DELFT UNIVERSITY OF TECHNOLOGY

Artificial Neural Networks

PROJECT REPORT

Authors:
Michiel Bongaerts
Marjolein Nanninga
Tung Phan
Maniek Santokhi

June 16, 2015

Contents

1. Introduction

Mapping the world around us has always been a human endeavour to advance economical output. A better understanding of the places around us makes for more efficient travelling and exploitation of the land. However, it has always been a very slow and tedious process to produce these maps, something technology has not changed just yet.

A new opportunity has arisen with the arrival of satellite imagery and an ever increasing amount of computational power. An opportunity where this mapping can be done automatically so that this tedious and slow job can be processed even more quickly and perhaps more accurately. It is with this in mind we further analyse any possibilities.

This paper proposes an update. Now a more straightforward approach has been chosen in which the emphasis lies on the actual Neural Network rather than the conversion of interpreted images to vector graphic maps. The new approach deals with image patches rather than pixels. This document discusses the newly acquired concept with a list of features and the actual implementation details. Also an updated schedule will be presented.

2. Concept

Earlier attempts to conceptualise the idea to automate map making resulted in a proposal that too heavily focussed on the actual map creation rather than the classification. For a Neural Network course this was deemed not befitting enough. The plan was also quite far reaching to start with. Thoughts were put into downscaling this ambitious plan. We played around a bit and came up with a new concept which will be discussed in this chapter. Firstly, an impression is given how the end user interacts with our system. This will lay the groundwork for how the Neural Network will be constructed. Secondly through the principles of MoSCoW a flexible requirements list is established. Actual talk about the classification is done in the subsequent chapter.

2.1 Impression

Below in figure 2.1 an impression is given of what the end user will interact with and a possible result that might come about from said interaction.

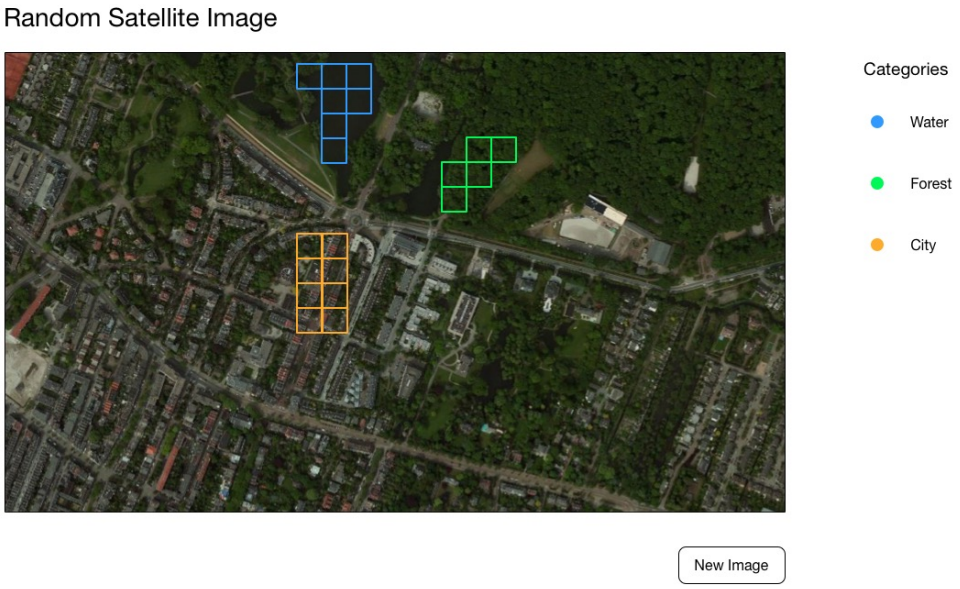

Figure 2.1: Impression of what the user interacts with and a possible outcome.

