

**Datensynthese für die Erkennung
mathematischer Ausdrücke mit tiefen
neuronalen Netzen**

Bachelorarbeit

**Michael Frichert
6. März 2023**

Supervisors:

Fabian Wolf, M.Sc.

Prof. Dr.-Ing. Gernot A. Fink

Fakultät für Informatik
Technische Universität Dortmund
<http://www.cs.uni-dortmund.de>

INHALTSVERZEICHNIS

1	Einleitung	3
2	Grundlagen	5
2.1	Mustererkennung und maschinelles Lernen	5
2.2	Neuronale Netze	6
2.2.1	Perzeptron	6
2.2.2	Multi-Layer Perceptron	7
2.2.3	Aktivierungsfunktionen	9
2.2.4	Convolutional Neural Networks	11
2.2.5	Recurrent Neural Networks	14
2.3	Optimierung	15
2.3.1	Verlustfunktion	16
2.3.2	Gradient Descent	16
2.3.3	Backpropagation	18
2.3.4	Overfitting	20
3	Verwandte Arbeiten	23
3.1	Handschriftliche Mathematikererkennung	23
3.2	Datensynthese	26
4	Modell	29
4.1	Encoder	29
4.1.1	DenseNet	29
4.1.2	Dense-Encoder	31
4.2	Decoder	32
4.2.1	Transformer-Decoder	32
4.2.2	Multi-Head-Attention	33
4.2.3	Positionskodierung	35
4.2.4	Sprachmodellierung und Training	36
4.2.5	Inferenz	38
5	Datensynthese	41
5.1	Datenextraktion	41
5.2	Synthesizer	42
5.3	Handschriftsynthese	44
5.4	Datenaugmentierungen	46
5.4.1	Affine Transformationen	47

5.4.2	Random Warp Grid Distortion	49
6	Experimente	51
6.1	Datensätze	52
6.1.1	Evaluierung	52
6.1.2	Ausdrucksextraktion	53
6.2	Metriken	54
6.3	Trainingsaufbau	55
6.4	Ergebnisse	56
6.4.1	Baseline	56
6.4.2	Ausdrucksquellen	56
6.4.3	Syntheseverfahren	59
6.4.4	Augmentierungen	60
6.4.5	Reduktion der Trainingsdaten	62
7	Fazit	65
A	Anhang	67

EINLEITUNG

Mathematische Ausdrücke sind für viele Menschen ein fester Bestandteil im schulischen, akademischen oder beruflichen Alltag. Mit der zunehmenden Verbreitung von Computern, Tablets und Smartphones lassen sich mathematische Ausdrücke nicht mehr ausschließlich gedruckt oder handschriftlich, sondern auch in digitaler Form wiederfinden. Obgleich bereits Eingabemöglichkeiten wie LaTeX, MathML oder grafische Nutzeroberflächen existieren, um mathematische Ausdrücke digital zu erfassen, erfordern sie für die Anwendung einen zusätzlichen Lern- und Eingabeaufwand. Eine maschinelle Erkennung von handschriftlichen mathematischen Ausdrücken (HMA) ist eine Möglichkeit, welche bei der Erfassung sowohl den Eingabe- als auch den Lernaufwand reduziert. Weiterhin kann sie bei einer Transkription von Dokumenten helfen [CY00].

Die maschinelle Erkennung von HMA reicht bis in das Ende der 1960er-Jahre zurück und wurde erstmals von Robert Anderson in [And67] formuliert. Seither haben sich sowohl die Eingabegeräte als auch die Lösungsansätze stark weiterentwickelt. In der Informatik ordnet sich die maschinelle Erkennung von HMA thematisch gesehen im Teilgebiet der Mustererkennung ein. Im vergangenen Jahrzehnt konnten für die Mustererkennung mittels maschineller Lernverfahren im Bereich des Deep-Learnings mit künstlichen neuronalen Netzen Durchbrüche erzielt werden [LBH15].

Bei der CROHME¹ [MZM⁺19], einem Wettbewerb für die Erkennung von HMA, ließ sich bei den vergangenen Austragungen eine Zunahme von Deep-Learning-basierten Lösungsansätzen beobachten, welche eine gute Performanz bei der Erkennung erzielen konnten. Deep-Learning-basierte Lösungsansätze benötigen für das Training jedoch umfangreiche Datensätze. Eine Schwierigkeit ist, dass eine Herstellung von umfangreichen Datensätzen durch Menschenhand sowohl zeit- als auch kostenintensiv ist [KUNP19, KJ16a]. Eine Datensynthese, bei der künstliche Trainingsdaten automatisiert hergestellt werden, ist eine potentielle Möglichkeit, um den Aufwand bei dem Herstellungsprozess von Trainingsdaten zu verringern.

In dieser Arbeit werden zunächst, angelehnt an [KJ16a] und [DKLR17], synthetische Trainingsdaten für die Erkennung von HMA hergestellt. Anschließend werden die Auswirkungen auf die Performanz mittels der synthetischen Daten trainierten Deep-

¹ „Competition on Recognition of Online Handwritten Mathematical Expressions“

Learning-Modelle untersucht. Dazu wird der in der Forschung etablierte CROHME-Benchmark eingesetzt.

Das Ziel in dieser Arbeit ist zu untersuchen, ob die Nutzung von synthetischen Trainingsdaten die Performanz bei der Erkennung von HMA bei dem CROHME-Benchmark verbessern kann und gegebenenfalls der Bedarf an echten Trainingsdaten reduziert werden kann.

Im Folgenden werden in Kapitel 2 zunächst die Grundlagen der künstlichen neuronalen Netze und die daraus resultierenden grundlegenden Architekturen vorgestellt. In Kapitel 3 folgt eine Erläuterung von Schwierigkeiten bei der Erkennung von HMA und eine Vorstellung einzelner Erkennungssysteme, welche in den vergangenen Jahren veröffentlicht wurden. Im Zuge dessen wird die CROHME [MZM⁺19] vorgestellt, welche in dieser Arbeit eine zentrale Rolle einnimmt. Weiterhin werden die Idee der Datensynthese mit einem Fokus auf die Bereiche Dokumentenanalyse und Mathematikererkennung erläutert und verschiedene Arbeiten vorgestellt, wobei sich die Datensynthese in dieser Arbeit an zwei der vorgestellten Arbeiten [KJ16a, DKLR17] orientiert. Im darauffolgenden Kapitel 4 wird der Aufbau des BTTR²-Modells [ZGY⁺21], einem Modell für die Offline-Erkennung von HMA, erörtert. Anschließend wird in Kapitel 5 ein Datensyntheseverfahren präsentiert, das im finalen Kapitel 6 in Verbindung mit dem BTTR-Modell experimentell zur Beantwortung der Forschungsfrage evaluiert wird. In einem Fazit in Kapitel 7 erfolgt eine Zusammenfassung der in dieser Arbeit festgestellten Ergebnisse.

² „Bidirectionally Trained Transformer“

2.1 MUSTERERKENNUNG UND MASCHINELLES LERNEN

Die Mustererkennung ist ein Teilgebiet der Informatik und beschäftigt sich nach Bishop [Bis06, S.1] mit der algorithmengestützten Entdeckung von Regelmäßigkeiten in Daten und anschließender Nutzung der Regelmäßigkeiten für Einsatzzwecke wie der Klassifikation von Daten. Klassifikation ist ein Problem, bei dem, gegeben eine Eingabe x und eine Anzahl an k vordefinierten Klassen, eine Zuweisung der Eingabe auf eine der k Klassen erfolgen soll [Bis95, S. 5]. Im Kontext der Funktionsapproximation lässt sich für ein Klassifikationsproblem annehmen, dass es eine unbekannte Funktion $y = f(x)$ gibt, welche für die Eingabe x eine Klasse y zuweist [Mur12, S. 3]. Durch maschinelle Lernverfahren kann eine solche Funktion f mit einer Menge von Daten $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$, zu denen bereits die Klassenzugehörigkeit bekannt ist, den sogenannten Trainingsdaten und einem mathematischen Modell [GBC16, S. 122] eine Approximation \hat{f} von f berechnet werden [Mur12, S. 2 ff.]. Anschließend kann \hat{f} verwendet werden, um Klassenvorhersagen für Eingabedaten x zu treffen. Der Vorteil der maschinellen Lernverfahren ist, dass die für die Klassifikation notwendigen Regelmäßigkeiten automatisiert [Mur12, S. 2] entdeckt werden. Ein Beispiel für die mathematischen Modelle, welche sich für Klassifikationsprobleme eignen, sind Support-Vector-Machines [Vap10], Decision-Trees [Qui86] oder die im Folgenden vorgestellten künstlichen neuronale Netze.

Eine Erkennung von handschriftlichen mathematischen Ausdrücken (HMA) kann realisiert werden, indem ein Bild, welches einen HMA enthält, eingegeben wird und eine Ausgabe in Form eines Ausdrucks in LaTeX [Lam94] formuliert wird. Dabei handelt es sich nicht um ein klassisches Klassifikationsproblem, da zu einem Ausdruck keine vordefinierte Klasse zugewiesen wird, welche für den jeweiligen LaTeX-String steht. Vielmehr handelt es sich um eine Erweiterung des Klassifikationsproblems, bei dem die Ausgabe eine Sequenz von LaTeX-Tokens ist, wobei jedes Element in der Sequenz zuvor einer der vordefinierten LaTeX-Token-Klassen zugewiesen wurde.

2.2 NEURONALE NETZE

Künstliche neuronale Netze bezeichnen eine Klasse von mathematischen Modellen, welche sich dadurch auszeichnen, dass sie universelle Funktionsapproximatoren sind [HSW89, GBC16, S. 164]. Daher eignen sich diese Modelle für das zuvor beschriebene Klassifikationsproblem.

Seit dem Ende des 20. Jahrhunderts konnten auf neuronalen Netzen basierte Systeme bei Wettbewerben in unterschiedlichen Domänen für verschiedene Problemstellungen herkömmliche Systeme in der Performanz übertreffen [Sch14]. Ein prominentes Beispiel ist ein von Krizhevsky et al. in [KSH12] vorgestelltes Convolutional Neural Network, das bei der ILSVRC¹ im Jahr 2012 [RDS⁺15] den ersten Platz bei der Klassifikations- und Lokalisierungsaufgabe belegte. Seither sind neuronale Netze für Probleme in diesem Bereich die dominanten Lösungsansätze [LBH15].

2.2.1 Perzeptron

Als Grundidee für die neuronalen Netze dient das von Frank Rosenblatt im Jahr 1958 in [Ros58] formulierte Perzeptron. Das Perzeptron ist ein mathematisches Berechnungsmodell und verallgemeinert durch die Einführung von Gewichten das im Jahr 1943 von Warren McCulloch und Walter Pitts in [MP43] vorgestellte Berechnungsmodell, welches durch echte biologische Neuronen inspiriert wurde [Roj96, S. 30, S. 55]. Ein Perzeptron lässt sich nach [KBB⁺15, S. 13] folgendermaßen definieren:

Seien die Eingaben x_1, \dots, x_n , die Gewichte w_1, \dots, w_n und der Schwellwert θ reellwertige Zahlen, dann lässt sich ein Perzeptron wie folgt definieren:

$$y = \begin{cases} 1 & \text{falls } \sum_{i=1}^n x_i \cdot w_i \geq \theta, \\ 0 & \text{sonst.} \end{cases} \quad (2.2.1)$$

Das Perzeptron nimmt eine Anzahl an Eingaben x_1, \dots, x_n und Gewichte w_1, \dots, w_n entgegen und bildet eine gewichtete Summe. Anschließend berechnet das Perzeptron eine 1, falls die Summe mindestens so groß wie der Schwellwert θ des Perzeptrons ist und 0 andernfalls. Das Perzeptron teilt somit den Eingaberaum in zwei Teile: einen Teil, für den die Berechnung der Eingaben 0 ergibt und einen anderen Teil, für den die Berechnung der Eingaben 1 ergibt [Roj96, S. 60].

