Learning Convolutional Neural Networks for Object Detection with Very Little Training Data

Christoph Reinders*, Hanno Ackermann*, Michael Ying Yang[†],
Bodo Rosenhahn*

*Institute for Information Processing, Leibniz University Hanover, Hanover, Germany [†]Scene Understanding Group, University of Twente, Enschede, The Netherlands

Contents

4.1	Introduction	66
4.2	Fundamentals	68
4.2.1	Types of Learning	68
4.2.2	Convolutional Neural Networks	70
4.2.2.1	Artificial neuron	70
4.2.2.2	Artificial neural network	71
4.2.2.3	Training	73
4.2.2.4	Convolutional neural networks	75
4.2.3	Random Forests	77
4.2.3.1	Decision tree	77
4.2.3.2	Random forest	79
4.3	Related Work	79
4.4	Traffic Sign Detection	81
4.4.1	Feature Learning	81
4.4.2	Random Forest Classification	82
4.4.3	RF to NN Mapping	82
4.4.4	Fully Convolutional Network	84
4.4.5	Bounding Box Prediction	85
4.5	Localization	86
4.6	Clustering	87
4.7	Dataset	89
4.7.1	Data Capturing	90
4.7.2	Filtering	90
4.8	Experiments	91

4.8.1	Training and Test Data	91
4.8.2	Classification	92
4.8.3	Object Detection	92
4.8.4	Computation Time	95
4.8.5	Precision of Localizations	96

4.9 Conclusion 98

Acknowledgment 98

References 98

4.1 Introduction

Cycling as a mode of transport has attracted growing interest. Cities are transforming urban transportation to improve their infrastructure. Amsterdam and Copenhagen, for example, are pioneers for cycling-friendly cities. While current development shows more and more infrastructure improvements, road conditions can vary greatly. Cyclists are frequently confronted with challenges such as absence of bicycle lanes, being overlooked by cars, or bad roads. Arising safety concerns represent a barrier for using bicycles. Thus, recommending fast and safe routes for cyclists has great potential in terms of environmental and mobility aspects. This, in turn, requires detailed information about roads and traffic regulations.

For cars, precise information has become available. Google, for example, started the Google Street View project in which data is captured by many cars. These are equipped with stereo cameras which already offer a good 3D estimation in a certain range, lidar, and other sensors. Additionally the cars provide computational power as well as power supply. In research, popular datasets like GTSRB [1], KITTI [2], and Cityscapes [3] have been published.

In recent years, users are increasingly involved in the data collection. Crowdsourcing data enables to create large amount of real-world datasets. For example, the smart phone app Waze collects data such as GPS-position and speed from multiple users to predict traffic jams. OpenStreetMap aims to build a freely available map of the world to which users can easily contribute.

Machine learning techniques have shown great success for analyzing this data. Most supervised methods especially convolutional neural networks, however, require large datasets of labeled data. While large datasets have been published regarding cars, for cyclists very few labeled data are available although appearance, point of view, and positioning of even relevant objects differ. Unfortunately, labeling data is costly and requires a huge amount of work.

Our aim is to collect information which is of interest to cyclists. Analyzing street data for cyclists cannot be straightforwardly done by using data captured for cars due to different perspectives, different street signs, and routes prohibited for cars but not for bicycles, as shown

Figure 4.1 : Real-world data has great potential to provide traffic information that is of interest to cyclists. For example, roads that are prohibited for cars but free for cyclists (left), bicycle lines in parks (middle), or bicycle boulevards which are optimized for cyclists (right). All three examples are recognized by our system.

in Fig. 4.1. For collecting real-world data, we involve users by using smart phones that are attached to their bicycles. Compared to other systems, like for example Google Street View, our recording system consists of a single consumer camera and can only rely on a limited power supply and little computational power. On the other hand, our system has very low hardware costs and is highly scalable so that crowdsourcing becomes possible.

Although capturing data becomes easy with this system, generating labels is still very expensive. Thus, in this chapter we further address the problem of learning with extremely little labeled data to recognize traffic signs relevant for cyclists. We combine multiple machine learning techniques to create a system for object detection. Convolutional neural networks (CNNs) have shown to learn strong feature representations. On the other hand, random forests (RFs) achieve very good results in regression and classification tasks even when little labeled data is available. To combine both advantages we generate a feature extractor using a CNN and train a random forest based on the features. We map the random forest to a neural network and transform the full pipeline into a fully convolutional network. Thus, due to the shared features, processing full images is significantly accelerated. The resulting probability map is used to perform object detection. In a next step, we integrate information of a GPS-sensor to localize the detections on the map.

Figure 4.2 : Supervised learning methods (A) are trained on input–target pairs to classify the data points into the classes red and blue. For semi-supervised learning methods (B) only a few labeled data points are given additional to a large amount of unlabeled data points. Unsupervised learning methods (C) process the training data without any labels and try to find structures in the data.

This chapter further extends our previous work [4]. We increased the traffic sign dataset for training and testing from 297 to 524 examples. Additionally, the feature generating CNN is improved and a larger network is used. Finally, the GPS-sensors from the smart phones have shown to be not very precise. To improve the localization accuracy, we added a clustering process which identifies and merges multiple observations of the same traffic sign.

4.2 Fundamentals

In this section, the fundamental concepts used throughout this chapter are presented. At first, a short overview over different types of learning is given. In the second section, the origins of neural networks and convolutional neural networks are presented as well as a brief introduction to the so-called *back-propagation* algorithm for training neural networks. In the last section, random forests which consist of multiple decision trees are explained.

4.2.1 Types of Learning

Machine learning algorithms can be broadly divided into supervised learning, semi-supervised learning, and unsupervised learning [5, pp. 102–105]. The division depends on the training data that is provided during the learning process. In this section the different types of learning algorithms are briefly presented. An example of training data for each method is illustrated in Fig. 4.2.

Supervised learning. In supervised learning, labeled data is provided to the learning process meaning that each example is annotated with the desired target output. The dataset for training consists of N input–target pairs,

$$X = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(N)}, y^{(N)})\}.$$

Each training pair consists of input data $x^{(i)}$ together with the corresponding target $y^{(i)}$. The goal of the algorithm is to learn a mapping from the input data to the target value so that the target y^* for some unseen data x^* can be predicted. Supervised learning can be thought of as a teacher that evaluates the performance and identifies errors during training to improve the algorithm.

Common tasks in supervised learning are classification and regression. In classification, the target is a discrete value which represents a category such as “red” and “blue” (see Fig. 4.2A). Another example is the classification of images to predict the object that is shown in the image such as for instance “car”, “plane”, or “bicycle”. In regression, the target is a real value such as “dollars” or “length”. A regression task is for instance the prediction of house prices based on data of the properties such as location or size.

Popular examples of supervised learning methods are random forests, support vector machines, and neural networks.

Semi-supervised learning. Semi-supervised learning is a mixture between supervised learning and unsupervised learning. Typically, supervised learning methods require large amounts of labeled data. Whereas the collection of data is often cheap, labeling data can usually only be achieved at enormous costs because experts have to annotate the data manually. Semi-supervised learning combines supervised learning and the usage of unlabeled data. For that, the dataset consists of N labeled examples,

$$X_l = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(N)}, y^{(N)})\},$$

together with M unlabeled examples,

$$X_u = \{x^{(N+1)}, x^{(N+2)}, \dots, x^{(N+M)}\}.$$

Usually, the number of labeled examples N is much smaller than the number of unlabeled examples M . Semi-supervised learning is illustrated in Fig. 4.2B for the classification of data points.

Unsupervised learning. In unsupervised learning, unlabeled data is provided to the learning process. The aim of the learning algorithm is to find structures or relationships in the data without having information about the target. The training set consists of N training samples,

$$X = \{x^{(1)}, x^{(2)}, \dots, x^{(N)}\}.$$

Popular unsupervised learning methods are for example k-means and mean shift. For unsupervised learning, usually, a distribution of the data has to be defined. K-means assumes that the data is convex and isotropic and requires a predefined number of classes. For mean shift a kernel along with a bandwidth is defined such as for example a Gaussian kernel. Typical tasks in unsupervised learning are density estimation, clustering, and dimensionality reduction. An example of a clustering task is shown in Fig. 4.2C.

4.2.2 Convolutional Neural Networks

Convolutional neural networks have shown great success in recent years. Especially since Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton won the ImageNet Large-Scale Visual Recognition Challenge in 2012 the topic of research has attracted much attention [6]. In this section an overview of convolutional neural networks is given. First, the origins of neural networks are described and afterwards the learning process *back-propagation* is explained. Finally, convolutional neural networks are presented.

