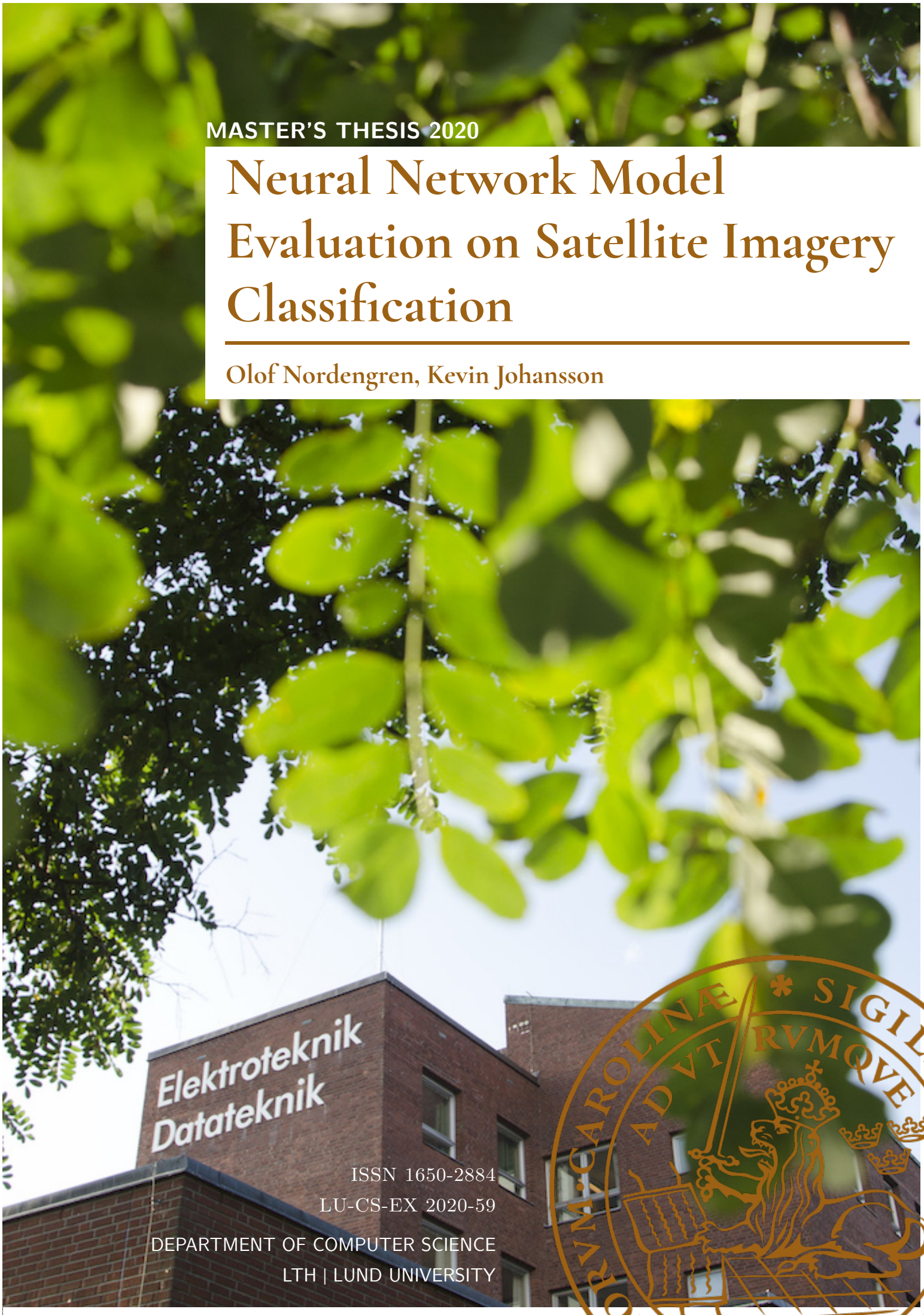


MASTER'S THESIS 2020

Neural Network Model Evaluation on Satellite Imagery Classification

Olof Nordengren, Kevin Johansson

ISSN 1650-2884
LU-CS-EX 2020-59

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX 2020-59

Neural Network Model Evaluation on Satellite Imagery Classification

Olof Nordengren, Kevin Johansson

Neural Network Model Evaluation on Satellite Imagery Classification

Olof Nordengren
dat13ono@student.lu.se

Kevin Johansson
dat14kjo@student.lu.se

October 3, 2020

Master's thesis work carried out at AFRY.

Supervisors: Frank Camara, frank.camara@afry.com
Jacek Malec, jacek.malec@cs.lth.se

Examiner: Jörn Janneck, jorn.janneck@cs.lth.se

Abstract

Digital maps contain a lot of information and heavily depend on satellite imagery. Since these maps are built manually, mistakes are inevitable. Within the maps there are various functional city zones, for example residential areas, industrial areas, forest areas and so on. These real-world areas are also subject to change, rendering the map inaccurate, which creates a need for maintenance.

In this paper, we evaluate eight neural network architectures on a classification problem, with the goal of finding the best performing architecture, to be used to automatically aid with the maintenance problem presented above. Our data consists of satellite imagery covering cities and the classes consist of six different functional city zones. We do coarse-grained image segmentation and make use of OpenStreetMap to create a process for automatic annotation. Our imagery consists of five bands; a greyscale band, colour bands (RGB) and an infra-red band. We evaluate each architecture with and without ImageNet based transfer learning. The models trained without transfer learning are fed all five bands of the imagery, while the transfer learning models are only fed the colour bands.

The results indicate that the models trained with all the available bands perform better, regardless of architecture. The best-performing architecture was DenseNet (89.5% test accuracy with all bands), which we argue makes sense because of the high detail-level that satellite imagery contains.

Keywords: Satellite imagery, Map Completeness, Machine Learning, Neural Networks, Transfer Learning, Network Evaluation, Architectures, OpenStreetMap

Acknowledgements

First and foremost, we would like to thank our supervisor, Jacek Malec, for the regular feedback that he has provided throughout this master's thesis. Secondly, we would also like to thank Jörn Janneck for his input on the report writing and work process, as well as Marcus Klang for his invaluable neural network insights on our problem. Finally, we would like to express our gratitude to AFRY for making this thesis possible, and especially Frank Camara, for his unwavering support and belief in our capabilities.

Contents

Chapter 1
Introduction

This chapter introduces our problem and what this thesis is all about. The background is presented along with the purpose and problem statement of this paper, followed by the outline.

1.1 Background

Satellites provide a wide range of services that are used for several common applications. Two services, satellite imagery and positional systems (GPSS), are essential components in the development of a *map service*. The main element of a map service is the actual map, that is constructed through utilising satellite imagery and area-specific information. A map conveys a detailed portraiture of the world, with the goal of resembling the real world with as high precision as possible. The degree of how accurately a map resembles the real world is also called *map completeness*.

Our world is very dynamic and is constantly changing, and differences in satellite imagery and the corresponding map area are therefore common, resulting in a need to regularly update the map to maintain a high degree of map completeness. Furthermore, it is generally simple for us to identify objects and areas in satellite imagery, and we make the assumption that this also is the case for machine learning methods, more specifically neural networks. This will be explored as a potential aid to help maintain a high degree of map completeness.

A neural network is a machine learning algorithm that is widely used for classification problems, i.e. the task of classifying objects into a predefined set of classes/type of objects. Neural networks are said to suffer from *the curse of dimensionality*, which means that there are a vast number of possibilities whilst working with neural networks, making it infeasible to iterate over all of them when trying to find the best match for any problem (Goodfellow, Bengio, and Courville 2016, p.155). We will therefore, when working on the map completeness issue, also try to narrow the search space when using neural networks with satellite imagery. An important aspect to consider when working with *computer vision* as in our case, is the

usage of *transfer learning* (Chollet 2018, p.143). Transfer learning is the concept of initialising a neural network with already learned knowledge from another problem, i.e. a pretrained network. This is conceptually meant to make training faster because images have common traits and thus the neural network algorithm does not have to learn from scratch. Applying transfer learning is particularly interesting in the case of satellite imagery because the images normally contain more image data than only colour, but transfer learning is only applicable to colour data (RGB), resulting in a trade-off since information in the satellite imagery has to be discarded to be able to use transfer learning.

The satellite imagery used for this thesis is acquired from DigitalGlobe, the global leader in satellite imagery. (*Global provider of advanced space technology solutions* 2020) Our imagery will be used with various neural networks, using functional city zones as classes. A functional city zone is an area within a city with a specific function, e.g. residential areas, industrial areas and parks. (LEMAR 2015) In order to map what area different parts of our imagery belong to, we will utilise OpenStreetMap (OSM), which is a nonprofit open-source project that creates and maintains geographical information, including functional city areas and location data. (OpenStreetMap 2020a) OSM is essential for our thesis because it allows us to automatically annotate our dataset, and it would probably be infeasible without it.

1.2 Purpose

The purpose of our thesis is twofold. Due to the vast search space of neural networks we aim to shed light on which existing neural network model architectures work well with satellite imagery, and which do not. This process is tiresome and time–consuming; therefore we reason that any progress in finding some starting point in this regard can save a lot of time for future projects. Also, we aim to find out what possibilities there might be in applying neural networks to satellite imagery with the specific purpose of helping with the process of maintaining a high degree of map completeness.

1.3 Problem Statement

There exist quite a few different neural network model architectures. We aim to evaluate a relevant set of these already developed models for a set of classes (functional city zones), with very high-resolution satellite imagery. The baseline of our evaluation will consist of models where transfer learning has been applied. The results from these models will be compared with models trained from scratch but that contain the full scope of the available satellite image data. We also aim to visualise our results, meaning that we wish to show how different areas in our satellite imagery are classified, rather than only display numbers representing performance.

The following is a formalisation of our problem description:

Q1. Is there a neural network model architecture that is superior when applied to satellite imagery? If so, why?

Q2. Are there gains to be made from applying transfer learning to each architecture but having to use less of the available satellite image data?

Q3. What applications could be developed through our work to help maintain a high degree of map completeness?

Our aim is that the answers to these questions will be useful for future work within this area since they may reduce the search space of neural network training with satellite imagery. Through our work we also aim to produce some potential starting points for future projects regarding different ways to help with maintaining map completeness.

1.4 General Constraints

The absolute majority of our constraints are bound to the fact that our time is limited. Training neural networks takes time, and the number of ways that one can do it is too big to cover. Further, to obtain good results it is necessary to have data that properly represents our problem. We use geographical data from OSM to do our annotation, which means that OSM dictates how well our images are labelled. This data is entered by people and is therefore not perfect. As we do not have the time to go over each segmented image to see if it is properly classified, we constrain ourselves by stating that the OSM data is sufficient (knowing it is not perfect). Neural network training also requires adequate hardware that is very financially expensive. Since we have to evaluate multiple neural network models with many different sets of parameters we require hardware that can reduce the time required to train each model. This adds yet another time-related constraint because we do not have unlimited access to powerful hardware, but have to gain access to some cloud service that meets our demands.

1.5 Contributions

The work conducted for this master's thesis can be split into four main areas: Data collection and processing, machine learning setup, training, and evaluation. Both authors of this paper have contributed to important design choices throughout the process and constant communication has taken place, discussing each part even as one or the other conducted the actual implementation. Thereby, whilst proceeding with the work, some areas have been focused on with more emphasis by one or the other. Specifically, the data collection and processing task was a long and daunting one. The work for collecting the data from OSM was mostly conducted by Olof, whilst assessing the quality of the data and the processing of the data was done jointly. The machine learning setup, i.e. virtual machine setup, machine learning implementation and training platform, was focused upon more by Kevin. The training and evaluation process was done with equal contribution.

1.6 Outline

This paper consists of chapters, both analytical and descriptive. Firstly, we describe the necessary theory to understand all of the decisions and trade-offs taken throughout our work process. The theory is covered in the two following chapters, satellite imagery and machine learning. The satellite imagery chapter describes what satellite imagery is and what makes it

special to work with, as well as the specific data for our thesis. The machine learning chapter briefly covers the area in general but describes neural networks in further depth, which makes the bulk of our work understandable. Furthermore, the fourth and fifth chapters describe and analyse our method. The data chapter describes what necessary preprocessing we have done to our data to achieve the right format for neural networks. The implementation chapter describes the practical parts of our thesis, how we executed our work and why we did it like we did. Finally, the last three chapters present our results, cover our discussion and summarise our conclusions.

Chapter 2
Satellite Imagery

This chapter describes satellite imagery and important aspects of it. A review of our specific satellite imagery is also given.

2.1 Spectral Bands

Satellite images are similar to conventional images, but have a few key differences. Conventional images have colours constructed using red, green and blue (RGB) values. Each pixel in an image has exactly three values, representing each RGB value. Satellite images are compositions of many images, photographed at different wavelengths in order to capture different light. The values from each wavelength are contained in something called *Spectral bands*. (David DiBiase 2018, ch. 8, p. 11) For example, a common combination is to use 3 bands representing the red, green and blue parts of the spectrum. These have wavelengths of about 600–690, 520–600, and 450–520 nanometers, respectively. (ibid., ch. 8, p. 11) These bands cover the visual spectrum, and are used to create the satellite images we are used to seeing. However, there are also bands covering other parts of the spectrum, invisible to the eye. Certain satellite images may have up to 16 bands, with wavelengths up to 15000 nanometers. (ibid., ch. 8, p. 11) In this thesis, we have utilised five bands: a greyscale composite of the colour bands, the three RGB colour bands, and a near–infrared band called the Alpha band. This Alpha band can be especially useful when detecting vegetation or water. This is due to the fact that reflectance from certain colours is similar in the visible spectrum, but differentiate in the near– and mid–infrared wavelengths. (ibid., ch. 8, p. 4) For each city, we were able to acquire two different images. One, which contained a single band with greyscale data taken with high resolution, and another image, with four bands, containing RGB colour bands and a single infrared band. We used both of these images as a representation of each city.

2.2 Location Data

The satellite imagery used in this project are provided as *.tif*–files. In the metadata of a .tif–file are, in addition to all bands, many other pieces of important data contained. (Sk. Sazid Mahammad 2009) All reading and writing to these files have been done through a Python API called *RasterIO*. (RasterIO 2016) One important piece of information is the location data, which represents where in the world the image is located. This information is represented by the coordinates of the corners of the image. While useless in a machine learning sense, this was crucial for us when annotating our dataset. The coordinates exist in a specific format and are defined by a coordinate system, and there are many such coordinate systems. (EPSG 2012) The most familiar of these is *EPSG:4236*, which uses longitude and latitude. This coordinate system is global, whereas some other systems may be specific to certain parts of the globe. A local coordinate system may give a more honest visual representation of distances within an image, but loses the ability to display objects correctly outside of its domain. Both the details of the corners, and the coordinate system, are contained in the image metadata. We have solely used EPSG:4236.

