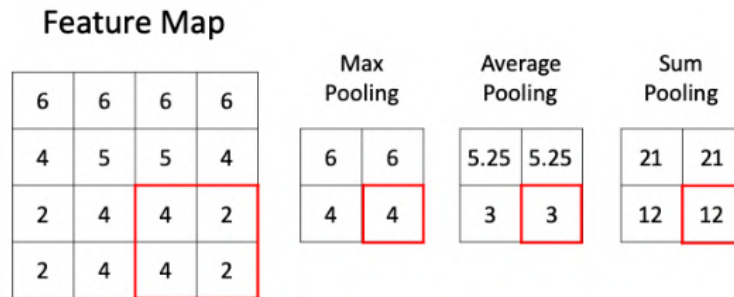


Figure 6.6: The three types of pooling layers [17]

6.1.3 Typical Architecture

Figure 6.7 shows a typical CNN architecture. It is made of two phases:

- A “feature learning phase” composed of Conv/ReLU and pooling layers which extracts meaningful features for the classification phase.

- A “classification phase”, after that the last 3D activation volume is flattened, just like a traditional deep-learning model. It is composed of successive fully-connected layers that produce a final output vector of K dimensions, where K is the number of classes to predict. In our case, there are two classes, "rail" or "non-rail".

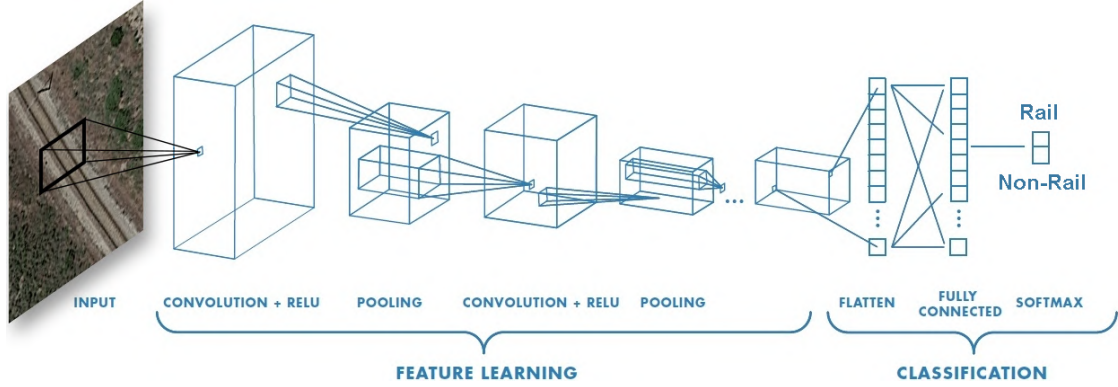


Figure 6.7: A typical CNN architecture, adapted to this task [13]

6.2 Comparison between Architectures

6.2.1 Use of Pre-defined Architectures

Although creating a convolutional network from scratch gives the most control over the network, it requires to have a deep understanding of the structure of a neural network and the configuration/options of each type of layer. There are many combinations of network layers and as we are not deep learning experts, we are not able to create a network configuration that has equal or higher performance than pre-defined networks. It is more pertinent for us to take network configurations made by researchers that have already proven their worth.

Moreover, the use of an existing network configuration allows us to use models that were pre-trained on another task and use their parameters for our task as a starting point. This process is known as “transfer learning”. The advantage of transfer learning is that the pre-trained model has already learned a set of meaningful features. We can thus keep up the training on our classification task and reduce the required computational cost and time.

6.2.2 Results

The first image classification model that has proven its worth is made by Krizhevsky et al. and is named AlexNet. This model is composed of five CNNs layers and three FC layers. Alexnet is an improved version of LeNet5 by Yann LeCun, it achieved 15,3% top-5 test error rate in ILSVRC 2012 challenge, which was much better than the 2nd place that reached 26,2% error-rate.[20] Its success kicked off a convolutional neural network renaissance in the deep learning community.

Ever since, many CNN models have seen the light of the day over the past few years. In this work, we chose five architectures available in Keras framework [21] that has a decent number of parameters ($< 25.000.000$). We consider the following architectures: Inception-v3 [22], ResNet-51 [23], MobileNet-v1 [24], MobileNet-v2 [25], Xception [26].

The ball chart of Figure 6.8 shows the top-1 accuracy on the ImageNet-1k validation set with respect to the computational complexity of their architectures. The ball size corresponds to the model complexity (number of parameters).

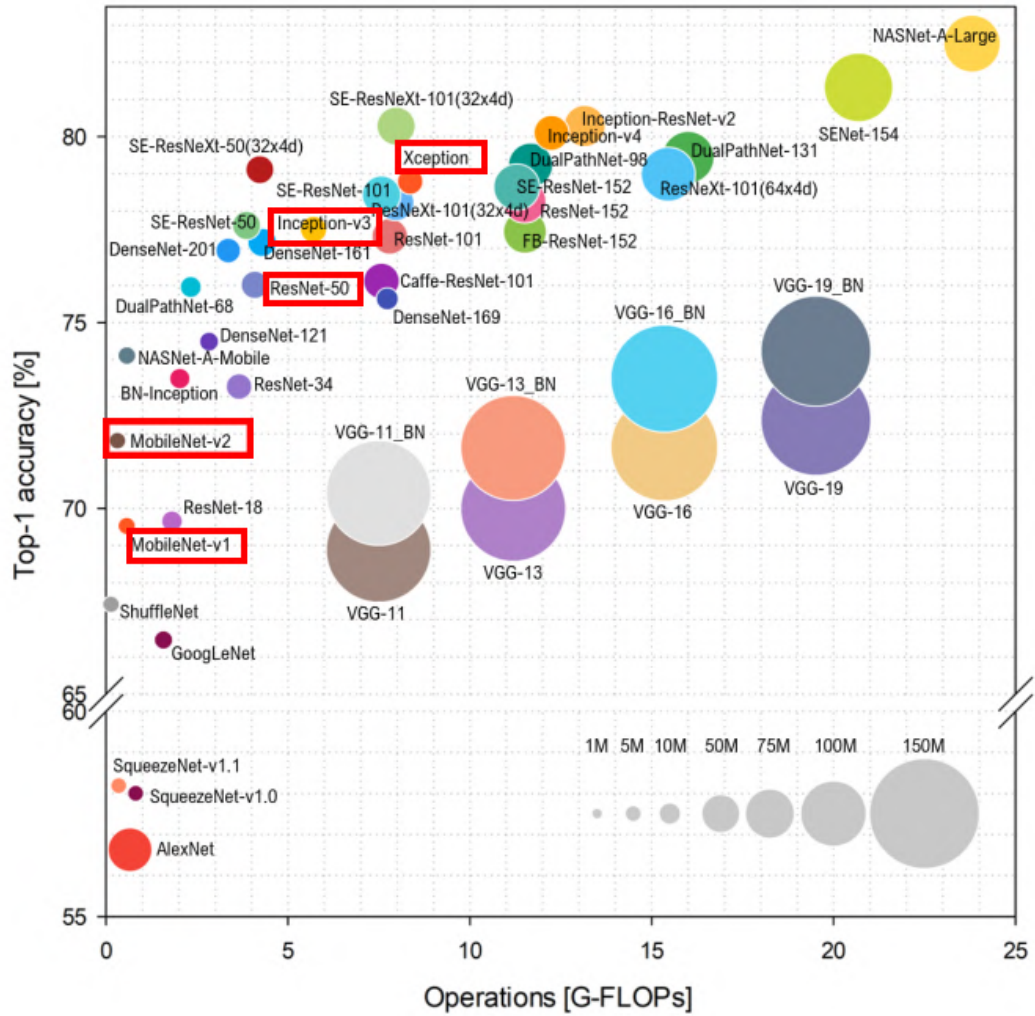


Figure 6.8: Ball chart reporting the Top-1 accuracy vs. computational complexity. The size of each ball corresponds to the model complexity (number of parameters). [27]

As you can see, MobileNet and MobileNetV2 are faster than the other models. This is because these are efficiency-oriented models, while the others are accuracy-oriented models. You can see on Table 6.1 that efficiency-oriented models have much fewer parameters than the other, leading to a reduction of the prediction time.

MobileNetV2	MobileNet	Xception	Resnet	Inception
5 145 154	5 853 890	22 961 706	23 591 810	23 903 010

Table 6.1: Total number of parameters of each architecture

The ball chart reported in Figures 6.9 shows the test accuracy on our test set with respect to the number of tiles predicted per second that the architecture can achieve. The ball size corresponds to the model complexity (number of parameters). The test set consists of a balanced dataset containing 6.000 tiles labeled "rail" and "non-rail".

Our trained models have a better accuracy on the test set than on the validation set. This is because the training+validation set has more mislabeled data. We removed the mislabeled data on the test set easily because we identified them after prediction. However, it is not worth the effort of reviewing the 55.000 examples that compose the training+validation set.

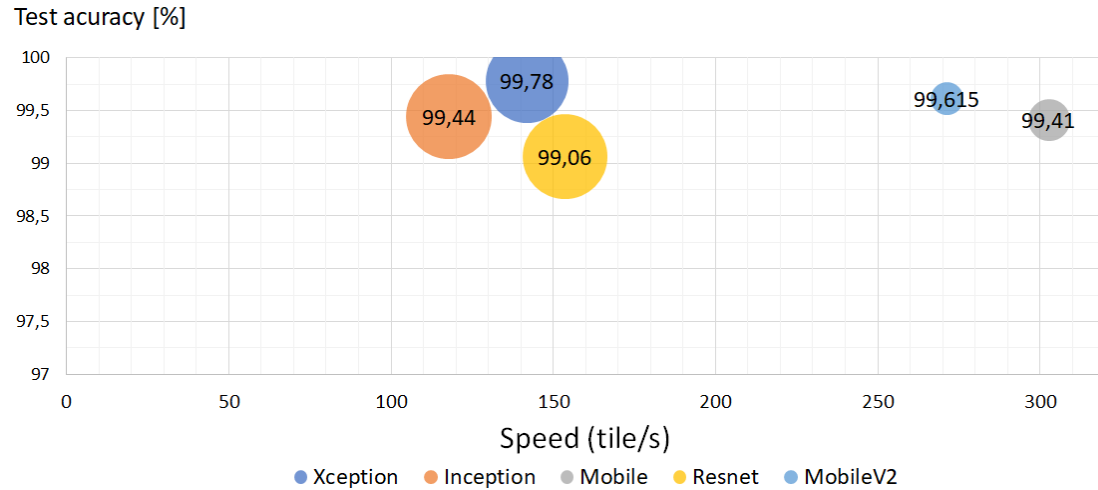


Figure 6.9: Ball chart reporting the test accuracy vs. tiles per second. The size of each ball corresponds to the model complexity (number of parameters).

The two models that stand out are Xception, on the accuracy-oriented side and MobileNetV2 on the efficiency-oriented side with respectively 99,78% and 99,65%. Choosing MobileNetV2 as final choice seems to be a good decision as it is much faster than Xception while staying very accurate. However, the speed of execution of the program is much more influenced by the number of requests sent to Google Maps API than by the number of predictions made by our model (see section 8.1.1). Thereby, having a faster but less accurate model does not make it more

time efficient. Indeed, the number of requests sent to Google Maps API increases when the prediction of a tile is wrong (see chapter 8).

Consequently, a simple comparison of the speed of the prediction is not enough, we must consider both models for the rail tracking algorithm and compare them with real tests.

6.3 Description of Used Architectures

Once again, this section does not claim to describe the full architecture of the used networks, but shows the key concepts brought by MobileNetV2 and Xception.

6.3.1 Xception

Xception was made by Google and published on 9 November 2017. Xception stands for “eXtreme version of Inception” as it uses the notion of depthwise separable convolution defined by Inception, slightly modified to give a better classification accuracy.

“Depthwise Separable Convolution” module consists of a depthwise convolution followed by a pointwise convolution as illustrated in Figure 6.10. In classic convolution, filters are applied on all channels simultaneously whereas in depthwise convolution, filters are applied independently on each channel and the output is then recombined.

This module leads to a reduction of the number of connections/parameters (and so the model size), without decreasing the network efficiency. The pointwise convolution is a 1x1 convolution and is used to change the dimension of the feature maps.

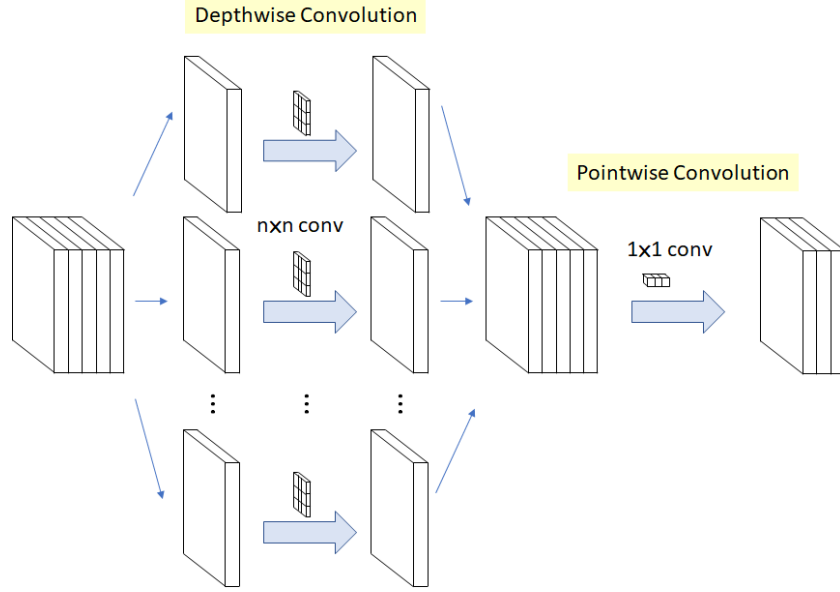


Figure 6.10: The depthwise separable convolution of Inception [28]

The two differences between the original depthwise convolution of Inception and the modified Depthwise Separable Convolution used in Xception are :

- The order of the operations: in Xception the pointwise convolution is performed before the depthwise convolution. The authors mention that this is a minor change as the modules are successively stacked in chained modules.
- The absence of non-linearity: in Xception there is no more ReLU function after the first operation, as it is the case in Inception.

The two combined changes give a better accuracy. Figure 6.11 shows the overall architecture of Xception. In this figure, SeparableConv is the modified depthwise separable convolution described above. The 1×1 convolutions on the left, called “residual connections”, are initially introduced by Resnet. In the Xception architecture, these shortcuts are essential in this architecture as removing these residual connections leads to a drop of accuracy.

For this task, we add on top of the last global average pooling layer, a fully-connected layer combined with a ReLU activation that outputs a vector of 1024 activation nodes. We then use a softmax layer to output the two class probabilities "rail" and "non-rail".

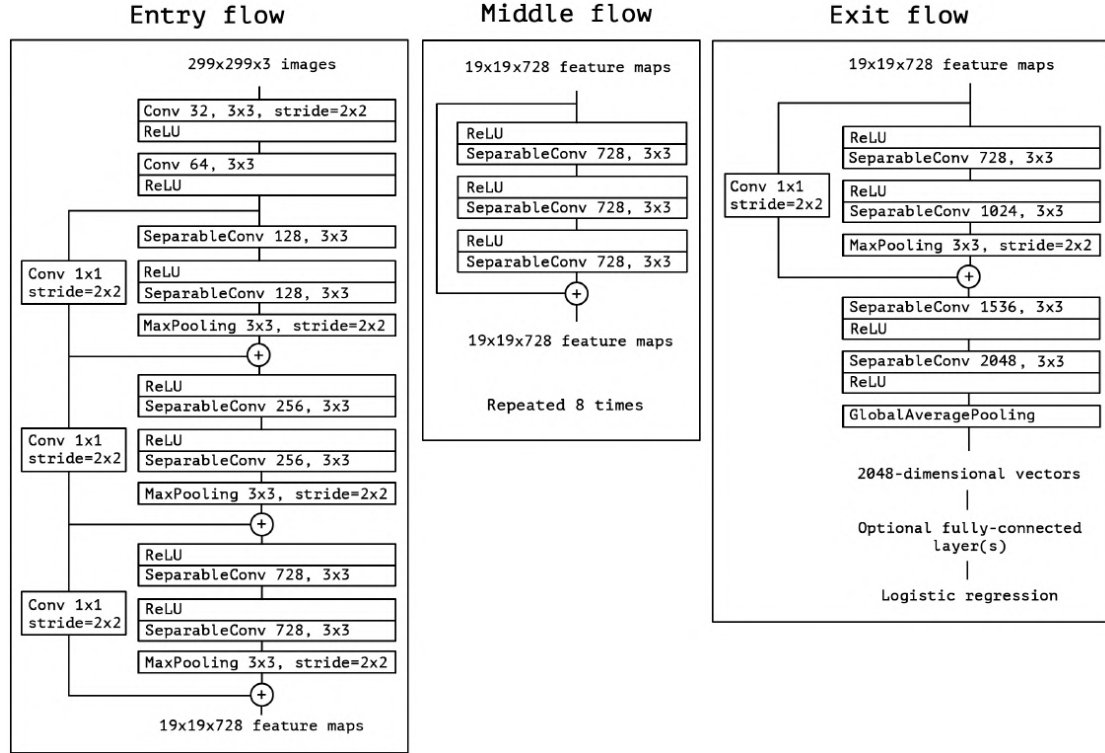


Figure 6.11: Overall architecture of Xception [28]

6.3.2 MobileNetV2

MobileNetV2 was also made by Google and was published on 17 December 2018. It is a direct improvement of MobileNetV1, where the depthwise separable convolution described in Xception is also used to reduce the complexity cost and the model size. MobileNetV2 uses inverted residual structure and remove the non-linearity in the layers.

