Recognition of Handwritten digits and mathematical expressions

Thesis submitted to
Visvesvaraya National Institute of Technology, Nagpur
In partial fulfillment of requirement for the award of
degree of

Bachelor of Technology
(Electronics and Communication Engineering)

by

Juhily Ashtikar **(BT12ECE005)**

Kandula Soujanya Priya **(BT12ECE025)**

Nihal Gandhi **(BT12ECE052)**

Sanskar Jhanwar **(BT12ECE069)**

Guide

Dr. Joydeep Sengupta

Department of Electronics and Communication Engineering
Visvesvaraya National Institute of Technology
Nagpur 440 010 (India)

May 2016

Declaration

We, hereby declare that the thesis titled "**Recognition of handwritten digits and mathematical expressions"** submitted herein has been carried out by us in the Department of Electronics and Communication Engineering of Visvesvaraya National Institute of Technology, Nagpur. The work is original and has not been submitted earlier as a whole or in part for the award of any degree / diploma at this or any other Institution / University.

JUHILY ASHTIKAR

KANDULA SOUJANYA PRIYA

NIHAL GANDHI

SANSKAR JHANWAR

Date:

Certificate

The thesis titled "**Recognition of handwritten digits and mathematical expressions"** submitted by

Ms. Juhily Ashtikar (BT12ECE005)

Ms. Kandula Soujanya Priya (BT12ECE025)

Mr. Nihal Gandhi (BT12ECE052)

Mr. Sanskar Jhanwar (BT12ECE069)

For the award of degree of Bachelor of Technology, has been carried out under my supervision at the Department of Electronics and Communication Engineering of Visvesvaraya National Institute of Technology, Nagpur. The work is comprehensive, complete and fit for evaluation.

Dr. Joydeep Sengupta
Assistant Professor,
Department of Electronics and
Communication Engineering,
VNIT, Nagpur

Head,
Department of Electronics and
Communication Engineering,
VNIT, Nagpur

Date:

ACKNOWLEDGEMENT

I take this opportunity to acknowledge with deep sense of gratitude towards my project guide, Dr. Joydeep Sengupta, Assistant Professor, Department of Electronics and Communication Engineering, VNIT, Nagpur, for his invaluable guidance, constant motivation and continuous support which has led to the successful completion of this project. I would also like to express my sincere thanks to Mr. Maharshi Gor for his continuous support and guidance throughout the year. I would also like to thank all the teaching and non-teaching staff of the department which has been directly or indirectly helpful to us. Finally I would like to thank friends and family for their constant support and encouragement. We also thank all those who have directly or indirectly contributed to this project. Their contribution has been very vital.

JUHILY ASHTIKAR

KANDULA SOUJANYA PRIYA

NIHAL GANDHI

SANSKAR JHANWAR

ABSTRACT

Nowadays, the need for a user friendly system is increasing exponentially. It has been two decades since there has been no significant change in the old scientific calculator except for its hardware and speed. Our approach is to change the way the calculations are performed. These days, with the increasing use of smart phones, smarter ways of performing calculations can be achieved. Many-a-times people want to eliminate the extra step of entering the values into calculator to compute faster. In this work, the goal is to eliminate the current interface between handwritten information on paper and electronic media and automate the process to bring about large potential savings in terms of manual effort and speed. We introduce a system, which efficiently digitizes handwritten mathematical expressions and encompasses a set of algorithms for preprocessing, feature extraction and classification and thus, predicting and solving the expressions the features represent with considerable accuracy. The system can further be improvised to achieve a fully functional and accurate handwritten mathematical calculator.

LIST OF FIGURES

LIST OF ACRONYMS

ANN	Artificial Neural Networks
C-HOG	Circular Histogram of Oriented Gradients
DET	Detail
DNA	Deoxyribo Nucleic Acid
Exp	Exponential
Fig	Figure
FPPW	False Positives Per Window
HD	Hamming Distance
HOG	Histogram of Oriented Gradients
KNN	K Nearest Neighbour
MNIST	Mixed National Institute of Standards and Technology
NaN	Not a Number
OCR	Optical Character Recognition
OS	Operating System
PDA	Personal Digital Assistant
RGB	Red Green Blue
R-HOG	Rectangular Histogram of Oriented Gradients
SIFT	Scale Invariant Feature Transformation
SVM	Support Vector Machines
Sqrt	Square Root

CONTENTS

CHAPTER-1

INTRODUCTION

Description of overall model

Recognizing handwritten expressions isn't easy. Rather, we humans are stupendously, astoundingly good at making sense of what our eyes show us. But nearly all that work is done unconsciously. And so we don't usually appreciate how tough a problem our visual systems solve. The difficulty of visual pattern recognition becomes apparent if you attempt to write a computer program to recognize digits like those above. What seems easy when we do it ourselves suddenly becomes extremely difficult. Simple intuitions about how we recognize shapes - "a 9 has a loop at the top, and a vertical stroke in the bottom right" - turn out to be not so simple to express algorithmically. When you try to make such rules precise, you quickly get lost in a morass of exceptions and caveats and special cases.

Optical character recognition, usually abbreviated as OCR, represents the mechanical or electronic translation of images containing handwritten, typewritten or printed text (usually captured by a scanner) into computer editable text.

Handwriting recognition is the ability of a computer to receive and interpret intelligible handwritten text from sources such as paper, photographs, touch-screen and other devices. The image of the text written can be processed “off-line”, from a paper, by optical scanning (optical character recognition) or by intelligent word recognition. Alternatively, the movement of the writing instrument can be processed “on-line”, for instance, by a touch screen with stylus.

Handwriting recognition mainly entails optical character recognition. However, a complete handwriting recognition system also handles formatting, performs correct segmentation into characters and finds the most plausible words. On-line handwriting recognition involves the automatic conversion of text, as it is written, on a special digital device or PDA, where a sensor captures the movements of the writing instrument and also the pen-up/pen-down switching. This kind of data is known as digital ink and can be regarded as a dynamic representation of handwriting. The obtained signal is converted into letter codes which can be used by the computer and in text processing applications. On-line character recognition is sometimes confused with optical character recognition. Off-line handwriting recognition involves the automatic conversion of text from an image into letter codes that can be used within a computer or other text processing applications. The data obtained in this form is regarded as a static representation of handwriting. This technology is successfully used in businesses that process a large amount of handwritten documents, such as insurance companies. The off-line handwriting recognition is difficult, because different people have

different handwriting styles. Nevertheless, limiting the types of input data allows improving the recognition process. Real time on-line handwriting recognition systems became well known as commercial products in the last few years. Among these systems can be enumerated the input devices for personal digital assistants, such as those running Palm OS. The Apple Newton pioneered this product. The algorithms used in these devices have the advantage that the order, speed and dimension of individual lines segments at the input are known. Also, the user can be constrained to use just certain shapes for the letters. These methods cannot be used in software scanning paper documents, so that the accurate recognition of handwritten documents is still an open problem. Accuracy rates of 80% to 90% for neat, clean characters can be achieved, but this rate still translates to dozens of errors per page, making the technology useful only in very limited applications.

A fair amount of work has been done thus far towards automatic human handwriting recognition. Digit recognition has been well-studied using the MNIST dataset, and its classification approximates human performance.

Neural networks approach the problem in a different way. The idea is to take a large number of handwritten digits, known as training examples, and then develop a system which can learn from those training examples. In other words, the neural network uses the examples to automatically infer rules for recognizing handwritten digits. Furthermore, by increasing the number of training examples, the network can learn more about handwriting, and so improve its accuracy. Perhaps we could build a better handwriting recognizer by using thousands or even millions or billions of training examples.

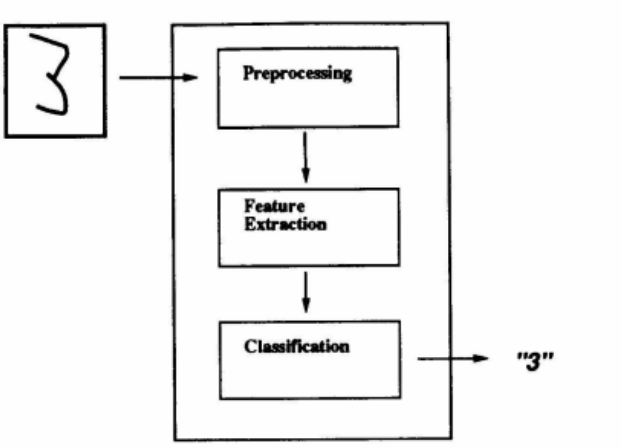


Figure 1.1- a basic model

1.1 Description of overall model

Our model works in three steps: 1) Preprocessing, 2) HOG features extraction and 3) Support vector machines classification.

In the preprocessing, we have some basic image processing to separate numbers from real samples or preparing data from dataset (which is reshaped from images to the vectors) and then in the second part, we extract HOG features which is very distinguishable descriptor for digits recognition where we divide an input image into 9×9 cells and compute then the histogram of gradient orientations thereby we represent each digit with a vector of 81 features. And finally in the third stage, a linear multiclass support vector machine has been employed to classify digits. In the last years, Support Vector Machines (SVMs) have gained a lot of attention of machine learning and pattern recognition communities. They have been successfully applied to several different areas ranging from face verification and recognition, speaker verification, text categorization, prediction, image retrieval, and handwriting recognition. The overall view of proposed approach has been illustrated in Figure 2. In general, the main contribution of our model is employing HOG features with SVM. HOG is a fast and reliable descriptor which can performs distinguishable features. And SVM is also a fast and powerful classifier that can be useful to classify HOG features. The most important benefit of above structure is that our model is fast and useful for real-time applications.

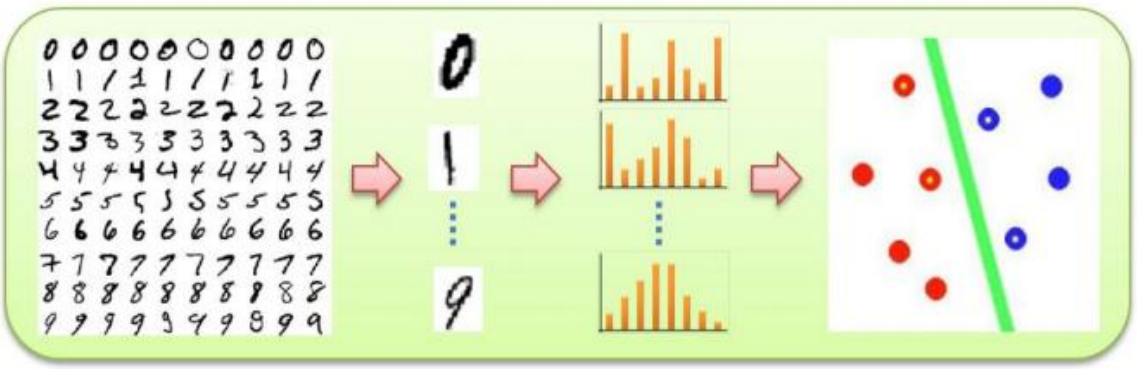


Figure1.2- The overall structure of our model

CHAPTER-2

SUPERVISED LEARNING

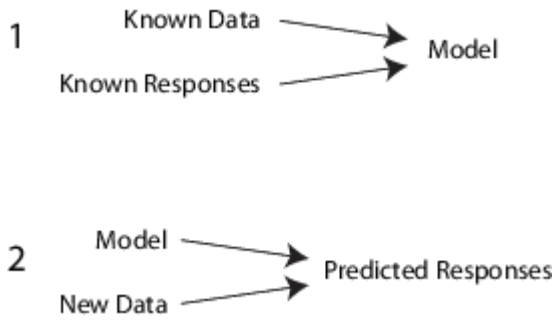
What is Supervised Learning?

Steps in Supervised Learning

2.1 What is Supervised Learning?

The aim of supervised, machine learning is to build a model that makes predictions based on evidence in the presence of uncertainty. As adaptive algorithms identify patterns in data, a computer "learns" from the observations. When exposed to more observations, the computer improves its predictive performance.

Specifically, a supervised learning algorithm takes a known set of input data and known responses to the data (output), and trains a model to generate reasonable predictions for the response to new data.



For example, suppose you want to predict whether someone will have a heart attack within a year. You have a set of data on previous patients, including age, weight, height, blood pressure, etc. You know whether the previous patients had heart attacks within a year of their measurements. So, the problem is combining all the existing data into a model that can predict whether a new person will have a heart attack within a year.

You can think of the entire set of input data as a heterogeneous matrix. Rows of the matrix are called observations, examples, or instances, and each contain a set of measurements for a subject (patients in the example). Columns of the matrix are called predictors, attributes, or features, and each are variables representing a measurement taken on every subject (age, weight, height, etc. in the example). You can think of the response data as a column vector where each row contains the output of the corresponding observation in the input data (whether the patient had a heart attack). To fit or train a supervised learning model, choose an appropriate algorithm, and then pass the input and response data to it.

Supervised learning splits into two broad categories: **Classification** and **Regression**.

- In classification, the goal is to assign a class (or label) from a finite set of classes to an observation. That is, responses are categorical variables. Applications include spam filters, advertisement recommendation systems, and image and speech recognition. Predicting whether a patient will have a heart attack within a year is a classification problem, and the possible classes are true and false. Classification algorithms usually apply to nominal response values. However, some algorithms can accommodate ordinal classes.
- In regression, the goal is to predict a continuous measurement for an observation. That is, the responses variables are real numbers. Applications include forecasting stock prices, energy consumption, or disease incidence.