The most notable attention grabber is the satellite image. This image will be acquired through a random call from a homogenous satellite image database on every new instantiation of the system. Another possibility to acquire new data is by clicking on the 'New image' button. Right of the satellite image one can see labels (Forest, City, Water) which correspond with the labels which are outputs of the classification. Results obtained from our algorithm, given the current satellite image as input, are graphically feed back to the user. The impression above does that by showing a correspondence between image patches and their respective label via color coding laid over the satellite image. This rendition just shows a few islands of resulsts as an example. Normally the entire image will have such arching (which will be a lot more subtle).

Figure 2.2 shows how it works internally. Two grids are maintained. One with patches the size of 25 by 25 pixels. The other by the size of 50 by 50 pixels. The latter is actually fed to the Neural Network from which will be decided for that patch the percentage of type of label it contains. Four 50 by 50 pixels will be grouped together to create a square. In the middle the 25 by 25 pixels patch is placed. This patch will eventually be coloured on the satellite image to indicate the type of label. To decide that, a weighted majority vote of the four 50 by 50 pixels surrounding that smaller patch is computed. Like a convolution filter this construction is shifted over the satellite image as to decide for every area on it. As one can imagine, 25

pixels at the borders are omitted.

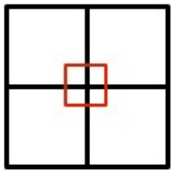

Figure 2.2: Grid structure which facilitates the algorithm.

2.2 MoSCoW

Since a limited amount of time is available and we thought of quite some experiments and features that could be added, we used the MoSCoW method to get our priorities straight and focus on the most important requirements. MoSCow stands for *Must have*, *Should have*, *Could have* and *Would have*. All the requirements are labeled in these four classes.

2.2.1 Must have

Must have requirements are critical to project success.

- Create code based on a *Convolutional Neural Network* (CNN) that enables automatic classification of patches from satellite images. Considering images acquired on provincial level and a substantial amount of pixels in one patch (at least 50 x 50 pixels).
- At least the following classes should be recognized: vegetation, city and water.
- Develop a way to visualize the automatic classifications clearly.

2.2.2 Should have

Requirements labeled as should have are important to book success, but not necessary for delivery.

- Create a clear interface in which the unlabeled images can be uploaded, and the output consists of labeled images.
- Calculate the uncertainty in the classification and ask user input for very uncertain patches.

2.2.3 Could have

It would be very nice if we would be able to reach the Could have feautures, but they are not critical.

- Experiment with pre-processed images (noise reduction, gradient calculations)
- Analyze images on city level, so with more details present. For this purpose new classes have to be added, such as roadways, cycle paths, buildings, distinct vegetations etc.
- Experiment with other models than the state-of-the art LeNet-1 CNN. For examples, a CNN in which Genetic Algorithms are incorporated, or implementing an Extreme Learning Machine for the training of the weights.

2.2.4 Would have

These requirements are implemented only in the most ideal situation. They are considered as the dream project, sometimes serving as a suggestion for further projects.

- Develop a method for high-detailed automatic vector graphics, in which segmentation of the distinct labeled classes is incorporated.
- Use the input of the users to improve the automatic classification.
- Sell the software package to Google.

3. Implementation

A plan has been established what kind of application should come about. The previous discussion pressed for a certain kind of structure. One where a Neural Network is central to the problem to be solved. But also a frontend is needed for certain user interaction as well as an infrastructure in the backend which facilitates the communication with said frontend. This chapter discusses these aspects.

3.1 Frontend and Backend

The impression given above in figure 2.1 is close to what the end result should look like (although it only displays a certain state). Yet an entire infrastructure outside of the Neural Network is needed to facilitate the application. Below in figure 3.1 an infrastructure is visualised via the communication of the two entities.

