Facade labelling using neural networks

by

Thomas Kolenbrander
Bart van Oort
Frank de Ruiter
Tim Yue

to obtain the degree of Bachelor of Science
at the Delft University of Technology,
to be defended publicly on Wednesday June 28, 2017 at 17:00.

Project duration:	April 24, 2017 – June 28, 2017	
Supervisors:	Dr. J. van Gemert	TU Delft
	Dr. S. Khademi	TU Delft
	Ir. M. Zaal	De Energiebespaarders
	Ir. J. Broek	De Energiebespaarders
Bachelor Project Coordinator:	Ir. O. W. Visser	TU Delft

This thesis is confidential and cannot be made public until June 28, 2017.

An electronic version of this thesis will be available at `http://repository.tudelft.nl/`.

Summary

De Energiebespaarders is a company that helps home owners make their homes more energy efficient. In order to estimate the costs and benefits of installing energy saving measures, a client can upload a picture of their house which is then analysed to determine the surface areas of windows, doors and walls. Following from this result, De Energiebespaarders are able to provide the client with a quotation, without the need for an installer to visit.

There was already a system in place to perform this task but it was slow, inconsistent and heavily dependent on user input. The aim of this project was to create a better system using machine learning techniques. In order to do this, we chose to use a convolutional neural network, more specifically a conditional generative adversarial network. The advantages of this type of network include fast training and learning its own loss function.

The implementation of this network together with its surrounding architecture, which allows the user to upload their picture, fulfil the needs of De Energiebespaarders as the system performs faster than their previous system. The accuracy is similar or better in most cases, but has not improved for all cases, although our system is more flexible than their previous system, as it is able to handle obstacles like trees and cars, taking away some user dependency by doing so. However, with more training data, our system could become more accurate.

Preface

This document is the final report for the bachelor project, conducted in the final year of the bachelor of Computer Science. It contains information on the problem proposed to us by De Energiebespaarders, a fast growing start-up company in Amsterdam with a focus on saving energy. The report describes our approach to applying machine learning in the form of a neural network for the recognition of windows, doors and walls from pictures of houses, and shows the results we have achieved. Finally, we conclude to what extent the problem was solved and provide recommendations on what can be done to further improve our product.

To conclude this preface, we would like to extend our thanks to a number of people who helped us through this project. First, we want to thank our supervisors Jeffry Broek and Mark Zaal from De Energiebespaarders for their hospitality within the company, their guidance and their support in the project. We also want to thank all other employees of De Energiebespaarders, in particular Carsten Griessmann who supplied key components for building a server to run our system within the company. Our thanks also go out to our supervisors Jan van Gemert and Seyran Khademi from the TU Delft, who guided and kept track of our progress throughout the project, and provided us with in-depth knowledge and insights into the techniques we were using. Finally, we want to thank Niels Heisterkamp for useful discussions.

Thomas Kolenbrander
Bart van Oort
Frank de Ruiter
Tim Yue

Delft, June 2017

Contents

1 Introduction **1**
1.1 Structure of this document . . . 1

2 Research **2**
2.1 Problem description . . . 2
2.1.1 Company description . . . 2
2.1.2 Current situation . . . 2
2.1.3 Desired solution . . . 3
2.2 Requirements in MoSCoW . . . 4

3 Design **5**
3.1 Design goals . . . 5
3.1.1 Performance . . . 5
3.1.2 Accuracy . . . 5
3.1.3 Maintainability . . . 6
3.1.4 Other . . . 6
3.2 Replacement of the current system . . . 6
3.3 Segmentation and classification . . . 6
3.4 Semantic segmentation using a neural network . . . 7
3.4.1 Convolutional Neural Networks . . . 7
3.4.2 Dilation . . . 8
3.4.3 pix2pix . . . 8
3.5 Area calculation . . . 11

4 Implementation **12**
4.1 Development process . . . 12
4.2 Software Architecture . . . 12
4.3 TrumPy . . . 13
4.3.1 Neural Network . . . 14
4.3.2 FeaturesFactory . . . 14
4.4 PutIn . . . 16
4.4.1 NodeJS Server . . . 16
4.4.2 ReactJS User Interface . . . 16

5 Testing **19**
5.1 Python . . . 19
5.2 JavaScript . . . 19
5.3 Software Improvement Group Feedback . . . 20

6 Evaluation **21**
6.1 Results . . . 21
6.2 Requirement evaluation . . . 24
6.2.1 Must have . . . 24
6.2.2 Should have . . . 24
6.2.3 Could have . . . 24
6.3 Design goals evaluation . . . 25
6.3.1 Performance . . . 25
6.3.2 Accuracy . . . 25
6.3.3 Maintainability . . . 25
6.3.4 Other . . . 26

7 Conclusion **27**

8 Recommendations **28**
8.1 Training data . . . 28
8.1.1 Colours . . . 28
8.1.2 Roofs . . . 29
8.2 Image rectification . . . 29
8.3 Specialist networks . . . 29
8.4 Window frame detection . . . 29

9 Discussion **30**
9.1 Training data . . . 30
9.2 Ethics . . . 30

Bibliography **31**

A Software Improvement Group feedback **33**
A.1 First feedback . . . 33
A.2 Second feedback . . . 33

B Original project description **34**

C Datasets **35**

D Extra results **37**

E Infosheet **39**

1 Introduction

With global warming becoming a bigger problem, fossil fuels becoming scarcer, thus more expensive, and clean energy solutions not being able to fully replace them as of yet, it does not come as a surprise that people have started looking for ways to save energy. This may start off with simple changes, such as turning off devices instead of leaving them idle, or switching from conventional light bulbs to LED lighting. Such simple changes are cheap and easy to implement and it is equally cheap and easy for a person to calculate how much it will save and cost.

But what about bigger changes? Staying close to home, what about renovating a house in order to make it more energy efficient? The home owner may want to install more energy efficient windows or better wall insulation. However, finding out how much this actually saves and how much this will cost is often time consuming, difficult and may even cost money in itself, as it is currently necessary for an installer to physically visit the house and perform measurements of the house. Based on these measurements, the installer can then create a preliminary quotation. The problem with this is that the home owner then also has to pick an installer, find out how much it costs to have this installer perform the required measurements and find out whether he is actually reliable and provides quality services. This is undesirable as it deters home owners from considering these options.

In the current age of computers, mobile phones, and freely available information and services over the internet, it is desirable to easily receive insight in the costs and savings of the previously mentioned measures. It should be just as easy as it is to rent and pay for an apartment with a few clicks from the comfort of your own home, or to gain insight into in-home energy usage with the help of a smart thermostat.

This is exactly what the company De Energiebespaarders is trying to achieve. They want to allow a client to supply a couple of pictures of their house along with information about the house itself, have the data automatically analysed and then provide the client with a preliminary quotation without the need for an installer to visit. This does, however, require a system that is able to recognise the required features in the pictures and calculate the information necessary for De Energiebespaarders to be able to provide the user with such a quotation. During our project we have developed such a system, with this document being a report on different aspects of the project.

1.1. Structure of this document

The project started by creating a description of the problem, analysing it and setting requirements for the final product, which is described in chapter 2. Chapter 3 then describes the design of the system, including what design goals were taken into account and how the current design came to be. Chapter 4 discusses how the design has been implemented, while chapter 5 discusses the way the implementation was tested. The results of the project are presented in chapter 6, along with an evaluation of the requirements and design goals. Chapter 7 provides a short summary of this project and the conclusions drawn from it, with chapter 8 providing recommendations on possible ways to improve our product, and chapter 9 providing a discussion on training data and ethics. A few appendices were added, starting with appendix A which contains the feedback received from the Software Improvement Group (in Dutch). Appendix B shows the original project description as drafted by the company and shown on BEPsys. The available datasets of facade images for machine learning are listed in appendix C. A brief overview of the results achieved by our system can be found in appendix D. Finally, the infosheet is attached in appendix E.

2 Research

This chapter elaborates upon the problem that the company faces and what a desired solution should look like, which is explained in section 2.1. The proposed solution is then split up in different features and a set of requirements for our final product is defined in section 2.2, prioritised according to the MoSCoW model [8].

2.1. Problem description

The project started by visiting the company and getting to know who they are, what their problem is and observing their current workflow. This section describes what was concluded from this visit and what the desired solution should look like. Section 2.1.1 describes the company and their activities. The current situation and their current system are described in 2.1.2. Finally, section 2.1.3 discusses what their desired solution should look like.

2.1.1. Company description

Our client is a fast growing start-up from Amsterdam called 'De Energiebespaarders'. They aim to help home owners make their homes more energy efficient. The main problems for households to go green are the lack of insight in the costs and benefits of what a home owner can do to save energy, and poor transparency in the price and quality of installers. De Energiebespaarders aims to solve this by providing an easy way for home owners to get such insight, by automatically analysing pictures of a house using data and algorithms in order to advise people what can be done to further save energy and what this will cost, without the need for an installer to physically visit the house. The home owners can then order such products, including installation services, from De Energiebespaarders, thus providing a one-stop shop for home energy saving.

2.1.2. Current situation

The development team at De Energiebespaarders has created a system called 'OBAMA', which can recognise the windows, doors and walls in a picture of the facade of a house, and calculate the surface areas of each of these features using a given scale. This system, however, does require the user to manually mark those features by placing a dot where the feature is located. Aside from that, the processing time is rather long and the accuracy of the system is not so great either, especially when there are obfuscations partially covering certain features, such as an ornament on the door, a tree or a bicycle obscuring parts of walls and windows.

Currently, OBAMA requires a specific way of providing a picture to be analysed, and the user interface of OBAMA, used for its image manipulation functionalities, is neither user friendly nor professional enough to be used by actual clients, as can be seen in figure 2.1. Therefore, the way De Energiebespaarders work now is that clients send in some pictures of all sides of their house and an employee of De Energiebespaarders then manually adjusts these pictures to conform to OBAMA's requirements. The adjusted image is then fed to OBAMA to be analysed, after which the employee performs some final adjustments to the features found. This is time-consuming and also too complicated for actual clients to use.

Figure 2.1: User interface for the current OBAMA system.

2.1.3. Desired solution

De Energiebespaarders want to improve OBAMA with machine learning, so that it will become faster and more accurate at recognising and classifying features in a facade. Ideally, they want their clients to be able to just provide a couple of pictures of their house on an intuitive web interface, after which these pictures are analysed to find the correct features, with minimal need to correct these. The user should not need to manually mark them in the picture as is the case in OBAMA, and the system should be able to handle previously mentioned obfuscations, while being able to provide a result in a short amount of time.

2.2. Requirements in MoSCoW

Following from the problem description and in consultation with De Energiebespaarders a number of requirements have been set for the software product following the MoSCoW model [8]. These are as follows:

- **Must have**
  - Automatic detection of windows, doors and walls.
  - Calculation of surface areas after user inputs scale reference.
- **Should have**
  - Possibility to add user's data to the training dataset.
  - Ability to determine whether picture is actually suitable for analysis by means of a confidence ratio.
  - Obstacle detection to correctly identify features and their areas even when partially obstructed but the area still remains a single surface.
- **Could have**
  - Material Recognition to aid feature detection and classification.
  - Automatic inference of image scale (e.g. using openly available geolocation data).
  - Obstacle detection to correctly identify features and their areas even when the feature is split by the obstacle.
- **Won't have**
  - Material Recognition to identify the materials of the features.
  - Generation of a 3D model of the house using the provided pictures
  - Ability to retrieve and use Google StreetView imagery for analysis after user inputs address.

3
Design

Following from the requirements and in consultation with De Energiebespaarders, we have set a number of design goals which were taken into account while designing and implementing the final product. These design goals are listed and explained in section 3.1. Following from these design goals, the requirements and the research done during the research phase, several designs for the product are proposed in sections 3.3 and 3.4.

3.1. Design goals

This section elaborates on the main design goals for our product. These design goals are guidelines to the design and implementation of the final product and provide us with a goal to work towards. The main design goals we have decided to aim for and will elaborate upon, are performance, accuracy and maintainability in sections 3.1.1, 3.1.2 and 3.1.3 respectively. Finally, we have two separate design goals that are discussed in section 3.1.4.

3.1.1. Performance

The first main design goal of our product is performance. The users of the system do not want to wait for several minutes until it has finally spit out an answer. In fact, in consultation with De Energiebespaarders we decided that the process of uploading an image to the system, having it analysed, and getting results in the specified output should preferably not take any longer than one minute, including the user interaction. In order to help achieve this, we have set the goal for the detection of features in a picture, such as windows, doors and walls, to take 30 seconds at most.

Aside from performance expressed in time, our system should also be able to perform in terms of robustness. It may occur that a client sent in photos where a feature is obstructed by an object, such as a bicycle or a tree. In the case of a partial obstruction, we want the system to still be able to correctly detect the specific feature. On the other hand, if this turns out to be impossible and the photos are therefore deemed unfit for analysis, then the system should be able to recognise this and let the user know.