2.2 Steps in Supervised Learning

Statistics and Machine Learning Toolbox supervised learning functionalities comprise a stream-lined, object framework. You can efficiently train a variety of algorithms, combine models into an ensemble, assess model performances, cross-validate, and predict responses for new data.

While there are many Statistics and Machine Learning Toolbox algorithms for supervised learning, most use the same basic workflow for obtaining a predictor model. The steps for supervised learning are:

- Prepare Data
- Choose an Algorithm
- Choose a Validation Method
- Examine Fit and Update Until Satisfied

2.2.1 Prepare Data

All supervised learning methods start with an input data matrix, usually called X here. Each row of X represents one observation. Each column of X represents one variable, or predictor. Represent missing entries with NaN values in X. Statistics and Machine Learning Toolbox supervised learning algorithms can handle NaN values, either by ignoring them or by ignoring any row with a NaN value.

You can use various data types for response data Y. Each element in Y represents the response to the corresponding row of X. Observations with missing Y data are ignored.

- For regression, Y must be a numeric vector with the same number of elements as the number of rows of X.
- For classification, Y can be any of the data types amongst numeric vector, categorical vector, character array, cell array of character vectors and logical vector.

2.2.2 Choose an Algorithm

There are tradeoffs between several characteristics of algorithms, such as:

- Speed of training
- Memory usage
- Predictive accuracy on new data
- Transparency or interpretability, meaning how easily you can understand the reasons an algorithm makes its predictions

2.2.3 Choose a Validation Method

The three main methods to examine the accuracy of the resulting fitted model are:

- Examine the re-substitution error.
- Examine the cross-validation error.
- Examine the out-of-bag error for bagged decision trees. For examples, see:

2.2.4 Examine Fit and Update Until Satisfied

After validating the model, you might want to change it for better accuracy, better speed, or to use less memory.

- Change fitting parameters to try to get a more accurate model.
- Change fitting parameters to try to get a smaller model. This sometimes gives a model with more accuracy.
- Try a different algorithm.

2.3 Methods of Supervised Learning

There are numerous supervised learning approaches and algorithms. When considering a new application, the engineer can compare multiple learning algorithms and experimentally determine which one works best on the problem at hand. Tuning the performance of a learning algorithm can be very time-consuming. Given fixed resources, it is often better to spend more time collecting additional training data and more informative features than it is to spend extra time tuning the learning algorithms.

For the present application, the most suitable approaches are –

1. Artificial Neural Networks
2. Support Vector Machines

CHAPTER-3

NEURAL NETWORKS

What is Artificial Neural Networks?

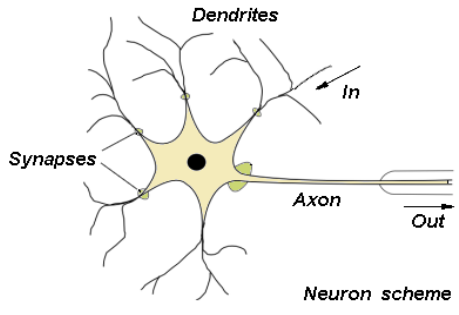
Auto Associative Memory

Backpropagation Algorithm

3.1 What is Artificial Neural Networks?

Artificial neural networks (**ANNs**) are a family of models inspired by biological neural networks and are used to estimate or approximate functions that can depend on a large number of inputs and are generally unknown. Artificial neural networks are generally presented as systems of interconnected "neurons" which exchange messages between each other. The connections have numeric weights that can be tuned based on experience, making neural nets adaptive to inputs and capable of learning.

Analogy between Biological Neuron and Artificial Neuron



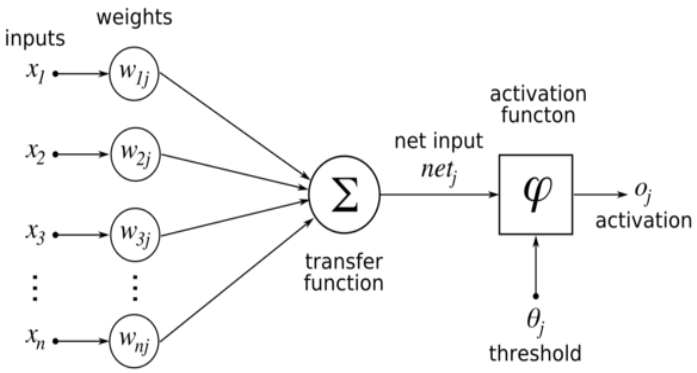


Figure 3.1 - Biological Neuron

Figure 3.2 - Artificial Neuron

An ANN is typically defined by 3 types of parameters: -

- The interconnection pattern between the different layers of neurons.
- The learning process for updating the weights of the interconnections.
- The activation function that converts a neuron's weighted input to its output activation.

Neural Networks can use a variety of topologies and learning algorithms:

- Auto-Associative Memory
- Backpropagation Algorithm

3.2 Auto Associative Memory

It is a feed-forward type network where the output is produced in a single feed-feedback computation. The model consists of 2 layers of processing units, one serving as the input layer while the other as output layer. The inputs are directly connected to the outputs, via a series of weights. The links carrying weights connects every input to every output. The sum of the products of the weights and the inputs is calculated in each neuron node.

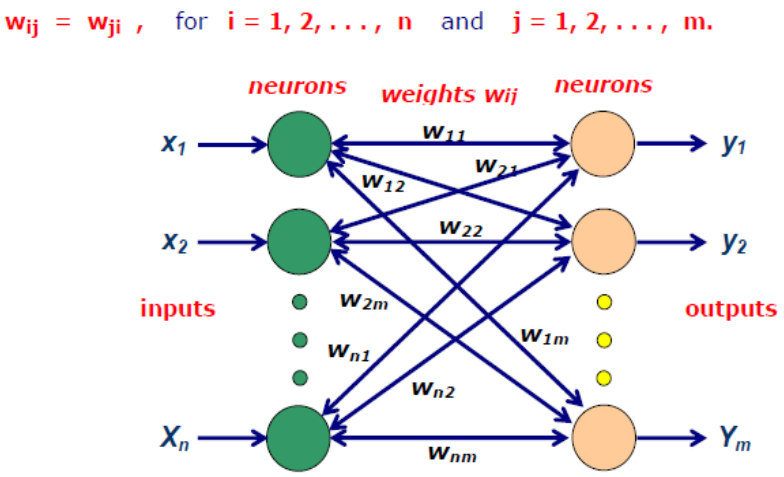


Figure 3.3 - The network architecture of the linear associator

Consider the three bipolar patterns A_1, A_2, A_3 to be stored as an auto-correlator.

$$A_1 = [-1 \quad 1 \quad -1 \quad 1]$$

$$A_2 = [\ 1 \quad 1 \quad 1 \quad -1]$$

$$A_3 = [-1 \quad -1 \quad -1 \quad 1]$$

Thus, the outer products of each of these three A_1, A_2, A_3 bipolar patterns are:

$$[A_1]^T_{4x1}\,[A_1]_{1x4} = \begin{Bmatrix} 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \\ 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \end{Bmatrix}$$

$$[A_2]^T_{4x1}\,[A_2]_{1x4} = \begin{Bmatrix} 1 & 1 & 1 & -1 \\ 1 & 1 & 1 & -1 \\ 1 & 1 & 1 & -1 \\ -1 & -1 & -1 & 1 \end{Bmatrix}$$

$$[A_3]^T_{4x1}\,[A_3]_{1x4} = \begin{Bmatrix} 1 & 1 & 1 & -1 \\ 1 & 1 & 1 & -1 \\ 1 & 1 & 1 & -1 \\ -1 & -1 & -1 & 1 \end{Bmatrix}$$

Therefore the connection matrix is

$$T = [t_{ij}] = \sum_{i=1}^{3} [A_i^T]_{4x1}\,[A_i]_{1x4} \quad \begin{Bmatrix} 3 & 1 & 3 & -3 \\ 1 & 3 & 1 & -1 \\ 3 & 1 & 3 & -3 \\ -3 & -1 & -3 & 3 \end{Bmatrix}$$

This is how the patterns are stored.

Consider a vector A' = (1, 1, 1, 1) which is a noisy presentation of one among the stored patterns.

- Find the proximity of the noisy vector to the stored patterns using Hamming distance measure.
- Note that the Hamming distance (HD) of a vector **X** from **Y**, where X = (x1, x2, …, xn) and Y = (y1, y2, …, yn) is given by
  HD (x, y) = $\sum_{i=1}^{m} |(x_i - y_i)|$

The HDs of A' from each of the stored patterns A_1, A_2, A_3 are

HD (A', A_1) = $\sum |(x_1 - y_1)|, |(x_2 - y_2)|, |(x_3 - y_3)|, |(x_4 - y_4)|$

= $\sum \left|\left(1 - (-1)\right)\right|, |(1 - 1)|, \left|\left(1 - (-1)\right)\right|, |(1 - 1)|$

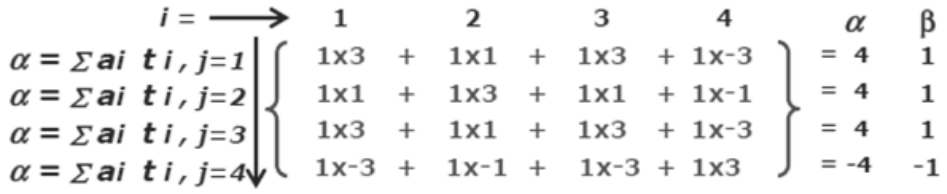
= 4

HD (A', A_2) = 6

HD (A', A_3) = 2

Therefore the vector A' is closest to A_2 and so resembles it. In other words the vector A' is a noisy version of vector A_2.

Computation of recall equation using vector A' yields:



Therefore $a_j^{new} = f\left(a_i\ t_{ij}\ a_j^{old}\right) \forall j = 1, 2, \ldots, p\ \ is\ \ f(\alpha, \beta)$

$$a_1^{new} = f(4, 1)$$

$$a_2^{new} = f(4, 1)$$

$$a_3^{new} = f(4, 1)$$

$$a_4^{new} = f(-4, -1)$$

The values of β is the vector pattern (1, 1, 1, -1) which is A_2.

3.3 Backpropagation algorithm

Backpropagation is a method of training artificial neural networks used in conjunction with an optimization method such as gradient descent. The method calculates the gradient of a loss function with respect to all the weights in the network. The gradient is fed to the optimization method which in turn uses it to update the weights, in an attempt to minimize the loss function.

Backpropagation requires a known, desired output for each input value in order to calculate the loss function gradient. It is therefore usually considered to be a supervised learning method, although it is also used in some unsupervised networks such as autoencoders. It is a generalization of the delta rule to multi-layered feedforward networks, made possible by using the chain rule to iteratively compute gradients for each layer. Backpropagation requires that the activation function used by the artificial neurons (or "nodes") be differentiable.

Intuition

Gradient descent algorithm

Linear Regression Model

$$h_\theta(x) = \theta_0 + \theta_1 x$$

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$

repeat until convergence {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

$$(\text{for } j = 1 \text{ and } j = 0)$$

}

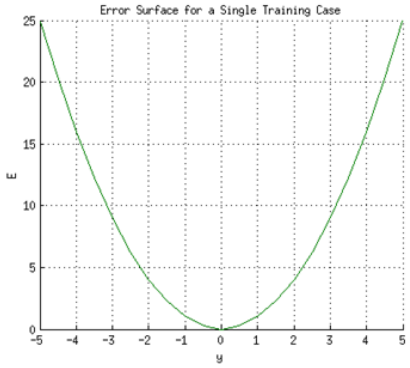


Figure 3.4 - Backpropagation

The Backpropagation learning algorithm can be divided into two phases:

Phase 1: Propagation

Each propagation involves the following steps:

1. Forward propagation of a training pattern's input through the neural network in order to generate the propagation's output activations.
2. Backward propagation of the propagation's output activations through the neural network using the training pattern target in order to generate the deltas (the difference between the input and output values) of all output and hidden neurons.

Phase 2: Weight update

For each weight-synapse follow the following steps:

1. Multiply its output delta and input activation to get the gradient of the weight.
2. Subtract a ratio (percentage) of the gradient from the weight.
3. This ratio (percentage) influences the speed and quality of learning; it is called the learning rate.

The greater the ratio, the faster the neuron trains; the lower the ratio, the more accurate the training is. The sign of the gradient of a weight indicates where the error is increasing; this is why the weight must be updated in the opposite direction.