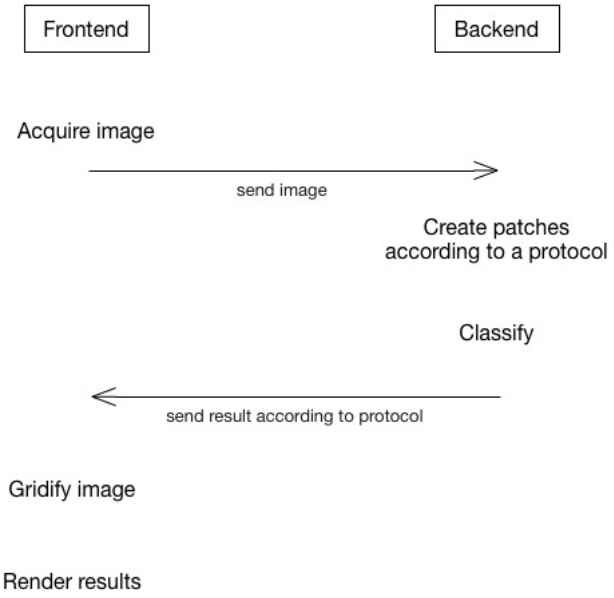

Figure 3.1: Communication visualization between frontend and backend.

First, at the frontend, an image is acquired by the user from a database call. The image is immediately send to the backend. There, patches are extracted according to a certain protocol (specific size and sequence). The patches are classified by our Neural Network. Send back to the frontend are the results in a list according to the sequence defined by the protocol. Also a codification of the results is part of the protocol. Back at the frontend the image is made into a grid which corresponds to the size of the protocol. According to the results of the Neural Network these grid windows are coloured.

3.2 Dataset

The image which is being acquired every time comes from the Bing Maps database. A predefined size of the map with satellite view enabled, the labels turned off, no other controls or logos visible and where the hight is 100 meters is put in the frontend. Randomly a location inside the Netherlands is generated. Now the actual image is taken from the map.

For training, satellite images from the same source and practice mentioned above are drawn on a per category/label basis. Patches are acquired code-wise over those images so that for each patch it is known what category/label belongs to it.

4. Convolutional Neural Network

During our research phase, we encountered a variety of papers that describes the use of image classification based on Neural Networks. Most of these papers have one thing in common which is the use of Convolutional Neural Networks (CNN). Since convolution operations are widely used to extract features from images and since these operations can be represented in terms of a Neural Network these properties lead to the existence of the Convolutional Neural Networks.

In general a CNN architecture consist of multiple Convolution layers and sub-sampling layers. It depends on the architecture how these layers are followed by each other. The convolution layer is the layer which results after the convolution operator is performed by the a convolution kernel. Since the gaol is to extract general features from our input image we want the CNN to generalize. This generalization is partly released by dimension reduction. Since convolution operation reduce the dimension of our input map with $\frac{M-N+1}{M}$, where N is the dimension of the convolution kernel and M the dimension of the input map, we want the dimension to be reduced more quickly. This is done by dub-sampling kernels which in general 'squeeze' or average the input map with a certain dimension. This operation is equivalent to a convolution operation but with a larger step-size (convolution uses a stepsize equal to 1) and equal weights for each element in the sub-sampling kernel.

4.0.1 LeNet-1

The model we implement is LeCun's LeNet-1. This model uses an alternating sequence of convolution and sub-sampling layers. The architecture of this network is shown in Figure 4.1. The convolution layers act as feauture maps, they consist of a window of a certain size, whose pixels are trainable weights. The sub-sampling layers reduce the dimensionality of the outputs.