MobileNetV1 uses the original depthwise separable convolution, while MobileNetV2 slightly changes the module, named “bottleneck” in this architecture. As illustrated in Figure 6.12, the pointwise convolution is applied before the depthwise convolution. A third layer is added, another 1×1 convolution but with a linear activation function. Authors mention that linear layers are crucial as it prevents non-linearity from destroying too much information. It then adds residual connections (shortcuts) between bottlenecks, enabling faster training and better accuracy.

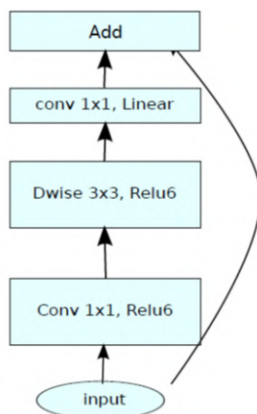


Figure 6.12: Bottleneck module in MobileNetV2 [29]

The architecture of MobileNetV2 contains one convolution layer, followed by 19 residual bottleneck layers and one global average pooling. MobileNetV2 is much faster in comparison to MobileNetV1. It uses half as many operations, has higher accuracy and needs 30 percent fewer parameters.

For this task, we add three fully-connected + ReLU layers on top of the last global average pooling layer. The first two layers output 1024 activation nodes while the last fully-connected layer outputs 512 nodes, these nodes are then connected to a softmax layer to make the final prediction. There are more FC layers with MobileNetV2 than with Xception, as we noticed that a simpler classification part leads to a slight accuracy drop.

6.4 Heatmap

6.4.1 Description of Heatmaps

Class Activation Map (CAM) makes our black box models more explainable by providing visualizations that support the predictions [30]. A method that uses CAM is called heatmap and is described in [31]. In a nutshell, it creates a 2D score grid associated to specific output class, computed for every location of an input image. This 2D grid reports, in our case, which regions in an image were relevant with respect to the rail class, for a specific layer.

Quite simply, it tells us which features the model extract and which regions of an image are used in the prediction. In CNNs, fully-connected layers loose the spatial information retained in convolutional layers, as the output of the last conv. layer is flattened (see Figure 6.7). Thus, we observe one of the last convolutional layers

since they have the best compromise between high-level semantics and detailed spatial information, as explained in [32].

6.4.2 Computation of the 2D Score Grid

We use a technique called Grad-CAM [32] to compute the 2D score grid associated with the presence of rail. Initially, we wanted to use CAM, but it works only when there is no additional fully-connected layers.

First, we compute $\frac{\partial y_c}{\partial A^k}$ which is the gradient of the score of class c (called y_c), with respect to feature map activations A^k of the layer we want to visualize. Basically, the gradient of a feature map is a 2D grid which tells how much we should increase the activations during the backpropagation steps to get a higher score y_c .

After obtaining these gradients, Equation 6.1 performs a global average pooling to compute a_c^k , the mean weight of each feature map k for the specific class c . In this case, c denotes the rail class. These weights capture the “importance” of feature map k for the target class c .

$$a_c^k = \overbrace{\frac{1}{Z} \sum_i \sum_j}^{\text{Global avg. pool.}} \underbrace{\frac{\partial y_c}{\partial A_{ij}^k}}_{\text{gradients}} \quad (6.1)$$

Then, once we obtain these weights, we perform a linear combination of the forward activation maps A^k , followed by a ReLU function (as shown in Equation 6.2). We are only interested in features having a positive influence on c , i.e. pixels whose intensity should be increased in order to increase y_c . Hence, the ReLU function is used to cancel features that are negatively correlated to the class of interest. Figure 6.13 shows the complete process in creating the 2D score grid.

$$L_{\text{Grad-CAM}}^c = \text{ReLU} \left(\underbrace{\sum_k a_c^k A^k}_{\text{linear combination}} \right) \quad (6.2)$$

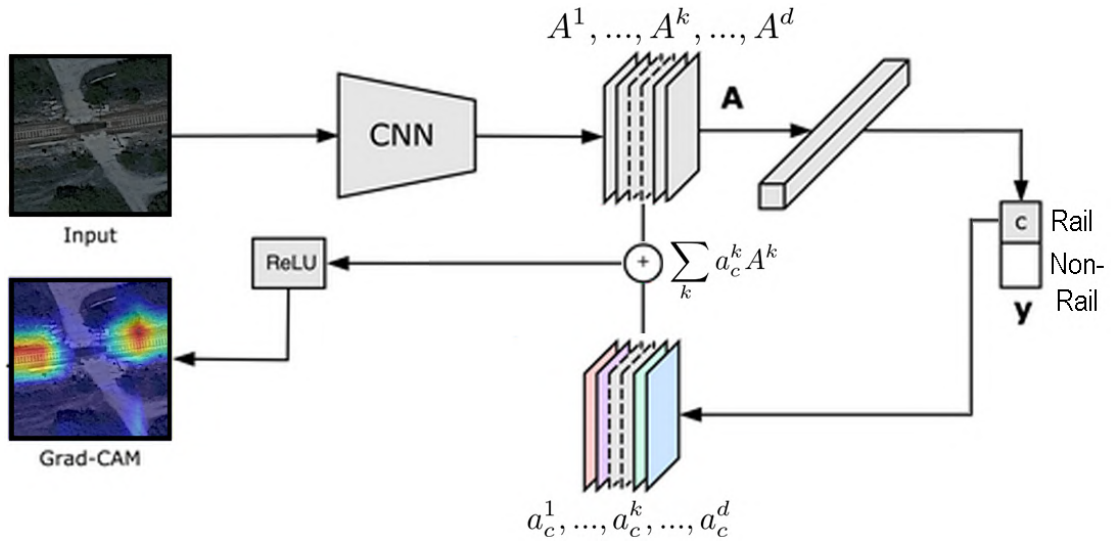


Figure 6.13: Steps in creating the 2D score grid using Grad-CAM.
Based on figure from [32]

This results in a heatmap with the size equal to the width and height of the analyzed feature maps. Eventually, this heatmap is upscaled using an interpolation function to obtain the final result having the same size as our input tile (256×256).

6.4.3 Results

Heatmaps are useful tools for debugging the decision process in convolutional neural networks. Indeed, it is interesting to see which features the model relies on to classify images. In the images below, the red region are the ones which were relevant to classify the tile as "rail". For the true positive examples, the region of relevant features should ideally only be the railway itself. Figure 6.14 shows true positive examples for Xception and MobileNetV2 models.

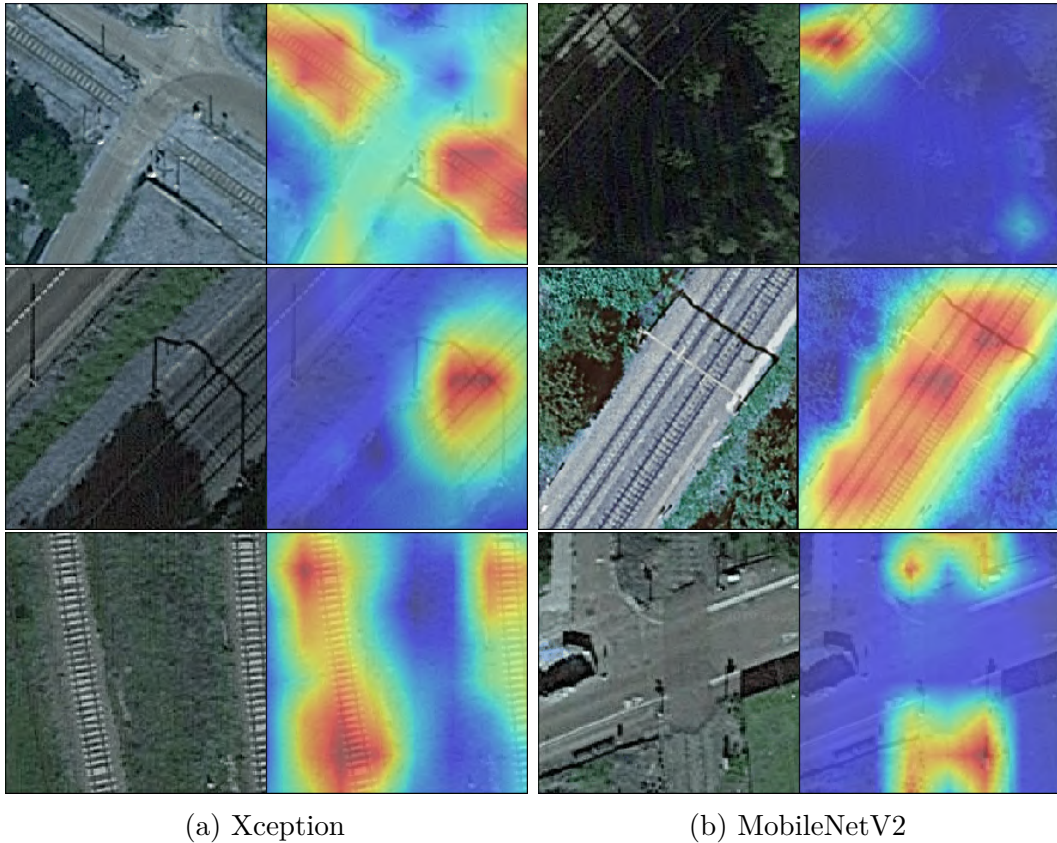
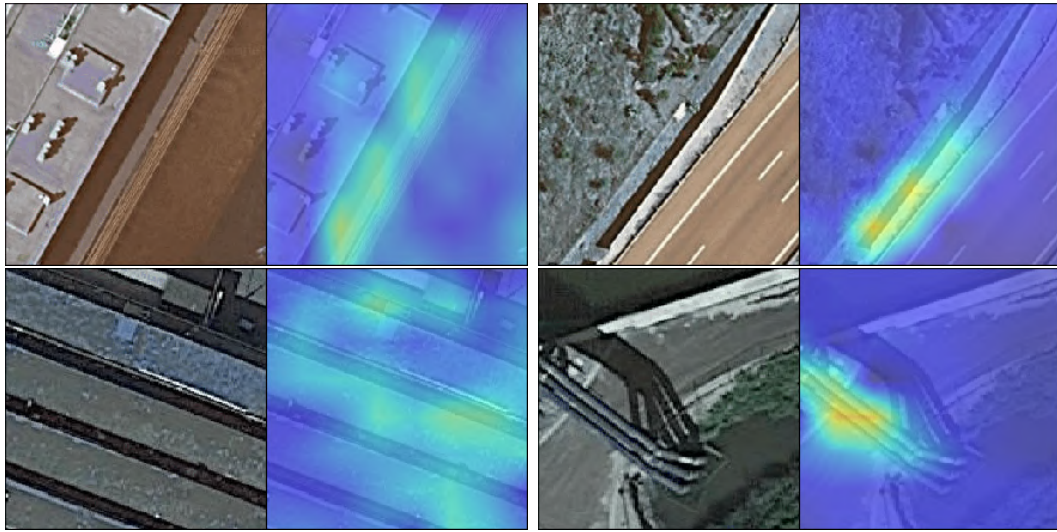


Figure 6.14: Heatmaps of true positive examples

It is even more useful for false positive examples where we see why the model made a wrong prediction. Figure 6.15 shows false positive examples. The two models detect patterns in the images that are close to the ones of rails, like straight lines, platforms or catenaries. These features are sufficiently significant for the models to predict the tile as "rail".

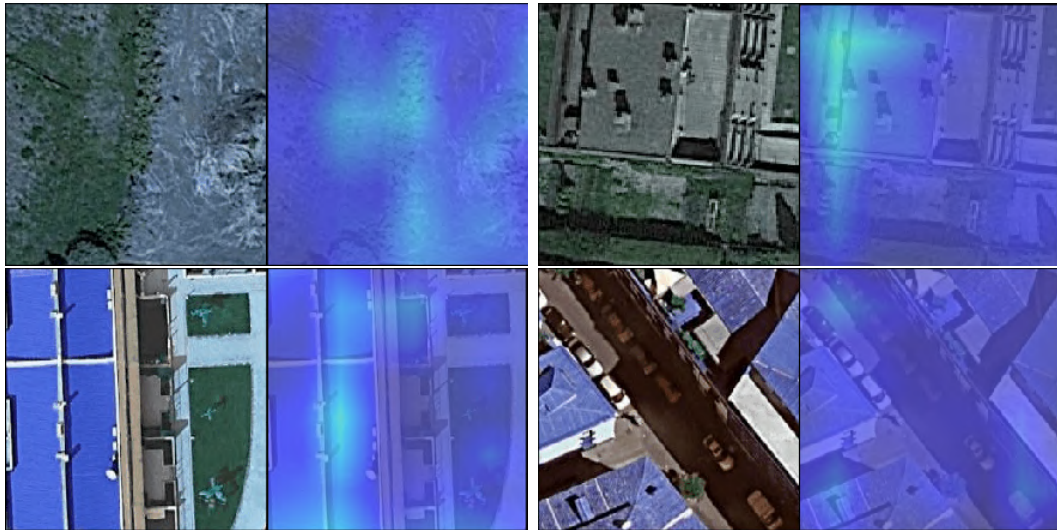


(a) Xception

(b) MobileNetV2

Figure 6.15: Heatmaps of false positive examples

Similarly, we can look at true negative examples to check which features are extracted by our models. In Figure 6.16, the green areas emphasize regions with straight lines or high contrast, but this time the features are not significant enough, so the model predicts the tile as a non-rail.



(a) Xception

(b) MobileNetV2

Figure 6.16: Heatmaps of true negative examples

Lastly, we can also observe what brought confusion when the classification lead to false negatives. In Figure 6.17, we observe two cases. We can see that Xception has some troubles to find rail patterns on the two tiles, while MobileNetV2 correctly localizes the features on these tiles, but it was probably considered too insignificant.

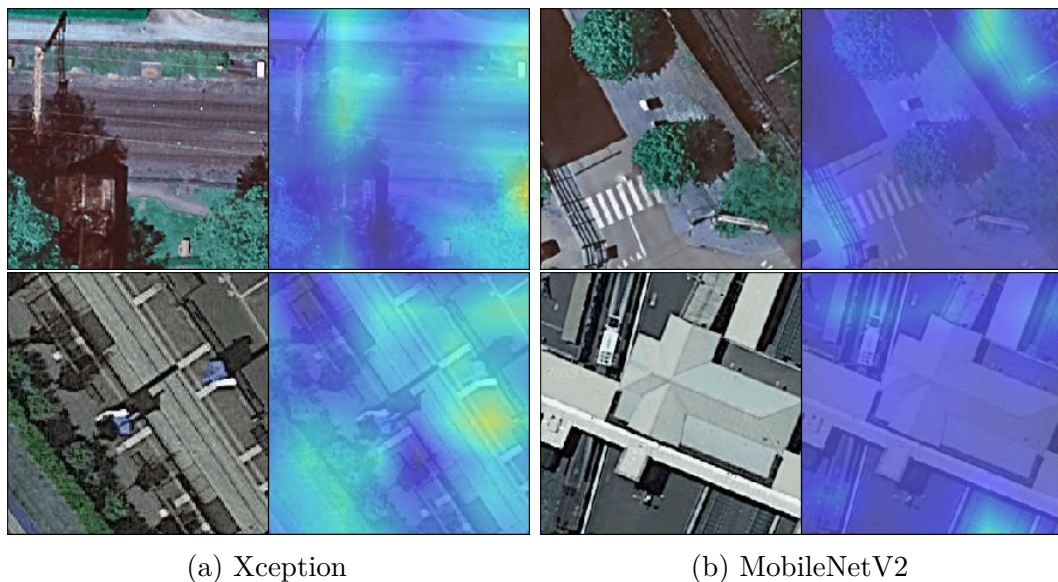


Figure 6.17: Heatmaps of false negative examples

6.4.4 Heatmap for Object Localization

Initially, we wanted to localize the rails on a tile by looking for the most relevant region on the associated heatmap, and thus make object localization without having to label the rails position in the training set (weakly-supervised object localization). However, this technique has sometimes a lack of accuracy, as shown in Figure 6.18

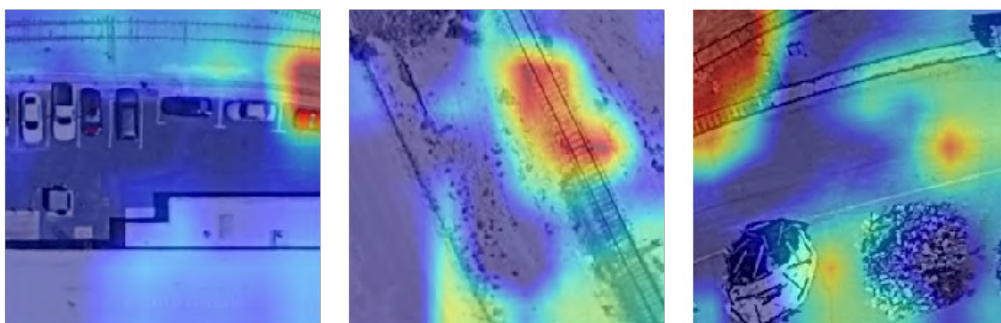


Figure 6.18: Heatmaps of true positives with a lack of accuracy

Hence, even though heatmaps are used for object localization for some tasks, we chose to use them as a debugging tool. Thanks to those visuals, we can ensure that our models are truly looking at rails. To localize the rails accurately on the tiles, we developed another approach explained in subsection 8.1.1.