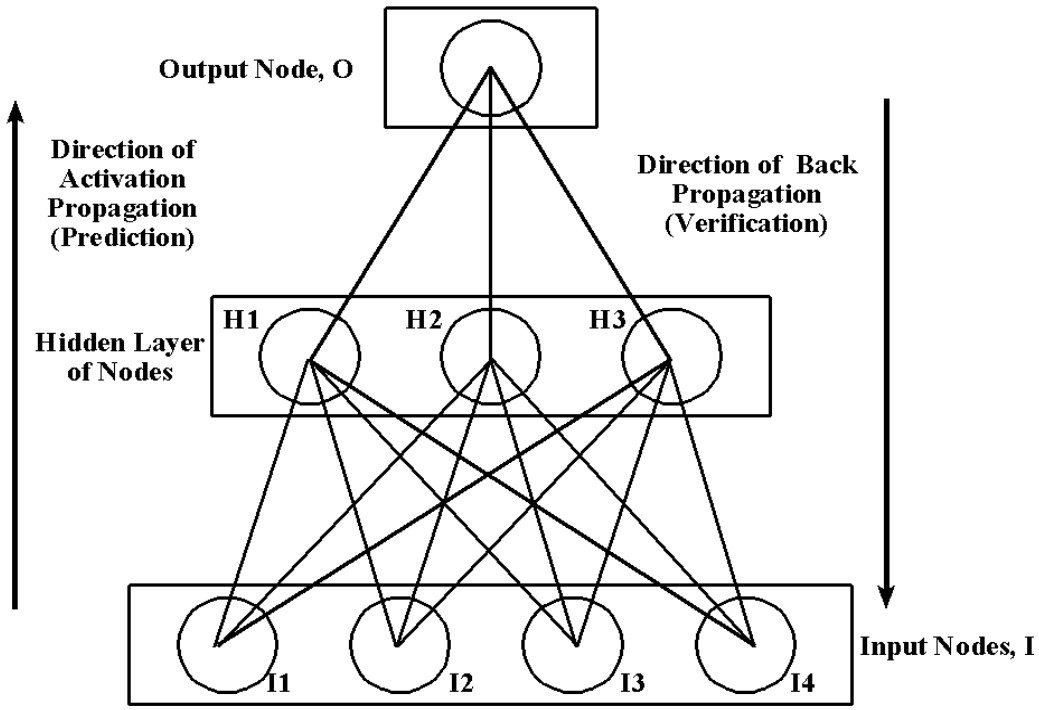


Figure 3.5 - Backward Propagation of errors

CHAPTER-4

SUPPORT VECTOR MACHINE

What is SVM?

Linear Classifier

Large Margin Classifiers

Kernels: from Linear to Non-Linear Classifiers

Understanding the effects of SVM and Kernel Parameters

Model Selection

4.1 What is SVM ?

The Support Vector Machine (SVM) is a widely used classifier. And yet, obtaining the best results with SVMs requires an understanding of their workings and the various ways a user can influence their accuracy. The Support Vector Machine (SVM) is a state-of-the-art classification method introduced in 1992 by Boser, Guyon, and Vapnik. The SVM classifier is widely used in bioinformatics (and other disciplines) due to its high accuracy, ability to deal with high-dimensional data such as gene expression, and flexibility in modeling diverse sources of data.

SVMs belong to the general category of *kernel methods*. A kernel method is an algorithm that depends on the data only through dot-products. When this is the case, the dot product can be replaced by a *kernel function* which computes a dot product in some possibly high dimensional feature space. This has two advantages: First, the ability to generate non-linear decision boundaries using methods designed for linear classifiers. Second, the use of kernel functions allows the user to apply a classifier to data that have no obvious fixed-dimensional vector space representation. The prime example of such data in bioinformatics are sequence, either DNA or protein, and protein structure.

Using SVMs effectively requires an understanding of how they work. When training an SVM the practitioner needs to make a number of decisions: how to preprocess the data, what kernel to use, and finally, setting the parameters of the SVM and the kernel. Uninformed choices may result in severely reduced performance.

4.2 Linear Classifiers

Support vector machines are an example of a linear two-class classifier. This section explains what that means. The data for a two class learning problem consists of objects labeled with one of two labels corresponding to the two classes; for convenience we assume the labels are +1 (positive examples) or -1 (negative examples). In what follows boldface **x** denotes a vector with components x_i. The notation x_i will denote the i^{th} vector in a dataset$\{(xi; yi)\}_{i=1}^{n}$, where y_i is the label associated with x_i. The objects x_i are called patterns or examples. We assume the examples belong to some set *X*. Initially we assume the examples are vectors, but once we introduce kernels this assumption will be relaxed, at which point they could be any continuous/discrete object (e.g. a protein/DNA sequence or protein structure).

A key concept required for defining a linear classifier is the *dot product* between two vectors, also referred to as an *inner product* or *scalar product*, defined as $w^T x = \sum_i w_i x_i$. A linear classifier is based on a linear *discriminant function* of the form

$$f(x) = w^T x + b. \quad (1)$$

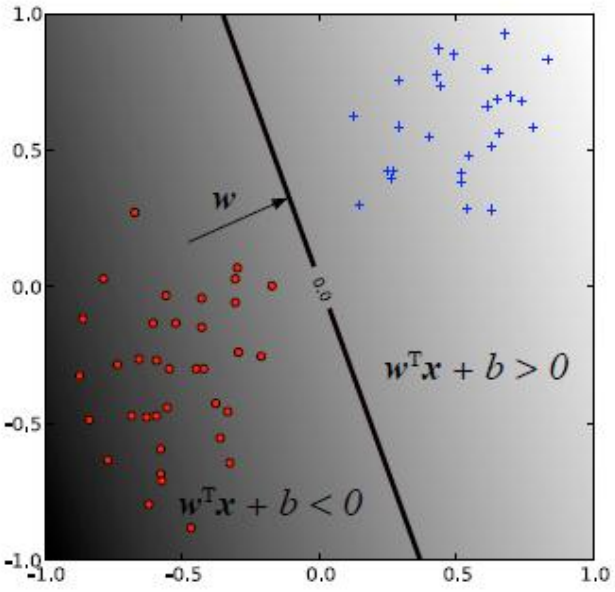


Figure 4.1- A linear classifier

The vector **w** is known as the *weight vector*, and b is called the bias. Consider the case b = 0 first. The set of points x such that $w^T x = 0$ are all points that are perpendicular to w and go through the origin - a line in two dimensions, a plane in three dimensions, and more generally, a *hyperplane*. The bias *b* translates the hyperplane away from the origin. The hyperplane

$$\{x : f(x) = w^T x + b = 0\} \quad (2)$$

divides the space into two: the sign of the discriminant function f(x) denotes the side of the hyperplane a point is on. The boundary between regions classified as positive and negativeis called the *decision boundary* of the classifier. The decision boundary defined by a hyperplane is said to be *linear* because it is linear in the input examples. A classifier with a linear decision boundary is called a linear classifier. Conversely, when the decision boundary of a classifier depends on the data in a non-linear way, the classifier is said to be non-linear.

4.3 Large margin classifiers

In what follows we use the term linearly separable to denote data for which there exists a linear decision boundary that separates positive from negative examples. Initially we will assume linearly separable data, and later indicate how to handle data that is not linearly separable.

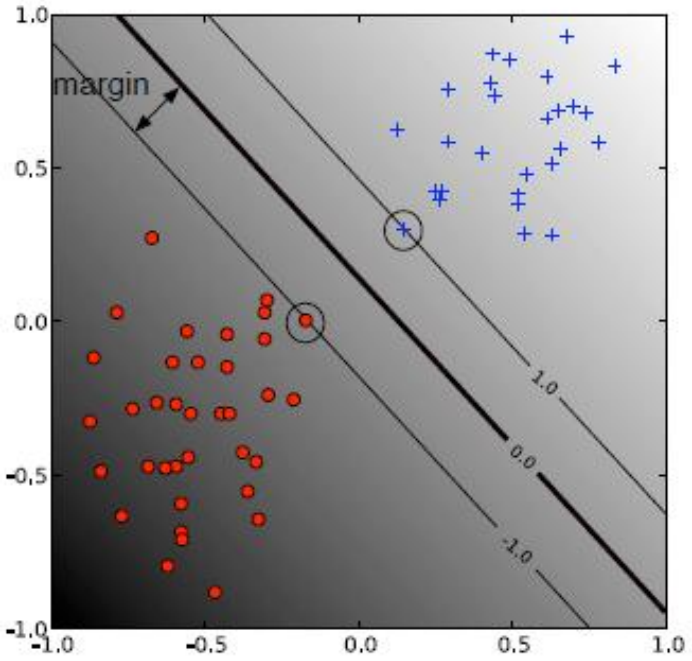


Figure 4.2- A linear SVM

4.3.1The Geometric Margin

In this section we define the notion of a margin. For a given hyperplane we denote by $x_+(x_-)$ the closest point to the hyperplane among the positive (negative) examples. The *norm* of a vector **w** denoted by ||w|| is its length, and is given by$\sqrt{w^T w}$. A unit vector $\hat{w}$ in the direction of w is given by w = ||w|| and has ||w|| = 1. From simple geometric considerations the margin of a hyperplane f with respect to a dataset D can be seen to be:

$$m_D(f) = \frac{1}{2}\hat{w}^T(x_+ - x_-) \qquad (3)$$

where $\hat{w}$ is a unit vector in the direction of w, and we assume that x_+ and x_- are equidistant from the decision boundary i.e.

$$f(x_+) = w^T x_+ + b = a$$
$$f(x_-) = w^T x_- + b = -a \qquad (4)$$

for some constant a > 0. Note that multiplying our data points by a fixed number will increase the margin by the same amount, whereas in reality, the margin hasn't really changed - we just changed the "units" with which we measure it. To make the geometric margin meaningful we fix the value of the decision function at the points closest to the hyperplane, and set a = 1 in Eqn. (8). Adding the two equations and dividing by ||w|| we obtain:

$$m_D(f) = \frac{1}{2}\hat{w}^T(x_+ - x_-) = \frac{1}{||w||}$$

4.3.2 Support Vector Machines

Now that we have the concept of a margin we can formulate the maximum margin classifier. We will first define the hard margin SVM, applicable to a linearly separable dataset, and then modify it to handle non-separable data. The maximum margin classifier is the discriminant function that maximizes the geometric margin 1/||w|| which is equivalent to minimizing $||w||^2$. This leads to the following constrained optimization problem:

$$\underset{w,\,b}{\text{minimize}} \quad \frac{1}{2}||w||^2$$

$$\text{subject to: } y_i(w^T x_i + b) \geq 1 \quad i=1,\ldots,n. \qquad (5)$$

The constraints in this formulation ensure that the maximum margin classifier classifies each example correctly, which is possible since we assumed that the data is linearly separable. In practice, data is often not linearly separable; and even if it is, a greater margin can be achieved by allowing the classifier to misclassify some points. To allow errors we replace the inequality constraints in Eqn. (5) with

$$y_i(w^T x_i + b) \geq 1 - \epsilon_i \quad i=1,2,\ldots,n$$

Where $\epsilon_i = 0$ are *slack variables* that allow an example to be in the margin ($0 \leq \epsilon_i \leq 1$, also called a margin error) or to be misclassified ($\epsilon_i > 1$). Since an example is misclassified if the value of its slack variable is greater than 1, $\sum_i \epsilon_i$ is a bound on the number of misclassified examples. Our objective of maximizing the margin, i.e. minimizing $\frac{1}{2}||w||^2$ will be

augmented with a term $C\sum_i \epsilon_i$ to penalize misclassification and margin errors. The optimization problem now becomes:

$$\underset{w,\,b}{\text{minimize}} \qquad \frac{1}{2}||w||^2 + C\sum_{i=1}^{n} \epsilon_i$$

$$\text{subject to: } y_i(w^T x_i + b) \geq 1 - \epsilon_i, \epsilon_i \geq 0 \qquad (6)$$

The constant C > 0 sets the relative importance of maximizing the margin and minimizing the amount of slack. This formulation is called the *soft-margin* SVM. Using the method of Lagrange multipliers, we can obtain the *dual* formulation which is expressed in terms of variable α_i:

$$\underset{\alpha}{\text{maximize}} \qquad \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} y_i y_j \alpha_i \alpha_j x_i x_j$$

$$\text{subject to: } \sum_{i=1}^{n} y_i \alpha_i = 0, \qquad 0 \leq \alpha_i \leq C \qquad (7)$$

The dual formulation leads to an expansion of the weight vector in terms of the input examples:

$$w = \sum_{i=1}^{n} y_i \alpha_i x_i \qquad (8)$$

The examples x_i for which $\alpha_i > 0$ are those points that are on the margin or within the margin when a soft-margin SVM is used. These are the so-called *support vectors*. The expansion in terms of the support vectors is often sparse, and the level of sparsity (fraction of the data serving as support vectors) is an upper bound on the error rate of the classifier.

The dual formulation of the SVM optimization problem depends on the data only through dot products. The dot product can therefore be replaced with a non-linear kernel function, thereby performing large margin separation in the feature-space of the kernel (see Figures 4.4 and 4.5). The SVM optimization problem was traditionally solved in the dual formulation, and only recently it was shown that the primal formulation, Equation (6), can lead to efficient kernel-based learning.

4.4 Kernels: from Linear to Non-Linear Classifiers

In many applications a non-linear classifier provides better accuracy. And yet, linear classifiers have advantages, one of them being that they often have simple training algorithms that scale well with the number of examples. This begs the question: Can the machinery of linear classifiers be extended to generate non-linear decision boundaries? Furthermore, can we handle domains such as protein sequences or structures where a representation in a fixed dimensional vector space is not available?

The approach of explicitly computing non-linear features does not scale well with the number of input features: when applying the mapping from the above example the dimensionality of the feature space F is quadratic in the dimensionality of the original space. This results in a quadratic increase in memory usage for storing the features and a quadratic increase in the time required to compute the discriminant function of the classifier. This quadratic complexity is feasible for low dimensional data; but when handling gene expression data that can have thousands of dimensions, quadratic complexity in the number of dimensions is not acceptable. Kernel methods solve this issue by avoiding the step of explicitly mapping the data to a high dimensional feature-space. The kernel can be computed without explicitly computing the mapping.

A degree-d polynomial kernel can be defined as:

$$k(x, x') = (x^T x' + 1)^d: \quad (9)$$

The feature space for this kernel consists of all monomials up to degree d, i.e. features of the form: $x_1^{d_1} x_2^{d_2} \ldots . x_m^{d_m}$ where $\sum_{i=1}^{m} d_i \leq d$. The kernel with d=1 is the *linear kernel*, and in that case the additive constant in Equation 9 is usually omitted. The increasing flexibility of the classifier as the degree of the polynomial is increased is illustrated in Figure 4.4. The other widely used kernel is the Gaussian kernel defined by:

$$k(x,x') = \exp(-\gamma \, || x - x' ||^2), \quad (10)$$

where $\gamma > 0$ is a parameter that controls the width of Gaussian. It plays a similar role as the degree of the polynomial kernel in controlling the flexibility of the resulting classifier (see Figure 4.5).

