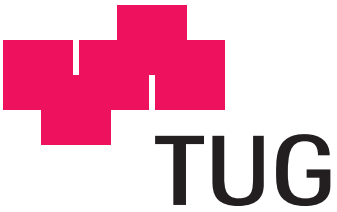
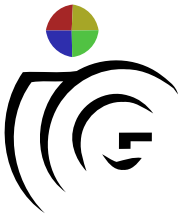


Graz University of Technology

Institute for Computer Graphics and Vision

Microsoft Photogrammetry

Master's Thesis

---

# Interactive Semantic Segmentation on Aerial Images

---

**Patrick Knöbelreiter**

Graz, May 2014

Supervisor:
Univ.-Prof. Dipl.-Ing. Dr.techn. Horst Bischof

Advisor:
Dipl.-Ing. Dr.techn. Christian Leistner

To my family.

It is not knowledge, but the act of learning, not possession but the act of getting there, which grants the greatest enjoyment.

---

*Carl Friedrich Gauss*

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

## EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am …… ……..
(Unterschrift)

Englische Fassung:

## STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

…… ……..
date (signature)

# Acknowledgments

First of all, I want to thank Konrad Karner from Microsoft Photogrammetry for giving me the opportunity to do my Master's Thesis in cooperation with such a great company. Special thanks go to Christian Leistner for his excellent mentoring and support. Independent of the difficulty of a question, he was always able to give me a clear and logical answers, which made my life a lot easier. Additionally, I want to thank Markus Unger, who is brilliant in everything considering GPU programming, Lukas Zebedin for a lot of subtle tips and tricks and the whole Photogrammetric Computer Vision team.

Last but not least I want to thank my supervisor Professor Horst Bischof from the Institute for Computer Graphics and Vision for the excellent support in technical and administrative issues.

# Abstract

Extracting semantic information out of images is one of the most challenging problems in computer vision. The goal is to identify all foreground and background objects visible on an image simultaneously. Every pixel on the image gets a logical class label assigned, such that a pixel accurate segmentation results, which is referred to as a semantic segmentation. Deriving a semantic segmentation of arbitrary input images allows a computer not just to see the images with a camera, but also to understand what the content of an image actually is. This allows to automate a lot of things. However, viewpoint variations, occlusions and different scales make semantic segmentation a very complex task.

In this thesis, a semantic segmentation of aerial images should be computed using an interactive approach. This requires the classifiers to be very fast at test time, such that the user gets immediate feedback after the classifier has been updated. Random forests and random ferns are classifiers fulfilling this property and have therefore been used in this thesis for the classification task. It is shown how random forests and random ferns can be used online, such that new training data can be incorporated at any time. To keep the interaction necessary as little as possible, a concept called active user guidance has been developed. This concept allows the user to update the classifier with those samples, which will have the greatest impact on performance.

With the application it is possible to semantically segment complete aerial projects in 2D as well as in textured 3D. Projects in 3D allow to incorporate additional features like pixel synchronous surface normals for example, which are highly discriminative and therefore a powerful information source for semantic segmentation.

**Keywords:** Interactive, Semantic Segmentation, Scene Parsing, Random Forest, Random Fern, Machine Learning, Aerial Photos

# Kurzfassung

Das Extrahieren von semantischer Information aus Bildern ist ein schwieriges Problem der Bildverarbeitung. Ziel ist es gleichzeitig alle Objekte, die sich sowohl im Vordergrund als auch im Hintergrund befinden, zu identifizieren. Das bedeutet, dass jeder einzelne Pixel des Bildes eine logische Kategorie zugewiesen bekommt. Das Ergebnis wird als semantische Segmentierung eines Bildes bezeichnet. Mit Hilfe einer semantischen Segmentierung ist es für einen Computer nicht nur möglich die Bilder mit einer Kamera zu betrachten, sondern auch zu erkennen, welche Objekte sich wo auf dem Bild befinden. Aufgrund unterschiedlicher Blickpunkte, Verdeckungen und unterschiedlicher Skalierungen ist die Berechnung einer semantischen Segmentierung ein sehr komplexes Problem.

In dieser Arbeit wird ein interaktiver Ansatz gewählt, um eine semantische Segmentierungen von Luftbildern zu berechnen. Dies impliziert, dass die Evaluierung von neuen Bildern sehr schnell erfolgen muss, sodass der Benutzer das Ergebnis seiner Interaktion unmittelbar sehen kann. Random Forests und Random Ferns sind Klassifikatoren, die diese Anforderung erfüllen und wurden deshalb in dieser Arbeit verwendet. Es wird gezeigt wie Random Forests bzw. Random Ferns online - neue Trainingsdaten können zu jedem Zeitpunkt in den Klassifikator integriert werden - verwendet werden können. Um die notwendige Interaktion so gering wie möglich zu halten wurde ein Konzept entwickelt, das dem Benutzer zeigt, welche annotierten Pixel die größte Qualitätsverbesserung bewirken.

Mit dieser Anwendung ist es möglich eine semantische Segmentierung von Luftbild-Projekten sowohl in 2D als auch in texturiertem 3D zu erstellen. Wenn es sich um ein Projekt in 3D handelt, ist es außerdem möglich, Normalvektoren der Pixel als zusätzliche Informationsquelle zu verwenden.

# Contents

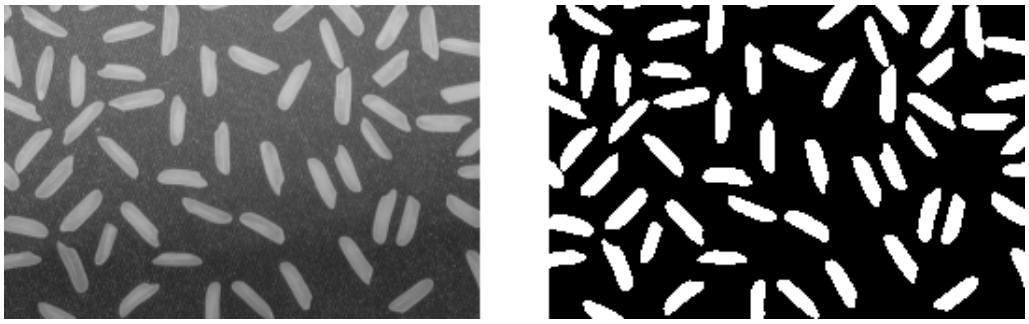
# Chapter 1

# Introduction

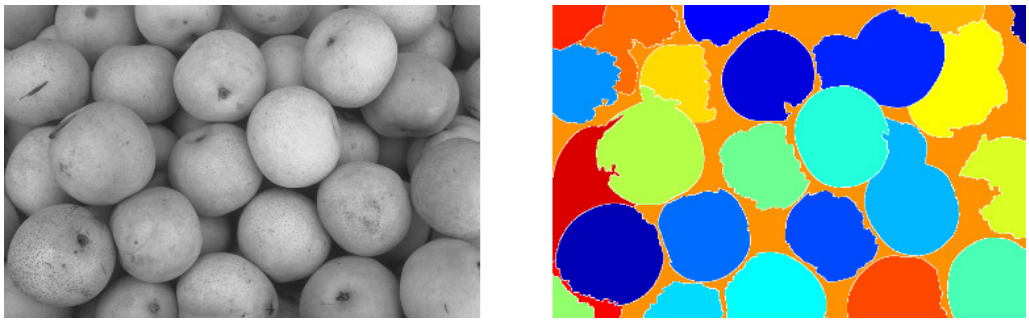
A fundamental problem in computer vision is image segmentation. Image segmentation is the task of grouping logical corresponding pixels in an image together. That is, for example, if pixels have a strong correlation to objects or regions in real world images, they should correspond to one group of pixels. Image segmentation is an important preprocessing step for further image analysis including classification, recognition, etc., i.e., it is the first step from raw pixel-based color information towards more meaningful information. There exist a lot of basic low level segmentation algorithms such as *thresholding*, *k-means clustering* or *watershed segmentation*. Figure 1.1 shows examples of these segmentation algorithms. For a more complete list and more details about basic image segmentation algorithms see [68], for a survey of segmentation methods see [41].

However, these basic methods are very sensitive to the parameters used. Bad image quality accompanied with noise, poor contrast, weak boundaries and highly textured objects make segmentation a very hard task. To overcome these difficulties, in the last decades, a lot of research has been done in this area and many more sophisticated image segmentation algorithms evolved (Level Sets [12], Graph-based segmentation [19, 64], Mean-Shift [11], Markov Random Fields [7], Variational Formulations [48]).

The huge number of publications reflects the importance of image segmentation and that it is not solved completely yet. However, for some very specific tasks the segmentation results are pleasing. Applications of image segmentation are found in various fields, like in aerial image processing, object recognition, industrial computer vision, medical image analysis and many others.



(a) Thresholding



(b) Watershed segmentation


(c) K-Means clustering

Figure 1.1: Examples of basic image segmentation algorithms. Image courtesy of [44], [43] and [5].

## 1.1 Definition of Image Segmentation

In mathematical terms, images are functions $I : \Omega \rightarrow \mathbb{R}^c$, where $\Omega$ is called the image domain. The dimensionality of $\Omega$ is 2 for 2D images. For gray-scale images $c = 1$ and for color images $c = 3$. Every pixel on an image is assigned to a region $\Omega_i$, where different

regions do not overlap. Hence, image segmentation can be defined as

$$\Omega = \bigcup_{i=1}^{K} \Omega_i, \qquad \Omega_i \cap \Omega_j = \emptyset \quad \forall i \neq j \tag{1.1}$$

All pixels in a region $\Omega_i$ should be homogeneous and different from surrounding regions $\Omega_j$. If $K = 2$, the segmentation is called binary segmentation. The aim of binary segmentations is to partition an image into foreground and background. For $K > 2$, the problem is called multi-label segmentation. The reminder of this work will deal with multi-label segmentations, if not other specified.

From the definition above, it can be seen that image segmentation is ambiguous, i.e., many segmentations do exist, which fulfil Equation (1.1). However, we are only interested in a small subset of them. More precisely, we want to get homogeneous region in some respect. This similarity could be color, spatial vicinity, or the correspondence to the same object for example. The question of which segmentations are "correct" solutions is not trivial to answer. Figure 1.2 shows some correct segmentations done by humans.

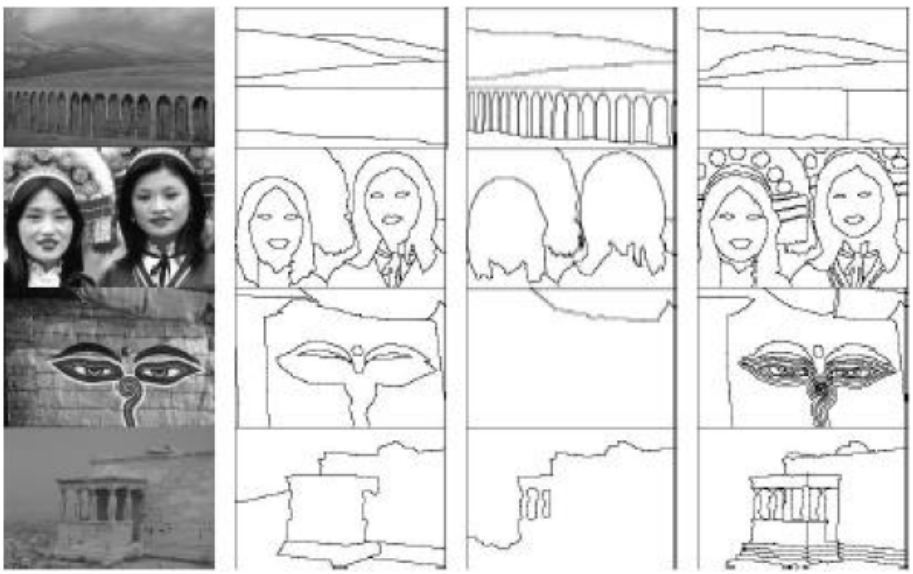


Figure 1.2: Segmentation ambiguity. Example of human-annotated images from the Berkley segmentation dataset [42]. Note the variation of annotations and the difference in the number of resulting segments. However, all of these segmentations are correct. Image courtesy of [3].

## 1.2 Machine Learning and Image Segmentation

Image segmentation can also be seen as a machine learning task. In general one distinguishes between *unsupervised* and *supervised* machine learning algorithms. The difference lies in the existence of class labels, where in the supervised setting target labels exist and in the unsupervised setting no class labels exist. In other words unsupervised machine learning algorithms are completely dependent on the data itself and supervised machine learning algorithms make use of additional information called labels. In the context of this work, machine learning algorithms are actually segmentation algorithms.

### 1.2.1 Unsupervised Segmentation

In the unsupervised segmentation configuration the target is to discover the data and group similar examples within the data together. Since in image segmentation the data is an image, the task is to group similar pixels together. The process of grouping similar data together is also called *clustering*. Actually, this fits exactly to the definition of image segmentation (see Equation (1.1)).

### 1.2.2 Supervised Segmentation

In the supervised setting, the input data comprises examples of the input vector along with their corresponding target vectors, i.e., the labels are available in addition. This enables the possibility to inject external information to the algorithms and therefore guide them and furthermore the resulting segmentation. Since the result of the segmentation are learnt class labels, supervised segmentation is a classification task. However, the result of this segmentation are not just regions as it is in the unsupervised setting, but regions with a label assigned to them. Therefore, also semantic information is incorporated into supervised segmentation which leads to a *semantic segmentation*. Semantic segmentation is discussed in Section 1.3.

### 1.2.3 Why Machine Learning

On this place, one could ask the question: "Why to use machine learning for semantic segmentation? Why don't use a sophisticated model designed specific for semantic segmentation?" Indeed, these questions are eligible. However, the answer to these questions is the following: It is unclear how to model different categories [39]. Take the category

WINDOW for example. The intra class variability is very high which makes it difficult or even impossible to define a model representing all variants of windows.

To overcome this problem a machine learning strategy is used. By using machine learning techniques, we want to learn what distinguishes different classes rather than manually specify their difference. The complexity of semantic segmentation and other recognition problems results from [39]:

- Viewpoint variations
- Different illuminations
- Occlusions
- Different scales
- Deformations of the objects
- Background clutter

Since the high complexity of semantic segmentation a lot of research focuses on machine learning techniques. Machine learning tackles these difficulties by trying to use statistical reasoning to find an approximate solution to the problem [33] (cf. Section 2.3.1).

## 1.3 Semantic Segmentation

Image segmentation (see Section 1.1) is the process of finding regions of corresponding pixels in an image. However, in many applications it is not enough to only know which pixels correspond to which segments in an image. More often, one wants to know which regions do correspond to which logical class in addition. In general, arbitrary class labels are possible, this is only a question depending on the task to be solved. Examples for such *semantic* classes could be

- Façade
- Door
- Window
- Roof
- Vegetation

- Street
- Car
- ...

In semantic segmentation all foreground (i.e., the "*things*") and background (i.e., the "*stuff*") objects, which are present on an image should be detected. From another point of view a thing is everything on the image, where it is possible to draw a bounding box around (e.g., car, window) and the stuff are the regions where it is not possible to draw a bounding box around (e.g., sky). However, one does not just want to detect the objects by drawing a bounding box around them, but to perform a pixel-wise segmentation. Hence, semantic segmentation tries to

- segment an image into regions, where each region represents either a thing or stuff
- find exact boundaries between things and stuff
- assign a class label to each region.

In other words, one could say that semantic segmentation turns visual data into a meaningful representation. Semantic segmentation performs a recognition task, which is in general very hard to do for computers, since a model does not exist. The step leads form numerical values, i.e., the colors of the pixels, to objects and meaningful regions present on an image.

By knowing what is on an image, it is possible to automate a lot of things. Semantic segmentation gives a computer with a camera not just the possibility to see, but also the possibility to understand the world as human do. This could be seen as one of the final targets of computer vision. With this ability a lot of things could be done automatically by computers.

**Semantic Segmentation on Aerial Images** In this work, the focus lies on aerial images. They differ from images shipped in a segmentation database. The images in labelling databases are almost designed for some predefined classes, i.e., that the images contain exactly the classes, for which one is looking. From a machine learning perspective of view, the most challenging part on aerial image is that there exist hardly any or even no ground truth annotations. An overview of differences is summarized in Table 1.1.


(a) Database image (b) Semantic segmentation

Figure 1.3: Example of a semantic segmentation. The semantic segmentation is pixel accurate, i.e., each pixel has a logical class label assign to it. Different colors encode different class labels. Image taken from the MSRC image database [57].

| | **Labelling database image** | **Aerial image** |
|---|---|---|
| Size | small - medium | medium - large |
| Resolution | medium - high | low - high |
| View | arbitrary | orthogonal, oblique |
| Visible Classes | predefined | arbitrary |
| Ground truth | available | not available |

Table 1.1: Differences of images from labelling databases and aerial images from a machine learning perspective of view.

How a semantic segmentation can be performed with a machine learning approach is discussed in Section 2. This application tackles the problem using an *online* approach (see Section 1.4 and Section 4). An example for an semantic segmentation on an aerial image is visualized in Figure 1.5.

## 1.4 Interactive Semantic Segmentation

In an interactive semantic segmentation system, the user has the possibility to interact with the algorithms. In fact, the user provides the labels for training and updating the classifier. Therefore, he can guide the algorithms what to learn. As illustrated in Figure 1.4, it is possible to distinguish different classes with very little annotation effort. A human operator has the possibility to guide the algorithm by intelligently providing the most useful samples as training data. For example, when the user provides the two red circles and the two green triangles (see Figure 1.4), then the classifier is able to separate the classes perfectly. These important samples defining the decision boundary are referred to as the *support vectors*. As can be seen from this toy example it is possible to achieve very good results

without the need of a huge amount of training data. In addition, by having *the user in the loop*, the ambiguity problem in image segmentation is not a problem anymore. The user can always teach the algorithms to learn, what he defines to be the ground truth. It is no problem, if this ground truth does change over time for example, since it is possible to update the knowledge of the classification algorithms on demand. Unsupervised methods are heavily dependent on the parameters used, like the number of segments in the final segmentation. In contrast, by using an interactive approach, the number of segments is implicitly defined by the number of labels the user uses.

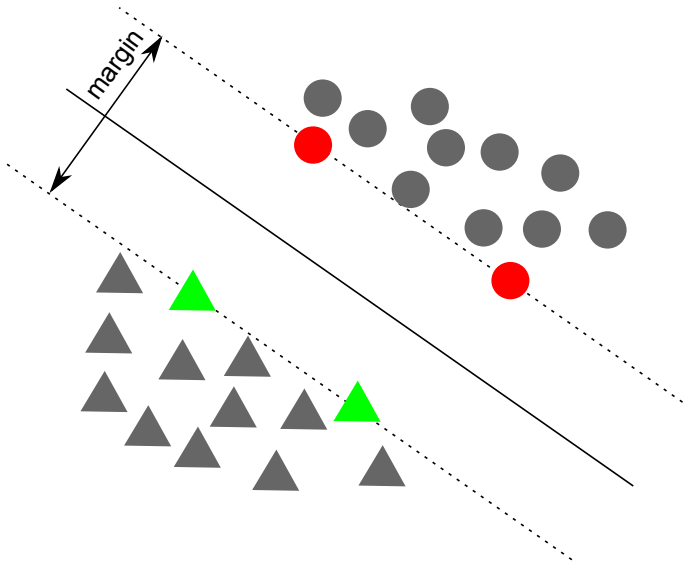



Figure 1.4: The advantage of an interactive semantic segmentation approach is that it is possible to achieve very good results with very little training data. In this case it is possible to split the two categories (circles and triangles), by annotating only the red and green coloured samples.

When a new classifier is trained from scratch, it is possible to create pleasing semantic segmentations with very little annotation effort. This is especially useful when hardly any ground truth annotations exist.

### 1.4.1 Classifier requirements

Based on the definition of interactivity, some properties, which must be fulfilled by the semantic segmentation algorithm can be derived:

- **Online capability of the classifier**
  The classifier must support trainings data arriving online, i.e., the classifier does not

see all training data in advance, but only some data in a stream-like setting.

- **Do not overfit on trainings data**
  Since only a subset of the data is available during the training stage, it is important that the classifier does not overfit to this data. The generalization (performance on previous unseen data) of the classifier should be as good as possible.

- **Fast evaluation**
  An application is only interactive, if the user gets immediate feedback after he has performed some operation. This requires a classifier that is very fast at test time, i.e., in deriving the semantic segmentation based on the samples provided by the user.

- **Multi-class**
  The classifier should be able to handle multiple classes and not just the binary case (foreground versus background), because in general one wants to split the data into more than two different classes. However, since the binary segmentation is just a special case of the multi-class segmentation problem, it can be solved with a multi-class classifier, too.

- **Probabilistic output**
  The classifier should output the complete posterior distribution over the class labels and not directly a class label. This is especially important, because these class label probabilities can be used almost directly to guide the user (see Section 5.3).

Here a short overview of classifiers commonly used throughout the literature in semantic segmentation is presented. It is shown whether they fulfil the required properties stated above or not. For a more complete list of classifiers used in machine learning the reader is referred to [27].

The first classifier presented is a *Support Vector Machine* (SVM). SVMs are known to be margin maximizing classifiers and therefore, they generalize very well on previously unseen data. However, SVMs are not inherently multi-class classifiers, i.e., SVMs can handle multiple classes only by training *1 versus all* SVMs. That is, for each class one SVM is trained against all other classes and the results of all SVMs are then combined to derive the final result. As Criminisi *et al.* pointed out in [14] this can lead to asymmetric decision boundaries which are not really justified by the training data. The performance of a SVM depends on the number of classes in the classification problem. The more classes

should be distinguished, the more SVMs are necessary and therefore the processing time increases. In addition, since semantic segmentation is a complex task, it is not possible to split the classes using a linear SVM, but a *kernel SVM* must be used. If there are a lot of support vectors to compare against, the performance suffers. Confidence output can only be produced with the usage of relevance vector machines [70], but only at the expense of further computations. SVMs are inherently online and therefore training data arriving any time can be incorporated easily.

*Boosting* is a classifier using the power of ensembles. There a strong classifier is constructed out of many weak learners. The most famous boosting algorithm is called AdaBoost [20]. Boosting concentrates on the difficult samples by re-weighting the samples appropriately. This makes boosting a powerful classifier. However, due to the re-weighting noise becomes more important too and therefore, this classifier is more noise sensitive. Due to the many weak learners which must be used to achieve a good performance, boosting is rather slow. As it is the case with SVMs, boosting is not inherently multi-class, i.e., the algorithm was designed for binary classification problems originally.

Another option to consider are *deep neural networks* (DNN). A neural network is said to be deep if it has more than one stage of non-linear feature transformation. The structure of DNNs is rather complex and it is difficult to comprehend what is actually happening in the DNN. However, DNNs are powerful classifiers which can be used in an online setting. To achieve interactive performance it is necessary to implement them on the GPU.

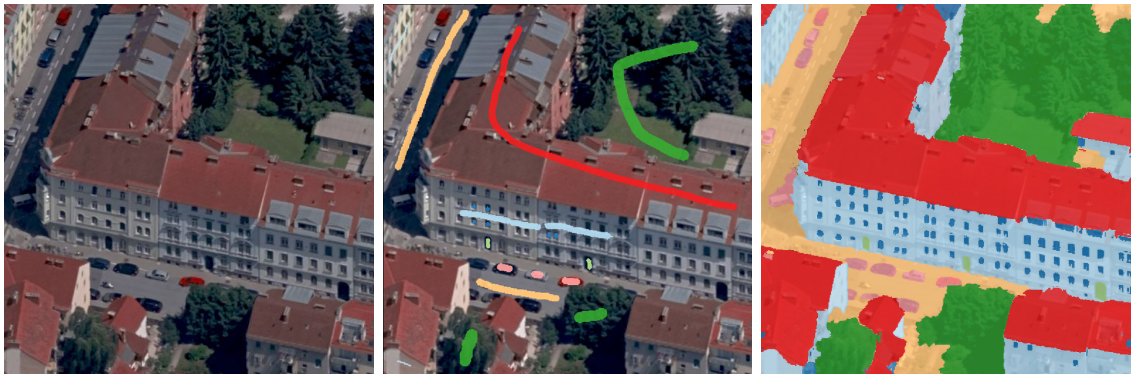
In this work, the main classifier used is a *random forest* (see Section 2.5). Random forests are inherent multi-class, very fast to evaluate and are known to be margin maximizing, which are great advantages. However, random forests are not inherently online, i.e., some tricks must be applied to use them in an online setting. Different variants of using random forests online are revealed in Section 4.2.

For comparison *random ferns* (see Section 2.6) are used in this thesis. Random ferns are similar to random forests. However, their structure is simplified and random ferns are generative classifiers. Therefore, they can be used in an online setting easily.

To summarize, on one hand interactive segmentation enables a lot of capabilities, due to the knowledge of a human, which can guide the semantic segmentation. However, on the other hand the *interactivity* also introduces some constraints on which algorithms are actually usable. Random forests and random ferns are best suited for interactive semantic segmentation. DNNs would be an interesting alternative, however, due to the simplicity of forests and ferns they are used in this thesis.

### 1.4.2 User Interaction

Obviously, in an interactive application some user interaction is required. The tool must provide a feature to incorporate the knowledge of the user into the learning algorithm. That is, the user must have the possibility to guide the algorithm by segmenting some of the data. In this work, the preference is given to brush strokes, i.e., *scribbles*, because of the several advantage they have compared to drawing bounding boxes or polygons or to mark the contours between objects. Drawing scribbles onto an object is very intuitive and it is easy to use them for different granularities. Furthermore, the annotation can be done very fast by using this technique. The scribbles are color coded and with different colors the user specifies the classes for the semantic segmentation. After the user did all annotations, i.e., he provided some samples of all classes he wants to segment, the classifier tries to learn the intent of the user and generalizes the gained knowledge to the complete image. If the user is not satisfied with the result, it is possible to enhance the segmentation by providing additional scribbles at incorrect segmented areas. In Figure 1.5b scribbles are visualized.



(a) Oblique aerial image (b) Interactive annotation (c) Semantic segmentation

Figure 1.5: Example of a semantic segmentation on an oblique aerial image. 1.5a shows the input image where a semantic segmentation should be derived. 1.5b shows the user annotation of the classes Roof, Vegetation, Façade, Window, Car, Ground, Entry. 1.5c shows the final semantic segmentation created based on the user input. Notice the pixel accurate class borders.

## 1.5 Outline

The further sections are organized as follows. Section 2 gives an overview of related work in semantic segmentation and how this problem can be solved. Section 2.1 introduces general

definition and notations used throughout this thesis. In Section 2.3 a general classifier is defined. The used classifiers in this thesis, random forests and random ferns, are reviewed in Section 2.5 and Section 2.6 in detail. Section 2.7 introduces the concept of boosting and how it can be used in the context of semantic segmentation. Since context is very important in a scene, Section 2.8 gives an overview of how context can be incorporated into the classification task.

In Section 3 the focus lies on the feature channels, i.e., the information source of the classifier, and on the features to be used for classification. The used feature channels are reviewed in Section 3.1 in detail. In Section 3.2 different features are presented.

In Section 4 the algorithms and concepts developed and used in this thesis are described. It starts with a general definition of online learning and reveals differences to the offline setting and the concept of continuous learning is presented. Section 4.2 shows how random forests can be used online, how the classifier can be updated and what can be done to achieve a good generalization. Section 4.3 reviews how random ferns can be used for interactive semantic segmentation. Some performance tricks incorporated in the implementation are shown in Section 4.5.

In Section 5 the application resulting from this thesis is presented. Section 5.1 defines the interactive semantic segmentation pipeline. The graphical user interface to be used for the semantic segmentation is stated in Section 5.2. To get the most out of the labels, i.e., the annotations made by the user should be as little as possible, in Section 5.3 a concept called *Active User Guidance* is described. Finally, Section 5.4 shows the algorithms used for regularizing the output of the classifier.

In Section 6 the used algorithms are evaluated and compared to state of the art algorithms. The performance evaluation of random forests is shown in Section 6.1 and the evaluation results for random ferns are shown in Section 6.2.

Section 7 summarizes and gives an outlook for further work.

# Chapter 2

# Related Work

## 2.1 Notations

Before diving deeper into the related work, the notations used in this thesis are introduced. In the following paragraphs, general definitions and machine learning definitions are done. Table 2.1 summarizes common used letters throughout this work.

**General Definitions** Lower case letters (e.g., $x$, $\lambda$) are used for scalar variables. Vectors are denoted by bold face lower case letters (e.g., $\mathbf{x}$) and unless other noted, vectors are column vectors. An example for a row vector would be $\mathbf{x}^T = (x, y)$. The $k$-th element of a vector is represented by a subscript (e.g., $x_k$). Matrices are denoted by bold face upper case letters (e.g., $\mathbf{M}$) and the elements are ordered row-major, i.e., the element in row $i$ and column $j$ is denoted by $M_{ij}$. Sets are represented by calligraphy letters like $\mathcal{X}$.

A random variable (RV) is denoted by upper case letters, such as $Y$. A probability distributions over a RV $Y$ is denoted as $\mathbb{P}[Y]$. In this thesis, the random variables correspond to the unknown class labels and are therefore discrete. The probability of a discrete random variable taking the value $y_i$ is denoted as $\mathbb{P}[Y = y_i]$ or in short $p(y_i)$. The expectation of a RV X is given by $\mathbb{E}[X = x_i]$.

**Machine Learning Definitions** A set of labelled data is defined as $\mathcal{X}_l = \{(\mathbf{x}, y)|\mathbf{x} \in \mathcal{X}, y \in \mathcal{Y}\}$, where $\mathcal{X}$ is a $d$-dimensional feature space $\mathbb{R}^d$. Each entry $x_i$ in the data point represents some attribute of it, called *feature*. Each labelled data point $\mathbf{x}$ has a label $y$ assigned. The class labels are discrete numbers $\mathcal{Y} = \{1, \ldots, K\}$ and therefore a $|\mathcal{Y}|$-class classification problem is given. In case $|\mathcal{Y}| = 2$, i.e., a binary problem is addressed, the labels are drawn from $y \in \mathcal{Y} = \{-1, +1\}$ to be compatible to other literature. A set of

unlabelled data is defined as $\mathcal{X}_u = \{\mathbf{x}|\mathbf{x} \in \mathcal{X}\}$. In an unlabelled data set, no class labels are available. One data point $\mathbf{x}_i \in \mathcal{X}_l$ or $\mathbf{x} \in \mathcal{X}_u$ is referred to as a *sample*.

**Likelihood and potentials** When dealing with classifiers directly, usually one is interested in the posterior probabilities over the class labels. The output of the classifier is the likelihood that a data point $\mathbf{x}$ belongs to class $y$. The greater the likelihood for class $y_i$, the more probable that $y_i$ is the correct label and therefore higher probabilities are better. However, some models require potentials, where the classification problem is stated as an energy minimization problem. Since this is a MINimization problem, lower values are better. Therefore, the probabilities must be transferred to potentials. Given the probability for class label $p_i$, the corresponding potential $f_i$ can be calculated easily by

$$f_i = 1 - p_i, \qquad p_i \in [0, 1] \tag{2.1}$$

The lower the potential $f_i$, the better.