3.1.2. Accuracy

The second main design goal is accuracy, as the results that our end product delivers are going to be used to give a preliminary quotation on what the home owner could do in order to save energy, and how much this will cost. While it is possible to manually adjust the exact bounds of the recognised features before calculating their size, this is time-consuming and inefficient. We aim for our system to be more accurate than OBAMA currently is, especially in regard to special situations, e.g. when there is a tree or bicycle (partially) obstructing a feature.

An often used metric for accuracy is to look at the percentage of pixels that correspond with the ground truth. However, with the implementation that we have chosen, this metric is in itself not completely accurate. The problem here is that the network should not just be judged upon whether it is able to correctly recognise whether a pixel represents a certain feature or not. Instead, it should also be judged upon whether it can grasp the context of the pixel, in the sense that the network should also be able to successfully identify whether the pixel represents a physical object that is (partially) covering a feature. Sadly, the CMP facades dataset [18] that we included in our own dataset sometimes did not label a window or other feature when it was occluded by a tree. This means that the network could be

punished unjustly when it predicts a window behind a tree, that does not occur in the ground truth, but does exist in the picture, thus making our metric less accurate.

With a large enough dataset of aptly labelled images, this metric may grow more accurate, but we do not have one sufficiently large. Therefore, in order to properly be able to gauge the network's ability to grasp context in a picture, it was decided to judge the accuracy of the neural network by manually comparing the resulting segmentations with the original images and the ground truths.

3.1.3. Maintainability

The third main design goal we have set is maintainability, seeing as it is important that other developers can also understand how our code works and how to improve or extend it in the future. In order to achieve this, we will document the functionality of every method in the code and also create a document explaining the software architecture. While on the subject of software architecture, we have also decided that the system should have an API in the same way OBAMA does now, so it can be run and developed separately from any user interface associated with it, which improves maintainability.

3.1.4. Other

Finally, two design goals were set that do not fall in the categories above. These were deemed less important, but will still be taken into account. The most important one of these is the goal that we want our system to have some form of a user interface, so that the employees of De Energiebespaarders can work efficiently with the system. It was deemed outside of the scope of our project to create an intuitive user interface for the clients of De Energiebespaarders, but a simple user interface was decided to be necessary, not only for De Energiebespaarders, but also for ourselves to allow for easier testing.

The last design goal is that of scalability. The idea is simple: The end product should be able to handle multiple users at the same time without crashing, while also providing the option of adding an extra server so the load on the system can be balanced easily. This should be rather easy to do, as our system should have an API, as previously stated. Then, each server could run a separate instance of the API and a central balancer will split the traffic between the different servers.

3.2. Replacement of the current system

One of the challenges of our project is the presence of an already existing system used by the company: OBAMA. The question soon arose whether we should extend the current system, or build a completely new one ourselves. In terms of the used algorithms, what we found was that the old implementation was improvable with machine learning. However, a neural network has the potential to be more flexible and robust, since it identifies correlations and therefore handles obstructions and noise better naturally, given that an appropriately labelled and sufficiently large dataset is fed into it. A second point is that the use of neural networks moves the workload to the training phase. When in production, the neural network processes an image much faster than the traditional image segmentation techniques used by OBAMA. This led to the decision to replace the image segmentation algorithms in OBAMA with a neural network.

The decision to replace the rest of OBAMA as well comes from a combination of difficulties in the briefing of the project. The difficulties mainly came from the fact that the lead developer of the particular system was no longer working there, so it got harder to get a clear overview on what was already implemented and how it worked. It was also difficult to get the system running ourselves. The front-end also left much to be desired. As such, with the freedom that De Energiebespaarders gave us for the project, the decision was made to replace OBAMA as a whole. OBAMA's successors are called TrumPy and PutIn.

3.3. Segmentation and classification

The initial design of the system consisted of three parts, namely image segmentation, a neural network to do classification and area calculation. The image segmentation would segment the image into different features, the neural network would decide whether the feature is a window, door or wall and the output of these two stages is used to calculate the surface area of the identified features. The image segmentation would be implemented using graph-based segmentation as described by Felzenszwalb and Huttenlocher [13]. The choice to use graph-based segmentation was made due to its simplicity

and speed compared to other traditional segmentation methods. More recent methods as proposed by Chen et al. [11] use neural networks to do segmentation. These segmentations are more accurate but this comes at the price of a higher complexity.

The results achieved by the graph-based segmentation were not as accurate as predicted. On some pictures the segmentation was almost perfect while on others key features like doors and windows were not detected, as shown in figure 3.1. As the system should be able to work on any type of house in the Netherlands, it became clear that another solution was needed. The results achieved by neural networks showed more potential, so it was decided that the neural network would have to take care of the segmentation as well. This new design will be described in the following section.

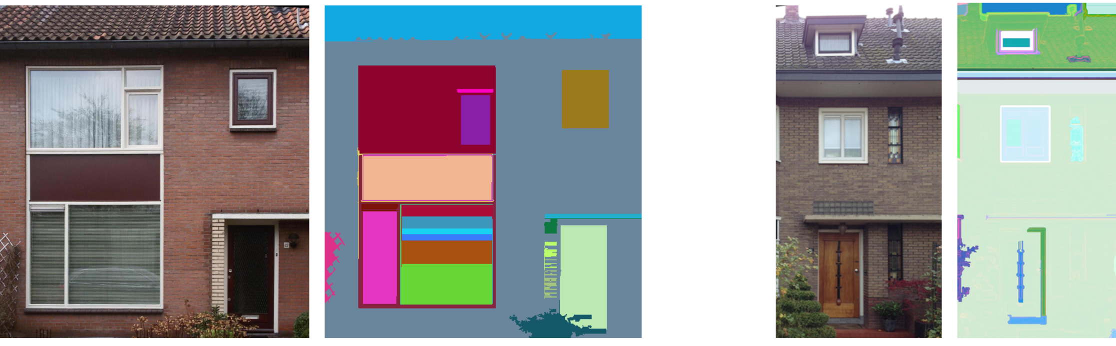


Figure 3.1: An accurate segmentation on the left side and an inaccurate segmentation on the right

3.4. Semantic segmentation using a neural network

Initially, the goal for the neural network would be to classify each feature discovered by the segmentation as a window or a door. Classification is a well documented task for convolutional neural networks, reaching accuracies as high as 93.72% for standard datasets like CIFAR-10 [14].
However, as traditional methods for segmentation failed to produce usable results, we decided to use a neural network for the entire pipeline from image to labels, and thus our goal shifted towards semantic segmentation. In their paper, Shelhamer et al. mentions two techniques that seemed especially useful to us. The first is a dilation based method from [23] that was specifically designed to tackle semantic segmentation tasks. The second is the U-net from [20] that caught our attention because of the low computational costs, since the company failed to provide us with any development hardware in time. Furthermore, since our client should be able to continue to use the solution after this project ends, we needed a technique that was easy to explain and use. The generative adversarial network from [15] combines a U-net with a relatively easy to use and understandable training system, since the results are immediately interpretable (no colour conversions needed), and the training data consists solely of images for both input and ground truth.

This section starts with an introduction to convolutional neural networks in section 3.4.1, followed by a description of the dilation method from [23] in section 3.4.2. This section ends with a description of pix2pix in section 3.4.3, the technique we have settled on for our neural network.

3.4.1. Convolutional Neural Networks

Both [23] and [15] are implementations of convolutional neural networks. Much like other neural networks they are modelled after biological brains, this kind in particular was modelled after the visual cortex in the brain, and is particularly suited for reasoning over images. Convolutional neural networks differ from other networks through the use of special layers, called convolutional layers. Each convolutional layer in a network adds a layer of abstraction to the network's understanding of the images it has seen. For example, the first layer generally learns to detect edges and blobs of colour. The second layer may learn simple combinations of these edges and blobs, thus learning to recognise simple shapes. Then, the third layer may learn to detect combinations of shapes, and so on. Convolutional layers are able to efficiently work with images by applying convolutions; they detect features in a small

area of the image and slide this window across the input image. Using these convolutions, the network can detect a certain feature anywhere in the image, while only learning the parameters to recognise that specific feature. As with all neural networks, they depend on having a large, accurate dataset to train on. Figure 3.2 illustrates the importance of the training dataset.

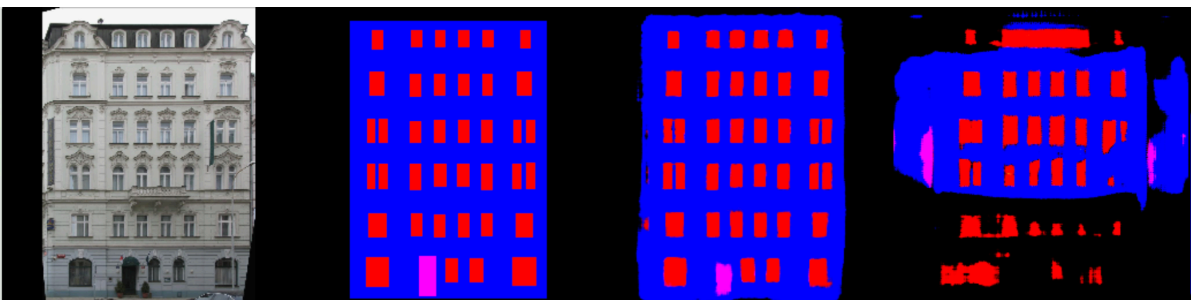


Figure 3.2: Left: a test image from the CMP facades dataset. Left-center: ground truth. Right-center: result from a model trained on both CMP and our own data. Right: result from a model trained only on our own data, which only contains terrace houses. These results clearly show the influence of the training dataset. The model on the right is not familiar with apartment buildings and is therefore at a significant disadvantage for images containing such buildings.

3.4.2. Dilation

In their paper, Yu and Koltun [23] note that most deep learning methods for image segmentation are adaptations of methods designed for image classification. They argue this is bound to yield suboptimal results and construct their own method for semantic segmentation from the ground up.

Yu and Koltun describe an adaptation of convolutional layers in which the receptive field is expanded through the use of a dilation. This adaptation results in a layer that uses an exponentially greater context during the convolution, while using only a linearly greater number of parameters. Overly simplified, the dilated convolutional layers use another convolution internally to add another level of context aggregation to the output of a single output vector.

Using this method, we achieved reasonable segmentation, as seen in 3.3. However, the labels had a tendency to fuse if they were too close to each other and the shapes of the labels did not correspond very well with the shapes of the original image. To resolve these issues, we investigated another technique, pix2pix [15], although it is not specifically targeted at semantic segmentation.

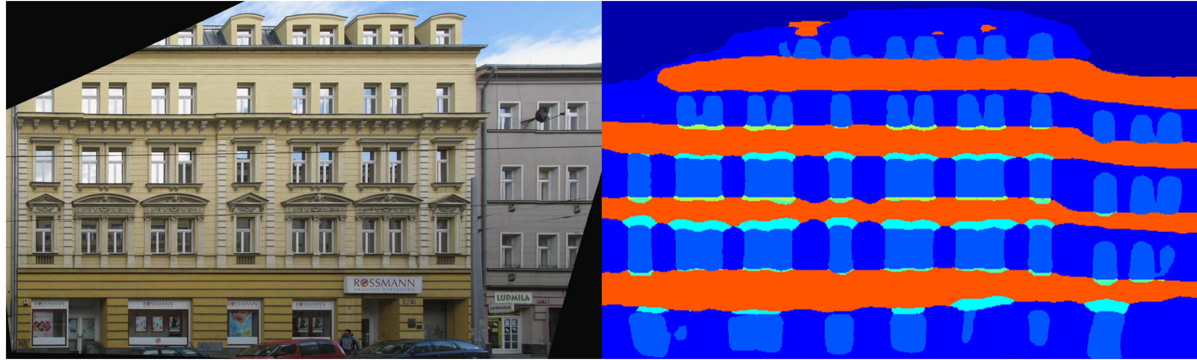


Figure 3.3: Sample result of semantic segmentation using a network with dilated convolutional layers from [23]. Note: this segmentation uses all the classes from CMP, we later decided to drop many of those and focus on walls, doors and windows.

3.4.3. pix2pix

Pix2pix [15] is a technique recently developed by Isola et al.. The primary goal of pix2pix is to provide a general solution for image to image machine learning tasks. Usually in these kinds of tasks, a developer must carefully construct a loss function to score the performance of the network, which can be very difficult to design. However, the derivative of such a loss function directly influences the way the network learns. So, if the loss function is imperfect, the consequences can range from slower learning

to suboptimal results due to over fitting. Pix2pix does not require a loss function because it learns its own loss function using another neural network. As such, the network is theoretically able to find an optimal loss function for any given image to image learning task. However, an optimal loss function does not mean a network cannot over fit, it only ensures the network will progress towards an exact optimum.

Pix2pix is a conditional Generative Adversarial Network (cGAN) and trains two networks simultaneously. One network, called the generator, generates images based on input images. The other, the discriminator, tries to determine whether the image is either generated, or the ground truth belonging to the input. The discriminator looks at patches of the output image and assigns each patch a rating of how sure it is that the patch is real or fake. These ratings, along with a per-pixel comparison of the output with the ground truth, are used by the generator to improve its results. The discriminator is afterwards told which image was real and improves itself based on that information.