4.2.2.1 Artificial neuron

Neural networks have been biologically inspired by the brain. The brain is a very efficient information-processing system which consists of single units, so-called *neurons*, which are highly connected with each other [7, pp. 27–30].

The first model of an artificial neuron was presented by Warren McCulloch and Walter Pitts in 1943 [8] and was called the *McCulloch–Pitts model*. The scientists drew a connection between the biological neuron and logical gates and developed a simplified mathematical description of the biological model. In 1957, Frank Rosenblatt developed the so-called *perceptron* [9], which is based on the work of McCulloch and Pitts. The model has been further generalized to a so-called *artificial neuron*. An artificial neuron has N inputs and one output a , as illustrated in Fig. 4.3. Each input is multiplied by a weight w_i . Afterwards all weighted inputs are added up together with a bias b , which gives the weighted sum z :

$$z = \sum_{i=1}^N x_i \cdot w_i + b. \quad (4.1)$$

Finally, the output is calculated using an activation function $\phi(\cdot)$:

$$a = \phi(z). \quad (4.2)$$

Early models such as the perceptron often used a step function $g(x)$ as activation function where g is defined as

Figure 4.3 : Model of an artificial neuron. The inputs are weighted and added together with the bias. Afterwards, the weighted sum is passed to an activation function to calculate the output.

$$g(x) = \begin{cases} 1 & x \geq 0, \\ 0 & \text{otherwise.} \end{cases} \quad (4.3)$$

Another activation function that is commonly used is the sigmoid function,

$$\text{sig}(x) = \frac{1}{1 + e^{-x}}. \quad (4.4)$$

Different from the step function, the sigmoid function is differentiable. This is an important property for training as shown later in this chapter. In recent years, another activation function, called the *rectified linear unit (ReLU)* became popular, which has been proposed by Krizhevsky et al. [6]. ReLU outputs 0 if the input value x is smaller than zero and otherwise the input value x :

$$\text{ReLU}(x) = \max(0, x). \quad (4.5)$$

Compared with the sigmoid function, ReLU is non-saturating for positive inputs and has shown to accelerate the training process.

This model of an artificial neuron can be used to classify input data into two classes. However, this only works for data which is linearly separable. To calculate more complex functions, multiple neurons are connected as presented in the next section.

4.2.2.2 Artificial neural network

Neural networks are arranged in layers. Each network has an input layer, one or more hidden layers, and an output layer. An example of a two-hidden-layer network is presented in Fig. 4.4. The design of the input and output layer is often straightforward compared to the design of the hidden layers. The input layer is the first layer in the network. Its so-called input neurons store the input data and perform no computation. If the input is for example an image of size 32×32 pixels, the number of input neurons is $1024 = 32 \cdot 32$. The output layer is the

Figure 4.4 : A multilayer perceptron with two hidden layers.

last layer in the network and contains the output neurons. For example, if a network classifies an image into one out of ten classes, the output layer contains one neuron for each class, indicating the probability that the input belongs to this class.

Formally, a neural network has L layers where the first layer is the input layer and the L th layer is the output layer. Neural networks with multiple layers are also called *multilayer perceptrons*. They are one of the simplest types of feed-forward neural networks, which means that the connected neurons build an acyclic directed graph.

Each layer l has n_l neurons and a neuron is connected to all neurons in the previous layer. The weight between the k th neuron in the $(l - 1)$ th layer and the j th neuron in the l th layer is denoted by $w_{k,j}^l$. Similarly, the bias of the j th neuron in the l th layer is denoted by b_j^l . Similar to Eq. (4.1) the weighted sum z_j^l of a neuron is calculated by

$$z_j^l = \sum_{k=1}^{n_{l-1}} a_k^{l-1} \cdot w_{k,j}^l + b_j^l. \quad (4.6)$$

Afterwards the activation function is used to calculate the output a_j^l of a neuron

$$a_j^l = \phi(z_j^l). \quad (4.7)$$

To simplify the notation, the formulas can be written in matrix form. For each layer, a bias vector b^l , a weighted sum vector z^l , and an output vector a^l are defined:

$$b^l = [b_1^l \quad b_2^l \quad \dots \quad b_{n_l}^l]^T, \quad (4.8)$$

$$z^l = [z_1^l \quad z_2^l \quad \dots \quad z_{n_l}^l]^T, \quad (4.9)$$

$$a^l = [a_1^l \quad a_2^l \quad \dots \quad a_{n_l}^l]^T. \quad (4.10)$$

The weights for each layer can be expressed by a matrix $w^l \in \mathbb{R}^{n_l \times n_{l-1}}$:

$$w^l = \begin{bmatrix} w_{1,1}^l & \cdots & w_{n_{l-1},1}^l \\ \vdots & \ddots & \vdots \\ w_{1,n_l}^l & \cdots & w_{n_{l-1},n_l}^l \end{bmatrix}. \quad (4.11)$$

The matrix has n_l rows and n_{l-1} columns which corresponds to the number of neurons in layer l and layer $(l - 1)$. The k th row of w^l contains all weights of the k th neuron in layer l connecting the neuron to all neurons in layer $(l - 1)$. Using the matrix form, the weighted sum for each layer can be calculated by multiplying the weight matrix with the output of the previous layer and adding the biases:

$$z^l = w^l a^{l-1} + b^l. \quad (4.12)$$

Applying the activation to each element of the weighted sum vector, the output of each layer is calculated by

$$a^l = \phi(z^l). \quad (4.13)$$

A neural network takes some input data x and calculates the output of the network $N(x)$, which is defined as the output of the last layer $N(x) = a^L(x)$. This is done by processing the network layer by layer. First of all the input data is passed to the input neurons $a^1(x) = x$. Afterwards all layers are processed by calculating the weighted sum and the output. Finally, the output of the network $a^L(x)$ is calculated.

4.2.2.3 Training

The goal of the training process is to automatically learn the weights and biases. Finding the optimal configuration can be challenging especially in larger networks which have thousands or millions of parameters. Whereas the McCulloch–Pitts model used fixed weights, Rosenblatt proposed a learning rule for adjusting the weights of a perceptron. In the 1970s and 1980s a much faster algorithm called *back-propagation* has been developed. Various researchers have worked towards similar ideas, including Werbos [10] and Parker [11]. The back-propagation algorithm has been popularized by a work of Rumelhart, Hinton, and Williams in 1986 [12].

Cost function. In order to learn the weights and biases, a training set of input vectors $x^{(i)}$ together with a corresponding set of target vectors $y^{(i)}$ are provided. During a *forward pass* the output $a^L(x)$ of a network is calculated. To quantify the error made by a network a loss

function C is introduced. A loss function that is often used is the *Euclidean loss*. It is defined as the sum over the squared differences between the target vector and the output vector

$$C = \frac{1}{2} \|y^{(i)} - a^L(x^{(i)})\|^2 = \frac{1}{2} \sum_j \left(y_j^{(i)} - a_j^L(x^{(i)})\right)^2. \quad (4.14)$$

Back-propagation. The objective of the training process is to adjust the weights and biases so that the loss is minimized. To understand how changes of the weights and biases change the cost function, let Δz_j^l be a small change that is added to the weighted sum of the j th neuron in the l th layer. Instead of $\phi(z_j^l)$ the neuron outputs $\phi(z_j^l + \Delta z_j^l)$, which is propagated through the network and leads to an overall change of the cost function by $\frac{\partial C}{\partial z_j^l} \Delta z_j^l$.

If the value of $\partial C / \partial z_j^l$ is large, Δz_j^l can be chosen to have opposite sign so that the loss is reduced. If the value of $\partial C / \partial z_j^l$ is small, Δz_j^l cannot improve the loss and the neuron is assumed to be nearly optimal. Let δ_j^l be defined as the error by the j th neuron in the l th layer,

$$\delta_j^l = \frac{\partial C}{\partial z_j^l}. \quad (4.15)$$

Back-propagation is based on four fundamental equations. First of all the error of each neuron in the last layer is calculated:

$$\delta^L = \nabla_a C \odot \phi'(z^L). \quad (4.16)$$

Afterwards the error is propagated backwards through the network so that step by step the error of the neurons in the $(l + 1)$ th layer is propagated to the neurons in the l th layer:

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \phi'(z^l). \quad (4.17)$$

The error can be used to calculate the partial derivatives $\partial C / \partial w_{k,j}^l$ and $\partial C / \partial b_j^l$ as follows:

$$\frac{\partial C}{\partial w_{k,j}^l} = \delta_j^l a_k^{l-1}, \quad (4.18)$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l, \quad (4.19)$$

which indicate how a change of a weight or bias influences the loss function. Finally, the weights and biases are adjusted by subtracting the corresponding partial derivative scaled with a learning rate α from the current value,

$$\hat{w}_{k,j}^l = w_{k,j}^l - \alpha \cdot \frac{\partial C}{\partial w_{k,j}^l}, \quad (4.20)$$

$$\hat{b}_j^l = b_j^l - \alpha \cdot \frac{\partial C}{\partial b_j^l}. \quad (4.21)$$

As a result, the weights and biases are optimized iteratively to minimize the loss function.