6.5 Training Phase

Many training configurations have been tested during the training phase. This section describes only those who gave the best accuracy on the validation and test set. Both models use the Adam Optimizer [33] and ImageNet for transfer learning (see section 6.2.1).

6.5.1 Xception

For Xception, we trained the first 40 iterations with 10^{-3} learning rate, and use 10^{-6} for the last 10 last iterations. The Xception model converged at iteration 50, this is why we stopped the training. Figure 6.19 shows the learning curve of the Xception training phase.

Adam updates any parameter with an individual learning rate, meaning that every parameter in the network has a specific learning rate associated. The learning rate that we provide to the optimizer, called λ , is used as an upper limit in Adam, every single learning rate can vary from 0 (no update) to λ (maximum update). We noticed that the computed loss stopped decreasing at iteration 40 with $\lambda = 10^{-3}$. So we lower λ to ensure that every update step does not exceed $\lambda = 10^{-6}$, this helps to decrease the loss during the last steps of training.

6.5.2 MobileNetV2

For MobileNetV2, we trained the first 60 iterations with 10^{-3} learning rate, and use 10^{-6} for the last 10 iterations. After looking at the heatmaps of the false negative examples, we saw that the feature extraction part detected the relevant region correctly, but the classification layers made a wrong prediction. We thus trained only the classification part for 5 iterations with 10^{-3} learning rate and 5 iterations with 10^{-6} learning rate. This process is known as “fine-tuning”. After that, the classification layers made the right choice. Figure 6.20 shows the learning curve of the MobileNetV2 training phase.

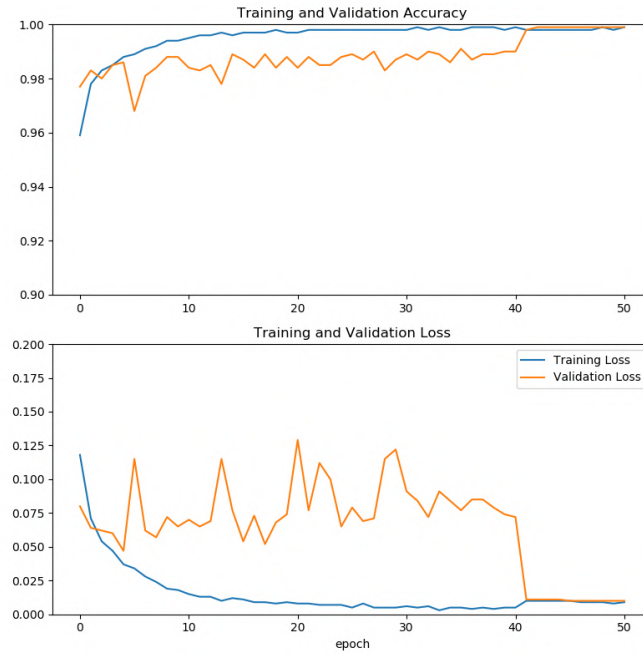


Figure 6.19: Learning curves of the Xception training phase, according to the training and validation accuracy and loss

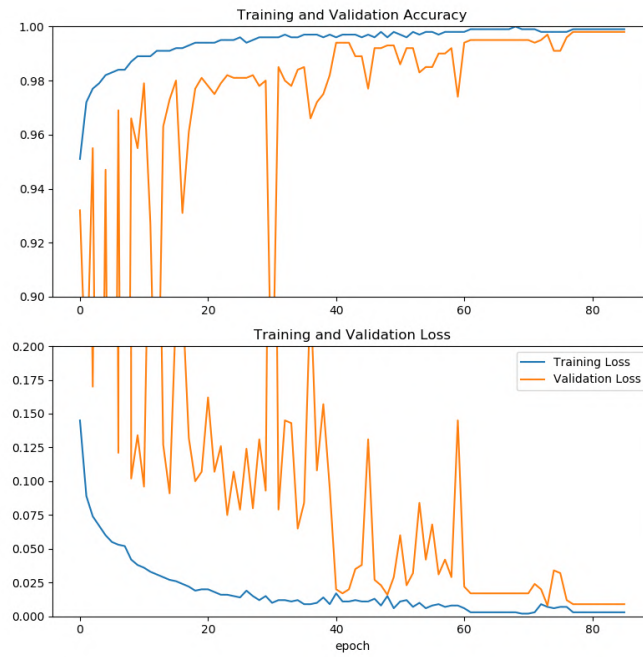


Figure 6.20: Learning curves of the MobilenetV2 training phase, according to the training and validation accuracy and loss

Chapter 7

Area Detection

In this chapter, we explain a first way to test our trained model for rail recognition by testing the model on a small area crossed by a railway track. This area is described by a grid of x by x tiles. Firstly, we explain how to get the precise coordinates of the neighboring tiles of an area, vertically and horizontally. Next, we extend it by finding the next tile given an azimuth, which will be useful for rail tracking. Finally, we show the results of the detection in some interesting areas.

7.1 Neighboring Tiles

Given a latitude and a longitude expressed in degrees, the problem is to get the precise coordinates of the next tile when traveling towards a direction. When moving northwards, the latitude increases (0° is the equator, 90° is the North Pole), while when moving eastwards, the longitude increases (0° is Greenwich, 180° is the International Date Line, or IDL for short). The aim is to get two tiles where the borders are juxtaposed, as shown in Figure 7.1.

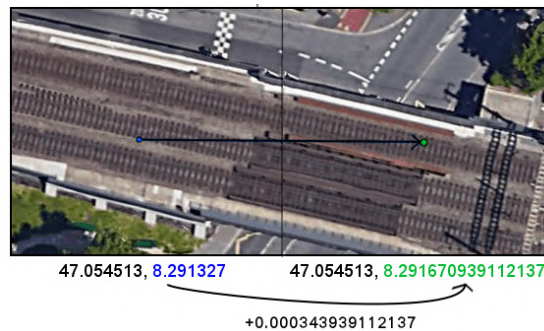


Figure 7.1: Two juxtaposed tiles obtained when moving eastwards

To understand those shifts, it is important to know that all major map services like Google Maps, Bing Maps, OpenStreetMap are based on the Mercator projection, but used a variant called “Web Mercator Projection” [34]. Just like the original projection, a distortion is created as we go further away from the equator. If we look at Figure 7.2, we see that Greenland is approximately as big as Africa while it is about 14 times smaller in reality [35].

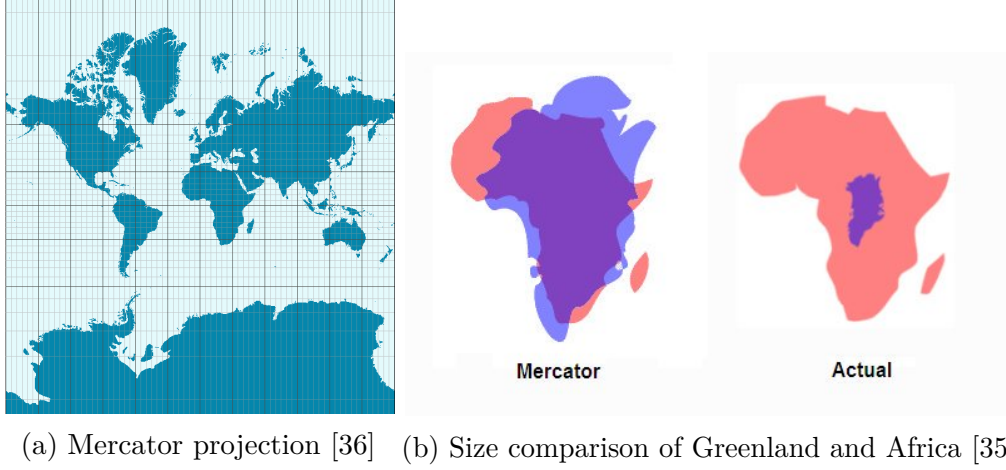


Figure 7.2: Side effect of Mercator projection

Traditionally, the Mercator projection is made on a sheet of paper, where the scale is expressed in kilometers by centimeter, for instance. On a screen, it is expressed in kilometers by pixels, and this scale varies with the zoom.

Regarding the projection in Figure 7.2a, we see that every longitude are separated by the same number of pixels. It is not the case of the latitudes, the interval between two parallel increases. For this reason, we use two different functions to determine the distance between two tiles, as reported in Equations 7.4 and 7.3.

$$MPP_{long} = \frac{MPP_0}{2^{zoom}} \quad (7.1)$$

$$MPP_{lat} = \frac{MPP_0 \times \cos(latitude)}{2^{zoom}} \quad (7.2)$$

$$distance_{long} = MPP_{long} \times 256 \quad (7.3)$$

$$distance_{lat} = MPP_{lat} \times 256 \quad (7.4)$$

MPP_0 is equal to 156.543,03392 m/px. It corresponds to the number of meters per pixel at zoom level 0 (the entire Earth) on a projection of 256 by 256 pixels.

Indeed, $MPP_0 \times 256px = 40.075.017 \text{ m}$, the Earth's circumference. The latitude, originally in degrees, is converted in radians. We simply have to multiply MPP_{lat} or MPP_{long} by the number of pixels of the shift (in our case 256) and we obtain the distance between the center of two tiles when moving northwards or eastwards on the projection.

Therefore, the horizontal shift is constant as the longitudes are equally separated (the North Pole is everywhere on top of the projection) while the vertical shift decreases when moving towards the poles, since the represented areas are not so large in reality.

Nevertheless, a tile of 256×256 pixels has always the same distance between each opposite edge, but this distance changes in accordance with the latitude. Otherwise, the tiles would be completely distorted in Scandinavia. Figure 7.3 shows this difference in some locations.

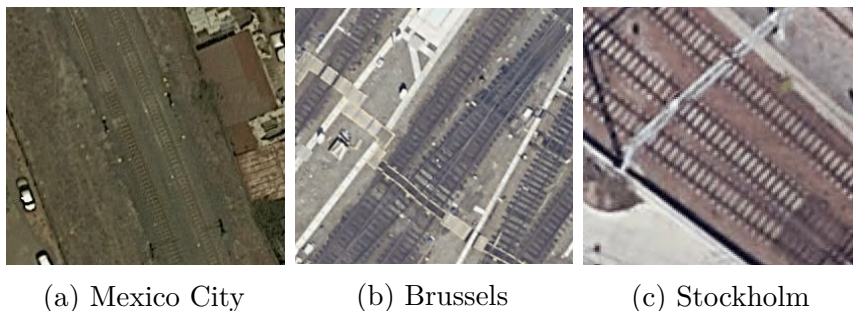


Figure 7.3: The distance between each opposite edge varies with the latitude

Note that our dataset is composed of images from different latitudes. Thus, this difference of perspective is taken into account during the training of our model. Table 7.1 summarizes these explanations in some locations.

Previous computation gives us the distance in meters on the projection. However, we still need the new latitude and longitude to get the next tile. To convert the distance in degrees, the following formula is used:

$$\Delta Coord = \frac{distance}{K} = \frac{distance}{1852 \times 60} \quad (7.5)$$

This translation is straightforward, knowing that 1 arc minute is equivalent to 1 nautical mile, which is equal to 1852 meters. This has to be multiplied by 60 to get the value in degrees [37].

Location	Latitude	MPP	Meters	Ratio
Equator	0	0,14929	38,22	100%
Mexico City	19,4	0,14081	36,05	94,32%
Tropic of Cancer	23,43671	0,13697	35,06	91,75%
Brussels	50,8	0,09436	24,15	63,21%
Stockholm	59,3	0,07622	19,51	51,05%
Arctic Circle	66,56333	0,05937	15,20	39,77%
North of Greenland	83,6	0,01664	4,26	11,15%

Table 7.1: Table showing the difference of meters per pixel in function of the latitude. The column “meters” corresponds to the distance between each opposite edge of the tile.

For instance, the example of Figure 7.1 is computed as follows:

$$MPP_{long} = \frac{MPP_0}{2^{zoom}} = \frac{156543,03392}{2^{20}} = 0,14929 \quad (7.6)$$

$$\Delta Coord = \frac{MPP_{long} \times 256}{K} = \frac{0,14929... \times 256}{1852 \times 60} = 0,0000343939112137 \quad (7.7)$$

$$New\ long = 8,291327 + \Delta Coord = 8,291670939112137 \quad (7.8)$$

Using equation 7.2 instead of 7.1, the same procedure can be used to get the new latitude of the neighboring tile. Obviously, inverting $\Delta Coord$ allows us to move towards the south and the west.

7.2 Neighboring Tiles Given an Azimuth

We now extend the problem of finding the neighboring tiles when we have to follow an azimuth as illustrated by Figure 7.4. An azimuth is the angle in the horizontal plane between the direction of an object and a reference direction. In this work, the reference direction is the north.

This is actually simple trigonometry. We quickly notice that whatever the azimuth, we have to move 256 pixels either vertically or horizontally to get the next tile. Then, for the other direction, we compute the length of the black arrow, which is the hypotenuse, using the following formula (the angle being in radians):

$$Length_{hypo} = 256 / \min[\cos(angle), \sin(angle)] \quad (7.9)$$

Finally, using Pythagorean theorem, we get the length of the red or purple arrow that we were looking for. This number of pixels is then used to multiply MPP_{lat} or MPP_{long} to get the next tile as explained in section 7.1. This extension is useful to follow the railway track, which is explained in the next chapter.



Figure 7.4: Two juxtaposed tiles obtained when moving according to an azimuth

7.3 Results

In this section we discuss the results of our railway detection in some interesting areas with Xception, the most accurate model we obtained in section 6.2.2. For better visualization, the images are available in appendix A. The blue tiles are the ones where rails are detected and the red numbers represent the confidence in the prediction. Those images were reconstituted after that the prediction of each tile was made independently.

For every location, the images come by pair. The first image of each pair is the result with the default threshold for binary classification, that is 0,5. In these cases, we notice that there are several false positives.

To get rid of them, we can choose to increase the threshold to choose when a tile is classified "rail". The second image of each pair shows the result when increasing this threshold to 0,999. Most of false positives become true negatives. Unfortunately, some true positives become false negatives, as best illustrated by Figure A.3. Those are the ones that are hardly recognized because they are partly masked by a train, for instance.

Nevertheless, in our case, it is preferable to have false negatives than false positives as we will see in chapter 8. Thus, we choose to keep the threshold of 0,999 to classify a tile as "rail".

7.4 Limitations

Unfortunately, we observe that it is very costly to load so many tiles (400 tiles for each image of appendix A) in comparison to the number of tiles that contain rails and the area covered. Indeed, if we had to find a railway line between two cities, the area would be much larger and we would test a huge number of tiles with no rail. In the next chapter, we will focus on rail tracking, i.e. following the railway track while minimizing the number of tiles downloaded and predicted.

Moreover, it is not only costly in terms of time, since Google and other services charges in proportion to the number of requests. That is why it is also important to minimize this number from an economic perspective.

Chapter 8

Rail Tracking

In this chapter we describe the second phase of our work. Even though the area detection explained in the last chapter can be helpful to detect multiple railway lines in a region, it is very costly in terms of number of requests. The procedure introduced in this chapter tries to minimize this number by tracking the rails between two locations. The algorithms explained in this section do not depend on the classification model used. The model is only used as a tool to detect whether there are rails. Of course, when results will be given at the end of this chapter, we used the best models we obtained.

8.1 Main Method

The main method is reported in Algorithm 1, it consists of starting from the origin (*org* in the algorithm) location and discover new rail coordinates at each iteration of the main loop, until reaching the destination (*dest*) location. To find the next coordinate, we use a method named *EstimateNextPoint* that estimates (on line 4) the next rail location to analyze, based on the previous rail coordinates discovered. This method returns the coordinate of the center of the tile where the rail is supposed to appear. Section 8.2 describes two techniques to make good estimations. The variable *speed* corresponds to the number of tiles that are skipped when estimating the next point. It is used to improve the performance (see details in section 8.1.2).

If the estimation is right and the *Detect* method used on line 5 localizes the rail on the estimated tile, we retrieve the position of the rail (line 6) and add it to the list of rail coordinates (line 31) and we carry on with the next iteration. Section 8.1.1 explains how the position of the railway is identified accurately thanks to the method named *EstimateNextPoint*.

In case of wrong estimation (line 8), we compute the perpendicular of the angle between the last detected rail point and the wrong estimation and look for the railway on this perpendicular (line 9). This process is described in section 8.1.3.

If no rail has been found (line 21), we enter in “discovery state” to find the track further (in case where the track is hidden by something). If this procedure is unsuccessful, we perform a rollback by entering the condition on line 24. More details on the discovery state and the rollback process are available in subsection 8.1.4.