2.3 Digital Globe Images

There are many sources of satellite imagery to be found online. The purpose of this thesis is to classify rather detailed components of a city, so we required images with a resolution no greater than 1–2 meters per pixel. An additional constraint was that the data had to be free of charge for research purposes. We used images from Digital Globe, which is a company that owns and operates some of the most high–resolution commercial Earth Imaging satellites. (*Global provider of advanced space technology solutions* 2020) Most of the images from these satellites are not free of cost, but they offer a selection of free samples which are free to use for research purposes.

The free samples from Digital Globe cover four cities: Stockholm, Washington, Tripoli, and Rio de Janeiro. (DigitalGlobe 2020) All images are available in two formats – the first containing only one band, with a greyscale composite, and the second containing four bands, with RGB colour image data and one infrared band called the Alpha band. The colour image from Washington had even more bands, with a total of 8, but we decided to discard the surplus bands and only use the first four bands. The greyscale data had four times the resolution of the colour data. In practical terms, this means that a greyscale pixel of Stockholm is 0.42 meters wide, whereas a colour pixel of Stockholm is 1.68 meters wide. However, because the images are shot in different coordinate systems, at different latitudes, the pixel height varies.

(OpenStreetMap 2020c) The Stockholm grey image consist of 39926x17398 pixels, with a pixel being 0.42 m wide and 0.84 m high. This means that the entire image covers an area of about 16809x14614 meters. In Figure 2.1, we can see the full Stockholm greyscale image – but with pixels forced to the same width and height. Because of this, the image gets somewhat distorted from the ground truth. This distortion is not the same for all cities, because they are on different latitudes – for example, a pixel in Washington is 0.40 m wide and 0.50 m high. This distortion has little practical effect, and the features and relationships within the image are preserved even if the images are drawn out vertically.

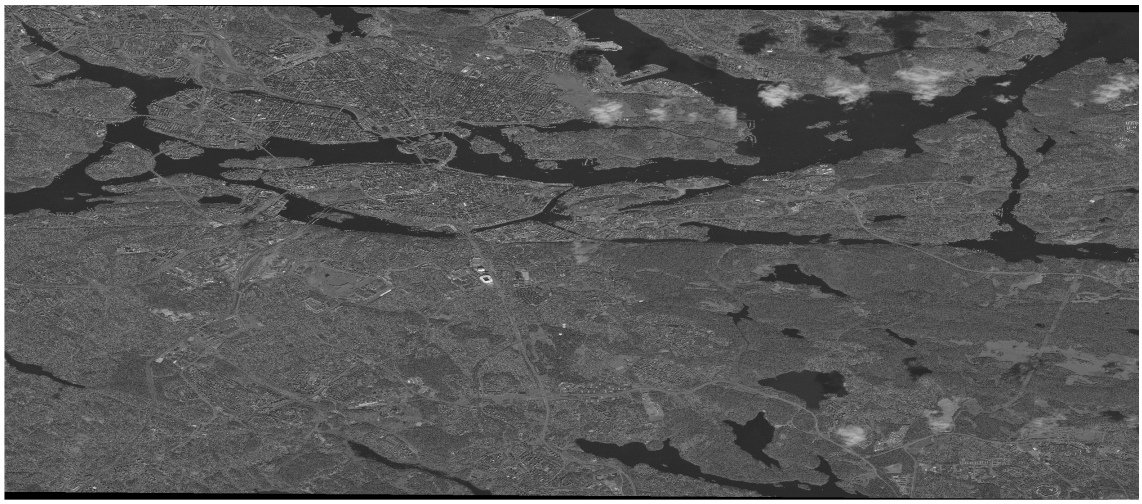


Figure 2.1: The full greyscale Stockholm satellite image

Chapter 3
Machine Learning

This chapter contains the bulk of the theory required to understand the conducted work. Firstly, a general overview of machine learning as an area is given, which is then diverged into the, in our case, more relevant area of neural networks. We have avoided the use of mathematical notation as much as possible, but still hope to convey the necessary understanding. The neural network section describes the essentials of a neural network, each specific architecture that we have evaluated for this thesis, as well as describing what practical issues arise.

3.1 Basics of Machine Learning

Machine learning belongs to the field of artificial intelligence and relies on mathematical principles. The field describes the development of programs with the capability of finding useful properties, patterns or rules in data (Russell and Norvig 2010).

The methods utilised in machine learning can enable an agent to do this without having any prior knowledge of the data; in other words, without being explicitly programmed to find said structure. This is more commonly known as *learning*. Learning represents an automatic search of the hypothesis space with the goal of finding a good representation of data. The search or learning algorithm is defined by the applied machine learning method.

The data applied to machine learning methods can basically consist of any data, for example, photos, tweets, audio recordings, newspapers or, in our case, satellite imagery. Data by itself is specifically known as unlabelled data, while labelled data combines each unlabelled data sample with a *tag*, *class*, or *label*. (Chollet 2018) This *label* provides some informative knowledge about the data and can be difficult to obtain.

The two main areas in machine learning are *supervised learning* and *unsupervised learning* (Russell and Norvig 2010, p.693-697). Supervised learning is primarily used for classification and regression problems and is the most commonly used technique. Unsupervised learning is mainly used for clustering, but also for anomaly detection and association learning problems and does not require labelled data. The purpose of unsupervised learning is to find a model

of the underlying structure in the data, without a predefined measure of success, to learn more about the data. This thesis will not describe unsupervised learning further as it is not the method we utilise.

Supervised learning, on the other hand, utilises training with labelled data. This enables making predictions as well as a way of measuring performance (and progress). The label of each data input, i.e. sample, is compared to each corresponding prediction to measure how accurate the model is. Furthermore, most supervised methods are *shallow*, meaning only one, sometimes a few, new representations are acquired (Chollet 2018). Images do not fit well with shallow methods because the structure of images is complex and detail-oriented. Therefore, finding useful representations of images often requires *depth*. This corresponds to using methods that can obtain tens, sometimes hundreds of representations, each based on the previous. This can be accomplished by a subfield of machine learning sometimes called *deep learning* (ibid., p.18). Deep learning utilises *neural networks* and resides at the core of our thesis work.

3.2 Neural Networks

The central problem when using neural networks is to find meaningful transformations of the data, each yielding a new representation that hopefully gets the agent closer to the expected output. Figure 3.1 illustrates a transformation which yields a new representation, and as such, a new representation is simply an alternative way to look at the same data; in this case, a simple coordinate change completely separates the two classes by the y axis, making them easily distinguishable.

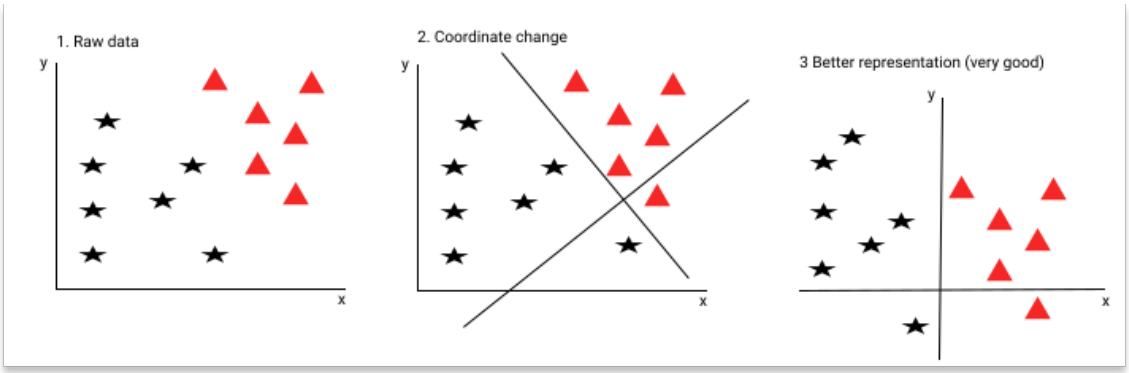


Figure 3.1: Transformation example

The idea behind applying neural networks on images is to find increasingly detailed representations that hopefully are useful for solving whatever the actual problem is (ibid.). A neural network performs function approximation based on simple mathematical properties. The network tries to approximate a function $f(x_1, x_2, x_3..., x_n)$, where n equals the number of variables in each data sample. The most important aspects of training a neural networks are listed below, (ibid.) and will be covered individually:

- The *input data* with corresponding labels (target values)
- The *layers*, which are combined into a network

- The *loss function*, which calculates the *loss score*, a value describing the network's performance
- The *optimiser*, that controls the learning process through the loss score

Figure 3.2 illustrates the training feedback loop and the components' relationships to one another.

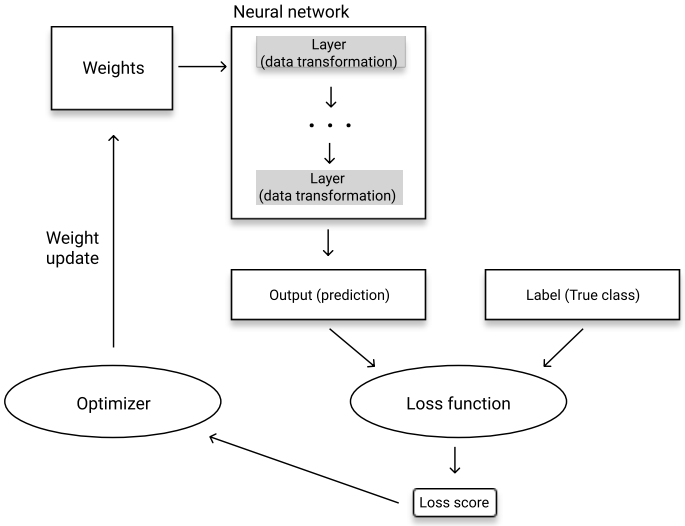


Figure 3.2: The neural network training process, or feedback loop. The network's weights are updated for each iteration with the goal of lowering the loss score yielded by the loss function, i.e. get the output as close to the true class as possible.

3.2.1 Input data, Tensors and Data Representations

The data fed to a neural network's first layer (the input layer) is called input data. Neural networks can be trained on any data, but certain *preprocessing* is necessary to transform the data into the necessary tensor format. A *tensor*, in machine learning, is a generalisation of the concept of scalars, vectors and matrices (Brownlee 2018). A 1D-tensor is equal to a vector, a 3D-tensor is equal to a vector of matrices, and so on. In addition, a tensor operation corresponds to any mathematical operation that is defined for the current N-dimensional tensors. All data applied to neural networks have to be tensors with predefined shape, which means that the raw data needs to be transformed into the correct tensor format before any training can take place.

To clarify, a transform of data, and its resulting data representation, is accomplished by neural networks through previously mentioned tensor operations. Each tensor operation attempts to extract useful information within the data that contributes to the prediction of the network. The tensor operations that are applied to the data at a certain point is defined by the corresponding layer.

3.2.2 Layers

A *layer* in a neural network consists of a set of *nodes*, or artificial *neurons*. A *neuron*, in turn, is essentially a mathematical function that executes a few operations on the input data. Firstly, the neuron applies the same tensor operations on all of its input values. Secondly, the results from these tensor operations are concatenated, commonly summed, to form a single value. Finally, the output from this is sent through a nonlinearity, the neuron's *activation* (Zhou 2019). There are various forms of activation functions, but somewhat simplified the functions decide whether the output of a neuron should be 1, or 0, based on a predefined threshold. Figure 3.3 illustrates a simple neuron receiving four inputs. The figure also reveals the essential *weights* of the neuron. A neuron has weights associated to each input value and the weights are a part of the neuron's function. The weights are updated during training and this is what enables the *learning* of neural networks. (Goodfellow, Bengio, and Courville 2016)

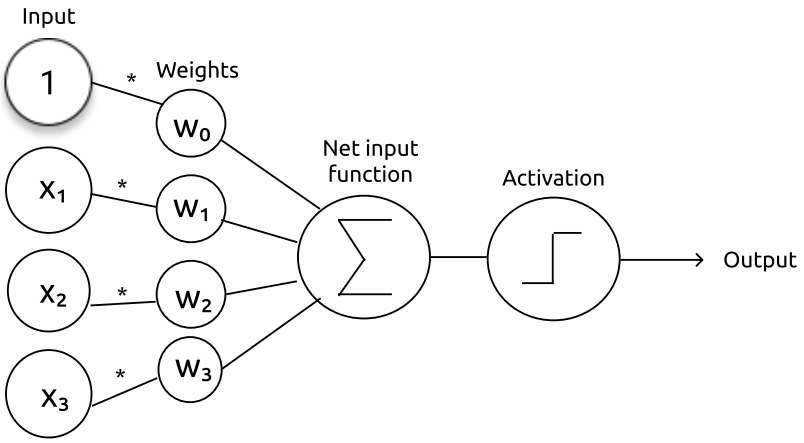


Figure 3.3: A simple neuron receiving four inputs, and therefore having four weights. The net input function is a weighted sum of the input and the activation function determines the output of the neuron based on the weighted sum.

Furthermore, Figure 3.4 illustrates a simple neural network consisting of five layers: the input layer, three intermediate layers and the output layer. The output from one layer acts as the input to the following layer until the final layer is reached. There are different types of layers, applying different tensor operations to its input. The primary layers used for image data are presented next, but be aware that there exist others.

A *dense* layer is the most simple, and commonly used. The neuron in figure 3.3 illustrates a neuron in a dense layer. Each input value is multiplied with its corresponding weight, then summed to a weighted sum. The dense layer is also called the *fully-connected layer* because each neuron in layer $l + 1$ is connected to each neuron in layer l. In our case, this layer is solely used as the output layer.