**Images** A grey scale image is represented as a matrix $\mathbf{I}$, where each entry $I_{ij}$ is called pixel and has an intensity of 8-Bit. In a color image $\mathbf{I}_{RGB}$, each pixel has a vector of size 3 assigned, where the values represent the red, green and blue intensity respectively.

| Symbol | Description |
|---|---|
| $\mathcal{X}_l$ | labelled set |
| $\mathcal{X}_u$ | unlabelled set |
| $\mathcal{P}$ | set of all patches in an image |
| $\mathcal{Y}$ | set of all labels |
| $\mathcal{X}$ | set of all features |
| $\mathcal{T}$ | set of all test parameters |
| $h(\mathbf{x}, \boldsymbol{\theta})$ | (weak) classifier |

Table 2.1: Predefined variables and notations

## 2.2 System Approach

In this section, we want to define the system approach, which is used in this thesis. Three important steps are necessary to perform a semantic segmentation on an arbitrary input image, where the first step can be considered as a preprocessing step and in step 2 and 3 the semantic segmentation is done based on the information calculated in step 1.

(1) **Feature channels**
The first step in a semantic segmentation framework is to extract so called feature channels. Feature channels can be extracted out of the RGB input image and abstract the pure color information for further processing. What feature channels are actually used and why they play a very important rule is described in Section 3.

(2) **Classification**
In the second stage, a powerful classifier is used to estimate the class likelihoods in a dense manner. The class likelihoods are often referred to as the *unaries*. By assigning the class label with the highest probability to each pixel, the maximum a posteriori (MAP) solution can be derived. However, this solution might be very noisy, since each pixel is treated completely individual. We used random forests (Section 2.5) as well as random ferns (Section 2.6) as classifiers in this thesis.

(3) **Regularization**
Because the pure MAP solution is visually not very pleasing, the third part is the *regularization* part (Section 5.4). Regularization means that the MAP solution is not used directly, but the class posteriors are smoothed. This leads to the effect that the labels are smooth in the final solution and label transitions are preferred to be at strong boundaries in the image.

**Patches for Semantic Segmentation** In order to do semantic segmentation, each pixel in the image must be presented as a data point $\mathbf{x}$. In semantic segmentation such a data point is represented through the actual pixel and the local neighbourhood of that pixel. This local neighbourhood is referred to as a *patch*. An example image and one contained patch in this image is visualized in Figure 2.1. In this application the size of such a patch is defined by $25 \times 25$ pixel, but in general an arbitrary patch size is possible. However, the data point $\mathbf{x}$ contains not the pixels of the patch directly, but properties of the patch called *feature channels* (Section 3). Each feature channel patch is saved as a row vector and the concatenation of all feature channel vectors is the resulting data point.

## 2.3 Classifier

As already proposed in Section 1.2, semantic segmentation can be solved using machine learning approaches. Since we want to learn discrete class labels, semantic segmentation can be considered as a classification task. The oracle, which actually does the learning is

(a) Patch (b) Complete Image

Figure 2.1: Patches for semantic segmentation. 2.1a: To classify one pixel in semantic segmentation a patch around this pixel (the red dot) is cropped out of the image. The data point $\mathbf{x}$ is a concatenation of the feature channels of the patch. 2.1b shows the source image where the visualized patch is marked with a red rectangle.

called *classifier*. Formally defined, a classifier is defined as a mapping

$$C(\mathbf{x}) = C(\mathbf{x}, \boldsymbol{\theta}) : \mathcal{X} \times \mathcal{T} \rightarrow \mathcal{Y} \tag{2.2}$$

where $\mathbf{x} \in \mathcal{X}$ is a feature vector, $\boldsymbol{\theta} \in \mathcal{T}$ are the parameters of the classifier and $\mathcal{Y}$ is the set of discrete class labels.

The aim of a classifier is to partition the feature space $\mathcal{X}$ into disjoint regions, where each region is associated with a certain class label. These regions are created during classifier training by finding the *decision boundaries* between distinct classes and denoted as *decision regions*. When the decision regions are defined (Figure 2.2), new data points can be classified. Classification of a feature vector $\mathbf{x}$ means to determine the decision region to which $\mathbf{x}$ corresponds to. After this region has been determined, the appropriate class label can be assigned to the feature vector.

In the case of semantic segmentation, the feature vector $\mathbf{x}$ can be derived from the patch. The label space $\mathcal{Y}$ contains discrete class label such as WINDOW, ROOF, FAÇADE for example.

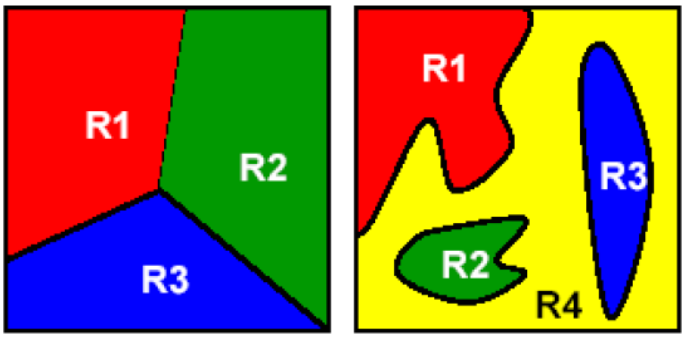



Figure 2.2: Visualization of two feature spaces partitioned into disjoint regions. The regions are derived during the training of the classifier.

### 2.3.1 Statistical Viewpoint

From a statistical point of view, a classifier tries to distinguish new data based on observed data during the trainings phase. The observed data can also be seen as samples from a probability distribution. The Bayes' theorem (Equation (2.3)) allows us to evaluate the uncertainty in $X$ after we have observed $Y$ in the form of the posterior distribution $p(\mathbf{x}|y)$ [5].

$$p(\mathbf{x}|y) = \frac{p(y|\mathbf{x})p(\mathbf{x})}{p(y)} \tag{2.3}$$

In words, the Bayes' theorem can be stated as

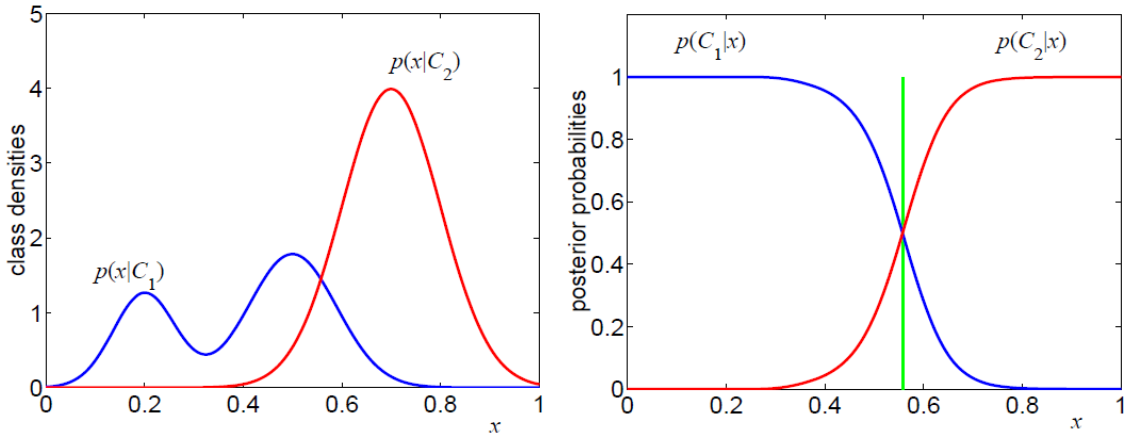
$$\begin{aligned} \text{posterior} &= \frac{\text{likelihood} \times \text{prior}}{\text{evidence}} \\ &\propto \text{likelihood} \times \text{prior} \end{aligned} \tag{2.4}$$

The denominator, i.e., the evidence, can be omitted, because this is simply a scaling factor and does not affect the outcome of the most probable class label. Depending on what terms of the Bayes' theorem a classifier actually models, they can be separated into two different kinds. In general one distinguishes between a generative and a discriminative classifier.

**Discriminative Classifier** A discriminative classifier models the posterior directly. Such a classifier tries to separate the data without modelling the underlying probability distribution of the data. This means that the classifier does not know how the data is distributed. A visualization of the posterior probability is shown in Figure 2.3b.

**Generative Classifier** In difference, a generative classifier does not model the posterior probability directly but tries to model the likelihood and the prior of the classes in the data.

Hence, a generative classifier knows how the data is distributed in the feature space. The corresponding posterior probabilities can be derived easily by using the Bayes' theorem (Equation (2.3)). A visualization of the likelihood modelled by a generative classifier is shown in Figure 2.3a.




(a) (b)

Figure 2.3: Generative versus discriminative classifier: In 2.3a a generative classifier was used. This classifier models the likelihood, i.e., the densities, of the different classes. 2.3b shows the corresponding posterior probabilities. A discriminative classifier does only model the posterior distribution without modelling the class densities. Notice, how the posterior distribution can be inferred from the likelihood. Image courtesy of [39].

### 2.3.2 Ensemble Methods

In *ensemble* methods the strengths of multiple learners are combined to build one strong classifier. The idea is to find combinations of base learners that overall perform better than each individual part. In general, ensemble learning can be broken into two phases [27]:

(1) Construct base learners with the training data

(2) Combine the base learners to form a strong classifier

Prominent ensemble methods are *random forests* (Section 2.5) and *random ferns* (Section 2.6). There, the individual base classifiers can be combined by the normalized combination of the base classifiers:

$$C(\mathbf{x}) = \frac{1}{N} \sum_{n=1}^{N} c_n(\mathbf{x}) \tag{2.5}$$

Another well known ensemble method is called *boosting* (Section 2.7). In this method the weak learners are combined using a weighted summation of the classifiers:

$$C(\mathbf{x}) = \sum_{n=1}^{N} \alpha_n c_n(\mathbf{x}) \tag{2.6}$$

where $\alpha_n$ corresponds to the influence of the $n$-th classifier. In difference to random forests and random ferns, in boosting the ensemble evolves over time, i.e., one weak classifier is added to the ensemble in each iteration.

## 2.4 Decision Tree

Decision trees exist for a long time [8, 54, 55]. A decision tree is a special type of graph. Formally spoken, a decision tree is a directed, acyclic graph connected from a root node (Figure 2.4). One distinguishes between the *root* node, *split* nodes and *leaf* nodes. Notice that the root node is a split node too. Except of the root node, all nodes in the tree have exactly one incoming edge and since we are dealing with *binary* decision trees, each node has exactly two outgoing edges. However, note that a general decision tree can have an arbitrary number of children at each node.

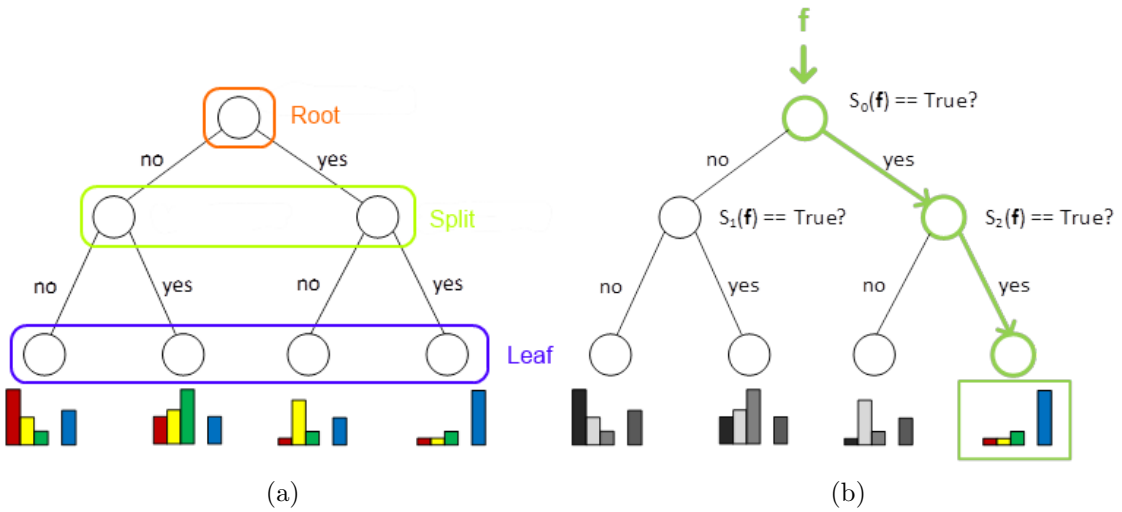
A decision tree is a set of binary questions, which are hierarchically organized. When an unknown feature vector is pushed through the decision tree, at each node the feature vector is handed over either to the left or to the right child node based on the outcome of a binary test. After pushing an unknown feature vector through the tree, the correspondence of the feature vector to different classes is estimated by the reached leaf node. Which questions are applied at each split node is defined by the tree parameters $\boldsymbol{\theta}$. Where it is possible to define $\boldsymbol{\theta}$ by hand for simple problems, due to the high complexity of semantic segmentation, $\boldsymbol{\theta}$ is learnt automatically from the available training data.

Each split node in the decision tree is also called *split function*, *weak learner* or *test function*. Note that these terms are used interchangeable and treated as synonyms in this work. The parameters of each weak learner, i.e., how the data is split in each weak learner, are discussed in Section 2.4.1. For now, just see a weak learner as a rule which encodes how data arriving at a specific node is split.

Precisely, a weak learner (e.g., the $j$-th node in the tree) is defined as the mapping from the Cartesian product of feature space $\mathcal{X}$ and the set of all test functions $\mathcal{T}$ to a binary output true or false.

$$h(\mathbf{x}, \boldsymbol{\theta}_j) : \mathcal{X} \times \mathcal{T} \rightarrow \{0, 1\} \tag{2.7}$$

where $\boldsymbol{\theta}_j \in \mathcal{T}$ denotes the parameters of the test function for the $j$-th split node. The binary decision whether a feature vector is propagated to the right child or to the left child is done using exactly this split function (Figure 2.5).




(a) (b)

Figure 2.4: Decision Tree: 2.4a shows a binary decision tree. The orange node is denoted as *root* node, the nodes in light green are *split* nodes or *internal* nodes and the purple nodes are the *leaf* nodes. 2.4b shows how a decision tree classifies an input vector $\mathbf{f}$ by a series of yes/no questions starting at the root node. At each node in the tree, the feature space is split according to the outcome of some binary decision criterion $S_i(\mathbf{f})$. The leaf nodes contain the posterior distribution over the class labels $p(y_i|\mathbf{f})$ created during training. The colors red, blue, green and yellow indicate the relative frequency of samples ending in a specific leaf node. The gap between the green and blue bar indicates that an arbitrary number of classes is possible when working with decision trees.

As can be seen from this definition, a decision tree can be thought of as a technique for spitting one complex problem into a set of simpler problems, because a data point is not classified directly but after a series of simple binary questions.

### 2.4.1 Tree Training

In machine learning the parameters of a classifier are derived from *training points*. A training point $\mathbf{x} \in \mathcal{X}$ is a point in a high dimensional feature space. As described in Section 2.2, in semantic segmentation data points correspond to features extracted from local patches. The set of all training points is called the *training set* and denoted by $\mathcal{S}$.

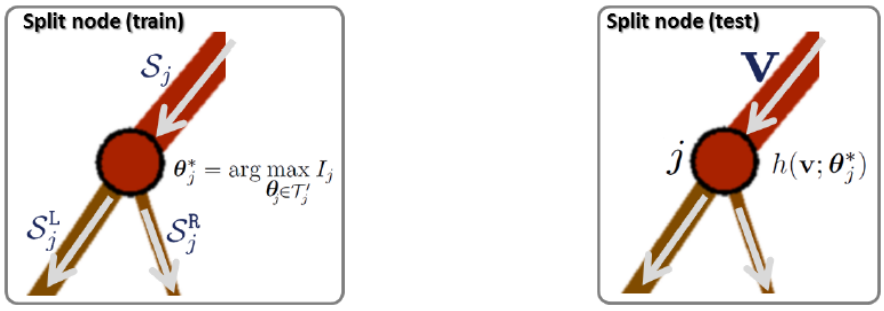
The trainings set $\mathcal{S}_j$ denotes the set of all training points arriving at the $j$-th node in the tree. Note that the nodes in the tree are enumerated in breadth first order starting with 0 at the root node. $\mathcal{S}_0$ is split into $\mathcal{S}_0^L$ and $\mathcal{S}_0^R$ denoting the subset going to the left and right child node respectively. Using these definitions, a binary decision tree fulfils the following properties [14]:

(1) $\mathcal{S}_j = \mathcal{S}_j^L \cup \mathcal{S}_j^R$

(2) $\mathcal{S}_j^L \cap \mathcal{S}_j^R = \emptyset$

(3) $\mathcal{S}_j^L = \mathcal{S}_{2j+1}$

(4) $\mathcal{S}_j^R = \mathcal{S}_{2j+2}$




(a) (b)

Figure 2.5: Split nodes. Visualization of a split node and its behaviour at train- (Figure 2.5a) and test-time (Figure 2.5b). During training the optimal parameter $\boldsymbol{\theta}_j^*$ is derived by optimizing the information gain $I_j$. During test time, the weak learner is applied to the incoming feature vector $\mathbf{v}$ and is propagated to the left or right child node depending on the outcome of the learnt test function $h(\mathbf{v}, \boldsymbol{\theta}_j^*)$. Image courtesy of [14].

The essential part of the trainings phase is how the individual weak learners in the tree are trained. This is important, because the split functions define the strength of the tree and how well data with different labels can be separated. To make the training of each weak learner most effective, we want to measure the quality of a proposed split. The most common used quality measures to measure the quality of a split are the *entropy*, the *Gini index* and the *misclassification error* [27]. Figure 2.6 visualizes these node impurity measures.

**Impurity Measures** The entropy is a measure of uncertainty of a random variable and comes from the mathematical sub-domain *information theory*. The higher the entropy of

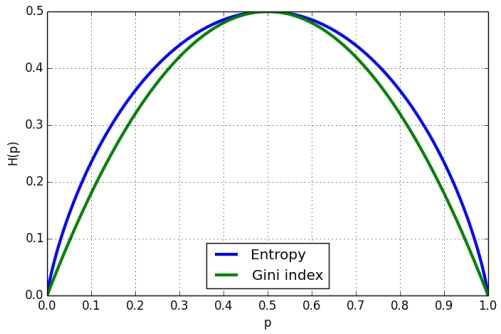



Figure 2.6: Visualization of the node impurity measures *Entropy* and *Gini index* as a function of the proportion $p$ in the $2^{nd}$ class. Note, the higher the uncertainty, the higher the impurity measure. The entropy has been scaled to pass through (0.5, 0.5) [27].

a random variable, the more uncertain is the outcome of it. The Shannon entropy [63] is defined as

$$H_E(\mathcal{S}) = -\sum_{y \in \mathcal{Y}} p(y) \log p(y) \tag{2.8}$$

The Gini index is very similar to the entropy. However, as visualized in Figure 2.6, the Gini index is less sensitive to changes in the node probabilities than the entropy. The Gini index is defined as

$$H_G(\mathcal{S}) = \sum_{y \in \mathcal{Y}} p(y)(1 - p(y)) \tag{2.9}$$

We decided to use the entropy in this work. In a classification task like semantic segmentation, we want to minimize the uncertainty of the predicted class label. This means, the purer $\mathcal{S}$ in terms of class labels, the smaller is the entropy. With a decision tree we try to separate data such that the leaf nodes are as pure as possible, i.e., they contain only one class in it. In terms of entropy, we are striving for small entropies which means that there is no uncertainty about the correct labelling. However, we will see that it is not always possible to get pure leaf nodes. This is due to the high complexity of semantic segmentation.

**Information gain** The impurity measures can further be used to calculate the so called information gain which results after a split. The information gain is a measure of the improvement of the purity after a split and can be defined as

$$I = H(\mathcal{S}) - \sum_{i \in \{L,R\}} \frac{|\mathcal{S}^i|}{|\mathcal{S}|} H(\mathcal{S}^i) \tag{2.10}$$

The smaller the impurity in the child nodes $\mathcal{S}^L$ and $\mathcal{S}^R$, the greater the information gained by the split and therefore the better the split.

**Data separation** As described in the previous paragraph the quality of a candidate split can be measured using the information gain. In this paragraph the model for actually splitting the data is defined. In general one distinguishes between data separation based on

- axis aligned hyperplanes
- oriented hyperplanes
- nonlinear surfaces

For example, in 2D the hyperplanes correspond to lines and surfaces correspond to curves respectively (Figure 2.7). In this application axis aligned hyperplanes are used to split the data, because they can be evaluated very fast and the performance is good if the used classifier is strong. The parametrization of a linear model is defined as

$$h(\mathbf{x}, \boldsymbol{\theta}_j) = \mathbb{1}_{[f(\mathbf{x},\boldsymbol{\phi}) \geq \tau]} \tag{2.11}$$

where $\mathbb{1}_{[.]}$ is the indicator function, $\boldsymbol{\theta}_j = (\boldsymbol{\phi}, \tau)$, where $\boldsymbol{\phi}$ are the parameters of the hyperplane, $f(.)$ computes the feature response of a data point $\mathbf{x}$ and $\tau$ is a threshold. Each split node (weak learner) applies exactly such a test to its incoming data to decide whether the data point is propagated to left or right child node. The splits are actually done using *features*. This is described in detail in Section 3.2.

At this place, we have all tools available to choose the best weak learner among all possible weak learners. We can do this by maximizing the information gain by doing an exhaustive search in the space of all possible parameters:

$$\boldsymbol{\theta}_j^* = \arg\max_{\boldsymbol{\theta}_j \in \mathcal{T}} I_j(\mathcal{S}_j, \mathcal{S}_j^L, \mathcal{S}_j^R, \boldsymbol{\theta}_j) \tag{2.12}$$

This means that depending on the subset $\mathcal{S}_j$, the function which splits the incoming data best in terms of the information gain is chosen. Then the weak learner is said to be *learnt*. After the training of one weak learner, two

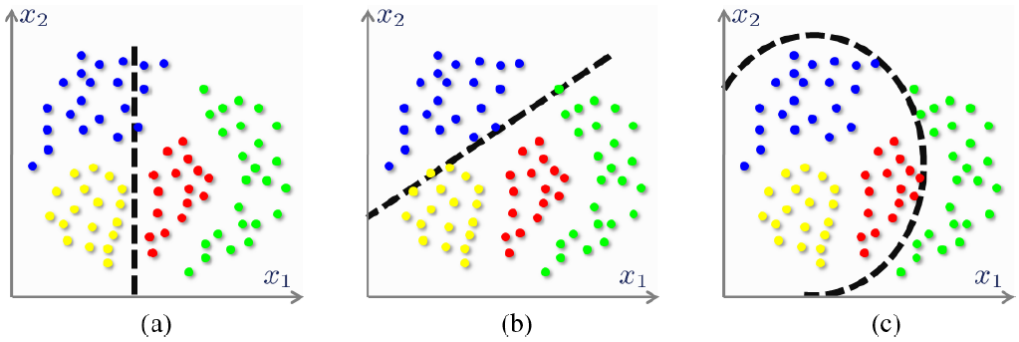



Figure 2.7: Visualization of data separation models in 2D. (a) axis aligned line (b) oriented line (c) quadric. Image courtesy of [14].

new nodes evolve. The training is continued recursively until a certain stopping criteria (e.g., maximum tree depth, information gain lower than a threshold, ...) is reached.

To summarize, after tree training we obtain

- optimized weak learners associated to each node in the tree
- a learnt tree structure
- different sets of training points at each leaf (forming the leaf statistics)

Note that a decision tree is a discriminative classifier (see Section 2.3.1), because the posterior distribution over the class labels is modelled directly. Hence, a decision tree tries to split the data as good as possible without considering the real distribution of the data.

### 2.4.2 Tree Testing

The aim of a classifier is to assign previously unseen data to specific regions in the feature space and then classify the data by assigning a logical class label to it. The weak learners trained during the training phase are now applied to a new incoming data point $\mathbf{x}$. Depending on the outcome of the binary test (Equation (2.11)) $\mathbf{x}$ is propagated either to the left or right child node. This procedure is continued until a leaf node is reached. Each leaf node contains a set of labelled training points forming a probability distribution. Figure 2.5b shows a visualization of how an input vector $\mathbf{x}$ is classified using a decision tree. Due to the fact that a decision tree outputs the posterior distribution over the class labels

$$p(y_i|\mathbf{x}) \tag{2.13}$$

it is not possible to directly assign a class label to the input vector $\mathbf{x}$. However, the data point can be classified to a discrete class label by finding the mode of the posterior distribution

$$C(\mathbf{x}, \boldsymbol{\theta}) = \arg\max_i \mathbb{P}[Y = y_i|\mathbf{x}] \tag{2.14}$$

## 2.5 Random Forests

Single (random) decision trees do not perform comparable to other classifiers like SVMs or neuronal networks for complex tasks. However, by using not only one decision tree, but an ensemble of decision trees, the performance can be increased significantly. Random Forests are basically ensembles of *Random Decision Trees* (Definition 2.5.1). The main classifier used in this thesis is a random forest. Random forests have been used in a broad range of applications in computer vision [38, 47, 58, 65]. The concept of Random Forests was first introduced by [4, 10]. Breiman defined Random Forests as follows:

**Definition 2.5.1 (Random Forest)** *A Random Forest is a classifier consisting of a collection of tree structured classifiers* $\{h(\mathbf{x}, \boldsymbol{\theta}_k), k = 1, \dots\}$*, where the* $\{\boldsymbol{\theta}_k\}$ *are independent identically distributed random vectors and each tree casts a unit vote for the most popular class at input* $\mathbf{x}$*.*

Each tree is a classifier $h(\mathbf{x}, \boldsymbol{\theta}_k)$, where $\mathbf{x}$ is an input vector and the parameter $\boldsymbol{\theta}_k$ is a random parameter vector. For the $k$-th tree a random vector $\boldsymbol{\theta}_k$ is generated independently of the past random vectors $\boldsymbol{\theta}_1, \dots, \boldsymbol{\theta}_{k-1}$ but with the same distribution. The number of trees in a forest is denoted as $T$ and is arbitrary in general. All trees in a forest are trained individually, i.e., they do not even know that there exist other trees.

The following sections define all necessary ingredients needed for a random forest. The concept of one decision tree was already discussed in Section 2.4. The reminder of this section will continue with how decision trees can be trained randomly and why this is a good idea. The last section derives properties of random forests like generalization and the strength.

### 2.5.1 Randomized Training

Semantic segmentation and computer vision problems are very hard and complex problems. Therefore it is not possible to select the parameters of a split function by searching the complete space of possible weak learners. Hence, some randomness is injected into the

training phase of each decision tree. Randomly trained decision trees are referred to as *random decision trees* and ensembles of random decision trees are referred to as *random forests*. Randomness can be injected into the training process in two different ways which are described in the next paragraphs. Note that both types of randomization can be used together. This was done in this work, too.

**Randomized Node Optimization** When randomized node optimization is used, the set of all test functions $\mathcal{T}$ is reduced. Especially for complex problems like semantic segmentation, this is necessary, due to the inherent complexity of the problem, i.e., there are too much parameters for exhaustively searching the complete space of test functions. Therefore, only a subset of all possible training parameters is used, i.e., $\mathcal{T}_R \subset \mathcal{T}$ is used for training. This can be defined mathematically as

$$\boldsymbol{\theta}_j^* = \underset{\boldsymbol{\theta}_j \in \mathcal{T}_R}{\arg\max}\ I_j(\mathcal{S}_j, \mathcal{S}_j^L, \mathcal{S}_j^R, \boldsymbol{\theta}_j) \tag{2.15}$$

where $\mathcal{T}_R$ is a random subset of split parameters. If the size of the subset $\mathcal{T}_j$, $|\mathcal{T}_j|$, is 1, i.e., only one random test is used for the weak learner selection, the tree is referred to as an *Extremely Randomized Decision Tree* [25]. Breiman suggested to use [27]

$$|\mathcal{T}_j| = \sqrt{|\mathbf{x}|} \tag{2.16}$$

where $|\mathbf{x}|$ is the dimension of the feature vector and corresponds to $patchSize^2 \times numFeatureChannels$.

**Random Training Set Sampling** Another possibility to incorporate randomness into the training phase is to sub-sample the set of all training samples. Three methods of how this sub-sampling could be done are outlined in the following paragraphs.

- **Random Sub-sampling**
  In this method completely random samples are drawn out of the training data for each weak learner.

- **Bagging**
  In *bagging* (= Bootstrap aggregating) [9] for each weak learner a random subset out of the complete training set is drawn *with replacement*. This reduces the variance of the data seen by each weak learner.

- **Per Node Sub-sampling**
  In difference to the previous revealed sub-sampling methods, Schulter *et al.* proposed to randomly sub-sample the training data in each node [62]. When the underlying structure of the data is represented reasonably by this subset, then the strength $s$ does not suffer too much and the correlation $\overline{\rho}$ of the trees is decreased further [62] (cf. Equation (2.22)).

All methods proposed above lead to a decorrelation of the trees, because each tree focuses on a specific set of training data. The combination of many trees ensures a strong classifier with improved accuracy compared to correlated (each tree gets the whole and same training data) trees. By sub-sampling the training data in each splitting node, the training speed can be highly increased, because small subsets of the whole training data are enough to achieve a good performance.

### 2.5.2 Testing

After all trees are trained, the forest forms a strong classifier. A new data point, i.e., a point without a class label, is now pushed through all trees until it reaches a leaf node in each tree. The predictions, i.e., the posterior distributions, of each leaf node must be combined to one final prediction for the data point. This can be done by an averaging operation [10]:

$$p(y|\mathbf{x}) = \frac{1}{T}\sum_{t=1}^{T} p_t(y|\mathbf{x}) \tag{2.17}$$

where $p_t(.)$ denotes the posterior distribution of the $t$-th tree in the forest. The final class output of a multi class classification problem can then be derived by using the mode of the posterior distribution of the forest:

$$C(\mathbf{x}, \boldsymbol{\theta}) = \underset{y \in \mathcal{Y}}{\arg\max}\ p(y|\mathbf{x}) \tag{2.18}$$

### 2.5.3 Properties

By introducing randomness in the forest training in the form of using different subsets of training data for different trees in the forest, the trees become decorrelated. The following definitions define a basis to analyse correlation and strength of random forests (based on [10]). Breiman defined the classification margin as

$$mr(\mathbf{x}, y) = p(y|\mathbf{x}) - \max_{\substack{k \in \mathcal{Y} \\ k \neq y}} p(k|\mathbf{x}) \tag{2.19}$$

The margin defines how secure a classifier is on its decision by comparing the most probable label with the second most probable label. Obviously, the greater the margin, the more secure the classification. It is also obvious that the classification is incorrect, if the margin is negative. Based on this thoughts, the generalization error can be defined as [10]:

$$GE = \mathbb{E}\left[mr(\mathbf{x}, y) < 0\right] \tag{2.20}$$

This error gives an estimate on how erroneous previously unseen data will be classified. However, not only the expected generalization error is interesting, but it would be interesting to derive an upper bound for this error. Breiman [10] actually calculated an upper bound for the generalization error. However, before we can state the inequality for the upper bound, the term *strength* must be defined. The strength of a classifier can be calculated based on the margin measure. The strength is defined as the expected value of the margin over the entire distribution:

$$s = \mathbb{E}\left[mr(\mathbf{x}, y)\right] \tag{2.21}$$

Finally, the generalization error can be bounded by

$$GE \leq \overline{\rho}\frac{1 - s^2}{s^2} \tag{2.22}$$

where $\overline{\rho}$ is the mean correlation between pairs of trees in the forest, where the correlation is actually measured in terms of the similarity of the predictions. This inequality confirms what was addressed in the previous section. The stronger the individual classifiers and the more decorrelated the classifiers, the lower the resulting classification error.