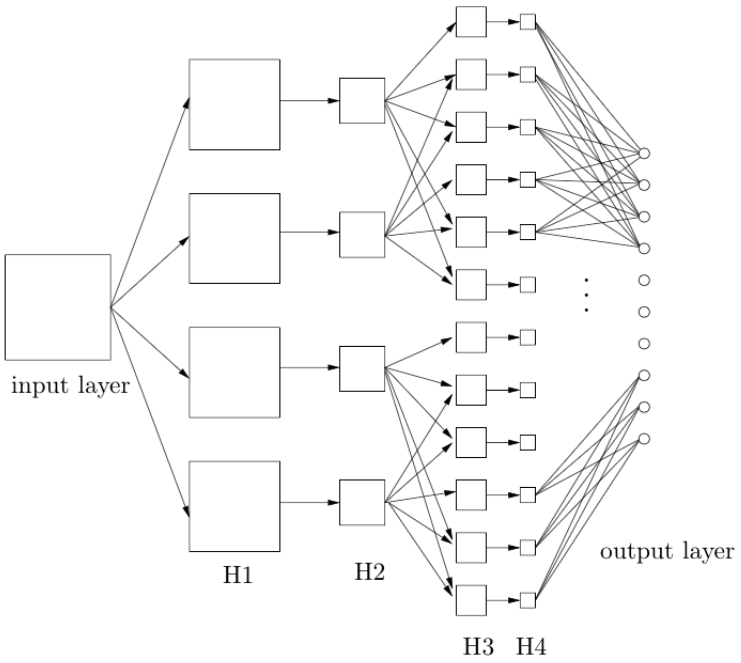

Figure 4.1: The architecture of the CNN proposed by LeNet. H1 and H3 are the convolution layers, H2 and H4 the sub-sampling layers to reduce the dimension. Source:

One of the biggest advantages of using CNN, and especially LeCun's LeNet-1 implementation, is the incorporation of back-propagation learning. Meaning that all the weights of the layers are adjusted iteratively,

eliminating the need to manually create the convolution masks.

4.0.2 Back-propagation

To train the network we have to adjust several parameters in the network. In this research we consider back-propagation from layer output to H4 to H3. In the whole network we use the Sigmoidal activation functions which is given by:

$$f(x) = \frac{1}{1+e^{-x}} \tag{4.1}$$

The error function is defined as:

$$E = \sum_k \frac{1}{2}|t_k - F_k|^2 \tag{4.2}$$

Where t_k represent the labelled class of the training. This variable is 1 if the corresponding class is trained and 0 if this is not the case. Below the functions are listed in which we go from the activation of our classifier neuron to value obtained in layer H3. In these equations r stands for row which is the first branch after the first convolution i.e. the total rows is equal to the amount of different feature maps in H1. b stands for the amount of branches per row. k stands for the amount of classes which are trained by the network.

$$\begin{aligned}
&F_k = f(x_k) \\
&x_k = \sum_{ij} W^{4rbk}[i,j]\ H^{4rb}[i,j] - b_k \\
&H^{4rb}[i,j] = \sum_{u,v} W^{3rb}[u,v]\ H^{3rb}[2i+u, 2j+v] \\
&H^{3rb}[2i+u, 2j+v] = f\left(x^{3rb}[2i+u, 2j+v]\right) \\
&x^{3rb}[2i+u, 2j+v] = \sum_{nm} W^{2rb}[n,m]\ H^{2rb}[n+(2i+u), m+(2j+v)] - b^{3rb}
\end{aligned} \tag{4.3}$$

From these equations we can calculate the update-rules for back-propagation. This is done by applying the gradient descent method in which we calculate the derivatives of out error function. The update-rule consists of the derivative with respect to the parameter we want to update with some

$$\begin{aligned}
\Delta W^{4rbk}[i,j] &= \sum_k -\eta \frac{dE}{dF_k}\frac{dF_k}{dx_k}\frac{dx_k}{dW^{4rbk}[i,j]} \\
&= \sum_k \eta(t_k - F_k)\frac{e^{-x_k}}{(1+e^{-x_k})^2}\frac{dx_k}{dW^{4rbk}[i,j]} \\
&= \eta(t_k - F_k)\frac{e^{-x_k}}{(1+e^{-x_k})^2}H^{4rb}[i,j]
\end{aligned} \tag{4.4}$$