A *Convolutional* layer constitute the key component of doing *computer vision* with neural networks, i.e. machine learning with images (Chollet 2018, p. 242). Convolutional layers enable sending the input data as 2D tensors, i.e. matrices, equivalent to image data, and their operations also preserve this shape, which results in the *spatial* and temporal information of

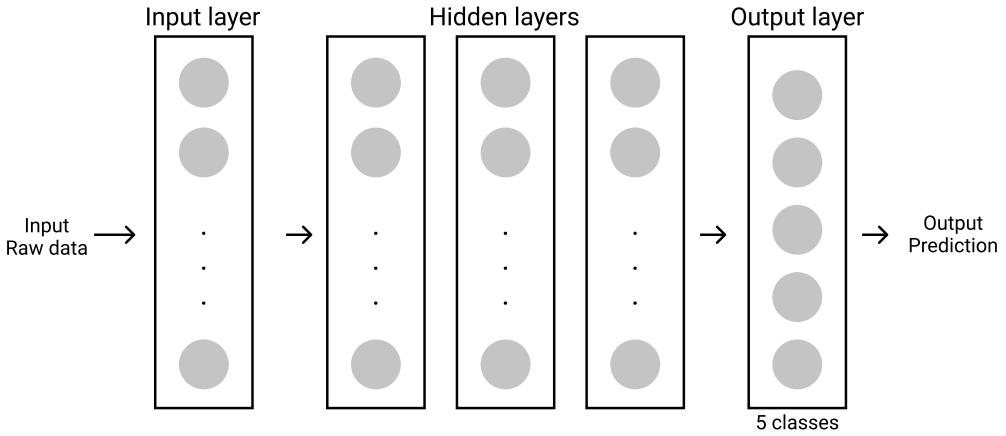


Figure 3.4: A simple neural network with three hidden layers and an output layer consisting of 5 nodes. The output from one layer is received as the input to the following layer.

the data being preserved, which is essential in learning from images. The convolutional layer iterates over the matrix with what is called a *kernel*, as illustrated in Figure 3.5. The kernel consists of an n times n matrix, each value in the kernel corresponding to the weights of the layer. Firstly, the kernel applies dot multiplication with the subregion of the input that it is currently "hovering" over, the result of each iteration is summed and becomes part of the resulting *feature map* that is formed from this process, hopefully extracting useful information. A convolutional layer has a predefined number of said kernels, each resulting in its own feature map. The yielded set of feature maps defines the output of a convolutional layer.

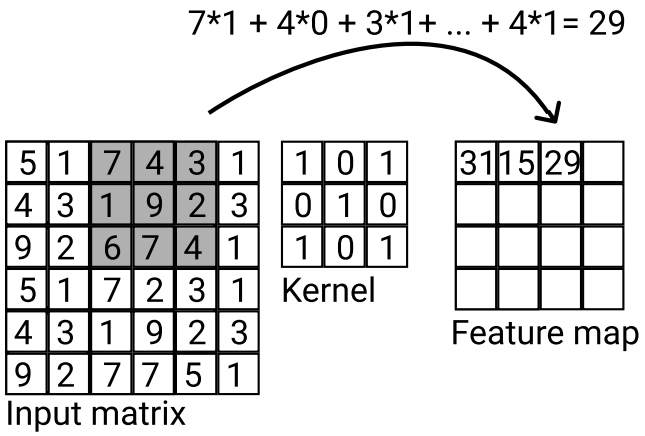


Figure 3.5: An illustration of the convolutional layer operation. The current kernel position is represented by the grayed out area. The kernel has already filtered over the two first positions, resulting in the values 31 and 15 for the feature map. In the figure you can also see the dot multiplication for the current iteration.

A *max-pooling* layer applies a max-filter to, usually non-overlapping, subregions of the input representation (ibid., p.321) The objective is to down-scale the input matrix (image)

whilst conserving the information maintained within that region, partly to increase performance, but also to act as a form of *regularisation*.

A *flatten* layer reduces the ND Tensor to a 1D Tensor, in our case, reducing the matrix data to a vector. This translates to a 10x10 matrix transforming to a 100-dimensional vector. This destroys the spatial structure of the data, but all features, useful representations, have already been extracted from the data. The output of the flatten layer is then sent to the output layer, which is a dense layer. The dense layer produces the output vector of the network, consisting of one value per existing class, interpreted as the probability that the current input sample belongs to each class. (Dependent on the activation function of the layer) The output vector becomes the input of the loss function, which is discussed next.

3.2.3 Loss Function

The *loss function* of a neural network acts as the jury, judge and executioner of a neural network, yielding a value describing the performance of the network for the current input sample (Goodfellow, Bengio, and Courville 2016, p.82). The loss function knows the correct label, or *target value*, of the input sample and can thus evaluate how "far off" the neural network is from predicting the correct label. The magnitude of the output describes the quality of the prediction, where a value of zero corresponds to a perfect prediction. Moreover, the output, or *loss score*, is a value representing all predictions, that is used by the optimiser with the purpose of helping the neural network to do better on the predicted samples, i.e. updating the weights of the network with the goal of lowering the loss score – recall the training process illustrated in Figure 3.2.

3.2.4 Optimiser

Optimisation, in a general sense, is the endeavour of improving the result or process of any task. As mentioned, a reduced loss score equals better predictions; therefore, in the case of neural networks, the optimisation algorithm strives to lower the loss score yielded from the loss function. This is accomplished by calculating the derivative (gradient) of the loss function with respect to each weight and then moving the weights in the negative direction of the derivative. This essentially corresponds to calculating how a numerical change to a weight alters the loss score, and then updating the weight values so that the loss score is reduced. This is a reference of *gradient-based optimisation*, which is what neural networks do by utilising *gradient descent* and *back propagation* (Murphy 2012, p.569).

Gradient descent is an optimisation method commonly used to train neural networks. Calculating the gradient of a neural network function is infeasible because the dimensions of the function is often in the millions. This means that solving the problem analytically, setting the weights to their optimal values, is not possible. Instead, the loss function is used and derived with respect to the network weights. The weights are then updated so that the loss function is minimised. The gradient descent algorithm that is covered below is known as stochastic mini-batch gradient descent (Chollet 2018, p. 49)

1. Randomly select N samples from the labelled dataset.
2. Do all calculations at each layer until a prediction is yielded. This step corresponds to a *forward pass*.

3. Evaluate the given loss function for the N samples. The yielded *loss value* is what the gradient descent algorithm is trying to minimise.
4. Evaluate the *gradient* of the loss function, i.e. the partial derivative of the loss function with respect to each of the parameters in the network. This is accomplished by the *back propagation* algorithm.
5. Update each weight by the negative of the partial derivative for the same weight. The magnitude of this update is controlled by a parameter known as the *learning rate*.

3.3 Neural Network Architectures

As depicted in Figure 3.4, layers, which consist of a bunch of neurons, are interconnected with other layers to form the architecture of a neural network model. These connections can look in various ways, each with its own merits. The architectures that we have evaluated are based on three kinds of structuring of layers, yielding different connections. The first is the *linear structure*, which basically is resembled in Figure 3.4. Here the layers are simply stacked upon each other, where the data representation from one layer acts as the input to the next. Secondly, a *residual layer* is a layer whose output can be redirected and connect to other layers deeper in the architecture. Finally, an *inception module* consist of at least two layers that each receive the same input. Instead of making the model deeper it adds width, applying different kernel sizes to the same input with the reasoning that extracting features of varying size requires different kernel sizes. The result from each layer in the inception module is then concatenated in a filter before being sent as input to the following layer (ibid., p.235). A residual layer and an inception module is illustrated in Figure 3.6.

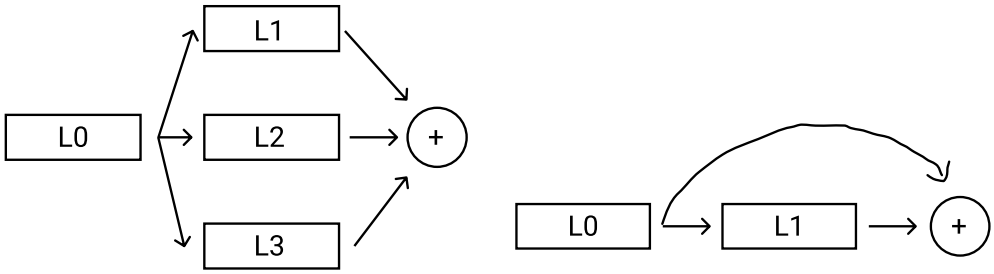


Figure 3.6: The left image illustrates a very simple inception module with three layers receiving the input of layer L0. The result from L1, L2 and L3 is then concatenated before being sent to the subsequent layer. The image to the right illustrates a residual layer, where the output of the first layer is sent to both the second and the third layer.

At least one of the presented ways to interconnect layers are used in each established architecture that we have used for our evaluation. Also, every network described below have been pretrained on ImageNet. ImageNet is a large dataset, standardised when used for evaluation to contain 1.3 million images split on 1000 classes (2016). These weights are saved and can be re-used, making it easy to apply transfer learning on our problem using these architectures.

The below listed architectures are the ones that are being evaluated in this paper. They were chosen due to their historical relevance and popularity, as well as breadth of different design choices.

- ***VGG16***: Published in 2015, VGG16 consists of 16 layers and expands on the classic AlexNet (Krizhevsky, Sutskever, and Hinton 2012) architecture by replacing some of the convolutional layers using larger kernels with multiple convolutional layers using 3x3 kernels. Overall, VGG16 is easy to understand; all layers are linearly stacked and consist of blocks of convolutional and max-pooling layers. VGG16 has 138 million trainable parameters, meaning we need a lot of GPU internal RAM. (Simonyan and Zisserman 2014)
- ***VGG19***: VGG19 was introduced in the same paper as VGG16 and is essentially the same architecture. The sole difference is that VGG19 is has another block of layers, meaning it ends up somewhat deeper, with 19 layers. (ibid.)
- ***ResNet50***: ResNet, which was introduced in 2015, was the first network to include residual layers. The number of trainable parameters in ResNet50 is 25.6 million, which is much lower than any VGG architecture even though it has 34 additional layers.(He et al. 2015)
- ***ResNet101***: ResNet101 was introduced at the same time as ResNet50 and they are essentially equal, but ResNet101 has 101 layers. The number of parameters is 44.5 million and takes longer to train than ResNet50. (ibid.)
- ***InceptionV3***: InceptionV3 was published in late 2015, and makes substantial use of inception modules. The network works really well on ImageNet but uses fewer parameters, only 25 million.(Szegedy, Vanhoucke, et al. 2015)
- ***Xception***: Xception, published in 2017, builds upon the same ideas as InceptionV3 with certain differences surrounding the inception modules. Xception also utilises residual layers in its first blocks.(Chollet 2016)
- ***DenseNet***: DenseNet was published in 2018. This architecture maximises the concept of residual layers, namely, each layer receives input from every preceding layer and passes on its own feature-maps to all subsequent layers. (Huang et al. 2016)
- ***InceptionResNetV2***: InceptionResNet combines the two designs used in the Inception and ResNet networks. It was published in 2016, and essentially implies that by training an inception network together with residual layers, training times are drastically reduced. This network also outperforms similarly expensive inception networks, albeit by a thin margin. (Szegedy, Ioffe, et al. 2016)

3.4 Neural Networks in Practice

Each element presented so far (the architecture (interconnected layers), the input, the choice of loss function and optimiser) add complexity to the process of achieving a neural network model that performs as desired. There are different kinds of optimisers and loss functions, infinite ways to select and interconnect layers and various ways to format the raw data. This

curse of dimensionality is obviously a problem, but there often exist general practices that typically work well, depending on the problem at hand. Instead of worrying too much about the choices of these broader elements, it is more commonly the *hyperparameters* of the network that are finetuned (varied) when trying to increase the performance of a model. The most common parameters are the number of *epochs*, the *batch size* and the *learning rate*.

3.4.1 Hyperparameters

An *epoch* determines how many times the model should iterate over all of the training data. Therefore, the number of epochs dictates how many times the model will get to update its weights before terminating. The *batch size* corresponds to the N chosen samples in the gradient descent algorithm and thus decides how many samples the loss function will be evaluated on. This parameter is very important and can heavily impact the performance of the agent and the time it takes to iterate over one epoch, and thus also the time it takes to train the agent. Finally, the *learning rate* is a constant that determines how large the weight updates should be during the optimisation step and can be static or dynamically changed during training, depending on the chosen optimisation algorithm.

The just mentioned hyperparameters add further complexity to the training process, and they have to be just right so that the neural network can learn proper representations of the data. This all boils down to achieving a model that can find general representations of the data, and perform well on unseen data. The problem of machine learning methods performing better on training data than on unseen data is known as *overfitting*, and is a problem for all methods, including neural networks.

3.4.2 Overfitting

The goal when training a neural network is to properly fit the neural network to the data. This "proper fitting" corresponds to approximating a function so that it will perform well on data it has not seen before, i.e. generalise to unseen data. A network that is *overfitting* is essentially updating its weights to map the training data too specifically, resulting in worse performance on unseen data. Figure 3.7 illustrates a simple two–dimensional example on how a function can overfit to the seen data. Changing the architecture, optimiser and acquiring more and/or more proper data for the problem at hand and finetuning hyperparameters can all affect the performance of the network and mitigate the effect of overfitting (Mitchell 1997, p.68). The key is to stop training just before overfitting starts, to achieve a model that performs as well as possible on unseen data.

3.4.3 Training, Validation and Test Data

Since the goal is to generalise to unseen data, evaluating the network on the training data is not good practice. Therefore, it is common to divide the prepared data into three sets: training, validation and test. The training data is used to train the network, to minimise the loss function. In turn, the validation data is fed to the network after each epoch to get an indication of how well the network is generalising to unseen data. Most importantly, no weights are updated using the validation data, i.e. no gradient descent step is done. Finally,

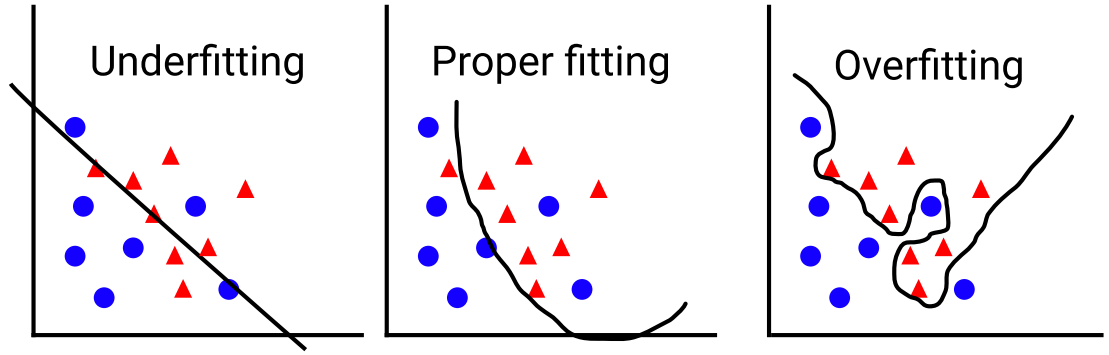


Figure 3.7: An illustration of three neural network function approximations. Going from the left, the first function did not learn anything about the data, and is thus underfitting. The middle function does a proper mapping that is not too accurate on the training data, which hopefully means that it generalises well. The function to the right is overfitting, i.e. being too specifically mapped to the training data, and will probably perform worse on unseen data.

the test data is an additional subset of all the data, acting as an unbiased final evaluation of the network. This test sample is important because the network gets biased to the validation data because the network can be altered to improve the performance on the validation data, meaning it essentially overfits on the validation data with enough alteration of the network. A common split of the training, validation and test samples are 60/20/20%, respectively, but it depends on the amount of available data and what the problem is.