Algorithm 1: Railway Tracking

Data: $org, dest$ the start and end of the railway to track

Result: A sequence of coordinates $R = x_1, \dots, x_n$ from org to $dest$

```
1  $R \leftarrow \{org\}, \quad x_{rail} \leftarrow org$ 
2  $discovery_{count} \leftarrow 0, \quad speed \leftarrow speed_{min}$ 
3 while  $dest \notin R$  do
4    $next \leftarrow EstimateNextPoint(R, speed)$  // Section 8.2
5   if  $Detect(next) \neq \emptyset$  then // Section 8.1.1
6      $x_{rail} \leftarrow Detect(next)$ 
7      $speed \leftarrow \min(speed * \gamma, speed_{max})$ 
8   else
9      $perp \leftarrow \perp$  of angle( $x_{rail}, next$ ) // Section 8.1.3
10    for  $i = 1$  to  $range_{max}$  do
11       $next_{left} \leftarrow$  the  $i^{th}$  tile on the left of  $next$  according to  $perp$ 
12       $next_{right} \leftarrow$  the  $i^{th}$  tile on the right of  $next$  according to  $perp$ 
13      if  $Detect(next_{left}) \neq \emptyset$  then
14         $x_{rail} \leftarrow Detect(next_{left})$ 
15         $speed \leftarrow \min(speed_{max}, \max(speed_{min}, speed * (\rho/i)))$ 
16        break
17      else if  $Detect(next_{right}) \neq \emptyset$  then
18         $x_{rail} \leftarrow Detect(next_{right})$ 
19         $speed \leftarrow \min(speed_{max}, \max(speed_{min}, speed * (\rho/i)))$ 
20        break
21  if No rail has been found in this iteration then // Section 8.1.4
22     $discovery_{count} \leftarrow discovery_{count} + 1$ 
23     $speed \leftarrow speed_{discovery}$ 
24    if  $discovery_{count} \geq discovery_{max}$  then // Section 8.1.3
25      if  $x_{rail}$  is too close to last rollback location then
26        break
27       $speed \leftarrow speed_{min}$ 
28       $discovery_{count} \leftarrow 0$ 
29       $R \leftarrow R \setminus$  Last three points
30  else
31     $R \leftarrow R \cup x_{rail}$ 
32     $discovery_{count} \leftarrow 0$ 
```

8.1.1 Detection of the Railway on a Tile

In this subsection, We describe how the *Detect* method works on lines 5, 14 and 16 of Algorithm 1.

After some observations, we notice that the speed of execution is much more influenced by the number of requests sent to Google Maps API than by the number of predictions of our model. From this conclusion, we improve the performance by decreasing the number of network requests. So instead of requesting tiles of 256×256 from Google, the program requests the maximum image size (620×620) and split these tiles in five 256×256 tiles. This is illustrated in Figure 8.1a and done on lines 1 and 2 of Algorithm 2.

This algorithm describes the process of detecting and localizing the rail in the 620×620 tile. In addition to decreasing the number of predictions, requesting larger tiles increases the area covered by one prediction and thus increases the speed of the rail tracking.

Algorithm 2: Rail Detection Method	
Data:	x the center of the 620×620 tile where the rail has to be detected
Result:	The coordinate of center the rail, $\emptyset$ if no rail has been detected
1	$Image \leftarrow ImageFromGoogle(x)$ // 620×620 size
2	$tiles \leftarrow$ split Image into five 256×256 tiles // Figure 8.1a
3	$R \leftarrow \{\}$
4	for $t \in tiles \wedge Class(t) \in rail$ do // Figure 8.1b
5	$sub_tiles \leftarrow$ split t into 128×128 tiles // Figure 8.1c
6	for $s \in sub_tiles \wedge Class(s) \in rail$ do
7	$R \leftarrow R \cup s$;
8	if $R \neq \{\}$ then
9	return $MeanCoordinate(R)$ // Figure 8.1d
10	else return $\emptyset$

It uses the CNN classifier on line 4 to predict tiles that contain rails from the five 256×256 images (see Figure 8.1b). On line 5, we split again each tile classified as "rail" in four 128×128 sub-tiles and classifies (see Figure 8.1c). Finally, we compute the mean coordinates of the center of each sub-tile classified as "rail" (on line 9). Since we take the center of the tile to refer to the position of the railroad, dividing each tile into four 128×128 sub-tiles increases the location accuracy, as illustrated in Figure 8.1d.

We can compute the number of predictions made for one 620×620 image. At most, the model has to make $5 + (4 \times 5) = 25$ predictions if it detects a rail on the five 256×256 tiles. At least, it needs 5 predictions if it detects no railway.

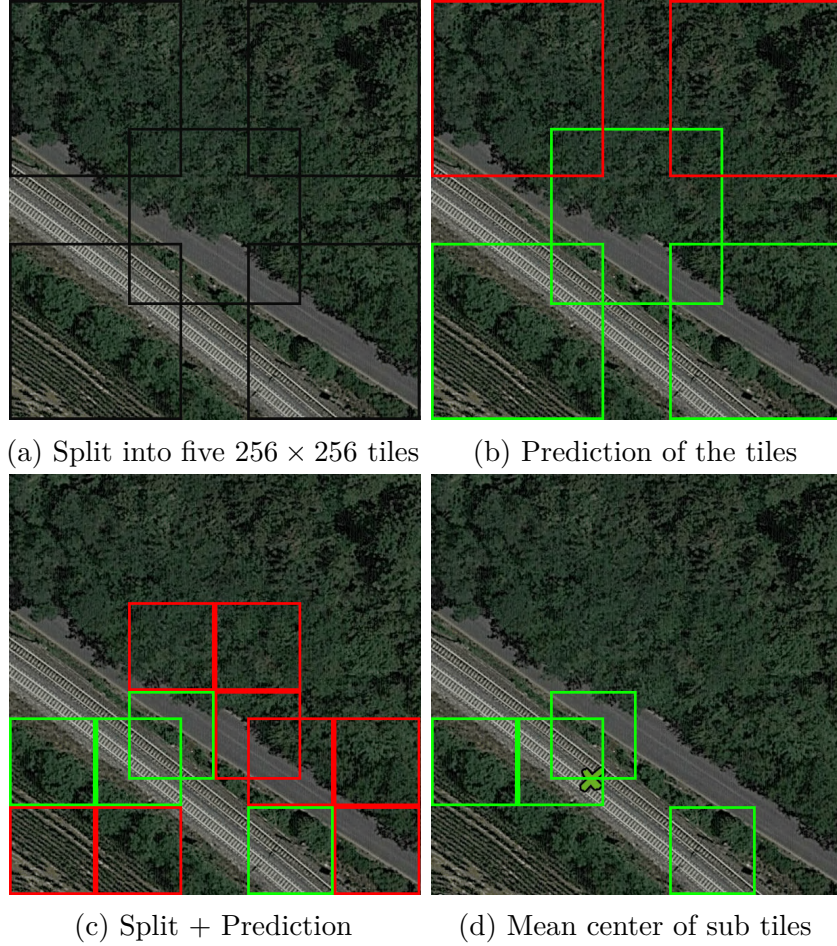


Figure 8.1: Main steps of the rail detection on a 620×620 tile

8.1.2 Speed Factor

The variable *speed* in Algorithm 1 corresponds to the number of tiles that are skipped when estimating the next point. This has been implemented to improve the performance, as tracking the railway tile by tile would be time and cost inefficient. It would result in many requests and many points that should be cleaned afterwards.

Therefore, when the route of the railway is simple (for example, a straight line) the speed increases as the next point is easy to estimate. However, the speed

decreases when the route is harder to determine (for example, a tight curve) to reduce the probability of missing the next estimation. Figure 8.2 shows the gap between the estimated points in accordance with the speed.

This idea is inspired by algorithms like Nelder-Mead [38] or the congestion control of TCP protocol [39]. The NM algorithm is commonly used to find the optimum of an objective function. There is an “expansion” phase where the simplex is expanded in one direction to take larger steps thanks to a factor γ (where $\gamma > 1$). When it has to “pass through the eye of a needle”, the simplex is contracted thanks to a factor ρ (where $0 < \rho \leq 0,5$).

Along the same lines, the idea behind the congestion control of TCP is to increase the transmission speed when there is no congestion until it reaches a defined maximum. When congestion is detected, the speed drops and increases gradually until the maximum is reached.

Similarly, on line 7 of Algorithm 1, we increase the speed by a factor γ (where $\gamma > 1$) when the predictions are correct. On lines 15 and 19, the further the estimation was from the railway, the more the speed is decreased, using a factor ρ (where $1 < \rho < 2$). When it rollbacks (explained in subsection 8.1.4), the speed drops to its minimum by entering the condition on line 24.



Figure 8.2: Evolution of the speed according to the railway route

8.1.3 Perpendicular Tracking

In this subsection, we describe the lines 9 to 18 of Algorithm 1, when the estimation is wrong, meaning that no rail has been detected on the tile.

In this case, we compute the perpendicular of the angle between the last detected point and the wrong estimation (line 9). This allows us to find the railway again on both sides of the wrongly estimated point (line 11 and 12). Figure 8.3 illustrates this process. The orange mark, the green tile and the green mark

correspond respectively to the wrong estimation, the neighboring tile containing the railway and the position used for the next iteration. The distance between the first estimation and the perpendicular tile depends on the variable i on line 10. This variable increases at each iteration of the for loop until a rail is detected on one of these perpendicular tiles (lines 13 and 17). We then exit the for loop on lines 16 or 20 and finally add the position to the list of rail coordinates (line 31) and continue with the next iteration.



Figure 8.3: Example of the perpendicular tracking

8.1.4 Wrong prediction and Hidden Rail

The method described so far has some limits. We must set a maximum range (named $range_{max}$ on line 10 of Algorithm 1) when tracking the rail on the perpendicular. This avoids searching the rail unnecessarily when the estimation is likely to be wrong. For example, when our classifier does not find the rail in a tile (false negative), or more frequently when the estimation of the next rail position is right but it is hidden by something. This is illustrated by Figure 8.4.

In case where no rail was found in one iteration, the algorithm should look further to find the rail again. This is why the algorithm does not halt when no rail is found. Instead, it decreases the speed (line 23) and starts the next iteration as before by estimating a new point further and keep looking for rails on the perpendicular. We named this phase “discovery state” and we leave this state as soon as the railway is found or when the number of discovery iterations exceeds a defined threshold (named $discovery_{max}$ on line 24). Increasing this threshold helps to deal with long tunnel cases but also increases the risks of following the path of another railway line.



Figure 8.4: Example of a case handled by the discovery state

The program must also handle false positive cases (when it wrongly detects rails in a tile). Figure 8.5a shows an example of this case, the green tile is the false positive: the CNN classifier detects a rail because of the pattern of the building roof. This induces a bias in the estimation of the next points. The algorithm stops the discovery state after that the number of discovery iterations in a row exceeds the threshold ($discovery_{max}$). It rollbacks by removing the last three rail points found (line 29) and set the speed to $speed_{min}$ (line 27) in order to ease the estimation of the rail position for the next iterations, as shown in Figure 8.5b.

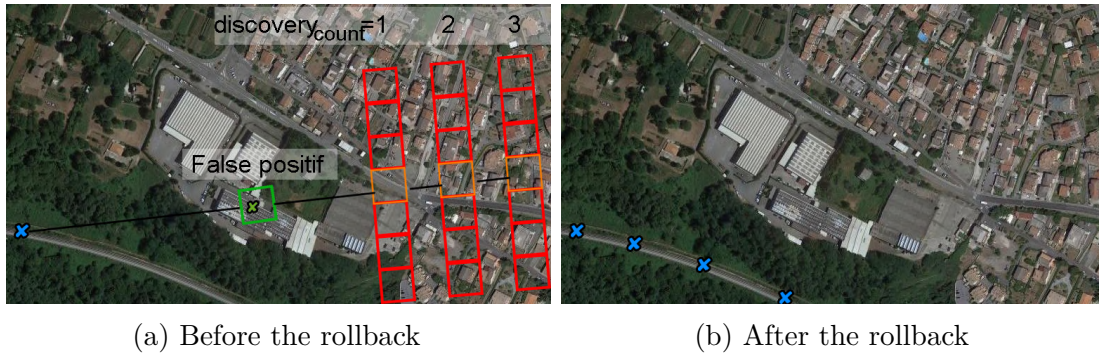


Figure 8.5: Example of a false positive, before and after the rollback

We now see why it is preferable to have more false negatives than false positives. First, a false positive could easily bring the program to another close railway line.

Also, the number of requested tiles is much larger when dealing with a false positive ($discovery_{max} \times range_{max}$ requests made before rolling back). In case of false negatives, usually one or two iterations suffice to find the track again.

8.1.5 Halting Cases

Finally, the program stops after it encounters three different situations. Firstly, when reaching the destination point provided by the user. Secondly, when the program rollbacks too close to the last roll back position (line 25). This case means that something covers the rail over a very long distance and the program is unable to find the railway in discovery mode, as illustrated in Figure 8.6.



Figure 8.6: Example of long tunnel where the end is not reached in discovery state

Last but not least, the program is stopped when it deviates too far from the destination (according to the minimum distance reached). This often means that it took the wrong path and that it will never reach the destination. Section 8.3 explains one solution to deal with this issue.

8.2 Estimation of the Next Point

This section describes two techniques designed to estimate the next position of the railroad as accurately as possible.

8.2.1 Weighted Average Angle

The first technique is very simple. We evaluate the direction that should be followed by the railway according to the previous rail positions. Next, we fetch the next tile by using the technique described in section 7.2, using the new azimuth.

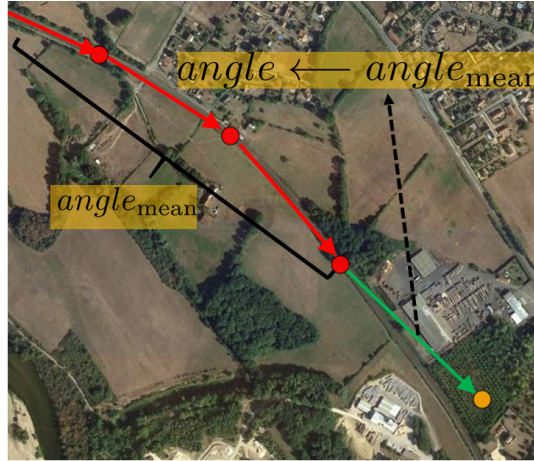
At each iteration, we update the angle with a weighted average between the old value of the weighted average and the new angle formed by the last point found and the new rail point. It is shown by Figure 8.7 and Equation 8.1 where α is the update coefficient. $\alpha = 0.7$ shows good result in our tests.

$$angle_{mean} \leftarrow \alpha \cdot angle(R.last, x_{rail}) + (1 - \alpha) \cdot angle_{mean} \quad (8.1)$$



(a) At the start of the iteration x , $angle_{mean}$ is based on the angles between last points in red. It is used to estimate the next point in orange.

(b) At the end of the iteration x , $angle_{mean}$ is updated based on the angle between the last point of R and the new rail point. It will be used in the next iteration.



(c) At iteration $x + 1$, the next point is estimated with the updated $angle_{mean}$

Figure 8.7: Steps of weighted average angle

The weighted average angle approach shows incredible performance in comparison to its simplicity. However, it is quite empirical. That is why we try another approach which is more mathematically founded to see if it could increase the estimation precision and thus reduce the number of requests.

8.2.2 Quadratic Lagrange Interpolating Polynomials

The Lagrange polynomial is defined as follows: “For a given set of points (x_j, y_j) with no two x_j values equal, the Lagrange polynomial is the polynomial of lowest degree that assumes at each value x_j the corresponding value y_j , so that the functions coincide at each point.” [40]

Therefore, thanks to three points, it is possible to obtain a quadratic function that can approximate a curve (or a straight line) between those points. We can thus use the same function to estimate the position of the next point, given one part of the coordinate (in our case, the latitude or the longitude) and compute the other part. We could use more than three points and obtain a more complex function, but it would not necessarily help to estimate the current curve of the railway because the further the points are from the new point, the more they can be irrelevant. Indeed, a railway curve does not depend on the ones behind it. Algorithm 3 describes this process of estimation and is reported further in this section.

The coefficient of the quadratic function $f(x) = ax^2 + bx + c$ can be computed using a system of three equations, given three coordinates. Let’s take an example with three basic coordinates. The same procedure can be made with geographical coordinates.

$$E = (-1, -2) \quad F = (1, 4) \quad G = (-2, 1)$$

$$\begin{cases} f(-1) = -2 \\ f(1) = 4 \\ f(-2) = 1 \end{cases} \Leftrightarrow \begin{cases} a - b + c = -2 \\ a + b + c = 4 \\ 4a - 2b + c = 1 \end{cases}$$

This second system can be transformed into matrices. The value of the coefficient is then obtained provided that the matrix A is invertible, which is always the case when three distinct points are provided.

$$\begin{bmatrix} 1 & -1 & 1 \\ 1 & 1 & 1 \\ 4 & -2 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} -2 \\ 4 \\ 1 \end{bmatrix} \Leftrightarrow AX = B \Leftrightarrow X = A^{-1}B$$

$$\Leftrightarrow \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 1 & -1 & 1 \\ 1 & 1 & 1 \\ 4 & -2 & 1 \end{bmatrix}^{-1} \begin{bmatrix} -2 \\ 4 \\ 1 \end{bmatrix} \Leftrightarrow \begin{cases} a = 2 \\ b = 3 \\ c = -1 \end{cases}$$

After having computed the quadratic function that passes through the last three points, we are now able to compute the latitude given the longitude with these following equations:

$$new_{long} \leftarrow last_{lng} + shift \quad (8.2)$$

$$new_{lat} \leftarrow f(new_{long}) \quad (8.3)$$

In Equation 8.2, we first compute the new longitude by adding a shift to the last rail point, this shift depends on the speed and the angle of the previous iteration. We can then compute in Equation 8.3 the new latitude with the computed quadratic function. We use these two equations on lines 7–11 of Algorithm 3.

The choice of requesting the latitude given the longitude or inversely depends on what we choose to place on the x-axis. We can use the longitude as x-axis and the latitude as y-axis and then compute the next latitude given a new longitude. Otherwise, we can switch the two axes and compute the next longitude given a new latitude (line 18).