Mit Definition 2.2.1 lassen sich bereits durch die Wahl geeigneter Gewichte w_1, \dots, w_n (mit Einschränkung der Variablen $x_1, \dots, x_n \in \{0, 1\}$) und eines Schwellwertes θ

¹ „ImageNet Large Scale Visual Recognition Challenge“

die logischen Funktionen AND, OR und NAND modellieren [Mit97, S. 87]. Allerdings existieren Funktionen wie die XOR-Funktion, welche sich nicht durch ein einzelnes Perzeptron modellieren lässt, da der Eingaberaum bei einer solchen Funktion nicht linear separierbar² ist [Mit97, S. 87].

Die XOR-Funktion, oder im Allgemeinen jede boolesche Funktion, lässt sich jedoch durch Zunahme weiterer Perzeptren modellieren, indem die Perzeptren miteinander vernetzt werden [Mit97, S. 87 ff.]. Bei einer solchen Vernetzung dient die Ausgabe eines oder mehrerer Perzeptren als Eingabe für weitere Perzeptren. Die Idee der Vernetzung ist die Grundlage für die heutigen neuronalen Netze und wird in Abschnitt 2.2.2 genauer betrachtet.

Die Definition 2.2.1 des Perzeptrons ist inzwischen überwiegend von historischer Relevanz. Heutzutage wird das Perzeptron (im Folgenden auch (künstliches) Neuron) verallgemeinert formuliert als (Definition in Anlehnung an [LWG])

$$y = f\left(\sum_{i=1}^n x_i \cdot w_i + b\right) = f(\mathbf{w} \cdot \mathbf{x} + b), \quad (2.2.2)$$

wobei f eine sogenannte Aktivierungsfunktion ist, deren Bedeutung in Abschnitt 2.2.3 näher erläutert wird. Anstelle eines Schwellwertes gibt es einen sogenannten Bias b , welcher unabhängig von den Eingaben x_1, \dots, x_n addiert wird. Weiterhin lassen sich die Eingaben und Gewichte zu Vektoren $\mathbf{x} = (x_1, \dots, x_n)^T$ und $\mathbf{w} = (w_1, \dots, w_n)^T$ zusammenfassen.

Ähnlich wie bei dem Perzeptron aus Definition 2.2.1 ist es mit der Vernetzung künstlicher Neuronen und der Wahl geeigneter Gewichte und Aktivierungsfunktionen möglich, jede stetige Funktion zu approximieren [KBB⁺15, S. 55].

2.2.2 Multi-Layer Perceptron

Multi-Layer Perceptrons oder auch (Deep-)Feedforward-Neural-Networks sind Modelle zur Approximation von Funktionen [GBC16, S. 164] und bestehen aus einer Menge von vernetzten Neuronen, welche zusammen Schichten (Layer) formen. Abbildung 2.2.1 skizziert den Aufbau eines vierschichtigen Feed-Forward-Networks (FFN) als einen azyklischen Graphen. Die Knoten im Graphen stehen für Neuronen und die gerichteten Kanten geben an, welche Ausgaben als Eingaben für weitere Neuronen dienen. Die Bezeichnung Feed-Forward bedeutet, dass es keine Rückverbindungen gibt und somit keine Ausgaben oder Zwischenrechnungen zurück in das Modell geführt werden [GBC16, S. 164].

² Eine formale Definition für die lineare Separierbarkeit befindet sich in [Roj96, S. 63].

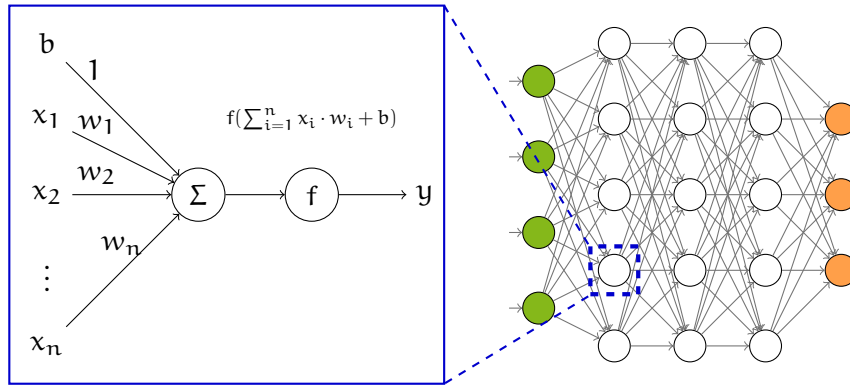


Abbildung 2.2.1: Grafische Veranschaulichung eines neuronalen Netzes. Links: Berechnungen eines einzelnen Neurons. Rechts: Die Vernetzung mehrerer Neuronen. Grafik in Anlehnung an [Vel23].

Die einzelnen Schichten lassen sich in eine Eingabeschicht (Input-Layer), drei versteckte Schichten (Hidden-Layer) und eine Ausgabeschicht (Output-Layer) aufteilen. Die Eingabe in das Netz wird von den Neuronen im Input-Layer entgegengenommen und an die Neuronen im ersten Hidden-Layer, ohne dass Berechnungen stattfinden, weitergereicht [LWG]. Aus diesem Grund wird der Input-Layer meistens nicht mitgezählt. In den Hidden-Layern werden dann Berechnungen auf Basis der Ausgaben der Neuronen im Vorgänger-Layer durchgeführt und anschließend im Output-Layer ausgegeben. Die Anzahl der Schichten wird als Tiefe (Depth) bezeichnet. Die einzelnen Schichten werden in dem Modell als Fully-Connected (FC) bezeichnet, da die Neuronen in einer Schicht als Eingabe die Ausgaben aller Neuronen der Vorgängerschicht bekommen [LWG].

Bei einer Eingabe $x = (x_1, \dots, x_n)$ als ein Spaltenvektor in ein dreischichtiges FFN mit FC Schichten werden folgende Operationen ausgeführt (angelehnt an [VSP⁺17]):

$$\text{FFN}(x) = \mathbf{W}_3 f_2(\mathbf{W}_2 f_1(\mathbf{W}_1 x + \mathbf{b}_1) + \mathbf{b}_2) + \mathbf{b}_3 . \quad (2.2.3)$$

Die Gewichte der Neuronen einer Schicht i werden jeweils als eine Zeile in einer Gewichtsmatrix \mathbf{W}_i zusammengefasst und ein Bias-Spaltenvektor \mathbf{b}_i enthält in jeder Zeile die passenden Bias der in \mathbf{W}_i zusammengefassten Neuronen. f_i ist eine Aktivierungsfunktion und wird üblicherweise, abhängig von der Definition von f_i , zeilenweise ausgeführt [GBC16, S. 171]. Obwohl 2.2.3 implizieren könnte, dass die Aktivierungsfunktionen f_1, f_2, f_3 unterschiedlich sind, ist es zwar nicht ausgeschlossen

unterschiedliche Aktivierungsfunktionen in einem FFN zu verwenden, jedoch in der Regel nicht üblich [LWG].

Mit 2.2.3 könnte man nach [LWG] bereits einen einfachen Bildklassifikator entwerfen, indem das Modell in der Eingabeschicht einen Spaltenvektor x mit Pixelwerten eines Bildes entgegennimmt und in der Ausgabeschicht einen k -dimensionalen Vektor ausgibt, welcher für jede der k Klassen jeweils einen Score angibt. Anschließend wird der Index mit dem größten Score als die vorhergesagte Klasse betrachtet [LBH15, Nie15, Kapitel 1]. Damit ein solcher Klassifikator jedoch möglichst genau die Eingaben auf die k Klassen abbildet, ist es nötig, die Bias und Gewichte oder im Folgenden auch Parameter geeignet zu wählen bzw. zu lernen.

2.2.3 Aktivierungsfunktionen

Die Wahl der Aktivierungsfunktion f eines Neurons ist eine relevante Entscheidung bei der Architektur bzw. dem Entwurf eines Modells. Die Anforderungen an eine Aktivierungsfunktion können in Abhängigkeit der Art und Architektur des Modells variieren. Die wichtigste Anforderung ist jedoch, dass eine Aktivierungsfunktion nichtlinear ist, denn andernfalls ließen sich die Gewichts-Matrizen unterschiedlicher Layer zu einer einzelnen Matrix zusammenfassen, sodass das Modell zu einem linearen Klassifikator wird [LWG, Bis95, S.121]. Im Hinblick auf die in Abschnitt 2.3 vorgestellte Optimierung ist ebenfalls die Differenzierbarkeit der Aktivierungsfunktion notwendig. In der Literatur ist die verwendete Aktivierungsfunktion oftmals namensgebend für das jeweilige Neuron. So wird beispielsweise ein Neuron, das die Sigmoid-Funktion verwendet, als Sigmoid-Neuron bezeichnet.

Abbildung 2.2.2 veranschaulicht drei geläufige Aktivierungsfunktionen: Sigmoid, tanh und ReLU. Diese werden im Folgenden genauer betrachtet.

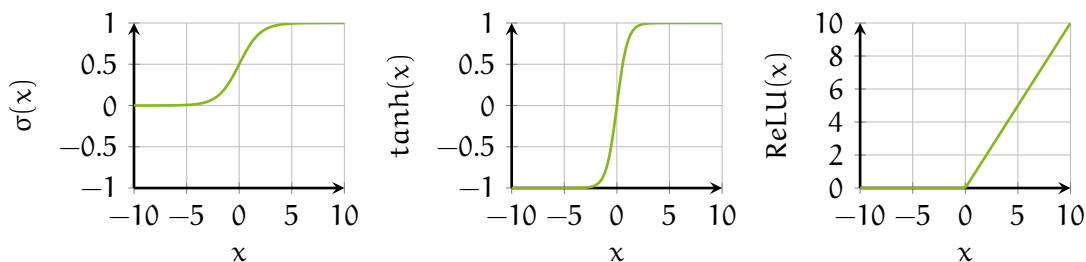


Abbildung 2.2.2: Graphen drei verschiedener Aktivierungsfunktionen: links: Sigmoid, Mitte: tanh, rechts: ReLU.

Die Sigmoid-Funktion σ ist definiert als:

$$\sigma(x) = \frac{1}{1 + e^{-x}} . \quad (2.2.4)$$

Betrachtet man die Fallunterscheidung des Perzeptrons aus Definition 2.2.1 mit Schwellwert $\theta = 0$ als Stufenfunktion, so ähnelt die Sigmoid-Funktion ihr [Nie15, Kapitel 1]. Der Unterschied ist jedoch, dass die Sigmoid-Funktion stetig ist [Bis95, S. 118] und die Eingabe x auf das gesamte Intervall $[0; 1]$ abbildet.