Linear decision boundary can be kernelized, i.e. its dependence on the data is only through dot products. In order for this to be useful, the training algorithms needs to be kernelizable as

well. It turns out that a large number of machine learning algorithms can be expressed using kernels-including ridge regression, the perceptron algorithm, and SVMs.

4.5 Understanding the Effects of SVM and Kernel Parameters

Training an SVM finds the large margin hyperplane, i.e. sets the parameters α_i and b. The SVM has another set of parameters called *hyperparameters*: The soft margin constant, C, and any parameters the kernel function may depend on (width of a Gaussian kernel or degree of a polynomial kernel). In this section we illustrate the effect of the hyperparameters on the decision boundary of an SVM using two-dimensional examples.

We begin our discussion of hyperparameters with the soft-margin constant, whose role is illustrated in Figure 4.3. For a large value of C a large penalty is assigned to errors/margin errors. This is seen in the left panel of Figure 4.3, where the two points closest to the hyperplane affect its orientation, resulting in a hyperplane that comes close to several other data points. When C is decreased (right panel of the figure), those points become margin errors; the hyperplane's orientation is changed, providing a much larger margin for the rest of the data.

Kernel parameters also have a significant effect on the decision boundary. The degree of the polynomial kernel and the width parameter of the Gaussian kernel control the flexibility of the resulting classifier (Figures 4.4 and 4.5). The lowest degree polynomial is the linear kernel, which is not sufficient when a non-linear relationship between features exists. For the data in Figure 4.4 a degree-2 polynomial is already flexible enough to discriminate between the two classes with a sizable margin. The degree-5 polynomial yields a similar decision boundary, albeit with greater curvature.

Next we turn our attention to the Gaussian kernel:k(x,x') = exp(- γ || x $- x'||^2$). This expression is essentially zero if the distance between x and x^0 is much larger than $1/\sqrt{\gamma}$; i.e. for a fixed x' it is localized to a region around x'. The support vector expansion is thus a sum of Gaussian “bumps" centred around each support vector. When γ is small (top left panel in Figure 4.5) a given data point x has a non-zero kernel value relative to any example in the set of support vectors. Therefore the whole set of support vectors affects the value of the discriminant function at x, resulting in a smooth decision boundary. As γ is increased the

locality of the support vector expansion increases, leading to greater curvature of the decision boundary. When γ is large the value of the discriminant function is essentially constant outside the close proximity of the region where the data are concentrated (see bottom right panel in Figure 4.5). In this regime of the γ parameter the classifier is clearly overfitting the data.

As seen from the examples in Figures 4.4 and 4.5 the parameter of the Gaussian kernel and the degree of polynomial kernel determine the flexibility of the resulting SVM in fitting the data. If this complexity parameter is too large, overfitting will occur (bottom panels in Figure 4.5).

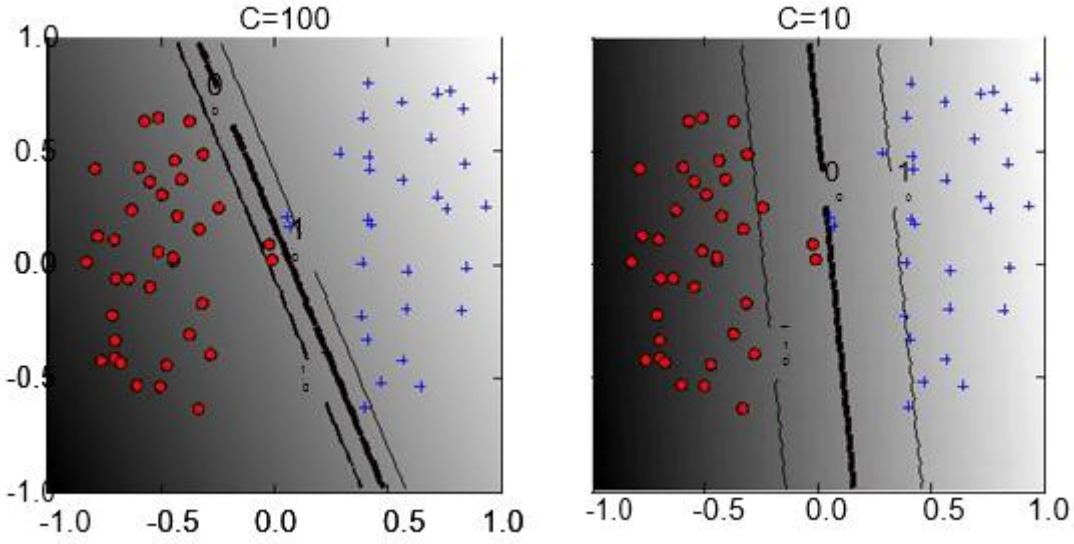


Figure 4.3- The effect of soft-margin constant on decision boundary

A smaller value of C (right) allows to ignore points close to the boundary, and increases the margin. The decision boundary between negative examples (red circles) and positive examples (blue crosses) is shown as a thick line. The lighter lines are on the margin (discriminant value equal to -1 or +1). The grayscale level represents the value of the discriminant function, dark for low values and a light shade for high values.

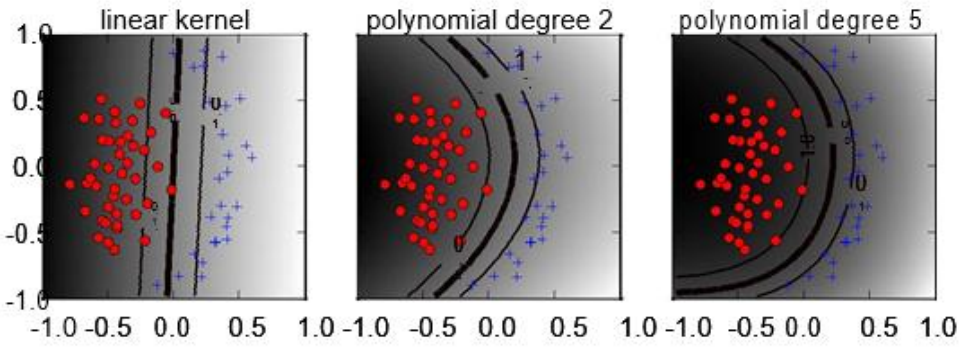


Figure 4.4- The effect of degree of a polynomial kernel on decision boundary

Higher degree polynomial kernels allow a more flexible decision boundary. The style follows that of Figure 4.3.

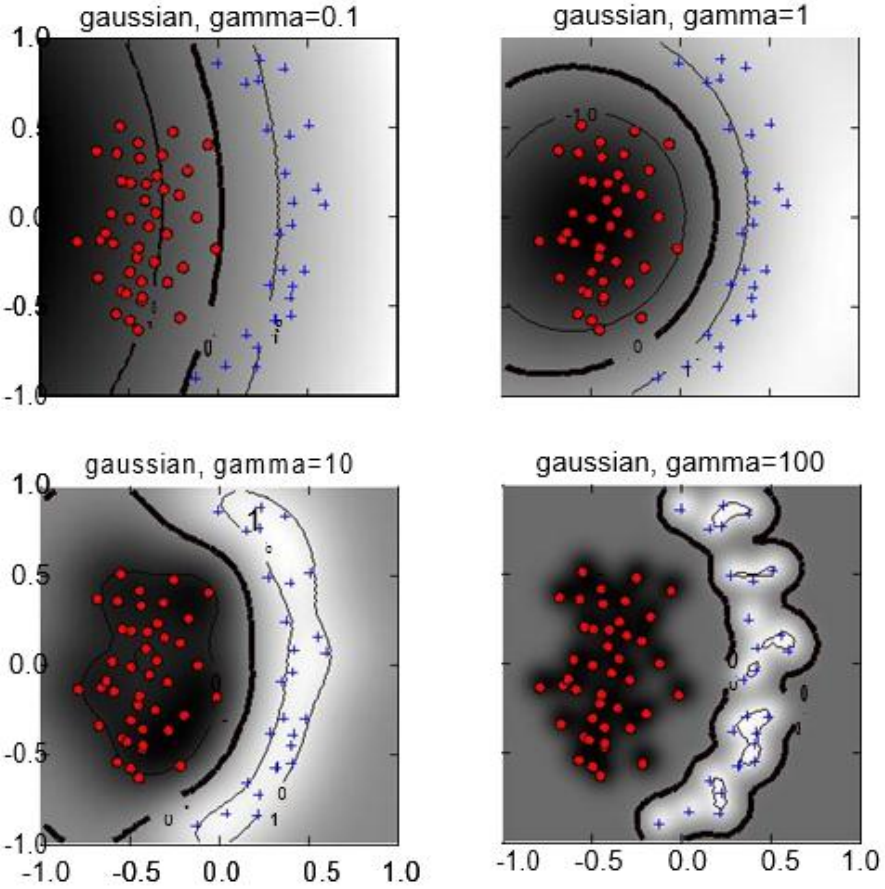


Figure 4.5-Effect of inverse-width parameter (γ) for fixed value of soft-margin constant

For small values of γ (upper left) the decision boundary is nearly linear. As γ increases the flexibility of the decision boundary increases. Large values of γ lead to overfitting (bottom). The figure style follows that of Figure 4.3.

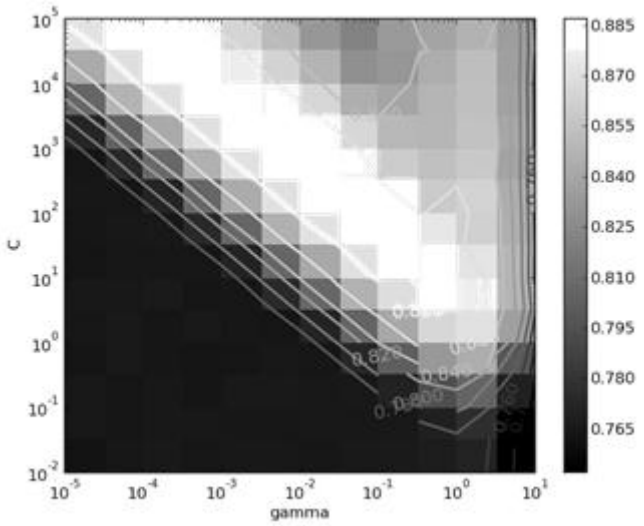


Figure 4.6- SVM accuracy on a grid of parameter values

A question frequently posed by practitioners is: which kernel should I use for my data?" There are several answers to this question. The first is that it is, like most practical questions in machine learning, data-dependent, so several kernels should be tried. That being said, we typically follow the following procedure: Try a linear kernel first, and then see if we can improve on its performance using a non-linear kernel. The linear kernel provides a useful baseline, and in many bioinformatics applications provides the best results: The flexibility of the Gaussian and polynomial kernels often leads to overfitting in high dimensional datasets with a small number of examples, microarray datasets being a good example. Furthermore, an SVM with a linear kernel is easier to tune since the only parameter that affects performance is the soft-margin constant. Once a result using a linear kernel is available it can serve as a baseline that you can try to improve upon using a non-linear kernel. Between the Gaussian and polynomial kernels, our experience shows that the Gaussian kernel usually outperforms the polynomial kernel in both accuracy and convergence time.

4.6 Model Selection

The dependence of the SVM decision boundary on the SVM hyperparameters translates into a dependence of classifier accuracy on the hyperparameters. When working with a linear classifier the only hyperparameter that needs to be tuned is the SVM soft-margin constant. For the polynomial and Gaussian kernels the search space is two-dimensional. The standard method of exploring this two dimensional space is via grid-search; the grid points are

generally chosen on a logarithmic scale and classifier accuracy is estimated for each point on the grid. This is illustrated in Figure 4.6. A classifier is then trained using the hyperparameters that yield the best accuracy on the grid.

The accuracy landscape in Figure 4.6 has an interesting property: there is a range of parameter values that yield optimal classifier performance; furthermore, these equivalent points in parameter space fall along a \ridge" in parameter space. This phenomenon can be understood as follows. Consider a particular value of (γ, C). If we decrease the value of γ, this decreases the curvature of the decision boundary; if we then increase the value of C the decision boundary is forced to curve to accommodate the larger penalty for errors/margin errors. This is illustrated in Figure 4.7 for two dimensional data.

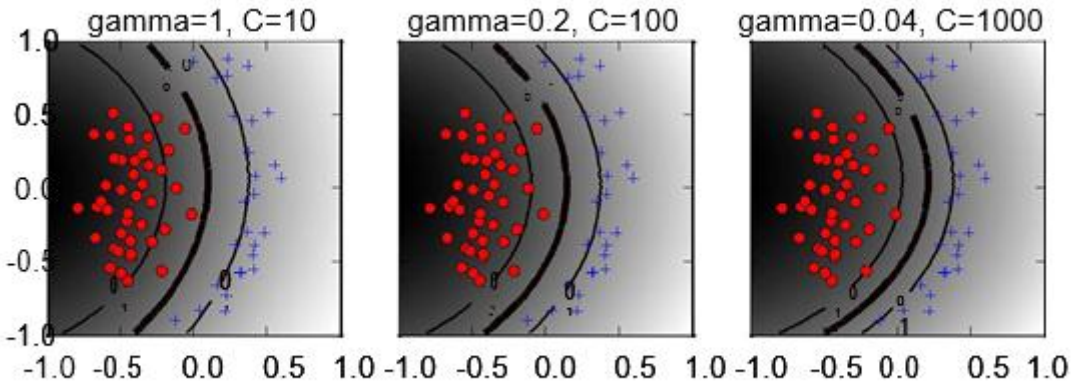


Figure 4.7- Obtaining similar decision boundaries using different combinations of SVM parameters

CHAPTER-5

HISTOGRAM OF ORIENTED GRADIENT

What is HOG?

