Multi-Language Online Handwriting Recognition

Daniel Keysers, Thomas Deselaers, Henry A. Rowley, Li-Lun Wang, Victor Carbune

Abstract—We describe Google's online handwriting recognition system that currently supports 22 scripts and 97 languages. The system's focus is on fast, high-accuracy text entry for mobile, touch-enabled devices. We use a combination of state-of-the-art components and combine them with novel additions in a flexible framework. This architecture allows us to easily transfer improvements between languages and scripts. This made it possible to build recognizers for languages that, to the best of our knowledge, are not handled by any other online handwriting recognition system. The approach also enabled us to use the same architecture both on very powerful machines for recognition in the cloud as well as on mobile devices with more limited computational power by changing some of the settings of the system. In this paper we give a general overview of the system architecture and the novel components, such as unified time- and position-based input interpretation, trainable segmentation, minimum-error rate training for feature combination, and a cascade of pruning strategies. We present experimental results for different setups. The system is currently publicly available in several Google products, for example in Google Translate and as an input method for Android devices.

Index Terms—online handwriting recognition, handwriting recognition

1 Introduction

In this paper we discuss *online* handwriting recognition. The goal is to output the best textual (Unicode) interpretation of a given sequence of handwritten strokes, where a stroke is a sequence of points (x, y, t) with position (x, y) and a timestamp t. We will use the term *ink* here to denote such an input, where one of the point sequences is called a *stroke*. Table 1 lists the terminology used in this paper.

Fig. 1 illustrates example inputs to our online handwriting recognition system in different languages and scripts. The left column shows examples in English with different writing styles and sampling rates. The center column shows examples from four different languages that share a lot of common structure with English: German, Russian, Vietnamese, and Greek. The right column shows scripts that are significantly different from English in various ways: Chinese and Japanese have a much larger character set, as does Korean. Hindi writing often contains a connecting 'Shirorekha' line, and Hindi (Devanagari) characters can form larger structures (grapheme clusters) which influence the written shape of the components. Arabic is written right-to-left and characters may change shape depending on the position within a word. Heart shapes (which are a commonly received input across many different languages) are a representative of 'general Unicode symbols' that we also recognize. Observing these inputs, we concur with Nagy [1] that we "*are often reminded that English is blessed with one of the simplest scripts in the world*".

Research in online handwriting recognition seems to have peaked in the late 1990s, but there are current reasons for a renewed interest, including:

- A strong growth in the usage of mobile computing devices, many of which are equipped with touch screens that allow for easy recording of online handwriting input. This is emphasized by the availability of numerous smart phone devices that ship with a stylus.

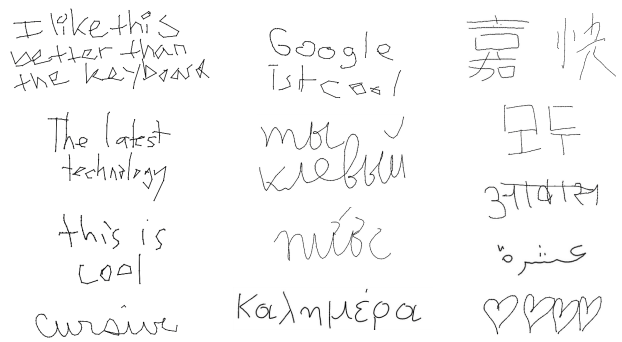


Fig. 1. Example inputs for online handwriting recognition in different languages. See text for details.

- The desire to support more scripts, some of which are particularly difficult to type on a soft keyboard, and for some of which no standard keyboard layout is commonly adopted, (e.g. Khmer).
- Advances in the underlying technology in areas like speech recognition, deep learning, machine translation, and optical character recognition, which can be applied to the task of online handwriting recognition.

In this paper we describe a system that was built to recognize online handwriting for text entry across many languages. Here, the input to the system is an ink along with the language it should be interpreted in[1]. Our primary focus is on mobile devices using touch screens, but the system is also used for other types of input. We build on results from related disciplines and extend existing techniques in various directions. We aim to use machine learning as much as possible but use heuristics where appropriate. In this paper we describe the complete recognition setup, also discuss novel components, and describe how we built a unified system to recognize 97 languages in 22 different scripts. We also describe the approaches used to address the language- and script-specific problems.

Daniel Keysers, Thomas Deselaers, and Victor Carbune are with Google Switzerland, Zurich, Switzerland. Henry Rowley and Li-Lun Wang are with Google Inc., Mountain View, California
E-mail: {keysers, deselaers, har, llwang, vcarbune}@google.com
Manuscript received Oct 5, 2015;

1. We also support a truely language-agnostic handwriting recognizer system in which the user does not need to specify the input language based on the system described here. However, the description of this system is beyond the scope of this paper.

One of our design principles was to build a modular training and recognition system, in which individual components can be adapted to the specific languages. Also, we try to share as much data (and code) between recognizers for various languages as possible. While this approach may have been taken before, here we discuss in detail which of the components are shared between languages and which differ in what way.

While the first commercial online handwriting recognition systems were already available in the 1980s, their success and capabilities were limited. We believe that online handwriting recognition is still not a solved problem and would like to highlight some of the challenges briefly:

- There is a strong variability in writing style between different groups of people and between individuals, for example preference for printed style vs. cursive writing or the age and location where writing was taught for the same language.
- There is a strong variability even for the same person, depending on the input speed, sloppiness, or context in which the text is input, e.g. while walking and on a mobile phone.
- Many ambiguities are inherent in the input, e.g. a small stroke could indicate an apostrophe or could just be accidental 'noise'. Also, many characters are virtually indistinguishable without context. e.g. a single round stroke could be any of the Unicode characters shown in Fig. 2. Similar ambiguities exist for other basic shapes such as horizontal or vertical strokes.
- For scripts with thousands of characters, machine learning is still a challenge, as is the gathering of proper training and evaluation data and the coverage of all common input symbols.
- Handwriting input often has a non-monotonic relationship to the output which is caused by delayed strokes (e.g. t-strokes and i-dots in English) or corrections that are entered later (e.g a forgotten letter is inserted into a word). This is in contrast to otherwise similar tasks like speech recognition, where the speaker can not 'go back in time' to correct input except by using explicit utterances, and OCR (or offline handwriting recognition), which does not use time information.

2 Related Work

Online handwriting recognition is a topic that has been studied in much detail in the research literature in the past and extensive reviews exist [2], [3]. More recently, [4] also summarized the current state of the art.

Kim et al. [4] state that "techniques are language and script-dependent". We believe that this is only partially true, but many of the underlying approaches can and should be shared in a multi-language recognition system.

In the literature, there are two main approaches to online handwriting recognition:

1) Over-segment and classify (or segment-and-decode): this approach is used in the Newton [5] and the TabletPC [6], for example.
2) Time-sequence interpretation: this approach includes hidden Markov models (HMMs) [2], time-delay neural networks (TDNN) [7], and recurrent neural networks (RNN) [8], of which long-short-term-memory networks (LSTM) [9] are a specific case currently receiving a lot of attention in machine learning.

Combinations of these approaches are possible, e.g. [6] uses a segment-and-decode approach at the global level and a TDNN for classification of the segments.

A variety of preprocessing steps are discussed in the literature [4], [10], of which we only employ very few (Sec. 4). Common steps include normalization of size, density, rotation (slope), and slant as well as resampling or smoothing of the strokes.

While not absolutely necessary for some approaches (e.g. when using LSTMs), traditionally features are extracted from the ink to represent the input in a way that lends itself to recognition. Two types of features are commonly used: (1) point-wise features that are computed for each point in the ink are very well suited to be used in the time-sequence interpretation approach and (2) global features that are computed for larger chunks of ink. The latter are commonly applied on the segments in the segment-and-decode approach. The point-wise features can easily be used as global features by accumulating them over a segment, for instance in a histogram.

Descriptions of many commonly used features can be found e.g. in [7], [9], [10], [11], [12], including normalized coordinates, curvature, aspect ratio, curliness, linearity, inflection points, stroke crossings, velocity, ascenders and descenders, directional features, moments, number of strokes, rendered bitmaps, and orientation maps. The feature set we employ is described in (Sec. 5.3.1).

Many research papers are restricted to single character entry, e.g. [13], and use the well known UNIPEN data set [14]. We briefly present a basic experiment on that data in Sec. 10.1. For single-character classification, almost any known classification technique has been evaluated in the literature, including nearest neighbor approaches, clustering, HMMs, support vector machines, and neural networks of various flavors.

The common approach of combining multiple classifier systems has been evaluated in the context of online handwriting recognition in the past as well, e.g. [15]. We use classifier combination at the character recognizer level for our Japanese and Chinese recognition systems.

Further, many approaches focus specifically on one language or script, for example Arabic [16], Tamil [17], Chinese [18], or Japanese [10].

Language modeling for online handwriting recognition bears many similarities with the use of language information in OCR and speech recognition, which often employ statistical n-gram models on the character- or word-level [19], [20]. Often in the online handwriting recognition literature, a restriction to a fixed dictionary is imposed, e.g. [21]. We believe that the capability to recognize arbitrary letter sequences, while hurting on some ambiguous inputs, can be viewed as an advantage over some other text input methods and we therefore use a combination of character-based n-grams and probabilistic state machines (Sec. 6.1).

