Real-Time Face Detection

Wolf Kienzle

Diplomarbeit

Real-Time Face Detection

Wolf Kienzle
Oktober 2003

Betreuer: Prof. Dr. Andreas Schilling [†] und Dr. Matthias Franz [‡].

[†)] Grafisch-Interaktive Systeme
Fakultät für Informations- und Kognitionswissenschaften
Wilhelm-Schickard-Institut für Informatik
Eberhard-Karls-Universität Tübingen

[‡)] Max Planck Institut für biologische Kybernetik
Abteilung Prof. Dr. Bernhard Schölkopf
Empirical Inference for Machine Learning and Perception

Erklärung

Mit der Abgabe der Diplomarbeit versichere ich, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Wolf Kienzle

Contents

Part I

Introduction

1.1 Motivation

Tracking humans in image sequences has many interesting applications, reaching from character animation to surveillance systems. The present work is part of the *imitation learning* project [28], a robot arm learns to imitate complex human arm movements. We try to restrict the learning input to images only, so that no special tracking hardware is needed. That way, our system has to be able to track shoulder, elbow and hand positions in image sequences. Using stereo images, we may then recover the joints' real world coordinates. Additionally, we want to automate the initialization process, that is, instead of manually specifying where each joint is located at system startup, we would like the system to determine the initial object positions by itself. As it is very hard to find regularities in hand, elbow or shoulder images, these objects are especially difficult to distinguish from the background. It is therefore helpful to incorporate prior information about the plausibility of certain joint positions, in other words, filter out false detections that lead to physically impossible arm configurations. This can be achieved as follows: we first detect the test person's face and then restrict the search areas for shoulder, elbow and hand accordingly. As the appearance of a human face is rather unique compared to that of other body parts, we expect face detection to be a reliable starting point. In fact, face detection is a thoroughly studied problem which has become a standard, relatively well-behaved application for classification algorithms.

The aim of this work is to implement an efficient face detection system. It should be designed such that computation times are close to real-time, allowing us to quickly re-initialize the tracking system if it should lose track. To this end, we first develop a preprocessing method that extracts skin colored image regions to which the face detector is then applied. The face detection method itself is based on a modified support vector machine classifier.

1.2 Previous Work

Face detection is a popular problem in image understanding. This is probably due to the large amount of applications and the relatively simple ways of collecting data. Moreover, from a more technical point of view, the dimensionality of the data is usually low enough for the problem to be tractable while its complexity is clearly non-trivial. This makes face detection a frequently used benchmark for classification methods and there are commonly used, almost standardized data sets (e.g. the CMU data set, as mentioned in [25]), allowing researchers to compare different methods. Two of the most important learning-based methods in face detection (or face recognition, face classification) are neural networks and support vector machines. In fact, there exists a close relationship between these approaches that will become clear in the second part of this text. In terms of error rates, the more recent support vector

machines have been shown to outperform neural networks in several fields. Neural networks in turn, tend to be computationally less complex and thus result in usually faster algorithms. Neural networks and support vector machines have been applied to the face detection problem for instance in [24, 25] and [22, 23, 26], respectively. In the course of these and related works, a number of image preprocessing techniques have evolved. They were designed to remove artifacts coming from occlusion, varying illumination and contrast.

While many famous face recognition systems operate on gray images, there is some work that incorporates color information as well (e.g. [12, 16]). Using color in this context usually boils down to separating skin colored pixels from non-skin colored pixels. This is often referred to as human skin color detection or skin color modelling [15, 18, 19]. What we have here is another classification problem, albeit only three dimensional and hence much easier to handle than face detection. A common technique is to approximate the skin color distribution by a normal or mixture of gaussians distribution [16, 20]. Color samples are then classified via Bayes' rule (against background colors) or merely empirical thresholds on the likelihood. There seems to be a great interest in the role of different color spaces. Most authors favor one of the commonly used color representations, RGB, HSV and YUV. In contrast, others (e.g. [13]) state that there is no significant difference between color classifiers based on different color spaces. In fact, we will show that this is the case in our environment.

1.3 Problem Setting

We use a stereo camera system as image source. It consists of two Basler CCD cameras, the left one yielding color images (A302fc) and the right one yielding monochrome images (A302fs). Both cameras are connected to a Windows XP machine via IEEE 1394 (firewire) interfaces. The software interface is a matlab library that implements various control functions as well as a simple snapshot routine. Within this work, we will not use disparity information and hence take snapshots from the left (color) camera only, where the video format will be 320x240 pixels. At this resolution the camera hardware is capable to provide 30 frames per second.

When recording arm movements, subjects are standing or sitting roughly three meters away from the camera. This distance comes from a tradeoff between a good depth resolution (if stereo information is used) and the constraint that the subject's head, torso and entire left arm are visible at any time. To collect the training and test data, we shot video clips (at approximately 10 frames/sec) of fifteen test persons from the lab. Each clip shows roughly 400 frames of one person moving his/her left arm, as if he/she was controlling the robot arm. After sorting out non-usable frames, we end up with a set of 5697 images, which we randomly divide into two sets of size 5197 and 500. In the following, we will refer to these sets as the

Figure 1.1: The training/testing data set. This figure shows two of the 5697 images in our database. Note that although we shut the blinds outside the lab windows (which can be seen in the back, on the upper right part of the images) in order to keep the illumination conditions constant, there are noticeable variations, complicating the classification process.

training and test images, respectively. Figure 1.1 shows two typical training/test images.

We found that even in indoor environments, it is impossible to create reproducible lighting conditions as long as daylight contributes to the overall illumination. This strongly decreases contrast and color constancy and thus imposes an additional difficulty on classification problems. In this work, we will not try to find a specific remedy; instead, we argue that training classifiers on this data will lead to functions that 'know' about the unwanted variations and 'ignore' them accordingly.

1.4 Thesis Overview

In the second part of this work, we give a formal definition of the classification problem and explain the concepts of learning and generalization within that context. We further introduce two important principles from statistical learning theory, namely empirical and structural risk minimization. Considering this theoretical background, we show how and why neural networks and support vector machines can learn to make correct decisions.

The third and fourth part describe our experiments in skin and face detection, respectively. We show how classifiers are trained, how they perform, and how we can deal with some difficulties that commonly arise during the design process of learning based methods. In particular, model selection and training techniques are examined, and the results are analyzed regarding the theoretical results from the preceding chapters.

Finally, in the fifth part, we discuss the results of our experiments and give an outlook to possible following work. We also have included a short appendix describing our software interface for those readers who are actually using the library.

Part II

Learning Theory

2.1 Pattern Classification

In pattern classification we try to assign predefined *class labels* to *patterns* that we observe. In most cases, patterns come as vectorial data that we obtain through some sort of measurement. The components of these vectors are called *features* and represent the properties which we use to make judgements about class memberships. Thus, a classifier can be seen as a function that maps feature vectors to class labels.

Throughout this text we will be dealing with *binary* classification problems, that is, two-class problems like separating skin colors from non-skin colors or face images from non-face images. In the latter case, for instance, our patterns will be image patches whose pixel intensities will serve as features. In order to tell if a patch contains a face we just plug in the corresponding feature vector into our classification function and check if the output says 'face' or 'non-face'.

The next section will illustrate why *learning* is important when dealing with such problems. We will then give an introduction to some basic concepts of statistical learning theory and examine two popular *induction principles*, namely *empirical* and *structural risk minimization*.

2.1.1 Why Learning?

The binary classification problem can be stated as follows: Given a set X of patterns, each of which is associated with a corresponding class label in $Y = \{-1, 1\}$, we want to find a mapping $f : X \to Y$ so that for any $\mathbf{x} \in X$ with label $y \in Y$,

$$f(\mathbf{x}) = y$$

that is, f predicts the correct label for each pattern. We call X input space, Y output space and f decision function. In our examples, we will always have $X \subseteq \mathbb{R}^n$.

How do we construct f? In a way, we want to write down a function that captures how y is determined by $\mathbf{x}$. If this feature-label relationship is sufficiently simple, it may readily be modelled - based only on our assumptions about the data. However, this is rarely the case. Most real-world classification problems (e.g. face detection) are too complex to make reasonable assumptions a priori. As a consequence, we have to look at the *data* itself and 'learn' f. We could, for instance, build a lookup table that holds the right label y for any pattern $\mathbf{x}$. Unfortunately, this turns out to be an impossible thing to do, since $X \times Y$ usually contains infinitely many elements and we only have finite time and space to determine and store f.

This leads us to idea of *inference*. We take only a *subset* $\subset X \times Y$ of so-called *training samples* and try to learn something about the regularities of the data. If there are any, that is,

if the feature-label relationship is not completely random, those samples will hopefully help us reveal some of this information. That way, we get an approximation to the 'true' f (the lookup table), that allows us to *infer* which class a previously unseen example $s \in (X \times Y) \setminus S$ belongs to.

Within that context, we call S the *training set*. When evaluating a classifier f, we test it on a set S' of unseen samples drawn from $(X \times Y) \setminus S$ (the *test set*) to see how well it *generalizes*, that is, to what extent it is able to use the learned information and make correct predictions. Generalization performance is a central issue in machine learning and its maximization will be one of our main goals.

2.1.2 Empirical Risk Minimization

In order to be able to compare different classifiers, we introduce a quality measure on decision functions. Assume we have learned some f from S. The error we make using f on $X \times Y$ is measured via a *loss* (or *cost*) function $c : X \times Y \times Y \to \mathbb{R}$. For a given $\mathbf{x} \in X$ with label $y \in Y$ and prediction $f(\mathbf{x}) \in Y$, the loss function $c(\mathbf{x}, y, f(\mathbf{x}))$ tells us how much we have to 'pay', if we believe that $y = f(\mathbf{x})$. In our examples, we will often use the so-called *zero-one-loss*

$$c(\mathbf{x}, y, f(\mathbf{x})) = \begin{cases} 0 & \text{if } y = f(x) \\ 1 & \text{otherwise} \end{cases}$$

which, summed over some set of examples, merely counts the number misclassifications. If we *average* a loss function over the training set, we get the *empirical risk* or *training error* R_{emp}. Let $S = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_m, y_m)\}$ be a training set of size m. Then

$$R_{emp}(f) = \frac{1}{m} \sum_{i=1}^{m} c(\mathbf{x}_i, y_i, f(\mathbf{x}_i)). \tag{2.1}$$

In the case of zero-one-loss, R_{emp} denotes the fraction of errors we make on the training set S. Similarly, if $S' = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{m'}\} \times \{y_1, y_2, \ldots, y_{m'}\}$ is our test set of size m', the *test error* (fraction of misclassifications on the test set) equals

$$R_{test}(f) = \frac{1}{m'} \sum_{i=1}^{m'} c(\mathbf{x}_i, y_i, f(\mathbf{x}_i)).$$

Note that these are not the quantities we are actually after - we want to classify well on $X \times Y$, not just on a training set S or some fixed test set S'. Instead, we look at the *expected value* of the test error. To this end, we make the assumption that all samples $(\mathbf{x}, y)$ are

independently drawn from some fixed (i.e. identical for all samples) probability distribution $P(\mathbf{x}, y)$ (often referred to as i.i.d. samples) to write:

$$R(f) = \mathbf{E}[R_{test}(f)] = \int_{X \times Y} c(x, y, f(\mathbf{x}))dP(\mathbf{x}, y),$$

where the value of $P(\mathbf{x}, y)$ equals the probability of observing an x with its true label being y. This quantity is called the *expected risk*. In our case, it denotes the fraction of misclassifications that we expect to make on some arbitrary test set $S \subset X$. In fact, it seems to be a suitable measure for choosing a 'good' f. If we can find some f that minimizes $R(f)$ for our problem, we have an optimal classifier.

But since our training set $S \subset X \times Y$ is only a sample of $X \times Y$, we cannot possibly know $P(\mathbf{x}, y)$ exactly. A common way to handle this is the following: We assume that there exists a density function $p(\mathbf{x}, y)$ for $P(\mathbf{x}, y)$ which allows us to rewrite the expected risk as

$$R(f) = \int_{X \times Y} c(\mathbf{x}, y, f(\mathbf{x}))p(\mathbf{x}, y)d\mathbf{x}dy. \tag{2.2}$$

Then we compute an estimate of $p(\mathbf{x}, y)$ on the training data S, namely the *empirical density*

$$p_{emp}(\mathbf{x}, y) = \frac{1}{m} \sum_{i=1}^{m} \delta(\mathbf{x} - \mathbf{x}_i)\delta(y - y_i)$$

where $\int \delta(x - \hat{x})f(x)dx = f(\hat{x})$. Now substituting $p_{emp}(\mathbf{x}, y)$ for $p(\mathbf{x}, y)$ in (2.2) leads to nothing else but the empirical risk $R_{emp}(f)$ (2.1) we have defined at the beginning of this section. So minimizing $R_{emp}(f)$ seems to be a reasonable approximation to minimizing $R(f)$, since, according to the law of large numbers, for any fixed f

$$R_{emp}(f) \to R(f)$$

as m grows to infinity. In other words, we might have found a tool for building a good classifier: Given the training set of labelled patterns, minimizing $R_{emp}(f)$ on these will yield an approximation to the optimal decision function. This technique is called *empirical risk minimization*. Unfortunately it has a major drawback: although it sounds quite promising at first, we will find that the underlying concept is generally too weak. In fact, empirical risk minimization alone can yield arbitrarily poor results (in terms of generalization). We will therefore have to extend it a little further.

2.1.3 Structural Risk Minimization

So far we have not said anything about properties of f other than the one that it should separate the two classes properly. Now suppose we are training a classifier on $S = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_m, y_m)\}$. Since there is no restriction on f, we can take our lookup table example (2.3), that is, make f return the right labels for all samples $s \in S$, which yields $R_{emp}(f) = 0$. But a lookup table does not contain information about samples not in the training set, so we cannot do any better than making f return an arbitrary value $\hat{y}$ (that is, $P(\hat{y} = 1) = P(\hat{y} = 0) = 0.5$) on samples from $(X \times Y) \setminus S$, for instance

$$f(\mathbf{x}) = \begin{cases} y_i & \text{if } \exists\ \mathbf{x}_i \in S, \text{ so that } \mathbf{x} = \mathbf{x}_i \\ \hat{y} & \text{otherwise} \end{cases} . \tag{2.3}$$

This means that f, although the empirical risk is zero, does nothing but *random guessing* on any pattern not contained in the training set. In fact, one will rarely have to classify a pattern that is exactly the same as one of the training patterns and so it turns out that our classifier does not do *anything* useful, even though the empirical risk minimization criterion was *optimally* satisfied. This phenomenon can be explained by the so-called 'no-free-lunch' theorem: If we only look at the training data S, we have no reason to judge whether one f is better than another. This implies the need for *prior knowledge* about the data, otherwise generalization is impossible.

Now the question is: how do we incorporate prior knowledge? In any case, minimizing the empirical risk seems to be necessary: We have shown that $R_{emp}(f)$ is an approximation to the actual risk $R(f)$, so it better be small. On the other hand we have seen in the preceding example that this cannot be the whole story. What we are missing here is *how* $R_{emp}(f)$ converges to $R(f)$. For one *particular* f, we have $R_{emp}(f) \to R(f)$ (*point-wise* convergence). But this does not mean that if f minimizes the empirical risk, it minimizes the true risk as well. In more technical terms, we say that empirical risk minimization is not *consistent* (Figure 2.1).

A solution to this problem has been proposed by *Vapnik*, who showed that one-sided *uniform* convergence is a both necessary and sufficient condition for consistency: If we can somehow make sure that for all $\varepsilon > 0$

$$P(\sup_{f \in F}(R(f) - R_{emp}(f)) > \varepsilon) \to 0 \tag{2.4}$$

as $m \to \infty$, empirical risk minimization will be consistent. Note how allowing f to be *any* function will violate this condition: In our lookup table example (2.3), we have no errors on the training set, i.e. $R_{emp} = 0$, but $R(f) > 0$ due to random guessing. As a consequence, if we want to ensure consistency, we have to put constraints on the space of possible solutions. What remains is the question of how these constraints look like.

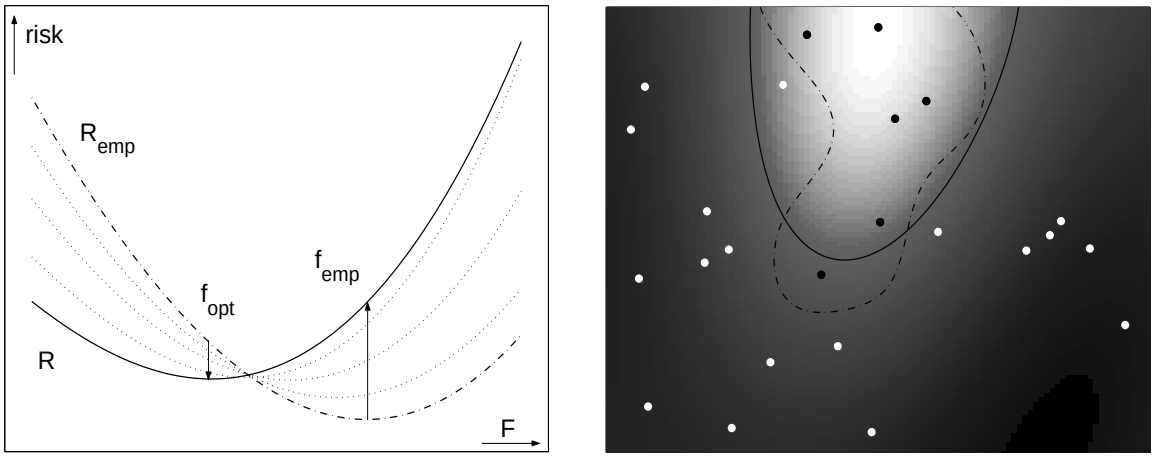

Figure 2.1: The problem with non-uniform convergence: On the left, we have plotted risk against function classes. The dash-dotted curve denotes the empirical risk on our training set. As we increase the number of examples, for any fixed f, $R_{emp}(f)$ converges point-wise to $R(f)$ (the solid curve) via the dotted curves, whereas the *minimum* moves to the left. Now empirical risk minimization tells us to choose f_{emp}, as it minimizes R_{emp}, but R is minimized by f_{opt}. The problem is: Not only does f_{emp} not coincide with the true optimum f_{opt}, but the expected risk $R(f_{emp})$ is also quite high. The right picture shows a toy problem where we try to separate withe dots from black dots in $X = \mathbb{R}^2$. The two decision boundaries (the place where f crosses zero, i.e. changes its sign) are the ones corresponding to f_{opt} (solid) and f_{emp} (dash-dotted), respectively. Note how the *true* optimal function has a *higher* training error on this training data than our estimated f_{emp}, while it yields (by definition) the highest possible generalization performance.