Die tanh-Funktion ist ähnlich wie die Sigmoid-Funktion definiert, mit dem Unterschied, dass sie x auf das Intervall $[-1; 1]$ abbildet [GBC16, S. 191]:

$$\tanh(x) = 2\sigma(2x) - 1 = \frac{2}{1 + e^{-2x}} - 1 . \quad (2.2.5)$$

Ein Problem, das bei der Optimierung unter der Verwendung der Sigmoid- oder tanh- Funktion in Modellen mit vielen Layern auftritt, ist das sogenannte Vanishing-Gradient-Problem³ [Sch14]. Die Ursache für das Problem ist, dass für die Ableitung der Sigmoid-Funktion $\sigma'(x) < 1$ gilt [HS97]. Dies hat zur Folge, dass bei der in Abschnitt 2.3.3 vorgestellten Backpropagation aufgrund der Anwendung der Kettenregel der Gradient 0 annähert und die Optimierung mit dem Gradient-Descent erschwert wird [HS97]. Selbiges tritt bei der tanh-Funktion auf, da $\tanh'(x) \leq 1$ gilt und nur für den Fall $\tanh'(0) = 1$ keine Verringerung während der Anwendung der Kettenregel stattfindet.

Nichtsdestotrotz ist die Nutzung beider Aktivierungsfunktionen im Allgemeinen nicht ausgeschlossen, denn für Architekturen im Bereich der Recurrent Neural Networks erweist sich die Nutzung der Funktionen als sinnvoll [GBC16, S.192].

Eine Aktivierungsfunktion, welche sich vom Aufbau von den bisher vorgestellten Funktionen unterscheidet, ist die Rectified Linear Unit (ReLU) oder im Folgenden auch: ReLU-Funktion. Sie ist definiert als:

$$\text{ReLU}(x) = \max(0, x) . \quad (2.2.6)$$

Die ReLU-Funktion hat für $x \geq 0$ den Wert x und 0 andernfalls. In dem vergangenen Jahrzehnt fand die ReLU-Funktion eine zunehmende Verbreitung [RN21, S. 810]. Einerseits gestaltet sich die Berechnung aus technischer Sicht einfacher, da für die Berechnung ausschließlich das Vorzeichen von x beachtet werden muss und im Vergleich zur Sigmoid- oder tanh- Funktion keine Potenzierung notwendig ist [GBC11, LWG].

³ Obwohl das Problem vermehrt in Verbindung mit Recurrent Neural Networks adressiert wird (siehe [HS97]), so kann es auch bei FFN mit vielen Layern auftreten [GBC16, S.286 ff.].

Andererseits vermeiden ReLU-Neuronen das Vanishing-Gradient-Problem [GBB11]. Im Vergleich mit den zuvor vorgestellten Funktionen hat die Ableitung der ReLU-Funktion entweder den Wert 0 für $x < 0$ oder den Wert 1 für $x > 0$ und ist für $x = 0$ nicht differenzierbar⁴ [GBC16, S. 188]. Ein Nachteil ist jedoch, dass es während der Optimierung vorkommen kann, dass die Gewichte und Bias so angepasst werden, sodass das jeweilige Neuron für weitere Trainingsdaten keinen Wert größer 0 annimmt und inaktiv ist [LWG].

2.2.4 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) sind eine Form von FFNs und eignen sich besonders gut für die Erkennung von Bilddaten. Erste Ansätze für CNNs wurden bereits Anfang der 1980er-Jahre von Fukushima et al. in [FM82] vorgestellt und später mehrfach von LeCun im Laufe der 1990er-Jahre, beispielsweise in [LBBH98], aufgegriffen. Die Arbeit von Fukushima und LeCuns Arbeiten beschäftigten sich unter anderem mit der Erkennung von Ziffern.

In CNNs wird die Eingabe, anders als der im Abschnitt 2.2.2 vorgestellte einfache Bildklassifikator, nicht als Vektor, sondern als ein Volumen $H \times B \times D$ mit einer Höhe H , Breite B und Tiefe D interpretiert. Kern der CNNs sind die Convolutional-Layer, welche mit ihren Neuronen solche Volumen entgegennehmen und transformieren [LWG]. Die Besonderheit der Layer ist, dass in ihnen die räumliche Struktur der Eingabe berücksichtigt wird, denn ein Neuron in einem Convolutional-Layer ist nur mit einem lokalen Teil des Eingabevolumens, dem sogenannten Receptive-Field des Neurons, vernetzt [LWG, LBBH98]. Die lokale Vernetzung erlaubt die Extraktion einfacher visueller Merkmale (Features) in den niedrigen Layern, welche in den höheren Layern zu komplexen Merkmalen zusammengesetzt werden [LBBH98].

In einem Convolutional-Layer sind die Neuronen zweidimensional angeordnet und haben jeweils unterschiedliche Receptive-Fields. Dabei führen die Neuronen auf den Werten des Eingabevolumens, welche im Receptive-Field des Neurons liegen, die aus Definition 2.2.2 bekannte Operation aus. Die Ausgaben der Neuronen formen eine Ebene, welche als Feature-Map bezeichnet wird [LBBH98]. Die Neuronen teilen sich untereinander dieselben Gewichte und realisieren damit eine Faltung (Convolution), gefolgt von der Addition eines Bias und der Anwendung einer Aktivierungsfunktion [LBBH98]. Aufgrund dessen bezeichnet man die Gewichte als Filter oder Kernel [Sch14]. Das Teilen der Gewichte ist ein wichtiges Architekturmerkmal in CNNs, da

⁴ Da in der Praxis Berechnungen, beispielsweise aufgrund von Rundungen, nicht exakt sind, kann in diesem Fall die Ableitung als 0 oder 1 definiert werden [GBC16, S. 187 ff.]

es, im Vergleich zu einem FFN mit FC-Layern, die Anzahl der Parameter drastisch reduziert und somit eine bessere Skalierung bezüglich der Eingaben und der Anzahl der Layer im Modell ermöglicht [LWG, GBC16, S. 333].

Mathematisch ist die zuvor beschriebene diskrete Faltungsoperation definiert als [GBC16, S. 329]

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(i + m, j + n) \cdot K(m, n), \quad (2.2.7)$$

wobei I die Eingabe in den Convolutional-Layer, K ein Filter und S die Feature-Map ist. Es ist möglich, mehrere solcher Filter in einem Convolutional-Layer zu verwenden, deren Feature-Maps durch eine Konkatenation in der Tiefe das Ausgabevolumen des Layers formen. Unterschiedliche Filter teilen sich die Gewichte jedoch nicht [LBBH98]. Abbildung 2.2.3 veranschaulicht die Convolution-Operation mit einer Eingabe I und Gewichten des Filters K . Nach der Berechnung von $I * K$ gilt es den Bias zu addieren und eine Aktivierungsfunktion anzuwenden.

Bei dem Entwurf eines CNN sind für jeden Convolutional-Layer die Hyperparameter: Filtergröße F , Stride S und die Anzahl der Filter zu wählen.

Die Filtergröße F beschreibt die Höhe und Breite der Receptive-Fields aller Filter im Layer. Die Tiefe wird durch F nicht festgelegt, da sich die Filter immer über die gesamte Tiefe des Eingabevolumens erstrecken [LWG]. Bei einer Eingabetiefe $d = 3$ und einer Filtergröße von beispielsweise $F = 5$ wird somit ein $5 \times 5 \times 3$ großer Filter über die Höhe und Breite der Eingabe durch die Conv.-Operation geschoben und eine Feature-Map erzeugt.

Die Schrittweite, welche die Abstände der Receptive-Fields regelt, wird als Stride S bezeichnet. Dabei kann S so gewählt werden, beispielsweise $S = 1$, dass sich die Receptive-Fields (Abhängig von der Filtergröße F) überlappen. Die Stride hat einen Einfluss auf die Höhe und Breite der resultierenden Feature-Map. Bereits mit einer kleinstmöglichen Stride $S = 1$, ist die Höhe und Breite leicht verringert und verringert sich weiterhin mit zunehmender Stride. Um diesen Effekt zu verhindern und die Ausgabedimensionen zu steuern, ist es üblich, die Eingabe vor der Convolution mit einem Wert, beispielsweise 0, aufzufüllen (siehe [HZRS16]). Dieses Vorgehen ist als (Zero-) Padding bekannt [LWG].

Nachdem die Stride S und Filtergröße F bei dem Entwurf eines Convolutional-Layers gewählt wurde, gilt es die Anzahl der zu nutzenden Filter und somit die Tiefe des Ausgabevolumens des Convolutional-Layers zu wählen.

Neben der Wahl einer ausreichend großen Stride S , um die Höhe und Breite des Ausgabevolumens zu reduzieren, wie es vorwiegend in [HZRS16] eingesetzt wird, gibt eine weitere Operation unter dem Namen Pooling. Pooling-Layer werden zwi-

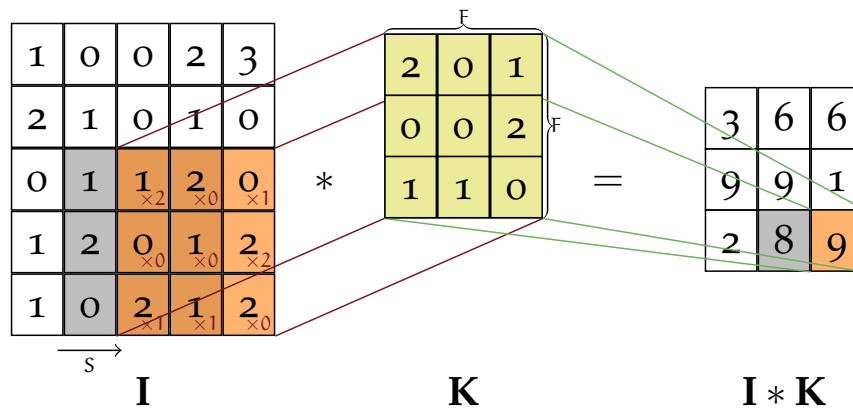


Abbildung 2.2.3: Convolution-Operation ohne Padding für ein $5 \times 5 \times 1$ großes Eingabevolumen I mit einem $3 \times 3 \times 1$ großen Filter K mit Stride $S = 1$. Grafik in Anlehnung an [Vel23].

schen Convolutional-Layer eingeschoben, um die Zahl der Parameter in den Folge-Layern zu reduzieren [LWG]. In einem Pooling-Layer werden, ähnlich wie in einem Convolutional-Layer, lokale Teile des Eingabevolumens betrachtet. Der Unterschied zu den Convolutional-Layern ist jedoch, dass keine lineare Operation auf Werten in einem lokalen Teil angewendet wird, sondern eine feste Pooling-Funktion ohne Parameter. Die Funktion berechnet dann beispielsweise das Maximum (Max-Pooling) oder den Mittelwert (Avg.-Pooling) der im lokalen Teil befindlichen Werte. Weiterhin erstreckt sich die Anwendung der Pooling-Funktion nicht über die gesamte Tiefe D , sondern findet ebenenweise statt, sodass für jede Feature-Map in der Tiefe des Eingabevolumens ein Wert berechnet wird [LWG, LBBH98]. Pooling-Layer transformieren somit die Feature-Maps aus dem Vorgänger-Layer in der Höhe und Breite. Ähnlich wie bei einem Convolutional-Layer ist bei dem Entwurf eines Pooling-Layers die Fenstergröße F (analog zur Filtergröße), die Stride S und zusätzlich die Funktion festzulegen. Pooling-Layer ermöglichen dem Modell eine gewisse Invarianz bezüglich der Positionen bestimmter Merkmale [GBC16, S. 336 ff.]. Abbildung 2.2.4 veranschaulicht die Anwendung zweier unterschiedlicher Pooling-Funktionen auf einem Eingabevolumen mit einer Feature-Map.

