Interactive Systems Labs

Carnegie Mellon University
Pittsburgh, PA, USA

University of Karlsruhe
Germany

Run-On Recognition in an On-line Handwriting Recognition System

REPORT

Ralph Groß

Supervisors:

Stefan Manke

Alex Waibel

June 1997

Contents

Acknowledgments

I would like to thank Alex Waibel for arousing my interest in handwriting recognition and Stefan Manke for his advice and guidance during this project. I would also like to thank Uwe Meier for his help with technical problems and Wolfgang Hürst for the various discussions we had on this subject.

Chapter 1

Introduction

In todays computer systems, the interaction between user and computer still has to be done by keyboard and mouse. On the other hand humans use a vast amount of different modalities when they communicate with each other. For example, they speak, write, point and gesture to express their thoughts, intentions or feelings. If the context does not impose any restriction, humans usually select the modality which appears to be the most natural in a given situation. To accommodate this fact, a lot of research has been conducted in the past years to enable computers to offer different input modalities. Especially the field of handwriting recognition has seen a lot of progress. As a matter of fact, there are already a lot of commercial systems available, which offer handwriting recognition. Applications range from mobile computer systems and special data entry tasks to the reading of address labels and verification of signatures.

1.1 Handwriting Recognition

In general, handwriting recognition systems can be classified according to the type of input data they are working on. So-called Optical Character Recognition Systems (OCR) deal with scanned text or words, whereas on-line Character Recognition Systems (OCLR) work on a sequence of data points generated by writing on a touch-screen or a graphic tablet. In the latter case, the dynamic writing information is available which facilitates the recognition task. Therefore, OCLR systems usually show a better performance than OCR systems (see [YMS92] and [OWM92] for surveys of OCR and OCLR systems).

1.2 Run-On Recognition

For the task of human-computer interaction, OCLR systems are usually used. In order to provide an easy-to-use natural interface, the handwriting recognizer

should impose as few restrictions on the input style as possible. Idealistically, the recognizer works in the background, constantly transforming the handwritten input into characters, words or, in the case of gestures, meanings. This kind of recognition, where the recognizer keeps up with the input of the user is called *Run-On Recognition*. Unfortunately, a lot of recognizer do not support it. Due to the way the preprocessing and the recognition component is designed, input can only be processed in certain portions. This usually forces the user to explicitly start the recognition by pushing a button, which makes the whole interaction slow and unpleasant.

1.3 Objective of this work

In this report we will describe the work we have been doing within the **NPen**$^{++}$ handwriting recognition system. **NPen**$^{++}$ is a writer independent large vocabulary on-line handwriting recognition system for the recognition of single words, written in any kind of writing style: printed (all characters separated from each other), cursive (no lifting of the pen within the word) or any mixture of both. It was mainly developed by Stefan Manke at the University of Karlsruhe, Germany ([MB94, MFW94, MFW95a, MFW95b, MFW96]). The purpose of our research was to enable **NPen**$^{++}$ to do run-on recognition. Because of the setup of the recognition engine, all further considerations should be seen in the context of a single word on-line handwriting recognition system for words written by using Latin characters (unless otherwise stated).
The report is organized as follows. Chapter 2 describes the **NPen**$^{++}$ system as it was available when this work started. In chapter 3 the incremental preprocessing and the changes to the recognition engine are presented. The results of the evaluation of the whole system are given in chapter 4. Finally, chapter 5 summarizes these results and outlines future work.

Chapter 2

The NPen^{++} system

In contrast to the architectural uniformity of todays state-of-the-art speech recognition systems, one can find a large variety in the way current handwriting recognizers are designed. They differ not only in the type of the recognition engine (e.g. template matcher, neural net, HMM), but also in fundamental aspects of the preprocessing (e.g. segmentation of the input before recognition or not). This diversity can even be found on the input level, as some systems require the user to write in a certain style. Bearing this in mind, **NPen**$^{++}$ can be considered as being one of the most powerful handwriting recognizers available. It accepts any kind of input style and recognizes words out of large vocabularies (up to 100 000 words). The system is writer independent, so there is no need to retrain or adapt the system to a new user.
This chapter describes the **NPen**$^{++}$ system in all its components. After a brief systems' overview in 2.1, the preprocessing is described in detail in section 2.2. The following sections deal with the recognition engine and the search component (2.3).

2.1 System Overview

As stated earlier, **NPen**$^{++}$ is an on-line handwriting recognizer. Therefore, it uses a time ordered sequence of data points as input, obtained by an LCD tablet or a touch-screen (an example of the input can be seen in figure 2.1).

Figure 2.2 shows the system with its two major modules: preprocessing and recognition. The preprocessing transforms the original point sequence into a still temporal sequence of feature vectors. This stream of feature vectors is then given to a neural network based recognizer.

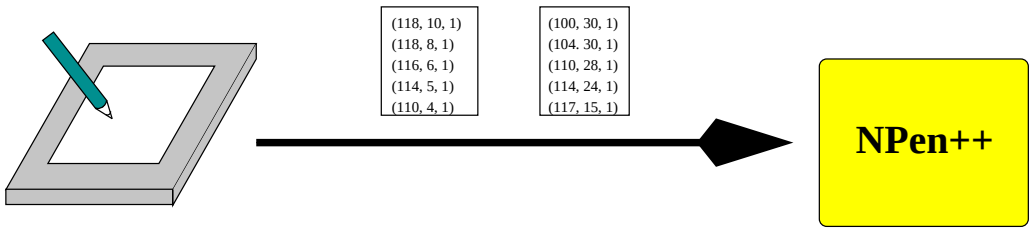

Figure 2.1: Diagram of the input format.

2.2 Preprocessing

The reason why handwriting recognition is a difficult task is obvious. Different people have significantly different handwritings and it is not likely that two words, written by two writers look the same. Therefore, recognizers have to deal with a large amount of variability (see figure 2.3).

Instead of using the raw data from the input device, all existing systems perform a couple of preprocessing steps before the recognition engine comes to work. This approach facilitates the recognition process significantly. The techniques applied during preprocessing can be divided into two groups:

- Normalization
  reduces meaningless variability which **does not** help to discriminate between classes, therefore builds up invariance

- Feature Extraction
  enhances variability which **does** help discrimination between classes

As already mentioned, there is a fundamental difference in todays handwriting recognizers regarding the "atomic" unit the recognizer is handling. One philosophy is to segment the input sequence into character or sub-character units and then perform recognition on those pieces. In most of the cases, the segmentation is done using heuristic rules, which is error-prone. Therefore, a lot of systems use whole words as basic units. The segmentation is implicitly done by the recognizer during the recognition run, which usually gives better results in terms of word accuracy. **NPen**$^{++}$ follows the latter approach and does not perform segmentation prior to recognition. For that reason, all preprocessing steps always involve the whole word.
The following subsections describe the essential preprocessing steps conducted by **NPen**$^{++}$.

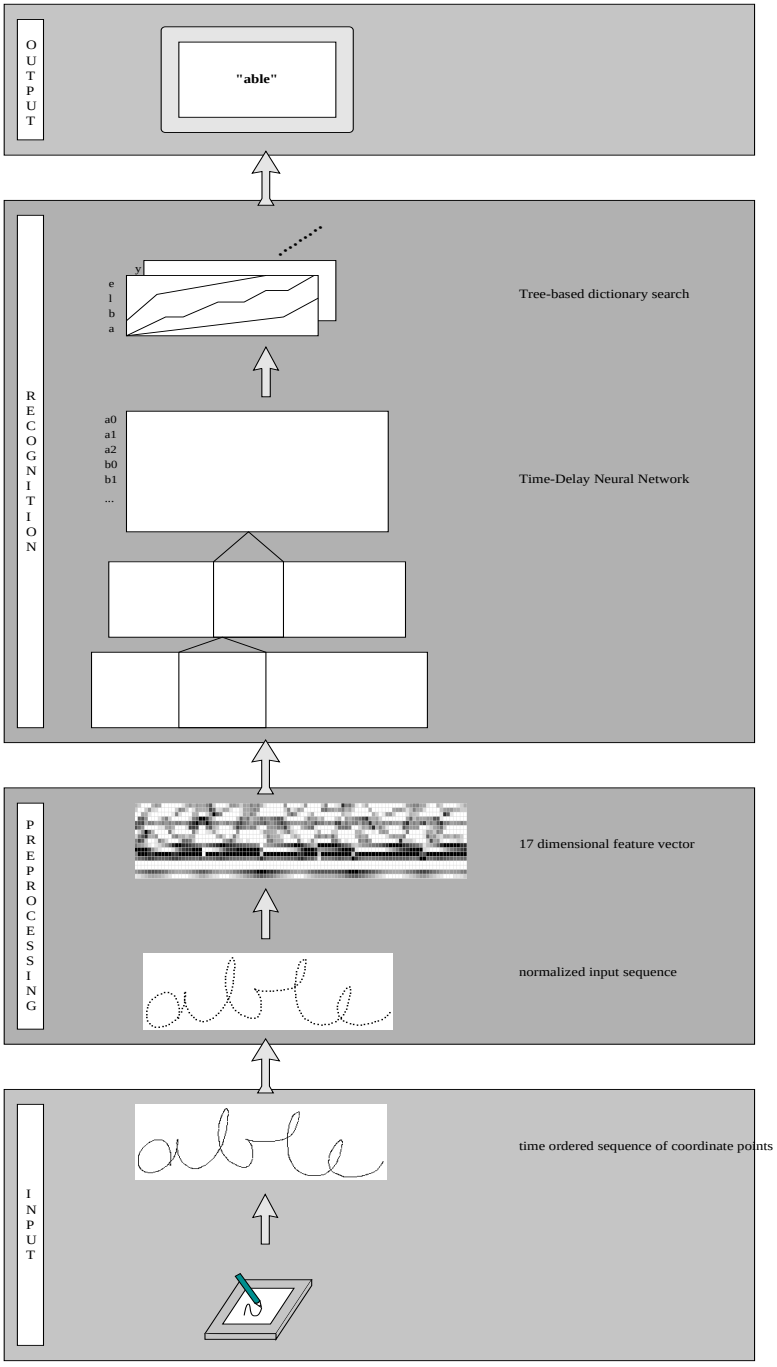

Figure 2.2: Overview of the **NPen**$^{++}$- System.

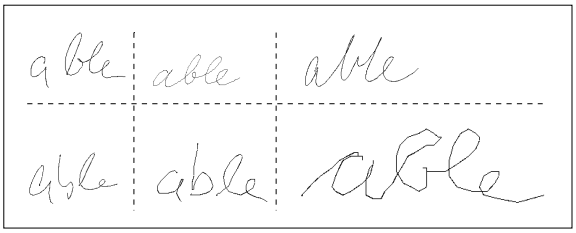

Figure 2.3: Examples for variations in handwriting.

2.2.1 Normalization

Resampling

The input device is sampled in fixed intervals, therefore, the raw data points are equidistant *in time*. As the writing speed is not constant (between different writers, for one writer within a single word or character), the number of points collected in different sections of a character varies a lot (see figure 2.4).

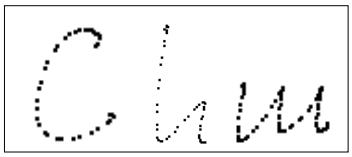

Figure 2.4: Examples for variations in writing speed.