4.2.2.4 Convolutional neural networks

Neural networks as explained in the last section are also called *fully-connected neural networks*. Every neuron is connected to every neuron in the previous layer. For a color image of size 32×32 pixels with three color channels, this means that each neuron in the first hidden layer has $3072 = 32 \cdot 32 \cdot 3$ weights. Although at first glance acceptable, for large images the number of variables is extremely large. For example, a color image of size 200×200 pixels with three color channels already requires $120\,000 = 200 \cdot 200 \cdot 3$ weights for each neuron in the first hidden layer. Additionally, fully-connected neural networks are also not taking the spatial structure into account. Neurons that are far away from another are treated equally to neurons that are close together.

Convolutional neural networks are designed to take advantage of the two-dimensional structure of an input image. Instead of learning fully-connected neurons, filters are learned that are convolved over the input image. The idea has been inspired by the visual cortex. Hubel and Wiesel [13] showed in 1962 that some neural cells are sensitive to small regions in the visual field and respond to specific features. In 1998, convolutional neural networks were popularized by Lecun, Botto, Bengio, and Haffner [14].

The first difference between regular neural networks and convolutional neural networks is the arrangement of the data. In order to take into account that images or other volumetric data are processed, the data in convolutional neural networks is arranged in 3D volumes. Each data blob in a layer has three dimensions: height, width, and depth. Each layer receives an input 3D volume and transforms it to an output 3D volume.

Convolutional layer. Whereas fully-connected neurons are connect to every neuron in the previous layer, neurons in convolutional layers will be connected to only a small region of the input data. This region is called *local receptive field* for a hidden neuron. Instead of applying different weights and biases for every local receptive field, the neurons are using shared weights and biases. The idea behind this is that a feature that is learned at one position might also be useful at a different position. Additionally, the number of parameters will decrease significantly. The shared weights and biases are defined as *filter*.

The size of the filters is usually small in height and width whereas the depth always equals the depth of the input volume. For example, a filter of size $3 \times 5 \times 5$ is often used in the first

hidden layer, i.e. 3 pixels in depth, because the input image has three color channels, 5 pixels in height, and 5 pixels in width. In general, a filter has depth C_{in} , height K_{h} , and width K_{w} , where C_{in} is the depth of the input volume. This results in $C_{\text{in}} \cdot K_{\text{h}} \cdot K_{\text{w}}$ weights plus one bias per filter.

Convolutional layers consist of a set of K filters. Each filter is represented as a three-dimensional matrix W_i of size $C_{\text{in}} \times K_{\text{h}} \times K_{\text{w}}$ and a bias b_i . To calculate the weighted sum Z of a layer, each filter is slid across the width and height of the input volume. Therefore the dot product between the weights of the filter W_i and the input volume I at any position is computed:

$$Z[i, y, x] = \sum_{c=0}^{C_{\text{in}}-1} \sum_{l=0}^{K_{\text{h}}-1} \sum_{m=0}^{K_{\text{w}}-1} W_i[c, l, m] I[c, y+l, x+m] + b_i \quad (4.22)$$

where i denotes the filter index and x, y the spatial position. Each filter produces a two-dimensional feature map. These feature maps are stacked along the depth and generate a three-dimensional volume Z . Afterwards the activation output A is calculated by applying the activation function to each element of the matrix Z ,

$$A[i, y, x] = \phi(Z[i, y, x]). \quad (4.23)$$

Each convolutional layer transforms an input volume of size $C_{\text{in}} \times H_{\text{in}} \times W_{\text{in}}$ to an output volume of size $C_{\text{out}} \times H_{\text{out}} \times W_{\text{out}}$. Additional to the number of filters K and the filter size $K_{\text{h}} \times K_{\text{w}}$, further parameters of a convolutional layer are the *stride* S and *padding* P . The stride S defines the number of pixels a filter is moved when sliding a filter over the input volume. While $S = 1$ refers to the standard convolution, $S > 1$ will skip some pixels and lead to a smaller output size. The padding P adds additionally P pixels to the border of the input volume. For example, it can be used to create an output volume that has the same size as the input volume. In general, because the feature maps are stacked, the output depth C_{out} equals the number of filters K that are used. The width and height of the output volume can be calculated by

$$W_{\text{out}} = (W_{\text{in}} - K_{\text{w}} + 2P)/S + 1, \quad (4.24)$$

$$H_{\text{out}} = (H_{\text{in}} - K_{\text{h}} + 2P)/S + 1. \quad (4.25)$$

Pooling layer. Another layer that is commonly used is the pooling layer. Usually, it is inserted directly after a convolutional layer. Pooling operates independently on every depth slice and applies a filter of size $K \times K$ to summarize the information. Similar to convolutional layers, a stride S controls the amount of pixels the filter is moved. Most frequently *max pooling* is used. As shown in Fig. 4.5, max pooling takes the maximum over each region and outputs

Figure 4.5 : Max pooling layer. Each depth slice is processed by taking the maximum of each 2×2 window.

a smaller feature map. For example, a filter size of 2×2 is commonly used which reduces the amount of information by a factor of 4. A pooling layer keeps the information that a feature has found but leaves out the exact position. By summarizing the features the spatial size is decreased. Thus, fewer parameters are needed in later layers which reduces the amount of memory that is required and the computation time. Other types of pooling functions are *average pooling* and *L2-norm pooling*.

Dropout layer. A common problem when using neuron networks is overfitting. Because of too many parameters, a network might learn to memorize the training data and does not generalize to unseen data. *Dropout* is a regularization technique to reduce overfitting [15]. A *dropout-ratio* p is defined and during training each neuron is temporarily removed with a probability p and kept with a probability $p - 1$, respectively. An example is illustrated in Fig. 4.6. This process is repeated so that in each iteration different networks are used. In general, dropout has shown to improve the performance of networks.

4.2.3 Random Forests

The random forest algorithm is a supervised learning algorithm for classification and regression. A random forest is an ensemble method that consists of multiple decision trees. The first work goes back to Ho [16] in 1995 who introduced random decision forests. Breiman [17] further developed the idea and presented the random forest algorithm in 2001.

4.2.3.1 Decision tree

A decision tree consists of *split nodes* $\mathcal{N}^{\text{Split}}$ and *leaf nodes* $\mathcal{N}^{\text{Leaf}}$. An example is illustrated in Fig. 4.7. Each split node $s \in \mathcal{N}^{\text{Split}}$ performs a split decision and routes a data sample x to the left child node $\text{cl}(s)$ or to the right child node $\text{cr}(s)$. When using axis-aligned split decisions the split rule is based on a single split feature $f(s)$ and a threshold value $\theta(s)$:

Figure 4.6 : Dropout temporarily removes neurons so that with each iteration different network structures are trained. (A) Standard neural network. (B) After applying dropout.

Figure 4.7 : An example of a decision tree that predicts whether or not to go hiking today. Split nodes (green) evaluate the data and route to the next node. Leaf nodes (blue) contain the possible outputs of the decision tree. Starting at the root node, the data is routed through the tree based on the split rules. Finally, a leaf node is reached which contains the decision output. In this example the output is “yes” or “no”.

$$x \in \text{cl}(s) \iff x_{f(s)} < \theta(s), \quad (4.26)$$

$$x \in \text{cr}(s) \iff x_{f(s)} \geq \theta(s). \quad (4.27)$$

The data sample x is routed to the left child node if the value of feature $f(s)$ of x is smaller than a threshold $\theta(s)$ and to the right child node otherwise. All leaf nodes $l \in \mathcal{N}^{\text{Leaf}}$ store votes for the classes $y^l = (y_1^l, \dots, y_C^l)$, where C is the number of classes.

