FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Bachelorarbeit

Eine Bibliothek zur handschriftlichen Eingabe mathematischer Ausdrücke

von

Daniel Ziegler

Matrikel-Nr.: 21909180

Betreuung:
Prof. Dr.-Ing. Jürgen Teich
Christian Schmitt, M.Sc.

3. November 2017

Dieses Dokument wurde mit dem Textsatzsystem LaTeX2e erstellt.

Inhaltsverzeichnis

Aufgabenstellung zur Bachelorarbeit

Grundlagen: Das Forschungsprojekt ExaStencils[1] verfolgt einen domänenspezifischen Entwurfsansatz für die Klasse von Stencil-Codes. Hierbei ist revolutionär, dass es sich nicht an die Beschleunigung von existierenden Programmen richtet, sondern eine neue Softwaretechnologie anstrebt, die eine automatische, domänen- und plattformspezifische Optimierung von Programmen ermöglicht und so auf einfache Weise von Anwendern ohne Expertenwissen genutzt werden kann. Stencils spielen eine zentrale Rolle in der Hochleistungssimulation. Sie sind reguläre Zugriffsmuster auf (i. d. R.) mehrdimensionalen Datengittern. Mehrgittermethoden arbeiten auf einer Hierarchie von erst sehr feinen und dann immer gröberen Gittern. ExaStencils schafft Programmierbarkeit für Exascale-Computing mittels der hierarchischen domänenspezifischen Sprache ExaSlang [SKH+14] und eines Codegenerators, der Anwenderprogramme mit Domänenwissen anreichert und dieses zur Optimierung nutzt. ExaStencils' hochentwickelte Werkzeugunterstützung beruht auf einer generatorbasierten Produktlinientechnologie, die automatisch eine für das Problem und die Ausführungsplattform maßgeschneiderte Implementierung erstellt [LAB+14]. Designiertes Ziel von ExaStencils ist es, aus einem Minimum von Benutzerspezifikation eine möglichst Optimale Implementierung des passenden Lösers maßzuschneidern. Für viele Nutzer ist eine Eingabe der zu lösenden partiellen Differentialgleichung die natürlichste und daher einfachste Variante.

Beschreibung: Um den Einstieg für neue und unerfahrene Nutzer zu vereinfachen wurde eine Web-basierte Oberfläche für den ExaStencils Code Generator (ECG) geschaffen, der neben der Eingabe in den vier ExaSlang-Ebenen auch die Spezifizierung von Zielarchitektur, Domänenwissen und Compilereinstellungen erlaubt. Um die Einstiegshürde weiter zu senken und dabei nachhaltig die umfassenden Automatismen des ECG zu demonstrieren soll es möglich sein, die Spezifikation des Problems in ExaSlang 1, d. h. die zu lösende Gleichung per Mobilgerät an das Web-Interface zu übermitteln. Hierfür existiert bereits eine prototypische Implementierung für Android-basierte Geräte, deren Erkennungsrate für handschriftliche Eingaben verbessert werden soll, basierend auf im Rahmen dieser Arbeit gewonnenen Erkenntnisse. Für die Handschriftenerkennung gibt es eine Vielzahl verschiedener Ansätze und existierender Bibliotheken. Zwei Beispiele stellen SESHAT[2] [Á15] und RNNLIB[3] dar. Darüber hinaus existieren verschiedene Ansätze um maschinelles Lernen einzusetzen, wie etwa [CS15] oder Caffe[4]. Einen ersten Vergleich und Test-Datensätze kann die Auswertung der Teilnehmer der *Competition on Recognition*

[1] http://www.exastencils.org

[2] https://github.com/falvaro/seshat

[3] https://sourceforge.net/p/rnnl/wiki/Home/

[4] https://github.com/BVLC/caffe/blob/master/examples/01-learning-lenet.ipynb

of Online Handwritten Mathematical Expressions[5] bieten.

Generell ist zu unterscheiden, wie die Eingaben erfolgen: Dies kann zum einem über ein Foto des handschriftlichen Ausdrucks erfolgen, oder über den Touchscreen des Geräts. Für Geräte mit kleinerem Bildschirm kann u. U. auch eine Gestenerkennung zur Beschreibung der mathematischen Symbole verwendet werden; für Konstrukte wie Matrizen und Brüche könnte eine spezialisierte Eingabemöglichkeit wie etwa ein Wizard geschaffen werden. Zum Teil sind solche Gestenerkennungen bereits in mobile Betriebssysteme integriert. Als leichtgewichtige Alternative bieten sich bspw. die $-Recognizer[6] der University of Washington an. Für gedruckte mathematische Ausdrücke ist in die App bereits Tesseract[7] integriert. In Frage kommende Ansätze und Bibliotheken sollen in Bezug auf die Erkennung der benötigten mathematischen Symbole und Konstrukte, aber auch in Hinblick auf ihre technische Umsetzung und Eignung für die Integration in eine gemeinsame Abstraktionsschicht evaluiert werden. Von ExaStencils benötigte mathematische Konstrukte umfassen:

- Lateinische und griechische Buchstaben in Groß- und Kleinschreibung,
- Arabische Zahlen,
- Standardfunktionen wie sin, cos, arctan
- Übliche mathematische Symbole wie etwa $+$, $-$, $=$, $\cdot$, $\rightarrow$, $\times$, $\sqrt{x}$, ∂, ∇, $\int$, Mengen ($\mathbb{R}$, $\mathbb{C}$, $\mathbb{N}$, $\mathbb{Z}$) , Klammern, und
- Übliche mathematische Konstrukte wie etwa Brüche, Indizes, Exponenten und Matrizen.

Im Einzelnen sind in der Arbeit die folgenden Aufgabenstellungen zu bearbeiten:

- Evaluierung verschiedener existierender Ansätze zur Eingabe von mathematischen Ausdrücken für Touchscreens verschiedener Größen und per Foto.
- Entwurf und Umsetzung eines Konzepts zur Integration der spezialisierten Ansätze in eine möglichst plattformunabhängige Bibliothek bzw. App, die unter Android und iOS, aber auch Desktop-Rechnern unter Windows, macOS und Linux genutzt werden kann.
- Evaluierung der entstandenen Lösung hinsichtlich Funktionsumfang, Bedienbarkeit, Verarbeitungsgeschwindigkeit und Erkennungsrate.
- Dokumentation und Zusammenschrift der Arbeit.

[5] http://www.isical.ac.in/~crohme/

[6] http://depts.washington.edu/madlab/proj/dollar/index.html

[7] https://github.com/tesseract-ocr/tesseract

Danksagung

An dieser Stelle möchte ich allen Personen danken, die mich bei der Erstellung meiner Bachelorarbeit fachlich sowie persönlich unterstützt haben. Mein Dank gilt insbesondere meinen zwei Betreuern Prof. Dr.-Ing. Jürgen Teich und Christian Schmitt des Lehrstuhls für Informatik 12 (Hardware-Software-Co-Design) der Friedrich-Alexander-Universität Erlangen-Nürnberg. Christian Schmitt hatte im Verlauf der Arbeit stets ein offenes Ohr für mich und stand mir bei Problemen mit den untersuchten Programmen (die leider keine Seltenheit waren) tatkräftig zur Seite.

Darüber hinaus gilt mein besonderer Dank meiner Familie, meiner Freundin und meinen Freunden für die viele Zeit, die sie zur Erstellung der umfangreichen Testdaten investiert haben. Ohne sie wäre die Evaluierung der implementierten $-Recognizer-Bibliothek in dieser Weise nicht möglich gewesen.

1 Einleitung

Beginnend mit der Entwicklung der ersten Computer gegen Mitte des 20. Jahrhunderts ergab sich in der Informatik die Herausforderung der Mensch-Maschine-Kommunikation (MMK). Zur Bewältigung des Medienbruchs zwischen der menschlichen und der digitalen Welt wurden stets neue und intuitive Lösungen gesucht. Durch die fortschreitende Entwicklung mobiler eingebetteter Systeme mit Touchscreen hat die MMK noch einmal deutlich an Bedeutung gewonnen. Da moderne Hardware über genügend Rechenleistung verfügt, können auch komplexere Eingabeformen wie die Handschriftenerkennung realisiert werden [SR10, KN09]. Diese stellt vor allem bei mathematischen Ausdrücken eine sehr natürliche und intuitive Form der Eingabe dar. Heutzutage werden fast ausschließlich auf Zeichenketten basierende Sprachen oder spezielle Editoren zur Eingabe mathematischer Ausdrücke in Computerprogramme genutzt. Besonders unerfahrene Nutzer sind mit diesen, für sie beschwerlichen Formen der Eingabe oft nicht vertraut und müssen sich in diese erst einarbeiten [Á15]. Ein System zur handschriftlichen Eingabe mathematischer Ausdrücke würde somit eine praktische und natürliche Benutzerschnittstelle zur Eingabe darstellen [MVGZ+13]. Zudem wird der Vorgang dadurch im Vergleich zu der Eingabe über die Tastatur deutlich beschleunigt.

Um die Einstiegshürde für unerfahrene Nutzer nachhaltig zu senken, soll solch ein Eingabesystem in das Framework des ExaStencils Code Generators (ECG) integriert werden. Dieses Programm ist ein zentraler Bestandteil des Advanced Stencil-Code Engineering (ExaStencils)-Projekts, dessen Forschungsgruppe sich aus Wissenschaftlern der Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), der Universität Passau, der Bergischen Universität Wuppertal und der University of Tokio zusammensetzt [Len].

Aktuell geht die Entwicklung im Bereich des Hochleistungsrechnens in Richtung zunehmend heterogener Systeme. Dies hat zur Folge, dass es immer schwieriger wird, Software für Hochleistungsrechner zu schreiben, die auf der einen Seite die bestmögliche Leistung erreicht, auf der anderen Seite aber portabel bleibt, um einen einfachen Wechsel des Ausführungssystems zu ermöglichen. Sobald der Quelltext des Programms manuell durch das Ausnutzen individueller Hardwareeigenschaften optimiert wird, geht dies zu Lasten der Portabilität [SKH+16]. Hieraus ergibt sich ein kostspieliger Doppelaufwand. Zum einen dauert es lange, den Quelltext von Hand für ein konkretes System unter Berücksichtigung seiner Architektur und Konfiguration zu optimieren. Zum anderen müssen umfangreiche personelle und finanzielle Ressourcen aufgebracht werden, um die notwendigen Anpassungen für eine spätere Portierung umzusetzen. Genau mit dieser Problematik beschäftigt sich das ExaStencils-Projekt. Ziel ist es, aus einer abstrakten High-Level-Spezifikation des Anwenders (z. B. für ein numerische Lösungsverfahren) automatisch den u. a. durch Parallelisierung optimierten Programmcode für die Ausführung auf dem Zielsystem zu generieren [SKH+14]. Zur Spezifikation des Problems wurde die domänenspezifische

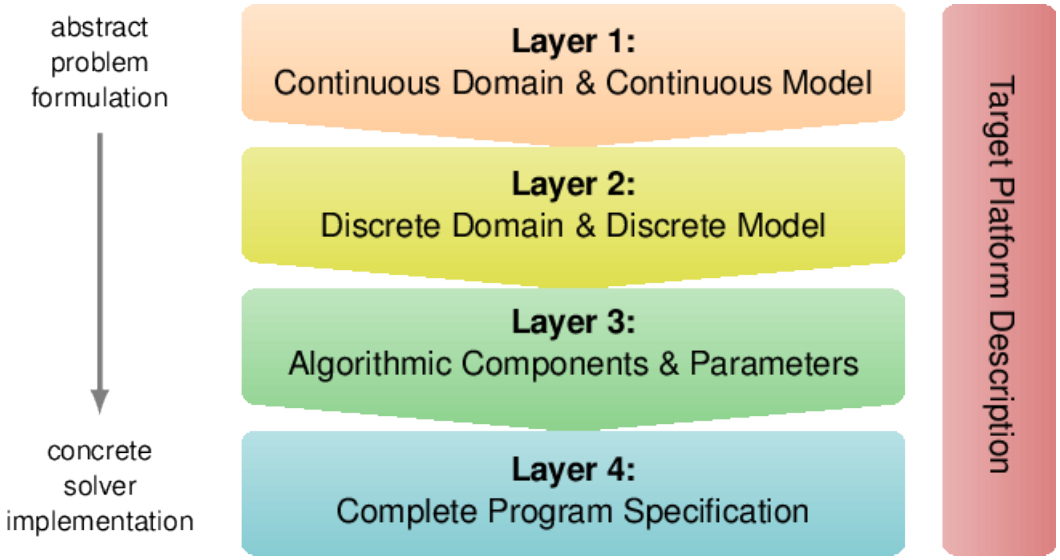


Abbildung 1.1: Strukturierung von ExaSlang in vier Ebenen [SKH+14]

Sprache (DSL) ExaSlang entwickelt, welche in vier Schichten untergliedert ist [SKH+14]. Die durch die DSL erreichte Trennung zwischen Algorithmus und Implementierung ermöglicht es in diesem Fall, dass Experten aus der Anwendungsdomäne Algorithmen spezifizieren können, ohne sich Gedanken über die Implementierung machen zu müssen. Abbildung 1.1 veranschaulicht die Schichtenhierarchie. Auf Ebene 1 (ExaSlang 1) kann das zu lösende Problem z. B. als partielle Differentialgleichung, welche gelöst werden soll, eingegeben werden. Der im Rahmen des Projekts entwickelte Kompiler überführt die Beschreibung in ExaSlang 1 schrittweise in den resultierenden maschinennahen C++-Programmcode. Dabei kann der Hardwareaufbau des Zielsystems mithilfe einer Target Platform Description Language (TPDL) genau beschrieben werden, sodass umfassende Möglichkeiten zur Optimierung ausgeschöpft werden können. Mithilfe des ECGs kann der gesamte Vorgang von Problemspezifikation auf der gewünschten ExaSlang-Ebene bis zur Code-Generierung ausgeführt werden. Eine Web-basierte Oberfläche des Programms ermöglicht unerfahrenen und neuen Nutzern eine einfache Bedienung. Für diese Nutzer würde die Möglichkeit zur handschriftlichen Eingabe der mathematischen Ausdrücke eine weitere Einstiegshürde abbauen. Darüber hinaus würde dies auch für erfahrene Anwender eine Bereicherung darstellen, da die handschriftliche Eingabe in Bezug auf die Geschwindigkeit anderen Varianten überlegen ist.

Nach einem kurzen Überblick über die wichtigsten Grundlagen werden in dieser Arbeit einige bestehende Programme und Ansätze zur Erkennung handschriftlicher mathematischer Ausdrücke untersucht. Darüber hinaus wird ihre Eignung für die Integration in die Benutzeroberfläche des ECGs evaluiert.

2 Allgemeines zur handschriftlichen Eingabe mathematischer Ausdrücke

Die Optical Character Recognition (Optische Zeichenerkennung) (OCR) befasst sich mit dem automatisierten Erkennen und Entziffern von handschriftlichen oder gedruckten Schriftstücken, welche eingescannt als Bitmap vorliegen. Der Inhalt wird in ein von Maschinen gut zu verarbeitendes Format umgewandelt, um eine Weiterverarbeitung zu ermöglichen [Cha13]. Typische Anwendungsbeispiele hierfür sind die automatische Erkennung der Adresse des Empfängers von Briefen oder das Auslesen von Überweisungsformularen [RB10a]. In beiden genannten Fällen können die Buchstaben und Ziffern sowohl handschriftlichen als auch maschinellen Ursprungs sein. Im Gegensatz zur OCR befasst sich die *Handschriftenerkennung* ausschließlich mit der Verarbeitung von Texten und Zeichen handschriftlicher Natur. Dabei können nicht nur eingescannte Schriftstücke verarbeitet werden, sondern die Eingabe kann auch direkt auf einem Touchscreen erfolgen, sodass es in diesem Anwendungsfall zu keinem Medienbruch kommt [KN09]. Die Erkennung von handschriftlichen Zeichen auf digitalen Bildern bildet die Schnittmenge der OCR und Handschriftenerkennung.

2.1 Grundlagen

Es gibt eine Vielzahl verschiedener Verfahren und Ansätze zur Handschriftenerkennung. Generell wird dabei zwischen der *Online-* und der *Offline-Erkennung* differenziert [SR10, ML13, KN09, Á15, PS00].

Bei der *Online-Erkennung* erfolgt die Eingabe meist auf einem Touchscreen mithilfe eines Eingabestiftes oder Fingers. Die so gezeichneten Pfade werden durch den Digitizer des Systems diskretisiert, gespeichert und dienen so als Grundlage für den anschließenden Verarbeitungsprozess [KN09]. Durch diesen Ablauf werden zeitliche Metadaten über den Zeichenvorgang gewonnen. Dazu zählt die Geschwindigkeit, die Reihenfolge und die Zeichenrichtung mit welcher einzelne Strokes bzw. Zeichen geschrieben wurden. Neben den zeitlichen Metadaten können weitere Informationen über den Zeichenvorgang erfasst werden. Dies kann z. B. der Druck oder die Neigung des Stiftes auf dem Eingabegerät sein [SR10, PS00]. Auch wenn es der Name „Online-Erkennung“ suggeriert, muss hierbei die Verarbeitung und Erkennung nicht während oder sofort nach der Benutzereingabe erfolgen. Es ist gut möglich, dass die Schriftzüge in digitaler Form gespeichert werden und erst später ausgewertet werden.

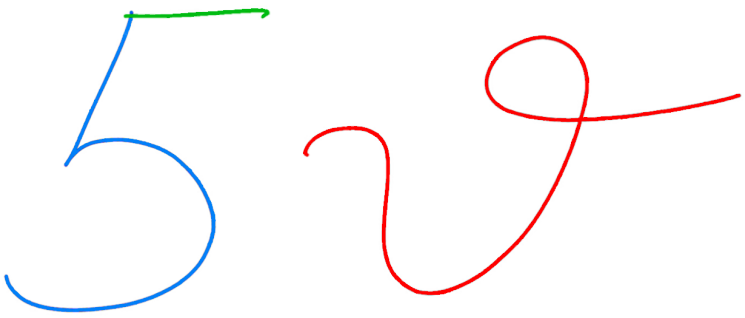


Abbildung 2.1: Farbliche Hervorhebung der einzelnen Strokes

Bei der *Offline-Erkennung* hingegen liegt die Eingabe im Allgemeinen als Bild bzw. Bitmap vor, die durch das Einscannen eines Dokuments (z. B. eines handschriftlichen Briefs) entstanden ist. Die Buchstaben und Symbole wurden dabei auf einem nicht-digitalen Medium (z. B. Papier oder Tafel) geschrieben und erst später digitalisiert [ML13]. Demzufolge stehen keinerlei Metadaten des Schreibprozesses zur Verfügung, die für die Verarbeitung hilfreich sein könnten. Bei der Offline-Erkennung kann die Auswertung immer erst nach dem vollständigen Abschluss der Eingabe erfolgen, wohingegen diese bei der Online-Erkennung schon während des Eingabevorgangs erfolgen kann, wodurch für den Benutzer sofort ersichtlich wird, ob das Geschriebene korrekt erkannt wurde oder nicht [KN09].

Als Grundlage für die Verarbeitung dienen bei der Online-Erkennung, wie in der Definition schon erwähnt wurde, die gezeichneten Pfade. Diese werden als *Strokes* (zu deutsch „Striche“) bezeichnet. Ein Stroke setzt sich aus einer geordneten Sequenz von Punkten zusammen, die vom Digitizer durch die Bewegung des Eingabemediums auf dem Touchscreen aufgezeichnet wurden. Ein Stroke beginnt damit, dass der Stift oder Finger den Touchscreen berührt, und endet, wenn dieser wieder davon abgehoben wird [Á15, DDN16, ZMVG16]. In der Abbildung 2.1 sind die einzelnen Strokes des Ausdrucks „5ϑ“ farblich hervorgehoben. Dabei wird deutlich, dass sich Symbole auch in der Anzahl der Strokes, aus denen sie zusammen gesetzt sind, unterscheiden können. Die Abbildung 2.2 veranschaulicht die digitale Repräsentation des handschriftlichen Ausdrucks aus der vorherigen Grafik, wofür nur die diskreten Punkte der Strokes in einer geordneten Liste gespeichert werden. Dabei ist gut erkennbar, dass die Abtastung nicht aus äquidistanten Punkten bestehen muss. Zusätzlich zu den Koordinaten der Punkte können optional noch Metadaten abgespeichert werden. Die Ink Markup Language (InkML) stellt ein gängiges Format zur Abspeicherung dieser handschriftlichen Eingaben auf einem digitalen Gerät dar. Sie ist eine XML-basierte Auszeichnungssprache, welche vom World Wide Web Consortium (W3C) entwickelt wurde [Wor11]. Listing 2.1 zeigt ein einfaches Beispiel einer InkML-Datei. Jedes „trace“-Element entspricht einem Stroke und speichert diesen als Liste von kartesischen Koordinaten.

Bezogen auf die Erkennungsrate liefern Verfahren der Online-Erkennung meist bessere Ergebnisse als Offline-Verfahren, da bei den letzteren keine zeitlichen Metadaten zur

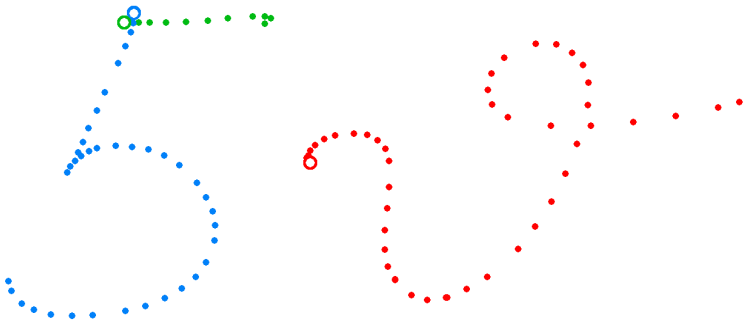


Abbildung 2.2: Digitale Repräsentation der Strokes. Der Anfang jedes Strokes ist mit einem Kreis gekennzeichnet.