Basically, we request the next longitude (resp. latitude) if the average gap between the previous latitudes (resp. longitudes) is greater than the gap between the previous longitudes (resp. latitudes), as described in the first condition of Algorithm 3.

Even though it is usually a correct choice, there are some exceptions. For example, we see that in Figure 8.8a the average gap of the longitudes is more important, but the prediction of the next latitude given by the quadratic function is wrong. Instead, we would prefer to revert the axes and compute the prediction made by the function of Figure 8.8b and request the longitude instead of the latitude. Indeed, we clearly notice in Figure 8.8a that the red curve is impossible in reality, the angle of curvature is way too large for a real railway curve, while the green curve is quite realistic.

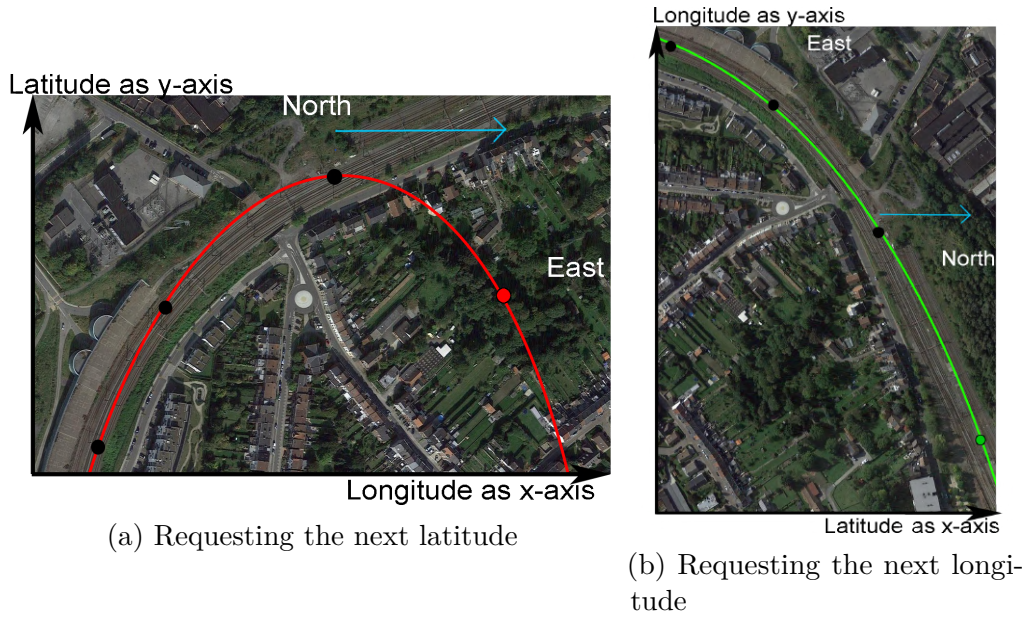


Figure 8.8: Example of the next prediction depending on the chosen axis

The issue is thus to estimate the railway curve radius. [41] gives a table of the minimum railway curve radii in the world. Thence, if the estimated radius of the rail is smaller than the minimum radius defined, we switch the axes as described on lines 17 and 20. The radius can be computed thanks to the tangents of the quadratic function at the first coordinate of the three points, and at the new coordinate (lines 13 – 15). The angle formed by these two tangents gives the central angle of the circle, as shown in Figure 8.9.

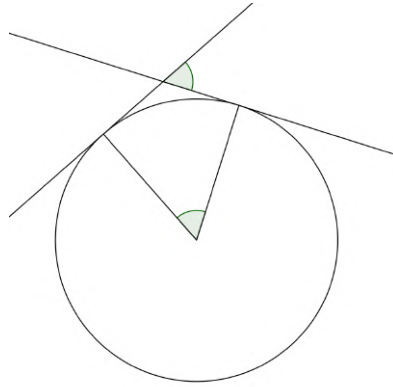


Figure 8.9: The angle between the two tangents is the same as the central angle [42]

This central angle is also known as the degree of curvature and can be used with the chord to compute the radius of curvature. In our case, the chord is simply the distance between the first of the three points and the new estimated point. Figure 8.10 illustrates this relation. From the degree of curvature, we can obtain the radius with the following formula: [43]

$$radius = \frac{chord}{2\sin(\frac{D_c}{2})} \quad (8.4)$$

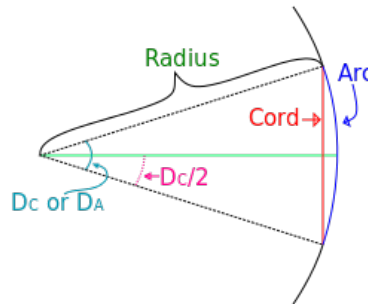


Figure 8.10: The relation between the degree and the radius of curvature [43]

Algorithm 3: Estimation of the estimated coordinate using Quadratic Lagrangian method

Data: R the list of found coordinates, $speed$ the current speed from Algorithm 1

Result: The coordinate of the new estimated point

```

1  $avg\_gap_{lat} \leftarrow$  average latitude gap between last three points
2  $avg\_gap_{long} \leftarrow$  average longitude gap between last three points
3  $last_{3points} \leftarrow$  the last three points of  $R$ 
4  $angle \leftarrow$  the angle between the last two points of  $R$ 
5
6 if  $avg\_gap_{lat} > avg\_gap_{long}$  then
    // Use long as x-axis and lat as y-axis
7      $last_{lat} \leftarrow$  the latitude of the last point found in  $R$ 
8      $shift_{NS} \leftarrow getShiftNS(last_{lat}, speed \cdot 256, angle)$ 
9      $new_{lat} \leftarrow last_{lat} + shift_{NS}$ 
10     $f \leftarrow$  The quadratic function that passes through  $last_{3points}$ 
11     $new_{lng} \leftarrow f(new_{lat})$ 
12
13     $chord \leftarrow$  distance between third last point of  $R$  and  $(new_{lat}, new_{lng})$ 
14     $D_c = \tan | \arctan f'(last_{lat}) - \arctan f'(new_{lat}) |$ 
15     $radius \leftarrow chord / (2 \sin(D_c/2))$ 
16
17    if  $radius < radius_{min}$  then
        // Use lat as x-axis and long as y-axis
18 else
    // Use lat as x-axis and long as y-axis
19
20    if  $radius < radius_{min}$  then
        // Use long as x-axis and lat as y-axis
21 return  $new_{lat}, new_{lng}$ 

```

8.3 Checkpoints

A recurrent problem we encountered when testing our solution is the choice that was made while it approaches intersections, often near the stations. The two methods proceed by evaluating the next point in accordance with the previously estimated positions. Therefore, the new estimated coordinate can be found on another close railway line if its location is closer to the estimation. Then, as the algorithm

cannot distinguish one line from another, it keeps up by tracking the wrong line and never reaches the destination. This usually occurs when the line we want to follow turns while another goes straight. Figure 8.11 shows some of those situations.



Figure 8.11: Intersections where checkpoints should be used

To counter this problem, we let to the user the possibility of specifying checkpoints along the path to clarify the right way. Thus, when the algorithm estimates a point close to a checkpoint (within a minimum range), it takes the checkpoint instead and re-estimates the direction according to the next checkpoint before continuing its job.

Following this addition, the program is now halted when it goes too far from the next checkpoint (instead of the final destination as described before) to avoid to follow the wrong path.

8.4 Comparison Between Estimation Methods

We compare the two rail tracking methods Weighted Average Angle (WAA) and Quadratic Lagrange Interpolating Polynomials (QLIP) combined with the two classification models MobileNetV2 and Xception on 15 tests with a mean distance of 30 km (505 km in total). We use the number of requests sent to Google Maps, the number of failures as well as the number of rollbacks during the rail tracking to make the comparison.

The number of requests sent gives a good indication of the performance since fewer requests means fewer prediction errors. Also, this is what has the biggest influence on the total execution time of the program because network requests are slow compared to the time needed for one prediction.

Nevertheless, if the program were to access the images differently, the objective would be different. For instance, with a local database, the time to retrieve the

images would thus be reduced and the execution time would be much more influenced by the prediction time.

The comparison is reported in Table 8.1. In this table, the last two columns reports the total number of rollbacks on the 15 tests and the number of times it did not reach the destination (number of failures) and can see that QLIP outperforms WAA on the three metrics. This difference is caused by the fact that WAA uses more requests because it needs more estimations to track rail turns correctly. The performance of the methods on each test is detailed in appendix B.

Class.	Model	Tracking Method	Nb. Requests	Nb. Roll.	Nb. Failures
MobileNetV2		QLIP	4278 (8,48 req./km)	2	1 (6,67%)
MobileNetV2		WAA	4659 (9,23 req./km)	3	2 (13,33%)
Xception		QLIP	4091 (8,11 req./km)	3	1 (6,67%)
Xception		WAA	4450 (8,82 req./km)	4	2 (13,33%)

Table 8.1: Comparison between WAA and QLIP combined with MobileNetV2 or Xception on 15 tests, with a mean distance of 30 km. In this Figure, Nb. Roll. means “Number of rollbacks”.

QLIP with MobileNetV2 needs more requests than QLIP with Xception as MobileNetV2 predicts more false positives or false negatives than Xception. However, MobilenetV2 makes a prediction almost twice faster. Table 8.2 reports an estimation of the time needed to reach the total distance of 505 km. On average, it needs 0,1594 seconds to get one 620×620 tile from Google Maps API. This time is given on an indicative basis because it depends on Google network, the program being hosted in Google Colab.

As described previously, the total number of predictions for one 620×620 tile is between 25 and 5 (see subsection 8.1.1). On the one hand, MobileNetV2 tests a 256×256 tile in 3,7 milliseconds while Xception does it in 7 milliseconds. Thus, MobileNetV2 needs between $5 \times 0,0037s + 0,1594s = 0,1779s$ and $25 \times 0,0037s + 0,1594s = 0,2519s$ to load and predict the 620×620 tile. On the other hand, Xception needs between $5 \times 0,0070s + 0,1594s = 0,1944s$ and $25 \times 0,0070s + 0,1594s = 0,3344s$. As a result, MobileNetV2 is faster but requests more tiles than Xception. Eventually, the total estimated time in both cases are reported in Table 8.2.

	MobilenetV2	Xception
Time to request a 620×620 tile	0,1594	0,1594
Time to Predict one 256×256 tile	0,0037	0,0070
Nb of prediction for a 620×620 tile	[5, 25]	[5, 25]
Total time for one request	[0,1779 ; 0,2519]	[0,1944 ; 0,3344s]
Number of requests for 505 km	4659	4450
Estimation time for 505 km	[828,8361 ; 1173,6021]	[865,08 ; 1488,08]

Table 8.2: Estimation of the time (in seconds) needed to fetch and predict the tiles used to track the railway on a total of 505 km, using QLIP with MobileNetV2 or Xception.

In conclusion, we see that QLIP performs better overall. The choice between MobilenetV2 and Xception represents a choice between speed and accuracy. One could use the fast model for clear areas and use Xception, which is more robust, for urban areas where more confusion with the rail pattern can arise. More importantly, it happens that one model fails on a path while the other succeeds. The models can thus be interchanged in order to give more chance to the completion of the task.

Notice that the estimation time considers only the time to fetch and predict the tiles. Actually, the execution time is longer than the theoretical estimation, it takes on average 2,15 seconds to track one kilometer of railway, but it depends on the complexity of the railway curve, the method used and the network quality.

8.4.1 Intermediate Results

Figure 8.12 shows a sample output from our rail tracking algorithm. You can see that the rail localization is accurate as each point is precisely on the railway line. Yet, a small minority is not accurately on the center of the rail, as illustrated in Figure 8.13. As you can notice, some points are superfluous and removing them does not impact the shape of the curve, more details are given in the next section.



Figure 8.12: Sample of the outputs of the rail tracking algorithm



Figure 8.13: A small minority of points are not exactly centered on the rails

8.5 Cleaning the Points

As partially mentioned before, reducing the number of found rail coordinates decreases the required time for the final user to review the railway found, as fewer points must be corrected. This reduction must not have an impact on the accuracy of the estimated path.

In this section, we introduce different techniques to reduce the number of points with the aim of keeping the accuracy of the path described by these points.

8.5.1 Naive Approach

The first approach we present is naive, we wrote a simple algorithm (described in Algorithm 4) that only keeps the key points where the railway changes its direction along the path, and the intermediate points between these key points are removed.

At each step, it iterates on all the remaining points until there is no more change in two successive iterations. Let ϑ_{i1} be the angle between $R[i]$ and $R[i] + 1$ and ϑ_{i2} be the angle between $R[i] + 1$ and $R[i] + 2$, we remove $R[i] + 1$ if the difference between ϑ_{i1} and ϑ_{i2} is smaller than a given threshold, as reported on lines 8 and 9.

Algorithm 4: Naive approach to clean intermediate points along the railway path

Data: R the list of found coordinates of the railway from *org* to *dest*,
threshold the minimal angle between two points so they are kept
Result: $R_{reduced}$ the reduced list of coordinates where only the coordinates that best describe the curve of the railway are kept.

```

1  $R_{reduced} \leftarrow R$ ;
2  $sizePreviousIteration \leftarrow 0$ ;
3 while  $|R_{reduced}| \neq sizePreviousIteration$  do
4    $sizePreviousIteration \leftarrow |R_{reduced}|$ ;
5   for  $i = 1$  to  $|R_{reduced}| - 2$  do
6      $\vartheta_{i1} \leftarrow angle(R_{reduced}[i], R_{reduced}[i + 1])$  ;
7      $\vartheta_{i2} \leftarrow angle(R_{reduced}[i + 1], R_{reduced}[i + 2])$  ;
8     if  $|\vartheta_{i1} - \vartheta_{i2}| \leq threshold$  then
9        $R_{reduced} \leftarrow R_{reduced} \setminus R_{reduced}[i + 1]$ 
10 return  $R_{reduced}$ 

```

8.5.2 Ramer-Douglas-Peucker Algorithm

The Ramer-Douglas-Peucker algorithm [44], or RDP for short, is a recursive algorithm dealing with the exact same problem. It reduces the number of points of a curve estimated by a list of points. It takes as inputs the list and a threshold parameter ε and returns the reduced list.

Figure 8.14 shows the different steps of the algorithm. It proceeds by using a “divide and conquer” approach. At step 1, it finds the furthest point from the line segment defined by the first and the last points (that are obviously kept). If the point is closer than ε , any intermediate points between the line segment not

marked to be kept can be discarded. Otherwise, the furthest point is marked to be kept and the algorithm calls itself with the first and the furthest point (step 2) and then with the furthest and the last point (step 3).

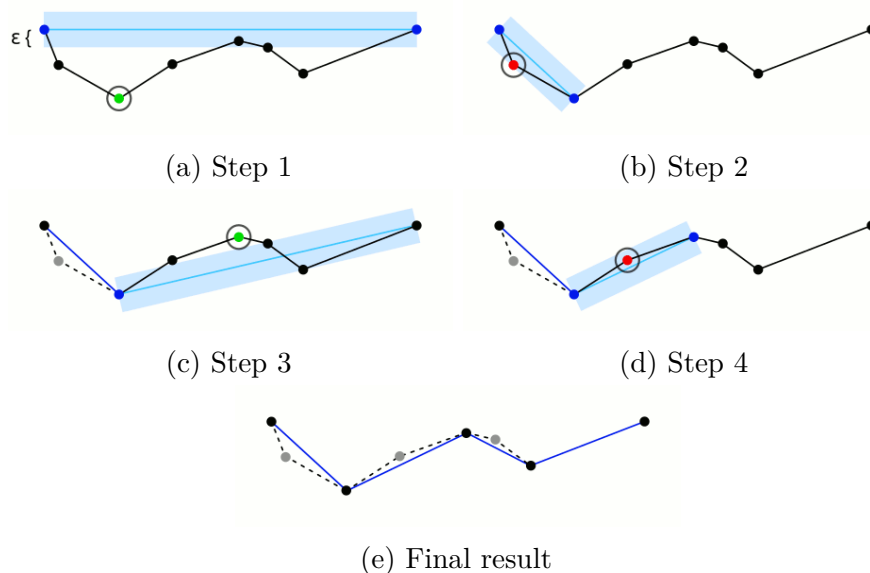


Figure 8.14: Steps of the RDP algorithm

8.5.3 Intermediate Interpolation

In order to improve the result obtained with the RDP algorithm, we add an intermediate step before cleaning that interpolates the spline that pass through the points. Consequently, RDP will get thousands of points as input and will be able to choose the best control points that define the spline .

Different algorithms can be used to perform this operation. We choose to use the “centripetal Catmull-Rom spline” algorithm (CCRS for short), introduced in [45]. Basically, it sequentially takes four successive points that were found by our rail tracking algorithm and interpolates one hundred values in-between.

We use the intermediate interpolation only in combination with RDP. We do not use it with our naive cleaning algorithm because if we give it many points close to each other as input, the angle difference between each successive point is tiny. As the naive algorithm iterates the points sequentially, it is much more complicated to find an effective threshold.