Decision trees are grown using training data. Starting at the root node, the data is recursively split into subsets. In each step the best split is determined based on a criterion. Commonly

used criteria are *Gini index* and *entropy*:

$$\text{Gini index: } G(E) = 1 - \sum_{j=1}^C p_j^2, \quad (4.28)$$

$$\text{entropy: } H(E) = - \sum_{j=1}^C p_j \log p_j. \quad (4.29)$$

The algorithm for constructing a decision tree works as follows:

1. Randomly sample n training samples with replacement from the training dataset.
2. Create a root node and assign the sampled data to it.
3. Repeat the following steps for each node until all nodes consists of a single sample or samples of the same class:
 - a. Randomly select m variables out of M possible variables.
 - b. Pick the best split feature and threshold according to a criterion, for example Gini index or entropy.
 - c. Split the node into two child nodes and pass the corresponding subsets.

For images, the raw image data is usually not used directly as input for the decision trees. Instead, features such as for instance HOG features or SIFT features are calculated for a full image or image patch. This additional step represents an essential difference between decision trees and convolutional neural networks. Convolutional neural networks in comparison are able to learn features automatically.

4.2.3.2 Random forest

The prediction with decision trees is very fast and operates on high-dimensional data. On the other hand, a single decision tree has overfitting problems as a tree grows deeper and deeper until the data is separated. This will reduce the training error but potentially results in a larger test error.

Random forests address this issue by constructing multiple decision trees. Each decision tree uses a randomly selected subset of training data and features. The output is calculated by averaging the individual decision tree predictions. As a result, random forests are still fast and additionally very robust to overfitting. An example of a decision tree and a random forest is shown in Fig. 4.8.

4.3 Related Work

In recent years, convolutional neural networks have become the dominant approach for many vision-based tasks such as object detection and scene analysis [18,19]. Girshick et al. [20]

Figure 4.8 : A decision tree (B) and a random forest (C) are trained to classify the data points in (A). The decision boundaries are shown in (B) and (C). The dark red and blue colors indicate areas which are clearly classified as red and blue points, respectively. The random forest additionally models the uncertainty which is indicated by a color between red and blue.

proposed a multistage pipeline called regions with convolutional neural networks (R-CNNs) for the classification of region proposals to detect objects. It achieves good results but the pipeline is less efficient because features of each region proposal need be computed repeatedly. In the SPP-net [21], this problem has been addressed by introducing a pooling strategy to calculate the feature map only once and generate features in arbitrary regions. Fast R-CNN [22] further improves the speed and accuracy by combining multiple stages. A drawback of these algorithms is their large dependence on the region proposal method. Faster R-CNN [23] combines the region proposal mechanism and a CNN classifier within a single network by introducing a region proposal network. Due to shared convolutions, region proposals are generated at nearly no extra cost. Other networks such as SSD [24], YOLO [25], and RetinaNet [26] directly regress bounding boxes without generating object proposals in an end-to-end network. These one-stage detectors are extremely fast but come with some compromise in detection accuracy in general. Overall, the one-stage and two-stage convolutional neural networks for object detection perform very well. However, they typically consist of millions of variables and for estimating those, a large amount of labeled data is required for training.

Feature learning and transferring techniques have been applied to reduce the required amount of labeled data [27]. The problem of insufficient training data has also been addressed by other work such as in [28] and [29]. Moysset et al. [28] proposed a new model that predicts the bounding boxes directly. Wagner et al. [29] compared unsupervised feature learning methods and demonstrated performance boosts by pre-training. Although transfer learning techniques are applied, the networks still have a large number of variables for fine-tuning.

A different approach is the combination of random forests and neural networks. Deep neural decision forests [30] unifies both in a single system that is trained end-to-end. Sethi [31] and Welbl [32] presented a mapping of random forests to neural networks. The mapping can be used for several applications. Massiceti et al. [33] demonstrated the application for camera localization. Richmond et al. [34] explored the mapping of stacked RFs to CNNs and an approximate mapping back to perform semantic segmentation.

4.4 Traffic Sign Detection

In this section, we present a system for detecting traffic signs. To overcome the problem of lack of data, we first build a classifier that predicts the class probabilities of a single image patch. This is done in two steps. First, we train a CNN on a different dataset where a large amount of data is available. Afterwards we use the generated features, extract the feature vectors, and train a random forest. The resulting classifier can be used to perform patch-wise prediction and to build a probability map for a given full image. Subsequently, all traffic signs are extracted and the detection system outputs the class and the corresponding bounding box.

Finally, the processing of full images is accelerated. By mapping the random forest to a neural network, it becomes possible to combine feature generation and classification. Afterwards we transform the neural network to a fully convolutional network.

4.4.1 Feature Learning

We learn features by training a convolutional neural network CNN^F . The patch size is 32×32 . We adopt the network architecture of Springenberg et al. [35], which yields good results on datasets like CIFAR-10, CIFAR-100, and ImageNet. The model *ALL-CONV-C* performed best and is used in this work. The network has a simple regular structure consisting of convolution layers only. Instead of pooling layers convolutional layers with stride of two are used. Additionally, the fully-connected layers that are usually at the end of a convolutional neural network, are replaced by 1×1 convolutional layers followed by an average pooling layer. Thus the output of the last layer has a spatial size of 1×1 and a depth C , where C equals the number of classes.

Because we have only very little labeled data available, we train the network on the larger dataset GTSRB [1]. After training, the resulting network CNN^F can be used to generate feature vectors by passing an input image to the network and performing a forward pass. The feature vectors can be extracted from the last convolutional layer or the last convolutional layer before the 1×1 convolutional layers, respectively. In our network this corresponds to the seventh convolutional layer, denoted by $\text{CNN}_{relu7}^F(x)$.

Figure 4.9 : A decision tree (left) and the mapped neural network (right). Each *split node* in the tree – indicated as circle – creates a neuron in the first hidden layer which evaluates the split rule. Each *leaf node*—indicated as rectangle—creates a neuron in the second hidden layer which determines the leaf membership. For example, a routing to leaf node 11 involves the split nodes (0, 8, 9). The relevant connections for the corresponding calculation in the neural network are highlighted.

4.4.2 Random Forest Classification

Usually, neural networks perform very good in classification. However, if the data is limited, the large amount of parameters to be trained causes overfitting. Random forests [17] have shown to be robust classifiers even if few data are available. A random forest consists of multiple decision trees. Each decision tree uses a randomly selected subset of features and training data. The output is calculated by averaging the individual decision tree predictions.

After creating a feature generator, we calculate the feature vector $f^{(i)} = \text{CNN}_{relu7}^F(x^{(i)})$ for every input vector $x^{(i)}$. Based on the feature vectors we train a random forest that predicts the target values $y^{(i)}$. By combining the feature generator CNN^F and the random forest, we construct a classifier that predicts the class probabilities for an image patch. This classifier can be used to process a full input image patch-wisely. Calculating the class probabilities for each image patch produces an output probability map.

4.4.3 RF to NN Mapping

Here, we present a method for mapping random forests to two-hidden-layer neural networks introduced by Sethi [31] and Welbl [32]. The mapping is illustrated in Fig. 4.9. Decision trees have been introduced in Sect. 4.2.3.1. A decision tree consists of *split nodes* $\mathcal{N}^{\text{Split}}$ and *leaf*

nodes $\mathcal{N}^{\text{Leaf}}$. Each split node $s \in \mathcal{N}^{\text{Split}}$ evaluates a split decision and routes a data sample x to the left child node $\text{cl}(s)$ or to the right child node $\text{cr}(s)$ based on a split feature $f(s)$ and a threshold value $\theta(s)$:

$$x \in \text{cl}(s) \iff x_{f(s)} < \theta(s), \quad (4.30)$$

$$x \in \text{cr}(s) \iff x_{f(s)} \geq \theta(s). \quad (4.31)$$

The data sample x is routed to the left child node if the value of feature $f(s)$ of x is smaller than a threshold $\theta(s)$ and to the right child node otherwise. All leaf nodes $l \in \mathcal{N}^{\text{Leaf}}$ store votes for the classes $y^l = (y_1^l, \dots, y_C^l)$, where C is the number of classes. For each leaf a unique path $P(l) = (s_0, \dots, s_d)$ from root node s_0 to leaf l over a sequence of split nodes $\{s_i\}_{i=0}^d$ exists, with $l \subseteq s_d \subseteq \dots \subseteq s_0$. By evaluating the split rules for each split node along the path $P(l)$ the leaf membership can be expressed as

$$x \in l \iff \forall s \in P(l) : \begin{cases} x_{f(s)} < \theta(s) & \text{if } l \in \text{cl}(s), \\ x_{f(s)} \geq \theta(s) & \text{if } l \in \text{cr}(s). \end{cases} \quad (4.32)$$

First hidden layer. The first hidden layer computes all split decisions. It is constructed by creating one neuron $H_1(s)$ per split node evaluating the split decision $x_{f(s)} \geq \theta(s)$. The activation output of the neuron should approximate the following function:

$$a(H_1(s)) = \begin{cases} -1, & \text{if } x_{f(s)} < \theta(s), \\ +1, & \text{if } x_{f(s)} \geq \theta(s), \end{cases} \quad (4.33)$$

where -1 encodes a routing to the left child node and $+1$ a routing to the right child node. Therefore the $f(s)$ th neuron of the input layer is connected to $H_1(s)$ with weight $w_{f(s), H_1(s)} = c_{\text{split}}$, where c_{split} is a constant. The bias of $H_1(s)$ is set to $b_{H_1(s)} = -c_{\text{split}} \cdot \theta(s)$. All other weights and biases are zero. As a result, the neuron $H_1(s)$ calculates the weighted sum,

$$z_{H_1(s)} = c_{\text{split}} \cdot x_{f(s)} - c_{\text{split}} \cdot \theta(s), \quad (4.34)$$

which is smaller than zero when $x_{f(s)} < \theta(s)$ is fulfilled and greater than or equal to zero otherwise. The activation function $a(\cdot) = \tanh(\cdot)$ is used which maps the weighted sum to a value between -1 and $+1$ according to the routing. The constant c_{split} controls the sharpness of the transition from -1 to $+1$.