3.5 Evaluation

Certain *metrics* are used to evaluate neural networks. The chosen evaluation metrics depend on the problem at hand, but as we have a classification problem, commonly used metrics for this case will be presented.

Accuracy

The *accuracy* of a model corresponds to the ratio of correctly classified samples. For example, given a data set of 1000 samples, a network achieves 90% accuracy. This means that 900 samples were correctly classified, while 100 were wrongly classified. This however, does not say anything about the underlying performance of the network, as it can be "lucky" when getting the prediction right. The reason that this is not a very good metric to really understand how well the network is performing, is that the model returns an N-dimensional vector (1D tensor) with one value for each class. This value represents the probability that the current sample belongs to each class. Given a problem with two classes and a sample, a network returns $[0.56, 0.44]$. Assuming the first class is correct, the accuracy of this model will increase. However, the model was not very sure about its prediction, since the two values were really close. This uncertainty can be displayed by looking at these values, but is also seen in the *loss score*.

Loss Score

The *loss score* is the value that the loss function returns and is what the neural network algorithm is trying to minimise. The loss function evaluates the loss score with respect to every value in the vector mentioned above. The loss score will only be 0 if a sample fed to the network returns a vector as [1, 0, ..., 0], given that the prediction is correct. This means that the even though a network might classify a sample correctly, the loss score describes how accurate the yielded prediction was. This is also the primary metric that we have used when we evaluated the performance of our networks. In addition to the accuracy and the loss score, there exist metrics that indicate how a network performs on individual classes.

Recall, Precision and F1 score

The *Recall*, *precision* and *F1 score* are closely related metrics that look at how a machine learning classification method performs on each class (Murphy 2012, p.182-183). The four values used to calculate these metrics are described given a neural network, two classes P and N and two samples S1 and S2, belonging to class P and N, respectively.

- **True Positive (TP)**: The network correctly classified S1 as belonging to P.
- **False Positive (FP)**: The network incorrectly classified S2 as belonging to P.
- **True Negative (TN)**: The network correctly classified S2 as belonging to N.
- **False Negative (FN)**: The network incorrectly classified S1 as belonging to N.

This can easily be extended to more than two classes, and basically describes how the network is predicting all samples. In turn, recall, precision and F1 score are evaluated as

$$Recall = \frac{TP}{TP + FN} \tag{3.1}$$

$$Precision = \frac{TP}{TP + FP} \tag{3.2}$$

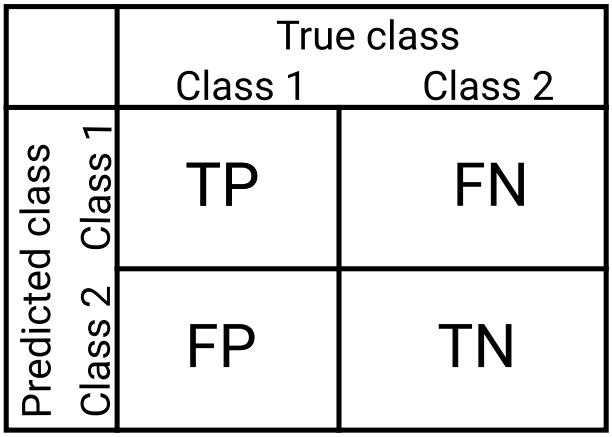
$$F_1 = 2 * \frac{Precision * Recall}{Precision + Recall} = \frac{2TP}{2TP + FP + FN} \tag{3.3}$$

In words, recall yields the hit rate, or true positive rate, of a class. Precision yields the ratio between the samples correctly classified as a class A, and all samples classified as belonging to A. Finally, the F1 score yields the harmonic mean of the two. We used these metrics to evaluate how our networks performed on individual classes. In particular, it is very easy to visualise this with a *confusion matrix*.

Confusion Matrix

A *confusion matrix* is an extremely useful table that displays the performance of a classifier. Figure **??** illustrates a confusion matrix. Recall and precision can easily be evaluated from looking at the table. Each column holds the true class and each row the predicted class. So summing up a column equals the number of samples for that class, while summing a row equals the number of predictions made for that class. This enables an easy way to evaluate

the four values used to evaluate recall, precision and F1 score. The tables also provides a great way to analyse and compare models when trying to conclude how they perform on certain classes, and why.



:
confMatrix

Chapter 4
Data

This chapter describes all of the preprocessing that we have conducted on our data to prepare it for neural network training. Firstly, the segmentation process is presented, i.e. the process of dividing, or tiling, our large satellite images to smaller pieces, which produces our dataset. Secondly, the annotation process is described and how we used OSM to do it. Thirdly, the final and more specific preprocessing of our actual dataset is described, to finally obtain the necessary tensor format.

4.1 Tiling

We had a multi–class image recognition problem to solve. For this, we segmented our satellite images into tiles, small enough to contain only one class, and big enough to contain enough detail. We decided a size of 256x256 pixels for the greyscale images. This translates to about 100 meters wide and 100–200 meters high, the height being inexact due to differing latitudes of the images. The size of a tile in Stockholm is 102.4x204.8 , but the same number for Washington is 99.8x133 . However, this discrepancy did not matter much – the only downside was that objects further from the equator were seen as more horizontally *drawn out* by the network. The tiling and other manipulation of our satellite imagery was done in QGIS. QGIS is a tool that handles raster data, i.e. spatial data, such as aerial photography, or satellite imagery, and allowed us to visualise our imagery. (QGIS 2020)When tiling, the geospatial data of the image was preserved, meaning each tile also contained the coordinates of its corners.

Since the resolution of the colour images was 4 times lower, the colour images were tiled with an image size of 64x64, to match the real-world size of the greyscale images. However, the original greyscale image did not perfectly overlap with the original RGB image. The offset was not large, but even a small shift would affect every single image. To counter this, the original satellite images were cut to the same exact coordinates, in all corners. Tiling, after the images were cut, yielded satellite images that had an almost perfect overlap.

We acquired 46012 segmented (tiled) images. For every greyscale 256x256 image, we had

a corresponding 64x64 colour image. The colour images for all cities except Washington had four bands: red, green, blue, and alpha. Washington had an additional four infrared bands, which we decided not to use, because these bands were not present in any of the other cities. The 46012 images were distributed as follows:

- *Stockholm*: 10613
- *Washington*: 9801
- *Rio de Janeiro*: 14397
- *Tripoli*: 11201

In Figure 4.1, the difference in resolution between the colour and greyscale images is made apparent, but it is also evident that a colour image contains information that the greyscale image does not. This is especially useful when separating lush trees and green areas from buildings, or separating roofs from roads. Part of our problem statement was intended to answer how big of an impact this difference in detail has, both in terms of resolution and colour, since we trained the models with and without this combined information.

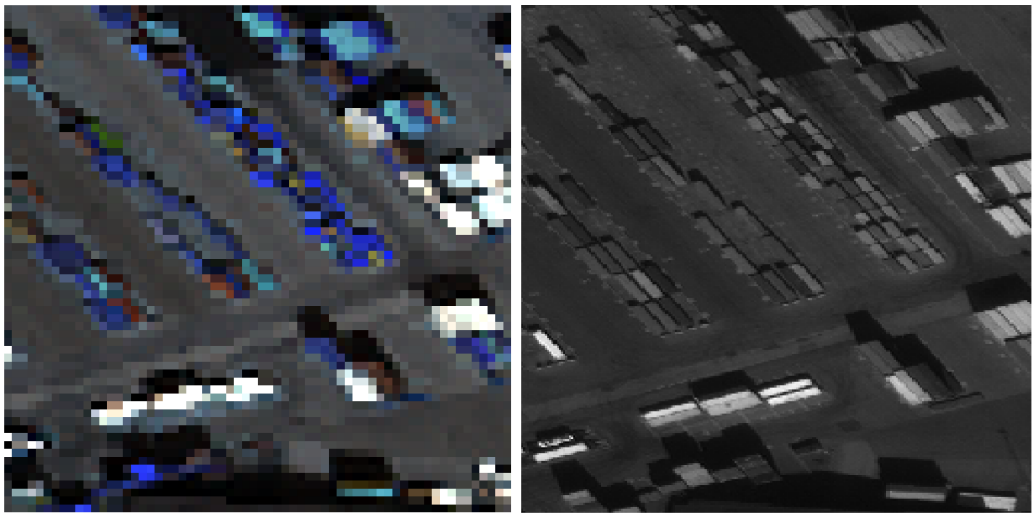


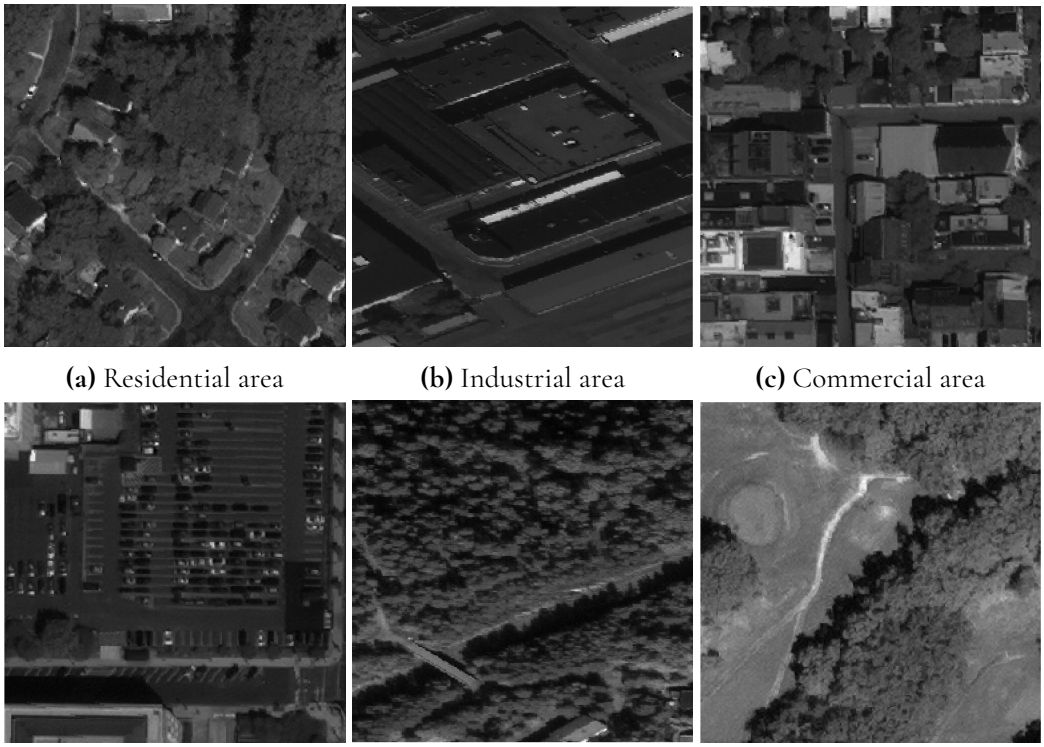
Figure 4.1: Corresponding colour and greyscale images of an industrial tile in Stockholm.

4.2 Annotation

When we decided on what city zones to include, we considered their relevance, abundance, and distinction. For example, classes that were discussed but discarded were cemeteries and pitches, because they were too few in numbers. Parks and forests often look similar, with abundant trees, but were still considered distinct enough to both be included. We ended up with the following six functional city zones:

- *Residential areas*: The most common area. These include both suburban areas with houses and villas, as well as urban areas with apartments.
- *Industrial areas*: Industrial areas include both naval industry and general factory areas.

- *Commercial areas*: Commercial areas are areas with shops, businesses, offices and such, often located centrally.
- *Parking areas*: Parking lots, does not include single parking spots alongside roads.
- *Forests*: Natural forests and general vegetation.
- *Parks*: Mostly large grass areas, but can also contain forest-like areas.



(a) Residential area **(b)** Industrial area **(c)** Commercial area

(d) Parking area **(e)** Forest **(f)** Park

Figure 4.2: One grey tile of each city zone.

To enable supervised machine learning on our dataset, the images had to be labelled. A label corresponds to a city zone to be classified. However, the tiled images are entirely unlabelled, and with 46000 images, we considered it infeasible to manually label each image. Therefore, we decided to design a way to automatically annotate our dataset, which compared the coordinates of each tiles to the OSM database that holds the important geospatial information of our classes, and labelled each image based on how much of the OSM data that overlapped with our tiles.

4.2.1 Extraction of Shapes

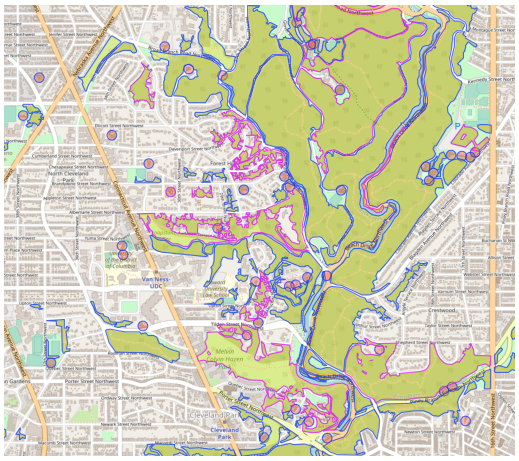
The first step in the process of automatic annotation was finding data related to where classes are, e.g. the whereabouts of each city zone in the image. We acquired spatial data from OSM, corresponding to manually annotated areas that equals our classes. As the OSM database is primarily manually built, it is not perfect. Open Street Map is in itself an interactive map, but by using a tool called Overpass, it is possible to extract the raw data of the elements.

(OpenStreetMap 2020b) Overpass works as a database, where a query returns a GeoJSON object, and contains spatial data of all elements returned by the query, which correspond to our *shapefiles* that are used for the annotation.

An example query is illustrated below:

```
way["natural"="wood"](38.88048,-77.11423,38.97260,-76.99918);
relation["natural"="wood"](38.88048,-77.11423,38.97260,-76.99918);
```

The above query returns ways and relations with the *natural* tag, tagged as *wood*, within the given coordinates. A *Way* equals a polygon, and a *relation* is a multi–polygon. Figure 4.3a, illustrates the visual results of the previously presented query. An extract of the JSON data for such a query can be seen in Figure 4.3b. This query refers to *nodes* in a way. The nodes are the points of which the polygon is composed and hold the location data, and they are also included in the resulting GeoJSON file, as seen in Figure 4.3c.