The author suggests that pix2pix as a whole is not particularly suitable for labelling, and suggests disabling the learning loss function for such tasks. However, the structured nature of our particular task suggests that a method that can learn the patterns specific to this task, could result in more accurate labelling. Consider, for example, figure 3.4. Clearly, the labelling is nowhere near perfect, but this picture does clearly show that the network has learned more than just pixel labelling. Although results like figure 3.4 are impressive and suggest that the network can learn such patterns, achieving stable results was impossible due to the small size and inconsistency of our dataset.

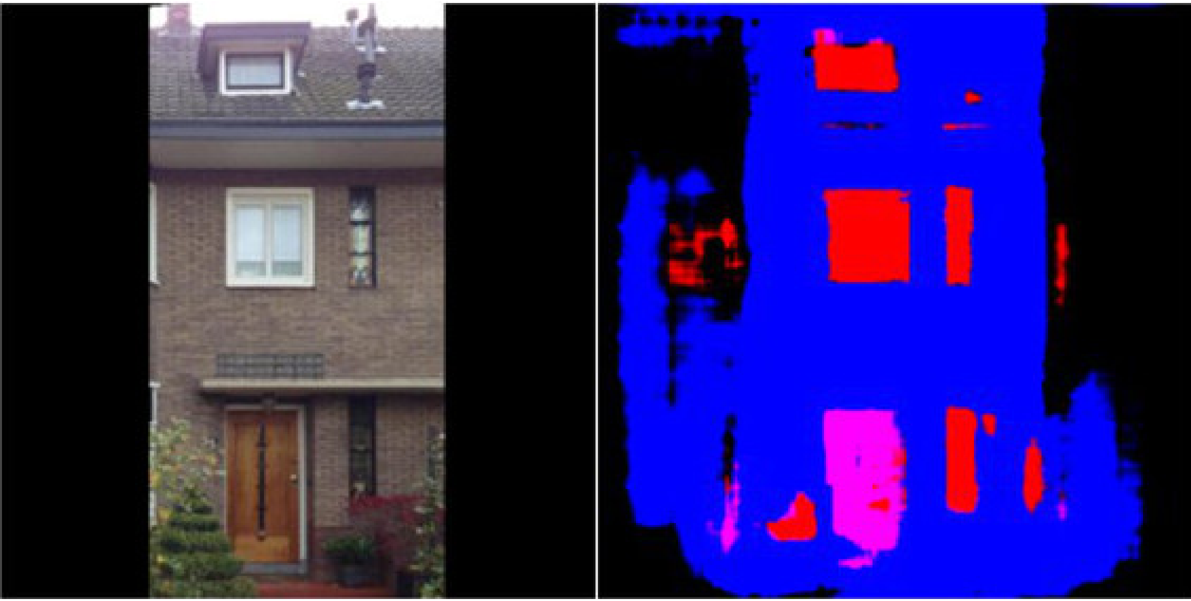


Figure 3.4: Example where the network has predicted a feature that is mostly outside of the image. Moreover, the bounding box of the left window is comparable to that of the right window. This suggests the network has learned that a window frame tends to be attached to a window, and that similar frames contain similarly sized windows.

Convolutional neural networks, unlike cGANs, can easily take over a month to train and afterwards, take up to a couple of seconds to process an image. This is another general problem with convolutional neural networks that pix2pix has overcome. Pix2pix uses a U-net for the generator, which consists of two stacks of convolutional layers of equal height, as seen in figure 3.5. One stack downsamples the input image until it is a single pixel, then passes that pixel to the other stack. The other stack upsamples the single pixel back to the resolution of the original image. Although pixels throughout this network can have far more dimensions than the RGB channels typically used in images, – we used 64 in most layers, as suggested by Isola et al. – it should be obvious that a lot of information is lost during the downsampling. This is overcome using skip-connections that append the channels of each downsampling layer to the channels of the corresponding upsampling layer. These shortcuts allow the network to find long range relations between pixels after downsampling, while also retaining the short

range information at every layer. This makes U-nets particularly suited for tasks such as labelling, where the input and output share an underlying structure. [20]

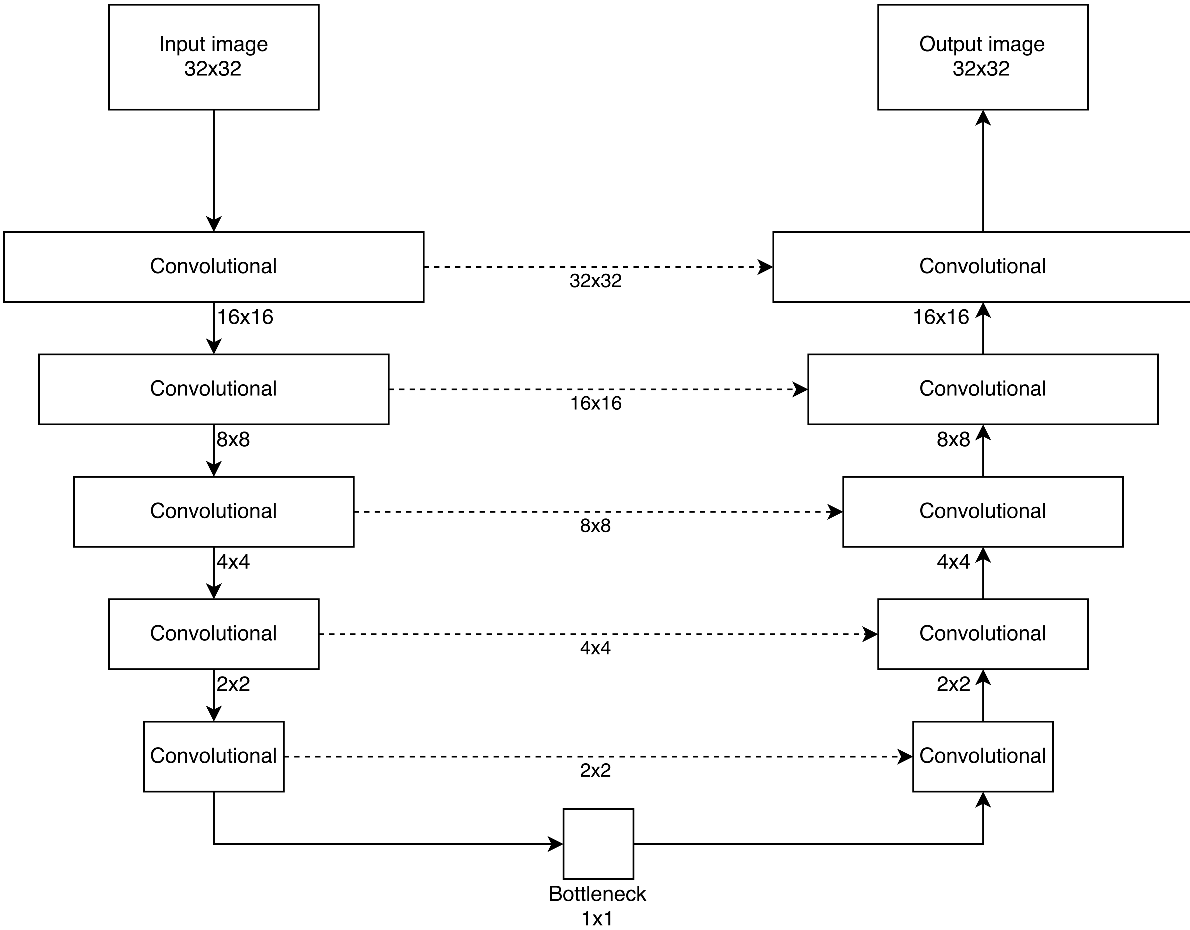


Figure 3.5: A U-net is a symmetric encoder-decoder network. To avoid data loss as information flows through the bottleneck, the channels of each layer i are appended to those of layer $n - i$.

Often, a pixel-wise loss function is used to determine how well a network is doing. The simplest form of such a loss function is the L1 loss function, which simply computes the sum of the errors in the dimensions of the output, in this case in each of the colour channels red, green, and blue. Pix2pix' loss function yields another advantage for highly structured labelling tasks that, for as far as we know, has not yet been described in literature. Pix2pix uses a patchGAN to learn its loss function along with an L1 loss that penalises the absolute error per pixel. For the sake of brevity, we will ignore the L1 loss in pix2pix for duration of this paragraph. A patchGAN looks at individual parts (or patches) of its input image and tries to guess whether they were generated by the network or belong to the ground truth. This means that the results of the generator, which are fed to this patchGAN, are not judged on a per-pixel basis, but instead are judged on whether they seem realistic according to the mean of the training dataset. Other methods usually measure accuracy by comparing the pixels produced by the network with the pixels in the ground truth. Although this seems like a reasonable way to rate the output of the network, the fact that it is based on only one ground truth image presents a problem. If the ground truth image is imperfect or deviates from the rest of the dataset, judging on a per-pixel basis is bound to confuse the network. Using a patchGAN resolves this issue for the generator, because it is judged on whether its results correspond to what the discriminator expects based on the entire dataset. The issue still remains for the discriminator, but the effect of this on the images produced by the generator should be negligible due to the gradient fading while propagating back through the discriminator. In other words, a wrongly labelled image is less harmful because it does not directly influence the generator.

3.5. Area calculation

In the final phase, the labelled output from the previous stage(s) is used to get the information that De Energiebespaarders is primarily interested in, which is the surface area of each feature. At the very least, the surface areas will be calculated separately for each feature with a scale reference input by the user. This scale reference is simply a number specifying the distance represented by the height or width of a single pixel, and can be calculated by having the user place two points on the image and input the distance between these points. Due to the simplicity of the output of the previous stages, neural networks are not required for the area calculation and traditional computer vision techniques can be used to process the segmented image. This has the benefit that it is easier to find out the border points of the features using these computer vision techniques and calculate the surface area from there.

In the area calculation phase, the individual features are also packed into a list of feature objects. In these objects, their feature type (window, door, wall), surface area in m^2 and their border points are recorded. The first two are to properly list the individual features with their respective surface areas to the user. The border points are there to make it easier to build in functionality which allows the user to tweak the resulting feature shapes. The segmentations will not reach perfection in the near future, so the ability for the user to easily tweak the shapes would help to bring the system into serious use by De Energiebespaarders.

4 Implementation

This chapter describes the techniques used to implement our product and how they have been applied. Section 4.1 describes the development process in general and how we worked together as a team. Section 4.2 explains the high-level software architecture and its implementation. Section 4.3 describes how TrumPy, our Python server and neural network, has been implemented. Finally, section 4.4 addresses the implementation of the NodeJS web server and its accompanying user interface, named PutIn.

4.1. Development process

The development process has been based on the Scrum framework [21]. As usual with Scrum, we worked in sprints, each of them one week long, as the time frame of the project was rather short. To keep track of our sprints, we used Atlassian's JIRA, which already used within the company. In order to ensure good communication within the team itself and between the team and the company, the team gathered almost every workday and worked on-site in Amsterdam for two days per week from 10:30 to 17:30. De Energiebespaarders also provided us with a Slack channel and a WhatsApp group was created for use within our team. The meetings in Amsterdam were especially useful as it allowed us to exchange ideas with the company, get feedback often and change the planning accordingly if necessary.

Although we used Scrum, it was only implemented in a general sense. We did not assign Scrum roles, such as product owner or scrum master, nor did we have a formal starting and ending meeting every day, as we deemed this unnecessary for a group of our size. We did perform a quick check with each other at the beginning and ending of each day to see what each member had been doing. Initially, we also did not have a division in development roles, but over the course of the project different focusses per team member emerged naturally, as described in table 4.1.

Team member	Subject focus
Thomas Kolenbrander	User interface, image processing
Bart van Oort	Server communication, user interface, general management
Frank de Ruiter	Neural network, back-end
Tim Yue	Image processing, back-end testing

Table 4.1: Division of focus of contribution per team member

4.2. Software Architecture

Our product consists mainly of three parts: the neural network (TrumPy), the web server and the user interface. These are programmed in different programming languages and are meant to communicate with each other using HTTP requests and remote procedure calls using Google's remote procedure call library (gRPC)[2], as can be seen in figure 4.1.

The user first selects an image in the user interface and specifies a scale reference. This data is then sent to the PutIn web server through an HTTP POST request, after which the web server transforms the data into a gRPC Request object and sends a remote procedure call to TrumPy's gRPC server. This

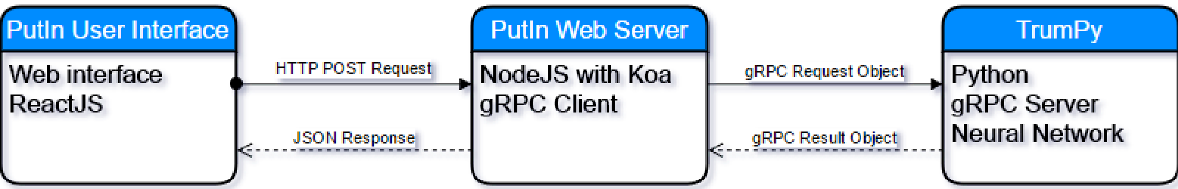


Figure 4.1: Software Architecture

in turn decomposes the request and calls upon its neural network to process the image and recognise windows, doors and wall area in the picture. It also calculates the surface areas of each detected feature. The resulting overlay image with the generated labelling and a list of these features are then wrapped in a gRPC Result object and returned to the PutIn web server. It then returns the result as a JSON response to the user interface, which then shows the results to the user.