Second hidden layer. The second hidden layer combines the split decisions from layer H_1 to indicate the leaf membership $x \in l$. One leaf neuron $H_2(l)$ is created per leaf node. It is connected to all split neurons $H_1(s)$ along the path $s \in P(l)$ as follows:

$$w_{H_1(s), H_2(l)} = \begin{cases} -c_{\text{leaf}} & \text{if } l \in \text{cl}(s), \\ +c_{\text{leaf}} & \text{if } l \in \text{cr}(s), \end{cases} \quad (4.35)$$

where c_{leaf} is a constant. The weights are sign matched according to the routing directions, i.e. negative when l is in the left subtree from s and positive otherwise. Thus, the activation of $H_2(l)$ is maximized when all split decisions routing to l are satisfied. All other weights are zero. To encode the leaf to which a data sample x is routed, the bias is set to

$$b_{H_2(l)} = -c_{\text{leaf}} \cdot (|P(l)| - 1), \quad (4.36)$$

so that the weighted sum of neuron $H_2(l)$ will be greater than zero when all split decisions along the path are satisfied and less than zero otherwise. By using the activation function $a(\cdot) = \text{sigmoid}(\cdot)$, the active neuron $H_2(l)$ with $x \in l$ will map close to 1 and all other neurons close to 0. Similar to c_{split} , a large value for c_{leaf} approximates a step function, whereas smaller values can relax the tree hardness.

Output layer. The output layer contains one neuron $H_3(c)$ for each class and is fully-connected to the previous layer H_2 . Each neuron $H_2(l)$ indicates whether $x \in l$. The corresponding leaf node l in the decision tree stores the class votes y_c^l for each class c . To transfer the voting system, the weights are set proportional to the class votes:

$$w_{H_2(l), H_3(c)} = c_{\text{output}} \cdot y_c^l, \quad (4.37)$$

where c_{output} is a scaling constant to normalize the votes as explained in the following section. All biases are set to zero.

Random forest. Extending the mapping to random forests with T decision trees is simply done by mapping each decision tree and concatenating the neurons of the constructed neural networks for each layer. The neurons for each class in the output layer are created only once. They are fully-connected to the previous layer and by setting the constant c_{output} to $1/T$ the outputs of all trees are averaged. We denote the resulting neural network as NN^{RF} . It should be noted that the memory size of the mapped neural network grows linearly with the total number of split and leaf nodes. A possible network splitting strategy for very large random forests has been presented by Massiceti et al. [33].

4.4.4 Fully Convolutional Network

Mapping the random forest to a neural network allows one to join the feature generator and the classifier. For that we remove the classification layers from CNN^{F} , i.e. all layers after *relu7*, and append all layers from NN^{RF} . The constructed network $\text{CNN}^{\text{F+RF}}$ processes an image patch and outputs the class probabilities. The convolutional neural network $\text{CNN}^{\text{F+RF}}$ is converted to a fully convolutional network CNN^{FCN} by converting the fully-connected layers into convolutional layers, similar to [36]. The fully convolutional network operates on

input images of any size and produces corresponding (possibly scaled) output maps. Compared with patch-wise processing, the classifier is naturally slid over the image evaluating the class probabilities at any position. At the same time the features are shared so that features in overlapping patches can be reused. This decreases the amount of computation and significantly accelerates the processing of full images.

4.4.5 Bounding Box Prediction

The constructed fully convolutional network processes a color image $I \in \mathbb{R}^{W \times H \times 3}$ of size $W \times H$ with three color channels and produces an output $O = \text{CNN}^{\text{FCN}}(I)$ with $O \in \mathbb{R}^{W' \times H' \times C}$. The output consists of C -dimensional vectors at any position which indicate the probabilities for each class. Due to stride and padding parameters, the size of the output map can be decreased. To detect objects of different sizes, we process the input image in multiple scales $S = \{s_1, \dots, s_m\}$.

We extract potential object bounding boxes by identifying all positions in the output maps where the probability is greater than a minimal threshold $t_{\min} = 0.2$. We describe a bounding box by

$$b = (b_x, b_y, b_w, b_h, b_c, b_s), \quad (4.38)$$

where (b_x, b_y) is the position of the center, $b_w \times b_h$ the size, b_c the class, and b_s the score. The bounding box size corresponds to the field of view which is equal to the size of a single image patch. All values are scaled according to the scale factor. The score b_s is equal to the probability in the output map.

For determining the final bounding boxes, we process the following three steps. First, we apply non-maximum suppression on the set of bounding boxes for each class to make the system more robust and accelerate the next steps. For that we iteratively select the bounding box with the maximum score and remove all overlapping bounding boxes. Second, traffic signs are special classes since the subject of one traffic sign can be included similarly in another traffic sign as illustrated in Fig. 4.10. We utilize this information by defining a list of parts that can occur in each class. A part is found when a bounding box b' with the corresponding class and an intersection over union (IoU) greater than 0.2 exists. If this is the case, we increase the score by $b'_s \cdot 0.2/P$, where P is the number of parts. Third, we perform non-maximum suppression on the set of all bounding boxes by iteratively selecting the bounding box with the maximum score and removing all bounding boxes with $\text{IoU} > 0.5$. The final predictions are determined by selecting all bounding boxes that have a score b_s greater than or equal to a threshold t_c for the corresponding class.

Figure 4.10 : The subject from class 237 (A) occurs similarly in class 244.1 (B) and class 241 (C). Due to very few training examples and the consequent low variability, parts of traffic signs are recognized. We utilize this information and integrate the recognition of parts into the bounding box prediction.

Figure 4.11 : The detections are projected to the map by integrating additional data. Based on the position $(i_{\text{lat}}, i_{\text{lon}})$ and heading i_h of the image, the position $(t_{\text{lat}}, t_{\text{lon}})$ and heading t_h of the traffic sign are determined. To approximate the geoinformation depending on the position and size of the bounding box, the relative heading Δt_h (green) and distance t_d (blue) between the image and traffic sign are calculated.

4.5 Localization

In this process we integrate additional data from other sensors to determine the position and heading of the traffic signs. For localization of the traffic signs, we use the GPS-position $(i_{\text{lat}}, i_{\text{lon}})$ and heading i_h of the images. The heading is the direction to which a vehicle is pointing. The data is included in our dataset, which is described in detail in Sect. 4.7. As illustrated in Fig. 4.11, we transform each bounding box

$$b = (b_x, b_y, b_w, b_h, b_c, b_s) \quad (4.39)$$

to a traffic sign

$$t = (t_{\text{lat}}, t_{\text{lon}}, t_h, t_c), \quad (4.40)$$

where $(t_{\text{lat}}, t_{\text{lon}})$ is the position, t_h the heading, and t_c the class. Since the position and viewing direction of the image are known, we approximate the traffic sign position and heading by calculating the relative heading Δt_h and distance t_d .