Nachdem mit Hilfe der Convolutional-Layer charakteristische Merkmale extrahiert und mittels der Pooling-Layer die Größe der Eingabe reduziert wurde, werden oftmals wenige FC-Layer zur Bestimmung der Klasse eingesetzt (siehe [KSH12, SZ15, HLMW17, HZRS16]).

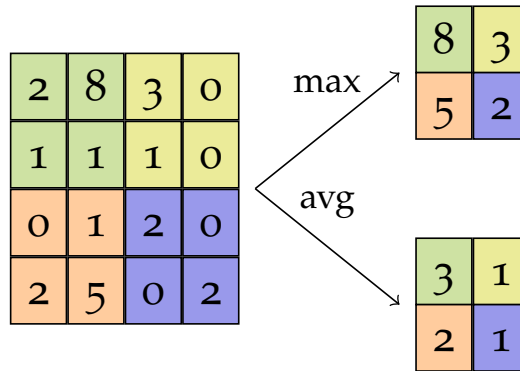


Abbildung 2.2.4: Darstellung zwei verschiedener Pooling-Operationen auf einer $4 \times 4 \times 1$ großen Eingabe mit Fenstergröße 2 und Stride 2. Die Fenster in der Eingabe und die resultierende Ausgabe sind jeweils farblich gekennzeichnet. Grafik in Anlehnung an [LWG].

2.2.5 Recurrent Neural Networks

Recurrent-Neural-Networks (RNNs) sind das Gegenstück zu den in 2.2.2 vorgestellten FFN, denn anders als in FFNs erlauben RNNs zeitversetzte Rückverbindungen zwischen Neuronen und können somit Berechnungen auf Basis vergangener Berechnungen treffen [PMB13]. Dies ist hilfreich, falls die Eingaben oder Ausgaben in einem neuronalen Netz Sequenzen mit variablen Längen sein sollen [CGCB14], wie es beispielsweise bei Problemen im Bereich des Natural-Language-Processing [HM15] oder der Handschrifterkennung [GFGSo6] der Fall ist.

Ein RNN lässt sich für eine Eingabesequenz $X = (x^{(1)}, \dots, x^{(n)})$ der Länge n verallgemeinert mit einer Rekursionsformel [CMG⁺14]

$$\mathbf{h}^{(t)} = f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}) \quad (2.2.8)$$

formulieren, wobei f die linearen Matrix-Vektor Operationen mit der Anwendung einer nichtlinearen Funktion beschreibt⁵. Die Rekursionsformel berechnet zu jedem Zeitpunkt t aus $1, \dots, n$ einen Hidden-State-Vektor $\mathbf{h}^{(t)}$ und realisiert damit die Rückverbindung, welche es ermöglicht, Informationen über vergangene Eingaben aufrechtzuerhalten [PMB13]. Eine Ausgabesequenz lässt sich durch Transformationen der berechneten Hidden-States erzeugen [LWG].

⁵ Beispiel: $f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}) = \tanh(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t)$ [LWG].

Zur Realisierung der rekurrenten Operationen in f gibt es RNN-Architekturen wie die „Long Short-term Memory“ [HS97] oder „Gated Recurrent Units“ [CMG⁺14] mit jeweils unterschiedlichen Funktionsweisen.

RNNs finden sich oftmals als Decoder in sogenannten Encoder-Decoder Architekturen wieder. Der Encoder bildet eine Eingabe auf eine oder mehrere Repräsentationen $Z = (z^{(1)}, \dots, z^{(n)})$ ab [VSP⁺17]. Der Decoder erzeugt daraufhin mittels Z eine Ausgangssequenz $Y = (y^{(1)}, \dots, y^{(m)})$ und berücksichtigt zusätzlich bei einer Vorhersage von y_t die Vorhersage aus y_{t-1} [VSP⁺17].

Des Öfteren nutzen RNN einen sogenannten Attention-Mechanismus [VSP⁺17]. Bei dem Attention-Mechanismus [BCB15] wird zu einem Zeitpunkt t , an Stelle der direkten Nutzung von Z , ein Kontextvektor c_t mittels einer gewichteten Summe aus den Elementen in Z berechnet. Der Kontextvektor c_t hebt wichtige Elemente aus Z hervor, welche für die Erzeugung des Hidden-States und der Ausgabe im Decoder zum Zeitpunkt t relevant sind [BCB15]. Die Gewichte für die Erzeugung von c_t werden während der Optimierung ermittelt.

2.3 OPTIMIERUNG

In den bisherigen Abschnitten wurde der grundlegende Aufbau von neuronalen Netzen vorgestellt und gezeigt, wie es mit Hilfe solcher Netze möglich ist, Bildklassifikatoren zu entwerfen. Dieses Kapitel behandelt die Anpassung der Parameter während des Trainings von neuronalen Netzen mit dem Ziel, möglichst gute Klassenvorhersagen sowohl für die Trainingsdaten als auch später für das Modell zuvor unbekannte Daten zu treffen.

Die Optimierung steht im Mittelpunkt bei dem Training bzw. Lernen des Modells. Als Optimierung wird die Maximierung oder Minimierung einer Zielfunktion bezeichnet [GBC16, S. 80 ff]. Sie gestaltet sich für neuronale Netze als schwierig, da die Zielfunktionen für neuronale Netze in der Regel nicht konvex sind [LWG].

Im Folgenden wird eine Verlustfunktion formuliert und der Gradientenabstieg (Gradient Descent), ein Algorithmus, um eine solche Verlustfunktion zu minimieren, vorgestellt [Bis95, S. 95]. Dieser nutzt, wie der Name andeutet, Informationen über den Gradienten der Verlustfunktion aus. Zur algorithmischen Berechnung eines Gradienten in einem neuronalen Netz wird die Backpropagation vorgestellt. Anschließend wird das Problem des Overfittings diskutiert, das infolge der in dieser Arbeit besprochenen Optimierung auftreten kann und Möglichkeiten vorgestellt, um das Problem zu vermindern.

2.3.1 Verlustfunktion

Um nachzuvollziehen, wie sicher ein Modell die Klassenvorhersage für ein Eingabedatum x_i trifft, lässt sich die Zielfunktion als eine Verlustfunktion (Loss Function) formulieren, welche einen hohen Wert annimmt, wenn die vorhergesagte Klasse \hat{y}_i der Eingabe nicht mit der tatsächlichen Klasse y_i (im Folgenden Label) übereinstimmt [LWG]. Somit ist das Ziel der in dieser Arbeit behandelten Optimierung, die Verlustfunktion zu minimieren. Das Label y_i lässt sich als sogenannter One-Hot-Vektor kodieren, dessen Komponenten die k Klassen repräsentieren, wobei die Komponente für das Label y_i den Wert 1 hat und die restlichen Komponenten den Wert 0 haben [CMG⁺14, SF16].

Sei \mathbf{f} ein k -dimensionaler Vektor mit Klassen-Scores für eine Eingabe x_i mit Label y_i , dann ist für ein einzelnes x_i der Cross-Entropy-Loss definiert als (Definition in Anlehnung an [LWG] und [GBC16, S. 180 ff.]):

$$L_i = -\log(\text{softmax}(x_i)_{y_i}) , \quad \text{softmax}(x)_j = \frac{e^{f_j}}{\sum_k e^{f_k}} . \quad (2.3.1)$$

Die Softmax-Funktion wird verwendet, um eine Wahrscheinlichkeitsverteilung über die k Klassen zu repräsentieren [GBC16, S. 180] und bildet die reellwertigen Scores im Vektor \mathbf{f} auf Pseudowahrscheinlichkeiten für die jeweilige Klassenzugehörigkeit ab. Zur Berechnung der Unterschiede zwischen der Verteilung der Trainingsdaten p und der geschätzten Verteilung des Modells q wird dann die Cross-Entropy $H(p, q) = -\sum_x p(x) \cdot \log(q(x))$ verwendet [LWG, GBC16, S. 130]. Aus diesem Grund eignet sich die Cross-Entropy als Verlustfunktion, denn wenn diese minimal ist, so trifft das Modell Vorhersagen, welche wenig von den Labels der Trainingsdaten abweichen. Da die Eingaben jeweils mit Wahrscheinlichkeit 1 genau einer Klasse y_i angehören, ist in diesem Fall für den Cross-Entropy-Loss nur die vorhergesagte Wahrscheinlichkeit des Labels y_i relevant.

Um den Loss für N Daten zu berechnen wird eine Summe über die Losses L_i berechnet $L = \sum_{i=1}^N L_i$ [Bis06, S. 242] oder das arithmetische Mittel [LWG] gebildet.

2.3.2 Gradient Descent

Die Gradient Descent Methode basiert auf der Idee, dass eine mit Hilfe von Trainingsdaten aufgestellte Verlustfunktion $L(\mathbf{w})$ minimiert werden kann, indem untersucht wird, welchen Einfluss marginale Veränderungen der Parameter auf den Funktionswert der Verlustfunktion haben [LBBH98]. Eine solche Beziehung des Funktionswertes hinsichtlich der Parameter im Modell wird durch den Gradienten der Funktion beschrieben

[LBBH98]. Der Gradient $\nabla_{\mathbf{w}}L(\mathbf{w})$ ist ein Vektor und enthält als Komponenten die partiellen Ableitungen der Funktion L nach den Gewichten \mathbf{w} [GBC16, S.82]. Die Parameter als ein Vektor \mathbf{w} werden nach folgender Vorschrift iterativ angepasst (Definition nach [GBC16, S. 83] und [LBBH98]):

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \epsilon \nabla_{\mathbf{w}_t} L(\mathbf{w}_t) . \quad (2.3.2)$$

Da die partiellen Ableitungen im Gradientenvektor Aufschluss über die Steigungen von $L(\mathbf{w})$ für die Parameter in \mathbf{w} geben, wird diese Information verwendet, um die Parameter \mathbf{w} jeweils in negativer Steigungsrichtung anzupassen. Die Learning-Rate ϵ gibt an, in welcher Schrittgröße die Anpassung der Gewichte erfolgen soll [Mit97, S. 91], da der Gradient in einem Punkt nur die Richtung und nicht die Entfernung angibt, in der sich ein Minimum befinden könnte [LWG].

Abbildung 2.3.1 veranschaulicht das Vorgehen für eine Funktion mit einem Parameter. Zu Beobachten ist, dass die Wahl von ϵ ein entscheidender Faktor bei der Annäherung an ein Minimum ist. Wird ϵ zu klein gewählt, sind viele Iterationen für die Annäherung notwendig. Andernfalls oszilliert der Parameter-Vektor \mathbf{w} bei einer zu großen Wahl um ein Minimum herum [Bis95, S. 95]. Um eine Oszillation nahe einem Minimum zu vermeiden, ist es üblich, die Learning-Rate nach einer Anzahl an Iterationen, beispielsweise um einen Faktor 0,1 zu verringern [LWG]. Neben der einfachen Anpassungsregel in Definition 2.3.2 gibt es komplexere Regeln wie ADAM [KB15], welche die Learning-Rate adaptiv an die Parameter in \mathbf{w} anpassen.