4.3. TrumPy

TrumPy is the main system in our product. It is written in Python 3.5 and contains the neural network implementation that has been trained to recognise windows, doors and walls in a picture of a house. More on how this neural network has been implemented is discussed in section 4.3.1. TrumPy also contains an implementation of a gRPC server, thus allowing other applications to use remote procedure calls in order to call the methods from TrumPy's API. Aside from all of this, TrumPy has Python modules that can rectify images, extract the recognised features from the generated image and transform this data into a response for the gRPC server. TrumPy runs inside a Docker container to allow for a stable and clean environment wherever it is hosted. It also allows the TrumPy system to be scaled, as multiple of these Docker containers can be deployed in parallel. The PutIn web server could then keep track of the load in each of these Docker containers and distribute its requests evenly across these multiple instances.

TrumPy is implemented with loosely coupled modules, making the system easy to maintain and extend. Currently, the control flow within TrumPy is as shown in figure 4.2. TrumPy's Launcher is run in order to start and stop the API server module, which hosts the gRPC server and implements the API methods that a remote procedure call may call. When a remote procedure call for analysing an image is received, it reads the image from the request and sends it to the neural network. The neural network processes this and generates a labelled image.

Next, the API server sends the labelled image to the FeaturesFactory module, which in turn extracts the recognised features from the image and converts them to a list of Feature objects. Each Feature object contains its type (window, wall or door), a list of points that form the bounding box around the specific feature and the surface area of the feature according to a given scale. In order to do this, the FeaturesFactory module makes use of different methods in the ImageProcessing and AreaCalculator modules. This was done in order to split the FeaturesFactory's responsibilities into smaller units, therefore improving the readability and maintainability of the code. Once the FeaturesFactory has created its list of features, then it returns this to the API server, which then forms the reply that is sent back to the caller by the gRPC server.

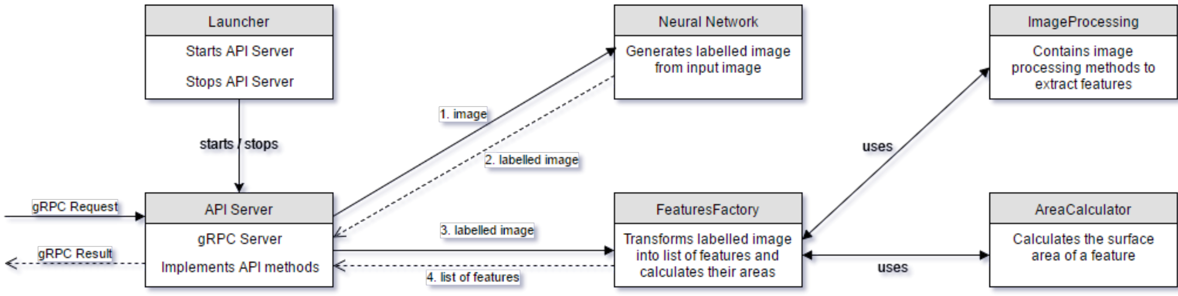


Figure 4.2: Control flow within TrumPy

Originally, TrumPy was meant to be implemented in Java, making use of the Deeplearning4j library [1] for the neural network, as it is well documented and uses a matrix library that is supposedly much faster than Python's equivalent Numpy. All of the group members also already had a large amount of experience with Java and image manipulation libraries, such as OpenCV, also have Java wrappers, so the choice seemed obvious. During the implementation, however, it turned out that Deeplearning4j did not yet fully implement and support some specific functionalities that we needed to have in order to implement the neural network of choice. This meant that we had to make the decision to switch to Python as our main programming language, even though none of our group members had experience with the language.

4.3.1. Neural Network

The segmentation is achieved through a conditional Generative Adversarial Network (cGAN), as described by [15]. Specifically, the PyTorch implementation by Zhu et al. [24] was used. The network was trained on a dataset of 619 images, containing images from the CMP facades dataset [18] and images that we manually labelled ourselves. The images from CMP facades were generated with labels for walls, windows, doors and background, in colours corresponding with the manually labelled images.

Using a U-net, we were able to train the network on a dataset of 907 images (divided into 612 training-, 129 validation-, and 166 test images) with a resolution of 256x256 in 7 hours, using a Nvidia GeForce GTX 1060 with 6GB memory. After training, the network can process an image in 0.06 seconds excluding overhead, using the same hardware.

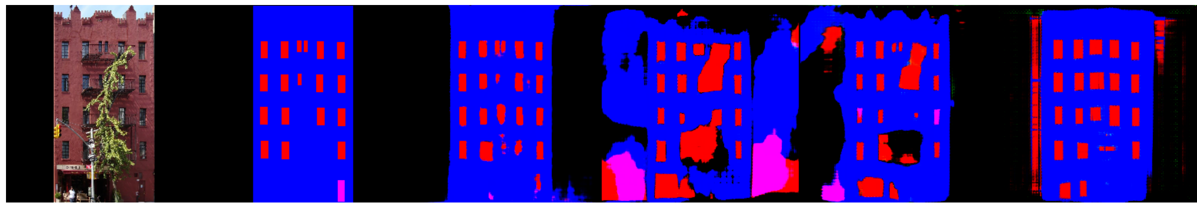


Figure 4.3: Left to right: (1) input, (2) ground truth, (3) Trump Standard + CMP with weight 500 on the L1 loss, (4) Trump Standard only with weight 500 on the L1 loss, (5) Trump Standard only with default weight 100 on the L1 loss, (6) CMP only with default weight 100 on the L1 loss. The input image was taken from the test partition of the CMP dataset. The influence having seen similar shapes in the training phase is clearly visible, but the influence of the L1 loss cannot be concluded from this comparison.

To determine the optimal configuration of the network, we experimented with training using seventeen different weights on the L1 loss, varying from 10 to 500, where 100 is the default. Although Isola et al. suggest using a high weight on the L1 loss, our experiments seem to suggest that a low value might produce better results. This can be explained by the differences in the datasets used. Isola et al. test the network on the Cityscapes dataset, which contains frames from video sequences of street scenes [12]. These images contain many different kinds of objects, with highly varying shapes. Our dataset, on the other hand, contains pictures of buildings, where the labels are nearly all quadrilaterals and even mostly rectangles. The results of our experiments are preliminary; since the deadline for this project prohibited training the network on the full dataset seventeen times, these results were obtained by training the network on a subset of only 120 images, which are visible in figure 4.4. Figure 4.3 shows the effect of varying the weight on the L1 loss in the case where the network was trained on different training sets. However, a definitive conclusion cannot be taken from these pictures.

4.3.2. FeaturesFactory

The extraction of the individual features in the labelled image is done with the use of the OpenCV library. The process is illustrated in figure 4.5. Per type of feature, an image mask is made where the pixels corresponding to this type of feature are coloured white, and the rest black, to separate the features from each other. From there, the contours of the white areas are taken per image mask and should represent a feature. Features with a surface area under a certain threshold are ignored to prevent noise from becoming features, such as stray red pixels being considered tiny windows.

Although the neural network can do some impressive labelling, the output is often times noisy. The labelled features hardly have regular shapes, despite most features being rectangular in reality. To get around this problem, a rectangle is drawn around the recognised contours of the feature so that it has the minimum possible area and is thus the tightest fitting rectangle around the given feature. This is

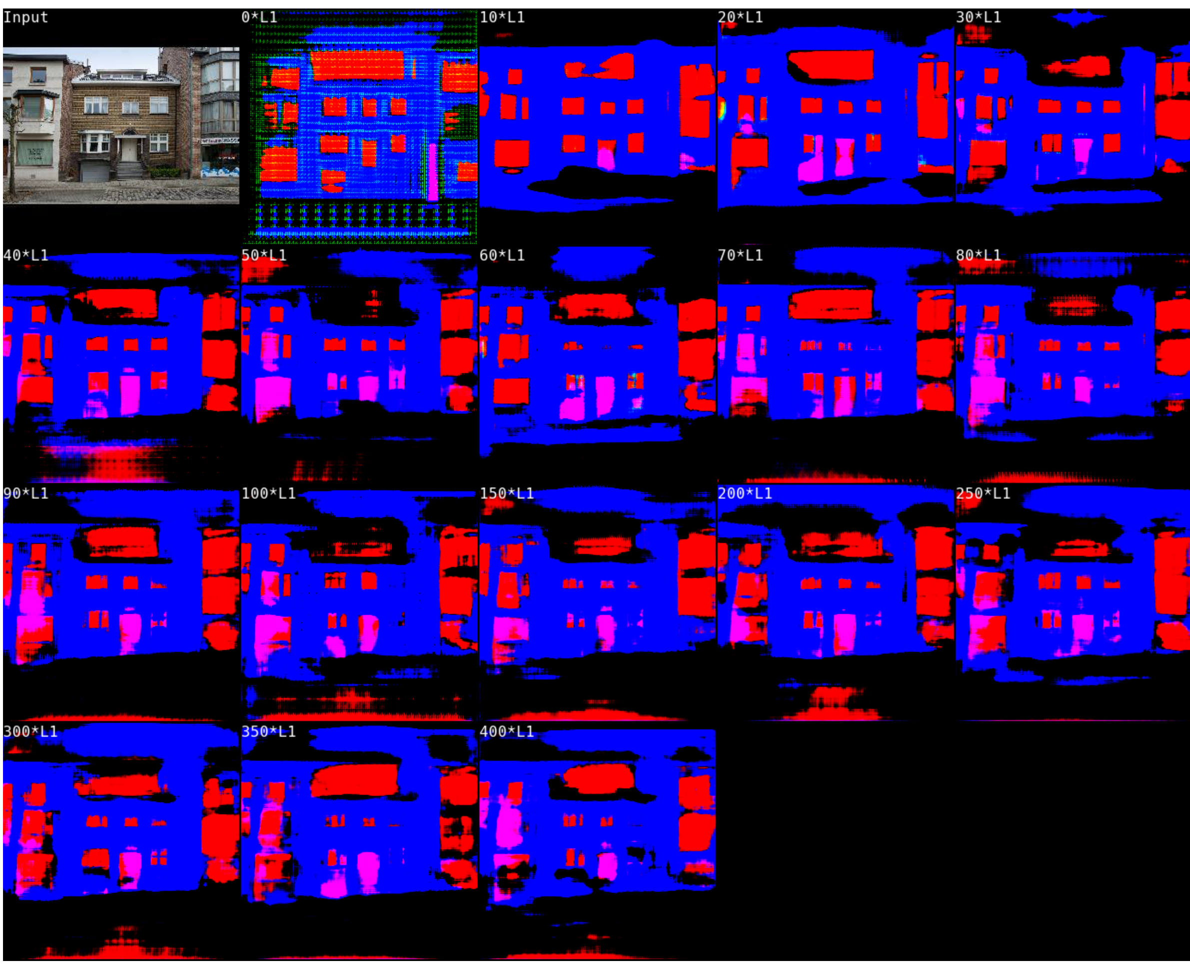


Figure 4.4: Comparison of using different weights on the L1 loss. Throughout the project, we used 100 as the default weight. This comparison suggests that a lower weight on the L1 loss might yield better results. N.B.: the results are preliminary and this trend was not present in all images.

called the 'minimum area rectangle' and is used to calculate the surface area of the feature, instead of using the contours directly. Although it is only an approximation of the contours, using these bounding boxes often times improves the surface area accuracy by completing the broken shapes outputted by the neural network. An added benefit of using these minimum area rectangles and thus only having four points instead of the many contour points, is the possibility to let the user adjust the features in a user interface when the end result is shown. Apart from this, the noise in the input image can cause for false positives and broken shapes. These are dealt with by morphological closing, where the shapes are first expanded and then shrunk again in processes called 'dilation' and 'erosion'.

The only exception to this workflow are the walls. Where doors and windows are usually rectangular, walls vary in their shape more often. Another factor is that the labelling of the outer edges of the walls can be very noisy. As such, taking the wall contours and making a polygonal approximation can have really inaccurate results. Moreover, due to the complexity of the shape of the wall, it is hard to enable the user to adjust the identified wall points afterwards. For these reasons, the decision was made to simply count the pixels in the picture that were the closest to the colour blue, and use that as the surface area of the wall. In the end, the scale reference from the user input is used to convert the calculated surface areas from squared pixels to squared meters. This is then used in the creation of the feature objects as defined in the protocol buffers used by gRPC, after which these feature objects are returned to the API server.