It turns out that the key is simplicity: If we want maximal generalization performance, we have to pick the simplest function possible that explains the data. This concept is not new and has many names. In information theory for instance, the *minimum description length principle* states that the more we are able to compress some data, the more we know about its regularities. In other words, if we need very little information to represent a function f that separates our data correctly, we must have learned quite a lot.

Looking at our toy problem (Figure 2.1), we see that the dash-dotted decision boundary is too complex for the problem at hand. It winds around the optimal curve and literally *memorizes* the training data, which is not appropriate here, since the structure we want to recover has obviously a different scale. As a result, such an *overfit* classifier suffers from large *variance*. If we classify points near the true boundary, the result will be more or less random, since at this scale, the decision will solely depend on the choice of the training set.

Furthermore, the two points that disagree with the optimal decision boundary may have

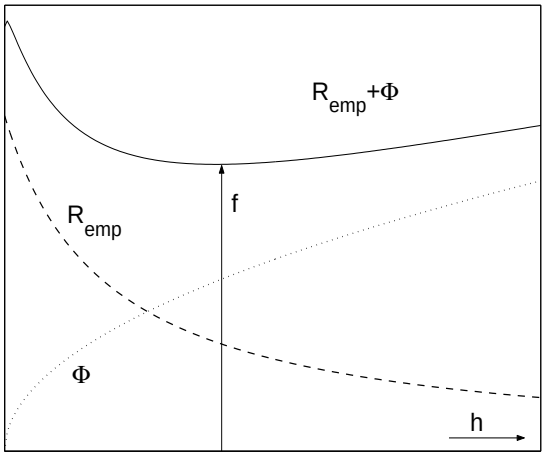

Figure 2.2: How structural risk minimization works: We have plotted the two right hand side terms of (2.5) against increasing function class complexity h. The dashed curve depicts the training error R_{emp}, which obviously approaches zero as we allow more complex functions. The dotted curve shows the behavior of the capacity term Φ, which grows roughly like $\sqrt{h}$. (2.5) tells us that the expected test error R is bounded by $R_{emp}+\Phi$, so a natural choice for f is the one minimizing that sum (solid curve). You may want to compare this to Figure 2.1.

been corrupted by noise or other measurement errors and are thus not representative. In contrast, the complex solution treats outliers as valid training data, yielding false classifications for all test samples in the vicinity of those points.

What exactly happens if we put this simplicity constraint on F? If we are too restrictive, none of the contained functions will be able to describe the data. The training error will be quite high and we will get an *underfit*, strongly *biased* (by our constraint) classifier. On the other hand, if the restrictions are too weak, the training error will be zero but the function will be overfitting the training data. To be able to deal with this effect mathematically, *Vapnik* incorporated the notion of function capacity into the consistency condition for empirical risk minimization (2.4) - a quite technical procedure which the interested reader can find for instance in [2]. He came to an impressive result, namely a bound on the expected risk

$$R(f) \leq R_{emp}(f) + \sqrt{\frac{h(\log \frac{2m}{h} + 1) - \log \frac{\delta}{4}}{m}} \tag{2.5}$$

which holds with probability of at least $1 - \delta$ for any $f \in F$ and $m > h$, where m is the number of training samples and h is a complexity measure on F called VC-dimension (after *Vapnik* and *Chervonenkis*). It equals the maximal number of points that the function class F can *shatter*. A function class is said to shatter k points if it is rich (complex) enough to implement any dichotomy (i.e. partition of the data into two classes) on any k points. By definition, a function class F with VC-dimension h can shatter any $r \leq h$ points, but also, we can find $h + 1$ points and a labelling $y_i \in \{-1, 1\}$ which cannot be predicted by any $f \in F$. An example: Linear functions in $\mathbb{R}^2$ (i.e. straight lines) can shatter any three points, but not any four points, so the VC-dimension of hyperplanes in $\mathbb{R}^2$ is three.

Now the benefit from (2.5) becomes clear: If we minimize the righthand side, we minimize the possible test error. This method is called *structural risk minimization* (the term 'structure' comes from the fact that h introduces a structure of nested subsets of increasing complexity on the function class). Note how this relates to the bias/variance tradeoff mentioned before: The bound comes as a sum of two competing functions: R_{emp} favors complex solutions, whereas the *capacity term* decreases as functions get simpler (Figure 2.2). This makes structural risk minimization a very powerful concept method in machine learning.

We are now equipped with two *induction principles*, namely empirical and structural risk minimization. In the following sections we will be more specific and show how they are actually implemented in popular *learning machines* such as neural networks and support vector machines.

2.2 Neural Networks

Neural Networks belong to the most widely used methods in machine learning. The concept is biologically motivated: We take a simple *neuron model* and plug several neurons together. That way, we try to simulate a learning environment similar to the brain. Even though the complexity of a neural network is extremely low compared to its biological counterpart, the performance is often quite acceptable.

We will briefly discuss how to formulate and train neural networks and see how they fit into the theoretical background we gave in the preceding section.

2.2.1 Neuron Model

Before we can build a neural network, we need an appropriate *neuron model*: A neuron has n scalar *inputs* $x_1, x_2, \ldots, x_n$ and one scalar *output* y. All n inputs are weighted by the so-called *synaptic weights* $w_i \in \mathbb{R}$ and summed up to an intermediate value v. We also keep a value $b \in \mathbb{R}$ for the *bias* of v:

$$v = \sum_{i=1}^{n} w_i x_i + b. \tag{2.6}$$

The output value y is computed using some *activation function* $\varphi : \mathbb{R} \to \mathbb{R}$, which is typically a threshold or sigmoid function. Here, we use the *tanh* function that clamps the output to $[-1; 1]$

$$y = \varphi(v) = \tanh\left(\sum_{i=1}^{n} w_i x_i + b\right). \tag{2.7}$$

As mentioned before, the idea of neural networks is to take these 'neurons' and plug them together, forming a network in which they interact with each other. The result can be viewed as a black box with some inputs and outputs. Its behavior is determined by the architecture (connectivity, type of activation function) of the underlying network and by its parameters (synaptic weights w_i and biases b_i). In its simplest form there is no interaction between different neurons (*single layer network*). We will now consider this easiest of all networks to show how setting the parameters, i.e training a neural network, works.

2.2.2 Training Neural Networks

Assume we have a single layer network consisting of just one neuron with n inputs plus some binary classification problem with a training set $S = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\}$. The network parameters are $\Theta = (w_1, w_2, \ldots, w_n, b)$. We denote the network output by $h(\mathbf{x})$ and say that a pattern $(\mathbf{x}, y)$ has been classified correctly, if $f(\mathbf{x}) = \operatorname{sgn}(h(\mathbf{x})) = y$. Training this network boils down to finding a parameter set $\hat{\Theta}$ which minimizes the training error on S and hence separates the training data best:

$$\hat{\Theta} = \arg\min_{\Theta} \sum_{i=1}^{m} c(\mathbf{x}_i, y_i, f(\mathbf{x}_i)). \tag{2.8}$$

Note that we are doing empirical risk minimization here. The term we are minimizing is basically R_{emp} ((2.1) without $1/m$), where f is parameterized by Θ. For the cost function c we have

$$c(\mathbf{x}, y, f(\mathbf{x})) = \frac{1}{2}|y - h(\mathbf{x})|$$

which can be viewed as a smooth version of the zero-one-loss: since h is smooth (tanh), the jump in the heaviside function is replaced by a differentiable transition.

To find $\hat{\Theta}$, we can do a simple *gradient descent*. This optimization problem (2.8) is *unconstrained* and we therefore do not have care about which values are valid for Θ and which are not. The method works as follows: We initialize $\Theta(0)$ with (preferably small) random values and present our training patterns to the network. Having ensured the differentiability of the risk function, we may compute its gradient with respect to Θ. At each iteration t, we evaluate

$$g(t) = \nabla \sum_{i=1}^{m} c(\mathbf{x}_i, y_i, f(\mathbf{x}_i))$$

which by definition equals the direction of *steepest ascend* of the risk in parameter space. Now we update our current parameter set $\Theta(t)$ by

$$\Theta(t+1) = \Theta(t) - \eta g(t),$$

which amounts to moving in the direction of steepest *descent* with a predefined step size η. Using a first-order Taylor series expansion, we can show that this reduces the empirical risk by $\Delta(t) \approx \eta \|g(t)\|^2$. We repeat this iteration until convergence, that is, until the progress $\Delta(t)$ falls below some ε.

The step size η is called *learning rate*. It must be chosen carefully, since it has significant impact on the behavior of the algorithm. This is already one drawback of our simple method. In fact, there exist a number of different methods for training neural networks, each one having its distinct advantages and problems. For a detailed discussion, see [3].

2.2.3 Multi-Layer Networks

In a neural network, the space of functions F from which we pick f is already limited by the fixed parameters, i.e. the network architecture plus the activation function φ. In particular, our single layer network can implement any linear function in $\mathbb{R}^n$, but linear functions are generally too simple and thus not suitable for real world problems.

If problems become more complex, we should increase F's capacity by adding more neurons. A very common architecture is the *layered, feedforward* model where groups of neurons can be viewed as layers between input and output nodes, i.e. the inputs of the first layer (*input layer*) are the inputs themselves, the outputs of the input layer become inputs of the second layer, and so on. The last layer is then called the *output layer*. We call all other layers *hidden*, since they are not directly accessible via the input/output ports.

When training the multi-layer network, we use a method called *backpropagation*, which is an efficient way to do gradient decent with feedforward networks. Basically, it implements the chain rule to calculate partial derivatives by propagating the error from the output layer back to the input layer. Again, for details see [3].

2.2.4 Discussion

Since the complexity of the available function class is encoded into the network architecture, it should embody our prior knowledge about the problem at hand (no free lunch). There are basically no restrictions on the network structure, allowing neural networks to approximate arbitrarily complex functions. This makes them a powerful and universal tool, but can also be disadvantageous: It is not clear at all which architecture solves which problem. In order to examine this within the learning theory framework, we may want to know its VC-dimension h and apply (2.5). In most cases the VC-dimension cannot be evaluated analytically. There are some *bounds* on h though, for instance

$$h \in O(|w|^2),$$

which holds for any multi-layer, feedforward network using a sigmoid activation function ($|w|$ denotes the number of free parameters). In practice, however, this bound is much too loose be beneficial.

Another problem is that the error surface implied by R_{emp} can have local minima. So the optimal solution we attain depends on the initial (random) parameter values and is not necessarily the global optimum. Moreover, there is the choice of the learning rate parameter η. If too small, the algorithm converges too slowly, if too large, it might not converge at all.

Despite these drawbacks, neural networks are very popular. The main reason is probably that, if we manage to build a network that is just sufficiently complex to solve the problem, there is almost no computational overhead at runtime, which results in very high classification speeds.

2.3 Support Vector Machines

The concept of *support vector machines* is a more recent one that contains several good ideas. As a main feature, support vector machines provide a mechanism for capacity control, in fact they implement structural risk minimization. Moreover, unlike in neural networks, the optimization procedure always yields a global optimum (the error surface has no local minima). A third advantage is the strong theoretical foundation that allows us to predict and control the behavior of the algorithm in a convenient way. To understand this, we take a look at some theory on linear classifiers.

2.3.1 Linear Classifiers

The power of support vector machine theory comes from the fact that they actually *are* linear classifiers. Recall that in binary classification, we seek a decision function f that separates the patterns $(\mathbf{x}, y) \in X \times Y$ into two classes via its value on $\mathbf{x}$. The data are said to be *linearly separable*, if this can be achieved with f being a linear function. Now if f is linear, it simply corresponds to a hyperplane H in X. The class membership of a point $\mathbf{x}$ is then determined by the side of H on which it lies. We can write linear functions in $\mathbb{R}^n$ as

$$H = \{\mathbf{x} \in \mathbb{R}^n, \quad \mathbf{w}^T\mathbf{x} + b = 0\},$$

where $\mathbf{w} \in \mathbb{R}^n$ and $b \in \mathbb{R}$. Note that scaling $\mathbf{w}$ and b by some nonzero value does not change H. This ambiguity can be easily removed, given some data. Consider a linearly separable data set $S = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_m, y_m)\} \subset \mathbb{R}^n \times \{-1, 1\}$. We define the *canonical hyperplane* as the parameter set $(\mathbf{w}, b)$ that satisfies

$$\min_{i=1\ldots m} y_i(\mathbf{w}^T\mathbf{x}_i + b) = 1,$$

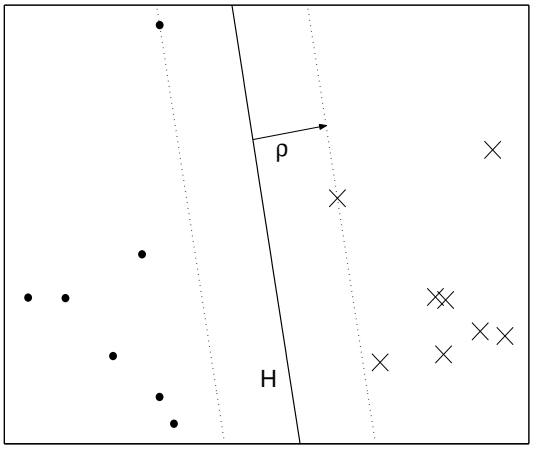

Figure 2.3: The margin of a hyperplane. H (solid line) separates the dots from the 'x's. The two data points (one dot, one 'x', lying on the dotted lines) that are closest to H define the margin ρ, that is, for the canonical hyperplane they satisfy $y(\mathbf{w}^T\mathbf{x} + b) = 1$.

in other words, the euclidian distance between the points which are closest to H and H itself must be equal to $1/\|\mathbf{w}\|$.

This also leads to the notion of the *margin*. The margin ρ of a hyperplane H is just its the minimum distance to any point in S (Figure 2.3). For the canonical hyperplane, this means that

$$\rho = \frac{1}{\|\mathbf{w}\|}.$$

Similarly, we say that the margin of an arbitrary point $(\mathbf{x}, y) \in X \times Y$ is defined as $y(\mathbf{w}^T\mathbf{x} + b)/\|\mathbf{w}\|$.

In fact, the margin plays an important role when it comes to generalization. A classifier with a large margin is expected to generalize better than classifier with a small margin. Intuitively, this can be explained in several ways [2]: Suppose we have a linear classifier H with a margin equal to ρ (Figure 2.3). What happens, if we add noise to the data points? As long as the noise displaces our points by less than ρ, the classifier will not get confused. So in a way, the margin measures the insensitivity to pattern noise, which means that we have learned quite a lot from the data if that value is large.

From the information theoretical viewpoint, we can relate this fact to the minimum description length principle. As mentioned above, the less information we need to represent the decision boundary, the more we know and the better we generalize. Suppose we *quantize* the representation of H. Although we obviously change the true H due to the quantization error, the classification result will not change if the margin is large enough.

Apart from these rather intuitive explanations there is a theorem by *Vapnik* that justifies the importance of the margin: Assume that the data lies within a sphere around the origin with radius r and that canonical hyperplane $H = \{\mathbf{x} \in \mathbb{R}^n, \ \mathbf{w}^T\mathbf{x} + b = 0\}$ has no bias,

i.e. $b = 0$ (this restriction is made only for simplicity and does not affect the general result). Then, the set of linear decision functions that have a margin of $\rho \geq 1/\Lambda$, has VC-dimension

$$h \leq r^2 \Lambda^2. \tag{2.9}$$

This means that if we have a set of linear functions that separate our data, all with a margin of at least $1/\Lambda$, its capacity is bounded from above via (2.9). According to the VC bound (2.5), it therefore seems advisable to maximize the margin if we want the expected risk to be as low as possible.