Bei dem Training eines Modells werden die Parameter mit zufälligen Werten initialisiert und die Initialisierung hat eine Auswirkung auf die Performanz des Modells [Bis06, S. 240].

Wird der Loss auf den gesamten N Trainingsdaten berechnet, spricht man von einem Batch-Gradient-Descent. Der Gegensatz zum Batch-Gradient-Descent ist der Stochastic-Gradient-Descent (SGD), bei dem der Loss iterativ über unterschiedliche Teilmengen der Trainingsdaten berechnet wird [RN21, S. 697]. Während eines Trainingsvorgangs nutzt man den Begriff „Epoche“, um anzugeben, wie oft über alle Trainingsdaten bereits iteriert wurde [LWG]. Dies hat zur Folge, dass der berechnete Gradient, bezogen auf den Trainingsdatensatz, nicht exakt ist und approximiert wird [LBBH98]. Die Optimierung mittels SGD wird bevorzugt, da das Verfahren schneller ist und oftmals zu besseren Ergebnissen führt [LBOM12]. Besonders bei großen Datensätzen mit vielen Trainingsdaten ist die Aufstellung des Losses über den gesamten Datensatz aufgrund des hohen Rechenaufwandes unpraktikabel. Außerdem kann eine Verlustfunktion in neuronalen Netzen unterschiedliche lokale Minima haben und die Approximation des Gradienten kann dabei helfen, weitere, potenziell bessere Minima zu finden [LBOM12].

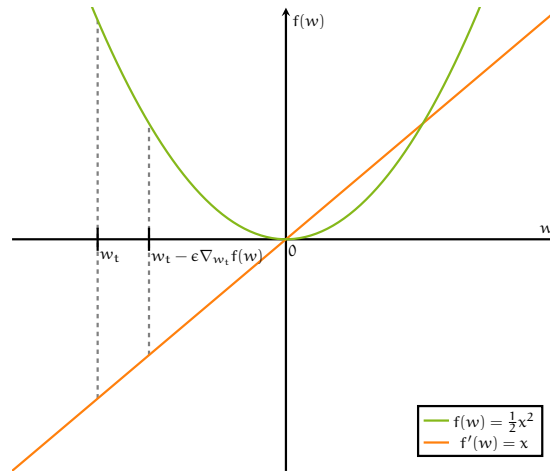


Abbildung 2.3.1: Veranschaulichung der Gradient Descent Methode anhand einer Funktion mit einem Parameter. Der Funktionswert der in grün dargestellten Funktion f an der Stelle w_t wird mit Hilfe der in orange dargestellten Ableitung $f'(w_t)$ verringert. Grafik in Anlehnung an [GBC16].

2.3.3 Backpropagation

Die Backpropagation ist ein algorithmisches Verfahren für die Berechnung eines Gradienten von einer Fehlerfunktion L in einem neuronalen Netz. Der Kern des Verfahrens ist die rekursive Anwendung der Kettenregel [LWG]. In diesem Kontext wurde sie erstmals 1986 von Rumelhart et al. [RHW86] vorgestellt und besteht aus zwei Schritten: Forward-Pass und Backward-Pass. Im Folgenden wird das Verfahren nach [Biso6] beschrieben.

Zunächst wird mittels der Trainingsdaten die Verlustfunktion L aufgestellt und im Forward-Pass der Wert von L berechnet. Im Folgenden wird der Fall betrachtet, dass L nur für eine einzelne Eingabe (x, y) berechnet wird. Während des Forward-Passes werden die Aktivierungen z_k aller k Neuronen im Netz zwischengespeichert.

Im Backward-Pass wird L dann nach den Gewichten w_{ji} durch die Anwendung der Kettenregel differenziert.

Sei w_{ji} ein Gewicht eines Neurons j , das in $a_j = \sum_i z_i w_{ji}$ mit der Aktivierung $z_i = h(a_i)$ weiterer i Neuronen multipliziert wird, wobei h die Aktivierungsfunktion eines Neurons i ist, dann kann L nach w_{ji} wie folgt differenziert werden:

$$\frac{\partial L}{\partial w_{ji}} = \frac{\partial L}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}. \quad (2.3.3)$$

Sei $\delta_j = \frac{\partial L}{\partial a_j}$ der Fehler des Neurons j und $\frac{\partial a_j}{\partial w_{ji}} = z_i$, dann ergibt sich für Gleichung 2.3.3:

$$\frac{\partial E}{\partial w_{ji}} = \delta_j z_i . \quad (2.3.4)$$

Aus Gleichung 2.3.4 folgt, dass sich der Gradient für alle Gewichte im Netz berechnen lässt, wenn es möglich ist, den Fehler δ_j für jedes Neuron j zu berechnen und δ_j mit der Aktivierung z_i zu multiplizieren [Biso6, S. 243]. An dieser Stelle wird die zwischengespeicherte Aktivierung aus dem Forward-Pass z_i abgerufen, sodass keine erneute Berechnung von z_i stattfinden muss. Zur Berechnung des Fehlers δ_j wird zwischen zwei Fällen unterschieden:

Falls ein Neuron j im Output-Layer liegt, muss für den Fehler L direkt nach a_j differenziert werden, da die Neuronen im Output-Layer keine Aktivierungsfunktion haben.

Falls ein Neuron j im Hidden-Layer liegt, muss eine Summe über die l Neuronen gebildet werden, weil ein Neuron j im Hidden-Layer mit allen l Neuronen im sukzessiven Layer verbunden ist und die Aktivierung z_j für Berechnungen im sukzessiven Layer genutzt wird. Für den Fehler des Neurons j im Hidden-Layer gilt:

$$\delta_j = \frac{\partial L}{\partial a_j} = \sum_l \frac{\partial L}{\partial a_l} \frac{\partial a_l}{\partial a_j} . \quad (2.3.5)$$

Für zwei direkt miteinander verbundene Neuronen j, l , wobei die Ausgabe von j nach l weitergeleitet wird, gilt $\frac{\partial a_l}{\partial a_j} = \frac{\partial a_l}{\partial z_j} \frac{\partial z_j}{\partial a_j} = w_{lj} \cdot h'(a_j)$. Setzt man dies in Gleichung 2.3.5 ein, so erhält man für den Fehler eines Neurons j im Hidden-Layer:

$$\delta_j = h'(a_j) \cdot \sum_l w_{lj} \delta_l . \quad (2.3.6)$$

Das Vorhandensein von δ_l aus einem sukzessiven Layer in δ_j bedeutet, dass Gleichung 2.3.6, unter Umständen zur Bestimmung des Fehlers erneut angewendet werden muss [Biso6, S. 243 ff.]. Dies bezeichnet die zuvor erwähnte rekursive Anwendung der Kettenregel.

Da sich die berechneten Fehler, ähnlich wie bei dem Forward-Pass, ebenfalls zwischenspeichern lassen, wird beim Backward-Pass bei der Differenzierung nach den Gewichten topologisch gesehen hinten angefangen, sodass die Zwischenergebnisse aus späteren Layern in früheren Layern wiederverwendet werden können.

Zu beachten ist, dass die zuvor vorgestellte Form der Backpropagation auch die Differenzierung nach den Bias einschließt, indem der Bias eines Neurons als spezielles

Gewicht betrachtet wird, das in der gewichteten Summe $a_j = \sum_i z_i w_{ji}$ mit einer Aktivierung $z_i = 1$ multipliziert wird [Biso6, S. 243].

Um $\frac{\partial L}{\partial w_{ji}}$ für den Fall, dass $L = \sum_{k=1}^N L_k$ über N Trainingsdaten aufgestellt wurde zu berechnen, muss für den gesamten Loss das zuvor beschriebene Vorgehen für jedes L_k ausgeführt werden und folgende Summe berechnet werden [Biso6, S. 245]:

$$\frac{\partial L}{\partial w_{ji}} = \sum_{k=1}^N \frac{\partial L_k}{\partial w_{ji}}. \quad (2.3.7)$$

2.3.4 Overfitting

Ein Problem, das bei der Optimierung von neuronalen Netzen auftreten kann, ist ein sogenanntes Overfitting, bei dem das Modell eine schlechte Generalisierung auf unbekannten Daten erreicht, obwohl es gute Vorhersagen auf den Trainingsdaten trifft [Mit97, S. 123]. Die Ursache für das Problem ist, dass sich das Modell mit den Parametern zu stark an die Charakteristiken der Trainingsdaten anpasst, anstelle eines Lernens der zugrundeliegenden Beziehungen, welche für die Klassifikationsaufgabe vonnöten sind [LWG]. Das Problem tritt verstärkt auf, wenn der Trainingsdatensatz über wenige Daten verfügt (siehe [SHK⁺14, Mit97, S. 111 ff.]) und das Modell eine hohe Kapazität aufweist, um Charakteristiken auswendig zu lernen [GBC16, S. 109 ff.]. Somit kann eine bessere Generalisierung mit Hilfe eines größeren Datensatzes erreicht werden. Regularisierung bezeichnet die Verminderung von Overfitting und es existieren verschiedene Techniken, um dies umzusetzen.

Eine Form, um eine Regularisierung zu realisieren, ist die Addition eines zusätzlichen Terms zum Loss L und wird als Weight-Decay bezeichnet [Biso6, S.256 ff.]:

$$\tilde{L} = L + \lambda \frac{1}{2} \mathbf{w}^T \mathbf{w}. \quad (2.3.8)$$

Während der Optimierung gibt es Gewichte, deren Wert zunimmt, um den Loss weiterhin zu verringern [Mit97, S. 111]. Die Addition des Terms hat zur Folge, dass große Gewichte in \mathbf{w} stärker zum Loss beitragen, als kleinere. Der Hyperparameter λ kontrolliert dabei die Auswirkungen des Regularisierungsterms auf den Loss.

Eine weit verbreitete Technik zur Regularisierung während des Trainings ist Dropout [SHK⁺14]. Bei dieser Technik wird ein Neuron in einem Netz während des Trainings zufällig mit einer Wahrscheinlichkeit, beispielsweise $p = 0,5$, deaktiviert und gibt als Aktivierung den Wert 0 aus [GBC16, S. 255].

Eine weitere Form von Regularisierung sind Datenaugmentierungen, bei denen bereits bestehende Trainingsdaten in jeder Epoche variiert werden, sodass sich das Modell während der Optimierung an jeweils unterschiedliche Merkmale anpassen muss. In dieser Arbeit werden unterschiedliche Datenaugmentationen in Verbindung mit der Datensynthese in Kapitel 5 diskutiert.