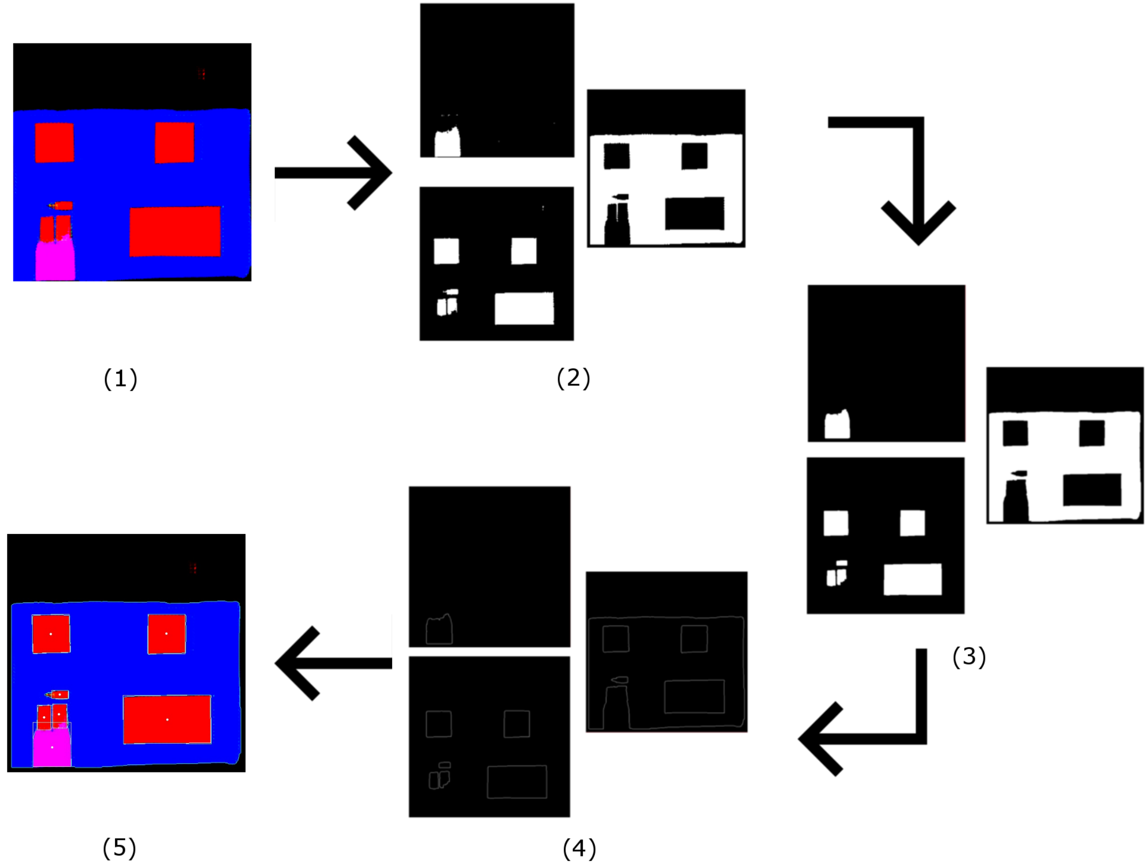


Figure 4.5: (1) Labelled image as returned by the neural network. (2) Binary image masks per feature type. (3) Noise reduction and gap closing by dilation/erosion. (4) Canny edge detection. The edges are representative of the contours used. (5) The source image with centroids per feature drawn and bounding boxes drawn around the features.

4.4. PutIn

PutIn is the name for both the web server and the web interface that are part of our product. Both have been written in JavaScript, with the web server making use of NodeJS and the web interface implementing the ReactJS framework. The implementation of the NodeJS web server is further discussed in section 4.4.1, while the implementation of the ReactJS web interface is discussed in 4.4.2.

4.4.1. NodeJS Server

The NodeJS web server makes use of the Koa framework [4] to serve the ReactJS web interface and is also a gRPC client that may connect to TrumPy's gRPC server. The server also hosts two services: on `/hello` and `/analyse`. The service on `/hello` accepts a GET request and calls TrumPy's gRPC server, which simply returns a text string containing the message `Hello GRPC!` It was first developed as a hello world for the gRPC server and client pair, but was kept in as it is currently mainly meant as a way to test the connection to the gRPC server. The service on `/analyse` accepts a POST request with an image and a scale reference sent as multipart form data. This data is then transformed into a gRPC request and sent to TrumPy for processing. Once it returns a result, the resulting image is converted to a data URI and then sent back to the web interface as a JSON response.

4.4.2. ReactJS User Interface

PutIn's web interface is written in ReactJS and makes use of the Material UI theme [5], which implements Google's material design in ReactJS components. The interface is meant for use by the employees of De Energiebespaarders. It therefore has a minimal and simple, yet functional design and provides an easy way to upload an image to the TrumPy server, as can be seen in figure 4.6.

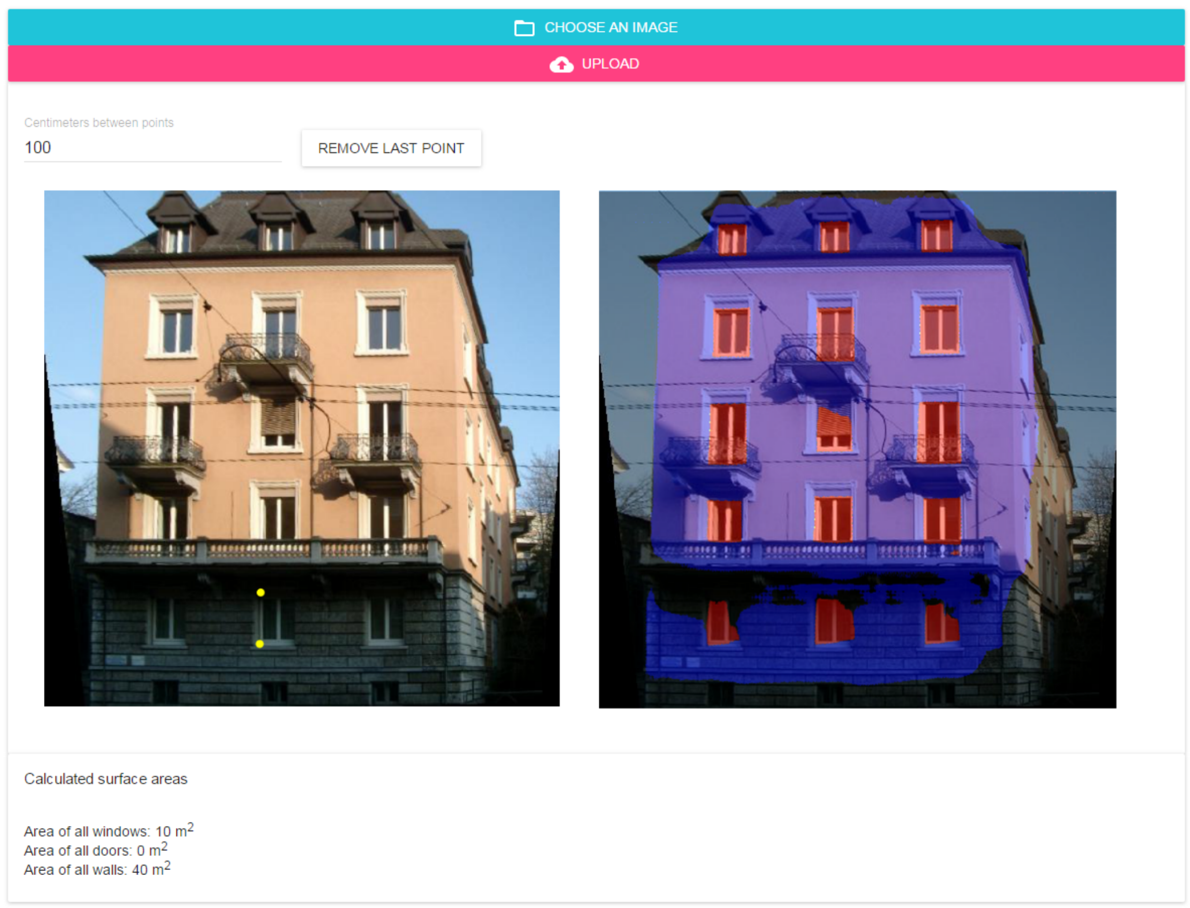


Figure 4.6: ReactJS User Interface

The ReactJS way of thinking prescribes that the structure of the application should be designed based on the single responsibility principle [17], which means that every component has one, and only one, responsibility. With this in mind, a structure of the user interface was created and can be seen in figure 4.7. The component App is the top-level component and contains four other components: ImagePicker, Uploader, RequestView and AreaCalculator. These each render their own parts of the interface.

To start off, the ImagePicker component consists of a button, with which the user can pick an image to start creating a request. The user then sees the image showing up in the RequestView by means of an ImageView, which is simply a container for an image while also allowing for children to be added, thus allowing for overlaying elements. In the RequestView, the user can compose a request by setting a scale reference, which is done by simply placing two points on the image and specifying the distance between these points. This scale reference is then calculated and passed through as the amount of meters represented by one pixel width / height.

Once the user has composed his request, then he can press the upload button, which is displayed by the Uploader component. It then collects the user's input and composes it into multipart form data, in order to send a POST request to PutIn's web server to have the data analysed by TrumPy.

When the web server JSON response with the resulting labelled image and the recognised features is received, it is saved in the App component and received by both the RequestView and the AreaCalculator. The RequestView then uses a second ImageView component to show the original image, overlayed with the labelled image from the neural network, which provides a quick and intuitive way for the user to see how much of a success the segmentation was. The AreaCalculator on the other hand aggregates the surface areas of all windows, walls and doors and displays these below the images. This component was originally designed to actually calculate the surface areas of the recognised features given its bounding points as returned by TrumPy, hence the name, but this feature had to be

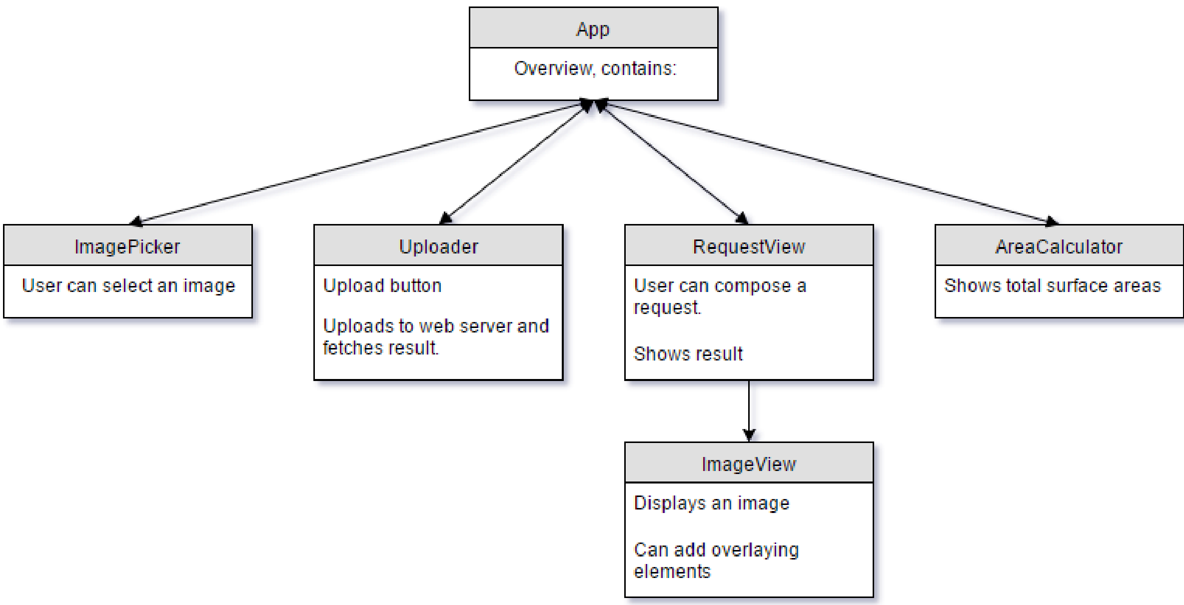


Figure 4.7: Components of the ReactJS web interface

postponed due to the difficulties encountered in the FeaturesFactory module of TrumPy, as described in section 4.3.2.

5 Testing

In order to verify the implemented functionalities, we have created several tests, both in TrumPy (Python) as well as in PutIn (JavaScript). The idea and implementation behind these tests will be explained in sections 5.1 and 5.2 respectively. During the course of the project, the complete code-base was submitted to the Software Improvement Group which provided feedback on our software architecture, maintainability and testing. This feedback and our response will be discussed in section 5.3.

5.1. Python

Testing in Python is done using the unittest framework [6]. One of the main components, the API server, is being tested on successfully starting, stopping and throwing the right errors when invalid requests are made.

The detection of the different features and calculation of their surface areas have been tested extensively as these functionalities are vital to producing good results. Nearly all Python code related to this has been unit tested. However, for a good part of the code, the functionality is to process images. For these functions, the only category of effective automated testing is regression testing; to verify that these functions keep the same functionality when something else was changed. To validate if the results are sufficient, manually testing the functions was required.

As for the neural network, its loss function and its way of training makes use of testing in the sense that the network uses a set of pictures different from its training images in order to validate its training. However, it is unfortunately impossible to write automated tests for a neural network that will fail or pass the network based on the end results, due to the complexity of the network and its unpredictable results. Any extra training or difference in training would invalidate automated tests, even if the results are better. Aside from that, at the start of its training, our neural network implementation makes use of a random initialisation of its weights, which means that results can differ even more when the network is retrained. Furthermore, the results from the network are rather subjective in a way. Labelling of the test data, the ground truths, is done by people, and per interest group there is a different focus and different nuances in its labelling, which may even vary from person to person. The only type of testing the neural network received has been visual inspection of the results.