2.3.2 Training Support Vector Machines

We have seen that among all linear functions, we expect the best generalization performance from the so-called *maximal margin classifier*. Now assume we are given a linearly separable training example $S = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_m, y_m)\} \subset X \times Y = \mathbb{R}^n \times \{-1, 1\}$. By definition of the margin, we have $\rho = 1/\|\mathbf{w}\|$. So if we solve

$$\begin{array}{lll} \text{minimize} & \frac{1}{2}\|\mathbf{w}\|^2 & \text{w.r.t. } \mathbf{w}, b \\ \text{subject to} & y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 & i = 1 \ldots m \end{array} \tag{2.10}$$

we maximize the margin under the constraint that the data are separated by the hyperplane $H = \{\mathbf{x} \in \mathbb{R}^n, \quad \mathbf{w}^T\mathbf{x} + b = 0\}$. Note that this is structural risk minimization: The m *inequality constraints* ensure that the data are separated, in other words, $R_{emp} = 0$. At the same time, F is restricted to the space of linear functions, so minimizing the capacity term in (2.5) amounts to finding the simplest $f \in F$, where simplicity is encoded into the *target function* $\|\mathbf{w}\|^2/2$. The resulting decision function writes

$$f(\mathbf{x}) = \operatorname{sgn}(\mathbf{w}^T\mathbf{x} + b). \tag{2.11}$$

The optimization problem (2.10) has the nice property that both the target function and the domain from which we pick $\mathbf{w}$ and b (implied by the constraints) are *convex*. Therefore, it does *not* suffer from local minima. We will now show how to find the optimal solution efficiently by turning (2.10) into its so-called *dual* problem. The following derivations are based on optimization theoretical ideas, in particular *Lagrangian* theory and the *Kuhn-Tucker* theorem [1]. The *Lagrangian function* of our optimization problem (2.10) writes

$$L(\mathbf{w}, b, \boldsymbol{\alpha}) = \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{m} \alpha_i (y_i(\mathbf{w}^T\mathbf{x}_i + b) - 1),$$

where the $\alpha_i \geq 0$ are called *Lagrange multipliers*. Due to the convexity of our problem we may apply the Kuhn-Tucker theorem [1, 2], which tells us that the optimum is a saddle point, more precisely, a minimum with respect to $\mathbf{w}$, b and a maximum with respect to $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_m)$. It is therefore necessary and sufficient to assert that

$$\frac{\partial L(\mathbf{w}, b, \boldsymbol{\alpha})}{\partial w} = 0, \qquad \frac{\partial L(\mathbf{w}, b, \boldsymbol{\alpha})}{\partial b} = 0,$$

leading to

$$\mathbf{w} = \sum_{i=1}^{m} \alpha_i y_i \mathbf{x}_i, \qquad \sum_{i=1}^{m} \alpha_i y_i = 0,$$

respectively. Finally, substituting the result for $\mathbf{w}$ into the Lagrangian function yields the dual problem

$$\begin{array}{ll} \text{maximize} & \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j \mathbf{x}_i^T \mathbf{x}_j \quad \text{w.r.t. } \boldsymbol{\alpha}, \\ \text{subject to} & \sum_{i=1}^{m} \alpha_i y_i = 0 \\ & \alpha_i \geq 0, \quad i = 1 \ldots m \end{array} \tag{2.12}$$

By construction, the dual optimization problem (2.12) leads to the same result as (2.10). Recovering the primal variables $\mathbf{w}$ and b is straightforward: by construction, we know that $\mathbf{w} = \sum_{i=1}^{m} \alpha_i y_i \mathbf{x}_i$. From the primal constraints we have

$$b = -\frac{1}{2}(\max_{y_i=-1}(\mathbf{w}^T \mathbf{x}_i) + \min_{y_i=1}(\mathbf{w}^T \mathbf{x}_i)).$$

Solving the dual problem has two major advantages: Firstly, by virtue of the so-called *Kuhn-Tucker complementary conditions* [1, 2], we know that at the optimum, the equation $\alpha_i(y_i(\mathbf{w}^T \mathbf{x}_i + b) - 1) = 0$ holds for all points $\mathbf{x}_i$, $i = 1 \ldots m$. So for any i, α_i can only be nonzero if $\mathbf{x}_i$ satisfies the primal constraint by equality, meaning that $\mathbf{x}_i$ is one of the points *on* the margin (Figure 2.4). In terms of the optimization problem, we say that these are *active* constraints. Now on the other hand, points with $\alpha_i = 0$ do *not* contribute to $\mathbf{w}$. Since the solution only depends on points with active constraints, we call these points *support vectors*.

If the number of support vectors is small compared to m, we say that the solution is *sparse*. Sparsity is an appealing property since it reduces the computational complexity of

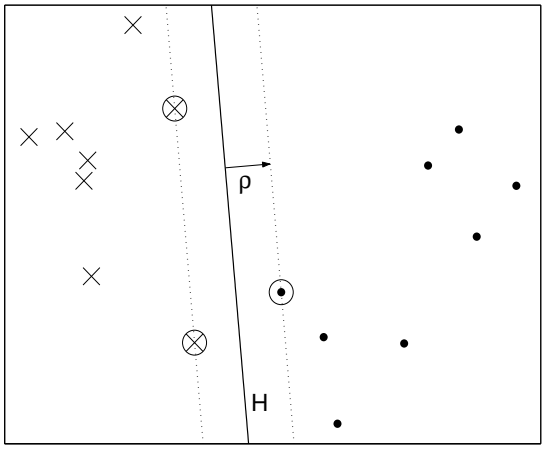

Figure 2.4: The maximum margin hyperplane. Like in Figure 2.3, we have a hyperplane H (solid line) separating the dots from the 'x's. But here, H is the hyperplane with the maximum margin possible. The encircled data points are the support vectors (active constraints). All other points have $\alpha_i = 0$ and are therefore 'useless' as they do not appear in the solution. From a geometrical point of view, this corresponds to the fact that perturbing those points by a small amount will not change H.

both finding the solution and applying the resulting decision function. Moreover, sparsity is motivated by the fact, that there exists a bound on the generalization error, depending on the fraction of support vectors [2].

As an aside, there is a nice interpretation of the α_i. The $\mathbf{x}_i$ with small corresponding α_i are *easy* points, they are clearly on the correct side of the hyperplane. In contrast, large α_i stand for *difficult* points, close to the decision boundary. From that viewpoint it makes perfect sense that support vector machines only use the difficult points in the decision function, as the maximum margin hyperplane is already defined by them alone.

The second advantage is based on *how* the algorithm (and the decision function) accesses the training (and testing) data, namely exclusively via *dot products*. We will show in the following section why this is a valuable property.

2.3.3 The Kernel Trick

Until now, we assumed that the data are linearly separable. This led us to the maximal margin classifier and its appealing statistical properties. Moreover, we have seen that such a classifier can be found by training a support vector machine. However, as mentioned before, real world problems are usually not linearly separable. The *kernel trick* allows us to extend the support vector machines idea to non-linear classification, while preserving all the advantageous properties we have seen so far.

Suppose we have solved the dual optimization problem. This gives us an *expansion* of $\mathbf{w}$

in terms of the support vectors, and we may rewrite the decision function (2.11) as

$$f(\mathbf{x}) = \operatorname{sgn}(\sum_{i=1}^{m} y_i \alpha_i \mathbf{x}_i^T \mathbf{x} + b). \tag{2.13}$$

We interpret (2.13) as follows: The decision function f *compares* the test pattern $\mathbf{x}$ to each support vector $\mathbf{x}_i$ and weights the respective *similarity* by α_i, where y_i provides the correct sign for this contribution (according to the side of the hyperplane that $\mathbf{x}_i$ lies on). In particular, f considers $\mathbf{x}$ to be more similar $\mathbf{x}_i$, the larger the value $\mathbf{x}_i^T \mathbf{x}$ gets. This means that for the linear case, the canonical dot product $\langle \cdot, \cdot \rangle$ provides the *similarity measure* on X.

Note that the optimization problem (2.12) uses the same notion of similarity. In fact, wherever an input space point occurs in our method, it will be within a dot product. This leads to the idea of the kernel trick, which amounts to substituting the dot product by a different similarity measure $k(\cdot, \cdot)$, called a *kernel*. It is motivated by the following consideration: Even if the data are not linearly separable in input space, it may well be possible to find a nonlinear function $\boldsymbol{\phi} : X \to F$ that maps input patterns to a so-called *feature space* F, where the problem *is* linearly separable.

Interestingly, it is not necessary to construct a specific mapping $\boldsymbol{\phi}$ for the problem at hand. We might as well use a generic kernel (e.g. the gaussian kernel, see below) as long as it allows the induced feature space to be high dimensional. This can be motivated by *Cover's* theorem [3]: It states that the probability of a classification problem being linearly separable increases with the dimensionality of the space F induced by our nonlinear mapping $\boldsymbol{\phi}$. Hence, if there exists k, such that

$$k(\mathbf{x}, \mathbf{z}) = \langle \boldsymbol{\phi}(\mathbf{x}), \boldsymbol{\phi}(\mathbf{z}) \rangle$$

we may *kernelize* our problem (2.12) by simply replacing any $\langle \cdot, \cdot \rangle$ by $k(\cdot, \cdot)$. That way, we implicitly solve our problem in F instead of X. Since we do not have to compute dot products in F explicitly, we can deal with arbitrarily high dimensional feature spaces at low computational cost. Moreover, if we work with off-the-shelf kernels, we avoid the difficulty of finding sophisticated feature maps.

Obviously, not every function $k(\cdot, \cdot)$ implies a dot product in some feature space. If we *have* a feature map $\boldsymbol{\phi}$, it is easy to write down a corresponding kernel. However, the opposite direction is much more convenient, that is, we take some $k(\cdot, \cdot)$ and *then* show that it implies a dot product in some high dimensional space F.

We will now discuss which conditions must hold for k to be a kernel. To this end, consider a function $k(\cdot, \cdot) : X \times X \to \mathbb{R}$ and a set of points in X, $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m\}$. The so-called *Gram Matrix* K with respect to these points has elements

$$K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j).$$

K is also referred to as the *kernel matrix*. The first thing we can say about K is that it must be *symmetric*, since dot products are. This, in turn, means that K must have real eigenvalues λ_i and m mutually orthogonal eigenvectors, whose normalized versions we denote by ψ_i. Now if we define a feature map

$$\boldsymbol{\phi}(\mathbf{x}_i) = (\sqrt{\lambda_1}\psi_{i,1}, \sqrt{\lambda_2}\psi_{i,2}, \ldots, \sqrt{\lambda_m}\psi_{i,m})^T$$

the dot product in feature space writes

$$\langle \boldsymbol{\phi}(\mathbf{x}_i), \boldsymbol{\phi}(\mathbf{x}_j) \rangle = \sum_{k=1}^{m} \lambda_k \psi_{i,k} \psi_{j,k} = (U^T \Lambda U)_{ij},$$

where U is an $m \times m$ matrix whose columns are the $\boldsymbol{\phi}_i$ and Λ is also $m \times m$, with the λ_i on the diagonal. But this is just the eigenvalue decomposition of K. Therefore,

$$\langle \boldsymbol{\phi}(\mathbf{x}_i), \boldsymbol{\phi}(\mathbf{x}_j) \rangle = k(\mathbf{x}_i, \mathbf{x}_j).$$

The second requirement for a dot product is *positive definiteness*. Without proof, this means that the kernel matrix K has to be positive definite as well, i.e. all its eigenvalues λ_i must be non-negative. As a result, the two required properties for k to serve as a kernel are symmetry and positive definiteness.

There is also contiguous version of this proposition, namely the famous *Mercer* theorem. Here, we have $X \subseteq \mathbb{R}^n$ and an integral operator $T_K : L_2(X) \to L_2(X)$ that can be expressed as

$$T_K(f)(\cdot) = \int_X k(\cdot, \mathbf{x}) f(\mathbf{x}) d\mathbf{x}.$$

If T_K is positive definite, i.e.

$$\int_{X \times X} k(\mathbf{x}, \mathbf{z}) f(\mathbf{x}) f(\mathbf{z}) d\mathbf{x} d\mathbf{z} \geq 0.$$

for all $f \in L_2(X)$, then k can be expanded in terms of the normalized eigenfunctions ψ_i of T_K and their (non-negative) eigenvalues λ_i by

$$k(\mathbf{x}, \mathbf{z}) = \sum_{i,j=1}^{\infty} \lambda_i \psi_i(\mathbf{x}) \psi_j(\mathbf{z})$$

which converges uniformly on $X \times X$. We may think of this as the generalization of our two kernel conditions. If we let f be a sum of delta functions centered on our training data, we can see the transition to the finite case.

There are a few kernels that have been proven particularly useful, for instance the so-called *gaussian radial basis function* kernel (which we have used to generate Figure 2.5)

$$k(\mathbf{x}, \mathbf{z}) = \exp(-\frac{1}{2\sigma^2}\|\mathbf{x} - \mathbf{z}\|^2). \tag{2.14}$$

The gaussian kernel has a single parameter σ, often referred to as the *kernel width*. It must be set to fit the problem at hand. We will see later how this can be done.

When using such kernels, we neither lose the convexity of the optimization problem (2.12), nor the maximum margin hyperplane concept. This is due to the fact that we are still computing maximum margin hyperplanes, albeit only implicitly. We are now ready to write down the kernelized dual optimization problem

$$\begin{aligned} &\text{maximize} && \textstyle\sum_{i=1}^m \alpha_i - \frac{1}{2}\sum_{i,j=1}^m y_i y_j \alpha_i \alpha_j k(\mathbf{x}_i, \mathbf{x}_j) \quad \text{w.r.t. } \boldsymbol{\alpha}, \\ &\text{subject to} && \textstyle\sum_{i=1}^m \alpha_i y_i = 0 \\ & && \alpha_i \geq 0, \quad i = 1 \dots m \end{aligned} \tag{2.15}$$

where the decision function becomes

$$f(\mathbf{x}) = \operatorname{sgn}(\sum_{i=1}^{m} y_i \alpha_i k(\mathbf{x}_i, \mathbf{x}) + b). \tag{2.16}$$

Note that the simple dot product $\langle \cdot, \cdot \rangle$ in X is also a kernel. This makes the linear support vector machine merely a special case of (2.15).

2.3.4 Soft Margins and Regularization

In connection with kernels, support vector machines are able to find nonlinear classifiers efficiently. This mechanism allows us to separate *any* set of points, if our kernel implies a feature map that is sufficiently complex to make the problem linearly separable. In fact, if it is not, there will be no feasible solution to our optimization problem (2.15). In that sense, the linear separability assumption disagrees with our preference for simple solutions: We *are* doing structural risk minimization (2.5), but due to the separability assumption we always assert that $R_{emp} = 0$. Thus, if the data are noisy (like in Figure 2.1), we will *have to* overfit if we want to solve (2.15).

Let us therefore introduce another extension to the maximum margin classifier, namely the notion of the *soft margin* (as opposed to the *hard margin*, which we have implicitly used

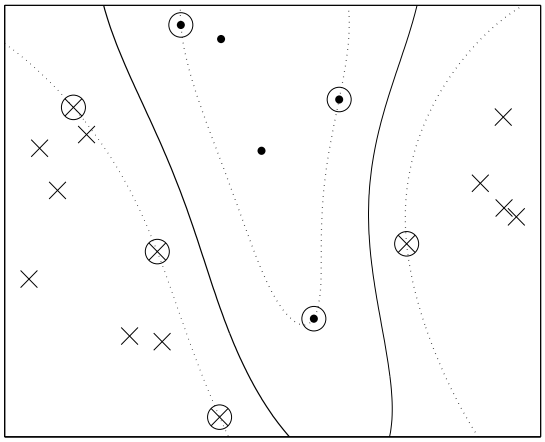

Figure 2.5: A nonlinear support vector machine. Again, we separate dots from 'x's. The decision boundary (solid curve) was found via (2.15), using a gaussian kernel. It corresponds to the maximum margin hyperplane in a feature space F implied by the gaussian kernel. The dotted curves show the place where $|f(\mathbf{x})| = 1$ (the margin in F). Note how the solution is expanded in terms of only six support vectors (encircled).

until now). The idea is that we would like to control not only the capacity of the solution, but also the *tradeoff* between low empirical risk and low capacity in (2.5). To this end, we introduce so-called *slack variables* ξ_i. In terms of the primal optimization problem (2.10) we write

$$\begin{array}{lll} \text{minimize} & \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^m \xi_i^d & \text{w.r.t. } \mathbf{w}, b \\ \text{subject to} & y_i(\mathbf{w}^T\mathbf{x}_i + b) + \xi_i \geq 1, & \\ & \xi_i \geq 0, & i = 1 \ldots m \end{array} \tag{2.17}$$

This means we allow each $\mathbf{x}_i$ to violate its constraint by ξ_i. If it does, we say that $\mathbf{x}_i$ has a *margin error* of ξ_i. In turn, we have to *penalize* these violations in the target function, otherwise any hyperplane would solve the problem, if only the slack is large enough. The constant C controls this tradeoff. If it equals infinity, we merely get back to the hard margin case. As C decreases, the contribution of the slack variables to the target function becomes smaller and the focus gets shifted from zero training error towards smoothness of the solution.

There are several ways we can account for margin errors. In (2.17), we have the sum of the ξ_i to the power of d. Usually, $d = 1$ (*1-norm soft margin*) or $d = 2$ (*2-norm soft margin*), that is, we penalize the slack vector $\boldsymbol{\xi} = (\xi_1, \xi_2, \ldots, \xi_m)$ by its l_1 or l_2 norm, respectively. Intuitively, a good approach would be a penalty term for $\boldsymbol{\xi}$ that *counts* the number of margin errors, but this turns out to yield an NP-complete problem.

Let us skip the necessary derivations and present the dual problem for the 1-norm soft margin support vector machine:

$$\begin{aligned} &\text{maximize} && \textstyle\sum_{i=1}^m \alpha_i - \frac{1}{2}\sum_{i,j=1}^m y_i y_j \alpha_i \alpha_j k(\mathbf{x}_i, \mathbf{x}_j) \quad \text{w.r.t. } \boldsymbol{\alpha}, \\ &\text{subject to} && \textstyle\sum_{i=1}^m \alpha_i y_i = 0 \\ & && 0 \leq \alpha_i \leq C, \quad i = 1 \ldots m \end{aligned} \tag{2.18}$$

Observe that the only difference to (2.15) is, that we require $\alpha_i \leq C$. This makes perfect sense: recall that a point $\mathbf{x}_i$ can be considered *difficult*, if its corresponding α_i is large. (2.18) places a constraint on the difficulty of single points, in other words, those $\mathbf{x}_i$ whose difficulty exceeds C become margin errors, instead of complicating the decision boundary. In contrast, for $C = \infty$, any point will lie on the correct side of the hyperplane (the hard margin case). For the 2-norm case, we have the dual

$$\begin{aligned} &\text{maximize} && \textstyle\sum_{i=1}^m \alpha_i - \frac{1}{2}\sum_{i,j=1}^m y_i y_j \alpha_i \alpha_j (k(\mathbf{x}_i, \mathbf{x}_j) + \frac{1}{C}\delta_{ij}) \quad \text{w.r.t. } \boldsymbol{\alpha}, \\ &\text{subject to} && \textstyle\sum_{i=1}^m \alpha_i y_i = 0 \\ & && \alpha_i \geq 0, \quad i = 1 \ldots m \end{aligned} \tag{2.19}$$

which can interpreted as follows: Here, the only difference to (2.15) is the term $1/C$ that is added to the each diagonal element of K. This amounts to using a slightly different kernel, namely

$$k'(\mathbf{x}_i, \mathbf{x}_j) = k(\mathbf{x}_i, \mathbf{x}_j) + \frac{1}{C}\delta_{ij} \tag{2.20}$$

which is still symmetric and positive definite. But K' is more diagonally dominant than K (we say, we add a *ridge* to the diagonal). This changes the spectrum of K such that all eigenvalues are increased by $1/C$. Now since in *ill-posed* problems like the ones that occur when we try to separate points with a too simple function, K is singular, we can use this method to assure that the smallest eigenvalue is at least $1/C$ and make the problem solvable.

