University of Oxford

Gaining Economic Insights Through Deep Learning on Satellite Imagery

by

Adrien Chkirate

St Hugh's College

A dissertation submitted in partial fulfilment of the degree of Master of Science in Statistical Science.

Department of Statistics, 24–29 St Giles,
Oxford, OX1 3LB

September 2023

This is my own work (except where otherwise indicated)

Candidate: Adrien Chkirate

Signed:.....

Date:.....September 11, 2023.....

Abstract

This work explores the potential of high-resolution satellite imagery for predicting urban wealth levels, as indicated by income per capita. Urban infrastructure such as, airports or roads, are recognised drivers of economic growth and are easily identifiable in overhead imagery. We therefore employ deep learning to extract information on urban infrastructure from satellite images. Our approach involves designing a two-component pipeline for information extraction from aerial images. The first component is a segmentation model that extracts road networks from overhead images, while the second is an object detection model that locates selected infrastructure. These two components work in tandem, with their outputs fed into a neural network architecture designed for wealth prediction. To evaluate our approach, we conducted tests using a novel dataset comprising images from 100 U.S. cities, which we curated ourselves. Results demonstrate the viability of this approach for predicting one city wealth level relative to another. This work also highlights the underexplored potential of high-resolution satellite imagery in economic analysis.

Keywords: Deep learning, Computer Vision, Satellite Imagery, Remote Sensing, Economic Forecasting.

Acknowledgements

I would like to thank Professor Xiaowen Dong for giving me the chance to work on this fascinating multi-disciplinary project and for his valuable guidance throughout this project. I also want to express my gratitude to Professor Tom Rainforth for his insightful contributions that were instrumental in developing the models presented in this work and in structuring this thesis. Lastly, I want to sincerely thank Valentina Semenova for her continuous support and for our regular meetings. This dissertation would not have been possible without her. Enfin, je tiens à remercier ma mère, mon père et mon frère pour leur soutien inconditionnel tout au long de mes études.

Table of contents

1 Introduction

The last decade has seen an unprecedented number of ground breaking successes for machine learning. The practical success of large neural networks, combined with the accessibility of affordable hardware, has expanded the horizons of what deep learning models can achieve, accelerating progress across various research domains. This surge in capabilities has notably delivered results in natural language processing and computer vision that were previously considered unattainable [1].

The deep learning revolution has not left the economics field untouched. From enhancing time series forecasting in financial economics [2] to extracting valuable insights from extensive textual data [3], and even examining spatial inequality using graph neural networks [4], deep learning is gradually becoming more common in economics. Another promising area of study involves the utilisation of satellite imagery and the valuable insights it can offer. The potential for using satellite imagery data in economics dates back to [5], who highlights how nightlights can help us to more precisely understand economic development. More recently, recent research has demonstrated the effectiveness of deep learning techniques applied to satellite imagery data in enhancing economic forecasting [6].

Nonetheless, satellite imagery remains a relatively underexplored resource in the field of economics, despite its numerous benefits. Overhead imagery provides a unique advantage as it offers comprehensive coverage of the entire globe. This makes such imagery an invaluable source for assessing the economic conditions of regions where reliable data may be scarce, inaccessible, or affected by conflicts. Furthermore, the frequent updates provided by many satellite sources allow for tracking and assessing economic development over time, adding another dimension to its utility in economic analysis. Moreover, satellite images are now available at a level of resolution that provides access to fine details, including the type of buildings observed. Surprisingly, the economic literature has yet to fully harness the potential of high-resolution aerial images, which offer the possibility of extracting much more detailed information than, for instance, nightlights. In this work, we aim to address this gap by applying advanced deep learning models on high-resolution satellite images of urban areas to predict wealth level, measured through income per capita.

In order to estimate the wealth level of a city using solely overhead imagery, we needed to carefully pick the information we desired to extract from our images. We opted to focus on identifying key infrastructure and the entire urban road network connecting these infrastructures. Urban infrastructure, particularly transportation infrastructure, is widely recognised as a fundamental driver of economic growth [7–9] which motivated our choice. Additionally, infrastructure such as airports, train stations, and stadiums is easily distinguishable in overhead imagery due to its substantial size and distinct features, making it an ideal target for extraction. We employed deep learning for both obtaining this information on aerial images and subsequently exploiting it to estimate the income per capita of a city. Thus, our first objective was to design a pipeline using deep learning models that takes raw satellite images of urban areas as inputs and extracts comprehensive information about the infrastructure present in the area. Then, we designed a separate architecture capable of utilising the outputs from the first pipeline to estimate the wealth of a given city. Our approach is summarised in figure 1.

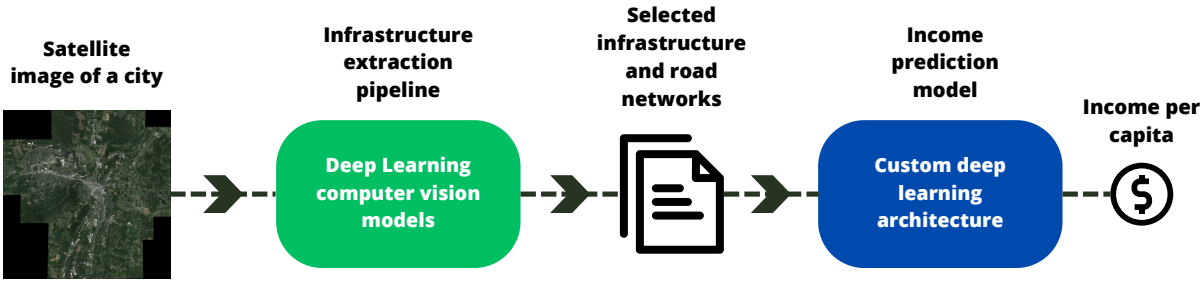


Figure 1: Schematic representation of the approach we have adopted to predict the wealth level of a city based on a satellite image of the city.

The first contribution of this work is the design of a pipeline for extracting road networks from urban satellite images. This pipeline leverages advanced computer vision architecture that have been proven to be both effective and reliable. Additionally, we developed and trained a neural network architecture capable of predicting per capita income based on the outputs of the pipeline. Our results demonstrate the utility of this information for predicting the relative wealth of a city, showcasing promising outcomes.

We have chosen to split our infrastructures extraction pipeline into two separate components, each addressing a distinct task. The first component will focus on extracting the comprehensive road network of an area from a satellite image while the second component will focus on locating various infrastructure that is considered indicative of economic growth, such as airports or storage tanks. Accordingly, the first part of this thesis is dedicated to the design of the road extraction component. We break down every aspect of its design, starting with the collection of a suitable dataset for the task, followed by the training process, and concluding with an assessment of its performance. The next section concentrates on building the detection component. We outline our data selection process and specify the types of infrastructure we intend to detect. We will then follow the same procedure as the first part, explaining our training approach and presenting the results of our detection model. The final part of this work combines our two components to create the infrastructure extraction pipeline and tests it on our manually curated dataset of satellite images. It then describes the architecture we designed to use the obtained information to predict the income per capita and assesses its results.

2 Extracting road networks

2.1 Overview of the literature and contributions

In this section, our focus is on extracting the road network from a satellite image that covers a specific location. This represents our initial step in constructing a pipeline with the ultimate goal of predicting a city's wealth using only overhead imagery. This task involves performing semantic segmentation on our input images, essentially teaching a model to classify each pixel as either a background or a road pixel. The desired output is an image of the same size as the input, highlighting the road network with white pixels against a black background, as illustrated in Figure 2.

Figure 2: On the left our input, a satellite image, and on the right the desired output with the highlighted road network.

Image segmentation has been extensively studied since the 1970s due to its diverse applications in various domains, including medicine and autonomous vehicles [10]. The substantial advancements in deep learning over the past decade have firmly established it as the leading and most effective approach for this task which, in turn, motivated our decision to address this challenge using deep learning. The literature has generated a plethora of potential architectures for segmentation, and these have been applied with considerable success in extracting road networks from aerial imagery [11]. Notably, we can mention the widespread use of the U-Net to this purpose [12–15]. In our work, we opted to implement DeepLabv3+ [16], which is part of the DeepLab series of models [17]. We selected this model due to its relatively straightforward implementation and its high level of accuracy. Indeed, the DeepLabv3+ architecture achieves state-of-the-art performance on the PASCAL Visual Object Classes (VOC) dataset which is a very popular segmentation benchmark [18].

Several papers have successfully applied DeepLabv3+ for road extraction in the past [19, 20]. However, we introduce two novel contributions in our implementation of DeepLabv3+. Our first contribution involves the use of Bayesian optimization to identify the optimal training settings for our model. The second, and perhaps the most crucial, contribution is the creation of a custom dataset for training our model, which combines data from multiple freely accessible sources. A significant challenge posed by many widely-used road extraction datasets is their limited geographical coverage, often concentrating on specific areas within a single country, as exemplified by the Massachusetts Road dataset [21]. Additionally, these datasets typically offer satellite images with fixed resolutions. This limitation hinders the training of a model from properly generalising, as road environments vary significantly between, for instance, a neighborhood in the US and a rural location in Egypt. To overcome this, we assembled our dataset by combining data from various sources. Thankfully, due to the popularity of the topic, many freely available datasets with satellite images and segmentation masks were accessible, enabling us to create a dataset encompassing diverse global regions.

In this section, we detail the creation of our curated dataset for aerial road segmentation, describe the key components of the DeepLabv3+ model for road extraction, explain our training methodology, and evaluate our trained model's road network extraction efficiency.

2.2 A curated dataset for road segmentation

To create our custom dataset, we are interested in datasets containing satellite images paired with their corresponding segmentation masks. These masks, matching the dimensions of their respective satellite images, employ just two values: one for road pixels and another for background pixels. Our custom dataset draws from five open resources that fulfill our criteria.

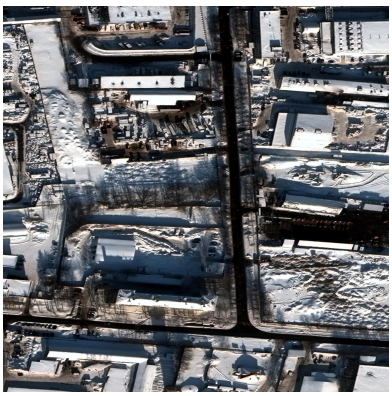
The initial dataset we selected comes from the DeepGlobe 2018 challenge [22]. This challenge encompasses classification, detection, and segmentation tasks for satellite images and presents three datasets accordingly. We are specifically interested in the dataset linked to the segmentation challenge, which involves extracting roads and street networks remotely from satellite images. The DeepGlobe road dataset consists of 8,570 images, each with dimensions of 1024x1024 pixels, captured over Thailand, Indonesia, and India at a high resolution of 0.5 meters per pixel. We will not use the testing set, as it lacks segmented masks. Therefore, we will work with 7,469 images from this dataset. Our second dataset is the Massachusetts Roads Dataset, a classic benchmark for road segmentation models introduced in [21]. It comprises 1,171 images, each measuring 1500x1500 pixels, and covers a variety of urban, suburban, and rural regions within the state of Massachusetts. The images have a resolution of 1 meter per pixel. We have also included data from the SpaceNet program, which provides an archive of readily accessible imagery [23]. To be more precise, we have collected data from both the SpaceNet Challenge 3 and SpaceNet Challenge 5, as both challenges were centered around road network extraction. This provides us with 5,151 aerial images, each measuring 1300x1300 pixels, captured over six cities: Las Vegas, Paris, Shanghai, Khartoum, Moscow, and Mumbai. The image resolutions span from 0.5 meters to 1.3 meters per pixel.

We then augment our data with two smaller datasets, in order to cover diverse regions and image resolutions. The first dataset we use is presented in [24], where a network capable of extracting road centerlines is introduced. This dataset comprises 224 images, with dimensions ranging from 600x600 to 800x1400 pixels, all having a spatial resolution of 1.2 meters per pixel. The second supplementary dataset, introduced by [25], encompasses 21 satellite images that cover the urban area of Ottawa. These images are relatively large spanning from 1300x2000 to 6500x5000 pixels. They have the highest spatial resolution within our selected dataset at 0.21 meters per pixel.

By merging these five resources, we gain access to a rich and diverse dataset to train our upcoming segmentation model. Figure 3 presents samples from each source. Our custom dataset includes aerial images from both urban and non-urban areas, spanning various settings across multiple continents as shown in Figure 4, with resolution ranging from 0.21m to 1.3m per pixel. Due to varying image dimensions in our dataset, we have opted to partition each image into smaller sections of size 512x512 pixels. This pre-processing step will also simplify subsequent training. We have excluded any corrupted images, yielding a final dataset comprising 53,146 satellite images, each accompanied by its corresponding segmented road network mask. We then split this dataset into a train set containing 41,186 images (77%), a validation set with 6,952 images (13%) and a test set with 5,008 images (10%). A table is available in the appendix summing up the contribution of each of our sources to our final dataset.

(a) DeepGlobe dataset

(b) Liu et al. dataset



(c) SpaceNet dataset

(d) Chen et al. dataset

(e) Massachusets Roads dataset

Figure 3: Examples of images from our various datasets demonstrating a range of settings and image resolutions.

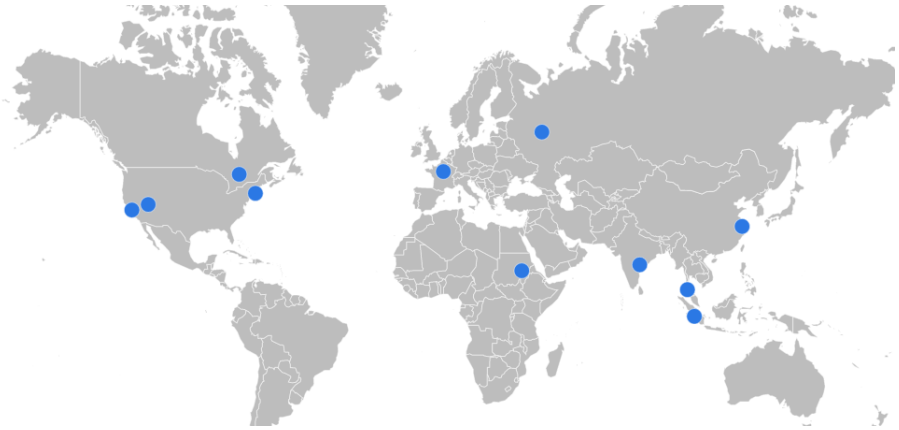


Figure 4: Locations of the areas covered in our custom dataset.

2.3 DeepLabv3+, a segmentation model

In the following, we propose to break down the main components of DeepLabv3+ that we selected to perform the task of extracting road networks on satellite images. The first specificity of our chosen model is that it possesses an encoder-decoder architecture. Images have traditionally been approached with neural networks using convolutional layers [26]. These layers produce feature maps, which are typically of lower dimension than the inputs. The goal of these feature maps is to extract features from the images that allow the network to learn how to recognize specific patterns and objects. However, a segmentation task requires outputting a segmented

image of the same dimensions as the input. This presents a challenge when relying solely on a sequence of convolutional layers, since the learned feature maps are of smaller dimensions. This is why an encoder-decoder architecture is efficient to perform segmentation.

The encoder part is in charge of learning to extract the important features of the image. It usually uses a pre-trained network containing a series of convolutional layers. Figure 5 illustrates, for example, the original ResNet architecture [27], which is a commonly used encoder. After that first section, the decoder part takes over and maps back the extracted features to the size of the original image through what is called an upsampling process.

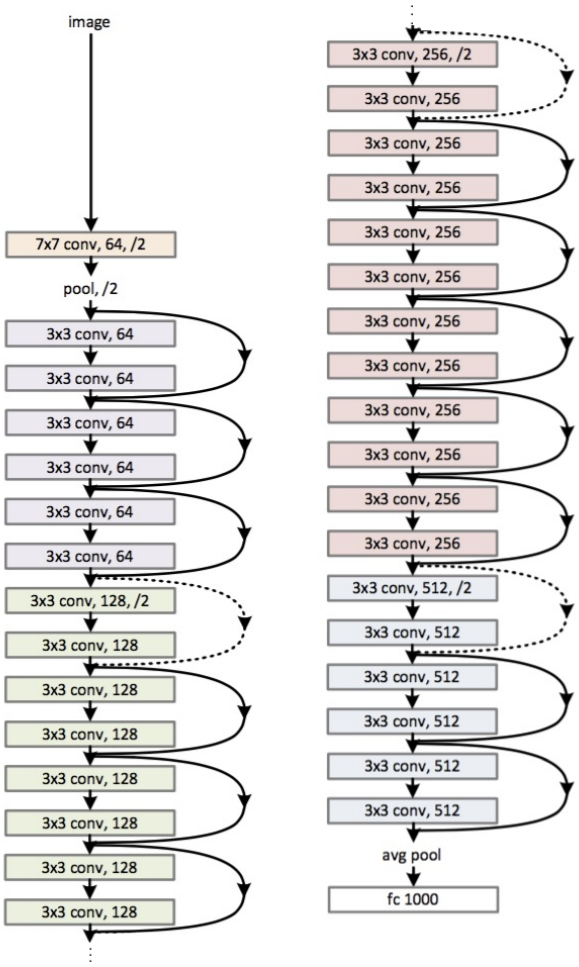


Figure 5: the ResNet original architecture with 34 layers. *Source : [27]*

One of the key component used in DeepLabv3+ is atrous convolution. Before the introduction of the DeepLab architecture, the main method to perform upsampling was to use deconvolution layers, which perform an operation similar to convolution but in reverse [28]. Instead of reducing the spatial dimensions, it increases them by using a filter that is larger than the input and applying a transposed operation. However, this process consumes additional memory, and the subsequent mapping back into the original image results in sparse feature extraction [17]. Atrous convolution offers a simple but powerful alternative. Also referred to as dilated convolution, atrous convolution closely resembles classic convolution by featuring parameters such as a kernel and stride. However, it introduces an additional parameter called the dilation rate. This rate controls the spacing between values within the kernel, achieved through the insertion of zeros. The complete atrous convolution formula is available in the appendix. Figure 6 showcases the difference between a 3x3 kernel of a convolution layer and a 3x3 kernel of an atrous layer with dilation rate of 2.

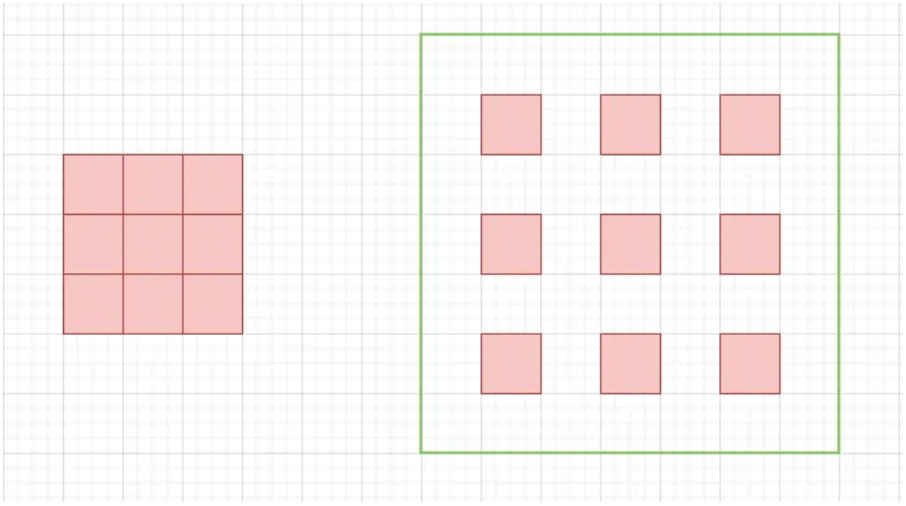


Figure 6: Difference between a 3x3 classic kernel and a 3x3 atrous kernel with a dilation of 2. *Source : [29]*

Atrous convolution enables us to capture larger input regions and control the spatial resolution of the output, all without increasing the kernel size parameters. The first DeepLab version used a pre-trained deep convolutional neural network (DCNN) to extract features, but the authors removed the downsampling operators from the last few max-pooling layers of the pretrained network and instead upsampled the feature maps using atrous convolutional layers [30].

On top of atrous convolution, another key component of DeepLabv3+ is the Atrous Spatial Pyramid Pooling (ASPP) introduced by the second iteration of the DeepLab series. This feature is inspired by the Spatial Pyramid Pooling (SPP) [31], but it is modified to integrate atrous convolution. The idea behind ASPP is to simply combine multiple atrous convolutions with different dilation rates on the same input in order to capture objects at multiple scales.

DeepLabv3+ builds on top of all the components introduced earlier and fuses them together while adding some innovations [16]. First of all, DeepLabv3+ incorporates components from the DeepLab series into an encoder-decoder architecture that has proven to be efficient for semantic segmentation tasks. The encoder block of the model uses the DeepLabv3 architecture, which means that the input goes through a DCNN employing atrous convolution. The output is then fed into the DeepLabv3 ASPP described earlier.