Mathematische Ausdrücke sind für viele Menschen ein fester Bestandteil im schulischen, akademischen oder beruflichen Alltag. Die maschinelle Erkennung von handschriftlichen mathematischen Ausdrücken (HMA) ist jedoch aufgrund verschiedener Herausforderungen schwierig. Mathematische Ausdrücke umfassen ein umfangreiches Vokabular bestehend aus Zahlen, griechischen und lateinischen Buchstaben [CY00, LT20] mit ähnlich aussehenden Buchstaben, wie beispielsweise: χ vs. x , ρ vs. p oder β vs. B . Weiterhin weist die Sprache der mathematischen Ausdrücke eine zweidimensionale Struktur auf [MZGV16, CY00]. Die Symbole in einem Ausdruck können unterschiedliche Anordnungsbeziehungen einnehmen, welche zusätzlich unterschiedliche Skalierungen aufweisen können [CY00]. In einem Ausdruck wie 3.0.1 befindet sich beispielsweise das Symbol k dreimal, allerdings in zwei unterschiedlichen Skalierungen und nimmt dabei jedes Mal eine unterschiedliche Anordnungsbeziehung ein: unterhalb eines weiteren Symbols, neben einem weiteren Symbol und als Superskript. Außerdem können unterschiedliche Schreibstile zwischen Personen die Erkennung zusätzlich erschweren [LT20].

$$\sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1} \quad (3.0.1)$$

In den folgenden Kapiteln wird zunächst die Entwicklung der Systeme zur HMA-Erkennung, insbesondere im Bereich Deep-Learning, besprochen. Im Zuge dessen wird die CROHME [MVK⁺11] vorgestellt, ein Wettbewerb, dessen Datensatz für die in dieser Arbeit behandelte Datensynthese eine zentrale Rolle einnimmt. Danach werden verschiedene Ansätze für die Datensynthese sowohl für unterschiedliche Forschungsbereiche als auch für die HMA-Erkennung vorgestellt.

3.1 HANDSCHRIFTLICHE MATHEMATIKERKENNUNG

Die Erkennung von HMA reicht zurück bis in die 1960er Jahre [And67]. Lösungsansätze zur Erkennung von HMA, insbesondere bis zum Anfang des 21. Jahrhunderts, bestehen hauptsächlich aus zwei Phasen [CY00]: der Symbolerkennung und der Strukturanalyse. Beide Phasen können sowohl sequentiell als auch global gelöst werden

[CY00, ZDD18]. Die Symbolerkennung umfasst die Segmentierung und Identifizierung der im Ausdruck befindlichen Symbole und die Strukturanalyse die Erzeugung einer hierarchischen Struktur, beispielsweise in Form eines (Parse-)Baumes [CY00].

Weiterhin lassen sich bei der Erkennung von HMA die Lösungsansätze meistens¹ in zwei verschiedene Kategorien aufteilen: Online-Erkennung und Offline-Erkennung [CY00]. Der Unterschied zwischen den Kategorien liegt in der zugrunde liegenden Repräsentation der Eingabedaten, auf denen die Erkennen arbeiten. Online-Daten bestehen aus Sequenzen von Koordinaten [CY00], welche die Strichfolgen im HMA repräsentieren. Durch die Sequenzen sind zusätzlich temporale Informationen über den HMA gegeben. Sie können durch verschiedene Eingabegeräte wie Touchscreenegeräte oder Grafiktablets erzeugt werden [MVZG16, MVZG14].

Bei Offline-Daten liegen die Schriftzüge ausschließlich als Bilder vor [PS00]. Diese können beispielsweise mit einem Foto oder Scan des HMA erzeugt werden. Anders als bei Online-Daten bieten Offline-Daten keine zusätzlichen Informationen, wie die Reihenfolge der Strichfolgen im Ausdruck. Eine weitere Möglichkeit Offline-Daten zu erzeugen, ist eine Umwandlung der Online-Daten in Offline-Daten. Dies wird als Rendering bezeichnet und in Abschnitt 5.3 näher betrachtet.

Um die Vergleichbarkeit verschiedener Systeme zu gewährleisten und den Stand der Technik zu beschreiben wurde im Jahr 2011 von Mouchère et al. ein Wettbewerb für die Online-Mathematikererkennung unter dem Namen „Competition on Recognition of Online Handwritten Mathematical Expressions“ (im Folgenden CROHME) [MVK⁺11] gegründet und fand bisher sechsmal in den Jahren 2011–2014 [MVK⁺11, MVK⁺12, MVZ⁺13, MVZG14], 2016 [MVZG16] und 2019 [MZM⁺19] statt. Die CROHME stellte in den vergangenen Jahren verschiedene Versionen ihres Online-Datensatzes und Evaluierungs-Tools für die Mathematikererkennung zur Verfügung. Neben der Hauptaufgabe Online-HMA als Ganzes zu erkennen, gab es in den vergangenen Competitions zusätzlich Aufgaben wie isolierte Symbole zu erkennen (2014, 2016, 2019), Matrizen zu erkennen (2014, 2016) oder zuletzt Ausdrücke in Dokumenten zu finden (2019). Forschungsgruppen können für die unterschiedlichen Aufgaben ihre Systeme einreichen, deren Funktionsweise und Performanz anschließend vorgestellt werden. In dieser Arbeit wird ausschließlich die Hauptaufgabe zur Erkennung von HMA betrachtet.

Zu beobachten ist, dass in den von Chan und Yeung in [CY00] im Jahr 2000 zusammengefassten Systemen und die Systeme in den Anfangsjahren der CROHME insbesondere bei der Strukturanalyse auf Regeln, wie beispielsweise Grammatiken,

¹ Eine klare Trennung ist nicht immer der Fall, da es Erkennen gibt, die sowohl mit Online- als auch Offline-Repräsentationen arbeiten (siehe beispielsweise Systeme in [MVZG14] und [MVZG16]).

basieren. Der manuelle Entwurf solcher Regeln gestaltet sich als schwierig, da er auf Vorkenntnissen [ZDZ⁺17] in der Domäne basiert.

Im Jahr 2019 wurden für die Hauptaufgabe der CROHME Online-Ausdrücke zu erkennen auch Systeme, welche ausschließlich auf offline gerenderten Versionen der Ausdrücke arbeiten gesondert berücksichtigt und die teilnehmenden Systeme konnten ihre Ausgaben als LaTeX-String oder Symbol-Layout-Tree [MZM⁺19] formulieren. Eine Besonderheit in dem Jahr war, dass erstmals vollständig auf Deep-Learning-basierte Encoder-Decoder Systeme teilnahmen und sowohl für die Online- als auch für die Offline-Erkennungsaufgabe jeweils den ersten Platz belegen konnten. Das Gewinnersystem der Hauptaufgabe setzte für die Offline-Erkennung das in [ZDZ⁺17] unter dem Namen WAP² vorgestellte Modell ein.

Das WAP-Modell [ZDZ⁺17] besteht aus einem Encoder und einem Decoder mit einem Attention-Mechanismus. Der Encoder ist ein Convolutional Neural Network (CNN) auf Basis der VGG-Architektur [SZ15] und bildet die Eingabe zunächst auf Feature-Maps ab, welche eine Sequenz von Feature-Vektoren formen [ZDZ⁺17]. Der auf Gated-Recurrent-Units [CMG⁺14] basierte Decoder nimmt die Feature-Vektoren entgegen und gibt eine Sequenz bestehend aus LaTeX-Token aus. Neben dem Attention-Mechanismus, gibt einen sogenannten Coverage-Mechanismus [TLL⁺16], der bei der Dekodierung eine redundante Betrachtung der Regionen in der Eingabe verhindern soll [ZDZ⁺17]. Nach Zhang et al. gibt es bei einer solchen Vorgehensweise folgende Hauptunterschiede zu herkömmlichen Systemen [ZDD18]: Das Modell ist daten-gestützt und die Segmentierung findet automatisch statt. Ersteres vermeidet den manuellen Entwurf von Regeln vollständig, da diese aus den Daten gewonnen werden. Abbildung 3.1.1 zeigt die Arbeitsweise des Decoders mit Attention-Mechanismus.

Um die Performanz zu steigern, wurde von Zhang et al. mit [ZDD18] eine verbesserte Version auf Basis des WAP-Modells unter dem Namen DenseWAP veröffentlicht. Für den Encoder wurde anstelle einer VGG-Architektur eine DenseNet-Architektur [HLMW17] eingesetzt. Eine weitere Verbesserung konnte erzielt werden, indem die Abbildung der Eingabe in den Encoder gleichzeitig auf zwei verschiedene Auflösungen stattgefunden hat [ZDD18].

Zhao et al. präsentierten in [ZGY⁺21] das sogenannte BTTR³-Modell. Aus Gründen der Vergleichbarkeit zu DenseWAP setzten die Autoren ebenfalls ein DenseNet (ohne Multi-Scale-Zweig) als Encoder ein [ZGY⁺21]. Als Decoder wird anstelle eines RNN die Transformer-Architektur [VSP⁺17] verwendet. Zur Verbesserung der Performanz findet das Training des Modells zusätzlich bidirektional statt.

² „Watch, Attend and Parse“

³ „Bidirectionally Trained Transformer“

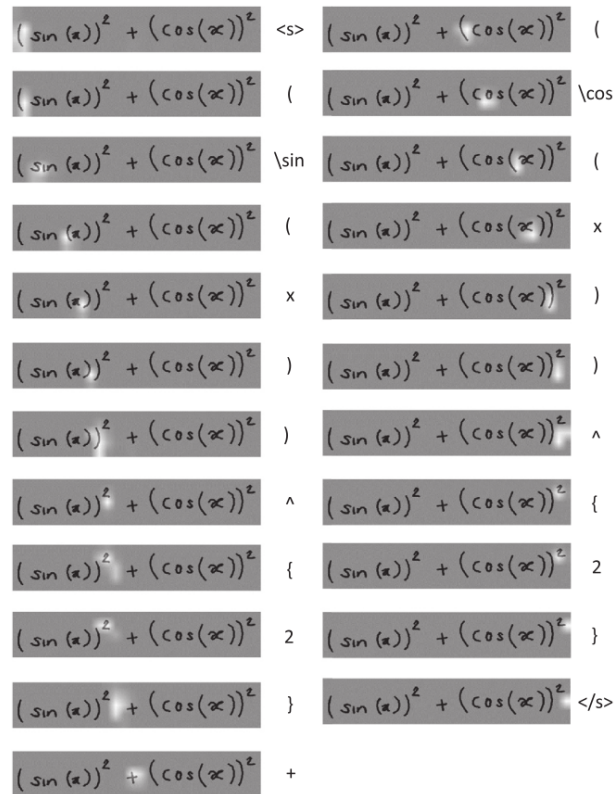


Abbildung 3.1.1: Visualisierte Arbeitsweise des Attention-Mechanismus des WAP-Modells. Helle Stellen sind vom Attention-Mechanismus hervorgehobene Regionen, welche für die Erzeugung der Ausgabe (jeweils rechts daneben) in einem Dekodierungsschritt relevant sind. Die Abbildung liest sich von oben nach unten und wurde entnommen aus [ZDZ⁺17].

Weiterhin konnte das BTTR-Modell in einer weiteren Arbeit [ZG22] durch die Erweiterung des Transformer-Decoders mittels eines Coverage-Mechanismus verbessert werden.

Die Gemeinsamkeit der vorgestellten Deep-Learning-basierten Modelle ist, dass für die Funktionsweise eine große Anzahl an Trainingsdaten benötigt werden.

3.2 DATENSYNTHESE

Der Arbeitsaufwand zur Herstellung eines umfangreichen Datensatzes durch Menschenhand erweist sich als äußerst zeit- und kostenintensiv [KJ16a, KUNP19]. Diese

Problematik beschränkt sich nicht ausschließlich auf die Erkennung im Bereich der Dokumentenanalyse, sondern erstreckt sich über viele Forschungsbereiche.