(a) Visual result, where woods are marked green with a pink outline

```
41969   {
41970     "type": "way",
41971     "id": 391291453,
41972     "nodes": [
41973       3944978029,
41974       3944978030,
41975       3944978031,
41976       3944978032,
41977       3944978033,
41978       3944978029
41979     ],
41980     "tags": {
41981       "natural": "wood"
41982     }
41983   },
41984   {
```

(b) An element in GeoJSON

```
178769  {
178770    "type": "node",
178771    "id": 811422898,
178772    "lat": 38.9266064,
178773    "lon": -77.0607619
178774  },
178775  {
178776    "type": "node",
178777    "id": 811422934,
178778    "lat": 38.9260074,
178779    "lon": -77.0601193
178780  },
```

(c) A node in GeoJSON

Figure 4.3: Results from an Overpass query for woods in Washington

The Overpass queries for most classes were straightforward – for example, *Parks* was queried by `'leisure'='park'`. However, due to different methods of annotation in different locations, the class *Forests* was composed of both areas marked as `'landuse'='forest'` and `'natural'='wood'`.

4.2.2 Tagging

At this point both tiles and the queried class areas were ready, and we were almost ready to automatically annotate the dataset. First, our queried data and the tiles had to be converted to a format where they could be compared with one another. For this, we used two Python libraries called Fiona and Shapely. (Gillies 2020a)(Gillies 2020b) With Fiona, we were able to convert each tile to a shape (.shp extension) from its corners, and each element from its nodes. This was also useful for error–checking, as we were able to display both elements and tiles in QGIS to visually see overlapping shapes. Shapely is used for set-theoretic analysis and manipulation of planar features. In our case, it was used to determine whether different shapes in shapefiles were disjoint or not.

By iterating through all the tiles, we compared each tile to all elements to see whether they intersected – if they did, that tile contained the class of the element. However, this was not enough to tag the tile, as only the label with the largest intersection should be chosen. If multiple elements of the same class intersect with the same tile, their intersected areas were added together. Furthermore, the tile was only tagged if the intersection would exceed 50% of the area of the tile. This to avoid tiles where only a small part of the tile was covered by a class, but at the cost of excluding many tiles since the threshold means that many tiles did not belong to any class at all, and were excluded from becoming part of the dataset.

4.2.3 Results From Annotation

For our annotation process to produce desirable results, there had to be reliable tags submitted to Open Street Map. In two of our four cities, Washington and Stockholm, the tags were abundant. Tiles from these cities were plentiful in most of our categories, and were of high quality. However, in Rio de Janeiro and Tripoli, the tags in Open Street Map were very few, and provided little to no value. In the classes where there were tags, there already were enough tags from Stockholm or Washington, whereas in the classes where we were lacking, Rio and Tripoli provided next to nothing. This was especially clear when we changed the threshold to 50%, as it dramatically reduced the number of tiles.

As a result, we decided not to use tiles from Rio de Janeiro or Tripoli. This, combined with the fact that a lot of tiles were marked as *Untagged*, due to the threshold, means that our original 46012 tiles had been reduced to 8741. Some classes, for example residential areas and forests, were abundant, whereas other classes, especially parking areas, were alarmingly few.

The final class count, after annotation, was as follows:

- *Residential areas*: 3113
- *Industrial areas*: 284
- *Commercial areas*: 403
- *Parking areas*: 149
- *Forests*: 2879
- *Parks*: 1913

4.3 Preprocessing

While the data was annotated, it was not yet ready to be fed to any machine learning algorithms. First and foremost, all the image data had to have the same dimensions. Currently, the greyscale images are 256x256 and the colour images are 64x64. By upscaling each colour pixel to a 4x4 copy of itself, we convert the 64x64 image to a 256x256 version. The image contains the same information, and looks the same when zoomed out, see Figure 4.4 for an illustration. This had the downside that the colour images now were 16 times bigger, and took up more space and made the training process slower. However, when upscaled the images had the same dimensions as the greyscale image, so that they can be combined to one tensor.

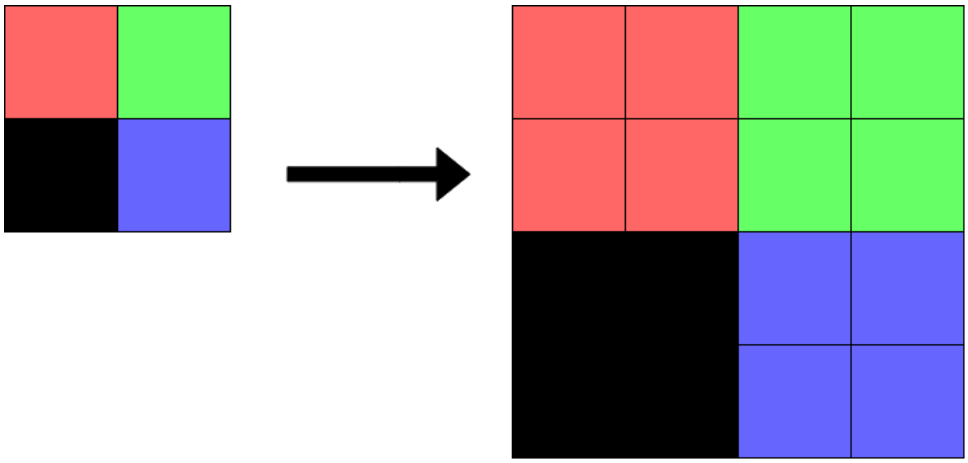


Figure 4.4: Upscaling 4 pixels to 16 via a 2x2 copy.

Secondly, the data had a serious disparity in the amount of images per class. Three classes had images in the thousands, while the other three classes were in the hundreds. Of course, it would be possible to train machine learning models with this data, but the resulting model would likely perform poorly. (Barnan Das 2013) There would be a heavy bias towards the numerous classes. A golden, informal rule is that if a class consists of less than 10% of the total data, then the model will likely not perform as well as it could. At this time, our parks constituted for a mere 1.6% of the data, which implied that something had to be done, and as a solution, we chose image augmentation. This is a method that copies an image, alters (augments) it in some way, and adds the augmented image to the dataset. A common augmentation is to simply flip the image horizontally, which preserves any internal structure and shape of the image, while still providing a new, fresh image to the network. The parking areas were the fewest, and to reach the 10% mark we decided to augment these images three times, increasing the image count from 149 to 1192. The three chosen augmentations were a horizontal flip, a vertical flip, and a perspective transform. The perspective transform zooms the image based on random distances from its corners, with the random numbers chosen from a normal distribution. (Jung 2020) The industrial and commercial areas were slightly more numerous than the parking areas, and only required two augmentations, and those two were the horizontal and vertical flips. After augmentation, industrial areas were the fewest, contributing to 9% of our data. Our dataset now consisted of 11845 images, and the final class count was as follows:

- *Residential areas*: 3113
- *Industrial areas*: 1136
- *Commercial areas*: 1612
- *Parking areas*: 1192
- *Forests*: 2879
- *Parks*: 1913

Thirdly, the .tif format is not particularly fast when reading large amounts of data. The spectral bands in the .tif images are essentially matrices of data, and can therefore be handled by the NumPy library. NumPy is a Python library widely used in scientific computing, and offers efficient multi–dimensional containers. (NumPy 2020) We read all images and created a NumPy object for each image, that contained both the greyscale and the colour bands in the same object. Each combined object was then saved to disk as .npy files, which allowed for much faster reading than the original .tif–files. The resulting .npy–files had a shape of [256,256,5]. The combined size of our dataset was 7.76GB ($256 * 256 * 5 * 2 * 11845$ bytes).

Furthermore, we also had to normalise the data. Normalising means rescaling the values to fit some interval, and in our case this interval was [0,1]. Initially, our pixel values were encoded with 11 bits per channel, taking a value between [0,2047]. The original distribution of a random tile, before any rescaling, can be seen in Figure 4.5. The distribution was heavily skewed towards the lower values, resulting in an average of 0.2 after normalising, but for the learning process to be as efficient as possible, the values should be evenly spread out over the interval [0,1]. Advised by an expert from our university, we combated this by taking the square root of each value to shift the distribution.

To conclude, all values were divided by 2047 to fit the new interval – then, we took the square root of each value. The result was a better spread, with an average of around 0.45, with internal features preserved.

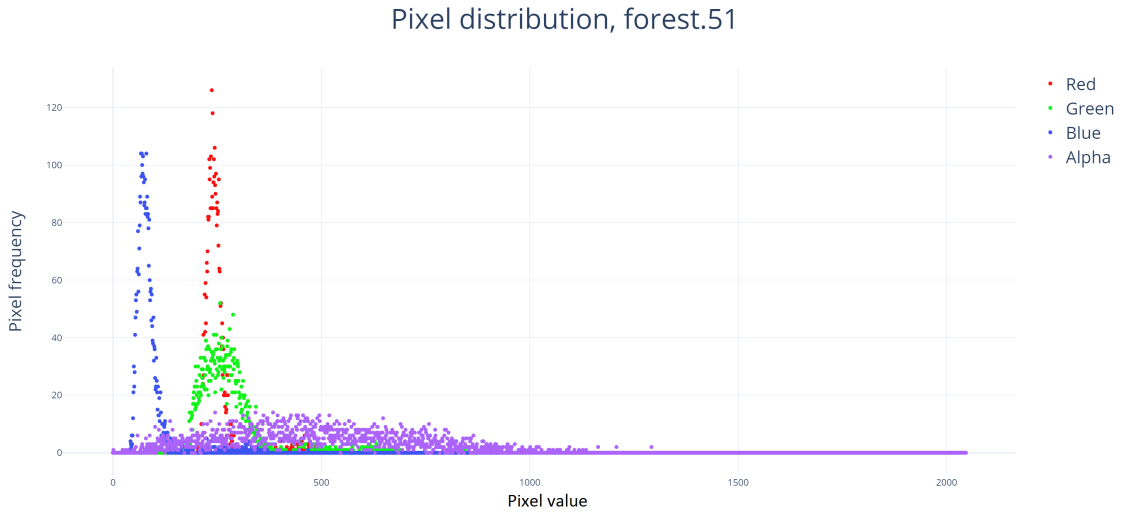


Figure 4.5: Initial pixel distribution, RGBA, random forest tile

Chapter 5
Implementation

This chapter describes more practical parts, with the data being ready. Our workstation is presented and how the practicality around it worked. In addition, our training process is described, followed by how we logged and evaluated our results.

5.1 Workstation

Both the parameter count of chosen architectures and the size of our data imply that the problem to be solved will be computationally heavy. Any attempts that were made to perform these calculations on our local machines was proven unsuccessful, due to insufficient GPU and RAM memory resources. Instead we decided to outsource our model training to a cloud platform.

We chose to use *Google Cloud Platform*, because the hardware we required was available, and because we were able to get access to that hardware without spending any money. (Google 2020) Google Cloud Platform allowed us to build a virtual machine, and upload our data and code to this machine. We had a Linux virtual machine where most of the necessary packages were preinstalled. To avoid the use of the terminal and to get a user-friendly workstation, we utilised a Jupyter Notebook server, started remotely on the virtual machine through a SSH connection. This enabled us to enter the notebook through the IP address of the virtual machine, yielding an interface where we were able to write and run code and download or upload data. Two pieces of hardware were crucial in successfully running the required computations. To efficiently feed all our data into the network, we wanted to simultaneously keep all of the data in RAM. When we normalised our data, the division and square root transformed each value to float32, which effectively doubled its size. This meant that we required at least 7.76 * 2 GB RAM, only for the data. We therefore chose to utilise 30GB of RAM for our virtual machine. The other important choice was what graphics processing unit (GPU) to use. We decided to use a *Nvidia Tesla P100*, which is an incredibly fast GPU. It also had enough RAM to hold all trainable parameters, as well as it enabled us to use an

adequate batch size. (NVidia 2016)

5.2 Training

To recall, we now had nearly 12000 images with the shape [256, 256, 5], with the last dimension, the 5, corresponding to the number of channels, or bands, in the data. The first band contained the greyscale, the following three the colour, RGB, and the last one the infra–red image. We did all of the preprocessing of our data on our local machine, and then transferred the .npy files to our virtual machine. When on the virtual machine, we constructed a new .npy file, that contained all of the images. The shape of this file was therefore [256, 256, 5, 11845]. This process was repeated each time we altered something in the data, but the transfer to the virtual machine was the most time consuming part. Furthermore, we had two results to retrieve, our baseline and our main results. Our baseline consisted of models trained with only colour, i.e. the three middle bands of our data, 1-4, together with transfer learning. Our main results was instead based on the full width of our data, without utilising previously trained weights. To answer our problem statement we evaluated models based on these two approaches and compared the yielded performance. We trained a model based on both approaches for each architecture that we wanted to evaluate. Our model evaluation was based on eight architectures, which equals eight models to be trained for the baseline and the main results. This, in turn, equals 16 models with only one set of parameters. For each parameter added, the number of models that had to be evaluated doubled.

We trained models based on one architecture at a time, experimenting with different hyperparameters and optimisers. Because of the large number of models that we had to train, we decided to vary the learning rate more freely and to focus on two different batch sizes. We also decided to only use one form of learning rate modification, namely *reducing learning rate on plateau*. This method monitors the learning rate and if there is no improvement the learning rate will be reduced by a factor. (Arnold 2019) Details, such as what is monitored, when to reduce and how much, can of course be specified upon use. We referred to the original publication of each architecture when we decided on what parameters to use, as each publication contains information of how the corresponding architecture was trained on ImageNet.

5.3 Evaluation

We added all relevant information of each trained model to a table. Partly, we saved the performance of the model, i.e. the loss score and the accuracy. We also saved the training setup, i.e. the hyperparameters and the choice of optimiser. During all of our training, we used the common training, validation and test splits from 3.4.3, namely 60/20/20.

For each model we tried an initial set of parameters based on the original paper for each architecture and experimented with a few different optimisers, until we achieved a model that learned from the data. Thereafter, we tried to optimise with a set of parameters that we tried to keep as fixed as possible. However, the parameters that worked well varied a lot with certain architectures, which led us to abandon our initial approach. Instead we chose the parameters more specific for what was working for each architecture.

Chapter 6
Results

This chapter describes the results we acquired from our experiments. First, we analyse the baseline and the 5–band model individually, presenting training history and a comparison of the architectures. Then, we analyse the top performing architecture more carefully, showing confusion matrices of predictions. Finally, we also present a practical classification, by reconstructing an original image base on predictions without labels.