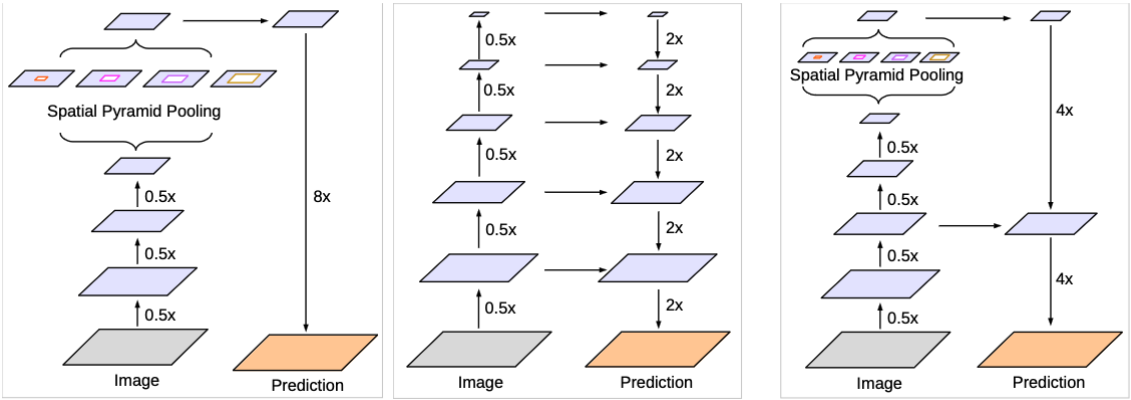


Figure 7: Traditionnal architecture with SPP followed by an encoder-decoder architecture and their merging introduced by DeepLabv3+. *Source : [16]*

The final key component behind the efficiency of DeepLabv3+ is a method called depthwise separable convolution [32] that significantly reduces the overall computational complexity of the model. Depthwise separable convolution breaks down the traditional convolution operation into two distinct steps: the depthwise convolution and the pointwise convolution. This process considerably reduces the complexity of convolutions by using fewer parameters and fewer computations to achieve the same result. DeepLabv3+ adapts the same approach to atrous convolutions to achieve what the authors have called atrous separable convolution.

The full architecture of DeepLabv3+ can be seen in Figure 8 where we easily distinguised the encoder and decoder block with the features described earlier. The DCNN part of the encoder, also called the backbone, can be chosen from a variety of pre-trained existing networks. In this work, we decided to test two different types of backbone for our segmentation model. The first one is ResNet101, which is a variant of the ResNet architecture [27] with 101 layers, and the second one is Xception71, which is a variant of the Xception architecture [33] with 71 layers. A diagram of Xception's architecture is available in the appendix.

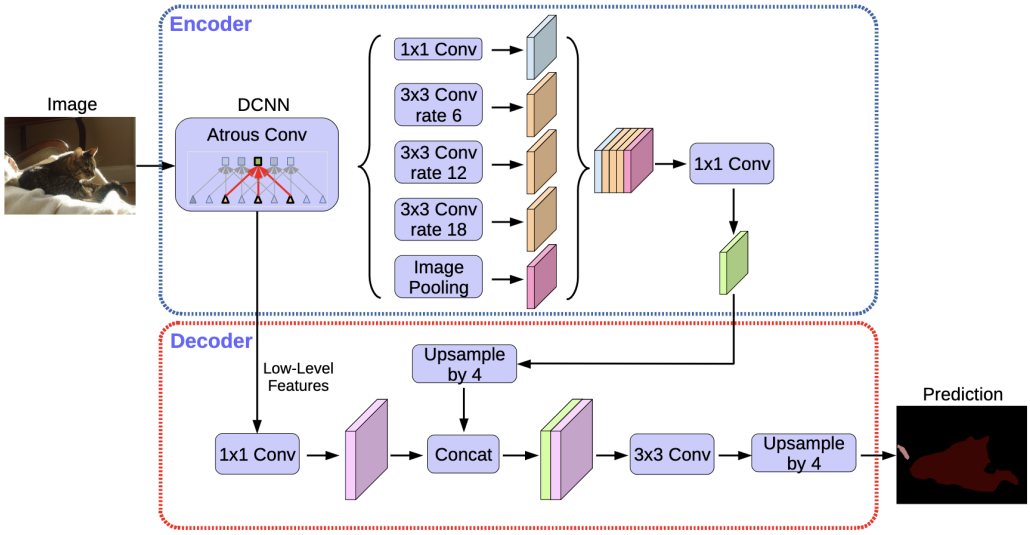


Figure 8: Deeplabv3+ full encoder-decoder architecture as presented in the original paper. *Source : [16]*

2.4 Training methodology

We will now proceed with a description of our methodology for training an efficient road extraction model based on DeepLabv3+. The inputs to our model will be the aerial images in our custom dataset of size 512x512 pixels with three RBG channels corresponding to the level of red, green and blue in each pixel. On the other end of our model, the outputs will be images of the same size, but with only one channel representing our binary classification result.

We implemented DeepLabv3+ using the open-source segmentation models repository [34], which contains several popular models coded using the deep learning library PyTorch [35]. This repository facilitates easy access to the architecture in a modular fashion, simplifying the testing of various backbones. All training was conducted using the classic minibatch approach, meaning we fed the model a minibatch of images before computing the loss between our prediction and the ground truth mask. We then computed the gradient of our network parameters using backpropagation, before updating the weights accordingly to minimize our selected loss. We used the Adam optimisation algorithm [36] in all our training and experiments. We will discuss later how we chose the learning rate, since it is a hyperparameter of our model. All experiments were conducted on a NVIDIA GeForce RTX 4090.

2.4.1 Loss

We now turn to the choice of the loss that we will try to minimize during training. Since our goal is essentially to perform binary classification on every pixel of the image, the Binary Cross-Entropy (BCE) loss is a natural choice. However, even if the BCE loss is used a lot for classification tasks, it fails to capture the full complexity of a segmentation task, especially the spatial aspect of the problem. A criterion that is often used to address this issue is the Dice loss, inspired by the Dice coefficient. Given two sets X and Y, the Dice coefficient measures their similarities using the following formula:

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

Since we are comparing a set representing the ground truth and a set being our prediction, the coefficient can be expressed in terms of true positives (TP), false positives (FP), and false negatives (FN):

$$D = \frac{2TP}{2TP + FN + FP}$$

The Dice loss L_D is simply the complement of the Dice coefficient. Using this loss pushes the model to output binary masks that have a higher overlap with the ground truth masks. We opted for a hybrid approach, incorporating both the BCE loss and the Dice loss. While the BCE loss centers on pixel-level precision, the Dice loss emphasises the overall broader alignment of masks. This dual strategy assists the model balance between individual pixel predictions and capturing the overall context of the segmentation. We are also confronted with a highly imbalanced dataset, where road pixels are notably rare in comparison to background pixels. The Dice loss is particularly efficient at handling class imbalance, as it takes into account both false positives and true positives. Therefore, the loss we will aim to minimise can be written :

$$L = \lambda L_{BCE} + (1 - \lambda) L_D$$

Both expressions for L_{BCE} and L_D in our specific context can be found in the appendix. We set $\lambda = 0.25$ drawing inspiration from [37], which utilises the same hybrid loss with success to train a U-Net segmentation architecture for road extraction in satellite images.

2.4.2 Finding the optimal setting for training

We have previously discussed one hyperparameter of our model, namely the learning rate. The learning rate is crucial as it significantly influences our convergence. While it is a common hyperparameter in deep learning tasks, we are also facing choices when it comes to picking our training strategy. Indeed, we have to pick the Deep Convolutional Neural Network (DCNN) acting as a backbone in the encoder part of DeepLabv3+. As explained earlier, the choice of the backbone is important because it directly influences the model's ability to capture relevant and informative features from the satellite images. We opted for two possible backbones, which are called ResNet101 and Xception71.

The choice of Xception was logical, as the authors of DeepLabv3+ employed the Xception architecture as a backbone network to achieve State-Of-The-Art (SOTA) performance on various benchmarks. Xception draws inspiration from the Inception architecture [38], incorporating depthwise separable convolutions that we previously mentioned, along with additional skip connections. The components of the Xception architecture

encompass convolution layers, depthwise separable convolution layers, Rectified Linear Units (ReLU) as activation functions, and several pooling layers. We decided to opt for Xception71, which is an Xception-based model featuring 71 layers and has been open-sourced by the PyTorch Image Models library [39]. This backbone consists of 42 million parameters.

We also decided to evaluate an alternative backbone. Despite Xception's proven efficiency as a backbone for DeepLabv3+, we sought to exploit the modular nature offered by the open-source implementation of DeepLabv3+ to investigate whether a DCNN with a distinct architecture could better capture the unique features of satellite images. As a result, we opted for a ResNet-based architecture [27], given its widespread adoption as a backbone in computer vision tasks. Our selection was ResNet101, an architecture comprising 101 layers and 45 million parameters.

Now, we are confronted with a third choice, as there are essentially two approaches to train our segmentation model. The first option involves training the model end-to-end from scratch. In this scenario, we input our data, compute the loss, and update all existing weights accordingly. The other possibility is to leverage a pre-trained backbone and freeze its weights before training the rest of our model. This approach, known as transfer learning, aims at enabling our DCNN to already be able to extract meaningful features of our aerial images. Therefore we collected pre-trained weights for both ResNet101 and Xception71. These weights were obtained by training both networks on the ImageNet dataset [40], achieving classification accuracies of 77.37% for ResNet101 and 79.88% for Xception71. We will test both training strategies: the first involves training from scratch, while the second leverages transfer learning by importing weights for the backbone and training the rest of our model.

We need to determine the optimal choices leading to the best performance. Our possible settings encompass all combinations of our backbone choice (Xception71 or ResNet101), our training approach choice (from scratch or using pre-trained weights), and our learning rate choice.

2.4.3 Using Bayesian optimisation

Finding the optimal settings for a deep learning model is typically a difficult task. The space of potential parameters can be extremely large, and exploring all possible combinations is generally not feasible due to resource and time constraints. In our case, the parameter space for the backbone choice and the training strategy is relatively limited since both have only two possible values, yielding only four possible combinations. However, for the learning rate, we have chosen to explore values ranging from 10^6 to 10^2, yielding an infinite number of possible values.

Various strategies exist to discover the configuration that results in the highest overall model performance. The most naive one, which is known as the grid search, consists in an exhaustive search over all the possible combinations of the selected hyperparameter space. Even if this method guarantees optimal results, it is often impractical or even not appropriate in the case of an infinite parameter space like ours. Therefore, a widely used strategy is the random search [41], which consists in testing randomly selected combinations of parameters. Even if this approach seems suboptimal compared to the grid search, it has proven to be surprisingly efficient. By randomly exploring the parameter space, the random search often encounters a combination that is near optimal and that the grid search would require significantly more time to discover. However, in this work we decided to use another strategy to find the best combination of our parameters that has become increasingly popular in recent years: Bayesian optimization (BO) [42].

As its name suggests, Bayesian Optimisation is founded on the Bayesian paradigm and employs observed data to guide the optimisation search. A BO algorithm has two key components: a probabilistic model for the function we are attempting to minimise and an acquisition function that determines where to sample next in

the hyperparameter space. The probabilistic model, which serves as an approximation of the true objective function, is referred to as a surrogate model. This surrogate model furnishes a posterior distribution that offers insights into potential values for the objective function. This posterior distribution is then updated each time we evaluate a new set of parameters for our objective function. Subsequently, the acquisition function, also known as the selection function, comes into play. It serves as the criterion to maximise when deciding which point to assess next, utilising the predictions and uncertainty estimates provided by the surrogate model. A general BO algorithm can be outlined as follows :

Algorithm 1 Bayesian Optimisation

1: **Initialisation** We select a surrogate model, which will be the prior to model the function we want to minimise. We evaluate the objective function on an initial configuration. We fix the number of total trials for the algorithm and we denote this number by N.
2: **while** $i \leq N$ **do**
3: We update the posterior on the objective given all the evaluations available.
4: We compute the acquisition function with the updated posterior and we find the setting $\mathcal{S}_0$ that maximises the updated acquisition function.
5: We evaluate the objective function with setting $\mathcal{S}_0$
6: i = i + 1
7: **end while**
8: Return the configuration that yields the minimum value of the objective.

Within the family of BO, there are multiple algorithms available that use different combinations of surrogate models and acquisition functions. We decided to use the Tree-structured Parzen Estimator (TPE) algorithm [43] since it has demonstrated effectiveness in similar settings. Details of how the algorithm works are given in the appendix. We implemented it using the HyperOpt library [44]. Nonetheless, we cannot efficiently apply TPE directly to our custom dataset, as it remains too large to support repeated training processes. Therefore, we created a smaller dataset, representing approximately one-fifth of the 50,000-image dataset we compiled. Our assumption is that this lightweight version containing around 10,000 images will suffice to identify the best-performing configuration for our model. We chose to fix the number of TPE algorithm evaluations at 35. Each evaluation involves training our model on the lightweight dataset with the set of parameters selected by the TPE algorithm. The objective function measures the validation loss of the model after training for 4 epochs. We selected this number of epochs based on empirical observations, noting that it was sufficient to distinguish the best-performing models from the others.

2.5 Experiments results

2.5.1 Bayesian optimisation results

The entire search process took approximately 11 hours, running on an NVIDIA GeForce RTX 4090. The results of the search are depicted in Figure 9.

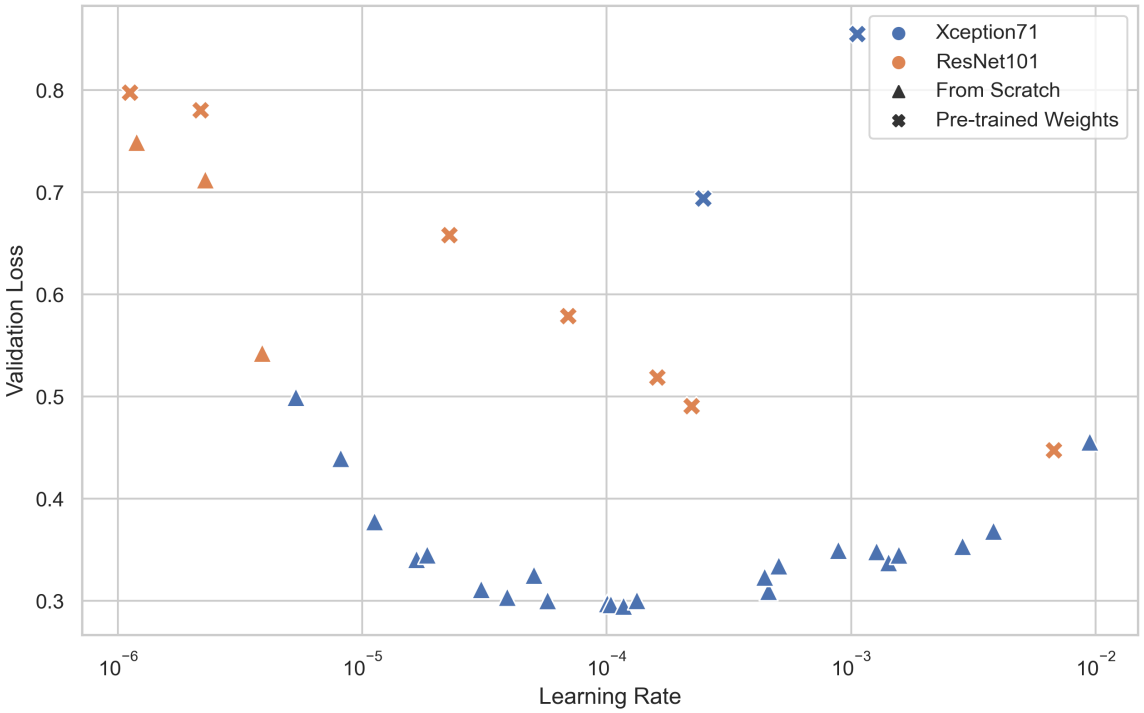


Figure 9: Validation loss against learning rate for various combination of our parameters.

The most notable observation is that training the model from scratch proved to be significantly more efficient than utilising pre-trained weights. Not a single evaluation using pre-trained weights managed to achieve a validation loss lower than 0.4. While this may seem surprising initially, it is a logical outcome considering the specific task we are attempting to solve. We are working with satellite images, whereas the weights we use in the pre-trained setting originate from models trained on ImageNet. Satellite images exhibit different types of features compared to traditional images, which typically capture profile information of objects. Figure 10 showcases the disparities in images for vehicles, ships, and airplanes between the previously mentioned PASCAL VOC dataset and aerial images extracted from the DIOR dataset, which we will use later [45]. This illustrates why effectively applying transfer learning within the context of satellite images is challenging, and our results reflect this challenge.

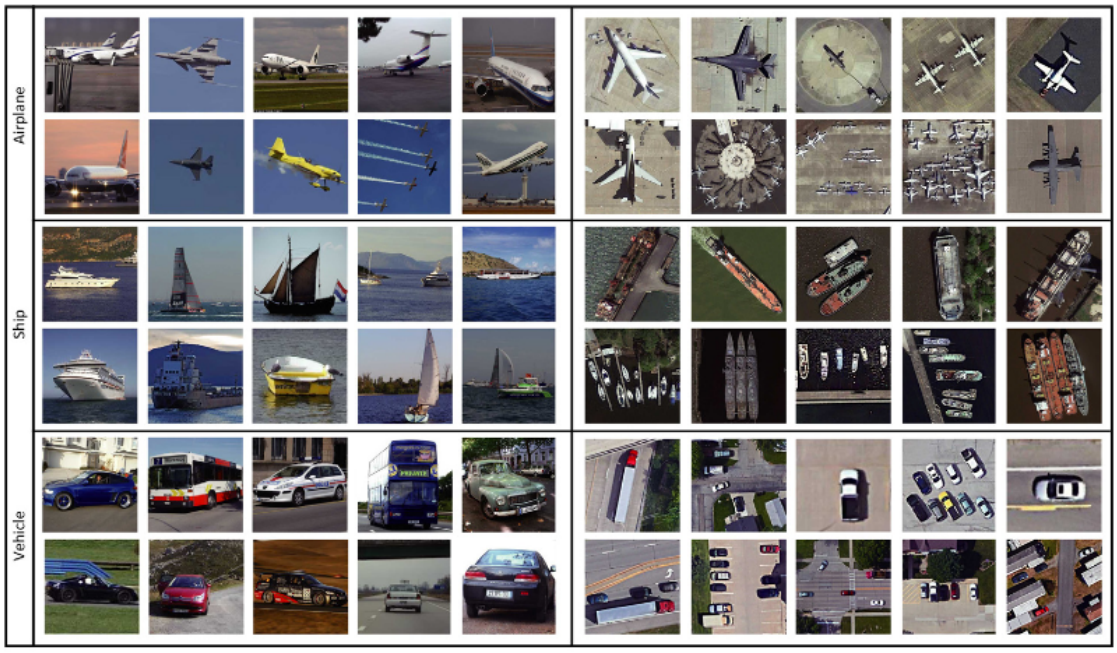


Figure 10: Difference between images from the PASCAL VOC dataset (left) and the DIOR dataset (right) for the same objects. *Source : [45]*

We also notice that Xception71 as a backbone always led to a lower validation loss. Therefore, we accordingly rule out ResNet101 as a possible backbone for our final model. Concerning the learning rate, the TPE search has also proven to be quite efficient. The Bayesian optimization approach quickly helped identify a region between 10^{-3} and 10^{-5} that was producing the best results. From then on, the algorithm was able to identify the optimal learning rate for our model, which is a value that can be roughly rounded to 10^{-4}.

2.5.2 Training results

Using the outcome of our Bayesian optimisation search, we trained our final model from scratch. We employed Xception71 as the backbone and set the learning rate to 10^{-4}. The model underwent training on the training set of our 50,000-image custom dataset for 25 epochs, with the validation set used for monitoring performance. The loss function used is the custom loss we described earlier, which combines the BCE Loss and the Dice Loss. The entire training process on our custom dataset took approximately 15 hours. Throughout the training process, we continuously tracked both the training loss and the validation loss, which are depicted in Figure 11.

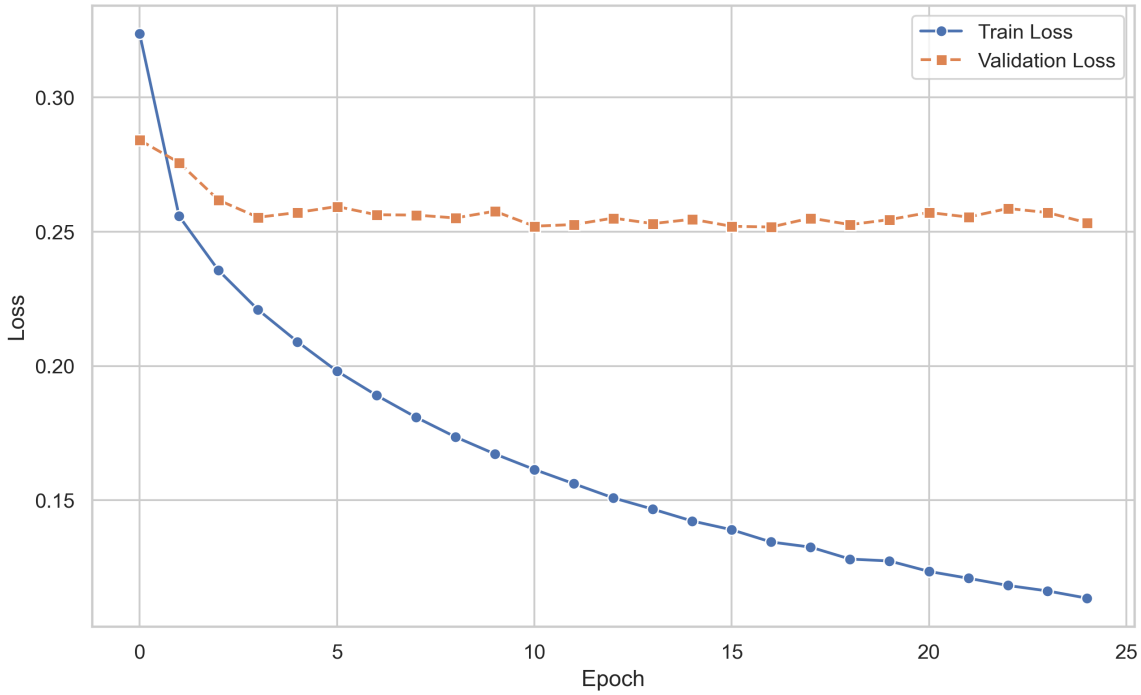


Figure 11: Training loss and validation loss against epochs for our DeepLabv3+-based segmentation model on our custom dataset.