The relative heading is based on the horizontal position b_x of the bounding box in the image. A traffic sign which is located directly in the center of the image has the same heading than the image. A traffic sign on the left or right border has a relative heading of half of the angle of view. To determine the relative heading, we calculate the horizontal offset to the center of the image normalized by the image width i_w . Additionally, we multiply the value by the estimated angle of view α_{aov} . Thereby, the relative heading is calculated by

$$\Delta t_h = \alpha_{\text{aov}} \cdot \left(\frac{b_x}{i_w} - 0.5 \right). \quad (4.41)$$

The distance t_d between the position of the image and the position of the traffic sign is approximated by estimating the depth of the bounding box in the image. Traffic signs have a defined size $t_w \times t_{ht}$, where t_w is the width and t_{ht} the height. Since an approximate depth estimation is sufficient, we use the information about the size and assume a simple pinhole camera model. Given the focal length f and the sensor width s_w of the camera obtained from the data sheet and a bounding box with width b_w , we calculate the approximated distance by

$$t_d = f \cdot \frac{t_w \cdot i_w}{b_w \cdot s_w}. \quad (4.42)$$

Lastly, a traffic sign $t = (t_{\text{lat}}, t_{\text{lon}}, t_h, t_c)$ is generated. The class t_c equals the bounding box class and the heading is calculated by adding the relative heading to the heading of the image $t_h = i_h + \Delta t_h$. The traffic sign position $(t_{\text{lat}}, t_{\text{lon}})$ is determined by moving the position of the image by t_d in the direction t_h .

4.6 Clustering

Traffic signs can be observed multiple times in different images, cf. Fig. 4.12. In this section, an approach for merging multiple observations of the same traffic sign is presented which makes the localization more robust and improves the localization accuracy. For that, the generated geoinformation is used and the unsupervised clustering algorithm mean shift [37] is applied. A mean shift locates maxima of a density function and operates without predefining the number of clusters. Supervised learning algorithms are not applicable because no labels exist.

The mean shift is applied to the set of traffic signs $T_c = \{t^{(1)}, t^{(2)}, \dots, t^{(N)}\}$ for each class c separately. As the traffic sign class is the same within the set, it can be omitted for clustering

Figure 4.12 : Multiple observations of the same traffic sign are grouped based on their position and heading (left: detections in images, right: positions on map). Because neither the labels nor the number of clusters are known, mean shift clustering is used for processing the detected traffic signs.

so that each traffic sign has a three-dimensional data vector $t^{(i)} = (t_{\text{lat}}^{(i)}, t_{\text{lon}}^{(i)}, t_{\text{h}}^{(i)})$ consisting of latitude, longitude, and heading. The mean shift algorithm is extended by introducing a general function $D(\cdot)$ to calculate the difference between two data points. This enables the processing of non-linear values and residue class groups such as for example angles. For the application in this work, the following difference function is used to calculate the difference between two traffic signs $t^{(a)}$ and $t^{(b)}$:

$$D(t^{(a)}, t^{(b)}) = \left(t_{\text{lat}}^{(a)} - t_{\text{lat}}^{(b)}, t_{\text{lon}}^{(a)} - t_{\text{lon}}^{(b)}, D_{\text{h}}(t_{\text{h}}^{(a)}, t_{\text{h}}^{(b)}) \right). \quad (4.43)$$

Latitude and longitude are subtracted, whereas the difference between the headings is defined by $D_{\text{h}}(\cdot)$. The function $D_{\text{h}}(\cdot)$ subtracts two headings α and β which lie within the interval $[-180, 180]$ and this ensures that the difference lies within the same interval:

$$D_{\text{h}}(\alpha, \beta) = \begin{cases} \alpha - \beta - 360 & \text{if } \alpha - \beta > +180, \\ \alpha - \beta + 360 & \text{if } \alpha - \beta < -180, \\ \alpha - \beta & \text{otherwise.} \end{cases} \quad (4.44)$$

The mean shift algorithm is generalized by integrating the difference function $D(\cdot)$ into the function for calculating the mean shift $m(y^t)$:

Figure 4.13 : Examples from the captured dataset. For instance, separate bicycle lines for cyclists (A) and (B), roads that are prohibited for cars but free for cyclists (C), or roads that allow for cycling in both directions (D).

$$m(y^t) = \frac{\sum_{i=1}^N K\left(\left\|\frac{D(t^{(i)}, y^t)}{b}\right\|^2\right) D(t^{(i)}, y^t)}{\sum_{i=1}^N K\left(\left\|\frac{D(t^{(i)}, y^t)}{b}\right\|^2\right)}, \quad (4.45)$$

where $K(\cdot)$ is the kernel, b the bandwidth, and y^t the cluster position in iteration t . In this work, a multivariate Gaussian kernel $K(x) = \exp\left(-\frac{1}{2}x\right)$ is used with different bandwidths for each dimension $b = (b_{\text{lat}}, b_{\text{lon}}, b_h)$. The bandwidths b_{lat} and b_{lon} are set to 0.00015 which equals approximately 10 meters for our geographic location and b_h to 30 degree.

The mean shift iteratively updates the cluster position by calculating the weighted mean shift. In each iteration the cluster position is translated by $m(y^t)$ so that the updated cluster position y^{t+1} is calculated by $y^{t+1} = y^t + m(y^t)$. A cluster is initialized at each data point. As a result, multiple instances of the same traffic sign are merged based on their position and heading.

4.7 Dataset

To collect data in real-world environments, smart phones are used for data recording because they can be readily attached to bicycles. Many people own a smart phone so that a large number of users can be involved. The recorded dataset consists of more than 60 000 images. Some examples are shown in Fig. 4.13.

4.7.1 Data Capturing

We developed an app for data recording which can be installed onto the smart phone. Using a bicycle mount, the smart phone is attached to the bike oriented in the direction of travel. While cycling, the app captures images and data from multiple sensors. Images of size 1080×1920 pixels are taken with a rate of one image per second. Sensor data is recorded from the built-in accelerometer, gyroscope, and magnetometer with a rate of ten data points per second. Furthermore, geoinformation is added using GPS. The data is recorded as often as the GPS-data is updated.

4.7.2 Filtering

After finishing a tour, the images are filtered to reduce the amount of data. Especially monotonous routes, e.g. in rural areas, produce many similar images. However, the rate with which images are captured cannot be reduced because this increases the risk of missing interesting situations.

We therefore introduce an adaptive filtering of the images. The objective is to keep images of potentially interesting situations that help to analyze traffic situations, but to remove redundant images. For instance, interesting situations could be changes in direction, traffic jams, bad road conditions, or obstructions like construction works or other road users. For filtering, we integrate motion information and apply a twofold filtering strategy based on decreases in speed and acceleration:

- i. **Decreases in speed** indicate situations where the cyclist has to slow down because of potential traffic obstructions such as for example traffic jams, construction works, or other road users. Speed is provided by the GPS-data. We apply a derivative filter to detect decreases in speed. As filter, we use a derivative of Gaussian filter with a bandwidth, i.e. standard deviation, of $2 \frac{\text{km}}{\text{h}^2}$.
- ii. **Acceleration** is used to analyze the road conditions and to detect for example bumps. It is specified per axis. Each data point consists of a three-dimensional vector. We calculate the Euclidean norm of the vector and apply two smoothing filters with different time spans: One with a large and one with a short time span. Thus, we filter the noisy acceleration data and detect the situations in which the short-term average acceleration relative to the long-term average acceleration exceeds a threshold of k . For smoothing, we use Gaussian filters with bandwidths of 1.5 g and 10 g, with standard gravitational acceleration $g = 9.81 \frac{\text{m}}{\text{s}^2}$, and set $k = 2.8$.

We filter the images by removing images if none of the two criteria indicate an interesting situation. The filtering process reduces the amount of data by a factor of 5 on average. Subsequently, the data is transferred to a server.

Figure 4.14 : Number of training and test samples in each class. On average only 26 samples are available per class for each set.

4.8 Experiments

Experiments are conducted to demonstrate the performance of the recognition system. Due to the limited amount of labeled data, the pipeline is trained on patches and then extended to perform object detection. First, results are presented on the classification of a single patch. Afterwards, the recognition performance is illustrated. The comparison of patch-wise processing and fully convolutional processing of full images is shown in the end. Random forests are trained and tested on an Intel(R) Core(TM) i7-7820X CPU @ 3.60GHz, and neural networks on a NVIDIA GeForce GTX 1080 Ti using *Tensorflow* [38] and *Keras* [39]. The proposed system is programmed in Python.

4.8.1 Training and Test Data

Ten different traffic signs that are interesting for cyclists are selected. Because the signs differ from traffic signs for cars, the availability of labeled data is very limited. Some classes come with little labeled data but for some classes no labeled data is available. To have ground truth data of our classes for training and testing, we manually annotated 524 bounding boxes of traffic signs in the images. The data is split into a training set and a test set using a split ratio of 50/50. In Fig. 4.14, the number of samples per class is shown. The training data consists of 256 samples for all 10 classes which corresponds to less than 26 samples per class on average. Please note that some traffic signs are very rare. Class 1000-32, for example, has only five

examples for training. Additionally, 2000 background examples are randomly sampled for training and testing. The splitting is repeated multiple times and the results are averaged.