The procedure introduces an artificial dissimilarity between points that are not exactly the same (i.e. all off-diagonal elements seem smaller). In fact, we can view the 2-norm method as adding m dimensions to the feature space, where for any $\mathbf{x}_i$ the kth additional component equals $\delta_{ik} y_i / \sqrt{C}$. If the data were not linearly separable before, they are now, since every point has been moved into its 'very own' dimension by $\delta_{ik} y_i / \sqrt{C}$. As a result, we are computing a true hard margin classifier, but in a slightly different feature space.

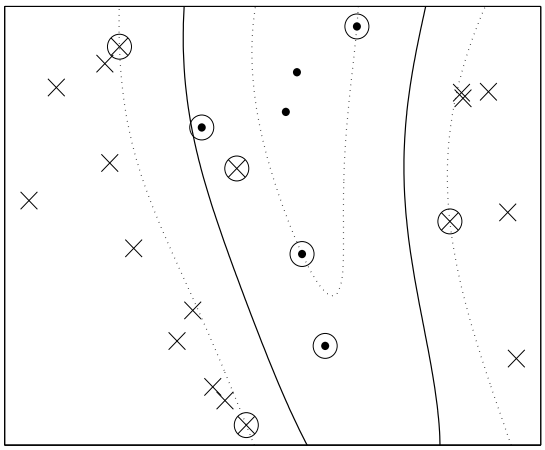

Figure 2.6: A soft margin classifier. This example shows a 1-norm soft margin support vector machine ($C < \infty$) using a gaussian kernel (2.14). Support vectors are encircled. There are five 'normal' support vectors (three 'x's, two dots) on the dotted lines which correspond to the active constraints with zero slack, i.e. $\xi_i = 0$ and therefore $y_i(\sum y_j \alpha_j k(\mathbf{x}_j, \mathbf{x})) = 1$. Additionally, we have a new type of support vectors here: The three remaining encircled points (two dots, one 'x') have become margin errors due to their unusual positions ($\xi_i > 0$). Note how the two dots violate the margin constraint, in particular, $0 < y_i(\sum y_j \alpha_j k(\mathbf{x}_j, \mathbf{x})) < 1$. There is a even one 'x' that lies on the wrong side of the decision boundary, i.e $y_i(\sum y_j \alpha_j k(\mathbf{x}_j, \mathbf{x})) < 0$.

Generally speaking, introducing soft margins amounts to *regularizing* the hard margin problem. It is not clear which regularization method performs better. The 1-norm version tends to yield sparser solutions, whereas the 2-norm version favors many small margin errors. The generalization results are very similar. Briefly, if we want sparsity, we use the 1-norm, if we need the hard margin interpretation (most theory, e.g. (2.5) applies to the hard margin case only), we use the 2-norm.

2.3.5 Discussion

We have shown the idea behind support vector machines in conjunction with some of the main theoretical advantages such as i) capacity control via implementation of the structural risk minimization principle, ii) convexity of the optimization problem, that is, no local minima, even in the nonlinear case and iii) sparsity of the solution.

Furthermore, there are fewer parameters that have to be chosen by hand compared to other methods. For instance, a soft margin SVM with a gaussian kernel needs only two (σ, C) parameters to be set a priori, whereas a neural network needs to have its whole topology designed manually.

A major drawback is the classification speed. Even though we get sparse solutions, they are by far not as sparse as it gets. In terms of computational complexity, support vector machines are clearly inferior to neural networks. So when it comes to time critical or even real-time applications, we might switch to such methods instead.

Part III

Skin Detection

3.1 Overview

In our system, we are doing skin detection for mainly two reasons. Firstly, detecting faces in images is a computationally expensive task. Sophisticated classifiers often are rather complex nonlinear mappings, whose evaluations require a significant amount of time, even on fast machines. If we have no prior knowledge about where faces occur within the image, we must evaluate the decision function at every position. This complicates the development of real time applications in this field. In our method, the idea is to restrict the search domain to image areas where pixels are likely to have skin color, since this is a necessary condition for a patch to contain a face.

Secondly, many neural network or support vector machine based face detection systems operate on gray scale images. Not only does this reduce the computational complexity of the classification process to one third (compared to three channel color images), but it also simplifies the training process, since the dimensionality of the data is reduced by just the same amount. However, it discards all chrominance information, which might actually be helpful for classification. In other words, we sacrifice accuracy for speed. Using skin color as a requirement for patches to contain faces reduces the chance that the classifier gets fooled by difficult patterns.

The present chapter describes the development of the skin detection component in our algorithm: we will show how the data are collected and how neural networks and support vector machines can be applied to this problem. We will look at their performance and discuss which method is more appropriate for our purposes. In addition, some standard image processing tricks will be applied in order to make the results more robust against noise.

3.2 Data Collection

To collect the training data, we use a matlab script that displays single frames from our sample video clips. It allows us to select several pixels from each frame, which are then automatically added to a data set. For the skin color examples, we mark pixels in face, neck, arm and hand regions, whereas for the non-skin examples, we select pixels from hair, cloth and background areas (Figure 3.1).

In order to make the data collection process less tedious, we choose a random subset of the training images, namely 100 out of 5197 available frames. Within each image, we mark between eight and ten pixels for each class, yielding 817 skin color examples and 908 non-skin color examples. These 1725 data points, together with their labels, serve as the training set that we will be using below.

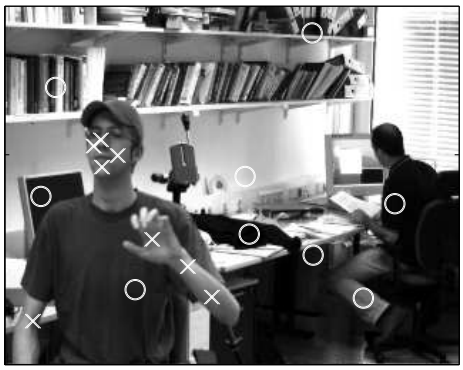

Figure 3.1: Data collection for training the skin color detector: This figure shows one of the 100 frames (out of 5197) from our video clips. The training samples are picked by hand and then stored automatically via a matlab program. The crosses/circles mark the skin/non-skin pixels that we collected from this image.

3.3 Feature Selection

The first step towards skin detection comprises *feature selection*. To this end, let us formulate skin detection as a binary classification problem: We have an input space X of pixel values, each element of which is associated with some label from $Y = \{-1, 1\}$. We say that *skin colored* pixels belong to class '1' and that *non-skin colored* pixels belong to class '-1'. Since color information is usually encoded in three numbers (for example, red-green-blue) we will have $X \subseteq \mathbb{R}^3$, in particular, when considering normalized color coordinates, $X \subseteq [0, 1]^3$. The goal is to find a function $f : X \to Y$ that predicts the skin/non-skin class membership for any color $\mathbf{x} \in X$.

Here, feature selection boils down to choosing *how* colors are encoded when we feed them into the classifier, that is, which meaning we give to the components (features) of elements in X. Such representations are called *color spaces*. In fact, there exist dozens of possible color spaces. A fair amount of research has been done on which color space is most suitable for separating skin from non-skin colors [11, 12, 13, 17, 18, 19]. The most frequently used representations are *RGB* (red, green, blue), *HSV* (hue, saturation, value) and *YUV* (luminance, redness, blueness).

Each model has its distinct properties and applications. RGB, for instance, is convenient when displaying color on cathode ray tube (CRT) or thin film transistor (TFT) displays, since there, color is physically composed from red, green and blue light. HSV can be found in image editing tools, as it is more intuitive for us humans to compose colors from hue, saturation and value than from RGB components. Finally, YUV is a standard for television systems. In particular, the luminance component is what we see on an old black-and-white television, where the two chrominance values encode the remaining color information.

However, it is not clear which model is best for *our* purpose, and different authors strongly disagree on that question - some claim that it does not make a difference at all [13]. We

believe that the choice depends too much on the particular setting to make a universally valid proposition. Therefore, we do feature selection in the following way: We compare the performance of the three models (RGB, HSV and YUV) in our own special environment and simply pick the one with the best result.

3.4 Model Selection

The second task is *model selection*. As we have seen in the previous chapter, we need prior knowledge about the data in order to generalize (recall that there is no free lunch). In case of neural networks, this information needs to be encoded into the network architecture, whereas with support vector machines, we have to fix the kernel and regularization parameters. In terms of learning theory, this means that we are modelling a specific class of decision functions, from which our learning machine picks the one that it considers the best.

To this end, we evaluate the performance of classifiers that use different combinations of color spaces (features) and function classes (models). In order to make judgements about the generalization performance resulting from certain feature/model combinations, we use a procedure called *cross-validation*. In cross-validation, we divide the training set into l subsets of equal size, called *folds*. For each feature/model combination, we train l classifiers, where each classifier uses a different set of $l-1$ folds as training data, more precisely, the ith classifier is trained on the all points that are not contained in fold i. Each classifier is then tested on that single fold (the *validation set*), which has been left out in the respective training process. Averaging the resulting test errors over the number of folds l yields the *cross-validation error*

$$R_{cv}(f) = \frac{1}{l} \sum_{\mathrm{i}=1}^{l} R^i_{test}(f^i)$$

where R^i_{test} denotes the test error on validation set i, and f^i is the decision function of the ith classifier. This quantity is used to measure the generalization performance that we expect from f, when using the respective feature/model combination.

The idea behind cross-validation is, that, since we test on unseen data, we hope to get a more reliable estimate of the expected test error compared to the empirical risk, i.e. $R(f) \approx R_{cv}(f)$. In fact, it has been shown that this *is* the case for the *leave-one-out* method, which is an extreme version of cross-validation where each fold contains only one element. For details, see [2, 3].

In the following, we will show how to pick the most suitable classifier, that is, the parameter set which leads to the most appropriate function class and thus to the smallest validation error. To this end, we use a very simple optimization technique called *grid search*. It amounts

to approximating the error surface in parameter space on a rough grid and choosing the parameter set that yields the minimal error.

3.4.1 Neural Networks

Let us first consider the neural network case. For simplicity, we restrict the possible architecture to fully connected feedforward multi layer networks with one hidden layer. In any case, the input layer has three neurons (one for each color channel) and the output layer has one neuron (for binary classification output). Thus, the only free parameter in the network structure is the number of neurons in the hidden layer k.

In our model selection example, we do not use all 1725 training points, but a random subset of size 512, color values are in $[0, 1]^3$. We use $l = 8$ folds and train eight neural networks for each $k \in \{1, 2, 3, 4, 5, 10, 20, 50\}$. All computations are done in matlab, using the *netlab toolbox* [4]. First, the network weights and biases are initialized with values drawn from a gaussian distribution with zero mean and unit variance. After a number of iterations, we stop the training process and monitor R_{emp} (i.e. the training error) and R_{tst} (i.e. the test error approximation). We continue training until a total of 5000 iterations. Figure 3.2 shows the results of this procedure.

We find that the lowest cross validation error on our parameter domain is around 9%, regardless of the choice of color space. Interestingly, the problem seems to be slightly easier in RGB space: In contrast to the HSV and YUV cases, here, 5000 iterations are sufficient to overfit the high capacity networks.

This means that, in general, we have to take care of the second parameter that influences the network capacity, namely the number of iterations. The synaptic weights, starting from small values, keep growing as the solution gains complexity. If we iterate too long and the network architecture has enough capacity to overfit the data, it will do so (Figure 3.3). This motivates the so-called *early stopping method*, where we stop the training process, once we reach the lowest validation error. However, it is quite difficult to tell when this is the case. As an aside, there are other methods for capacity control that do not take into account the number of iterations. For instance, we can penalize the synaptic weights by $\|\mathbf{w}\|^2$ - similar to support vector machines - or successively delete neurons with small weights, in order to smoothen out the decision boundary.

Luckily, the skin color classification problem is a rather simple one. In the RGB case for instance, the small ($k = 3, 4$) networks seem to never overfit, but yield a similarly low test error compared to the best values of the large ($k > 10$) networks. In other words, if we pick a small k, there is no need for additional complexity control as the classifier still performs well. Moreover, the fewer hidden neurons we have, the higher classification speeds we get.

The result of the model selection procedure is therefore as follows: For skin detection

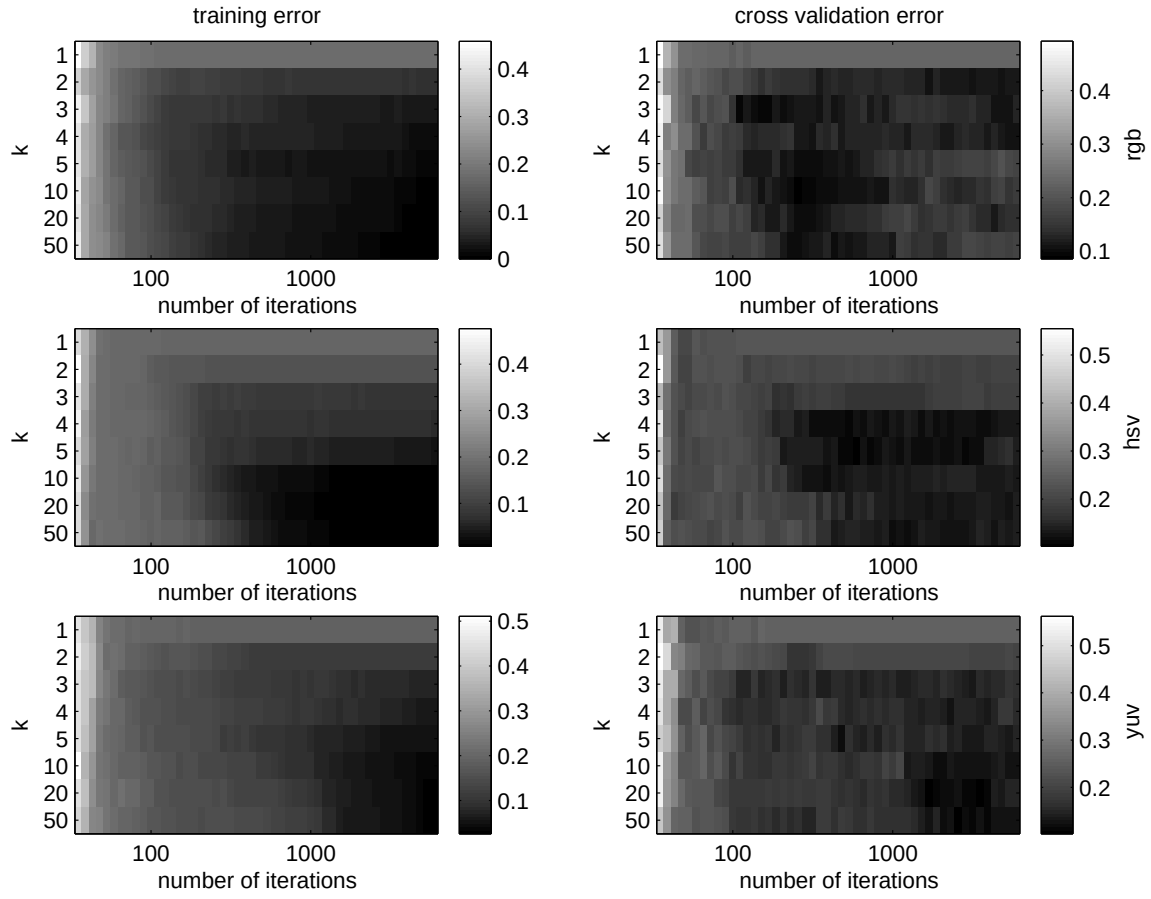

Figure 3.2: Model selection for the neural network classifier. We have tested three colorspaces RGB, HSV and YUV (rows). The left/right column shows the training/validation error, respectively. In each image, the model capacity (implied by k) increases from top the bottom, whereas the number of training iterations increases from left to right. We have chosen the evaluation points in a way that the scale of these images is roughly logarithmic for both directions. Note how the training error smoothly goes down as we increase the complexity, that is, towards the lower right of each image. The test error surfaces are more interesting, especially the RGB (leftmost) case: Towards higher capacities, the increase of validation error shows that we start to overfit.

with feedforward multi layer networks, we expect optimal results from the feature/model combination RGB color space and $k = 4$ hidden neurons.