The literature also suggests several ways of handling delayed strokes (in scripts that make use of them) [4]:

- assigning to a character hypothesis based on a heuristic or fixed position based reordering
- removing delayed strokes and performing recognition without them

0	U+0030	DIGIT ZERO	○	U+0D20	MALAYALAM LETTER TTHA
O	U+004F	LATIN CAPITAL LETTER O	๐	U+0E50	THAI DIGIT ZERO
o	U+006F	LATIN SMALL LETTER O	◌	U+17F0	KHMER SYMBOL LEK ATTAK SON
°	U+00B0	DEGREE SIGN	○	U+25CB	WHITE CIRCLE
ο	U+03BF	GREEK SMALL LETTER OMICRON	。	U+3002	IDEOGRAPHIC FULL STOP
о	U+043E	CYRILLIC SMALL LETTER O	〇	U+3007	IDEOGRAPHIC NUMBER ZERO
૦	U+0AE6	GUJARATI DIGIT ZERO	ㅇ	U+3147	HANGUL LETTER IEUNG
ଠ	U+0B20	ORIYA LETTER TTHA	０	U+FF10	FULLWIDTH DIGIT ZERO

Fig. 2. Examples of ambiguities: a selection of symbols that look like a circle with their Unicode code-point number and short descriptions.

- explicit training of delayed-strokes-free models (e.g. for 'dot-free i') and capturing nearby delayed strokes
- running two recognitions in parallel (with and without reordering)

We propose a novel integrated approach in which a re-ordered and a non-reordered decoding is done in a single pass with little overhead (Sec. 5.2).

There exist many end-to-end systems. Some of the first commercial handwriting recognition systems were based on unistroke single-character inputs [22] and applications that use similar approaches still exist today. We have already mentioned the systems developed by Apple [5] and Microsoft [6]. There are other commercially developed systems for online handwriting recognition that are worth mentioning, although we are not aware of public descriptions of their technology at a detailed level. The most prominent current technology is probably the system developed by MyScript/Vision Objects. Apple's iOS products ship with a handwriting recognition system for Chinese. For the specific case of Chinese handwriting recognition, there exist several additional systems (e.g. for the Android operating system) by various parties. Some websites on the world wide web also show a handwriting recognition input field, for example `http://hk.yahoo.com` allows their users to enter single characters that can then be used for searching. There are also a variety of open source systems available, but an in-depth discussion of these is out of the scope of this document.

3 System Architecture

Our system follows the segment-and-decode approach: it first over-segments the ink into character hypotheses, then classifies the character hypotheses as characters, and finally recognizes the entire ink in a lattice decoding phase.

The main steps are the following:

1) preprocessing (Sec. 4)
2) segmentation and search lattice creation (Sec. 5)
3) generation & scoring of character hypotheses (Sec. 5.3)
4) best path search in the resulting lattice using additional knowledge sources (Sec. 6)

4 Preprocessing

The preprocessing we apply to the ink is fairly simple and consists of only two steps.

1) Resampling: We apply a length-based resampling along the strokes of the ink with a resampling rate of 5 percent of the height of the writing area. This means that a fully vertical stroke across the entire writing area will be represented using roughly 20 points. The writing area size can be set by the application and is considered part of the input. If the writing area is not set, or if the input is very small or very large with respect to the writing guide, we use a heuristic to adjust the writing area.
2) Slope correction: To handle inks that are not written along a horizontal base line, we project the input points onto rotated y-axes between -15 and $+15$ degrees and score them using the variance of the projection with a bias towards horizontal input. We rotate the input according to the best scoring slope.

TABLE 1
Terminology used in this paper.

Point	A location in x and y coordinates plus a timestamp t.
Stroke	A sequence of points where a pen (or finger, stylus, ...) consecutively touched the writing surface.
Ink	A user input, a sequence of one or more strokes.
Segmentation/cut point	A point at which another character may start. These points can be at the beginning of a stroke, but also in the middle of strokes.
Segment	A segment represents the (partial) strokes between two consecutive segmentation points. Segments may consist of one or more complete or partial strokes.
Character hypothesis	A set of one or more segments (not necessarily consecutive).

These preprocessing steps deliberately do not restrict the input to be written on a fixed baseline, with a specific height, or on a single line. This has the advantage that our system is robust to handwriting from a variety of different applications without specific tuning. We chose not to apply other, potentially helpful, preprocessing methods to keep our system as flexible as possible with respect to the types of input it can process. In the course of developing the system we evaluated several other techniques from the literature (e.g. removal of erroneous touch inputs, noise filtering, or slant correction) but did not find them to consistently improve accuracy on the datasets that we evaluated on.

5 Search Lattice Creation

Once the ink has been preprocessed we determine a set of character hypotheses to create a *segmentation lattice*. We first apply a segmenter to find an overcomplete set of potential cut points between characters (Sec. 5.1). With these cut points we create a segmentation lattice by grouping sets of ink segments into character hypotheses (Sec. 5.2), which we then label using the classification of the character hypotheses (Sec. 5.3) and additional feature functions (Sec. 5.4).

5.1 Segmentation

During segmentation the goal is to obtain high recall of all actual character boundaries. High precision, while beneficial for the computational cost and overall accuracy, is not as important because we will drop cut points and merge segments during later stages but we will not be able to recover

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPAMI.2016.2572693, IEEE Transactions on Pattern Analysis and Machine Intelligence

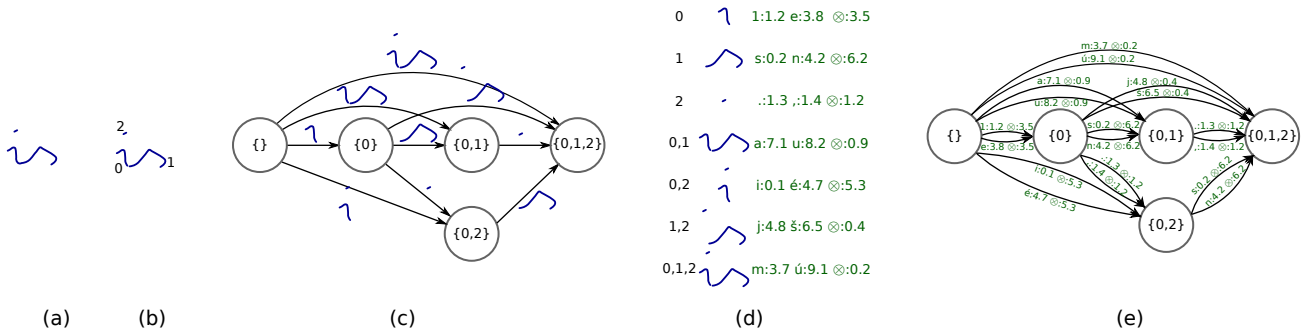


Fig. 3. Lattice creation: on the input ink (a), we determine cut points and create three segments (b). From the segments, we create a segmentation lattice (c) by grouping adjacent segments in character hypotheses. The character hypotheses are classified (d) and the lattice is labeled (e).

missing cut points and thus missing character boundaries will likely affect recognition success in later stages.

In the presented system, segmentation is a two-step process: a heuristic creates a set of potential cut points and then a neural network assigns a score to each cut point. Segmentation points with cut scores under a learned threshold are dropped from further consideration.

The heuristic in the first step is chosen based on the script. E.g. for Latin script, the heuristic proposes points at local minima of the y-coordinate and additionally inserts cut points between two minima if the segment is longer than a threshold (Fig. 3(b)). In contrast, for Chinese script, consecutive characters are usually not connected and thus, the heuristic only proposes to cut between strokes. An overview of our segmentation heuristics and for which script they are used is given in Table 2 and Sec. 9.

In the second step, a neural network is used to compute a score for each cut point. This neural network uses a set of simple geometric features of the point and its surrounding as input. This cut classification network is trained during the normal training procedure (see Sec. 7.2) and is tuned to have the desired balance between precision and recall.

5.2 Creating a Segmentation Lattice

From the resulting segments, we create a segmentation lattice (Fig. 3(c)). The segmentation lattice is a graph in which both edges and nodes correspond to sets of ink segments: The edges correspond to the ink segments which are grouped in a character hypothesis and the nodes correspond to the part of the ink that has been consumed up to the node.

The segmentation lattice has one start node (the empty set of segments) and one target/sink node (the set of all segments). For each node in the lattice, the set of segments represented by it is the union of the segments of the edges along each path to that node from the start node. Equivalently, each node corresponds to the part of the input ink that has been consumed along each path up to this node.

A character hypothesis often consists of more than one segment: character hypotheses are formed by selecting a set of consecutive segments from a sorted list of segments, and are influenced by two parameters:

1) the maximum number of segments that are allowed to form a character hypothesis, and
2) the order in which the segments are sorted to determine succeeding segments.

The maximum number of segments is chosen such that complete characters can be formed from the segments. Making this number smaller will make the recognizer faster, as the resulting lattices are smaller, but making the number too small will have an impact on accuracy and may make it impossible to write some characters.

For the sort-order of the segments we use two strategies or their combination: *time order* which sorts the segments according to their temporal writing order, and *spatial order* which sorts the segments according to their x-coordinates. The time order approach is useful to support overlapping writing, e.g. when a user writes subsequent letters on top of each other in the same location. The spatial order approach is useful to support delayed strokes, e.g. when a user writes cursively and adds t-strokes and i-dots later. The combination of the two approaches is shown in Fig. 3(c), where the segmentation lattice contains paths for both time order (segments 0, 1, 2) and spatial order (segments 0, 2, 1).

For Chinese script, which is not usually written with delayed strokes, using just time order is sufficient. For Latin script and several others, we use both strategies in parallel, because this allows support for both cursive writing with delayed strokes and overlapping writing. For bidirectional languages (such as Arabic and Hebrew) using both orders would make bidirectional text (e.g. mixed Arabic script and numbers) ambiguous during decoding, and because we want to handle delayed strokes, we use spatial ordering. Which orders are used in which script is detailed in Table 2.