**Maximum Margin Property** A very important property in classification is how well a classifier generalizes to previously unseen data. To achieve a good generalization even with little trainings data, the *maximum margin property* is crucial. SVMs are known to be margin maximization classifiers. However, also in the context of random forests, the margin maximization property was discovered [14, 37]. It has been shown in [14] that random forests exhibit the maximum margin property. Consider a two class classification problem as shown in Figure 2.8. Criminisi *et al.* [14] showed that the separating line tends

to lie within the gap between the two classes if the training parameter subset $\mathcal{T}_R \subset \mathcal{T}$ for the weak learners is big enough. Due to the fact that the information gain is equal within the gap, each separation line within the gap is optimal and equal likely. This actually means that the separating lines are randomly distributed within the gap. The result is a maximum margin random forest. For a formal deviation see [14].

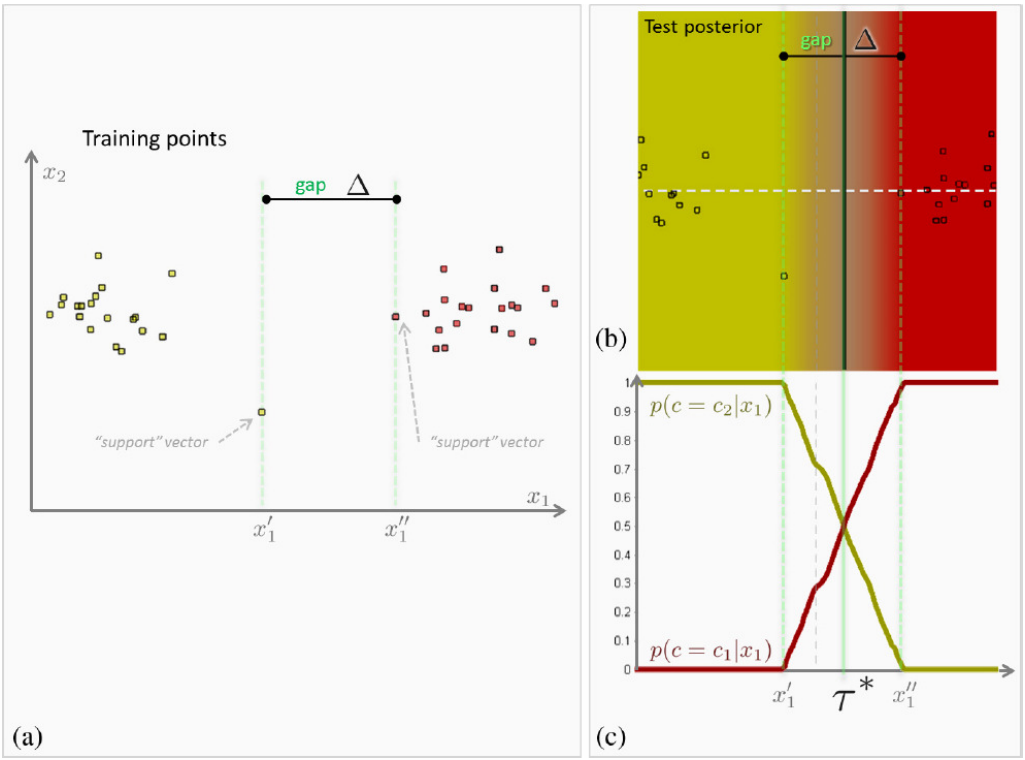
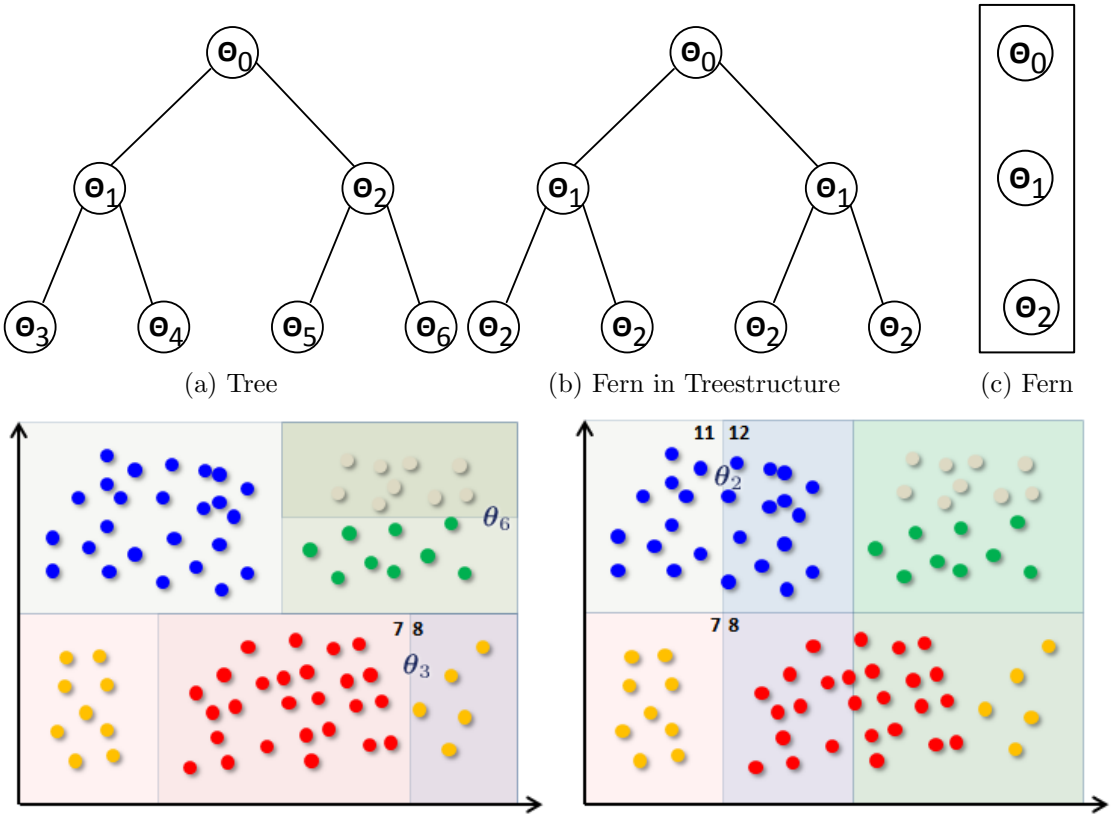



Figure 2.8: Maximum margin property of a random forest. (a) shows the input training points separated by a gap $\Delta$. The two "support vectors" are highlighted. In (b) the posterior distribution of the forest is visualized. The uncertainty remains just between the gap. (c) shows the cross section of the class posteriors along the white dashed line. Image courtesy of [14].

## 2.6 Random Ferns

Another classifier used in this thesis is called *Random Fern* [79, 80]. They can be considered as a special case of random forests, because each stage, i.e., each depth in a tree has its own split function, rather than each node. This is visualized in Figure 2.9. Özuysal *et al.* argued that the classification power of random forests does not lie in the tree structure, but in the groups of binary tests used. Therefore, they proposed a non hierarchical ar-

chitecture called *Fern*. Originally, ferns were used to detect keypoints in images, however they have been used in image classification too [6]. The following section shows how ferns are fit to a Bayes model.




(a) Tree (b) Fern in Treestructure (c) Fern

(d) Forest Splits (e) Fern Splits

Figure 2.9: Forests versus ferns: A forest as well as a fern can be visualized using binary decision trees. The difference lies in how the parameters $\boldsymbol{\theta}_j$ are chosen. In a forest each node has its own split function (Figure 2.9a), where in a fern each stage, i.e., each depth in the tree, has the same split function assigned (Figure 2.9b). In Figure 2.9c the compact representation of a fern is visualized. In Figure 2.9d and Figure 2.9e respectively a data set is visualized with the splits assigned by the forest and the fern. Image courtesy of [14].

### 2.6.1 Mathematical Formulation

In the following sections, the mathematical concept of ferns is derived. The start point is given by the Bayes' theorem (Equation (2.3)), then we will derive *Naive Bayes* until we reach the definition of a fern.

In semantic segmentation, we want to infer the class label based on a feature vector computed out of a pixel position. Formally, this can be defined as

$$C(\mathbf{x}) = \arg\max_i \mathbb{P}[Y = y_i|\mathbf{x}] \tag{2.23}$$

where $\mathbf{x}$ is a feature vector and $Y$ is a random variable representing the class. In probability theory spoken, Equation (2.23) is nothing but a posterior probability distribution over the class labels and the output of the classifier is the mode of this distribution. The rule of Bayes tells us that the posterior is equal to the likelihood times the prior (see Equation (2.5)). Using Bayes' theorem yields to

$$\mathbb{P}[Y = y_i|x_1, x_2, \ldots, x_N] = \frac{\mathbb{P}[x_1, x_2, \ldots, x_N|Y = y_i]\mathbb{P}[Y = y_i]}{\mathbb{P}[x_1, x_2, \ldots x_N]} \tag{2.24}$$

Here, the denominator can be omitted, because this is only a scaling factor independent from the classes and therefore it does not change anything of the result. After simplifying this we get

$$\begin{aligned} \mathbb{P}[Y = y_i|x_1, x_2, \ldots, x_N] &\propto \mathbb{P}[x_1, x_2, \ldots, x_N|Y = y_i]\mathbb{P}[Y = y_i] \\ &= \mathbb{P}[x_1, x_2, \ldots, x_N, y_i] \end{aligned} \tag{2.25}$$

Equation (2.25) tells us now that we can infer the posterior distribution by learning the joint likelihood distribution over all class labels. The problem arising here is that modelling the complete joint distribution is most likely intractable. For example, assume $N = 300$, i.e., 300 features are used. Modelling the complete joint probability would require to compute and store $2^N = 2^{300}$ entries for each class. One could decrease this high complexity by assuming complete independence of the random variables. This rather extreme step yields to the so called *Naive Bayes* assumption. Formally, the Naive Bayes formulation is

$$\mathbb{P}[x_1, x_2, \ldots, x_N, y_i] \overset{\text{Naive Bayes}}{\approx} \mathbb{P}[Y = y_i] \prod_{j=1}^{N} \mathbb{P}[x_j|Y = y_i] \tag{2.26}$$

However, by using the Naive Bayes assumption, the dependence between the features is completely ignored, even if it is actually there. This means that this approximation underestimates the true posterior distribution grossly. From this start point, Özuysal *et al.* proposed the concept of ferns. They tried to keep the simplicity of Naive Bayes with introducing only some dependent features. In detail they defined a fern as a set of dependent features, and used ensembles of ferns which are independent from each other.

Based on this definition, a fern can be seen as a *Semi-Naive Bayes* classifier, because it is not completely naive. Consider $M$ groups with size $S = \frac{N}{M}$, where each group is referred to as one fern and denoted as $F_k$. Then formally, the conditional probability becomes

$$\begin{aligned} \mathbb{P}[x_1, x_2, \ldots, x_N, y_i] &\overset{\text{Fern}}{\approx} \mathbb{P}[Y = y_i] \prod_{k=1}^{M} \mathbb{P}[F_k | Y = y_i] \\ &= \mathbb{P}[Y = y_i] \prod_{k=1}^{M} \mathbb{P}[x_1, \ldots x_S | Y = y_i] \end{aligned} \tag{2.27}$$

With this approach the joint probability of each fern can be modelled completely with an appropriate $S$. This gives $M \cdot 2^S$ parameters, which can be handled easily.

### 2.6.2 Usage

The procedure of using ferns instead of forests is very similar. In the following paragraphs, fern training and testing is described and the similarities are referenced accordingly.

**Training** As it is the case with random forests, ferns are also trained randomly and therefore they are called random ferns. The fern training is very similar to the training of a random decision tree. The only difference is that the weak learners do not have a hierarchy. Therefore, when a fern with size $S = 10$ is trained, 10 weak learners (cf. Equation (2.7)) are trained, i.e., a binary split function is assigned to each node in the fern. In the original way, each weak learner is trained completely independent by maximizing the information gain (2.10). In other words, each weak learner can be seen as a *decision stump* (Figure 2.10). For more information on how the training is done and how an optimal split can be found see Section 2.5.1.

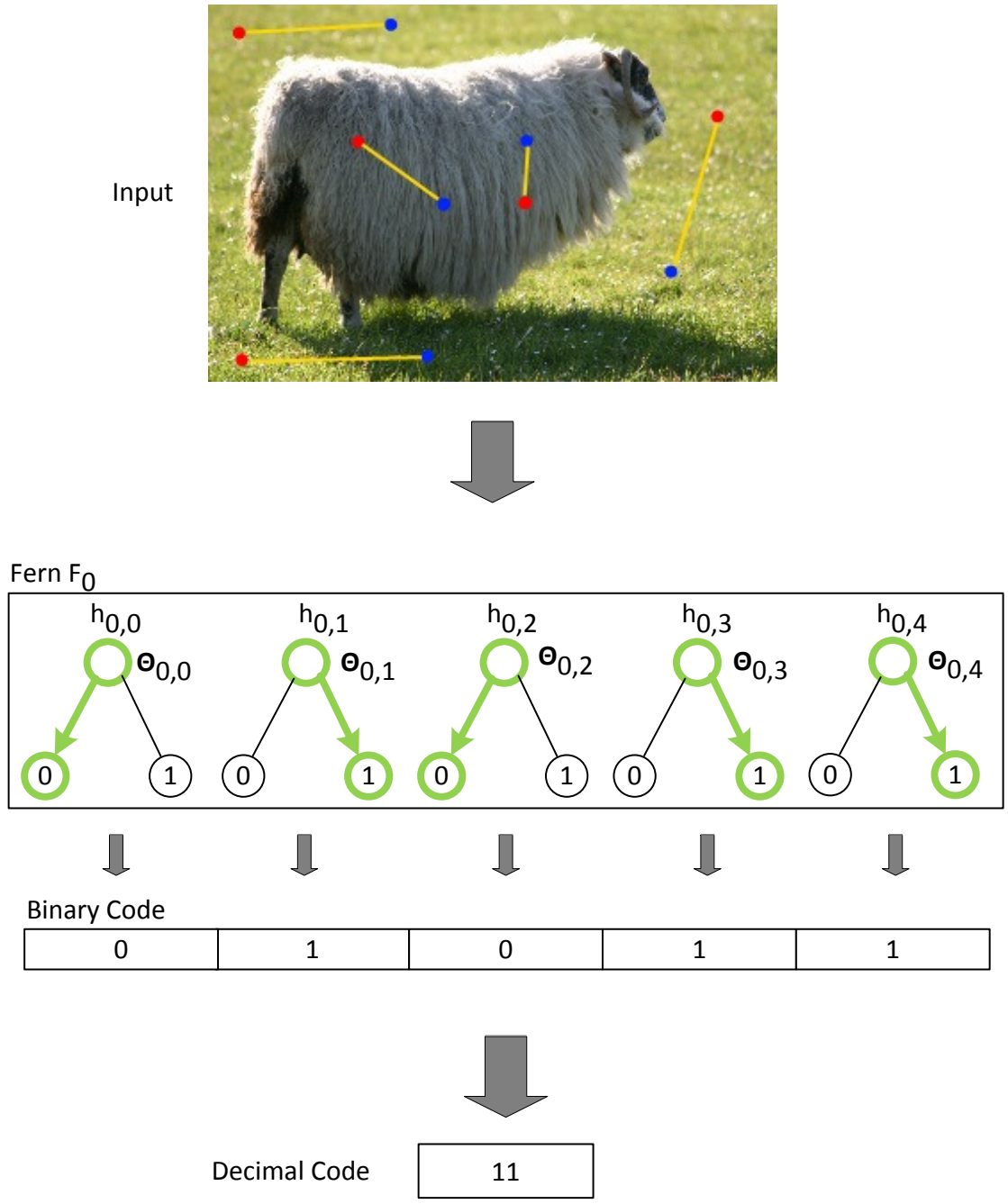



Figure 2.10: Code generation using a fern: In this example the input image is a sheep. Each yellow line represents one binary test, where some property of the red point is compared to some property of the blue point. These tests are actually applied by the weak learners $h_{0,0}, \ldots h_{0,4}$ (decision stumps) in the fern $F_0$. Depending on the outcome of this binary tests, a binary code is generated. To find the entry in the fern statistics, the binary code is converted into a decimal number (see Figure 2.11). The sheep image is taken from the MSRC image database [57].

After all split functions in each fern are defined, the training data is used to generate statistics of the codes. For each of the $|\mathcal{Y}|$ classes an own statistic exists. When a fern is applied to a trainings sample, a code is generated as shown in Figure 2.10. If the sample corresponds to class $y_1$, then the entry at position `code` is incremented by 1. This procedure is done for all trainings samples to form valid statistics for each class. After the training has finished, code statistics have been aggregated as shown in Figure 2.11. And this is a major difference compared to a forest. A forest models directly the posterior distribution over the class labels, i.e., no individual statistics for the classes are generated as it is with ferns.

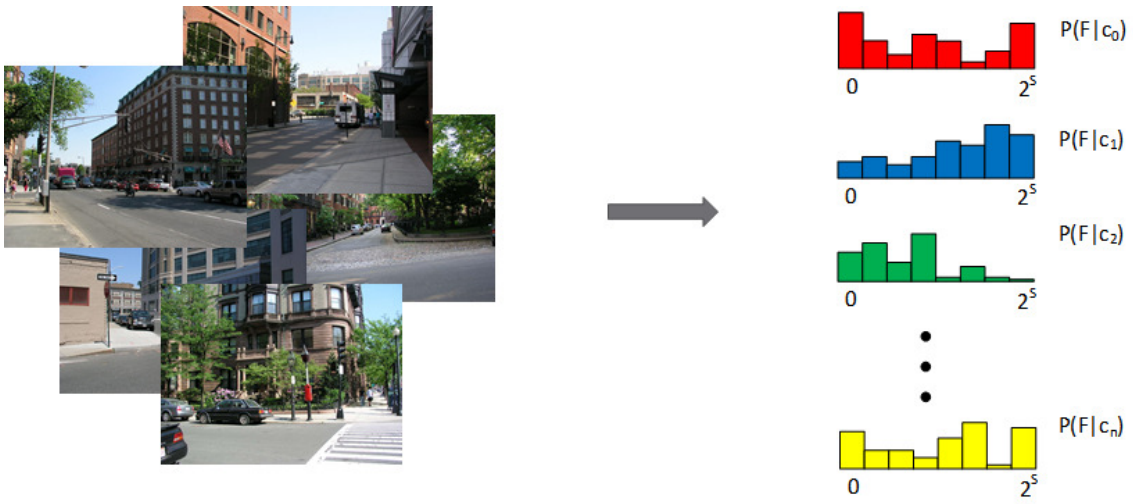



Figure 2.11: Statistics generation: By applying the fern to a bunch of input images, a statistic is generated for each class. Notice that one statistic is generated for each individual class. If a training sample belongs to class $c_1$, then the generated code (see Figure 2.10) is used to adapt the statistic for the appropriate class. The size of the statistic of each class is $2^S$, where $S$ corresponds to the size of the fern. Sample images are taken from the LabelMe database [59].

**Testing** The testing procedure in a fern is different to that of a decision tree, because we do not have the posterior distribution to directly infer the most probable class label. When a new sample should be tested, the fern is applied to the input to generate the binary code. Due to the fact that we do not know the true class label of the sample, we simply use the generated code and construct the posterior distribution over the class labels. This can be done by using the bin of each class statistics where the code points to. This bins can be used to form a new probability distribution where it is possible to look up the most probable class. This concept is visualized in Figure 2.12.

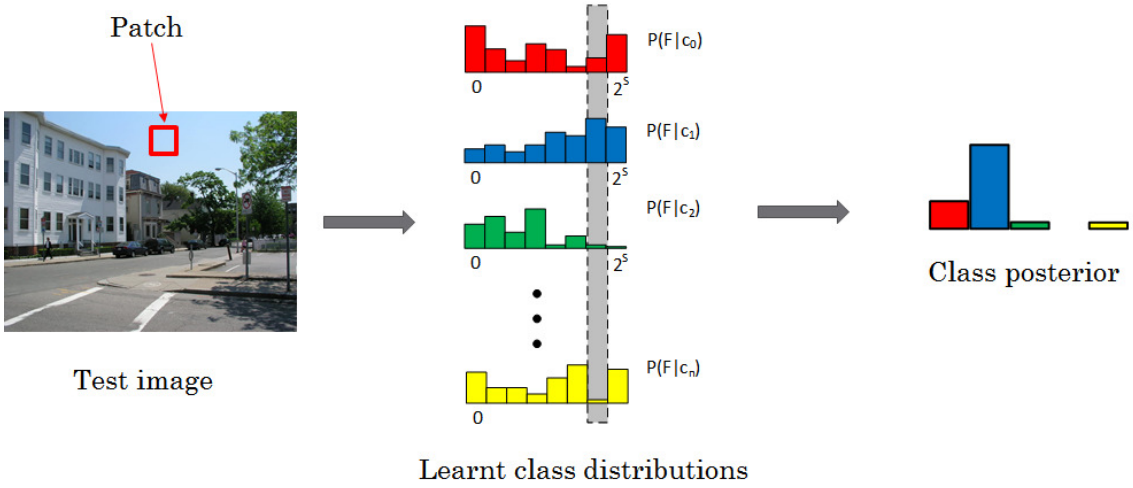



Figure 2.12: Fern testing: When a new sample is tested (e.g., a patch of an image), the fern is applied to the patch to generate the binary code. This code is then used to look up the corresponding bin in each of the class statistics to construct the posterior distribution over the class labels. Test image taken from [59].

**Ensembles of Ferns** Ferns can be combined into ensembles as it is done with decision trees. The outcome of the individual ferns can be combined using Equation (2.17) as it is done by using decision trees.

**Decision Trees versus Ferns** As visualized in Figure 2.9, a fern can be depicted as a tree. This visualization already indicates the similarity of decision trees and ferns. However, there are subtle differences. One difference is that a fern is not hierarchically organized, as it is within a tree. In a decision tree the current weak learner is dependent on the outcome of the previous weak learner, where in a fern each weak learner is applied to incoming data, indifferent of the outcome of the previous weak learner. This simply means that using a fern, each sample gets exactly the same binary questions, where in a forest each sample gets different binary questions until a leaf node is reached.

Second, as [80] pointed out, ferns do need more samples to be as discriminative as forests. However, this seems intuitive, because a decision tree is a *discriminative* classifier, where a fern tries to model the likelihood, i.e., a fern is a *generative* classifier and the posterior probability is constructed from the likelihood.

## 2.7 Boosting

Boosting is a powerful technique to combine multiple "base" classifiers (weak learners) to one powerful strong classifier. The performance of this combined strong classifier can be significantly better than that of any weak learner. If each weak learner performs just a little better than chance, i.e., the probability that the prediction is correct is greater than $\frac{1}{2}$, then the training error of the combined strong classifier drops exponentially fast [21, 22]. However, this will not be the case on the test set. The most widely used boosting algorithm is called adaptive boosting or short *AdaBoost* [20]. AdaBoost can be used for binary and multiclass problems. However, when multiple classes are used, it is rather difficult to reach the error criteria, i.e., that the error is $< \frac{1}{2}$. Friedman *et al.* proposed other boosting variants: RealBoost, GentleBoost and LogitBoost [24].

Precisely, boosting is an additive model. The weak learners are weighted according to their performance and then added together to build the strong classifier. Mathematically, this can be defined as

$$C_M(\mathbf{x}, \boldsymbol{\theta}) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m h(\mathbf{x}, \boldsymbol{\theta}_m)\right) \tag{2.28}$$

where $M$ is the number of weak classifiers, $\mathbf{x}$ is an input vector, $\alpha_m$ is the weight and $\boldsymbol{\theta}_m$ are the parameters for the $m$-th weak learner.

The strong classifier $C_M(\mathbf{x}, \boldsymbol{\theta})$ is created during training. First, each point in the training data gets an equal weight assigned. The training dataset and the weights are used to train a weak classifier, i.e., to find the optimal parameters $\boldsymbol{\theta}_m$ according to a quality criterion (cf. Equation (2.10)). The training of such a weak classifier can be done like we would learn a node in a decision tree (Section 2.4.1). After the weak learner is trained, its weight parameter $\alpha_m$ is calculated depending on the performance of the weak learner. In the next step, the weights of all training points classified incorrectly are increased and the weights of the training points classified correctly are decreased. This procedure ensures that the next classifier will concentrate on those samples which are classified incorrectly with the current strong classifier $C_m$. This procedure continues until $M$ classifiers are trained. An abstract visualization of the training of a boosted classifier is illustrated in Figure 2.13. Figure 2.14 shows real data and how a boosted classifier is trained on it. Below, the AdaBoost algorithm is just outlined. For a more detailed version see [5]. Note that the re-weighting of incorrect classified samples can be problematic when the input data is noisy. In comparison to boosting, a random forest is more noise tolerant.

**Algorithm 1** AdaBoost

**Require:** Training set $\mathcal{X}_l = \{(\mathbf{x}, y)|\mathbf{x} \in \mathcal{X}, y \in \mathcal{Y}\}$
**Require:** Number of weak learners $M$

1: Initialize weights $w_i = \frac{1}{N} \quad \forall i$
2: **for** $m = 1 \ldots M$ **do**
3: Fit a weak classifier $h(\mathbf{x}, \boldsymbol{\theta}_m)$ by optimizing the parameters $\boldsymbol{\theta}_m$
4: Add the current weak classifier $h(\mathbf{x}, \boldsymbol{\theta}_m)$ to the strong classifier $C(\mathbf{x})$
5: Compute weak classifier weight $\alpha_m$ according to the performance of the weak classifier
6: Update weights for all examples
7: **return** The boosted classifier $C(\mathbf{x})$

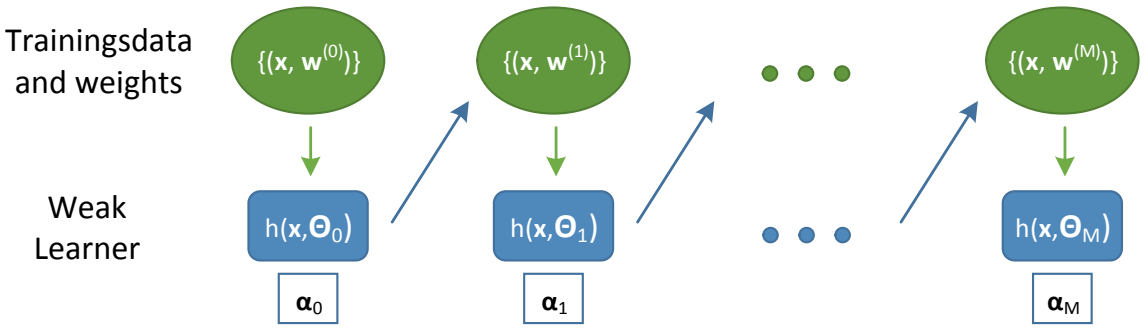



Figure 2.13: Creating a boosted strong classifier. The current weak classifier is dependent on the outcome of all previous classifiers, i.e., the current strong classifier. After a weak classifier is trained, the data points are re-weighted according to the current performance of the strong classifier and a weight is assigned to the currently trained classifier.

Boosting has become famous in computer vision due to the great success of the face detector developed by Viola and Jones [76]. However, boosting is a very general concept and can be used in various disciplines of computer vision. Shotton *et. al* for example used the JointBoost algorithm proposed in [71] to build a powerful strong classifier for semantic segmentation [66]. They incorporated texture, layout and context in their framework and used so called texture-layout filters to capture texture.

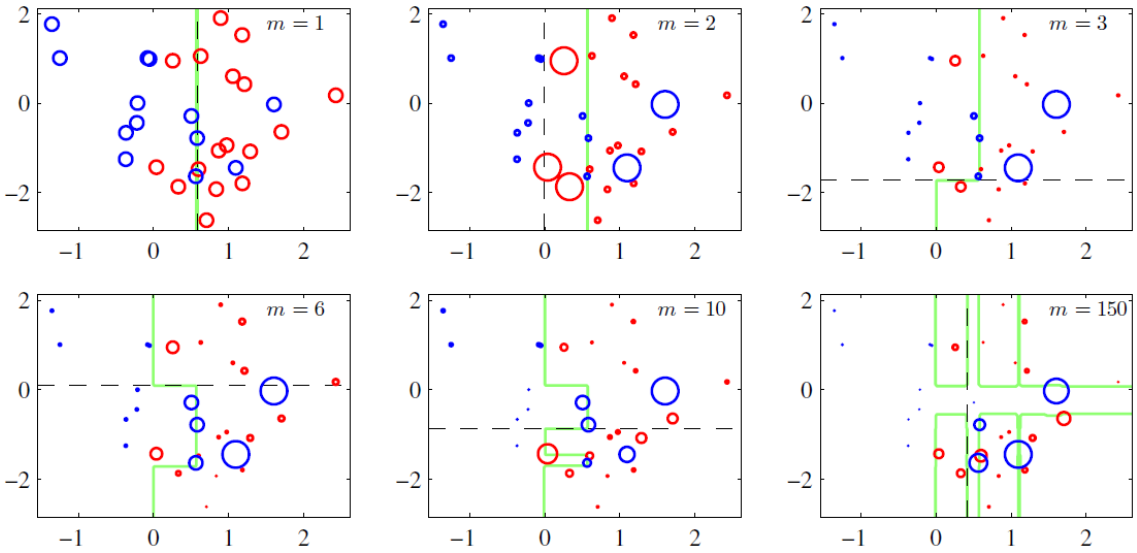



Figure 2.14: Boosting on real data. Each weak learner is a simple threshold applied to one of the axis. The most recent weak learner is shown as a dashed line and the current combined strong classifier is indicated using the green solid line. The size of the data points represents the current weight of the sample. Notice that the weight of incorrect classified samples is increased, where the weight of correct classified samples is decreased. Image courtesy of [5].

## 2.8 Context

In computer vision, the *context* of an object is defined by any data or meta-data not directly produced by the object [29]. Following this definition, context is

- **Nearby image data**
  The local neighbourhood of an object or a pixel.
- **Scene information**
  Where in the scene is the pixel/object?
- **Presence of other objects**
  Are there any other objects present on the image and if, where are they?

In other words, context is everything what stays in junction with an object, without the object itself. Humans use a lot of context information. The less a human knows about a given scene, the more important is the context (see Figure 2.15).

When a single pixel is classified in semantic segmentation, actually a patch around this pixel is used to perform the classification task. This means that the class correspondence

of one pixel is determined only by the appearance of the local neighbourhood of a pixel. This procedure works considerably good as long as the patch contains salient points. However, especially in low textured regions this is not the case. In order to achieve good performance, *context* is very important. Figure 2.15 shows why context is very important to distinguish similar looking local patches.