Theoretical Background

Overview of the method

Algorithm Implementation

Final Descriptor size

Implementations and Performance Study

5.1What is HOG?

Histogram of Oriented Gradient descriptors, or **HOG** descriptors, are feature descriptors used in computer vision and image processing for the purpose of object detection. The technique counts occurrences of gradient orientation in localized portions of an image.

The purpose is to implement the algorithm of extracting Histogram of Gradient Orientation (HOG) features. These features will be used then for classification and object recognition.

5.2 Theoretical background

HOG features have been introduced by Navneet Dalal and Bill Triggs who have developed and tested several variants of HOG descriptors, with differing spatial organization, gradient computation and normalization methods.

The essential thought behind the Histogram of Oriented Gradient descriptors is that local object appearance and shape within an image can be described by the distribution of intensity gradients or edge directions. The implementation of these descriptors can be achieved by dividing the image into small connected regions, called cells, and for each cell compiling a histogram of gradient directions or edge orientations for the pixels within the cell. The combination of these histograms then represents the descriptor. For improved performance, the local histograms can be contrast-normalized by calculating a measure of the intensity across a larger region of the image, called a block, and then using this value to normalize all cells within the block. This normalization results in better invariance to changes in illumination or shadowing.

5.3 Overview of the Method

This section gives an overview of our feature extraction chain, which is summarized in fig.5.1. Implementation details are postponed until x6. The method is based on evaluating well-normalized local histograms of image gradient orientations in a dense grid. Similar features have seen increasing use over the past decade. The basic idea is that local object appearance and shape can often be characterized rather well by the distribution of local intensity gradients or edge directions, even without precise knowledge of the corresponding gradient or edge positions. In practice this is implemented by dividing the image window into

small spatial regions ("***cells***"), for each cell accumulating a local 1-D histogram of gradient directions or edge orientations over the pixels of the cell. The combined histogram entries form the representation. For better invariance to illumination, shad-owing, ***etc***., it is also useful to contrast-normalize the local responses before using them. This can be done by accumulating a measure of local histogram "energy" over somewhat larger spatial regions ("***blocks***") and using the results to normalize all of the cells in the block. We will refer to the normalized descriptor blocks as ***Histogram of Oriented Gradient (HOG)*** descriptors. Tiling the detection window with a dense (in fact, overlapping) grid of HOG descriptors and using the combined feature vector in a conventional SVM based window classifier gives our human detection chain (see fig.5.1).

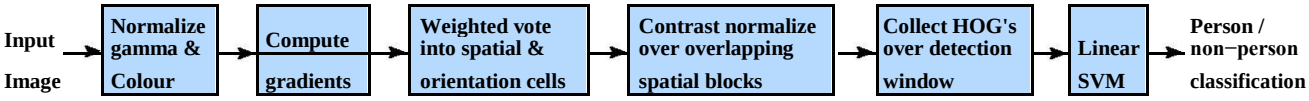


Figure 5.1 - An overview of our feature extraction and object detection

The use of orientation histograms has many precursors, but it only reached maturity when combined with local spatial histogramming and normalization in Lowe's ***Scale Invariant Feature Transformation (SIFT)*** approach to wide baseline image matching, in which it provides the underlying image patch descriptor for matching scale-invariant key points. SIFT-style approaches perform remark-ably well in this application. The ***Shape Context*** work studied alternative cell and block shapes, albeit initially using only edge pixel counts without the orientation histogramming that makes the representation so effective. The success of these sparse feature based representations has somewhat overshadowed the power and simplicity of HOG's as dense image descriptors. We hope that our study will help to rectify this. In particular, our informal experiments suggest that even the best current key point based approaches are likely to have false positive rates at least 1–2 orders of magnitude higher than our dense grid approach for human detection, mainly because none of the key point detectors that we are aware of detect human body structures reliably.

The HOG/SIFT representation has several advantages. It captures edge or gradient structure that is very characteristic of local shape and it does so in a local representation with an easily controllable degree of invariance to local geometric and photometric transformations: translations or rotations make little difference if they are much smaller that the local spatial or orientation bin size. For human detection, rather coarse spatial sampling, ne orientation sampling and strong local photometric normalization turns out to be the best strategy,

presumably because it permits limbs and body segments to change appearance and move from side to side quite a lot provided that they maintain a roughly upright orientation.

5.4 Algorithm Implementation

5.4.1 Gradient Computation

The first step of calculation is the computation of the gradient values.

One application of gradient vectors is to edge detection. You can see in the gradient images how large gradient values correspond to strong edges in the image. The other application is feature extraction. When we base our feature descriptors on gradient vectors instead of just the raw pixel values, we gain some "lighting invariance". We'll compute the same descriptor (or at least closer to the same descriptor) for an object under different lighting conditions, making it easier to recognize the object despite changes in lighting.

The most common method is to apply the 1D centred point discrete derivative mask in both the horizontal and vertical directions. Specifically, this method requires filtering the grayscale image with the following filter kernels:

$$D_X = [-1 \quad 0 \quad 1] \text{ and } D_Y = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}$$

Figure 5.2 - Initial Image

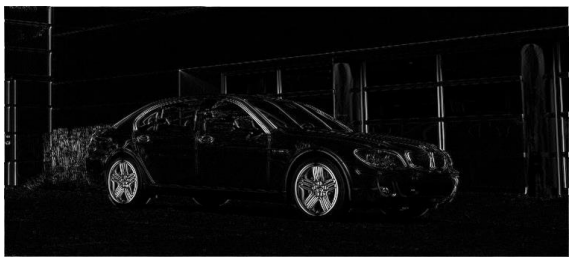


Figure 5.3-X-derivative of the initial image *Figure 5.4-Y-derivative of the initial image*

So, being given an image *I*, we obtain the *x* and *y* derivatives using a convolution operation:

$I_X = ID_X$ and $I_Y = ID_Y$

The magnitude of the gradient is $|G| = \sqrt{I_X^2 + I_y^2}$

The orientation of the gradient is given by: $\theta = tan^{-1}(I_Y/I_X)$

5.4.2 Orientation Binning

The second step of calculation involves creating the cell histograms. Each pixel within the cell casts a **weighted vote** for an orientation-based histogram channel based on the values found in the gradient computation. The cells themselves are rectangular and the histogram channels are evenly spread over 0 to 180 degrees or 0 to 360 degrees, depending on whether the gradient is "unsigned" or "signed". Dalal and Triggs found that unsigned gradients used in conjunction with 9 histogram channels performed best in their experiments. As for the vote weight, pixel contribution can be the gradient magnitude itself, or the square root or square of the gradient magnitude.

Figure 5.5 - Initial Image

Figure 5.6 - Magnitude of Gradient

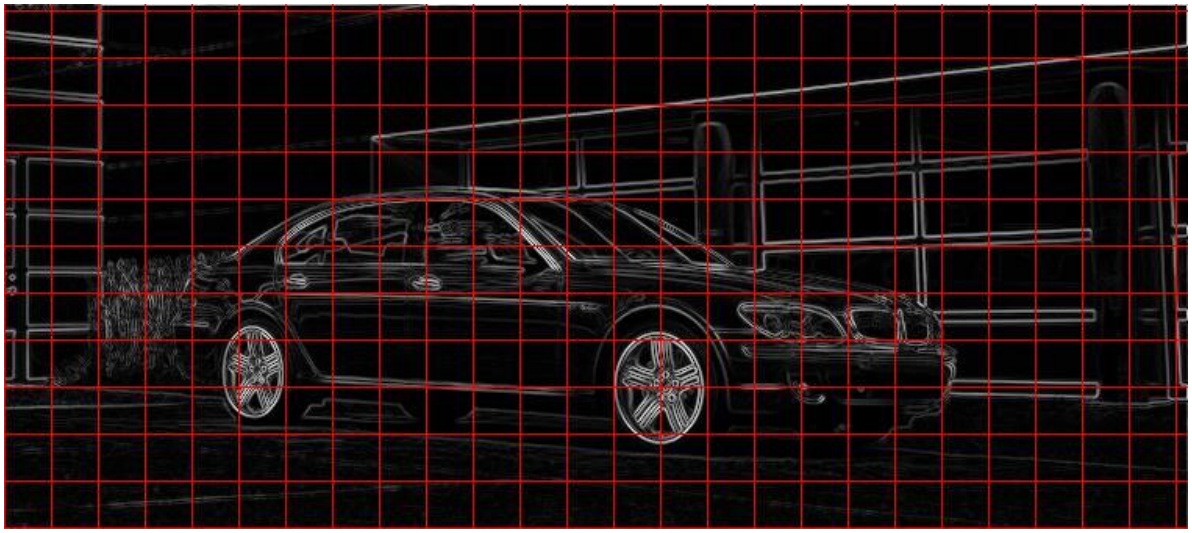


Figure 5.7 - Cell division example

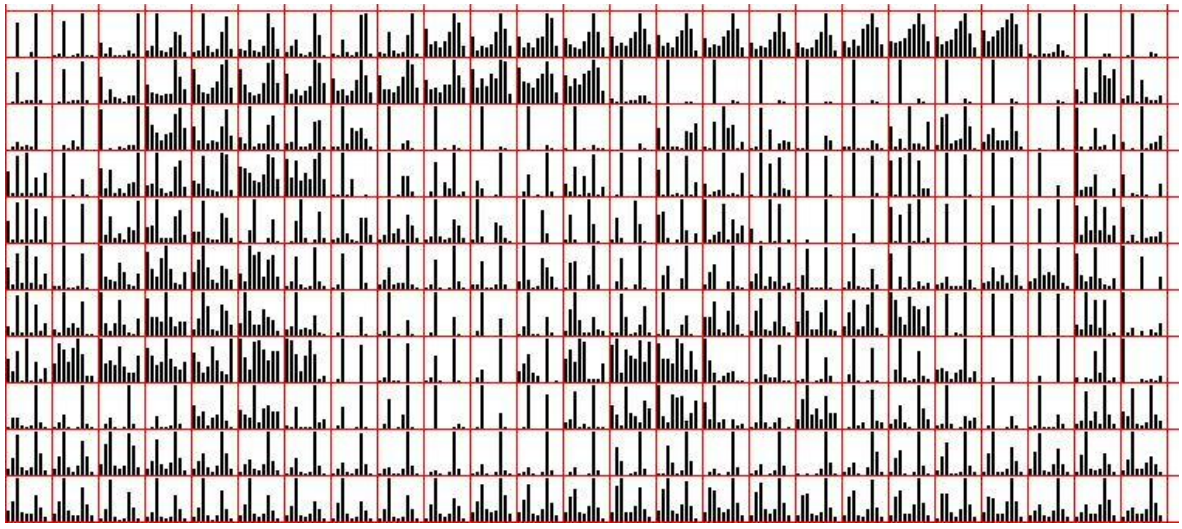


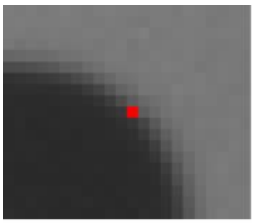
Figure 5.8 - Histogram of Oriented Gradient descriptors (normalized inside each cell)

5.4.3 Normalizing Gradient Vectors

The next step in computing the descriptors is to normalize the histograms. Let's take a moment to first look at the effect of normalizing gradient vectors in general. It turns out that by normalizing your gradient vectors, you can also make them invariant to *multiplications* of the pixel values. Take a look at the below examples. The first image shows a pixel, highlighted in red, in the original image. In the second image, all pixel values have been increased by 50. In the third image, all pixel values in the original image have been multiplied by 1.5.

Notice how the third image displays an increase in contrast. The effect of the multiplication is that bright pixels became much brighter while dark pixels only became a little brighter, thereby increasing the contrast between the light and dark parts of the image.

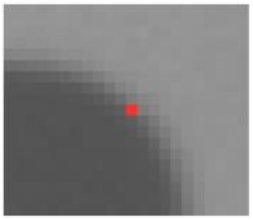
Let's look at the actual pixel values and how the gradient vector changes in these three images. The numbers in the boxes below represent the values of the pixels surrounding the pixel marked in red.



	93	
56		94
	55	

$$\nabla f = \begin{bmatrix} 38 \\ 38 \end{bmatrix}$$

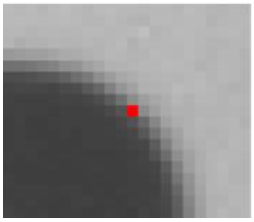
$$|\nabla f| = \sqrt{(38)^2 + (38)^2} = 53.74$$



	143	
106		144
	105	

$$\nabla f = \begin{bmatrix} 38 \\ 38 \end{bmatrix}$$

$$|\nabla f| = \sqrt{(38)^2 + (38)^2} = 53.74$$



	140	
84		141
	83	

$$\nabla f = \begin{bmatrix} 57 \\ 57 \end{bmatrix}$$

$$|\nabla f| = \sqrt{(57)^2 + (57)^2} = 80.61$$

The gradient vectors are equivalent in the first and second images, but in the third, the gradient vector magnitude has increased by a factor of 1.5. If you divide all three vectors by their respective magnitudes, you get the same result for all three vectors: [0.71 0.71].