There are several interesting insights to draw from the figure summarising the training stage. First, we can see that the training loss is steadily decreasing at each epoch which indicates that our model is effectively learning from our satellite images. However, the rate of reduction in the training loss diminishes with each epoch. For example, the training loss decreases from 0.33 to 0.25 after the first epoch, whereas it only decreases from 0.146 to 0.142 by the 15th epoch. This behaviour makes sense as the model has gradually less to learn from the training data as the training progresses. The validation loss curve is more complex to interpret. Initially, we can clearly discern the validation loss steadily decreasing during the initial epochs. However, after 5 epochs, the validation loss starts oscillating around 0.25 without any significant decrease. Despite this relative stagnation, we observe the lowest validation loss at the 17th epoch, where it reaches 0.251. There are several possible hypotheses that can explain such behavior for our validation loss. The interpretation that we find most plausible is that our model, due to its complexity and the diversity of our dataset, quickly reaches its optimal performance after a few epochs. It then becomes robust enough to avoid overfitting the training data to the extent that it would result in a significant increase in the validation loss. Once the training stage was completed, it became evident that we needed to conduct further analysis on our trained model to assess its performance and validate our assumptions.

2.5.3 Performance on the test set

BCE Loss, Dice Loss, and their combination are not the only ways to assess the performance of image segmentation. Instead of the losses we mentioned, other metrics evaluate segmentation accuracy positively, meaning the higher the value, the better. We discussed earlier how the Dice Loss is based on the complement of the Dice Coefficient. Thus, it is natural to use the Dice Coefficient as a performance metric. When employed as a segmentation metric, the Dice Coefficient is often referred to as the F1-Score since it is computed using precision ($\frac{TP}{TP+FP}$) and recall ($\frac{TP}{TP+FN}$) of a prediction. Another widely used metric is the Intersection over Union (IoU), also known as the Jaccard index. The Jaccard Index for two sets X and Y is simply given by:

$$\frac{|X \cap Y|}{|X \cup Y|}$$

In the context of semantic segmentation, using the same confusion matrix notations (TP, FP, and FN), we derive an expression similar to the F1-Score

$$\frac{TP}{TP + FP + FN}$$

We use our trained model to extract the road network in each image of the test set, resulting in a predicted mask. Subsequently, we employ the ground truth mask to compute the confusion matrix of our predictions, which contains the values used in the Dice Coefficient and the IoU. The obtained confusion matrix is visible in Figure 12. It turns out that True Negatives represent nearly 94% of our model's predictions, which is a logical result considering the heavy class imbalance in our dataset, where background pixels are represented by the value 0. The errors of our models, False Positives and False Negatives, occur in roughly the same proportion, indicating no clear trend in the errors. We then computed the F1-Score and the IoU on the test set, resulting in values of 0.70 and 0.54, respectively.

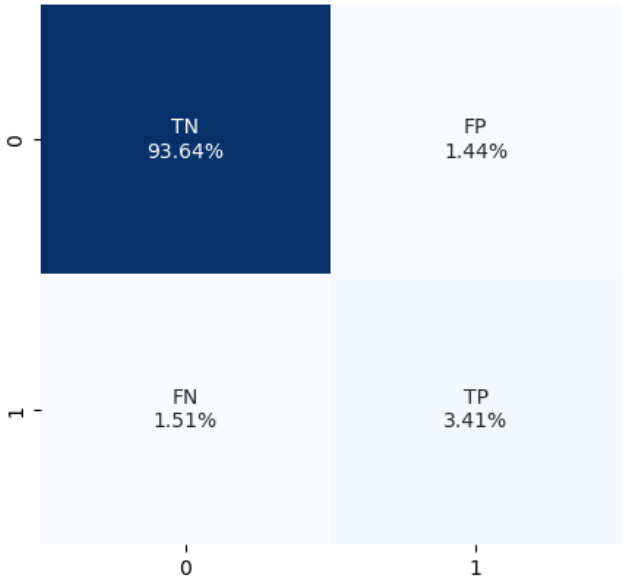
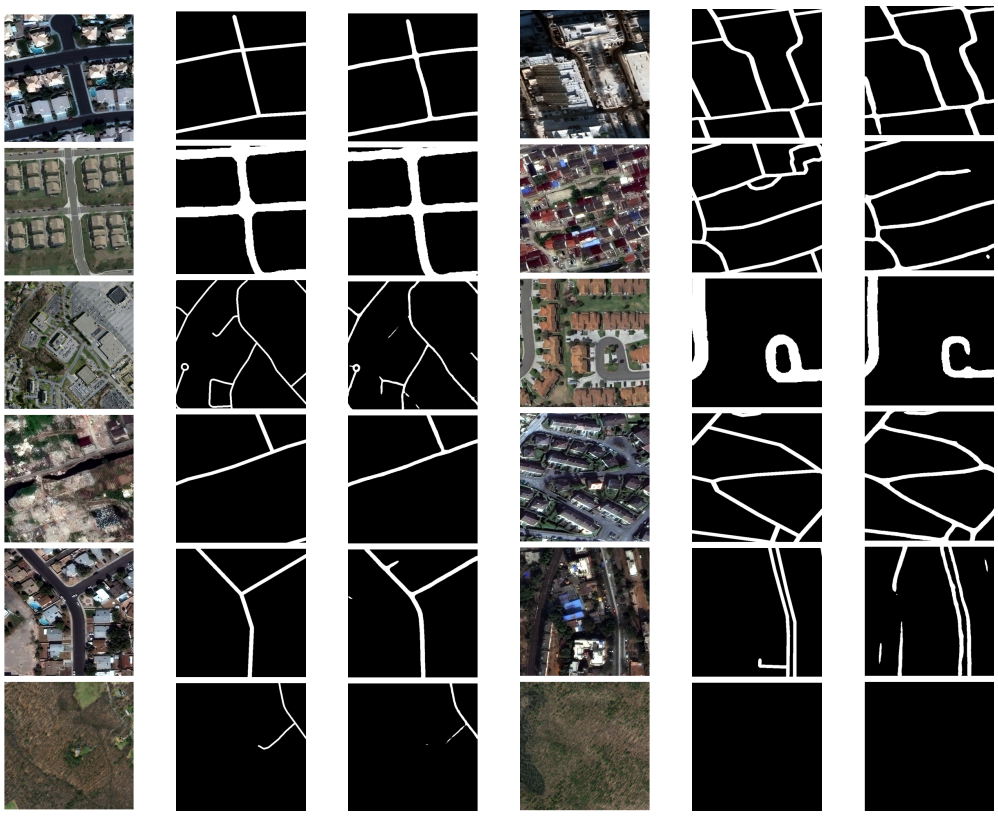


Figure 12: Confusion matrix for our prediction.

Although metrics provide useful quantitative insights into the performance on the test set, it can be even more insightful to visualise the segmented masks and compare them to the ground truth. Figure 13 illustrates the model's outputs for several satellite images in the test set. The model proves to be efficient in most cases, achieving segmentation very close to the ground truth. It can not only correctly classify straight lines, as seen in the first rows, but also handle more complicated network patterns with multiple junctions and various angles. The model is also capable of detecting areas with few or no roads, as showcased by the predictions in the images of the last row. Moreover, we have identified some weaknesses of the model. It sometimes generates noise that results in pixels being labelled as roads when they are not connected to any road segment. Our trained model can also confuse certain objects for roads, as illustrated by the second image in the penultimate row. In this image, a stretch of road runs alongside railroad tracks, which appear very similar to roads in aerial imagery.



(a) (b) (c) (d) (e) (f)

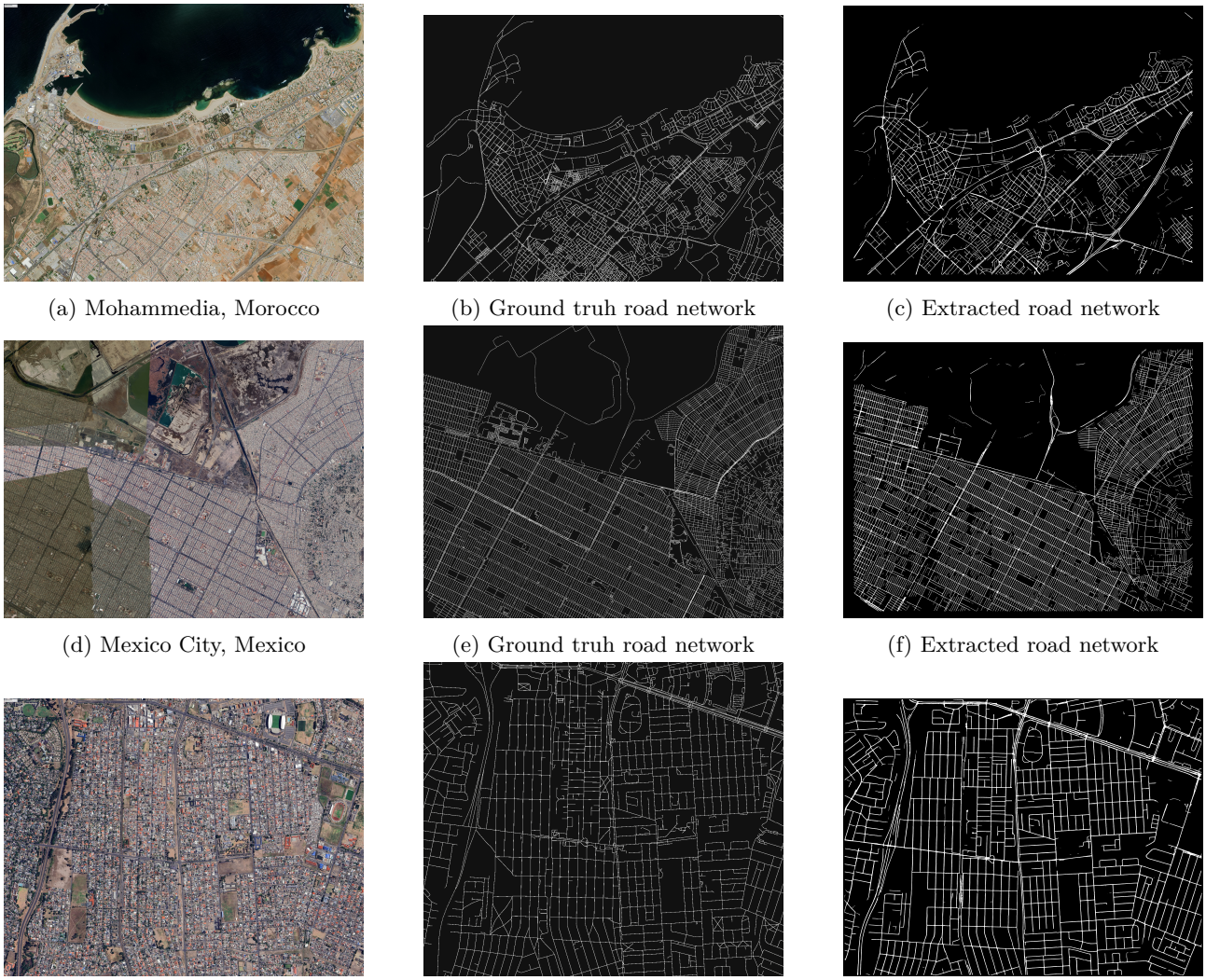
Figure 13: Images from our test set are depicted in a) and d), accompanied by their respective ground truth masks shown in b) and e), and our predicted road network in c) and f).

Consequently, our model incorrectly classifies some parts of the rails as roads. This visual inspection provides possible explanations for the classification errors observed in the confusion matrix.

2.5.4 Performance on selected areas

One of the strengths of our model is that it has been trained on a very diverse dataset covering numerous regions of the world. To test the robustness of our model on satellite images not linked to the datasets we relied on to build our training set, we collected our own test set, including regions around the world our model has not been trained on. We used Google Earth as our primary source, as it offers freely accessible high-resolution satellite imagery. We selected three different locations: Mohammedia in Morocco, Mexico City in Mexico, and Cape Town in South Africa. We chose locations in Africa and South America since they are the least represented continents in our custom dataset. We also ensured that the selected locations contained very different topography and elements. The satellite image of Mohammedia incorporates a significant portion of the coast, with the city's roads winding and not following a precise grid pattern. The image of Mexico City showcases an extremely dense grid-shaped road network alongside another area containing concentric lines. It also includes a large natural area with a lake and almost no construction. Finally, the Cape Town image focuses on a residential neighborhood with roads laid out in a grid pattern.

We needed to obtain the ground truth road networks of our selected areas for comparison with our model's predictions. We decided to use data from OpenStreetMap (OSM), which is a freely available open-source geographic database. More specifically, we utilised the OSMnx Python package [46], which provides tools to easily download and visualise street networks provided by the OSM community. Figure 14 presents the satellite images for each selected location, along with the ground truth road network and our trained model's predictions



(a) Mohammedia, Morocco (b) Ground truh road network (c) Extracted road network

(d) Mexico City, Mexico (e) Ground truh road network (f) Extracted road network

(g) Cape Town, South Africa (h) Ground truh road network (i) Extracted road network

Figure 14: Comparisons between the ground truth road network and the one extracted using our segmentation model at three selected locations.

on the images. This experiment with our own satellite images demonstrates that our model is capable of segmenting almost the entire street network in dense areas of major cities around the world. It can handle various settings and urban configurations and correctly ignore natural landscapes, even when they are close to constructed areas.

3 Detecting infrastructure

3.1 Overview of the literature and contributions

We now proceed to the second component of our final pipeline. This element is responsible for the detection and precise mapping of various infrastructure within the city. As previously emphasised, infrastructure have a considerable influence over the economic well-being of urban areas. Consequently, we anticipate that identifying specific infrastructure types, as we will elaborate on shortly, will yield valuable insights into a location's wealth status. To achieve this objective, we will need to perform object detection on satellite images.

Object detection constitutes a separate task from the semantic segmentation discussed in the prior section. Rather than classifying each pixel in an image, our aim here is to detect instances of particular objects within an image class and define bounding boxes around each object to precisely indicate their positions. Our desired output comprises the coordinates of these bounding boxes, along with the corresponding object class. For instance, if we were to detect and locate airplanes within a satellite image of an airport, the desired output would resemble the illustration in Figure 15.

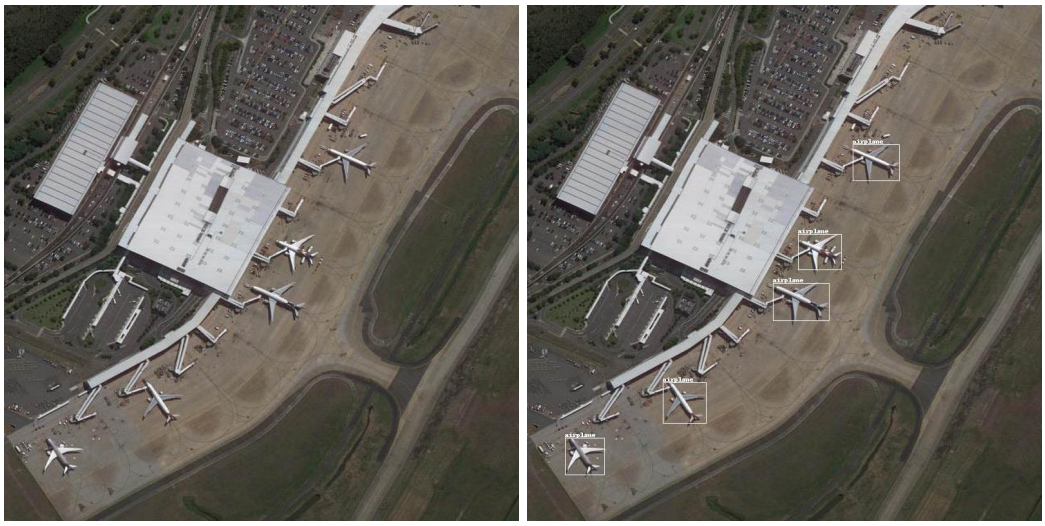


Figure 15: A satellite image of an aiport and the same image annoted with bounding boxes locating airplanes.

Object detection has been one of the most crucial challenge for computer vision since its emergence. Just as we saw with semantic segmentation in the first part of this work, deep learning has widely revolutionised the field, resulting in models that deliver outstanding performance while becoming the standard approach to tackle the task. Therefore, selecting an architecture for object detection can appear daunting given the extensive literature and the multitude of models it has generated [47]. Detection models can be broadly categorized into two groups: two-stage detectors and one-stage detectors. Two-stage detectors, pioneered by the RCNN model [48], marked the early exploration of convolutional neural network (CNN) architectures for object detection. These architectures use a two-step process: first, selecting potential regions of interest using a predefined algorithm, and second, employing a CNN to predict the classes of objects within each selected region proposal. However, in our approach, we opted for a one-stage detector stemming from the the You Only Look Once (YOLO) model [49]. The latter introduced a significant advancement by proposing a single-stage object detection model that can be trained end-to-end, resulting in a substantial increase in overall speed.

Our decision to employ a one-stage detector from the YOLO family was driven by substantial evidence in the literature demonstrating the efficiency of these models in detecting infrastructure in satellite images [50, 51]. We chose to implement YOLOv5 [52], and we will provide the reasons for this decision later in this section. Prior

research has successfully applied this model or slightly modified versions to the task of infrastructure detection in overhead imagery, demonstrating its effectiveness [53], [54]. Our main contribution is the training of YOLOv5 on a curated dataset, featuring infrastructure that were thoughtfully selected for the specific signals we anticipate them to carry regarding a city's wealth. Once again, we have compiled this dataset by amalgamating various freely available sources.

In this section, we will first outline the process of creating our curated dataset for aerial infrastructure detection, discuss why we choose YOLOv5 and its key components, then explain our training methodology, and finally, evaluate the efficiency of our trained model in detecting urban infrastructure.

3.2 Creating a dataset for object detection

We construct an infrastructure detection dataset by relying on two major sources. Our initial source is the DOTA dataset [55], which is well-recognized within the remote sensing community. In particular, we have opted to use the second version of the dataset (DOTA-v2.0), comprising over 2,000 labelled satellite images with corresponding bounding boxes spanning 18 classes. These classes encompass a broad spectrum of objects, from large infrastructure such as airports to smaller entities like helipads or ships. We will delve further into the classes we have chosen to retain when discussing the integration of DOTA with our second selected dataset. Despite its limited image count, the DOTA dataset encompasses images with dimensions ranging from 800x800 to 20,000x20,000 pixels. This diversity allows the dataset's labels to include over 300,000 instances of selected classes, showcasing a wide range of scales and shapes. The DOTA dataset stores labels as text file having the same name as their corresponding images and containing one row for every instance formatted as follows : x_1, y_1, x_2, y_2, x_3, y_3, x_4, y_4, `class`, `difficult`. The eight first points give the coordinates of the vertices of the bounding box, arranged in a clockwise order. The last two text items give, respectively, the class of the object and whether the instance is difficult to detect or not. It can be noted that the presence of the latter is not always present in object detection datasets, and we will remove it during the merging with our second dataset.

The second dataset we have incorporated is the DIOR dataset [45], which we previously utilised to illustrate the differences between objects captured on the ground and from an aerial perspective. This dataset comprises over 23,000 satellite images, each sized at 800x800 pixels, and includes annotations for more than 190,000 object instances. DIOR encompasses 20 object classes, some of which overlap with the DOTA dataset (e.g., airport, bridge, storage tank), while also introducing unique classes (e.g., chimney, dam, train station). Object detection datasets lack a uniform standard for organising labels, thus the DIOR dataset structures its annotations differently than the DOTA dataset storing them in .xml files.

With our objective of predicting the wealth level of a city in mind, we carefully selected the classes to include in both datasets. First, we chose all the infrastructure related to transportation, whether they were transport hubs like airports or large means of transportation like ships. Within the literature, These types of infrastructure have consistently been shown to be key factors of economic well-being [56, 57]. We also decided to track every amenities linked to sports practice, as it could be a strong indicator of wealth differences from one city to another. For instance, we expect a city with numerous golf and tennis courts to be wealthier than a city with only basketball courts. Lastly, we include all infrastructures related to industry, such as storage tanks, dams, and chimneys, as they undoubtedly contain valuable information about the economic activity within an area. Our selection comprises a total of 20 classes, encompassing transport hubs, industrial infrastructure, sports facilities, and more : *Airplane*, *Airport*, *Helipad*, *Harbor*, *Ship*, *Bridge*, *Train station*, *Golf course*, *Baseball field*, *Basketball court*, *Ground track field*, *Soccer ball field*, *Swimming pool*, *Tennis court*, *Stadium*, *Wind mill*, *Container crane*, *Chimney*, *Dam*, *Storage tank*.

In order to merge the two datasets with our desired classes, we needed to save the annotations in a standard format so that DIOR and DOTA annotations, which were stored differently, could coexist. We decided to use a format known as the YOLO format, obviously motivated by our choice of detection model. This format consists of saving the annotations in text file with the same name as its corresponding image, with each row representing an identified instance. Each row has the following format: `class-number x-center y-center width height`. All of the values describing the bounding box represent relative positions and sizes within the image, so they are all between 0 and 1. After converting DIOR and DOTA annotations to the YOLO format, we obtained a dataset with 20,915 images. We further split the dataset into a training set with 16,198 images (77%), a validation set with 3,652 images (17%) and a test set with 1,065 images (6%). Figures 16 presents four samples of satellite images taken from our merged dataset, labeled with their annotations.

Figure 16: Four images taken from our final dataset annotated with their corresponding bounding boxes.

3.3 YOLOv5, an object detection model

We have several reasons for choosing the fifth iteration of the YOLO model. Firstly, we believe our detection dataset is sufficiently diverse, large, and well-annotated, to ensure that our detection task does not fall on the difficult end of the spectrum. Therefore, our criteria for selecting the model extend beyond complexity and accuracy. Speed is also crucial to us because we intend to integrate our infrastructure detection model into a larger pipeline alongside the segmentation component we presented earlier. YOLOv5 appeared to meet all our requirements while being relatively straightforward to apply. It is an efficient fast detection model but it is also particularly convenient to implement in Python through its active open-source repository [58].