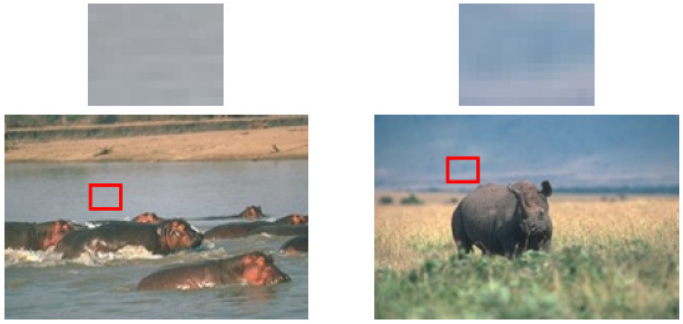


Figure 2.15: Importance of context. By looking at the two image patches on top of the images, even for human it is very difficult to distinguish the categories WATER and SKY based only on the local appearance of the patches. However, if context is incorporated, i.e., looking at the whole image, it is easy to distinguish the two categories. Image courtesy of [28].

Due to the high importance of context, several researchers discovered how to incorporate context in a natural way. One way to incorporate context information is to use Markov Random Fields (MRF) or Conditional Random Fields (CRF) [36]. In such approaches the modelling and computing stages are studied in isolation. Due to the high computational inference cost, context is limited to a fixed neighbourhood structure, which is not enough in many cases. Context is used widely in the literature in semantic segmentation and scene understanding [28, 30, 56, 66].

In difference to many energy minimization algorithms, *auto-context* uses the same procedure during modelling and computing stages. In this approach numerous classifiers are trained where the confidence output for each class of the previous classifier is used as additional (context-) features for training the subsequent classifier. In other words each learnt classifier provides new context information for the next classifier to be trained. Note that the concept is similar to that of boosting (Section 2.7), since the output of the previous classifier is used to generate the input for the following classifier. Auto-context has several advantages [72]:

(1) Auto-context can be incorporated easily into classifier training

(2) Auto-context is significant faster than much existing algorithms (MRF, CRF)

(3) Context is not restricted to the local neighbourhood, but context information is available all over the image

**Entangled Forest** The concept of using the probabilistic output of a classifier as additional feature maps was used by Montillo *et al.* They removed the necessity to train several classifiers to incorporate context information. The used classifier was a random forest trained breadth first (Algorithm 4). This enables the possibility to use the knowledge of stage $n$ to train stage $n+1$. Each depth is denoted as one stage during classifier training. This concept is called *entanglement* and such a random forest is called *entangled decision forest* [47]. With this approach context is incorporated and can be used as it is available. They showed that an entangled decision forest outperforms a forest trained with auto-context and a forest without any context information on a medical CT dataset.

## 2.9 Summary

In this chapter, the most important concepts regarding semantic segmentation are discussed. After the preliminary specification of the used notations in this work, it is described how a semantic segmentation can be derived in a three stage manner based on (1) feature computation, (2) estimation of the MAP probability for all classes and (3) regularizing the result. In Section 2.3, the formal definition of a classifier is stated and how classification fits into a statistical model. The main part of this chapter deals with the classifiers used throughout this work. The first classifier used is a random forest (Section 2.5), which consists of several randomized trained decision trees. It is discussed in detail how a random forest can be constructed automatically based on training data and how it is possible to infer the most probable class label using a random forest. The second classifier reviewed is called random fern (Section 2.6). A random fern can be seen as a semi-naive Bayes classifier, because only groups of features are considered to be dependent and the individual groups are considered to be independent of each other. In Section 2.7, the famous AdaBoost algorithm is described and how many weak classifiers can be combined to one strong classifier. The section is closed with review of context in images and why context is very important to understand what is actually visible on an image.

# Chapter 3

# Feature Channels and Features

In the previous sections the terms feature channel and features often appeared. However, what these terms are actually stand for, was not discussed yet. The term *feature channel* is used for an abstract representation of an image, which can either be derived directly form the RGB source image or it encodes some additional pixel synchronous knowledge corresponding to the source image. Exactly these feature channels are responsible for the quality of the final segmentation. The better the feature channels generalize what is on the image, the easier it is for the classifier to achieve a good performance. Up to now, we have some abstract representations of the image, where the information for the classification task comes from. To actually use this information, *features* are used. A feature combines values in the feature channels at different locations and computes a feature response. The classification is done using these feature responses by comparing them with learnt responses.

To summarize, feature channels in combination with features enable a classifier to actually classify. The next two sections will investigate the feature channels and features used in more detail.

## 3.1 Feature Channels

Feature channels are used to compute feature responses on it instead of computing them directly on the color pixels. With feature channels it is easier to encode ad-hoc domain knowledge that is difficult to learn using a finite set of training data [75].

Fundamentally, one distinguishes between invariant and covariant feature channels. Covariant means that the measure of the feature channel changes in a way consistent with

the image transformation. Invariant is the opposite and means that a measure remains unchanged under some image transformation. For easier understanding, imagine two images with one and the same rose on it. The only difference is that these images have been taken with two different cameras. Hence, the rose on the first image is a little bit more reddish than the rose on the second image. A covariant feature channel would map the differences in color directly to the feature channel, i.e., the feature channel of the first image differs form the feature channel of the second image. In difference, an invariant feature channel would not map the color differences to the resulting feature channel, i.e., the feature channel of the first image is equal to the feature channel of the second image.

In general, invariant feature channels are preferred, since the feature channels itself are not affected by small appearance differences of an object. And that is exactly what one wants. To reuse the example from above, it should be indifferent how the rose does exactly look like, to get a correct classification of it. Invariant feature channels are the key to achieve a good generalization of the classifier.

In the following sections a brief overview of the used feature channels of the application is given. Additionally, they are categorized whether they are covariant or invariant.

### 3.1.1 L*a*b*

Feature channels should provide the information, such that it is possible for the classifier to separate different classes. Obviously, color information can give a good hint of how objects and classes can be separated.

Instead of using the RGB color space, each image is converted into the L*a*b* (also called CIELAB) color space. In the L*a*b color space, the red, green, and blue components of the RGB model are separated into luminance and two color components a and b. The L*a*b* color space takes account of the roughly logarithmic response of the human visual system. Actually, humans can perceive relative luminance differences of about 1% [69]. The advantages of L*a*b* are summarized in the following list:

- Separation of luminance and chrominance
- Perceptually uniform (differences in color correspond to differences in perception)

L*a*b* is a covariant feature channel, since it uses color information directly to generate the feature channel. However, the real texture of objects is an important feature, since many objects can be distinguished just by there texture. In Figure 3.1 the individual

channels L (lightness), A and B (the color-opponent dimensions) are shown. For more information about color, Poynton answered frequently asked questions about color in [53].

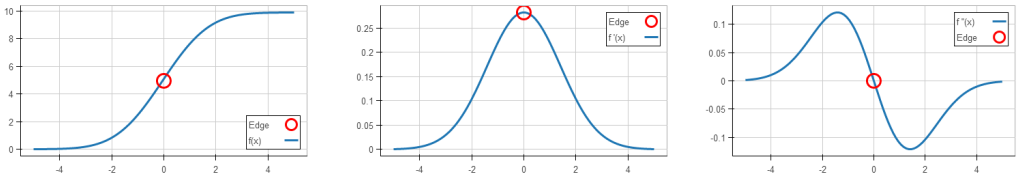
(a) Luminance (b) A (c) B

Figure 3.1: Visualization of the L*a*b* colorspace. L corresponds to lightness, a and b to the opponent color dimensions. Image taken from the MSRC database [57].

### 3.1.2 $1^{st}$ order derivatives

As described in the previous section, color information can contain powerful information for semantic segmentation. However, a classifier should not just split different classes dependent on its texture, but also boost the confidence of categories, where the texture of the objects do not match. To achieve this, invariant feature channels are necessary. The color information can be abstracted by using the first derivative of an image.

Before it will be demonstrated how to calculate the gradient of a 2D image, imagine a 1D image. In Figure 3.2, such a 1D image, i.e., a 1D signal, is shown.




(a) Intensity change (b) First derivative (c) Second derivative

Figure 3.2: Derivatives of a 1D image. In 3.2a the edge is at the point of inflection, in 3.2b the edge is at the maximum response and in 3.2c the edge is located at the zero crossing. Notice that it is easier to locate the exact position of the edge in 3.2b and even easier in 3.2c.

To actually calculate the derivatives of a 2D image the Sobel operator can be used. With this operator it is possible to calculate the gradient in $x$- and in $y$-direction. This operator is nothing but a *kernel* and therefore, it is possible to calculate the gradient images with a convolution. Convolution is defined as

$$g(i,j) = \sum_{k,l} f(i-k, j-l)h(k,l) = \sum_{k,l} f(k,l)h(i-k, j-l) \tag{3.1}$$

where $f(i,j)$ is the input image, $h(k,l)$ is called *kernel* or *mask* and $g(i,j)$ is the result, i.e., the input image convolved with the mask. Convolution is often written as

$$g = f * h \tag{3.2}$$

where * is the convolution operator.

To actually compute the gradient of an image, the Sobel operator is used. Since an image is a 2D signal, the derivative in $x$- and $y$-direction respectively must be computed. The Sobel operator in $x$-direction is defined in Equation (3.3) and the Sobel operator in $y$-direction is defined in Equation (3.4).

$$\mathbf{S}_x = \begin{array}{|c|c|c|}\hline -1 & 0 & 1 \\ \hline -2 & 0 & 2 \\ \hline -1 & 0 & 1 \\ \hline \end{array} \tag{3.3}$$

$$\mathbf{S}_y = \begin{array}{|c|c|c|}\hline -1 & -2 & 1 \\ \hline 0 & 0 & 0 \\ \hline 1 & 2 & 1 \\ \hline \end{array} \tag{3.4}$$

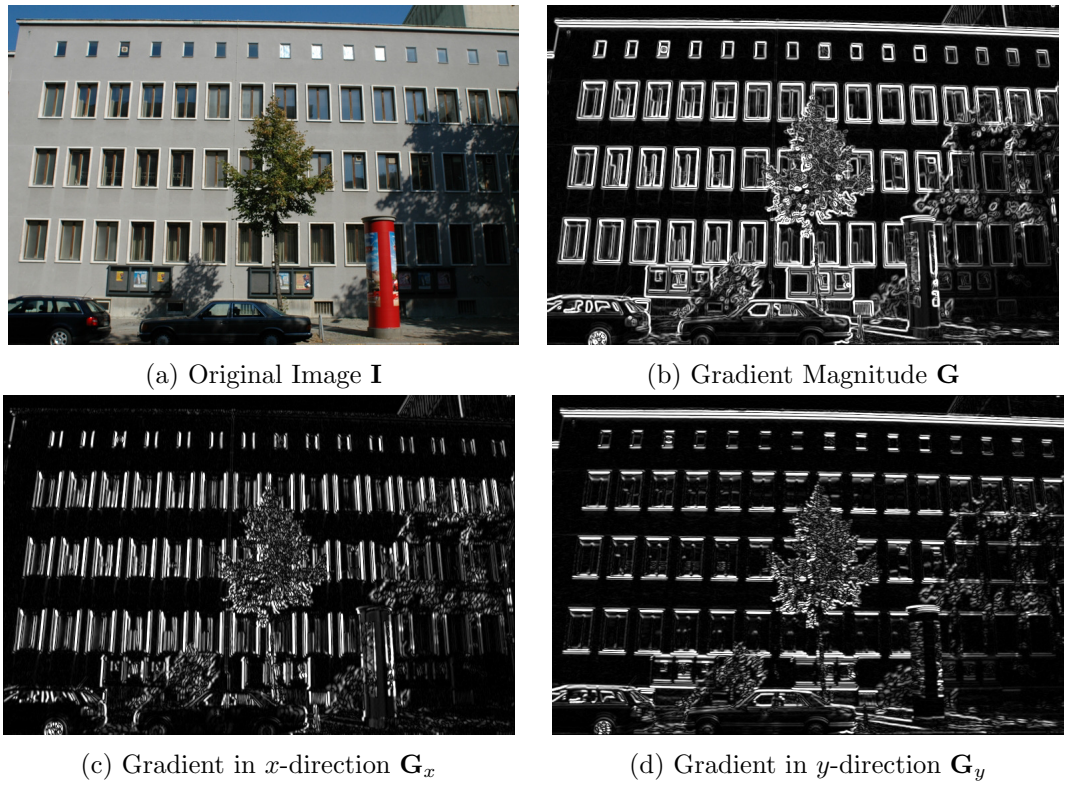
The gradient images $\mathbf{G}_x = \mathbf{S}_x * \mathbf{I}$ and $\mathbf{G}_y = \mathbf{S}_y * \mathbf{I}$ can be calculated by a convolution of the input image $\mathbf{I}$ with the filter kernels $\mathbf{S}_x$ and $\mathbf{S}_y$. Finally, the gradient magnitude image is calculated using Equation (3.5).

$$\mathbf{G} = \sqrt{\mathbf{G}_x^2 + \mathbf{G}_y^2} \tag{3.5}$$

As can be seen in Figure 3.3, the Sobel operator is an edge detector. Edge detection is equivalent to find significant changes in image intensity. And this is exactly what the Sobel operator does. The greater the change in intensity, the greater the response of the Sobel operator. The position of the edge is considered to be exactly where the response of the Sobel operate reaches the local maximum.

### 3.1.3 $2^{nd}$ order derivatives

With the $1st$ derivative of an image $\mathbf{I}$, we get changes in the intensity of an image. To get the true location of an edge, the local maximum must be found. If one does differentiate



(a) Original Image **I**

(b) Gradient Magnitude **G**

(c) Gradient in $x$-direction $\mathbf{G}_x$

(d) Gradient in $y$-direction $\mathbf{G}_y$

Figure 3.3: Image gradient computed with the Sobel operator. Image taken from the eTRIMS database [35].

the gradient image again, the edge is located exactly at the zero crossing of the resulting image. Therefore, with the second derivative, it is easier to find the exact location of an edge. For the use in the semantic segmentation framework, the second derivative provides a powerful feature channel. It is possible to calculate the second derivative of an image directly by using the so called *Laplace* operator. The kernel of the Laplace operator is defined in Equation (3.6). And as before, the Laplace filtered image can be generated by convolving the input image **I** with the Laplace filter kernel. The Laplacian of an image is visualized in Figure 3.4.

$$\mathbf{L} = \begin{array}{|c|c|c|}\hline 0 & 1 & 0 \\ \hline 1 & -4 & 1 \\ \hline 0 & 1 & 0 \\ \hline \end{array} \tag{3.6}$$

(a) Original Image **I** (b) Laplacian

Figure 3.4: Image **I** filtered with the Laplace operator. Notice that the filter response is equal to zero, exactly where an edge is located. Image taken from the eTRIMS database [35].

### 3.1.4 Local Binary Pattern

Local Binary Pattern (LBP) [50] is an other coded representation of an image. LBP considers the local 8-neighbourhood of a pixel. The intensity value of the center pixel is compared with all neighbouring pixels and the outcome of the test is binary. Each of the neighbouring pixels gets an index of a corresponding 8-bit number assigned. If the test outcome is positive, the bit is set in the 8-bit number. After doing this procedure for all neighbouring pixels, an 8-bit LBP code results. For clarity, Figure 3.5 visualizes the LBP procedure. An extended rotation invariant version of LBP is proposed in [51].

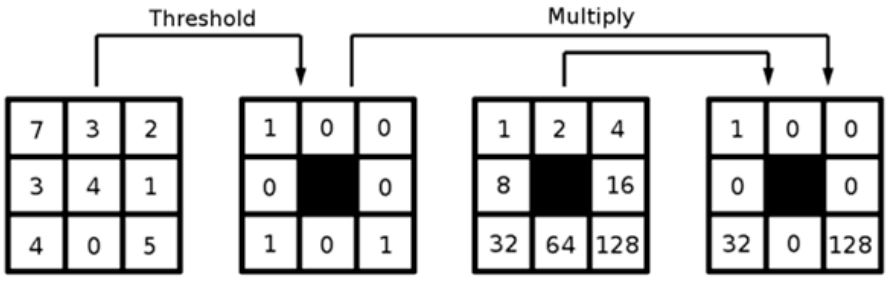



Figure 3.5: Local Binary Pattern (LBP) [50]: the center pixel is compared with its 8-neighbourhood. After the test an 8-bit LBP code is the result. Notice that instead of the multiply-step it is equal to just set the corresponding bit in the 8-bit code.

### 3.1.5 Histogram of Oriented Gradients

One of the most powerful descriptors in object detection is called Histogram of oriented gradients (HOG). In general, HOG is an adaption of scale invariant feature transform

(SIFT) [40]. However, in the original form it was specifically designed to detect upright object categories in images [15]. The procedure can be summarized as follows:

1. **Gradient calculation**
   Compute image gradients in $x$- and $y$-direction. This can be done using the Sobel operator, as described in Section 3.1.2.

2. **Orientation binning**
   In this step, the window is partitioned into a grid of cells, where the cells can either be rectangular or circular (see Figure 3.6). Each cell is contained within a block. A common block size is $3 \times 3$ with a cell size of $6 \times 6$ pixels as it has been used in [15].


(a) R-HOG (rectangular) (b) C-HOG (circular)

Figure 3.6: HOG descriptor blocks. Image courtesy of [16].

   After the window is split into blocks and cells, each pixel casts a weighted vote to an orientation based histogram for the corresponding cells. The weight is based on the gradient magnitude in the specific cell. A histogram with 9 bins between 0° and 180° achieve best results [15].

3. **Normalization of blocks**
   The key in the normalization process is that the blocks are overlapping (see Figure 3.7) and that the normalization is done within each block. After the normalization, the HOG feature vector is a concatenation of the normalized cell histograms for all blocks. Therefore, the HOG descriptor is high dimensional.

By looking at visualizations of the HOG descriptor (Figure 3.8), it is obvious that the shape of an object is encoded with this descriptor. Therefore it is very suitable for object detection and has been used in many recent works.

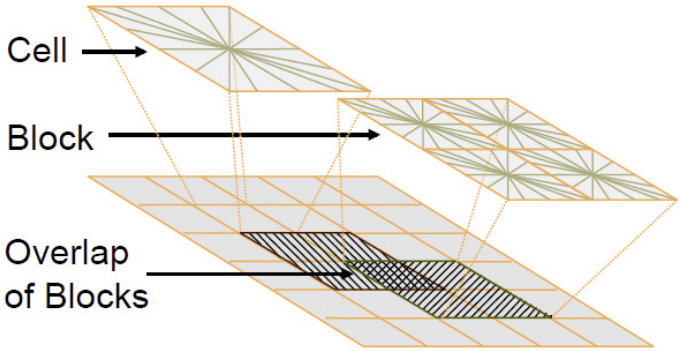
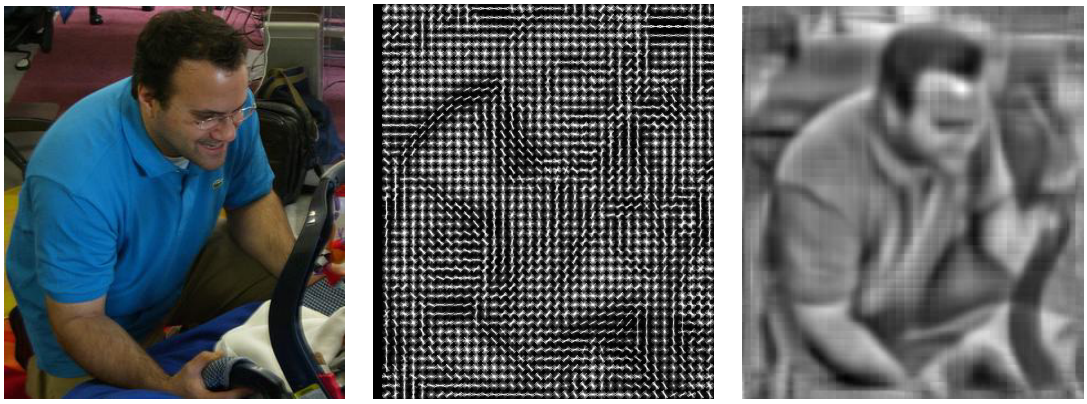



Figure 3.7: Visualization of blocks and cells in the HOG descriptor. Image courtesy of [16].



(a) Original Image **I** (b) HOG (c) Hoggles

Figure 3.8: Different visualizations of the HOG descriptor. 3.8b shows the original visualization presented in [15], 3.8c shows a new visualization presented in [77].

**HOG for Semantic Segmentation** Since the original HOG descriptor was designed for object detection, it is not directly applicable for semantic segmentation. In semantic segmentation, one does not want to have a global descriptor, but a feature channel. In this application a HOG-like feature channel is used. Therefore, the size of a cell is defined to be odd, such that there exists a real center pixel. The histogram of oriented gradients of one cell is assigned to the pixel at the center of the cell. The cell is slid over the image, such that it is possible to extract a HOG for each pixel within the image (without the border). The result of this procedure is a 9-dimensional histogram for each pixel. Now each bin of the histograms forms an individual feature layer. This means, the first bin becomes one feature channel, the second bin becomes one feature channel and so on. Hence, for each image 9 feature channels result, because the orientations are binned into histograms of size 9.

### 3.1.6 Location

The pixel position is another powerful feature channel. This is especially useful in standard databases, since the images in those databases are somehow standardized and focus on some classes present on the images which should be semantically segmented. Location feature are useful, since they encode that it is more probable that the sky appears on top of an image and a street appears rather on the bottom of an image. However, since the images are different in size, the locations are encoded form 0 to 1 in $x$- and $y$-direction and then scaled in a proper way. From the location information, two feature channels can be constructed. The first feature channel encodes the $x$-position of the pixel and the second feature channel encodes the $y$-position of the pixel respectively.

### 3.1.7 Confidence Maps

All feature channels which have been presented in the previous sections can be extracted directly from an input image. The confidence feature channel is the first feature channel presented, where this is not the case. Confidence maps are defined as images, where at each pixel of the image the confidence is encoded, with that a specific class is present at this position on the real input image. Such a confidence map exists for each class. From this definition, it can be seen that the confidence maps cannot be computed out of the RGB input image directly.

Confidence maps have a big impact to the overall performance of the classifier. By using confidence maps as additional feature channels, the classifier learns also context, i.e., what classes are around a specific class.

**Confidence Map Generation** Confidence maps are usually generated by using a second classifier (cf. Section 2.8). Consider one image $\mathbf{I}$, where the confidence maps for $|\mathcal{Y}|$ classes should be generated. In the common classification task after routing all patches $\mathbf{p}_i \in \mathbf{I}$ through the classifier the result is a discrete probability distribution indicating how probable a specific class $y \in \mathcal{Y}$ is. From this probability distribution the confidence that $\mathbf{p}_i$ corresponds to label $y_j$ can be derived. Hence, after the evaluation, each patch is assigned with a discrete probability distribution. Assume now that these probability distributions are aligned in $z$-direction. Then each layer of this tensor corresponds to the confidence map of the $j$-th class. This concept is visualized in Figure 3.9. Examples for confidence maps are shown in Figure 3.10.

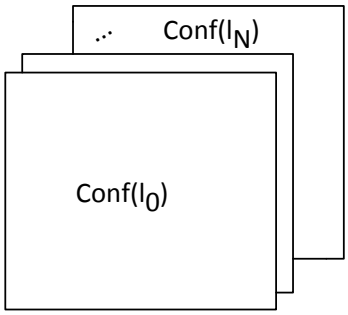



Figure 3.9: Abstract visualization of the layout of the confidence maps

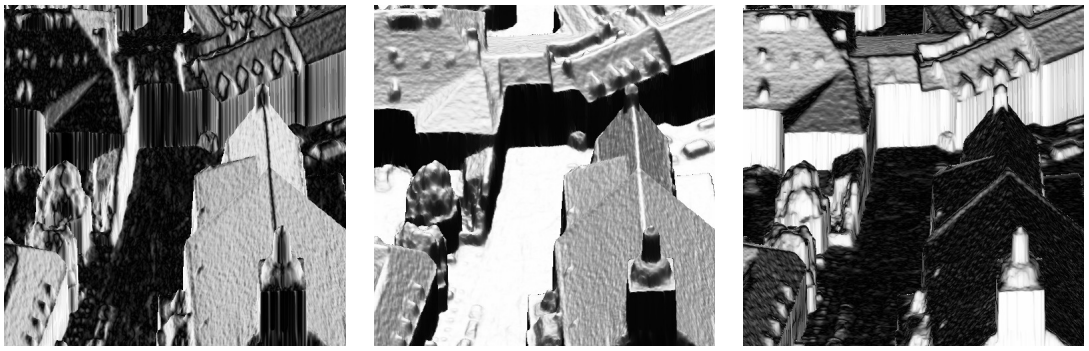
(a) (b) (c)

Figure 3.10: Visualization of confidence maps. In 3.10a, the input image is shown, 3.10b shows the confidences for class CAR and 3.10c shows the confidences for class WINDOW. The whiter a pixel in confidence map $c$, the more confident is the classifier that the pixel corresponds to class $c$. Input image taken from [59].

### 3.1.8 Surface Normals

The surface normals are another feature channel, which cannot be derived directly from an image. However, since the framework provides the capability to do semantic segmentation on textured 3D environments (see Section 5), it is possible to extract pixel synchronous normals. As one can imagine surface normals provide a powerful information source. Just from this features and without any other knowledge it is e.g., possible to split FAÇADE pixels from STREET pixels, since the normals are completely different, i.e., the normals of FAÇADE pixels are horizontal, where the normals of STREET pixels are vertical. To improve generality the normalized absolute values of the normals are used.

**Feature Channel Construction** The surface normal feature channels can be constructed in a similar way as the confidence feature channels has been constructed (see Section 3.1.7). Each component of a normal vector corresponds to one feature channel. Figure 3.11 shows the surface normals of a tile from a 3D scene.



(a) $|n_x|$ (b) $|n_y|$ (c) $|n_z|$

Figure 3.11: Visualization of the feature channels constructed from the surface normals

## 3.2 Features

To actually use the feature channels as information source features are used. In this application *Pixel Pair features* and *Generalized Haar Features* are used. By applying these features on feature channels, a numerical value, i.e., the feature response can be computed. Existentially, this is how $f(\mathbf{x}, \phi)$ in Equation (2.11) is computed.

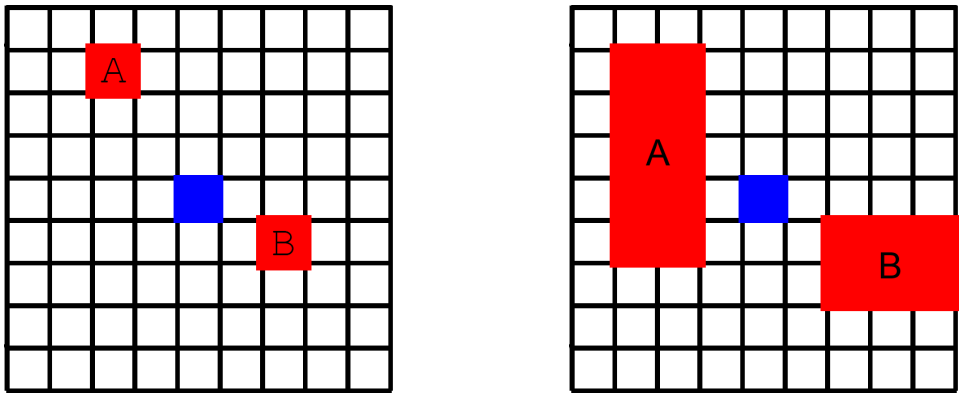
### 3.2.1 Pixel Pair Feature

Pixel pair features are very simple features. As the name already suggests, the response of a pixel pair feature can be computed by comparing two pixels on a patch. The pixel positions on the patch and the actual feature channel used to compute the feature response is determined in the training stage. The parameter $\boldsymbol{\theta}$ captures all these properties of the feature. A visualization of a pixel pair feature is shown in Figure 3.12a. The values of the pixels can be compared using one of the following comparison modes:

- Difference: $A - B$
- Sum: $A + B$
- Absolute difference: $|A - B|$

### 3.2.2 Generalized Haar Feature

Haar features have been proven to achieve very good results in face detection [75]. They used a set of predefined Haar-like features, to find different interest points in images. This set contains edge features, line features and the four-rectangle features (see Figure 3.13).




(a) (b)

Figure 3.12: Features. In 3.12a a pixel pair feature on a patch is visualized. The feature response can be determined by comparing the two selected pixels, A and B, using one of the above defined comparison modes. In 3.12b one sample of a Haar feature is visualized. Because we are using generalized Haar features, the size of the rectangles A and B, i.e., their width and height respectively, is arbitrary. The response of a Haar feature is determined by applying a comparison mode on the sum of all pixels under the rectangles. Image courtesy of [23].

These features can be seen as convolution kernels (cf. Equation (3.1)). Each feature responses a single value obtained by comparing the sum of pixels under the black rectangle with the sum of pixels under the white rectangle.

Figure 3.13: Visualization of the simple Haar features used in [75]. First and second are edge features, third and fourth line features and the fifth is a four-rectangle feature.

In this application we do not use pre-defined Haar features as shown in Figure 3.13, but generalized Haar features. Generalized means that the size and the position of the rectangles in the feature is arbitrary. This properties are determined during the training stage of a classifier and optimized to generate a feature which results in an information gain as big as possible. A generalized Haar feature as used in this thesis is visualized in Figure 3.12b. Note also that the pixel pair feature is a special case of the generalized Haar feature, where each rectangle is quadratic with the side length equal to 1.

**Efficient Computation of Rectangular Features** One might think that the computational cost of Haar features is high in comparison to pixel pair features. In fact, when *integral images* are used, it is possible to compute the response of Haar features in constant time. An integral image contains the sum of pixel values of an image as defined in Equation (3.7). An arbitrary rectangle can be calculated using Equation (3.8) in $\mathcal{O}(1)$. Figure 3.14 shows how an integral image works.

$$\mathbf{I}_\Sigma(x, y) = \sum_{x' \leq x, y' \leq y} \mathbf{I}(x, y) \tag{3.7}$$

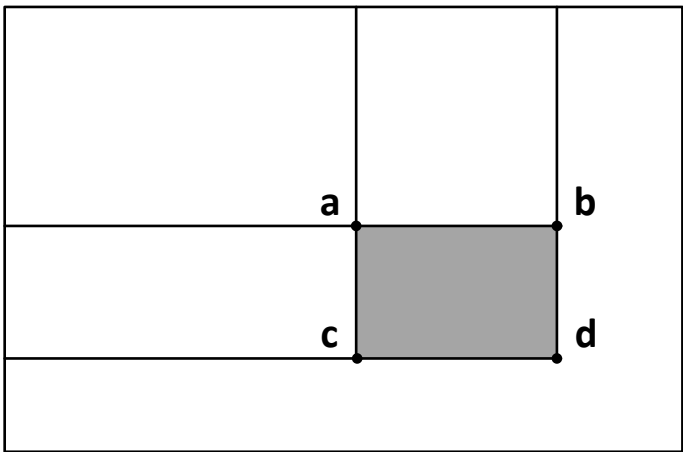



Figure 3.14: The value of the integral image at point **a** contains the sum of all pixels from the top left point to **a**. The same holds for **b**, **c** and **d** respectively. The sum contained in the rectangle $\overline{\mathbf{abcd}}$ can now calculated using Equation (3.8). Hence, independent of the size of the rectangle only three operations are necessary, which results in constant runtime $\mathcal{O}(1)$.

$$\mathbf{I}_\Sigma\left(\overline{\mathbf{abcd}}\right) = \mathbf{I}_\Sigma(\mathbf{d}) - \mathbf{I}_\Sigma(\mathbf{b}) - \mathbf{I}_\Sigma(\mathbf{c}) + \mathbf{I}_\Sigma(\mathbf{a}) \tag{3.8}$$

## 3.3 Summary

In this chapter, the feature channels used in this application (see Section 5) are presented and it was shown, how they can be computed out of an image. The feature channels define an abstract representation of the image content. Through this abstraction it is possible to distinguish different classes not only on texture, but also on more advanced information like histograms of oriented gradients or local binary patterns. It is important that invariant feature channels are used as discussed in Section 3.1. The second part of

the chapter dealt with the features applied on the feature channels. Therefore pixel pair features and generalized Haar features have been presented and it was shown how they can be computed very efficiently by using integral images.