Listing 2.1: Beispiel einer InkML-Datei

```
1 <ink xmlns="http://www.w3.org/2003/InkML">
2    <trace id="0">
3       130 155, 144 159, 158 160, 170 154, 179 143, 179 129
4    </trace>
5    <trace id="1">
6       227 50, 226 64, 225 78, 227 92, 228 106, 228 120, 229 134, 230 148,
       234 162, 235 176, 238 190, 241 204
7    </trace>
8    <trace id="2">
9       282 45, 281 59, 284 73, 285 87, 287 101, 288 115, 290 129, 291 143
10   </trace>
11 </ink>
```

Verfügung stehen, welche die Verarbeitung unterstützen würden [SR10]. Ein weiterer Grund dafür ist, dass beim Digitalisierungsprozess vor der Offline-Erkennung Rauschen hinzukommen kann, welches anschließend aufwendig herausgefiltert werden muss. So kann man anschließend nicht mehr sicher sein, dass alle schwarzen Pixel - angenommen der Benutzer hat mit einem schwarzen Stift auf weißem Papier geschrieben - auch wirklich von dem Anwender stammen. Zudem lässt sich mit Online-Verfahren eine deutlich höhere Interaktivität mit dem Nutzer erreichen, da das Geschriebene sofort verarbeitet werden kann. Sollte dann das Erkannte nicht mit dem übereinstimmen, was der Benutzer schreiben wollte, kann dies sofort korrigiert werden [KN09]. Aufgrund der Unterschiede in der Erkennungsrate wird im Rahmen dieser Arbeit verstärkt auf Tools zur Online-Erkennung eingegangen.

2.2 Anforderungen des ExaStencils-Projektes

Ein für das ExaStencils-Projekt adäquates Werkzeug zur Unterstützung der handschriftlichen Eingabe mathematischer Ausdrücke muss alle erforderlichen Symbole und Kon-

Tabelle 2.1: Symbole

Ziffern	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Lateinische Buchstaben (klein)	a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z
Lateinische Buchstaben (groß)	A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z
Griechische Buchstaben (klein)	α, β, γ, δ, ε, ζ, η, θ, ι, κ, λ, μ, ν, ξ, o, π, ρ, σ, τ, υ, ϕ, χ, ψ, ω
Griechische Buchstaben (groß)	A, B, Γ, Δ, E, Z, H, Θ, I, K, Λ, M, N, Ξ, O, Π, P, Σ, T, Υ, Φ, X, Ψ, Ω
Mathematische Zeichen	$+$, $-$, $=$, $\cdot$, $\rightarrow$, $\times$, ∂, ∇, $\sqrt{\ }$, $\int$, ','
Klammern	(,), [,], {, }
Mengenzeichen	$\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, $\mathbb{C}$
Funktionsnamen	sin, cos, arctan, ...

strukte unterstützen. Einschränkungen hierbei wirken sich sehr schnell auf die Benutzerfreundlichkeit und Effizienz aus, da dann für diese Symbole auf alternative Eingabeformen zurückgegriffen werden muss. Aus dem Überblick über die Funktionsweise und Einsatzgebiete des ExaStencils-Projekts in der Einleitung wird ersichtlich, dass die Erkennung von Differentialgleichungen, Funktionen und allgemeinen Gleichungen unterstützt werden muss. Dabei dürfen diese aus allen üblichen mathematischen Symbolen aufgebaut sein.

In der Tabelle 2.1 sind diese Zeichen aufgeführt. Neben den arabischen Ziffern werden auch die lateinischen und griechischen Buchstaben benötigt, jeweils in Groß- und Kleinschreibung. Es fällt auf, dass das Aussehen einiger griechischer und lateinischer Buchstaben übereinstimmt, sodass eine Unterscheidung zwischen diesen nicht möglich [BLL06], aber auch nicht notwendig ist. So sieht z. B. das Omikron genauso aus wie das lateinische „O“. Aus Gründen der Vollständigkeit sind in der Tabelle jeweils die gesamten Alphabete aufgeführt, obwohl bei der Handschriftenerkennung die Duplikate nicht berücksichtigt werden müssen. Zu den genannten Symbolen kommen noch die gebräuchlichen mathematischen Zeichen (hier nur die wichtigsten), sowie Klammern und Mengenzeichen hinzu. Die Funktionsnamen wurden bei dieser Auflistung als Symbole eingeordnet. Allerdings könnten sie auch als Konstrukte betrachtet werden, da es sich um eine Zeichenkette aus mehreren Buchstaben handelt. Bei den aufgeführten Konstrukten in Tabelle 2.2 ist zu beachten, dass diese auch beliebig rekursiv geschachtelt werden dürfen. Ausdrücke wie z. B.

$$f(x) = \frac{\sqrt{x^2}+1}{\frac{1}{\frac{2x}{\sqrt{\pi}}}}$$

dürfen folglich keine Schwierigkeiten bereiten.

Tabelle 2.2: Konstrukte

Exponent	a^b
Index	a_i
Bruch	$\frac{x}{y}$
Wurzel	$\sqrt{n}$
Integral	$\int_a^b f(x)\,dx$
Matrix	$\begin{bmatrix} a_1 & a_2 \\ b_1 & b_2 \end{bmatrix}$

2.3 Schwierigkeiten

Die Erkennung handschriftlicher Zeichen stellt bereits seit langem einen zentralen Forschungsgegenstand der Mustererkennung dar. Das maschinelle Lesen von einzelnen Buchstaben und Ziffern ist bereits weit fortgeschritten und liefert gute Ergebnisse. Dieselbe Entwicklung ist bei der Erkennung handschriftlicher oder gedruckter Texte zu beobachten. Bei mathematischen Formeln ist die Forschung aktuell jedoch noch nicht vergleichbar vorangeschritten, obwohl die ersten Versuche auf diesem Gebiet in die 60er Jahre zurückreichen [ML13, MVGK+11]. Der Grund hierfür ist, dass eine ganze Reihe an Schwierigkeiten zu bewältigen sind.

Die Tatsache, dass mathematische Ausdrücke aus beliebig tiefen Verschachtelungen der genannten Konstrukte bestehen dürfen, stellt ein großes Problem für die Erkennung dar. Bei der klassischen OCR, bei der z. B. der Text auf einer eingescannten Buchseite erkannt werden soll, gibt es dieses Problem nicht. In der Abbildung 2.3 sieht man einen eingescannten, englischen Text, welcher mit einem OCR-Programm verarbeitet wurde. Hier gibt es nur einfache, zeilenweise angeordnete Wörter, was die Texterkennung deutlich

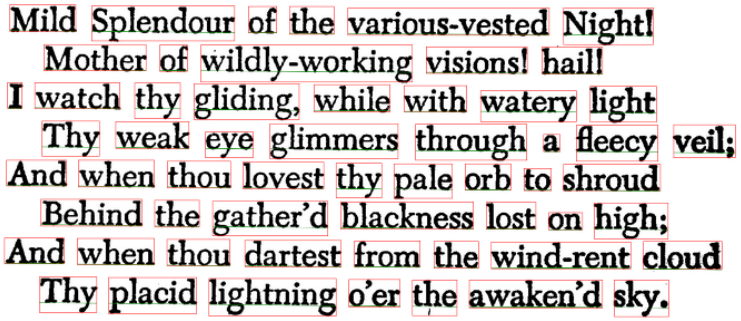


Abbildung 2.3: Erkannte Wörter auf einer eingescannten Buchseite [Pro16]

vereinfacht. In diesem Beispiel überlappen sich zudem die Bounding-Boxes der Wörter nicht, welche durch rote Rechtecke eingezeichnet sind, was zusätzlich zu einer einfacheren Verarbeitung beiträgt. Im Gegensatz dazu handelt es sich bei der Erkennung mathematischer Ausdrücke um ein 2D-Problem, wodurch es zu einer Vielzahl an Unsicherheiten und Mehrdeutigkeiten kommt [Á15, MVGK+12]. Es können zum einen teilweise keine klaren Zeilen erkennbar sein, wie etwa bei:

$$\sum_{i=0}^{N} a_i + \frac{1}{2}$$

Zum anderen können sich Zeichen gegenseitig vertikal betrachtet überlappen, wie an folgendem Ausdruck deutlich wird:

$$\sqrt{\pi}$$

Schon die Tatsache, dass es eine relativ hohe Anzahl zu unterscheidender Zeichen gibt (mehr als 120), stellt ein Problem dar, da die Komplexität eines Systems mit der Anzahl zu erkennender Zeichen zunimmt [SKC08]. In Kombination mit drei weiteren Faktoren wird dies jedoch zu einer ernst zu nehmenden Schwierigkeit für die Handschriftenerkennung.

So gibt es unter den in Tabelle 2.1 aufgeführten Symbolen einige, die sich sehr ähnlich sehen, wie z. B. „x“ und „χ“. Zudem gibt es Zeichen, die die gleiche Form haben, wie z. B. der Buchstabe „O“ und die Ziffer „0“ [Á15]. In Abbildung 2.4 sind zur Verdeutlichung dieses Problems zwei handschriftliche Zeichen sowie einige derer möglichen Interpretationen visualisiert.

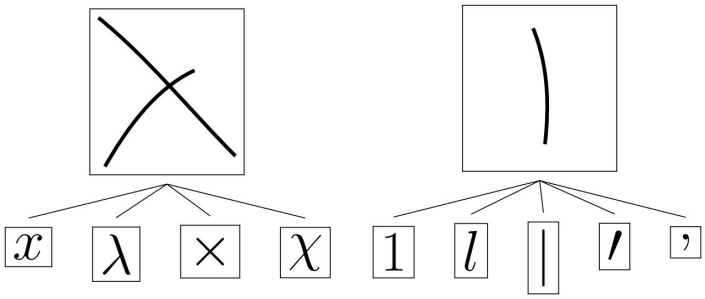
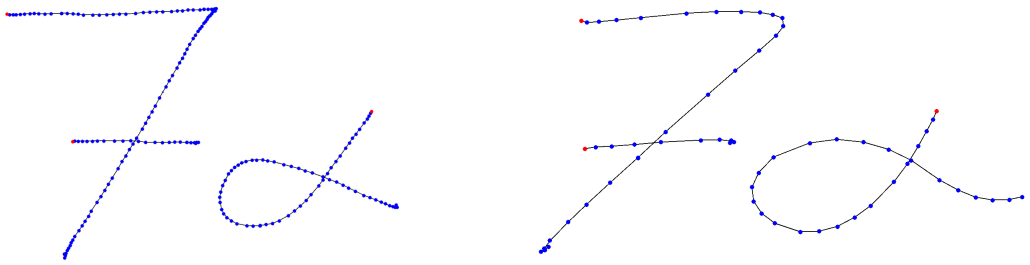


Abbildung 2.4: Zuordnungsproblem für ähnlich aussehende Zeichen [Á15]

Ein weiterer Faktor, der das Problem komplizierter macht, ist, dass die menschliche Handschrift nie exakt gleich aussieht. Selbst wenn dieselbe Person zwei Mal direkt hintereinander den gleichen Text oder die gleiche Formel schreibt, wird es immer kleine, aber für den Computer erkennbare Unterschiede geben. Bei der Online-Erkennung spiegelt sich dies in Abweichungen bezüglich der Anzahl, Form, Größe und Reihenfolge der Strokes sowie der Schreibgeschwindigkeit wieder. Einige dieser Größen, wie z. B. die Anzahl der Strokes, spielen allerdings eine wichtige Rolle bei der Klassifikation eines Zeichens [KN09]. Abbildung 2.5 verdeutlicht in diesem Zusammenhang den Einfluss der Schreibgeschwindigkeit auf Stroke-Ebene.

Neben den zwei genannten Faktoren muss ebenfalls beachtet werden, dass der Schreibstil je nach Nutzer stark variieren kann. Dies beginnt bereits damit, dass manche Personen



(a) Langsam (b) Schnell

Abbildung 2.5: Unterschiedliche Ergebnisse, je nach Zeichengeschwindigkeit (bei gleichem Anwender)

lieber in Druckschrift und andere lieber in Schreibschrift schreiben. Beim letzteren werden die einzelnen Buchstaben bewusst miteinander verknüpft, um den Stift möglichst selten absetzen zu müssen, was zur Folge hat, dass sich die Grenzen zwischen den einzelnen Buchstaben nur sehr schwer detektieren lassen. Bei der Druckschrift setzt sich das Schriftbild aus einzelnen abgesetzten Strichen zusammen, die nicht in einem Fluss gezeichnet werden [KN09]. Letzteres ist für die maschinelle Verarbeitung deutlich besser geeignet. Neben diesen zwei Schriftarten hat jeder Mensch einen mehr oder weniger stark individuellen Schreibstil. In Abbildung 2.6 sind unterschiedliche Schreibweisen des kleinen griechischen Buchstabens Theta (Schreibweisen: ϑ oder θ) von acht verschiedenen Personen zu sehen. Hierbei lassen sich teilweise sehr starke Diskrepanzen erkennen. Dies ist der Grund, weshalb bei Programmen zur Online-Erkennung zwischen benutzerabhängig (writer dependent) und benutzerunabhängig (writer independent) unterschieden wird. Bei benutzerabhängigen Systemen darf die Eingabe nur von der Person stammen, für die es trainiert wurde, wohingegen dies bei benutzerunabhängigen keine Rolle spielt [KN09].

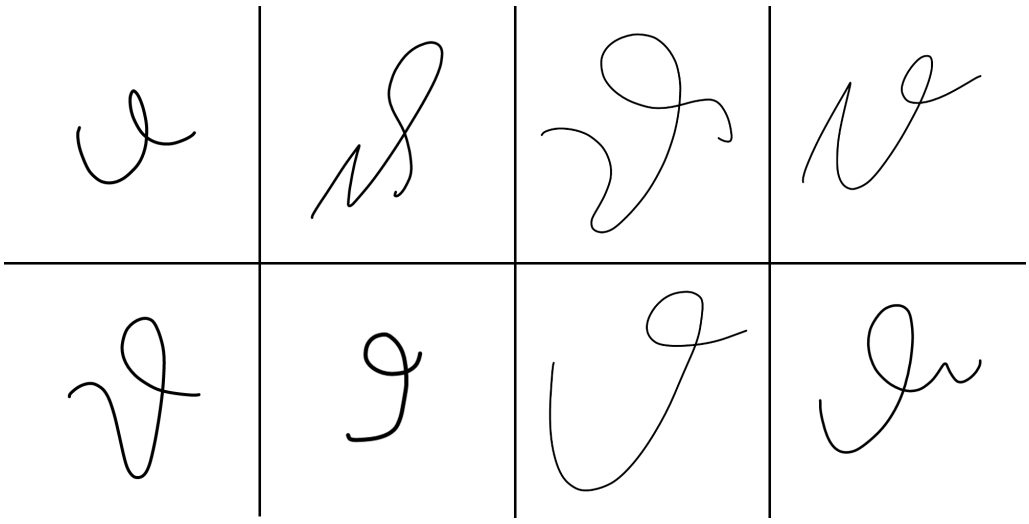


Abbildung 2.6: Unterschiedliche Schreibweisen des kleinen Thetas von verschiedenen Personen

3 Existierende Ansätze

Es gibt aktuell eine Vielzahl verschiedener Ansätze und Implementierungen von Systemen zur Erkennung handschriftlicher mathematischer Ausdrücke. Zudem werden jedes Jahr eine Vielzahl wissenschaftlicher Papiere zu diesem Thema auf Konferenzen und in Fachzeitschriften publiziert [MVGK+11].

Auf dem Markt sind einige kommerzielle Softwarelösungen wie z. B. MyScript[1] oder WIRIS[2] erhältlich, die oft sehr gute Ergebnisse liefern. Die Web-Demo von MyScript[3] erkennt auch komplizierte mathematische Ausdrücke und zeigt die Resultate noch während des Schreibens an. Die von der Firma entwickelten Programme finden in einem breit gefächerten Anwendungsgebiet von Bildung über die Automobilindustrie bis zum Einsatz in Büros Verwendung. Dies zeigt, dass solche Hilfsmittel für die MMK sehr gefragt sind und eine nachhaltige Bereicherung für den Benutzer darstellen. Nachdem diese Programme jedoch leider nicht frei verfügbar sind, scheiden sie für eine weitere Betrachtung aus.

Zur Entwicklung einer Bibliothek zur handschriftlichen Eingabe mathematischer Ausdrücke für das ExaStencils-Projekt wird eine Open-Source-Lösung benötigt. Um auf diesem Gebiet einen guten Überblick über den aktuellen Stand der Forschung und die verfügbaren Anwendungen zu bekommen, lohnt es sich, die CROHME auszuwerten.

3.1 CROHME

Die Competition on Recognition of Online Handwritten Mathematical Expressions wurde zum ersten Mal im Jahr 2011 veranstaltet und findet seitdem, mit Ausnahme des Jahres 2015, jährlich statt. Der Wettbewerb wurde im Rahmen der International Conference on Document Analysis and Recognition (ICDAR) (2011 & 2013) bzw. der International Conference on Frontiers in Handwriting Recognition (ICFHR) (2012, 2014, 2016) ausgetragen, da die meisten Forscher auf dem Gebiet der handschriftlichen Erkennung mathematischer Ausdrücke der Gemeinschaften dieser Veranstaltungen angehören [MVGK+11, MVGZG14, MVGZG16, MVGK+12, MVGZ+13].

Ziel der CROHME ist es, den aktuellen Stand der Forschung auf diesem Gebiet auszuloten und einen Vergleich der Leistungsfähigkeit der verschiedenen Programme zu ermöglichen [ZMVG16]. Es gibt aktuell eine Vielzahl verschiedener Forschungsprojekte, die sich mit dieser Disziplin der Handschriftenerkennung beschäftigen. Jedoch veröffentlichen die meisten Forschungsgruppen Testergebnisse ihrer Tools, welche sie auf Basis eigener, nicht-öffentlicher Datensätze gewonnen haben. Zudem werden die

[1] http://myscript.com/

[2] http://www.wiris.com/

[3] http://webdemo.myscript.com/views/math.html

Datensätze oft nicht zusammen mit den gesammelten Arbeitsergebnissen veröffentlicht. Andere Arbeitsgruppen haben dadurch nicht die Möglichkeit, ihre eigenen Werkzeuge mit diesen Daten zu testen oder die Ergebnisse die publizierten Ergebnisse zu reproduzieren. Aus diesem Grund ist kein direkter Vergleich der Erkennungsprogramme auf Basis ihrer publizierten Testergebnisse möglich. Genau bei diesen Problemen soll die CROHME Abhilfe schaffen [MVGK+11, MVGK+12].

Der Wettbewerb gliederte sich ursprünglich in zwei bzw. später drei übergeordnete Disziplinen. Bei den ersten drei Austragungen gab es die Disziplinen „Isolated Symbol Recognition“ (also die Erkennung einzelner, isolierter Symbole) und „Mathematical Expression Recognition“. Ab 2014 wurde zusätzlich als neue Herausforderung die „Matrix Recognition“ eingeführt [MVGZG14]

Für die Teilnehmer der CROHME gibt es jeweils einheitliche Testdaten[4] [5], welche zudem öffentlich zugänglich sind. Zusätzlich werden Trainingsdaten bereitgestellt, welche jedoch nicht verpflichtend verwendet werden müssen. So gibt es immer wieder Teams, die ausschließlich oder zusätzlich dazu private Trainingsdaten verwenden [MVGZG16]. Im Laufe der Jahre wurden die Datensätze immer umfangreicher. So bestand das Testset für die Formelerkennung beim ersten Wettbewerb noch aus 348 Ausdrücken, wohingegen es 2013 schon 671 und 2016 schließlich über 1000 waren. Die bereitgestellten Trainingsdaten wuchsen in noch größerem Umfang von anfangs 921 auf 8836 Ausdrücke im Jahr 2016 [MVGZG14, MVGZG16].

Die Trainings- und Testdaten liegen im InkML-Format vor und enthalten zudem Mathematical Markup Language (MathML)-Annotationen, die es erlauben, die einzelnen Strokes den Symbolen in den mathematischen Ausdrücken zuzuordnen. Um einen automatisierten Trainingsprozess zu ermöglichen, sind in den Dateien der Trainingsdaten die repräsentierten Formeln zusammen mit der Segmentierung gespeichert. Die Segmentierung legt fest, welche Gruppen von Strokes zusammen gehören und ein Zeichen repräsentieren. In Abbildung 3.1 und Abbildung 3.2 sind einige Beispiele für Formeln und Matrizen aus dem CROHME-Testset zu sehen. Die enthaltenen mathematischen Ausdrücke unterliegen einer bestimmten Grammatik, d. h. es sind nicht alle beliebigen Formeln und Gleichungen möglich. Bei der ersten CROHME bestand die Grammatik dafür aus 57 Terminalsymbolen, darunter Ziffern, einige lateinische und griechische Buchstaben, Funktionsnamen (tan, log) und mathematische Symbole. Die Ausdrücke durften aus Summen, Produkten, Funktionen, Brüchen, Wurzeln, Exponenten, Indizes, etc. bestehen und es gab keine Einschränkungen bezüglich der Rekursionstiefe [MVGK+11]. Die Grammatik für diesen Teil des Wettbewerbs wurde genau wie die Trainings- und Testdaten einige Male erweitert. Bei der dritten CROHME wurden weitere Zeichen, hauptsächlich lateinische und griechische Buchstaben, aufgenommen, sodass die Grammatik insgesamt 101 Terminalsymbole umfasste [MVGZ+13]. Bei einem Vergleich dieser Vorgaben mit den in Kapitel 2.2 beschriebenen Anforderungen fällt auf, dass diese bis auf einige kleine Unterschiede übereinstimmen. Somit wird deutlich, dass die CROHME einen guten Ausgangspunkt für die Suche nach einer geeigneten Erkennungssoftware darstellt.