In addition, different input devices work with different sampling rates. In order to eliminate this variability, the original point sequence is resampled to be equidistant *in space*. This means that after resampling the Euclidean distance d between any two consecutive points has the same value. The value of d is determined from the height of the center of the word (the so-called *core height*, see following paragraph on baseline normalization). The algorithm performing the resampling uses simple linear interpolation. For some of the feature extraction steps, it is important that the input sequence is continuous, which is obviously not the case between the locations of a pen-lift and a pen-down. Therefore, these points are included in the resampling, which means that the algorithm interpolates a connecting line between every pen-up and the following pen-down. Figure 2.5 shows an original and a resampled point sequence.

Smoothing

Just like any real world system, **NPen**$^{++}$ has to deal with noise or imperfections in input data. These imperfections originate from hardware problems, limited

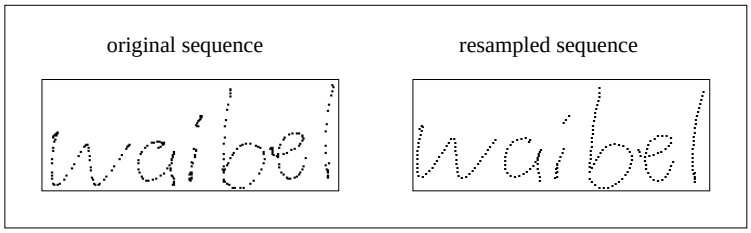

Figure 2.5: Example for a resampling step.

accuracy of the input device, erratic hand motions and inaccuracies of pen-down indications. A popular technique to deal with these problems is *smoothing*. This technique averages the point position of every data point with a certain number of neighbor points:
with x_{p_i} being the x-position of point p_i, the new x-coordinate of p_i, $\hat{x}_{p_i}$ can be calculated as:

$$\hat{x}_{p_i} = c_{i-n} \cdot x_{p_{i-n}} + \ldots + c_i \cdot x_{p_i} + \ldots + c_{i+m} \cdot x_{p_{i+m}}$$

n, m: number of preceding / succeeding points used for the calculation
$c_{i-n}, \ldots, c_{i+m}$: weighting factors
The new value $\hat{y}_{p_i}$ of y_{p_i} is calculated accordingly.

In **NPen**$^{++}$ the best results could be obtained by using $n = m = 2$. Figure 2.6 shows a word before and after smoothing.

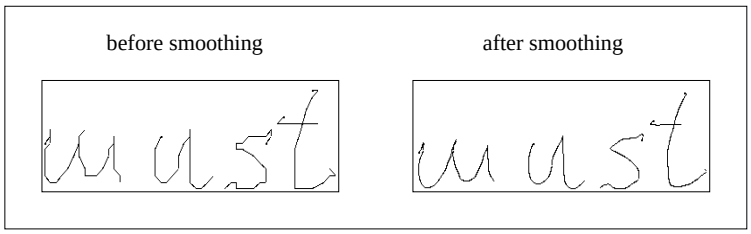

Figure 2.6: Example for smoothing.

Removal of crosses and dots

A further source for unwanted variability in the input data are diacritical marks such as dots (as found on "i", "j") or crosses (like in "t"). Unfortunately, people are not following consistent rules on *when* to draw these points or crosses. They do it:

- right after the "main" part of the character was written

- after the whole word was written
- after any pen-up following this character

This obviously is a problem for time-ordered representations. The standard procedure to address this problem is to remove delayed diacritical marks (e.g. [Sch93]) or trying to reinsert them at a better position in the stroke sequence (e.g. [MBPV93]). In **NPen**$^{++}$ the decision was made to remove delayed diacritical marks (using a heuristic approach), but capture the information of their presence by a binary feature, the so-called *hat feature*. Experiments show that this leads to improvements in word accuracy in the order of $1.5-2\%$ depending on the task. These removal procedures have been left out in the incremental system.

Baseline normalization

One of the major source of variance between the handwriting of different writers is the difference in size. Therefore, it is very important to create a normalized input representation, which is invariant against scaling. In **NPen**$^{++}$, this is achieved by performing a so-called baseline normalization (the idea follows [BL94]). Here, the first step is to fit a geometric model to the input data. The model comprises four "flexible" lines representing respectively the ascender line, the core line, the baseline and the descender line (see figure 2.7).

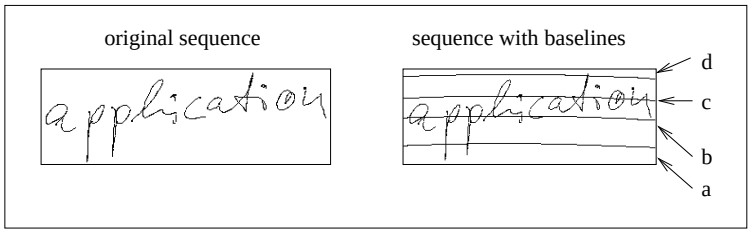

Figure 2.7: Original word and word with baselines: a (descender line), b (baseline), c (core line), d (ascender line).

The adaption is done by using an Expectation Maximization (EM) approach. Once the lines are determined, the baseline can be used to rotate the word to a nearly horizontal orientation. This ensures that the normalized word is invariant against rotation. The distance between the baseline and the centerline (the so-called *core height*) is scaled to a fixed value, so that the size of the whole word is normalized (see figure 2.8).

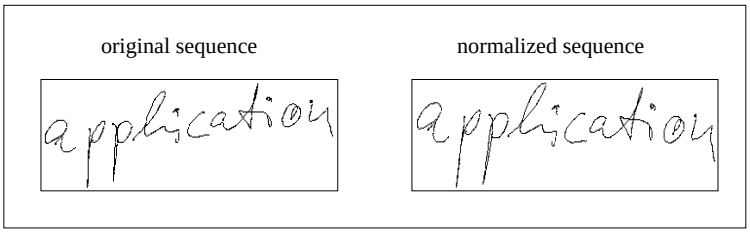

Figure 2.8: Original word and word after baseline normalization.

2.2.2 Feature Extraction

The purpose of the feature extraction is to enhance the variability which helps to discriminate between classes. For each data point of the normalized coordinate sequence, a N-dimensional feature vector is calculated. The system uses $N = 17$ features. The general idea of the selected features is to extract low level topological information and leave the extraction of high level features to the connectionist recognizer.

Absolute y position

The lines computed during baseline normalization divide the word into three areas: upper, medium and lower area (see above paragraph on baseline normalization). This information can be fed into the recognizer with the y position of the data points. Instead of using the pure y-coordinate the vertical position of the coordinate points is expressed relative to the centerline and baseline. The x position of the input points has been left out of the feature set. Even calculated relative to the predecessor point, the x position depends on the word length, which is *not* normalized during preprocessing. Therefore a feature based on the x position would not be bound as the described y-feature is.

Local angle information

The features described in this section provide information about the direction and the curvature of the trajectory at a given instant of time.
Direction: The direction of a stroke is determined by a discrete approximation of the first derivatives with respect to the arc length, $\frac{dx}{ds}$ and $\frac{dy}{ds}$, where $ds = \sqrt{dx^2 + dy^2}$
The approximations can be calculated as:

$$\cos\theta(n) = \frac{\Delta x(n)}{\Delta s(n)}$$

$$\sin\theta(n) = \frac{\Delta y(n)}{\Delta s(n)}$$

where

$$\Delta x(n) = x(n+1) - x(n-1)$$

$$\Delta y(n) = y(n+1) - y(n-1)$$

$$\Delta s(n) = \sqrt{\Delta x(n)^2 + \Delta y(n)^2}$$

(see figure 2.9).

This representation has various advantages: it does not require the computation of transcendental functions, the feature values are bound and for a smooth curve the parameters change smoothly. Therefore, it is important that the resampling also includes the points where a pen-up/pen-down takes place.

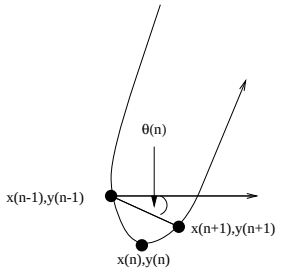

Figure 2.9: Estimation of writing direction.

Curvature: As the second derivatives $\frac{d^2x}{ds^2}$ and $\frac{d^2y}{ds^2}$ are not bound, the local curvature is approximated by the angle between two elementary segments: $\phi(n) = \theta(n+1) - \theta(n-1)$ (see figure 2.10). This angle is also encoded by its cosine and sine. Using the subtraction formulas for sine and cosine these values can be computed as:

$$\begin{aligned} \cos\phi(n) &= \cos(\theta(n+1) - \theta(n-1)) \\ &= \cos\theta(n+1) \cdot \cos\theta(n-1) + \sin\theta(n+1) \cdot \sin\theta(n-1) \end{aligned}$$

$$\begin{aligned} \sin\phi(n) &= \sin(\theta(n+1) - \theta(n-1)) \\ &= \sin\theta(n+1) \cdot \cos\theta(n-1) - \cos\theta(n+1) \cdot \sin\theta(n-1) \end{aligned}$$

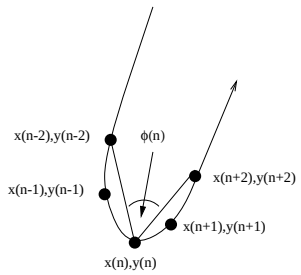

Figure 2.10: Estimation of curvature.

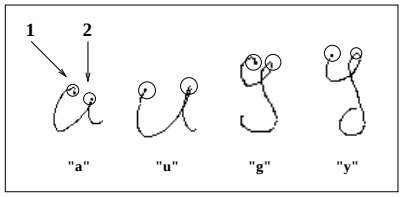

Figure 2.11: Examples for hard to detect differences between cursive characters.

Context Bitmaps

All the features described so far are strictly local in both time and space. Recognizers using only local features showed significant problems in discriminating between cursive letters like "a" and "u" or "g" and "y", which differ only in very small regions of the character (see figure 2.11).

Since the feature vectors are ordered in time and the points of region 2 were written a lot later than the points of region 1 there is no immanent connection between them in the data. Therefore, the recognizer does not obtain any information on the relative position of those points to each other and the characters remain indistinguishable. To overcome these problems, the recognizer has to get a notion of the vicinity of each data point. This information can be derived from a grey scale bitmap representation $B = b(i, j)$ of the normalized coordinate sequence. The bitmap is created by counting the number of points (x_t, y_t) of the sequence, which fall into pixel (i, j) (see figure 2.12).

For every data point (x_t, y_t) a $d \times d$ section centered around the corresponding pixel (i, j) is determined. This section is then down sampled to a 3×3 bitmap. The resulting nine values are the new feature values. These features are still local in space but global in time (see figure 2.13). Adding these so-called context bitmaps to the feature set of **NPen**$^{++}$ resulted in a 50% error reduction. They are able to model temporal long range and spatial short range dependencies as they occur in pen trajectories.

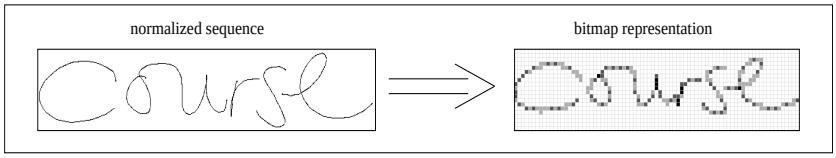

Figure 2.12: Example for bitmap representation of normalized coordinate sequence.