# Chapter 4

# Online Learning

As the title of the thesis already suggests, the target in this thesis is to build an interactive segmentation tool. The term "interactivity" indicates that an online capable classifier is necessary to perform this task. Before going into more detail about how random forests (Section 2.5) and random ferns (Section 2.6) can be used as (semi-)online classifiers, it is described what online learning actually is, how it differs from offline learning and what special requirements are made to an online classifier.

**Offline Classifier** Offline learning is the common setting in machine learning applications. The term "offline" indicates that all the data on which the classifier is trained is available at training time. That is, the classifier is said to work in *batch mode*. The advantage of this setting seems obvious. All training data is available and therefore the classifier sees the complete diversity of the dataset and is able to choose the best separation criterion according to all data. After the training phase, nothing of the classifier changes anymore. The one and the same classifier is then used on test data to predict the most probable class labels based on the initial learnt training data.

**Example** To give a real world example, assume for now that a human is a classifier and a library is the training data. In the offline setting described above, the human goes into the library and reads every book without even leaving just one out. This corresponds to the training stage in the machine learning process. After this stage, the human leaves the library and tests his knowledge in the real world to see, what he has learnt.

**Online Classifier** In the online setting, there is nothing like a training stage or a test stage. Online means that data used for training the classifier can arrive at arbitrary

time. When new training data arrives, the classifier uses the newly available information to *update* the current classifier. This requires to change the classifier dependent on the information given by the new training data. Therefore, an online classifier has the ability to change over time, which gives the possibility to adapt the underlying model until a satisfactory result can be achieved completely automatically.

**Example** Again, for a real world example, assume that a human is a classifier for now. But this time, he does not go to the library, since all training data is not available at once. In this setting, the human has a guide, who teaches him what to learn. This is very similar as it is in our real world. Human go to school and a teacher provides selected information, what should be learnt. And exactly this happens in the online setting. The human in the loop teaches the classifier with selected information, i.e., information which is not known yet by the classifier, and provides them on demand.

To summarize, by using an interactive approach to do semantic segmentation, it is required that the chosen classifier can deal with data arriving *online* or *sequentially*. Table 4.1 reveals the differences between an offline and an online classifier. It can be seen that an online classifier must be able to handle training data arriving at any time. However, online classifiers also expand the domain, where they could be used. Their big advantage is that it is not necessary that all training data is available at once. And this is the case in many real world scenarios. It has been shown in [60] that the performance of an online classifier converges to the performance of an offline classifier when the number of samples increases. In addition, the performance of an online classifier can be enhanced when additional training samples are provided.

| | **Online** | **Offline** |
|---|---|---|
| Training | when new training data arrives | only during training stage |
| Testing | every time | after training stage |

Table 4.1: Offline versus online: Training and Testing

**Continuous Learning** By using an online classifier, the learning procedure can be seen as continuous learning. In difference to offline learning, this means that the classifier is not trained once, but continuously. Hence, the first challenge is to allow the classifier to adapt if the underlying classes change over time. How this adaption can be performed is discussed in Section 4.2. A second, even more difficult problem is how new labelled

data can be extracted, such that the classifier can be trained with the new data. In this application (Section 5) the training data is selected by a human operator and therefore this task is no problem. The user selects the samples, such that the classification performance of the classifier can be increased as much as possible or until no performance increase is possible anymore.

## 4.1 Online Random Forests

One of the requirements on a classifier used in an interactive approach is to incorporate training data arriving at any time (cf. Section 1.4.1). However, standard random forests as discussed in Section 2.5 are designed to learn in batch or offline mode. The following example demonstrates why random forests are not inherently online.

**Example** This example shows, why a random forest is not inherently online. Therefore, two forests, forest $f_1$ and forest $f_2$, are trained with the same training data. However, the first forest will see all training data in advance as it is in the offline setting whereas the second forest sees only a subset of the training data in advance and the rest of the data arrives online. Without loss of generality a non randomized forest is used in this example. This means, that the very best split function is used at every node. The training set $\mathcal{X}$ consists of $n$ data points.

The first forest, $f_1$, is trained with $n$ training points, i.e., all training data is used to construct the structure of the trees in the forest.

The second forest, $f_2$, is trained with $n-1$ data points, i.e., the structure of the trees in the forest is established with a subset of all training points. Afterwards, the missing $n$-th data point arrives. An online classifier is able to incorporate this $n$-th data point, such that the resulting classifier is equal to a classifier trained with all data points at once, i.e., after incorporating the $n$-th data point $f_1$ should be equal to $f_2$. Nevertheless, due to the recursive structure of a forest, the new data point affects the complete structure, i.e., $f_1 = f_2$ is only possible by completely retraining $f_2$ with *all* training points. Therefore, a forest is not inherently online.

Saffari *et al.* proposed a method to use random forests online [60]. This method can be seen as a tree growing approach. In fact, they start with a single root node in each tree and maintain statistics for a set of candidate splits. When a new sample arrives, these statistics are updated and the potential information gain of all candidate splits is

measured. A split is actually done, if there have been enough samples in a node to form a robust statistic and the information gain is above a threshold. In this setting new training data can be incorporated online without the need of completely retraining the complete forest.

## 4.2 Semi-Online Random Forests

When some data (with or without labels) is available in advance, this data can be used to build a random forest with more robust statistics than in the completely online case. In the following sections two possible ways to make a random forest *semi-online* are discussed. They are said to be semi-online, because the structure of the tree is built once with the available data and only the statistics in the leaf nodes are updated accordingly when new training data arrives.

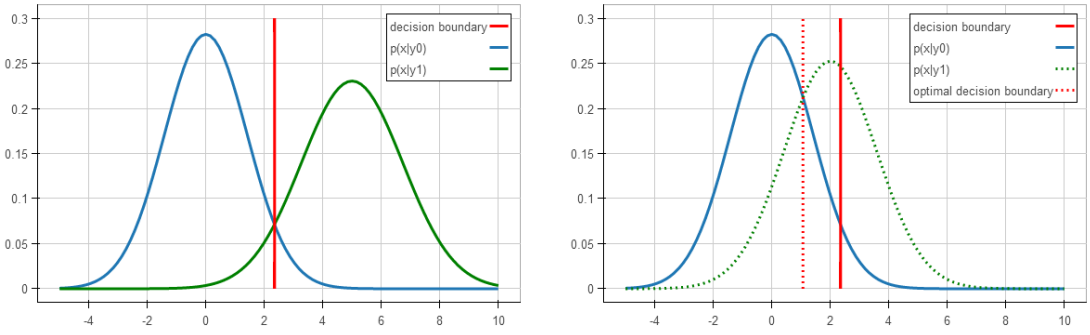
### 4.2.1 Supervised pre-training

Consider a set of labelled training data $\mathcal{X}_l$ and a set of unlabelled test sets $\{\mathcal{X}_u^i\}$. First, the classifier is trained exactly with the same procedure as described in Section 2.4.1, i.e., the structure of each tree and the statistics in the leaf nodes are constructed with the labelled training data $\mathcal{X}_l$. This can be seen as a pre-training step, where the classifier learns the underlying distribution of the labelled input data. Note that the structure of the trees learnt in this step keeps fixed.

**Classifier Update** In difference to an offline forest, in the semi-online setting the statistics of the leaf nodes are not frozen. When a new training point arrives, this data point is routed through the trees in the forest until it ends in a leaf node. However, in this case the statistic in the leaf node is not used to infer the most probable class label, but since the class label is known, the new sample is used to update the statistics with the new training point in the appropriate leaf nodes. With this approach, it is possible to incorporate new training data online.

A similar concept as described above has been used by [34]. The advantages of this approach are visualized in Figure 4.1. When a test set $\mathcal{X}_u^i$ is similar to the training set, then the classifier will perform very well, because the learnt distribution will represent the data in the test set as well (see Figure 4.1a). In this context, the term “similar” means that the test images were taken e.g., with the same camera or at a similar daytime. However,

the difficulty arises, when the test data differs from the training data. In this case, the performance of the classifier will not be that good. Due to the usage of the labels in the training stage, the classifier is very dependent on the training data and its distribution. However, in this semi-online approach, the optimal decision boundary can be found by updating the classifier with new samples (see Figure 4.1b).




(a) (b)

Figure 4.1: Class likelihood of training and test set. Figure 4.1a shows the class likelihood for the classes $y_0$ and $y_1$ of the data used for classifier training. The red vertical line indicates the optimal decision boundary learnt by the classifier during the pre-training stage. When the test set is similar to the training set, the class likelihoods will also be similar and therefore the classification will be good. However, if the class likelihoods of the test data differ (Figure 4.1b), then the learnt decision boundary is not optimal anymore. In 4.1b, the likelihood of class $y_1$ is shifted. By using a semi-online approach it is possible to shift the decision boundary towards the dashed red line.

### 4.2.2 Unsupervised pre-training

Another alternative used throughout the literature is to train a forest without the use of labels. In [26] the structures of the trees are initialized completely random, i.e., they did not optimize the splitting tests or thresholds in the tree nodes to avoid overfitting. However, this strategy allows splits, which do not separate any data at all and this contradicts with the discriminative behaviour of forests.

Vezhnevets *et al.* went one step further with their extremely randomized hashing forests [74]. They constructed the forest without looking at any class labels. They forced the split to actually separate some data, such that at least one sample is split from all others. This is in contrast to [26], where trivial splits are possible.

In this work, we want to go even further. We want to built an unsupervised pre-trained

random forest which is as strong as possible. By following the maximum entropy principle, the information gain can be calculated based on how balanced the resulting leafs are (see Equation (4.1)). The more balanced the resulting child nodes, the better the split. This enables the forest to improve its strength while maintaining a compact representation.

$$I = 1 - \frac{||\mathcal{S}^L| - |\mathcal{S}^R||}{|\mathcal{S}^L| + |\mathcal{S}^R|} \tag{4.1}$$

As can be seen in Equation (4.1), the splits are computed only based on the size of the samples travelling to the left and to the right child after the split. No label information is used. This is, only the structure of the tree is built during this stage, i.e., there do not exist statistics in the leaf nodes after this stage. The advantage considering generalization of the forest is that the decision trees do not look on labels to distinguish different classes, but try to group similar data together. Note that the structure of the pre-trained forest stays fixed. However, this is no disadvantage, because the structure of the forest is general (no class labels have been used) and discriminative.

**Classifier Update** The update of the random forest can be done exactly as in the case of supervised pre-training. When a new sample arrives, it is routed through all trees in the forest and updates the statistics accordingly.

Note that it is possible built the structure of the trees in the forest with unsupervised pre-training even if the training set consists of labelled data. In this case, the labelled training set can be used to immediately *update* the classifier with the samples in the training set to initialize the statistics in the leaf nodes.

### 4.2.3 Implementation Details

In this section, it is described, how a random forest can be actually implemented. There are some additional subtleties to consider implementing a random forest. To begin, assume the standard supervised scenario.

**Weights for Training Samples** Each sample, i.e., each data point $\mathbf{x}$, from the training set has a label $y$ assigned to it. However, the number of labels of each class is different. This leads to a prior to that classes, where more samples exist (e.g., SKY) in comparison to classes where rather few samples exist (e.g., ENTRY). In general, one does not want to give a prior to some classes in semantic segmentation. To get an uniform prior for all

classes independent of their appearance frequency in the training set, each sample gets a weight $w$ assigned. Before the samples from the training set are used in the trainings procedure, the samples are re-weighted to get an uniform prior distribution. This can be done using Equation (4.2) or Equation (4.3).

$$w_{y\in\mathcal{Y}}^{(t+1)} = \frac{\sum_{y\in\mathcal{Y}} w_y^{(t)}}{w_y^{(t)}} \tag{4.2}$$

where $w_y^{(t)}$ is the initial weight for class $y$ and $w_y^{(t+1)}$ is the updated weight after the normalization. This can also be expressed in terms of probabilities as

$$w_y = \frac{1}{\mathbb{P}[Y = y]} \tag{4.3}$$

where $w_y$ is the weight of class $y$ after normalization.

The first algorithm presented is the original algorithm proposed by Breiman [10]. The pseudo code of the algorithm can be seen in Algorithm 2.

**Algorithm 2** Random Forest [10]

**Require:** Training set $\mathcal{S} = \{(\mathbf{x}, y)|\mathbf{x} \in \mathcal{X}, y \in \mathcal{Y}\}$
**Require:** Forest Size $T$
**Require:** Maximum tree depth $D$

1: **for** $t = 1, \dots T$ **do**
2: Bootstrap training samples (= sample with replacement) from $\mathcal{X}_l$
3: TrainDepthFirst($\mathcal{S}$, $D$) OR TrainBreadthFirst($\mathcal{S}$, $D$)
4: **return** The final forest F

As can be seen in Algorithm 2, the real training procedure can be done either with depth first training or with breadth first training. In fact, the difference lies in the order of when a specific node in the tree is trained (see Figure 4.2).

Depth first training is easier to implement then breadth first training. The first step in the depth first training algorithm is to check, whether the training should be continued. When this is the case, one node is trained, i.e., the optimal parameters are derived by sampling a number of different weak learner. The parameters of the weak learner with the highest information gain (cf. Equation (2.10) and Equation (4.1)). Afterwards this weak learner is used to split the data into two sets $\mathcal{S}^L$ and $\mathcal{S}^R$. The node training is continued recursively until all nodes are trained (Algorithm 3).

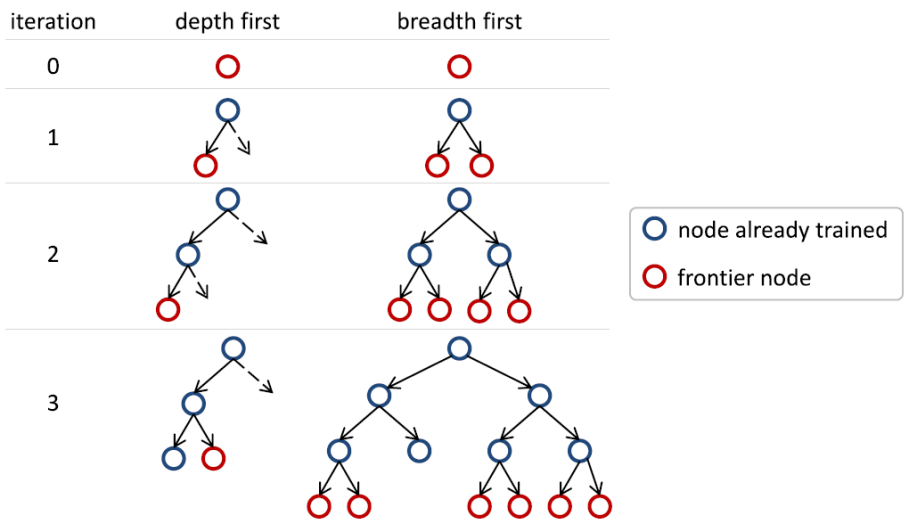



Figure 4.2: Depth first vs. breadth first training. In depth first training the nodes are recursively trained. In breadth first training, all nodes of depth $d$ are trained before the training continues with depth $d+1$. Frontier nodes define a set of nodes trained in the next stage. Image courtesy of [13].

**Algorithm 3** Depth First Training

**Require:** Set of training data $\mathcal{S}$
**Require:** Maximum tree depth $D$

**function** TrainDepthFirst($\mathcal{S}$, $D$)
  **if** $d < D$ and $\exists$ non-pure node **then**
    $\boldsymbol{\theta}^* =$ FindBestSplit($\mathcal{S}$)
  **else**
    GenerateLeafStatistics($\mathcal{S}$)

  *// Call recursively on left and right subset*
  $(\mathcal{S}^L, \mathcal{S}^R) =$ SplitData($\mathcal{S}$, $\boldsymbol{\theta}^*$)
  TrainDepthFirst($\mathcal{S}^L$, $d+1$)
  TrainDepthFirst($\mathcal{S}^R$, $d+1$)
**return** depth first trained tree

An alternative to depth first training is breadth first training. In this algorithm, all nodes of a specific depth $d$ in the tree are trained in common. Breadth first training is more difficult to implement. However, a breadth first trainings algorithm is required to implement an entangled decision forest [47], because based on the output of one specific depth $d$ additional feature channels are computed and used for classifier training. The first step in the algorithm is to compute the so called frontier nodes of a specific level with

their learnt parameters $\boldsymbol{\theta}_j$ and the subset of samples present at each node $\mathcal{S}_l \subset \mathcal{S}$. If the maximal depth $D$ is reached, the statistics for all leaf nodes are generated. Otherwise, all frontier nodes are trained by searching the best parameters $\boldsymbol{\theta}^*$ for the weak learner. The breadth first training algorithm is listed in Algorithm 4.

**Algorithm 4** Breadth First Training

**Require:** Set of training data $\mathcal{S}$
**Require:** Maximum tree depth $D$

```
1: function TrainBreadthFirst(S, D)
2:     for d = 0, ..., D do
3:         L = GetAllFrontierNodes(S, d)
4:         if d == D then
5:             for l ∈ L do
6:                 GenerateLeafStatistics(S_l)
7:             return Breadth first trained tree
8:         // train level d in tree
9:         for l ∈ L do
10:            if l non-pure then
11:                θ* = FindBestSplit(S_l)
12:                θ_l = θ*
```

## 4.3 Random Ferns

The second classifier used in this thesis is a random fern classifier (Section 2.6). Random ferns were chosen (a) due to their similarity to random forests, i.e., only few modifications in the code are necessary to implement random ferns and (b) because they are inherently online. A random forest is highly discriminative and tries to split the data as good as possible. Then each leaf node has its individual statistics, which is used to infer the most probable class label for an unknown sample. In contrast, a random fern does not try to split data, but one fern builds a global statistic for each class. Therefore a fern is a semi-generative classifier. It is not completely generative, because the individual ferns are considered to be independent from each other.

**Classifier Update** Due to the generative behaviour of a random fern, the update procedure is easy. In fact, always when a labelled sample is used for training, the fern is actually

updated. This can be done simply by applying the fern on the new sample, calculate the corresponding fern code and update the statistics of the appropriate class.

**Implementation Details** To implement a random fern classifier, a lot of code from the random forest implementation can be used. Each weak learner in the fern is selected by optimizing the split on a bootstrapped set of training data. After the structure of the ferns is built, the statistics are constructed using all available labelled data (Algorithm 5).

**Algorithm 5** Fern Training

**Require:** Set of training data $\mathcal{S}$
**Require:** Number of Ferns $M$
**Require:** Size of each Fern $S$

**for** $m = 1 \dots M$ **do**
    *// built fern structure*
    **for** $s = 1 \dots S$ **do**
        $\mathcal{S}_B = \text{BOOTSTRAP}(\mathcal{S})$
        $\boldsymbol{\theta}_{m,s} = \text{FINDBESTSPLIT}(\mathcal{S}_B)$

    *// built statistics with training data*
    **for** $\mathbf{x} \in \mathcal{S}$ **do**
        class = GETCLASS($\mathbf{x}$)
        code = APPLYFERN($\mathbf{x}$)
        UPDATESTATISTIC(class, code)

**return** Fern Classifier (set of $M$ ferns)

To infer the most probable class of an unknown test sample (Algorithm 6), the posterior distribution over the class labels must be constructed. This is different to a random forest, where the posterior distribution is saved directly in the leafs. Therefore, each fern is applied on the sample to construct the binary code (see Section 2.6.2 and Figure 2.10). Then the posterior distribution can be formed by using the bins of this code and aggregating the statistics over all ferns.

## 4.4 Domain Adaption

Due to the high complexity of semantic segmentation, the classifiers are often limited to certain kinds of images which are e.g., in the same image database. When a classifier trained on database $A$ is used on another database $B$, the performance of the classifier is

**Algorithm 6** Fern Testing

**Require:** Test sample $\mathbf{x}$
**Require:** Number of Ferns $M$
**Require:** Size of each Fern $S$
**Require:** Number of classes $C$

1: **for** $c = 1, \ldots, C$ **do**
2: $P[c] = 0$

3: **for** $m = 1 \ldots M$ **do**
4: code = ApplyFern($\mathbf{x}$)
5: *// Form posterior distribution*
6: **for** $c = 1 \ldots C$ **do**
7: $P[c] = P[c]$ + GetBin(code, $m$, $c$)

8: **return** Aggregated posterior distribution $P[c]$

decreased usually. However, when a new classifier is trained completely from scratch, the knowledge of the old classifier is lost. Domain adaption tries to adapt a classifier from a *source* domain to a *target* domain (cf. [17]). In this work, the knowledge of a trained classifier is transferred to a new classifier. Therefore, a classifier which was trained on the source domain (e.g., database $A$) is used to compute confidence maps for each class in the target domain (e.g., database $B$). These confidence maps are then used as additional feature channels when the new classifier is trained. With this approach, it is easy to incorporate the knowledge of a trained classifier into a new classifier in a soft way. The only requirement is that both classifiers support the same classes.

## 4.5 Performance Tricks

In order to achieve interactive performance, the implementation is highly optimized. Because the application runs on the central processing unit (CPU), a multi-threaded version was implemented to fully utilize all cores of the CPU during training, updating and testing. A good reference for efficient implementations of decision forests can also be found in [13].

**Tree Node Training** The usage of a randomized forest encourage the computation of the responses on the fly, because it is not known in advance which features will be chosen as candidates actually and it is impossible to pre-compute *all* feature responses due to the

huge search space. This also decreases the memory consumption during classifier training.

For each candidate feature, a threshold needs to be found. This threshold is then used to decide whether a sample is pushed to the left or the right child node respectively. When the tree is not extremely randomized, then in general multiple candidate thresholds between the minimal and maximal feature response are chosen as candidates and the best separating threshold is used [13, 14]. However, we found that it is even more efficient to directly use the median of the responses of all data. The median of an unsorted list can be computed without completely sorting the list. In C++ this can be done as shown below:

```
std::vector<int> responses{5, 6, 4, 3, 2, 6, 7, 9, 3};

std::nth_element(responses.begin(),
                 responses.begin() + responses.size()/2,
                 responses.end());
std::cout << "Median: " << responses[responses.size()/2];
```

**Memory Organization** A further very important thing is how the tree is organized in the system memory. In fact, if the tree is organized well in the memory, the caching architecture of the CPU is used heavily and therefore the overall performance is increased. A simple implementation of one tree is the following. Each node in the tree stores a pointer to the left and right child of it. However, due to the dynamic allocations during the training stage, the nodes will not lie coherent in the memory. Therefore, the performance can be increased by pre-allocating all nodes for the tree in an array. The first node in this array is the root node. Due to the usage of binary decision trees, all other tree nodes can be determined by

$$\mathrm{Idx}(LeftChild) = 2 \cdot \mathrm{Idx}(Parent) + 1 \tag{4.4}$$

$$\mathrm{Idx}(RightChild) = 2 \cdot \mathrm{Idx}(Parent) + 2 \tag{4.5}$$

This inherent relation between parent and child nodes also eliminates the necessity of saving references to parent and child in each node and therefore decreases the memory usage. Additionally, all nodes of the tree lie compact in the system memory and thus the cache is used.

**Parallelization** Modern CPUs have multiple cores which gives the possibility to perform independent computations in parallel. Nowadays, libraries do exist to easily incorporate the power of multiple threads (Microsoft's Parallel Pattern Library (PPL) [46], Intel's Thread Building Blocks (TBB) [31]). These libraries allow to run a `for`-loop in parallel. However, each iteration in the loop must be independent from the others, because each iteration is executed using its own thread.

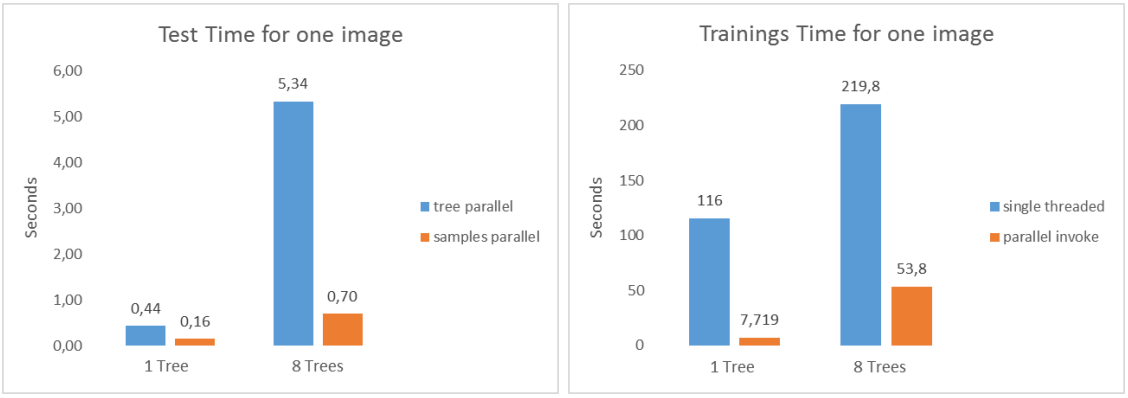
Despite the easy usage of these libraries, it is not ensured that parallelization will increase the performance. Consider for example that threads work on shared data, i.e., they read and/or write to the same memory address. In such a case, the programmer is responsible for a correct synchronisation between the threads. In fact, if there is too much synchronization necessary, a lot of additional computations must be performed and therefore the single threaded version might be more performant. However, the synchronization problem can be mitigated when each thread writes e.g., to its own predefined memory address and the results of all threads are combined in a post-processing step.

In order to parallelize tree training and testing, a lot of different possibilities arise. It makes a big difference, whether one parallelizes over the

- Weak learner
- Images
- Samples
- Trees

In order to get a performance boost through parallelization, keep the following basic principle in mind: The task to perform must be more time consuming than the creation and destruction of the thread itself. Following this principle, it has turned out by an empirical evaluation that parallelization (`parallel_for`) over the samples gives an speed-up of 2.7x - 7.6x compared to parallelization over the trees on evaluating one image (Figure 4.3).

In depth first training, after a node has been trained, two new child nodes are created. The training of these nodes can be parallelized using `parallel_invoke`. Then each node is trained in its own thread leading to a speed-up of 4.8x - 15x on training a forest for one image. All speed tests were performed with an Intel Xeon E5430 CPU running at 2.66 GHz.




(a) (b)

Figure 4.3: Train and test performance. The effect of suitable parallelization can be seen in 4.3a. While parallelizing over the trees is no good idea, parallelization over the samples works very well. When the parallelization is done over the trees, there are too many threads for very little work and therefore the overhead of thread management is big. In 4.3b it is shown how the training time of a forest can be decreased by parallelism. All tests were made on an image of size $640 \times 480$, tree depth $D = 15$ and 128 candidate features are used to find the best split.

## 4.6 Summary

In this chapter, the algorithms used in this work are discussed in detail. The first section clarifies the difference between offline and online classifiers and defines the term continuous learning. Afterwards, the concept of semi-online random forests is revealed. In the first case, a random forest is pre-trained supervised which leads to a high dependence on the labelled training data. In the second case, the pre-training is done unsupervised, such that the labels of the training data are abstracted further. The statistics in the leaf nodes are constructed by updating the pre-trained classifiers with the labelled training data. The concepts of supervised pre-training and unsupervised pre-training are very similar. However, the resulting tree and the performance of both variants is quite different.

Furthermore, different random forest training algorithms (depth first training and breadth first training) as well as algorithms for creating and testing random ferns are presented and outlined. The last section is concerned with performance, because performance is crucial in an interactive classification scenario. It is shown how the training can be done efficiently, how the memory should be organized to allow optimal caching and what the impact of parallelization over different primitives (trees, samples, ...) is.

# Chapter 5

# Interactive Semantic Segmentation

In this chapter, the applications which is the result of this thesis is described in detail. The algorithms and ideas presented in Section 4 are used to build a powerful interactive labelling tool.

## 5.1 Pipeline

Since the training can take quite a lot of time, a classifier is pre-trained on a representative dataset. The training is done unsupervised, which implies a better generalization on different datasets. During this pre-training stage, the structure of the classifier is constructed, such that data can be split best based on unsupervised observations. In other words, in the pre-training stage, the features, which are used in the classifier are selected. At this stage a domain adaption can be done additionally, i.e., the knowledge of a trained classifier can be transferred to the new classifier (see Section 4.4). Notice that after this training stage, the statistics in the leaf nodes are equally distributed, since there are no labels used during the training phase (see Section 4.2.2). However, in an interactive semantic segmentation tool, the user provides the labels and actually constructs the statistics with the annotations.

After the application is started, the user loads the pre-trained classifier and the data for semantic segmentation. Now, the user can draw some scribbles, to teach the classifier the classes, which the classifier should learn and also how they look like.

When the user did all his annotations, i.e., he provided samples of all different labels, which he wants to classify, all annotated samples are used to update the statistics of the classifier. Therefore patches are cropped out of the image and pushed through the classifier

until they update the statistics in the appropriate leaf nodes. After the update is finished, the complete project area, i.e., all images which are currently loaded, is segmented using the current classifier. Hence, the user gets feedback of what the classifier has learnt immediately. Now the user decides, whether the semantic segmentation is already satisfying or not. If there are numerous false classified pixels, it is possible to re-annotate these pixels by drawing some scribbles with the correct labels to update the classifier. This loop can be continued as long as the result is satisfactory. The complete pipeline is visualized in Figure 5.1.

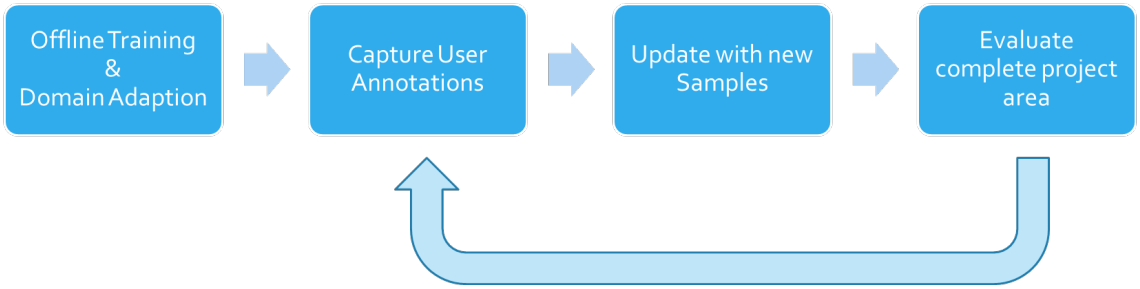



Figure 5.1: The annotation pipeline for interactive semantic segmentation. The initial training of the classifier is performed offline. After the training phase the user can annotate and update the classifier with new labels until the results are satisfying. Note that the domain adaption is optional.