5.2. JavaScript

Testing the JavaScript code provided a small extra challenge, as the JavaScript code is partially NodeJS and partially ReactJS. The `react-scripts` package used to develop and build the ReactJS code, was first also used to create a few tests for a ReactJS component. However, when tests were added for the NodeJS web server, it turned out that `react-scripts` does not pick up these tests. Thus, we made the obvious choice to switch to the Jest testing framework [3]. "Obvious", because the underlying testing framework used by `react-scripts` is also Jest, and because both ReactJS, `react-scripts` and Jest come from the same company, namely Facebook, making it easy to set up with the existing tests at the time. Jest also allowed us to test asynchronous methods and components with ease.

While the user interface was mainly tested manually due to its simple nature, a number of automated tests were created in order to test the methods used for scale calculation. The NodeJS web server got more tests, namely for the Routes and RequestFactory modules, which are responsible for the implementation of the routes (`/hello` and `/analyse`) used by the system, and for creating gRPC

requests from incoming data, respectively. The tests employ mocking of the gRPC library in order to test whether the routes perform correctly and return the correct results, but also to test whether it correctly handles erroneous or even non-existent input, in the case that it is called incorrectly.

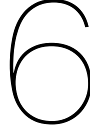
Aside from the manual testing of the interface, no extensive user testing was done. Since the interface that was built is not for use by clients of The Energiebespaarders yet, the user interaction and styling built into the interface is minimal and barebones. Therefore, user testing was deemed unnecessary and issues could be found out by manually testing it ourselves.

5.3. Software Improvement Group Feedback

The first feedback received by the Software Improvement Group (SIG) was mostly positive as the code got a four out of five rating on their maintainability scale. The full feedback (in Dutch) is included in appendix A. The maximum score was not reached due to “a lower score for unit size”, though they did not specify exactly which classes suffer from this. They did note that the Python code generally scored better on unit size than the JavaScript code, as the JavaScript code contained a large amount of logic in anonymous callback functions making the code less readable and testable. Their last remark was regarding test code, as at the time they did not find any for the JavaScript used.

In response to this feedback the JavaScript code received a refactor, which consisted of splitting the larger callback methods into smaller methods, moving them to modules, and adding tests to the now more testable code. As the code-base of TrumPy kept growing, additional tests in Python have also been added in order to ensure the quality of the Python code.

The final feedback received by SIG stated that our maintainability score remained the same while the code-base has been growing significantly. They were pleased to see that we had reduced the unit size in our previously existing code. However, the newly added code still suffered from a large unit size. The implemented tests for JavaScript and Python were also noted and contributed to our high score. All in all, they concluded that we implemented their feedback for the most part, but the unit size could still be improved, even though they once again did not specify where.



Evaluation

This chapter evaluates our final product. To start off, the results achieved by our final product are shown in section 6.1. In order to determine whether the project has been a success or not, the current system is also compared to the requirements and design goals, as defined in chapters 2 and 3. This evaluation is done in section 6.2 for the requirements, and section 6.3 for the design goals.

6.1. Results

The created solution consists of an online user interface where the user is able to upload a picture for analysis, as shown in figure 4.6. After the user has selected two points and specified the distance between these points, the picture can be uploaded to the web server, which then sends it to TrumPy's API server. The picture will then be analysed by the neural network and returned to the user with the calculated surface areas. This is similar to how the current system works, but requires less user input.

In order to determine our success, the results achieved by our system are compared with those achieved by OBAMA, the company's current system, through manual inspection of the results. Figure 6.1 includes pictures of the returned segmentation. Note that OBAMA marks walls red, windows blue, and doors green, whereas our system (TrumPy) marks walls blue, windows red, and doors purple. The first two rows show that our system performs comparable or slightly less than OBAMA. The first row is an image that the OBAMA system was designed for; a simple facade without obstructions. OBAMA performs very well on this image, whereas our system is less accurate. Especially the doors in this picture are not properly recognised, due to the fact that our training data simply is not large enough to contain an adequate amount of doors for the network to learn. More training data would allow the system to perform more accurately on these kind of pictures.

While the door in the picture in the second row of figure 6.1 does not get recognised for the same reason as before, this picture does show that our system is capable of distinguishing between the street and the wall, while OBAMA is not. It is also able to recognise that the wall continues behind the car in the picture. This suggests that our system is able to learn the context of an image. Figure 3.4 in chapter 3 shows this as well. It suggests that the network has learned that a window frame tends to be attached to a window, and that similar frames contain similarly sized windows.

The third and fourth row of figure 6.1 show examples where our system outperforms OBAMA. The third row contains an image taken from the CMP dataset, a dataset that was largely included in our training. OBAMA seems to be struggling with the shadows and balconies in the picture. As our system was trained on similar buildings, the result it produces is almost perfect. This shows that with enough training data, the system could easily outperform OBAMA.

The fourth row also shows an interesting case, as some of the windows in the original picture are obstructed by bushes and are therefore more difficult to detect. It is clear that OBAMA is not designed to handle such cases, seeing as almost half of the windows are not detected, and the other half are merged or inaccurately approximated. Our system on the other hand is able to see through the bushes and returns an almost perfect result. This example shows the flexibility of our system and further shows its ability to grasp the context of a picture, which is also part of the accuracy design goal.

Through our inspection, we come to the general conclusions laid out in table 6.1. For more segmentation examples, see appendix D where images are segmented, and it is determined if the segmentation

Category	OBAMA	TrumPy
Walls	+ Works well when cropped tightly - Often times also labels ground and sky as wall when included.	+ Can usually distinguish walls from ground, sky or other categories and works well generally. - Has a vertical bias with the current training dataset due to tall buildings. Can therefore sometimes label the space above flat buildings as wall
Doors	+ Generally identifies neatly - Ornaments, windows or mailboxes in the door can make the identified door smaller	- Often identified as a window
Windows	+ Generally good window identification and also identifies window frames - Window frames are given the same colour as windows and their identification is very varying in accuracy	+ Usually acceptable segmentations - Sometimes hallucinates windows in the sky
Obstructions	- Either features behind obstructions are not recognised at all, or such features track the shape of the obstruction, or random shapes are identified as features	+ Can see through and estimate features behind most smaller obstructions and some larger obstructions, e.g. bicycles, bushes, lamp posts, tree branches with a low amount of leaves, and tree stems. - Can sometimes present false positives behind obstructions
General	+ Identified feature areas are neat - System is very sensitive to where exactly the user selects the feature, which can lead to high variance in results	+ No variance in results and handles photos taken at different angles to the houses better - Noisy features, often with broken shapes (can be partially solved by post-processing)

Table 6.1: General findings from our manual inspection of the results.

is accepted and what the preferred result is. A segmentation is considered 'accepted' when the result is accurate enough to be used when making a quotation. From this small scale evaluation we see that the results from TrumPy and OBAMA are equally much preferred, though TrumPy has one more accepted case. Due to the small sample size, however, it is not possible to draw any conclusions on which system is ultimately better. From this evaluation, it can be concluded that the accuracy of TrumPy and OBAMA is very similar, with each having their own strengths and weaknesses.

Figure 6.1: Results achieved by OBAMA and TrumPy

6.2. Requirement evaluation

Following from our research, a number of requirements for our final product were set, as defined in section 2.2. This section evaluates each of these requirements to determine whether they were fulfilled. It starts with the 'must have' requirements in section 6.2.1, which were deemed necessary for the project to be successful. Then, the important – though not vital – 'should have' requirements are evaluated in section 6.2.2. Finally, the 'could have' requirements are evaluated in section 6.2.3.

6.2.1. Must have

The requirements defined as 'must have' are functionalities that were deemed necessary for the system to contain. As the MoSCoW model states, it is impossible to deliver a viable solution without an implementation of these requirements [8]. In the requirements two 'must have' functionalities were defined:

- Automatic detection of windows, doors and walls.
- Calculation of surface areas after user inputs scale reference.

Both these requirements are met as the system can successfully recognise windows, doors and walls in a picture, within a certain degree of accuracy. The user is also able to specify a scale reference, which is used to calculate the surface areas of the different features, thus the system fulfils the must-have requirements.

6.2.2. Should have

The requirements defined as 'should have' are important but not vital. The MoSCoW model states that while these functionalities are important and it may be painful to leave these out, the solution is still viable [8]. The requirements contained the following three 'should have' functionalities:

- Possibility to add user's data to the training dataset.
- Ability to determine whether picture is actually suitable for analysis by means of a confidence ratio.
- Obstacle detection to correctly identify features and their areas even when partially obstructed but the area still remains a single surface.

One of these functionalities has currently been implemented, namely the obstacle detection. The system is able to ignore obstacles that partially obstruct a feature, such as trees, traffic signs and lamp posts.

Regarding the requirement of adding a confidence ratio, this was dropped in consultation with the company. With the research we had done at the time, it seemed that implementing this feature would be rather simple, as the classification would practically assign a set of numbers to each pixel, which would represent the chance of that pixel belonging to the specific classes. When these chances are too evenly distributed, then the neural network is not sure which class the pixel belongs to. If this happens with too many pixels, then the image could be deemed unfit for analysis. However, this cannot be done with the pix2pix implementation that we used, because it produces images instead of probability vectors. Instead, guidelines for what pictures are suitable should be made, and it is assumed that the user is competent enough to judge the result and will try again if it is not satisfactory. The last requirement is the possibility to add the user's data to the training dataset. This would entail that the user can tell the system whether the resulting segmentation is accurate enough for it to be added to the training dataset and trained upon. This feature is currently not implemented due to time constraints, but the company could easily do this themselves.

6.2.3. Could have

The requirements defined as 'could have' are desirable, but less important than the 'must have' and 'should have' requirements. These requirements are only met in the best case scenario [8]. The requirements contain the following three 'could have' functionalities:

- Material Recognition to aid feature detection and classification.

- Automatic inference of image scale (e.g. using openly available geolocation data).
- Obstacle detection to correctly identify features and their areas even when the feature is split by the obstacle.

The ‘could have’ requirements should actually not be implemented as long as the ‘should have’ requirements are not fulfilled. However, we still did fulfil the requirement for advanced obstacle detection, as it was included in the implementation of the ‘should have’ obstacle detection. Due to time constraints and prioritisation of the ’must have’ and ’should have’ requirements, the other two ‘could have’ requirements have not been implemented.

6.3. Design goals evaluation

At the start of the project, a set of design goals was drafted in order to guide the design of our product, which can be found in section 3.1. This section evaluates these design goals to see whether they have been implemented in the system. First, in section 6.3.1, the design goals for performance are evaluated. Section 6.3.2 then discuss the accuracy design goal, whereas section 6.3.3 discusses the maintainability design goal. Finally, section 6.3.4 evaluates the two other design goals that were set: a user interface and scalability.

6.3.1. Performance

The first performance design goal prescribed that the detection of windows, doors and walls should take at most 30 seconds. This design goal is easily fulfilled, as it takes only a matter of seconds for the detection to occur. The network takes on average 4.15 seconds to parse an image on our development machine. The total processing time for a request to the API server was on average 7.25 seconds. This performance is achieved using an NVidia geforce 1060 GTX mini as the GPU. If no GPU were to be used, we estimate the training of our network to last around two weeks as opposed to the two hours that we need now. For the processing of a user image we expect a processing time of 30-60 seconds. This is higher than our set threshold and leads to impractical training duration. We therefore highly recommend use of a GPU.

The second performance design goal stated that the system should be able to handle partial obstructions. This goal has been fulfilled as well, as described in sections 6.1 and 6.2.2.

6.3.2. Accuracy

As explained in section 3.1.2, we cannot use a metric such as intersection over union or percentage of pixels. While it could be used to give a quantified comparison between our results and OBAMA’s, we cannot automate this process as OBAMA requires the user to click every feature in every picture. Then there is also a lot of variance in where exactly the feature is clicked when operating OBAMA. Slight deviations can have big differences in results, resulting in different scores every run. Furthermore, OBAMA also labels the window frames the same colour as the windows, which makes it impossible to distinguish them from each other in automated testing. Therefore, we had to resort to manual inspection of the results.

As shown in section 6.1, our system performs on par with OBAMA, despite our limited set of training data. The accuracy design goal stated that we aimed for our system to be more accurate than OBAMA is, especially in regard to special situations, where parts of the house are obstructed. As explained in section 6.1, our solution is much more flexible and is able to perform better than OBAMA in such special situations. The accuracy should only increase when more training data is added to our dataset and trained on. Therefore, the design goal of accuracy can be considered fulfilled.

6.3.3. Maintainability

The third design goal stated that the system should be well maintainable. This design goal has been fulfilled as every function has been documented accordingly and an API has been implemented to ensure compatibility with other systems. The two main components of the system, TrumPy and PutIn, are able to run separately while continuing communication through an API. This makes it easy to update or even replace either of them, without breaking the other as long as they communicate through the same API. Also, a document on the software architecture of the system is included in this report in section 4.2.