$$\Delta b_k = -\eta \frac{dE}{dF_k}\frac{dF_k}{dx_k}\frac{dx_k}{b_k} = \eta(t_k - F_k)\frac{e^{-x_k}}{(1+e^{-x_k})^2}(-1) \tag{4.5}$$

$$\begin{aligned}
&\Delta W^{2rb}[n,m] = \\
&\sum_k -\eta \frac{dE}{dF_k}\frac{dF_k}{dx_k}\sum_{ij}\frac{dx_k}{dH^{4rb}[i,j]}\sum_{uv}\frac{dH^{4rb}[i,j]}{dH^{3rb}[2i+u,2j+v]}\frac{dH^{3rb}[2i+u,2j+v]}{dx^{3rb}[2i+u,2j+v]}\sum_{nm}\frac{dx^{3rb}[2i+u,2j+v]}{dW^{2rb}[n,m]} \\
&= \sum_k\sum_{ij}\sum_{uv}\sum_{nm}\eta(t_k - F_k)\frac{e^{-x_k}}{(1+e^{-x_k})^2}W^{4rbk}[i,j]W^{3rb}[u,v]\frac{e^{-x^{3rb}[2i+u,2j+v]}}{(1+e^{-x^{3rb}[2i+u,2j+v]})^2} \\
&H^{2rb}[n+(2i+u), m+(2j+v)]
\end{aligned} \tag{4.6}$$

$$\begin{aligned}
&\Delta b^{3rb} = \\
&\sum_k -\eta \frac{dE}{dF_k}\frac{dF_k}{dx_k}\sum_{ij}\frac{dx_k}{dy^{4rb}[i,j]}\sum_{uv}\frac{dy^{4rb}[i,j]}{dy^{3rb}[2i+u,2j+v]}\frac{dy^{3rb}[2i+u,2j+v]}{dx^{3rb}[2i+u,2j+v]}\frac{dx^{3rb}[2i+u,2j+v]}{db^{3rb}} \\
&= \sum_k\sum_{ij}\sum_{uv}\sum_{nm}\eta(t_k - F_k)\frac{e^{-x_k}}{(1+e^{-x_k})^2}W^{4rbk}[i,j]W^{3rb}[u,v]\frac{e^{-x^{3rb}[2i+u,2j+v]}}{(1+e^{-x^{3rb}[2i+u,2j+v]})^2}(-1)
\end{aligned} \tag{4.7}$$

(4.8)

5. Method

The architectures of the Convolutional Neural Networks we use for this research are based on the LeNet-1 described in chapter 4. The architectures could differ in several ways from each other. First, we can adjust the sizes and types of the first convolution kernels. Second, the dimension reduction for both pooling layers in the network can differ. The size of the second convolution kernels can be changed. And the dimension of the output map (H4) can be chosen differently. At last, the amount of classifier neurons at the end of the network can be modified but this necessarily depends on the amount of classes the network has to resolve. It is possible to vary more parameters for these networks but this research is limited to these cases.

The networks were programmed in *Python*. No special packages related to Neural Networks were used such as *Theano*. For the convolution operation we used a package *Scipy.signals* which contains a convolution function. Furthermore, some additional packages were used for basic operations in the programme such as *Numpy* and *PIL*. The latter is used to import images for the root. Since we use patches to train the network we write a function which divides the imported images in patches. Each patch get the label corresponding to the class it belongs to so it can be used to train the network. The images were obtained from *Bing maps*. Suitable images were gathered by selecting only places (all around the world) which contains the favoured class. A side note has to be included for the images with the class city since these images may include some tree, grassland or water patches and thus will be labelled with the class city. Most other classes forest, grassland and water does not entail this problem since a great proportion of the world purely contains these classes.

In the previous chapter 4 we have seen that Convolutional Neural Networks can be used to classify images. Back-propagation is used to update the parameters to train the network. In this research we made a distinction between two types of training one in which only the weights between the output map H4 and the classifier neurons are trained including the biases on these classifier neurons. Second, a training in which the latter is trained and the parameters of the convolution kernels W^{2rb} from layer H2 to H3 and their biases b^{2rb} (see section 4.0.2). Since back-propagation costs al lot of computational power we have limited our research to these types of training. Of course, it is theoretically possible to use back-propagation till layer H1. For the first type of training the parameters of the second convolution kernels W^{2rb} were chosen from a random distribution.

5.1 Cross-validation

To compare the different architectures we used cross-validation. This includes the training of the network on 90% of the data/total amount of patches. The remaining 10% is used to validate the performance of the network. The training was performed in three steps. In the first step the learning rates were quite high such that the network roughly learns to recognize the classes. The following steps were trainings in which the learning rates were smaller. In other words, we did not used decaying learning rates or another method. To summarize, each patch from the (90%) training set has been 'seen' by the network three times and each time with a different learning rate.

From the network we want to know how it performance in general and on each class. So from the cross-validation four values were subtracted general performance and the performance on each class. Since this validation is executed for three classes we obtain this four values. The values are calculated by the relative fraction of patches which were classified correctly, or:

$$P = \frac{N_{right}}{N_{right} + N_{wrong}} \tag{5.1}$$

In which N_{right} determines the number of correctly classified patches and N_{wrong} the number of patches which were classified wrong. This equation is used to calculate the general performance in which we include all patches and for the performance for each class by only including the corresponding patches.

6. Results and Discussion

In the first phase of our research we made a convolutional neural network based on LeNet-1. We experimented with different convolution kernels in the first layer (H1) of the network and differed the amount of branches in the third layer (H3). We started with manually chosen convolution kernels for both layers H1 and H3. We began with input patches of 50 x 50 followed by a convolution kernel of 5x5 resulting in a feature map of dimensions 46x46. Next, sub-sampling with a dimension reduction of 2 results in a feature map of dimension 23x23. Again we did different 4x4 convolution kernel operation creating branches in H3 resulting in 20x20 feature maps. Sub-sampling with a dimension reduction of 5 resulted in an 4x4 output map. These outputmaps are then connected to two diffent classifcation neurons for Forest and City.

We used the Sigmoid function as activation function only for our classifaction-neurons. Thus, no activation functions were used earlier in the network (compared to LeNet-1). We performed back-propagation on the weights connecting the output maps with the classification neurons and biases. The first functioning results where found with 4 feature maps in H1 and 2 feature maps in H3 each where the network was enable to recognize some forest/city patches from an new image. These results were too poor to include in this report. We used the convolution kernels shown in figure 6.1. The Network was trained on structure only so no color feautures were included.

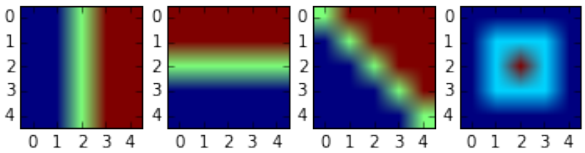

Figure 6.1: The four filters used in the first stage where the CNN was able to distinguish forest and city.

In the second phase we tried to extend our classes with the class *water* or *grassland*. We tried to investigate whether training only the last weights and biases were sufficient to train the network to classify three classes. During this phase we tried some adjustments to the network. We noticed that the last sub-sampling with a dimension reduction of a factor 5 could lead to problems since more information might be lost in this reduction. The underlying reasoning is that in each layer of the network more generalizations are made by the network which should represent the information of your input patch. When you downsize the feature maps at the end of the network this more contained information will be lost in this sub-sampling compared with sub-sampling earlier in the network. To test this hypothesis we trained different networks with different sub-sampling operations. Furthermore, we varied the size of the convolution kernels and the dimension of the output maps as explained in chapter 5.