Für die Herstellung von Trainingsdaten für die HMA-Erkennung gibt es verschiedene Vorgehensweisen. Eine Möglichkeit ist die Vorgabe von Labels, zu denen Personen gebeten werden, passende Daten zu erzeugen (siehe CROHME [MVZG16]). Eine weitere Möglichkeit ist, dass bereits Daten gegeben sind und Personen den Daten Labels zuweisen oder durch einen automatisierten Vorgang bereits zugewiesene Labels verifizieren oder korrigieren (siehe HME100k [YLD⁺22] Supplementary Material). Obwohl Ansätze wie Crowdsourcing existieren [KUNP19], um den Erstellungsprozess zu unterstützen, so ist bei einer solchen Vorgehensweise dennoch menschliche Arbeit involviert, welche mit der Zunahme von Trainingsdaten ebenfalls zunimmt.

Ein Lösungsansatz für die zuvor genannte Problematik ist eine Nutzung von synthetischen Trainingsdaten. Die Datensynthese beschäftigt sich mit der Generierung künstlicher Trainingsdaten. Ziel der Datensynthese ist, dass die menschliche Arbeit bei der Erstellung der Datensätze reduziert wird und bietet somit eine kostengünstige [KUNP19] Möglichkeit, um zusätzliche Trainingsdaten zu erzeugen. In einer Domäne, wie die Erkennung von HMA, in der bisher nur wenige Datensätze existieren ist dies besonders attraktiv.

Die Datensynthese grenzt sich von der Datenaugmentierung in dem Aspekt ab, dass bei der Datensynthese neue Trainingsdaten erzeugt und bei der Datenaugmentierung hingegen bestehende echte Daten durch unterschiedliche Techniken verändert werden [Nik19].

Synthetische Daten werden in verschiedenen Forschungsbereichen mit unterschiedlichen Vorgehensweisen eingesetzt. Das semantische Segmentierungsproblem [LSD15] ist ein Beispiel, bei dem sich die Erstellung von Trainingsdaten durch Menschenhand besonders aufwändig ist, da pixelgenaue Annotation notwendig sind [RSM⁺16]. In [RSM⁺16] wurde beispielsweise für die semantische Segmentierung im Bereich des autonomen Fahrens eine Stadt zunächst 3D-modelliert und anschließend mit Hilfe des 3D-Modells Bilder und Videos erzeugt. Weitere Forschungsbereiche in denen synthetische Daten eingesetzt werden können sind die Scene Text Recognition [JSVZ14] oder die Bilderkennung im medizinischen Bereich [FDK⁺18].

Im Bereich der Dokumentenanalyse gibt es verschiedene Vorgehensweisen. Eine Vorgehensweise ist die Nutzung von Fonts zur Darstellung von Text. Krishnan et al. verfolgten in [KJ16a] einen solchen Ansatz für das Handwritten-Word-Spotting und erstellten einen Datensatz mit Bildern, die einzelne Worte enthalten. Mit einem Vokabular aus 90.000 Worten und einer Auswahl aus 750 Fonts wurden somit 9 Millionen Trainingsbilder erzeugt. Da das Aussehen von einem natürlichen handschriftlichen Text von Faktoren wie der Schreibgeschwindigkeit, dem Schreibwinkel, dem Schreib-

untergrund und motorischen Fähigkeiten beeinflusst wird, nutzten die Autoren bei der Nachbearbeitung zur Nachahmung eines handschriftlichen Schreibstils zufällig gewählte affine Transformationen [KJ16a].

Für die Erkennung von druckschriftlichen mathematischen Ausdrücken präsentierten Deng et al. [DKLR17], ähnlich zu den in Abschnitt 3.1 präsentierten Modellen, ebenfalls ein Encoder-Decoder Modell mit einem Attention-Mechanismus. Im Zuge dessen wurde neben dem druckschriftlichen Corpus auch ein handschriftlicher Corpus erstellt, indem alle Symbole in einem Ausdruck durch zufällig gewählte handschriftliche Symbole ersetzt wurden.

Eine weitere Vorgehensweise im Bereich der Dokumentenanalyse ist die Generierung von Trainingsdaten mittels neuronaler Netze. Zur Generierung von handschriftlichen Online-Daten nutzte Graves [Gra13] ein RNN auf Basis der LSTM-Architektur, das auf den Online-Daten des IAM Datensatzes [LB05] trainiert wurde.

Springstein et al. [SME21] nutzten für die Generierung von Offline-HMA mittels neuronaler Netze eine Generative Adversarial Network (GAN) Architektur [GPM⁺20], welche sich zur Generierung von Bildern eignet. Das GAN bekommt als Eingabe einen in Druckschrift gerenderten mathematischen Ausdruck und übersetzt ihn in einen HMA.

Eine Vorgehensweise, die sich von den beiden zuvor genannten Kategorien unterscheidet, ist die Arbeit von Khuong et al. [KUNP19]. Die Autoren präsentierten einen Synthesizer, welcher mittels eines mathematischen Ausdrucks in LaTeX oder MathML einen Online-HMA erzeugt. Der Synthesizer konstruiert aus dem mathematischen Ausdruck als Eingabe zunächst einen Symbol-Relation-Tree (SRT), der die Anordnungsbeziehungen der in der Eingabe befindlichen Symbole repräsentiert. Danach werden mittels des SRT die Größen und Positionen der Symbole bestimmt. Anschließend wird der Online-HMA erzeugt, indem Online-Symbole aus einer Datenbank eingesetzt werden.

MODELL

Zur Untersuchung der Auswirkungen eines Trainings auf synthetischen Offline-Daten bei der Erkennung von handschriftlichen mathematischen Ausdrücken (HMA) wird in dieser Arbeit das BTTR-Modell [ZGY⁺21] und dessen in [Zha21] veröffentlichte Implementierung verwendet. Das BTTR-Modell nimmt ein Bild eines HMA entgegen und erzeugt einen zum Bild passenden LaTeX-String.

LaTeX ist ein System, welches häufig zur Formulierung von wissenschaftlichen Dokumenten verwendet wird [Lam94]. Im Folgenden wird LaTeX vorübergehend als eine Sprache zur Formulierung von mathematischen Ausdrücken aufgefasst und der Teil betrachtet, welcher ausschließlich zur Formulierung von mathematischen Ausdrücken mit den Symbolklassen des CROHME-Datensatzes [MZM⁺19] relevant ist.

Das BTTR-Modell besteht aus einem Encoder und einem Decoder, welche im Folgenden vorgestellt werden.

4.1 ENCODER

Der Encoder des BTTR-Modell ist ein CNN und hat die Aufgabe, Merkmale des Eingabebildes zu extrahieren. Das CNN folgt der von Huang et al. in [HLMW17] vorgestellten DenseNet-Architektur. Aus Gründen der Vergleichbarkeit zum DenseWAP-Modell [ZDD18] wird der gleiche Encoder eingesetzt mit dem Unterschied, dass die Ausgabe des Encoders in ausschließlich einer Auflösung vorliegt [ZGY⁺21].

Im Folgenden wird zunächst die allgemeine DenseNet-Architektur diskutiert und im Anschluss der Aufbau des DenseNets im BTTR Modell vorgestellt.

4.1.1 DenseNet

Die Idee der DenseNet-Architektur ist, dass ein Layer ℓ , anders als in einem klassischen CNN, mit allen seinen $\ell - 1$ Vorgänger-Layern verbunden ist.

Seien $x_0, \dots, x_{\ell-1}$ die von den Layern $0, \dots, (\ell - 1)$ erzeugten Feature-Maps und \cdot eine Konkatenation in der Tiefe, dann ist die Ausgabe des ℓ -ten Layers x_ℓ definiert als (Definition nach [ZDD18])

$$x_\ell = H_\ell([x_0; x_1; \dots; x_{\ell-1}]), \quad (4.1.1)$$

wobei $H_\ell(\cdot)$ die Anwendung einer 3×3 Convolution bezeichnet [ZDD18]. Im Folgenden wird, sofern nicht anders erwähnt, die Convolution mit einer Stride $S = 1$ und mit Padding ausgeführt. Somit berechnet der ℓ -te Layer seine Ausgabe auf Basis der Ausgaben aller vorherigen Layer. Durch den Zugriff auf Feature-Maps aus früheren Layern wird eine Wiederverwendung von Features ermöglicht. Die Anzahl an erzeugten Feature-Maps innerhalb eines Convolutional-Layers im DenseNet wird durch den Hyperparameter Growth-Rate k bestimmt [HLMW17].

Eine Schwierigkeit ist, dass zwischen den Layern $0, \dots, \ell - 1$ kein Pooling angewendet werden kann, da andernfalls die Konkatenation aufgrund abweichender Größen der Feature-Maps infolge eines Poolings nicht möglich wäre. Um ein Pooling dennoch anwenden zu können, wird aus diesem Grund das DenseNet in unterschiedliche sogenannte Dense-Blocks eingeteilt, in denen die Operation von Definition 4.1.1 ausgeführt wird. Abbildung 4.1.1 veranschaulicht den Aufbau eines Dense-Blocks. Zwischen den Dense-Blocks findet das Pooling in sogenannten Transition-Layern statt. Der Hyperparameter Depth D regelt die Anzahl an Convolutions in den Dense-Blocks [HLMW17].

Für eine Verringerung des Berechnungsaufwands aufgrund einer steigenden Anzahl der Parameter in den höheren Layern können vor den 3×3 Convolutions sogenannte Bottleneck-Layer eingeschoben werden. Ein Bottleneck-Layer wendet eine 1×1 Convolution an, welche $4k$ Feature-Maps erzeugt [HLMW17].

Nachdem die Convolutions in einem Dense-Block ausgeführt wurden, werden alle m Feature-Maps des Blocks ausgegeben. Zur weiteren Reduktion der Größe wird im Transition-Layer vor der Anwendung des Poolings ebenfalls eine 1×1 Convolution angewendet, welche $\lfloor cm \rfloor$ Feature-Maps erzeugt, wobei $0 < c \leq 1$ gilt. Dies wird als Kompression bezeichnet und der Kompressionsfaktor c ist ein Hyperparameter, welcher für den Fall $c < 1$ den Grad der Kompression angibt [HLMW17]. Die Transition-Layer sind somit zusammengesetzte Layer aus Convolution und Pooling und haben neben der Aufgabe die Höhe und Breite des Ausgabevolumens mittels Pooling zu reduzieren, zusätzlich die Aufgabe die Tiefe zu reduzieren.

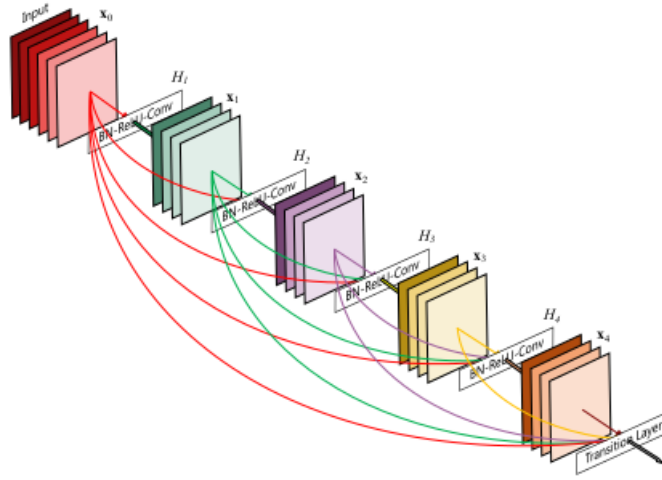


Abbildung 4.1.1: Aufbau eines Dense-Blocks mit Growth-Rate $k = 4$ und Depth $D = 5$. Zu beachten ist, dass die Reihenfolge der Operationen nach einer Convolution nicht mit der Reihenfolge im Dense-Encoder übereinstimmt. Abbildung entnommen aus [HLMW17].