6.3.4. Other

The first design goal in this section was that the system needs to have some form of a user interface. As can be seen in figure 4.6 in section 4.4, the system has a simple yet functional and good looking user interface and therefore this design goal has definitely been achieved.

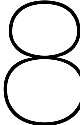
The final design goal regards scalability. The web server (PutIn) and neural network system (TrumPy) are two separate pieces of software communicating via an API. This, combined with the fact that TrumPy is run inside a Docker container, allows multiple instances of TrumPy to be run on different machines when needed. The web server could then act as a load balancer for each instance. If necessary, the web server itself could be deployed on multiple machines as well, with each machine managing its own cluster of TrumPy instances, thus allowing for full scalability.

7 Conclusion

The goal of this project was to create a system that is able to detect certain features in a frontal picture of a house; namely windows, walls and doors. The system also had to be able to calculate the surface areas of these features and display these in a user interface. Furthermore, the system should be able to handle (partial) obstructions and perform faster than OBAMA, the system that De Energiebespaarders already had.

Following from the problem description and the requirements set up in chapter 2, and the design goals as described in section 3.1, we created an initial design. The decision was made to replace the system that the company already had, instead of extending it, as explained in section 3.2. A total of three designs have been proposed, segmentation and classification as described in section 3.3, dilation as explained in 3.4.2 and finally settling on pix2pix which can be found in section 3.4.3. Pix2pix delivered good results, as shown in 6.1, and thereby became the preferred solution. Advantages include fast training and not needing to manually optimise a loss function. The pix2pix technique was adopted into our final design and implemented together with the surrounding architecture as described in chapter 4.

The implementation has been tested as described in chapter 5 and fulfils all ’must have’ requirements, one ’should have’ requirement and one ’could have’ requirement, following from the evaluation of the requirements in section 6.2. On the other hand, all design goals have successfully been implemented, as shown in section 6.3. While more training data would see the system perform even better, the accuracy of our product is in most cases similar to that of the company’s current system. The system has also shown that it is more flexible than OBAMA, seeing as it is much better at handling (partial) obstructions. In combination with the performance, our product has considerable benefits over the old system. Nonetheless, the calculated surface areas are still just an estimation due to the fact that differences in depth are not taken into account. To make it more accurate, it would be necessary to find ways to record the depth in the image. Still, De Energiebespaarders will take our system into production, so in conclusion, the project can definitely be considered a success.



Recommendations

This chapter provides a number of recommendations that can be applied to the system in order to improve its performance and accuracy. These recommendation range from improving the training data, as discussed in section 8.1, to splitting the network into multiple specialist networks, as discussed in section 8.3. Another recommendation is to use a secondary neural network to distinguish windows from their frames. This idea is elaborated in section 8.4.

8.1. Training data

The first and most important recommendation is to gather more training data, especially training data that is representative of houses in the Netherlands. Doing so should vastly improve the accuracy of the results produced by the neural network. An easy way to obtain more training data would be to allow the user to rate the produced result. Results with high ratings can then be added to the training set and trained upon, while results with low ratings can be added to the test set used during the training, provided that the low rating is not caused by user errors. In the next training run of the network, the highly rated results will aid the training of the network, and the poorly rated results can be inspected afterwards to see if the accuracy of the system has improved. It is recommended to manually inspect the rated results before they are actually used for training, to verify that the given ratings were indeed correct. Otherwise the network may actually mistakenly be learning its own faults, instead of improving on them.

Another way in which more training data can be gathered, is to create a module in the user interface that provides an easy and fast way of manually labelling training data. Such an interface could also be used in order to edit already labelled data, whether this was done manually, or by the system itself. In combination with rating the accuracy of the system's results, this could allow the company to improve the labelling of the rated results, before they are trained upon, which should increase the overall accuracy of the system.

Aside from simply adding more data, the training data could also be modified and extended so the network can receive better training. These separate recommendations are discussed in section sections 8.1.1 and 8.1.2.

8.1.1. Colours

The current labelling colours were chosen out of necessity, because our dataset was too small to use the colours we initially intended. We started out using red for the windows, blue for the walls and green for the doors. However, the very limited number of doors in the CMP dataset and the small size of our own dataset caused the network to fail to recognise doors. This was overcome by using purple for the door labels, because then the recognition of doors could be seeded by the edges of window and wall labels. This solution works because purple is in between blue and red in colour space, and therefore boosted the networks performance on doors. Sadly, this solution causes another problem for exactly the same reason: the network often has trouble producing sharp and correct the edges between adjacent doors and windows, though this is less of a problem than not recognising doors at all. As the dataset grows larger, and enough doors are present in it, we recommend switching back to green labels for doors, so all of the label colours are perpendicular in colour space.

A solution to this problem could also be to drop the L1 norm, which would mean the distance and

orientation of the labelling colours within the colour space is not relevant any more. However, as can be seen in the top-left corner of figure 4.4 in chapter 4, this introduces artefacts in the image. In this particular case, the weight of the L1 loss is set to 10, so setting the weight of the L1 loss to zero would probably produce even worse results. Unfortunately, it is therefore not possible to drop the L1 norm.

8.1.2. Roofs

The current dataset contains labels for windows, walls, doors and background. We recommend adding labels for roofs as well. Allowing the network to learn roofs should help the network to distinguish the sky from the windows, as the network sometimes has the tendency to label parts of the sky as windows. By adding roof labels, the network can learn that windows should always be surrounded by roofs or walls.

8.2. Image rectification

With the current method for surface area calculation, the system would give a more accurate estimation if the input images would be rectified. What this means is to change the perspective of the picture so that it is a perfectly frontal picture of the house in question. This would remove part of the depth differences in the individual features and between the different features, resulting in a more relevant input scale and therefore more accurate surface areas. This feature also makes feature shapes more regular, which can help segmentation accuracy as well. Accompanying this feature would be to also train the network on rectified images, but it is not mandatory for the whole dataset to be rectified.

The old system did have a way to do this which already gave back decent results. It sometimes gives weird results however, so we did not want to integrate it with our system before having an error check in the user interface, as we would like to automatise it. The port from Python 2 to Python 3 did not give much trouble, so integrating it would be a good and not too complicated addition.

If the results are still not satisfactory, there are also preprocessing methods used in [7], including rectification. This can be found online, but is written in MATLAB with their Computer Vision System Toolbox, which requires a license. For use outside of MATLAB it would need to be ported over manually between MATLAB and Python, which will require some work.

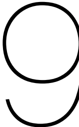
8.3. Specialist networks

As seen in figure 4.3 in chapter 4, the similarity between the data the network has trained on and its input has a major impact on the performance. The system could potentially become significantly more accurate by splitting it up into multiple specialist networks. Each network can then be trained on a specific type of house. Another neural network could decide which of the networks may best be able to label this specific house in the picture, or the user could be prompted to select one of a small number of pictures to describe their house.

It may also be possible to simply add an extra label to the training data that specifies what type of house the picture contains, instead of having this done through specialist networks. The neural network could then learn the different characteristics of different types of houses with these extra labels, while still being able to learn the definition of windows, walls and doors from the entire training dataset, instead of just a subset. However, both methods do require a significant amount of extra training data, and it is unknown which of these methods may perform better.

8.4. Window frame detection

Finally, one more way to improve the system, more specifically the window detection, is to use another neural network trained to distinguish a window from its frame, as our system currently does not take the window frames into account when detecting windows and calculating their surface areas. If the window frames can be deducted from the surface areas, then the estimation of the window surface area would become more accurate. This cannot accurately be detected by our current network, because the images are scaled down to 256x256 pixels in order to fit into the limited memory of the graphics card. At this resolution, the information is just too compressed to accurately distinguish such fine features. The solution would be to run the main network, extract the windows from the original image, and feed those in the original resolution of the image to the other network, which may then provide a more detailed definition of the window.



Discussion

This chapter provides a discussion on two aspects of the project. First of all, the subject of training data for the neural network and the implications regarding this are discussed in section 9.1. Section 9.2 then provides a discussion on the ethical aspects of our product.

9.1. Training data

Training data is of huge importance when making use of neural networks, as better and more training data will result in more accurate results. During the course of the project, we have started collecting data for our own dataset that was called the 'Trump Standard'. This was necessary as there were no datasets available that were representative of facades in the Netherlands. The labelling of our own dataset took a large amount of time, but with the help of the company we were able to label 192 images. However, this is still a relatively small dataset, and with more (and also accurate) training data the accuracy would definitely improve.

9.2. Ethics

Neural networks often raise concerns regarding ethics. One of these concerns is that neural networks may learn to discriminate on for example, race or sex, when it is trained with biased data. However, our data does not contain any sort of attributes that can be used to discriminate in such a way.

A second concern with neural networks is when it needs to take moral decisions, as it is a moral obligation for these decisions to be taken by humans and not machines. However, calculating the surface areas of doors, windows and walls is definitely not a moral decision. The only concern this may raise, comes from the fact that De Energiebespaarders use these calculations in order to give their clients a preliminary quotation. Therefore, the accuracy of the results of the neural network does influence what their clients will have to pay and what the installers will receive. If the system predicts higher surface areas on average, then this could mean that the client is overpaying. The other way around would mean that the installer is underpaid for the work that he delivers.

This brings us to another concern with neural networks, which is that it is often not clear who is responsible for the outcomes of a neural network. In our case the company is responsible for the produced results, as the results are checked manually and the quotation is also composed manually before being sent to the client. As the results produced by our network are merely an estimation and not in any way binding decisions, there are virtually no concerns regarding the ethics of this project.

Bibliography

[1] Deeplearning4j: Open-souce, Distributed Deep Learning for the JVM. URL https://deeplearning4j.org/.

[2] gRPC open-source universal RPC framework. URL http://www.grpc.io/.

[3] Jest - Delightful JavaScript Testing. URL https://facebook.github.io/jest/.

[4] Koa - next generation framework for node.js. URL http://koajs.com/.

[5] Material-UI. URL http://www.material-ui.com.

[6] unittest - unit testing framework for python. URL https://docs.python.org/3/library/unittest.html.

[7] Lama Affara, Liangliang Nan, Bernard Ghanem, and Peter Wonka. Large scale asset extraction for urban images. In *ECCV (3)*, pages 437–452, 2016.

[8] Agile Business Consortium. MoSCoW Prioritisation, 2014. URL https://www.agilebusiness.org/content/moscow-prioritisation.

[9] Sean Bell, Paul Upchurch, Noah Snavely, and Kavita Bala. Opensurfaces: A richly annotated catalog of surface appearance. *ACM Transactions on Graphics (TOG)*, 32(4):111, 2013.

[10] Sean Bell, Paul Upchurch, Noah Snavely, and Kavita Bala. Material recognition in the wild with the materials in context database. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3479–3487, 2015.

[11] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L Yuille. Semantic Image Segmentation with Deep Convolutional Nets and Fully Connected CRFs. *Iclr*, pages 1–14, 2014. ISSN 0162-8828. doi: 10.1109/TPAMI.2017.2699184.

[12] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The Cityscapes Dataset for Semantic Urban Scene Understanding. apr 2016. URL http://arxiv.org/abs/1604.01685.

[13] Pedro F Felzenszwalb and Daniel P Huttenlocher. Efficient Graph-Based Image Segmentation. *International Journal of Computer Vision*, 59(2):167–181, 2004.

[14] Benjamin Graham. Fractional Max-Pooling. dec 2014. URL http://arxiv.org/abs/1412.6071.

[15] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. Image-to-image translation with conditional adversarial networks. *arxiv*, 2016.

[16] Filip Korč and Wolfgang Förstner. eTRIMS Image Database for Interpreting Images of Man-Made Scenes. 2009. URL http://www.ipb.uni-bonn.de/projects/etrims.

[17] R.C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Alan Apt series. Pearson Education, 2003. ISBN 9780135974445. URL https://books.google.nl/books?id=0HYhAQAAIAAJ.

[18] Radim Šára Radim Tyleček. Spatial pattern templates for recognition of objects with regular structure. In *Proc. GCPR*, Saarbrucken, Germany, 2013.

[19] Hayko Riemenschneider, Ulrich Krispel, Wolfgang Thaller, Michael Donoser, Sven Havemann, Dieter Fellner, and Horst Bischof. Irregular lattices for complex shape grammar facade parsing. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 1640–1647, 2012. ISBN 9781467312264. doi: 10.1109/CVPR.2012.6247857. URL https://pdfs.semanticscholar.org/a947/0f9cf832b1cf385e4ccc424b1fd6051f5aa8.pdf.

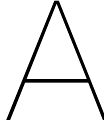
[20] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation. URL http://lmb.informatik.uni-freiburg.de/http://lmb.informatik.uni-freiburg.de/people/ronneber/u-net.

[21] K Schwaber and Jeff Sutherland. The Scrum Guide - The Definitive Guide to Scrum: The Rules of the Game. 2011. URL http://www.scrumguides.org/docs/scrumguide/v2016/2016-Scrum-Guide-US.pdf{#}zoom=100.