[4] http://www.isical.ac.in/~crohme/CROHME_data.html
[5] http://tc11.cvc.uab.es/datasets/CROHME-2014_2

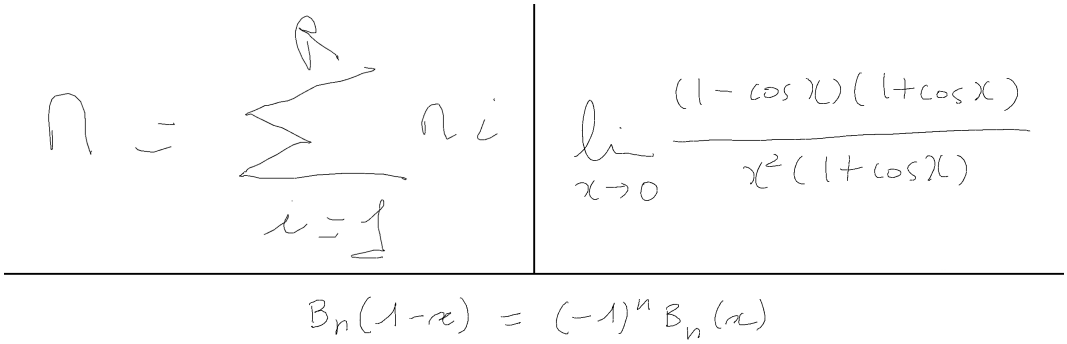


Abbildung 3.1: Beispiele für Formeln aus dem CROHME-Testset 2012

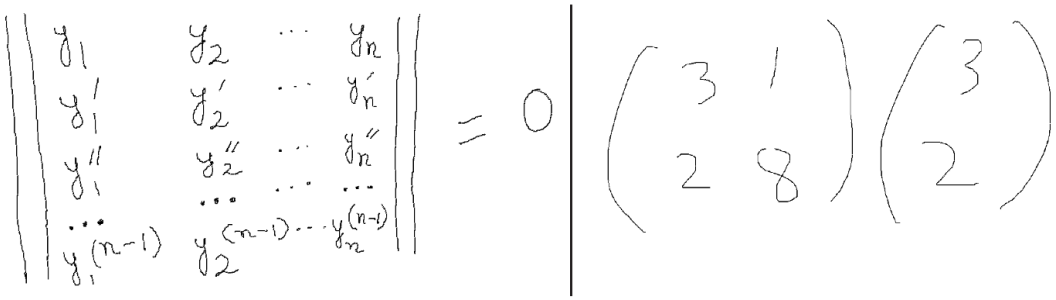


Abbildung 3.2: Beispiele für Matrizen aus dem CROHME-Testset 2014 [MVGZG14]

In der Tabelle 3.1 sind alle teilnehmenden Teams aufgeführt. Seit Beginn des Wettbewerbs traten insgesamt elf verschiedene Teams gegeneinander an, wobei Vision Objects und MyScript dabei zusammen als nur ein Team gezählt wurden, da sich das erste Team 2014 zu MyScript umbenannte [MVGZG14]. Jedes eingereichte Programm unterstützte mindestens die Erkennung einzelner, isolierter Symbole, weshalb dies in der Tabelle 3.1 nicht gesondert aufgeführt ist. An der Disziplin der Formelerkennung (FE) nahmen immer alle Teilnehmer, mit Ausnahme des ILSP/Athena Research and Innovation Center bei der CROHME 2011 & 2014 sowie dem RIT 2016, teil. An die im Zuge des vierten Wettbewerbs eingeführte Matrixerkennung (ME) wagten sich insgesamt nur drei Teams heran, von denen zwei kommerzielle Absichten verfolgen [MVGZG14, MVGZG16]. Zu jedem Teilnehmer ist in Tabelle 3.1 zudem aufgeführt, ob der Programmcode unter einer Open-Source-Lizenz veröffentlicht wurde. Bei vielen Teilnehmern ließ sich leider weder Quelltext noch ein Hinweis darauf finden, ob sie die Absicht haben, diesen gemeinfrei zu veröffentlichen oder zu kommerzialisieren, was mit einem Fragezeichen gekennzeichnet ist. Das Kreuz hingegen bedeutet, dass die Software sicher zu ökonomischen Zwecken genutzt wird. So ist es möglich, auf den Webseiten von MyScript und WIRIS Lizenzen für ihre Programme zu kaufen [MyS, Mat].

In der folgenden Tabelle 3.2 sind die Erkennungsraten der Formelerkennung aufgeführt. Die Rahmenbedingungen für diese Disziplin wurden mit der Zeit zunehmend schwieriger, da über die Jahre neue Zeichen und kompliziertere Ausdrücke zugelassen wurden. Für die in der Tabelle 3.2 aufgelisteten Ergebnisse wurde stets die herausfordernste Teildiszi-

Tabelle 3.1: CROHME Teilnehmerübersicht

Team	'11	'12	'13	'14	'16	FE	ME	Open Source
Vision Objects		x	x			✓	✗	✗
MyScript				x	x	✓	✓	✗
WIRIS					x	✓	✓	✗
RIT	x	x	x	x	x	(✓)	✗	✓
Tokyo University			x	x	x	✓	✗	?
University Nantes	x	x	x	x	x	✓	✗	?
University Sao Paulo			x	x	x	✓	✗	?
University Valencia	x	x	x	x		✓	✓	✓
Athena RIC	x	x		x		(✓)	✗	?
Czech University			x			✓	✗	?
Sabanci University	x	x	x			✓	✗	?
University Waterloo		x				✓	✗	?
Σ Teilnehmer	5	7	8	7	6			

Tabelle 3.2: CROHME - Erkennungsrate für Formeln (in %)

Team	'11	'12	'13	'14	'16
Vision Objects/MyScript		62,50	60,36	62,15	67,65
WIRIS					49,61
RIT	2,59	9,43	14,31	19,52	
Tokyo University			19,97	25,04	43,94
University Nantes	22,41	25,61	18,33	17,59	13,34
University Sao Paulo			9,39	15,05	33,39
University Valencia	19,83	22,75	23,40	30,70	
Athena RIC		3,69			
Czech University			2,68		
Sabanci University	0,29	4,92	8,35		
University Waterloo		40,16			

plin betrachtet. Damit lässt sich erklären, weshalb z. B. Vision Objects 2013 schlechter abschnitt als im Jahr zuvor.

Tabelle 3.3 listet die Ergebnisse der Matrixerkennung auf. Die deutlich geringere Teilnehmerzahl hier lässt sich dadurch erklären, dass diese Aufgabe deutlich schwieriger ist als die vorher betrachtete Erkennung einfacher Formeln oder Gleichungen. Eine Matrix kann als Ausdruck mit tabellarisch angeordneten Subausdrücken betrachtet werden. Für die Erkennung muss hierbei zunächst festgestellt werden, dass eine Matrix vorliegt und anschließend müssen die Zeilen und Spalten korrekt detektiert werden, sodass die Inhalte der einzelnen Zellen (wiederum eigenständige mathematische Ausdrücke) geparst werden

Tabelle 3.3: CROHME - Erkennungsrate für Matrizen (in %)

Team	**'14**	**'16**
Vision Objects/MyScript	53,28	68,40
WIRIS		56,40
University Valencia	31,15	

können [MVGZG14].

Die Zahlen aus Tabelle 3.2 und Tabelle 3.3 stellen nicht die einzigen Resultate des Wettbewerbs zur automatischen Handschriftenerkennung dar. Neben ihnen wurde z. B. auch ausgewertet, wie viele einzelne Zeichen eines Ausdrucks richtig erkannt wurden oder in wie vielen Fällen die Segmentierung (ohne die Erkennung) korrekt war.

Auf der Suche nach einer geeigneten Lösungen für das ExaStencils-Projekt scheiden schon aufgrund des essentiellen Kriteriums, dass der Quelltext Open Source sein muss, alle außer zwei Arbeitsgruppen aus. Es verbleiben nur die Teams der Universitat Politècnica de València (UPV) und des RIT. Dies ist ein sehr mageres Ergebnis, wenn man die hohe Zahl an Teilnehmern des Wettbewerbs bedenkt. Das Programm der UPV unterstützt, anders als das des RIT, auch die Matrixerkennung und liefert bezüglich der Erkennungsrate deutlich bessere Ergebnisse. Die Erkennungsraten waren hier immer mindestens 50% und anfangs sogar um über 650% (2,59% im Vergleich zu 19,83%) besser.

Im Folgenden werden die zwei genannten Implementierungen genauer evaluiert, wobei aufgrund der genannten Vorteile ein besonderer Fokus auf das Programm der UPV gelegt werden soll.

3.2 Evaluierung bestehender Programme

Nachdem aus der Analyse der CROHME nur zwei Tools hervorgingen, bei denen sich eine weitere Untersuchung lohnt, wurde nach weiteren Kandidaten gesucht. Zudem bezog sich die CROHME, wie der Name schon sagt, nur auf die Erkennung von handschriftlichen, mathematischen Ausdrücken, die als Online-Daten vorliegen. Dementsprechend galt es, auch Ausschau nach Programmen für die Offline-Erkennung zu halten. Das Programm Seshat, welches von der UPV entwickelt wurde und den Favoriten des vorherigen Kapitels darstellt, baut zu großen Teilen auf RNNLIB auf, weshalb dieses System in der Evaluierung zuerst betrachtet wird.

3.2.1 RNNLIB

RNNLIB ist eine Bibliothek zur Lösung von Sequence-Learning-Problemen, wie Sprach- oder Handschriftenerkennung, welche auf rekurrenten neuronalen Netzen aufbaut [Gra]. Alex Graves entwickelte diese Software im Rahmen seiner Dissertation [Gra14] an der Technische Universität München (TUM), welche er im Januar 2008 fertigstellte. Der

Quelltext von RNNLIB wurde unter der GNU General Public License, version 3.0 (GPLv3) veröffentlicht und kann auf der Plattform Sourceforge[6] heruntergeladen werden.

Neben der modernen Long Short Term Memory (LSTM)-Architektur werden auch andere, traditionellere Konzepte neuronaler Netze implementiert. LSTM ist eine Architektur für rekurrente neuronale Netze (RNN), die eine bessere Speicherung und einen leichteren Zugriff auf Daten ermöglicht als normale RNN [Gra14, Gra13]. Auf die genaueren Konzepte hinter den neuronalen Netzen wird an dieser Stelle jedoch nicht eingegangen, da sie für die Funktionalität nicht von primärem Interesse sind.

Eine große Stärke der Bibliothek ist ihre Flexibilität. Neben der Sprach- und Handschriftenerkennung wurde RNNLIB unter anderem auch für die Klassifikation von Bildern, Erkennung von Gesichtsausdrücken, Verifikation von Unterschriften und Objekterkennung eingesetzt. Dabei ist es möglich, dass die Eingabedaten in verschiedenen Formaten, wie z. B. Bilddateien oder unverarbeiteten Sensordaten, vorliegen [Gra].

Im Bezug auf die Handschriftenerkennung kann RNNLIB sowohl für die Online- als auch für die Offline-Erkennung verwendet werden [Gra14]. Bei der Offline-Erkennung arabischer Texte lieferte die Software beispielsweise sehr gute Ergebnisse [GS09]. Die veröffentlichte Version des Programms unterstützt zudem ein individuelles Training des neuronalen Netzes, sodass dieses auch für andere Sprachen und Zeichen trainiert werden kann. Allerdings ist das Tool sehr generisch gehalten, weshalb es für die komplexe Aufgabe der Erkennung mathematischer Ausdrücke alleine nicht gut eignet ist und eine Realisierung mit viel Aufwand verbunden wäre. Das spezialisierte Programm Seshat, welches im Folgenden untersucht wird, verwendet RNNLIB intern für die Klassifikation von Zeichen. Allerdings kombiniert es diese mit domänenspezifischem Wissen, was die Ergebnisse deutlich verbessert. Ein weiterer negativer Aspekt der Bibliothek ist, dass der veröffentlichte Quelltext seit 2013 nicht mehr weiterentwickelt bzw. gepflegt wurde. Darüber hinaus gibt es nur eine kurze Anleitung zur Verwendung der Software, jedoch keine umfassende Dokumentation.

3.2.2 Seshat

Das bereits angesprochene Programm Seshat wurde von Francisco Álvaro Muñoz im Rahmen seiner Dissertation [Á15] im Jahr 2015 an der Universitat Politècnica de València entwickelt. Es handelt sich um ein Werkzeug zur Online-Erkennung handschriftlicher mathematischer Ausdrücke und nahm sehr erfolgreich an der CROHME teil. Bei dem Wettbewerb 2011 gewann es den ersten Platz (das Team der University of Nantes blieb bei der Wertung außen vor, da der Wettbewerb von dieser Universität veranstaltet wurde). In den Jahren 2013 und 2014 gewann das Programm wieder den ersten Platz, allerdings diesmal in der Wertung mit allen Teams, die für das Training nur die von CROHME bereit gestellten Datensätze verwendeten. In der Gesamtwertung über alle Teams wurde es in den beiden Jahren Zweiter [FL, MVGK$^+$11, MVGZ$^+$13, MVGZG14]. Zudem unterstützt Seshat viele der in Abschnitt 2.2 genannten Anforderungen. Neben 92 von 120 Symbolen werden alle Konstrukte, darunter auch Matrizen, unterstützt. Dies ist bei keinem anderen

[6] https://sourceforge.net/projects/rnnl/

Open-Source-Programm, welches an der CROHME teilnahm, der Fall. Aufgrund dieser positiven Aspekte wird auf Seshat im Folgenden genauer eingegangen.

Funktionsweise

Die Erkennung mathematischer Ausdrücke wird von Seshat, wie es auch bei anderen Lösungen üblich ist, in drei Schritte zerlegt. Diese sind: Symbolsegmentierung (symbol segmentation), Symbolerkennung (symbol recognition) und Strukturanalyse (structural analysis) [Á15, ÁSB14, HZ13]. Die Symbolsegmentierung versucht herauszufinden, welche Strokes zusammen gehören, damit anschließend im zweiten Schritt der Symbolerkennung das repräsentierte Symbol erkannt werden kann. Ein mathematischer Ausdruck setzt sich meist aus mehreren Zeichen zusammen, die in unterschiedlichen Beziehungen zueinander stehen. Die Strukturanalyse soll diese teilweise komplexen Beziehungen herausfinden. Alle drei Schritte hängen wechselseitig voneinander ab, sodass es nicht möglich ist, sie wie in einer Pipeline hintereinander abzuarbeiten. Das Verwenden von Informationen über den Kontext eines Strokes oder Zeichens kann in allen Teilen des Prozesses zu einer deutlichen Verbesserung der Ergebnisse beitragen. Beispielsweise wird durch die Kenntnis des vorhergehenden und nachfolgenden Symbols eine Unterscheidung zwischen „x“ und „$\times$“ deutlich erleichtert. Der verwendete Ansatz versucht die drei grundlegenden Schritte simultan zu optimieren. Neben den Kontextinformationen werden zur Unterstützung auch kontextfreie Grammatiken, nach denen mathematische Ausdrücke gebildet werden, herangezogen [Á15].

An dieser Stelle folgt eine kurze Beschreibung der konkreten Umsetzung. Für die Symbolerkennung wird eine Kombination aus Online- und Offline-Features verwendet, da dies signifikant bessere Ergebnisse liefert als eine Einzelne der beiden Varianten [ÁSB14]. Die Online-Features werden direkt aus den gespeicherten Punkten eines Strokes berechnet. Zur Bestimmung der Offline-Features wird aus den Strokes ein Bild mit vordefinierter Höhe generiert, welches dann als Grundlage für die Berechnung der Features dient. Ein Bidirectional Long Short Term Memory (BLSTM)-RNN wurde jeweils für beide Feature-Mengen trainiert. Für die Klassifikation der Zeichen werden die Ergebnisse anschließend kombiniert. Der Parser stellt zunächst einige Hypothesen für die Symbolsegmentierung auf Stroke-Ebene auf. Hierfür werden verschiedene stochastische Modelle und die Klassifikation der Zeichen herangezogen. Im nächsten Schritt werden neue Hypothesen durch Kombination von Teilproblemen aufgestellt, die zudem die Strukturanalyse miteinbeziehen. Dies wird so lange fortgeführt, bis der gesamte Ausdruck betrachtet wird. Von diesen Hypothesen wird diejenige ausgewählt, welche die höchste Wahrscheinlichkeit besitzt [MVGZG14].

Implementierung

Die Implementierung von Seshat, welche komplett in C++ geschrieben wurde, ist als Open Source unter der GPLv3 auf Github[7] verfügbar [ÁSB16]. Als RNN wurde die zuvor vorgestellte RNNLIB Bibliothek verwendet. Weitere Abhängigkeiten sind die Boost C++

[7] `https://github.com/falvaro/seshat`

Libraries[8] und die Xerces-C++ Library[9] zum Parsen der Eingabe, welche im InkML- oder im SCGINK-Format vorliegen darf [Áb].

Das Programm wurde so entworfen, dass eine Anpassung an andere Anforderungen ohne Änderungen am Quelltext möglich ist. Die Listen mit den unterstützen Symbolen sind wie die verwendeten Grammatiken in Konfigurationsdateien gespeichert. Auch die Parameter für die neuronalen Netze sind in solchen Dateien gespeichert. Die bei der CROHME 2014 eingesetzte Konfiguration, welche die Erkennung von Matrizen unterstützt, wurde allerdings nicht veröffentlicht. Auch bei der Online-Demonstration[10] sind keine Matrizen als Eingabe erlaubt. Aufgrund des Designs ließe sich dieses Problem genau wie die fehlende Unterstützung für manche anderen benötigten Symbole eigentlich leicht beheben. Allerdings gibt es so gut wie keine Dokumentation zum Aufbau der Konfigurationsdateien und es ist teilweise nicht möglich, sich diesen selbst zu erschließen. Dazu kommt, dass an keiner Stelle beschrieben wird, wie und mit welchen Parametern RNNLIB genau trainiert wurde. Dies macht es sehr schwierig, die Konfiguration von Seshat anzupassen. Neben diesem Aspekt gibt es zudem noch eine Reihe weiterer Schwächen:

1. Abgesehen von einer kurzen Anleitung zum Kompilieren und Ausführen des Programms gibt es keine Dokumentation. Zudem enthält der von Francisco Álvaro Muñoz geschriebene Quellcode kaum Kommentare, was es sehr schwierig gestaltet, diesen nachzuvollziehen.

2. Seit Ende 2015 gab es keine Änderungen mehr an dem öffentlich zugänglichen Quelltext. Dies deutet zum einen darauf hin, dass es keine weitere Unterstützung für das Programm gibt. Zum anderen führt es dazu, dass beim Übersetzen Fehler auftreten, wenn man einen aktuellen Kompiler verwendet und aktuelle Versionen der Abhängigkeiten installiert hat. Durch kleine Änderungen am Programmcode lässt sich dieses Problem allerdings beheben.

3. Die verwendeten Bibliotheken erschweren es zusätzlich das Programm für andere Systeme (speziell Android) zu portieren. Dieser Vorgang war nur mit erheblichem Aufwand und Änderungen am Quelltext möglich.

4. Am Gravierendsten ist jedoch die geringe Qualität der Implementierung, was ein Test mit dem Analyse-Werkzeug Valgrind[11] bestätigt. Abbildung 3.3 zeigt das Resultat der Analyse. Neben uninitialisierten Sprungvariablen wurden insgesamt über 45 Millionen „errors“ detektiert und in Summe verursachte das Programm ein Speicherleck von ca. 2,4 MB.

Die genannten Punkte machen deutlich, dass ein umfassendes Refactoring der Implementierung nötig wäre, bevor Seshat in eine qualitativ hochwertige Bibliothek zur Erkennung handschriftlicher mathematischer Ausdrücke integriert werden könnte.

[8] http://www.boost.org/
[9] http://xerces.apache.org/xerces-c/
[10] http://cat.prhlt.upv.es/mer/
[11] http://valgrind.org/

```
LaTeX:
\tan ( \frac{\pi}{4} ) = 1
==3037==
==3037== HEAP SUMMARY:
==3037==     in use at exit: 2,585,006 bytes in 1,209 blocks
==3037==   total heap usage: 1,089,894 allocs, 1,088,685 frees, 213,774,897 bytes allocated
==3037==
==3037== LEAK SUMMARY:
==3037==    definitely lost: 75,432 bytes in 402 blocks
==3037==    indirectly lost: 2,436,318 bytes in 805 blocks
==3037==      possibly lost: 0 bytes in 0 blocks
==3037==    still reachable: 73,256 bytes in 2 blocks
==3037==         suppressed: 0 bytes in 0 blocks
==3037== Rerun with --leak-check=full to see details of leaked memory
==3037==
==3037== For counts of detected and suppressed errors, rerun with: -v
==3037== Use --track-origins=yes to see where uninitialised values come from
==3037== ERROR SUMMARY: 45821337 errors from 401 contexts (suppressed: 0 from 0)
```

Abbildung 3.3: Ausgabe von Valgrind bei der Ausführung von Seshat mit einer der mitgelieferten Beispielformeln

3.2.3 RIT

Das Team des Rochester Institute of Technology (RIT) war bei der CROHME neben den Entwicklern von Seshat das Einzige, welches seinen Quelltext auch als Open Source veröffentlichte. Das aktuellste Programm heißt DPRL CROHME 2014 (DPRL14), wobei DPRL als Abkürzung für „Document and Pattern Recognition Lab“ steht, und ist unter der GPLv3 auf Github[12] erhältlich. Auch die Versionen der Jahre 2011, 2012 und 2013 können dort heruntergeladen werden [HDÁZ]. Im Folgenden wird allerdings nur auf die neueste Version eingegangen, da diese die besten Ergebnisse erzielt und am weitesten entwickelt ist.

Funktionsweise

DPRL14 ist ein Programm zur Online-Erkennung handschriftlicher mathematischer Ausdrücke. Die Verarbeitung gliedert sich dabei wie bei Seshat in drei Schritte: Symbolsegmentierung, Symbolerkennung und Strukturanalyse. Für die Symbolsegmentierung werden die Strokes in der zeitlichen Reihenfolge ihrer Entstehung betrachtet. Es wird also angenommen, dass ein Zeichen immer nur aus zeitlich aufeinander folgenden Strokes besteht. Ein handschriftlicher Ausdruck wie in Abbildung 3.4 wird somit ausgeschlossen. Eine Einschränkung bezüglich der Anzahl der Strokes, aus denen sich ein Zeichen zusammensetzt, gibt es nicht. Für die Segmentierung werden bei N Strokes alle Möglichkeiten betrachtet, zeitlich konsekutive Strokes in einer Gruppen der Größe 1 bis N zusammenzufassen, was zu einer Komplexität von $\mathcal{O}(N^2)$ führt. Mithilfe eines AdaBoost-Klassifikators wird bestimmt, welche Strokes zusammengehören [HZ13, HDÁZ].

Für die anschließende Symbolerkennung wird eine Kombination aus Online- und Offline-Features verwendet. Die in den verarbeiteten Online-Daten enthaltenen Metadaten über den Verlauf der Zeichengeschwindigkeit werden hierbei verworfen, da sie von Kenny Davila et al. aufgrund ihrer hohen Varianz bei unterschiedlichen Schreibstilen

[12] https://github.com/DPRL/CROHME_2014

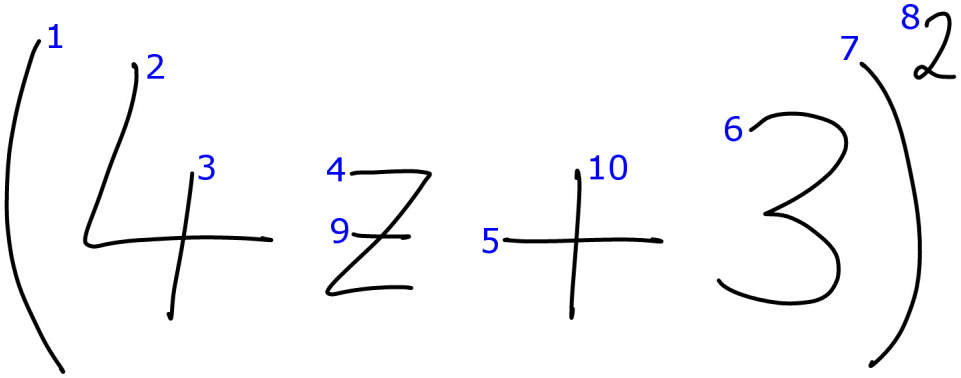


Abbildung 3.4: Beispiel für einen Ausdruck, der Schwierigkeiten bereiten würde, da „z“ und „+“ nicht aus zeitlich aufeinander folgenden Strokes bestehen. Die Nummern geben die Reihenfolge der Entstehung der Striche an.

als unbrauchbar angesehen werden [DLZ14]. Die mit den Online-Features, welche z. B. die normalisierte Länge des Pfades und die Anzahl der Strokes enthalten, kombinierten Offline-Features werden für die Klassifikation durch eine Support Vector Machine (SVM) verwendet [MVGZG14].

Bei der Strukturanalyse werden fünf verschiedene Typen räumlicher Beziehungen unterschieden: horizontal (AB), Index (A_B), Exponent (A^B), untereinander ($\frac{A}{B}$) und ineinander ($\sqrt{B}$, wobei A durch $\sqrt{\ }$ repräsentiert wird). A und B stellen dabei jeweils einen Teilausdruck dar. Der Parser arbeitet rekursiv nach dem Top-Down-Ansatz. Die Klassifikation findet jeweils auf Basis geometrischer Features statt [AZ13, HDÁZ].

Implementierung

Bei der Implementierung entschied sich das Team des RIT für eine Kombination aus C++ und Python. Im Hinblick auf eine Erweiterung oder Integration von DPRL14 in eine für unsere Zwecke adäquate Bibliothek ergeben sich bei genauerer Untersuchung einige Probleme. Der veröffentlichte Quelltext auf Github wurde seit Anfang 2014 nicht mehr verändert, was zum einen ein klares Indiz dafür ist, dass keine Weiterentwicklung des Programms mehr stattfindet. Zum anderen führt es dazu, dass mit aktuellen Versionen der verwendeten Python-Module Fehler auftreten. Darüber hinaus gibt es keine Dokumentation des Programms und der Quellcode selbst ist ebenfalls nur äußerst spärlich kommentiert, was es sehr kompliziert gestaltet, diesen anzupassen. Deutlich schwerer wiegt, dass es keine Informationen dazu gibt, wie die Konfigurationsdateien für die verschiedenen Klassifikatoren erzeugt wurden, sodass es fast nicht möglich ist, DPRL14 um weitere mathematische Symbole oder Konstrukte zu erweitern. Da die Software jedoch nur die Zeichen der CROHME unterstützt, wäre eine Erweiterung des Zeichensatzes unumgänglich.

Neben den genannten Nachteilen spricht auch die relativ schlechte Erkennungsrate von 19,52% bei der CROHME 2014 gegen eine Verwendung des Programms des RIT.

3.2.4 Caffe

Caffe ist ein namenhaftes Deep-Learning-Framework, welches 2014 vom Berkeley Vision and Learning Center (BVLC) entwickelt wurde. Es ist komplett in C++ implementiert und verfügt über Application Programming Interfaces (APIs) und Benutzerschnittstellen in Python sowie MATLAB, welche die Bibliothek leicht verwendbar und flexibel einsetzbar machen. Mithilfe von Caffe können generische Convolutional Neural Networks (CNNs) und andere Deep-Learning-Modelle effizient trainiert und eingesetzt werden [JSD+14]. Der gesamte Quelltext ist als Open Source unter der BSD 2-Clause license (BSD2) auf Github[13] verfügbar. Zudem gibt es eine gute Dokumentation und einige Tutorials zu seiner Verwendung auf der Website der Berkley Universität[14].

Aufgrund der effizienten Implementierung, die sich wahlweise auf der CPU oder GPU ausführen lässt, eignet sich Caffe auch für industrielle Anwendungen [JSD+14]. Facebook verwendet die Caffe-Bibliothek zur Erkennung der dargestellten Szene auf den von Nutzern hochgeladenen Fotos. Die Beschreibung kann von Personen mit Sehbehinderung als automatischer Alternativtext abgerufen werden und ermöglicht es ihnen so, den Inhalt des Bildes nachzuvollziehen, womit ein Beitrag zur Barrierefreiheit geleistet wird. Auch andere Unternehmen, wie z. B. Pinterest, setzen auf diese Open-Source-Bibliothek [SDL].

Im April 2017 veröffentlichte Facebook Caffe2, welche eine eigene weiter entwickelte Version des ursprünglichen Frameworks darstellt [Fac]. Im Vergleich zum Vorgänger ist Caffe2 leichtgewichtiger und besser skalierbar [NG17]. Bezüglich der API und den grundlegend verwendeten Techniken gab es jedoch keine Änderungen. Obwohl Caffe2 von Facebook betreut und weiter entwickelt wird, ist der Quelltext unter der Apache License, version 2.0 (ALv2) verfügbar[15].

Als Vorteile dieser Deep-Learning-Bibliothek sind die umfangreiche Dokumentationen und leichte Verwendbarkeit hervorzuheben. Darüber hinaus lässt sich das Programm, dank guter Anleitung für alle üblichen Betriebssysteme, leicht installieren. Zudem gibt es einige Anleitungen für erste Projekte zur Einarbeitung [Fac17].

Caffe und sein Nachfolger sind primär für maschinelles Sehen (Machine Vision) konzipiert und eignen sich weniger für die Text- oder Spracherkennung [NG17]. Das Ziel von Machine-Vision-System ist es, automatisiert Informationen über die auf Bildern dargestellte Szene zu sammeln, um damit ein Modell der realen Welt erstellen zu können [JKS95]. Damit lassen sich z. B. folgende Aufgaben lösen [SDL]:

- **Generelle Klassifikation:** Bestimmung, um was für einen Typ von Bild es sich handelt (Landschaftsaufnahme, Porträt, etc.) und welche Arten von Objekten zu sehen sind.
- **Detektion der Objekte:** Erkennung der Dinge, die auf dem Foto abgebildet sind, und wo sie sich befinden.

Auch wenn die vom BVLC entwickelte Bibliothek nicht schwerpunktmäßig für die Texterkennung konzipiert ist, gibt es dennoch einige Programme, die Caffe für die OCR

[13] https://github.com/BVLC/caffe
[14] http://caffe.berkeleyvision.org/
[15] https://github.com/caffe2/caffe2

einsetzen[16]. Diese Anwendungen gehen jedoch nicht über die Erkennung einzelner Zeichen oder Textzeilen hinaus. Somit ist ein Einsatz von Caffe bzw. Caffe2 als Werkzeug zur handschriftlichen Erkennung mathematischer Offline-Ausdrücke (aktuell) nicht geeignet. Eventuell ließe sich mit der Bibliothek ein solches Programm realisieren, was jedoch mit großem Aufwand vor allem in Bezug auf die Segmentierung der Formeln und Gleichungen verbunden wäre.

3.2.5 Tesseract

Die Entwicklung von Tesseract begann 1984 als PhD-Projekt von Ray Smith [Smi87] bei HP Labs, welche das damals noch proprietäre Programm bis 1994 weiter entwickelten. Tesseract dient zur Offline-Erkennung maschinell geschriebener Texte und war ursprünglich als mögliches Software-Add-On für die Scanner von HP gedacht [Smi07]. 2005 veröffentlichten das Unternehmen in Kooperation mit der University of Nevada den Quelltext, womit dieser frei zugänglich wurde. Seitdem wird er von Google weiterentwickelt und mitfinanziert [Smi16, RB10a].

Die Verarbeitung eines eingescannten Dokumentes gliedert sich in mehrere Schritte. An den ersten Schritt der Vorverarbeitung (Anwenden von Filtern, etc.) schließt sich die Segmentierung des Bildes, also das Auffinden der Textzeilen an. Dieser Prozess stellt bei der OCR allgemein eine große Schwierigkeit dar. Das 1984 entwickelte Programm verwendet hierfür einen mehrstufigen Ablauf, der zunächst größere Blöcke (Blobs) findet, die dann weiter unterteilt werden, bis eine Segmentierung in einzelne Wörter bzw. Buchstaben vorliegt. Für die eigentliche Erkennung der Zeichen dienen dann die Umrisse der Buchstaben. Um dabei gute Ergebnisse zu erzielen, werden auf Wortebene Sprachmodelle wie Wörterbücher verwendet [Smi07].

Anfangs unterstütze Tesseract nur die Erkennung eingescannter, maschinell erstellter Schriftstücke, die aus einer einzigen Textspalte (also mehrere untereinander angeordnete Textzeilen gleicher Länge) bestehen. Im Laufe der Weiterentwicklung durch Google wurden leistungsfähigere Techniken zur Analyse des Seitenlayouts eingeführt, wodurch es möglich wurde, z. B. ganze Zeitungsseiten auf einmal zu verarbeiten, da die Bilder oder Tabellen als solche erkannt werden. Auch unregelmäßig angeordnete Textblöcke stellen damit kein Problem mehr dar [Smi16, Smi09]. In Abbildung 3.5 ist dieser Vorgang anhand eines Beispiels visualisiert.

Darüber hinaus wurde die Software für den internationalen Gebrauch erweitert. Neben dem Englischen werden aktuell über 100 weitere Sprachen unterstützt. Dafür waren zum einen neue Sprachmodelle notwendig, zum anderen mussten auch weitere Zeichen erkannt werden [Smi16]. Dies stellt bei manchen Sprachen wie z. B. Chinesisch eine große Herausforderung dar, da sie einige Tausend verschiedene Symbole umfassen. Ungeachtet dessen strebt Google an, so viele Sprachen wie möglich zu unterstützen [SAL09]. Zur Realisierung dieser Erweiterungen wurden einige Neuerungen am Kern von Tesseract nötig. Ein wichtiger Schritt war dabei die Integration neuronaler Netze zur Klassifikation der Zeichen [Smi16].

[16] z. B. `https://github.com/mateogianolio/ocr`

(a) Erkannte Textspalten (b) Partitionierung in Zeilen (c) Detektierte Textblöcke

Abbildung 3.5: Layout Analyse einer Zeitungsseite [Smi16]

Es wird deutlich, dass es sich bei Tesseract um ein vielseitig einsetzbares Tool zur Offline-Erkennung gedruckter Texte handelt. Zudem gibt es eine gute Unterstützung für die Verwendung des Programms. Unter gängigen Betriebssystemen wie Linux oder MacOS lässt es sich bequem über den Paketmanager installieren. Der offizielle Programmcode ist unter der Apache License, version 2.0 (ALv2) auf Github[17] abrufbar. Neben den verfügbaren Konfigurationsdateien für über 100 Sprachen, darunter Englisch, Deutsch, Arabisch und historische Schriftarten, kann man die Anwendung auch mit eigenen Daten trainieren. Für diesen Vorgang gibt es neben guten Anleitungen auch verschiedene Werkezuge zur Unterstützung [Smi].

Diese Features ermöglichen es, Tesseract auch für andere Zwecke zu verwenden. Amey Chavan et al. nutzten das Programm, um eine Android-App zur Lösung linearer Gleichungen zu entwickeln. Dabei wird ein Foto der gedruckten Gleichung aufgenommen, vorverarbeitet und mit Tesseract analysiert. Anschließend wird die erkannte Gleichung gelöst, wobei sie auch dem Nutzer zur Verifikation angezeigt wird. Die möglichen mathematischen Ausdrücke sind in diesem Projekt jedoch stark eingeschränkt, sodass nur Gleichungen, wie z. B.

$$5x + 3y = 9$$

zulässig sind. Diese dürfen zudem nur aus den Zeichen „a-z“, „0-9“, „+“, „-“und „=“bestehen. Die Erkennungsrate lag hier bei 85-90% [CN13]. Auch Michael Young präsentiert in seinem Blog[18] eine Möglichkeit, wie Tesseract zur Erkennung von Gleichungen verwendet werden kann. Dabei gibt es weniger Einschränkungen, sodass auch Exponenten und Klammern in den mathematischen Ausdrücken vorkommen dürfen.

Neben diesen Ansätzen, die das OCR Programm für mathematische Ausdrücke verwenden, untersuchten Wissenschaftler, ob damit auch eine Handschriftenerkennung realisierbar

[17] `https://github.com/tesseract-ocr/tesseract`
[18] `http://blog.ayoungprogrammer.com/2013/01/equation-ocr-part-2-training-characters.html`

ist. Um dies herauszufinden, trainierten sie Tesseract mit handschriftlichen Mustern lateinischer Buchstaben. Dabei verwendeten sie eingescannte Texte von drei verschiedenen Testpersonen mit jeweils über 1000 Zeichen. Mit diesem Versuchsaufbau ließ sich eine benutzerspezifische Erkennungsrate von ca. 78% erreichen [RB10a, RB10b].

Auch wenn anhand der genannten Beispiele der Eindruck entsteht, dass Tesseract ein Potential für die Erkennung handschriftlicher mathematischer Ausdrücke besitzt, ist dies aktuell jedoch nur äußerst eingeschränkt der Fall. Dies ist vor allem auf zwei Faktoren zurück zu führen:

1. Tesseract wurde von Grund auf für die Erkennung maschinell erstellter Zeichen und Texte konzipiert. Diese sind sehr regelmäßig aufgebaut, haben gerade Zeilen mit konstantem Zeilenabstand und abgesehen von verschiedenen Fonts sehen die Buchstaben immer gleich aus. Deshalb führen die Unregelmäßigkeiten, welche bei handschriftlichen Schriftstücken unweigerlich auftreten, zu Problemen. Besonders bei der Segmentierung ist dies der Fall, was bei einem Fehler zur Folge haben kann, dass eine korrekte Erkennung der Buchstaben anschließend von vornherein nicht mehr möglich ist [RB10a].

2. Da mathematische Ausdrücke deutlich schwerer zu erkennen sind als Texte, versucht Tesseract in seiner aktuellen Version nicht, diese zu erkennen [LS13]. Das liegt im Wesentlichen daran, dass es sich hier um ein 2D-Problem handelt und somit keine klare Strukturierung in Zeilen vorliegt (siehe Abschnitt 2.3). Um Interferenzen bei der Verarbeitung eingescannter Buchseiten, welche Formeln enthalten, zu vermeiden, wurden in das Programm Algorithmen integriert, welche solche Bereiche detektieren und von der Verarbeitung ausschließen. Hierfür wird die Dichte mathematischer Symbole mit einigen Heuristiken kombiniert [LS13]. Abbildung 3.6 zeigt das Ergebnis dieses Vorgangs.

Für zukünftige Versionen ist jedoch geplant, dass die Formeln erkannt und nicht von der Verarbeitung ausschlossen werden [LS13]. Solange dies allerdings nicht der Fall ist, eignet sich Tesseract nicht für den in dieser Arbeit untersuchten Zweck.

3.2.6 PME Parser

Der PME Parser wurde wie Seshat von Francisco Álvaro Muñoz im Rahmen seiner Dissertation [Á15] entwickelt. Zuerst untersuchten er und sein Team dabei das Problem, gedruckte mathematische Ausdrücke mithilfe probabilistischer Grammatiken zu erkennen. Später wendeten sie sich der komplizierteren Handschriftenerkennung zu [Á15]. Außer den Besonderheiten, welche sich aus dem Unterschied zwischen der Erkennung gedruckter und handschriftlicher Zeichen ergeben, funktioniert das Programm nach dem gleichen Prinzip wie Seshat. Aufgrund dieser Tatsache und da es möglicherweise eine gute Basis oder Inspiration für ein Programm zur Offline-Erkennung handschriftlicher mathematischer Ausdrücke darstellen könnte, wird hier nur kurz auf den PME Parser eingegangen.

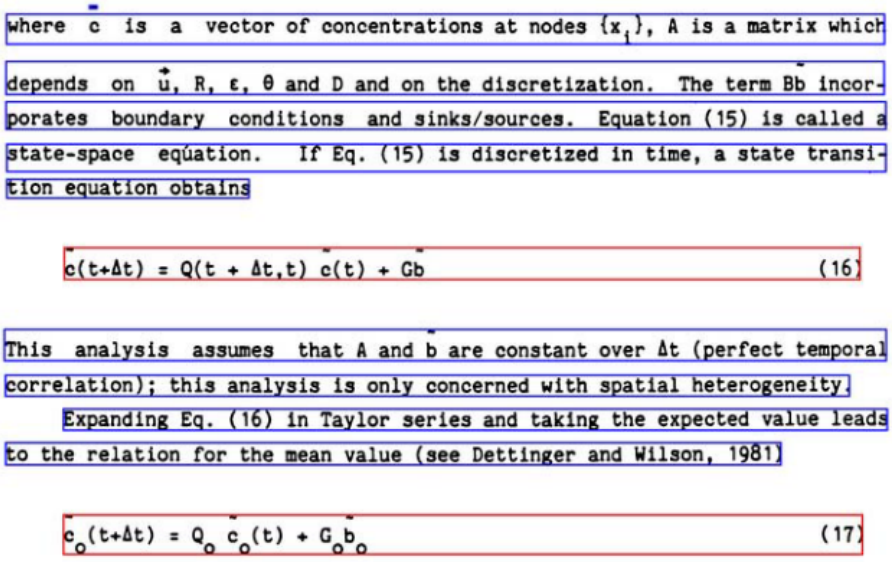


Abbildung 3.6: Detektierte Bereiche mit Formeln (rot markiert) [LS13]

Der in C++ geschriebene Quelltext ist unter der GPLv3 auf Github[19] erhältlich. Um das Programm kompilieren und ausführen zu können, wird noch die Magick++ Library[20] benötigt. Die Eingabe kann in verschiedenen Formaten als Bilddatei vorliegen. Die vom Programm erkannte Formel wird anschließend im LaTeX-Format ausgegeben [ÁSB11, Áa, Á15].

Allerdings unterstützt der PME Parser nur gedruckte Ausdrücke und funktioniert hier auch nicht sehr zuverlässig. Zudem wurde der veröffentlichte Programmcode seit Oktober 2014 nicht mehr aktualisiert oder weiter entwickelt, was vermuten lässt, dass seine Entwicklung eingestellt wurde.

3.2.7 $-Recognizer

Die Familie der $-Recognizer unterscheidet sich in einigen zentralen Merkmalen von den bisher untersuchten Programmen zur Handschriftenerkennung. Erstens werden keine komplexen Ansätze wie neuronale Netze verwendet. Zweitens liegt der Fokus weniger auf den konkreten Implementierungen, sondern auf den Algorithmen dahinter. Es wird großen Wert darauf gelegt, dass diese einfach und leicht zu verstehen sind. Aufgrund der genannten Gesichtspunkte werden die $-Recognizer an dieser Stelle ganz bewusst betrachtet, um eine mögliche Alternative zu den bisherigen Ansätzen aufzuzeigen. Allerdings können mit diesen Algorithmen immer nur einzelne Zeichen bzw. Gesten erkannt werden, weshalb eine Verarbeitung ganzer mathematischer Ausdrücke auf einmal nicht möglich ist.

Die erste Variante der $-Recognizer, der $1-Recognizer, wurde 2007 von zwei Forschern der University of Washington, Jacob O. Wobbrock und Yang Li, sowie Andrew D. Wilson

[19] https://github.com/falvaro/pme_parser
[20] http://www.imagemagick.org/Magick++/

von Microsoft Research veröffentlicht. Ihr Ziel war es, eine einfache, billige und leicht zu bedienende Lösung zur Erstellung von User Interface (UI)-Prototypen mit Gestensteuerung zu entwickeln. Zuvor blieb die Integration einer Gestensteuerung in Benutzeroberflächen trotz der Omnipräsenz von mobilen Geräten mit Touchscreen meist nur Experten auf dem Gebiet der Mustererkennung vorbehalten. Um diesem Zustand Abhilfe zu schaffen, verzichteten Wobbrock et al. in den entwickelten Algorithmen vollständig auf neuronale Netze und jegliche Art von Feature-basierten Klassifikatoren, um auch eine für Neulinge und UI-Designer einfach zu verstehende Lösung zu schaffen [WWL07].

Neben dem $1-Recognizer gehören der Familie noch der $N- und $P-Recognizer an. Alle drei sind in die Kategorie der Online-Erkennung einzuordnen.

$1-Recognizer

Der $1-Recognizer kann nur mit Zeichen umgehen, die aus einem einzigen Stroke bestehen (daher auch der Name). Der Algorithmus dahinter ist in vier Schritte gegliedert, wobei immer nur grundlegende Geometrie verwendet wird. Die ersten drei Schritte bilden zusammen die Normalisierung der Eingabedaten und werden deshalb immer durchlaufen, egal ob es sich um einen Trainings- oder Erkennungsprozess handelt. Der vierte Schritt stellt schließlich die eigentliche Erkennung des Zeichens dar.

Schritt 1: Resampling Die zu verarbeitenden Strokes können sich in den Koordinaten der gespeicherten Punkten unterscheiden, auch wenn sie objektiv betrachtet den identischen Pfad repräsentieren. Dies hängt mit der Zeichengeschwindigkeit und der Abtastrate des Digitizers zusammen. Damit dies im weiteren Ablauf keine Rolle spielt, wird jeder Stroke in einen annähernd formgleichen Stroke mit einer festen Anzahl an Punkten (N), die zudem äquidistant verteilt sind, umgewandelt. In Abbildung 3.7 ist das Ergebnis dieses Vorgangs für verschiedene Werte von N veranschaulicht. Die Größe N liegt hierbei in einem Zielkonflikt zwischen der Genauigkeit und der Verarbeitungsgeschwindigkeit. Bei Tests hat sich herausgestellt, dass Werte $32 \leq N \leq 256$ für einen Kompromiss gut geeignet sind.

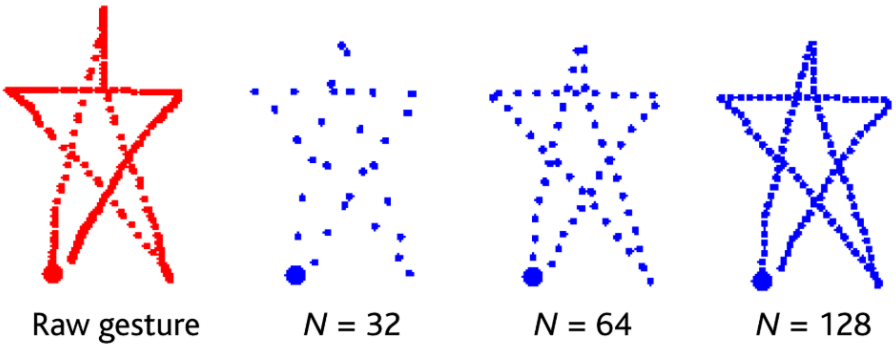


Abbildung 3.7: Resampling eines Sterns für verschieden viele Abtastpunkte N. Der Anfang des Strokes ist durch einen großen Punkt gekennzeichnet. [WWL07]

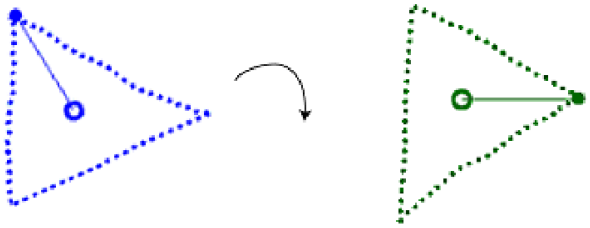


Abbildung 3.8: Rotation anhand des Indikativ-Winkels. Der Ring markiert den Schwerpunkt des Strokes. [WWL07]

Schritt 2: Rotation In diesem Schritt wird der Stroke so gedreht, dass der Winkel zwischen dem Schwerpunkt und dem ersten Punkt des Strokes 0° beträgt. Dieser Winkel wird auch als Indikativ-Winkel bezeichnet. Abbildung 3.8 stellt diesen Vorgang am Beispiel eines Dreiecks dar.

Schritt 3: Skalierung und Verschiebung Nach dem Resampling und der Rotation wird der Stroke so skaliert, dass seine Bounding-Box dem Einheitsquadrat entspricht. Es ist dabei möglich, dass die vorgenommene Skalierung in X- und Y-Richtung voneinander abweicht, was einige Einschränkungen bezüglich der unterscheidbaren Symbole zur Folge hat. Anschließend werden die Punkte des Pfades so verschoben, dass ihr Schwerpunkt auf dem Ursprung des Koordinatensystems zu liegen kommt. Sofern es sich um einen Trainingsvorgang handelt, endet der Algorithmus an dieser Stelle und der normalisierte Stroke wird zusammen mit dem Namen des repräsentierten Zeichens gespeichert, was im Folgenden als Template bezeichnet wird.

Schritt 4: Finden der höchsten Übereinstimmung Der normalisierte Stroke S wird nun mit allen gespeicherten Templates $\mathcal{T}$ verglichen. Als Maß für die Ähnlichkeit zwischen einem Template T und S wird die Pfad-Distanz verwendet, welche die mittlere euklidische Distanz zwischen korrespondierenden Punkten ist und sich nach Formel (3.1) berechnet. Dabei ist $S_{k,x}$ bzw. $T_{k,x}$ die X-Koordinate des k-ten Punktes des Strokes S bzw. Templates T.

$$d = \frac{\sum_{k=0}^{N} \sqrt{(S_{k,x} - T_{k,x})^2 + (S_{k,y} - T_{k,y})^2}}{N} \tag{3.1}$$

Um die bestmögliche Übereinstimmung zwischen einem Template T und dem Stroke S zu finden, müssen alle möglichen Drehungen von S berücksichtigt werden. Die einfachste Variante wäre es hier, eine Brute-Force-Suche über dem Raum aller möglichen Ausrichtungen durchzuführen, d. h. alle Drehungen von 0° bis 360° durchzuprobieren, um so den kleinstmöglichen Wert für die Pfad-Distanz zu ermitteln. Dieses Vorgehen wäre für bis zu 30 Templates in Bezug auf den Berechnungsaufwand akzeptabel, jedoch gibt es auch bessere Alternativen. Durch die Rotation im Schritt 2 der Normalisierung wird sich der optimalen Ausrichtung bereits relativ gut angenähert. Anhand einer Studie von Testdaten erwies sich das Verfahren des Goldenen Schnittes als gut geeignet für die anschließende

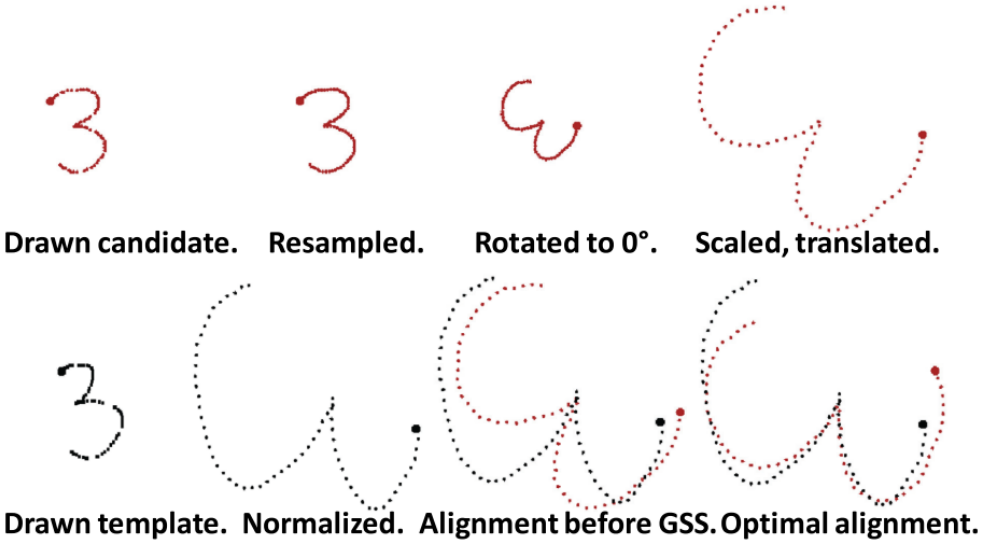


Abbildung 3.9: Skizze des vollständigen Ablaufs [AW10]

Feinausrichtung des Strokes. Dieses ist ein iteratives Verfahren zur Bestimmung von Extremwerten einer Funktion auf einem vorgegebenen Intervall. Wenn für jedes Template aus $\mathcal{T}$ die bestmögliche Übereinstimmung mit dem Stroke S berechnet wurde, werden diese aufsteigend sortiert und als Ergebnis zurückgeliefert [WWL07]. Abbildung 3.9 zeigt eine gelungene Visualisierung der einzelnen Schritte des genannten Algorithmus.

Die Implementierung dieses Algorithmus kann in ca. 100 Zeilen realisiert werden, was aber abhängig von der verwendeten Programmiersprache variieren kann. Auf der zugehörigen Website der University of Washington[21] sind der Pseudocode, sowie verschiedene Referenzimplementierung, die unter der New BSD License (BSDn) stehen, verfügbar. Zudem gibt es eine Web-Demo der JavaScript-Implementierung des $1-Recognizers [WWL16]. Das gleiche Angebot ist für die zwei anderen Algorithmen zur Zeichenerkennung auf dieser Website verfügbar. Zu beachten ist jedoch, dass immer nur Zeichen, die aus einem einzigen Stroke bestehen, verarbeitet werden können. Zudem können aufgrund der während der Normalisierung vorgenommenen Skalierung keine eindimensionalen Zeichen (z. B. „|“ oder „–“) erkannt werden. Die später entwickelten $N- und $P-Recognizer besitzen diese Einschränkung nicht mehr, bei ihnen handelt es sich somit um Multistroke-Recognizer.

$N-Recognizer

Der $N-Recognizer stellt eine wesentliche Erweiterung des zuvor vorgestellten Algorithmus dar und hebt einige seiner Einschränkungen auf. So können von ihm Zeichen aus mehreren Strokes erkannt werden, wobei hier automatisch alle möglichen Anordnungen und Reihenfolgen der einzelnen Strokes beachtet werden. Dadurch ist es z. B. unerheblich,

[21] `http://depts.washington.edu/madlab/proj/dollar/index.html`

ob beim „T“ zuerst der horizontale oder vertikale Strich gezeichnet wird. Auch spielt es keine Rolle, ob man einen Strich von oben nach unten oder anders herum malt. Des Weiteren wurde die Rotationsinvarianz eingeschränkt, sodass nur noch Zeichen mit geringer Abweichung in der Drehung als gleiche erkannt werden [AW10]. Dadurch wird u. a. die Unterscheidung von „A“ und „∀“ möglich. Darüber hinaus können auch eindimensionale Zeichen erkannt werden. Dies wird dadurch ermöglicht, dass vor der Skalierung anhand eines Schwellenwertes überprüft wird, ob es sich um ein 1D-Symbol handelt oder nicht. Wenn dies der Fall ist, erfolgt keine Skalierung auf ein Einheitsquadrat, sondern nur die längere Seite der Bounding-Box wird auf die Normlänge skaliert.

Jedes gespeicherte Symbol wird intern als Menge von Singlestrokes repräsentiert. Zur Erstellung der Singlestrokes werden alle möglichen Permutationen der ursprünglichen Strokes nach Reihenfolge und Zeichenrichtung betrachtet, welche dann jeweils zu einem einzigen Pfad aneinander gefügt werden. Ein Zeichen, das aus N Strokes zusammen gesetzt ist, wird intern somit aus $2^N \cdot N!$ Singlestrokes repräsentiert und gespeichert. Grundsätzlich funktioniert der restliche Algorithmus genau wie der $1-Recognizer. Allerdings muss jedes zu erkennende Symbol mit allen Singlestrokes eines gespeicherten Zeichens verglichen werden, was zu sehr hohem Rechenaufwand führen kann. Deshalb wurden einige Mechanismen zur Beschleunigung dieses Vorgangs eingebaut. So wird eine Vorauswahl an passenden gespeicherten Zeichen anhand des Startwinkels getroffen. Auch ist es möglich, dass nur Zeichen, die aus der selben Anzahl an Strokes zusammen gesetzt sind, als identisch erkannt werden [AW10].

Die Autoren testeten die Leistungsfähigkeit des $N-Recognizers anhand handschriftlicher mathematischer Zeichen. Dabei wurden folgende 20 verschiedene algebraischen Symbole verwendet: 0-9, a, b, c, x, y, =, +, -, (,). Von Schülern sammelten sie von diesen insgesamt über 15.000 Muster. Bei einem, mit 15 Mustern trainierten $N-Recognizer, wurde eine sehr gute Erkennungsrate von 96,6% erreicht [AW10].

Genau wie beim $1-Recognizer gibt es Open-Source-Referenzimplementierungen (u. a. in JavaScript und C#), Pseudocode sowie eine Web-Demonstration[22] [AW12].

$P-Recognizer

Der $N-Recognizer behob zwar das Problem, dass nur Zeichen, die aus einem Stroke bestehen, erkannt werden können, führte aber sogleich ein neues ein. So ist die Laufzeit und die Speicherkomplexität deutlich schlechter geworden. Der $P-Recognizer bedient sich eines neuen Ansatzes, um weniger Kosten bei der Ausführung zu verursachen und dennoch Zeichen aus mehreren Strokes verarbeiten zu können [VAW12].

Symbole werden nun nicht mehr als Menge von Sinlgestrokes betrachtet, sondern als ungeordnete Punktewolke. Somit spielt es keine Rolle mehr, in welcher Reihenfolge oder Richtung die Strokes gezeichnet wurden. Auch die Anzahl der Pfade, aus denen sich ein Zeichen zusammensetzt, ist nicht mehr relevant. Dieses Vorgehen ähnelt der Offline-Erkennung, bei der diese Metadaten von vornherein nicht zur Verfügung stehen. Jedoch ist zu beachten, dass z. B. die Zuordnung der Punkte zu den Strokes weiterhin für

[22] http://depts.washington.edu/madlab/proj/dollar/ndollar.html

einzelne Berechnungen benötigt wird, weshalb man nicht komplett ohne diese zusätzlichen Informationen auskommt. Der Algorithmus gliedert sich grob in zwei Schritte.

Schritt 1: Normalisierung Als erstes wird das zu trainierende oder zu erkennende Zeichen normalisiert. Dieser Vorgang ist identisch zur Normalisierung beim $1-Recognizer mit der Ausnahme, dass keine Drehung vorgenommen wird. Wenn es sich um einen Trainingsprozess handelt, endet der Algorithmus an dieser Stelle und das vorverarbeitete Zeichen wird als Template abgespeichert.

Schritt 2: Erkennung Wie beim $1-Recognizer wird ein zu erkennendes Zeichen S mit allen gespeicherten Templates verglichen. Dafür muss jeweils das Matching mit der besten Güte $d_{opt}(T,S)$ bestimmt werden. Ein Matching zweier Punktewolken S und T wird dabei durch eine bijektive Zuordnung zwischen den beiden Punktemengen erzeugt. Dies wird von Abbildung 3.10 illustriert. Die Güte $d(T,S)$ des Matchings ist durch die Summe der euklidischen Distanzen zwischen den Punktepaaren definiert. Es lässt sich mit der Formel (3.2) berechnen, wobei j in der Formel von i abhängt und das Ergebnis der bijektiven Abbildung $h(i)$, $h : \{0,\ldots,N-1\} \to \{0,\ldots,N-1\}$ ist.

$$d = \sum_{i=0}^{N} \|S_i - T_j\| = \sum_{i=0}^{N} \sqrt{(S_{i,x} - T_{j,x})^2 + (S_{i,y} - T_{j,y})^2} \tag{3.2}$$

Für jedes gespeicherte Template muss die bestmögliche Güte berechnet werden. Jenes mit dem kleinsten Wert für $d_{opt}(T,S)$ hat die höchste Übereinstimmung mit dem verarbeiteten Symbol. Es handelt sich jedoch nicht um ein triviales Problem, das beste Matching zwischen zwei Punktewolken zu berechnen. Hierfür können komplexe Algorithmen wie der Kuhn-Munkres-Algorithmus (auch Ungarische Methode genannt) verwendet werden. Da dies aber nicht der Philosophie der $-Recognizer entsprechen würde, entschieden sich Wobbrock et al. für eine einfachere Heuristik, welche auch sehr gute Ergebnisse erzielt [VAW12]. Auf der Website[23] des $P-Recognizers wird der Zuordnungsvorgang zwischen zwei Punktewolken in einem Video gut visualisiert.

Vergleich

Die vorgestellte Familie der $-Recognizer bietet drei einfache Algorithmen, welche auf grundlegender Trigonometrie aufbauen und für die eine ganze Reihe von Open-Source-Implementierungen verfügbar sind. Aufgrund ihrer Einfachheit sind sie gut in bestehende Projekte integrierbar und leicht erweiterbar. Ein weiterer Vorteil ist, dass zur Laufzeit einfach neue Symbole trainiert und weitere Muster zu bestehenden gespeichert werden können. Somit spielt die Tatsache, dass es keine bereits trainierte Version für die hier benötigten mathematischen Zeichen gibt, keine Rolle. Tabelle 3.4 zeigt eine Gegenüberstellung der drei Varianten. Für die Erkennungsrate wurden jeweils Zeichensätze mit 16 verschiedenen Zeichen getestet [VAW12].

[23] `http://depts.washington.edu/madlab/proj/dollar/pdollar.html`

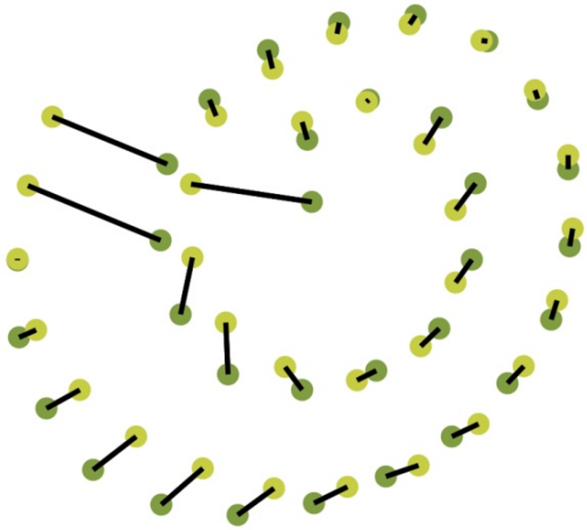


Abbildung 3.10: Matching der Punkte zweier normalisierter Strokes [WAV17]

Der $P-Recognizer ist für die Erkennung handschriftlicher mathematischer Zeichen vermutlich am besten geeignet, da er dem anderem Multistroke-Recognizer in vielen Punkten überlegen ist. In Kapitel 4 wird dies anhand eigener Tests genauer untersucht.

3.3 Übersicht

In diesem Kapitel wurden insgesamt sieben verschiedene Programme vorgestellt, die alle für die OCR bzw. Handschriftenerkennung verwendet werden können. In Bezug auf die Erkennung handschriftlicher mathematischer Ausdrücke sind manche Anwendungen besser und manche weniger gut geeignet. Im Folgenden wird noch einmal ein kurzer Überblick über die vorgestellten Werkzeuge gegeben. In Tabelle 3.5 sind die Online-Tools und in Tabelle 3.6 die Offline-Tools aufgeführt. RNNLIB ist dabei in beiden Tabellen zu finden, da es sich für beide Verfahren eignet. Der Bereich „Wertung (0-5)“ soll einen individuellen Vergleich der Programme, aufgeschlüsselt nach verschiedenen Kriterien, ermöglichen. Die Skala geht hierbei von 0 (schlecht) bis 5 (sehr gut). „Erkannte Zeichen“ und „Erkannte Konstrukte “ sind jeweils auf die handschriftliche Erkennung mathematischer Zeichen bzw. Ausdrücke bezogen. Sofern es nur Unterstützung für gedruckte Zeichen gibt, führt dies zu der Wertung 0.

Tabelle 3.4: Vergleich der $-Recognizer Familie [VAW12]

Kriterium	**$1**	**$N**	**$P**
Typen von Symbolen			
Ein-Stroke-Zeichen	✓	✓	✓
Multi-Stroke-Zeichen	✗	✓	✓
Größeninvariant	✓	✓	✓
Rotationsinvariant	✓	✓/✗	✓
Richtungsinvariant	✗	✓	✓
Erkennungsrate			
benutzerabhängig (Ein-Stroke-Zeichen)	99,5%	98,0%	99,3%
benutzerabhängig (Multi-Stroke-Zeichen)		97,7%	98,4%
benutzerunabhängig (Ein-Stroke-Zeichen)	97,1%	95,2%	96,6%
benutzerunabhängig (Multi-Stroke-Zeichen)		96,4%	98,0%
Komplexität			
Zeitkomplexität des Algorithmus	$\mathcal{O}(n \cdot T \cdot R)$	$\mathcal{O}(n \cdot S! \cdot 2^S \cdot T)$	$\mathcal{O}(n^{2,5} \cdot T)$
Platzkomplexität	$\mathcal{O}(n \cdot T)$	$\mathcal{O}(n \cdot S! \cdot 2^S \cdot T)$	$\mathcal{O}(n \cdot T)$
Ungefähre Anzahl an Programmzeilen	100	200	70

S: Anzahl der einzelnen Strokes eines Zeichens
n: Anzahl der Punkte eines Strokes
T: Anzahl der Muster, mit denen der Recognizer für jedes Zeichen trainiert wurde
R: Anzahl der Iterationen des Verfahren des Goldenen Schnittes (die Implementierung verwendet $R = 10$)

Tabelle 3.5: Übersicht der analysierten Online-Tools

Kriterium	**RNNLIB**	**Seshat**	**RIT**	**$-Recognizer**
Lizenz	GPLv3	GPLv3	GPLv3	BSDn
Programmiersprache	C++	C++	C++/Python	Algorithmus
Unterstützung				
Handschriftenerkennung	✓	✓	✓	✓
Erkennung math. Ausdrücke	✗	✓	✓	✗
Erlernen neuer Zeichen	✓	✗	✗	✓
Wertung (0-5)				
Erkannte Zeichen	0	4	4	0
Erkannte Konstrukte	0	4	4	0
Dokumentation	2	1	1	5
Tutorials	2	0	0	4
Codequalität	2	1	3	5
Erweiterbarkeit	3	1	1	3
Integrierbarkeit	3	3	2	5
Installation	4	3	2	5
Weiterentwicklung	1	2	1	3

Tabelle 3.6: Übersicht der analysierten Offline-Tools

	RNNLIB	**Caffe(2)**	**Tesseract**	**PME Parser**
Lizenz	GPLv3	BSD2/ALv2	ALv2	GPLv3
Programmiersprache	C++	C++/Python	C++	C++
Unterstützung				
Handschriftenerkennung	✓	✗	✗	✗
Erkennung math. Ausdrücke	✗	✗	✗	✓
Erlernen neuer Zeichen	✓	✗	✓	✗
Wertung (0-5)				
Erkannte Zeichen	0	0	0	0
Erkannte Konstrukte	0	0	0	0
Dokumentation	2	5	4	1
Tutorials	2	5	5	0
Codequalität	2	4	5	2
Erweiterbarkeit	3	3	4	1
Integrierbarkeit	3	4	3	2
Installation	4	5	5	2
Weiterentwicklung	1	5	5	1

4 Implementierungsarbeit

In diesem Kapitel wird die Familie der $-Recognizer genauer untersucht und in Hinblick auf ihre Leistungsfähigkeit bei der Erkennung handschriftlicher mathematischer Zeichen evaluiert. In Abschnitt 3.2.7 wurden die Algorithmen der drei verschiedenen $-Recognizer im Detail erläutert. Der $1-Recognizer unterstützt nur Zeichen, die aus einem einzigen Stroke bestehen und eine zweidimensionale Gestalt aufweisen. Aufgrund dieser Einschränkungen eignet sich der $1-Recognizer nicht für die Erkennung mathematischer Symbole. Der $N- und der $P-Recognizer besitzen diese Einschränkungen nicht, weshalb im Folgenden nur auf diese beiden Algorithmen eingegangen wird. Ziel der Implementierungsarbeit ist es, eine flexibel einsetzbare Bibliothek für die zwei genannten $-Recognizer zu entwerfen und umzusetzen. Mithilfe dieser Bibliothek soll anschließend ermittelt werden, in welchem Ausmaß dieser Ansatz für die Handschriftenerkennung geeignet ist.

4.1 Entwicklung der Bibliothek

Die von Wobbrock et al. entwickelten Algorithmen verwenden hauptsächlich einfache geometrische Berechnungen, wodurch es zu keinen Einschränkungen bei der Wahl der Programmiersprache zur Umsetzung der Bibliothek kommt. Um diese möglichst plattformunabhängig zu gestalten und den Einsatz auf eingebetteten System zu ermöglichen, wird C++ als Programmiersprache verwendet. Neben der hohen Performance aufgrund der Hardwarenähe war auch die Objektorientierung ausschlaggebend für diese Programmiersprache. Auf der Website der University of Washington gibt es zwar eine C++-Implementierung für den $N-Recognizer, jedoch ist diese eher als Demonstration gedacht und somit nicht geeignet. Da auch der $P-Recognizer unterstützt werden soll, entschieden wir uns dafür, die Bibliothek von Grund auf neu zu entwerfen.

4.1.1 Anforderungen

Bezüglich der Ein- und Ausgabeformate soll die Bibliothek die selben wie die der CROHME unterstützen, d. h. die Daten der Online-Eingabe liegen als InkML-Datei vor und das Ergebnis wird als LaTeX-Zeichenkette zurück geliefert [MVGK+11]. Dieser Weg der Datenübergabe ist bei einer Bibliothek zur Handschriftenerkennung jedoch unpraktisch und erzeugt unnötigen Overhead, da die Eingabe vom Benutzer meist unmittelbar im Anschluss an die Erstellung verarbeitet werden soll und nicht erst gespeichert wird. Aus diesem Grund sollte die Schnittstelle zum Aufruf der Recognizer direkt mit den Punkten

der Strokes arbeiten. Dennoch sollte es zusätzlich die Möglichkeit geben, gespeicherte InkML-Dateien zu verarbeiten.

4.1.2 Design

Um einen flexiblen Wechsel zwischen den zwei Recognizer-Typen zu ermöglichen, verfügen beide über die selbe grundlegende Schnittstelle, die sie von der abstrakten Klasse „DollarRecognizer“ erben. In der Abbildung 4.1 sind diese Schnittstellen und die Beziehungen zwischen den wichtigsten Klassen beschrieben. Die Klasse „DollarRecongizerLib“ dient als Benutzerschnittstelle zu der gesamten Bibliothek. Mit ihr lassen sich Objekte der Recognizer-Klassen erzeugen und sie verfügt über eine Methode, um InkML-Dateien zu parsen, was mithilfe der Xerces-C++ Library erfolgt. Dieses Feature wird allerdings nicht immer benötigt und führt auf manchen Systemen zu Problemen. Das kann z. B. der Fall sein, wenn wenig Speicherplatz verfügbar ist oder keine weitere Software installiert werden soll. Deshalb wurde die Aufteilung so gewählt, dass nur eine Klasse eine Abhängigkeit zur Xerces-Bibliothek besitzt, wodurch diese leicht deaktiviert werden kann. Die Klasse „Stroke“ repräsentiert einen gezeichneten Pfad. Dabei wird neben den Punkten des Strokes auch seine Kennung (ID) gespeichert, damit die Reihenfolge der Strokes nicht verloren geht. Die andere dargestellte Klasse „Result“ dient zur Übergabe des Ergebnisses der Methode „recognizeSymbol“. Neben den im UML-Diagramm (Abbildung 4.1) dargestellten Klassen und Funktionen werden noch weitere für die Realisierung benötigt, welche aber an dieser Stelle nicht von Bedeutung sind.

Wenn die Bibliothek mit einem Zeichen trainiert wird, soll dieses persistent gespeichert werden. Dafür werden die trainierten Symbole intern in Konfigurationsdateien gespeichert. Diese sind in einem einfachen Format aufgebaut und enthalten neben dem Namen des Symbols Listen mit den Punkten der Strokes, aus denen sich das Zeichen zusammensetzt. Beim Erzeugen eines Objekts der „DollarPRecognizer“- oder „DollarNRecognizer“-Klasse werden die gespeicherten Symbole automatisch geladen und werden bei zukünftigen Aufrufen der Methode „recognizeSymbol“ verwendet.

Der $N-Recognizer ist im Gegensatz zum $P-Recognizer flexibel und kann anhand folgender drei Parameter konfiguriert werden:

1. *Eingeschränkte Rotationsinvarianz*: Es kann ausgewählt werden, ob der Algorithmus komplett invariant bezüglich Rotationen ist oder nicht. Wenn die eingeschränkte Rotationsinvarianz deaktiviert ist, werden radialsymmetrische Zeichen als identisch erkannt, auch wenn sie um 90° oder mehr gedreht sind. Ansonsten ist dies nur in einem Intervall von 45° der Fall.

2. *Identische Stroke-Anzahl*: Wenn dies aktiviert ist, werden nur Zeichen, die aus derselben Anzahl an Strokes bestehen, als gleich erkannt.

3. *Verwendung des Protractors*: Der $N-Protractor ist eine von Wobbrock et al. entwickelte Optimierung des ursprünglichen Algorithmus, welche deutlich schneller arbeitet [AW12]. Er basiert auf dem Protractor, eine von Yang Li entwickelte Erweiterung des $1-Recognizers [Li10].

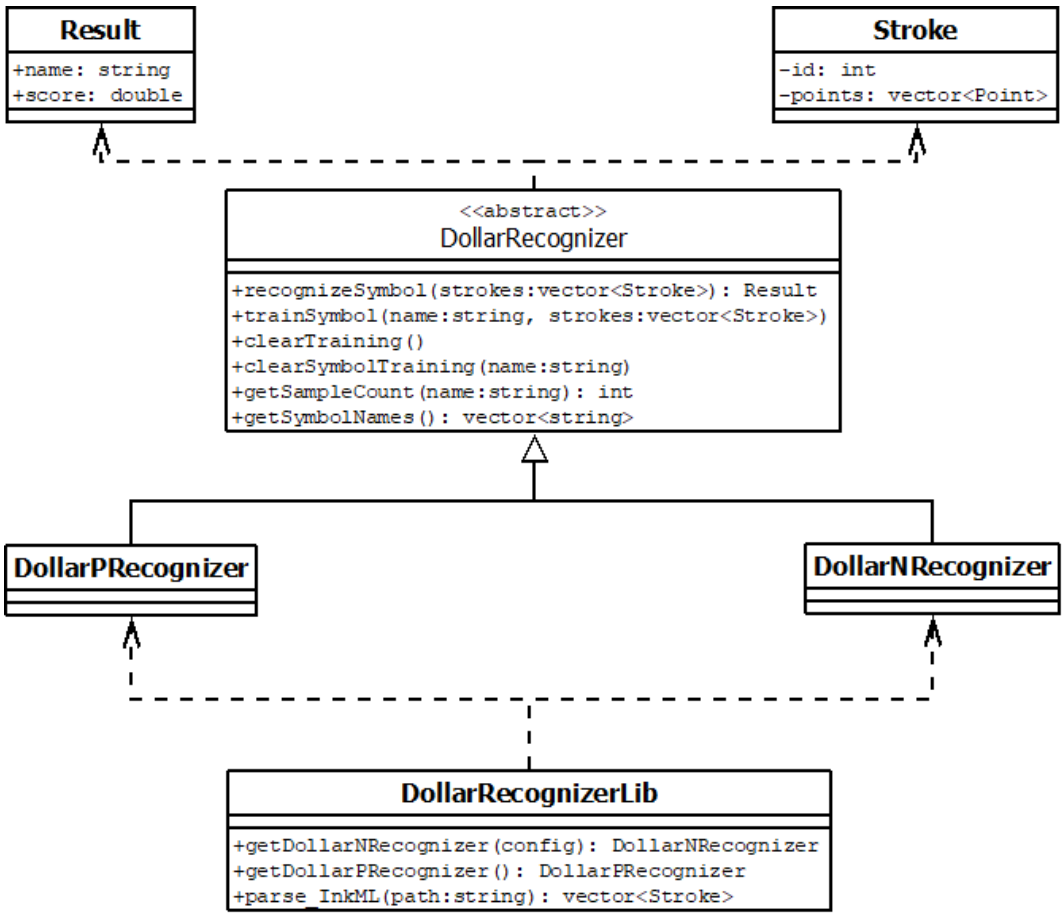
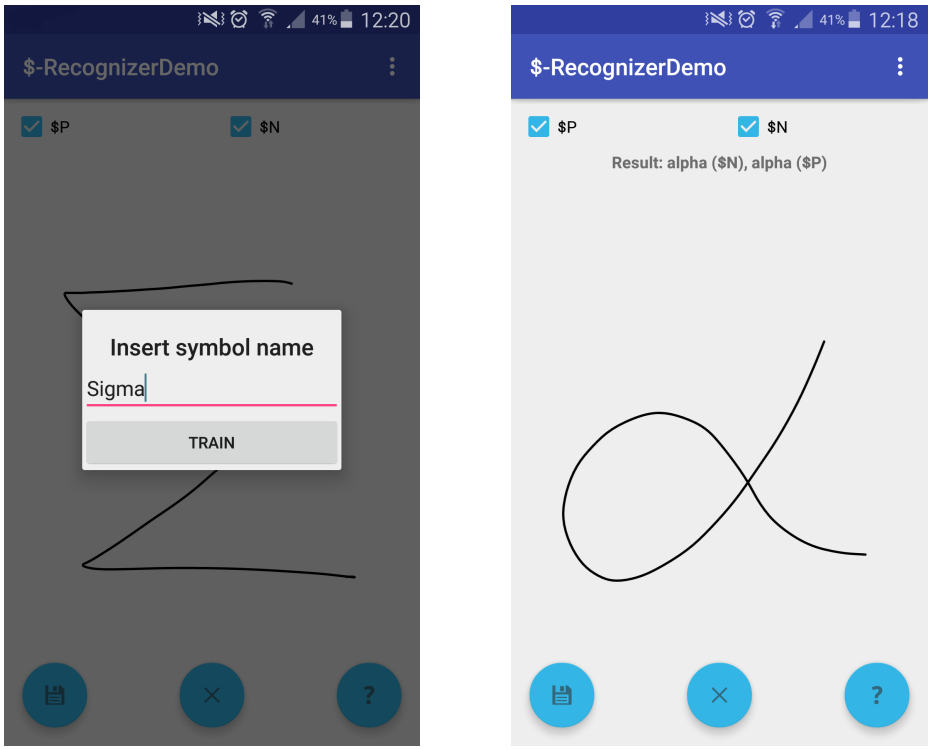


Abbildung 4.1: UML-Diagramm mit den wichtigsten Klassen und Funktionen

Die erste Einstellung muss zwingend bei der Initialisierung des Recognizers festgelegt werden und kann später nicht mehr verändert werden. Bei den anderen zwei kann dies bei jedem Erkennungsvorgang individuell festgelegt werden.

4.1.3 Umsetzung

Der von Wobbrock et al. veröffentlichte Pseudocode sowie die Referenzimplementierung in JavaScript dienten bei der Implementierung der Bibliothek als Orientierung. Zum Erzeugen einer statischen Programmbibliothek aus der $-Recognizer-Bibliothek wurde ein Makefile erstellt. Für das Kompilieren kann konfiguriert werden, ob die Xerces-C++ Library für die Unterstützung von InkML-Dateien eingebunden wird oder ob auf dieses Feature verzichtet werden soll. Zur Demonstration der Bibliothek wurde eine Android-App entwickelt, die es ermöglicht, das Erkennungsprogramm mit gezeichneten Symbolen zu trainieren und diese zu erkennen. Abbildung 4.2 zeigt zwei Screenshots der App „DollarRecognizerDemo“. Das auf der integrierten Zeichenfläche gemalte Symbol „Σ“ wird auf dem linken Bild zum Training des $P- sowie $N-Recognizers verwendet. Auf dem rechten Bild wurde das gezeichnete „α“ erfolgreich von beiden Algorithmen erkannt.



(a) Speichern eines neuen Symbols (b) Erkennen des Gezeichnetem

Abbildung 4.2: Screenshot der Android-App zur Demonstration der Dollar-Recognizer-Bibliothek

Darüber hinaus gibt es zwei kleine Kommandozeilenprogramme, welche für Testzwecke verwendet werden können und eine gute Möglichkeit zur Einarbeitung in die Bibliothek darstellen. Mithilfe des ersten Programms können Symbole aus einzelnen Dateien trainiert und erkannt werden. Das zweite Werkzeug ermöglicht die Verarbeitung ganzer Ordner. Dieses Tool wurde in Kombination mit einigen für diesen Zweck geschriebenen Python-Skripten zur Evaluierung der Erkennungsrate verwendet, worauf im nächsten Abschnitt eingegangen wird.

4.2 Evaluierung der Bibliothek

In den publizierten wissenschaftlichen Papieren über die $-Recognizern wurde die Erkennungsrate immer nur anhand 16 bzw. 20 verschiedenen Symbolen getestet [AW10, VAW12, AW12, WWL07]. Der für unsere Zwecke benötigte Zeichensatz (siehe Abschnitt 2.2) umfasst jedoch mehr als 120 zu unterscheidende Symbole. Somit lassen sich die Ergebnisse aus den genannten Veröffentlichungen hierauf nicht übertragen. Um zu bestimmen, wie sich die $-Recognizer bei dieser deutlich größeren Anzahl an Symbolen verhalten, wurden mithilfe der zuvor implementierten Bibliothek eigene Tests durchgeführt.

Abbildung 4.3: Screenshot der Android-App zum Erfassen der Testdaten

4.2.1 Erhebung der Testdaten

Zur Durchführung der Evaluation werden zunächst genügend Muster der 120 aufgelisteten mathematischen Zeichen benötigt. Die verfügbaren Datensätze der CROHME oder der Mathematical Formulae Database (MfrDB)[1] waren hierfür nicht geeignet, da sie zum einen nicht alle benötigten Zeichen enthalten und zum anderen hauptsächlich Dateien mit ganzen mathematischen Ausdrücken beinhalten. Daher wurde der Beschluss gefasst, eine Android-App zu programmieren, um selbst passende Trainings- und Testdaten sammeln zu können. In dieser App „CollectHandwriting“ werden nacheinander die zu zeichnenden Symbole angezeigt und können jeweils auf der integrierten Malfläche gezeichnet werden. Die vom Benutzer generierten Strokes werden anschließend im InkML-Format abgespeichert. Abbildung 4.3 zeigt einen Bildschirmfoto der App.

Insgesamt wurden auf diesem Weg 4560 Muster der Symbole erhoben, welche von acht verschiedenen Personen stammen. Dabei sollten die mathematischen Symbole so gezeichnet werden, wie es in einer alltäglichen Anwendung der Fall wäre. Dies soll dazu beitragen, dass realistische Ergebnisse bei den Erkennungsraten erzielt werden. Fast alle Testpersonen besitzen einen technischen oder naturwissenschaftlichen akademischen Hintergrund, was dem potentiellen Anwenderkreis dieser Software entspricht.

[1] `http://mfr.felk.cvut.cz/Database_download.html`

4.2.2 Auswertung

Für die Auswertung der implementierten Bibliothek wurde ein Computer mit folgender Konfiguration verwendet:

- Betriebssystem: Ubuntu 16.04 LTS (64-Bit)
- Prozessor: Intel® Core™ i7-7500U
- Arbeitsspeicher: 16,0 GB

Vergleich aller Konfigurationsmöglichkeiten des $N-Recognizers

Zu Beginn der Auswertung werden zunächst die Unterschiede zwischen den verschiedenen Konfigurationen des $N-Recognizers ermittelt. Dabei sind folgende drei Einstellungsoptionen möglich, welche detailliert in Kapitel 4.1.2 beschrieben wurden:

a) *Eingeschränkte Rotationsinvarianz*

b) *Identische Stroke-Anzahl*

c) *Verwendung des Protractors*

Jede dieser Optionen kann entweder aktiviert oder deaktiviert sein. Für die Beschreibung der Konfiguration des $N-Recognizers wird im Folgenden die Notation „$N=abc“ verwendet, wobei $a, b, c \in \{t, f\}$ und anzeigen, ob die jeweilige Einstellung aktiviert ist oder nicht. „*t*“ steht dabei für true, also aktiviert, und „*f*“ für false. Die Reihenfolge ist dabei die Gleiche, in der die drei Konfigurationsmöglichkeiten oben aufgelistet sind. Beispielsweise bedeutet die Konfiguration „$N=tft“, dass bei dem $N-Recognizer die eingeschränkte Rotationsinvarianz aktiviert ist, nur Zeichen mit der gleichen Anzahl an Strokes als identisch erkannt werden und der Protractor aktiviert ist. Abbildung 4.4 zeigt die Unterschiede der acht möglichen Gesamtkonfigurationen in Bezug auf die Laufzeit für das Training und Erkennen von jeweils 119 Zeichen, wobei vor dem Erkennungsvorgang jeweils ein Muster pro Zeichen trainiert wurde. „arctan“ wurde an dieser Stelle von den 120 gesammelten Zeichen nicht verwendet - der Grund hierfür wird später erläutert. Beim Trainingsvorgang gibt es keine Laufzeitunterschiede bezogen auf die verschiedenen Konfigurationen. Eine Deaktivierung des Protractors führt vor allem in Kombination mit der Einstellung, dass Symbole zur Übereinstimmung nicht aus der gleichen Anzahl an Strokes bestehen müssen, zu einer massiven Verlangsamung des Vorgangs. Die anderen Einstellungen spielen hier keine signifikante Rolle. Abbildung 4.5 zeigt einen Vergleich der Erkennungsraten. Hierfür wurde das Programm jeweils mit vier Mustern der 119 Symbole trainiert. Anschließend wurde die Erkennungsrate mit einem Satz der Zeichen, die nicht für das Training verwendet wurden, ermittelt. Es wird deutlich, dass die Aktivierung der eingeschränkten Rotationsinvarianz - unabhängig von den anderen Konfigurationen - die Erkennungsrate stets um ca. 12 Prozentpunkte erhöht. Dies liegt im Wesentlichen daran, dass dadurch Symbole wie „+“ und „x“ deutlich besser unterschieden werden können. Wenn der Recognizer so konfiguriert ist, dass nur Zeichen mit der gleichen Anzahl an

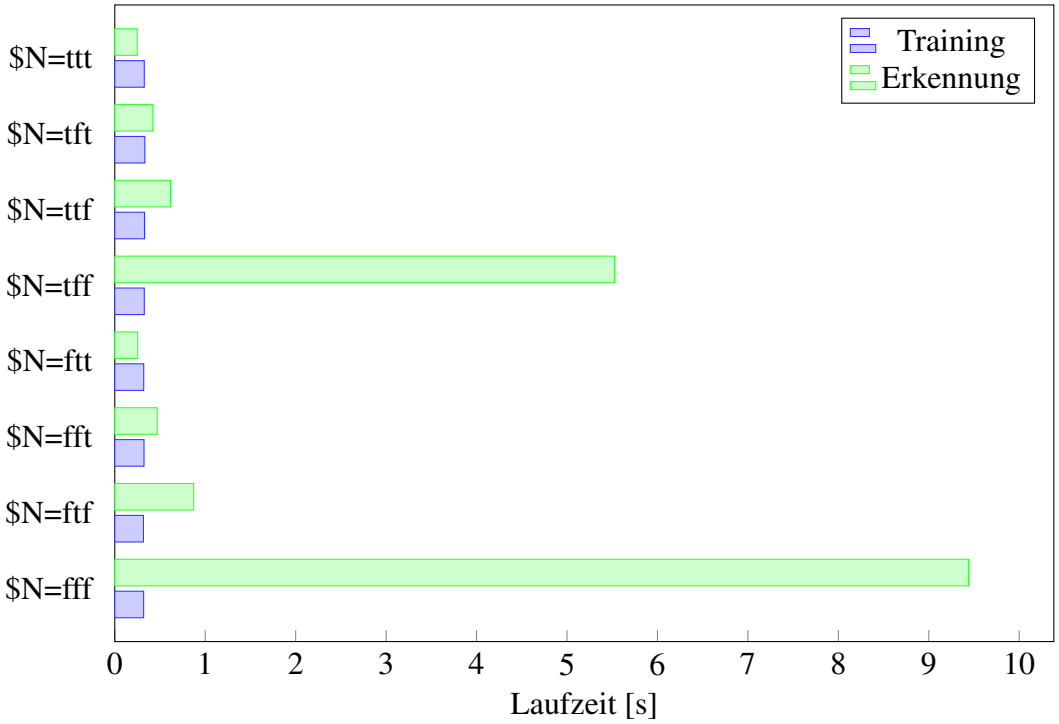


Abbildung 4.4: Vergleich der Laufzeiten für Training (119 Symbole) und Erkennung (119 Symbole, wobei je ein Muster trainiert war) der verschiedenen Konfigurationen des $N-Recognizers.

Strokes als identisch erkannt werden, führt das in diesem Fall stets zu einer Erhöhung der Erkennungsrate um ca. zwei bis vier Prozentpunkte. Man sollte dabei jedoch bedenken, dass bei der vorliegenden Auswertung alles vom selben Anwender geschrieben wurde. Bei unterschiedlichen Benutzern kommt es deutlich öfter vor, dass es Abweichungen in der Anzahl der Strokes beim gleichen Symbol gibt. Zusammen mit der zuvor betrachteten Grafik lassen sich dennoch zwei Schlüsse ziehen.

- Es sollte bei der Erkennung mathematischer Zeichen immer die eingeschränkte Rotationsinvarianz verwendet werden, da dies zu höheren Erkennungsraten führt.

- Die Verwendung des Protractors verringert die Erkennungsrate zwar um 0,2 bis 1,7 Prozentpunkte, führt aber zu deutlich kürzeren Laufzeiten. Diese sind abhängig von der restlichen Konfiguration um den Faktor 2,5 bis 20 besser, was eine erhebliche Zeitersparnis bedeutet. Deshalb sollte diese Erweiterung auch immer aktiviert sein.

Mithilfe weiterer Messungen soll im Weiteren geklärt werden, welche Option bei der zweiten Einstellungsmöglichkeit sinnvoll ist.

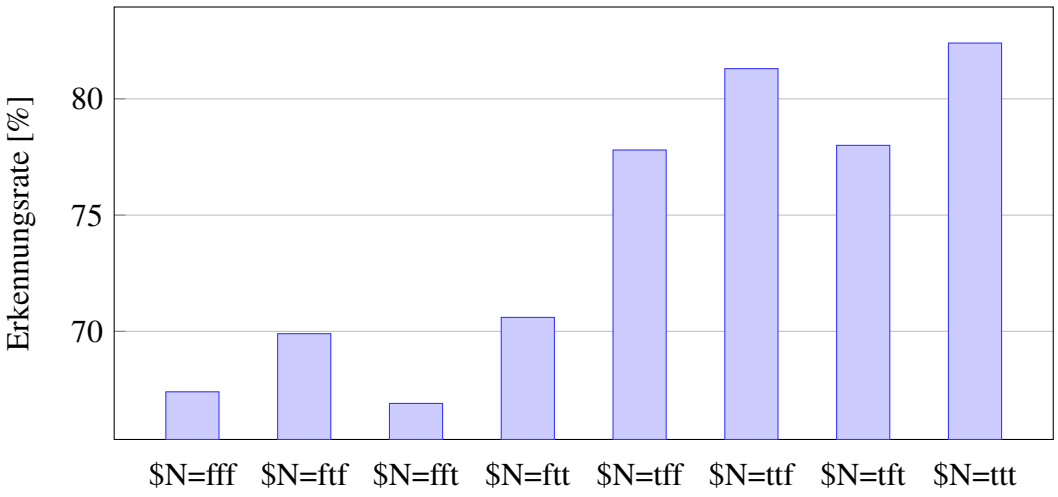


Abbildung 4.5: Vergleich der Erkennungsraten für die verschiedenen Konfigurationen des $N-Recognizers

Vergleich des $P- und $N-Recognizers bezüglich Laufzeit und Speicherplatzbedarf

Bislang wurden nur die verschiedenen Konfigurationen des $N-Recognizers untereinander verglichen. Die zwei erfolgversprechendsten Varianten ($N=ttt und $N=tft) werden nun mit dem $P-Recognizer verglichen. Wie in Tabelle 3.4 deutlich wird, unterscheiden sich diese zwei Vertreter der $-Recognizer-Familie grundlegend in der Laufzeit- und Speicherkomplexität. Dies soll beim Training der Erkennungsprogramme mit Zeichensätzen à 120 Zeichen, die sich in der maximalen Anzahl an Strokes unterscheiden, aus denen sich ein Symbol zusammensetzt, evaluiert werden. Da die Konfiguration der $N-Recognizer für den Trainingsprozess so gut wie keine Rolle spielt, wird in den zwei Grafiken nicht danach differenziert. Abbildung 4.6 visualisiert die zur Ausführung benötigte Zeit. Beim $N-Recognizer lässt sich ein deutlicher Anstieg der Laufzeit mit zunehmender maximaler Stroke-Anzahl beobachten. Ab acht Strokes war kein Training mehr möglich, da hierfür der Arbeitsspeicher von 16 GB des zur Auswertung verwendeten Computers nicht mehr ausreichte. Bezogen auf den benötigten Speicherplatz lässt sich in Abbildung 4.7 der gleiche Zusammenhang beobachten. Zur besseren Visualisierung wurde die X-Achse in dieser Grafik logarithmisch skaliert. Im Gegensatz dazu ist der $P-Recognizer sowohl in der Laufzeit als auch bezüglich des benötigten Arbeitsspeichers unabhängig von der untersuchten Größe. Der $N-Recognizer hat eine Laufzeit- und Platzkomplexität von $\mathcal{O}(n \cdot S! \cdot 2^S \cdot T)$, da alle Permutationen der Strokes eines Zeichens bezogen auf Malrichtung und Reihenfolge betrachtet werden. Der entscheidende Faktor S steht in der Formel für die Anzahl der Strokes, aus der ein Zeichen besteht. Für Symbole, die aus mehr als vier bis fünf Pfaden bestehen, eignet sich diese Variante also nicht mehr. In Realität ist dies aber nur selten der Fall. Bei den mit „CollectHandwriting“ gesammelten Schriftzügen tritt dies z. B. nur bei „arctan“ auf, weshalb diese trigonometrische Funktion bei einigen Evaluationen des $N-Recognizers außen vor gelassen wird.

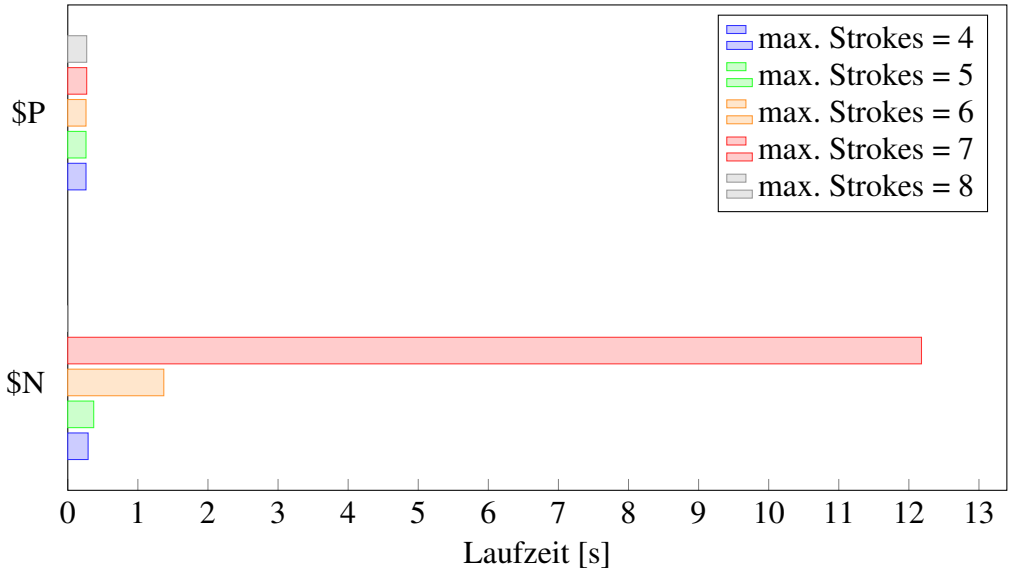


Abbildung 4.6: Laufzeit für das Training mit 120 Zeichen, wobei die maximale Anzahl der Strokes, aus denen sich ein Zeichen zusammen setzt, variiert.

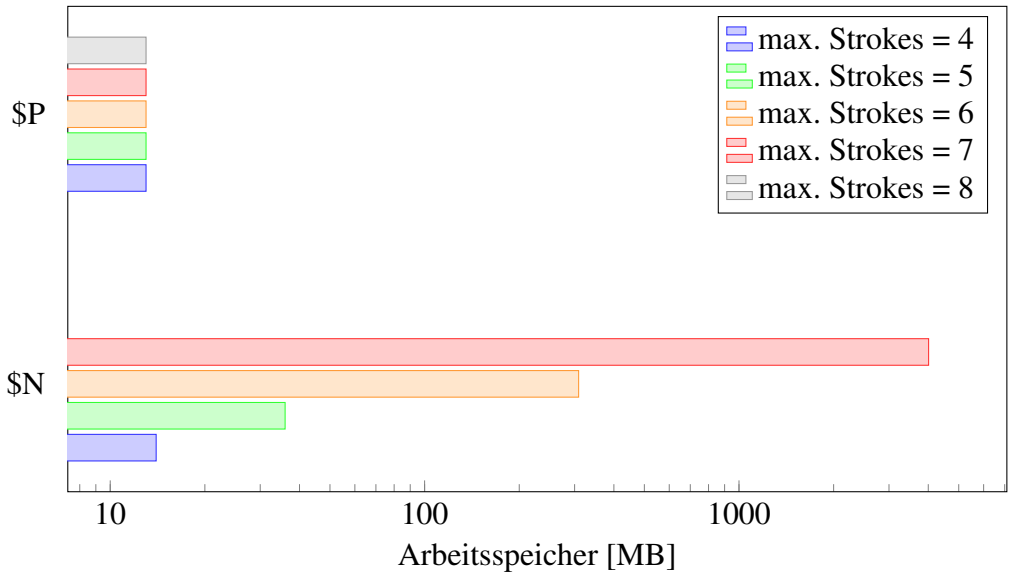


Abbildung 4.7: Benötigter Arbeitsspeicher für das Training mit 120 Zeichen, wobei die maximale Anzahl der Strokes, aus denen sich ein Zeichen zusammen setzt, variiert. Zur besseren Darstellung ist die X-Achse logarithmisch skaliert.

Evaluierung der Erkennungsrate des $P- und $N-Recognizers

In den nächsten drei Grafiken (Abb. 4.8 bis 4.10) ist der Verlauf der Erkennungsrate in Abhängigkeit von der Anzahl trainierter Muster pro Symbol für den $P- sowie für die zwei Konfigurationen ($N=ttt, $N=tft) des $N-Recognizers zu sehen. Dabei wurde die benutzerabhängige Erkennungsrate bestimmt, d. h. die Trainings- und Testdaten stammen ausschließlich von demselben Anwender. Als Testdaten wurden die Schriftzüge der drei Personen verwendet, von denen am meisten Muster vorliegen. Zur Auswertung eines Datensatzes mit n Samples pro Zeichen wurde das Erkennungsprogramm jeweils n-Mal mit je x ($x \in \{1, \ldots, n-1\}$) unterschiedlichen Mustern trainiert. Nach jedem Trainingsvorgang wurde ein zufälliger Zeichensatz aus den verbleibenden für den Test ausgewählt. Die ermittelten Erkennungsraten wurden anschließend für jedes x durch Berechnung des Mittelwertes zusammengefasst. Die Werte sind in den Abbildungen 4.8, 4.9 und 4.10 zu sehen.

Aus den Grafiken lässt sich unter den drei betrachteten Erkennungsverfahren kein klarer Gewinner ermitteln. Bei dem ersten und zweiten Anwender liegt $N=ttt meist vorne, wohingegen beim dritten Anwender $P immer die beste Erkennungsrate liefert. Die Laufzeit außen vorgelassen lässt sich nur mit Bestimmtheit sagen, dass so viele Trainingsdaten wie möglich verwendet werden sollten. Die höchste Erkennungsrate liegt beim zweiten Benutzer mit ca. 90% vor. Beim ersten Anwender liegt das Maximum, ähnlich wie bei der dritten Person, etwas niedriger bei ca. 85%.

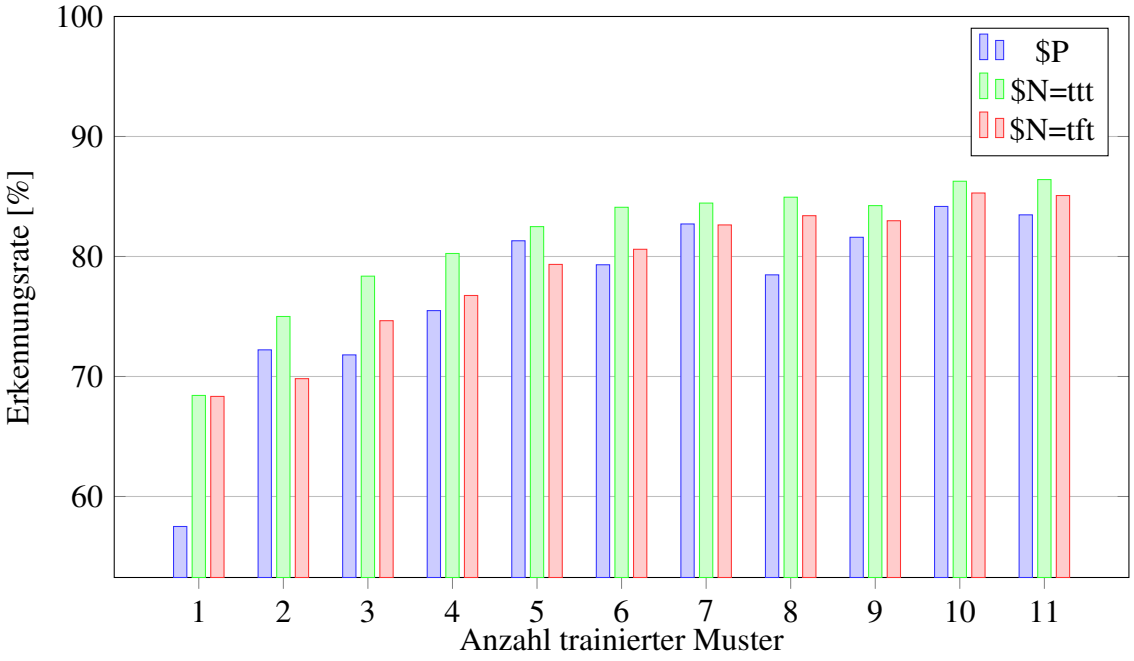


Abbildung 4.8: Erkennungsrate in Abhängigkeit der Anzahl trainierter Muster pro Zeichen für den 1. Anwender (21, w., Eingabe vorwiegend mit Stift)

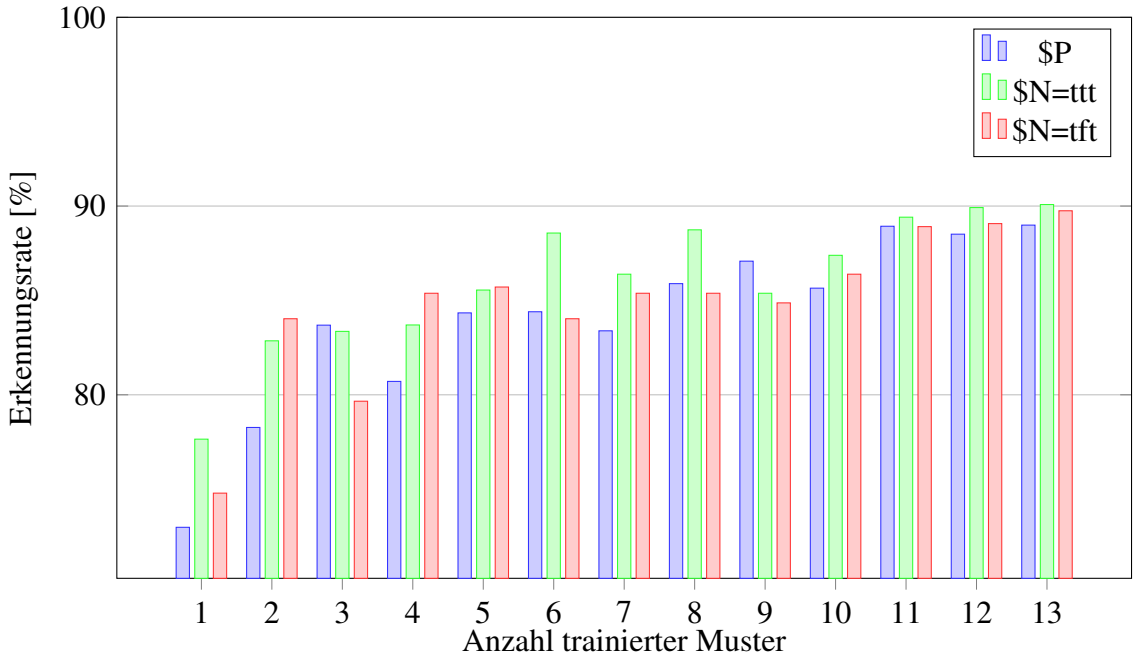


Abbildung 4.9: Erkennungsrate in Abhängigkeit der Anzahl trainierter Muster pro Zeichen für den 2. Anwender (21, m., Eingabe vorwiegend mit Finger)

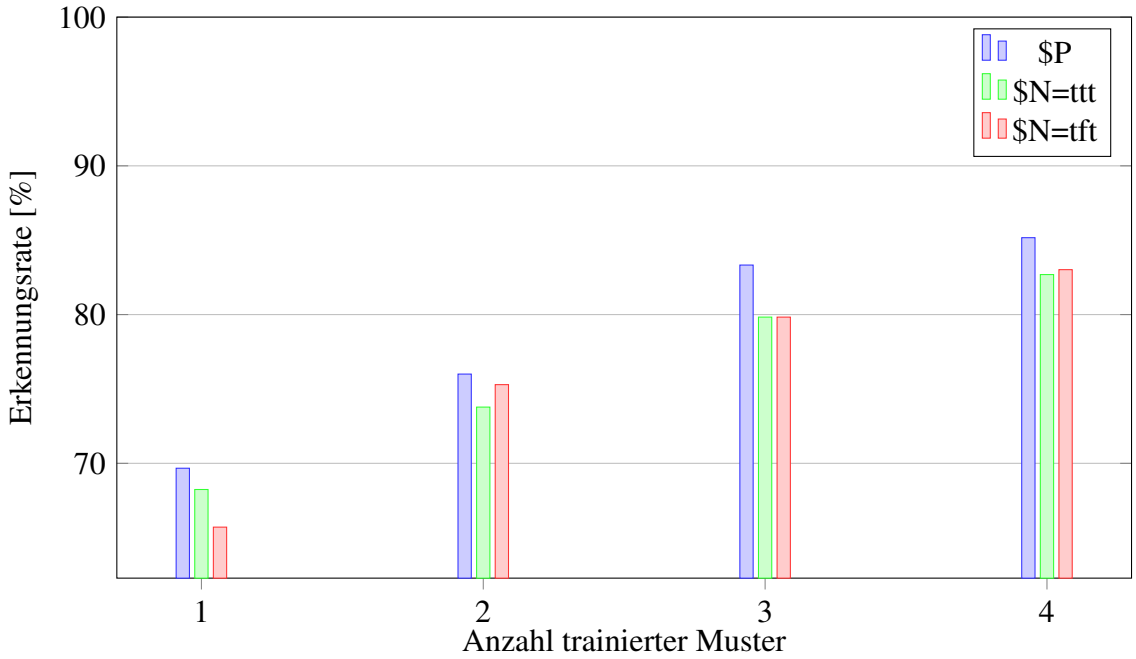


Abbildung 4.10: Erkennungsrate in Abhängigkeit der Anzahl trainierter Muster pro Zeichen für den 3. Anwender (56, w., Eingabe nur mit Stift)

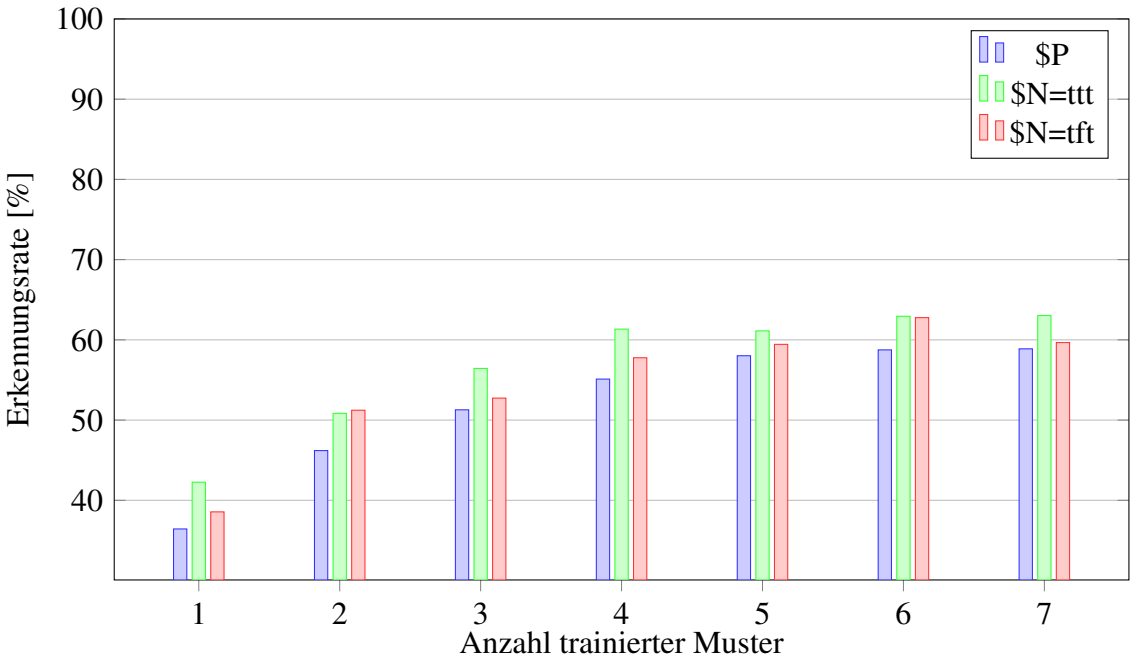


Abbildung 4.11: Erkennungsrate in Abhängigkeit der Anzahl trainierter Muster pro Zeichen, welche von verschiedenen Benutzern stammen.

Bei den meisten Systemen zur Handschriften-Erkennung stammen die Trainingsdaten nicht von den Personen, die diese später verwenden. Somit ist die benutzerunabhängige Erkennungsrate von großem Interesse. Für die Auswertung wurden aus den $B = 8$ Testpersonen jeweils $b \in \{1, \ldots, B-1\}$ für das Training ausgewählt. Dabei wurde für jede Person zufällig einer ihrer geschriebenen Zeichensätze ausgewählt. Anschließend wurde für alle verbleibenden Personen mit je einem Zeichensatz die Erkennungsrate ermittelt. Dieser Ablauf wurde einige Male wiederholt. Im Anschluss daran wurden alle erhobenen Erkennungsraten zu Durchschnittswerten zusammengefasst, was in Abbildung 4.11 zu sehen ist. Im Gegensatz zur benutzerabhängigen Auswertung liegen die Erkennungsraten hier deutlich niedriger bei maximal 63%. Zudem gab es deutlich größere Schwankungen innerhalb der gemessenen Erkennungsraten. Bei sieben trainierten Mustern schwankte die Erkennungsrate je nach den ausgewählten Personen für Training und Erkennung um bis zu 20 Prozentpunkte.

Auswertung der falsch erkannten Zeichen

Neben den durchschnittlichen Erkennungsraten ist es insbesondere interessant, welche Zeichen besonders häufig falsch erkannt wurden. Tabelle 4.1 zeigt jeweils für die verwendeten Recognizer-Typen die drei Zeichen mit der höchsten Falscherkennungsrate. Zudem ist auch aufgeführt, als welches Symbol diese fälschlicherweise erkannt wurden und wie hoch der prozentuale Anteil an der Falscherkennung war.