So in the above example we see that by dividing the gradient vectors by their magnitude we can make them invariant (or at least more robust) to changes in contrast.

Dividing a vector by its magnitude is referred to as normalizing the vector to unit length, because the resulting vector has a magnitude of 1. Normalizing a vector does not affect its orientation, only the magnitude.

5.4.4 Histogram Normalization

Recall that the value in each of the nine bins in the histogram is based on the magnitudes of the gradients in the 8×8 pixel cell over which it was computed. If every pixel in a cell is multiplied by 1.5, for example, then we saw above that the magnitude of all of the gradients in the cell will be increased by a factor of 1.5 as well. In turn, this means that the value for each bin of the histogram will also be increased by 1.5x. By normalizing the histogram, we can make it invariant to this type of illumination change.

5.4.5 Block Normalization

Rather than normalize each histogram individually, the cells are first grouped into blocks and normalized based on all histograms in the block.

The blocks used by Dalal and Triggs consisted of 2 cells by 2 cells. The blocks have "50% overlap", which is best described through the illustration below.

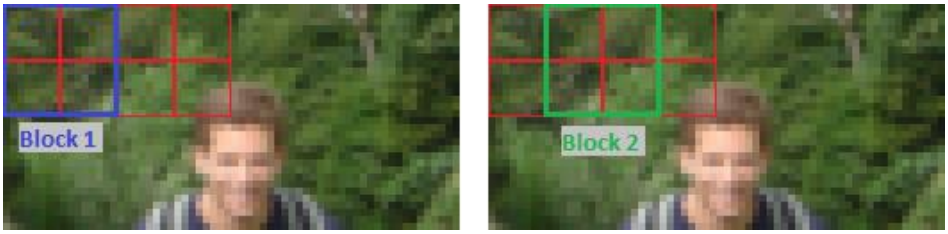


Figure 5.9 - Block Normalization with 50% overlap

This block normalization is performed by concatenating the histograms of the four cells within the block into a vector with 36 components (4 histograms x 9 bins per histogram). Divide this vector by its magnitude to normalize it.

The effect of the block overlap is that each cell will appear multiple times in the final descriptor, but normalized by a different set of neighbouring cells. (Specifically, the corner cells appear once, the other edge cells appear twice each, and the interior cells appear four times each).

In the earlier normalization example, we multiplied every pixel in the image by 1.5, effectively increasing the contrast by the same amount over the *whole* image. The rationale in the block normalization approach is that, changes in contrast are more likely to occur over smaller regions within the image. So rather than normalizing over the entire image, we normalize within a small region around the cell.

5.5 Final Descriptor Size

If we have 64 x 128 pixel detection window, it will be divided into 7 blocks across and 15 blocks vertically, for a total of 105 blocks. Each block contains 4 cells with a 9-bin histogram for each cell, for a total of 36 values per block. This brings the final vector size to 7 blocks across x 15 blocks vertically x 4 cells per block x 9-bins per histogram = 3,780 values. These 3,780 values will be passed on to SVM classifier for the classifying purpose.

5.6 Implementation and Performance Study

We now give details of our HOG implementations and systematically study the effects of the various choices on detector performance. Throughout this section we refer results to our default detector which has the following properties, described below: RGB colour space with no gamma correction; [-1, 0, 1] gradient filter with no smoothing; linear gradient voting into 9 orientation bins in 0° –180° ; 16x16 pixel blocks of four 8x8 pixel cells; Gaussian spatial window with $\sigma = 8$ pixel; ***L2-Hys*** (Lowe-style clipped L2 norm) block normalization; block spacing stride of 8 pixels (hence 4-fold coverage of each cell); 64x128 detection window; linear SVM classifier.

Fig.5.10 summarizes the effects of the various HOG parameters on overall detection performance. These will be examined in detail below. The main conclusions are that for good performance, one should use ne scale derivatives (essentially no smoothing), many orientation bins, and moderately sized, strongly normalized, overlapping descriptor blocks.

5.6.1 Gamma/Colour Normalization

We evaluated several input pixel representations including grayscale, RGB and LAB colour spaces optionally with power law (gamma) equalization. These normalizations have only a modest effect on performance, perhaps because the subsequent descriptor normalization achieves similar results. We do use colour information when available. RGB and LAB colour spaces give comparable results, but restricting to grayscale reduces performance by 1.5% at 10^{-4} FPPW. Square root gamma compression of each colour channel improves performance at low FPPW (by 1% at 10^{-4} FPPW) but log compression is too strong and worsens it by 2% at 10^{-4} FPPW.

5.6.2 Gradient Computation

Detector performance is sensitive to the way in which gradients are computed, but the simplest scheme turns out to be the best. We tested gradients computed using Gaussian smoothing followed by one of several discrete derivative masks. Several smoothing scales were tested including σ = 0 (none). Masks tested included various 1-D point derivatives (uncentred [-1, 1], centred [-1, 0, 1] and cubic-corrected [1, 8, 0, 8, 1]) as well as 3x3 Sobel masks and 2x2 diagonal ones $\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}, \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$ (the most compact centred 2-D derivative masks). Simple 1-D [-1; 0; 1] masks at σ =0 work best. Using larger masks always seems to decrease performance, and smoothing damages it significantly: for Gaussian derivatives, moving from σ =0 to σ =2 reduces the recall rate from 89% to 80% at 10^{4} FPPW. At σ =0, cubic corrected 1-D width 5 filters are about 1% worse than [-1, 0, 1] at 10^{-4} FPPW, while the 2x2 diagonal masks are 1.5% worse. Using uncentred [-1, 1] derivative masks also decreases performance (by 1.5% at 10^{-4} FPPW), presumably because orientation estimation suffers as a result of the x and y filters being based at different centres.

For colour images, we calculate separate gradients for each colour channel, and take the one with the largest norm as the pixel's gradient vector.

5.6.3 Spatial / Orientation Binning

The next step is the fundamental nonlinearity of the descriptor. Each pixel calculates a weighted vote for an edge orientation histogram channel based on the orientation of the gradient element centred on it, and the votes are accumulated into orientation bins over local spatial regions that we call ***cells***. Cells can be either rectangular or radial (log-polar sectors). The orientation bins are evenly spaced over 0° – 180° ("unsigned" gradient) or 0° –360° ("signed" gradient). To reduce aliasing, votes are interpolated bi-linearly between the neighboring bin centres in both orientation and position. The vote is a function of the gradient magnitude at the pixel, either the magnitude itself, its square, its square root, or a clipped form of the magnitude representing soft presence/absence of an edge at the pixel. In practice, using the magnitude itself gives the best results. Taking the square root reduces performance slightly, while using binary edge presence voting decreases it significantly (by 5% at 10^{-4} FPPW).

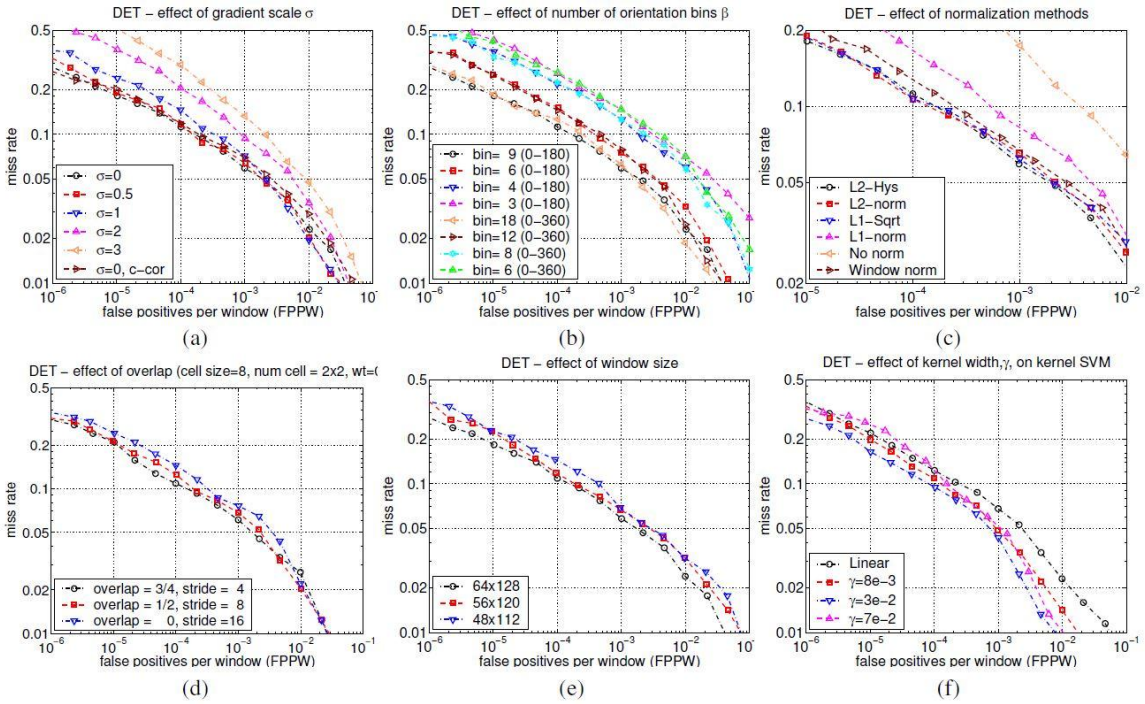


Figure 5.10 - HOG Implementation and Performance Study

(a) Using ne derivative scale significantly increases the performance. ('c-cor' is the 1D cubic-corrected point derivative). (b) Increasing the number of orientation bins increases performance significantly up to about 9 bins spaced over 0° – 180°. (c) The effect of different block normalization schemes. (d) Using overlapping descriptor blocks decreases the miss rate by around 5%. (e) Reducing the 16 pixel margin around the 64x128 detection window decreases the performance by about 3%. (f) Using a Gaussian kernel SVM, $exp(-\gamma\|x_1 - x_2\|^2)$, improves the performance by about 3%.

Fine orientation coding turns out to be essential for good performance, whereas spatial binning can be rather coarse. As fig 5.10(b) shows, increasing the number of orientation bins improves performance significantly up to about 9 bins, but makes little difference beyond this. This is for bins spaced over 0° –180°, ***i.e***. the `sign' of the gradient is ignored. Including signed gradients (orientation range 0° – 360°, as in the original SIFT descriptor) decreases the performance, even when the number of bins is also doubled to preserve the original orientation resolution. For humans, the wide range of clothing and background colours presumably makes the signs of contrasts uninformative. However note that including sign information does help substantially in some other object recognition tasks, ***e.g***. cars, motorbikes.

5.6.4 Normalization and Descriptor Blocks

Gradient strengths vary over a wide range owing to local variations in illumination and foreground-background contrast, so effective local contrast normalization turns out to be essential for good performance. We evaluated a number of different normalization schemes. Most of them are based on grouping cells into larger spatial blocks and contrast normalizing each block separately. The final descriptor is then the vector of all components of the normalized cell responses from all of the blocks in the detection window.

In fact, we typically overlap the blocks so that each scalar cell response contributes several components to the final descriptor vector, each normalized with respect to a different block. This may seem redundant but good normalization is critical and including overlap significantly improves the performance. Fig. 5.10(d) shows that performance increases by 4% at 10^{-4} FPPW as we increase the overlap from none to 16-fold area/ 4-fold linear coverage.

We evaluated two classes of block geometries, square or rectangular ones partitioned into grids of square or rectangular spatial cells, and circular blocks partitioned into cells in log-polar fashion. We will refer to these two arrangements as R-HOG and C-HOG (for rectangular and circular HOG).

R-HOG. R-HOG blocks have many similarities to SIFT descriptor but they are used quite differently. They are computed in dense grids at a single scale without dominant orientation alignment and used as part of a larger code vector that implicitly encodes spatial position relative to the detection window, whereas SIFT's are computed at a sparse set of scale-invariant key points, rotated to align their dominant orientations, and used individually.

SIFT's are optimized for sparse wide baseline matching, R-HOG's for dense robust coding of spatial form. Other precursors include the edge orientation histograms of Freeman & Roth. We usually use square R-HOG's, ***i.e***. ς x ς grids of η x η pixel cells each containing β orientation bins, where ς, η, β are parameters.

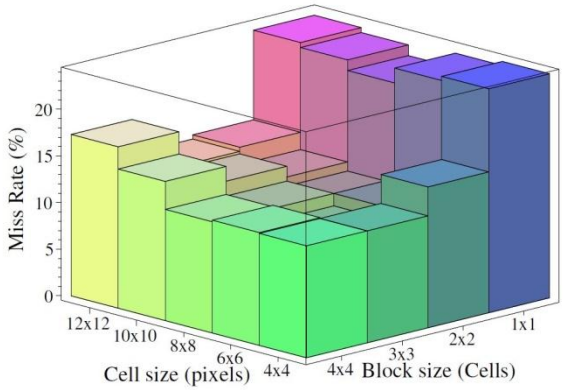


Figure 5.11 - Miss rate at 10^{-4} *FPPW as the cell and block sizes change.*

Fig. 5.11 plots the miss rate at 10^{-4} FPPW w.r.t. cell size and block size in cells. For human detection, 3x3 cell blocks of 6x6 pixel cells perform best, with 10.4% miss-rate at 10^{-4}FPPW. In fact, 6–8 pixel wide cells do best irrespective of the block size – an interesting coincidence as human limbs are about 6–8 pixels across in our images. 2x2 and 3x3 blocks work best. Beyond this, the results deteriorate: adaptivity to local imaging conditions is weakened when the block becomes too big, and when it is too small (1x1 block / normalization over orientations alone) valuable spatial information is suppressed.