Hat Feature

As already mentioned, delayed crosses and dots are removed from the point sequence. The information on where these points were located is captured by the binary hat feature. It marks every remaining point of the point sequence which was originally covered by a delayed cross or dot (see figure 2.14).

Since removing of dots and crosses is not done in the incremental system, this feature is left out of the feature set of the incremental system.

Pen Feature

The resampling algorithm also includes the points where pen-ups and pen-downs take place, so that a sequence of points is interpolated between those locations. The binary pen feature indicates which of the data points are is generated by a real pen movement and which are interpolated points between a pen-up and a pen-down. An example for a whole feature vector can be found in figure 2.15.

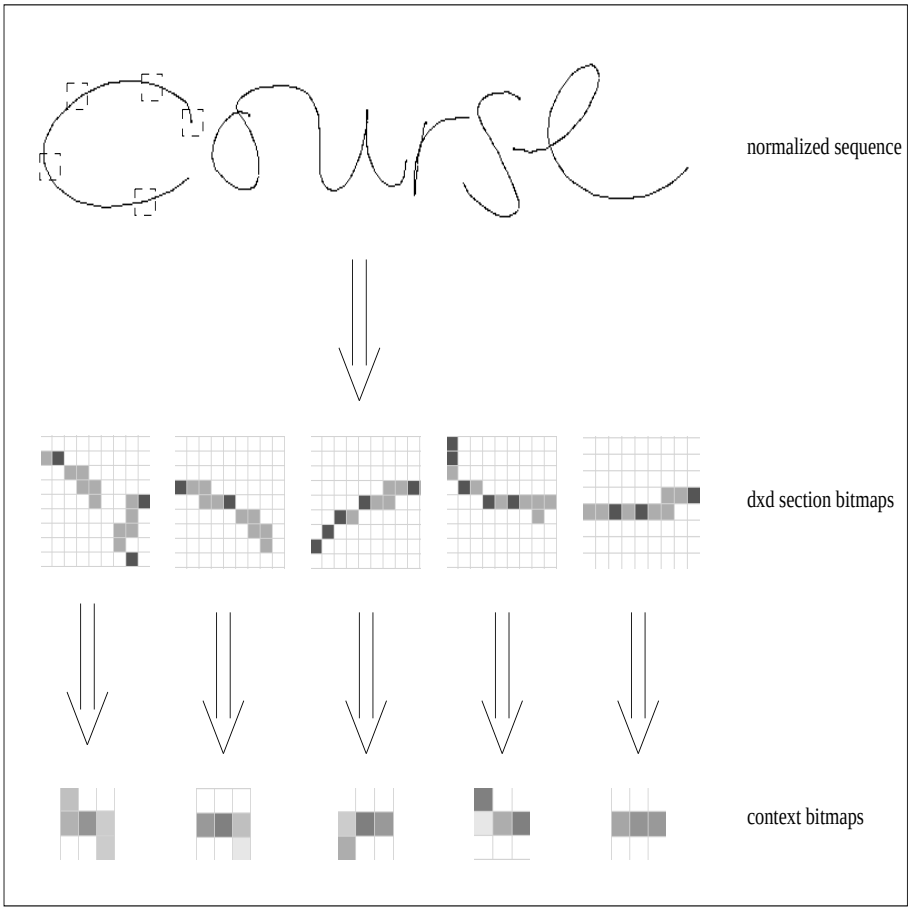

Figure 2.13: Calculation of context bitmaps with $d = 9$.

2.3 The NPen^{++} recognizer

Like in speech recognition, the main problem of recognizing continuous words is that character or stroke boundaries are not known (especially if there are no pen lifts or white spaces that indicate these boundaries). The **NPen^{++}** recognition engine integrates recognition and segmentation of words into a single network architecture, the so-called Multi-State Time Delay Neural Network (MS-TDNN), which was originally proposed for continuous speech recognition tasks (see [BHMW93, HW92]). The Time Delay Neural Network (TDNN) [WHH$^+$89] with its time-shift invariant architecture has been applied successfully to on-line *single* character recognition [GAL$^+$91]. The MS-TDNN combines the high ac-

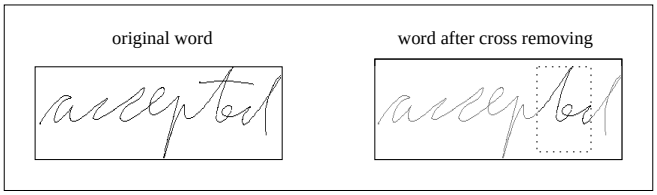

Figure 2.14: Original word and word after cross removing. Emphasized part represents the area where the value of the hat feature is 1.

curacy pattern recognition capabilities of a TDNN with a non-linear time alignment algorithm (dynamic time warping) in order to find an optimal alignment between stroke and characters in handwritten words. The following section describes the network architecture, the training procedures and the search techniques used in the recognition component of **NPen**$^{++}$.

2.3.1 Modeling assumptions

Let $W = w_1, \ldots, w_K$ be a dictionary consisting of K words. Each of these words w_i is represented as a sequence of characters $w_i \equiv c_{i_1} c_{i_2} \ldots c_{i_n}$. Each character c_j itself is modeled by a three state Hidden Markov Model (HMM) $c_j \equiv q_j^0 q_j^1 q_j^2$ (for an introduction to HMMs see [Rab89]). Inside these models, only self-loops $q_{j_k} \rightarrow q_{j_k}$ and state transitions $q_{j_k} \rightarrow q_{j_{k+1}}$ are allowed. The self-loop probability $p(q_{i_j} \mid q_{i_j})$ and the transition probabilities $p(q_{i_j} \mid q_{i_{j-1}})$ are both set to $\frac{1}{2}$, while all the other transition probabilities are set to zero. The idea behind using three states per character is to model explicitly the initial, middle and final section of the character. A word w_i is then modelled by the sequence of the states of its characters: $w_i \equiv q_{i_0} q_{i_1} \ldots q_{j_{3k}}$.
During recognition of an unknown sequence of feature vectors $x_0^T = x_0 \ldots x_T$ the recognizer has to find the word $w_j \in W$ in the dictionary that maximizes the *a posteriori* probability $p(w_i \mid x_0^T, \theta)$ given a fixed set of parameters θ and the feature sequence x_0^T. The recognized word then satisfies:

$$w_j = \text{argmax}_{\,w_i \in W} p(w_i \mid x_0^T, \theta)$$

Using Bayes' rule, the probability $p(w_i \mid x_0^T, \theta)$ can be expressed as

$$p(w_i \mid x_0^T, \theta) = \frac{p(x_0^T \mid w_i, \theta) P(w_i \mid \theta)}{p(x_0^T \mid \theta)}$$

Instead of approximating $p(w_i \mid x_0^T, \theta)$ directly the MS-TDNN models the likelihood of the feature vector sequence $p(x_0^T \mid w_i, \theta)$.

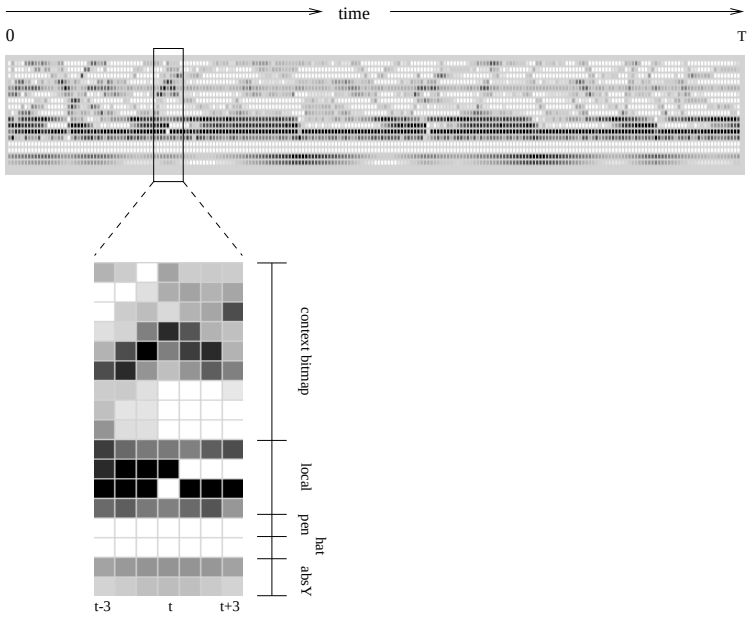

Figure 2.15: Complete feature vector for the word "able".

2.3.2 The MS-TDNN architecture

As stated earlier, the TDNN and the MS-TDNN architectures were originally developed for speech recognition. The objective was to develop a neural network able to represent relationships between events in time. The actual features or abstractions learned by the neural network should be invariant under translations in time. In order to achieve that every neuron is not only exposed to the current input, but also to the temporal vicinity (through so-called *time delays*). This enables a TDNN unit to relate and compare the current input to events happening shortly before or afterwards. Another way of seeing the net is depicted in figure 2.16. In this case a time window is shifted over the time varying input and over the activations in the hidden layer as well. The output of the TDNN are activations in the so-called *states layer*, which can be interpreted as an estimate of the probabilities of the states q_j given the input window $x_{t-d}^{t+d} = x_{t-d} \dots x_{t+d}$ for each time frame t. In order to recognize words, which are sequences of characters, the best sequence through the states in the states layer has to be found. This is done by the so-called *dynamic time warping (DTW)* algorithm. The activations from the states layer are simply copied into the DTW layer. The DTW algorithm finds an optimal alignment path for each word and the sum of all activations along this optimal path is taken as the score for the word output unit.

The MS-TDNN is trained with standard back-propagation. In a first step, the network is trained in a forced alignment mode, during which hand-segmented training data is used. The back-propagation here starts at the states layer of the front-end TDNN with fixed state boundaries. After a certain number of iterations, the forced alignment is replaced by a free alignment found by the DTW algorithm. Now training starts at the word level of the MS-TDNN and is performed on unsegmented training data.