8.5.4 Comparison of the Cleaning Methods

Finally, we compare the three cleaning techniques with the ground-truth paths that we manually created. We choose two railway lines to observe the general behavior:

- A 42-kilometer journey with a few large curves in Italy
- An 18-kilometer journey with many tight curves in Spain

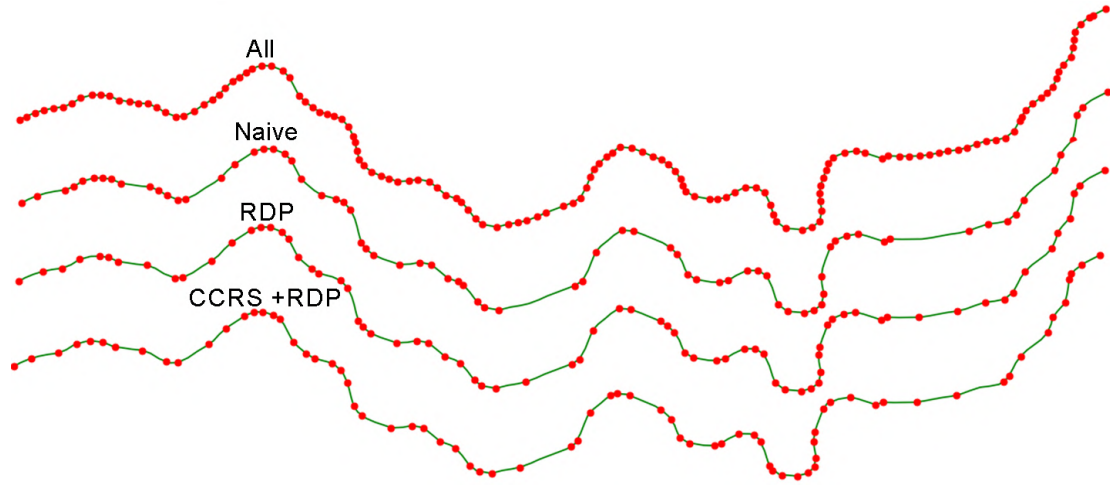
Table 8.3 reports the values of the Area Between the Curves (see section 2.4 for details) for each technique. The ABC is computed using a final interpolation (with Catmull-Rom spline) on the remaining points. The use of splines gives a more realistic representation of the real railway curve, making the creation of the ground truth easier (as much fewer points are needed to describe it accurately) and is closer to what is used in the Track Editor Tools.

We choose to report the values using different thresholds for each technique so that they output the same number of points. Since the minimization problem is a trade-off between the number of estimated points and the desired accuracy, we leave the final choice of the threshold to the end user in accordance with the complexity of the railway line and its preferences.

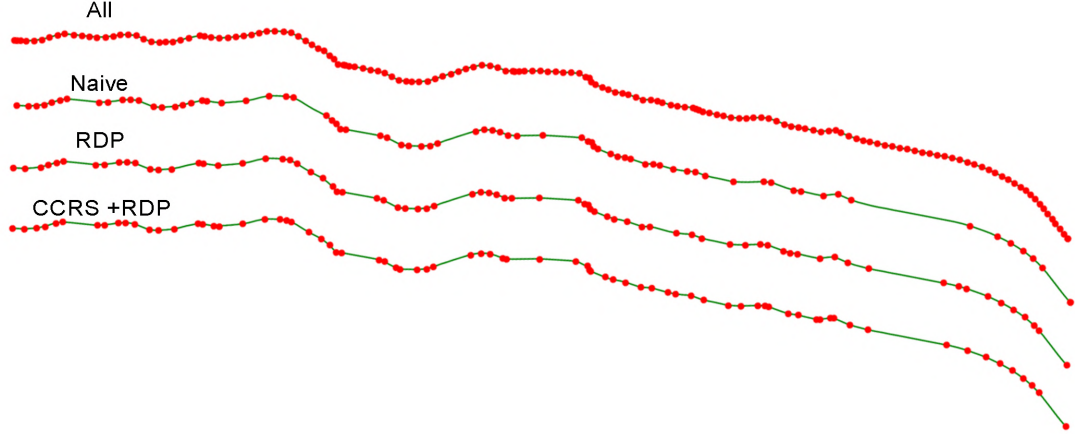
Method	E Italy	ABC Italy (m ²)	E Spain	ABC Spain (m ²)
Naive	50	625.927	50	377.531
RDP	50	564.162	50	288.608
CCRS + RDP	50	517.044	50	281.650
Naive	75	450.833	75	183.479
RDP	75	351.720	75	147.510
CCRS + RDP	75	374.680	75	146.360
Naive	100	303.591	100	114.678
RDP	100	309.294	100	115.261
CCRS + RDP	100	322.271	100	119.196
Naive	125	290.877	125	94.776
RDP	125	289.566	125	94.575
CCRS + RDP	125	292.927	125	100.747
No cleaning	158	283.092	144	93.320

Table 8.3: Comparison between the ABC obtained with the cleaning techniques as a function of the number of estimated points ($|E|$).

The observation is quite similar for both journeys. The naive and the RDP algorithms give the best results when removing only a small number of points. When we clean more intensively, the performance of the naive algorithm becomes poor and RDP combined with CCRS starts to give better results. Yet, the areas between the curves, when using a small number of points, are very large compared to the ones without cleaning.



(a) Example in Spain (18 km)

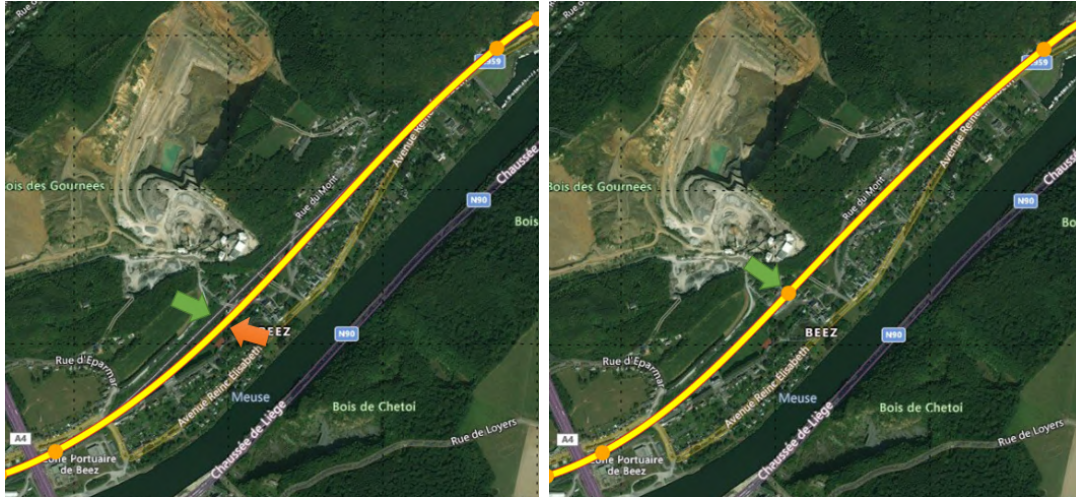


(b) Example in Italy (42 km)

Figure 8.15: Example of the computed cleaned point, depending on the used method, with 75 remaining points

Figure 8.15 shows, for each method, the positions of the points after cleaning. Overall, the original path (in green) remains the same in all cases. Yet, the intervals

between the points are quite different, especially on the straight lines. This is important to keep intermediate points on those straight lines. Otherwise the spline could go too far away from the railway, as illustrated in Figure 8.16. This is not the case of the naive algorithm and this is why it is outperformed by CCRS+RDP when we remove many points.



(a) The path is less accurate without an intermediate point (b) The path is more accurate when we keep an intermediate point

Figure 8.16: Shows the difference of accuracy when adding an intermediate point on a straight line

Chapter 9

Integration into the TET

The integration of the rail tracking algorithm into the Track Editor Tools (TET), provided by Transurb, is not part of this work. We leave them the freedom to integrate it into their own software.

Nevertheless, we wrote a procedure that creates an XML file, readable by the TET. This file contains the final rail predictions found by our algorithm. It is constructed by filling an XML template with the list of predicted coordinates after converting them into X,Y offsets (in meters) relative to the origin.

Figure 9.1 illustrates an example of a track loaded in the TET, where orange points are the cleaned points computed by the rail tracking algorithm. The red curve is the spline computed by the TET, based on the provided points. As one might have guessed, the TET correctly places the rail points on the map, although the two programs do not use the same satellite map providers. Indeed, the TET uses Bing Map to display the map while we use Google Maps to make the detection.

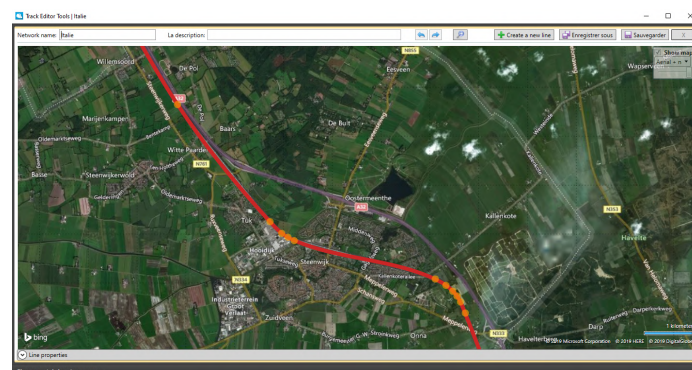


Figure 9.1: Integration of a path computed by the rail tracking into the Track Editor Tools

Chapter 10

Future Improvements

This chapter presents a non-exhaustive list of enhancements that could be made to improve our work, or external factors that could improve the current solution. We always tried to provide a maximum number of features that we thought could be worthwhile. Yet, some of them could only be added with additional efforts and time while others depend on the quality of external services.

10.1 Image Quality

We decided to use Google Maps for our solution. At the best of our knowledge, this service provides the best image quality as we write these lines. In the recent years, we saw great improvements of the image quality and of the zoom level, especially in big cities, where Google has even implemented a “bird’s eye view”. In the concerned regions, we notice an outstanding image quality, where the images are brighter and rails can be recognized easily.

In Belgium, we can find this view in cities like Brussels and Antwerp, but we expect that it will be available for many more cities in the future, even smaller ones. Figure 10.1 shows this quality difference. Getting such image quality can only help the algorithm to perform better. Moreover, the shadows that mask the railroads are less often a nuisance. As the image is brighter, we can usually recognize the rails.



Figure 10.1: Transition between classic view on the lower left side, and bird's eye view on the upper right side, near Antwerp

Of course, our artificial intelligence performs already very well with the classic view when the zoom level 20 is available. Unfortunately, this zoom level is not available in most of the developing countries we explored, neither in some regions of developed countries. In this case, the images are upscaled and they become so blurry that it makes impossible for our solution to work there. Figure 10.2 shows this difference. Again, the enhancement of this image quality is the only way to make possible the rail recognition and we expect to see gradual improvements in the near future.

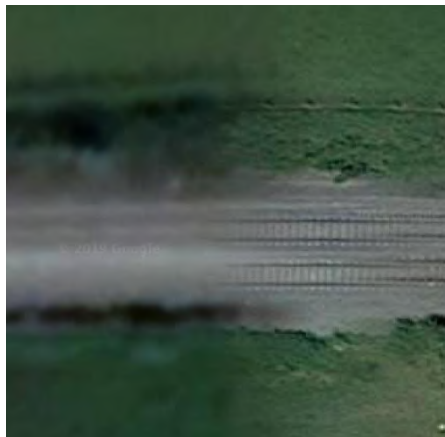


Figure 10.2: Quality difference with upscaled images near Kildare, Ireland

Obviously, those improvements do not only concern Google Maps. We expect enhancements from other services, even free ones, which could lead to the usage of other providers in a cheaper or more accurate way.

10.2 Additional Data

During this project, we used about 55.000 images captured in different locations to train and validate our models. This is a fairly large number of images and it took some time to obtain. Although we could not add more examples without exceeding the RAM provided with our notebooks, progress is likely to be seen with more training examples. Indeed, adding more rail types from additional countries and at different latitudes could lead to a more robust solution.

Also, we focused on creating a dataset with examples of good quality. It would be interesting to see how the classification model deal with poor quality images if we add more examples of second-rate. Yet, this could lead to the misclassification of some roads, as railways looks like unmade roads in poor quality images.

As we mentioned previously, mislabeled examples are present in our dataset even though we tried to clean it as much as possible. It is not a big issue when this number of examples is limited. However, with the accuracy we have managed to achieve, it is now a more significant concern because they might be responsible for our residual mistakes. Thus, besides adding more data, cleaning the current dataset may also lead to a more robust artificial intelligence.

10.3 Altitude

A nice addition to our rail tracking algorithm would be to consider the elevation of the railways, this for two reasons. First, the Track Editor Tools is capable of using the altitude to simulate the railway line in a more realistic fashion. Second, it would allow the validation of the estimated point. As we perfectly know, trains are not capable of crossing a large difference in altitude in a short distance.

The major issue we perceived when considering the addition of this feature concerns bridges. Given that there can be a huge difference between the altitude at the level of the bridge and that of the ground below. It would lead to erroneous data since the elevation APIs gives the altitude at the ground level.

Furthermore, the elevation APIs are not as accurate as we would need. When we investigated the idea, we had a look at the API of Google and we noticed that it returns a resolution value, indicating the maximum distance between data points from which the altitude was interpolated. As this value can reach a few tens of meters, we could get inaccurate data because railways have width of several meters and the ground level is often adapted to deal with their elevations.

Nonetheless, it could be used after cleaning the spline. Since the number of points would be less numerous, the probability of keeping a point on a bridge is lower as well as keeping two points with large altitude difference. Thus, it could give a general trend of the altitude variation. The validation could be made afterwards, where the suspicious sections could be marked for the end user.

10.4 Multiple Classes Classification

This work focused on binary classification because that was the most straightforward approach given the problem we had to solve. However, our classifier could be extended to predict multiple classes after recognizing whether a tile contains rails.

With this aim in mind, a second classification model could be built on top of the first, to predict if a tile contains a level crossing, a bridge, a catenary (to know if the line is electrified) or platforms (which could help to detect a station). Figure 10.3 shows that these elements can be easily spotted on good quality images.



Figure 10.3: Additional elements that could be recognized by a second classifier

To be efficient, this would require to travel the railway line tile by tile. Otherwise, such elements would be easily skipped. Also, it has to be noted that those elements are not all mutually exclusive (a tile may contain platforms and a catenary). Finally, the dataset should be created accordingly where some of these elements are not easy to find in large numbers.

Chapter 11

Conclusion

As part of this research work, we introduced the rising field of deep learning with focus on convolutional neural networks applied to image recognition. We applied recent deep learning techniques to classify satellite images with a strong accuracy, and we built an algorithm on top of it to track a specific railway line.

We showed that CNN could be highly efficient and accurate with a reasonable amount of data and training time. The results obtained with our classification models were beyond our expectations. We reached 99,8% of accuracy on a test set composed of 6.000 balanced examples, by choosing appropriate deep neural architectures and by tuning associated hyperparameters. Thanks to the great availability of satellite data, our models are trained to detect rails of many countries. Yet, classification can be unsatisfying in some regions, either because they use specific types of rail that are not covered in our training set, or because the image quality is poor.

Our rail tracking algorithm can follow railways for hundreds of kilometers as long as the rails are sufficiently visible. This includes an adequate image quality and a suitable railway path, i.e. a path with a reasonable amount of difficulties along the route (like tunnels, shadows, clouds, etc.). The objectives of the rail tracking algorithm was twofold: first, reproduce the actual railway as precisely as possible and second, minimize the number of requests sent to the image provider. We then proposed two efficient approaches to solve this problem. Both come with many parameters that allows the user to set up the tracking depending on the line features. It was designed to let the user choose between speed, accuracy and reliability due to the large set of environments that can be encountered.

Overall, this project is a success from a research point of view. We proved its feasibility, which is always cheering in a research work. We addressed most faced

issues and satisfied the initial needs of Transurb. Furthermore, this project leads to many interesting research directions. As a starting point, we cited possible enhancements at the end of this document. Thereafter, we think that further tasks could focus on finding a more convenient alternative to the checkpoints method used to deal with the intersections. Still, we feel that this work has provided enough content for future inspirations and hope that our solution will be a useful time-saving tool for Transurb.

Bibliography

- [1] Cambridge Dictionary. *Definition of Big Data*. URL: dictionary.cambridge.org/dictionary/english/big-data.
- [2] A. Correa Bahnsen. *Building AI Applications Using Deep Learning*. deeplearning.ai, 2017. URL: blog.easysol.net/building-ai-applications/.
- [3] A. Ng. *Why is Deep Learning taking off?* URL: blog.easysol.net/building-ai-applications/.
- [4] J. Redmon et al. *You Only Look Once: Unified, Real-Time Object Detection*. 2015. arXiv: 1506.02640 [cs.CV].
- [5] Mapbox. *Zoom level*. URL: docs.mapbox.com/help/glossary/zoom-level/.
- [6] OpenStreetMap contributors. *Zoom level*. 2012. URL: wiki.openstreetmap.org/wiki/Zoom_levels.
- [7] S. Shah. *Do we really need GPU for Deep Learning?* 2018. URL: medium.com/@shachishah.ce/do-we-really-need-gpu-for-deep-learning-47042c02efe2.
- [8] M. Heller. *What is CUDA? Parallel programming for GPUs*. 2018. URL: infoworld.com/article/3299703/what-is-cuda-parallel-programming-for-gpus.html.
- [9] O. Bar-El. *Getting the Most Out of Your Google Colab*. 2018. URL: medium.com/@oribarel/getting-the-most-out-of-your-google-colab-2b0585f82403.
- [10] S. Sharma. *Activation Functions in Neural Networks*. URL: towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6.
- [11] A. Karpathy. *CS231n: Convolutional Neural Networks for Visual Recognition*. (cs231n.github.io/convolutional-networks/).
- [12] J. Brownlee. *How Do Convolutional Layers Work in Deep Learning Neural Networks?* 2019. URL: machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/.