In the following, we propose to briefly describe the primary components of the YOLOv5 architecture. The groundbreaking idea that initiated the YOLO series of models was to reframe the object detection problem as a regression task, with the aim of predicting bounding box coordinates and class probabilities. The original YOLO model divides the input image into a grid, with each grid cell responsible for predicting the coordinates of a fixed number of bounding boxes as well as the class probabilities of any detected objects within the cell. In this initial iteration, predictions for each cell are managed by a convolutional neural network drawing inspiration from GoogLeNet [38]. The process introduced by YOLO is summarised in Figure 17, as depicted in the original paper.

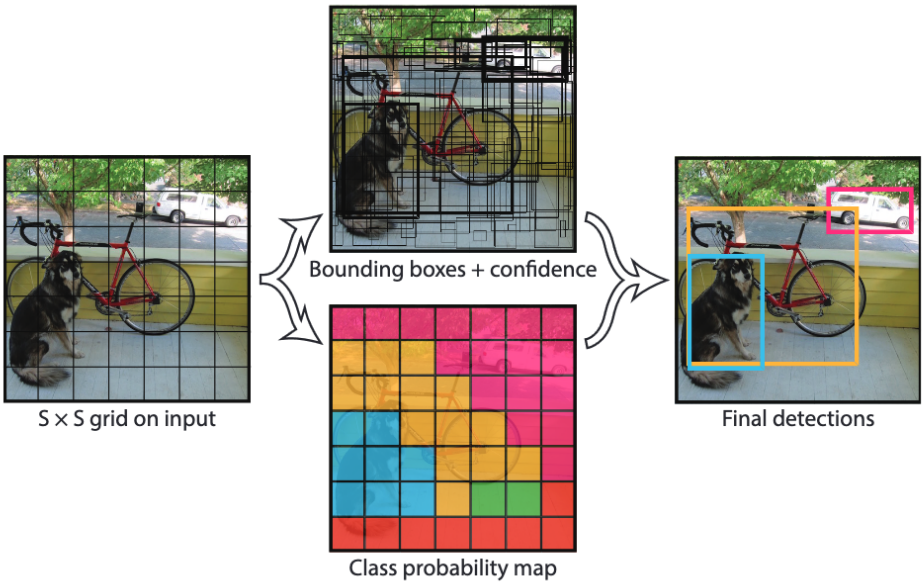


Figure 17: The process used by YOLO to detect objects corresponding to a dog, abike, and a car. *Source : [49]*

YOLOv5 builds on this fundamental approach while incorporating innovations introduced by YOLOv2 [59], YOLOv3 [60] and YOLOv4 [61]. YOlOv5 possesses an architecture made up of three key componenents: the backbone, the neck, and, the head. The backbone plays the same role as usual, extracting insighful features from images using convolutional layers. YOLOv5 uses CSPDarknet53 [62] as a backbone which is the backbone used in YOLOv4. The central component of CSPDarknet53 is the CBS, which stands for Convolutional Batchnorm SiLU. While we have discussed convolutional layers extensively in this work, this is the first mention of batchnorm and SiLU. Batchnorm is widely used in modern deep learning. These components were also present in the DeepLabv3+ architecture, but we chose not to focus on them, as they were not at the core of the model's innovations. Batchnorm is simply a normalisation layer for our mini-batch during training, as explained in [63]. These layers have learnable parameters that enable the layer to adapt its normalisation. Batchnorms are widely used due to their numerous benefits: they accelerate training by normalising values between layers as well as providing a mild regularization effect. The last component of the essential block of YOLOv5 backbone is the SiLU, which stands for Sigmoid Linear Units [64]. It simply multiplies the input with the well-known sigmoid function.

The output of YOLOv5's backbone consists of feature maps of different sizes, which are then passed to the neck responsible for merging these inputs. The neck employs two crucial components: a Feature Pyramid Network (FPN) [65] and a Path Aggregation Network (PAN) [66]. We decided not to delve into extensive detail regarding these two components as they are quite complex and describing them would require a lengthy technical description, which would not effectively contribute to the explanation of how our pipeline was designed.

However, The key takeaway here is that by using these two networks, the neck creates a pyramid containing a hierarchy of feature maps at varying spatial resolutions. This enables the model to capture information about objects at different scales and locations.

The output of the neck is finally fed to the the head, also called the prediction head. This block is responsible for predicting the object class, the bounding box coordinates as well as a class confidence score reflecting the model confidence in its prediction. YOLOv5 uses the head introduced in YOLOv3, which consists of simply three parallel convolutional layers with different sizes, aiming to detect small, medium, and large-sized objects. A simplified architecture of YOLOv5 with its three components is shown in Figure 18 :

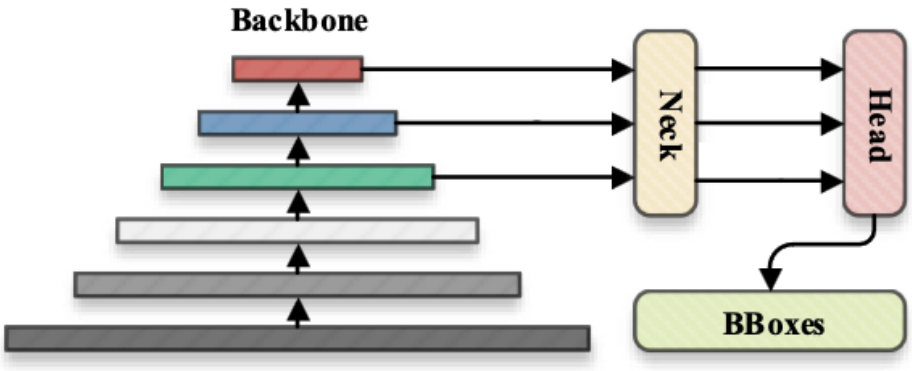


Figure 18: YOLOv5 architecture with the CNN backbone, the neck and the prediction head. *Source : [67]*

3.4 Training methodology

As previously mentioned, we implemented YOLOv5 using the dedicated open-source repository in Python. This repository not only provides us access to the model but also offers a collection of convenient tools for analyzing our results. There are various versions of pre-trained YOLOv5 models available with different sizes and speed-performance trade-offs. We chose YOLOv5x6, which is the largest available version, comprising more than 86 million parameters. We opted for the heavier model due to its capability to handle the largest images, supporting sizes up to 1280x1280, while maintaining relatively fast processing speed. Given the extensive size of this model, we chose to fine-tune a pre-trained version of YOLOv5x6, rather than training our detection model from scratch, as we did with our segmentation model. We will fine-tune it using pre-trained weights on the COCO dataset, which is a large-scale object detection dataset containing over 300,000 images [68].

Since YOLOv5 produces three distinct outputs — bounding box coordinates, object class, and confidence — the loss function employed for training the model comprises a combination of three individual losses, each addressing one of these outputs. The loss function addressing the classification and confidence outputs is the same, which is the BCE Loss. The loss used to assess the accuracy of the model in predicting the coordinates of the bounding boxes is the GIoU loss. GIoU stands for Generalized Intersection over Union and it uses the notion of IoU we already explored earlier. Building on the IoU, it addresses some of its limitations by not only measuring the overlap between bounding boxes but also considering their differences in size and position [69]. It achieves this by using a third box covering the real box and the predicted box, thus measuring their spacing. We introduce the following notations: B for the ground truth bounding box, $\hat{B}$ for the predicted bounding box, and C as the smallest box covering both B and $\hat{B}$. With these notations, we can define the GIoU Loss as:

$$L_{GIoU} = 1 - \frac{|B \cap \hat{B}|}{|B \cup \hat{B}|} + \frac{|C \setminus B \cup \hat{B}|}{|C|} = 1 - \text{IoU}(B, \hat{B}) + \frac{|C \setminus B \cup \hat{B}|}{|C|}$$

We made a slight modification to our dataset before training. Following the recommendations in the YOLOv5 literature, we included background images in the training set, which are images without any instances of our selected classes. We used images from DOTA and DIOR that contained classes we did not select to add these background images to our training and validation sets. After this modification, images without any instances account for approximately 8% of the training and validation sets.

We trained our model using the default configuration provided by the implementation of our YOLOv5x6 model. This default configuration not only spares us from hyperparameter tuning but also automatically selects hyperparameters optimised for our task. It notably determines the optimal batch size based on the available memory space of the hardware used in our experiments. Our model was trained for 25 epochs using the Adam optimiser we employed previously [36], with an initial learning rate of 10^{-3}.

3.5 Experiments results

3.5.1 Training results

The training process took approximately 10 hours. The evolution of the classification loss (BCE Loss) and the loss concerning the bounding box prediction (GIoU Loss) for both the training and validation sets can be observed in Figure 19. Both types of losses indicate efficient training. There is no sign of overfitting as the validation losses steadily decrease while the model gradually learns from the training set.

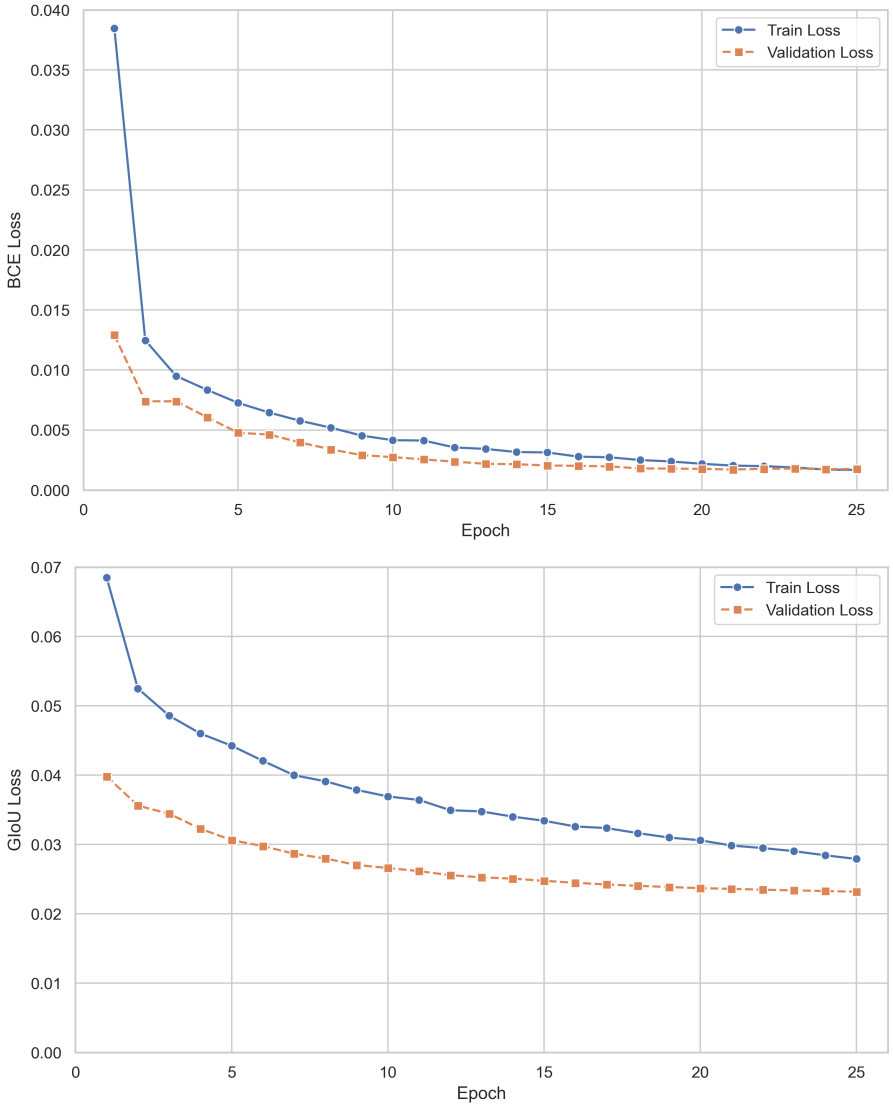


Figure 19: Evolution of object classification and bounding box regression losses during YOLOv5 training on our dataset.

Another way to assess the evolution of our model's performance during training is by using the mean Average Precision (mAP) metric. This widely used metric in object detection is based on recall ($\frac{TP}{TP+FN}$) and precision ($\frac{TP}{TP+FP}$). The first one essentially measures the ability of the model to predict the positives while the second measures how generally accurate the predictions are. We define Average Precision (AP) as the area under the precision-recall curve. With this definition, the mean Average Precision (mAP) is simply the average of the AP values calculated for all the classes. However, one might wonder what constitutes a true prediction for the bounding boxes in the context of object detection. We simply set a threshold on the Intersection over Union (IoU) metric above which we consider predictions for the boxes to be true. Figure 20 shows the evolution of the mAP on the validation set during training, with predictions considered true when the IoU is above 0.5. This indicates that the model effectively learns from the data and suggests that 25 epochs were sufficient to achieve optimal performance, as the mAP starts to plateau around the 20th epoch.

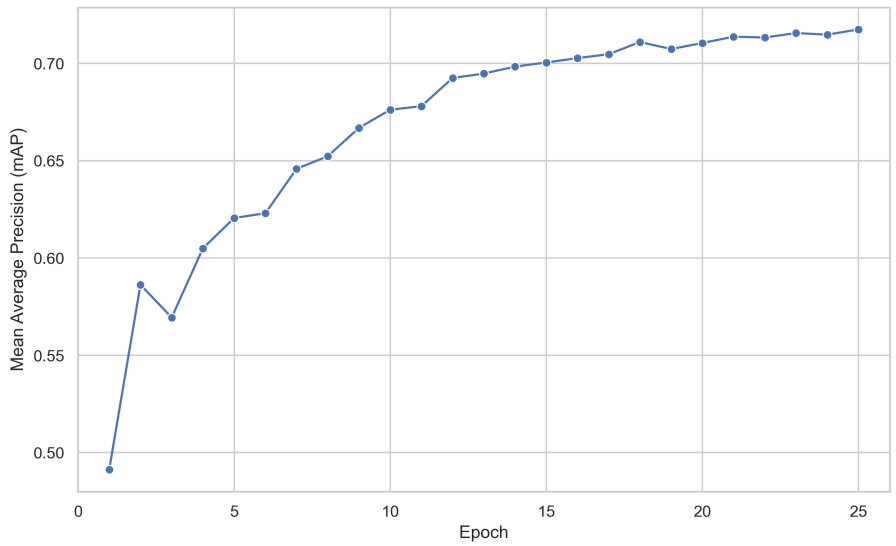


Figure 20: Evolution of the mean Average Precision (mAP) evaluated on the validation set during training.

3.5.2 Performance on the test set

We then evaluate the performance of our trained detection model on our test set. The model achieves a mAP of 0.868 on the test set, indicating its high efficiency in detecting our selected infrastructure. Figure 21 illustrates the Precision-Recall curve used to calculate the AP for each infrastructure class in our test set. It is worth noting that while the model performed well overall, there are variations in performance across classes. For instance, the trained model demonstrated a higher proficiency in detecting airports (with an AP of 0.962) compared to train stations (with an AP of 0.712).

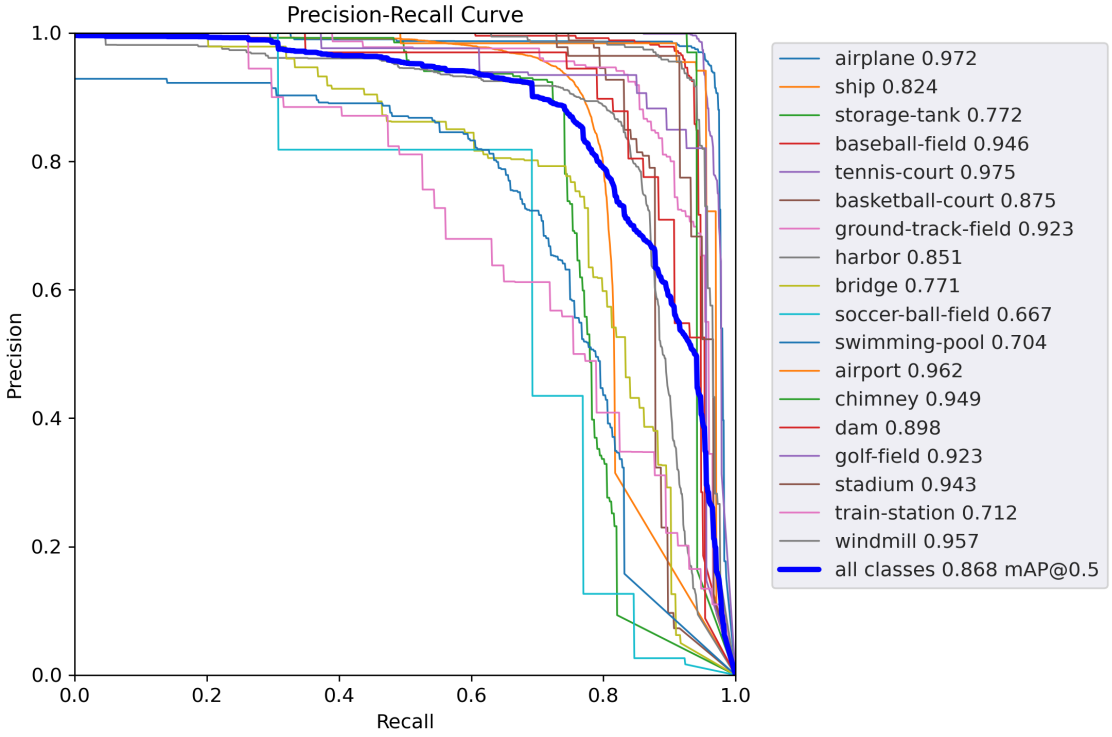


Figure 21: Precision against recall for the prediction of our trained YOLOv5 on the test set.

As we did previously to assess our segmentation model, we also inspect visually the predictions made by our model. The latter proved to be highly efficient and robust, accurately predicting objects with satisfactory bounding boxes in most cases. Figure 22 displays some examples of images from the test set annotated with the model's predictions. The model is capable of detecting objects in close-up settings, such as the image with the storage tanks, as well as in higher-altitude images, as illustrated by the detection of the dam in the first picture.

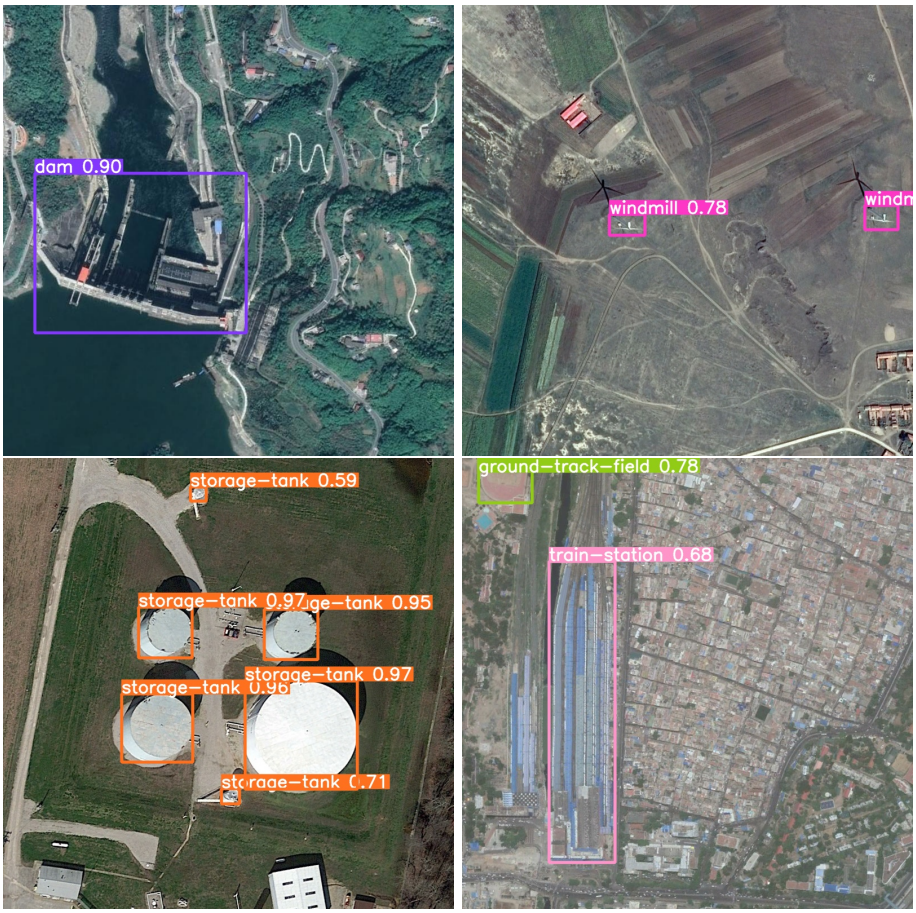
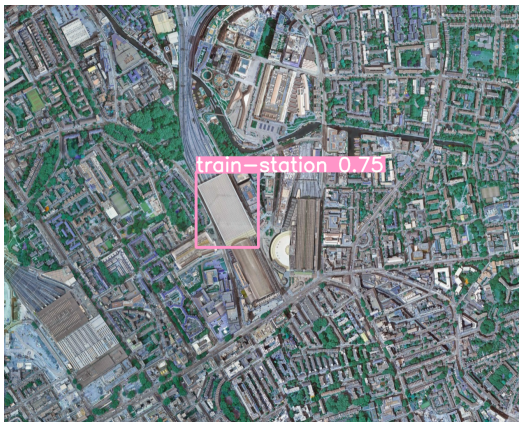


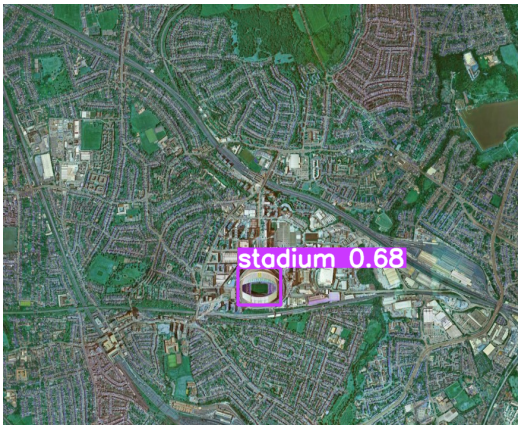
Figure 22: Some images from our test set annoted with the prediction of our model.