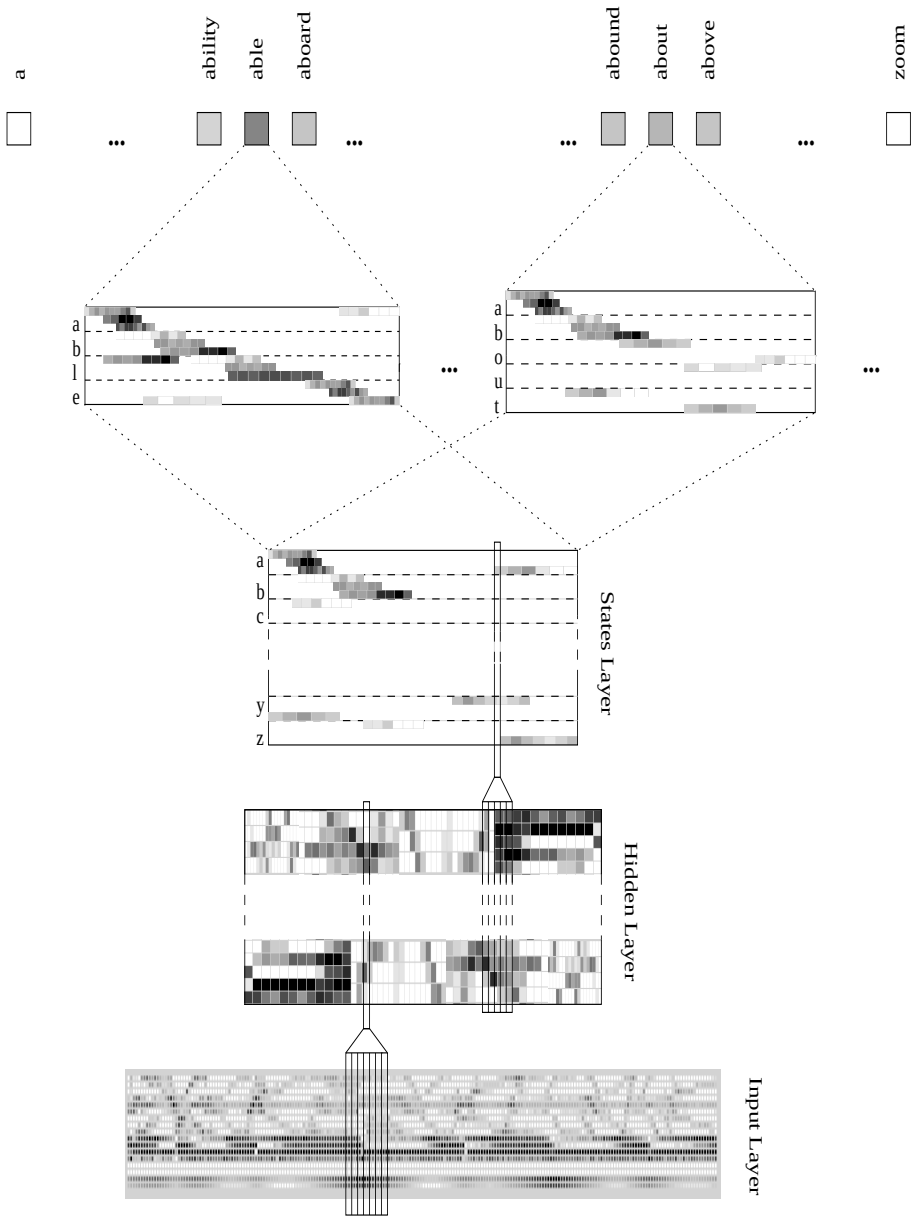

Figure 2.16: The Multi-State TDNN architecture used in this work. It comprises 17 input units, 40 units in the hidden layer and 78 state output units. There are 7 time delays in the input layer and 5 time delays in the hidden layer. The representation of the search is simplified. See section 2.3.3 for an exact version.

2.3.3 Search Technique

The search for the most likely word out of the dictionary consists of two steps: First, the most likely state sequence in *every word of the dictionary* has to be determined and then the word with the highest total score has to be selected. If the search is executed exactly as described one ends up searching through a *flat dictionary* structure as depicted in figure 2.17. This is only feasible as long as the number of words in the dictionary is small.

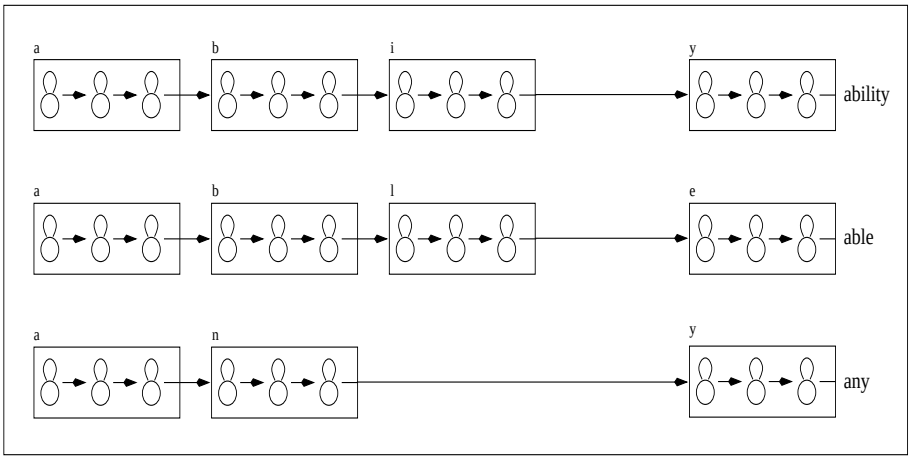

Figure 2.17: Flat dictionary structure.

Since the run time for a recognition pass scales linearly with the number of entries in the dictionary, this approach can not be used for building very large vocabulary neural network recognizer. For a 100k dictionary, the search engine would have to evaluate 968,005 different HMMs. To reduce the number of different HMMs the organization of the dictionary was changed, extending the MS-TDNN to a tree-based TDNN. For each letter a search tree for all words starting with this character is built. The nodes in each tree consist of HMMs representing the individual letters (see figure 2.18).

This technique reduces the ratio of the number of HMMs to the size of the dictionary from 10 for the flat structure to 3. If a 100k dictionary is used, the number of different HMMs to be evaluated drops down to 277,382. In order to further reduce this number a beam search is performed instead of a search over all models. This means that all branches whose accumulated scores are below a certain threshold (called beam) are pruned. Experiments show that the tree search with pruning is about 15 times faster than the flat search. This allows running **NPen**$^{++}$ in real-time with dictionary sizes up to 100,000 words [MFW96].

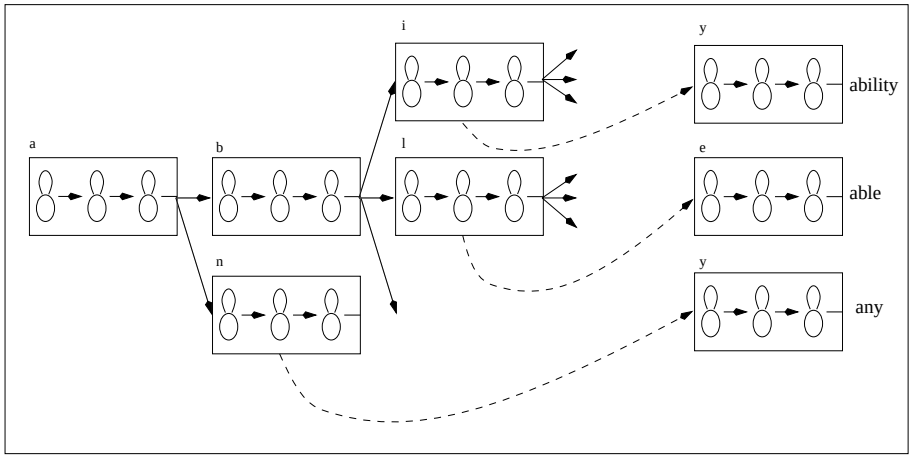

Figure 2.18: Tree structured dictionary.

Chapter 3

The Run-On System

In general, users want computer systems to be fast. This is especially true for interactive systems, where input and output are interleaved, allowing the user's input to depend on earlier output from the same run. In this case, small differences in response times can decide whether a system is usable or not. The most obvious way to tackle this problem is to speed up the recognizer. Of course, this can not be done ad infinitum especially if the system is supposed to handle large vocabularies with a high recognition accuracy. If the recognizer itself can not be made any faster, the next idea is to just start the recognition process earlier. This can only be done by overlapping the user input with the recognition. Thus, the recognizer has to be enabled to do *run-on recognition*. Being convenient for an isolated word recognizer, the ability to do run-on recognition is almost a must for continuous word recognizers.
This chapter is organized as follows. The first section describes the basic architectural concepts of the run-on system. Sections 2 and 3 then show how incremental preprocessing and recognition are done in **NPen**$^{++}$.

3.1 System Architecture

In order to be able to accept user input and to do recognition of the previous input at the same time the recognizer is split into two separate processes (see figure 3.1).

The first process is constantly recording the coordinates of the pen movement. After a certain number of data points has been collected (and the recognizer has finished the processing of the previous input), the data collecting process sends the points to the recognition engine, where preprocessing and recognition is done. An important question here is how the coordinates are grouped. The optimization of this process has to deal with two conflicting goals. First, the recognizer should produce results as early and as often as pos-

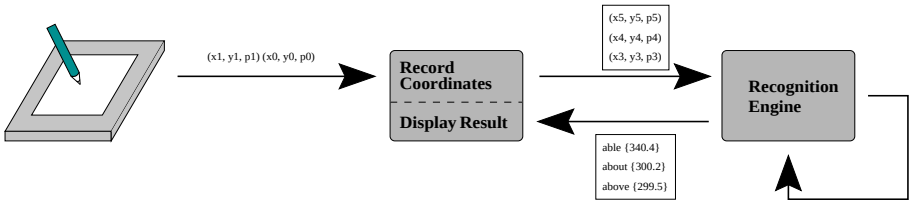

Figure 3.1: Overview of the Run-On System.

sible, which implies sending the coordinate bunches as soon as possible (with small amounts of coordinates). On the other hand, it only makes sense to start a recognition run if there is enough information in the coordinate sequence, e.g. data points of at least one complete character. Since it usually can not be determined where a character ends (for cursive script), a simple strategy would be to just put together as much coordinates as possible. The compromise implemented in **NPen**$^{++}$ is to group the data points according to the number of local extrema in the sequence. After a certain number of local minima and maxima is reached the coordinates are passed to the recognizer. The value of these thresholds together with the length of the initial sequence is optimized on the given data set (see section 4.3).
As will be shown in section 3.2, incremental preprocessing is not always possible for every coordinate group. First of all the initial coordinate sequence is processed using the normal methods. Subsequent coordinate bunches have to fulfill certain conditions in order to be preprocessed incrementally. In case the corresponding check fails, the whole previous sequence (including the new data points) has to be preprocessed and recognized together. If incremental preprocessing is possible the new bunch of feature vectors is given to the recognizer. The result of the recognition run is then presented to the user. Figure 3.2 depicts this approach for the incremental preprocessing.