4.8.2 Classification

The first experiment evaluates the performance of the classification on patches. The evaluation is performed in two steps. First, the training for learning features is examined and, secondly, the classification on the target task.

For feature learning, the GTSRB [1] dataset is used since it is similar to our task and has a large amount of labeled data. The dataset consists of 39 209 examples for training and 12 630 examples for testing over 43 classes. After training, the convolutional neural network CNN^F achieves an accuracy of 97.0% on the test set.

In the next step, the learned features are used to generate a feature vector of each training example of our dataset, and then to train a random forest. For evaluation, the test data is processed similarly. A feature vector is generated for each example from the test set using the learned feature generator CNN^F and classified by the random forest subsequently.

Since the class distribution is imbalanced, we report the overall accuracy and the mean accuracy. The mean accuracy calculates the precision for each class independently and averages the results. The random forest classification achieves an accuracy of 96.8% and a mean accuracy of 94.8% on the test set. The confusion matrix is shown in Fig. 4.15. Six classes are classified without errors. All other classes, except from the background class, only contain a single, two, or three misclassified examples. Class 1000-32 which consist of five examples has larger error. Additionally, some background examples are classified as traffic signs and vice versa. Please refer to Fig. 4.15 for more information about the traffic signs the classes correspond to.

4.8.3 Object Detection

The next experiment is conducted to demonstrate the recognition performance of the proposed system. The task is to detect the position, size, and type of all traffic signs in an image. The images have a high diversity with respect to different perspectives, different lighting conditions, and motion blur.

The recognition system is constructed by extending the CNN for patch-wise classification to a fully convolutional network so that fast processing of full images is enabled. A filtering strategy is applied subsequently to predict bounding boxes. No additional training data is required

Figure 4.15 : Confusion matrix showing the performance of the classifier on the test set. The absolute number of samples are shown in the matrix.

during this process so that only 256 examples over 10 classes are used for training the recognition system. We process the images in eight different scales. Starting with the scale $s_0 = 1$, the image size is decreased from scale to scale by a factor of 1.3.

To evaluate the recognition performance, we process all images in the test set and match the predicted bounding boxes with the ground truth data. Each estimated bounding box is assigned to the ground truth bounding box with the highest overlap. The overlap is measured using the IoU and only overlaps with an $\text{IoU} > 0.5$ are considered.

All bounding boxes come with a score and the class specific threshold t_c determines if a bounding box is accepted or rejected as described in Sect. 4.4.5. For each class, the threshold t_c is varied and precision and recall are calculated. Precision measures the accuracy of the predictions i.e. how many of the predicted objects are correct. It is defined as

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \quad (4.46)$$

where TP is the number of true positives and FP the number of false positives. Recall measures the amount of objects that are found relative to the total number of objects that are

Figure 4.16 : Precision-recall curves for evaluating the recognition performance. (A) Standard traffic signs. (B) Info signs. The shape of the curves is erratic because few labeled data is available for training and testing.

Figure 4.17 : Selected failure cases for class 267.

available and is defined as follows:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \quad (4.47)$$

where TP is the number of true positives and FN the number of false negatives. The resulting precision-recall curves are shown in Fig. 4.16. To facilitate understanding these results, two graphs are shown. In the first, the precision-recall curves of a group of standard traffic signs are plotted. The results are good. Some classes are detected almost perfectly. In the second graph, the precision-recall curves of a different group of traffic signs are plotted. These signs are much more difficult to recognize as they are black and white and do not have a conspicuous color. The performance of each class correlates with the number of examples that are available for training. Class 9001 with 35 training examples performs best, class 1022-10 with 22 training examples second best, and class 1000-32 with only 5 training examples worst. In Fig. 4.17 failure cases for class 267 are shown. Patches with similar appearance are extracted due to the limited variability with few training samples and missing semantic information since the broader context is not seen from the patch-wise classifier. To summarize the performance on each class the average precision (AP) is calculated. The results are presented in

Table 4.1: Average precision of each class on the test dataset.

class	237	239	240	241	242.1	244.1	267	1000-32	1022-10	9001
AP	0.901	0.930	0.964	0.944	0.801	0.996	0.677	0.023	0.215	0.679

Table 4.1. In total, the recognition system achieves a good mean average precision (mAP) of 0.713.

In the last step, the final bounding box predictions are determined. The threshold t_c of each class is selected by calculating the F1 score,

$$F1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}, \quad (4.48)$$

for each precision-recall pair and choosing the threshold with the maximum F1 score. Some qualitative results are presented in Fig. 4.18. In each column, examples of a particular class are chosen at random. Examples that are recognized correctly are shown in the first three rows, examples which are recognized as traffic sign but in fact belong to the background or a different class are shown in the next two rows. These bounding box patches can have a similar color or structure. Examples that are not recognized are shown in the last two rows on the bottom. Some of these examples are twisted or covered by stickers.

4.8.4 Computation Time

In the third experiment we evaluate the computation time. Random forests are fast at test time for the classification of a single feature vector. When processing a full image, the random forest is applied to every patch in the feature maps. For an image of size 1080×1920 the feature maps are produced relatively fast using CNN^F and have a size of 268×478 so that 124 399 patches have to be classified to build the output probability map. The images are processed in eight different scales. All together, we measured an average processing time of more than 10 hours for a single image. Although the computation time could be reduced by using a more efficient language than Python, the time to access the memory represents a bottleneck due to a large overhead for accessing and preprocessing each patch.

For processing all in one pipeline, we constructed the fully convolutional network CNN^{FCN}. The network combines feature generation and classification and processes full images in one pass. The time for processing one image in eight different scales is only 4.91 seconds on average. Compared with the patch-wise processing using random forest, using the fully convolutional network reduces the processing time significantly.

Figure 4.18 : Recognition results for randomly chosen examples of the test set. In each column, the ground truth traffic sign is shown on top along with correctly recognized traffic signs (first three rows), false positives (next two rows), and false negatives (last two rows on the bottom). Some classes have less than two false positives or false negatives, respectively, however.

4.8.5 Precision of Localizations

The last experiment is designed to demonstrate the localization performance. The localization maps the predicted bounding boxes in the image to positions on the map. Position and heading of a traffic sign are calculated based on the geoinformation of the image and the position and size of the bounding boxes.

For evaluation, we generate ground truth data by manually labeling all traffic signs on the map that are used in our dataset. In the next step, correctly detected traffic signs are matched with the ground truth data. The distance between two GPS-positions is calculated using the *haversine formula* [40]. The maximal possible difference of the heading is 90° because larger differences would show a traffic sign from the side or from the back. Each traffic sign is assigned to the ground truth traffic sign that has the minimum distance and a heading difference within the possible viewing area of 90° . The median of the localization error, i.e. the distance between the estimated position of the traffic sign and its ground truth position, is 6.79 m. Since the recorded GPS-data also includes the inaccuracies of each GPS-position, we can

Figure 4.19 : The distance error with respect to GPS-inaccuracy (A) and distance between the recording device and the traffic sign (B). The black lines indicate the medians, the upper and bottom ends of the blue boxes the first and third quantile.

remove traffic signs which are estimated by more inaccurate GPS-positions. If traffic signs with a GPS-inaccuracy larger than the average of 3.79 m are removed, then the median of the localization error decreases to 6.44 m.

The errors of the localizations (y-axis) with respect to the GPS-inaccuracies (x-axis) are plotted in Fig. 4.19A. The orange dots indicate estimated positions of traffic signs. The black lines indicate the medians, the upper and bottom ends of the blue boxes the first and third quartiles. It can be seen that the localization error does not depend on the precision of the GPS-position as it does not increase with the latter. The localization errors (y-axis) with respect to the distances between the positions of the traffic signs and the GPS-positions (x-axis) are shown in Fig. 4.19B. It can be seen that the errors depend on the distance between traffic sign and bicycle as they increase with these distances. This can be explained by the fact that the original inaccuracies of the GPS-position are extrapolated, i.e. the larger the distances, the more the GPS-inaccuracies perturb the localizations.

Since smart phones are used as recording devices, the precision of the GPS-coordinates is lower than those used in GPS-sensors integrated in cars or in high-end devices. As the inaccuracies of the GPS-positions have a large influence on the localizations, we identify multiple observations of the same traffic sign in a clustering process to reduce the localization error. The performance is evaluated as before. The overall median of the localization error improves to 5.75 m. When measuring the performance for all traffic signs which have multiple observations the median of the localization error even decreases to 5.13 m.