As in, it is useful to downweight pixels near the edges of the block by applying a Gaussian spatial window to each pixel before accumulating orientation votes into cells. This improves performance by 1% at 10^{-4} FPPW for a Gaussian with = 0.5*block-width.

We also tried including multiple block types with different cell and block sizes in the overall descriptor. This slightly improves performance (by around 3% at 10^{-4} FPPW), at the cost of greatly increased descriptor size.

Besides square R-HOG blocks, we also tested vertical (2x1 cell) and horizontal (1x2 cell) blocks and a combined descriptor including both vertical and horizontal pairs. Vertical and vertical+horizontal pairs are significantly better than horizontal pairs alone, but not as good as 2x2 blocks (1% worse at 10^{-4}FPPW).

C-HOG. Our circular block (C-HOG) descriptors are reminiscent of Shape Contexts except that, crucially, each spatial cell contains a stack of gradient-weighted orientation cells instead of a single orientation-independent edge-presence count. The log-polar grid was originally suggested by the idea that it would allow ne coding of nearby structure to be combined with coarser coding of wider context, and the fact that the transformation from the visual field to the V1 cortex in primates is logarithmic. However small descriptors with very few radial bins turn out to give the best performance, so in practice there is little inhomogeneity or context. It is probably better to think of C-HOG's simply as an advanced form of centre-surround coding.

We evaluated two variants of the C-HOG geometry, ones with a single circular central cell (similar to the GLOH feature of), and ones whose central cell is divided into angular sectors as in shape contexts. We present results only for the circular-centre variants, as these have fewer spatial cells than the divided centre ones and give the same performance in practice. A technical report will provide further details. The C-HOG layout has four parameters: the numbers of angular and radial bins; the radius of the central bin in pixels; and the expansion factor for subsequent radii. At least two radial bins (a centre and a surround) and four angular bins (quartering) are needed for good performance. Including additional radial bins does not change the performance much, while increasing the number of angular bins decreases performance (by 1.3% at 10^{-4}FPPW when going from 4 to 12 angular bins). 4 pixels is the best radius for the central bin, but 3 and 5 give similar results. Increasing the expansion factor from 2 to 3 leaves the performance essentially unchanged. With these parameters, neither Gaussian spatial weighting nor inverse weighting of cell votes by cell area changes the performance, but combining these two reduces slightly. These values assume ne orientation sampling. Shape contexts (1 orientation bin) require much finer spatial subdivision to work well.

Block Normalization schemes. We evaluated four different block normalization schemes for each of the above HOG geometries. Let v be the unnormalized descriptor vector, $\|v\|_k$ be its k-norm for k=1, 2, and ϵ be a small constant. The schemes are:

(a) ***L2-norm***, $v \rightarrow v/\sqrt{\|v\|_2^2 + \epsilon^2}$

(b)***L2-Hys***, L2-norm followed by clipping (limiting the maximum values of v to 0.2) and renormalizing;

(c)***L1-norm***, $v \rightarrow v/(\|v\|_1 + \epsilon)$; and

(d) ***L1-sqrt***, L1-norm followed by square root $v \rightarrow \sqrt{v/(\|v\|_1 + \epsilon)}$,

which amounts to treating the descriptor vectors as probability distributions and using the Bhattacharya distance between them. Fig. 5.10(c) shows that L2-Hys, L2-norm and L1-sqrt all perform equally well, while simple L1-norm reduces performance by 5%, and omitting normalization entirely reduces it by 27%, at 10^{-4} FPPW. Some regularization ϵ is needed as we evaluate descriptors densely, including on empty patches, but the results are insensitive to ϵ's value over a large range.

Centre-surround normalization. We also investigated an alternative centre-surround style cell normalization scheme, in which the image is tiled with a grid of cells and for each cell the total energy in the cell and its surrounding region (summed over orientations and pooled using Gaussian weighting) is used to normalize the cell. However as fig. 5.10(c) ("***window norm***") shows, this decreases performance relative to the corresponding block based scheme (by 2% at 10^{-4} FPPW, for pooling with σ =1 cell widths). One reason is that there are no longer any overlapping blocks so each cell is coded only once in the final descriptor. Including several normalizations for each cell based on different pooling scales provides no perceptible change in performance, so it seems that it is the existence of several pooling regions with ***different*** spatial offsets relative to the cell that is important here,not the pooling scale.

5.6.5 Detector Window and Context

Our 64x128 detection window includes about 16 pixels of margin around the person on all four sides. Fig. 5.10(e) shows that this border provides a significant amount of context that helps detection. Decreasing it from 16 to 8 pixels (48x112 detection window) decreases performance by 6% at 10^{-4} FPPW. Keeping a 64x128 window but increasing the person size within it (again decreasing the border) causes a similar loss of performance, even though the resolution of the person is actually increased.

5.6.6 Classifier

By default we use a soft (C =0:01) linear SVM trained with SVM Light (slightly modified to reduce memory usage for problems with large dense descriptor vectors). Using a Gaussian kernel SVM increases performance by about 3% at 10^{-4}FPPW at the cost of a much higher run time.

5.6.7 Discussion

Overall, there are several notable findings in this work. The fact that HOG greatly out-performs wavelets and that any significant degree of smoothing before calculating gradients damages the HOG results emphasizes that much of the available image information is from ***abrupt edges at fine scales***, and that blurring this in the hope of reducing the sensitivity to spatial position is a mistake. Instead, gradients should be calculated at the nest available scale in the current pyramid layer, rectified or used for orientation voting, and only then blurred spatially. Given this, relatively coarse spatial quantization suffices (8x8 pixel cells / one limb width). On the other hand, at least for human detection, it pays to sample orientation rather finely: both wavelets and shape contexts lose out significantly here.

Secondly, strong ***local*** contrast normalization is essential for good results, and traditional centre-surround style schemes are not the best choice. Better results can be achieved by normalizing each element (edge, cell) ***several times*** with respect to different local supports, and treating the results as independent signals. In our standard detector, each HOG cell appears four times with different normalizations and including this `redundant' information improves performance from 84% to 89% at 10^{-4}FPPW.

CHAPTER-6

IMPLEMENTATION

MNIST database of handwritten digits

Training a classifier

Testing the classifier

Assumptions during testing

Here we succinctly enumerate the steps that are needed to detect handwritten digits -

1. Create a database of handwritten digits.
2. For each handwritten digit in the database, extract HOG features and train a Linear SVM.
3. Use the classifier trained in step 2 to predict digits.

6.1 MNIST database of handwritten digits

The first step is to create a database of handwritten digits. We are not going to create a new database but we will use the popular MNIST database of handwritten digits. The MNIST database is a set of 70000 samples of handwritten digits where each sample consists of a grayscale image of size 28×28. There are a total of 70,000 samples. We will use sklearn.datasets package to download the MNIST database from mldata.org. This package makes it convenient to work with toy datasbases.

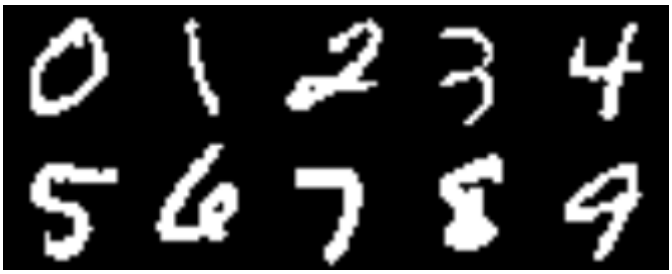


Figure 6.1 - Sample for each handwritten digit in MNIST database

There are approximate 7000 samples for each digit. The actual samples for each digit was -

Digits	Number of samples
0	6903
1	7877
2	6990
3	7141
4	6824
5	6313
6	6876
7	7293

8 6825

9 6958

Later the database of mathematical operators (+, -, x, /) were also added by scanning approximately 7000 sample images of each operator.

Now, we will write 2 python scripts – one for training the classifier and the second for test the classifier.

6.2 Training a Classifier

Here, we will implement the following steps –

1. Calculate the HOG features for each sample in the database.
2. Train a multi-class linear SVM with the HOG features of each sample along with the corresponding label.
3. Save the classifier in a file

We will use the sklearn.externals.joblib package to save the classifier in a file so that we can use the classifier again without performing training each time. Calculating HOG features for 98000 images is a costly operation, so we will save the classifier in a file and load it whenever we want to use it. As discussed above sklearn.datasets package will be used to download the MNIST database for handwritten digits. We will useskimage.feature.hog class to calculate the HOG features and sklearn.svm.LinearSVC class to perform prediction after training the classifier. We will store our HOG features and labels in numpy arrays. The next step is to download the dataset using the sklearn.datasets.fetch_mldata function.

Once the dataset is downloaded we will save the images of the digits in a numpy array features and the corresponding labels i.e. the digit in another numpy array labels.

Next, we calculate the HOG features for each image in the database and save them in another numpy array named hog_feature.

To calculate the HOG features, we set the number of cells in each block equal to one and each individual cell is of size 14×14. Since our image is of size 28×28, we will have four blocks/cells of size 14×14 each. Also, we set the size of orientation vector equal to 9. So our HOG feature vector for each sample will be of size 4×9 = 36. We are not interested in visualizing the HOG feature image, so we will set the visualize parameter to false.

The next step is to create a Linear SVM object. Since there are 10 digits and 4 operators, we need a multi-class classifier. The Linear SVM that comes with sklearn can perform multi-class classification.

We perform the training using the fit member function of the clf object.

The fit function requires 2 arguments –one an array of the HOG features of the handwritten digit that we calculated earlier and a corresponding array of labels. Each label value is from the set — [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]. When the training finishes, we will save the classifier in a file named digits_cls.pkl.

The compress parameter in the joblib.dump function is used to set how much compression is done.

Up till this point, we have successfully completed the first task of preparing our classifier.

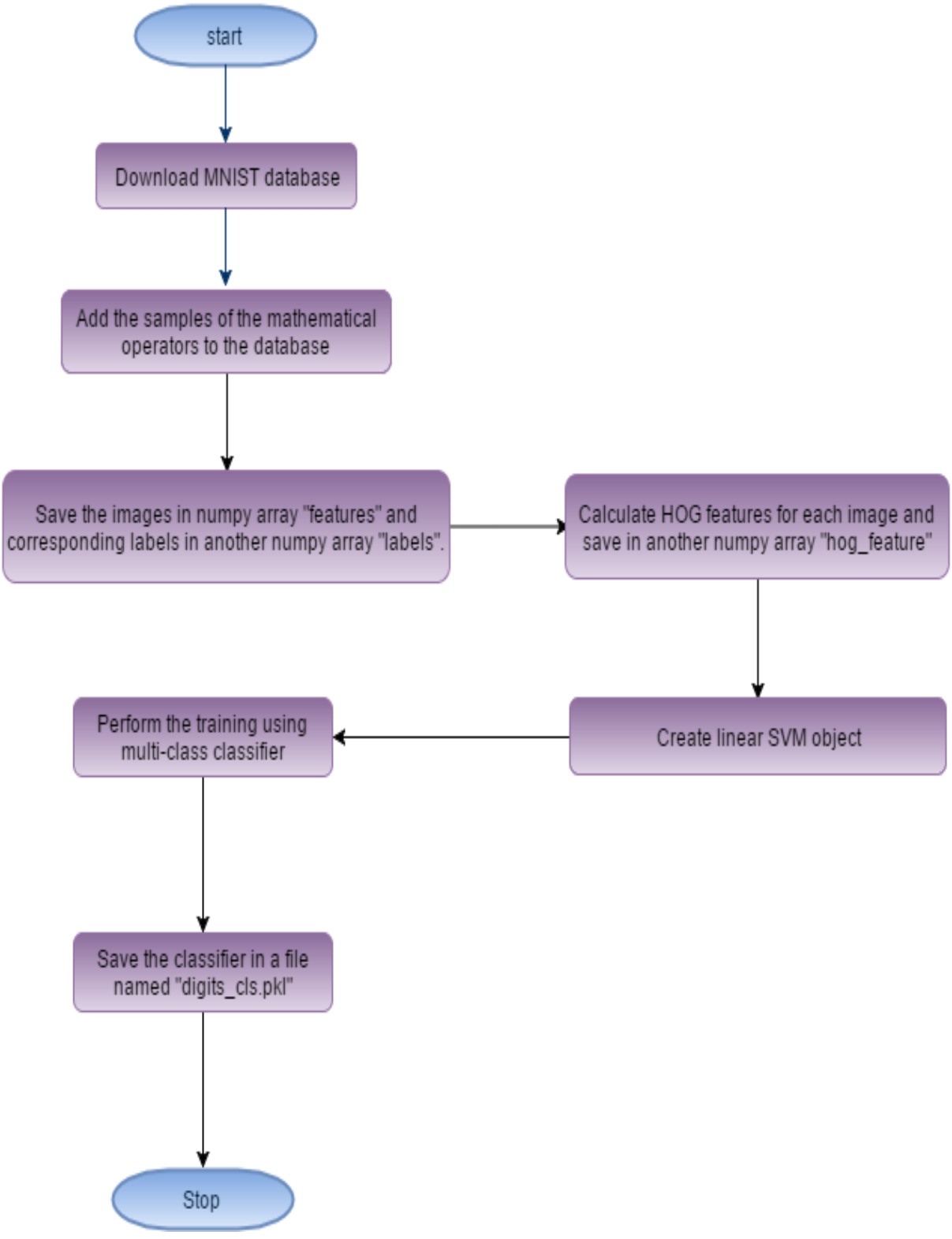


Figure 6.2 - Training a classifier

6.3 Testing the Classifier

Now, the second task is to test the classifier.