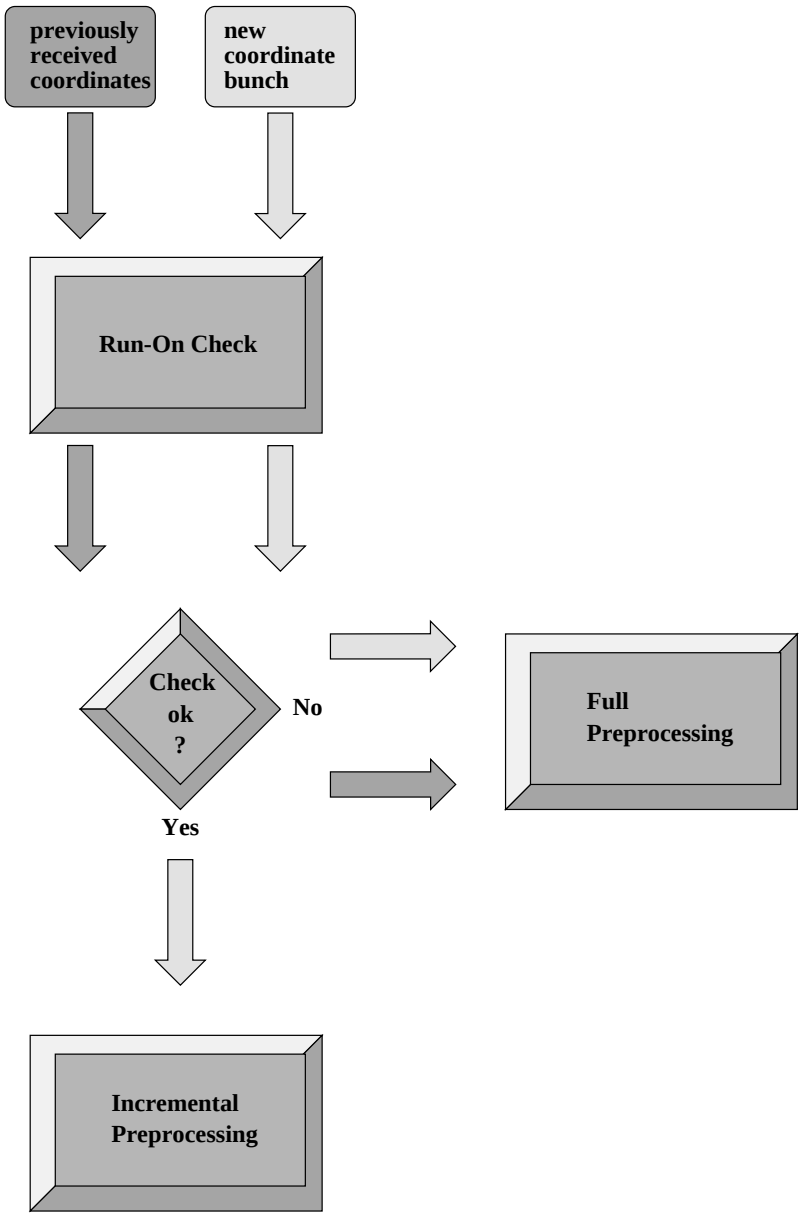

Figure 3.2: Preprocessing cycle of the run-on system.

3.2 Incremental Preprocessing

Seen in a more abstract way, doing incremental preprocessing means doing preprocessing without knowing the whole word. This already leads to the first problem: scaling. As it is stated in section 2.2, scaling the handwritten word to a fixed size and therefore eliminating size variations *between words* is an important preprocessing step. It is very likely that the height of a word is not constant along its trajectory. By scaling different parts to the same size, which would happen if the parts are scaled independently from each other, the variations present in the original word would disappear (remove size variations *within* words). The consequence is a distorted word. An example for this erroneous processing can be found in figure 3.3.

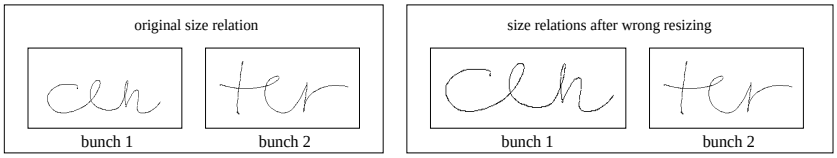

Figure 3.3: Example for wrong scaling of the word "center".

Therefore, the scaling has to be done relative to the maximum height of the word. All coordinate bunches processed prior to the point of maximal height have to be scaled again, which means that the whole preprocessing has to be redone. For the parts of the input processed after this point, the scaling factor determined earlier has to be applied. In this case the preprocessing can be done incrementally.
The following subsection explains how incremental preprocessing is done for the different preprocessing steps. Subsection 3.2.2 then describes our approach for detecting and handling of height changes.

3.2.1 Techniques for the various preprocessing functions

Local Techniques

As shown in section 2.2, most of the preprocessing algorithms operate locally. Only a very limited number of data points out of the temporal vicinity of the point to be processed is involved in the calculation. The fact that not all coordinates from the input sequence are available only matters at the end of a given sequence. During incremental preprocessing the points which the algorithm can not process because of missing corresponding points are stored. As soon as the next coordinate group is available, these points are simply added at the beginning of the new bunch. (see figure 3.4). The algorithms where this approach is possible produce the same results for normal and incremental preprocessing. As

can be concluded from the description in section 2.2 this technique is applicable to resampling (see also next section), smoothing and for the local features: y-position, curvature, writing direction and pen indication[1]

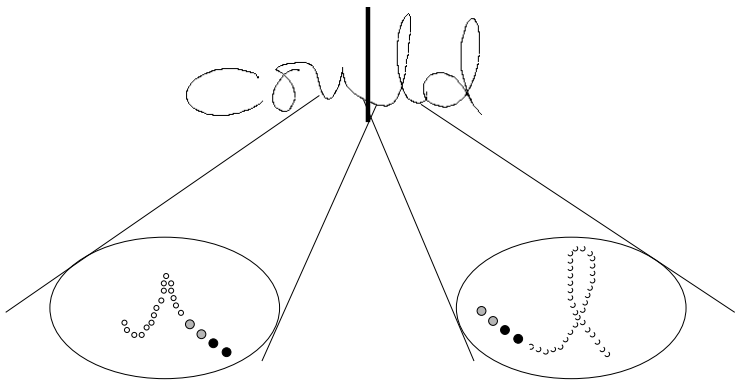

Figure 3.4: Approach for local preprocessing techniques. White dots are already processed, black dots could not be processed at the end of the first bunch and are therefore copied to the beginning of the second bunch. The grey dots are auxiliary points needed for processing the black dots. The grey and black dots have been enlarged for better understanding.

Context Bitmaps

In contrast to the algorithms described in the last section, context bitmaps are local in space, but global in time. They are especially designed to capture temporal long range dependencies (see figure 2.2.2). It is obviously impossible to include future coordinate points into the calculation of the bitmaps. Therefore, only the data points of the current coordinate bunch and the points collected so far can be considered. As can be seen in figure 3.5, the information about the existence of points in region *2* is not accessible from region *1* at the time when the points in that region are processed. For the processing of region *2* on the other hand, the points of region *1* are available.

There is no means to compensate for a loss of information of this kind.

3.2.2 Detection and Handling of word height changes

The easiest way to measure the height of a word is to simply calculate the vertical distance between the points with minimum and maximum values in the y coordinate (see *absolute word height* in figure 3.6). Unfortunately, this

[1] As the removal of the diacritical marks has been left out in the incremental system, the hat feature is not used.

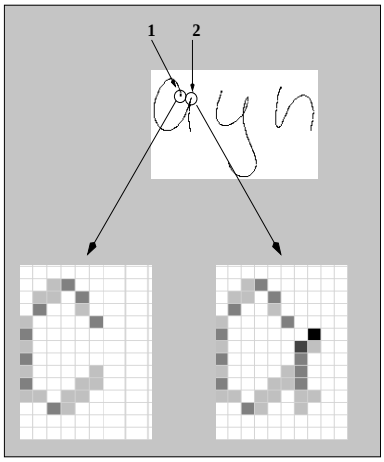

Figure 3.5: Example for incomplete context bitmaps occurring during incremental processing.

approach fails if the baseline of the word deviates a lot from the horizontal. In this case, changes to the absolute word height (in terms of the distance just mentioned) are not necessarily related to true word height changes (measured orthogonal to the baseline). An example for these cases can be found in figure 3.6. Therefore, we made the decision to use the word core height as a criterion to detect word height changes. Besides being invariant against rotation of the word, the core height is also more robust against very small variations in the word height, which can be tolerated for the matters of scaling.

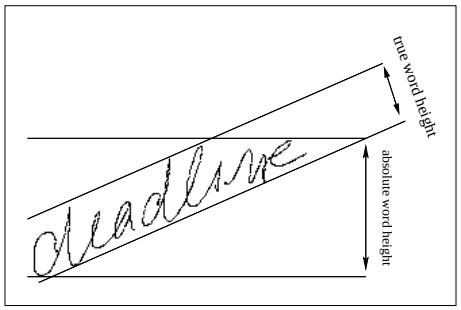

Figure 3.6: Example for a case where absolute word height and true word height differs.

This approach only works if the core height is not constantly changing throughout the word. Figure 3.7 shows that after a certain point the changes to the core height are very small. This follows our expectations, as the true word

height is not changing any more after the highest and the lowest points of the word are reached.

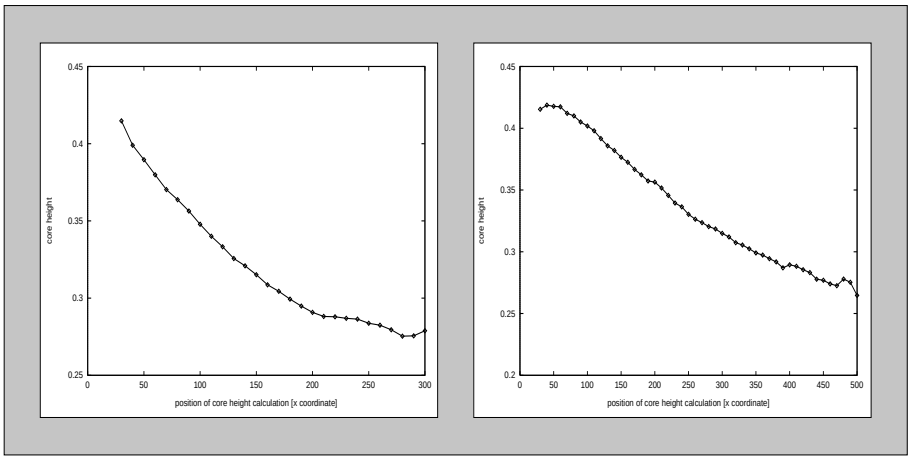

Figure 3.7: Graphs showing the core height for words starting with capital letters (left) and small letters (right) respectively.

The approach taken in **NPen**$^{++}$ can be described as follows (see also figure 3.2). The baselines and the core height are calculated for the first coordinate bunch. Every further group of coordinates is added to the points collected so far and the core height of the whole sequence is determined. If the difference between this core height and the previously calculated core height exceeds a certain threshold, the whole sequence together is preprocessed again. If this is not the case, the previously fixed core height and baselines are used for the preprocessing of the new coordinate bunch (for resampling and baseline normalization).

The threshold used to decide on whether a new full preprocessing of the coordinates collected so far is necessary or not is very crucial. If it is assigned to a low value the number of full preprocessings performed while processing a word overweight the number of incremental preprocessings. For a high threshold value the opposite is true. The effect of this threshold on the overall system performance is analyzed in section 4.2.

3.3 Incremental Recognition

The changes to the recognition engine are a lot less extensive than the changes to the preprocessing. In order to perform incremental recognition, the neural network had to be enhanced by the ability to add frames to the various layers. In each forward pass of the net, the time windows are shifted to the very end

of the available input and the last feature vectors and activations are stored. Then the new feature frames are just appended to the stored vectors and the windows are shifted out of the position where they stopped at the end of the last pass. Figure 3.8 depicts this strategy. The neural net was trained with normally preprocessed training data (see 4.1 for a description of the data used for training).

The search component had to be enabled to find the single best path out of the active paths at the end of the last search and to issue the characters along this path. The fact that the search is done along a dictionary tree can be used to perform *word completion*. If the tree does not branch off after the node with the highest score at the end of the search run, the whole word is emitted and not just the beginning. As can be seen in chapter 4 in a considerable amount of cases, this technique saves the user from writing the whole word[2].

[2]The work described in this section was predominantly done by Stefan Manke.