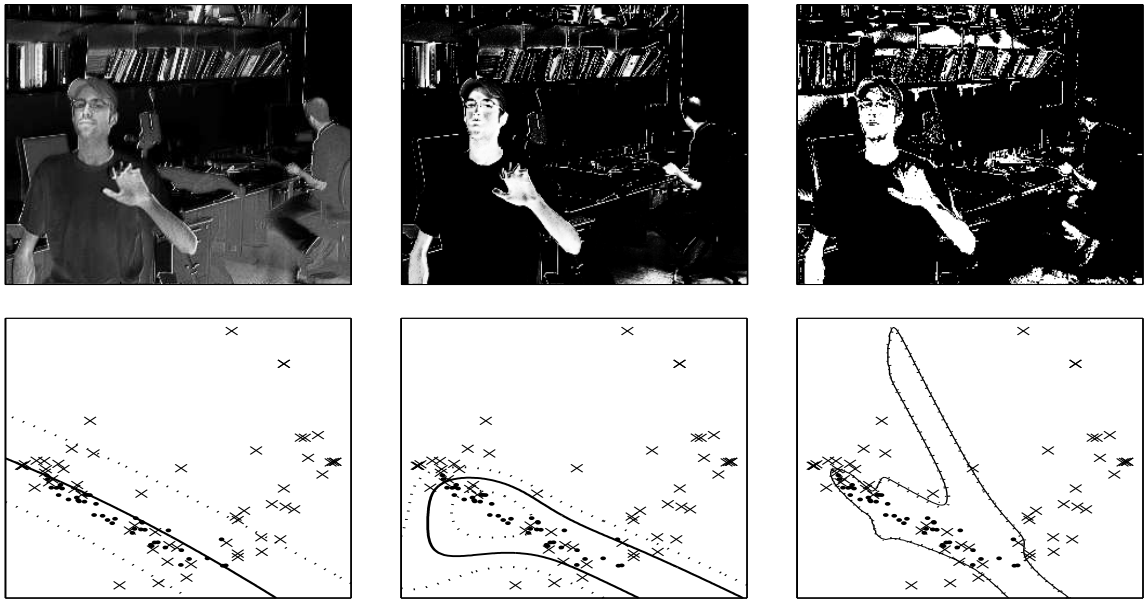

Figure 3.3: The problem with too large function class capacities in neural network training. The top row shows the output (evaluated at every pixel on the image from Figure 3.1) of a neural network with $k = 50$ hidden neurons at three stages during training, namely, after 10, 300 and 1500 iterations. The corresponding decision boundary is plotted below. To illustrate the this three-dimensional problem, we projected the RGB coordinates onto the plane spanned by the two first principal directions of the training data and plotted 100 training points. We observe that after 10 iterations (left column), the synaptic weights are still small, yielding a very simple, underfit solution. After 300 iterations (middle column), the result (top) and the complexity (bottom) seem just right. If we continue training, the decision function overfits, resulting in a too complex solution and a high test error (right). Finally, note how the notion of smoothness nicely applies to both the decision function (bottom row) and the surface implied by its outputs on the test image (top row).

3.4.2 Support Vector Machines

As the second approach to skin detection, we apply support vector machines. Again, we try all three color spaces together with various function classes. We use a gaussian kernel (2.14) in combination with a 1-norm soft margin classifier (2.15). This gives us two model parameters: the kernel width σ and the regularization constant C (2.16). Like in the neural network case, we pick values in a way that the error surface has logarithmic scale in both directions, in particular, $\log \sigma \in \{-1.5, -1, -0.5, 0, 0.5, 1\}$ and $\log C \in \{-1, -0.5, 0, ..., 3, 3.5\}$. For these

experiments, we use matlab and the *SPIDER* [8] toolbox for training and testing. We do cross validation with $l = 8$ folds, using the same training set as with the neural network classifier, that is, 512 training points. Results are plotted in Figure 3.4.

It turns out that support vector machines perform slightly better than neural networks. The lowest validation errors are approximately 8% for all color spaces. From the error surfaces in Figure 3.4, we expect an optimal kernel width around $\sigma = 0.06$, the best suited regularization parameter appears to be $C = 10$ for RGB/HSV and $C = 100$ for YUV color space. Using these values, the 1-norm soft margin algorithm yields roughly 150 support vectors for each color space.

And again, color space does not seem to be an important issue when it comes to generalization performance. We therefore decide that our optimal feature/model combination for the 1-norm soft margin classifier with a gaussian kernel is an RGB color space and $\sigma = 0.06$, $C = 10$.

3.5 Performance

Let us compare the two optimal classifiers, that is, the neural network with $k = 4$ hidden neurons and the support vector machine with $\sigma = 0.06$, $C = 10$. This time, we use all available training data (1725 points) and evaluate both classifiers via eight fold cross validation. We get an expected test error of 9.8% for the neural network and 7.4% for the support vector machine (with 234 support vectors on average). Figure 3.5 shows the classification results on a test image.

Although the support vector machine performs better on the training data, in Figure 3.5 we hardly notice a difference to the neural network results. Looking at the computational complexity of the decision functions, it becomes clear that the advantage of the support vector solution cannot justify the large number of computations needed for its evaluation. With 234 support vectors, we have to compute 234 dot products (in three space). In contrast, the neural network needs only four dot products.

During training, we have seen that for simple classification problems like skin detection, the theoretical advantages of support vector machines have no noticeable effect on the convenience of model selection (or training itself) since this turns out to be quite simple anyway. During testing, though, one difference is apparently the higher computational cost of the support vector solution. We therefore chose the neural network for our implementation.

In terms of feature selection, our results agree to [13] in the sense that choice of color space does not significantly change the results. Intuitively, since the mappings between different color spaces are rather simple or even linear, we expect that a sophisticated classifier can easily compensate for the peculiarities of each color space. Our choice will be RGB.

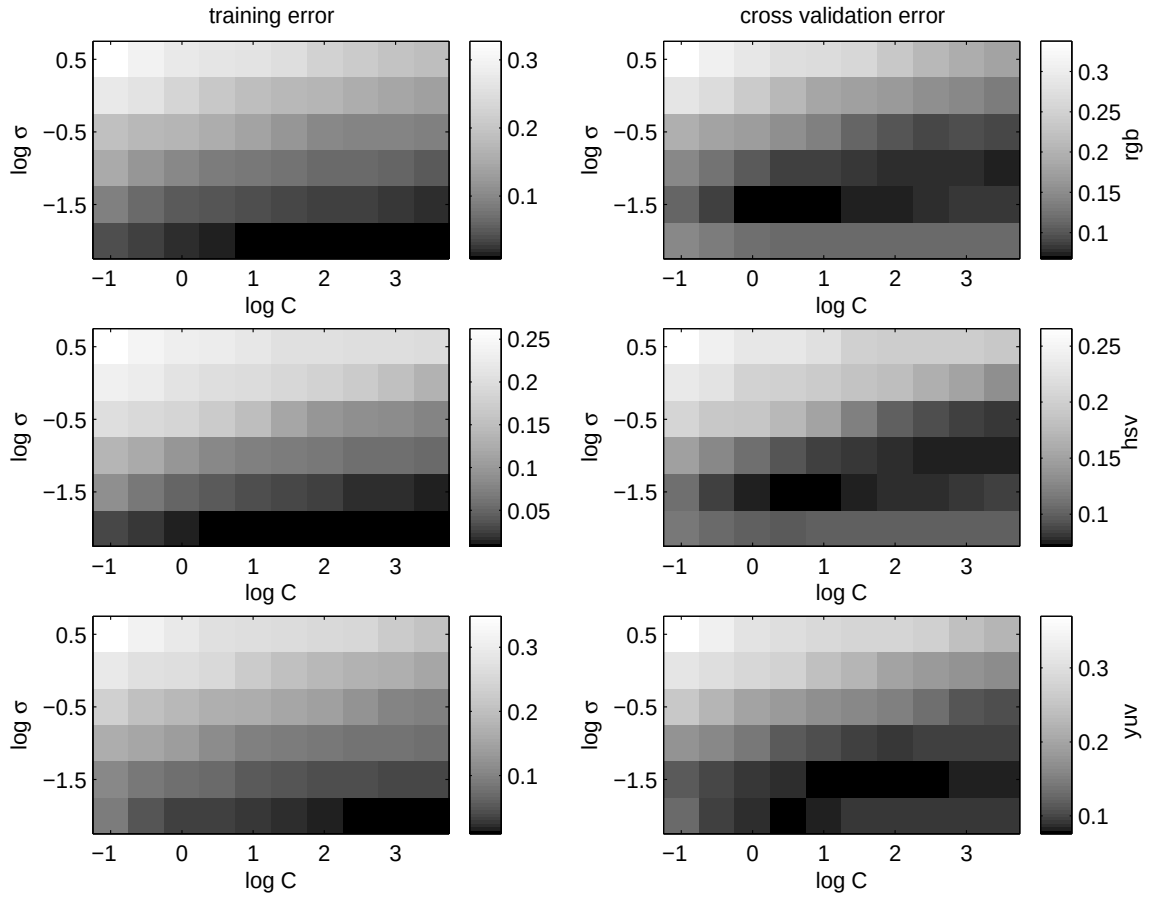

Figure 3.4: Model selection for the support vector machine classifier. Like in Figure 3.2, we have plotted training and validation errors for different parameter sets σ, C. Here, $\log \sigma$ varies along the vertical where $\log C$ increases from left to right. We can see that the validation error surfaces seem to have one possibly global minimum around $\log \sigma = -1.2$ and $\log C = 1, 2$ for RGB/HSV and YUV, respectively.

We have tested a C implementation of the neural network on an Intel Pentium 4 machine (2.8 GHz) under Windows XP. It is able to detect skin colored pixels in 320x200 images within approximately 2.1 milliseconds.

Figure 3.5: Support vector machines vs. neural networks in skin detection. The leftmost picture shows our test image. In the middle, we have the output of the SVM decision function, on the right we have the corresponding result from the neural network classifier. Note that both approaches lead to a very similar (visual) result, even though we expected the neural network solution to be slightly inferior.

3.6 Postprocessing

Recall that the purpose of our skin detection method is to stabilize and accelerate the subsequent classification procedures, in particular, face detection. To this end, we would like the skin color preprocessing to yield a *skin map* of the image, that is, a binary image that tells us where it makes sense to search for faces. Since the skin color detection operates purely local, we get a quite noisy skin map in the first place (Figure 3.5), especially at object boundaries. Many of these skin colored yet isolated pixels are superfluous - relatively large objects like faces are usually found within larger skin colored regions. Moreover, our skin detection does not detect eyes, eye brows or mouths. This results in holes within face areas that may mislead the face detection method.

So we have to remove two kinds of noise: Isolated skin colored pixels in non-skin regions and non-skin colored pixels within skin areas. To this end, we use some standard techniques from image processing.

We first apply a 5x5 *median filter* to the raw classification output. This is a common technique for removing speckle noise in images. Formally, the filter output at position (u, v) equals the median of the values inside the box defined by $[u-2, u+2] \times [v-2, v+2]$. In our binary skin maps, we therefore set the output at (u, v) to 'skin colored' if and only if more than 12 pixels in the 5x5 neighborhood are also skin colored.

Then, to fill the larger holes, we *close* the image with the same 5x5 rectangular mask. The closing transform amounts to the subsequent application of *dilation* and *erosion*. When dilating our skin map, we set the output at (u, v) to 'skin colored', if at least one pixel in the 5x5 neighborhood is also skin colored. Erosion can be viewed as the opposite thereof:

Figure 3.6: The three postprocessing stages. The left image shows the network classification output (Figure 3.5, rightmost image) after 5x5 median filtering. The subsequent closing transform yields the middle image. After re-opening the image, most of the unwanted skin pixels have been removed (right image).

Here, all neighbor pixels have to be skin colored if the output is 'skin colored'. Finally, we *open* the image again, that is, we erode and *then* dilate. This makes the skin regions more 'blobby' and eventually re-separates regions that have been accidently merged by the closing operation. The effect of all these operations is illustrated in Figure 3.6.

The postprocessing procedure can be implemented very efficiently and yet improves the results significantly, that is, at least intuitively - when we look at the resulting skin maps. Note that median filtering usually involves sorting the pixel values. In the binary case, we may merely count the pixels that are skin colored and check if this number exceeds 12 (i.e. more than one half the total number of pixels on the filter mask). Closing and opening can be implemented by comparing the same number to 1 or 24, respectively.

In our C implementation, the postprocessing takes another 2.2 ms, so the overall cpu time needed to get from the RGB input image to the final de-noised skin map equals roughly 4.3 milliseconds. We will see during the next sections that this investment does pay off.

Part IV

Face Detection

4.1 Overview

Face detection has been studied by many researchers in image understanding and machine learning and there exist a number of quite different approaches to this problem (e.g. [23, 24, 25]). For instance, in component based methods, we search for face elements like noses, eyes and mouths and then try to find configurations that match some face model. Another idea is to fit circles or ellipses into skin colored image regions. In this work, we use a patch based approach, that is, we classify rectangular image patches as a whole. The patches are divided in two groups, depending on whether they show a face or not. Then, we train a binary classifier (a support vector machine) with those two classes. This method is a very simple one, in the sense that we do not need to model the face geometry by hand. Instead, we let the SVM training do this work.

Note that formally, face and skin detection are almost identical: Both problems are solved by training a binary classifier, the only difference being the dimensionality of the data. In the skin color case, we chose $X \subset \mathbb{R}^3$. Now, we choose $X \subset \mathbb{R}^{361}$ (we will see below, why exactly 361), since we obviously need much more features in order to encode the great variety of image patches. In this case, we start to notice the effects of the so-called *curse of dimensionality*. It basically means that the complexity of functions in some space increases exponentially with the dimension of the latter. Although this is advantageous in terms of data separability (Cover's theorem, see previous chapter), we run into problems. If we want the data to be somewhat dense in X (or even F), the number of training points has to increase exponentially as well. This, in turn, implies a very time consuming training phase and/or an under-determined problem which needs a substantial amount of regularization. Moreover, we will have many support vectors (on the order of thousands), making real-time applications impossible.

As a consequence, the present chapter focusses on how to approach these problems. The resulting algorithm is based on [21, 22]. While the first sections will be similar to those in the skin detection chapter, we will go into more detail during model selection and find some interesting properties of the gaussian kernel. Additionally, we will introduce two concepts we have not mentioned yet: *bootstrapping* and *reduced set methods*.

4.2 Data Collection

Like before, we build our face database from the sample video clips. To this end, we use another matlab program that cuts and stores the 19x19 patch around the image position we click on. This yields 5197 face patterns. To further increase the variety, we double the face set size by mirroring each pattern vertically (10394 patches). For the non-faces, we use another

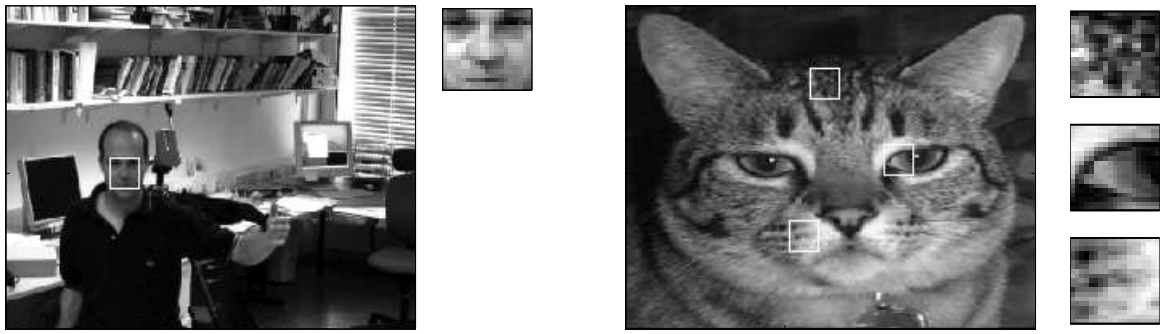

Figure 4.1: Data collection for the face classifier: The left part of the figure shows a sample frame from our training video clips. The selected face patch is marked by the white square and is also shown in a magnified version (small image). On the right half we have one of the images we used for extracting non-face patches. Here, we randomly picked three patterns (right small images). In a way, this shows how face patterns much more valuable, since their acquisition involves considerable human effort, whereas non-face patterns can be easily collected automatically.

program that automatically collects 15000 patterns from 134 randomly selected images that do *not* contain faces. An example is shown in Figure 4.1.

Note that the number of possible non-face patterns is much larger than the number of possible face patterns. In other words, the class sizes are *unbalanced* and in fact, we should use a similar ratio in the training set (since we assume the training data to be i.i.d, see section 2.1.2). Unfortunately, this is an impossible thing to do: we would have to train on millions of patterns, where the time complexity of (2.15) is equal to that of a matrix inversion, i.e. $O(m^3)$. So the actual number of non-face patches, 15000, is a tradeoff between accuracy and computational tractability. The idea here is to use these roughly 25000 patterns for model selection. Then, the solution with the optimal parameter set is bootstrapped in order to incorporate additional information of other possible non-face patterns in a computationally efficient way.

4.3 Feature Selection

We did feature selection quite straightforwardly in the face detection case. Given a 19x19 color image, we merely convert it to gray scale and use each pixel value as one component of a 361 dimensional input vector. We basically chose this representation because of its frequent use in related work, so no testing or comparison to other approaches has been made.

Still, there are some thoughts about why these features might be preferable over others. The use of gray images instead of color images can be motivated by the fact, that it reduces the

pattern dimensionality to a third (and thus reduces the effects of the curse of dimensionality). That way, we we may train with a far smaller training set, which makes a great difference in terms of how tedious data collection becomes. Another topic is computational cost. Rather than processing plain pixel values, we could use a different image representation, for instance wavelet or fourier coefficients. But then, we would have to transform every image patch we want to classify, resulting in longer computation times.

This brings us to another important point. Many robust face detection implementations preprocess each patch before classification. Major issues here are illumination and contrast. In fact, illumination and contrast variations can change an image by a surprising amount. There are a few common techniques to keep image patches *invariant* under these changes. In [24], for example, the authors fit a plane to the brightness surface (i.e. do a linear regression) and subtract it from the image. That way, they reduce the effect of non-uniform illumination of the faces. They also modify the gray value histogram of each patch so that all patterns have equal contrast.

We will not apply any of these preprocessing techniques for the sake of speed. Note that each patch has to be preprocessed before being passed to the classifier, which might be computationally too expensive for a real-time application. Instead, we try to keep the illumination in our setting as constant as possible and hope that the classifier will learn to ignore the unavoidable minor deviations.