It should be noted that the initial training can be done with the graphical user interface too, despite the training will take some time. For small tasks, where just one image should be segmented, it is useful to train the forest directly on that image. Here the user can decide to train the classifier supervised with initially provided labels. However, by doing this, the classifier will not generalize as well as if it would be trained unsupervised, due to overfitting on the viewed data.

## 5.2 Graphical User Interface

Apart from the algorithmic presented in the previous section, the graphical user interface (GUI) is very important in an interactive semantic segmentation scenario. The GUI handles how images and complete projects are displayed on the screen and provides the tools to allow the user to do a semantic segmentation on an image. In this thesis the

target is to build a strong labelling tool, which can be used to do a semantic segmentation on aerial imagery. The idea is to semantically segment this data into

- Façade
- Window
- Roof
- Street
- Car
- Vegetation
- Water
- Sky

Therefore, the classifier is trained once on a representative dataset and this classifier is then used again and again and updated with new labels on demand. Since the graphical user interface (GUI) allows to intervene and teach the classifier what it should learn at any time, this approach can be seen as life long learning. The concept behind this idea is called *continuous learning* as described in detail in Section 4. The idea is to provide a good structure of the classifier when it is trained. In fact, the structure of the classifier is very important for the strength of the classifier. The better the structure of the classifier, the less tests are necessary to confidentially split data based on their local feature responses. The more projects the classifier has seen yet, the fewer user interaction is necessary to achieve good results. However, if necessary, it should be possible to train a classifier completely from scratch and tune it for a specific task too.

Note that it is possible to learn arbitrary classes with this application. The algorithms do not change when different classes are used. The user defines the classes and their semantic meaning when he annotates an image.

The compute intensive algorithms, classifier training, update and evaluation, are performed in the background by starting additional threads. Therefore, the user can interact with the GUI at any time. For example, it is possible to re-annotate some incorrect labelled pixels during the evaluation of some other images in the project.

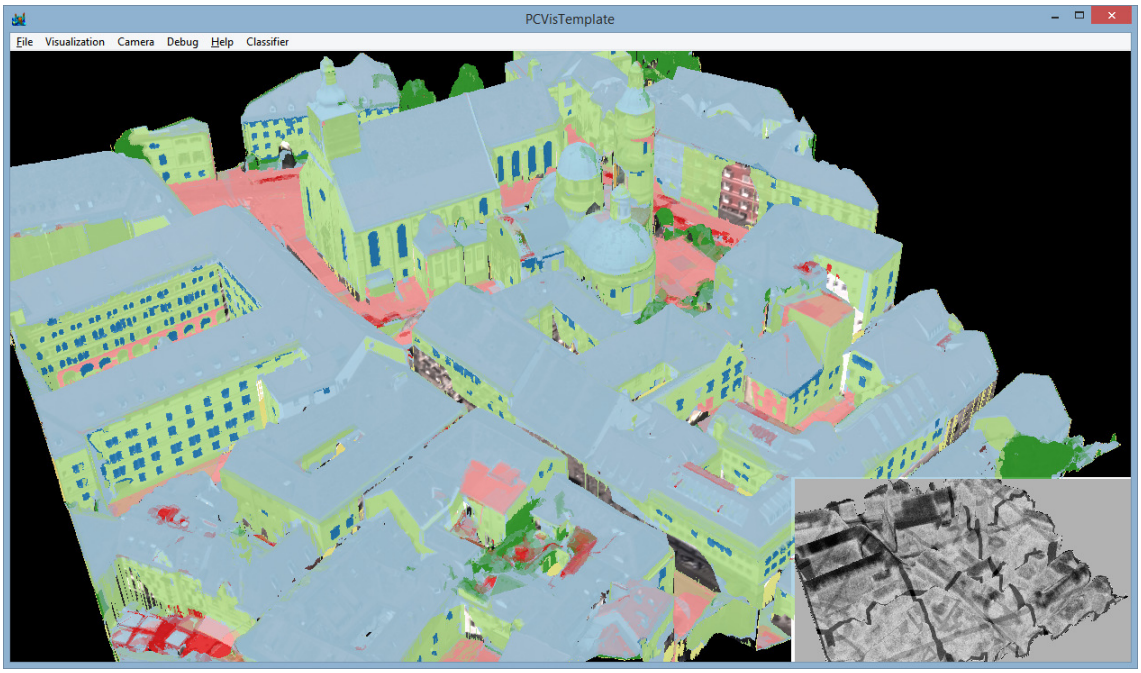



Figure 5.2: Graphical User Interface (GUI) to do a semantic segmentation in 2D and 3D. The Active User Guidance (AUG) is visualized as a map on the bottom right corner (see Section 5.3).

### 5.2.1 Visualization of the Projects

There already exists a viewer to visualize geo-referenced projects and the segmentation functionality should be incorporated into this viewing application. With this viewer it is possible to explore the project area with the mouse. An overview of the basic functionality can be seen in Table 5.1. Actually there are two different types of images to be used for semantic segmentation. These types will be presented shortly in the next paragraphs.

| Key | Action |
|---|---|
| Left mouse button | pan view |
| Right mouse button | rotate view |
| Mouse wheel | zoom in/out |
| Ctrl + left mouse button | draw scribbles |
| Ctrl + mouse wheel | change cursor size |

Table 5.1: Basic functionality of the viewer

**Ortho Images** Orthophotos are geometrically corrected images, i.e., topographic relief, lens distortion and the tilt from the camera position is adjusted. Orthophotos are

constructed synthetically and the procedure of creating such an image is called *orthorectification*. The result of orthorectification is an image where each viewing ray through the camera is orthogonal to the image plane. This equals to an orthographic projection (=orthogonal parallel projection). Since an orthophoto has uniform scale, it is possible to measure distances, areas, angles and positions. The construction of an orthophoto is visualized in Figure 5.3 and an example of an orthophoto can be seen in Figure 5.4.

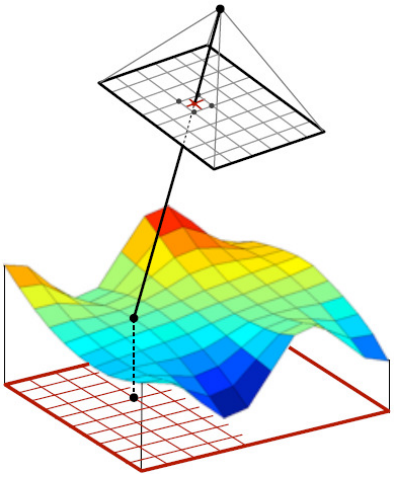


Figure 5.3: Orthophoto construction: The terrain is projected to the $xy$-plane to compute the ortho photo [61].

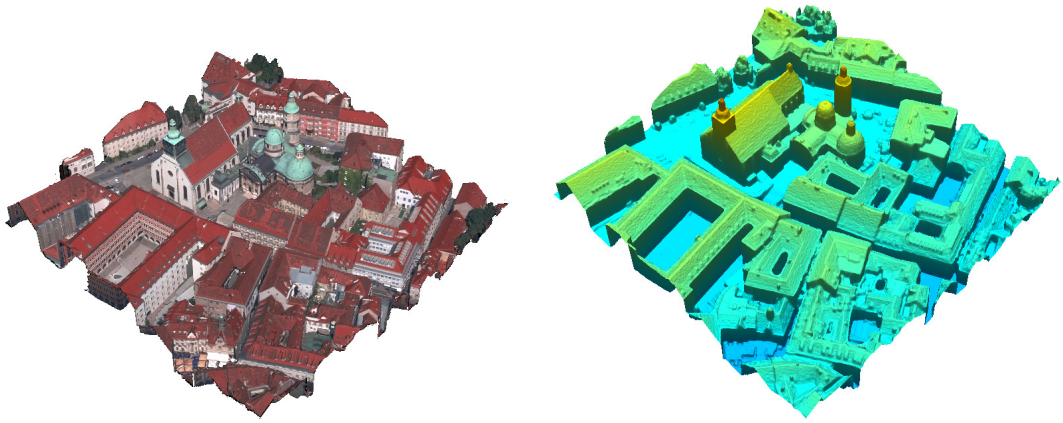
**Oblique Images** Photographs which are taken from an angle are called oblique images. Oblique images are constructed using an oblique projection (=oblique parallel projection). In this projection, the viewing rays are in parallel as it is in the orthographic projection. However, the parallel viewing rays intersect the image plane at an angle and therefore this projection is called oblique. The difference to orthophotos is that not only the top view of a scene is visible, but also façades, windows, etc. can be seen on oblique images.

With the viewer it is possible to load single images and perform an interactive semantic segmentation on them. Due to the fact that the projects are geo-referenced, i.e., each tile of the complete project area has its own global position in a global coordinate system, it is also possible to use multiple images at once. In addition to the images and the global positions, a digital surface model (DSM) is also available. This enables the possibility to project ortho- and oblique-images onto the DSM to get a complete textured 3D city. Technically this is realized using projective texture mapping [18], which is an image based

(a) Orthophoto (b) Oblique photo

Figure 5.4: Example of an ortho photo and an oblique photo. The oblique image was taken under an angle of 45°.

rendering (IBR) method [67].



(a) Textured 3D city (b) Digital Surface Model (DSM)

Figure 5.5: The images are projected onto the DSM using projective texture mapping.

### 5.2.2 Using the GUI for Interactive Semantic Segmentation

To actually perform semantic segmentation on those images, the viewer was extended with new features. After a project has been loaded by the user, it is possible to train a classifier on the images visualized in the GUI. To adjust the settings, i.e., what classifier should be used with what parameters, a configuration dialog was designed (see Figure 5.6).

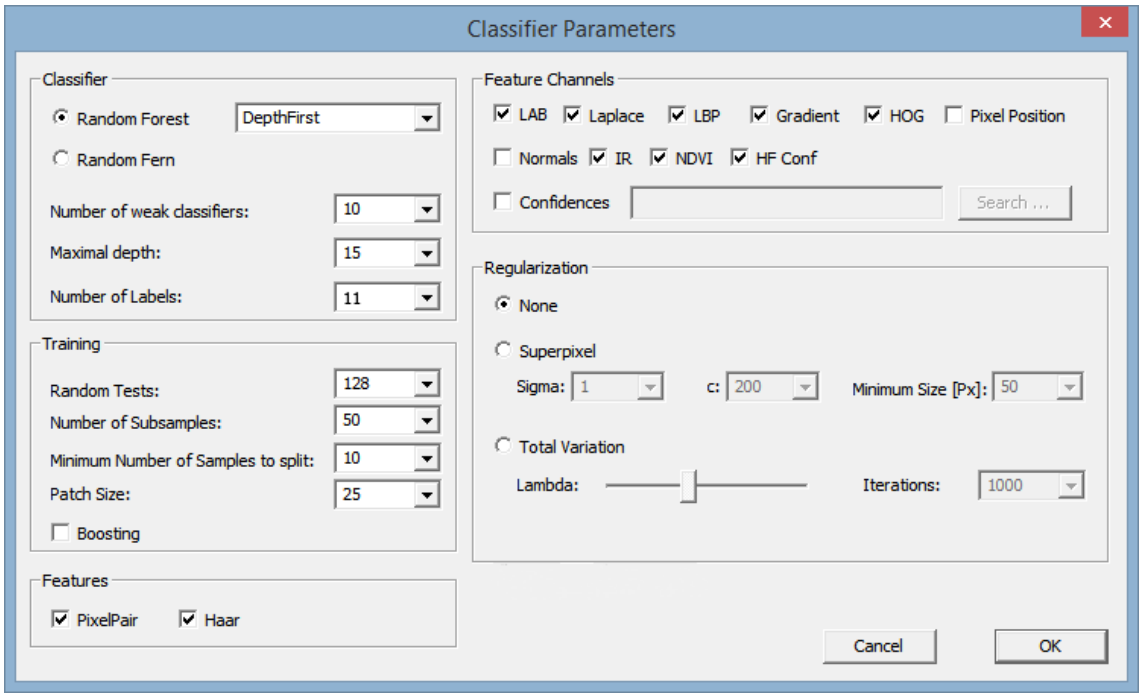



Figure 5.6: Classifier dialog: All settings regarding the classifier for semantic segmentation can be adjusted in this dialog. Notice that the parameters are grouped together logically into Classifier, Training, Features, Feature Channels (see Section 3) and Regularization (see Section 5.4). Regularization defines how the subsequent postprocessing is done using the confidence maps of all labels.

If this dialog is opened the first time, all values are initialized with default values. They are chosen, such that they fit to common semantic segmentation problems directly. In addition, it is possible to choose from predefined values in drop-down menus, such that the user has the possibility to try different settings. It is also possible to type values directly into the combo boxes and therefore use values not present in the drop down menu. This option is indented for expert users only. When a classifier is already loaded into the GUI, the classification dialog shows all the settings, with that the imported classifier was trained.

By pressing the OK-button the configured settings are applied and by clicking Classifier→Train Unsupervised in the menu bar the specified classifier is trained using the defined settings.

**Interactive Labelling** The labelling functionality is the core of interactive semantic image segmentation. The user can teach the classifier, what it should learn actually, by

drawing some scribbles directly on the loaded image. The drawing functionality is enabled by holding the CTRL key. While this key is pressed, the size of the cursor can be changed using the mouse wheel and a scribble is drawn on the image by simply pressing the left mouse button. If the annotation mode is activated, i.e., the CTRL key is pressed, a circle is projected onto the image (see Figure 5.7), such that the user sees where the scribble will be drawn by pressing the left mouse button immediately.

(a) Street (b) Façade (c) Roof

Figure 5.7: The cursor projected onto different surfaces. Notice how accurately the cursor is projected onto the geometry.

The last thing missing for the user to actually draw the scribbles with semantic meaning are the labels. The labels are the logical classes into that the input data should be segmented. For that purpose another dialog, called the "Annotation Dialog" is used. In this dialog, the user can choose between different labels and assign meaningful names to them (see Figure 5.8). The colors in the dialog correspond to the colors used for the annotation in the labelling GUI.

Since the concept of continuous learning is used, i.e., the classifier has been updated at many different project with a lot of labels, the designations of the labels are saved in combination with the classifier. This is especially useful if there exist different classifiers for different purposes. Every time an existing classifier is loaded, the designations used in the current classifier are loaded too and therefore the user immediately knows which colors do correspond to which labels and what they stand for. In Figure 5.2 and Figure 1.5 examples for semantic segmentations made with the GUI presented here are shown.

Depending on the used classifier, it might be possible that pixels do not get the correct label assigned, although exactly these pixels were annotated by the user. However, in fact this is very unlikely. To minimize these situations, the user has the possibility to *boost* his annotations. This actually means that the weight of an annotated sample is increased, such that an incorrect label is hardly ever assigned to an annotated pixel (cf. Section 2.7).

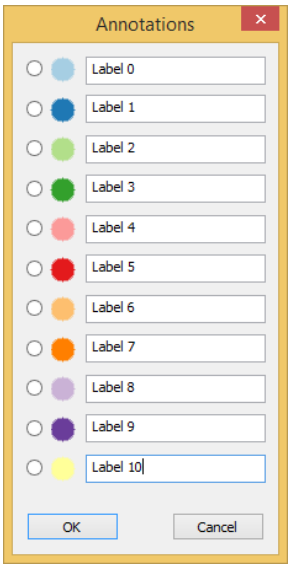



Figure 5.8: Annotation Dialog: this dialog is used to choose a label. It is possible to assign a logical class name to each label in addition.

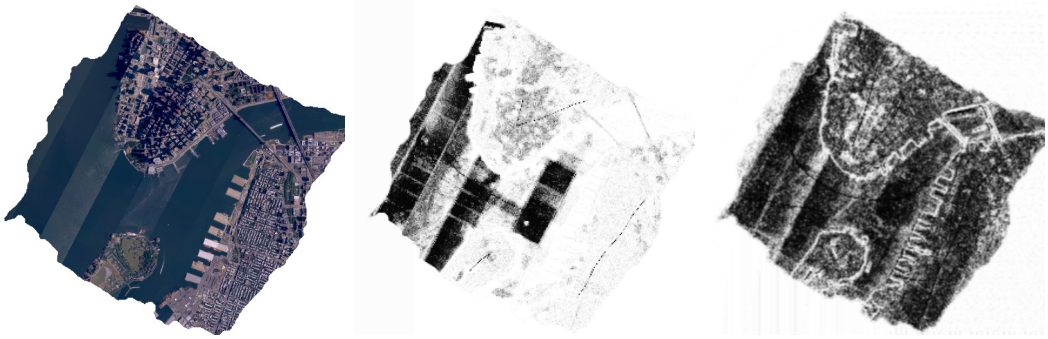
Even if the annotations are boosted, the result is still computed using Equation (2.18), i.e., the prediction is constructed through averaging the posterior distributions of all trees. In real boosting, the label is inferred by a summation over the weak learners as defined in Equation (2.28).

## 5.3 Active User Guidance

Since this is an interactive labelling application, user interaction is necessary to teach the classifier. Obviously, the less user input is affordable, the better. To actually minimize the necessary user interaction, a concept called "Active User Guidance" (AUG) has been developed (see Figure 5.9). AUG shows the confidence of the classifier to the user. This means, by looking to the AUG, the user immediately sees, where the classifier is not that confident and needs some additional help from the human in the loop. With this technique, it is possible to minimize the afford to achieve a good semantic segmentation on given imagery. Actually, the whiter an entry in the AUG, the more equally is the posterior distribution over the class labels distributed. Hence, by providing samples at exactly these positions, the best impact can be achieved. Figure 5.2 shows, how the AUG is rendered in the GUI.

Consider the following example: Without enabling AUG, the user provides samples by drawing some scribbles on the input images. However, it might be the case that the

user always draws scribbles at positions, where the classifier is already very confident. In this case, the statistics in the corresponding leafs will become even clearer, but there is no global information gain. Hence, the quality of the semantic segmentation will not increase, when the classifier is trained with "false" samples. By using AUG, exactly this is avoided, since the user always knows, what samples will have the greatest impact considering the global performance of the classifier.



(a) Input Image (b) Low confident classifier (c) High confident classifier

Figure 5.9: Visualization of the Active User Guidance (AUG). The whiter an entry in the AUG, the more insecure is the classifier. This means that the posterior distribution over the class labels used to classify the pixel is almost an equal distribution. Hence, by guiding the classifier in exactly these regions with new annotated samples, the classifier will become stronger with less annotation effort. In 5.9b the classifier is very insecure especially on LAND pixels. In 5.9c the classifier knows exactly which pixel correspond to which label.

## 5.4 Regularization

As already proposed in Section 2.2, this application uses a two-stage approach to compute the final semantic segmentation. When a sample is propagated through a classifier, the output is a class label $y \in \mathcal{Y}$. However, by directly using a class label as output of the classifier a hard decision is made. This causes noisy semantic segmentations with a lot of small isolated classes (see Figure 6.1). Despite this result is most of the times visually not pleasing, it is actually the most likely labelling. Nevertheless, in general one does want to get smooth results with no or hardly any small and isolated regions. Regularization is the concept of producing a smooth result based on the posterior distributions inferred for each pixel. This means that the input for the regularization is a $|\mathcal{Y}|$-dimensional probability vector $\mathbf{p}$, where the entry $p_i$ is the probability that the sample corresponds to the $i$-th

class. In other words, the confidence maps (see Section 3.1.7) for the classes are the input for the regularization.

In the following paragraphs, the available regularization methods are described in detail. Notice that there exist a lot of different possibilities how to actually regularize the output of a classifier. Some of them are Superpixels, Markov Random Fields (MRF), Conditional Random Fields (CRF), Potts model, etc. In this application two methods, Superpixel regularization and a regularization with the Potts model, which are described next, are used.

### 5.4.1 Superpixels

Before a superpixel segmentation can be used for regularization the terms must be defined in advance. A superpixel can be defined as a set of pixels, where all the pixels contained in this set are homogeneous in some respect. This homogeneity could be locality and coherency for example. In other words superpixels are nothing but small regions, which have a similar appearance. When a complete image is segmented into superpixels, the result is called *superpixel segmentation*. An example for such a superpixel segmentation can be seen in Figure 5.10. Notice, how the pixels are grouped into perceptual meaningful atomic regions. The superpixel algorithm proposed by Felzenszwalb and Huttenlocher [19] was chosen in this application. This has several reasons:

- **Boundary recall**
  The superpixel algorithm of Felzenszwalb and Huttenlocher has the best boundary recall rate when compared with other superpixel algorithms [2].
- **Superpixel size**
  The size of the generated superpixels is adjusted automatically based on the image. This is especially important, because it is more likely that also fine structures are captured correctly as superpixels.
- **Speed**
  The computation time of the superpixel algorithm of Felzenszwalb and Huttenlocher is $\mathcal{O}(n \log n)$.

**Regularization with Superpixels** In this section it is described how these superpixels can be used to do a regularization on the output of the classifier. When using superpixels

Figure 5.10: Visualization of superpixels generated by the algorithm of Felzenszwalb and Huttenlocher [19] with two different zoom-levels. Image courtesy of [1].

for the regularization, it is assumed that one superpixel does contain only one logical class label. This means instead of predicting the label for each pixel individually as it is done by the classifier, now the label for one superpixel, i.e., one region, is predicted. To get a prediction for a complete segment, the posterior distribution of the complete segment is constructed out of the individual posterior distributions of the individual pixels. The posterior probability of a superpixel can be calculated using Equation (5.1).

$$\mathbb{P}(Y = y_k|\mathbf{x}_1, \ldots, \mathbf{x}_N) = \frac{1}{N} \sum_{i=1}^{N} \mathbb{P}_i(Y = y_k|\mathbf{x}_i) \tag{5.1}$$

where $\mathbb{P}$ is the posterior distribution over the class labels of the complete superpixel, $\mathbb{P}_i$ is the posterior distribution over the class labels of the $i$-th pixel in the superpixel, $\mathbf{x}_i$ is the data point corresponding to the $i$-th pixel and $N$ is the number of pixels in the superpixel.

To get the actual class prediction for the complete segment, the class label with the highest probability is assigned to all pixels in the superpixel.

$$C(\mathbf{x}_1, \ldots, \mathbf{x}_N) = \arg\max_k \mathbb{P}(Y = y_k|\mathbf{x}_1, \ldots, \mathbf{x}_N) \tag{5.2}$$

By using superpixels for regularization, the results are improved both, visually and also quantitatively. However, if the superpixel segmentation does not recover the original object boundaries, i.e., one superpixel contains actually more than one label, than an error is introduced. At this stage, there is no possibility to overcome this problem anymore. However, the superpixel segmentation is most of the times accurate enough and yields to very smooth and pleasing results.

### 5.4.2 Potts Model

Another option to regularize the classifier output is to use the method proposed by Unger [73]. This variational regularization approach does not deal with probabilities directly, but with potentials. However, the probabilities $p_i$ can be converted into potentials $f_i$ easily by using Equation (2.1). Since the labelling problem is defined as an energy minimization problem, potentials can be seen as costs, where a high potential corresponds to a high cost and low potentials correspond to low costs respectively.

The general minimal partitioning problem for $K$-class labelling problem in the continuous setting is given by

$$\begin{aligned} &\min_{\Omega_i} \left\{ \frac{1}{2} \sum_{i=1}^{K} Per(\Omega_i) + \sum_{i=1}^{K} \int_{\Omega_i} f_i(\mathbf{x}) d\mathbf{x} \right\}, \\ &s.t.\ \Omega = \bigcup_{i=1}^{K} \Omega_i,\ \Omega_i \cap \Omega_j = \emptyset \quad \forall i \neq j \end{aligned} \tag{5.3}$$

where $Per(\Omega_i)$ is the perimeter of the set $\Omega_i$ and $f_i$ are the potentials. In the discrete setting Equation (5.3) is referred to as the Potts model [52], which is a generalization of the Ising model [32] for multiple labels. In these models pairwise costs are defined, i.e., if there is a label switch in the neighbourhood of a pixel, a cost $g(i, j)$ is assigned:

$$\psi_{ij}(y_i, y_j) = \begin{cases} 0 & \text{if } y_i = y_j \\ g(i,j) & \text{otherwise} \end{cases} \tag{5.4}$$

To actually solve the labelling problem stated in Equation (5.3), the relaxation proposed by [78] augmented with a binary term to the total variation (TV) can be used (see Equation (5.5)). After transforming the relaxed version into a primal-dual problem, it can be solved efficiently. For more information about variational multi-label image segmentation the reader is referred to the work of Unger [73].

$$\begin{aligned} &\min_{u_i} \left\{ \sum_{i=1}^{K} \int_{\Omega} c_b \underbrace{|\nabla u_i|}_{TV} + \sum_{i=1}^{K} \int_{\Omega} u_i f_i d\mathbf{x} \right\}, \\ &s.t.\ \sum_{i=1}^{K} u_i = 1,\ u_i \geq 0 \quad \forall i = 1, \ldots, K \end{aligned} \tag{5.5}$$

## 5.5 Summary

In this chapter, the application resulting of this thesis is described in detail. An available viewer, which is capable to visualize images in 2D and complete project areas in 3D is extended with the semantic segmentation functionality. To facilitate the annotation task, appropriate dialogs have been designed. In order to do a semantic segmentation on such aerial projects the first step is to load or train a new classifier. Then the user can teach the classifier until the semantic segmentation is satisfactory. The segmentation itself is generated using a two stage approach. In the first stage, the unaries are generated with a classifier and in the second stage, called regularization, the probabilities generated by the classifier are used to infer a smooth labelling for each image. Two types of regularization, superpixel regularization and a regularization based on the Potts model, are described in detail.

# Chapter 6

# Evaluation

In this chapter, the algorithmic part of the application is evaluated. The performance in terms of accuracy of random forests and random ferns are compared. A supervised version of a random forest is used as a base line, because this setting is most common in the literature [65, 66]. However, the accuracy of the final segmentation is not the only goal in this application. Due to the fact that a user should interact with the algorithms, an interactive performance is desired. Therefore, the parameters (forest size, maximal tree depth, etc.) is chosen, such that the behaviour of the application remains interactive.

The performance of the different algorithms is evaluated on the eTRIMS [35] database and on the MSRC database [57]. When the eTRIMS database is used, 40 images are used for training and 20 images are used for testing. This is exactly the same setting as in [23]. When the MSRC database is used, the split is chosen exactly as it was done by Shotton *et al.* in [57]. Since the training procedure incorporates some randomness, each test is done five times and the result is averaged.

In order to evaluate the online capability of the classifiers, it is necessary to simulate the interaction done by a human operator. The user will always annotate pixels which are incorrect classified to teach the classifier. Hence, in the evaluation an intelligent procedure for producing new samples online must be used. To tackle this problem, the active user guidance (AUG) (see Section 5.3) can be used again. In this case the AUG does not help the user, but the computer to create online samples intelligently.

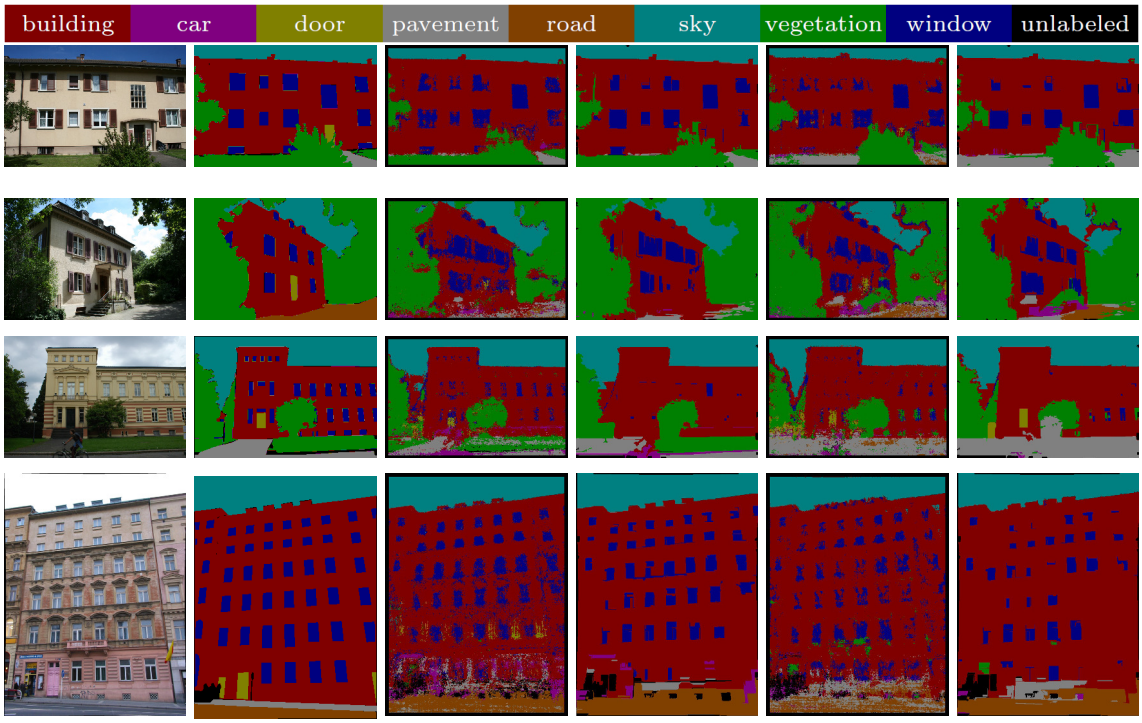
To quantify the performance of the different settings and different classifiers, the global and average recognition rate are measured. The global recognition rate reveals the fraction of pixels labelled correctly compared to all pixels:

$$\text{Global Recognition Rate} = \frac{\#\text{correct labelled pixels}}{\#\text{all pixels}} \tag{6.1}$$

However, this quantity prefers bigger categories over small ones, because by labelling just the big categories like SKY for example correctly, a good performance can be achieved even if small classes are labelled completely incorrect. Therefore, the second quantity measured are the average correct pixels. The average recognition rate is defined in Equation (6.2). This quantity also incorporates how well small categories are labelled:

$$\text{Average Recognition Rate} = \frac{\sum_{y \in \mathcal{Y}} \frac{\#\text{Correct pixels of class } y}{\#\text{all samples of class } y}}{\#\text{classes}} \tag{6.2}$$

Figure 6.1 shows some visual results obtained with the implementation of this thesis. In the following sections, the results of the evaluations are presented for random forests and random ferns. The detailed test settings for each evaluation are described in the specific sections.




(a) Image (b) Ground truth (c) Forest (d) Forest SP (e) Fern (f) Fern SP

Figure 6.1: Visual results from the eTRIMS database. In (a) the input image is shown, (b) shows the corresponding ground truth annotation, (c) and (d) show the unary result and the superpixel-regularized result achieved with the forest of test 6.2. (e) and (f) show the unary result and the superpixel-regularized result achieved with the fern of test 6.10b.