We load the required modules. Then, we load the classifier from the file digits_cls.pkl which we had saved in the previous script. After that, we load the test image and we convert it to a grayscale image using cv2.cvtColor function. We then apply a Gaussian filter to the grayscale image to remove noisy pixels. Later, we convert the grayscale image into a binary image using a threshold value of 90. All the pixel locations with grayscale values greater than 90 are set to 0 in the binary image and all the pixel locations with grayscale values less than 90 are set to 255 in the binary image. We calculate the contours in the image and then we calculate the bounding box for each contour. For each bounding box, we generate a bounding square around each contour. Then, we resize each bounding square to a size of 28×28 and dilate it. We calculate the HOG features for each bounding square. Remember here that the HOG feature vector for each bounding square should be of the same size for which the classifier was trained; else you will get an error. We predict the digit using our classifier. We also draw the bounding box and the predicted digit on the input image. Finally, we display the image.

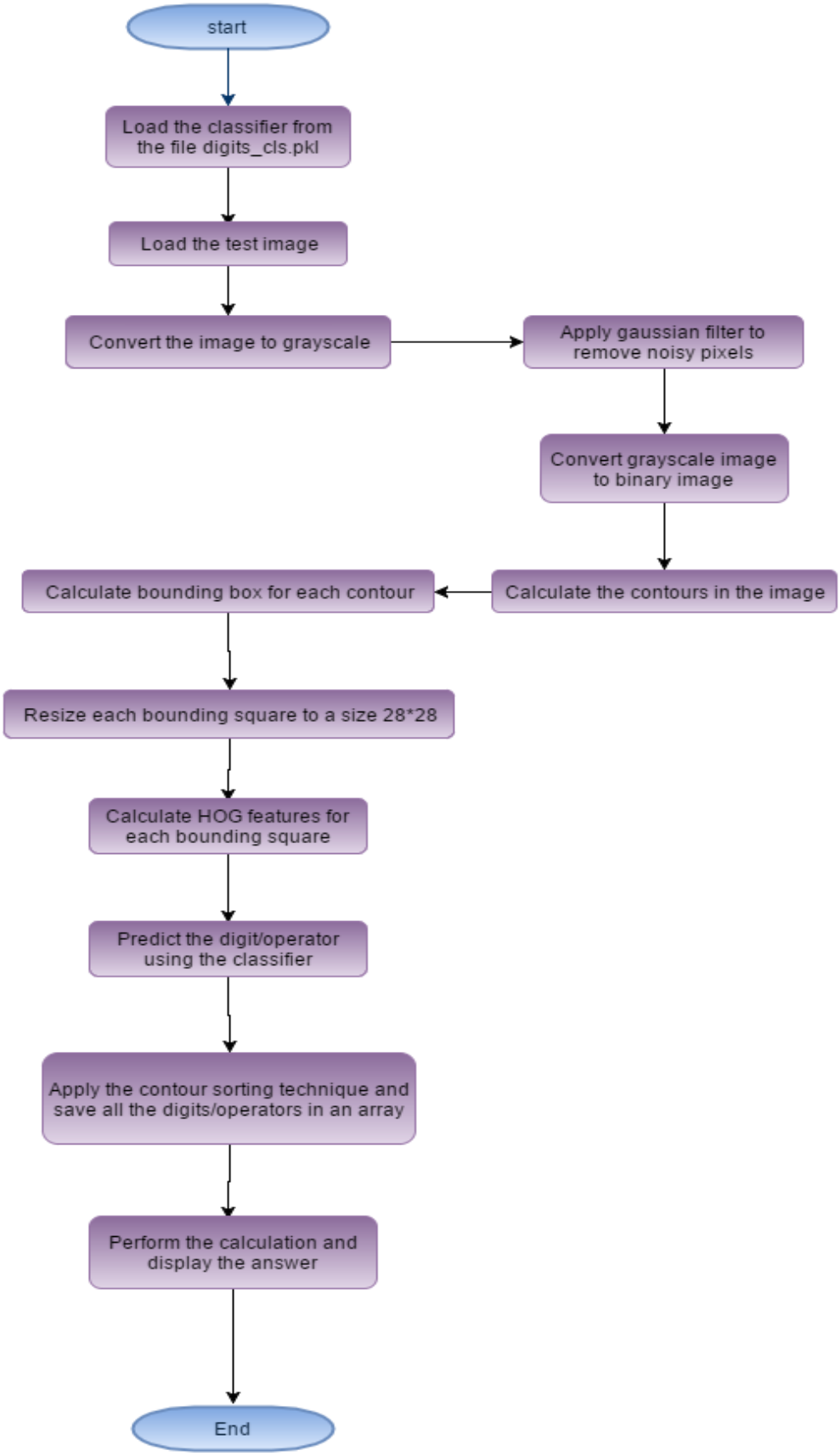


Figure 6.3 - Testing a classifier

Testing the classifier on this image -

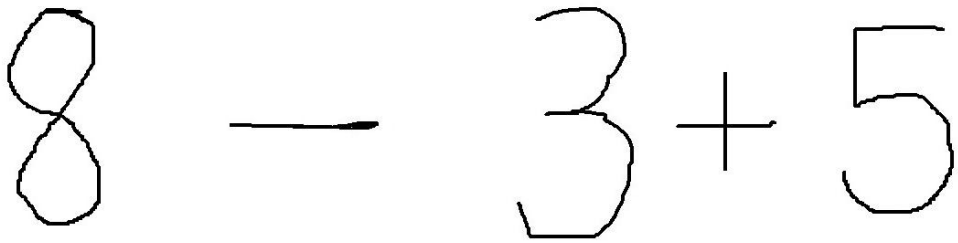


Figure 6.4 - Input image for testing the classifier

The resulting image, after running the second script was –

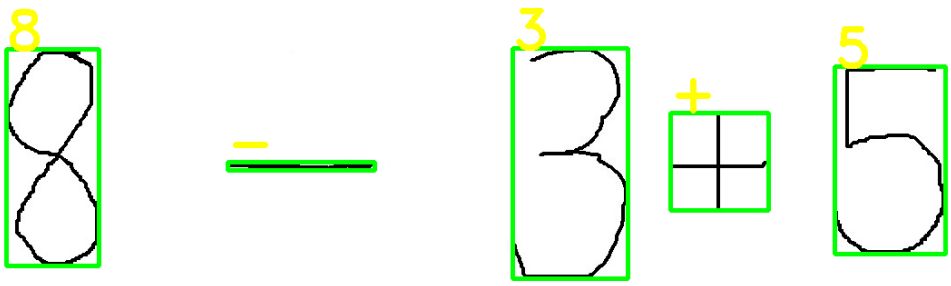


Figure 6.5 - Resultant image

So, the results are pretty good. Here is another result –

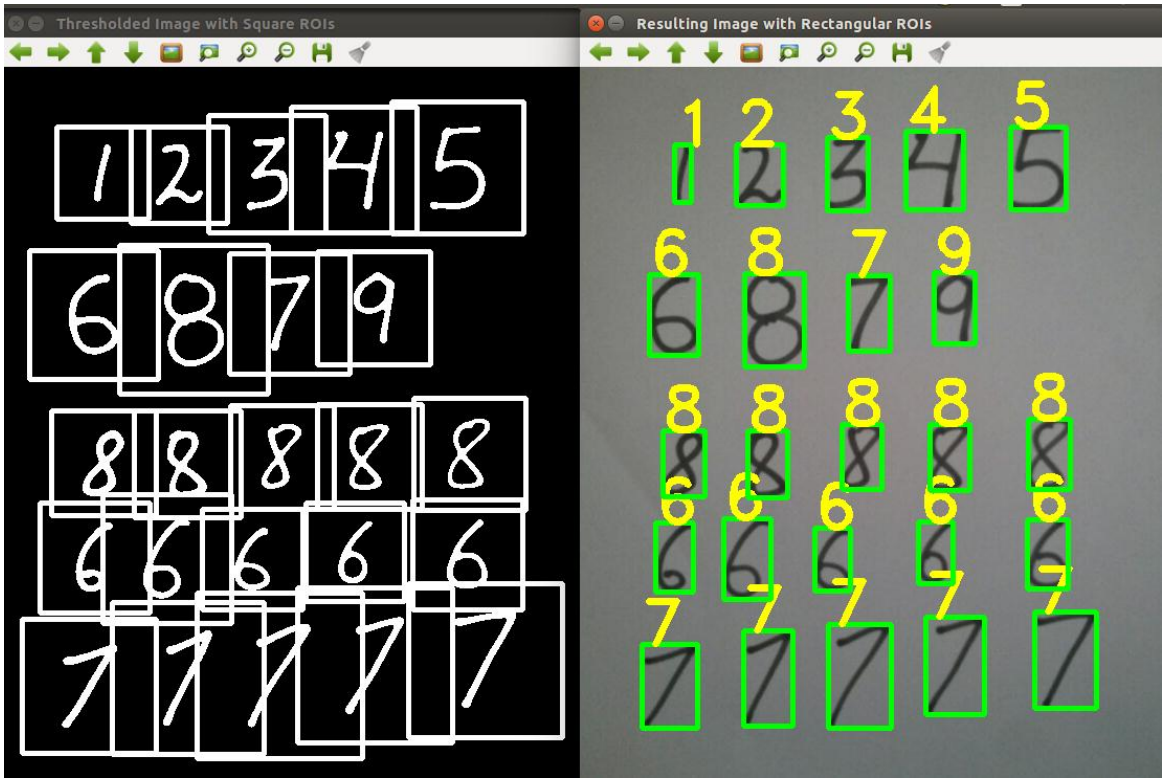


Figure 6.6 - Results of recognized digits

In the image on the left hand side, we display the thresholded image with a square around each digit. Each of this square region is then resized to a 28×28 image. After resizing, we calculate the HOG features of this square region and then using these HOG features we predict the digit. In the image on the right hand side, we display the predicted digit for each handwritten sample bounded in the rectangular box.

6.4 Assumptions during testing

There are a few assumptions, we have assumed in the testing images –

1. The digits should be sufficiently apart from each other. Otherwise if the digits are too close, they will interfere in the square region around each digit. In this case, we will need to create a new square image and then we need to copy the contour in that square image.

2. For the images which we used in testing, fixed thresholding worked pretty well. In most real world images, fixed thresholding does not produce good results. In this case, we need to use adaptive thresholding.
3. In the pre-processing step, we only did Gaussian blurring. In most situations, on the binary image we will need to open and close the image to remove small noise pixels and fill small holes.

CHAPTER-7

CONCLUSION

Automatic handwriting recognition is of academic and commercial interest. Current algorithms already excel at learning to recognize handwritten digits. Post offices use them to sort letters; banks use them to read personal checks. Some predict that in the near future billions of handheld devices such as cell phones will have handwriting recognition capabilities. In recent decades neural networks have been overshadowed by the very useful but principally less general and less powerful support vector machines as well as other more specialized machine learning methods.

The system described in this thesis has already performed well in a practical setting, as it was used to efficiently typeset many of the mathematical expressions in this thesis. Furthermore, recognition accuracy rates within its domain have been very promising. However, it should be viewed as only a first step towards truly robust recognition of handwritten mathematics. If these techniques are to be extended to a truly practical setting, a number of limitations will need to be over-come. Some of these limitations constitute simple improvements to various parts of the system, while other limitations arise at a more fundamental level.

CHAPTER-8

FUTURE SCOPE OF PROJECT

The current model is still a single user system, since the symbol recognition algorithm is naturally sensitive to variations in writing that were not present in the training data, such as that between examples written by different writers. Modelling this variation is difficult, and still an active area of research in handwriting recognition. One possible solution is a user adaptive system, which uses erroneously classified symbols to adjust its symbol models to a particular user. Another is to use a mixture model for each individual symbol. No matter how it is approached, the problem of creating a truly user in-dependent system lies largely in improving isolated symbol recognition, since the other elements of the system are not nearly as sensitive to the variations in other users' writing.

Of the three main sub-problems of this thesis, the stroke partitioning algorithm is perhaps the most robust. The system is currently able to correctly partition the symbols in expressions which it is entirely unable to parse, indicating that this portion of the research should scale very well to more complex expressions. There still remains some work to be done on normalizing the costs of multiple stroke symbols with those of single stroke symbols. I view this problem to actually be a problem in the symbol recognition algorithm though, rather than a limitation of the stroke partitioning algorithm.

In addition, the area where the most research still needs to be done is in correctly parsing complex expressions. In particular, for the system to interpret more than simple expressions it will be necessary to incorporate a more complete use of symbol baselines into the parsing algorithm. The expressions in this thesis which could not be typeset using the system typically had either complex superscript or subscript forms or matrix notations. A better understanding of how the baseline of symbols effects the structure of the expression will be necessary if this problem is to be solved.

Finally, there is still much work to be done on the user interface. As more complex functionality is added to the system, the limitations of a pen interface will quickly become apparent. At the very least, a full functioned equation editor will need to be incorporated within the system.

BIBLIOGRAPHY

- http://cath.lu/files/CS221-Project.pdf
- http://neuralnetworksanddeeplearning.com/chap1.html
- http://yann.lecun.com/exdb/publis/pdf/matan-92.pdf
- http://www.ace.tuiasi.ro/users/103/7_F4_09_Sandu.pdf
- https://chrisjmccormick.wordpress.com
- http://www.learnpython.org/