The combined set of character hypotheses for the enabled segment orders are then used to create one joint lattice. This is straightforward because nodes and edges are represented as sets of segments of the same input. Combining segment orders in one lattice in this manner allows to hypothesize all orderings jointly in a single recognition pass. Note that this approach introduces no additional cost should the orderings coincide, e.g. if the time order and the spatial order are identical in a piece of Latin-script input.

In informal experiments we observed relative reductions in the error rate of 2% for various languages including Latin-script languages like English, Spanish, and German when switching from spatial order to the combination of spatial and time order. In this comparison the overall search space was kept fixed; improvements were larger when the search effort was allowed to increase.

In the resulting segmentation lattice, there will be character hypotheses that correspond to incomplete characters, combinations of incomplete characters, or multiple characters together. (See Fig. 3(c), where the hypothesis of only segment 0 is an incomplete character, and the hypothesis that combines the segments 0 and 1 corresponds to part of the 'i' and part of the 's'.) To handle these cases, we model

these non-characters explicitly using a garbage class.

5.3 Character Classification

After the segmentation lattice has been created, the character hypotheses are classified with the goal of determining the characters most likely to have been written.

5.3.1 Feature Extraction

To classify the character hypotheses created during the lattice creation into characters (or non-characters), we use a fixed-length, dense feature vector representation of the ink. We distinguish between two general classes of features:

1) Pointwise features are extracted for each point along a stroke. In our approach, the pointwise features are then accumulated into histograms for a complete character hypothesis to arrive at a fixed-length representation.
2) Character-global features are extracted for entire character hypotheses.

The full feature vector is linearly normalized to have a distribution (per-dimension) with mean zero and a range not exceeding the interval [-1, 1] on the training data.

5.3.2 Pointwise Features

As pointwise features, we compute 23 simple geometric features following ideas from the literature [5], [7] including normalized x/y–coordinates, derivatives, curvature, and curliness. Further, we compute local image features following [23] for each point by rendering the ink into a bitmap and using a local neighborhood (scaled to 3×3 pixels) of each point as a feature vector. A similar approach is called 'context map' in [7].

Based on the 23 geometric and 9 bitmap features, we compute an additional $23 \cdot 22 \,/\, 2 = 253$ and $9 \cdot 8 \,/\, 2 = 36$ second-order (product) features, yielding a total of 321 pointwise features. Then we accumulate a 10-bin histogram per feature for each character hypothesis leading to a 3210 dimensional feature vector. The value of 10 bins was determined in informal experiments with varying numbers of bins and showed improvements over the direct use of the values. The histogram bin boundaries are determined on the training data to have equal frequencies per bin on average. We use the square roots of normalized histogram counts as features, following [24].

5.3.3 Character Global Features

In addition to the pointwise features, we compute a set of features on the entire character hypothesis. These help to capture spatial structure, long-range correlation, and character shape.

Bitmap features (3×64-dimensional). For each character hypothesis we compute a feature vector based on an image representation of the ink. We render the ink into a binary 32×32 pixel bitmap and then scale it to a gray scale image of 8×8 pixels. We then also compute Sobel filter responses along x and y directions, which have been used for image-based character classification [23].

Simple Statistics (384-dimensional). For each character hypothesis we compute a set of simple statistical features: a bounding box feature that captures the size of the hypothesis; the mean, variance, and x/y covariance of the points; the number of stroke-crossings and dots; the sum of lengths, widths, heights, histogram of directions, and average curvatures of each stroke; the minimum, maximum, mean, and variance of the curvature along points; an estimate of the height of the baseline of the writing and how many points are above and below the baseline; the number of strokes in the character hypothesis.

Water Reservoir Features (64-dimensional). We compute a water reservoir bitmap feature in four directions based on ideas from [25].

Stroke Direction Features (180-dimensional). We extract stroke direction features similar to those described in [5].

Quantized Stroke Direction Maps (512-dimensional). We compute quantized maps of stroke directions in an 8×8 spatial grid quantized to 8 directions [26].

5.3.4 Classification

The main character classifier we use is an artificial neural network based on the framework described in [27]. The network topology is simple: a single, fully-connected hidden layer of size 1024. The hidden layer uses $\tanh$ activation, and the output layer uses softmax activation. This approach is in many ways similar to the approach taken in the Newton [5]. In informal experiments we did not see accuracy gains from deeper network structures, which may be related to the use of a rich feature set. Note that the use of deeper network structures also tends to increase latency at the time of classification. We train the network using distributed stochastic gradient descent using up to 200 epochs.

For Chinese and Japanese the very large character sets pose a special challenge: some of the characters are written only rarely but are still relevant to the language. When using only the neural network model these characters can be very hard to recognize. Therefore, for Chinese and Japanese we use two character classifiers: the neural network, as described above, and an approximate nearest neighbor classifier [28] which improves accuracy for rare characters. To compute meaningful scores for the nearest neighbor classification we use a modified quadratic discriminant function [29] which we use to compute the scores for the top 50 classes returned by the nearest neighbor classifier. We use these scores as an additional feature function in the lattice alongside the scores returned by the neural network.

5.3.5 Labeling the Lattice

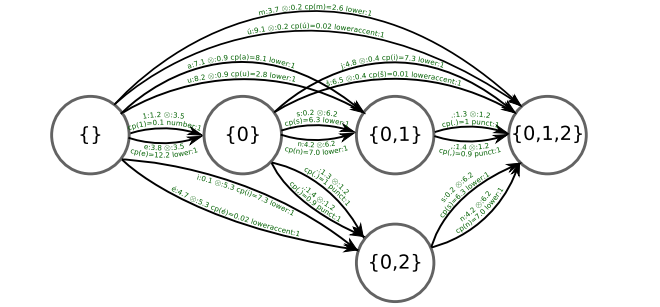
In the next step we create a labeled lattice, which will later be decoded to find the final recognition result.

Starting from the segmentation lattice (Fig. 3(c)), we create a multi-graph by adding edges for each potential label for each character hypothesis (Fig. 3(e)). These edges are assigned the classification result for the character hypotheses (Fig. 3(d)) as a label and the score for that label from the classifier as a cost which is later linearly combined with the other feature functions. The garbage class is treated specially: each edge is assigned an additional garbage feature function with a value corresponding to the classifier score for the garbage class.

For each character hypothesis we consider the top 20 classification results, thus the labeled lattice will have 20 times the edges as the segmentation lattice and the same set of nodes.

5.4 Spaces, Relative Size, and other Feature Functions

Beyond the scores computed from the character classifier we use a variety of additional feature functions (extra scores



Beam 0 steps:

edge label	edge cost	lm cost	total cost	path
	0.0	0.0	0.0	{}

Beam 1 steps:

edge label	edge cost	lm cost	total cost	path
1	-1.2	9.7	8.5	{}-0
i	3.1	7.3	10.4	{}-02
é	0.4	13.5	13.9	{}-02
m	7.1	13.4	20.5	{}-012
u	11.1	9.9	21.0	{}-01
e	13.5	7.9	21.4	{}-0
ú	9.9	12.0	21.9	{}-012
a	15.3	7.0	22.3	{}-01

Beam 2 steps:

edge label	edge cost	lm cost	total cost	path
is	1.3	8.3	19.9	{}-02-012
1.	2.1	9.4	20.0	{}-0-02
m			20.5	{}-012
1,	2.1	10.5	21.1	{}-0-012
1s	1.3	13.5	25.3	{}-0-01
in	6.0	7.9	24.2	{}-02-012
1n	6.0	14.0	28.5	{}-0-01
és	1.3	17.7	32.9	{}-02-012
1j	12.7	14.0	35.2	{}-0-02
lš	7.1	24.0	39.6	{}-0-02
én	6.0	19.9	39.8	{}-02-012

Beam 3 steps:

edge label	edge cost	lm cost	total cost	path
is			19.9	{}-02-012
m			20.5	{}-012
1.s	1.3	36.3	37.6	{}-0-02-012
1,s	1.3	41.2	42.5	{}-0-02-012
1.n	6.0	39.2	45.2	{}-0-02-012
1,n	6.0	40.1	46.1	{}-0-02-012

Fig. 4. Additional features in the lattice (Sec. 5.4) and beam search on this lattice with all weights set to 1.0, except for the garbage weight set to -1. The beam search uses a beam size of 4 which means that in each step it expands the top-4 entries of the beam. The edge costs in the beam correspond to the newly added costs from traversing an edge and the newly added language model cost. The total cost contains the sum of these and the total cost of the path before this last step. The gray entries in the beam are those that are not expanded in the next step.

per character recognition hypothesis) to guide the recognition and to introduce additional prior knowledge. Some of these feature functions resemble what are called 'geometric contexts' in [30]. These additional feature functions can be added to the lattice explicitly or they can be generated implicitly during decoding.

Each edge with a character hypothesis is assigned extra scores based on the character that it corresponds to:

1) A *character prior* feature that captures how well the character was represented in the training data. This feature is useful to allow for balancing the prior which the discriminative character classifier has learned. We use the negative logarithm of the relative frequency of occurrence in the training data as value.
2) A *character class* feature that scores characters based on their meaning within a language, e.g. whether a character is a commonly used character, a numeric symbol, a currency symbol, a punctuation symbol, or an undesired symbol. This feature allows for an easy adaption of a *script-specific* system to a *language-specific* system. For instance, the sets of common characters in the Spanish and French recognizers are slightly different but otherwise these two recognizers share the same character classifier. A specific example is the use of the highly confusable characters LATIN CAPITAL LETTER ETH (U+00D0, Ð) used in Icelandic and the LATIN CAPITAL LETTER D WITH STROKE (U+0110, Đ) used e.g. in Bosnian.
3) A *string length* feature that is useful to allow for learning a weight to balance long vs. short recognitions (similar to the word penalty in speech recognition [31] or the fertility model in machine translation [32].)
4) *Cut features* — one feature for each possible type of cut from the segmenter that precedes the character (e.g. did the cut happen within a stroke or between two separate strokes) as the cut score assigned by the neural network.