3.5.3 Performance on selected areas

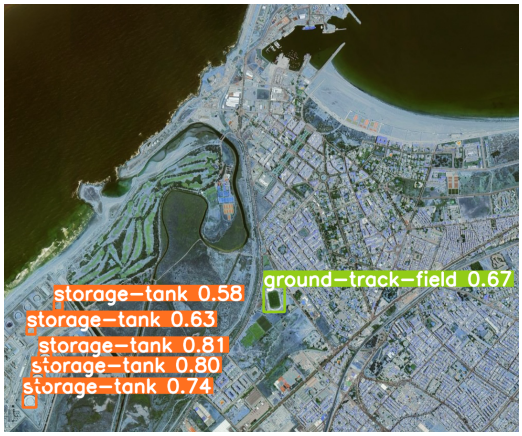
Naturally, we proceed to test the model with our own selected images just as we did with the segmentation model. We chose various locations with different scales, each containing infrastructure of interest that our model was trained to detect. Our selections included two dense urban settings in London: one in close proximity to King's Cross station and the other providing a wider view of the famous Wembley Stadium. Additionally, we decided to reuse a satellite image of the city of Mohammedia in Morocco to assess how the model would perform in a coastal environment. Finally, we captured an image of the Oxford airport to test how the model would perform in a less densely populated urban area. The images labelled with the model's predictions can be seen in Figure 23. The model's predictions are satisfying, correctly detecting King's Cross, Wembley, and Oxford Airport. The Mohammedia example is particularly insightful because this image did not have any specific features that were expected to be detected, unlike the others. We can see that the model has detected storage tanks and a ground track field that we were not previously aware of. This showcases the model's ability to help us identify key infrastructure in satellite images. However, the model failed to detect the golf course that is visible above the storage tanks, indicating that our model is not infallible.



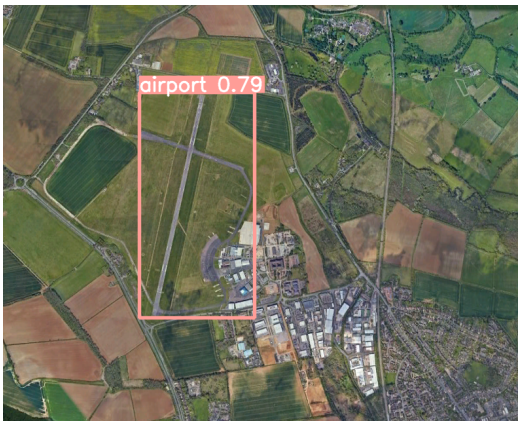
(a) King's Cross train station, London



(b) Wembley stadium, London



(c) Close up on Mohammedia



(d) Oxford Airport

Figure 23: Predictions made by our trained detection model on four selected locations.

4 From satellite images to economics

4.1 Bridging our components to predict the wealth level of an area

This section integrates the two components we designed previously before attempting to use their outputs to predict the wealth level of urban areas. We chose to evaluate our approach using American cities due to the abundance of available city-level economic indicators, which facilitated the collection of income per capita data for every city in our dataset. Additionally, American urban centers are situated in diverse environmental settings, making them an ideal testing ground to assess the robustness of our models. This final part demonstrates that the information extracted from our outputs, namely the road network and infrastructure, provide valuable insights into detecting variations in wealth among cities.

The key contributions of this section encompass the development of a distinctive dataset containing satellite images from diverse US cities, the assembly of our urban infrastructure extraction pipeline, and the design of a neural network architecture capable of predicting income per capita using both road network and infrastructure data as inputs.

4.2 A dataset of satellite images covering US cities

We picked one hundred American cities, spanning different states and reflecting various levels of wealth based on each city's income per capita as measured by Bureau of Economic Analysis (BEA) of the U.S. Department of Commerce. Our models require high-resolution images, particularly for the detection component. Therefore, we opted to extract our data from Google Earth, which offers satellite images up to 1 meter per pixel.

For each city on our list, we crafted large satellite images that encompass the entire urban area. These extensive images were created by stitching together multiple high-resolution overlapping captures of a given city at various locations. The resulting images are massive with some reaching a size of 20,000x20,000 pixels. One of these reconstructed images is shown in Figure 24.

Figure 24: Our reconstructed image of the city of Rome located in Georgia. This image was created by stitching 8 different high resolution satellite images of the city.

4.3 The infrastructure extraction pipeline

4.3.1 Putting the components together

We now assemble the components we designed in the previous section. One significant technical challenge our pipeline encounters is managing the size of our inputs. This poses memory issues and, for the detection component, impacts performance since the model was originally trained on 1000x1000 images, making it challenging for it to handle such a substantial increase in size.

We address this challenge by dividing the inputs into individual tiles, sequentially processing them through the pipeline. We have adopted distinct tile sizes for the segmentation and detection components to accommodate the unique requirements of each. Specifically, we use 512x512 tiles for the segmentation block and 1280x1280 tiles for the detection block. These tile sizes have demonstrated the best performance in our experiments, aligning closely with the dimensions on which the models were originally trained.

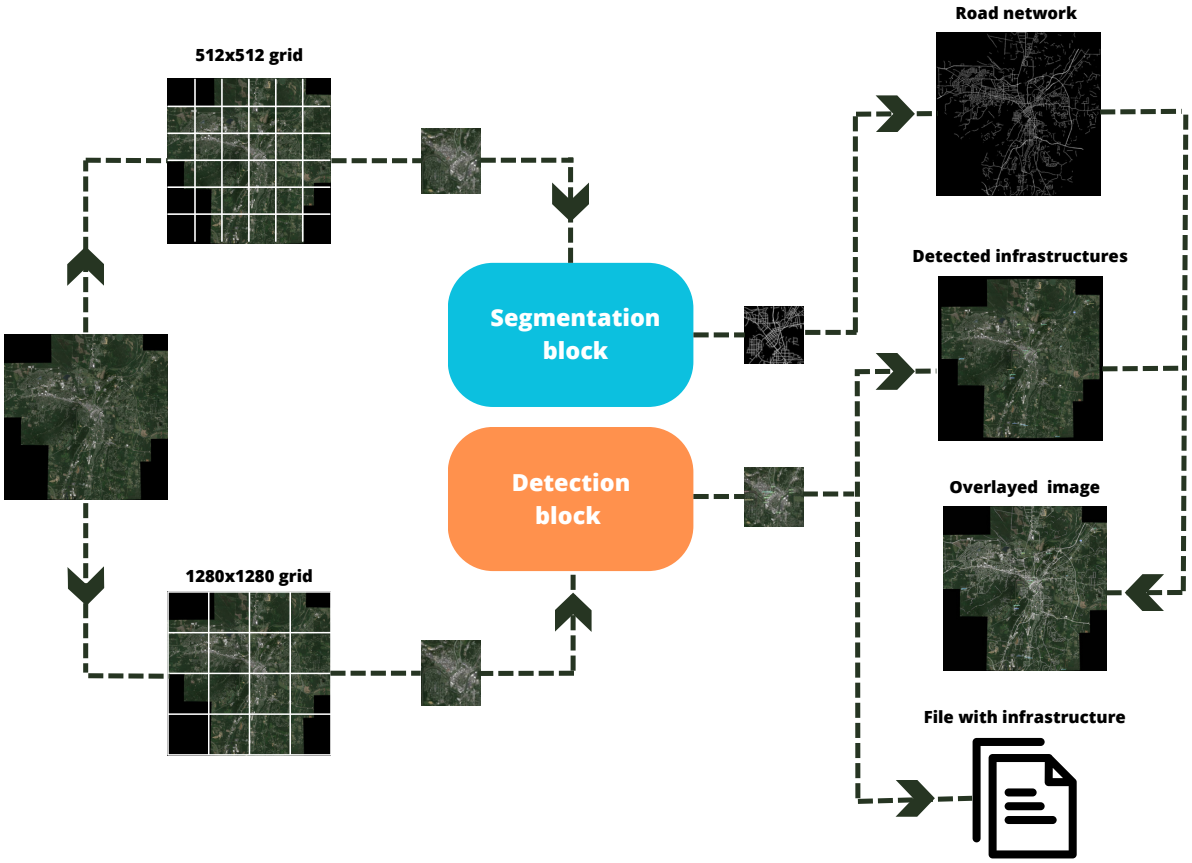


Figure 25: Simplified representation of our final infrastructure extraction pipeline utilising the segmentation and detection model we built in the previous sections.

Our pipeline produces four distinct outputs. After the segmentation block, we merge the segmented tiles to create an image of the same size as the original input, representing the complete extracted road network of the city. Simultaneously, the detection block generates two outputs. It initially draws bounding boxes on each tile before combining them to reproduce the original input with all detected infrastructure marked. Additionally, it generates a .csv file storing the detected infrastructure, their locations, and types in a structured format. This

dual output approach enables us to visualise the infrastructure while maintaining a convenient format to store them for further analysis. The pipeline ultimately overlays the segmented road network onto the satellite image marked with detected infrastructure, resulting in a final image of the city annotated with all the extracted information. A schematic representation of the pipeline can be seen in Figure 25.

4.3.2 Testing the pipeline on the US cities dataset

We applied our pipeline to our handcrafted US Cities dataset. Interestingly, the pipeline demonstrates notable efficiency by processing an entire city with a 15,000x15,000-pixel satellite image in an average time of approximately 30 seconds. Moreover, the quality of these outputs is highly satisfactory. Our image-splitting strategies and the efficiency of our two main blocks have proven effective, allowing the pipeline to accurately detect most infrastructure while efficiently extracting the road network of the area. The overlaid image, our third output, provides valuable visual insights for result assessment. We have included examples of these outputted images from selected areas in four cities in Figure 26. It is notable that the pipeline is capable of detecting infrastructure in various settings, such as isolated instances, as shown in the image of the dam in the first picture, or in densely populated areas. Additionally, the pipeline proves adept at identifying industrial zones, as demonstrated in the fourth picture, where numerous storage tanks are detected.

Figure 26: Samples from the overlaid images generated by our pipeline. In these images, the road network is represented by white pixels, and the model also highlights detected infrastructure with indicating their confidence levels.

4.4 Predicting income per capita using our extracted information

4.4.1 Predicting income per capita using only the outputs of the detection block

We start by attempting to predict city income per capita using only the output of the detection block. This choice was influenced by the ease of handling the outputs from this block initially, as the detected infrastructure are stored in a file. This allows us to create a vector for each city, indicating how many of each type of infrastructure has been detected. We employ a straightforward Random Forest Regressor (RFR) to predict a city's income per capita based on its infrastructure vector.

After training the Random Forest Regressor (RFR) on our training set, we find that the model produces an average error of \$5,235 in predicted income when applied to the validation set. To put this in perspective, if we had instead predicted the mean income value of our training set for every city in the validation set, we would have achieved a lower average error of around \$5,000. Taking this into account, it might seem that our infrastructure vectors do not contain significant signals for determining city wealth. However, it appears that even though our toy model may not accurately predict income, it still provides valuable information. Upon closer examination of our predictions, we find that the model seems to perform better at estimating a city's wealth relative to others.

To assess this effect, we conducted a test to determine how accurately the trained RFR could predict wealth comparisons between cities. Our results are presented in a matrix in Figure 27, where a 1 at the intersection of two cities signifies that the model correctly predicted which one was wealthier, while 0 indicates otherwise.

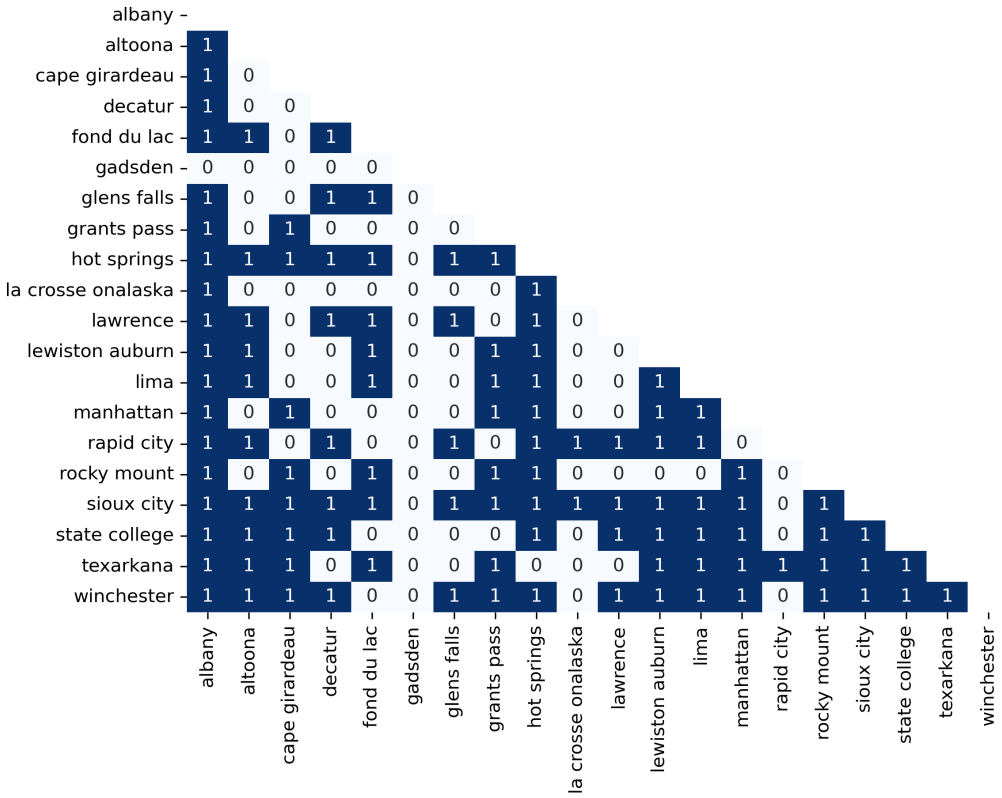


Figure 27: Heatmap illustrating the accuracy of our Random Forest model trained solely on infrastructure vectors for predicting the wealthier city in pairwise comparisons, with true predictions represented as 1s and false predictions as 0s.

We found that the model accurately predicts which city is wealthier in a comparison about 58% of the time, outperforming random guessing. Despite our limited training dataset of 80 cities, our experiment yields promising results. The infrastructure vectors derived from satellite images appear to be useful in assessing the relative wealth levels of cities.

4.4.2 Predicting income per capita using infrastructure and road networks

After experimenting the usefulness of detected infrastructure with our previous simple model, we are now aiming to leverage all the outputs generated by our pipeline to predict a city's income per capita. While incorporating the detected infrastructure into algorithms was relatively straightforward due to the use of a .csv file, the same cannot be said for our extracted road networks. These road networks are stored as segmented images with the same dimensions as the original input, consisting of binary pixels that indicate the presence of roads or the background. So, we must find a way to extract meaningful information from the segmented images before combining them with the infrastructure vectors. Our choice was to use a Convolutional Neural Network (CNN) to create a vector embedding of the road network images, which will then be concatenated with the infrastructure vectors.

This design choice was natural, as we have seen multiple times in this work how CNNs excel at extracting relevant information from images. Our idea is to produce a single vector that combines both the infrastructure and the road network image embedding before learning from this unified data structure. We then accordingly feed this embedding to three fully connected layers with ReLU activation functions that produce the income prediction.

The CNN used in our architecture to turn the road network images into a vector is ResNet18, an 18-layer ResNet network. We opted for this relatively lightweight backbone because our images are binary, and we believe that 18 layers suffice to capture the essential features. To accommodate our large images, we employed the same splitting strategy we previously used. We divided the segmented image into a grid of 224x224 cells, passing each cell through the backbone. Subsequently, we aggregated all the resulting vectors to create the image embedding. An illustration of our architecture for predicting income per capita using both road networks and infrastructure information is depicted in Figure 28.

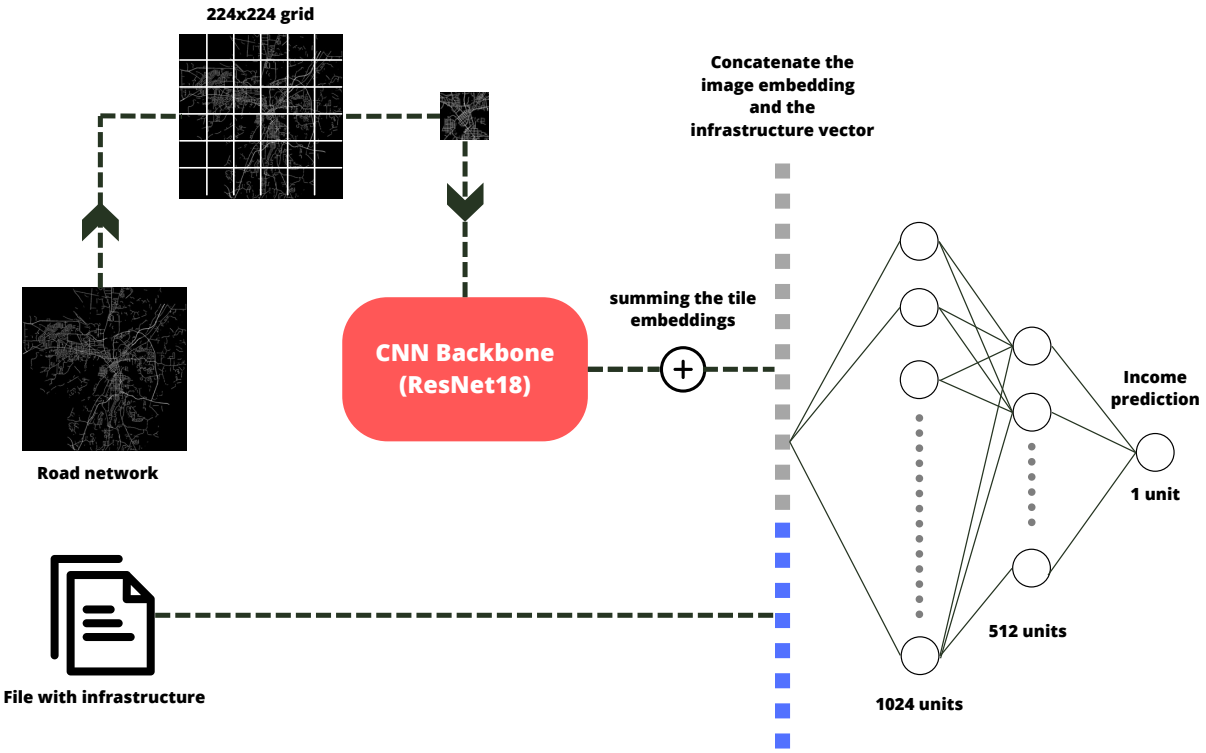


Figure 28: An overview of the architecture we designed to translate the information we obtained from a satellite image of a city (road network and detected infrastructure) into a prediction for the city's income per capita.

We trained the model presented above for 7 epochs trying to minimise the Mean Squared Error (MSE) Loss on the income prediction. We updated our neural network weights using Adam optimiser with a learning rate of 10^{-2}, and weight decay set to 10^{-4}. Additionally, we integrated dropout layers into our fully connected layers to mitigate overfitting, considering the size of our training set. The training phase results are depicted in Figure 29. The model reaches a plateau relatively quickly for both the training and the validation loss, which is understandable given the limited size of our dataset.

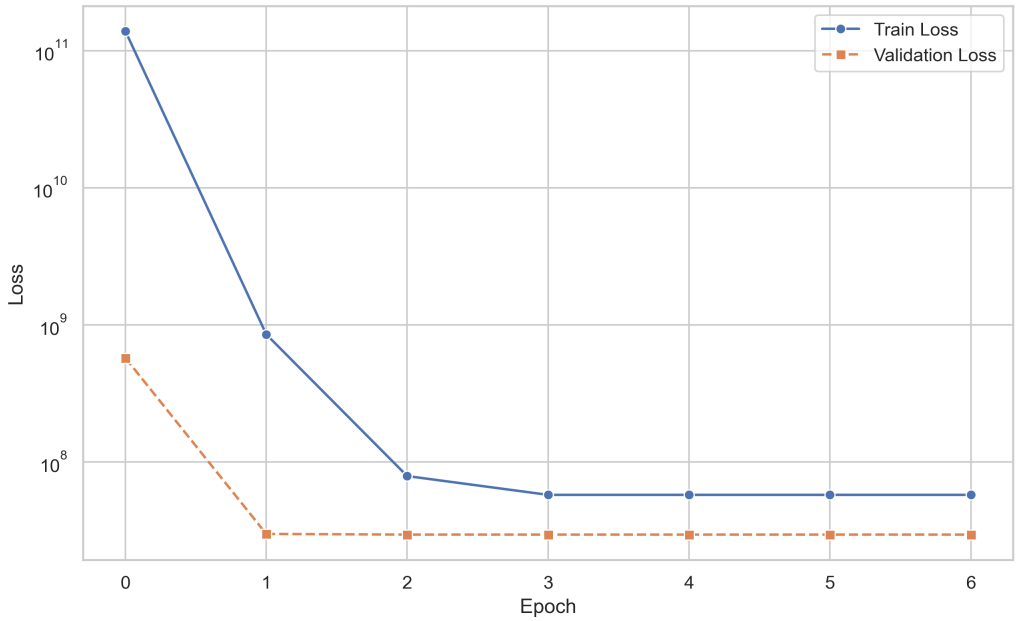


Figure 29: Training and validation loss against epochs for our final income prediction architecture.

The trained model yields noteworthy results in terms of pure prediction. It surpasses the Random Forest trained solely on infrastructure data, resulting in an average prediction error of 4,500 dollars, which is also lower than the error associated with just predicting the mean income per capita for every cities. Nonetheless, the true potential of our final model lies in its ability to predict a city's income level relative to another. Figure 30 illustrates the model's accuracy in predicting which city is wealthier in pairwise comparisons, using the same format as before.