4.4 Model Selection

We use a 2-norm soft margin support vector machine (2.19) with a gaussian kernel (2.14) for face detection. Even though the 1-norm regularizer usually leads to sparser solutions, we chose the 2-norm version here, since we wanted to examine the behavior of the function capacity (via (2.9)) in comparison to the validation error. Like before, we do a grid search to find the optimal capacity parameters. Here, we use all available training data for model selection, that is, 10394 faces and 15000 non-faces. We vary the logarithm of the kernel width σ between -0.5 and 2.5, and the logarithm of C between -1 and 3 where the step size equals 0.5 on a logarithmic scale and we use $l = 8$ folds for cross validation.

The results are depicted in Figure 4.2. For each pair of parameters, the first two images show the training and validation error surfaces, the left picture in the middle shows the number of support vectors. For $\log \sigma = 0.5$ and $\log C \geq 0$, the validation error goes down to as low as 0.01, meaning that the training set can be easily separated. Interestingly, the surface implied by the number of support vectors (middle left image) has a very similar shape (which makes sense, since the fraction of support vectors can be used as a bound on the probability of test error [2]).

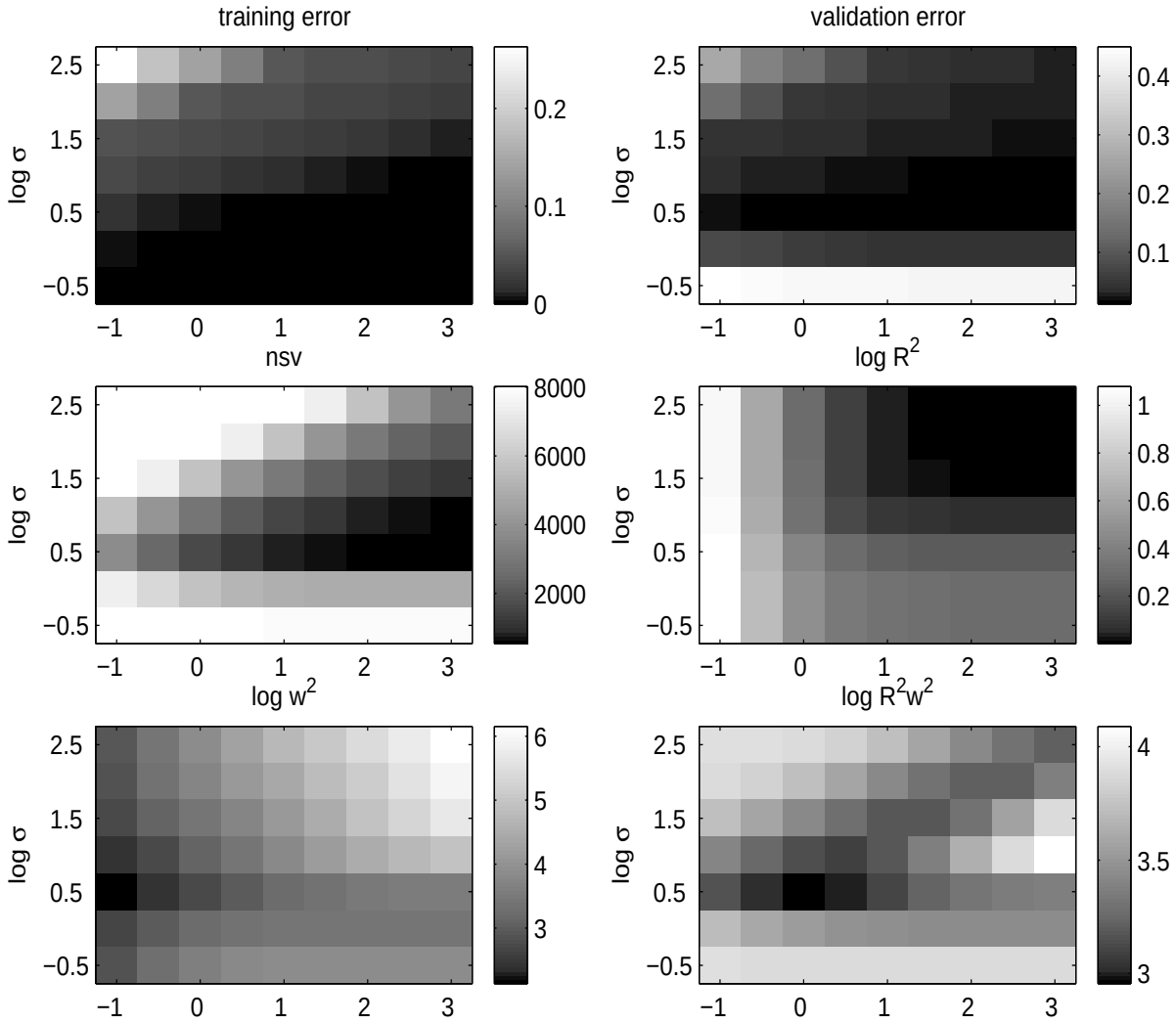

Figure 4.2: Model selection for the face detection SVM. The top row shows the training and validation error, the left middle image shows the number of support vectors. Like before, the horizontal axes denote the amount of regularization $\log C$ and the vertical axes show the kernel width $\log \sigma$. In the remaining three pictures, we have plotted the ingredients for the margin-based bound on the test error (2.9 into 2.5). Note that we use the logarithm of these quantities only for visualization purposes.

To illustrate the behavior of (2.9), we have plotted the squared radius R^2 of the data in feature space, the inverse margin squared $\|\mathbf{w}\|^2$, and the product of the two. Note that the derivation of the squared radius leads to another optimization problem, namely finding the *smallest enclosing sphere* of m points in feature space. Instead of calculating the exact solution, we apply a common trick and compute the *variance* of the data in feature space, which provides an approximation to the radius. The inverse margin is evaluated directly via a simple dot product. As a result, we have for gaussian kernels

$$\begin{aligned} \mathrm{R}^2 &\approx \mathbf{V}\left[k(\mathbf{x},\cdot)\right] \\ &= \mathbf{E}\left[k(\mathbf{x},\cdot)^2\right] - \mathbf{E}\left[k(\mathbf{x},\cdot)\right]^2 \\ &= \tfrac{1}{m}\textstyle\sum_{i=1}^m (k(\mathbf{x}_i,\mathbf{x}_i) + \tfrac{1}{C}) - \tfrac{1}{m^2}\sum_{i,j=1}^m (k(\mathbf{x}_i,\mathbf{x}_j) + \tfrac{1}{C}\delta_{ij}) \\ &\approx 1 + \tfrac{1}{C} - \tfrac{1}{m^2}\textstyle\sum_{i,j=1}^m k(\mathbf{x}_i,\mathbf{x}_j) \end{aligned} \tag{4.21}$$

and

$$\begin{aligned} \|\mathbf{w}\|^2 &= \|\textstyle\sum_{i=1}^m y_i\alpha_i k(\mathbf{x}_i,\cdot)\|^2 \\ &= \textstyle\sum_{i,j=1}^m y_i y_j \alpha_i \alpha_j k(\mathbf{x}_i,\mathbf{x}_j) \\ &= \textstyle\sum_{i=1}^m \alpha_i - \tfrac{1}{C}\|\boldsymbol{\alpha}\|^2, \end{aligned} \tag{4.22}$$

where, according to (2.9), the product of these quantities $R^2\|\mathbf{w}\|^2$ bounds the VC dimension h of the function class implemented by the respective support vector machine. For a more detailed description of the derivations, see [1, 2].

The R^2, $\|\mathbf{w}\|^2$ and $R^2\|\mathbf{w}\|^2$ plots provide further insight into what exactly happens, if we vary σ and C. Formally, from (4.21) we have

$$\frac{\partial R^2}{\partial \log C} = -\frac{1}{C} < 0$$

and

$$\frac{\partial R^2}{\partial \log \sigma} = -\frac{1}{2\sigma^2}\frac{1}{m^2}\sum_{i,j=1}^m k(\mathbf{x}_i,\mathbf{x}_j)\|\mathbf{x}_i - \mathbf{x}_j\|^2 < 0,$$

since for the gaussian kernel, $k(\cdot,\cdot)$ is strictly positive. In particular, we see that the radius of the data in feature space decreases, if we increase C and/or σ. This comes from the fact, that i) we are adding $1/C$ to the squared length of each point in feature space (2.20) and

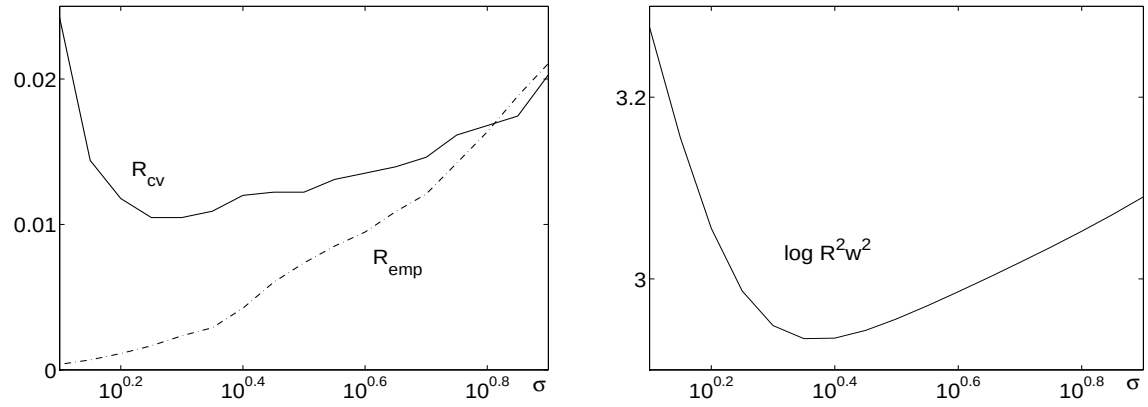

Figure 4.3: Model selection part II. Here, we have fixed $C = 1$ and search for the optimal σ on a finer grid (compared to the one in Figure 4.2). In the left picture, we have plotted the training and validation error against the kernel width. The right plot shows the capacity bound $R^2\|\mathbf{w}\|^2$. Note how these curves correspond to the ones in Figure 2.1 and 2.2 in the theory chapter.

ii) to a wide gaussian kernel all points look quite similar, and so under the corresponding mapping, the data do not spread much.

Interpreting the margin image is not that simple. The derivatives of $\|\mathbf{w}\|^2$ depend on $\boldsymbol{\alpha}$, which in turn depends on the solution of the optimization problem. The only thing we can say a priori is that the margin decreases monotonically if C grows. This is because for larger C, the kernel matrix K becomes less diagonally dominant, i.e. patterns become more similar to each other. We may justify this by taking the derivative of (4.22) with respect to $\log C$, which yields

$$\frac{\partial\|\mathbf{w}\|^2}{\partial \log C} = \frac{1}{C}\|\boldsymbol{\alpha}\|^2 > 0.$$

The lower right image illustrates the $R^2\|\mathbf{w}\|^2$ surface. It has a noticeable minimum at $\log\sigma = 0.5$ and $\log C = 0$.

Moreover, for $\log\sigma \gg 0.5$, we observe that the level curves are almost straight lines with slope 0.5 (as in the R_{emp}, R_{cv} and nsv images). This comes from the fact, for larger σ, that the gaussian kernel becomes approximately linear: from to the Taylor expansion we have

$$k(\mathbf{x}_i, \mathbf{x}_j) \to -0.5\|\mathbf{x}_i - \mathbf{x}_j\|^2/\sigma^2$$

for $\sigma \to \infty$. Now assume that at some $(\sigma,\, C)$, we change σ to $\sigma' = \sigma(1+\varepsilon)$. Since our kernel is close to linear here, this does not change the geometry of the data, but merely scales all distances by $1/(1+\varepsilon)^2$. Interestingly, in (2.17), this change can be made 'undone'

by setting $\mathbf{w}' = \mathbf{w}(1+\varepsilon)$ and $C' = C(1+\varepsilon)^2$. The only difference now lies in the target function, which has been scaled by $(1+\varepsilon)^2$. But re-scaling the target function does not change the solution. Hence, on a logarithmic scale, the initial solution is retained if we change $\log\sigma' = \log\sigma + \log(1+\varepsilon)$ and $\log C' = \log C + 2\log(1+\varepsilon)$. In the pictures of Figure 4.2, this amounts to moving along a line with slope 0.5. Note that this phenomenon can be observed on the training and validation error surfaces as well as on the number of support vector and $R^2\|\mathbf{w}\|^2$ surfaces.

Finally, let us examine how the pictures are related to each other: The $R^2\|\mathbf{w}\|^2$ surface on the lower right combines then notions of scale (R^2) and separability ($\|\mathbf{w}\|^2$). By virtue of (2.5), we know that the expected test error $R \approx R_{cv}$ (upper right image) is bounded by R_{emp} (upper left image) plus a capacity term that involves the VC dimension h, which is in turn bounded by $R^2\|\mathbf{w}\|^2$ (lower right image). Interestingly, even though (2.5) is not tight, people usually get good results when using the minimizer of $R^2\|\mathbf{w}\|^2$.

In our case, the classification performance seems to be quite robust to changes in C once we have the right kernel width σ. We therefore fix $C = 1$ (as it is the minimizer of $R^2\|\mathbf{w}\|^2$) and search for the optimal σ on a finer grid (between $\log\sigma = 0$ and $\log\sigma = 1$ with log step size 0.05). The results are shown in Figure 4.3: According to the validation error, we would chose $\log\sigma \approx 0.28$. The $R^2\|\mathbf{w}\|^2$ bound favors smoother solutions - here, we would decide for $\log\sigma \approx 0.39$. In the following, we use $\log\sigma \approx 0.38$ ($\sigma = 2.4$), which means that we are rather trusting the margin bound.

4.5 Bootstrapping

From the model selection procedure, we expect the optimal parameters to be $\sigma = 2.4$ and $C = 1$. The resulting classifier has 5077 support vectors and achieves an average *class rate* (fraction of correct predictions, i.e. $1 - R_{cv}$) of 98.9% on the validation sets. As mentioned before, our training set does not represent the huge amount of non-face patterns that can occur in real-life situations. We therefore keep a separate, 'worst case' test set with 1000 face (500 + mirrored) and 23577 'difficult' non-face patches (taken from the MIT CBCL database [27]). Here, we get a class rate of 94.1%, which still seems satisfactory. But in fact, the performance is rather poor, once we examine the fractions of *true positives* (or *hits*, i.e. faces that are correctly classified as faces) and *false positives* (i.e. non-faces that are misclassified as faces). On the test set, we achieve a hit rate of 98.8% but also a false positive rate of 5.7%. Now 5.7% of 23577 is already more than 1300 false classifications. Even worse, since we are looking at images which contain only one face each, the classifier should (ideally) return 'face' at only *one* position within each image - now if 5.7% of the remaining positions (more than 60000, at a 320x240 resolution) are misclassified, our method is not usable.

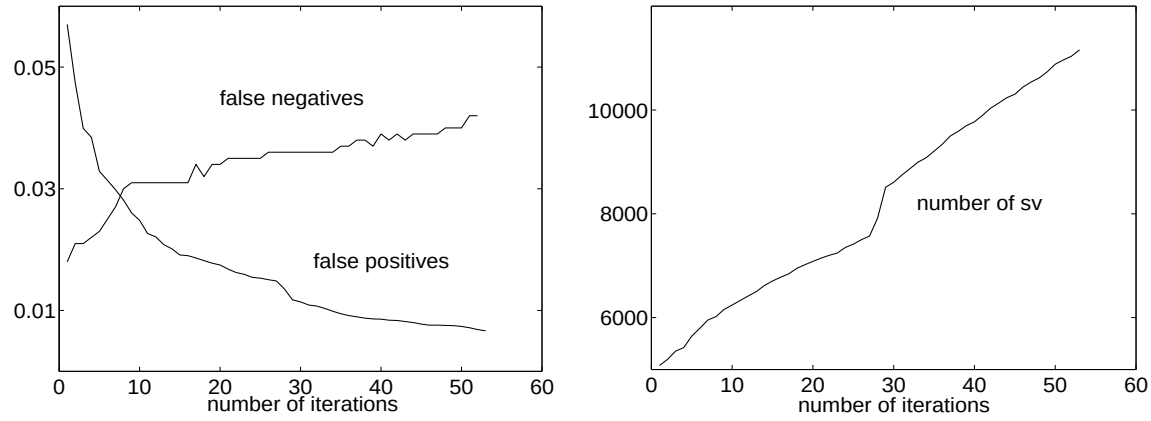

Figure 4.4: Bootstrapping. These plots show how hit rate, false positive rate and number of support vectors evolve during the bootstrap iterations. In the latter two plots we can see the gap at iterations=27, where we changed the number of random non-faces added to the training set.

We use a technique called *bootstrapping* [24] to further reduce the number of false positives. Recall that the SVM solution depends only on the support vectors. If we retrain the classifier with these 5077 points, we will find exactly the same solution. In bootstrapping, we create a new training set by adding new non-face examples to the set of support vectors and retrain the classifier. We expect that, if we repeat this procedure, the classifier will learn more and more about the non-face patterns and thus yield a lower false positive rate.

This process can be automated in the following way. At each iteration, we randomly pick 20 of our 134 images that do not contain faces (cf. 4.2) and randomly extract 150 patches from each image. These 3000 patches are added to the current set of support vectors. Now as the size of the solution grows, the effect of adding new points becomes weaker. So after 27 iterations, we changed the numbers so that 300 patches are picked from each of 30 images. After 54 iterations (11159 support vectors) we stopped the process, since training became too time-consuming. The results are plotted in Figure 4.4.

Although the hit rate of the final classifier has dropped to 95.8%, it achieves a false positive rate of only 0.67%. While this value is still significant, given the large number of possible non-face patterns, the result is almost ten times better than what we got from the initial version. To illustrate this, we have plotted the outputs of the support vector machine on a test image, before and after bootstrapping, respectively (see Figure 4.4).