Additionally, we use two 'bigram' feature functions that are applied not at the level of single edges but for pairs of successive edges on a decoding path:

1) A *relative size* feature function that looks at pairs of character hypotheses on successive edges on the decoding path. For the relative size information we use a prior about the bounding boxes of successive candidates and compute a score that depends on how well the bounding boxes of the character hypotheses align with their respective priors.
2) A *space insertion* feature function that looks at the horizontal space between successive character hypotheses relative to their size and allows to hypothesize spaces during recognition.

When computational resources are very limited (as is the case on mobile devices) we do not use bigram feature functions. In that case the space feature function can be replaced by a function that looks only at the previous and following ink for a space (in contrast to looking at the previous and next character hypotheses, which depend on the path taken during decoding). Relative size information could be used in a post-processing step but we currently just omit this knowledge source for on-device recognition.

6 Lattice Decoding

Given the labeled lattice (Fig. 4) along with an additional language model, we search for the best path to determine the recognition result.

6.1 Language Models

To model the likelihood of the character sequences that are recognized, we use two types of language models (LMs). The unit at which we determine the probability estimates in the LMs is a character (more precisely, a Unicode codepoint), which is in contrast to the word-based LMs often used in speech recognition or machine translation.

The main LM we apply is a stupid-backoff [33] entropy-pruned 9-gram character LM. Note that we do not claim this to be the optimal choice of language model for online handwriting recognition in general (which would be outside of the scope of this paper), but we believe it is a reasonable choice in the current design of the overall system. The LMs are trained on a large corpus of web-mined text for the specific language [33]. In our cloud recognizers, we use 100M n-grams for most languages. For some languages that get comparatively few requests (e.g. Esperanto) we trade off memory use vs. accuracy and use a smaller model with 10M n-grams. For ondevice recognition we use 5M n-grams for all languages and we compress them into a space-efficient data structure using ideas from [34].

The second LM we use is a word-based probabilistic finite automaton created using the 100,000 most frequent words of a language, determined from a web-mined corpus. In this LM, back-off is handled using an additional feature function for out-of-vocabulary events and word frequencies are used to assign scores to letter sequences.

Note that the main type of request that our system handles consists of very short phrases or even single words or characters as they are typically entered on a mobile device. This makes a longer-range LM less useful than it would be for long sentences or even paragraphs. We plan to

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPAMI.2016.2572693, IEEE Transactions on Pattern Analysis and Machine Intelligence

evaluate more types of LMs including longer-range LMs in the future as we believe that they may be useful for certain types of input.

In some cases, the input to the recognizer may additionally contain context information about the text that was written before the current input. We use this pre-context information (which is restricted to 20 characters of text) as history for the LM scoring. This also allows us to hypothesize and predict a leading space in the recognition without any indication in the actual stroke input. For example, the written sequence 'is' may be best recognized as ' is' with a leading space if it follows the word 'she' (to form a combined string of 'she is'), but as 'is' without leading space for a pre-context of 'th' to yield the combined string 'this'.

The language model scores are not stored in the lattice but are used implicitly during search.

6.2 Search

At this point we have a graph that represents the entire input and by finding the best path from the source node (no ink recognized) to the target node (all ink recognized) we obtain a recognition result for the entire ink. During the search we implicitly expand the lattice to consider the history for the language model and the bigram feature functions described in the previous section.

The search algorithm is an ink-aligned beam search that starts at the start node and proceeds through the lattice in topological order. The partial order is given by set-inclusion on the nodes (which are sets of segments, compare Fig. 3). At the beginning, the set of open search states consists of only the start node with a language model history corresponding to already written preceding text (if any exists). In each node expansion pass, the decoding algorithm considers all edges that leave the current node and expands the states into new states by computing a combined weighted score for the feature functions and the language model. Once all states of the current node are expanded, the set of open states is pruned to only include the states with the best scores and state recombination happens. During this process, the open states may correspond to different sets of segments, but the differences are limited by the maximal number of segments in a character hypothesis and the topological sorting. At the end of the iteration, the open states all correspond to the lattice end node, i.e. the full set of segments has been processed. At that point, the state with the best score corresponds to the determined best recognition result. Alternative recognition results can be obtained from states with worse scores that correspond to alternative decoded strings. Fig. 4 shows an example for the beam search on a graph including the language model scores.

6.3 Pruning strategies

Pruning is a well-known method to speed up tasks like speech recognition, OCR, and many others. Besides search pruning, we apply a variety of different pruning strategies at different stages of recognition to allow for a fluent interaction of users with the recognizer while keeping the accuracy as high as possible. In this tradeoff, there are often different pareto-optimal points that can e.g. be used for on-device recognition vs. server-based recognition, where the latter can use more computational resources. In the following we describe the different pruning strategies in the order which they are applied in our recognition pipeline.

6.3.1 Segmenter Pruning

As described in Sec. 5.1, the aim of the segmenter is to have a high recall. To reduce the size of the segmentation lattice, segmentation points with very low scores from the segmenter are not considered, which directly reduces the size of the segmentation lattice and thus avoids any of the subsequent steps.

6.3.2 Lattice Edge Pruning before Classification

Before we perform (computationally costly) feature extraction and classification on an edge of the lattice, we use a simple (log-)linear classifier to discard some of the edges. The input to this classifier consists of simple features such as the number of segments, the type of cut at start and end, the size of the ink with respect to the input, and the number of dots and intersections. When removing edges, we ensure that we do not disconnect any node from the lattice by leaving at least one incoming and one outgoing edge for each node. We prune up to 20% of the edges at this point, depending on the desired speed/accuracy tradeoff.

6.3.3 Lattice Edge Pruning after Classification

After character hypotheses have been classified by the character model, we use the character scores as additional input to prune edges from the labeled lattice. Here we use the values of all feature functions that do not depend on the search history, which mainly excludes the language model feature functions. When an edge between two nodes has a much higher cost than other edges between the same nodes it is very unlikely that this edge is chosen during the final search unless it has a much better language model probability. We prune up to 90% of the edges at this point, depending on the desired speed/accuracy tradeoff, again, ensuring that all nodes can be traversed by not removing all incoming or all outgoing edges from any node.

6.3.4 Edge Factor Pruning

Next, we limit the number of total edges in the lattice to be at most 100 times the length of the longest path. This is done using an A-star search without LMs. Edges that are traversed by the search are kept until this limit is reached and there are no partially connected nodes. Edges that are not traversed in this process are removed from the lattice.

6.3.5 Beam Search Pruning

When we perform the search through the lattice to identify the most promising hypothesis, we use a conventional beam search approach and limit the number of expanded states (including LM history) to a maximum between 40 and 120.

7 Training

The full system is trained by alternately training the individual system components. Some components are considered as fixed during a training run, e.g. the language models that are derived from separate data sources.
There are three main training components:

- training the character model ('C')
- training the segmentation model ('S')
- training the feature weights ('W')

A typical training run may consist of a sequence of these steps following each other, e.g. C-C-W-C-W-S-W-C-W-S-W.

We will briefly discuss each of these steps here. The number of steps in a typical training run usually varies between 5 and 20. The iterative improvement structure of the training seems reminiscent of the EM-algorithm [35], though we do not claim any convergence or optimality properties.

7.1 Training the Character Models

We determine a forced alignment of each training observation by recognizing it while restricting the decoding to allow only the correct label. From this forced alignment we determine a segmentation of the training data, which gives us a mapping of character hypotheses to recognition labels (characters or grapheme clusters, see Sec. 9).

We then train the character models aiming to classify the force-aligned character hypotheses. In addition we add a 'garbage' (non-character) class $\otimes$ from character hypotheses that were not chosen in the decoding. Since the total amount of potential garbage training data is very large we subsample it by a small integer factor. Nonetheless, it usually is the most common class in the resulting training data.

The very first character model may be trained using either a linear alignment of segments to characters, an alignment proportional to the number of segments observed in the training data using a linear regression, or restricting the training set to only instances of single recognition classes.

7.2 Training the Segmenter

Training the segmenter bears many similarities to character model training. We train the neural network cut classifier using the segmentation points that were selected during the forced alignment as positive samples and all other segmentation points as negative samples.

7.3 Training the Feature Weights

To balance the weight that each knowledge source has in the final decoding process, the individual feature functions that are associated with each edge are weighted (log-)linearly. This is similar to the use of a language model weight in speech recognition or the log-combination of translation and language model weights in machine translation. We use the MERT (minimum error rate training) framework [36] to estimate the weights on a hold-out set separate from training, validation, and test data sets. In MERT weight training, we repeatedly decode the items in the weight-training set with the current weights and construct the decoding lattices, which are combined over several iterations. For the full set of segmentation graphs of all items, the MERT algorithm then improves the weights with respect to the character error rate by considering changes in one weight at a time. When considering one weight (e.g. the language model weight), the algorithm optimizes the error rate with respect to this one weight while keeping all other weights fixed. In this sub-optimization the error rate is a piecewise constant function of the weight. Finding the optimal weight becomes feasible by considering the following property: the score of each hypothesis that is represented in the set of lattices is a linear function of the weight that is considered. Determining the values of the parameter at which best scoring hypothesis changes is therefore equivalent to finding the intersections of linear functions. This makes it feasible to enumerate all possible values at which the error rate changes, and therefore to determine all possible values of the error rate. Picking the optimum is then also feasible. Note that varying only one parameter at a time means that this procedure cannot guarantee finding the global optimum over all parameter combinations, although it works well in practice.