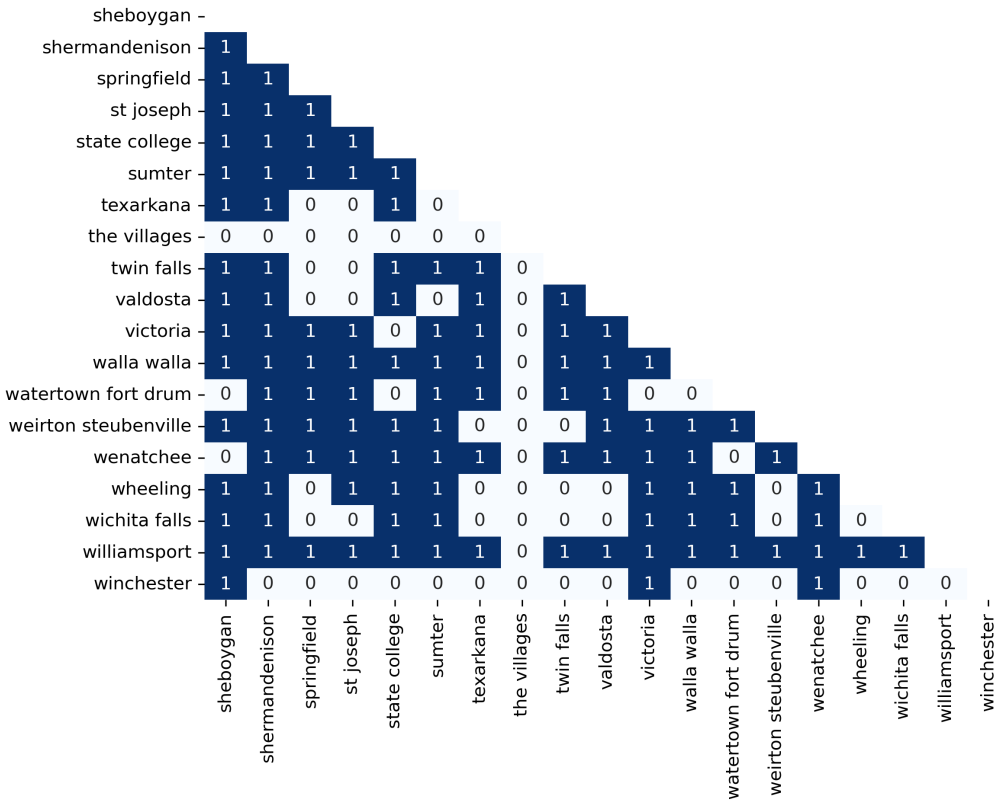


Figure 30: Heatmap illustrating the accuracy of our custom final architecture for predicting the wealthier city in pairwise comparisons, with true predictions represented as 1s and false predictions as 0s.

Our model effectively predicts the wealthier city in a comparison in 65% of cases, surpassing the performance of the RFR. However, we observe that the model consistently mispredicts every comparison involving the city called The Villages. This city located in Florida has a substantial population of retired individuals with high incomes, resulting in an income per capita significantly above the mean ($61,000 compared to the mean of $49,000). Given our training dataset of 80 cities, our model struggles to adapt to such outliers, resulting in inaccurate predictions for this specific city.

However, when we exclude this unusual city from our validation set, our model's accuracy improves to 71%. These results are highly encouraging, as they demonstrate that in a substantial number of cases, it is possible to distinguish which city is richer in a comparison using only the road network and the list infrastructure provided by our satellite images. We believe that if we were able to achieve these results with a dataset of only 80 US cities, this approach has even greater potential.

5 Conclusion

In this work, we have demonstrated the potential of high-resolution satellite imagery for economic purposes. We developed an infrastructure extraction pipeline comprising two trained deep learning models: an image segmentation model based on Deeplabv3+ and an object detection model based on YOLOv5. These two components proved to be efficient and robust, performing well not only on our test datasets but also on various global locations. When combined, they offer an effective means to extract valuable information from satellite images in a convenient manner. Furthermore, we utilised the information outputted by our pipeline to achieve promising results in estimating the relative wealth levels of cities. We accomplished this by designing a neural network architecture that takes our pipeline outputs as inputs and combines them to produce an income per capita estimate for a city. We trained this proposed architecture on a unique dataset of satellite images covering 100 US cities, which we believe can be used to further explore the use of satellite imagery for economic purposes.

Nevertheless, our approach has room for improvement on several fronts. A primary enhancement would involve gathering a significantly larger dataset to train our income prediction architecture, encompassing cities not just within the US but across multiple countries. Additionally, we could refine the utilisation of our infrastructure information to capture more signals. We used only part of the information available on infrastructure detected by our object detection component because we only use the presence of infrastructure, but not its location outputted by the pipeline. Therefore, one potential avenue for improvement is the development of an architecture that seamlessly integrates the road network with the detected infrastructure. This approach could consider factors such as the connectivity of industrial areas or airports with the rest of the city. Additionally, we could explore employing a graph-based structure to store road networks instead of using traditional images, a choice that might better represent the natural graph-like structure of city roadmaps.

Future work could address the limitations of this study and expand its scope. In addition to utilising satellite imagery as a proxy for wealth levels, there is potential to explore its applications in various other economic domains. For example, it could be used to predict export levels by analysing port activities or assess commodity production by examining industrial areas. Finally, it would be valuable to investigate how this method can help address data scarcity in many regions of the world, potentially scaling our approach at the regional or national level.

References

[1] S. Dong, P. Wang, and K. Abbas, "A survey on deep learning and its applications," *Computer Science Review*, vol. 40, p. 100379, 2021.

[2] S. Nosratabadi, A. Mosavi, P. Duan, P. Ghamisi, F. Filip, S. S. Band, U. Reuter, J. Gama, and A. H. Gandomi, "Data science in economics: comprehensive review of advanced machine learning and deep learning methods," *Mathematics*, vol. 8, no. 10, p. 1799, 2020.

[3] J. E. Ball, D. T. Anderson, and C. S. Chan, "Comprehensive survey of deep learning in remote sensing: theories, tools, and challenges for the community," *Journal of applied remote sensing*, vol. 11, no. 4, pp. 042609–042609, 2017.

[4] A. Merchant and C. Castillo, "Disparity, inequality, and accuracy tradeoffs in graph neural networks for node classification," *arXiv preprint arXiv:2308.09596*, 2023.

[5] S. Keola, M. Andersson, and O. Hall, "Monitoring economic development from space: using nighttime light and land cover data to measure economic growth," *World Development*, vol. 66, pp. 322–334, 2015.

[6] A. Khachiyan, A. Thomas, H. Zhou, G. Hanson, A. Cloninger, T. Rosing, and A. K. Khandelwal, "Using neural networks to predict microspatial economic growth," *American Economic Review: Insights*, vol. 4, no. 4, pp. 491–506, 2022.

[7] S. Polyzos and D. Tsiotas, "The contribution of transport infrastructures to the economic and regional development," *Theoretical and Empirical Researches in Urban Management*, vol. 15, no. 1, pp. 5–23, 2020.

[8] B. Ghosh and P. De, "How do different categories of infrastructure affect development? evidence from indian states," *Economic and Political Weekly*, pp. 4645–4657, 2004.

[9] C. Ng, T. Law, F. Jakarni, and S. Kulanthayan, "Road infrastructure development and economic growth," in *IOP conference series: materials science and engineering*, vol. 512, p. 012045, IOP Publishing, 2019.

[10] Y. Yu, C. Wang, Q. Fu, R. Kou, F. Huang, B. Yang, T. Yang, and M. Gao, "Techniques and challenges of image segmentation: A review," *Electronics*, vol. 12, no. 5, p. 1199, 2023.

[11] Z. Chen, L. Deng, Y. Luo, D. Li, J. M. Junior, W. N. Gonçalves, A. A. M. Nurunnabi, J. Li, C. Wang, and D. Li, "Road extraction in remote sensing data: A survey," *International Journal of Applied Earth Observation and Geoinformation*, vol. 112, p. 102833, 2022.

[12] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III 18*, pp. 234–241, Springer, 2015.

[13] Y. Ren, Y. Yu, and H. Guan, "Da-capsunet: a dual-attention capsule u-net for road extraction from remote sensing imagery. remote sens 12 (18): 2866," 2020.

[14] P. Li, X. He, M. Qiao, D. Miao, X. Cheng, D. Song, M. Chen, J. Li, T. Zhou, X. Guo, *et al.*, "Exploring multiple crowdsourced data to learn deep convolutional neural networks for road extraction," *International Journal of Applied Earth Observation and Geoinformation*, vol. 104, p. 102544, 2021.

[15] Z. Zhang, Q. Liu, and Y. Wang, "Road extraction by deep residual u-net," *IEEE Geoscience and Remote Sensing Letters*, vol. 15, no. 5, pp. 749–753, 2018.

[16] L.-C. Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam, "Encoder-decoder with atrous separable convolution for semantic image segmentation," in *Proceedings of the European conference on computer vision (ECCV)*, pp. 801–818, 2018.

[17] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, "Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs," *IEEE transactions on pattern analysis and machine intelligence*, vol. 40, no. 4, pp. 834–848, 2017.

[18] M. Everingham, S. A. Eslami, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes challenge: A retrospective," *International journal of computer vision*, vol. 111, pp. 98–136, 2015.

[19] H. Wang, F. Yu, J. Xie, and H. Zheng, "Road extraction based on improved deeplabv3 plus in remote sensing image," *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. 48, pp. 67–72, 2022.

[20] Q. Wu, F. Luo, P. Wu, B. Wang, H. Yang, and Y. Wu, "Automatic road extraction from high-resolution remote sensing images using a method based on densely connected spatial feature-enhanced pyramid," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 14, pp. 3–17, 2020.

[21] V. Mnih, *Machine learning for aerial image labeling*. University of Toronto (Canada), 2013.

[22] I. Demir, K. Koperski, D. Lindenbaum, G. Pang, J. Huang, S. Basu, F. Hughes, D. Tuia, and R. Raskar, "Deepglobe 2018: A challenge to parse the earth through satellite images," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 172–181, 2018.

[23] A. Van Etten, D. Lindenbaum, and T. M. Bacastow, "Spacenet: A remote sensing dataset and challenge series," *arXiv preprint arXiv:1807.01232*, 2018.

[24] G. Cheng, Y. Wang, S. Xu, H. Wang, S. Xiang, and C. Pan, "Automatic road detection and centerline extraction via cascaded end-to-end convolutional neural network," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 55, no. 6, pp. 3322–3337, 2017.

[25] Y. Liu, J. Yao, X. Lu, M. Xia, X. Wang, and Y. Liu, "Roadnet: Learning to comprehensively analyze road networks in complex urban scenes from high-resolution remotely sensed images," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 57, no. 4, pp. 2043–2056, 2018.

[26] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[27] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[28] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3431–3440, 2015.

[29] A. Sankar, "A primer on atrous(dilated) and depth-wise separable convolutions." `towardsdatascience.com/a-primer-on-atrous-convolutions-and-depth-wise-separable-convolutions-443b106919f5`.

[30] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, "Semantic image segmentation with deep convolutional nets and fully connected crfs. arxiv 2014," *arXiv preprint arXiv:1412.7062*, 2014.

[31] K. He, X. Zhang, S. Ren, and J. Sun, "Spatial pyramid pooling in deep convolutional networks for visual recognition," *IEEE transactions on pattern analysis and machine intelligence*, vol. 37, no. 9, pp. 1904–1916, 2015.

[32] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1251–1258, 2017.

[33] J. Dai, H. Qi, Y. Xiong, Y. Li, G. Zhang, H. Hu, and Y. Wei, "Deformable convolutional networks," in *Proceedings of the IEEE international conference on computer vision*, pp. 764–773, 2017.

[34] P. Iakubovskii, "Segmentation models pytorch." `https://github.com/qubvel/segmentation_models.pytorch`, 2019.

[35] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, pp. 8024–8035, Curran Associates, Inc., 2019.

[36] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[37] V. Yerram, H. Takeshita, Y. Iwahori, Y. Hayashi, M. Bhuyan, S. Fukui, B. Kijsirikul, and A. Wang, "Extraction and calculation of roadway area from satellite images using improved deep learning model and post-processing," *Journal of Imaging*, vol. 8, no. 5, p. 124, 2022.

[38] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.

[39] R. Wightman, "Pytorch image models." `https://github.com/rwightman/pytorch-image-models`, 2019.

[40] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255, Ieee, 2009.

[41] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization.," *Journal of machine learning research*, vol. 13, no. 2, 2012.

[42] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. De Freitas, "Taking the human out of the loop: A review of bayesian optimization," *Proceedings of the IEEE*, vol. 104, no. 1, pp. 148–175, 2015.

[43] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," *Advances in neural information processing systems*, vol. 24, 2011.

[44] J. Bergstra, D. Yamins, and D. Cox, "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures," in *International conference on machine learning*, pp. 115–123, PMLR, 2013.

[45] K. Li, G. Wan, G. Cheng, L. Meng, and J. Han, "Object detection in optical remote sensing images: A survey and a new benchmark," *ISPRS journal of photogrammetry and remote sensing*, vol. 159, pp. 296–307, 2020.

[46] G. Boeing, "Osmnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks," *Computers, Environment and Urban Systems*, vol. 65, pp. 126–139, 2017.

[47] Z. Zou, K. Chen, Z. Shi, Y. Guo, and J. Ye, "Object detection in 20 years: A survey," *Proceedings of the IEEE*, 2023.

[48] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 580–587, 2014.

[49] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 779–788, 2016.

[50] A. Hammas, S. T. Brassai, A. Németh, and S. L. Gábor, "Training yolo v4 on dota dataset," in *2023 IEEE 21st World Symposium on Applied Machine Intelligence and Informatics (SAMI)*, pp. 000109–000114, IEEE, 2023.

[51] Z. Zakria, J. Deng, R. Kumar, M. S. Khokhar, J. Cai, and J. Kumar, "Multiscale and direction target detecting in remote sensing images via modified yolo-v4," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 15, pp. 1039–1048, 2022.

[52] G. Jocher, A. Stoken, J. Borovec, NanoCode012, ChristopherSTAN, L. Changyu, Laughing, tkianai, A. Hogan, lorenzomammana, yxNONG, AlexWang1900, L. Diaconu, Marc, wanghaoyang0106, ml5ah, Doug, F. Ingham, Frederik, Guilhen, Hatovix, J. Poznanski, J. Fang, L. Y. , changyu98, M. Wang, N. Gupta, O. Akhtar, PetrDvoracek, and P. Rai, "ultralytics/yolov5: v3.1 - Bug Fixes and Performance Improvements," Oct. 2020.

[53] Y. Liu, G. He, Z. Wang, W. Li, and H. Huang, "Nrt-yolo: Improved yolov5 based on nested residual transformer for tiny remote sensing object detection," *Sensors*, vol. 22, no. 13, p. 4953, 2022.

[54] J. Xue, Y. Zheng, C. Dong-Ye, P. Wang, and M. Yasir, "Improved yolov5 network method for remote sensing image-based ground objects recognition," *Soft Computing*, vol. 26, no. 20, pp. 10879–10889, 2022.

[55] J. Ding, N. Xue, G.-S. Xia, X. Bai, W. Yang, M. Y. Yang, S. Belongie, J. Luo, M. Datcu, M. Pelillo, *et al.*, "Object detection in aerial images: A large-scale benchmark and challenges," *IEEE transactions on pattern analysis and machine intelligence*, vol. 44, no. 11, pp. 7778–7796, 2021.

[56] J. Hong, Z. Chu, and Q. Wang, "Transport infrastructure and regional economic growth: evidence from china," *Transportation*, vol. 38, pp. 737–752, 2011.

[57] Y. T. Mohmand, F. Mehmood, K. S. Mughal, and F. Aslam, "Investigating the causal relationship between transport infrastructure, economic growth and transport emissions in pakistan," *Research in Transportation Economics*, vol. 88, p. 100972, 2021.

[58] G. Jocher, "YOLOv5 by Ultralytics," May 2020.

[59] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 7263–7271, 2017.

[60] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv preprint arXiv:1804.02767*, 2018.

[61] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "Yolov4: Optimal speed and accuracy of object detection," *arXiv preprint arXiv:2004.10934*, 2020.

[62] C.-Y. Wang, H.-Y. M. Liao, Y.-H. Wu, P.-Y. Chen, J.-W. Hsieh, and I.-H. Yeh, "Cspnet: A new backbone that can enhance learning capability of cnn," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition workshops*, pp. 390–391, 2020.

[63] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*, pp. 448–456, pmlr, 2015.

[64] S. Elfwing, E. Uchibe, and K. Doya, "Sigmoid-weighted linear units for neural network function approximation in reinforcement learning," *Neural networks*, vol. 107, pp. 3–11, 2018.

[65] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature pyramid networks for object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2117–2125, 2017.

[66] S. Liu, L. Qi, H. Qin, J. Shi, and J. Jia, "Path aggregation network for instance segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8759–8768, 2018.

[67] H. Liu, F. Sun, J. Gu, and L. Deng, "Sf-yolov5: A lightweight small object detection algorithm based on improved feature fusion mode," *Sensors*, vol. 22, no. 15, p. 5817, 2022.

[68] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*, pp. 740–755, Springer, 2014.

[69] H. Rezatofighi, N. Tsoi, J. Gwak, A. Sadeghian, I. Reid, and S. Savarese, "Generalized intersection over union: A metric and a loss for bounding box regression," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 658–666, 2019.

Appendix

Overview of our custom dataset composition

Table 1: Dataset sources contribution to the final dataset.

Source	**Number of Images - Resolution**	**Contribution in the merged dataset**
DeepGlobe dataset [22]	7,469 - 0.5 m/pix	24,903 images (46 %)
Liu et al. dataset [25]	21 - 0.21 m/pix	716 images (2 %)
SpaceNet dataset [23]	5,151 - 0.5 to 1.3 m/pix	17,770 images (33 %)
Chen et al. dataset [24]	224 - 1.2 m/pix	1,621 images (4 %)
Massachusetts Roads dataset [21]	1,171 - 1 m/pix	8,136 images (15 %)

Atrous convolutions formula

The convolution operation remains unchanged, with the only distinction being the introduction of a spaced kernel. For a given input x_i, the resulting output y_i of an atrous convolution using a kernel w of length K and a dilation rate r is defined as:

$$y_i = \sum_{k=0}^{K} x_{i+r\cdot k} w_k$$

We see that if we set $r = 1$, we obtain the operation performed by a standard convolution layer.

Losses formulas

If we say that our images contain n pixels and we denote each pixel value by y_i, then with $\hat{y_i}$ being our predicted probability of pixel i, the binary cross-entropy loss can be written:

$$L_{BCE} = -\frac{1}{n}\sum_{i=1}^{n} y_i \log(\hat{y_i}) + (1 - y_i)\log(1 - \hat{y_i})$$

Using the same notation as before for the ground truth pixels and their predicted values, the Dice Loss can be written :

$$L_D = 1 - \frac{1}{n}\frac{2\sum_{i=1}^{n} y_i\hat{y_i}}{\sum_{i=1}^{n} y_i + \hat{y_i}}$$

Xception architecture

The input initially traverses the entry flow before progressing to the middle flow, which is iterated eight times. The final segment, the exit flow, involves flattening the feature maps into vectors, potentially ready for input

to fully connected layers. In our context, the last layers of the exit flow are substituted with atrous convolution layers and the ASPP.

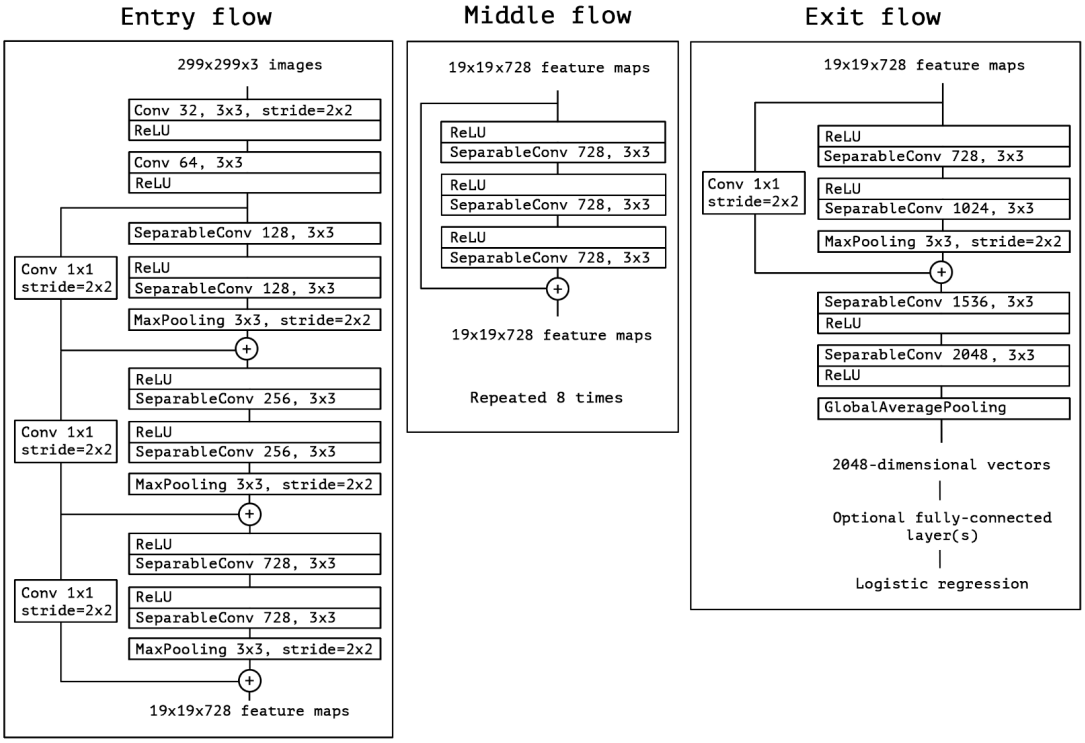


Figure 31: The original Xception architecture. *Source : [32]*

Details on the TPE algorithm

In the context of BO, the hyperparameter space will not be represented as a deterministic grid of parameters, as in Random Search, but rather as distributions over parameters that reflect our prior knowledge. For example, since we aim to equally test our two backbone choices, each of them will have an initially equal probability of being selected. We will follow the same approach for selecting the training strategy. As for the learning rate, we have opted to employ a log-uniform distribution for the initial distribution. This decision was motivated by our desire to uniformly explore multiple magnitudes of learning rates. The density function, denoted as f, for the initial distribution of the learning rate with $x \in [10^{-8}, 10^{-2}]$ can be expressed as :

$$f(x) = \frac{1}{x \cdot [log(10^{-2}) - log(10^{-8})]}$$

In TPE, the acquisition function is the Expected Improvement (EI). The EI for a given set of hyperparameters $\mathcal{S}$ can be expressed as:

$$\text{EI}(\mathcal{S}) = \mathbb{E}[\max(f_{min} - f(\mathcal{S}), 0)] = \int \max(f_{min} - f(\mathcal{S}), 0) p(f(\mathcal{S})|\mathcal{S}, \mathcal{D}) df(\mathcal{S})$$

Where f_{min} is the lowest value for our objective function that we have found so far, and $p(f|\mathcal{S}, \mathcal{D})$ is the posterior distribution for the objective function, taking into account past evaluations denoted as $\mathcal{D}$. The EI assigns non-null weights to points that the surrogate model predicts will yield a value below our current best,

f_{min}. Therefore, it is logical to maximize it to guide our exploration of the hyperparameter space, essentially directing the search towards the direction that yields the most significant improvement over our current best.