Figure 4.5: The effect of the bootstrapping procedure. In this figure we have plotted a test image (left) together with the support vector output before (middle) and after (right) bootstrapping. Bright pixels denote positions with large output values. After bootstrapping, the positive response on for example the bookshelf and the subject's shoulder is damped by a noticeable amount.

4.6 Reduced Set Methods

The problem now is that we have 11159 support vectors, which is far more than what we can handle computationally in a real-time setting. As mentioned in the theory chapter, solutions of (2.15) tend to be sparse, but not necessarily as sparse as possible. In the following sections we will see that the size of our solution may be reduced to less than 1%. The resulting simplified algorithm is an example of a so-called *reduced set method*. A crucial ingredient here is the computation of *pre-images*, that is, finding points $\mathbf{x} \in X$ that correspond to given points $\boldsymbol{\psi} \in F$. We will show how to use this idea in order to chose a new set of support vectors which is much smaller than the initial solution and at the same time keeps the classification performance at a reasonable level.

4.6.1 Pre-Images

Recall that a kernel $k(\cdot, \cdot)$ denotes a dot product in some feature space F, spanned by the image of X under the implied feature map $\boldsymbol{\phi}$. In this setting, the *pre-image problem* is as follows: Given $\boldsymbol{\psi} \in F$, find $\mathbf{z} \in X$ so that $\boldsymbol{\psi} = \boldsymbol{\phi}(\mathbf{z})$ (Figure 4.6). To this end, an important topic is the *existence* of $\mathbf{z}$. If it exists, it can be easily recovered (as shown in [2]).

We can make use of this idea in order to speed up our face classifier: The decision function (2.16) is basically a dot product in F. We may write it as

$$f(\mathbf{x}) = \operatorname{sgn}\left[\langle \boldsymbol{\phi}(\mathbf{x}), \boldsymbol{\psi} \rangle + b\right],$$

where

$$\boldsymbol{\psi} = \sum_{i=1}^{m} y_i \alpha_i \boldsymbol{\phi}(\mathbf{x}_i).$$

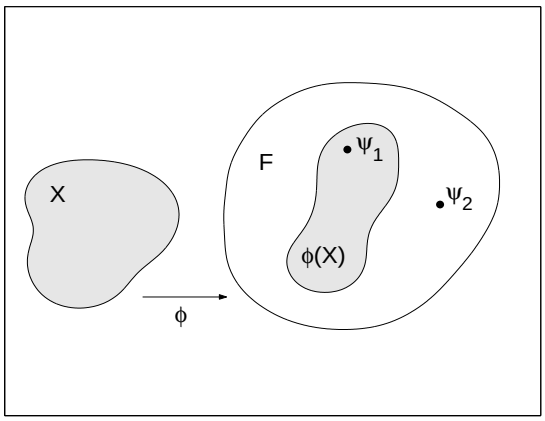

Figure 4.6: The setting for the pre-image problem. We have a space X and a mapping $\boldsymbol{\phi}$ (in our case, $\boldsymbol{\phi}(\cdot) = k(\mathbf{x}, \cdot)$). The image of X under $\boldsymbol{\phi}$ is a subspace of F (the closure of the span of $\boldsymbol{\phi}(X)$). If the feature map is implied by a kernel function, there exists a pre-image for any $\boldsymbol{\psi}$ which can be written as $k(\mathbf{x}, \cdot)$. Here, this is the case for $\boldsymbol{\psi}_1$. In contrast, $\boldsymbol{\psi}_2$ does not have a pre-image in X.

In other words, the vector from the origin of F to $\boldsymbol{\psi}$ equals the normal of the hyperplane.

The high computational complexity of our solution comes from the fact, that we have to express $\boldsymbol{\psi}$ as a linear combination of images of 11159 support vectors. The idea is to simplify this expression by reducing the number of necessary points. Ideally, if we manage to find a pattern $\mathbf{z} \in X$ such that $\boldsymbol{\psi} = \boldsymbol{\phi}(\mathbf{z})$, the decision function reduces to $f(\mathbf{x}) = \operatorname{sgn}(k(\mathbf{x}, \mathbf{z}) + b)$ and can therefore be computed more than 10000 times faster.

However, the existence assumption does not hold in our case. It can be shown that for the gaussian kernel, any $\boldsymbol{\phi}(\mathbf{x}_i)$ and $\boldsymbol{\phi}(\mathbf{x}_j)$ are linearly independent, if $\mathbf{x}_i \neq \mathbf{x}_j$. Hence, there is no non-trivial linear combination $\sum \alpha_i \boldsymbol{\phi}(\mathbf{x}_i)$ which has a pre-image in X (the $\boldsymbol{\psi_2}$ case in Figure 4.6).

4.6.2 Approximate Pre-Images

Instead of computing exact pre-images, we may in this case switch to finding *approximate pre-images*. We say that for a 'good' approximate pre-image $\mathbf{z}$ of $\boldsymbol{\psi}$,

$$\|\boldsymbol{\psi} - \boldsymbol{\phi}(\mathbf{z})\|^2 \tag{4.23}$$

should be small. More precisely, we compute approximate pre-images via minimizing the squared distance in feature space (4.23) with respect to $\mathbf{z}$.

In our special case however, observe that the $\boldsymbol{\psi}$ for which we want to find a pre-image, only occurs within a dot product, meaning that we are interested in a *direction* rather than a position. We may therefore loosen the constraints such that we seek for a pattern $\mathbf{z}$ whose image is close to a *multiple* of $\boldsymbol{\psi}$. In other words, we minimize the squared *projection distance*

$$\|\boldsymbol{\psi} - \beta\boldsymbol{\phi}(\mathbf{z})\|^2 \tag{4.24}$$

with respect to $\mathbf{z}$ and β (Figure 4.7). By setting the partial derivatives of (4.24) w.r.t $\mathbf{z}, \beta$ to zero, we derive a necessary condition for the optimal $\mathbf{z}, \beta$-combination:

$$\beta = \frac{\boldsymbol{\psi}^T \boldsymbol{\phi}(\mathbf{z})}{\|\phi(\mathbf{z})\|^2}. \tag{4.25}$$

If we substitute this into (4.24), we notice that maximizing

$$\frac{(\boldsymbol{\psi}^T \boldsymbol{\phi}(\mathbf{z}))^2}{\|\phi(\mathbf{z})\|^2}. \tag{4.26}$$

yields the same solution as minimizing (4.24), if β is recovered via (4.25). A solution to (4.26) can be found via gradient descent or similar off-the-shelf optimization methods.

If $\boldsymbol{\phi}(\mathbf{z}) = k(\mathbf{z}, \cdot)$, where k is gaussian kernel (2.14), we can solve (4.26) more conveniently. Observe that for all $\mathbf{z}$, $\|\boldsymbol{\phi}(\mathbf{z})\| = 1$, so (4.26) reduces to maximizing $(\boldsymbol{\psi}^T \boldsymbol{\phi}(\mathbf{z}))^2$, whose derivative with respect to $\mathbf{z}$, in turn, vanishes for

$$\mathbf{z} = \frac{\sum_i \alpha_i \exp(-\|\mathbf{x}_i - \mathbf{z}\|^2/(2\sigma^2)) \cdot \mathbf{x}_i}{\sum_i \alpha_i \exp(-\|\mathbf{x}_i - \mathbf{z}\|^2/(2\sigma^2))},$$

which can be casted into a fixed point iteration

$$\mathbf{z}_{k+1} = \frac{\sum_i \alpha_i \exp(-\|\mathbf{x}_i - \mathbf{z}_k\|^2/(2\sigma^2)) \cdot \mathbf{x}_i}{\sum_i \alpha_i \exp(-\|\mathbf{x}_i - \mathbf{z}_k\|^2/(2\sigma^2))}.$$

In practice, we start with a random pattern and iterate until convergence. Note that there is no guarantee that the method converges, in such cases we simply restart the iteration with a different random pattern.

4.6.3 Iterated Pre-Images

Back to our application, it turns out that it is generally not sufficient to approximate the decision normal with a single pre-image. This is due to the fact that for linear combinations of gaussians, exact pre-images do not exist and their approximations usually are very poor. Clearly, if a single support vector was sufficient, the data would be linearly separable in input space already. Hence, we approximate $\boldsymbol{\psi}$ by a *linear combination*

$$\boldsymbol{\psi}' = \sum_{i=1}^{m'} \beta_i \boldsymbol{\phi}(\mathbf{z}_i)$$

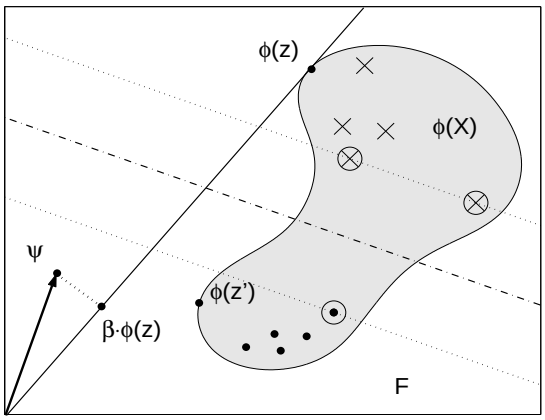

Figure 4.7: Pre-image approximation via minimization of the projection distance. The gray area denotes the image of X in feature space F (the origin being the lower right image corner). Additionally, we have plotted the feature-mapped training data of a toy problem and its optimal hyperplane (dash-dotted), whose normal is shown at the origin, pointing towards $\boldsymbol{\psi} = \sum y_i \alpha_i \boldsymbol{\phi}(\mathbf{x}_i)$. The pre-image of the point maximizing (4.24) is $\mathbf{z}$. In contrast, (4.23) is maximized by the obviously worse $\mathbf{z}'$.

such that

$$\|\boldsymbol{\psi} - \boldsymbol{\psi}'\|^2 \tag{4.27}$$

is minimized. The idea is to find a so-called *reduced set* of $\mathbf{z}_i$, which is more 'suitable' than the support vector set in the sense that $\boldsymbol{\psi}'$ gets very close to $\boldsymbol{\psi}$ whereas its expansion uses significantly fewer points.

At this point, we are left with another optimization problem, namely that of choosing the best reduced set, which is known to be NP-complete. In other words, given a set of possible expansion points, we have to compare all (exponentially many) possible subsets against each other in order to find the optimum. Even worse, in case of arbitrary pre-images, that is, if we want to choose *any* set of points, there are additional numerical difficulties (many dimensions, local minima).

Hence, we use a suboptimal but efficient greedy strategy called *iterated pre-images*: We initialize the target

$$\boldsymbol{\psi}_0 = \sum_{i=1}^{m} y_i \alpha_i \boldsymbol{\phi}(\mathbf{x}_i)$$

and an empty reduced set

$$R_0 = \emptyset,$$

which is successively extended in the following way: At iteration k, we add the best approximate pre-image (4.26) for the current $\boldsymbol{\psi}_k$

$$R_{k+1} = R_k \cup \{\arg\max_{\mathbf{z} \in X} \frac{\boldsymbol{\psi}_k^T \boldsymbol{\phi}(\mathbf{z})}{\|\boldsymbol{\phi}(\mathbf{z})\|^2}\}.$$

Then, we compute the optimal expansion coefficients $\boldsymbol{\beta}$ (4.28) for the linear subspace spanned by the points in R_{k+1}. For a given set of points $\mathbf{z}_i$, $i = 1 \ldots m'$, the optimal expansion

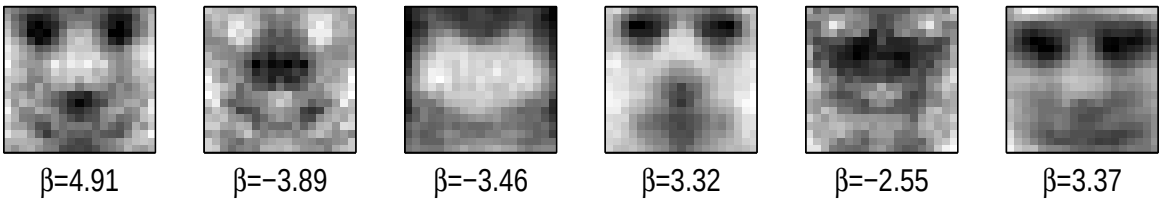

Figure 4.8: The first 6 pre-images. This figure shows the results from the iterated pre-image procedure described in section 4.6.3 applied to our face detection support vector machine. The corresponding optimal expansion coefficient (from the reduced set R_{100}) is shown below each image. Note how the sign of β corresponds to the respective side of the decision hyperplane that each pattern lies on (i.e. face/anti-face). Furthermore, the absolute value of β encodes how much (in terms of (4.27)) we gain from each expansion point.

coefficients β_i (in the sense of (4.27)) are computed as follows. We write down (4.27) with $\boldsymbol{\psi}$, $\boldsymbol{\psi}'$ replaced by their respective expansions and set its gradient with respect to $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_{m'})$ to zero. This leads to

$$\boldsymbol{\beta} = (K^{\mathbf{z}})^{-1} K^{\mathbf{xz}} \boldsymbol{\alpha} \tag{4.28}$$

where $K^{\mathbf{z}}_{ij} = k(\mathbf{z}_i, \mathbf{z}_j)$ and $K^{\mathbf{xz}} = k(\mathbf{x}_i, \mathbf{z}_j)$ are $m' \times m'$ and $m \times m'$ matrices respectively, and $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_m)$ [2]. As an aside, note that (4.25) is merely a special case of (4.28) for $m' = 1$ and $\boldsymbol{\alpha} = 1$.

With the optimal expansion coefficients $\boldsymbol{\beta}$ for the current R_{k+1} at hand, we may set $\boldsymbol{\psi}_{k+1}$ to the residual

$$\boldsymbol{\psi}_{k+1} = \sum_{i=1}^{m} y_i \alpha_i \boldsymbol{\phi}(\mathbf{x}_i) - \sum_{i=1}^{k} \beta_i \boldsymbol{\phi}(\mathbf{z}_i).$$

This is repeated until the reduced set has reached a certain size or until the length of the residuum falls below some threshold.

We applied the iterated pre-image method to our face detection SVM to compute the first 100 reduced set vectors. Figure 4.8 shows the corresponding image patches for $i = 1 \ldots 6$ together with their expansion coefficients β_i.

Based on these results, we would like to examine how the classification performance evolves through the different stages, that is, for different classifiers using different set sizes. As mentioned before, the face/non-face data are somewhat unbalanced and we therefore divide the test error into two values, namely the hit rate and the false positive rate. To this end, we have to be careful about the bias b: If we increase b, the decision (2.16) will be biased towards 'faces', i.e. the false negative rate will decrease but the false positive rate will

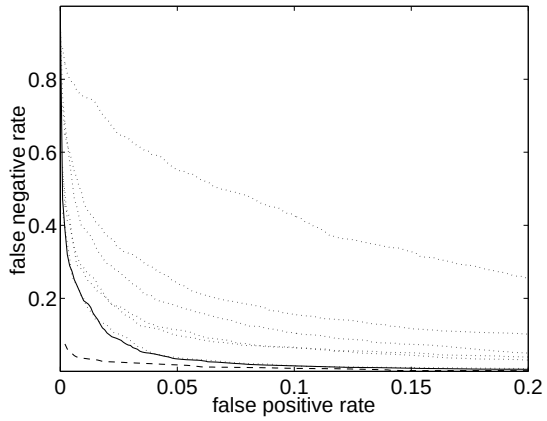

Figure 4.9: Receiver operator characteristics for various reduced set classifiers. The dotted lines correspond to the face detectors of size $m' = 1, 2, 5, 10, 20, 50$, respectively, whereas the solid line denotes the $m' = 100$ case. Furthermore, we have plotted the performance of the non-reduced, original classifier (dashed line). Remarkably, the $m' = 50$ version already allows for detection rates of $\approx 80\%$ at a false positive rate of $\approx 1\%$.

increase. Accordingly, a decrease in b will lead to an opposite result. Now instead of trying to readjust the biases in order to compare different classifiers, we introduce a new performance representation called *ROC* (receiver operator characteristic) curves [6]. ROC curves plot false negatives against false positives as shown in Figure 4.9. As we change b, the false positive/negative rate moves along the curve in the respective direction. Figure 4.9 compares the performances of classifiers with different reduced set sizes ($m' = \{1, 2, 5, 10, 20, 50, 100\}$) together with the characteristic of the original, full classifier. We notice that for increasing m', the performance reaches a reasonable level rather quickly and eventually converges to the original curve. On the other hand, a fast increasing m'-step size is required, if we want to keep the convergence rate constant.

4.6.4 Sequential Evaluation

Although we have significantly reduced the size of the solution, it is still prohibitive to use *as is*, since for real-time performance, we cannot afford more than a few (say, $5 \ldots 10$) support vectors per patch, even on state of the art machines (≈ 2.5GHz). We will therefore follow an approach proposed in [22].

In the so-called *sequential evaluation* scheme, we build a cascade of classifiers with increasing complexity. The fastest classifier, which uses only $m' = 1$ support vector, is applied first. During subsequent (more and more complex) stages, only those patches which have not been rejected by the preceding ones, are classified. For instance, we can adjust the bias of the $m' = 1$ classifier (Figure 4.9) such that it rejects $\approx 80\%$ of the non-face patches and yet recognizes over 70% of the face patches. This method has been shown to quickly sort out non-face patches and significantly improve the classification speed [22].