Tabelle 4.1: Zeichen mit der höchsten Falscherkennungsrate (FER) für Anwender 1

Zeichen	**FER** (in %)	**Erkannt**	**Anteil** (in %)	**Erkannt**	**Anteil** (in %)
Recognizer: $P					
. (Punkt)	81.06	-	14.02	C	14.02
z	81.06	Z	43.93	g	19.63
v	75.00	ν	47.47	V	37.37
Recognizer: $N=ttt					
. (Punkt)	91.67	-	11.57	l	10.74
x	83.33	X	50.00	$\times$	36.36
v	75.00	ν	51.52	V	38.38
Recognizer: $N=tft					
. (Punkt)	92.42	E	15.57	$\mathbb{C}$	11.48
x	88.64	X	45.30	$\times$	35.90
w	70.45	W	67.74	ω	15.05

Tabelle 4.2: Zeichen mit der höchsten Falscherkennungsrate (FER) bei der benutzerunabhängigen Auswertung

Zeichen	**FER** (in %)	**Erkannt**	**Anteil** (in %)	**Erkannt**	**Anteil** (in %)
Recognizer: $P					
. (Punkt)	91.27	-	20.00	arctan	6.96
κ	88.49	K	23.32	X	12.56
l	87.70	ι	32.58	(	13.12
Recognizer: $N=ttt					
. (Punkt)	96.43	Γ	12.59	–	11.85
ν	89.29	v	28.80	V	21.60
z	87.86	Z	40.65	2	18.70
Recognizer: $N=tft					
. (Punkt)	93.57	–	12.98	I	9.16
0	89.29	O	40.80	o	20.80
σ	89.29	Ξ	8.80	E	8.80

Besonders auffällig ist, dass der Punkt meist falsch erkannt wurde. Dies ist hauptsächlich auf die geringe Größe und fehlende Struktur des Zeichens zurückzuführen. Da die Stroke-Repräsentation eines Symbols auf die gleiche Größe skaliert wird, führt dies bei dem Punkt zu einer hohen Varianz der normalisierten Form. Tabelle 4.2 zeigt die am häufigsten falsch erkannten Zeichen für die benutzerunabhängige Auswertung. Genau wie im vorherigen Fall führt der Punkt wieder die Liste mit der höchsten Falscherkennungsrate an. Wenig verwunderlich ist die hohe Verwechslungsrate zwischen „0“, „O“ und „o“ aus dem dritten Abschnitt der Tabelle.

4.2.3 Ergebnis

Die $-Recognizer liefern trotz des großen Zeichensatzes von 120 Symbolen immer noch relativ gute Ergebnisse, obwohl nur wenige Samples trainiert wurden. Die wichtigsten Schlüsse, die sich aus den Statistiken ziehen lassen, werden an dieser Stelle noch einmal kurz zusammengefasst.

- Beim $N-Recognizer sollte in diesem Kontext nur die Konfiguration $N=ttt oder $N=tft verwendet werden.
- Beim Einsatz des $N-Recognizer dürfen Symbole maximal aus vier bis fünf Strokes bestehen, da der Speicherbedarf sonst explosionsartig ansteigt.
- Bei Systemen mit stark beschränktem Arbeitsspeicher sollte deshalb nur der $P-Recognizer eingesetzt werden.
- Manche Zeichen, wie vor allem der Punkt, sind aufgrund ihrer Gestalt nur sehr schwer zu erkennen. Hierfür könnte der Algorithmus mit einer zusätzlichen Komponente erweitert werden, die zusätzlich die Größe des Zeichens miteinbezieht.
- Um hohe Erkennungsraten zu erzielen, sollte das Programm von derselben Person trainiert werden, die dies später auch verwendet.

5 Fazit

Im Rahmen dieser Arbeit wurden mehrere verschiedene Ansätze zur Erkennung handschriftlicher mathematischer Ausdrücke untersucht. Dabei wurden neben Programmen, die exakt für diese Aufgabe konzipiert sind, auch Implementierungen in Betracht gezogen, die eigentlich für die OCR gedruckter Schriftstücke entwickelt wurden. Es stellte sich jedoch heraus, dass es kein Open-Source-Programm gibt, das alle in Abschnitt 2.2 gestellten Anforderungen erfüllt. Darüber hinaus sind hier die Erkennungsraten oft unzureichend und eine Erweiterung, sodass alle benötigten Zeichen und Konstrukte unterstützt werden, ist oftmals nicht oder nur mit sehr viel Aufwand möglich. Das größte Hindernis hierbei sind meist die fehlenden Informationen zur Erstellung der Konfigurationsdateien, was es schwierig gestaltet, diese anzupassen. Ein weiteres Defizit stellen die teilweise nicht veröffentlichten Trainingsdaten dar.

Die genannte Problematik ergibt sich häufig bei Open-Source-Software, welche auf neuronalen Netzen basiert. Der Quelltext wird zwar als frei verwendbar veröffentlicht, allerdings gilt dies nicht für die zugrundeliegenden Trainingsdaten oder die Werkzeuge, die für das Training verwendet wurden. Dadurch ist es für den Anwender des Programms zwar möglich, es mit der mitgelieferten Konfiguration des neuronalen Netzes auszuführen, allerdings kann er diese nur sehr eingeschränkt oder nicht für seine eigenen Zwecke anpassen bzw. erweitern.

Zusammenfassend lässt sich sagen, dass die maschinelle Handschriftenerkennung vor allem im Bereich mathematischer Ausdrücke ein sehr schwerwiegendes Problem darstellt, für das es aktuell noch keine ausgereifte Lösung gibt [RB10a, MVGZG16, ZMVG16]. Besonders bei den Open-Source-Programmen gibt es noch viel Potential zur Verbesserung. Unter ihnen gibt es aktuell leider kein bestehendes Programm, welches alle benötigten Zeichen sowie Konstrukte unterstützt und zudem gute Erkennungsraten liefert. Bei der Analyse der Competition on Recognition of Online Handwritten Mathematical Expressions hat sich zudem herausgestellt, dass viele Tools im Rahmen von Dissertationen oder anderen universitären Arbeiten entwickelt wurden, wobei die Projekte mit Abschluss der Forschungsarbeit meist eingestellt wurden. Das folgende Zitat von Ray Smith spiegelt den aktuellen Stand auf diesem Gebiet sehr gut wieder.

> „If there are a lot of papers on a topic, there is most likely no good solution, at least not yet, so try to use something else.“ [Smi16]

Wir haben das Beste aus dieser Situation gemacht, indem wir als Alternative zu den komplexen Programmen, die auf neuronalen Netzen und Machine-Learning basieren, die Familie der $-Recognizer untersuchten. Diese einfach aufgebauten Algorithmen sind auf die Erkennung einzelner, isolierter Zeichen ausgerichtet. Anhand der detaillierten

Untersuchung der Funktionsweise in Abschnitt 3.2.7 stellte sich heraus, dass der $N- und der $P-Recognizer für die Erkennung handschriftlicher mathematischer Zeichen geeignet sind. Basierend auf diesen Erkennungsalgorithmen wurde eine von Grund auf neue Bibliothek konzipiert und implementiert, welche an unsere Anforderungen angepasst ist. Die entwickelte, flexibel einsetzbare C++-Bibliothek wurde mithilfe umfangreicher Testdaten, welche ausschließlich für diesen Zweck erhoben wurden, hinsichtlich ihrer Erkennungsrate evaluiert. Die in Kapitel 4.2 präsentierten Ergebnisse der Evaluierung machen deutlich, dass in der benutzerabhängigen Auswertung selbst bei dem verwendeten Zeichensatz von 120 Symbolen relativ gute Erkennungsraten erzielt werden können. Darüber hinaus ist ein großer Vorteil der $-Recognizer, dass neue Zeichen sehr einfach zur Laufzeit trainiert werden können. Das schwerwiegende Problem, dass immer nur einzelne Zeichen erkannt werden können, ließe sich durch die Entwicklung eines Wizards zur schrittweisen Eingabe mathematischer Ausdrücke beheben. Eine Möglichkeit wäre es hier, dem Anwender eine Art Raster mit Feldern auf dem Bildschirm anzuzeigen. In die Felder könnten dann einzelne Zeichen bzw. ganze Funktionsnamen wie *sin* oder *cos* eingegeben werden. Dadurch müsste die Eingabe nicht mehr segmentiert werden und die Zeichen können einzeln vom Algorithmus erkannt werden. Diese Erweiterung der $-Recognizer kann im Rahmen zukünftiger Arbeiten weiter fortgeführt werden. Die implementierte Dollar-Recongnizer-Bibliothek bietet hierfür zusammen mit ihrer Evaluierung eine gute Grundlage.

Literatur

[Áa] Francisco Álvaro Muñoz. Printed math expression parser. `https://github.com/falvaro/pme_parser`. Zugegriffen: 28.09.2017.

[Áb] Francisco Álvaro Muñoz. Seshat: Handwritten math expression parser. `https://github.com/falvaro/seshat`. Zugegriffen: 28.09.2017.

[Á15] Francisco Álvaro Muñoz. *Mathematical Expression Recognition based on Probabilistic Grammars*. Dissertation, Universitat Politècnica de València, 2015.

[ÁSB11] Francisco Álvaro Muñoz, Joan-Andreu Sánchez, and José-Miguel Benedí. Recognition of Printed Mathematical Expressions Using Two-dimensional Stochastic Context-Free Grammars. In *International Conference on Document Analysis and Recognition (ICDAR)*, pages 1225–1229, 2011.

[ÁSB14] Francisco Álvaro Muñoz, Joan-Andreu Sanchez, and Jose-Miguel Benedi. Offline Features for Classifying Handwritten Math Symbols with Recurrent Neural Networks. In *2014 22nd International Conference on Pattern Recognition*, pages 2944–2949. IEEE, 2014.

[ÁSB16] Francisco Álvaro Muñoz, Joan-Andreu Sánchez, and José-Miguel Benedí. An integrated grammar-based approach for mathematical expression recognition. *Pattern Recognition*, 51:135–147, 2016.

[AW10] Lisa Anthony and Jacob O. Wobbrock. A Lightweight Multistroke Recognizer for User Interface Prototypes. In *Proceedings of Graphics Interface 2010*, GI '10, pages 245–252, Toronto, Ont., Canada, Canada, 2010. Canadian Information Processing Society.

[AW12] Lisa Anthony and Jacob O. Wobbrock. $N-protractor: A Fast and Accurate Multistroke Recognizer. In *Proceedings of Graphics Interface 2012*, GI '12, pages 117–120, Toronto, Ont., Canada, Canada, 2012. Canadian Information Processing Society.

[AZ13] Francisco Alvaro and Richard Zanibbi. A Shape-based Layout Descriptor for Classifying Spatial Relationships in Handwritten Math. In *Proceedings of the 2013 ACM Symposium on Document Engineering*, DocEng '13, pages 123–126. ACM, 2013.

[BLL06] Klaus Braune, Joachim Lammarsch, and Marion Lammarsch. *LaTeX*. X.systems.press. Springer-Verlag, 2006.

[CFJ] Christian Schmitt, Frank Hannig, and Jürgen Teich. A Multi-layered Domain-specific Language for Stencil Computations.

[Cha13] Tom Chatfield. *50 Schlüsselideen Digitale Kultur*. Springer Spektrum, 2013.

[CN13] Amey Chavan and Asad Naik. Linear equation solver in Android using OCR. *IOSR Journal of Engineering*, 03:42–44, 2013.

[CS15] Dan Ciresan and Jürgen Schmidhuber. Multi-Column Deep Neural Networks for Offline Handwritten Chinese Character Classification. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–6, 2015.

[DDN16] Hai DAI NGUYEN, Anh DUC LE, and Masaki Nakagawa. Recognition of Online Handwritten Math Symbols Using Deep Neural Networks. *IEICE Transactions on Information and Systems*, E99.D:3110–3118, 2016.

[DLZ14] Kenny Davila, Stephanie Ludi, and Richard Zanibbi. Using Off-Line Features and Synthetic Data for On-Line Handwritten Math Symbol Recognition. In *ICFHR*, 2014.

[Fac] Facebook Open Source. Caffe2. `https://github.com/caffe2/caffe2`. Zugegriffen: 01.10.2017.

[Fac17] Facebook Open Source. Caffe2 - A New Lightweight, Modular, and Scalable Deep Learning Framework. `https://caffe2.ai/`, 23.09.2017. Zugegriffen: 2017-10-01.

[FL] Engine Falvarez and Frontend Luis A. Leiva. Mathematical Expression Recognition. `http://cat.prhlt.upv.es/mer/#publications`. Zugegriffen: 04.10.2017.

[Gra] Alex Graves. RNNLIB. `https://sourceforge.net/projects/rnnl/`. Zugegriffen: 28.09.2017.

[Gra14] Alex Graves. *Supervised Sequence Labelling with Recurrent Neural Networks*. Dissertation, Technischen Universität München, 2008-01-14.

[Gra13] Alex Graves. Generating Sequences With Recurrent Neural Networks. *CoRR*, abs/1308.0850, 2013.

[GS09] Alex Graves and Juergen Schmidhuber. Offline Handwriting Recognition with Multidimensional Recurrent Neural Networks. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *Advances in Neural Information Processing Systems 21*, pages 545–552. Curran Associates, Inc, 2009.

[HDÁZ] Lei Hu, Kenny Davila, Francisco Álvaro Muñoz, and Richard Zanibbi. RIT DPRL CROHME 2014. `https://github.com/DPRL/CROHME_2014`. Zugegriffen: 05.10.2017.

[HZ13] Lei Hu and Richard Zanibbi. Segmenting Handwritten Math Symbols Using AdaBoost and Multi-scale Shape Context Features. In *2013 12th International Conference on Document Analysis and Recognition*, pages 1180–1184. IEEE, 2013.

[JKS95] Ramesh C. Jain, Rangachar Kasturi, and Brian G. Schunck. *Machine vision*. McGraw-Hill series in computer science. McGraw-Hill, 1995.

[JSD+14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe. *CoRR*, abs/1408.5093, 2014.

[KN09] Santosh K.C. and Cholwich Nattee. A comprehensive survey on on-line handwriting recognition technology and its real application to the Nepalese natural handwriting. *Kathmandu University Journal of Science, Engineering, and Technology*, 5:31–55, 2009.

[LAB+14] Christian Lengauer, Sven Apel, Matthias Bolten, Armin Größlinger, Frank Hannig, Harald Köstler, Ulrich Rüde, Jürgen Teich, Alexander Grebhahn, Stefan Kronawitter, Sebastian Kuckuk, Hannah Rittich, and Christian Schmitt. ExaStencils. In *Proceedings of Euro-Par 2014: Parallel Processing Workshops*, volume 8806 of *Lecture Notes in Computer Science (LNCS)*, pages 553–564. Springer, 2014.

[Len] Christian Lengauer. Advanced Stencil-Code Engineering (ExaStencils). `http://www.exastencils.org`. Zugegriffen: 17.10.2017.

[Li10] Yang Li. Protractor. In Elizabeth D. Mynatt, Scott E. Hudson, and Geraldine Fitzpatrick, editors, *CHI Conference*, pages 2169–2172. Association for Computing Machinery, 2010.

[LS13] Zongyi Liu and Ray Smith. A Simple Equation Region Detector for Printed Document Images in Tesseract. In *2013 12th International Conference on Document Analysis and Recognition*, pages 245–249. IEEE, 2013.

[Mat] Maths for More SL. Store — wiris — math & science. `http://www.wiris.com/en/store`. Zugegriffen: 28.09.2017.

[ML13] Karanveer Mohan and Catherine Lu. Recognition of Online Handwritten Mathematical Expressions. Technical report, Stanford University, 2013-12-13.

[MVGK+11] H. Mouchere, C. Viard-Gaudin, D. H. Kim, J. H. Kim, and U. Garain. CROHME2011. In *2011 International Conference on Document Analysis and Recognition*, pages 1497–1500, 2011. Sept,.

[MVGK+12] Harold Mouchere, Christian Viard-Gaudin, D. H. Kim, J. H. Kim, and U. Garain. ICFHR 2012 Competition on Recognition of On-Line Mathematical Expressions (CROHME 2012). In *2012 International Conference on Frontiers in Handwriting Recognition*, pages 811–816. IEEE, 2012.

[MVGZ+13] Harold Mouchere, Christian Viard-Gaudin, Richard Zanibbi, Utpal Garain, Dae Hwan Kim, and Jin Hyung Kim. ICDAR 2013 CROHME. In *2013 12th International Conference on Document Analysis and Recognition*, pages 1428–1432. IEEE, 2013.

[MVGZG14] H. Mouchere, C. Viard-Gaudin, R. Zanibbi, and U. Garain. ICFHR 2014 Competition on Recognition of On-Line Handwritten Mathematical Expressions (CROHME 2014). In *2014 14th International Conference on Frontiers in Handwriting Recognition*, pages 791–796. IEEE, 2014.

[MVGZG16] H. Mouchere, C. Viard-Gaudin, R. Zanibbi, and U. Garain. ICFHR2016 CROHME. In *2016 15th International Conference on Frontiers in Handwriting Recognition (ICFHR)*, pages 607–612, 2016. Oct,.

[MyS] MyScript. MyScript Developer - Pricing for on-device and cloud handwriting recognition. `https://developer.myscript.com/pricing`. Zugegriffen: 28.09.2017.

[NG17] Chris Nicholson and Adam Gibson. Deep Learning Comp Sheet: Comparing Top Deep Learning Frameworks: Deeplearning4j, PyTorch, TensorFlow, Caffe, Keras, MxNet, Gluon & CNTK. `https://deeplearning4j.org/compare-dl4j-torch7-pylearn#caffe`, 29.09.2017. Zugegriffen: 01.10.2017.

[Pro16] Project Naptha. Tesseract.js — Pure Javascript OCR for 62 Languages! `http://tesseract.projectnaptha.com/`, 18.10.2016. Zugegriffen: 03.10.2017.

[PS00] R. Plamondon and S. N. Srihari. Online and off-line handwriting recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22:63–84, 2000. Jan,.

[RB10a] Sandip Rakshit and Subhadip Basu. Development of a multi-user handwriting recognition system using Tesseract open source OCR engine. *CoRR*, abs/1003.5886, 2010.

[RB10b] Sandip Rakshit and Subhadip Basu. Recognition of Handwritten Roman Script Using Tesseract Open source OCR Engine. *CoRR*, abs/1003.5891, 2010.

[SAL09] Ray Smith, Daria Antonova, and Dar-Shyang Lee. Adapting the Tesseract open source OCR engine for multilingual OCR. In Venu Govindaraju, Prem Natarajan, Santanu Chaudhury, and Daniel Lopresti, editors, *Proceedings of the International Workshop on Multilingual OCR - MOCR '09*, page 1. ACM Press, 2009.

[SDL] Evan Shelhamer, Jeff Donahue, and Jon Long. Caffe in a Day Tutorial. `https://docs.google.com/presentation/d/1HxGdeq8MPktHaPb-rlmYYQ723iWzq9ur6Gjo71YiG0Y/edit#slide=id.g1189b5eb19_4_67`. Berkeley Artificial Intelligence Research, Zugegriffen: 08.08.2017.

[SKC08] Fotini Simistira, Vassilis Katsouros, and George Carayannis, editors. *A Template Matching Distance for Recognition of On-Line Mathematical Symbols*, 2008.

[SKH+14] Christian Schmitt, Sebastian Kuckuk, Frank Hannig, Harald Köstler, and Jürgen Teich. ExaSlang. In *Proc.\ Int.\ Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*, pages 42–51. IEEE Computer Society, 2014.

[SKH+16] Christian Schmitt, Sebastian Kuckuk, Frank Hannig, Jürgen Teich, Harald Köstler, Ulrich Rüde, and Christian Lengauer. Software for exascale computing - SPPEXA 2013-2015. In Hans-Joachim Bungartz, Philipp Neumann, and Wolfgang E. Nagel, editors, *Software for exascale computing - SPPEXA 2013-2015*, number 113 in Lecture Notes in Computational Science and Engineering, pages 211–235. Springer International Publishing, 2016.

[Smi] Ray Smith. Tesseract OCR. `https://github.com/tesseract-ocr/tesseract`. Zugegriffen: 28.09.2017.

[Smi87] Ray Smith. *The extraction and recognition of text from multimedia document images*. Dissertation, University of Bristol, 1987.

[Smi07] Ray Smith. *Ninth International Conference on Document Analysis and Recognition, 2007*. IEEE Computer Soc, 2007.

[Smi09] Ray Smith. Hybrid Page Layout Analysis via Tab-Stop Detection. In *2009 10th International Conference on Document Analysis and Recognition*, pages 241–245. IEEE, 2009.

[Smi16] Ray Smith. Tesseract Blends Old and New OCR Technology. `https://github.com/tesseract-ocr/docs/tree/master/das_tutorial2016`, 2016. DAS2016 Tutorial - Santorin, Griechenland.

[SR10] Joachim Schenk and Gerhard Rigoll. *Mensch-Maschine-Kommunikation*. Springer, 2010. Rigoll, Gerhard (VerfasserIn).

[VAW12] Radu-Daniel Vatavu, Lisa Anthony, and Jacob O. Wobbrock. Gestures as Point Clouds. In Louis-Philippe Morency, editor, *Proceedings of the 14th ACM international conference on Multimodal interaction*, pages 273–280. ACM, 2012.

[WAV17] Jacob O. Wobbrock, Lisa Anthony, and Radu-Daniel Vatavu. $P Recognizer. `http://depts.washington.edu/madlab/proj/dollar/pdollar.html`, 17.01.2017. Zugegriffen: 30.09.2017.

[Wor11] World Wide Web Consortium. Ink Markup Language (InkML). `https://www.w3.org/TR/InkML/`, 20.09.2011. Zugegriffen: 27.09.2017.

[WWL07] Jacob O. Wobbrock, Andrew D. Wilson, and Yang Li. Gestures Without Libraries, Toolkits or Training: A $1 Recognizer for User Interface Prototypes. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology*, UIST '07, pages 159–168, New York, NY, USA, 2007. ACM.

[WWL16] Jacob O. Wobbrock, Andrew D. Wilson, and Yang Li. The $1 Unistroke Recognizer (JavaScript version). `http://depts.washington.edu/madlab/proj/dollar/dollar.js`, 21.12.2016. Zugegriffen: 29.09.2017.

[ZMVG16] Ting Zhang, Harold Mouchere, and Christian Viard-Gaudin. Online Handwritten Mathematical Expressions Recognition by Merging Multiple 1D Interpretations. In *2016 15th International Conference on Frontiers in Handwriting Recognition (ICFHR)*, pages 187–192. IEEE, 2016.

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 3. November 2017

_____

Daniel Ziegler