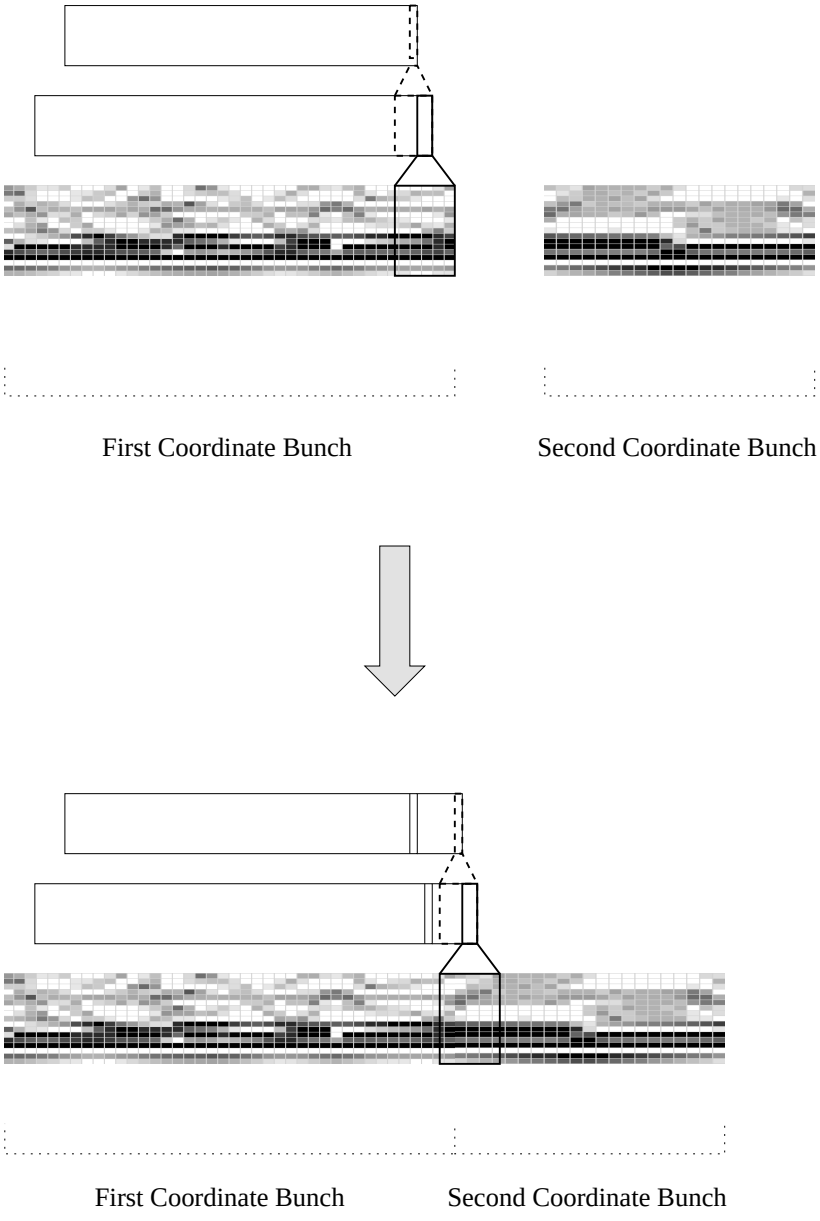

Figure 3.8: Incremental version of the TDNN.

Chapter 4

Evaluation of the System

This chapter describes the various tests conducted with the run-on system. The exact results of the presented tests can be found in appendix A.

4.1 Setup for the Experiments

4.1.1 Training of the Neural Net

The neural net we used for the experiments was trained on approximately 18300 samples written by 191 different writers. The writers were only provided with minimal instructions. The training set contains all types of writing styles, namely cursive, hand-printed and mixtures of both. As stated earlier, the neural net was trained on normally preprocessed data.

4.1.2 Test Environment

All tests performed with the run-on system use the same test set. It consists of 2345 words written by 45 different writers. The set was build from test data with which the normal **NPen**$^{++}$system is evaluated. None of the writers in the test set contributed to the training data. The tests were done using a $50,000$ word dictionary with lower and upper case letters. It was selected randomly from the ARPA Wall Street Journal Task (WSJ), which was originally defined for speech recognition evaluations. The test data consists of coordinate sequences of whole words. In order to test the incremental system the coordinate sequence is chopped into bunches following the strategy described in section 3.1. The parameters controlling this grouping and their optimization is described in section 4.3.

4.2 Recognition Performance of the Run-On System

As described in section 3.2.2, the value of the core height threshold is very important, since it determines the ratio between the number of full preprocessings and the number of incremental preprocessings. This can be seen in figure 4.1.

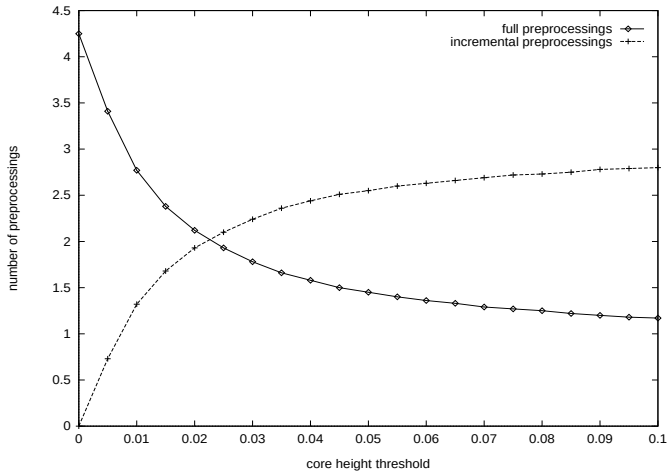

Figure 4.1: Number of full and incremental preprocessings plotted against the word core height threshold. Refer to table A.1 for exact values.

If the threshold value is raised up 0.1 the number of full preprocessings approaches 1. This is the lower boundary because the first coordinate bunch is always fully preprocessed. The core height threshold also affects the average x position of the last full preprocessing as shown in figure 4.2.

In order to determine the recognition capability of the run-on system, two different kind of tests are conducted. The first test examines the incremental preprocessing, excluding the influence of the incremental recognition. This is done by simply putting together the individually preprocessed bunches to get the feature vector of the whole word. The whole vector is then recognized simply as the feature vector of a conventionally preprocessed word would be recognized. The results of these experiments are shown in figure 4.3. As expected the recognition rate drops with raising values of the core height threshold.

In the second set of experiments, the effects of the incremental recognition are included. Every coordinate bunch determined by the splitting routine is given to the recognizer and processed according to the preprocessing mode (full or incremental recognition). The recognition rates achieved at the different points of the input sequence are shown in figures 4.4 and 4.5 (again depending on the core height threshold).

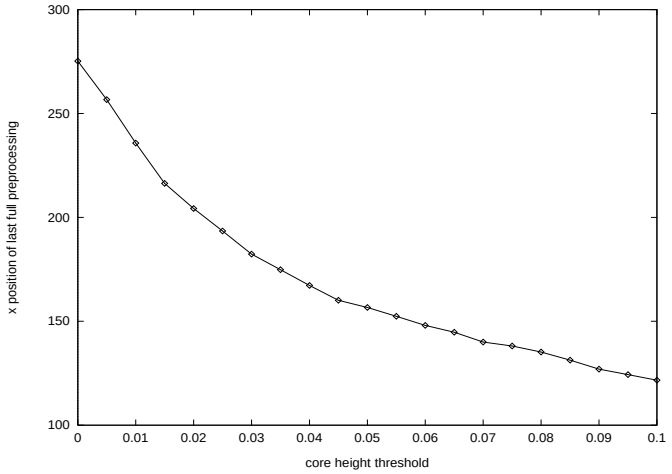

Figure 4.2: x position of the last full preprocessing plotted against the word core height threshold. The average length of the words in the test set is 270.5. Refer to table A.2 for exact values.

For the first set of graphs only the words which are correctly recognized when processed like in the first experiment were taken into account. It might be the case that the recognizer gets on the wrong path towards the end of the word after following the right character sequence before, but usually one would expect the recognizer to be off all the way through if it recognizes the wrong word while processing the whole feature vector. This can also be concluded from the second set of graphs where all words are considered for the recognition rates. There the accuracies are significantly lower. The average length of the sections between the points where the recognition rates are measured is 60 for both sets of graphs.

The shape of the graphs in figure 4.4 are very similar to each other. The recognition rate for the first recognizer run is significantly higher than the rate for the second one, due to the fact that the first run is always done with full preprocessing. After this first break the rate rises because more and more information is available. The maximal accuracy depends on the core height threshold. It falls with raising values of that parameter (as expected). For the graphs in figure 4.5 the situation is not as homogeneous as it was just described. The effects sketched for the first set of graphs get washed out by the high number of wrong recognition results incorporated in the second set of graphs.

During these recognition experiments the number of word completions was also determined. For all tested systems the completion succeeded for approximately 36% of the correctly recognized words. On average, the correct result was found after 74% of the word was written. The word completion rate can be expected to be even higher if the dictionary size is smaller.

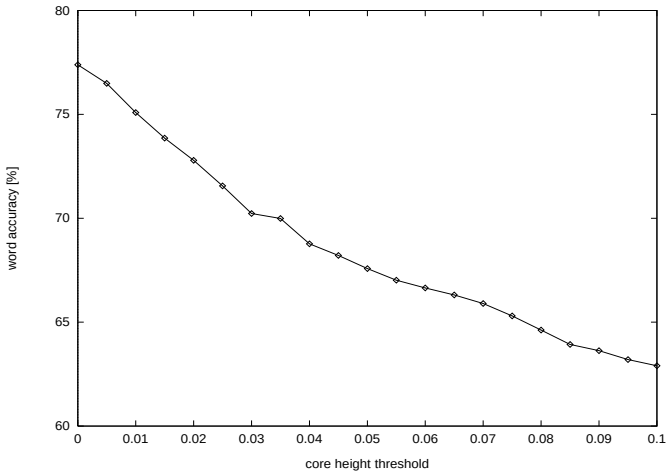

Figure 4.3: Word accuracies of the incremental system as they are achieved at different values of the core height threshold. See table A.3 for exact values.