4.9 Conclusion

We presented a system for object detection that is trained with very little labeled data. CNNs have shown great results in feature learning and random forests are able to build a robust classifier even if very few training examples are available. We combined the advantages of CNNs and random forests to construct a fully convolutional network for predicting bounding boxes. The system is built in three steps. First, we learned features using a CNN and trained a random forest to perform patch-wise classification. Second, the random forest is mapped to a neural network. Lastly, we transform the pipeline to a fully convolutional network to accelerate the processing of full images. Whereas deep learning typically depends on the availability of large datasets, the proposed system significantly reduces the required amount of labeled data.

The proposed system was evaluated on crowdsourced data with the aim of collecting traffic information for cyclists. We used our system to recognize traffic signs that are relevant for cyclists. The system is trained with only 26 examples per class on average. Furthermore, we showed how additional sensor information can be used to locate traffic signs on the map.

Acknowledgment

This research was supported by German Research Foundation DFG within Priority Research Program 1894 Volunteered Geographic Information: Interpretation, Visualization and Social Computing.

References

- [1] J. Stallkamp, M. Schlipsing, J. Salmen, C. Igel, Man vs. computer: benchmarking machine learning algorithms for traffic sign recognition, *Neural Networks* 32 (2012) 323–332, <https://doi.org/10.1016/j.neunet.2012.02.016>.
- [2] A. Geiger, P. Lenz, C. Stiller, R. Urtasun, Vision meets robotics: the KITTI dataset, *The International Journal of Robotics Research* 32 (11) (2013) 1231–1237.
- [3] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, B. Schiele, The Cityscapes dataset for semantic urban scene understanding, in: *CVPR*, 2016.
- [4] C. Reinders, H. Ackermann, M.Y. Yang, B. Rosenhahn, Object recognition from very few training examples for enhancing bicycle maps, in: *2018 IEEE Intelligent Vehicles Symposium, IV*, 2018.
- [5] I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning*, MIT Press, 2016.
- [6] A. Krizhevsky, I. Sutskever, G.E. Hinton, Imagenet classification with deep convolutional neural networks, in: *NIPS*, 2012.
- [7] J. Zurada, *Introduction to Artificial Neural Systems*, Jaico Publishing House, 1994, <https://books.google.de/books?id=cMdjAAAACAAJ>.
- [8] W.S. McCulloch, W. Pitts, A logical calculus of the ideas immanent in nervous activity, *The Bulletin of Mathematical Biophysics* 5 (4) (1943) 115–133, <https://doi.org/10.1007/BF02478259>.
- [9] F. Rosenblatt, *The Perceptron, a Perceiving and Recognizing Automaton Project Para*, Report, Cornell Aeronautical Laboratory, Cornell Aeronautical Laboratory, 1957, https://books.google.de/books?id=P_XGPgAACAAJ.

- [10] P.J. Werbos, Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences, Ph.D. thesis, Harvard University, 1974.
- [11] D. Parker, Learning Logic: Casting the Cortex of the Human Brain in Silicon, Technical report, Center for Computational Research in Economics and Management Science, Massachusetts Institute of Technology, Center for Computational Research in Economics and Management Science, 1985, <https://books.google.de/books?id=2kS9GwAACAAJ>.
- [12] D.E. Rumelhart, J.L. McClelland, Parallel Distributed Processing: Explorations in the Microstructure of Cognition, vol. 1: Foundations, MIT Press, 1986.
- [13] D. Hubel, T. Wiesel, Receptive fields, binocular interaction, and functional architecture in the cat's visual cortex, *Journal of Physiology* 160 (1962) 106–154.
- [14] Y. Lecun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition, *Proceedings of the IEEE* 86 (11) (1998) 2278–2324, <https://doi.org/10.1109/5.726791>.
- [15] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, Dropout: a simple way to prevent neural networks from overfitting, *Journal of Machine Learning Research* 15 (1) (2014) 1929–1958, <http://dl.acm.org/citation.cfm?id=2627435.2670313>.
- [16] T.K. Ho, Random decision forests, in: *Proceedings of the Third International Conference on Document Analysis and Recognition*, vol. 1, ICDAR '95, IEEE Computer Society, Washington, DC, USA, 1995, p. 278, <http://dl.acm.org/citation.cfm?id=844379.844681>.
- [17] L. Breiman, Random forests, *Machine Learning* 45 (1) (2001) 5–32, <https://doi.org/10.1023/A:1010933404324>.
- [18] F. Kluger, H. Ackermann, M.Y. Yang, B. Rosenhahn, Deep learning for vanishing point detection using an inverse gnomonic projection, in: *GCPR*, 2017.
- [19] M.Y. Yang, W. Liao, H. Ackermann, B. Rosenhahn, On support relations and semantic scene graphs, *ISPRS Journal of Photogrammetry and Remote Sensing* 131 (2017) 15–25.
- [20] R. Girshick, J. Donahue, T. Darrell, J. Malik, Rich feature hierarchies for accurate object detection and semantic segmentation, in: *Computer Vision and Pattern Recognition*, 2014.
- [21] K. He, X. Zhang, S. Ren, J. Sun, Spatial pyramid pooling in deep convolutional networks for visual recognition, in: *European Conference on Computer Vision*, 2014.
- [22] R. Girshick, Fast R-CNN, in: *ICCV*, 2015.
- [23] S. Ren, K. He, R. Girshick, J. Sun, Faster R-CNN: towards real-time object detection with region proposal networks, in: *NIPS*, 2015.
- [24] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, A.C. Berg, SSD: single shot multibox detector, in: *ECCV*, 2016.
- [25] J. Redmon, A. Farhadi, Yolo9000: better, faster, stronger, arXiv preprint, arXiv:1612.08242.
- [26] T. Lin, P. Goyal, R.B. Girshick, K. He, P. Dollár, Focal loss for dense object detection, *CoRR*, arXiv:1708.02002, arXiv:1708.02002, <http://arxiv.org/abs/1708.02002>.
- [27] M. Oquab, L. Bottou, I. Laptev, J. Sivic, Learning and transferring mid-level image representations using convolutional neural networks, in: *CVPR*, 2014, pp. 1717–1724.
- [28] B. Moysset, C. Kermorant, C. Wolf, Learning to detect and localize many objects from few examples, *CoRR*, arXiv:1611.05664.
- [29] R. Wagner, M. Thom, H. Schweiger, G. Palm, A. Rothermel, Learning convolutional neural networks from few samples, in: *IJCNN*, 2013, pp. 1–7.
- [30] P. Kotschieder, M. Fiterau, A. Criminisi, S.R. Bulò, Deep neural decision forests, in: *ICCV*, 2015, pp. 1467–1475.
- [31] I.K. Sethi, Entropy nets: from decision trees to neural networks, *Proceedings of the IEEE* 78 (10) (1990) 1605–1613, <https://doi.org/10.1109/5.58346>.
- [32] J. Welbl, Casting random forests as artificial neural networks (and profiting from it), in: *GCPR*, Springer, Cham, 2014, pp. 765–771.
- [33] D. Massiceti, A. Krull, E. Brachmann, C. Rother, P.H.S. Torr, Random forests versus neural networks – what's best for camera localization?, in: *ICRA*, IEEE, 2017, pp. 5118–5125.

- [34] D.L. Richmond, D. Kainmueller, M.Y. Yang, E.W. Myers, C. Rother, Relating cascaded random forests to deep convolutional neural networks for semantic segmentation, CoRR, arXiv:1507.07583.
- [35] J.T. Springenberg, A. Dosovitskiy, T. Brox, M. Riedmiller, Striving for simplicity: the all convolutional net, in: ICLR, 2015, pp. 1–14, arXiv:1412.6806.
- [36] E. Shelhamer, J. Long, T. Darrell, Fully convolutional networks for semantic segmentation, IEEE Transactions on Pattern Analysis and Machine Intelligence 39 (4) (2017) 640–651, <https://doi.org/10.1109/TPAMI.2016.2572683>.
- [37] D. Comaniciu, P. Meer, Mean Shift: a robust approach toward feature space analysis, IEEE Transactions on Pattern Analysis and Machine Intelligence 24 (2002) 603–619.
- [38] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G.S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng, TensorFlow: large-scale machine learning on heterogeneous systems, software available from [tensorflow.org](https://www.tensorflow.org), 2015, <https://www.tensorflow.org/>.
- [39] F. Chollet, et al., Keras, <https://keras.io>, 2015.
- [40] J. Inman, Navigation and Nautical Astronomy, for the Use of British Seamen, F. & J. Rivington, 1849.