6.1 Baseline

We trained a baseline for each model, with a finetuned set of parameters each. The individual results of this training can be seen in Figure 6.1, where ResNet50 achieves the highest result of 86.3% test accuracy. It is also clear, that out of the chosen architectures, VGG19 performs the worst, even worse than its simpler version, VGG16. The full set of parameters used can be seen in appendix A.

In Figure 6.2, the training history for the baseline is shown. Certain models reach a high accuracy already after a single epoch, whereas others are slower. Keep in mind, the baseline was initialised with ImageNet weights, which means much of the training was already done, just not adapted to satellite images. Many of the features from ImageNet can be used, however, which leads to a lower learning time. A key thing to notice here is that all models keep improving with more epochs. This indicates that a somewhat higher learning rate could be appropriate at that time of training, but a low learning rate was needed at earlier epochs.

6.2 5–Band Model

Similar to the approach for the baselines, we trained a number of models for each architecture, with finetuned parameters. The best model for each architecture was chosen as the representative, which resulted in Figure 6.3. Here we see that instead of ResNet50, it is

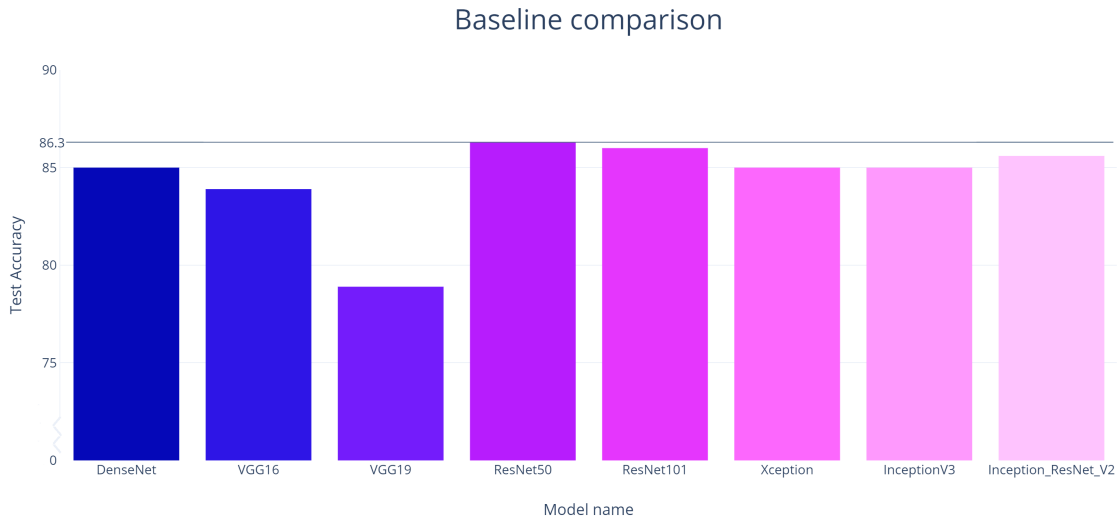


Figure 6.1: Baseline comparison of models

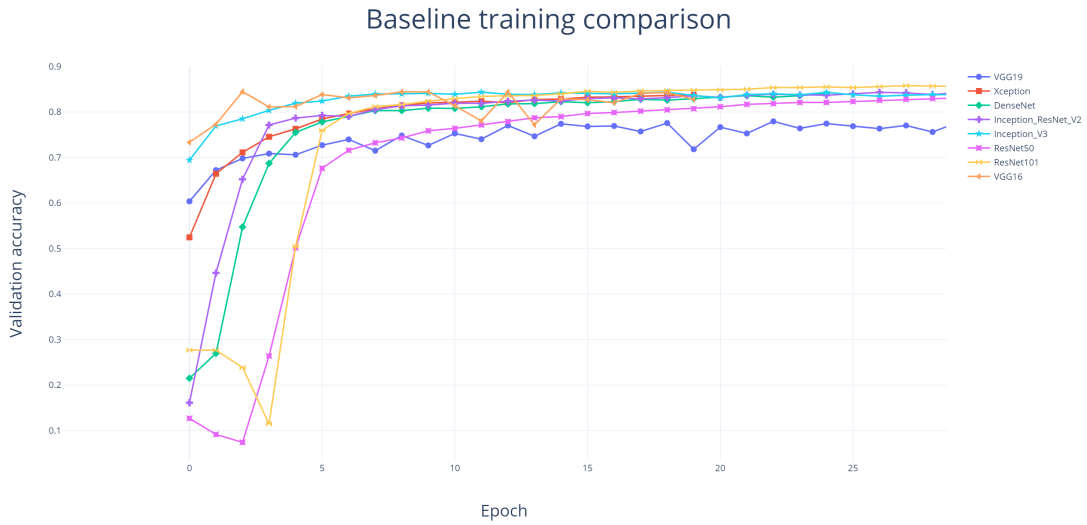


Figure 6.2: Baseline training history

now DenseNet that outperforms the rest, but VGG19 is still at the bottom. Unfortunately, we were unable to train VGG19, no matter which parameters we chose. The test accuracy of VGG19 was around 25%, which corresponds to the number of residential areas in the dataset, and is exactly what VGG19 predicted. Furthermore, all models improved when training with the extra bands, except for VGG. VGG16 also got worse, and although it was not to the same extent as with VGG19, test accuracy dropped from 83.9% to 83.5%.

6.3 Top Performing DenseNet Model

Because this DenseNet model performed the best, and was trained with all bands, we decided to evaluate this model further. The set of parameters and settings used for this model was:

- *Learning rate*: $6e^{-}6$

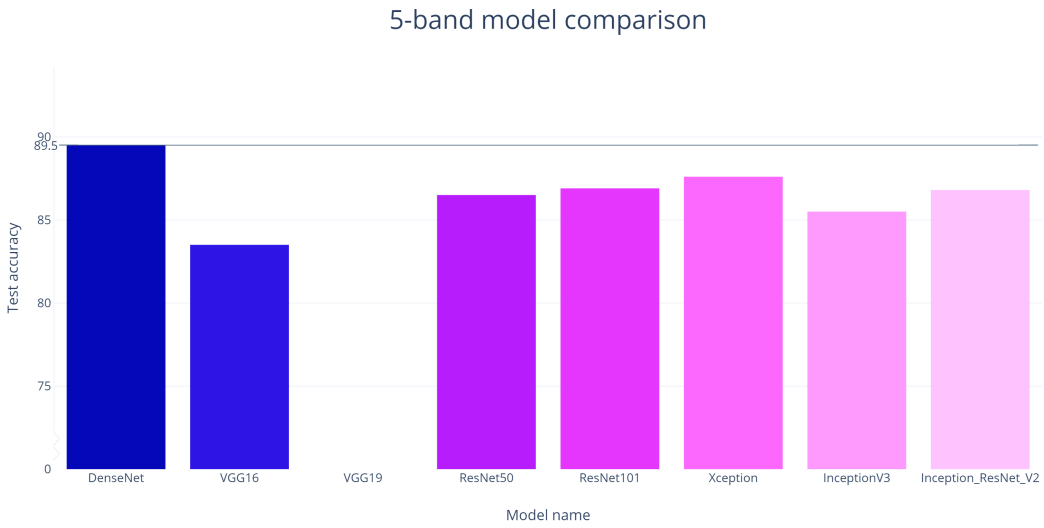


Figure 6.3: 5-band model comparison

- *Batch size*: 32
- *Optimiser*: RMSProp
- *Learning rate modifiers*: Reduce_lr_on_plateau
- *Dropout*: None

It reached a loss of 0.32 and 89.5% test accuracy after 40 epochs. In fact, a virtually identical accuracy was reached already after 20 epochs as seen in Figure 6.4, but the validation loss would drop until epoch 40. In other words, the predictions for the last 20 epochs were similar, but the model would be more convinced of its predictions. However, as in the case of the training history for the baseline models, the model keeps improving with time, albeit slowly. Again, this indicates the learning rate is too low. In the baseline case we started with relatively low learning rates, as they were already pretrained, but with the 5–band model we did not have to worry about ruining any weights from the start. We initially tried a higher learning rate, but again and again found that lower initial learning rates produced better results. This is how we reached a learning rate as low as $6e^{-}6$.

In Figure 6.5, we can see the predictions for each class by the DenseNet baseline. This matrix is based on the test set, of around 2000 samples. A confusion matrix shows how many of each class was correct, and what the incorrect classifications were predicted as. The diagonal shows the correct classifications, and a well trained model will therefore have a strong diagonal. For example, in the top left corner, we see the number 0.76. This means 76% of images containing commercial areas were correctly classified as commercial images. To the right, 0.065, means that 6.5% of commercial areas were incorrectly classified as industrial areas. Down from 76%, we find 0.069, meaning 6.9% of industrial areas were incorrectly classified as commercial areas. This system allows us to see which classes are similar to the model, and which are easier and harder to predict. Note that a row always sums up to 1, but that the same is not true for the columns.

As we compare the DenseNet baseline to the 5–band model, we can see improvement in all classes, save forests, as shown in Figure 6.6. In both models, we expected to see a large

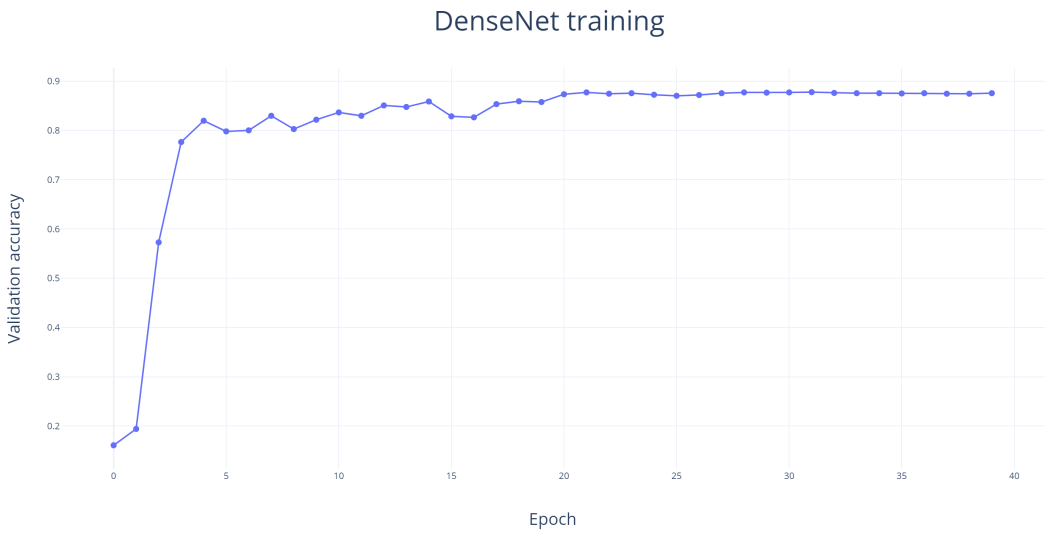


Figure 6.4: DenseNet best model training history

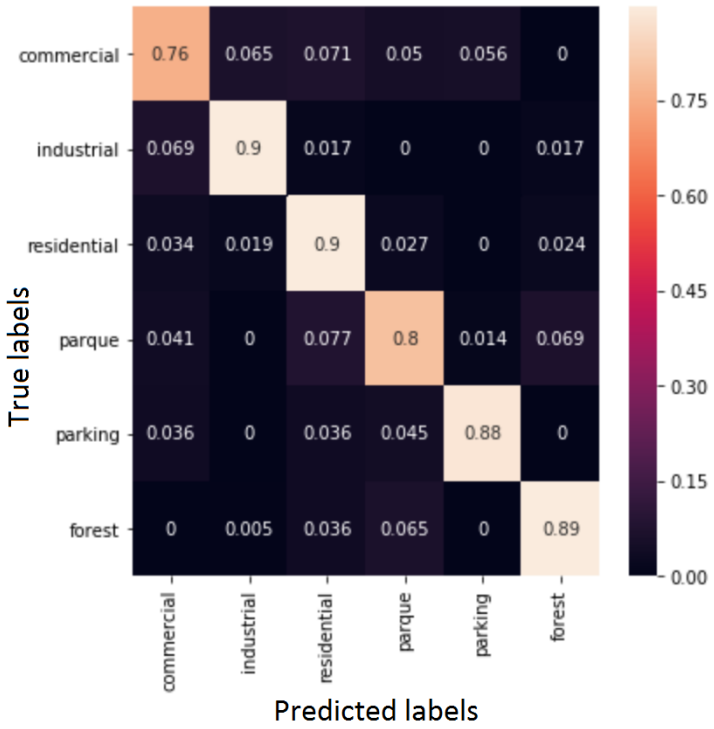


Figure 6.5: DenseNet baseline confusion matrix

overlap in the classification of parks and forests, as they both contain a lot of vegetation. This prediction holds true, in that the most common incorrect classification of both those classes were each other, in both the baseline and the 5–band model. There is also a high rate of error when classifying the commercial areas – to all classes except forests. This is probably due to the varying structure of commercial areas. For example, certain urban residential areas are likely to be indistinguishable from commercial areas, and many commercial areas contain large parking lots. An example of the latter is shown in Figure 6.7. This image shows

a commercial area, which was incorrectly classified as a parking area by the best DenseNet model. However, we were impressed by the models being able to classify around 90% of images correctly, considering the variance in the images and the similarity between the classes.