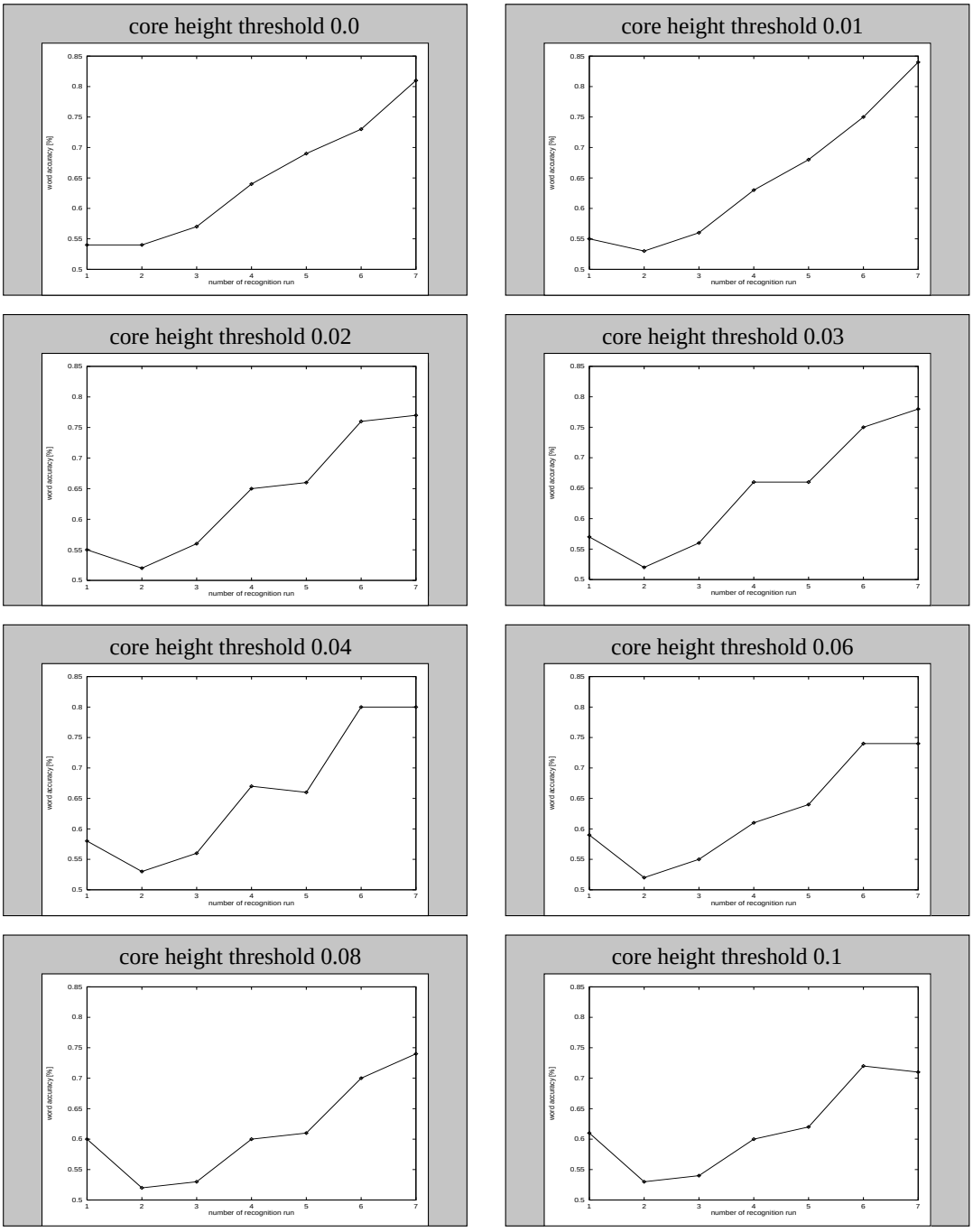

Figure 4.4: Overview of the intermediate recognition results for different core height thresholds if only the correctly recognized words are taken into account. The exact values can be found in table A.4.

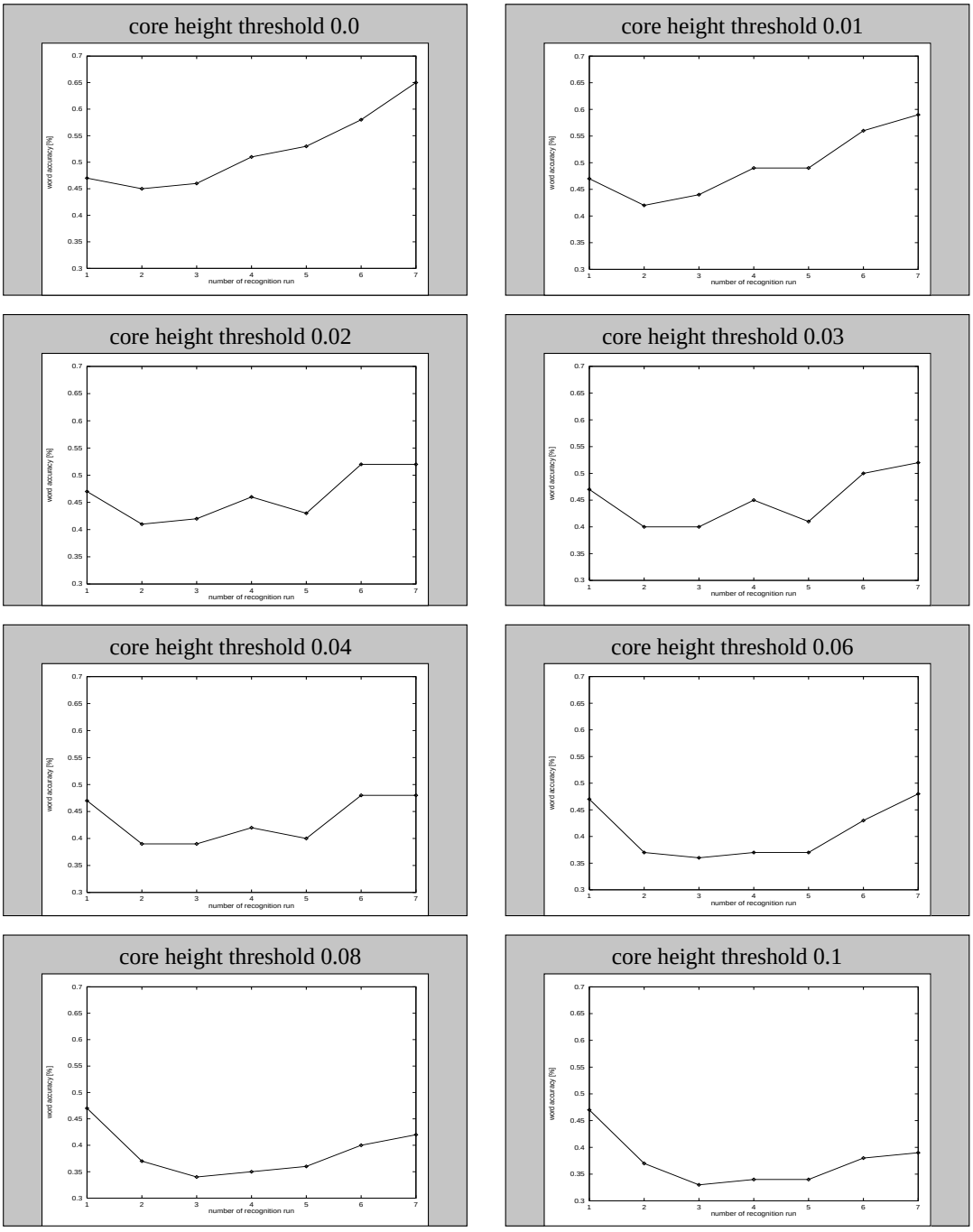

Figure 4.5: Overview of the intermediate recognition results for different core height thresholds if all words are taken into account. The exact values can be found in A.5.

4.3 System selection and parameter optimization

As shown in the last section, the task of choosing one system, which means choosing one core height threshold, involves optimization of at least two conflicting values, namely:

- recognition accuracy
- ratio of full / incremental preprocessings

As a compromise between a system with high recognition accuracy and a system with a high number of incremental preprocessings we selected the system with a 0.03 core height threshold. This recognizer combines a recognition rate of 70.23% with a reasonable ratio of full and incremental preprocessings (1.38 versus 2.25).
For this setup, we optimized the parameters controlling the grouping of the coordinates. During the previously described tests we used default values. The parameters are:

- minimal length of the first coordinate bunch
- minimal length of any further sequence
- number of minima and maxima per sequence

In order to keep the computational costs within reasonable limits, we assume these parameters to be independent. Therefore, the following graphs are only supposed to give an impression on how the parameters are related to word accuracy and preprocessing ratio. Figures 4.6 , 4.7 and 4.8 show the whole word recognition accuracies and the ratio between the number of full and incremental preprocessings depending on the said parameters. This ratio should be low since one of the aims is to have a high number of incremental preprocessings (compared to the number of full preprocessing). Following the optimization goals, we selected the system using:

- minimal length of the first coordinate bunch: 70
- minimal length of further sequences: 30
- number of extrema per sequence: 2

This system reaches a word accuracy of 70.6% while performing 1.42 full and 2.35 incremental preprocessing on average.

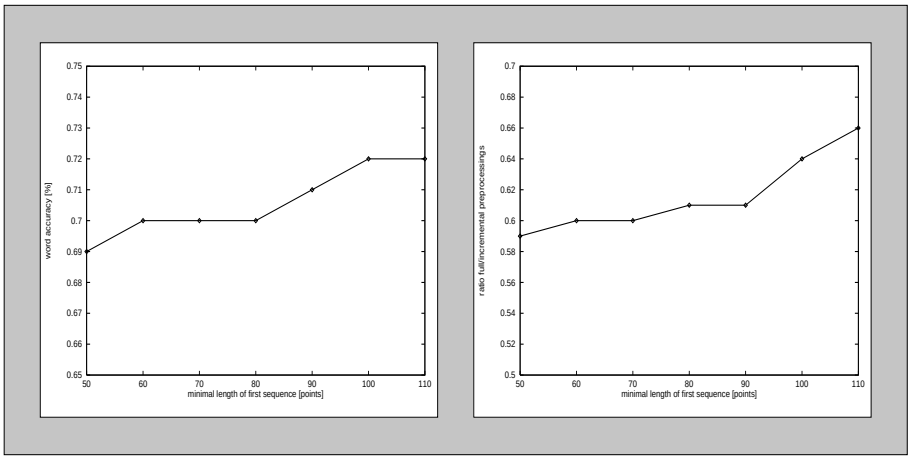

Figure 4.6: Graphs showing the word accuracy (left) and the ratio between the number of full and incremental preprocessings (right) depending on the minimal length of the first coordinate bunch.

4.4 Timings

In order to determine the speed-up gained by using incremental preprocessing and recognition instead of full preprocessing and full recognition, we measured the time the system spent in those modules. We used the recognizer described in the last section and determined how long it takes to process a word using full preprocessing and full recognition for all coordinate bunches versus applying incremental preprocessing and incremental recognition whenever possible. The incremental system only needs 54.1% of the time of the full system (averaged over 1560 words). Test runs with an interactive application also showed, that this speed-up reaches the user-level.

4.5 Discussion

The results given in this chapter show that it is possible to perform run-on recognition with **NPen**$^{++}$. The system we presented achieves a recognition accuracy of 70.2% on a 50k dictionary. Compared to a system with full preprocessing and full recognition this result constitutes a loss of 7.2% in word accuracy. Considering the limited information the incremental system gets while processing a word and the fact that the neural network was not trained on incrementally preprocessed data, this result is very encouraging.

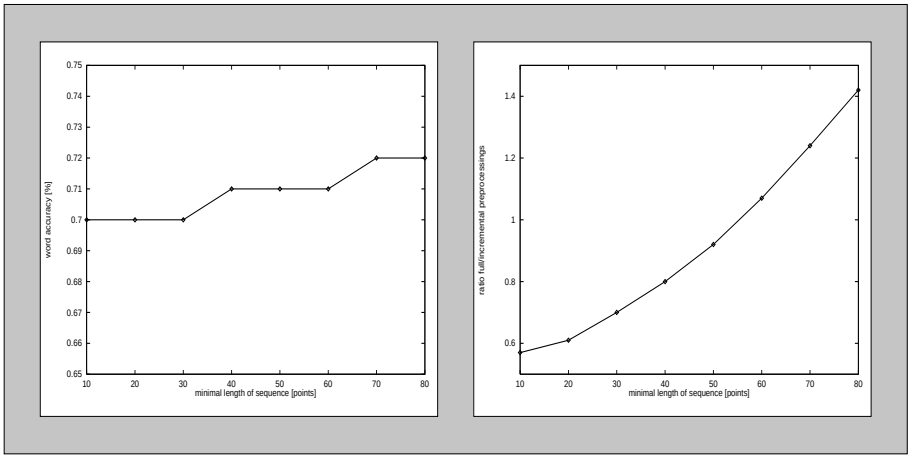

Figure 4.7: Graphs showing the word accuracy (left) and the ratio between the number of full and incremental preprocessings (right) depending on the minimal length of all further coordinate bunches.