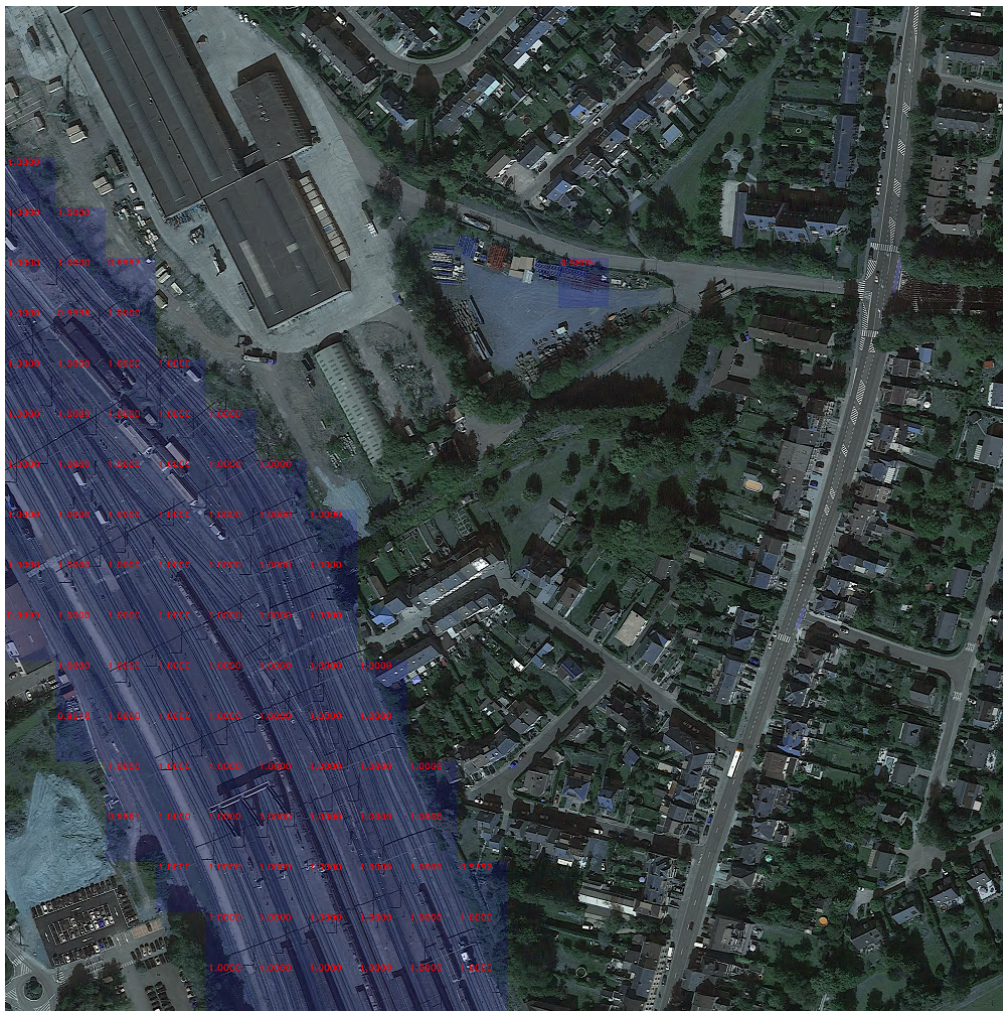
- [13] R. Prabhu. *Understanding of Convolutional Neural Network (CNN) — Deep Learning*. 2018. URL: medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148.
- [14] MathWorks. *Convolutional Neural Network*. URL: mathworks.com/solutions/deep-learning/convolutional-neural-network.html.
- [15] A. Deshpande. *A Beginner's Guide To Understanding Convolutional Neural Networks*. 2016. URL: adeshpande3.github.io/adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks/.
- [16] A. Dertat. *Applied Deep Learning - Part 4: Convolutional Neural Networks*. (towardsdatascience.com). 2017.
- [17] M. West. *Convolutional Neural Networks : The Theory*. 2019. URL: bouveto.com/bouvet-deler/understanding-convolutional-neural-networks-part-1.
- [18] M. Nielsen. *Deep learning*. 2019. URL: neuralnetworksanddeeplearning.com/chap6.html.
- [19] Md Zahangir Alom et al. *The History Began from AlexNet: A Comprehensive Survey on Deep Learning Approaches*. URL: arxiv.org/ftp/arxiv/papers/1803/1803.01164.pdf.
- [20] Stanford Vision Lab. *Result of ILSVRC2012*. 2012. URL: <http://www.image-net.org/challenges/LSVRC/2012/results.html>.
- [21] François Chollet et al. *Keras*. keras.io. 2015.
- [22] C. Szegedy et al. "Rethinking the Inception Architecture for Computer Vision". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. 2016, pp. 2818–2826. DOI: 10.1109/CVPR.2016.308. URL: doi.org/10.1109/CVPR.2016.308.
- [23] K. He et al. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90. URL: doi.org/10.1109/CVPR.2016.90.
- [24] A.G. Howard et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: *CoRR* abs/1704.04861 (2017). arXiv: 1704.04861. URL: <http://arxiv.org/abs/1704.04861>.

- [25] M. Sandler et al. “MobileNetV2: Inverted Residuals and Linear Bottlenecks”. In: *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. 2018, pp. 4510–4520. DOI: 10.1109/CVPR.2018.00474. URL: http://openaccess.thecvf.com/content%5C_cvpr%5C_2018/html/Sandler%5C_MobileNetV2%5C_Inverted%5C_Residuals%5C_CVPR%5C_2018%5C_paper.html.
- [26] F. Chollet. “Xception: Deep Learning with Depthwise Separable Convolutions”. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. 2017, pp. 1800–1807. DOI: 10.1109/CVPR.2017.195. URL: doi.org/10.1109/CVPR.2017.195.
- [27] S. Bianco et al. “Benchmark Analysis of Representative Deep Neural Network Architectures”. In: *IEEE Access* 6 (2018), pp. 64270–64277. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2877890.
- [28] S. Tsang. “Review: Xception — With Depthwise Separable Convolution, Better Than Inception-v3 (Image Classification)”. In: 2018. URL: towardsdatascience.com/review-xception-with-depthwise-separable-convolution-better-than-inception-v3-image-dc967dd42568.
- [29] S. Tsang. “Review: MobileNetV2 — Light Weight Model (Image Classification)”. In: 2019. URL: towardsdatascience.com/review-mobilenetv2-light-weight-model-image-classification-8febb490e61c.
- [30] N. Shukla. “Gradient weighted Class Activation Map(Grad-CAM)”. In: (2019). URL: medium.com/@ninads79shukla/gradcam-73a752d368be.
- [31] B. Zhou et al. “Learning Deep Features for Discriminative Localization.” In: (2016).
- [32] R.R. Selvaraju et al. “Grad-CAM: Why did you say that? Visual Explanations from Deep Networks via Gradient-based Localization”. In: *CoRR* abs/1610.02391 (2016). arXiv: 1610.02391. URL: <http://arxiv.org/abs/1610.02391>.
- [33] D.P. Kingma and J. Ba. “Adam: A Method for Stochastic Optimization”. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. 2015. URL: <http://arxiv.org/abs/1412.6980>.
- [34] Wikipedia contributors. *Web Mercator projection*. 2018. URL: en.wikipedia.org/wiki/Web_Mercator_projection.
- [35] Rob Lammle. *Web Mercator projection*. 2014. URL: <https://www.mentalfloss.com/article/57050/actual-size-greenland>.

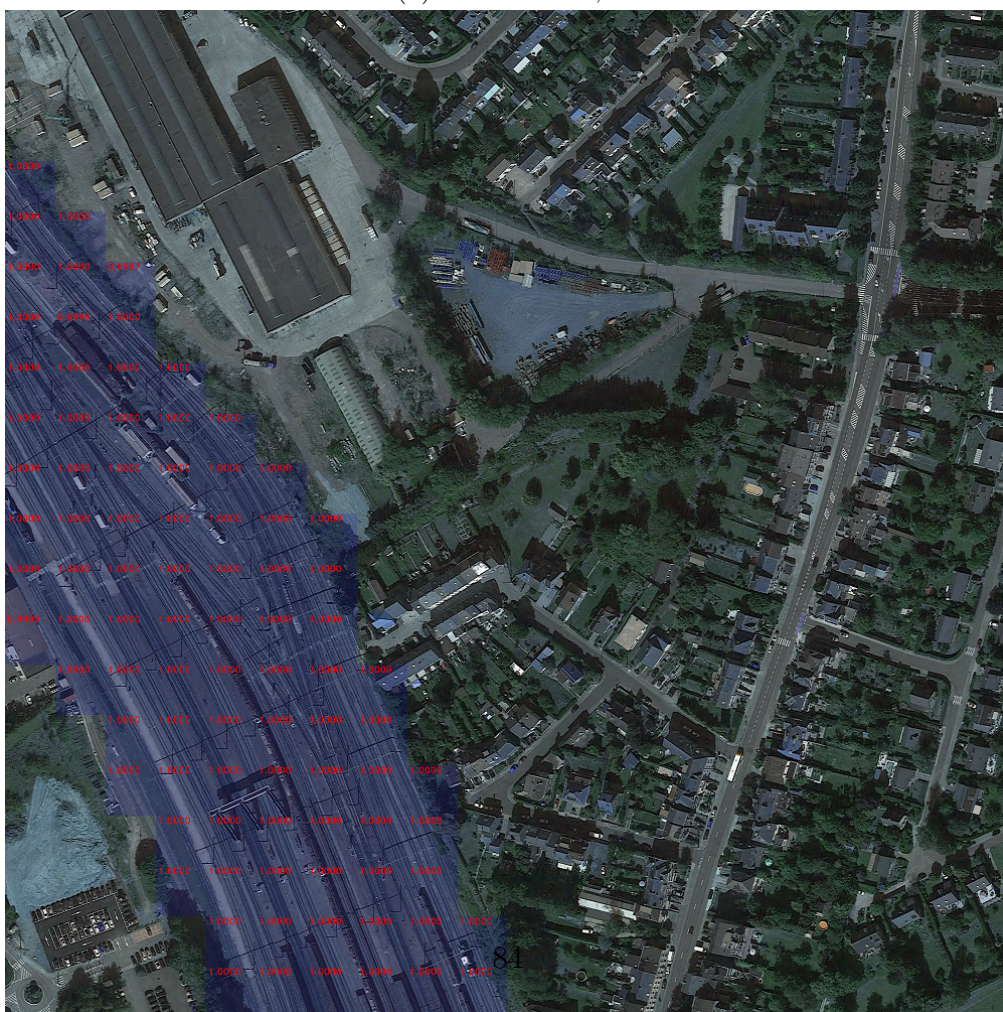
- [36] Wikimedia contributors. *Classical Mercator projection of the Earth*. URL: <https://commons.wikimedia.org/wiki/File:Mercator-proj.png>.
- [37] C. Veness. *Calculate distance, bearing and more between Latitude/Longitude points*. 2002. URL: movable-type.co.uk/scripts/latlong.html.
- [38] J. A Nelder and R. Mead. “A Simplex Method for Function Minimization”. In: *The Computer Journal* 7.4 (Jan. 1965), pp. 308–313. ISSN: 0010-4620. DOI: 10.1093/comjnl/7.4.308. URL: doi.org/10.1093/comjnl/7.4.308.
- [39] V. Cerf and R. Kahn. “A Protocol for Packet Network Intercommunication”. In: *IEEE Transactions on Communications* 22.5 (May 1974), pp. 637–648. ISSN: 1558-0857. DOI: 10.1109/TCOM.1974.1092259.
- [40] E. Waring. *Problems concerning interpolations*. 1779. DOI: 10.1098/rstl.1779.0008.
- [41] Wikipedia contributors. *Minimum Railway Curve Radius*. URL: en.wikipedia.org/wiki/Minimum_railway_curve_radius.
- [42] Wikipedia contributors. *Courbure d’un arc*. URL: fr.wikipedia.org/wiki/Courbure_d’un_arc.
- [43] Wikipedia contributors. *Degree of Curvature*. URL: en.wikipedia.org/wiki/Degree_of_curvature.
- [44] U. Ramer. “An iterative procedure for the polygonal approximation of plane curves”. In: *Computer Graphics and Image Processing*. 1972, pp. 244–256. DOI: 10.1016/S0146-664X(72)80017-0.
- [45] E. Catmull, R. Rom. “A Class of Local Interpolatin Splines”. In: 1974. DOI: 10.1016/B978-0-12-079050-0.50020-5.