It is not clear how to set the biases in order to get a specific *overall* performance. In this

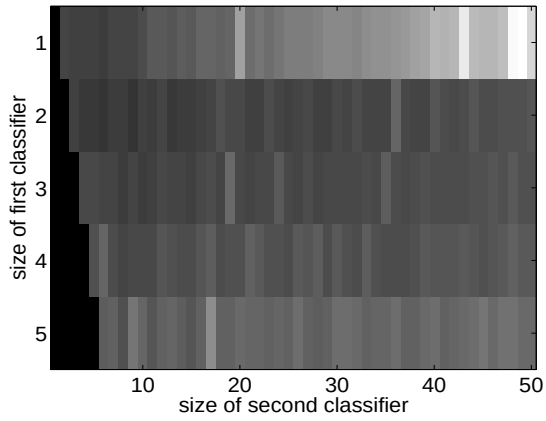

Figure 4.10: The reduced set size selection procedure. We have plotted the average computation time for cascades (of three classifiers) with reduced set sizes $\{1 \dots 5,\ 1 \dots 50,\ 50\}$. The best (lowest) value is attained at (2,6). Note that if the first classifier has already 4 or 5 support vectors, the computation time hardly depends on the complexity of the second. This means that a great amount of the negative patches are already rejected by classifiers of size ≈ 5.

work, we will not deal with this question in detail (for more information, see [22]). Instead, we merely adjust the biases so that the hit rate equals 90% within every stage. What remains is to fix the *length* of the cascade and the respective classifier *sizes*. Since our main goal is speed, we minimize the computation time with respect to these quantities. For the sake of simplicity, we only test cascades of length 3 where the size of the first classifier equals $1 \dots 5$ and the size of the last classifier is always 50.

Figure 4.10 shows the computation times with respect to the sizes of the first two reduced sets (averaged over 100 random test images and evaluated only within skin regions) for all different cascades that we have tested. Here, the lowest value ($4.4ms$) is achieved by the $(2, 6, 50)$ combination, so we select this cascade for our implementation.

4.7 Performance

We applied the face detection cascade to our test patches. It recognized 80.4% of the faces at a false positive rate of 1.8%. Obviously, the three different stages have slightly different notions of what is a face and what is not: each single classifier yields (by construction) a 90% hit rate, i.e. almost 10% more than the combination of the three. The loss comes from those faces that are not recognized by *every* single classifier independently. This shows how the sequential evaluation scheme sacrifices accuracy for speed.

Before we give the experimental results on whole images rather than face/non-face patches, some words about *scale* seem to be in order. Usually, one generates a pyramid of differently scaled versions of the test image and presents them to the classifier. That way, faces may be detected at arbitrary scale (*multi-scale* approach). However, in our application, the distance of the subject from the camera, and thus the scale, is known. This distance is bounded

from above by the depth resolution of our stereo camera system, and bounded from below by the finite size of the image (i.e. the subject's distance to the camera must exceed a certain threshold in order to him/her fit into the image). In our setting, this tradeoff distance is at $\approx 3m$. Hence, before applying the classifier, we merely adjust the scale of each image such that the face patch size (19x19) matches the expected average face size within the image. In particular, our camera parameters yield a scale factor of 0.8, which in turn leads to 256x192 images.

Another important issue in face detection systems is the *global* interpretation of the classifier output on a test image. If several face patterns have been detected, we have to distinguish between i) *redundant* detections, all coming from the same, single face and ii) *isolated* detections coming from different faces. Usually, people employ heuristic algorithms that sort out redundant (i.e. physically impossible) detections or train additional classifiers for this task. In our method, we use a very simple strategy: since we only allow exactly one face to appear within an image, the algorithm merely outputs the position with the highest SVM response.

Let us look at the overall computation time. In our C implementation, the re-scaling and rgb-to-gray transform take $0.7ms$ and $0.3ms$, respectively. Together with an average face detection time of $4.4ms$, we have $5.4ms$. Furthermore, since these statistics were generated by classifying only the skin colored regions, we have to add the required preprocessing time (see previous chapter), yielding a total of $9.7ms$ per frame. If we leave out the skin detection stage, the computation time goes up to $38ms$, justifying the preprocessing overhead.

Figure 4.11 illustrates the final algorithm applied to a test image. First, we apply the skin color classifier to the input image (top left) to get the skin map (top right). The input is then converted to gray scale and passed on to the first face detector (with 2 support vectors). Faces are only sought for in skin regions. This yields the left picture in middle row. The next two images show the sequential evaluation scheme: The two subsequent face classifiers are only applied to positions where the preceding output was positive. The bottom right image shows the input with a bounding box drawn at the position of maximal output of the final (with 50 support vectors) face classifier (bottom left).

Figure 4.11: The face detection method. The six images show different stages of the face detection process. For a detailed explanation, see text.

Part V

Conclusion

5.1 Discussion

We have implemented a software library for real time face detection. In our system, the key ingredients for a low computational complexity are

- a neural network driven preprocessing step for discarding image regions that are not skin-colored,
- an approximation technique for reducing the size of a sophisticated support vector machine solution, applied to our face classifier.

With neural networks and support vector machines, we focussed on algorithms which are general in the sense that they are based on both optimization (learning) and prior knowledge (modelling) rather than on heuristics alone. In contrast, we have seen that at some points (e.g. post-processing of the skin map) it is convenient and legitimate to use conventional techniques and so-called magic numbers (for filter widths, etc.) without regarding their optimality.

The algorithm is capable of detecting faces in 320x240 color images within roughly 10 milliseconds. It should be emphasized, though, that we designed our system to detect a single face at a single scale only. In order to get a multi-scale solution, we have to apply the face classifier to differently scaled versions of the input image. Fortunately, this extension is simple to implement and, if written carefully (scaling of the image/support vectors for larger/smaller scales), will not slow down the method too much. Moreover, our method can be easily extended such that it detects more than one face per image. This may be achieved via an additional, yet not time critical, mechanism (as described in [25]), which combines overlapping detections.

A possible drawback of our system is that we cannot compare it to other face detection systems via the standardized test sets, since the available training examples are very much biased towards the conditions in our lab (lighting, camera, only 15 different faces, etc.). As a result, the performance on a more general data set (e.g. CMU faces) will not be representative. Also, unlike in [23, 24, 25], we do not preprocess our patches, which makes a comparison even more difficult.

Still, we would like to compare the classification results within the *respective environments*. In [24], *Sung et al.* achieved 94.6% and only 2 false detections on their database (313 high-quality images containing one face each, implicitly yielding $4,669,960$ negative examples), based on clustering in connection with neural networks. The same database was used by *Osuna et al.* [23], who applied a 2nd degree polynomial kernel SVM. Their system detected 97.1% of the faces with 4 false alarms. Our non-reduced classifier yielded a 95.8% hit rate and 0.67% false alarms on our 'worst case' test set (cf. section 4.5). Please note that the

relatively high false positives rate (0.67% means 158 false detections) comes from the fact that our test set consists of mostly 'hard' non-face examples, i.e. non-face patches that almost look like faces, which occur less frequently in real-world images.

As for the reduced set/sequential evaluation method, we compare our results to those presented in [22] (*Romdhani et al.*). Here, the authors achieved 80.7/0.001% hits/false positives (on 130 images containing 507 faces). On the worst case test set, our system yielded 80.4/1.8%. In order to measure the performance in a more general setting (and compare it to [22]), we also classified our 1000 test *images*, each containing one face. Here, face is considered as correctly classified if the distance between the true face position and the maximum SVM output is less than three pixels, whereas any positive SVM output more than three pixels away from the true face position is regarded a false alarm. We achieved a 88%/0.082% rate. Compared to [22], the larger number of false positive is possibly due to the fact that we did not bootstrap the non-reduced SVM as extensively as *Romdhani et al.*. Moreover, their method uses a maximum of 100 support vectors, we only apply at most 50.

5.2 Further Work

We found that the learnt insensitivity to illumination/contrast changes is valid within a small range only. To remedy this, we could try to record the training data under more strongly varying lighting conditions or retrain the SVM on artificially varied copies of support vectors (*virtual* support vectors, see [2]). We could also try to switch from plain pixel values to more illumination invariant face features, such as edges or texture.

Another goal will be the development of a similar method for hand detection. The question here is how much of our present algorithm we can re-use for this task. While it is certainly helpful to use the skin color map, we suspect that the image resolution might be too low, as most of the hand structure (i.e. the fingers) is at a scale close to the pixel size. Moreover, it is questionable whether a patch based classifier like our face SVM is suitable for this problem. We might run into difficulties due to the large variety of possible shapes and the orientations compared to the face detection case.

Furthermore, we will need to implement an algorithm for shoulder and elbow detection. This is probably the most difficult task, since the regularities of shoulders and elbows are much more subtle and therefore require significantly larger patch sizes and/or much more training data, if not a completely different classification approach. To this end, it might be necessary to incorporate disparity information.

Finally, there is the tracking algorithm itself, for which the methods mentioned above provide the initial joint positions. It remains to be seen whether the joint detection code runs sufficiently fast. If this is the case, we may calculate the joint positions (together with

their real world coordinates) on every frame and track them with a simple kalman filter, for instance. However, if running the detection code turns out to be too time consuming, we may have to switch to faster (i.e. simpler) feature detectors for tracking purposes and only use our joint detection for restarting the tracker in case it gets lost.

5.3 Acknowledgements

I would like to thank Prof. Andreas Schilling, Dr. Matthias Franz and Prof. Bernhard Schölkopf for supervising this work and for giving me the opportunity to work in a very distinguished and, as I found, very pleasant environment at the Max Planck Institute. I also thank Dr. Kwang In Kim for his valuable advice on the most diverse subjects and Gökhan Bakır for the many hours of insightful discussion. Finally, I greatly appreciate the way all the lab members were willing to help me out whenever I had a question or whenever I needed subjects for my experiments.

Part VI

Appendix

6.1 Implementation Details

We have implemented all algorithms in Matlab and/or C for the code to be compatible with already existing software components. More precisely, the neural network and support vector training code is written in Matlab (using netlab [4] and libsvm [5]), whereas the classification algorithms are written in C for performance reasons. We wrapped the latter into a Matlab `mex` file, a mechanism which allow us to implement our methods efficiently (that is, in C) and yet invoke them directly via Matlab function calls (for details, see the Matlab website at www.mathworks.com).

The classification code is in `fdetect.dll`. It is accessed through two function calls, namely an initialization and a classification routine. The initialization code must be executed (once) in order to set up the classifiers. It has the following syntax

```
fdetect('init', skinnet, facesvm, scaling);
```

where `skinnet` and `facesvm` are cell arrays containing the parameters for the skin detection network (one hidden layer, feedforward) and the face detection SVM. They are initialized via

```
skinnet = {weight1, bias1, weight2, bias2};
```

and

```
facesvm = {nsv, sv, sigma, rssize, coef, bias};
```

respectively. The sizes of the `skinnet` components depends on the number of neurons in the hidden layer `h`. `weight1` is a `3x4` matrix, where `weight1(i,j)` denotes the synaptic weight from the ith input node (i.e. the ith color channel) to the jth hidden neuron. The biases for the hidden layer are stored in `bias1` which has dimension `1x4`. Similarly, the connection between the hidden neurons and the output node is encoded in `weight2` (`4x1`) and `bias2` (`1x1`). For the `facesvm` initialization, we have to provide the number of support vectors `nsv`, the support vectors themselves `sv` (`361 x nsv`), and the kernel width `sigma`. Note that the support vectors have to be in `uint8` format. Furthermore, for the sequential evaluation scheme, we add the sizes of the different classifiers together with their coefficients and biases in `rssize` (`3x1` cell array with scalar entries), `coef` (`1x3` cell array with coefficient vectors of size `rssize{i}x1`) and `bias` (`3x1` cell array with scalar entries).

The additional scalar parameter `scaling` is used for re-scaling input images in order to adjust the expected face size to roughly 19x19 pixels. The classification function is called via

```
[skinmap, facemap] = fdetect(image);
```

Here, `image` denotes the input image (RGB, `uint8` format, `width x height x 3`) and `facemap` is the classification result, that is, another image whose pixel values correspond to the the face SVM output at each image position. The post-processed neural network output (`skinmap`) is also provided for possible follow-up applications, e.g. hand or arm detection.

We wrote the dll code in Microsoft Visual C++ 6.0 under Windows XP. The image pipeline (Figure 4.11) starts with a 24 bit RGB (8 bits per channel) input provided by the `fdetect(image);` call. First, we re-scale the input by `scaling` using nearest neighbors. All following operations are performed at the resulting resolution. The implementation of the skin detection network is straightforward, i.e. we simply compute 2.7 for each hidden neuron (with `weight1, bias1`) and combine the results in the output neuron (with `weight2, bias2`). As mentioned already in the skin detection chapter, the postprocessing filters (median, open, or close) basically count the number of skin pixels within a 5x5 window which are subsequently compared to the respective constant (12, 1, or 24).

At each stage of the face detection cascade, we store the SVM output in a binary image which serves as a mask for the next classifier. The detection code itself is also straightforward, as it mainly involves the subtraction of two 19x19 image patches, a dot product, a multiplication (by $-1/2\sigma^2$) and one function evaluation (exp) for each image position and support vector (see 2.16). Note that the performance bottleneck here is *not* the exp function call, but the large number of memory accesses and subtractions/multiplications needed for the exp argument. Therefore, we compiled this part of the code with the Intel C++ Compiler 7.1 for Windows (available at www.intel.com/software/products/compilers), which is able to vectorize the code. In fact, this decreased the computation time by a noticeable amount.

The final face detection SVM stores its output without sgn, so that we can compute the position of maximal detection response in the output image `facemap`, yielding the most probable face coordinates.

Bibliography

[1] Nello Cristianini and John Shawe-Taylor, *An Introduction to Support Vector Machines*, Cambridge University Press, 2000.

[2] Bernhard Schölkopf and Alex Smola, *Learning With Kernels*, MIT Press, 2002.

[3] Simon Haykin, *Neural Networks*, 2nd Edition, Prentice Hall, 1999.

[4] Neural network toolbox, Neural Computing Research Group, Aston University, Birmingham, United Kingdom, http://www.ncrg.aston.ac.uk/netlab/

[5] libsvm - A Library for Support Vector Machines, Chih-Chung Chang and Chih-Jen Lin, www.csie.ntu.edu.tw/∼ cjlin/libsvm/

[6] Richard O. Duda, Peter E. Hart, David G. Stork, *Pattern Classification*, 2nd Edition, Wiley-Interscience,2001.

[7] Bernhard Schölkopf, *Support Vector Learning*, Oldenburg, 1997.

[8] SPIDER, Object Oriented Machine Learning Library, http://www.kyb.tuebingen.mpg.de/bs/people/spider/index.html.

[9] Olivier Chapelle, Vladimir Vapnik, Olivier Bousquet, Sayan Mukherjee, *Chosing Multiple Parameters for Support Vector Machines*, 2001.

[10] Vladimir Vapnik, Olivier Chapelle, *Bounds on Error Expectation for Support Vector Machines*, from *Advances in Large Margin Classifiers* by Smola, Bartlett, Schölkopf and Schuumans, MIT Press, 2000.

[11] Jie Yang, Weier Lu, Alex Waibel, *Skin-Color Modeling and Adaptation*, 1997

[12] Ming-Hsuan Yang, Narendra Ahuja, *Detecting Human Faces in Color Images*, 1998

[13] Min C. Shin, Kyong I. Chang, Leonid V. Tsap, *Does Colorspace Transformation Make Any Difference On Skin Detection?*, 2002

[14] J. Birgitta Martinkauppi, Maricor N. Soriano, Mika H. Laaksonen, *Behavior of skin color under varying illumination seen by different cameras at different color spaces*, 1997

[15] Christophe Garcia, Georgios Tziritas, *Face Detection Using Quantized Skin Color Regions Merging and Wavelet Packet Analysis*, IEEE Transactions on Multimedia, September 1999

[16] Rein-Lien Hsu, Mohamed Abdel-Mottaleb, Anil K. Jain, *Face Detection in Color Images*, 2002

[17] Michael J. Jones, James M. Rehg, *Statistical Color Models with Application to Skin Detection*, Cambridge Research Laboratory, Technical Report Series CRL 98/11, December 1998

[18] Hannes Kruppa, Martin A. Bauer, Bernt Schiele, *Skin Patch Detection in Real-World Images*, ETH Zürich, 2002

[19] Jean-Christophe Terrillon, Shigeru Akamatsu, *Comparative Performance of Different Chrominance Spaces for Color Segmentation and Detection of Human Faces in Complex Scene Images*, ATR Human Information Processing Laboratories, Kyoto, Japan, 2000

[20] Jean-Christophe Terrillon, Martin David, Shigeru Akamatsu, *Automatic Detection of Human Faces in Natural Scene Images by Use of a Skin Color Model and of Invariant Moments*, ATR Human Information Processing Laboratories, Kyoto, Japan, 1998

[21] Chris J.C. Burges, *Simplified Support Vector Decision Rules*, Bell Laboratories, Lucent Technologies, 1996

[22] Sami Romdhani, Philip Torr, Bernhard Schölkopf, Andrew Blake, *Computationally Efficient Face Detection*, Microsoft Research Ltd., ICCV 2001

[23] Edgar Osuna, Robert Freund, Federico Girosi, *Training Support Vector Machines: An Application To Face Detection*, MIT, Cambridge, 1997

[24] Kah-Kay Sung and Tomaso Poggio, *Example-Based Learning for View-Based Human Face Detection*, IEEE transactions on pattern analysis and machine intelligence, Vol. 20, No. 1, January 1998

[25] Henry A. Rowley, Shumeet Baluja, Takeo Kanade, *Neural Network-Based Face Detection*, Computer Vision and Pattern Recognition, 1996.

[26] Bern Heisele, Tomaso Poggio, Massimiliano Pontil, *Face Detection in Still Gray Images*, MIT, 2000

[27] MIT CBCS Face Database, http://www.ai.mit.edu/projects/cbcl/software-datasets

[28] Learning of sensory-motor control strategies for a vision-guided robot arm using kernel methods. G. Bakir, Robotics Lab, MPI Tuebingen, Germany, www.kyb.tuebingen.mpg.de/bs/people/gb/index.html