## 6.1 Random Forest

A random forest is the main classifier used in this application. Here, it is evaluated how the impact of different forest sizes $T$ is and what maximum depth $D$ should be used in order to achieve the best generalization on previously unseen data. Table 6.1 shows common parameters used throughout the evaluation.

| **Property** | **Value** |
|---|---|
| Maximum tree depth $D$ | $5, \ldots, 20$ |
| Number of trees | $2, 4, 6, 8, 10$ |
| Feature Channels | LAB, Gradients, HOG, LBP, Location, [Confidence] |
| Features | Pixelpair, Generalized Haar |
| Random tests | 256 |
| Threshold selection | Median |
| Regularization | Superpixel |

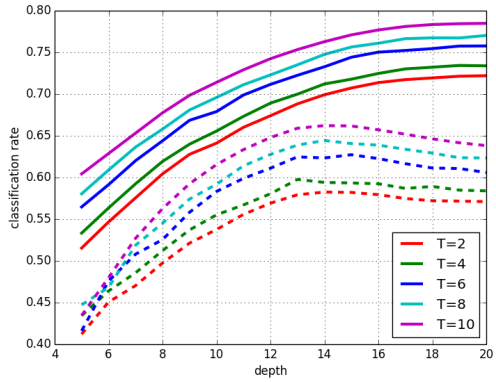
Table 6.1: Random forest parameters used for the evaluation

### 6.1.1 Offline Performance

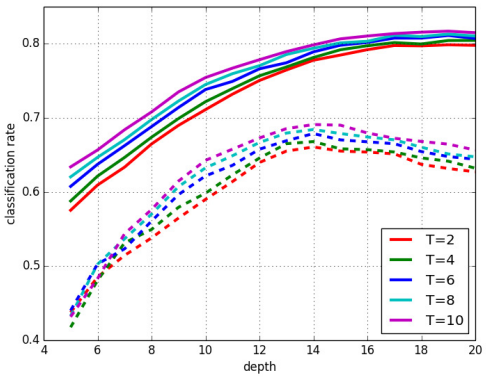
In the offline setting, the classifier is trained with a training set and then its performance is measured on a test set. Here, different settings have been investigated. As a base line for all evaluations, a supervised forest has been trained on the eTRIMS database for easy comparison with the literature. All other tests will be compared with this test scenario. The plots show the global and average correct labelled pixels respectively for different parameters.

#### 6.1.1.1 Supervised, Depth First

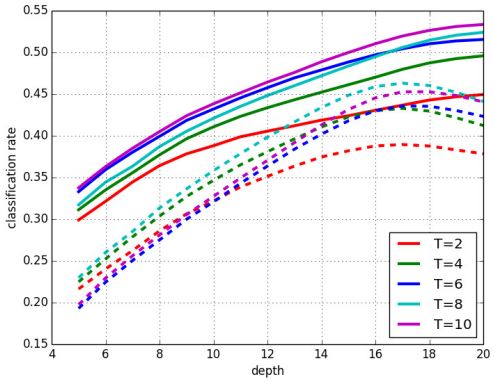
In this test run, each forest is trained completely supervised. The plot (Figure 6.2) shows the results for different numbers of trees in the forest and different depths.



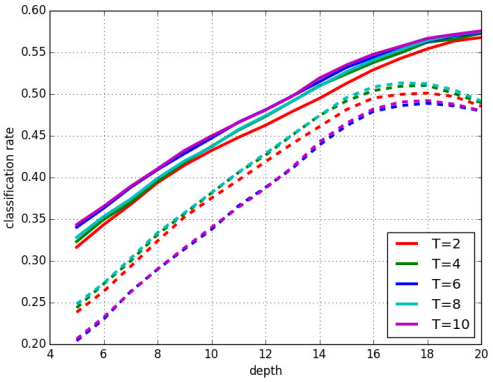

(a) eTRIMS: Unary




(b) eTRIMS: Superpixel




(c) MSRC: Unary

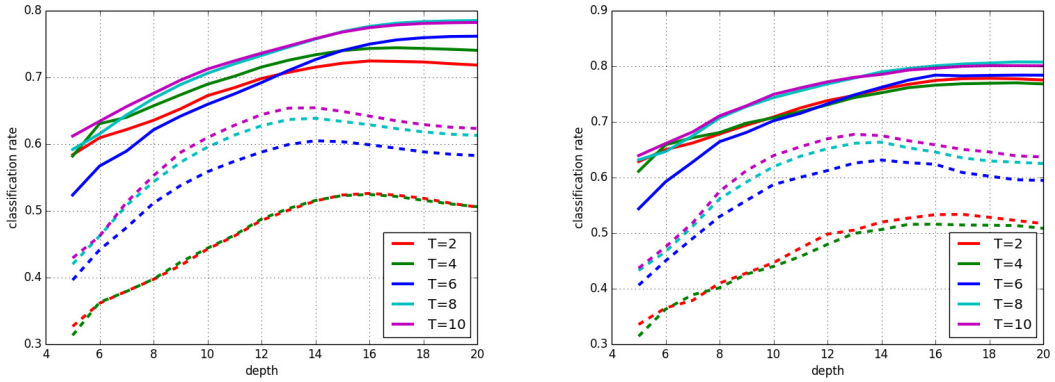



(d) MSRC: Superpixel

Figure 6.2: Comparison of the effect of different forest parameters (forest size $T$, maximum depth $D$) on the eTRIMS and MSRC database. Solid lines indicate the global recognition rate and dashed lines indicate the average recognition rate.

#### 6.1.1.2 Supervised, Entangled

In this test run entangled forests are used to incorporate context (see Section 2.8). In this scenario, the confidence output for all classes $y \in \mathcal{Y}$ of depth $n$ are used as addition feature channels during the training of depth $n + 1$. In Figure 6.3, the results of an entangled decision forest are shown.



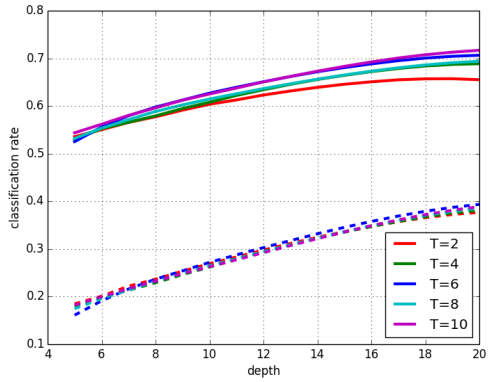

(a) eTRIMS: Unary

(b) eTRIMS: Superpixel

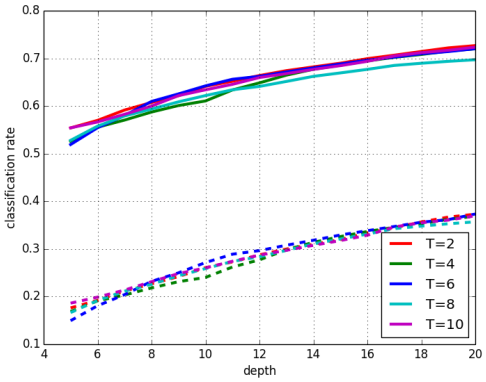
Figure 6.3: Comparison of different parameters when an entangled random forest is used on the eTRIMS database. Solid lines indicate the global recognition rate and dashed lines indicate the average recognition rate.

#### 6.1.1.3 Unsupervised, Depth First, Update

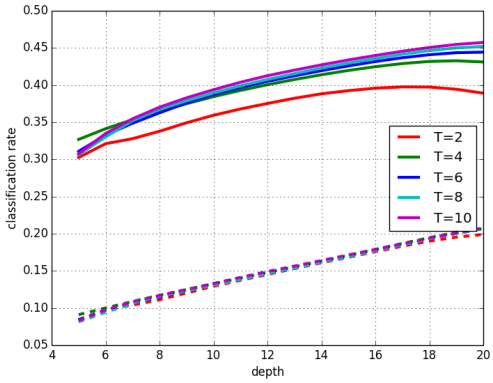
In this experiment, the tree structure is built unsupervised, i.e., no labels are used in the training stage. Therefore, the structure of the trees is more general than in the supervised case. However, different patches are also split in the nodes of the trees. Actually, after the training of the unsupervised forest, no statistics exist in the leaf nodes. The statistics are generated by a semi-online update of the forest with the labels of the training data (see Section 4.2). Note, the structure is trained without considering the labels and the statistics of the unsupervised trained forest is constructed using the labels of the training set. As can be seen in Figure 6.4, especially the average recognition rate is worse compared to the supervised trained forest (cf. Figure 6.2).



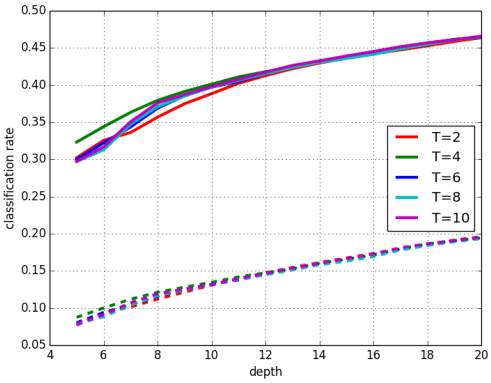

(a) eTRIMS: Unary




(b) eTRIMS: Superpixel




(c) MSRC: Unary



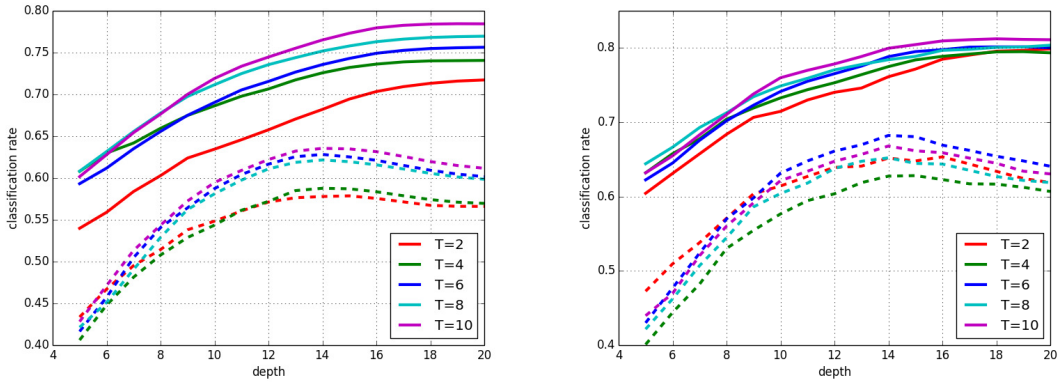

(d) MSRC: Superpixel

Figure 6.4: Comparison of the performance of an unsupervised trained forest where the statistics are generated through a semi-online forest update with the labelled training data on the eTRIMS and MSRC database. Solid lines indicate the global recognition rate and dashed lines indicate the average recognition rate.

#### 6.1.1.4 Supervised, Depth First, Additional Confidence Maps

The next test run tries to incorporate the knowledge of a classifier trained on a source database into the training phase of a new classifier for a different target database (see Section 4.4). Therefore, in this test case, the first classifier providing the confidence maps was trained on the LabelMe database which contains the same classes as the eTRIMS database. With this classifier, confidence maps for images in the eTRIMS database are created and used as additional feature channels during the training of a new classifier for the eTRIMS database. This is similar to the concept of auto-context (cf. Section 2.8) ex-

cept that the classifier which provides the confidence maps is trained on another database. The performance of the new trained classifier is shown in Figure 6.5.



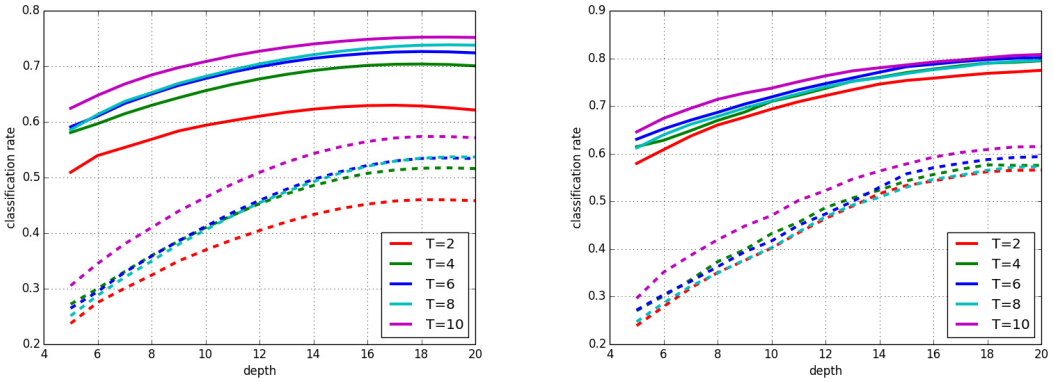

(a) eTRIMS: Unary

(b) eTRIMS: Superpixel

Figure 6.5: Knowledge transfer from the LabelMe database to the eTRIMS database using a supervised trained forest. Solid lines indicate the global recognition rate and dashed lines indicate the average recognition rate.

#### 6.1.1.5 Unsupervised, Depth First, Additional Confidence Maps

This test run is equal to the previous one, except of that the new forest is trained unsupervised instead of supervised. In comparison to Figure 6.4, especially the average recognition rate can be improved by using additional confidence maps even if the classifier which provides these confidence maps is trained on a different dataset (see Figure 6.6). Therefore, it is possible to use learnt information from a previously trained classifier to construct a more powerful new classifier.




(a) eTRIMS: Unary

(b) eTRIMS: Superpixel

Figure 6.6: Knowledge transfer from the LabelMe database to the eTRIMS database using an unsupervised trained forest. Solid lines indicate the global recognition rate and dashed lines indicate the average recognition rate.

#### 6.1.1.6 Summary

Here, the best parameters from each evaluation are gathered and summarized. Table 6.2 shows the maximum reached *global* recognition rate and with which parameters the results were achieved on the eTRIMS database. Table 6.3 shows the maximum reached *average* recognition rate and with which parameters the results were achieved on the eTRIMS database. Table 6.4 and Table 6.5 show the maximum results of the global and average recognition results on the MSRC database.

| Test Run | Depth | #Trees | Global | Average |
|---|---|---|---|---|
| 6.2a | 20 | 10 | 0.7848 | 0.6382 |
| 6.2b | 19 | 10 | **0.8170** | **0.6646** |
| 6.3a | 20 | 8 | 0.7852 | 0.6131 |
| 6.3b | 19 | 8 | 0.8078 | 0.6274 |
| 6.4a | 20 | 10 | 0.7169 | 0.3890 |
| 6.4b | 20 | 2 | 0.7267 | 0.3730 |
| 6.5a | 19 | 10 | 0.7845 | 0.6149 |
| 6.5b | 18 | 10 | 0.8123 | 0.6448 |
| 6.6a | 19 | 10 | 0.7525 | 0.5735 |
| 6.6b | 20 | 10 | 0.8080 | 0.6150 |

Table 6.2: Performance of different supervised trained random forests on the eTRIMS database. The table shows the best *global* recognition rate and the corresponding average recognition rate.

| Test Run | Depth | #Trees | Global | Average |
|---|---|---|---|---|
| 6.2a | 14 | 10 | 0.7628 | 0.6622 |
| 6.2b | 14 | 10 | **0.7985** | **0.6911** |
| 6.3a | 14 | 10 | 0.7582 | 0.6546 |
| 6.3b | 13 | 10 | 0.7802 | 0.6776 |
| 6.4a | 20 | 6 | 0.7065 | 0.3936 |
| 6.4b | 20 | 6 | 0.7203 | 0.3734 |
| 6.5a | 14 | 10 | 0.7651 | 0.6355 |
| 6.5b | 14 | 6 | 0.7883 | 0.6822 |
| 6.6a | 18 | 10 | 0.7524 | 0.5738 |
| 6.6b | 20 | 10 | 0.8080 | 0.6150 |

Table 6.3: Performance of different supervised trained random forests on the eTRIMS database. The table shows the best *average* recognition rate and the corresponding global recognition rate.

| Test Run | Depth | #Trees | Global | Average |
|---|---|---|---|---|
| 6.2c | 20 | 10 | 0.5335 | 0.4406 |
| 6.2d | 20 | 10 | **0.5758** | **0.4801** |
| 6.4c | 20 | 10 | 0.4574 | 0.2080 |
| 6.4d | 20 | 10 | 0.4659 | 0.1952 |

Table 6.4: Performance of different supervised trained random forests on the MSRC database. The table shows the best *global* recognition rate and the corresponding average recognition rate.

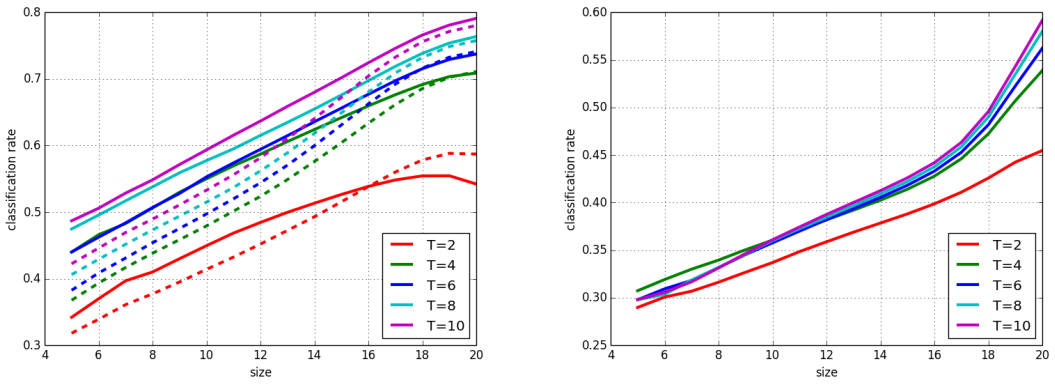
| Test Run | Depth | #Trees | Global | Average |
|---|---|---|---|---|
| 6.2c | 17 | 8 | 0.5059 | 0.4631 |
| 6.2d | 17 | 8 | **0.5531** | **0.5134** |
| 6.4c | 20 | 10 | 0.4574 | 0.2080 |
| 6.4d | 20 | 2 | 0.4638 | 0.1960 |

Table 6.5: Performance of different supervised trained random forests on the MSRC database. The table shows the best *average* recognition rate and the corresponding average recognition rate.

### 6.1.2 Online Performance

Since in this application, all classifiers are used in an online setting, the online performance is evaluated in addition. Therefore, the samples to update the classifier are chosen based on the confidence of the classifier. This means that the least confident samples are used to update the forest.

In this test run, each forest is trained unsupervised on the training data (see Section 4.2.2). The advantage of this approach is that a semantic segmentation can be computed without the necessity of a labelled set of training data. To evaluate the online performance of such a forest, the test images are used and each image is updated exactly ten times. Initially, 500 random samples from the first image are used to update the classifier. Then the classifier is updated with the 500 least confident samples of each class. This is equal to 1.14% of all pixels in each image in the eTRIMS database (average image size = $512 \times 768$ pixels). However, as can be seen in Figure 6.7, the performance is very good, even if there are very few samples provided. It can also be seen that 2 trees are not enough to capture the variability of the data, because the red curve decreases at depth 18. This means that the forest starts to overfit to the training data. The forests with $T > 2$ perform very well and do not overfit. However, it is not feasible to increase the depth of the forest further, due to the exponential growth of the number of nodes. The best performing forests in the online setting are summarized in Table 6.6.




(a) Unary: eTRIMS

(b) Unary: MSRC

Figure 6.7: Online performance of a semi-online random forest on the eTRIMS and the MSRC database. Note that the performance is almost equal as in the completely supervised setting in offline mode, where all samples are used to generate the statistics. In difference to the supervised setting, the average recognition rate is better in the online setting.

Another interesting property of an online classifier is how fast it adapts to new data. To investigate this behaviour, the best performing forest ($T = 10$) has been used to plot the adaption of a semi-online forest to new unseen data (Figure 6.8).

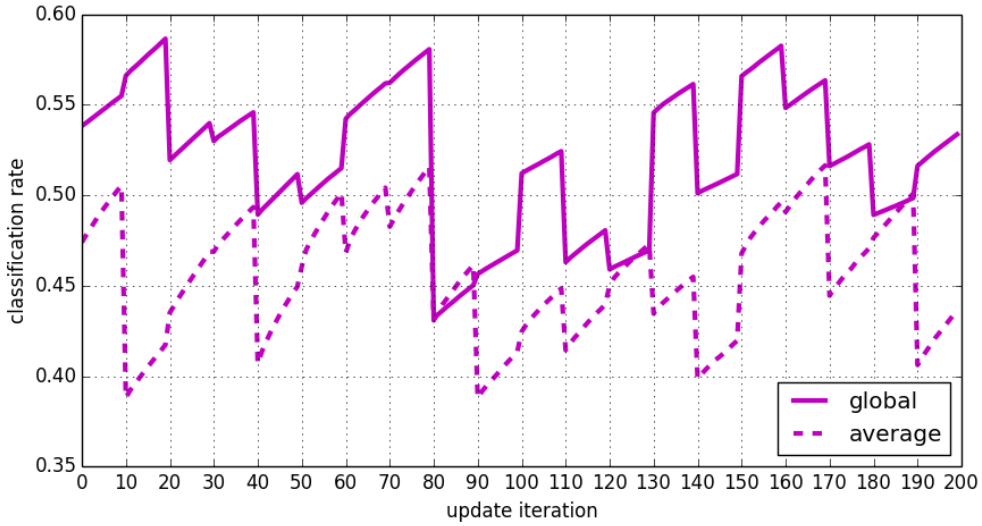



Figure 6.8: Adaption of a semi-online random forest. Each image is updated exactly 10 times. After each local minimum, the performance increases very fast. This indicates that it is possible to learn new information very fast.

| Test Run | Depth | #Trees | Global | Average |
|---|---|---|---|---|
| 6.7a | 20 | 10 | 0.7908 | 0.7801 |
| 6.7b | 20 | 10 | 0.5915 | - |

Table 6.6: Results of the unsupervised pre-trained semi-online random forest on the eTRIMS and MSRC database. The performance is measured by averaging the performance after the online update of each image.

## 6.2 Random Ferns

To compare the performance of the main classifier (random forest), random ferns have been chosen, because they are inherently online and therefore very well suited for an interactive approach. Common parameters used during the evaluation are shown in Table 6.7. These parameters show that the number of ferns is greater than the number of trees in a forest to achieve comparable performance. The weak learners of the fern are chosen according to optimize the information gain and additionally, the training procedure tries to split classes, which could not be split up to now. This procedure is similar to the procedure used in a forest. But in difference to a forest, all tests in a fern are applied to the input data, independent of the outcome of the previous test.

| Property | Value |
|---|---|
| Maximum fern size $S$ | $10, \ldots, 20$ |
| Number of ferns | $10, \ldots, 30$ |
| Feature Channels | LAB, Gradients, HOG, LBP, Location, [Confidence] |
| Features | Pixelpair, Generalized Haar |
| Random tests | 256 |
| Threshold selection | Median |
| Regularization | Superpixel |

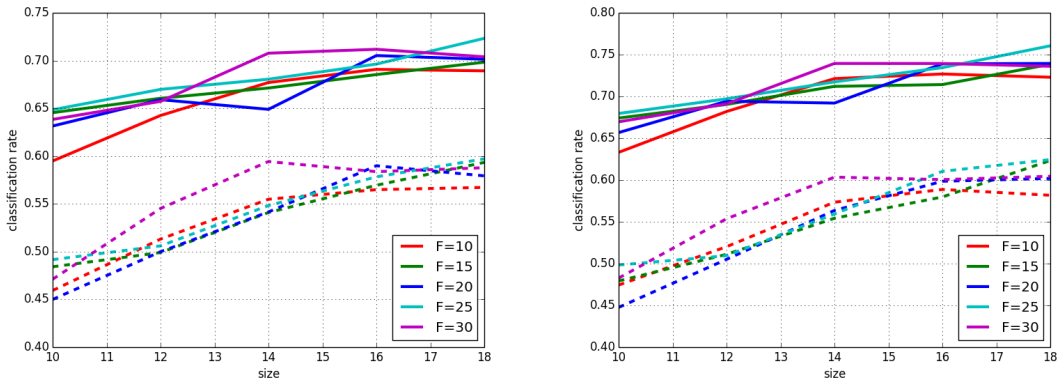
Table 6.7: Random fern parameters used for the evaluation

### 6.2.1 Offline Performance

As it was done with the random forest, the random ferns are tested in an offline setting too. The same tests are done with the ferns to directly compare their performance with the performance of a random forest.

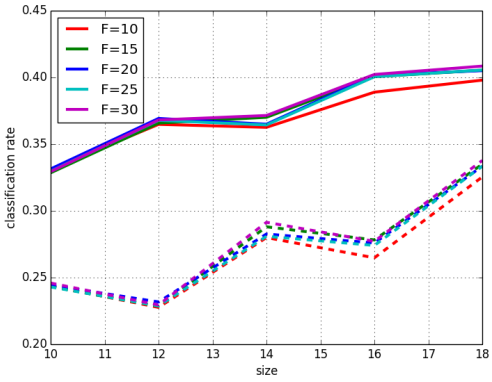
#### 6.2.1.1 Supervised

In this test run, the random fern is trained in a supervised manner. Figure 6.9 shows how random ferns perform in this setting. See Figure 6.2 for the result of the random forest in this experiment.

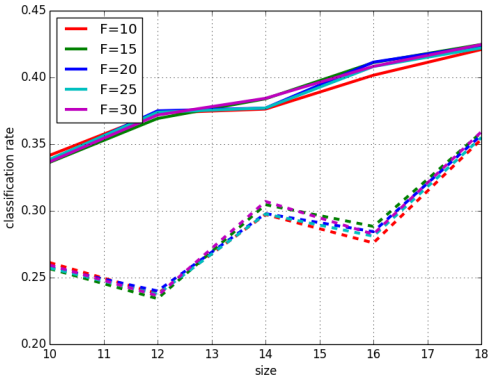



(a) eTRIMS: Unary

(b) eTRIMS: Superpixel




(c) MSRC: Unary

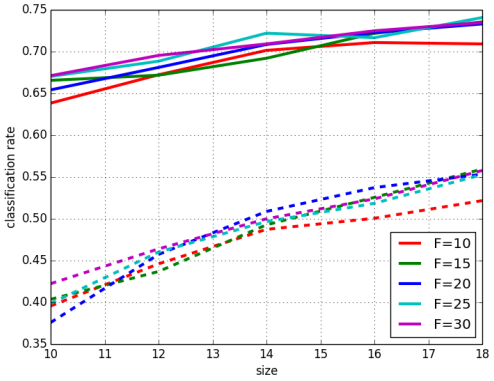



(d) MSRC: Superpixel

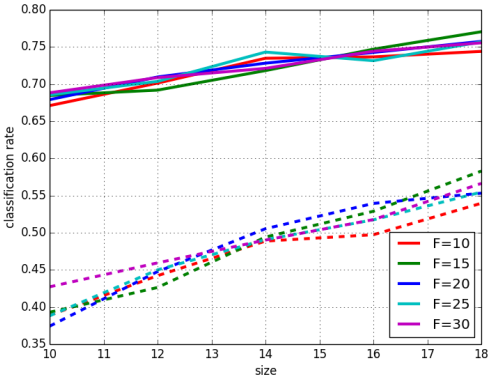
Figure 6.9: Comparison of the effect of different fern parameters (number of ferns $N$, fern size $S$) on the eTRIMS and MSRC database. Solid lines indicate the global recognition rate and dashed lines indicate the average recognition rate.

#### 6.2.1.2 Unsupervised, Update

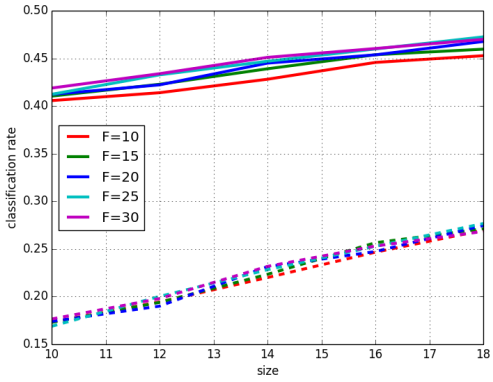
In this test case, it is evaluated how a random fern performs, when it is trained unsupervised, i.e., no labels are used during the training stage. The statistics are generated by updating the ferns with the labels available in the training data. The performance of an unsupervised trained fern is shown in Figure 6.10. Figure 6.4 shows the result of a random forest in this experiment.



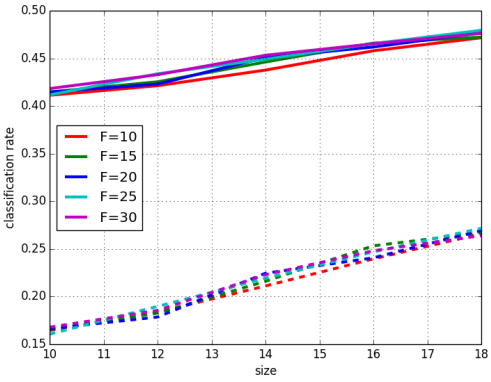

(a) eTRIMS: Unary




(b) eTRIMS: Superpixel




(c) MSRC: Unary

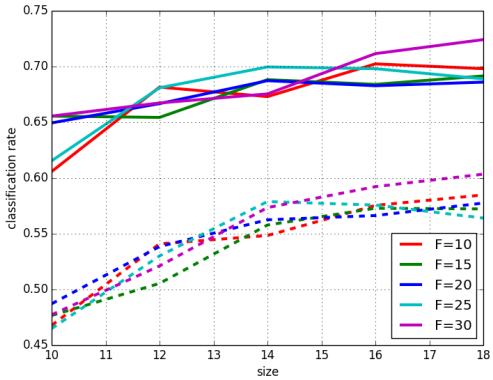



(d) MSRC: Superpixel

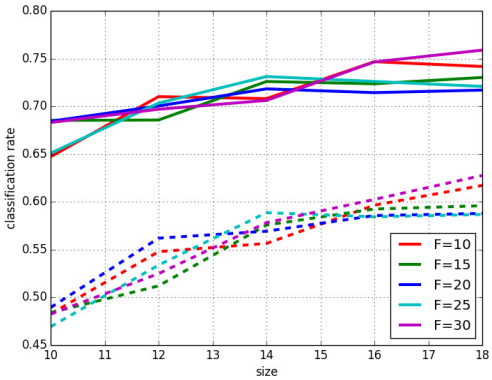
Figure 6.10: Comparison of the performance of an unsupervised trained ferns where the statistics are generated through an online update with the labelled training data on the eTRIMS and MSRC database. Solid lines indicate the global recognition rate and dashed lines indicate the average recognition rate.

#### 6.2.1.3 Supervised, Additional Confidence Maps

In this test run, the knowledge of another classifier is incorporated in the learning procedure of the fern, which should be trained. Therefore, a random forest has been trained on the LabelMe database and the confidence maps of this classifier are used as additional feature channels during the supervised fern training. Figure 6.11 shows the results of this test case. See 6.5 for the performance of a random forest in this experiment.