[22] Evan Shelhamer, Jonathan Long, and Trevor Darrell. Fully Convolutional Networks for Semantic Segmentation. URL https://arxiv.org/pdf/1605.06211.pdf.

[23] Fisher Yu and Vladlen Koltun. Multi-Scale Context Aggregation by Dilated Convolutions. nov 2015. URL http://arxiv.org/abs/1511.07122.

[24] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A. Efros. Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks. mar 2017. URL http://arxiv.org/abs/1703.10593.



Software Improvement Group feedback

A.1. First feedback

De code van het systeem scoort bijna 4 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code bovengemiddeld onderhoudbaar is. De hoogste score is niet behaald door een lagere score voor Unit Size.

Voor Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, te testen en daardoor eenvoudiger te onderhouden wordt.

In jullie project valt het op dat de Python-code hier over het algemeen beter scoort dan de JavaScript-code. Dit komt voornamelijk door het gebruik van callbacks, zoals in de aanroep naar router.post in app.js. Dit is een typisch probleem in JavaScript, op het moment dat je te veel logica in de callback plaatst wordt de code al snel moeilijk leesbaar. In dit soort gevallen kun je het afhandelen van het resultaat van de aanroep beter in een aparte functie plaatsen. Naast de leesbaarheid komt dit ook de testbaarheid van de code ten goede.

Als laatste nog de opmerking dat we voor de Python-code wel testcode hebben aangetroffen, maar voor de JavaScript-code niet. Ook hier zien we dus een verschil in kwaliteit tussen de gebruikte technologieën. Hopelijk lukt het nog om dit tijdens het vervolg van het project wat meer gelijk te trekken. Over het algemeen scoort de code dus bovengemiddeld, maar dit is dus voornamelijk te danken aan de Python.

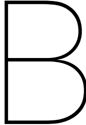
A.2. Second feedback

In de hermeting zien we dat het codevolume sterk is gegroeid. In vergelijking met de eerste upload zit de groei in zowel de Python-code als in de JavaScript-code. De score voor onderhoudbaarheid is in vergelijking met de eerste upload ongeveer gelijk gebleven.

Ondanks dat de totaalscore gelijk is gebleven zien we Unit Size wel degelijk verbetering in de aspecten die we in de feedback op de eerste upload hebben genoemd. Die verbeteringen zijn echter weer grotendeels teniet gedaan door het introduceren van nieuwe lange methodes in de nieuwe code. Het is belangrijk om dit soort verbeteringen structureel te doen, dus ook voor nieuwe code en niet alleen op het moment dat je aan het opruimen/refactoren bent.

Wat betreft het testen hebben jullie advies opgevolgd en nu ook unit tests voor de JavaScript code geschreven. Daarnaast is het goed om te zien dat jullie daarnaast ook de hoeveelheid testcode voor Python verder hebben vergroot.

Uit deze observaties kunnen we concluderen dat onze aanbevelingen grotendeels zijn meegenomen.



Original project description

We use Data + Algorithms to analyse homes remotely, give advice and sell products. We offer turnkey solutions (product + installation services) for insulation, boilers, glass to solar panels.

Context of the problem

Amongst others, our dev team has developed an Image Recognition Algorithm that is able to measure surface areas (e.g. of facades, windows, doors) via photos (O.B.A.M.A.) This enables us to provide homeowners with an energy report and remote quotes without the need for an installer to physically visit the house. Much more sustainable! Besides reducing costs and time for both installer and homeowner.

Description of the challenge

Next up, is the challenge to add Machine learning to our Algorithm.

We have access to a large data-set of images. All of these images are of a single facade from houses here in The Netherlands. All contain a mix of similar features, windows, doors, wall, roof and sky. Some also contain trees, bicycles, cars or other unwanted features. Our algorithms find these features and classify them, however, further work is required to categorise the features correctly so that the most accurate depiction of the house can be determined. This information is then used to calculate the real world physical sizes of the features which are input into our energy calculations.

Your tasks will likely involve:

- Building machine learning algorithms into the current software
- Deciding on the best features to track in the images
- Tuning the algorithms and running tests on new datasets
- Providing input into discussions on best practices
- Helping to define what a good result looks like

C
Datasets

Training is an essential part of developing artificially intelligent applications. Table C.1 provides a tabular summary of the available datasets. The size is the number of pictures and the labels show which features are labelled within the dataset. The following datasets are pictures of facades with different sets of labels. Most datasets consist of facades of large capital city buildings such as the ones found in CMP. Some datasets are more varied, such as eTRIMS.

Dataset	Size	Labels	Comments
CMP	328 + 228	facade, molding, cornice, pillar, window, door, sill, blind, balcony, shop, deco, background	
eTRIMS [16]	60	building, pavement/road, sky, vegetation	
ZuBuD	1005	none	taken in Zurich, 5 images per building, includes obfuscations and different seasons
Graz50 [19]	50	wall, door, window, sky	
ECP	109	Window, Wall, Balcony, Door, Roof, Chimney, Sky, Shop, Outlier	taken in major cities around the world.
VarCity	428	window, wall, balcony, door, roof, sky, shop	Includes 2D and 3D versions
LabelMe Facade	945	various, building, car, door, pavement, road, sky, vegetation, window	subset of LabelMe, only for non-commercial and research experiments

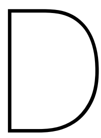
Table C.1: Summary of available datasets of facade images for machine learning. The size is the number of available images. Note that some datasets contain more than one image per facade.

Material datasets

Possible training for material recognition requires labels for the different materials and therefore other datasets. Table C.2 presents a number of important datasets. FMD and OpenSurfaces present images of objects, which are labelled with a material. Materials In Context (MINC) presents environmental images in which more objects are labelled by material.

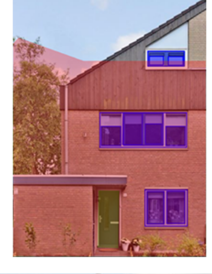
Dataset	Size	Labels	Comments
FMD	1000	fabric, foliage, glass, leather, metal, paper, plastic, stone, water, wood	100 samples per category
OpenSurfaces [9]	105,000	wood, painted, fabric/cloth, metal, glass, tile, carpet/rug, plastic - opaque, granite/marble, ceramic, food, paper/tissue, leather, plastic - clear, brick, concrete, laminate, wallpaper, hair, mirror, cardboard, stone, skin, wicker, fur, rubber/latex, foliage, wax, linoleum, sponge, sky, water, chalkboard/blackboard, cork/corkboard, fire, styrofoam	
MINC [10]	7061	brick, carpet, ceramic, fabric, foliage, food, glass, hair, leather, metal, mirror, other, painted, paper, plastic, polished stone, skin, sky, stone, tile, wallpaper, water, wood	number of images is 7061, but it does have 3mln point samples and is growing

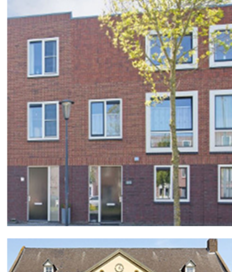
Table C.2: Summary of available datasets of material recognition for machine learning. Again, the size is the number of images in the dataset.



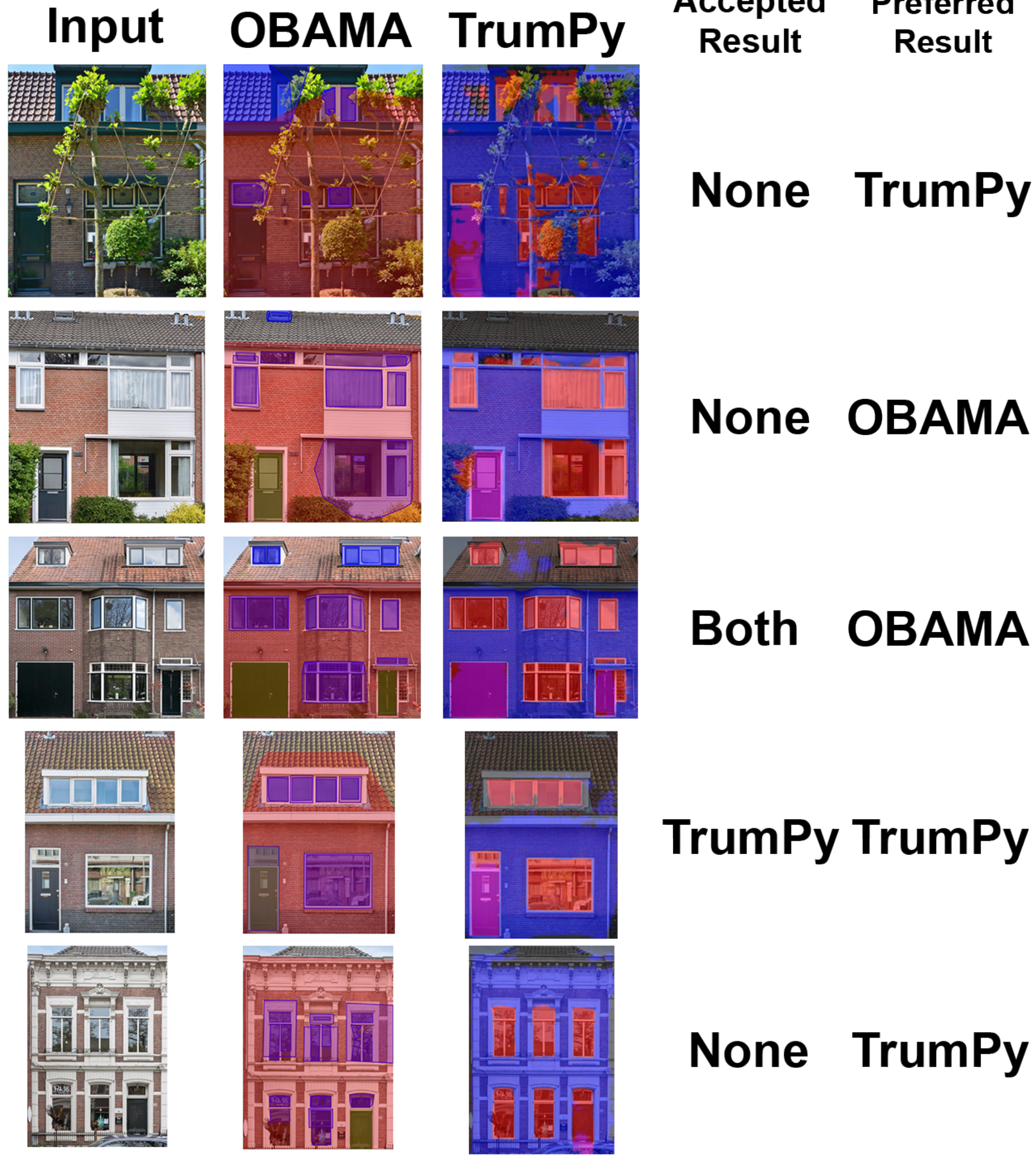
Extra results

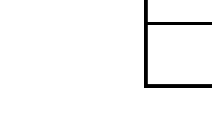
Input | OBAMA | TrumPy | Accepted Result | Preferred Result





None TrumPy


Input
OBAMA
TrumPy
Accepted Result
Preferred Result
None
TrumPy
None
OBAMA
Both
OBAMA
TrumPy
TrumPy
None
TrumPy



Infosheet

Title: Facade labelling using neural networks
Client: De Energiebespaarders
Presentation date: June 28, 2017
Description:
De Energiebespaarders is a company that helps home owners make their homes more energy efficient. In order to estimate the costs and benefits of installing energy saving measures, a client can upload a picture of their house which is then analysed to determine the surface areas of windows, doors and walls. Using this result, De Energiebespaarders are able to provide the client with a quotation, without the need for an installer to visit. There was already a system in place to perform this task, but it was slow, inconsistent and heavily dependent on user input. The aim of this project was to create a better system using machine learning techniques.

During the research phase it was decided to first use classic segmentation techniques and then classify these segmentations using a neural network. However, it turned out that the results achieved by the segmentation were not sufficient, so it was decided that both the segmentation and classification would be done using a conditional generative adversarial neural network.

Our solution currently performs faster than their previous system, though with similar accuracy. Another advantage of our system is its flexibility as it is able to handle obstructions like trees and cars. It should be noted that with more training data for the neural network, the accuracy of the results can improve vastly. The company has therefore been recommended to increase the amount of training data when they take the system in production.

The final report for this project can be found at: `http://repository.tudelft.nl`

Team members

Name	**Role and contributions**	**E-mail address**
Thomas Kolenbrander	User interface, Image processing	t.j.kolenbrander@student.tudelft.nl
Bart van Oort	Server communication, User interface, General management	b.vanoort@student.tudelft.nl
Frank de Ruiter	Neural network, User interface, Back-end	moghammed@gmail.com
Tim Yue	Image processing, Testing	tim_yue@hotmail.com

Client contacts

Name	**Affiliation**
ir. M. Zaal	De Energiebespaarders
ir. J. Broek	De Energiebespaarders

TU Delft coaches

Name	**Department**	**Group**
dr. J. van Gemert	Computer vision	Pattern Recognition and Bioinformatics
dr. S. Khademi	Computer vision	Pattern Recognition and Bioinformatics