Appendix A

Results of Area Detection



(a) Threshold 0,5



(b) Threshold 0.999

Figure A.1: Results in Ottignies, Belgium

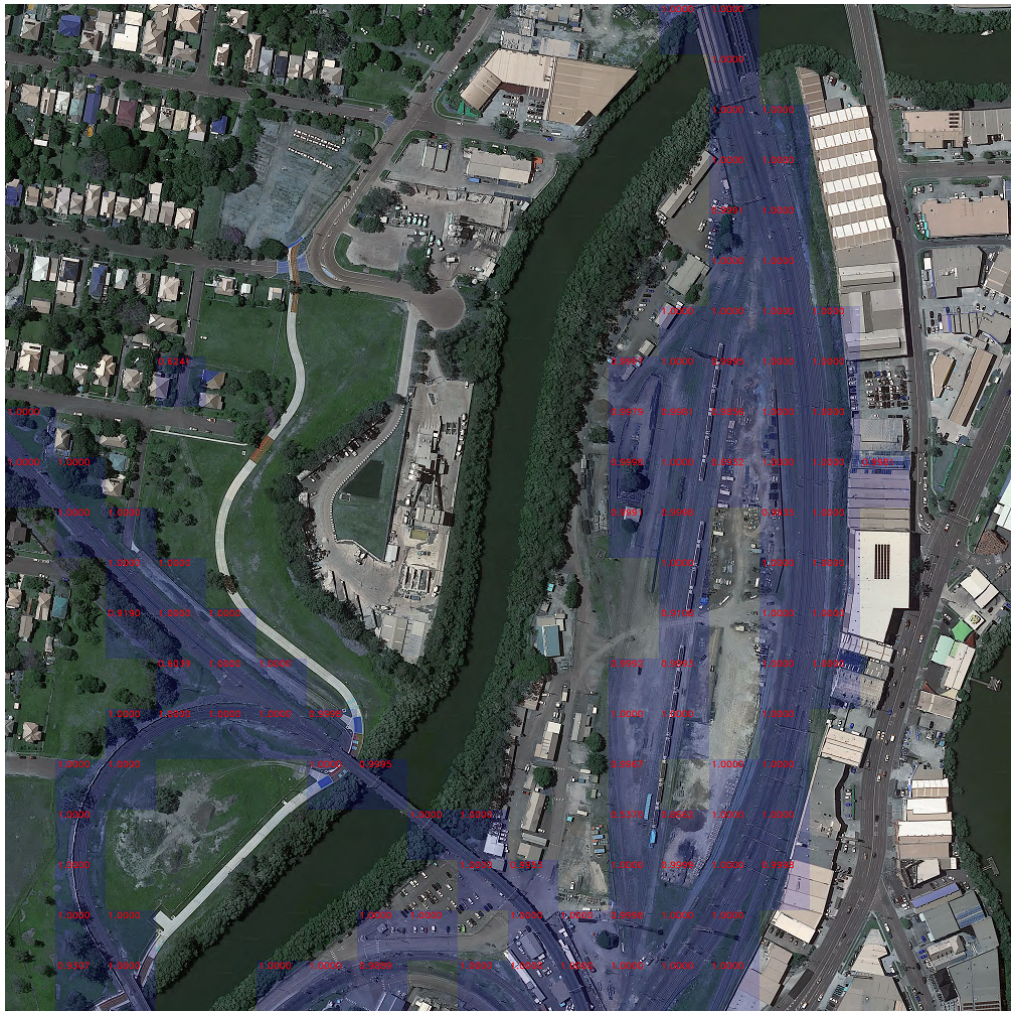


(a) Threshold 0,5

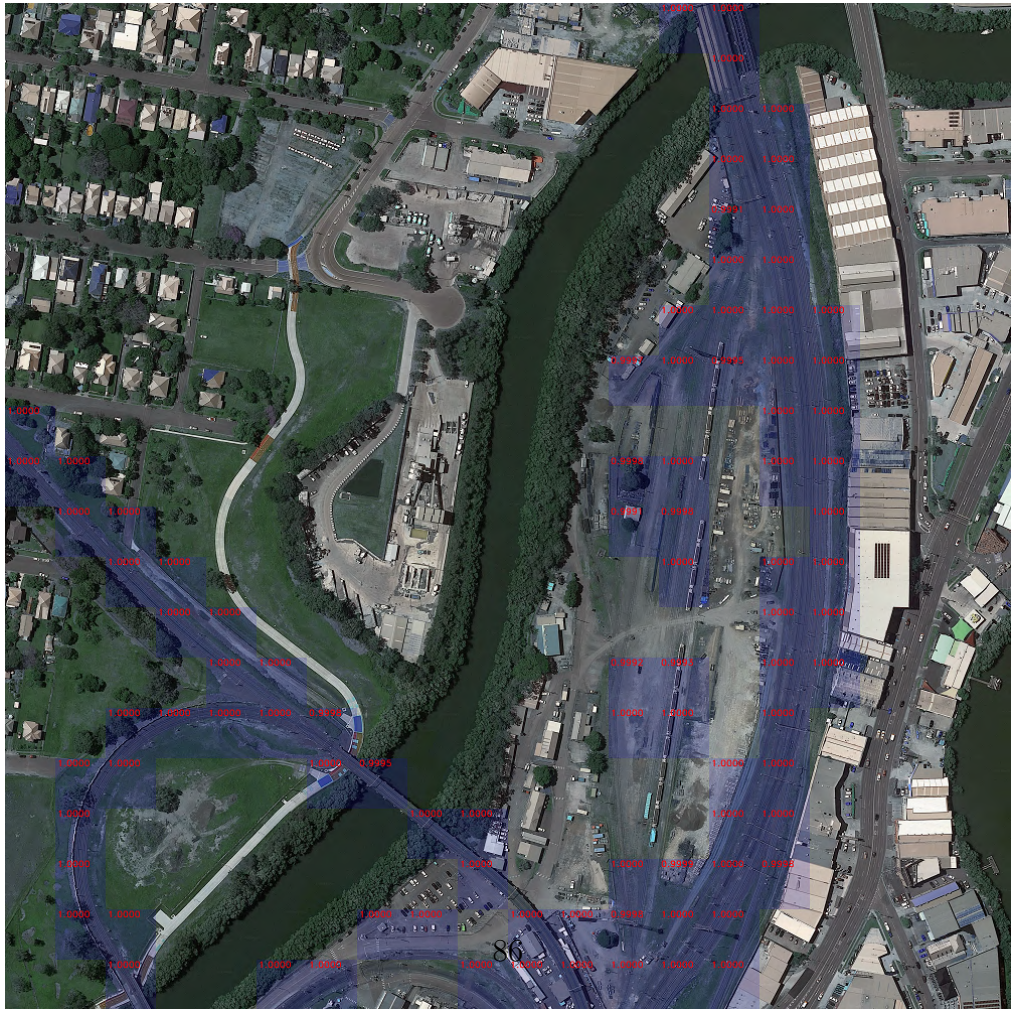


(b) Threshold 0.999

Figure A.2: Results in Villers-la-Ville, Belgium

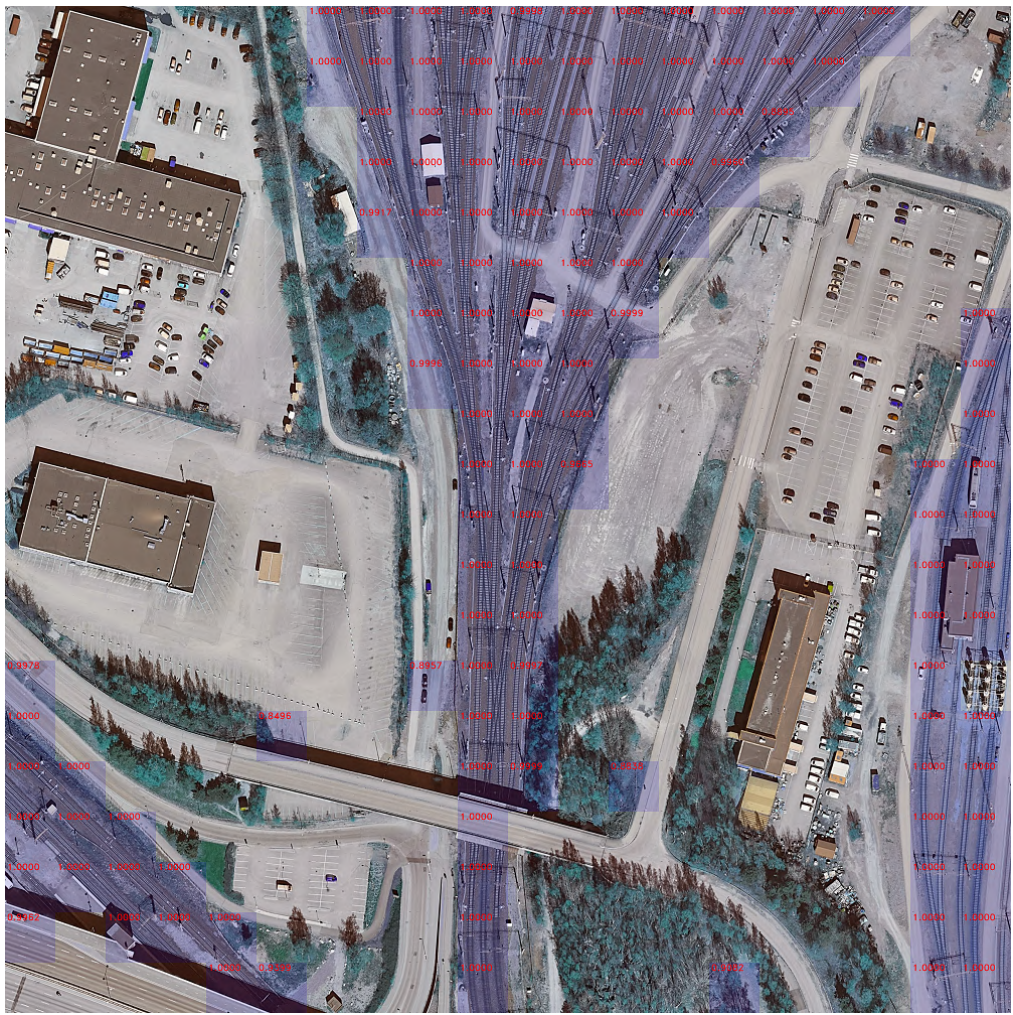


(a) Threshold 0,5

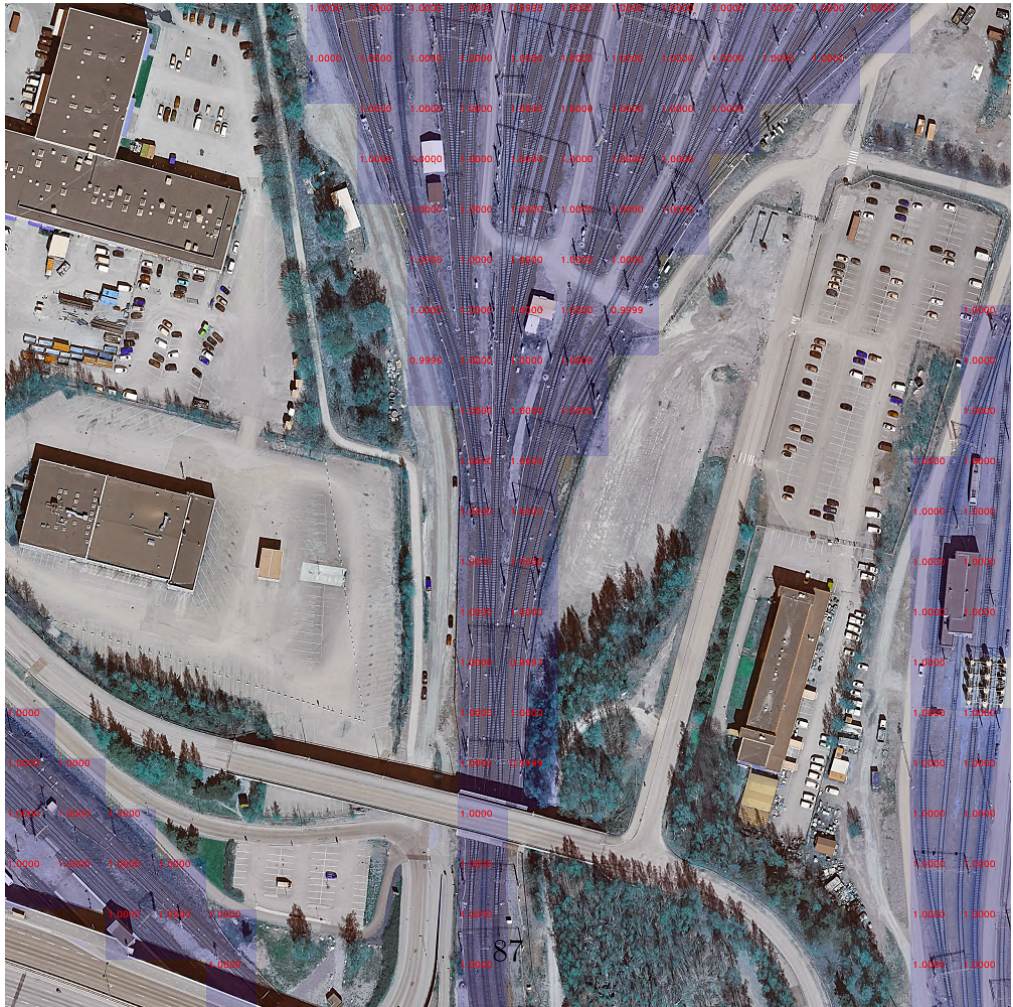


(b) Threshold 0.999

Figure A.3: Results in Brisbane, Australia

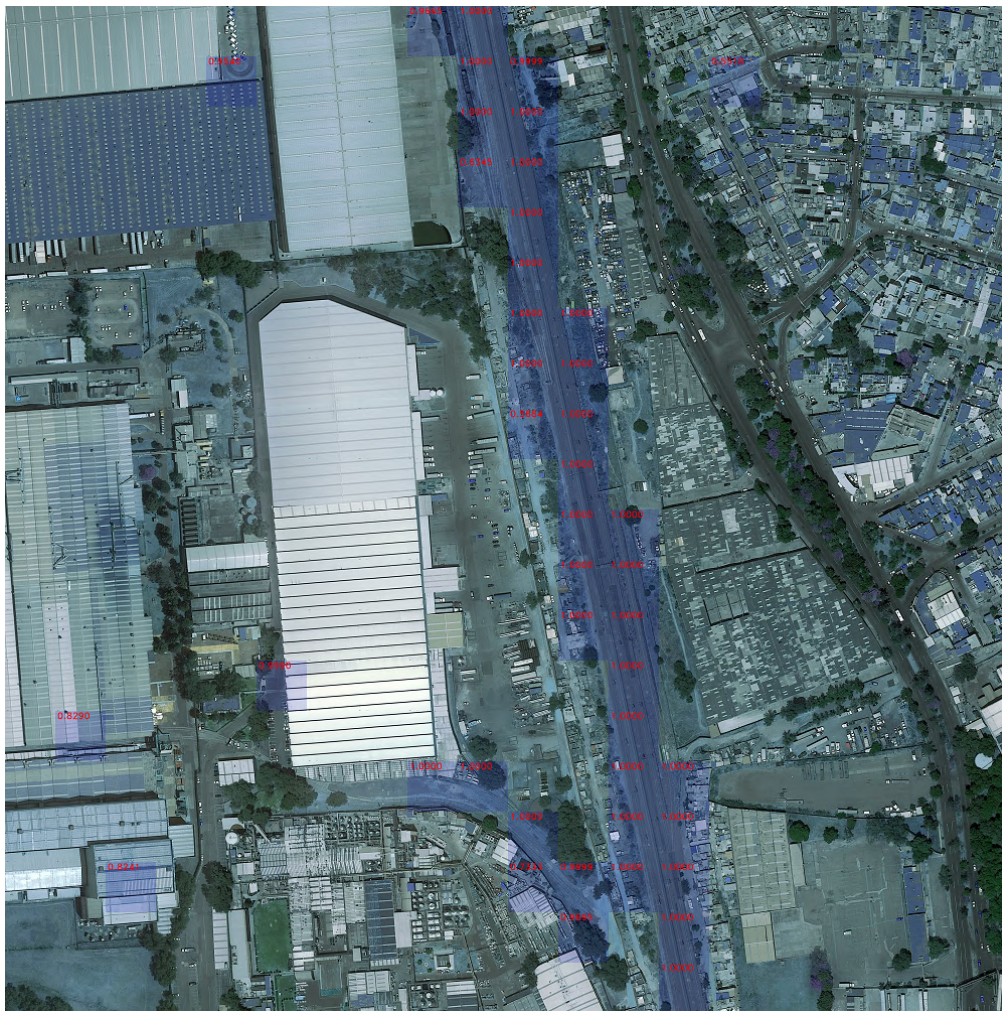


(a) Threshold 0,5



(b) Threshold 0.999

Figure A.4: Results in Helsinki, Finland



(a) Threshold 0,5



(b) Threshold 0.999

Figure A.5: Results in Mexico City, Mexico

Appendix B

Comparison of Rail Tracking Methods

Nb. km	Nb. iterations				Nb. requests				Nb. rollbacks			
	M+Q	M+W	X+Q	X+W	M+Q	M+W	X+Q	X+W	M+Q	M+W	X+Q	X+W
28	130	133	135	131	170	193	195	185	0	0	0	0
28	133	110	127	125	205	182	185	249	0	0	0	1
47	216	196	213	218	274	284	253	260	0	0	0	0
83	363	327	365	327	385	375	395	377	0	0	0	0
50	222	215	237	237	288	361	353	353	0	0	1	1
7	49	43	38	32	199	141	108	80	0	0	0	0
8	43	x	41	100	123	x	109	200	0	x	0	5
23	106	83	91	83	208	147	151	145	1	0	0	0
17	76	65	71	65	170	129	139	129	0	0	0	0
43	187	205	204	156	381	324	452	266	0	0	1	0
17	x	59	67	59	x	157	191	159	x	0	0	0
31	129	106	x	97	295	238	x	207	1	1	x	0
41	183	x	174	x	437	x	310	x	0	x	1	x

Table B.1: Comparison between WAA and QLIP combined with MobileNetV2 or Xception on 15 tests. On this Figure, M+Q = MobileNetV2+QLIP, M+W = MobileNetv2+WAA, X+Q = Xception+QLIP, X+W = Xception+WAA. The "x" indicate that it did not reach the destination.

List of Figures

2.1	Three simulator solutions proposed by Transurb	4
2.2	Example of a track in the Track Editor Tools	5
2.3	Interaction between our application and the TET	5
2.4	Illustration that highlights the importance of keeping only a few coordinates	6
2.5	Example of modules of railway topology	6
2.6	Modeling the network into the TET	7
2.7	Illustration of the Area Between the Curves	8
2.8	Illustration of the ABC, with much fewer points	8
3.1	Performance of machine learning methods as a function of the amount of training data	9
3.2	Object detection using the YOLO algorithm	10
3.3	Mask segmentation, manually labeled	11
3.4	Work flow	12
4.1	Tiles showing the different zoom levels provided by Google Maps . .	14
4.2	Tiles showing the quality provided by Google Maps in different countries	15
4.3	Tiles showing the quality provided by different satellite image services	17
4.4	Tiles showing hiding issues encountered for the classification	18
6.1	A simple deep-learning model	23
6.2	CNN adapted to this task	23
6.3	Convolution of an RGB image with a filter	24
6.4	Steps of the application of a convolutional filter	25
6.5	The ReLU activation function	26
6.6	The three types of pooling layers	26
6.7	A typical CNN architecture, adapted for this task	27
6.8	Ball chart reporting the Top-1 accuracy vs. computational complexity	29
6.9	Ball chart reporting the test accuracy vs. tiles per second	30

6.10	The depthwise separable convolution of Inception	32
6.11	Overall architecture of Xception	33
6.12	Bottleneck module in MobileNetV2	34
6.13	Steps in creating the 2D score grid using Grad-CAM	36
6.14	Heatmaps of true positive examples	37
6.15	Heatmaps of false positive examples	38
6.16	Heatmaps of true negative examples	38
6.17	Heatmaps of false negative examples	39
6.18	Heatmaps of true positives with a lack of accuracy	39
6.19	Learning curves of the Xception training phase, according to the training and validation accuracy and loss	41
6.20	Learning curves of the MobilenetV2 training phase, according to the training and validation accuracy and loss	41
7.1	Two juxtaposed tiles obtained when moving eastwards	42
7.2	Side effect of Mercator projection	43
7.3	The distance between each opposite edge varies with the latitude . .	44
7.4	Two juxtaposed tiles obtained when moving according to an azimuth	46
8.1	Main steps of the rail detection on a 620×620 tile	52
8.2	Evolution of the speed according to the railway route	53
8.3	Example of the perpendicular tracking	54
8.4	Example of a case handled by the discovery state	55
8.5	Example of a false positive, before and after the rollback	55
8.6	Example of long tunnel where the end is not reached in discovery state	56
8.7	Steps of weighted average angle	57
8.8	Example of the next prediction depending on the chosen axis	60
8.9	The angle between the two tangents is the same as the central angle	61
8.10	The relation between the degree and the radius of curvature	61
8.11	Intersections where checkpoints should be used	63
8.12	Sample of the outputs of the rail tracking algorithm	66
8.13	A small minority of points are not exactly centered on the rails . . .	66
8.14	Steps of the RDP algorithm	68
8.15	Example of the computed cleaned point, depending on the used method, with 75 remaining points	70
8.16	Shows the difference of accuracy when adding an intermediate point on a straight line	71
9.1	Integration of a path computed by the rail tracking into the Track Editor Tools	72

10.1	Transition between classic view on the lower left side, and bird's eye view on the upper right side, near Antwerp	74
10.2	Quality difference with upscaled images near Kildare, Ireland	74
10.3	Additional elements that could be recognized by a second classifier .	76
A.1	Results in Ottignies, Belgium	84
A.2	Results in Villers-la-Ville, Belgium	85
A.3	Results in Brisbane, Australia	86
A.4	Results in Helsinki, Finland	87
A.5	Results in Mexico City, Mexico	88

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl