Classification of Textures Using Convolutional Neural Networks

Ivar Persson
tpi12ipe@student.lu.se

Supervisors:
Magnus Oskarsson, Kalle Åström

Master's thesis

Lund University

June 17, 2017

Abstract

This Master's thesis have concerned the segmentation and classification of background textures in images. In order to segment the images we have used the SLIC algorithm to create superpixels. These are a sort of oversegmentation of the image where pixels close to each other and similar in colour are considered to be the same texture. All superpixels were then classified using a Convolutional Neural Network which was trained as a part of this thesis. As this network had about 30% errors a second stage was added to the classification program, a spatial bias. The first attempt at this spatial bias used the neighbouring superpixels' classification in order to make the image more homogeneous. Secondly, as a comparison, a neural network was also trained as the spatial bias.

When using the neural network followed by the neural network for spatial biases the errors decreased to a little more than 10%, while the ordinary spatial bias only decreased error by a few percent. Even though the best performing network had a low error rate we were not able to replicate these results with unknown images as the network most likely were severely overfitted to the relatively small training set. The method in it self showed potential and with more training data and possibly smaller superpixels we could get more consistent results between training images and other unknown data.

Sammanfattning

Denna Masteruppsats har behandlat segmentering och klassificering av bakgrundstexturer i bilder. För att segmentera bilderna har vi använt en algoritm, kallad SLIC, för att skapa så kallade superpixlar. En superpixel är en samling pixlar som är nära varandra rent fysiskt och med liknande färg. Dessa superpixlar klassificerades sen av ett Faltande Neuralt Nätverk (*eng.* Convolutional Neural Network) som tränades upp genom att låta nätverket se superpixlar som redan var manuellt klassificerade. Eftersom detta nätverket gjorde ca. 30% fel lades ett andra steg till i klassificeringsalgoritmen. Detta andra steg bestod i att angränsande superpixlars ursprungliga klassificering ökade chansen för de typerna av texturer i suprpixeln i fråga. Som en jämförelse tränades också ett neuralt nätverk för att hantera angränsande suprpixlar.

Genom att använda det faltade neurala nätverket följt av det neurala nätverket för grannliggande superpixlar sänktes felen till ca. 10%, medan den vanliga hanteringen av grannliggande pixlar endast sänkte felen med några få procentenheter. Trots att felen var så låga kunde dock inte resultaten återupprepas på okända bilder, antagligen på grund av att de neurala nätverken var övertränade på de relativt låga antalet superpixlar. När ett neuralt nätverk är övertränat innebär det att det har väldigt stor förmåga att känna igen, i det här fallet superpixlar, som liknar de som är använda vid träningen men helt oförmögen att korrekt klassificera andra superpixlar. Metoden med två neurala nätverk har potential och genom att ha fler träningsbilder och eventuellt mindre superpixlar, så kanske resultaten från träningen kan återupprepas på okänd data.

Acknowledgements

I would like to thank my supervisors Magnus Oskarsson and Kalle Åström, who helped me plan this thesis work and helped me throughout the project with all different kinds of problems. I would also like to direct many thanks to Anna Broms and Anna Åberg. Anna Broms for your wizardry with LaTeX, Anna Åberg for your uncanny ability to find all of my spelling errors and both of you for acting sounding boards willingly or not.

Contents

1 Introduction

In this Master's thesis we will implement a program to segment and classify textures in images. With textures we mean everything in the background – grass, water, sky etc – not the objects. This thesis goal is part of a much bigger problem, where we want to automatically create meaningful captions to images. First of all, this requires a good segmentation and classification of all parts of the image, not just the foreground objects. Secondly, a caption writer that produces reasonable sentences is needed. The entire problem of meaningful captions has attracted attention from some of the biggest corporations in the world, e.g. Facebook, Microsoft and Google, that has lead to the COCO (Common Objects in Context) Captioning Challenge [12], a competition to create automatic image captions. Research teams at Google [20] and Microsoft [3] tied first place in the 2015 challenge. Besides the COCO Captioning Challenge, there are several benchmark tests and competitions for only the segmentation and classification (so called semantic segmentation), and not the captioning, for example the IAPR TC-12 [6] benchmark with over 20,000 images and the Imagenet [16], a sort of computer olympics for object detection, with over one million images. This thesis does however not concern objects at all, but rather the background textures of an image. As an example, as competing program in the COCO Challenge should probably caption Figure 1 as "Two people sitting on a beach under some palm trees" or similar. We will ignore the people and just display what areas are sand, water, forest and possibly urban area in the background.

In order to make the classifications we need to divide the image into small pieces that can be analysed separately. We will use so called superpixels, an oversegmentation of an image into small pieces where each superpixel covers an area depicting roughly the same thing. Once the segmentation is done, a Convolutional Neural Network (CNN) will be used to classify the superpixels. This thesis will present a state-of-the-art algorithm to create superpixels and, by also presenting the latest research on CNNs and all its parts, we design a CNN and train it to be able to make the classifications.

Visual results will be mixed with numerical error rates and types of miss-classifications in order to get a complete picture of the performance of the network. An important question to investigate in this thesis, apart from how many errors the network makes, is whether the CNN can distinguish similar looking textures from each other and what types of error it will make. The implementation of the CNN will use MatConvNet [19], a prewritten code for matlab where building blocks of the CNN are defined, in order to speed up the implementation.

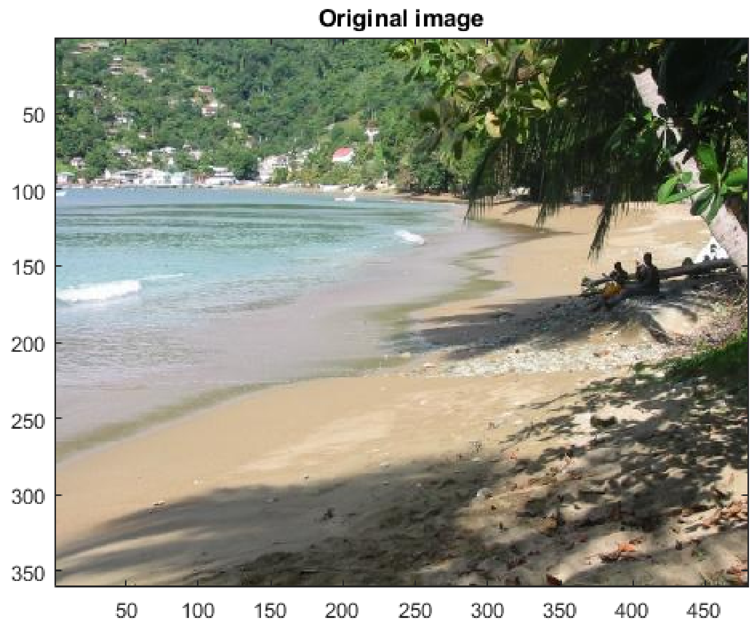


Figure 1: A typical image where the background textures were to be segmented and classified.

1.1 Texture classification

As noted we will classify background textures. There will be a total of 15 classes including grass, forest, bushes, sky, water and an object class. These are all textures from the outside and there is a good reason for that. Imagine any room indoors; the entire room is filled with objects, for example a couch, a book shelf, a table etc. but there are very few open spaces with for example just floor or wall, as there are probably some rugs and paintings. Outdoors we can often find images with big sections of a forest or an ocean which will work to our advantage as we want images with a lot of background textures and few objects.

When we speak of texture in every day life, we often speak of the feel of a material, a fabric or a piece of wood for example. This idea is what we want to use CNNs for to differentiate between similar textures. As an example we can consider blue skies and water. How do we separate them even if they are both blue? The water will often have ripples and other small waves while the blue skies will be a smoother, or a more homogeneous, blue colour. Traits like this will hopefully separate similar textures from each other.

1.2 Related work

Neither image segmentation nor neural networks are new research areas. CNNs have been around since the 1990s with some of the first articles from Yann LeCun et al. [10, 21] where handwritten zip codes were analysed. A lot of work on neural networks has been on object detection and classification [7, 9], where classification errors have drastically dropped since 2012 when Krizhevsky et al. [9] published groundbreaking results on the Imagenet data set. Less work seems to have been put into background textures. However Tivive and Bouzerdoum [18] has shown the advantages of using CNNs over traditional texture classification methods. The so called superpixels have also been examined closely, for example in [15, 1], where several different algorithms are compared. The methods we will use; CNNs and superpixels for texture classification, are all known theories on their own, however, little or no work in this particular combination seems to have been done.

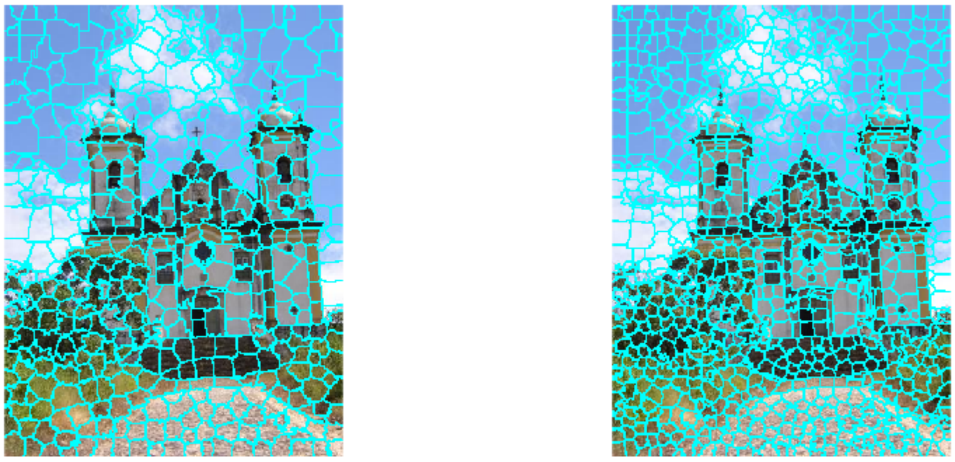
2 Theory

In this theory segment we will present the relevant theory about superpixels, why we want them and how we calculate them. Next we will explain how CNNs works and how the training of a CNN is a big optimisation problem. We will not cover general artificial neural networks, however it can be pointed out that some of the theory is the same. As a finalisation we present the second stage in our classification algorithm, the spatial biases and how they can help reduce error rates.

2.1 Superpixels

As we are classifying textures it is logical that we need to analyse a larger area, not just a single pixel, to determine which texture it belongs to. A quick and easy solution would be to divide the image into squares and analyse each square by it self, however, texture boundaries do not follow straight lines and therefore we need a robust system to select a collection of pixels that all describes one texture only. This is why we want superpixels. The results of a superpixel segmentation can be seen in Figure 2, where two different numbers of superpixels have been used. As we can see, pixels close to each other spatially and similar in colour are bundled together, just as we were interested in.

There exists many different algorithms to calculate suprpixels, such as a known graph based algorithm by Shi and Malik [15]. However, Achanta et al. [1] shows that their proposed algorithm, Simple Linear Iterative Clustering (SLIC), is both quicker, more memory efficient and more robust in terms of obeying boundaries. In MATLAB, there is a prewritten implementation (`superpixels.m`) of the SLIC algorithm which we are able to use. The algorithm by Achanta et al. is described in Algorithm 1.



(a) *500 superpixels* (b) *1000 superpixels*

Figure 2: *An image of a building with two examples of different number of superpixels in the segmentation.*

Algorithm 1 SLIC algorithm by Achanta et al. [1]

```
1: Set all labels $l(i) = 0$
2: Create and set a distance measure $d(i) = \infty$ for all pixels $i$.
3: Set error $E = \infty$
4: Set error limit $\delta$ to some constant
5: Initialise cluster centres $C_k$ by sampling an equidistant grid, i.e. the grid intervals will be $S = \sqrt{N/k}$,
   where $N$ is the number of pixels in the image and $k$ the desired number of superpixels. The $C_k =
   [l_k, a_k, b_k, x_k, y_k]$, belong to the CIELAB colour space. The CIELAB or LAB colour space is a different
   way of describing colour compared to the well known form of red (R), green (G) and blue (B) colour
   channels. Here we have the three channels brightness (L), red/magnta (A) and blue/yellow (B). The
   $x_k$ and $y_k$ are spatial coordinates while $l_k$, $a_k$ and $b_k$ describe the colour.
6: Move centres $C_k$ in a $3 \times 3$ spatial neighbourhood to lowest gradient position, i.e. the gradient with
   respect to $l_k$, $a_k$ and $b_k$. This is to avoid centres to be on lines or in noisy pixels.
7: while $E > \delta$ do
8:     for $k = 1 : N$ do
9:         for all pixels $i$ in a $2S \times 2S$ region around $C_k$ do
10:            Calculate a distance $D$
11:            if $D < d(i)$ then
12:                $d(i) = D$
13:                $l(i) = k$
14:    Calculate new $C_k$ based on current labels $l$
15:    $E = \sum_i d(i)$
```

The authors state that the *while*-statement on row 7 can be replaced by a *for*-statement of ten iterations. It is usually enough for most images [1] and this is also the default method in MATLAB. Very consciously we have, until now, referred to the distance measure as some sort of measure. When we are calculating the distance between a cluster centre C_k and a pixel $i = [l_i, a_i, b_i, x_i, y_i]$ we cannot use an ordinary 5D Euclidean distance as it would give different importance to spatial proximity for different sizes of superpixels. Therefore, we define a modified distance D' as a combination of the colour proximity d_c and the spatial proximity d_s:

$$\begin{aligned} d_c &= \sqrt{(l_k - l_p)^2 + (a_k - a_p)^2 + (b_k - b_p)^2}, \\ d_s &= \sqrt{(x_k - x_p)^2 + (y_k - y_k)^2}, \\ D' &= \sqrt{\left(\frac{d_c}{N_c}\right)^2 + \left(\frac{d_s}{N_s}\right)^2}. \end{aligned} \tag{1}$$

Here we get a 3D Euclidean measure for the colour difference and a 2D Euclidean measure for spatial differences, which are then weighed against each other. The maximum expected width and height of a superpixel is the grid interval, therefore we set $N_s = S$. The maximum colour distance N_c is set to some constant m as it is harder to define a maximum colour distance. Every pixel is different and therefore we leave m as a constant. By inserting the values of N_s and N_c into D' in equation (1) and multiply with m, we get the distance measure used in the algorithm

$$D = \sqrt{d_c^2 + \left(\frac{d_s}{S}\right)^2 m^2}.$$

We have m as a tunable parameter. Small m promotes colour proximity and large m promotes spatial proximity [1]. Achanta et al. also tells us that $m \in [1, 40]$ for images in CIELAB space. In the algorithm in MATLAB, $m = 10$ by default and we will also use this value. These default values made the superpixel segmentation in Figure 2.

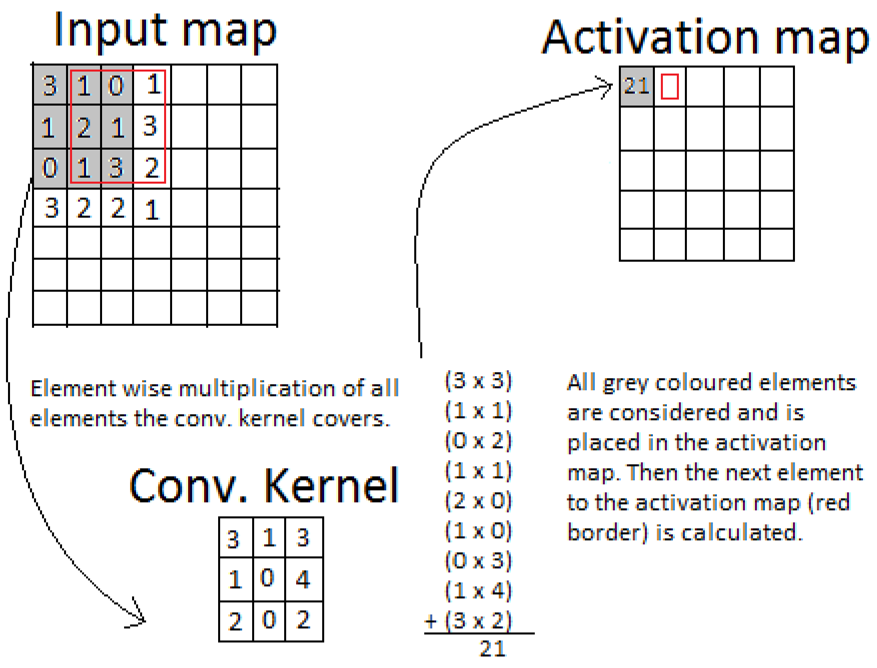
2.2 Convolutional neural network

Convolutional neural networks, or CNNs, are built up by neurons, just as any other neural network and these neurons are connected to each other in a predetermined way. In the CNN each neuron usually consists of a convolution layer which can be paired with so called pooling, activation functions and many other types of layers. In this section we will describe these layers, how they work and also explain why we choose to use them. Almost each neuron consist of a convolutional layer. This layer is a feature extractor for the input image. We have a two-dimensional convolution

$$y(m,n) = x(m,n) * h(m,n) = \sum_j \sum_i x(i,j) h(m-i, n-j), \tag{2}$$

where h is the convolution kernel and x is the input image. The output y is known as the activation map or featue map. An illustration of this convolution can be seen in Figure 3, where the convolution kernel (in the middle) is element wise multiplied with all elements across the image [13]. The values of the elements in the convolution kernel are known as weights w_{ji}. The subscripts to w is indicated as w_{ji} is the ith weight belonging to neuron j. The weight w_{j0} is known as the bias to neuron j, that is added to each element in the activation map. The new image created from the convolution, the activation map, is seen to the right in Figure 3.

When constructing a convolution kernel, the network designer has a few choices. First, the size of the kernel has to be considered. A large kernel will analyse the larger context of the image, while a small one will capture structures at a finer level. In Figure 3 we see a fairly small kernel with size 3×3. Secondly, one can choose to use zero-padding on some or all sides of the image in order to control the size of the output from the convolution. For example, in Figure 3 we seem to have one line of zero padding on the left, right and top. In an ordinary two dimensional convolution, such as equation (2), the summations are over the range of $[-\infty, \infty]$. However the input image x is not infinitely large and with the convolutions here no calculations are made where the kernel does not completely cover the image. Once we have



Input map
Activation map
21
Element wise multiplication of all elements the conv. kernel covers.
Conv. Kernel
(3 x 3)
(1 x 1)
(0 x 2)
(1 x 1)
(2 x 0)
(1 x 0)
(0 x 3)
(1 x 4)
+ (3 x 2)
21
All grey coloured elements are considered and is placed in the activation map. Then the next element to the activation map (red border) is calculated.

Figure 3: *Two-dimensional convolution. The convolution kernel (bottom) is element wise multiplied with each element the kernel covers, in this case a* 3×3 *area, before it moves on to the next location.*

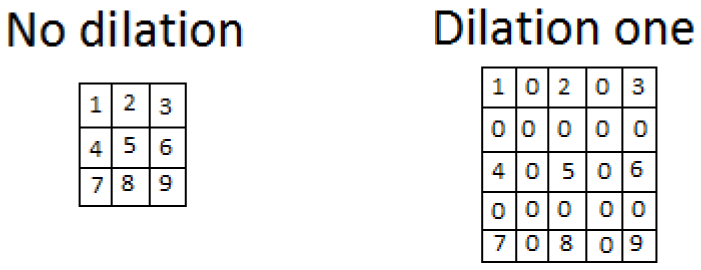


Figure 4: *Illustration of how a non-dilated convolution kernel looks compared to one with one dilation.*

chosen filter size and zero-padding it is also common to consider the stride. A filter is usually calculated at some point and then moved one step. This may not be the best way in all cases, as larger steps of more than one pixel is needed. This can, among other methods still to be described, be used to control the reduction of the dimensions of the activation map [8]. Furthermore, one can choose to dilate the convolution kernel, i.e. inserting rows and columns with zeros in them, as seen in Figure 4. This has proven to lower errors in semantic segmentation [22] where textures are considered.

We have chosen to use convolution kernels of size 5×5 and 6×6. These are a little bigger than the ones shown in figure 3. For example, since we are dealing with textures we do not want to analyse individual tree branches, we want to analyse the entire tree. The superpixels generated by the SLIC algorithm can have various shapes and therefore they are padded with zeros to become rectangular. This rectangular shape is needed when running the superpixels through the network. Since we already have some zeros around the pixels any additional zero padding is deemed unnecessary. We opted to use a stride of one since the images were not very big and we do not need rapid dimension reduction. Although we are doing semantic segmentation of images and should consider dilated kernels as proposed in [22], the neural network will classify entire superpixels in a similar manner to object detection networks and

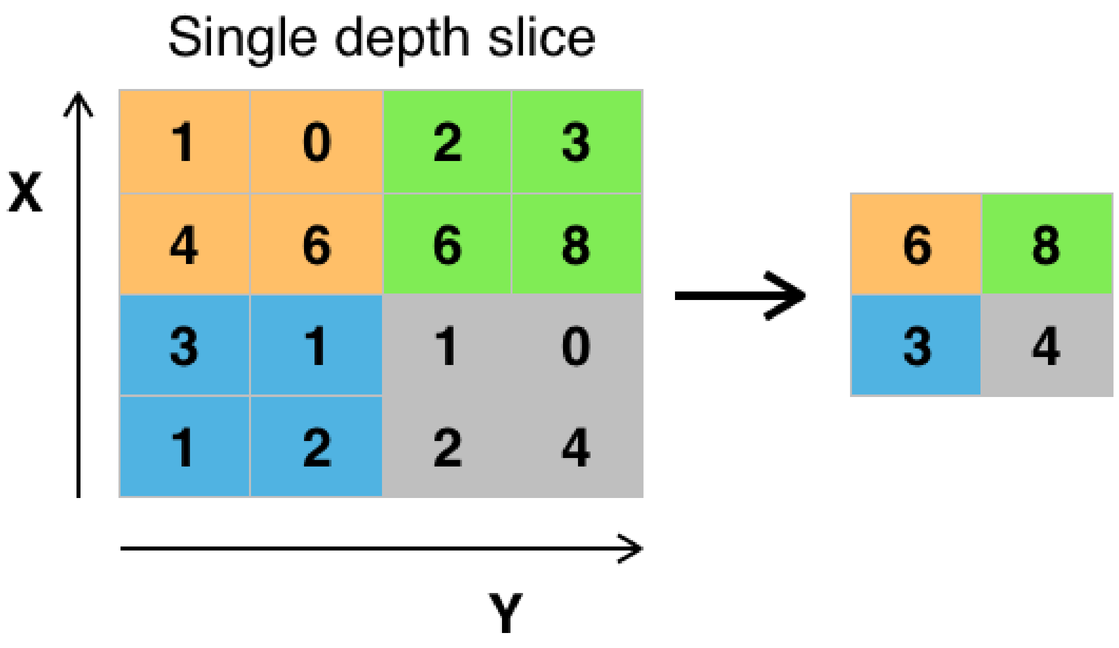


Figure 5: *Illustration of the max pooling operation.* 2×2 *filters with stride 2 are placed over the activation map of size* 4×4*. The maximum value in each area is passed on to the next step.*

therefore we will not dilate the convolution kernels. One can say that the network does not consider the entire image or its neighbouring parts, it only classifies the superpixel.

Pooling is a type of down-sampling of an activation map from the convolution. Some common types of pooling are max, average and sum pooling. This kind of layers are used to reduce spatial dimensions of activation maps to the next layer [14]. Pooling is conducted by running a filter, usually of size 2×2 or 3×3 over the activation map in a similar manner as the convolution kernel. However, instead of convolving all values in the range of the kernel, the maximum, average or sum is used as the value for the activation map in the next layer. In Figure 5 we can see an example of the effect of max pooling with 2×2 filters and stride 2. Scherer et al. [14] show that max pooling is often superior and max pooling is now the standard pooling type in many networks [9, 7, 17]. As with the convolution kernel, the network designer has to choose size and stride of the pooling filter. Krizhevsky et al. [9] show that overlapping pooling layers, i.e. pooling with for example size equal to 3 and stride equal to 2, reduces error rates and decrease the risk of overfitting the network to training data. Overfitting is the phenomenon where you train you network to recognise your training data very well but the network is not stable nor capable to classify object not belonging to the training set. In accordance with the article by Krizhevsky et al. [9] we use pooling with size 3, stride 2 and since Scherer et al. [14] shows that max pooling is superior we have used that as well.

Within the field of neuroscience it has been noted that the firing intensity of a neuron is not linearly dependent on the input activation [5]. If we translate this fact about biological neurons into these artificial ones, it would mean that the actual value of the activation map will not be linearly dependent on the value calculated from the convolution and pooling. Therefore, a non-linear activation function, applied to each element in the activation map, has been introduced to better simulate the natural behaviour of biological neurons. Historically a sigmoid or hyperbolic tangent function have been used [5]. This is however not the best method, as Glorot et al. [5] shows in his article where the non-linear and non-symmetric function

$$f(x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0, \end{cases} \tag{3}$$

is used. This is known as the Rectified linear unit (ReLU). LeCun et al. [11] describes how the ReLU has a quicker learning process than the sigmoid function. It has been noted that cortical neurons (brain cells) rarely fire at full intensity and that is why a non-saturating activation function (a function without supremum), such as the ReLU can be suitable [5]. In 2015, He et al. [7] proposed the Parametric Rectified Linear Unit (PReLU) to replace the ReLU. The PReLU is similar to the ReLU in equation (3), however, for inputs smaller than zero we still have some non-zero output. We have

$$f(x) = \begin{cases} x, & x \geq 0 \\ a_i x, & x < 0, \end{cases} \tag{4}$$

where a_i is some trainable scalar connected to that neuron. The PReLU outperformed the ordinary ReLU on the ImageNet (the unofficial object recognition olympiad) [7]. The used implementation in MATLAB from MatConvNet, have a more static version of the PReLU is used, called Leaky Rectified Linear Unit (LReLU), where a_i is set to some constant for all neurons[19]. The LReLU does, however, not increase performance of the network significantly [7].

In order to limit the implementation needed we have chosen the ReLU as activation function and not PReLU or LReLU. This function will be applied to the activation map after every convolution to mimic nature as much as possible. Even though the PReLU outperforms the ReLU, many systems use the ReLU and still get very low error rates [9, 17] and therefore we can choose to save implementation time without jeopardising the performance of the network. As the LReLU does not significantly increase performance there is no need to use it.

2.3 Training a CNN

The training of a CNN is a time consuming process where a few steps need to be iterated many times. We begin by initialising all weights and biases in the convolutional kernels. A common practice is to sample the weights from a normal distribution with mean zero and standard deviation 0.01 and all biases to zero, as Krizhevsky et al. [9] do, where they also claim that an initialisation of the weights will in this manner increase learning rate in the early stages compared to initialisation with uniform distributions. He et al. [7] proposes a type of robust initialisation method for very deep neural networks (more than 8 convolution layers), since ordinary initialisation can have convergence issues. However, we will settle with ordinary initialisation as in [9] since our network will not be very deep. Convergence issues could appear anyhow but it should be easy to notice these issues and be able to correct them for example by lowering the learning late, a concept discussed later in section 2.3.1.

The training can then be divided into three parts; forward pass, backward pass and weight update. The forward pass is simply just running through the network with the current weights and biases to get a classification prediction. Next, the backward pass calculates the local gradient of the error, i.e. the gradient in the neuron, before this result is used for the last stage, to update the weights. The backward pass and weight updates will be done using the back propagation algorithm, which we will get back to. The forward pass, backward pass and weight update are usually called an epoch (iteration) and the training will be conducted over many such epochs.

The second to last layer of the neural network will have an resulting activation map with dimensions $1 \times 1 \times D \times N$, where D is the number of classes we can assign and N is the number of images we used in training [19]. The goal here is to have high scores for the most probable classes and low scores for the less likely and this is where we have a ground truth to compare the predictions from the network. Before, in the so called hidden neurons, the output had no ground truth to compare to. The last layer in the network is a so called loss layer, where a total loss, or error, of the network is calculated. This value is low when the network is performing well and high when it is not [8]. This extra neuron also makes sure we can calculate the local gradient in the second to last neuron. In the next section we will show how the network can learn through back propagation.

One last thing to consider when designing a CNN is dropout. When using dropout, connections from one neuron to the next only exist with a certain probability for each training epoch. Krizhevsky et al. [9] describes the importance of this in, where severe overfitting otherwise occurs since the network is very big compared to the number of training examples. We chose to use dropout on the fully connected layers, just as in [9], however at a rate of 30% instead of 50% as we have a limited amount of data. Dropout of 30% should be enough since our fully connected layers are much smaller than the ones Krizhevsky et al. uses. This value is not definite and other dropout rates will be discussed later.

2.3.1 Back propagation

The back propagation algorithm is based on several partial differentiations along with the chain rule. This derivation is based on the derivation in [4] with some extra comments. In a neuron we have several weights connecting other neurons to this neuron. High values on the weights indicate a strong connection. In general, we have a convolution and get an activation map $v_j(k)$ as

$$v_j(k) = \sum_{i=0}^{n} w_{ji}(k) x_i(k), \tag{5}$$

where $v_j(k)$ is the activation map belonging to the j:th neuron in epoch k. For each k and j, v is a matrix. We have as input into neuron j $x_i(k)$ from neuron i, which is connected to neuron j with the weights $w_{ji}(k)$. We can compare with Figure 3, where neuron i is to the left, the convolution kernel is composed by the values $w_{ji}(k)$ and the resulting activation map is to the right. In total n weights connects neuron i to neuron j. A schematic image of the connections between neuron can be seen in Figure 6, where our neuron j is connected to 4 neurons. The final output from the neuron j is then given by

$$y_j(k) = \Phi_j(v_j(k)), \tag{6}$$

where Φ_j is the activation function (sigmoid, ReLU etc) belonging to neuron j. The output $y_j(k)$ can then be used as input in equation (5) for some other neuron. We define an error

$$e_j(k) = d_j(k) - y_j(k) \tag{7}$$

and an error energy

$$E_j(k) = \frac{1}{2} e_j^2(k) \tag{8}$$

in neuron j, given the desired output $d_j(k)$ and the actual output $y_j(k)$. The total error energy of the network so far is then given by $E(k) = \sum_j E_j(k)$, for all previous neurons j.

The goal of back propagation is to minimise the error energies $E_j(k)$ with respect to the weights in the network. Minimising an unknown function can be done by following the negative gradient of E until we find the minimum. The chain rule tells us that

$$\frac{\partial E_j(k)}{\partial w_{ji}(k)} = \frac{\partial E_j(k)}{\partial e_j(k)} \frac{\partial e_j(k)}{\partial y_j(k)} \frac{\partial y_j(k)}{\partial v_j(k)} \frac{\partial v_j(k)}{\partial w_{ji}(k)}.$$

We can now identify the partial derivatives from equations (8), (7), (6) and (5)

$$\frac{\partial E_j(k)}{\partial e_j(k)} = e_j(k), \quad \frac{\partial e_j(k)}{\partial y_j(k)} = -1, \quad \frac{\partial y_j(k)}{\partial v_j(k)} = \Phi_j'(v_j(k)), \quad \text{and} \quad \frac{\partial v_j(k)}{\partial w_{ji}(k)} = y_i(k). \tag{9}$$

Combining these partial derivatives in equation (9), we get

$$\frac{\partial E_j(k)}{\partial w_{ji}(k)} = -e_j(k)\Phi_j'(v_j(k)) y_i(k),$$

where we define

$$\delta_j(k) = -\frac{\partial E_j(k)}{\partial v_j(k)} = e_j(k)\Phi_j'(v_j(k))$$

as the local gradient in neuron j. No ground truth exist in our CNN as we have hidden neurons. We cannot form an error or error energy, how could then the local gradients for the hidden layers be calculated?

We begin by considering the error $e_l(k)$ in some non-hidden neuron and the total error energy $E(k) = \sum_j E_j(k)$ in the network. We obtain

$$\frac{\partial E(k)}{\partial y_j(k)} = \sum_l e_l \frac{\partial e_l(k)}{\partial y_j(k)} = \sum_l e_l \frac{\partial e_l(k)}{\partial v_l(k)} \frac{\partial v_l(k)}{\partial y_j(k)}. \tag{10}$$

We know from equation (9) that

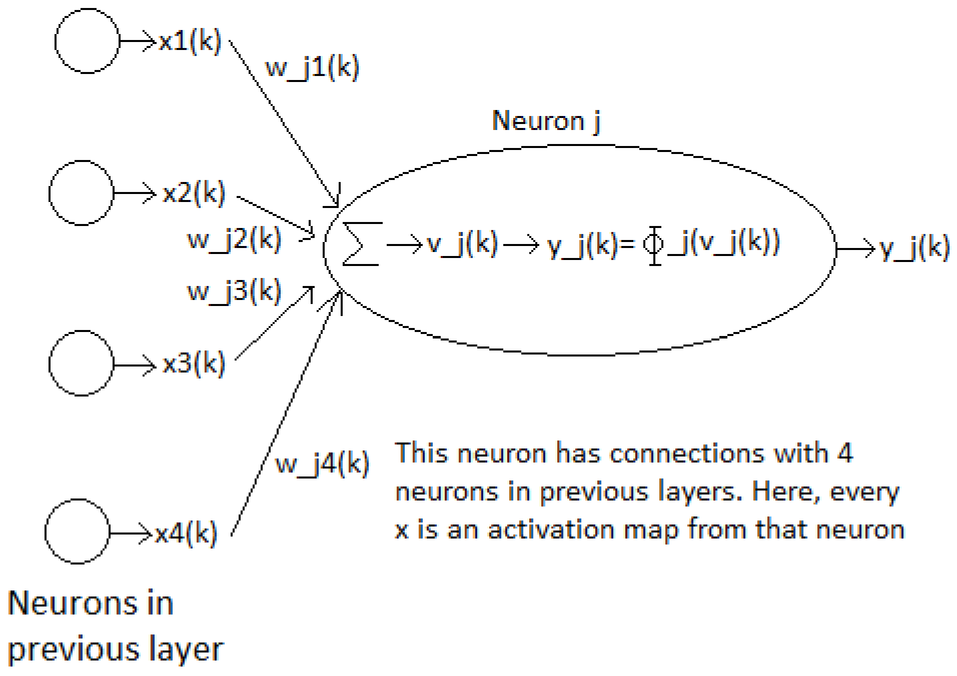

x1(k)
w_j1(k)
Neuron j
x2(k)
w_j2(k)
∑ → v_j(k) → y_j(k)= Φ_j(v_j(k))
y_j(k)
w_j3(k)
x3(k)
w_j4(k)
This neuron has connections with 4 neurons in previous layers. Here, every x is an activation map from that neuron
x4(k)
Neurons in previous layer

Figure 6: *Schematic image of connections between neurons. Inside neuron j we see the summation from equation* (5)*, the activation function from equation* (6) *and the output $y_j(k)$.*

$$\frac{\partial e_l(k)}{\partial v_l(k)} = \frac{\partial e_l(k)}{\partial y_l(k)} \frac{\partial y_l(k)}{\partial v_l(k)} = -1 \times \Phi_l'(v_l(k)). \tag{11}$$

From equation (5) we get that

$$\frac{\partial v_l(k)}{\partial y_j(k)} = w_{lj}(k). \tag{12}$$

Substituting (11) and (12) into equation (10) we get

$$\frac{\partial E(k)}{\partial y_j(k)} = -\sum_l e_l \Phi_l'(v_l(k)) w_{lj} = -\sum_l \delta_l(k) w_{lj},$$

where $\delta_l(k)$ is the local gradient. We can now describe the gradient in a hidden neuron j as

$$\delta_j(k) = -\frac{\partial E(k)}{\partial v_j(k)} = -\frac{\partial E(k)}{\partial y_j(k)} \frac{\partial y_j(k)}{\partial v_j(k)} = \Phi_j'(v_j(k)) \sum_l \delta_l(k) w_{lj}(k),$$

i.e. the sum of previous gradients times the weights and times the derivative of the activation function in neuron j. To update the weights we will need two more parameters; the learning rate and the momentum. We define the size of the weight update on the weight from neuron i to j, $\Delta w_{ji}(k)$, as

$$\Delta w_{ji}(k) = \beta \Delta w_{ji}(k-1) + \alpha \delta_j(k) y_i(k), \tag{13}$$

where α is the learning rate and β is the momentum. If we first consider only the second term, we have the local gradient in the j:th neuron times the output from the i:th neuron. This is the gradient with respect to the weight. The gradient is in turn multiplied with α, to regulate the size of the imposed changes to the weights. The learning rate could be lowered during the training in order to to come even closer to the local minimum [8, 19]. Now we consider the entire expression in equation (13). We use the last update $\Delta w_{ji}(k-1)$ in the previous epoch $k-1$, and multiply it with our momentum β, commonly set to 0.9 [9, 17, 22]. The second term is now a correction term to steer the updates in the correct direction. By using this momentum we avoid getting stuck when the local gradient is small.

The observant reader might recognise the weight update as simple gradient search with an extra momentum. An obvious problem with a gradient search is of course the risk of getting stuck in a local minimum in the error function. This is, however, not usually a problem as LeCun et al. [11] mentions. Choromanska et al. [2] even proves that recovering the global minimum is harder when the network is bigger and also irrelevant as a too good minimum probably will cause a lot of overfitting.

2.3.2 Measuring the error

We have mentioned earlier that we compare the results given from the network and the ground truth when we are training the network with the back propagation algorithm. However, we have not mentioned what we are measuring when we talk about the error. This error is a crucial starting point when calculating the back propagation. In section 4.1 we will see three different types of plots. The leftmost plot is the objective function (see Figure 7a). It is the output from the very last layer, the loss layer:

$$L(X,c) = -\ln\left(\frac{e^{X(c)}}{\sum_q e^{X(q)}}\right), \tag{14}$$

where c is the correct classification label and X is the output vector from the second to last layer. Another common thing is to calculate the loss as $L(X,c) = -\ln(X(c))$ called the log loss. The softmax log loss normalises the predictions apart from the log loss [19]. This means that our resulting vector will contain the probabilities that the superpixel belongs to some class. In other words, these number all sum to one.

A perfect network where the output vector is one for the correct texture and zero for everyone else will become $L(X,c) = 0$ as both the numerator and the denominator will be one. For a truly random weights we will get $X(c) = \frac{1}{15}$, for 15 classes, and $X(q) = \frac{1}{15}$ for all q. Inserting this into equation (14) will yield $L(X,c) = -ln(\frac{e^(1/15)}{15e^(15)}) \approx 2.708$. With 7 classes we get $L(X,c) \approx 1.946$ with random weights.

2.4 Introducing spatial biases

As the neural network is not perfect we will use our knowledge about background textures to try to reduce errors. Background textures cover large areas of an image and neighbouring superpixels are often depicting the same texture. Our method will therefore be to use the classification of the neighbouring pixels. Usually when using a neural network we run through the network and classify depending on what is the most probable alternative. However, we will consider what the top three choices from the network are for some superpixel. Then we will investigate if any of the neighbouring pixels are classified as one of these top three choices. If a pixel is classified as one of these three choices we add a bias to that choice. We choose to introduce this bias in order to force neighbouring superpixels to be classified in the same way as its neighbouring superpixels, but only if there is doubt from the network. In other words, it is possible that we have some superpixels surrounded by nothing but other textures. If the network is sure enough about its classification, then the bias will have no effect on the classification but if the network is not sure the bias will kick in. By adding this bias, we should cause the images to become more homogeneous, a trait we seek for these images. In Section 4.1 we will show error rates with both the firsthand choice and these top three choices. It is common when presenting results from neural networks to present both the firsthand error and a top-3 or top-5 error. This shows how many times the network is close to the correct answer. We hope that a low top-3 error will drastically drop the overall classification errors.

This proposed rather static bias, just presented, should improve error rates, however, a more flexible solution might be needed to handle different cases. As a contender to the static bias we will also present a neural network to handle the spatial dependencies. These two solutions can then be compared to each other. This neural network handling spacial dependencies, from now on referred to as the spatial network, will differ the spatial bias in many ways. Instead of just using the top-3 choices and search for similarities we start by extracting the entire probability vectors from the 10 closest superpixels, the superpixel itself and 9 closest neighbours. Here we want to find robust similarities as to how the network makes its errors and try to correct them. We stack the probablility vectors, in order of proximity to the superpixel, as an "image" of size $1 \times 15 \times 10$. We trick the CNN into believing that our 15 classes in 10 superpixels are an image with 15 pixels in a row and 10 colour channels. This "image" is first sent into a neural network with a convolution layer of size $1 \times 1 \times 10$ in order to sum up the probabilities for each of the classes by them self. Following, there are two fully connected convolution layers in order two change the size from the input of $1 \times 15 \times 10$ to an output of $1 \times 1 \times 15$. This network is trained on the results that the best classification network produced. In Section 2.3 we decided that the initialisation of the weights for the CNN would be drawn from a normal distribution, however with this non-conventional spatial network it was not possible as we did not reach convergence. Instead the initialisation was that the network would send the result vector from the pixel it self without any alterations from the network. We then trained and optimised the weights from that starting point. We should expect a smaller initial value of the objective function compared to the inital for the classification network.

3 Implementation

Here we will present the final details of the classification program, including the overall structure, what textures we considered and how we defined each of them. We will also present the used image material that the network was trained on.

All images in the data set are of size 360×480 pixels (or transposed) and by visual evaluation it is deemed reasonable to have approximately 260 superpixels in each image. Due to the nature of the SLIC algorithm it never produced exacly 260 superpixels but close to it. If we choose the superpixels to be too small, creation of the ground truth will take a very long time and too large superpixels will not segment the image enough and we run the risk of having several textures in one superpixel.

In order to save RAM during training we will use stochastic gradient descent (SGD) when back propagation is performed. This means that a small set of images are evaluated at a time and the weights are then updated after each small set [11]. This will cause a bit of a jiggle in the way the weights are optimised, however SGD has been proven to work for reasonable batch sizes [8]. We will use batch sizes of 100 superpixels.

3.1 Image material

In this project we will work with the large IAPR TC-12 data set provided by ImageCLEF [6]. This dataset consists of 20,000 colour images depicting humans, scenery, buildings and many other things. The reason why we use IAPR TC-12 and not for example COCO [12] is that the images in COCO are filled with many different objects as it is part of the COCO challenge, while we want a lot of background and scenery to analyse. As this is a very large dataset, only some of these images were used to create a ground truth that was used to train the neural network, while other images were used for visual validation instead. A total of 7967 superpixels from 31 images were manually classified into the 15 texture classes and 10% of the superpixels were withheld for validation.

3.2 Structure of program

The average superpixel was approximately 30×30 but was resized to 50×50 pixels with the help of a built in command in MATLAB (`imresize.m`) in order to make the network function. The network is dependent on that each superpixel is of the same size to ensure that the output from the network is of the correct size. We can follow, in Table 1 how the size of the output from each step changes by looking at the fifth column. The superpixels were processed by the CNN to get a classification of each superpixel. With this initial classification we can introduce our two proposed spatial biases to get the final classification. Since we are dealing with textures that cover larger areas we can expect that neighbouring pixels, with a high probability, should be classified the same. We will also assume that the SLIC algorithm, the superpixel creator, will conform well with boundaries, just as the authors show [1]. Without this assumption many pixels will contain data from two different classes and therefore be very difficult to classify.

In accordance with the suggestion Karpathy gives in [8] we will construct the network as a flow of convolutional layers followed by ReLU units and sometimes also followed by a max pool layer. The complete structure is shown in Table 1. The second column describes what type of layer we have. The third and fourth column describes the general size of filters, while the last describes the total size of the activation map once that layer has been calculated.

Table 1: *Complete structure of the CNN with all layers. The convolution layers 12 and 14 are known as the fully connected layers as all neurons have connections with all the data from the previous layer. As explained in section 2.3 this is why the dropout layers are placed in layer 11 and 13.*

#	**Type**	**Size**	**# filters**	**Data size**
0	Input			$50 \times 50 \times 3$
1	Conv	$6 \times 6 \times 3$	64	$45 \times 45 \times 64$
2	ReLU			$45 \times 45 \times 64$
3	Conv	$5 \times 5 \times 64$	64	$41 \times 41 \times 64$
4	ReLU			$41 \times 41 \times 64$
5	Max pool	3×3, stride 2		$20 \times 20 \times 64$
6	Conv	$6 \times 6 \times 64$	64	$15 \times 15 \times 64$
7	ReLU			$15 \times 15 \times 64$
8	Conv	$5 \times 5 \times 64$	64	$11 \times 11 \times 64$
9	ReLU			$11 \times 11 \times 64$
10	Max pool	3×3, stride 2		$5 \times 5 \times 64$
11	Dropout	Drop rate: 30%		$5 \times 5 \times 64$
12	Conv	$5 \times 5 \times 64$	256	$1 \times 1 \times 256$
13	Dropout	Drop rate: 30%		$1 \times 1 \times 256$
14	Conv	$1 \times 1 \times 256$	15	$1 \times 1 \times 15$
15	Loss	Soft-max-loss		$1 \times 1 \times 1$

3.3 Texture choices

We choose the textures, as we are also creating the ground truth. The choices are based on what type of images there are and what reasonable textures we would want to detect. The textures are exclusively outdoor textures since there are usually many objects that would need separate classification indoors. The following 15 textures were considered:

- **Grass:** All types of grass and cultured farmland.
- **Water:** All types of water, oceans, lakes, rivers, ponds, swimming pools etc.
- **Forest:** Forests and stand alone trees, as long as they are not bushes.
- **Bushes:** Bushes, not trees.
- **Other vegetation:** All other vegetation including cactuses and water lily pads.
- **Soil/Dirt:** Soil and dirt, but also mud, earth and non-cultured farmland.
- **Sand:** Beaches, desert etc.
- **Sky/Clouds:** Everything in the skies including clouds.
- **Mountain:** Mountains and mountain ranges.
- **Snow/Glacier:** All types of snow and ice.
- **Stone/Rock:** Stone and rock formations that are not considered mountains.
- **Crowd (of people):** Large groups of people.

- **Wall:** Background walls in images, when they are not considered objects (in a house for example).
- **Urban area:** Everything man made that is not considered an object, not including walls.
- **Other Object:** Cars, people, houses, animals etc.

Many of these textures require subjective analysis from the creator of the ground truth in order to assign the correct class, for instance if something is an object or not, or if we have a tree or a bush. These very probable clashes are chosen consciously in order to test the performance of the network. A human should differentiate a tree from a bush but it is more interesting to know how the network performs. As we are aware of these potential problems we will also train a network to only distinguish a few textures. Here we mash similar textures together into one:

- **Vegetation:** All types of vegetation.
- **Water:** All types of water, oceans, lakes, rivers, pond, swimming pools etc.
- **Earth:** All types of soil, dirt and mud etc.
- **Sky/Clouds:** Everything in the skies including clouds and blue skies.
- **Mountains:** Mountains, hills, rock formation, glaciers and snow.
- **Urban area:** Everything man made that is not considered an object.
- **Other Object:** Cars, people, houses, animals etc.

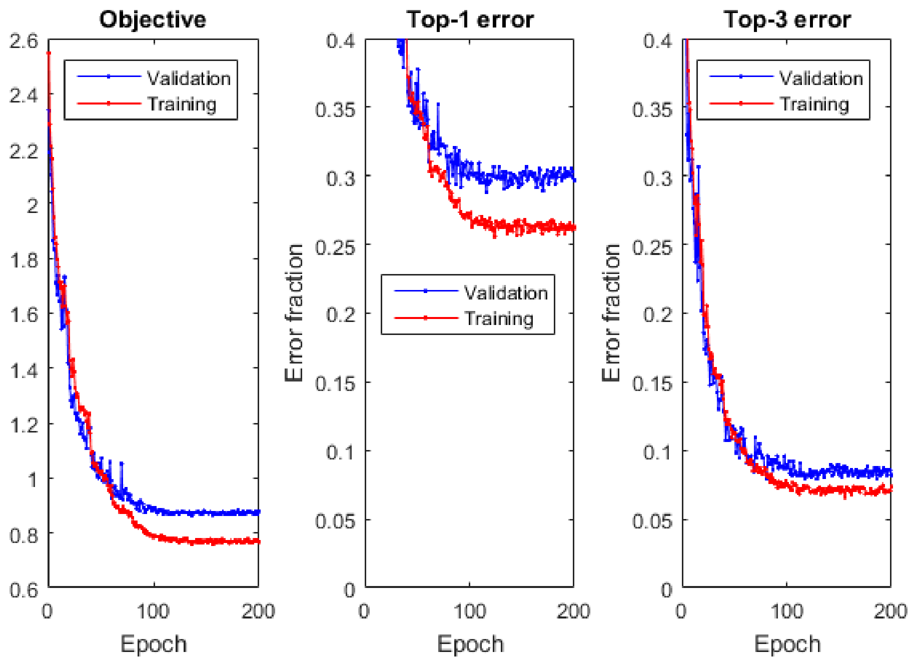
These new textures are chosen because of two reasons. First, textures that probably will be miss-classified to each other are collected and secondly, textures that describe similar things are collected. These two reasons are sometimes mutual, as with for example Forest and Bushes, where we can expect some problems. When considering Snow and Mountains we can expect few or none miss-classifications as a mountain is grey and uneven while snow is white and smooth, but they are usually found in similar places. Some textures, for example Water and Sky/Clouds, were kept apart from each other even though there is a risk of miss-classifications since the meaning of the classification would be lost if fewer than these classes are used. It is very reasonable to assume that this smaller network will have a lot fewer classification errors, however, the classification will of course be less detailed.

4 Results

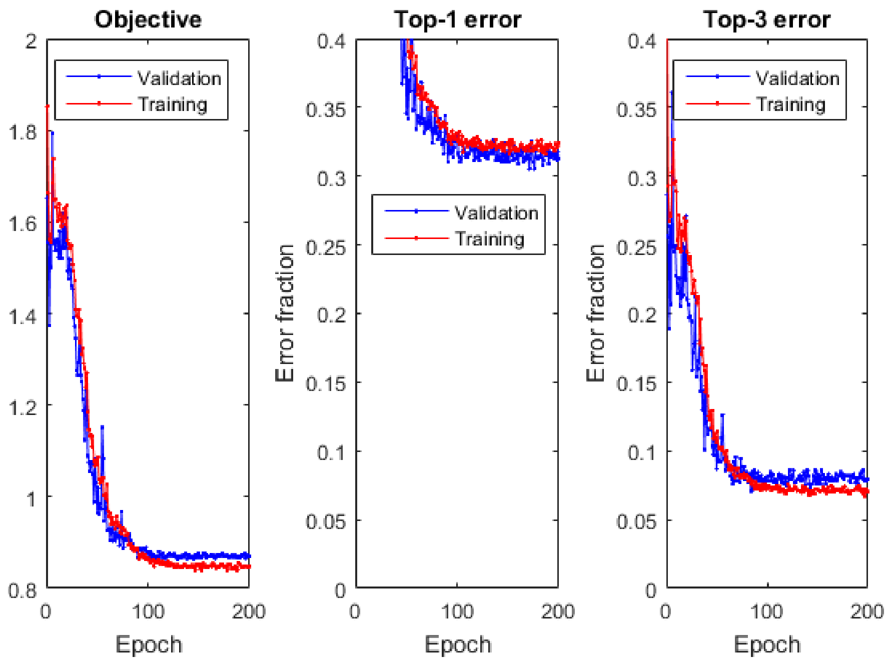
The results are divided into three parts. First we present the overall performance of the network with general error rates for the network. Then we investigate the network's performance on entire images and visually compare resulting segmentations to each other. Finally we will show in what manner the network makes its errors. At first, in this results section, we will see the number of right and wrong but in the final subsection to the results we will investigate what types of errors the network makes and also what errors it never makes.

4.1 Overall performance of the network

In Figure 7 we see the training results for the classification network with both 15 classes (in 7a) and 7 classes (in 7b). We can see that the error rate for the Top-1 error, i.e. the firsthand choice, is approximately 30% for both 15 and 7 classes. The Top-3 error, where the error rate is shown for the Top 3 choices of the network, is hardly surprising lower at 10%. This value is crucial for the static bias, as we presented in Section 2.4, where a low top-3 error will increase our chances of getting good results. We can also note the leftmost plot in Figures 7a and 7b, the objective function. It is the measurement from Section 2.3.2 as a total measure of how the network is performing. We that the starting points for the objective functions are close to the analytical result for random weights and that they are getting smaller.

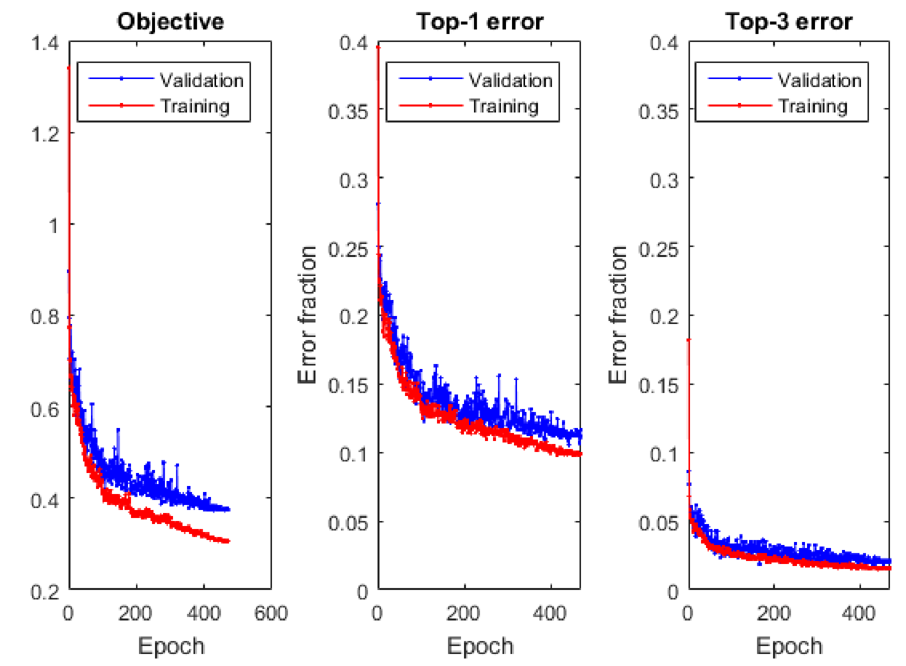


(a) *Training and validation errors with 15 texture classes. Shown is both the Top-1 error (left) and Top-3 error (right).*

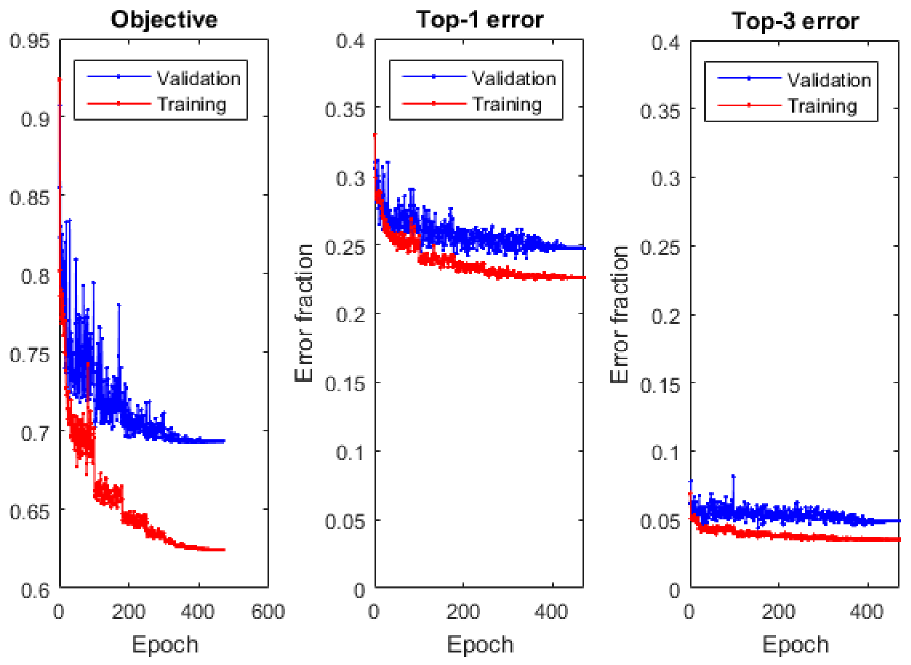


(b) *Training and validation errors with 7 texture classes. Shown is both the Top-1 error (left) and Top-3 error (right).*

Figure 7: *The error of the classification network.*



(a) *Training and validation errors of the bias network for spatial dependence with 15 texture classes. Shown is both the Top-1 error (left) and Top-3 error (right).*



(b) *Training and validation errors with 7 texture classes. Shown is both the Top-1 error (left) and Top-3 error (right).*

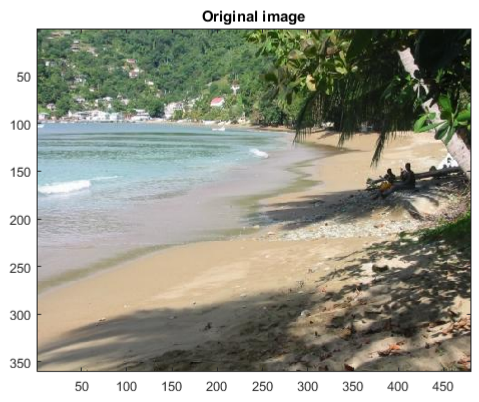
Figure 8: *The error of the bias network for spatial dependence.*

The network for spatial bias had as its input the output from the classification network. A trivial network would get the same error rate as in Figure 7 and therefore it is obvious that if the spatial network were to be of any use, it needs to drop the error rate to under 30%. In Figure 8 we can see that the Top-1 and Top-3 errors are lower for both 15 and 7 classes compared to Figure 7. The Top-1 error has now dropped to approximately 10% instead of 30%, and a similar story is true for the Top-3 error.

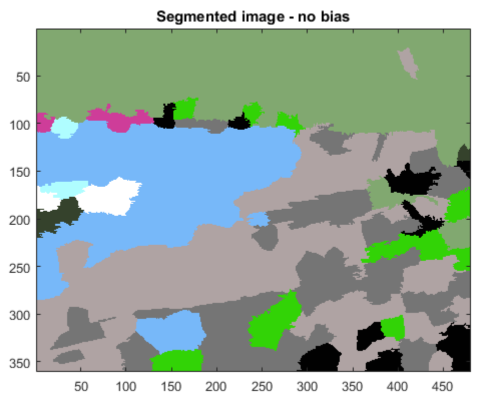
4.2 Visual illustration of program performance

Now we have seen numbers and percentages of how the network performs of a abstract level. In the real world and in the case of segmentation we are interested to see how the network performs on real images and how the results look.

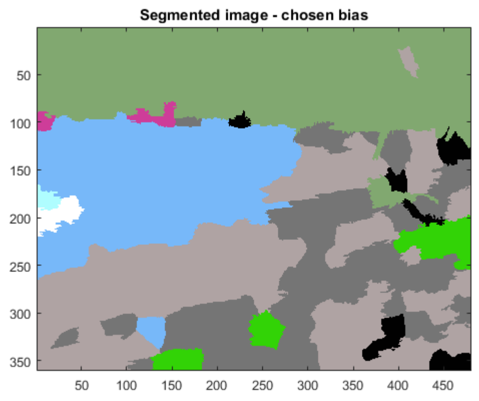
In Figure 9 we can see the suggested classification from the classification network (in 9b), the static bias (in 9c) and the network bias (in 9d) along with their error rates. All results can distinguish the big forest in the background and all of them detect that there is water and something else in the image. Both with no bias and with the static bias a lot of the sand (light grey) is instead classified as stone (dark grey), however the spatial network removes almost all of the stone. Every network have some issue with the breaking waves. With no bias and the static bias we get some sky (light blue) and snow (white) in the water (blue), while the spatial network removes the snow.



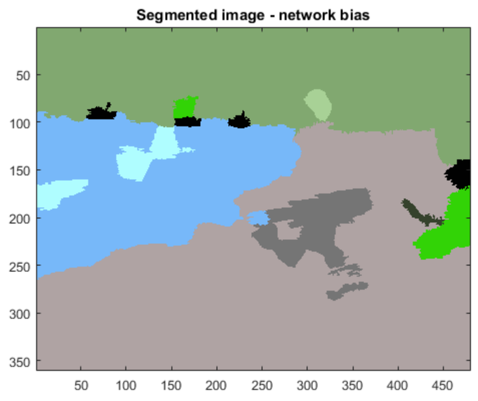
(a) *The original image, a tropical beach.*



(b) *Results from the classifier network. Total error rate: 36%*



(c) *Results with the static bias. Total error rate: 32%*



(d) *Results with the spatial network. Total error rate: 12%*

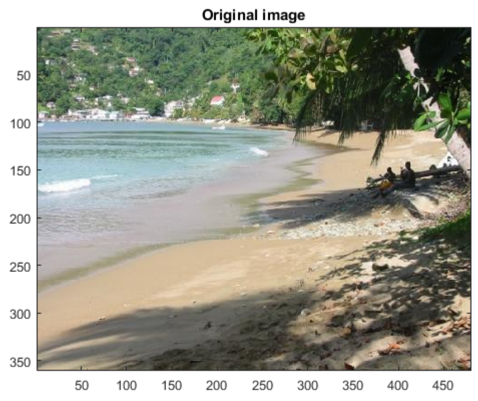
Figure 9: *Results from an image belonging to the training set with all 15 classes. Bright green is Grass, green is Forest, light gray is Sand, dark gray is Stone, blue is Water, light blue is Sky/Clouds and black is Urban area.*

One other aspect to investigate is the comparison between Figure 9c, static bias and Figure 9d, the spatial network. Both of them are reducing the error rates, however, the spatial network has almost a third of the error rate compared to the static bias.

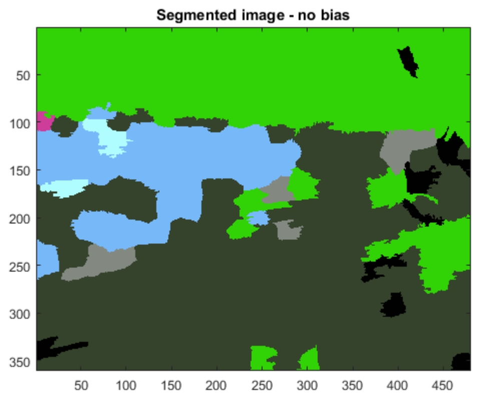
The results from Section 4.1 had similar error rates with 7 or 15 classes, however, when using images the segmentation with 7 classes yields higher error rates than all the methods for 15 classes, as can be seen in Figure 10. Precisely as in Figure 9 all methods can distinguish the vegetation in the background, however a large part of mountains can be seen (dark green) where it should be sand and water. The resulting error rates are higher for every method while using the smaller network with 7 textures compared to 15 textures.

The networks were then tested on images not belonging to any training data or similar to any training data to see how well they performed on unknown data. As the image is unknown no ground truth exists and therefore we have no error rates. Even though the error rates are missing we can visually compare the methods in Figure 11. It is obvious that the landing pool for the waterfall have not been classified correctly. We have either sand or stone from all methods and no water. All methods are more or less capable of classifying the vegetation in one way or another, while the waterfall itself is not consistently classified as water. The same problems exist for the sky.

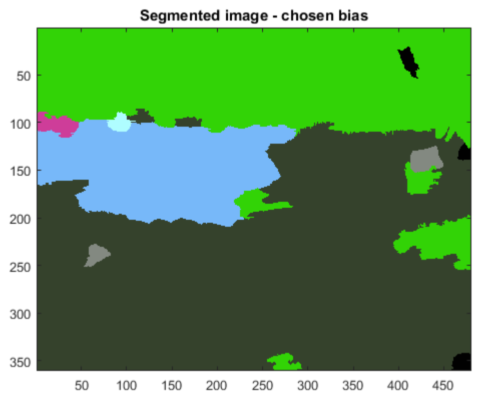
When using 7 classes in Figure 12 we do not get any errors on the type of vegetation, as there is only on class, however the water in the landing pool is still considered to be mountains (including stone) or earth (including sand).



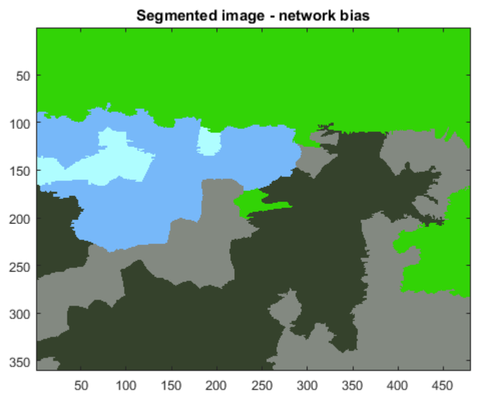
(a) *The original image, a tropical beach.*



(b) *Results from the classifier network. Total error rate: 59%*

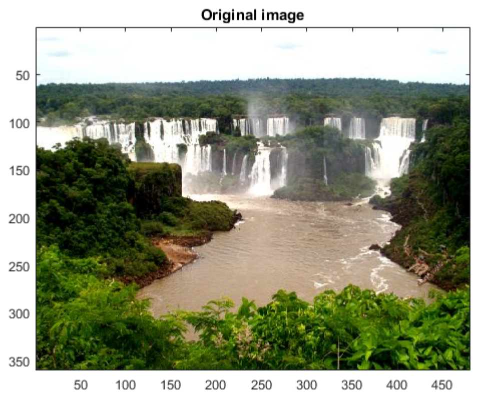


(c) *Results with the static bias. Total error rate: 61%*

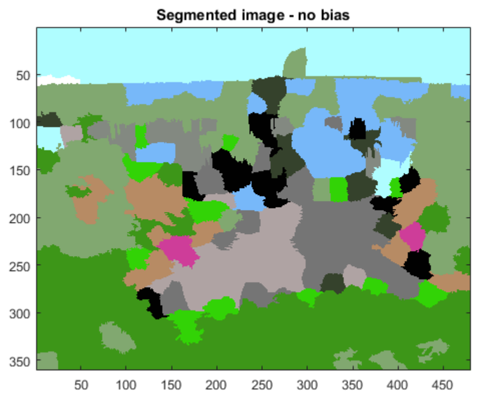


(d) *Results with the spatial network. Total error rate: 42%*

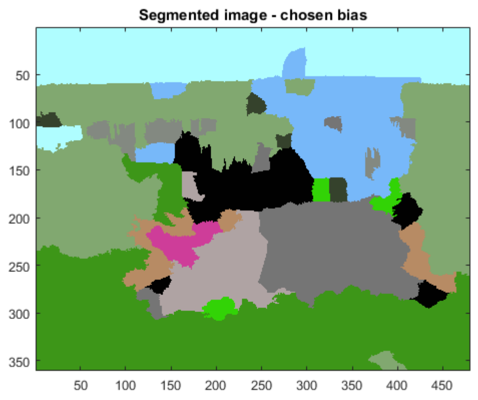
Figure 10: *Results from an image belonging to the training set with 7 classes. Light green is Vegetation, dark green is Mountains, grey is Earth, blue is Water and light blue is Sky/Clouds.*



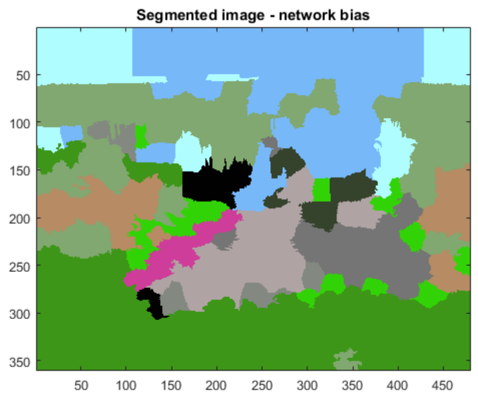
(a) *The original image, a waterfall in a djugle.*



(b) *Results from the classifier network.*



(c) *Results with the static bias.*



(d) *Results with the spatial network.*

Figure 11: *Results from an image unknown to the network with 15 classes. Bright green is Grass, pale green is Forest, green is Other vegetation, light gray is Sand, dark gray is Stone, blue is Water, light blue is Sky/Clouds and black is Urban area.*

4.3 Different types of miss-classifications

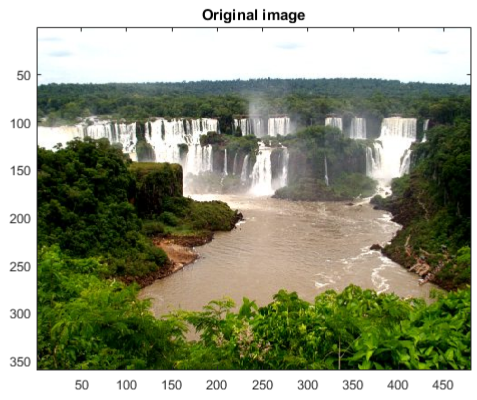
As mentioned in earlier sections the types of error can be considered. Here we show the results when testing all textures one at a time to see what type of textures the network considers. In Table 2 we see these errors when using all 15 classes and in Table 3 when using 7 classes. We find the correct classification rates on the diagonals of the tables and the errors on each row, i.e. given that a pixel belongs to class i, it will be classified as class j with the chance found in row i and column j.

In table 2 we see that the best type of superpixel, i.e. the type that gets the least errors is Snow. More than 96% of all superpixels with snow are correctly classified as snow. The worst pixel is bushes since every pixel that is bushes will be classified as forest. Apart from the bushes Soil is the worst with 47% correct classification, where many erroneous classifications are stone (19%). Even though soil has a low correct classification rate, there are few pixels that are not soil who are miss-classified as soil. Urban area an Forest are the two textures where almost every other texture runs a risk of miss-classifying a superpixel as one of them.

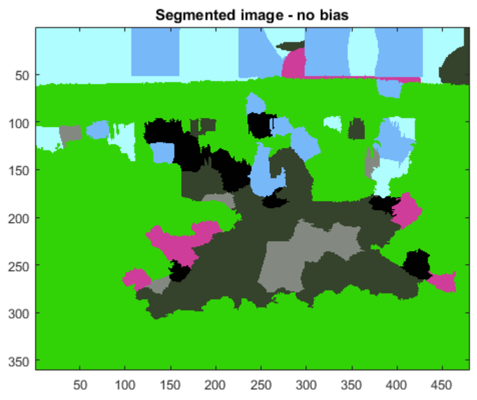
The much smaller Table 3 has all error types collected for 7 classes. Here the best texture type is mountain with a little more than 84% correct classifications, while the poorest performing texture is Other objects with 43% correct classifications.

Table 2: *Proportion of classifications for 15 classes. Given that a pixel belongs to class i, it will be classified as class j with the chance found in row i and column j, i.e. the chance of a correct classification can be found on the diagonals.*

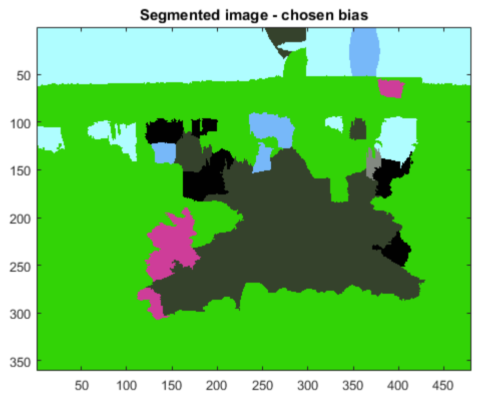
Texture	**Grass**	**Water**	**Forest**	**Bushes**	**Soil**	**Sand**	**Sky**	**Mount.**	**Snow**	**Stone**	**Crowd**	**Wall**	**Objects**	**Oth. veg.**	**Urban**
Grass	0.7679	0	0.1071	0	0	0.0179	0	0	0	0.0714	0	0	0.0179	0	0.0179
Water	0	0.7213	0.0984	0	0	0.0164	0.0164	0.1184	0	0	0	0.0164	0	0	0.0164
Forest	0	0.0230	0.6897	0	0	0	0.0230	0.0115	0.0115	0.0690	0.0345	0	0.0115	0.0575	0.0690
Bushes	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
Soil	0	0	0.0476	0	0.4762	0.0476	0	0.0476	0	0.1905	0	0	0	0	0.1905
Sand	0	0	0	0	0	0.7143	0	0	0	0.1964	0.0179	0	0.0179	0	0.0536
Sky	0	0.0522	0	0	0	0	0.8731	0.0224	0.0224	0	0	0	0.0224	0	0.0075
Mount.	0.0625	0.0417	0.0417	0	0	0.0417	0.0625	0.5208	0	0.1250	0	0.0208	0	0	0.0833
Snow	0	0.0323	0	0	0	0	0	0	0.9677	0	0	0	0	0	0
Stone	0.0299	0.0149	0.0448	0	0.0448	0.0746	0	0.0597	0	0.5821	0.0299	0	0.0149	0	0.1045
Crowd	0	0	0.0952	0	0	0	0	0	0	0	0.8571	0	0	0	0.0476
Wall	0	0.0500	0	0	0	0.0250	0.0500	0.0500	0	0.0250	0	0.7250	0.0500	0	0.0250
Objects	0.0519	0.0130	0.0909	0	0	0.0260	0.0130	0	0.0390	0.0130	0	0.0649	0.5064	0	0.1818
Oth. veg.	0	0	0.1250	0	0	0	0	0	0	0	0.0208	0	0.0417	0.7917	0.0208
Urban	0.1136	0.0227	0.0909	0	0.0227	0	0	0.0909	0	0.1136	0.0227	0	0.0455	0	0.4773



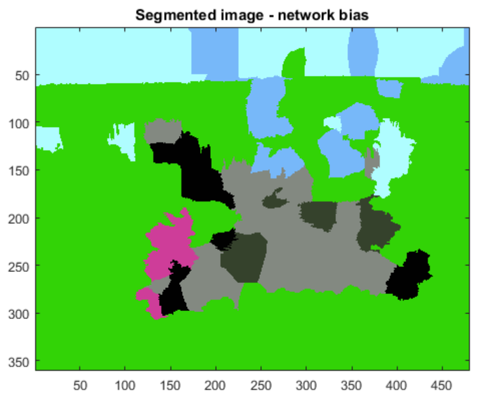
(a) *The original image, a waterfall in a djugle.*



(b) *Results from the classifier network.*



(c) *Results with the static bias.*



(d) *Results with the spatial network.*

Figure 12: *Results from an image unknown to the network with 7 classes. Light green is Vegetation, dark green is Mountains, grey is Earth, blue is Water and light blue is Sky/Clouds.*

Table 3: *Proportion of classifications for 7 classes. Given that a pixel belongs to class i, it will be classified as class j with the chance found in row i and column j, i.e. the chance of a correct classification can be found on the diagonals.*

Texture	**Vegetation**	**Water**	**Earth**	**Sky**	**Mountain**	**Urban area**	**Other objects**
Vegetation	0.8063	0	0	0.0052	0.0995	0.0105	0.0785
Water	0.1667	0.5303	0.0152	0.0758	0.1818	0	0.0303
Earth	0.0108	0.0215	0.5376	0	0.3548	0.0645	0.0108
Sky	0.0238	0.0397	0	0.7460	0.1508	0.0238	0.0159
Mountain	0.0255	0.0127	0.0191	0.0382	0.8471	0.0191	0.0382
Urban area	0.0588	0.0294	0.0294	0.0294	0.2647	0.4853	0.1029
Other objects	0.2282	0	0.0217	0.0109	0.1957	0.1087	0.4348

5 Discussion and Conclusions

Some of the results have been quite expected and some more surprising. In our training data we had very few superpixels depicting bushes and therefore it was not surprising to see that many were miss-classified. In fact, when we look at Table 2 we see that all superpixels with bushes in the validation set were classified as forest. When choosing the textures to investigate, we purposely divided trees and bushes into different texture classes in order to see if the network could distinguish them but at first glance this does not seem to be the case. When investigating further the number of pixels with bushes were a big problem. The sheer number of trees compared to bushes made it favourably for the network to never class anything as a bush and everything as trees, and still keep the error rate low. This is a very common problem when training neural networks if you do not have a balanced number of superpixels in each class. Everything regarding the network is however not bad, a good behaviour from the network can also be seen in Table 2. There are quite a lot of zeros in the table. This means that there are many types of pixels that are never classified as some other type of texture. For example if we look at the column with Soil, we have only non-zero elements in the rows with Soil, Stone and Urban. This means that if a pixel is classified as Soil, it will be either a correct classification, Stone or Urban, nothing else. We see the same behaviour in the columns with Grass, Sky and Snow to mention a few. Furthermore if we compare Grass with Crowd for example we notice that no Grass pixels are ever classified as Crowd or vice versa, they have apparently nothing in common. This happens with multiple other texture pairs. Comparing this behaviour with many zeros with the network with 7 classes in Table 3 there are hardly any zeros at all and there are no pairs of texture never classified as each other.

In Section 4.1 we can compare the results of the classification network and the network for spatial bias for 15 and 7 classes in Figures 7 and 8. Worth noting is that the Top-1 error and Top-3 error behaves roughly the same, with roughly the same errors when comparing 15 and 7 classes, both with the classification network and the spatial bias. This leads us to believe that the networks for 7 and 15 classes will have the same error rates when tested on images, however, Figures 9 and 10 tells another story. Here we see a much higher error rate for 7 classes regardless of method, a very surprising result when considering the error rates in Figures 7 and 8. When we instead consider what we discussed in the previous paragraph this result should not come as a surprise, there are many more zeros in Table 2 than Table 3 and many of the classification rates for correct classification are surprisingly low. The interesting question that still remains unanswered is why 7 classes behaves in a much worse way with the images compared to 15 classes despite similar error rates in Figures 7 and 8.

A very pleasing result is that the network for spatial biases reduces error rates more than the static bias. The spatial network can handle more different cases than the static bias. This shows the flexibility of the network compared to the static bias. However an alarming behaviour can be seen in Figure 11d, where errors seems to increase compared to the static bias, or at least not reducing the errors. It would seem as the spatial network is overfitted to the training data. It is not especially overfitted when we see Figure 8, however, other problems have possible occurred. We will get back to this thought later in this section. Another observation we can do with the spatial network is that is does not smooth out the image just as much as the static bias. We noted in Section 2.4 that a desired trait of the images is smoothness, however, too much smoothing will cause us to smooth over smaller objects or narrow texture areas. A drawback to this lack of smoothing from the spatial network seem to be some pixels changing texture classification in a weird and unpredictable way.

Earlier, we stated that the superpixels will conform well to borders, just as the authors of the article explained [1], and that this is a necessity for us to not have to deal with multiple textures in each superpixel. This statement has not always worked and the part we described as a necessity has not always been a necessity. For example, fuzzy lines on the horizon or similar clothing on people compared to their surroundings were issues that caused problems when the superpixels were created. Some superpixels contained both sky and water and some others contained both grass and trees. This caused some problems when ground truth was created, since the creator of the ground truth needed to subjectively choose which texture it depicted most. An idea, that never was implemented, was to simply throw away these superpixels from the training set. This could have been successful, however it would still need a subjective choice of "throw away" or not. The second reason forit to not be implemented was due to the lack of superpixels, we needed everyone we could get our hands on. A more reasonable solution to investigate in the future could possibly be to reduce the size of the superpixels or even change the tunable constant in the SLIC algorithm to promote close colour proximity rather than spatial proximity. We would get more jagged superpixels, but they would hopefully only contain one texture. When considering changes to the superpixels we could also choose not to enlarge them as much as we do now. This would have no effect on the superpixel quality, but by halving the size to 25×25 or even smaller we could save a few hundred thousand weights, which would lead to quicker training and possibly less overfitting.

When designing new algorithms it is common to blame your data for many problems, such as lack of data or contradicting data but in this case the data is really a problem. In this project we trained a network using 7967 superpixels from 31 images, a tiny amount of data. The benchmarks tests mentioned in the Introduction and other tests not included have training sets with 50.000, 100.000 or even millions of data points. Some of these are of course tests with a lot more classes but this still brings our size of the training set into perspective. The root of the problem is the low number of images used. As an example, all the superpixels of Sand are from just two or three of these images and it is very naive to think that all types and all looks of sand can be contained in just a few images, we should probably have had at least 31 images just for the sand. This problem exists for all textures. We can clearly see the issues when looking at Figures 11 and 12, where the water is completely wrong. The network either thinks it is sand or mountains which is totally wrong. This is an example of what happens when the data for water is not diverse enough. This lack of data can explain part of the problem where we have low validation error in Figure 7 but still high errors with other images, such as Figure 11. Our network is fairly stable when we only consider the type of data we have given but since the validation data shown in Figures 7 and 8 we are tricked by Figures 7 and 8 despite that the network is overfitted.

5.1 Future work

In this project we have segmented and classified the background textures and left the objects as unknown. An obvious next step is of course to try to classify these objects as well. One method can be to use a pretrained object classifier, for example by Krizhevsky et al. [9] or a newer network. When an object is detected a portion around this detected object can then be cut out from the image and classified through the object classification network. We will then have a complete segmentation of the entire image. This will mean that there is a third stage with a third network in this classification program, an obvious disadvantage concerning memory and calculation time. If the objects were to be handled a different method would possibly have been more suitable.

As we mentioned in the discussion the shortage of data was an issue when the network was trained. In the future it would be very interesting to test how the network's performance with a larger data set. This will demand more manually classified ground truth, a process that is already tiresome.

In the very beginning of this thesis we mentioned Tivive and Bouzerdoum [18] who, in their article, showed the advantages of using CNNs over traditional texture classification methods. It would have interesting, as an extra comparison, to compare our classifications with a neural network to a good classification algorithm based on classical features.

5.2 Conclusions

In the introduction we said that an important question to investigate with this thesis was whether our CNN could differentiate similar looking textures. The short answer is yes and no. The long answer is that our CNN could distinguish sky and water in many cases and only had serious trouble with breaking waves. Trees and bushes were to textures that was not differentiated well. The second part of the question was what types of errors the network made. Some errors were expected, such as trees and bushes, while other were more unexpected. The issues with breaking waves and snow was unexpected but when you gave it a second thought it was fairly reasonable.

To conclude and summarise this thesis we should remember three topics for the future. First of all, the classification network with the added spatial network did classify many of the textures and did achieve a fairly low error rate in the end. Secondly we do have to mention the superpixels. If we would have used smaller superpixels with more emphasis on colour proximity than we had here. By using smaller superpixels the creation of ground truth would demand fewer subjective choices from the creator. Thirdly and to finalise this report we do mention the low error rates we had and see it as a good sign to continue investigating CNN for future classification programs.

References

[1] Radhakrishna Achanta, Appu Shaji, Kevin Smith, Aurelien Lucchi, Pascal Fua, and Sabine Suüsstrunk. Slic superpixels compared to state-of-the-art superpixel methods. *IEEE Transactions on pattern analysis and machine intelligence*, 34(11):2274–2281, November 2012.

[2] Anna Choromanska, Mikael Henaff, Michaël Mathieu, Gérard Ben Arous, and Yann LeCun. The loss surface of multilayer networks. *CoRR*, abs/1412.0233, 2014.

[3] Hao Fang, Saurabh Gupta, Forrest N. Iandola, Rupesh Kumar Srivastava, Li Deng, Piotr Dollár, Jianfeng Gao, Xiaodong He, Margaret Mitchell, John C. Platt, C. Lawrence Zitnick, and Geoffrey Zweig. From captions to visual concepts and back. *CoRR*, abs/1411.4952, 2014.

[4] David B Fogel, Derong Liu, and James M Keller. *Multilayer Neural Networks and Backpropagation*, pages 35–60. John Wiley & Sons, Inc., 2016.

[5] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Aistats*, volume 15, page 275, 2011.

[6] Michael Grubinger, Paul Clough, Henning Müller, and Thomas Deselaers. The iapr tc-12 benchmark: A new evaluation resource for visual information systems. *International Conference on Language Resources and Evaluation*, May 2006.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *The IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[8] Andrej Karpathy. Cs231n convolutional neural networks for visual recognition - lecture notes. *http://cs231n.github.io*, 2017.

[9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[10] Yann LeCun. Generalization and network design strategies. *Technical Report CRG-TR-89-4*, 1989.

[11] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. In *Nature*, volume 521, pages 436–444, May 2015.

[12] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014.

[13] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *CoRR*, abs/1511.08458, 2015.

[14] Dominik Scherer, Andreas Müller, and Sven Behnke. *Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition*, pages 92–101. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[15] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Transactions on pattern analysis and machine intelligence*, 22(8):888–905, August 2000.

[16] Princeton University Stanford Vision Lab, Stanford University. Imagenet. *http://image-net.org/index*, 2016.

[17] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

[18] F. H. C. Tivive and A. Bouzerdoum. Texture classification using convolutional neural networks. In *TENCON 2006 - 2006 IEEE Region 10 Conference*, pages 1–4, Nov 2006.

[19] A. Vedaldi and K. Lenc. Matconvnet – convolutional neural networks for matlab. In *Proceeding of the ACM Int. Conf. on Multimedia*, 2015.

[20] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. *CoRR*, abs/1411.4555, 2014.

[21] J.S. Denker D. Henderson R.E. Howard W. Hubbard L.D. Jackel Y. LeCun, B. Boser. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1:541–551, 1989.

[22] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. *CoRR*, abs/1511.07122, 2015.