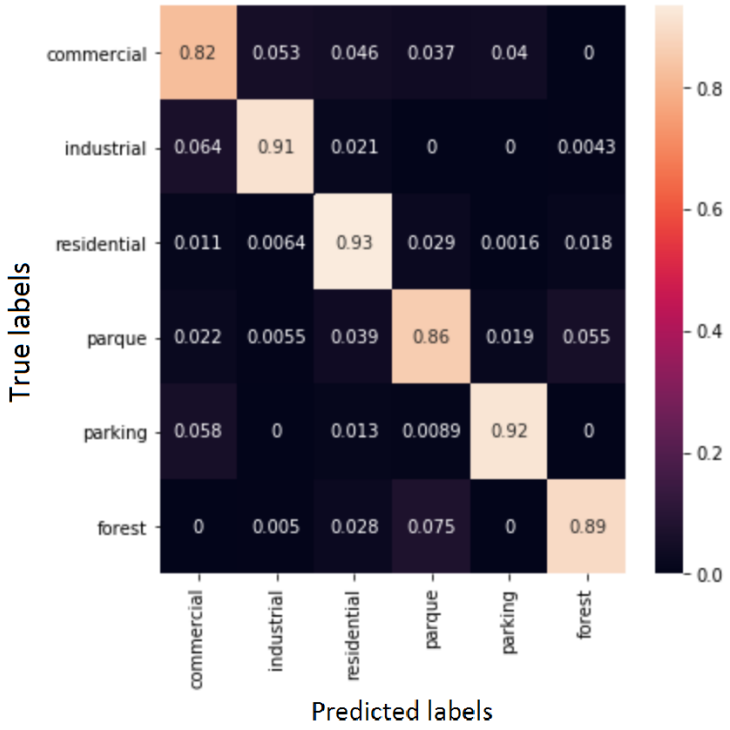


Figure 6.6: DenseNet 5-band confusion matrix

6.4 Colour Reconstruction

Using the best DenseNet model, we let it assign a class to each of the tiles from the full dataset before tagging. Note that this set includes thousands of tiles that are untagged, i.e. that do not belong to any class according to Open Street Map. Each class was assigned a colour, and the result was a reconstruction of all tiles, with an overlay of the colour belonging to each class. A small part of this reconstruction can be seen in Figure 6.8. This area in particular shows a suburban residential area, with a forest in the northeastern corner. A park can be seen in the midst of the residential region, and two industrial tiles can be seen to the right. These industrial tiles show a couple of football pitches, but *pitch* was not included as a class when training the models.

Another part of the full reconstruction can be seen in Figure 6.9. This image shows Gamla Stan, in central Stockholm, as a mostly commercial area, with industrial elements towards the docks and the roads. A curious forest appears in the northwestern corner, but there are always errors, especially when such a large part of the image is just water. Furthermore, the segmentation of tiles proves to be a rather inexact method. Many tiles belong to two classes, and it would be much more informative to mark important areas within the images, rather than marking a single tile as either one class or another.

Figure 6.7: Commercial area predicted as parking area by DenseNet

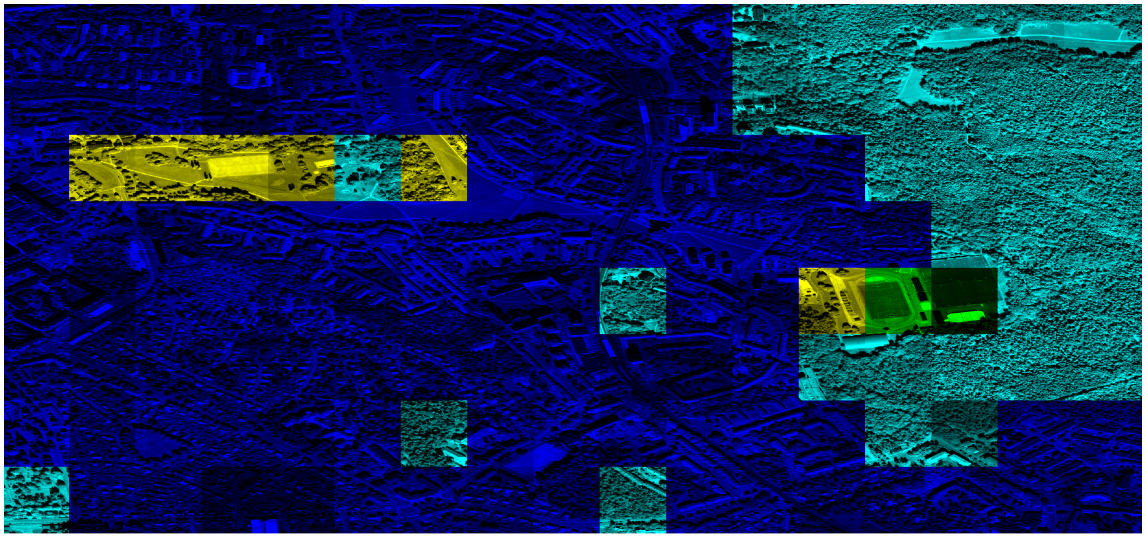


Figure 6.8: Reconstruction of a suburban area. *Blue* = Residential, *Yellow* = Park, *Teal* = Forest, *Green* = Industrial

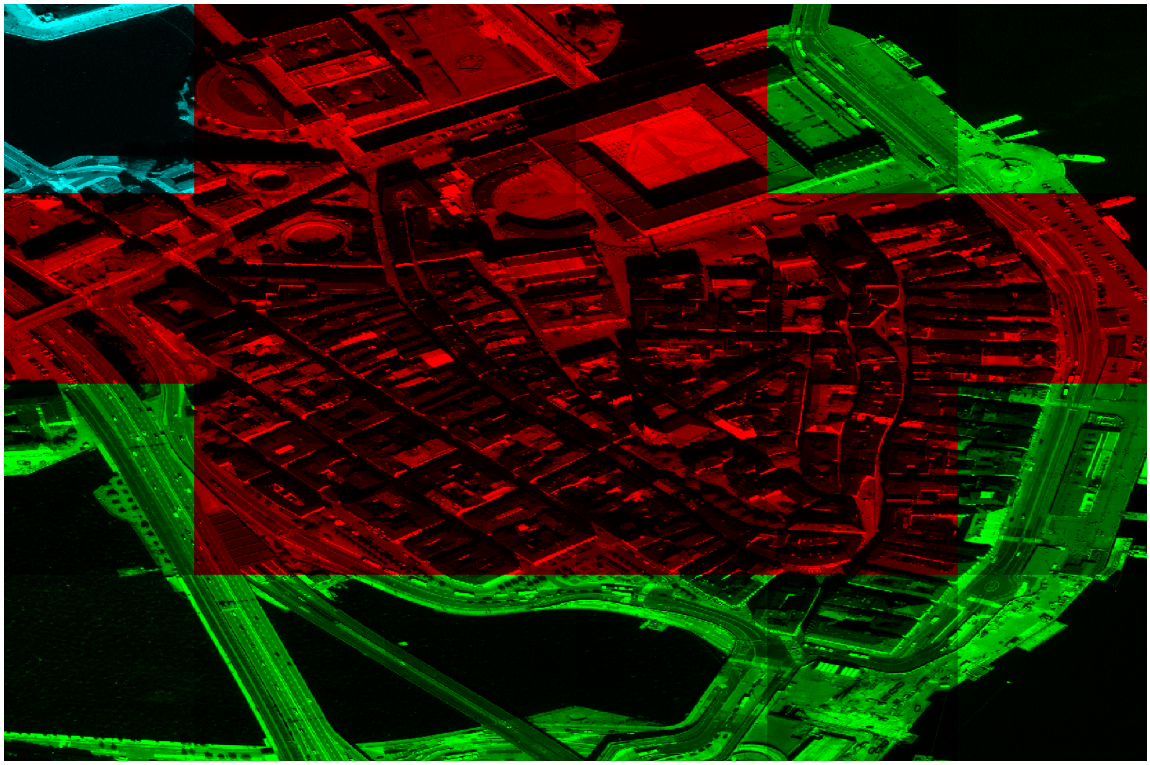


Figure 6.9: Reconstruction of Gamla Stan. *Red* = Commercial, *Teal* = Forest, *Green* = Industrial

Chapter 7

Discussion

In this chapter, we discuss our results and try to answer our problem statement. Initially, we describe, analyse, and draw conclusions from our results, both considering the baseline, the 5–band model and our reconstruction. We also describe what issues we encountered on the way, and how we circumvented these. Finally, we discuss what could be done in the future, building upon this thesis, both in terms of further studies and in practical applications.

7.1 Result analysis

7.1.1 Baseline

Analysing our baseline, we expected the more advanced, newer architectures to outperform the older ones. However, our problem does not seem to be too complicated, so any of the architectures ought to perform well. From the results we can conclude that the VGG models were the ones performing the worst, while still achieving around 80% accuracy, which agrees with our theory. However, the top performing models in the baseline were not the most modern architectures, which was a little bit surprising. Furthermore, for both ResNet and VGG, the models with fewer layers (VGG16 and ResNet50) performed better than their deeper counterparts (VGG19 and ResNet101). This could be due to over-capacity, which means that if the architecture is deeper than the problem requires, results will deteriorate because of vanishing gradients. This implies our problem is relatively simple, which further explains why the more modern architectures are not necessarily better than the older ones. Furthermore, choosing the perfect set of parameters is time–consuming, and it may be that we were "lucky" when we tried parameters for ResNet50, and therefore finding better results than we did for ResNet101. The implications of this margin of error is that our results have a larger margin of error, which makes it more difficult to draw definitive conclusions.

7.1.2 5–Band Model

Training the 5–band model yielded DenseNet as the top performer, at 89.5%, with Xception in second at 86.7%. These were the two most modern architectures and performed the best, all according to theory. The VGG nets performed the worst, as expected, but what was surprising was that VGG19 was untrainable. However, it would probably be trainable had we spent more time configuring the parameters. It must have been very sensitive, because no improvements were made in either direction upon changing learning rates, adding or removing dropout layers, or changing the batch size. Learning rates ranging from $1e^{-2}$ all the way to $1e^{-8}$ were tried, but to no avail. DenseNet, on the other hand, gave excellent results without desperately searching. This of course has the side effect that we may very well have been able to find a set of parameters that would outperform our best model, but with sufficiently good results we decided it was enough.

7.1.3 Comparison

Recall our problem statement, specifically question Q2:

- Are there gains to be made from applying transfer learning to each architecture but having to use less of the available satellite image data?

In order to adequately answer this question, we compared the baseline to the 5-band model, and specifically took two things in consideration:

- What difference do the two extra bands, grey and Alpha, make?

Adding the two extra bands adds complexity to the problem, especially the greyscale, which in itself has 16 times more unique information than any other band. This could explain why the more complex architectures performed better on the 5–band model than the baseline. Moreover, it seems that the two extra bands contribute positively to the final model. We could see improvement across all models but VGG, and all classes but *forests* (which was static at 89%), and because pretrained weights virtually cannot make training worse, the improvement in result must be attributed to the two extra bands. Which band then helps the most, grey or Alpha? This is somewhat difficult to answer, as we have not done any more thorough investigation into individual layers, and what the models *actually* look at. For example, we expect large weights on neurons that see green colour when predicting forests, and large weights on neurons that detect densely packed straight lines when predicting parking lots. Furthermore, to truly see the effect of individual bands, we should have investigated the performance of only the greyscale band, and also investigated a model with RGB + A, and perhaps even a model with the greyscale band and the Alpha band. These models, along with our results, could provide an adequate base to draw conclusions on which bands are the most useful. In our confusion matrices, we could see that the forests were almost exclusively misclassified as parks, which was expected due to the similarity between classes. However, there was no improvement from the baseline to the 5-band model regarding forests – the predictions were almost identical. Improvements were instead seen in classes that tend to misclassify as residential areas - namely commercial areas and parks. These dropped by several percentage points each when adding the two extra bands. This is likely due to the fact

that the difference between commercial and residential areas is much more exact and on a detailed level, compared to forests, where its features are much broader.

- What difference does having / not having pretrained weights make?

Unfortunately, it is difficult to say exactly how big the impact of transfer learning has been. We can monitor training times, and perhaps conclude that the required number of epochs is lower if we use transfer learning, but that does not say anything about whether the end model has improved as a result or not. In all our cases except VGG, the 5-band model outperformed the baseline, in some cases significantly. As seen by the training history, all models required quite many epochs to reach the final result. However, they may have led the training into the right track, saving valuable time. This may have been the case with VGG19, which was untrainable without transfer learning. However, our results are somewhat distorted by the fact that we add additional bands when examining whether transfer learning contributes or not. With this in mind, we conclude that it cannot be said that transfer learning improves the final model. Note that this does not necessarily mean that transfer times do not improve the final model – just that we cannot draw the definitive conclusion that they do. Were these tests done independently, it would be easier to draw reliable conclusions.

Usually, the initial weights will be entirely randomised, and such was the case when we were training without transfer learning as well. This means that it is very unlikely that any transfer learning contributes negatively to the end result – the question is rather whether it contributes positively or even at all. With this is is mind, and by receiving significant improvements on multiple models when using several bands, we can conclude that adding the extra bands has a positive contribution in itself.

7.1.4 Reconstruction

As many of the original tiles are untagged, any evaluation of them has to be done manually. Even if the images in the test set always were unseen to the models, the images in the set would always have 50% belonging to a class. The untagged classification is therefore much more difficult, and will assign a class even if the image in itself is nonsense. For example, practically every image that was covered in water (i.e. almost completely black) was predicted as an industrial area. What was especially interesting to see, was predictions of tiles that clearly belonged to a class that was not included amongst the original classes. The most obvious example is seen in Figure 6.8, where two industrial tiles show a couple of football pitches. Why they have been predicted as industrial areas is unknown, but it still shows that there is probably room for more classes and more complexity in the problem. Furthermore, this classification does not take any neighbouring tiles into consideration when assigning colours. For example, in figure 6.8, there are stray tiles of forests in the midst of the larger residential region. While those predictions of forest tiles may be reasonable, perhaps the tile should be a residential tile if all surrounding tiles are residential tiles. This way, it would be possible to create more accurate, larger regions with just one class.

7.2 Issues

Naturally, we had to overcome several issues whilst conducting our work, some of which significantly affected our approach, of which the most important are discussed below.

Annotation Threshold

Throughout the majority of our work process, our annotation was done without a threshold. This meant that an overlap of a single pixel between a tile and a shapefile was enough to classify that tile. This led to many labelled images, and a larger dataset. However, as a consequence of the non-existing threshold, the quality of our dataset was really bad. Many of the images did not represent the city zone that it was labelled as, which also led to our neural networks performing really poorly. We soon realised that our negligence in adding a threshold early in our process most likely was to blame for this, and therefore we remade our entire dataset from the start with a added 50% threshold. The number of images that qualified for labelling was significantly less, but the quality was infinitely better. Time could have been spent to find a more fitting threshold, maximising both quality and quantity, but the initial threshold of 50% already gave us an incredible increase in performance in comparison to not using one, so we decided to be content and move on.

Keras Generator

Initially, we wanted to use a generator to feed data to the network. Keras accepts tif-files as input to its inherent generator but for some reason it only accepted 8-bit resolution. This meant that our 16-bit resolution images got reduced to 8-bit. Unfortunately, for some reason, the 8 most significant bits got removed, instead of the 8 least significant bits, essentially destroying our data. To combat this, we implemented our own generator, integrable with the Keras framework. In this way, we could handle our data in whatever way we wanted, dividing each data sample with 2047, which was the largest value in our images (so our actual pixel-resolution was 11 bits). Even though this solved our initial problem, the use of a generator essentially made it so that the CPU became the bottleneck instead of the GPU, which is ridiculous since the GPU is the expensive part. Since a generator creates a batch through reading each sample from disc, this simply took much longer time than actually training our neural network on that batch. This finally resulted in us acquiring enough RAM in our virtual machine so that we could store all of our data there as a large npy file, virtually rendering our generator moot, and allowing us to fully utilise our GPU.