8 Training Data

We use various types of training data, and the amount we use varies by script. Sources of training data include:

- Data collected through prompting volunteers to write a set of short phrases or characters on a mobile device.
- Data that we could obtain a commercial license for.
- Artificial inflation of data from existing samples to increase variability, especially for characters with very few training samples, compare e.g. the concept of 'elastic distortions' [37].
- Recognition requests which are not associated with a user and are labeled by humans.
- Recognition requests which are not associated with a user and are self-labeled by one of the recognizers.

We apply a simple binary log-linear classifier as a filter to the self-labeled data to exclude items that are likely to be just 'scribbles'. Self-labeled data is then used in the training process in the same way as labeled data.

The number of training samples varies from tens of thousands to several million, depending on the script. We tend to use more training items for languages that have many possible recognition classes (like Chinese or Devanagari script) and for languages that have a lot of usage.

9 Language-specific Adaptations

Table 2 gives an overview of the different languages and scripts that we support in our system, their characteristics, and how we adapted our system to handle them.

The *Latin* script recognizer serves the largest family of a total of 52 languages. Beyond the basic 2×26 characters, it supports the accented (or otherwise modified) characters needed to support the 52 languages. The total set of supported labels in the character classifier is slightly larger to support numbers, punctuation marks, and a few special characters such as a heart, a smiley, and a checkmark.

Vietnamese, while a Latin script language, is handled in a recognizer separate from the other Latin script recognizers because of the large number of diacritics on many of the characters in that language which are not used in any other Latin script language (Fig. 5b). Otherwise it is nearly identical to the Latin script recognizer in its configuration.

Similarly, the *Cyrillic and Greek* script recognizers also are nearly identical in configuration to the Latin script recognizer. The larger gap between the alphabet sizes and the number of supported labels in these recognizers is due to our decision to support the basic Latin script alphabet in all of our recognizers independently of whether or not they are part of the official alphabet because nearly all languages seem to occasionally mix with Latin script text.

Chinese script is quite different from the previously discussed scripts in many respects. First, the total number of characters is about 50,000 but many of these are mostly used in historical scripts. Most Chinese actively know about 3,000–4,000 characters and we chose to support a superset of about 12,000 characters including the most commonly used characters in simplified Chinese (used mostly in mainland

China), traditional Chinese (mostly used in Taiwan), and Cantonese (mostly used in Hong Kong). In addition, like all our recognizers, the Chinese recognizer supports the basic Latin alphabet, numbers, and a variety of punctuation marks. Second, Chinese characters are often far more complex than e.g. Latin script characters and an individual character can consist of up to 36 strokes. The individual characters often have a rectangular shape and when writing cursive, characters are not connected but only the strokes within a character are connected. In addition, a standard stroke order of writing Chinese characters is taught in schools and most writers employ this ordering. To account for these properties, we adapted the Chinese system in terms of preprocessing (none), features to represent the ink, and classifiers that we use. As features, we use only bitmap features and quantized stroke direction maps (cp. Sec. 5.3.3). The stroke direction maps are used in two configurations: once in the same way as described before and once after we insert pen-up strokes (connections between the last point of one stroke and the first point of the next one) into the ink. These help to reduce the variation between cursive and printed writing, and emphasize the common stroke order. To support the large number of different characters we use an approximate nearest neighbor classifier [28] which improves the accuracy for rare characters (Sec. 5.3.4).

The *Japanese* recognizer is configured identically to the Chinese one. Japanese uses three different alphabets: Hiragana (46 characters), Katakana (48 characters), and Kanji which are adopted Chinese characters. The set of Chinese characters that we use in Japanese contains 6355 characters which we support in addition to the full Hiragana and Katakana alphabet as well as the basic Latin alphabet, numbers and punctuation.

The *Korean* script is explicitly combinatorial: it combines individual pieces (28 Jamo) into larger syllabic units called Hangul (Fig. 5a). In theory, 11,172 distinct Hangul are defined but in practice only about 2,000 are used. Our recognizer supports about 3,000 Hangul and all Jamo, as well as Latin script, numbers, and punctuation symbols. To allow for adding support for some not very commonly used Hangul we generate some artificial training data by recombining Jamo segmented from other characters into new Hangul. When building the recognizer we experimented with settings similar to the Chinese recognizer and settings similar to our Latin script system. We found that accuracy for Korean was best for a system similar to the Latin script system with only the segmenter based on the same heuristic as the Chinese system.

Arabic script is another that poses several challenges: First, Arabic script is generally written cursively, i.e. nearly all characters within a word are connected to their neighbors, making segmentation hard. Interestingly, for segmenting Arabic script, the same heuristic as used for cursive Latin script can be applied. Second, most characters have different representations that depend on their position within a word, e.g. the characters look different when written isolated, at the beginning of a word, in the middle of a word, or at the end of a word (Fig. 5c). While the basic Arabic alphabet consists of only 28 letters (and an additional 18 letters to allow for supporting Urdu and Persian) and no cases, due to the presentation forms and common ligatures we support a total set of about 250 character variations, in addition to both Latin script numerals and two variants of Arabic script numerals, the basic Latin script alphabet, and punctuation marks. Third, the differences between characters often are subtle, e.g. depend on the exact position of dots, which may be accidentally misplaced in handwriting and thus Arabic character recognition is more ambiguous than most other scripts. Fourth, Arabic script is written from right-to-left, but numbers (or words in Latin script) are written left-to-right, which effectively makes the language bidirectional. To support the bidirectional input we chose to disable the time-order in our lattice creation and use the spatial ordering only. To make the handling of Arabic script easier, we flip the ink horizontally, which makes Arabic script left-to-right for recognition purpose and potentially embedded numbers or Latin script regions are right-to-left, which we reorder as a special case in postprocessing.

Hebrew script, like Arabic, is written bidirectionally but doesn't share the cursiveness. Hebrew has only 22 letters (five of which take a different form at the end of a word) and no cases. In Hebrew handwriting, characters are rarely connected which allows for having the smallest set of supported labels in our Hebrew recognizer including basic Latin script support, numbers, and punctuation marks.

The remaining *Indic and South-east Asian scripts* all are Brahmic-derived scripts and share the property that they formally have a relatively small alphabet but the effective number of character hypothesis labels is much larger. E.g. up to 12 consecutive characters (in Odiya) can form a *grapheme cluster*. Examples for grapheme clusters are shown in Devanagari script (Fig. 5d), Khmer (Fig. 5e), and Lao (Fig. 5f). Grapheme cluster forming can be complex, non-monotonic (i.e. later Unicode codepoints in the output string are rendered to the left of earlier ones), two-dimensional (layout above or below), or subtractive (adding a code point removes parts of a character). In these scripts the consonants typically carry an implicit vowel sound and in many of them, the Virama is a special character that removes the implicit vowels and in writing forms grapheme clusters.

Devanagari script, which is used for Hindi, Marathi, Nepali, and many other languages, has 33 consonants and 14 vowels and defines a large variety of grapheme clusters. It is also characterized by the Shirorekha, a line connecting letters at the top. While many users write the Shirorekha line in handwriting, not all do.

Gurmukhi (used for *Punjabi*, 41 consonants and 10 vowels), *Bengali* (39 consonants and 11 vowels), and *Gujarati* (36 consonants and 15 vowels) scripts are very similar to Devanagari script. Gurmukhi and Bengali also use the Shirorekha, Gujarati does not.

Telugu (35 consonants + 18 vowels) and *Kannada* (25 structured + 11 unstructured consonants, 14 vowels) share a lot of similarity. So do *Tamil* (24 consonants, 12 vowels), *Malayalam* (41 consonants + 15 vowels), and *Sinhala* (20 consonants, 23 vowels) and to a lesser degree *Odiya* (38 consonants 14 vowels) and *Burmese* (33 consonants, 21 vowels).

Languages like Malayalam, Odia (Oriya), Tamil, and others with vowels which appear to the left of (or on both sides of) a cluster of consonants present special challenges. We typically recognize the left vowel in the order it is written with respect to the consonant cluster, and as a post-processing step reorder the sequence to the standard Unicode order placing the vowel at the end. In some languages like Malayalam, there are multiple ways to write consonant clusters (depending on the usage of traditional or reformed

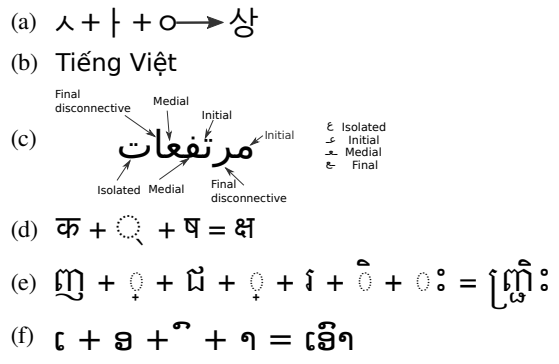


Fig. 5. Illustrations of script specific properties.

orthography), which can result in the left vowel appearing at different locations. The post-processing normalizes these variations to the standard Unicode encoding order.

Finally at the bottom of Table 2 we note the three Asian languages that complete the list: *Khmer* (33 consonants, 16 vowels) is a predecessor of *Lao* (34 consonants and 33 vowels) and *Thai* (44 consonants and 15 vowels).

10 Experiments and Results

One problem with comparing online handwriting recognition systems is that there are only a few publicly available databases to compare results on. Several publications in the scientific community use the IAM database [38] for comparison for the English language and we include comparison results on the corresponding test set below. Other existing databases with more suitable licenses often have very few if any comparison results or are otherwise restricted. One database that was used frequently in the past is the UNIPEN-1 dataset [14], but most evaluations were restricted to single character recognition. We also present some results on this dataset below.