Regarding the surrogate model, TPE employs kernel density estimators (KDE) to model the distribution of the objective function. Utilizing Bayes' rules, we know that:

$$p(f(\mathcal{S})|\mathcal{S},\mathcal{D}) \propto p(f(\mathcal{S})|\mathcal{D}) \cdot p(\mathcal{S}|f(\mathcal{S}),\mathcal{D})$$

We will model $p(\mathcal{S}|f(\mathcal{S}),\mathcal{D})$ using a combination of kernels evaluations, with a kernel denoted k_σ with parameter σ the combination can be written :

$$g(\mathcal{S},\mathcal{D},\sigma) = w_0 p_0(\mathcal{S}) + \sum_{i=1}^{N} w_i k_\sigma(\mathcal{S}_i,\mathcal{S})$$

Where N is the number of trials conducted so far, w_i represents the weights updated at each iteration, summing up to 1, and $\mathcal{S}i$ denotes the hyperparameter values that have already been tested, meaning that $\mathcal{D} = \{(\mathcal{S}_i, f(\mathcal{S}_i))\}_{i=1}^{N}$. In our hyperparameter space, we have categorical parameters (backbone choice and training strategy) as well as continuous numerical parameters (learning rate). Therefore, TRE employs two different types of kernels for each kind. For numerical parameters, it uses a Gaussian kernel with a single parameter, σ:

$$k_\sigma(x,x_i) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(x-x_i)^2}$$

While for categorical features, TPE uses a Aitchison-Aitken kernel with a parameter $\sigma \in [0,1]$:

$$k_\sigma(x,x_i) = \begin{cases} 1-_\sigma & \text{if } x = x_i \\ \frac{\sigma}{C-1} & \text{Otherwise} \end{cases}$$

Where C is the number of choices for the parameter.

A dependence on $f(\mathcal{S})$ is then introduced in the expression of $p(\mathcal{S}|f(\mathcal{S}),\mathcal{D})$ by splitting our past evaluations into two groups using a threshold and having a different expression for the posterior of each group. We set a threshold f_* that leads to the partition of $\mathcal{D}$ into $\mathcal{D}^{(1)} = \{(\mathcal{S}_i, f(\mathcal{S}_i))|f(\mathcal{S}_i) \leq f_*\}$ and $\mathcal{D}^{(2)} = \{(\mathcal{S}_i, f(\mathcal{S}_i))|f(\mathcal{S}_i) > f_*\}$. Finally, the expression for $p(\mathcal{S}|f(\mathcal{S}),\mathcal{D})$ is given by :

$$p(\mathcal{S}|f(\mathcal{S}),\mathcal{D}) = \begin{cases} g(\mathcal{S},\mathcal{D}^{(1)},\sigma_1) & \text{if } f(\mathcal{S}) < f_* \\ g(\mathcal{S},\mathcal{D}^{(2)},\sigma_2) & \text{Otherwise} \end{cases}$$

Code

A Creating the segmentation dataset

```
import os
import numpy as np
import matplotlib.pyplot as plt
import cv2
import pathlib
from patchify import patchify
import shutil

# ============= Massachusets Roads dataset ============

os.chdir('../Data/Segmentation/Massachuset-roads/tiff/')

folder = 'val'
img_path = [image_id for image_id in os.listdir(folder)]


# keeping only the images without a lot of white pixels meaning that they are corrupted
for img in img_path:

    image = cv2.cvtColor(cv2.imread(os.path.join(folder, img)), cv2.COLOR_BGR2RGB)
    mask = cv2.cvtColor(cv2.imread(os.path.join(folder + '_labels', img)[:-1]),
    ↪  cv2.COLOR_BGR2GRAY)


    patches = patchify(image, (480, 480, 3), step=480)
    patches = np.squeeze(patches)
    mask = np.expand_dims(mask, axis=-1)
    patches_mask = patchify(mask, (480, 480, 1), step=480)
    patches_mask = np.squeeze(patches_mask,axis=2)

    rows, cols, _, _, _ = patches.shape

    os.chdir('../../Merged-roads/')
    for i in range(rows):
        for j in range(cols):

            if np.average(patches[i, j]) < 110:

                cv2.imwrite(os.path.join(folder, str(i) + str(j) + img) , patches[i, j])
                cv2.imwrite(os.path.join(folder + '_labels', str(i) + str(j) + img),
                ↪  patches_mask[i,j])


    os.chdir('../Massachuset-roads/tiff/')

# ============= DeepGlobe dataset ============

```

```
os.chdir('../Data/Segmentation/DeepGlobe-roads/')

folder = 'train'
img_path = [image_id for image_id in os.listdir(folder) if pathlib.Path(image_id).suffix ==
↪  '.jpg']

for img in img_path:

    image = cv2.cvtColor(cv2.imread(os.path.join(folder, img)), cv2.COLOR_BGR2RGB)
    mask = cv2.cvtColor(cv2.imread(os.path.join(folder, img[:-7]+'mask.png')),
    ↪  cv2.COLOR_BGR2GRAY)

    patches = patchify(image, (512, 512, 3), step=512)
    patches = np.squeeze(patches)
    mask = np.expand_dims(mask, axis=-1)
    patches_mask = patchify(mask, (512, 512, 1), step=512)
    patches_mask = np.squeeze(patches_mask,axis=2)

    rows, cols, _, _, _ = patches.shape

    os.chdir('../Merged-roads/')
    for i in range(rows):
        for j in range(cols):

            cv2.imwrite(os.path.join('deep', str(i) + str(j) + img) , patches[i, j])
            cv2.imwrite(os.path.join('deep' + '_labels', str(i) + str(j) + img),
            ↪  patches_mask[i,j])

    os.chdir('../DeepGlobe-roads/')


os.chdir('../Merged-roads/')

# 80% in train, 15% in val, 5% in test

n = len(img_path)
size_train = int(0.8 * n)
size_val = int(0.15 * n)
size_test = int(0.05 * n)
img_path_train = img_path[:size_train]
img_path_val = img_path[size_train:size_train + size_val]
img_path_test = img_path[size_train + size_val:size_train + size_val + size_test]

for img in img_path_test:
    shutil.move('deep/' + img, 'test/' + img)
    shutil.move('deep_labels/' + img, 'test_labels/'+ img)


for img in img_path_val:
    shutil.move('deep/' + img, 'val/' + img)
    shutil.move('deep_labels/' + img, 'val_labels/'+ img)

```

```
95
96  for img in img_path_train:
97      shutil.move('deep/' + img, 'train/' + img)
98      shutil.move('deep_labels/' + img, 'train_labels/'+ img)
99
100
101 # ============= Chen et al. dataset ============
102
103 os.chdir('../Data/Segmentation/GE-roads/')
104
105 folder = 'train'
106 img_path = [image_id for image_id in os.listdir(folder) if image_id != '.DS_Store']
107 images = []
108 for img in img_path:
109
110     image = cv2.cvtColor(cv2.imread(os.path.join(folder, img)), cv2.COLOR_BGR2RGB)
111     images.append(image)
112
113 folder = 'train'
114 img_path = [image_id for image_id in os.listdir(folder) if image_id != '.DS_Store']
115
116 for img in img_path:
117
118     image = cv2.cvtColor(cv2.imread(os.path.join(folder, img)), cv2.COLOR_BGR2RGB)
119     mask = cv2.cvtColor(cv2.imread(os.path.join(folder + '_labels', img[5:])),
        ↪  cv2.COLOR_BGR2GRAY)
120
121     patches = patchify(image, (256, 256, 3), step=256)
122     patches = np.squeeze(patches)
123     mask = np.expand_dims(mask, axis=-1)
124     patches_mask = patchify(mask, (256, 256, 1), step=256)
125     patches_mask = np.squeeze(patches_mask,axis=2)
126
127     if (np.ndim(patches),np.ndim(patches_mask)) == (5,5):
128
129         rows, cols, _, _, _ = patches.shape
130
131         os.chdir('../Merged-roads/')
132         for i in range(rows):
133             for j in range(cols):
134                 cv2.imwrite(os.path.join(folder, str(i) + str(j) + img) , patches[i, j])
135                 cv2.imwrite(os.path.join(folder + '_labels', str(i) + str(j) + img),
                    ↪  patches_mask[i,j])
136
137     os.chdir('../GE-roads/')
138
139
140 # ============= Liu et al. dataset ============
141
142 os.chdir('../Data/Segmentation/Ottawa-roads')
143
144 folder = 'train'
```

```
img_path = [image_id for image_id in os.listdir(folder) if image_id != '.DS_Store']

for img in img_path:

    image = cv2.cvtColor(cv2.imread(os.path.join(folder, img)), cv2.COLOR_BGR2RGB)
    mask = 1 - cv2.cvtColor(cv2.imread(os.path.join(folder + '_labels', img[:-4] + '.png')),
    ↪  cv2.COLOR_BGR2GRAY)

    patches = patchify(image, (512, 512, 3), step=512)
    patches = np.squeeze(patches)
    mask = np.expand_dims(mask, axis=-1)
    patches_mask = patchify(mask, (512, 512, 1), step=512)
    patches_mask = np.squeeze(patches_mask,axis=2)

    if (np.ndim(patches),np.ndim(patches_mask)) == (5,5):

        rows, cols, _, _, _ = patches.shape

        os.chdir('../Merged-roads/')
        for i in range(rows):
            for j in range(cols):
                cv2.imwrite(os.path.join('ottawa', str(i) + str(j) + img) , patches[i, j])
                cv2.imwrite(os.path.join('ottawa_labels', str(i) + str(j) + img),
                ↪  patches_mask[i,j])

    os.chdir('../Ottawa-roads/')


# 85% in train, 10% in val, 5% in test
os.chdir('../Merged-roads/')
folder = 'ottawa'
img_path = [image_id for image_id in os.listdir(folder)]

n = len(img_path)
size_train = int(0.85 * n)
size_val = int(0.10 * n)
size_test = int(0.05 * n)


img_path_train = img_path[:size_train]
img_path_val = img_path[size_train:size_train + size_val]
img_path_test = img_path[size_train + size_val:size_train + size_val + size_test]

for img in img_path_test:
    shutil.move('ottawa/' + img, 'test/' + img)
    shutil.move('ottawa_labels/' + img, 'test_labels/'+ img)


for img in img_path_val:
    shutil.move('ottawa/' + img, 'val/' + img)
    shutil.move('ottawa_labels/' + img, 'val_labels/'+ img)

```

```

for img in img_path_train:
    shutil.move('ottawa/' + img, 'train/' + img)
    shutil.move('ottawa_labels/' + img, 'train_labels/'+ img)


# ============= SpaceNet dataset ============

os.chdir('../Data/Segmentation/SpaceNet-roads/')

folder = 'val'
img_path = [image_id for image_id in os.listdir(folder) if image_id != ".DS_Store"]

# keeping only the images without a lot of black pixels
for img in img_path:

    image = cv2.cvtColor(cv2.imread(os.path.join(folder, img)), cv2.COLOR_BGR2RGB)
    mask = np.where(cv2.cvtColor(cv2.imread(os.path.join(folder + '_labels', img)),
    ↪  cv2.COLOR_BGR2GRAY) <10,0,255)

    patches = patchify(image, (512, 512, 3), step=512)
    patches = np.squeeze(patches)
    mask = np.expand_dims(mask, axis=-1)
    patches_mask = patchify(mask, (512, 512, 1), step=512)
    patches_mask = np.squeeze(patches_mask,axis=2)

    rows, cols, _, _, _ = patches.shape

    os.chdir('../SpaceNet-roads-curated/')
    for i in range(rows):
        for j in range(cols):

            if np.average(patches[i, j]) > 60:

                cv2.imwrite(os.path.join(folder, str(i) + str(j) + img) , patches[i, j])
                cv2.imwrite(os.path.join(folder + '_labels', str(i) + str(j) + img),
                ↪  patches_mask[i,j])


    os.chdir('../SpaceNet-roads/')

```

B Using Bayesian optimisation to find the best configuration for the segmentation model

```

import os
import torch
import numpy as np
from torch.utils.data import DataLoader,Subset
from torchvision import transforms

```

```
import cv2
import albumentations as album
from tqdm import tqdm
import gc
from hyperopt import fmin, tpe, hp, STATUS_OK, Trials
import pickle
import segmentation_models_pytorch as smp
from functools import partial

path = 'final-roads/'

os.chdir(path)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Accessing the custom segmentation dataset

class roadsDataset(torch.utils.data.Dataset):

    def __init__(self, folder, transform=None):

            self.image_paths = [os.path.join(folder, image_id) for image_id in os.listdir(folder)
            ↪  if image_id != ".DS_Store"]
            self.mask_paths = [os.path.join(folder + '_labels', image_id) for image_id in
            ↪  os.listdir(folder) if image_id != ".DS_Store"]

            self.transform = transform

    def __getitem__(self, i):


        image = cv2.cvtColor(cv2.imread(self.image_paths[i]), cv2.COLOR_BGR2RGB)
        mask = cv2.cvtColor(cv2.imread(self.mask_paths[i]), cv2.COLOR_BGR2GRAY)
        mask = np.where(mask >= 128, 255, 0) // 255

        if self.transform:
            sample = self.transform(image=image, mask=mask)
            image, mask = sample['image'], sample['mask']

        return transforms.ToTensor()(image.astype('float32')),
        ↪  transforms.ToTensor()(mask.astype('float32'))

    def __len__(self):

        return len(self.image_paths)


def transform_divisible(image, mask):

    transform = [
        album.PadIfNeeded(min_height=512, min_width=512, always_apply=True, border_mode=0),
    ]

```

```
    return album.Compose(transform)(image=image, mask=mask)

# Creating the training loop for TPE evaluations

def training_loop(n_epochs, optimizer, model, loss_fn,
↪  train_loader,val_loader,preprocessing,decoderOnly=False):

    for epoch in range(1, n_epochs + 1):

        print('Epoch {}'.format(epoch))

        model.train()
        loss_train = 0.0
        n_batches_train = len(train_loader)
        train_loop = tqdm(train_loader, desc="Training")

        for imgs, masks in train_loop:

            if decoderOnly:
                imgs =  torch.stack([preprocessing(image.permute(1, 2, 0)).permute(2,0,1).float()
                ↪  for image in imgs])
            else:
                imgs = (imgs - imgs.mean())/imgs.std()

            imgs, masks = imgs.to(device), masks.to(device)

            outputs = model(imgs)
            loss = loss_fn(outputs, masks)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            loss_train += loss.item()

        model.eval()
        with torch.no_grad():

            loss_val = 0.0
            n_batches_val = len(val_loader)
            val_loop = tqdm(val_loader, desc="Validation")

            for imgs, masks in val_loop:

                imgs = (imgs - imgs.mean())/imgs.std()
                imgs, masks = imgs.to(device), masks.to(device)

                outputs = model(imgs)
                loss = loss_fn(outputs, masks)
                loss_val += loss.item()

        val_loss_t = loss_val / n_batches_val
        train_loss_t = loss_train / n_batches_train
```

```
105
106         print('Training loss {} | Validation loss {}'.format(train_loss_t, val_loss_t))
107
108     return val_loss_t,train_loss_t
109
110
111 # Custom loss combining BCE Loss and Dice Loss
112
113 class CustomLoss(torch.nn.Module):
114
115     def __init__(self,alpha):
116         super(CustomLoss, self).__init__()
117         self.alpha = alpha
118         self.loss1 = torch.nn.BCELoss()
119         self.loss2 = smp.losses.DiceLoss(mode="binary",from_logits=False)
120
121     def forward(self, output, target):
122
123         loss = self.alpha*self.loss1(output,target) + (1-self.alpha)*self.loss2(output,target)
124
125         return loss
126
127 # Defining the objective function
128
129 def objective(params,train_loader,val_loader):
130
131     print(params)
132
133     gc.collect()
134     torch.cuda.empty_cache()
135
136     encoder = params['encoder']
137     encoder_weights = "imagenet"
138
139     model = smp.DeepLabV3Plus(
140             encoder_name= encoder,
141             encoder_weights= encoder_weights,
142             activation="sigmoid",
143             in_channels=3,
144             classes=1,
145         )
146
147     model.to(device)
148
149     n_epochs = 4
150     loss_fn = CustomLoss(0.25)
151
152     if not params['decoder-only']:
153         optimizer = torch.optim.Adam([ dict(params=model.parameters(),
        ↪  lr=params['learning-rate'])])
154     else:
```

```
        optimizer = torch.optim.Adam([dict(params=model.decoder.parameters(),
        ↪  lr=params['learning-rate'])])

    preprocessing = smp.encoders.get_preprocessing_fn(encoder_name=encoder,
    ↪  pretrained=encoder_weights)

    val_loss,train_loss = training_loop(n_epochs, optimizer, model, loss_fn,
    ↪  train_loader,val_loader,preprocessing,params['decoder-only'])

    return {
        'loss': val_loss,
        'status': STATUS_OK,
        'params': params,
        'train_loss': train_loss
        }

# Defining the space of possible training settings

trials = Trials()
space = {
        'encoder': hp.choice('encoder',['resnet101','tu-xception71']),
        'decoder-only': hp.choice('decoder-only',[False,True]),
        'learning-rate': hp.loguniform('learning_rate',
                                                np.log(0.000001),
                                                np.log(0.01))
        }

train_dataset = roadsDataset('train',transform=transform_divisible)
valid_dataset = roadsDataset('val',transform=transform_divisible)

train_dataset_lite = Subset(train_dataset, torch.randperm(len(train_dataset))[:8000])
valid_dataset_lite = Subset(valid_dataset, torch.randperm(len(valid_dataset))[:2000])

train_loader = DataLoader(train_dataset_lite, batch_size=10, shuffle=True)
val_loader = DataLoader(valid_dataset_lite, batch_size=5, shuffle=False)

# Running TPE using HyperOpt

hyperopt_objective = partial(objective, train_loader=train_loader,val_loader=val_loader)

best = fmin(hyperopt_objective,
            space=space,
            algo=tpe.suggest,
            max_evals=50,
            trials=trials)

filename = 'hyperopt_trials.pkl'

with open(filename, 'wb') as f:
    pickle.dump(trials, f)