From the first runs it turned out that the networks found it hard to distinguish the classes *forest* and *water*. By comparing some patches of both classes it became quite evident that this distinction is hard since there structure is similar (see figure 6.2). For this reason we included the class *grassland* instead of *water*. From figure 6.3 we can see that these patches are less similar and thus we would expect the network to perform better on these these combination of classes.

6.1 Results: Cross-validation

In this part of the research we investigated how different architectures of the network performance compared to each other. This comparision is made by using cross-validation explained in chapter 5. All network had similar convolution kernels for the first layer. If we used 3 rows or convolution kernels in the first layer we used the kerneltypes shown in figure 6.4. For the networks with 5 and 7 the kernels shown in figure 6.5 and 6.6 respectively.

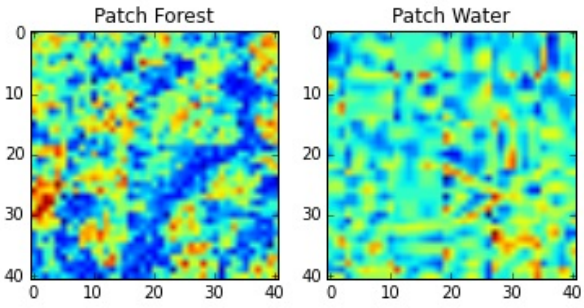

Figure 6.2: A forest and water patch are shown. Their similarity makes it hard for the CNN to distinguish these two classes.

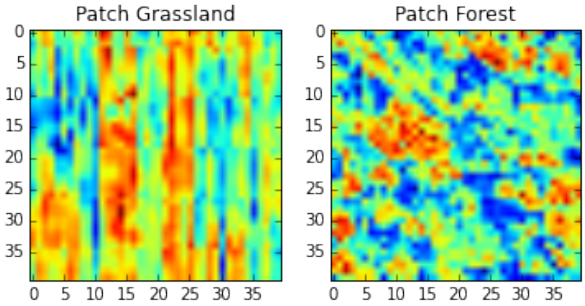

Figure 6.3: A grassland and forest patch are shown.

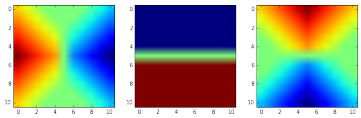

Figure 6.4: The three convolution kernels used for the architectures with three rows.

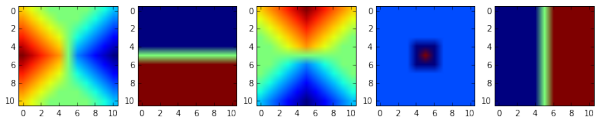

Figure 6.5: The five convolution kernels used for the architectures with five rows.

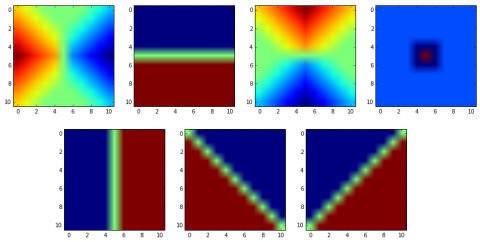

Figure 6.6: The seven convolution kernels used for the architectures with seven rows.

7. Time schedule

Week	Description of work	Deadline	Labour (hours)
20	• Implement backpropagation output layer • Start examining how to take into account RGB values • Train on more pictures and examine performance on other classes like city and water • Start developing frontend/ backend architecture	Finish updated proposal	40
21	• Work on implementing communication protocol • Experiments with connecting the nodes of the different layers of the Neural Networks	Finish backpropagation output layer, and frontend implementation of the communication protocol	40
22	• In-between overall evaluation • Backend and frontend implementation • Experiment with preprocessed images (noise reduction) • Implement backpropagation further, until layer H3, see Figure 4.1	Finish model with complete RGB values and the frontend/backend architecture must be completely finished	40
23	Start report writing and implement last adjustments	Finished product for demonstration	40
24	Preparation for demo	Demo	30
25	Project finalization	-	30
26	Project finalization	Project report + individual document	40
			Total: 468