Bad Class Distribution

The distribution of our classes after annotating with a threshold of 50% was really problematic. We initially tried to combat this by forcing each batch to contain an equal amount of each class through reprogramming of our generator. However, this made the CPU bottleneck even larger and the utilisation of our GPU worse. Instead, we chose to augment our images (as we did not even use a generator at all in the end) to shift the distribution. This process was done after the tiling and annotation of our images. This added a lot more data, requiring even more RAM to maintain all of the data in memory at one time, but it was worth it. The

result was better performing models as well as lower training times for the models, when all of the data was in RAM, enabling our GPU to maximize its potentialt

7.3 Future Work

Related to the work that we have done, there are plenty of areas that could be explored further. Firstly, additional data should be desired to get more accurate models, and more specifically, a geographically spread out set of cities could be added to add diversification on the data, as the cities might look very differently based on location and culture. Secondly, more classes could be added to increase complexity, such as roads, pavements, public structures/areas, to further enable the mapping of a city. Thirdly, the classification could be done on a per-pixel basis instead of our large tiles, increasing the level of detail that the model can achieve. Also, a more detailed analysis of our annotation process could be necessary to judge how accurate it actually is, and if more time were available, a large scale manual annotation process could take place. Finally, applications that use neural networks could be developed to help with the map completeness problem.

Additional Data

More data is nearly always equivalent to a better, more generalising model. This is because more data almost always represents the problem more accurately, covering more edge cases, which means that the model could approximate a more general function, resulting in increased performance before overfitting. In addition, adding more data could result in a more diverse model, because the data added could be of cities with vastly varying architecture and surrounding biome. Adding more data could also simplify the addition of more specific classes. It would be interesting to include roads, pavements, churches and other public entities such as schools, to create a more specific mapping of the city. Also, it could be worth exploring adding features, in this case point of interest information, that is also available in OSM. Then, in cases where the network is unsure about the spatial data, features about an area could help with the classification.

Manual Annotation

An investigation on how accurate our annotation actually is and whether it would be worth the time to do manual annotation, given enough resources, could be worthwhile. Also, our tiles could be manually constructed, resulting in more accurate information in each tile for each class. The process is naturally very time-consuming and would require plenty of resources and some judgement whether it would be financially justifiable. Although the data that we have looked at after adding the 50% threshold on our tagging process seems to be quite good, there are cases where the annotation is poor, and manually remedying them could be a good idea. The effect that these cases might have on the performance might also be mitigated if vastly more data was available.

Pixel-wise Image Segmentation and Object Detection

Doing classification on each pixel is a more complicated process but enables masked object detection and a lot more accurate and smooth reconstruction as in Figure 6.8 and 6.9. To classify individual houses, for example, this process could be very good as the model could map the exact whereabouts of each individual house. If this is done for all the classes, on a more detailed level, say, roads, buildings, parks, etc., a very detailed depiction of a city can be done. This would probably also be a better approach for the map completeness issue. We also believe that, theoretically, a map could be generated, in a semi or fully automated process, from the satellite imagery, based on the results of pixel-wise image segmentation and object detection. Then the map could be validated manually through various tools, and the information, such as addresses, could be added manually.

ImageNet for Satellite Imagery

Given the fact that our results are not statistically conclusive, they imply that applying transfer learning on our satellite imagery is not worthwhile. Instead, training the model with all bands from scratch yielded better performance. As such, this indicated that there could be features in satellite imagery that is not represented in ImageNet, especially since we can not use all the available image data when using ImageNet weights. Therefore, since satellite imagery normally has more bands and that there are plenty of use-cases for satellite imagery and neural networks, we argue that it would be worthwhile to construct a standard dataset for satellite imagery, to enable the use of transfer learning with all the available data.

Map Completeness

There are several applications that could be developed to help with the map completeness issue. For one, a prediction from a network could be used to hint on differences between the satellite imagery (snap-shot of real world) and the current map version. A comparison with the prediction and the corresponding map area is made and if the comparison suggests a difference, a notification should be sent, highlighting this area of the map to be investigated to see whether the map actually differs with the satellite imagery, and in that case, update the map.

Chapter 8
Conclusion

In summary, we have evaluated neural network architectures on functional city zones. We have trained eight different architectures with and without applying ImageNet based transfer learning and analysed the effects. We have also used the best model to predict on the entire Stockholm image and colour each segment by class to visualise the predictions of the network and to illustrate the structure of Stockholm.

To conclude our results, it is indeed possible to do classification on satellite imagery with functional city zones as classes. There are gains to be made from including the greyscale and infra-red images but training from scratch. However, we probably should have also trained on the greyscale image only to be able to see how much it adds in comparison to colour. DenseNet was the architecture that performed the best when training from scratch, reaching a test accuracy of 89.5%. This is probably due to the fact that DenseNet maintains a lot of the abstract detail levels throughout the training process, which seems to be important when training on data with a lot of variation and detail. The networks that performed the worst, however, were the simple linear architectures of VGG. This also seems natural because they do not have any inception or residual layers, which both seem to be useful when doing computer vision, and necessary when the problem is complex enough.

Finally, from our discussion we conclude that the map completeness issue most certainly can be alleviated through the use of neural networks. There are many potential ways to address this and substantial further work is required to realise the full potential of helping with maintaining an up to date map. We hope that our model evaluation can help the potential future work with a starting point, hopefully reducing the necessary time to reach promising results.

Bibliography

(2016). Stanford Vision Lab. URL: http://www.image-net.org/ (visited on 01/30/2019).

Arnold, Taylor B. (2019). *Reduce learning rate when a metric has stopped improving*. URL: https://rdrr.io/cran/kerasR/man/ReduceLROnPlateau.html (visited on 03/01/2020).

Barnan Das Narayanan C. Krishnan, Diane J. Cook (2013). "Handling Class Overlap and Imbalance to Detect Prompt Situations in Smart Homes". In: *IEEE 13th International Conference on Data Mining Workshops, pp. 266-273*.

Brownlee, Jason (2018). *A Gentle Introduction to Tensors for Machine Learning with NumPy*. Machine Learning Mastery. URL: https://machinelearningmastery.com/introduction-to-tensors-for-machine-learning/ (visited on 12/13/2019).

Chollet, François (2016). "Xception: Deep Learning with Depthwise Separable Convolutions". In: arXiv: 1610.02357 [cs.CV].

— (2018). *Deep Learning with Python*. Shelter Island, NY 11964, 20 Baldwin Road: Manning Publications Co.

David DiBiase, John A. Dutton (2018). *The Nature of Geographic Information*. Penn State Department of Geography. URL: https://www.e-education.psu.edu/natureofgeoinfo (visited on 02/21/2020).

DigitalGlobe (2020). Digital Globe - free samples. URL: https://www.digitalglobe.com/samples?search=Imagery (visited on 02/24/2020).

EPSG (2012). *Using the EPSG Geodetic Parameter Dataset*. International Association of Oil and Gas Producers. URL: http://www.epsg.org/Portals/0/373-07-1.pdf (visited on 03/01/2020).

Gillies, Sean (2020a). *Fiona API*. URL: https://pypi.org/project/Fiona/ (visited on 02/29/2020).

— (2020b). *Shapely API*. URL: https://pypi.org/project/Shapely/ (visited on 02/29/2020).

Global provider of advanced space technology solutions (2020). Digital Globe. URL: https://www.digitalglobe.com/company/maxar-family (visited on 02/21/2020).

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. http://www.deeplearningbook.org. MIT Press.

Google (2020). *Google Cloud Platform - Overview*. URL: https://cloud.google.com/docs/overview (visited on 02/29/2020).

He, Kaiming et al. (2015). "Deep Residual Learning for Image Recognition". In: arXiv: 1512.03385 [cs.CV].

Huang, Gao et al. (2016). "Densely Connected Convolutional Networks". In: arXiv: 1608.06993 [cs.CV].

Jung, Alexander (2020). URL: https://imgaug.readthedocs.io/en/latest/source/api_augmenters_geometric.html#imgaug.augmenters.geometric.PerspectiveTransform (visited on 03/01/2020).

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012). "ImageNet Classification with Deep Convolutional Neural Networks". In: ed. by F. Pereira et al., pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

LEMAR, ANIKA SINGH (2015). "Zoning as Taxidermy: Neighborhood Conservation Districts and the Regulation of Aesthetics". In: *Indiana Law Journal* Vol. 90 : Iss. 4 , Article 4. (4). ISSN: 0019-6665. URL: https://www.repository.law.indiana.edu/ilj/vol90/iss4/4/.

Mitchell, Tom M. (1997). *Machine Learning*. McGraw-Hill Science/Engineering/Math.

Murphy, Kevin P. (2012). *Machine Learning: A Probabilistic Approach*. Cambridge, Massachusetts, London, England: The MIT Press.

NumPy (2020). NumPy. URL: https://numpy.org/ (visited on 03/01/2020).

NVidia (2016). *Data Sheet: Tesla P100*. URL: https://images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-PCIe-datasheet.pdf (visited on 02/29/2020).

OpenStreetMap (2020a). *OpenStreetMap Foundation- About*. OpenStreetMap. URL: https://www.openstreetmap.org/about (visited on 02/21/2020).

— (2020b). *Overpass API*. OpenStreetMap. URL: https://wiki.openstreetmap.org/wiki/Overpass_API (visited on 02/29/2020).

— (2020c). *Zoom Levels*. OpenStreetMap. URL: https://wiki.openstreetmap.org/wiki/Zoom_levels (visited on 02/24/2020).

QGIS (2020). *QGIS - About*. QGIS. URL: https://www.qgis.org/en/site/about/index.html (visited on 02/24/2020).

RasterIO (2016). *RasterIO - Introduction*. RasterIO. URL: https://rasterio.readthedocs.io/en/latest/intro.html (visited on 02/21/2020).

Russell, Stuart and Peter Norvig (2010). *Artificial Intelligence: A Modern Approach*. Upper Saddle River, New Jersey 07458.: Pearson Education, Inc.

Simonyan, Karen and Andrew Zisserman (2014). "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: arXiv: 1409.1556 [cs.CV].

Sk. Sazid Mahammad, R. Ramakrishnan (2009). *GeoTIFF – A standard image file format for GIS applications*. GeoSpatial World. URL: https://www.geospatialworld.net/article/geotiff-a-standard-image-file-format-for-gis-applications/ (visited on 02/21/2020).

Szegedy, Christian, Sergey Ioffe, et al. (2016). "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning". In: arXiv: 1602.07261 [cs.CV].

Szegedy, Christian, Vincent Vanhoucke, et al. (2015). "Rethinking the Inception Architecture for Computer Vision". In: arXiv: 1512.00567 [cs.CV].

Zhou, Victor (2019). *Machine Learning for Beginners: An Introduction to Neural Networks*. Towards Data Science. URL: https://towardsdatascience.com/machine-learning-

for-beginners-an-introduction-to-neural-networks-d49f22d238f9 (visited on 12/16/2019).

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2020-06-11

EXAMENSARBETE Neural Network Model Evaluation on Satellite Imagery Classification
STUDENTER Olof Nordengren, Kevin Johansson
HANDLEDARE Jacek Malec(LTH)
EXAMINATOR Jörn Janneck(LTH)

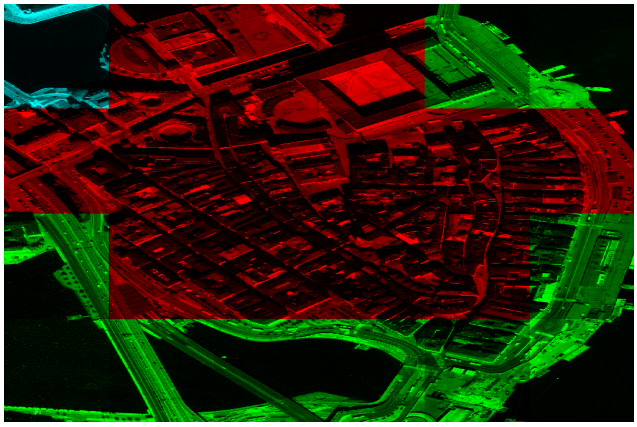
Klassificering av stadszoner med artificiell intelligens

POPULÄRVETENSKAPLIG SAMMANFATTNING **Olof Nordengren, Kevin Johansson**

Artificiell intelligens kan appliceras i många områden - inte minst inom bildigenkänning. Vi har använt neurala nätverk för att klassificera satellitbilder av olika stadsområden, och därigenom analyserat vilka neurala nätverk som presterar bäst.

Moderna karttjänster är till stor grad automatiserade, där olika program automatiskt upptäcker fel i kartdatan. Vårt examensarbete handlade om att utnyttja framstegen inom neurala nätverk för att jobba mot att upptäcka fel i kartdatan gentemot de faktiska satellitbilderna, i vårt fall när och hur olika områden i städer förändrats. Neurala nätverk är en form av maskininlärning, som är ett fält inom artificiell intelligens som är utomordentligt bra på bildigenkänning. Det neurala nätverket får titta på ett stort antal bilder och gissa på vilken stadstyp den nuvarande bilden representerar. Med ett tillhörande facit blir nätverket med tiden bättre och bättre på att göra de här klassificeringarna.

Facit togs fram genom att de stora satellitbilderna delades upp i mindre bilder som sedan med hjälp av en databas klassificerades till den stadszon som varje liten del av satellitbilden tillhörde. Våra nätverk tränades sedan på dessa bilderna med målet att de ska kunna bestämma stadszonen av bilder som nätverket inte sett förut. I bilden kan vi se hur nätverket klassificerar Gamla Stan i Stockholm. Dem röda och gröna områdena motsvarar kommerciella respektive industriella områden. Det blåa området i vänstra hörnet motsvarar en felklassificering och ska representera en skog. Andra områden vi klassificerade var bostadsområden, parker samt parkeringar.



I själva verket finns många olika typer av neurala nätverk som är bra på olika saker, beroende på hur komplext problemet är. En del av vår tes var att undersöka vilken av arkitekturerna fungerar bäst på satellitbilder. Vi undersökte åtta olika konstruktioner av neurala nätverk, s.k. arkitekturer. Arkitekturen som presterade bäst av dessa var DenseNet, som också är en av de nyare, mer moderna arkitekturerna. Nätverket med denna arkitektur nådde en träffsäkerhet på 89.5 procent.