(a) Unary: Global Correct

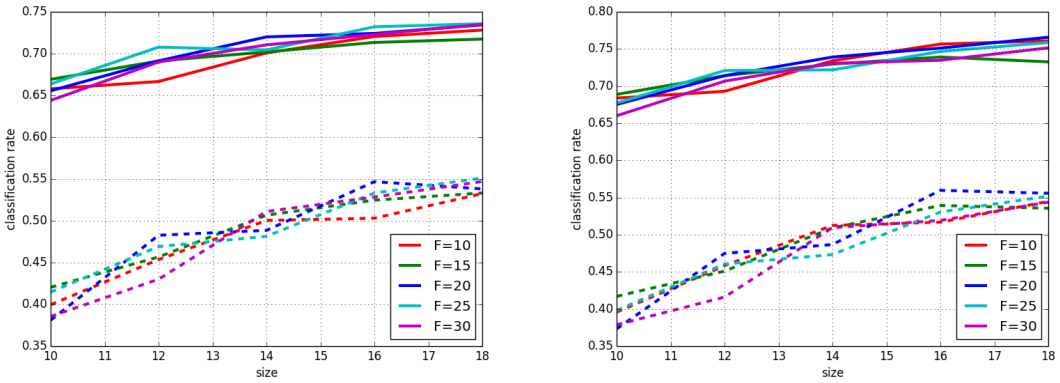



(b) Superpixel: Global Correct

Figure 6.11: Knowledge transfer from the LabelMe database to the eTRIMS database using a supervised trained fern. Solid lines indicate the global recognition rate and dashed lines indicate the average recognition rate.

#### 6.2.1.4 Unsupervised, Additional Confidence Maps

This test run is quite similar to the test run above, except that the random fern is trained unsupervised. The results are shown in Figure 6.12. Figure 6.6 shows the results of a random forest in this experiment.




(a) Unary: Global Correct

(b) Superpixel: Global Correct

Figure 6.12: Knowledge transfer from the LabelMe database to the eTRIMS database using an unsupervised trained fern. Solid lines indicate the global recognition rate and dashed lines indicate the average recognition rate.

#### 6.2.1.5 Summary

Here an overview of the maximum performances achieved with random ferns in terms of global and average recognition rate respectively are summarized. Table 6.8 and Table 6.9 show the results on the eTRIMS database and Table 6.10 and Table 6.11 show the results on the MSRC database.

| Test Run | Size | #Ferns | Global | Average |
|---|---|---|---|---|
| 6.9a | 18 | 25 | 0.7233 | 0.5972 |
| 6.9b | 18 | 25 | 0.7605 | 0.6244 |
| 6.10a | 18 | 25 | 0.7407 | 0.5529 |
| 6.10b | 18 | 15 | **0.7705** | 0.5830 |
| 6.11a | 18 | 30 | 0.7242 | 0.6034 |
| 6.11b | 18 | 30 | 0.7589 | **0.6277** |
| 6.12a | 18 | 25 | 0.7358 | 0.5511 |
| 6.12b | 18 | 20 | 0.7658 | 0.5558 |

Table 6.8: Performance of different supervised trained random ferns on the eTRIMS database. The table shows the best *global* recognition rate and the corresponding average recognition rate.

| Test Run | Size | #Ferns | Global | Average |
|---|---|---|---|---|
| 6.9a | 18 | 25 | 0.7233 | 0.5972 |
| 6.9b | 18 | 25 | 0.7605 | 0.6244 |
| 6.10a | 18 | 15 | 0.7355 | 0.5595 |
| 6.10b | 18 | 15 | **0.7705** | 0.5830 |
| 6.11a | 18 | 30 | 0.7242 | 0.6034 |
| 6.11b | 18 | 30 | 0.7589 | **0.6277** |
| 6.12a | 18 | 25 | 0.7358 | 0.5511 |
| 6.12b | 16 | 20 | 0.7512 | 0.5598 |

Table 6.9: Performance of different supervised trained random ferns on the eTRIMS database. The table shows the best *average* recognition rate and the corresponding average recognition rate.

| Test Run | Size | #Ferns | Global | Average |
|---|---|---|---|---|
| 6.9c | 18 | 30 | 0.4085 | 0.3378 |
| 6.9d | 18 | 15 | 0.4247 | **0.3588** |
| 6.10c | 18 | 25 | 0.4725 | 0.2767 |
| 6.10d | 18 | 25 | **0.4795** | 0.2717 |

Table 6.10: Performance of different supervised trained random ferns on the MSRC database. The table shows the best *global* recognition rate and the corresponding average recognition rate.

| Test Run | Size | #Ferns | Global | Average |
|---|---|---|---|---|
| 6.9c | 18 | 30 | 0.4085 | 0.3378 |
| 6.9d | 18 | 30 | 0.4246 | **0.3593** |
| 6.10c | 18 | 25 | 0.4725 | 0.2767 |
| 6.10d | 18 | 25 | **0.4795** | 0.2717 |

Table 6.11: Performance of different supervised trained random ferns on the MSRC database. The table shows the best *average* recognition rate and the corresponding average recognition rate.

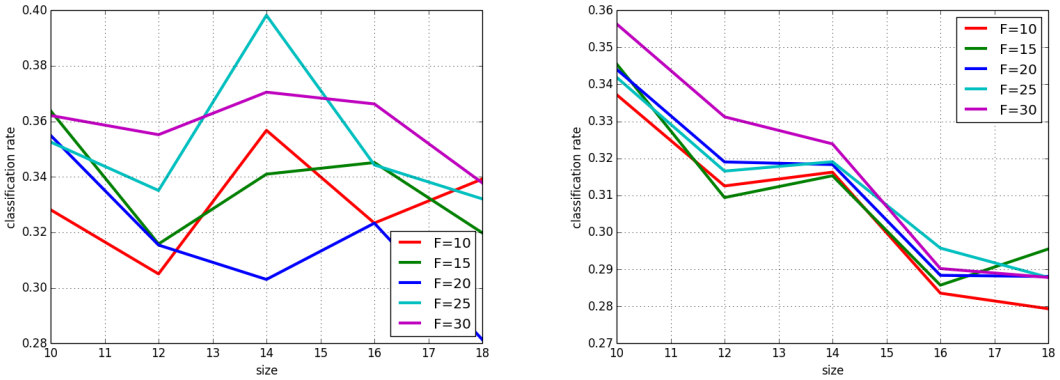
### 6.2.2 Online Performance

The random ferns are evaluated in an online setting too. Therefore, all ferns are trained unsupervised and the update is done based on the confidence of the current classifier, i.e., the update is done exactly as it was done with the semi-online random forest (cf. Section 6.1.2).

The adaption of the best random fern ($M = 20$, $S = 14$) is shown in Figure 6.14.

However, the performance of the online fern is worse compared to the semi-online forest. The fern suffers from the the fact, that only little training data is used for the update.

The best performing ferns in the online setting are summarized in Table 6.12.




(a) Unary: eTRIMS (b) Unary: MSRC

Figure 6.13: Online performance of a random fern on the eTRIMS and the MSRC database. The performance is far worse compared to the performance of the semi-online random forest (Figure 6.7). This is due to the generative behaviour of the ferns. They need a lot of training samples to achieve a good performance.

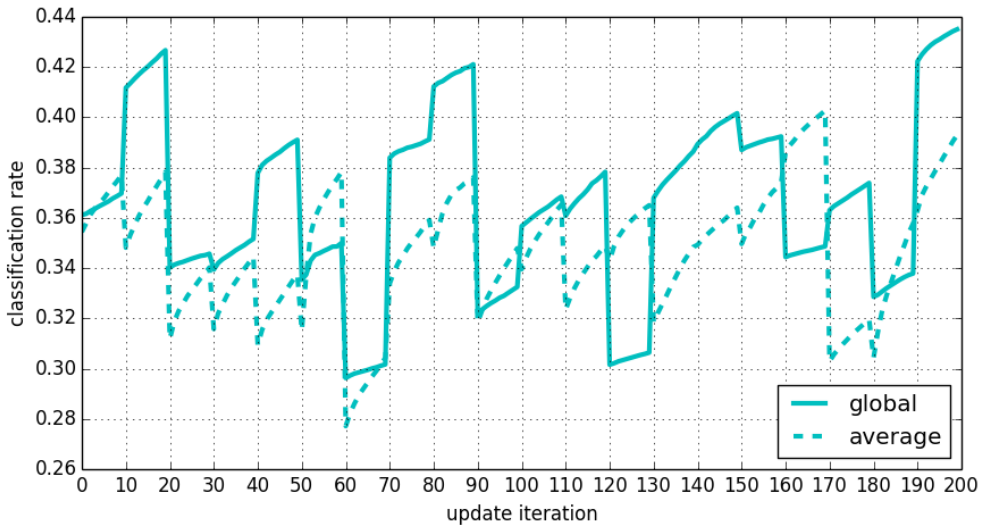



Figure 6.14: Adaption of a random fern demonstrated using the best performing online fern (F = 25, S = 14) on the eTRIMS database. Each image is updated exactly 10 times. Here it can be seen that the fern learns slower than a random forest.

| Test Run | Size | #Ferns | Global | Average |
|---|---|---|---|---|
| 6.13a | 14 | 25 | 0.3982 | 0.3823 |
| 6.13b | 10 | 30 | 0.3563 | - |

Table 6.12: Results of the unsupervised pre-trained semi-online random forest on the eTRIMS and MSRC database. The performance is measured by averaging the performance after the online update of each image.

## 6.3 Summary

In this chapter, the performance of the algorithms used in this thesis are evaluated. The basis is given by a standard offline forest, which was trained in a supervised manner. On this base line the performance of an entangled decision forest, unsupervised trained forests with and without additional confidence maps is evaluated and compared. To reveal the differences in performance, the same tests are also performed with the random ferns. The best performing classifiers are summarized in Table 6.13 (eTRIMS) and Table 6.14 (MSRC). In the offline tests, the forest is better in most of the experiments. The fern is better, when the classifier is trained unsupervised.

| Setting | Classifier | Test Run | Depth/ Size | #Trees/ #Ferns | Global | Average |
|---|---|---|---|---|---|---|
| Supervised | Forest | 6.2b | 20 | 10 | **0.8170** | **0.6646** |
| Supervised, ACM | Forest | 6.5b | 18 | 10 | 0.8123 | 0.6448 |
| Unsupervised | Fern | 6.10b | 18 | 15 | 0.7705 | 0.5830 |
| Unsupervised, ACM | Forest | 6.6b | 20 | 10 | 0.8080 | 0.6150 |

Table 6.13: Parameters and evaluation results of the best performing classifiers on the eTRIMS database. ACM = Additional Confidence Maps.

| Setting | Classifier | Test Run | Depth/ Size | #Trees/ #Ferns | Global | Average |
|---|---|---|---|---|---|---|
| Supervised | Forest | 6.2d | 17 | 8 | **0.5531** | **0.5134** |
| Unsupervised | Fern | 6.10d | 18 | 25 | 0.4795 | 0.2717 |

Table 6.14: Parameters and evaluation results of the best performing classifiers on the MSRC database.

Except from this offline evaluations, the semi-online random forest and the random fern are tested under true online conditions. To generate reliable samples like a human operator would do, the least confident samples are chosen using the active user guidance (AUG). The evaluation has shown that a semi-online random forest with ten trees of depth

20 yields best results in the online setting. The performance of the semi-online random forest is comparable with the fully supervised setting. This is different to the behaviour of the fern. When the fern is used online, the performance is worse than in the offline setting. Due to the generative behaviour of the fern, many more samples are necessary to generate reliable statistics for classification. In Table 6.15 the results of the best performing online classifier are summarized.

| **Database** | **Classifier** | **Test Run** | **Depth** | **#Trees** | **Global** | **Average** |
|---|---|---|---|---|---|---|
| eTRIMS | Forest | 6.7a | 20 | 10 | 0.7908 | 0.7801 |
| MSRC | Forest | 6.7b | 20 | 10 | 0.5915 | - |

Table 6.15: Parameters and evaluation result of the best performing classifiers in an *online* setting on the eTRIMS and MSRC database.

To summarize, with the algorithms presented in Section 4.2.2, a random forest achieves very good results in an online setting. Actually a random forest performs better than the fern which is inherently online.

# Chapter 7

# Conclusion & Further work

## 7.1 Conclusion

In this master thesis, an existing viewing application was extended with the functionality to perform a semantic segmentation. The main target images are aerial images.

In Section 1.1 image segmentation is defined and some examples of image segmentations are visualized. However, no information about what is visible on the image is available. Therefore, the concept of semantic segmentation is introduced in Section 1.3. In semantic segmentation, the task is to group pixels, which are similar in some respect, together and assign a logical class label to every pixel. This means in a semantic segmentation, all foreground and all background objects should be identified pixel accurately. Due to the high complexity of semantic segmentation, machine learning approaches are used to tackle the problem. Usually, a semantic segmentation is derived using supervised machine learning approaches, i.e., a classifier is trained with a huge amount of training data and uses the knowledge to semantically segment previously unseen images. However, as shown in Figure 1.4 a very good semantic segmentation can be achieved using few selected samples for training the classifier. Therefore, an interactive approach has been chosen (see Section 1.4). In this setting, the categories and the appropriate samples are selected by a human operator. New information is incorporated when it is available.

In Section 2 an overview of relevant related work for this thesis is given. A classifier is defined and it is shown, how a classifier can be used for semantic segmentation. The main part of this chapter describes random forests (Section 2.5) and random ferns (Section 2.6). Random forests are powerful discriminative classifiers, which are very fast to evaluate which makes them perfectly usable in an interactive approach. A random forest is the

main classifier used in this thesis and as can be seen, it is also the most powerful classifier in an offline as well as in an online setting (see Section 6). To compare the performance of random forests, random ferns are used. They are similar to forests, except that not each node in a tree has an individual split function, but each depth. A random fern is a semi-naive Bayes classifier and therefore more generative then a random forest.

The information sources used by the classifier in semantic segmentation are called feature channels. Feature channels abstract the pure color information. In this thesis, LAB, image gradients, HOG and LBP are used as feature channels. If a project in 3D is semantically segmented, surface normals are incorporated additionally as feature channels, since they give a powerful information source. To actually use the feature channels, pixel pair features and generalized Haar features are used. The usage of generalized Haar features allows to select the best Haar feature automatically and therefore no predefined Haar features must be specified.

Section 4 starts with a comparison of offline learning and online learning. The differences are demonstrated by a real world example. Section 4.2 shows, how a random forest can be used in an online approach. Therefore two different concepts, supervised pre-training (Section 4.2.1) and unsupervised pre-training (Section 4.2.2) are revealed. The core of an online approach is always how training data arriving at any time can be incorporated as new information. In Section 4.2.3, the algorithms of how random forests and random ferns can be actually implemented are listed. In an interactive approach, the performance of evaluating unseen data is always crucial. Therefore, in Section 4.5 some performance considerations are described.

The application, into which the semantic segmentation functionality is incorporated is described in Section 5. By using an interactive approach, the labels are provided by the human operator. Actually, the user specifies the categories into that the input images should be segmented and provides samples of these categories. The categories can be selected using the so called “Annotation Dialog”, (see Figure 5.8), which was specifically designed for that purpose. The annotations itself can be done comfortably by simply drawing some scribbles (see Figure 1.5b) with the appropriate label onto the source image. Note that a rough annotation of the desired categories is enough. The hard work of doing the exact semantic segmentation is done by the classifier after some labels have been provided by the user. A quality criterion of online algorithms is always how much user interaction is necessary to achieve good results. To minimize the user interaction necessary, a concept called active user guidance has been developed (see Section 5.3). This allows

the user to select the samples, where the classifier is not sure about the correct labelling. Because the MAP classifier output is usually noisy, regularization methods used in this thesis are described in Section 5.4.

In Section 6, the algorithmic part of the thesis is evaluated. The algorithms are evaluated using the eTRIMS database and the MSRC database. The basic evaluation is defined by a supervised trained random forest. Different configurations are evaluated (maximum tree depth, number of trees, unsupervised trained, supervised trained, additional confidence maps, entangled). The same evaluations are done with random ferns too. The evaluations have shown that a random forest with ten trees yields to the best results in the offline setting and in the online setting. Although ferns are more generative than forests, ferns are worse compared to forests in the online setting. However, this is due to the small number of samples used for updating the classifiers. A fern needs a lot more samples to achieve a good performance.

To summarize, the application can be used to perform a semantic segmentation in 2D as well as in a textured 3D environment. Unsupervised semi-online random forests yield to very good results. In 3D pixel synchronous surface normals are used as additional feature channels. It should be noted that entangled decision forests are not usable in an interactive approach, because then for each input image and each depth in the tree, confidence maps must be created and stored, which makes the application slow and memory hungry.

## 7.2 Further Work

Although the performance of the application is very good, we want to implement the evaluation of the classifiers on the GPU. This would lead to an additional speed-up, which is always welcome. Therefore, general purpose graphics processing libraries like Nvidia's Compute Unified Device Architecture (CUDA)[49] or Microsoft's C++ Accelerated Massive Parallelism (AMP) [45] could be used.

In the current configuration, the forest is pre-trained and than the samples are used to update the statistics in the leaf nodes. Another possibility would be to make the random forest online and not only semi-online. This would require to adapt the tree-structure in some way.

When semantic segmentation is performed in 3D, further feature channels could be incorporated. For example digital surface model (DSM) minus digital terrain model (DTM) should give another powerful information source. Additionally, due to the overlap of the images, the votes of all images seeing one pixel should be aggregated to derive the final

label.

# Bibliography

[1] Radhakrishna Achanta, Appu Shaji, Kevin Smith, Aurelien Lucchi, Pascal Fua, and Sabine Süsstrunk. SLIC Superpixels. Technical Report 149300, EPFL, 2010.

[2] Radhakrishna Achanta, Appu Shaji, Kevin Smith, Aurelien Lucchi, Pascal Fua, and Sabine Süsstrunk. SLIC Superpixels Compared to State-of-the-Art Superpixel Methods. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(11):2274–2282, 2012.

[3] Sharon Alpert, Meirav Galun, Ronen Basri, and Achi Brandt. Image Segmentation by Probabilistic Bottom-Up Aggregation and Cue Integration. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, 2007.

[4] Yali Amit and Donald Geman. Shape Quantization and Recognition with Randomized Trees. *Neural Computation*, 9(7):1545–1588, 1997.

[5] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[6] A. Bosch, A. Zisserman, and X. Munoz. Image Classification using Random Forests and Ferns. In *IEEE International Conference on Computer Vision*, pages 1–8, 2007.

[7] Yuri Boykov and Gareth Funka-Lea. Graph Cuts and Efficient N-D Image Segmentation. *International Journal of Computer Vision*, 70(2):109–131, 2006.

[8] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth, 1984.

[9] Leo Breiman. Bagging Predictors. *Machine Learning*, 24(2):123–140, 1996.

[10] Leo Breiman. Random Forests. *Machine Learning*, 45(1):5–32, 2001.

[11] Dorin Comaniciu and Peter Meer. Mean Shift: A Robust Approach Toward Feature Space Analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(5):603–619, 2002.

[12] Daniel Cremers, Mikael Rousson, and Rachid Deriche. A Review of Statistical Approaches to Level Set Segmentation: Integrating Color, Texture, Motion and Shape. *International Journal of Computer Vision*, 72(2):195–215, 2007.

[13] Antonio Criminisi and Jamie Shotton. *Decision Forests for Computer Vision and Medical Image Analysis*. Springer, 2013.

[14] Antonio Criminisi, Jamie Shotton, and Ender Konukoglu. Decision Forests: A Unified Framework for Classification, Regression, Density Estimation, Manifold Learning and Semi-Supervised Learning. *Foundations and Trends in Computer Graphics and Vision*, 7(2–3):81–227, 2011.

[15] Navneet Dalal and Bill Triggs. Histograms of Oriented Gradients for Human Detection. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 1, pages 886–893, 2005.

[16] Navneet Dalal, Bill Triggs, and Cordelia Schmid. Histogram of Oriented Gradients (HOG) for Object Detection. `http://people.cs.missouri.edu/~duanye/cs8690/lecture-notes/HoG.pdf`, 2005. Accessed: 2014-05-16.

[17] Hal Daumé III. Frustratingly Easy Domain Adaptation. In *ACL*, 2007.

[18] Cass Everitt. Projective texture mapping. *White paper, NVidia Corporation*, 4, 2001.

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

[20] Yoav Freund and Robert E. Schapire. A Decision-theoretic Generalization of On-line Learning and an Application to Boosting. In *Computational Learning Theory*, pages 23–37, 1995.

[21] Yoav Freund and Robert E. Schapire. Experiments with a New Boosting Algorithm. In *International Conference on Machine Learning*, pages 148–156, 1996.

[22] Yoav Freund and Robert E. Schapire. A Decision-theoretic Generalization of On-line Learning and an Application to Boosting. 55(1):119–139, 1997.

[23] Björn Fröhlich, Erik Rodner, and Joachim Denzler. Semantic Segmentation with Millions of Features: Integrating Multiple Cues in a Combined Random Forest Approach. In *Asian Conference of Computer Vision*, pages 218–231. Springer, 2013.

[24] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Additive Logistic Regression: a Statistical View of Boosting. *Annals of Statistics*, 28(2):337–407, 2000.

[25] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely Randomized Trees. *Machine Learning*, 63(1):3–42, 2006.

[26] Martin Godec, Peter M. Roth, and Horst Bischof. Hough-based Tracking of Non-Rigid Objects. In *International Conference on Computer Vision*, pages 81–88. IEEE, 2011.

[27] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*, volume 2. Springer, 2009.

[28] Xuming He, Richard S. Zemel, and Miguel Á. Carreira-Perpiñán. Multiscale Conditional Random Fields for Image Labeling. In *Computer Vision and Pattern Recognition*, volume 2, pages 695–702, 2004.

[29] Derek Hoiem. Machine Learning in Computer Vision. `www.cs.uiuc.edu/~dhoiem/presentations/putting_context_into_vision_final.pdf`, 2004. Accessed: 2014-04-24.

[30] Derek Hoiem, Alexei A. Efros, and Martial Hebert. Putting Objects in Perspective. In *Computer Vision and Pattern Recognition*, volume 2, pages 2137–2144, 2006.

[31] Intel. Thread Building Blocks. `https://www.threadingbuildingblocks.org/`, 2014. Accessed: 2014-05-01.

[32] Ernst Ising. Beitrag zur Theorie des Ferromagnetismus. *Zeitschrift für Physik*, 31(1):253–258, 1925.

[33] Ajay Joshi, Anoop Cherian, and Ravishankar Shivalingam. Machine Learning in Computer Vision - A Tutorial. `http://www-users.cs.umn.edu/~cherian/ppt/MachineLearningTut.pdf`. Accessed: 2014-04-08.

[34] Zdenek Kalal, Jiri Matas, and Krystian Mikolajczyk. P-n learning: Bootstrapping binary classifiers by structural constraints. In *Computer Vision and Pattern Recognition*, pages 49–56, 2010.

[35] Filip Korc and Wolfgang Förstner. eTRIMS Image Database for interpreting images of man-made scenes. *Dept. of Photogrammetry, University of Bonn, Tech. Rep. TRIGG-P-2009-01*, 2009.

[36] John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In *International Conference on Machine Learning*, pages 282–289, 2001.

[37] C. Leistner, A. Saffari, J. Santner, and H. Bischof. Semi-Supervised Random Forests. In *International Conference on Computer Vision*, pages 506–513, 2009.

[38] Vincent Lepetit and Pascal Fua. Keypoint Recognition Using Randomized Trees. *Pattern Analysis and Machine Intelligence*, 28(9):1465–1479, 2006.

[39] Fei-Fei Li. Machine Learning in Computer Vision. `http://www.cs.princeton.edu/courses/archive/spr07/cos424/lectures/li-guest-lecture.pdf`. Accessed: 2014-04-08.

[40] David G. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[41] L. Lucchese and S. K. Mitra. Color Image Segmentation: A State-of-the-Art Survey. 2001.

[42] D. Martin, C. Fowlkes, D. Tal, and J. Malik. A Database of Human Segmented Natural Images and its Application to Evaluating Segmentation Algorithms and Measuring Ecological Statistics. In *International Conference on Computer Vision*, volume 2, pages 416–423, 2001.

[43] Mathworks. Correcting Nonuniform Illumination. `http://www.mathworks.de/de/help/images/examples/correcting-nonuniform-illumination.html`, 2014. Accessed: 2014-05-05.

[44] Mathworks. Marker-Controlled Watershed Segmentation. `http://www.mathworks.de/de/help/images/examples/marker-controlled-watershed-segmentation.html`, 2014. Accessed: 2014-05-05.

[45] Microsoft. C++ AMP (C++ Accelerated Massive Parallelism). `http://msdn.microsoft.com/en-us/library/hh265137.aspx`, 2014. Accessed: 2014-05-14.

[46] Microsoft. Parallel Pattern Library. `http://msdn.microsoft.com/en-us/library/dd492418.aspx`, 2014. Accessed: 2014-05-01.

[47] Albert Montillo, Jamie Shotton, John Winn, Juan Eugenio Iglesias, Dimitri Metaxas, and Antonio Criminisi. Entangled Decision Forests and Their Application for Semantic Segmentation of CT Images. In *International Conference on Information Processing in Medical Imaging*, pages 184–196, 2011.

[48] D. Mumford and J. Shah. Optimal Approximations by Piecewise Smooth Functions and Associated Variational Problems. *Communications on Pure and Applied Mathematics*, 42(5):577–685, 1989.

[49] Nvidia. CUDA C Programming Guide. Technical report, Nvidia, 2014.

[50] Timo Ojala, Matti Pietikäinen, and David Harwood. A Comparative Study of Texture Measures with Classification Based on Featured Distributions. *Pattern Recognition*, 29(1):51–59, 1996.

[51] Timo Ojala, Matti Pietikainen, and Topi Maenpaa. Multiresolution Gray-Scale and Rotation Invariant Texture Classification with Local Binary Patterns. *Pattern Analysis and Machine Intelligence*, 24(7):971–987, 2002.

[52] R. B. Potts. Some Generalized Order-Disorder Transformations. *Mathematical Proceedings of the Cambridge Philosophical Society*, 48:106–109, 1952.

[53] Charles Poynton. Colorfaq. `http://www.poynton.com/ColorFAQ.html`. Accessed: 2014-03-18.

[54] J. R. Quinlan. Induction of Decision Trees. *Machine Learning*, 1(1):81–106, 1986.

[55] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[56] A. Rabinovich, A. Vedaldi, C. Galleguillos, E. Wiewiora, and S. Belongie. Objects in Context. In *International Conference on Computer Vision*, pages 1–8, 2007.

[57] Microsoft Research. MSRC Database. `http://research.microsoft.com/en-us/projects/objectclassrecognition/`. Accessed: 2014-03-25.

[58] Grégory Rogez, Jonathan Rihan, Srikumar Ramalingam, Carlos Orrite, and Philip H. S. Torr. Randomized trees for human pose detection. In *Computer Vision and Pattern Recognition*, pages 1–8, 2008.

[59] Bryan C. Russell, Antonio Torralba, Kevin P. Murphy, and William T. Freeman. LabelMe: A Database and Web-Based Tool for Image Annotation. *International Journal of Computer Vision*, 77(1-3):157–173, 2008.

[60] Amir Saffari, Christian Leistner, Jakob Santner, Martin Godec, and Horst Bischof. On-line Random Forests. In *International Conference on Computer Vision Workshops*, pages 1393–1400, 2009.

[61] Konrad Schindler. Orthophoto. `http://www.igp.ethz.ch/photogrammetry/education/lehrveranstaltungen/PhotogrammetryFS14/coursematerial/Photo-FS2014-11-ortho.pdf`. Accessed: 2014-03-04.

[62] Samuel Schulter, Christian Leistner, Peter M. Roth, Horst Bischof, and Luc Van Gool. On-line Hough Forests. In *Brithish Machine Vision Conference*, pages 1–11, 2011.

[63] Claude Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 27:379–423, 623–656, 1948.

[64] Jianbo Shi and Jitendra Malik. Normalized Cuts and Image Segmentation. *Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000.

[65] J. Shotton, M. Johnson, and R. Cipolla. Semantic Texton Forests for Image Categorization and Segmentation. In *Computer Vision and Pattern Recognition*, pages 1–8, 2008.

[66] Jamie Shotton, John Winn, Carsten Rother, and Antonio Criminisi. TextonBoost: Joint Appearance, Shape and Context Modeling for Multi-class Object Recognition and Segmentation. In *European Conference on Computer Vision*, pages 1–15. 2006.

[67] Heung-Yeung Shum and Sing Bing Kang. A Review of Image-based Rendering Techniques. In *Visual Communications and Image Processing (VCIP)*, pages 2–13, 2000.

[68] Milan Sonka, Vaclav Hlavac, and Roger Boyle. *Image Processing, Analysis, and Machine Vision*. 2008.

[69] Richard Szeliski. *Computer Vision: Algorithms and Applications*. Springer, 2010.

[70] Michael E Tipping. Sparse bayesian learning and the relevance vector machine. *The journal of machine learning research*, 1:211–244, 2001.

[71] Antonio Torralba, Kevin P. Murphy, and William T. Freeman. Sharing Visual Features for Multiclass and Multiview Object Detection. *Pattern Analysis and Machine Intelligence*, 29(5):854–869, 2007.

[72] Zhuowen Tu and Xiang Bai. Auto-Context and Its Application to High-Level Vision Tasks and 3D Brain Image Segmentation. *Pattern Analysis and Machine Intelligence*, 32(10):1744–1757, 2010.

[73] Markus Unger. *Convex Optimization for Image Segmentation*. PhD thesis, Graz University of Technology, 2012.

[74] Alexander Vezhnevets, Vittorio Ferrari, and Joachim M. Buhmann. Weakly Supervised Structured Output Learning for Semantic Segmentation. In *Computer Vision and Pattern Recognition*, pages 845–852. IEEE, 2012.

[75] Paul Viola and Michael Jones. Rapid Object Detection using a Boosted Cascade of Simple Features. In *Computer Vision and Pattern Recognition*, volume 1, pages 511–518, 2001.

[76] Paul Viola and Michael J. Jones. Robust Real-Time Face Detection. *International Journal of Computer Vision*, 57(2):137–154, 2004.

[77] C. Vondrick, A. Khosla, T. Malisiewicz, and A. Torralba. HOGgles: Visualizing Object Detection Features. *International Conference on Computer Vision*, pages 1–8, 2013.

[78] Christopher Zach, David Gallup, Jan-Michael Frahm, and Marc Niethammer. Fast Global Labeling for Real-Time Stereo Using Multiple Plane Sweeps. In *VMV*, pages 243–252, 2008.

[79] Mustafa Özuysal, Michael Calonder, Vincent Lepetit, and Pascal Fua. Fast Keypoint Recognition Using Random Ferns. *Pattern Analysis and Machine Intelligence*, 32(3):448–461, 2010.

[80] Mustafa Özuysal, Pascal Fua, and Vincent Lepetit. Fast keypoint recognition in ten lines of code. In *Computer Vision and Pattern Recognition*, pages 1–8, 2007.