For quantitative evaluations of other factors we present results on internal datasets of items in the given languages that we use for evaluation of our system. These datasets contain items which resemble the traffic that we observe in our system, e.g. many short phrases, word-parts, or single letters, also some words that are more frequent in other languages (e.g. Spanish text in the English recognizer) and many words that are not dictionary words such as uncommon proper names. As the measure of performance we use the character error rate (CER) or word error rate (WER) obtained from an edit distance alignment of the output to the label of the item. As usual, we accumulate insertions, deletions, and substitutions and divide by the number of characters/words in the label.

One question in this context is whether a recognition system should return a 'spell-corrected' version of the input or try to be as true to the written text as possible. For our experiments, we have chosen the latter approach and generally try to transcribe any user input exactly as written without an integrated spelling correction, and do the same in labeling evaluation sets. If an application wants spell-corrected user input, this often can be performed as a postprocessing step.

10.1 Single-Character Experiments on UNIPEN-1

We used the R01/V07 version of the UNIPEN-1 data with a fixed 2:1 random partition into training and test data (not writer-independent) and applied a simple aspect-ratio based cleaning method according to the protocol from [11]. We trained a single-character recognizer similar to those that we use as character classifier in our full system and observed competitive error rates as shown in Table 3. The table shows comparison results for the best 'multi-writer' experiment from [11] following the same experimental protocol. We also included results from [13]. For additional comparison results, we refer to [11] which compares to a wide range of state-of-the-art comparison results. We observe that the resulting error rates are very competitive but admit that the comparison results are more than a decade old.

TABLE 3
Error rates on UNIPEN-1 data in comparison to the state of the art.

	error rates [%]		
UNIPEN dataset	this work	[11]	[13]
1a	0.8	1.1	2.9
1b	3.1	4.3	7.2
1c	5.1	7.9	9.3

TABLE 4
Error rates on IAMDB test set in comparison to the state of the art. A '*' in the 'system' column indicates the use of an open training set.

system	CER[%]	WER[%]
HMM [38]	-	33.4
commercial-1 [15] *	-	28.7
LSTM (comparison, see Sec. 10.3) *	8.8	26.7
commercial-2 [15] *	-	20.8
LSTM [9]	11.5	20.3
LSTM [15]	-	18.9
combination [15] *	-	13.8
this work *	4.3	10.4

10.2 Word Recognition Experiments on IAM-OnDB

We also performed experiments on the test set IAM-OnDB-t2 of the IAM database [38] that contains 3,859 items. We used our existing English recognizer, so this comparison should be regarded as 'open training set'. The results are summarized in Table 4. No optimization with respect to the data was done on our side for these experiments. Again, we observe that the resulting error rates are very competitive with respect to other published results we are aware of.

10.3 Word Recognition Experiments on Internal Data

We evaluate the recognition accuracy of our systems for all supported languages regularly on validation data sets and less frequently on test/holdout sets to be able to spot any signs of overfitting to the validation data. Here we show results for the validation sets in a few languages to illustrate the effect of several of the system components and the importance of a sufficiently large training set. The set of languages that we are evaluating in this paper as well as the sizes of the validation sets are given at the top of Table 5.

Table 5 also shows the improvement of the character error rate for our system when successively adding more of the system components described above to a baseline system for each of the languages.

The baseline system in line (1) uses only the character classifier and the simple frequent-word-based LM as feature function in the search. The character classifier uses only the first-order pointwise features as a reduced set of ink features to represent the character hypotheses. In line (2) we

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPAMI.2016.2572693, IEEE Transactions on Pattern Analysis and Machine Intelligence

TABLE 2
Summary of the characteristics of the different languages and scripts and how we adjust our system to handle them.

script	example	# languages	alphabet size	total supported labels	writing direction	delayed strokes	stroke orders[1]	whitespace	fixed stroke-order	grapheme clusters	segmenter heuristic[2]	comments
Latin script	English	59	$2 \times 26 + 2 \times 92 + 1$	293	LTR	yes	T/S	yes	no	no	min-and-between	
Vietnamese	tiếng Việt	1	$2 \times 26 + 2 \times 67$	223	LTR	yes	T/S	yes	no	no	min	
Cyrillic script	русский	11	$2 \times 24 + 2 \times 33$	237	LTR	yes	T/S	yes	no	no	min-and-between	
Greek	ελληνικά	1	$2 \times 33 + 3$	183	LTR	yes	T/S	yes	no	no	min-and-between	
Chinese	中文	3	$\sim$50,000	12362	LTR	no	T	no	yes	no	strokes	[3]
Japanese	日本語	1	$46 + 48 + \sim$50,000	6710	LTR	no	T	no	yes	no	strokes	[3]
Korean	한국말	1	$11{,}172 + 28$	3318	LTR	no	T	yes	yes	no [4]	strokes	
Arabic script	العربية	3	$28 + 18$ [5]	353	bidi	yes	S	yes	no	yes [6]	min-and-between	
Hebrew	עברית	1	$22 + 5$	117	bidi	yes	S	yes	no	no	min	
Devanagari script	हिन्दी	3	47	4625	LTR	Shirorekha	T/S	yes	no	yes [7]	strokes	[8]
Punjabi script	ਪੰਜਾਬੀ	1	51	678	LTR	Shirorekha	T/S	yes	no	yes [7]	min-and-between	[8]
Bengali	বাংলা	1	50	2471	LTR	Shirorekha	T/S	yes	no	yes [7]	min	[8]
Gujarati	ગુજરાતી	1	51	892	LTR	yes	T/S	yes	no	yes [7]	min	
Telugu	తెలుగు	1	53	978	LTR	yes	T/S	yes	no	yes [7]	min	
Kannada	ಕನ್ನಡ	1	50	1293	LTR	yes	T/S	yes	no	yes [7]	strokes	
Tamil	தமிழ்	1	36	807	LTR	yes	T/S	yes	no	yes [7]	min	
Malayalam	മലയാളം	1	56	1987	LTR	yes	T/S	yes	no	yes [7]	min	
Sinhala	සිංහල	1	43	2998	LTR	yes	T/S	yes	no	yes [7]	strokes	
Odiya	ଓଡ଼ିଆ	1	42	1808	LTR	yes	T	yes	no	yes [7]	min	
Burmese	မြန်မာအက္ခရာ	1	54	1148	LTR	yes	T	yes	no	yes [7]	min-and-between	
Khmer	ភាសាខ្មែរ	1	49	3966	LTR	yes	T	no	no	yes [7]	min-and-between	
Lao	ພາສາລາວ	1	67	855	LTR	yes	T	yes	no	yes [7]	min-and-between	
Thai	ภาษาไทย	1	59	1305	LTR	yes	T	yes	no	yes [7]	min	

[1]: The stroke orderings considered when creating the segmentation lattice (Sec. 5.2): T = time order, S = spatial order.
[2]: The segmenter heuristics are: *strokes*: insert cut points between each pair of strokes; *min*: in addition to the 'stroke' cuts, insert cut points at local minima of y coordinates of the ink; *min-and-between*: in addition to the 'min' cuts, insert additional cuts between two cuts if the segment is long enough.
[3]: We use not only a neural network for character classification, but also an approximate nearest neighbor classifier. To make that effective, we reduced the features to represent character representations to use only the bitmap features and quantized stroke direction histograms (Sec. 5.3.3).
[4]: Hangul could be seen as grapheme clusters of the underlying Jamo. We use the Hangul Unicode representation directly.
[5]: While the Arabic script only has fewer letters we learn explicit models for the presentation forms.
[6]: We use presentation forms and ligatures to represent characters during training.
[7]: We use grapheme clusters as defined per Unicode as well as subgraphemes that we defined to allow for more data sharing.
[8]: We remove the Shirorekha line as a preprocessing step to make segmentation easier.

added additional feature functions that are used in decoding: cut-types, relative size features, and the character prior (cp. Sec. 5.4). This leads to an improvement of 43% relative in character error rate for English. In line (3), we added more character features to the feature extraction: second order pointwise features (cp. Sec. 5.3.2) and all character global features (cp. Sec. 5.3.3). Using the full set of features leads to 12% relative improvement here for English. In line (4), making the system complete by adding the n-gram-based character language model leads to another 41% relative improvement for English.

To provide a perspective of how our system compares to other systems described in the literature we give results for an LSTM-system using connectionist temporal classification that closely follows the system described in [9]. This system uses 40 pointwise and bitmap features (cp. Sec. 5.3.2) to represent the ink and has 100 LSTM nodes. It was trained on the same (supervised) training set as our main system and uses the same character-based language model as our main system. As shown in Table 4, this system obtains a better CER on the IAM dataset than the original implementation but it does have a higher WER. This difference in results may be partially explained by our Latin-script training set having a higher diversity in the used vocabulary (including many non-English words). On our in-house datasets, our system performs better than the LSTM-system on all of the evaluated languages but Thai and Hindi, where the LSTM-system has a small edge. In particular in Korean, the big gap is probably explained by the fact that our system is able

TABLE 5
Character error rates on the validation data using successively more of the system components described above for English (en), Spanish (es), German (de), Arabic (ar), Korean (ko), Thai (th), and Hindi (hi) along with the respective number of items and characters in the sets.

Language	en	es	de	ar	ko	th	hi
Items	5,320	4,137	11,408	8,581	19,955	20,599	6,030
Characters	24,287	25,578	66,289	71,637	45,803	96,726	25,548
System	CER [%]						
(1) Baseline	26.1	23.5	21.1	25.4	43.9	6.9	26.4
(2) + feature functions	14.8	16.0	13.5	23.1	35.3	6.4	24.4
(3) + more features	12.9	12.1	9.4	17.4	17.4	5.0	16.6
(4) + n-gram LM	7.5	7.2	6.0	14.8	13.8	4.1	15.7
LSTM (comparison)	10.2	12.4	9.5	18.2	44.2	3.9	15.4

to handle the large number of characters and the complex character structure better. Note that we did not engineer the LSTM system as carefully as our system, so these results cannot be used to claim that our approach is inherently more suitable for this task. However, the competitive results on Thai and Hindi show that this bias is not large.

Fig. 6 shows the dependence of the error rate on the size of the training set. We ran the training pipeline on datasets that resulted from repeatedly and randomly subsampling the training data by a factor of 2. Similar to results presented in [39], we observe an expected improvement of the classifier with increasing training set size. Note that adding about the same amount of self-labeled (or unlabeled) training data in addition to the labeled data improves the results about as much as the last doubling step. In experiments with further increased amounts of unlabeled data we did not observe

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPAMI.2016.2572693, IEEE Transactions on Pattern Analysis and Machine Intelligence

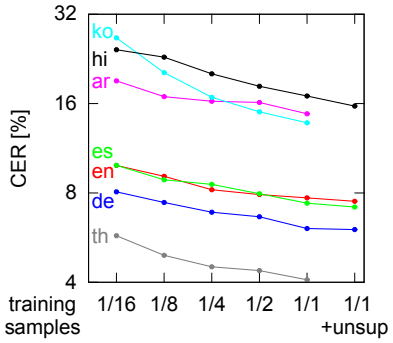


Fig. 6. Character error rates for our system with respect to the number of training samples used. Each point on the lines corresponds to doubling the training set. The point at the position "1/1 + unsup" corresponds to using the full set of labeled data along with a similar amount of unsupervised (self-labeled, unlabeled) data.

TABLE 6
Languages by usage (cloud recognizer) over a week in Jan. 2016.

1	English	9	French
2	Chinese	10	Hindi
3	Japanese	...	...
4	Spanish	17	Malayalam
5	Russian	...	...
6	Arabic	25	Gujarati
7	German	...	...
8	Korean	28	Khmer

further improvements for this particular data set. The full English training set consists of about one million labeled samples and one million self-labeled samples. The last point of each line corresponds to the full system (4) in Table 5.

10.4 Language Use

Table 6 shows the most used languages in our cloud recognizer ordered by usage. We included Malayalam, Gujarati, and Khmer because as far as we know there are no other word-based online handwriting recognition systems for these languages. The list does not hold great surprises, but it shows that the system is used in a wide variety of scripts and languages. Usage seems to be influenced by availability of mobile devices and the ease of use of handwriting in comparison to other mobile input methods.

Without going into more detail we would like to make two statements underscoring the importance of online handwriting recognition technology and research:

- In many languages, the most-entered phrase uniformly seems to be 'I love you' or its translation in the respective language. In that sense, handwriting recognition brings some love to the world.
- In some products, the usage of online handwriting recognition can be compared to other input methods like speech recognition or camera OCR. In some languages and for some products, we observe that the usage is of the same order of magnitude. In that sense, online handwriting recognition adds significant user choice to the set of available input methods.

10.5 Example Errors

Figure 7 shows some example errors made by our English language system. Each item is annotated with the 'ground

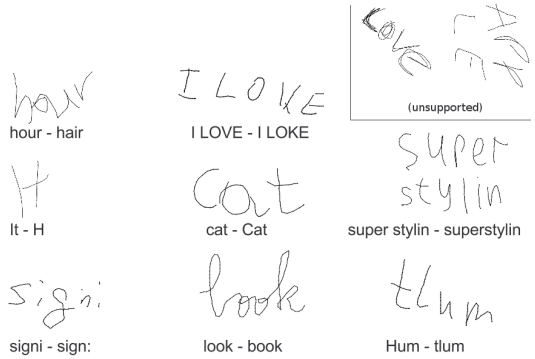


Fig. 7. Example errors. See text for details.

truth' label and the recognition result. The top right shows two cases of input that our system currently does not handle well: characters written with many repeated strokes and full 90-degree rotation. These types of input are infrequent enough that a special handling does not seem warranted at the moment, even though it would be possible to handle these cases specifically. Other examples show that both the human transcription and the recognition can be valid decodings of the input. E.g. both 'signi' (as a prefix of 'significant') or 'sign:' can be valid interpretations. Based on this (not surprising) observation, we do allow multiple correct labels in evaluations. An interesting special case is the recognition of 'I LOKE' which may seem like a problem that should be easily fixed by using a better language model. However, a stronger emphasis on the language model than the one picked by the MERT weight learning would lead to more errors on some other out-of-vocabulary items. One example for this effect is that the interpretation of the example in the lower left of Fig. 7 becomes 'signs' when the language model weight is increased.

10.6 Test Data Drift over Time

One interesting problem, which does not typically occur in academic research, but which we have observed with respect to system evaluation is that the (anonymous) traffic we see in our recognizers changes over time. This drift makes it important to continuously adapt the system to new requirements. The existence of the drift is not surprising (compare e.g. the discussion of 'Population Drift' in [40]) and there are several reasons for this, including

- Users observe that a specific style of writing is recognized better and prefer that in the future (e.g. printed English is often recognized with fewer errors than cursive English).
- Some users observe that the recognizer doesn't recognize their handwriting well and only try it a few times, then never use this feature again.
- Handwriting support gets included in new products and the population of users is different, e.g. handwriting input was first launched in Google Translate for Android, and somewhat later in the iOS version (which have different numbers of users in each country/language).
- Handwriting support gets included in new products and the input characteristics are different, e.g. handwriting input on the Google Translate desktop version

has many users entering text by handwriting with their mouse rather than on a touch screen device.
- New features are added to the recognizers and users start using them, e.g. the concurrent recognition of overlapping and delayed-strokes writing for Latin script.

We address this problem by evaluating against an 'evolving' set of validation and test data. We regularly add more items to the validation and test data sets based on the current traffic that we observe. We also allow to add alternative labels, e.g. for the item in the lower left corner of Fig. 7 we may allow both labels as correct, because we do not want to base a comparison of quality between two recognizers on this difference. Lastly, we sometimes discover obvious label errors, i.e. the originally associated label did not correspond to what was written. When comparing results across two data sets, we always use the most current version of the labels and alternative labels that we have in our data base.

11 On-Server vs. On-Device Recognition

The presented system was originally designed to work well on high-performance servers that allow the use of many cores, large cpu-caches, and a comparably large amount of memory. For applications that need access to the web, it is very reasonable to also let the recognition happen in the cloud. Examples for these are entering a query for web-search using handwriting and entering text to be translated into another language. In other cases, such as an input method for a mobile phone or tablet, the recognition should happen on the device that records the handwriting. These devices typically have much more limited computational resources. To allow recognition to be performed on a mobile device, we followed the general principle of sharing as much of the code and models as possible, but to make recognition fast, a few changes were necessary:

- Use slightly fewer features.
- Use fewer hidden nodes in the neural network for classification.
- Use a smaller language model and space-optimized representation; use only a character n-gram.
- Do not use feature functions that require second-order dependencies in search.
- Use adapted pruning thresholds and fewer segmentation hypotheses.
- Do not use the additional nearest neighbor classifier for Chinese and Japanese.

Because usability as an input method is strongly correlated to the latency of recognition, we aim to recognize the inputs quickly. The cloud recognition system for English usually takes around 40ms per recognized character. The user-perceived latency for the cloud system also includes the network round trip time, so much further speedup is not as impactful as it is for on-device recognition. Recognition on-device (on an 'LG Nexus 5' device, equipped with a 2.26 GHz quad-core processor) for English takes around 25ms per character using a system that is reduced in the ways described above. (Note that recognition on devices with a less powerful processor may of course take longer.) The error rate for the on-device recognizer is typically between 10% and 40% worse than the cloud recognizer, relative. For English, the relative difference is around 25%.

12 Conclusion

We presented the architecture of a real-world, multi-script and multi-language online handwriting recognition system. The system combines several existing and some new components. A key emphasis of the system is on the re-use of components across many scripts and languages, which makes the problem tractable as an engineering problem. We discussed a variety of script- and language-specific elements that are used. Novel ideas presented include the following:

- We combine time- and position-based interpretation of the input in a single lattice for joint decoding. This allows to recognize overlapping writing and handle delayed diacritical marks.
- We train a segmenter as a filter to determine which hypothetical cut points are valid character segmentations, which allows to keep the search space small but still achieves a high recall segmentation.
- We use MERT, a method for learning feature weights known from machine translation, to combine up to 35 knowledge sources (such as character classifiers, language models, segmenter scores, and others).
- A flexible hierarchy of pruning approaches allows the same system to run both on powerful servers and on less powerful mobile devices.
- Because we use only a minimal amount of preprocessing, the system is able to recognize handwriting inputs from a large variety of input sources, including text entry on mobile devices with touch screens, mouse-drawn handwriting inputs on the web, and input using reconstructed writing paths from gyroscope data [41].

The system described in this paper is currently used in several publicly available products which include mobile and desktop variants of 'Google Translate', 'Google Handwriting Input' for Android mobile devices.

Acknowledgments

We would like to thank the numerous people within and outside of Google who have made the development of this system possible through their help. To name a few, we would like to thank Ashok Popat, Dhyanesh Narayanan, Dmitriy Genzel, Duc Ngo, Franz Och, Jan Bulanek, Kester Tong, Lawrence Chang, Linne Ha, Masakazu Seno, Matej Vecerik, Michael Jahr, Pavol Safarik, Remco Teunen, Sanjiv Kumar, Stefano Pellegrini, Tomasz Wozniak, Wolfgang Macherey, and Yang Li.

References

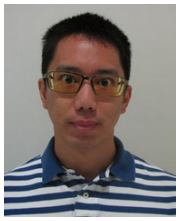
[1] G. Nagy, "Twenty years of document image analysis in PAMI," *IEEE Trans. Pattern Analysis & Machine Intelligence*, vol. 22, no. 1, pp. 38–62, 2000.
[2] C. C. Tappert, C. Y. Suen, and T. Wakahara, "The state of the art in online handwriting recognition," *IEEE Trans. Pattern Analysis & Machine Intelligence*, vol. 12, no. 8, pp. 787–808, Aug. 1990.
[3] R. Plamondon and S. N. Srihari, "On-line and off-line handwriting recognition: A comprehensive survey," *IEEE Trans. Pattern Analysis & Machine Intelligence*, vol. 22, no. 1, pp. 63–84, Jan. 2000.
[4] J. H. Kim and B. Sin, "Online handwriting recognition," in *Handbook of Document Image Processing & Recognition*, 2014, pp. 887–915.
[5] L. Yaeger, B. Webb, and R. Lyon, "Combining neural networks and context-driven search for on-line, printed handwriting recognition in the Newton," *AAAI AI Magazine*, 1998.
[6] J. A. Pittman, "Handwriting recognition: Tablet PC text input," *IEEE Computer*, vol. 40, no. 9, pp. 49–54, 2007.
[7] S. Jaeger, S. Manke, J. Reichert, and A. Waibel, "Online handwriting recognition: the NPen++ recognizer," *International Journal on Document Analysis and Recognition*, vol. 3, no. 3, pp. 169–180, 2001.
[8] J. Schenk and G. Rigoll, "Novel hybrid NN/HMM modelling techniques for on-line handwriting recognition," in *IWFHR*, 2006.

[9] A. Graves, M. Liwicki, S. Fernandez, R. Bertolami, H. Bunke, and J. Schmidhuber, "A novel connectionist system for unconstrained handwriting recognition," *IEEE Trans. Pattern Analysis & Machine Intelligence*, vol. 31, no. 5, pp. 855–868, 2009.
[10] S. Jäger, C. Liu, and M. Nakagawa, "The state of the art in Japanese online handwriting recognition compared to techniques in western handwriting recognition," *International Journal on Document Analysis and Recognition*, vol. 6, no. 2, pp. 75–88, 2003.
[11] E. H. Ratzlaff, "Methods, report and survey for the comparison of diverse isolated character recognition results on the UNIPEN database," in *ICDAR*, 2003, pp. 623–628.
[12] A. Delaye and É. Anquetil, "HBF49 feature set: A first unified baseline for online symbol recognition," *Pattern Recognition*, vol. 46, no. 1, pp. 117–130, 2013.
[13] C. Bahlmann and H. Burkhardt, "The writer independent online handwriting recognition system frog on hand and cluster generative statistical dynamic time warping," *IEEE Trans. Pattern Analysis & Machine Intelligence*, vol. 26, no. 3, pp. 299–310, Mar. 2004.
[14] I. Guyon, L. Schomaker, R. Plamondon, M. Liberman, and S. Janet, "UNIPEN project of on-line data exchange and recognizer benchmarks," in *ICPR*, 1994, pp. 29–33.
[15] M. Liwicki, H. Bunke, J. A. Pittman, and S. Knerr, "Combining diverse systems for handwritten text line recognition," *Mach. Vis. Appl.*, vol. 22, no. 1, pp. 39–51, 2011.
[16] F. Biadsy, R. Saabni, and J. El-Sana, "Segmentation-free online Arabic handwriting recognition," *IJPRAI*, vol. 25, no. 7, pp. 1009–1033, 2011.
[17] A. Bharath and S. Madhvanath, "Hidden Markov models for online handwritten Tamil word recognition," in *ICDAR*, 2007, pp. 506–510.
[18] C. Liu, S. Jäger, and M. Nakagawa, "Online recognition of Chinese characters: The state-of-the-art," *IEEE Trans. Pattern Analysis & Machine Intelligence*, vol. 26, no. 2, pp. 198–213, 2004.
[19] I. Bazzi, R. Schwartz, and J. Makhoul, "An omnifont open-vocabulary OCR system for English and Arabic," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 21, no. 6, pp. 495–504, 1999.
[20] A. Stolcke *et al.*, "SRILM-an extensible language modeling toolkit." in *INTERSPEECH*, 2002.
[21] G. Seni, R. K. Srihari, and N. M. Nasrabadi, "Large vocabulary recognition of on-line handwritten cursive words," *IEEE Trans. Pattern Analysis & Machine Intelligence*, vol. 18, no. 7, pp. 757–762, 1996.
[22] D. Goldberg and C. Richardson, "Touch-typing with a stylus," in *INTERCHI*, 1993, pp. 80–87.
[23] D. Keysers, T. Deselaers, C. Gollan, and H. Ney, "Deformation models for image recognition," *IEEE Trans. Pattern Analysis & Machine Intelligence*, vol. 29, no. 8, pp. 1422–1435, 2007.
[24] R. Arandjelovic and A. Zisserman, "Three things everyone should know to improve object retrieval," in *CVPR*, 2012, pp. 2911–2918.
[25] U. Pal, A. Belaïd, and C. Choisy, "Touching numeral segmentation using water reservoir concept," *Pattern Recognition Letters*, vol. 24, no. 1-3, pp. 261–272, 2003.
[26] Z. Bai and Q. Huo, "A study on the use of 8-directional features for online handwritten Chinese character recognition," in *ICDAR*, 2005, pp. 262–266.
[27] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. W. Senior, P. A. Tucker, K. Yang, and A. Y. Ng, "Large scale distributed deep networks," in *NIPS*, 2012, pp. 1232–1240.
[28] D. Vanderkam, R. Schonberger, H. Rowley, and S. Kumar, "Nearest neighbor search in Google Correlate," Google, Inc., Tech. Rep., 2013. [Online]. Available: http://research.google.com/pubs/archive/41694.pdf
[29] F. Kimura, K. Takashina, S. Tsuruoka, and Y. Miyake, "Modified quadratic discriminant functions and the application to chinese character recognition," *IEEE Trans. Pattern Analysis & Machine Intelligence*, vol. 9, no. 1, pp. 149–153, 1987.
[30] X. Zhou, J. Yu, C. Liu, T. Nagasaki, and K. Marukawa, "Online handwritten japanese character string recognition incorporating geometric context," in *ICDAR*, 2007, pp. 48–52.
[31] L. Rabiner and B.-H. Juang, *Fundamentals of Speech Recognition*. Prentice Hall, 1993.
[32] P. Brown, S. Della Pietra, V. Della Pietra, and R. Mercer, "The mathematics of statistical machine translation: Parameter estimation," *Computational Linguistics*, vol. 19, no. 2, pp. 263–311, 1993.
[33] T. Brants, A. Popat, P. Xu, F. Och, and J. Dean, "Large language models in machine translation," in *EMNLP-CoNLL*, 2007, pp. 858–867.
[34] G. Jacobson, "Space-efficient static trees and graphs," in *Foundations of Computer Science*, 1989, pp. 549–554.
[35] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the EM algorithm," *Journal of the Royal Statistical Society: Series B*, vol. 39, pp. 1–38, 1977.
[36] W. Macherey, F. Och, I. Thayer, and J. Uszkoreit, "Lattice-based minimum error rate training for statistical machine translation," in *EMNLP 2008*, 2008, pp. 725–734.
[37] P. Y. Simard, D. Steinkraus, and J. C. Platt, "Best practices for convolutional neural networks applied to visual document analysis," in *ICDAR*, 2003, pp. 958–962.
[38] M. Liwicki and H. Bunke, "IAM-OnDB-an on-line English sentence database acquired from handwritten text on a whiteboard," in *ICDAR*, 2005, pp. 956–961.
[39] H. A. Rowley, M. Goyal, and J. Bennett, "The effect of large training set sizes on online Japanese Kanji and English cursive recognizers," in *IWFHR*, 2002, pp. 36–40.
[40] D. J. Hand, "Classifier technology and the illusion of progress," *Statistical Science*, vol. 21, no. 1, pp. 1–14, 2006.
[41] T. Deselaers, D. Keysers, J. Hosang, and H. A. Rowley, "Gyropen: Gyroscopes for pen-input with mobile phones," *IEEE T. Human-Machine Systems*, vol. 45, no. 2, pp. 263–271, 2015.

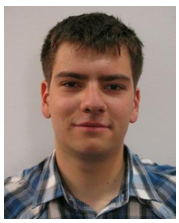
Daniel Keysers is a software engineer at Google Switzerland. Before that, he was a postdoc at the German Research Center for Artificial Intelligence (DFKI) in Kaiserslautern, Germany. He received his PhD and diploma degrees from RWTH Aachen University in Germany and spent parts of his studies at Univ. Complutense de Madrid and Univ. Politecnica de Valencia, both in Spain.

Thomas Deselaers is a software engineer at Google Switzerland. Prior he was a postdoc in the computer vision group of ETH Zurich in Switzerland. He received his PhD and diploma degrees from RWTH Aachen University in Germany. His personal interests include applied machine learning and mobile computing and combinations of these.

Henry A. Rowley is a research scientist at Google, Inc. where he has worked on handwriting recognition, computer vision, approximate nearest neighbor search, and machine learning. He received his Masters and PhD in Computer Science from Carnegie Mellon University, and BS degrees in Electrical Engineering and Computer Science from the University of Minnesota.



Li-Lun Wang is a software engineer at Google Inc. He received his PhD degree from University of Illinois at Urbana-Champaign and his B.S. degree from National Taiwan University. His interests are diverse, including artificial intelligence and machine learning.



Victor Carbune is a software engineer at Google Switzerland. He received his master's degree in Computer Science at ETH Zurich, doing his master thesis at the Learning and Adaptive Systems Group. He received B.S. degree from Politehnica University of Bucharest, Romania. His interests lie at the boundary between research and large-scale engineering, related to machine learning and data mining algorithms.