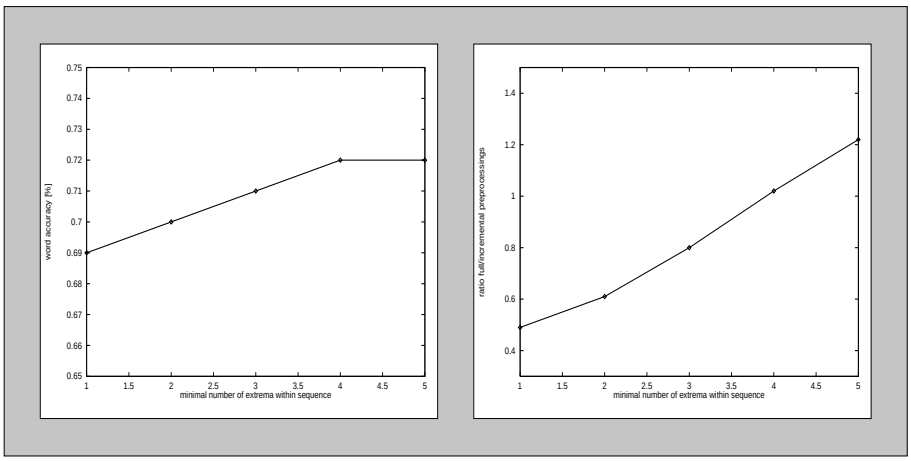

Figure 4.8: Graphs showing the word accuracy (left) and the ratio between the number of full and incremental preprocessings (right) depending on the minimal number of extrema for all further coordinate bunches.

Chapter 5

Summary

5.1 Summary

We showed that it is possible to do run-on recognition with the **NPen**$^{++}$ on-line handwriting recognition system. We changed the system architecture and all parts (preprocessing, recognition, search) in order to enable them to perform incremental processing. The system we built achieved a recognition accuracy of 70.2% on a 50k dictionary. For 36% of the correctly recognized words the result was already found before all coordinate points were passed to the system (using word completion). The run-on system showed a significant speed up compared to a system performing full preprocessing and full recognition on the coordinate bunches.

5.2 Future Work

The major aim for the future is the improvement of the recognition accuracy. We achieved the results presented in this report by using a neural network recognizer for data which deviates significantly from the data the net was trained on. This can also be concluded from the fact that the recognition accuracy depends on the core height threshold. This threshold basically controls the amount of deviation the incrementally preprocessed data has from the "optimal" data, which is the normally preprocessed data. Therefore, the next step will be to train the recognizer on incrementally preprocessed data. We especially hope that the strong dependency on the core height threshold can be removed or at least weakened. We also plan to integrate run-on recognition into the continuous version of **NPen**$^{++}$.

Appendix A

Detailed Test Results

core height threshold	Nbr full preproc.
0.00	1.01
0.005	1.47
0.01	1.58
0.015	1.55
0.02	1.49
0.025	1.42
0.03	1.38
0.035	1.33
0.04	1.29
0.045	1.24
0.05	1.22
0.055	1.20
0.06	1.18
0.065	1.15
0.07	1.14
0.075	1.12
0.08	1.11
0.085	1.09
0.09	1.08
0.095	1.07
0.1	1.06

core height threshold	Nbr incr. preproc.
0.00	0.0
0.005	0.74
0.01	1.32
0.015	1.68
0.02	1.93
0.025	2.11
0.03	2.25
0.035	2.36
0.04	2.44
0.045	2.52
0.05	2.55
0.055	2.60
0.06	2.64
0.065	2.66
0.07	2.70
0.075	2.72
0.08	2.74
0.085	2.76
0.09	2.78
0.095	2.79
0.1	2.81

Table A.1: Number of full (left) and incremental (right) preprocessings depending on the core height threshold.

	Position of last full preprocessing
core height threshold	
0.00	275.23
0.005	256.69
0.01	235.76
0.015	216.37
0.02	204.28
0.025	193.44
0.03	182.34
0.035	174.82
0.04	167.27
0.045	160.11
0.05	156.66
0.055	152.39
0.06	148.05
0.065	144.74
0.07	140.01
0.075	138.13
0.08	135.21
0.085	131.30
0.09	127.00
0.095	124.32
0.1	121.64

Table A.2: Point of last full preprocessing depending on core height threshold.

	Word accuracy
core height threshold	
0.00	77.39
0.005	76.49
0.01	75.09
0.015	73.86
0.02	72.79
0.025	71.56
0.03	70.23
0.035	69.99
0.04	68.77
0.045	68.21
0.05	67.58
0.055	67.02
0.06	66.65
0.065	66.31
0.07	65.90
0.075	65.30
0.08	64.62
0.085	63.93
0.09	63.63
0.095	63.20
0.1	62.90

Table A.3: Word accuracy depending on core height threshold.

	Number of recognition run	Word accuracy
core height 0.0	1	0.54
	2	0.54
	3	0.57
	4	0.64
	5	0.69
	6	0.73
	7	0.81
core height 0.01	1	0.55
	2	0.53
	3	0.56
	4	0.63
	5	0.68
	6	0.75
	7	0.84
core height 0.02	1	0.55
	2	0.52
	3	0.56
	4	0.65
	5	0.66
	6	0.76
	7	0.77
core height 0.03	1	0.57
	2	0.52
	3	0.56
	4	0.66
	5	0.66
	6	0.75
	7	0.78
core height 0.04	1	0.58
	2	0.53
	3	0.56
	4	0.67
	5	0.66
	6	0.80
	7	0.80
core height 0.05	1	0.59
	2	0.52
	3	0.56
	4	0.63
	5	0.66
	6	0.76
	7	0.73
core height 0.06	1	0.59
	2	0.52
	3	0.55
	4	0.61
	5	0.64
	6	0.74
	7	0.74
core height 0.07	1	0.60
	2	0.52
	3	0.54
	4	0.59
	5	0.62
	6	0.71
	7	0.72
core height 0.08	1	0.60
	2	0.52
	3	0.53
	4	0.60
	5	0.61
	6	0.70
	7	0.74
core height 0.09	1	0.61
	2	0.52
	3	0.54
	4	0.61
	5	0.62
	6	0.71
	7	0.74
core height 0.1	1	0.61
	2	0.53
	3	0.54
	4	0.60
	5	0.62
	6	0.72
	7	0.71

Table A.4: Intermediate recognition results for different core height thresholds if only the correctly recognized words are taken into account.

	Number of recognition run	Word accuracy
core height 0.0	1	0.47
	2	0.45
	3	0.46
	4	0.51
	5	0.53
	6	0.58
	7	0.65
core height 0.01	1	0.47
	2	0.42
	3	0.44
	4	0.49
	5	0.49
	6	0.56
	7	0.59
core height 0.02	1	0.47
	2	0.41
	3	0.42
	4	0.46
	5	0.43
	6	0.52
	7	0.52
core height 0.03	1	0.47
	2	0.40
	3	0.40
	4	0.45
	5	0.41
	6	0.50
	7	0.52
core height 0.04	1	0.47
	2	0.39
	3	0.39
	4	0.42
	5	0.40
	6	0.48
	7	0.48
core height 0.05	1	0.47
	2	0.38
	3	0.38
	4	0.39
	5	0.45
	6	0.47
	7	0.49
core height 0.06	1	0.47
	2	0.37
	3	0.36
	4	0.37
	5	0.37
	6	0.43
	7	0.48
core height 0.07	1	0.47
	2	0.37
	3	0.35
	4	0.35
	5	0.36
	6	0.41
	7	0.47
core height 0.08	1	0.47
	2	0.37
	3	0.34
	4	0.35
	5	0.36
	6	0.40
	7	0.42
core height 0.09	1	0.47
	2	0.37
	3	0.34
	4	0.35
	5	0.35
	6	0.39
	7	0.40
core height 0.1	1	0.47
	2	0.37
	3	0.33
	4	0.34
	5	0.34
	6	0.38
	7	0.39

Table A.5: Intermediate recognition results for different core height thresholds if all words are taken into account.

List of Figures

List of Tables

Bibliography

[BHMW93] C. Bregler, H. Hild, S. Manke, and A. Waibel. Improving connected letter recognition by lipreading. In *Proceedings of the ICASSP-93*, Minneapolis, April 1993.

[BL94] Y. Bengio and Y. LeCun. Word normalization for on-line handwritten word recognition. In *Proceedings of the ICPR-94*, Jerusalem, October 1994.

[GAL+91] I. Guyon, P. Albrecht, Y. LeCun, W. Denker, and W. Hubbard. Design of a neural network character recognizer for a touch terminal. *Pattern Recognition*, 24(2), 1991.

[HW92] P. Haffner and A. Waibel. Multi-state time delay neural network for continous speech recognition. In *Advances in Neural Information Processing Systems (NIPS-4)*. Morgan Kaufman, 1992.

[MB94] S. Manke and U. Bodenhausen. A connectionist recognizer for on-line cursive handwriting recognition. In *Proceedings of the International Conference on Acoustic, Speech and Signal Processing*, 1994.

[MBPV93] P. Morasso, L. Barberis, S. Pagliano, and D. Vergano. Recognition experiments of cursive dynamic handwriting with self-organizing networks. *Pattern Recognition*, 26, 1993.

[MFW94] S. Manke, M. Finke, and A. Waibel. Combining bitmaps with dynamic writing information for on-line handwriting recognition. In *Proceedings of the International Conference on Pattern Recognition*, 1994.

[MFW95a] S. Manke, M. Finke, and A. Waibel. npen++: A writer independent, large vocabulary on-line cursive handwriting recognition system. In *Proceedings of then International Conference on Document Analysis and Recognition*. IEEE Computer Society, 1995.

[MFW95b] S. Manke, M. Finke, and A. Waibel. The use of dynamic writing information in a connectionistic on-line cursive handwriting recognition system. In *Advances in Neural Information Processing*, number 7. MIT Press, Cambridge (MA), 1995.

[MFW96] S. Manke, M. Finke, and A. Waibel. A fast search technique for large vocabulary on-line handwriting recognition. In *International Workshop on Frontiers in Handwriting Recognition*, Colchester, England, 1996.

[OWM92] K. Odaka, T. Wakahara, and H. Murase. On-line handwriting recognition. *Proceedings of the IEEE*, 80(7), 1992.

[Rab89] L.R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proc. IEEE*, 77(2):257–286, February 1989.

[Sch93] L. Schomaker. Using stroke- or character-based self-organizing maps in the recognition of on-line, connected cursive script. *Pattern Recognition*, 26, 1993.

[WHH+89] A. Waibel, T. Hanazawa, G. Hinton, K. Shinao, and K. Lang. Phoneme recognition using time-delay neural networks. *IEEE Transactions on Acoustics, Speech and Signal Processing*, March 1989.

[YMS92] K. Yamamoto, S. Mori, and C. Y. Suen. Historical review of ocr research and development. *Proceedings of the IEEE*, 80(7), 1992.