```

C Training our segmentation model on our merged dataset

```
import os
import torch
import numpy as np
import cv2
import albumentations as album
import segmentation_models_pytorch as smp
from tqdm import tqdm
from torch.utils.data import DataLoader
from torchvision import transforms
import pickle

path = 'final-roads/'

os.chdir(path)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
torch.cuda.empty_cache()

# Accessing the custom segmentation dataset

class roadsDataset(torch.utils.data.Dataset):

    def __init__(self, folder, transform=None):

            self.image_paths = [os.path.join(folder, image_id) for image_id in os.listdir(folder)
            ↪  if image_id != ".DS_Store"]
            self.mask_paths = [os.path.join(folder + '_labels', image_id) for image_id in
            ↪  os.listdir(folder) if image_id != ".DS_Store" ]

            self.transform = transform

    def __getitem__(self, i):


        image = cv2.cvtColor(cv2.imread(self.image_paths[i]), cv2.COLOR_BGR2RGB)
        mask = cv2.cvtColor(cv2.imread(self.mask_paths[i]), cv2.COLOR_BGR2GRAY)
        mask = np.where(mask >= 128, 255, 0) // 255


        if self.transform:
            sample = self.transform(image=image, mask=mask)
            image, mask = sample['image'], sample['mask']

        return transforms.ToTensor()(image.astype('float32')),
        ↪  transforms.ToTensor()(mask.astype('float32'))

    def __len__(self):

        return len(self.image_paths)

def transform_divisible(image, mask):
```

```
48
49      transform = [
50          album.PadIfNeeded(min_height=512, min_width=512, always_apply=True, border_mode=0),
51      ]
52
53      return album.Compose(transform)(image=image, mask=mask)
54
55  # Creating the training loop for the segmentation model
56
57  def training_loop(n_epochs, optimizer, model, loss_fn, train_loader,val_loader):
58
59      train_loss_list = []
60      val_loss_list = []
61
62      for epoch in range(1, n_epochs + 1):
63
64          print('Epoch {}'.format(epoch))
65
66          model.train()
67          loss_train = 0.0
68          n_batches_train = len(train_loader)
69          train_loop = tqdm(train_loader, desc="Training")
70
71          for imgs, masks in train_loop:
72
73              imgs = (imgs - imgs.mean())/imgs.std()
74
75              imgs, masks = imgs.to(device), masks.to(device)
76
77              outputs = model(imgs)
78              loss = loss_fn(outputs, masks)
79
80              optimizer.zero_grad()
81              loss.backward()
82              optimizer.step()
83              loss_train += loss.item()
84
85          model.eval()
86          with torch.no_grad():
87
88              loss_val = 0.0
89              n_batches_val = len(val_loader)
90              val_loop = tqdm(val_loader, desc="Validation")
91
92              for imgs, masks in val_loop:
93
94                  imgs = (imgs - imgs.mean())/imgs.std()
95                  imgs, masks = imgs.to(device), masks.to(device)
96
97                  outputs = model(imgs)
98                  loss = loss_fn(outputs, masks)
99                  loss_val += loss.item()
```

```
100
101         val_loss_t = loss_val / n_batches_val
102         val_loss_list.append(val_loss_t)
103         train_loss_t = loss_train / n_batches_train
104         train_loss_list.append(train_loss_t)
105
106         os.chdir('../')
107         print('Training loss {} | Validation loss {}'.format(train_loss_t, val_loss_t))
108         torch.save(model.state_dict(), 'deeplabv3+-xception-{}-{}.pth'.format(epoch,val_loss_t))
109         os.chdir('final-roads/')
110
111     results = {
112         'train_loss': train_loss_list ,
113         'val_loss': val_loss_list,
114     }
115
116     with open('training_res.pkl', 'wb') as f:
117         pickle.dump(results, f)
118
119 # Custom loss combining BCE Loss and Dice Loss
120
121 class CustomLoss(torch.nn.Module):
122
123     def __init__(self,alpha):
124         super(CustomLoss, self).__init__()
125         self.alpha = alpha
126         self.loss1 = torch.nn.BCELoss()
127         self.loss2 = smp.losses.DiceLoss(mode="binary",from_logits=False)
128
129     def forward(self, output, target):
130
131         loss = self.alpha*self.loss1(output,target) + (1-self.alpha)*self.loss2(output,target)
132
133         return loss
134
135 # Defining our dataset and instantiating DeepLabv3+
136
137 train_dataset = roadsDataset('train',transform=transform_divisible)
138 valid_dataset = roadsDataset('val',transform=transform_divisible)
139
140 train_loader = DataLoader(train_dataset, batch_size=10, shuffle=True)
141 val_loader = DataLoader(valid_dataset, batch_size=5, shuffle=False)
142
143 encoder = "tu-xception71"
144 encoder_weights = "imagenet"
145 decoderOnly=False
146
147 model = smp.DeepLabV3Plus(
148     encoder_name= encoder,
149     encoder_weights= encoder_weights,
150     activation="sigmoid",
151     in_channels=3,
```

```
    classes=1,
)

model.to(device)

# Train the model for 25 epochs

n_epochs = 25
loss_fn = CustomLoss(0.25)

optimizer = torch.optim.Adam([ dict(params=model.parameters(), lr=0.00012316897323716 27)])


training_loop(n_epochs, optimizer, model, loss_fn, train_loader,val_loader)

```

D Creating the detection dataset

```
from IPython.display import Image
import os
from PIL import Image
import numpy as np
import PIL.Image
import shapely.geometry as shgeo
import xml.etree.ElementTree as ET
import shutil

os.getcwd()

# ============= DOTA dataset ============

os.chdir('../Data/Detection/DOTA-merged/val/')

PIL.Image.MAX_IMAGE_PIXELS = 806504000

selected = ['plane', 'ship', 'storage-tank',
↪  'baseball-diamond','tennis-court','basketball-court','ground-track-field', 'harbor','bridge',
'soccer-ball-field','swimming-pool','container-crane','airport','helipad']

class_name_to_id_mapping_dota = {
    'plane': 0,
    'ship': 1,
    'storage-tank': 2,
    'baseball-diamond': 3,
    'tennis-court': 4,
    'basketball-court': 5,
    'ground-track-field': 6,
    'harbor': 7,
    'bridge': 8,
    'soccer-ball-field': 9,
```

```
    'swimming-pool': 10,
    'container-crane': 11,
    'airport': 12,
    'helipad': 13
}

# Functions to turn DOTA original annotations into YOLO annotations format. Code taken from
↪ https://github.com/ringringyi/DOTA_YOLOv2/tree/master then modified

def GetFileFromThisRootDir(dir,ext = None):
  allfiles = []
  needExtFilter = (ext != None)
  for root,dirs,files in os.walk(dir):
    for filespath in files:
      filepath = os.path.join(root, filespath)
      extension = os.path.splitext(filepath)[1][1:]
      if needExtFilter and extension in ext:
        allfiles.append(filepath)
      elif not needExtFilter:
        allfiles.append(filepath)
  return allfiles

def parse_dota_poly(filename):
    """
        parse the dota ground truth in the format:
        [(x1, y1), (x2, y2), (x3, y3), (x4, y4)]
    """
    objects = []

    f = []
    fd = open(filename, 'r')
    f = fd

    while True:
        line = f.readline()

        if line:
            splitlines = line.strip().split(' ')
            object_struct = {}

            if (len(splitlines) < 9):
                continue
            if (len(splitlines) >= 9):
                    object_struct['name'] = splitlines[8]
            if (len(splitlines) == 9):
                object_struct['difficult'] = '0'
            elif (len(splitlines) >= 10):

                object_struct['difficult'] = splitlines[9]

            object_struct['poly'] = [(float(splitlines[0]), float(splitlines[1])),
                                     (float(splitlines[2]), float(splitlines[3])),
```

```
                                    (float(splitlines[4]), float(splitlines[5])),
                                    (float(splitlines[6]), float(splitlines[7]))
                                    ]
            gtpoly = shgeo.Polygon(object_struct['poly'])
            object_struct['area'] = gtpoly.area
            objects.append(object_struct)
        else:
            break
    return objects

def dots4ToRec4(poly):
    xmin, xmax, ymin, ymax = min(poly[0][0], min(poly[1][0], min(poly[2][0], poly[3][0]))), \
                             max(poly[0][0], max(poly[1][0], max(poly[2][0], poly[3][0]))), \
                              min(poly[0][1], min(poly[1][1], min(poly[2][1], poly[3][1]))), \
                              max(poly[0][1], max(poly[1][1], max(poly[2][1], poly[3][1])))
    return xmin, ymin, xmax, ymax

def dots4ToRecC(poly, img_w, img_h):
    xmin, ymin, xmax, ymax = dots4ToRec4(poly)
    x = (xmin + xmax)/2
    y = (ymin + ymax)/2
    w = xmax - xmin
    h = ymax - ymin
    return x/img_w, y/img_h, w/img_w, h/img_h


def dota2yolo(imgpath, txtpath, extractclassname):
    """
    :param imgpath: the path of images
    :param txtpath: the path of txt in dota format
    :param dstpath: the path of txt in YOLO format
    :param extractclassname: the category you selected
    :return:
    """
    filelist = GetFileFromThisRootDir(txtpath)
    for fullname in filelist:
        objects = parse_dota_poly(fullname)
        name = os.path.splitext(os.path.basename(fullname))[0]
        img_fullpath = os.path.join(imgpath, name + '.png')
        img = Image.open(img_fullpath)
        img_name = img_fullpath[18:]
        img_w, img_h = img.size

        print_buffer = []
        for obj in objects:

            poly = obj['poly']
            b_center_x, b_center_y, b_width, b_height = dots4ToRecC(poly, img_w, img_h)
            bbox = np.array([b_center_x, b_center_y, b_width, b_height])
            if obj['name'] in extractclassname and not ((sum(bbox <= 0) + sum(bbox >= 1)) >= 1) :
                class_id = class_name_to_id_mapping_dota[obj['name']]
```

```
                print_buffer.append("{} {:.3f} {:.3f} {:.3f} {:.3f}".format(class_id, b_center_x,
                ↪  b_center_y, b_width, b_height))


        save_file_name = os.path.join("labels-yolo", img_name.replace("png", "txt"))

        if print_buffer:
            print("\n".join(print_buffer), file= open(save_file_name, "w"))
            shutil.move('images-background/' + img_name, 'images-yolo/' + img_name)

dota2yolo('images-background',
            'labels',
                 selected)


# ============= DIOR dataset ============

os.chdir('../Data/Detection/DIOR/test/')

# Functions to turn DIOR original annotations into YOLO annotations format. Code taken from
↪  https://github.com/hai-h-nguyen/Yolo2Pascal-annotation-conversion then modified
def extract_info_from_xml(xml_file):
    root = ET.parse(xml_file).getroot()

    info_dict = {}
    info_dict['bboxes'] = []

    for elem in root:

        if elem.tag == "filename":
            info_dict['filename'] = elem.text


        elif elem.tag == "size":
            image_size = []
            for subelem in elem:
                image_size.append(int(subelem.text))

            info_dict['image_size'] = tuple(image_size)


        elif elem.tag == "object":
            bbox = {}
            for subelem in elem:
                if subelem.tag == "name":
                    bbox["class"] = subelem.text

                elif subelem.tag == "bndbox":
                    for subsubelem in subelem:
                        bbox[subsubelem.tag] = int(subsubelem.text)
            info_dict['bboxes'].append(bbox)

```

```
    return info_dict

class_name_to_id_mapping_dior = {"airplane": 0,
                                 "ship": 1,
                                 "storagetank": 2,
                                 "baseballfield": 3,
                                 "tenniscourt": 4,
                                 "basketballcourt": 5,
                                 "groundtrackfield": 6,
                                 "harbor": 7,
                                 "bridge": 8,
                                'soccer-ball-field': 9,
                                'swimming-pool': 10,
                                'container-crane': 11,
                                 "airport": 12,
                                 "helipad": 13,
                                "chimney": 14,
                                "dam": 15,
                                "golffield": 16,
                                "stadium": 17,
                                "trainstation": 18,
                                "windmill": 19
                            }


def convert_to_yolov5(info_dict):
    print_buffer = []


    for b in info_dict["bboxes"]:

        if b["class"] in class_name_to_id_mapping_dior:
            class_id = class_name_to_id_mapping_dior[b["class"]]

            b_center_x = (b["xmin"] + b["xmax"]) / 2
            b_center_y = (b["ymin"] + b["ymax"]) / 2
            b_width    = (b["xmax"] - b["xmin"])
            b_height   = (b["ymax"] - b["ymin"])

            image_w, image_h, image_c = info_dict["image_size"]
            b_center_x /= image_w
            b_center_y /= image_h
            b_width    /= image_w
            b_height   /= image_h

            print_buffer.append("{} {:.3f} {:.3f} {:.3f} {:.3f}".format(class_id, b_center_x,
            ↪  b_center_y, b_width, b_height))

    save_file_name = os.path.join("labels-yolo", info_dict["filename"].replace("jpg", "txt"))

    if print_buffer:
        print("\n".join(print_buffer), file= open(save_file_name, "w"))
```

```
        shutil.move('images-background/' + info_dict["filename"], 'images-yolo/' +
        ↪  info_dict["filename"])

annotations = [os.path.join('labels', x) for x in os.listdir('labels') if x[-3:] == "xml"]
annotations.sort()

for ann in annotations:
    info_dict = extract_info_from_xml(ann)
    convert_to_yolov5(info_dict)

```

E Training our detection model on our merged dataset

We used the `train.py` file in the YOLOv5 repository available on GitHub. We define the training instructions in an .xml file, which is then passed to the training file along with our training parameters using the following command:

```
python3 train.py -batch -1 -epochs 25 - 'dior-dota-curated.xml' -weights 'yolov5x6.pt'
```

F Building our infrastructure extraction pipeline and running it on the US Cities dataset

```
import os
import torch
import numpy as np
from torchvision import transforms
import cv2
import pandas as pd
import albumentations as album
from tqdm import tqdm
import segmentation_models_pytorch as smp
import numpy as np
from patchify import patchify
from patchify import unpatchify


device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Creating the pipeline combining the trained segmentation model and the trained detection model
class SegmentationDetectionPipeline(torch.nn.Module):

    def __init__(self, segmentation_weights,detection_weights,device):
        super().__init__()

        # Loading the weights of the trained models
        self.segmentation_model = smp.DeepLabV3Plus(
```

```
                                        encoder_name= "tu-xception71",
                                        activation="sigmoid",
                                        in_channels=3,
                                        classes=1,
                                    )
        self.segmentation_model.load_state_dict(segmentation_weights)
        self.segmentation_model.to(device)
        self.segmentation_model.eval()

        self.detection_model = torch.hub.load('ultralytics/yolov5', 'custom',
        ↪  path=detection_weights)
        self.detection_model.conf = 0.5
        self.detection_model.to(device)

    def forward(self, image,city):

        # Creating the 512x512 grid for the segmentation component
        (h,w,_) = image.shape
        size = 512
        min_h = h + size-h%size; min_w = w + size-w%size

        transform = album.PadIfNeeded(min_height=min_h, min_width=min_w, always_apply=True,
        ↪  border_mode=0)
        image2 = transform(image=image)['image']

        patches = patchify(image2, (512, 512, 3), step=512)
        patches = np.squeeze(patches)
        rows, cols, _, _, _ = patches.shape
        patches_segmentation = patches.copy()

        for i in tqdm(range(rows)):
            for j in tqdm(range(cols)):

                img_tensor = transforms.ToTensor()(patches[i,j].astype('float32'))
                img_tensor = (img_tensor-img_tensor.mean())/img_tensor.std()
                img_tensor = img_tensor.to(device)
                pred_masks = self.segmentation_model(img_tensor.unsqueeze(0))
                pred_binary = torch.where(pred_masks >= 0.5,1,0)
                patches_segmentation[i,j] =
                ↪  cv2.cvtColor(pred_binary[0].squeeze().numpy(force=True).astype('uint8')*255,
                ↪  cv2.COLOR_GRAY2RGB)

        stitched_segment = unpatchify(np.expand_dims(patches_segmentation, axis=2), image2.shape)

        # Segmented road network
        cv2.imwrite(os.path.join('../../segmented', city + '.jpg'), stitched_segment)

        # Creating the 1280x1280 grid for the detection component
        size = 1280
        min_h = h + size-h%size; min_w = w + size-w%size

```

```
        transform = album.PadIfNeeded(min_height=min_h, min_width=min_w, always_apply=True,
        ↪  border_mode=0)
        image3 = transform(image=image)['image']

        patches = patchify(image3, (size, size, 3), step=size)
        patches = np.squeeze(patches)
        rows, cols, _, _, _ = patches.shape
        patches_detection = patches.copy()

        columns = ['xmin', 'ymin', 'xmax', 'ymax', 'confidence', 'class', 'name']
        detection_df = pd.DataFrame(columns=columns)

        for i in tqdm(range(rows)):
            for j in tqdm(range(cols)):

                detection = self.detection_model(patches[i,j], augment=True)
                patches_detection[i,j] = detection.render()[0]
                detection_df = pd.concat([detection_df , detection.pandas().xyxy[0]],
                ↪  ignore_index=True)

        stitched_detect = unpatchify(np.expand_dims(patches_detection, axis=2), image3.shape)

        # Image with the detected infrastrucure marked
        cv2.imwrite(os.path.join('../../final_pipeline', city + '_detect.jpg'), stitched_detect)
        # CSV file with the detected infrastrucure
        detection_df.to_csv(os.path.join('../../infrastructures', city + '.csv'), index=False)

        transform2 = album.CenterCrop(h, w)
        stitched_detect = transform2(image=stitched_detect)['image']
        stitched_segment = transform2(image=stitched_segment)['image']

        final = cv2.addWeighted(stitched_detect, 1, stitched_segment, 1, 1)

        # Image overlayed with all the information
        cv2.imwrite(os.path.join('../../final_pipeline', city + '_final.jpg'), final)


segmentation_weights= torch.load('DeepLabV3Plus-best.pth')
detection_weights = 'YOLOv5x6-best.pt'
pipeline = SegmentationDetectionPipeline(segmentation_weights,detection_weights,device)

list_cities = [city for city in os.listdir('Cities/stitched_cities/') if city != '.DS_Store' and
↪  city != 'stitched_cities']
os.chdir('Cities/stitched_cities/')

# Running the pipeline on our US Cities dataset
for city in list_cities:
    image = cv2.imread(city)
    pipeline(image,city[:-4])

```

G Building our architecture to predict income per capita and training it

```
import os
import torch
import numpy as np
import cv2
import timm
import re
import pickle
import pandas as pd
import albumentations as album
from tqdm import tqdm
from torchvision import transforms
from torch import nn
from patchify import patchify

path = 'forecasting-income/'
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
os.chdir(path)

# Accessing the data outputted by the infrastructure extraction pipeline and the income per capita
class forecastingDataset(torch.utils.data.Dataset):

    def __init__(self, folder,path_csv):

            self.image_paths = [image_id for image_id in os.listdir(folder + '/segmented') if image_id != ".DS_Store"]
            self.df = pd.read_csv(path_csv)
            self.folder = folder

    def __getitem__(self, i):

        city_name = re.sub(r'[_]+$', '', re.sub(r'[A-Z]|-|_+$', '', self.image_paths[i][:-4]))
        row = self.df[self.df['Cities'] == city_name]
        infra_tens = torch.tensor(row.iloc[:, 2:-1].values.astype('float32'))

        income = self.df[self.df['Cities'] == city_name]['income'].values

        return os.path.join(self.folder + '/segmented', self.image_paths[i]),infra_tens,torch.tensor(income).to(torch.float32)

    def __len__(self):

        return len(self.image_paths)

# Training loop for the forecasting pipeline
def training_loop(n_epochs, optimizer, model, loss_fn, train_dataset,valid_dataset):

    train_loss_list = []
    val_loss_list = []
    val_pred = []
```

```
48
49      for epoch in range(1, n_epochs + 1):
50
51          print('Epoch {}'.format(epoch))
52
53          model.train()
54          loss_train = 0.0
55          n_batches_train = len(train_dataset)
56
57          for i in tqdm(range(n_batches_train), desc="Training"):
58
59              image_path,vec,income = train_dataset[i]
60              image = cv2.imread(image_path)
61
62              outputs = model(image,vec)
63              income = income.to(device)
64
65
66              loss = loss_fn(outputs, income.unsqueeze(0))
67
68              optimizer.zero_grad()
69              loss.backward()
70              optimizer.step()
71              loss_train += loss.item()
72
73          model.eval()
74          with torch.no_grad():
75
76              loss_val = 0.0
77              n_batches_val = len(valid_dataset)
78
79              for i in tqdm(range(n_batches_val), desc="Validation"):
80
81                  image_path,vec,income = valid_dataset[i]
82                  image = cv2.imread(image_path)
83
84                  outputs = model(image,vec)
85
86                  income = income.to(device)
87
88                  loss = loss_fn(outputs, income.unsqueeze(0))
89                  loss_val += loss.item()
90
91                  if epoch == 2:
92                      val_pred.append((image_path,outputs.cpu()))
93
94          val_loss_t = loss_val / n_batches_val
95          val_loss_list.append(val_loss_t)
96          train_loss_t = loss_train / n_batches_train
97          train_loss_list.append(train_loss_t)
98
99          print('Training loss {} | Validation loss {}'.format(train_loss_t, val_loss_t))
```

```
        torch.save(model.state_dict(), 'nnincome-{}-{}.pth'.format(epoch,val_loss_t))

    results = {
        'train_loss': train_loss_list ,
        'val_loss': val_loss_list,
        'pred': val_pred
    }

    with open('training_res.pkl', 'wb') as f:
        pickle.dump(results, f)

# Neural network architecture predicting the income per capita for a given city
class incomeModel(nn.Module):
    def __init__(self, backbone_name,custom_bias):
        super(incomeModel, self).__init__()

        self.backbone = timm.create_model(backbone_name,
        ↪  pretrained=True,num_classes=0,in_chans=3, global_pool='catavgmax')
        self.finallayer = nn.Linear(512, 1)
        with torch.no_grad():
            self.finallayer.bias.copy_(custom_bias)

        self.fc_layers = nn.Sequential(
            nn.LazyLinear(1024),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(1024, 512),
            nn.ReLU(),
            nn.Dropout(0.5),
            self.finallayer
        )

    def forward(self, image,y):

        (h,w,_) = image.shape
        size = 224
        min_h = h + size-h%size; min_w = w + size-w%size

        transform = album.PadIfNeeded(min_height=min_h, min_width=min_w, always_apply=True,
        ↪  border_mode=0)
        image2 = transform(image=image)['image']

        patches = patchify(image2, (size, size, 3), step=size)
        patches = np.squeeze(patches)
        rows, cols, _, _, _ = patches.shape
        x = torch.zeros(1,1024)

        for i in range(rows):
            for j in range(cols):

                img_tensor = transforms.ToTensor()(patches[i,j].astype('float32'))

```

```
                if img_tensor.std() != 0:
                    img_tensor = (img_tensor - img_tensor.mean()) / img_tensor.std()

                img_tensor = img_tensor.to(device)

                temp = self.backbone(img_tensor.unsqueeze(0))
                x += temp.cpu().detach()


        x = x.to(device)
        y = y.to(device)
        x = torch.cat((x, y), dim=1)
        x = self.fc_layers(x)

        return x


# Training the architecture for 7 epochs

train_dataset = forecastingDataset('train','infrastructures-income.csv')
valid_dataset = forecastingDataset('val','infrastructures-income.csv')

model = incomeModel('resnet18',49587)

model.to(device)

n_epochs = 7
loss_fn = nn.MSELoss()

optimizer = torch.optim.Adam([dict(params=model.parameters(), lr=0.01,weight_decay=1e-4)])


training_loop(n_epochs, optimizer, model, loss_fn, train_dataset,valid_dataset)

```