4.1.2 Dense-Encoder

Bei einer Eingabe in den Encoder des BTTR-Modells wird zunächst eine 7×7 Convolution mit einer Stride 2 ausgeführt und 48 Feature-Maps erzeugt. Danach findet ein 2×2 Max-Pooling mit Stride 2 statt [ZDD18]. Daraufhin folgen drei Dense-Blocks mit einer Growth-Rate, $k = 24$. Jeder Dense-Block hat eine Depth $D = 16$ und verwendet Bottleneck-Layer. Somit besteht ein Dense-Block insgesamt aus 32 Layern: 16 Convolutional-Layern und zusätzlich 16 Bottleneck-Layern [ZGY⁺21, ZDD18]. Zwischen den Dense-Blocks befinden sich zwei Transition-Layer, jeweils mit einem Kompressionsfaktor $c = 0,5$ und 2×2 Average-Pooling mit Stride 2 [ZDD18].

Jede Convolution im gesamten Encoder wird gefolgt von einer Batch Normalization und ReLU-Aktivierungsfunktion [ZDD18].

Zusätzlich folgt nach dem letzten Dense-Block eine 1×1 Convolution, um die Ausgabe-Dimensionen des Encoders an den Hyperparameter d_{model} des Transformer-Decoders anzupassen [ZGY⁺21]. Anschließend werden die Feature-Maps im Ausgabevolumen des CNNs mit der Dimension $H \times W \times d_{\text{model}}$ in $L = H \times W$ Feature-Vektoren der Dimension d_{model} umgeformt und ausgegeben [Zha21].

4.2 DECODER

Der Decoder des BTTR-Modells ist ein Transformer und hat die Aufgabe, die vom Encoder extrahierten visuellen Merkmale und Wort-Sequenzen entgegenzunehmen und eine Sequenz von LaTeX-Tokens zu erzeugen. Die Transformer-Architektur wurde von Vaswani et al. in [VSP⁺17] ursprünglich als ein Sequence-Model für die Machine-Translation [HM15] präsentiert. Während für Modelle in diesem Problembereich typischerweise RNNs sowohl als Encoder als auch als Decoder eingesetzt werden, setzten die Autoren anstelle von RNNs die Transformer ein und konnten die damaligen State-Of-The-Art Modelle in der Performanz übertreffen [VSP⁺17].

Die Besonderheit der Transformer-Architektur ist, dass sie vollständig aus Attention-Mechanismen besteht und dabei auf Rekurrenzen verzichtet [VSP⁺17]. Ersteres erlaubt es der Architektur, im Vergleich zu einem RNN, das insbesondere bei langen Sequenzen auftretende Vanishing Gradient Problem zu vermeiden [ZGY⁺21]. Letzteres ermöglicht eine höhere Parallelisierbarkeit bei den Berechnungen im Modell [VSP⁺17, ZGY⁺21]. Seit der Veröffentlichung von Vaswani et al. im Jahr 2017 haben Transformer-Modelle eine weite Verbreitung in unterschiedlichen Problemstellungen [DCLT19, CMS⁺20, PVU⁺18].

Der Transformer-Decoder im BTTR-Modell [ZGY⁺21] orientiert sich an dem ursprünglichen Decoder von Vaswani et al. [VSP⁺17].

Im Folgenden wird zunächst der Aufbau der Decoder-Layer vorgestellt. Danach folgt die Vorstellung der Funktionsweise von den sogenannten Multi-Head-Attention-Layern. Anschließend wird die bidirektionale Sprachmodellierung, welche während des Trainings des BTTR-Modells eingesetzt wird, präsentiert. Schließlich werden die Inferenz eines LaTeX-Ausdruckes und die Techniken, welche bei der Eingabe eines Bildes eingesetzt werden, erläutert.

4.2.1 Transformer-Decoder

Ein Transformer-Decoder besteht aus einem oder mehreren zusammengesetzten Layern. Abbildung 4.2.1 zeigt den Aufbau des BTTR-Modells. Im rechten Teil der Abbildung befindet sich der Transformer-Decoder. Der Decoder bekommt als Eingabe die Feature-Vektoren des Encoders und zusätzlich eine Sequenz $y = \{y_1, \dots, y_T\}$ von Tokens¹ y_i und gibt ein neues Token aus. Der Decoder besteht aus mehreren in Grau dargestellten Decoder-Layern. Ein Decoder-Layer besteht aus drei Sub-Layern:

¹ Obwohl in Abbildung 4.2.1 zwei konkatenierte Sequenzen angedeutet sind, so werden diese getrennt in den Decoder eingegeben, da sie in der Batch-Dimension (siehe [ZGY⁺21]) konkateniert werden.

zwei Multi-Head-Attention-Layer (MHA) und einem Feed-Forward-Network. Vor jedem Sub-Layer existiert jeweils eine Vorwärtsverbindung zu einer sogenannten Layer-Normalization [BKH16], welche auf der Summe zwischen Ein- und Ausgabe des Sub-Layers berechnet wird. Um die Summen berechnen zu können, müssen die Dimensionen aller Eingaben in den Decoder-Layer und die Ausgaben der Sub-Layer vorab durch den Hyperparameter d_{model} festgelegt werden [VSP⁺17].

Jeder Decoder-Layer erhält zwei Eingaben: die Feature-Vektoren des Encoders und die Ausgabe des Vorgänger-Layers. Die Feature-Vektoren des Encoders werden vor der Eingabe positionskodiert und dann an einen MHA-Layer weitergeleitet. Falls der Decoder-Layer keinen Vorgänger-Layer hat, so ist die Eingabe die Sequenz y . Die Tokens werden zunächst durch ein sogenanntes Embedding in Vektoren der Dimension d_{model} abgebildet und ebenfalls positionskodiert [VSP⁺17]. Daraufhin werden die kodierten Vektoren in den Masked-MHA-Layer eingegeben, dessen normalisierte Ausgabe in den MHA-Layer weitergeleitet wird.

Schließlich wird jede Zeile in der Ausgabe des MHA-Layers durch ein zweischichtiges Fully-Connected Feed-Forward-Network² mit einer ReLU-Aktivierungsfunktion verarbeitet. Die Anzahl der Neuronen im Hidden-Layer wird durch den Hyperparameter d_{ff} bestimmt [VSP⁺17].

Die finale Ausgabe des Decoders wird mittels der Ausgabe des letzten Decoder-Layers, gefolgt von einem Linear-Layer und der Anwendung einer Softmax-Funktion, bestimmt.

Der Transformer-Decoder im BTTR-Modell nutzt drei Decoder-Layer und die Dimensionen $d_{\text{model}} = 256$ und $d_{\text{ff}} = 1024$ [ZGY⁺21].

4.2.2 Multi-Head-Attention

Der Kern der Transformer-Architektur sind die sogenannten Multi-Head-Attention-Layer, welche, wie der Name bereits andeutet, die Attention-Mechanismen im Modell realisieren. In einem MHA-Layer wird die sogenannte Scaled-Dot-Product Attention-Funktion berechnet [VSP⁺17]. Die Attention-Funktion ist wie folgt definiert:

Gegeben seien n Query-Vektoren $\mathbf{q} \in \mathbb{R}^{d_k}$ und jeweils m Paare von Key-Vektoren $\mathbf{k} \in \mathbb{R}^{d_k}$ und Value-Vektoren $\mathbf{v} \in \mathbb{R}^{d_v}$. Die Query-, Key- und Value-Vektoren werden jeweils als Zeilen in den Matrizen $\mathbf{Q} \in \mathbb{R}^{n \times d_k}$, $\mathbf{K} \in \mathbb{R}^{m \times d_k}$, $\mathbf{V} \in \mathbb{R}^{m \times d_v}$ zusammen-

² Zu beachten ist, dass aufgrund der zeilenweisen Anordnung der Ausgaben des MHA-Layers ein FFN wie in Definition 2.2.3 aufgrund der Reihenfolge zwischen \mathbf{x} und \mathbf{W}_i inkompatibel ist. Für eine kompatible Definition des FFN siehe [VSP⁺17].

gefasst und dienen als Eingabe in die folgende Attention-Funktion (Definition nach [VSP⁺17]):

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_v}}\right)\mathbf{V}. \quad (4.2.1)$$

In der Attention-Funktion werden für jede Query in \mathbf{Q} zunächst durch die Multiplikation mit \mathbf{K}^T die Ähnlichkeiten [ZGY⁺21] zwischen der Query und allen Keys in \mathbf{K} berechnet. Eine Query weist zu einem Key eine hohe Ähnlichkeit auf, wenn das Skalarprodukt zwischen den beiden Vektoren größer ist als das Skalarprodukt zwischen der Query und anderen Keys. Alle Werte in der resultierenden $n \times m$ Matrix werden daraufhin durch einen Faktor $\frac{1}{\sqrt{d_k}}$ skaliert um die Größenordnung der Werte zu reduzieren [VSP⁺17]. Danach werden die skalierten Werte durch eine zeilenweise Anwendung der Softmax-Funktion auf Werte zwischen 0 und 1 normalisiert. Die normalisierten Werte in jeder Zeile werden im nächsten Schritt als Gewichte verwendet, um mittels der Multiplikation mit \mathbf{V} eine gewichtete Summe aller Values zu bilden. Somit formt die Ausgabe der Attention-Funktion eine $n \times d_v$ Matrix, dessen Zeilen auf Values abgebildete Queries enthält.

Anstelle einer einfachen Anwendung der Attention-Funktion findet die Anwendung der Funktion parallel in h sogenannten Heads statt [VSP⁺17]. Dazu werden zunächst die Queries \mathbf{Q} , Keys \mathbf{K} und Values \mathbf{V} vor der Anwendung der Attention-Funktion durch gelernte Matrizen $\mathbf{W}_Q^i \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $\mathbf{W}_K^i \in \mathbb{R}^{d_{\text{model}} \times d_k}$ und $\mathbf{W}_V^i \in \mathbb{R}^{d_{\text{model}} \times d_v}$ projiziert³ und die Attention-Funktion für jeden Head $i \in 1, \dots, h$ berechnet [VSP⁺17]:

$$\text{head}_i = \text{Attention}(\mathbf{Q}\mathbf{W}_Q^i, \mathbf{K}\mathbf{W}_K^i, \mathbf{V}\mathbf{W}_V^i). \quad (4.2.2)$$

Danach werden die einzelnen Heads spaltenweise konkateniert und erneut durch eine Matrix $\mathbf{W}_O \in \mathbb{R}^{h d_v \times d_{\text{model}}}$ projiziert und vom Layer ausgegeben:

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = [\text{head}_1; \dots; \text{head}_h]\mathbf{W}_O. \quad (4.2.3)$$

Der Einsatz verschiedener Heads ermöglicht es dem Modell, die Attention-Funktion auf unterschiedlich gelernten Repräsentationen der Queries, Keys und Values anzuwenden [VSP⁺17]. In der Arbeit von Vaswani et al. [VSP⁺17] und der Implementierung des BTTR-Modells [Zha21] sind die Dimensionen der Queries, Keys und Values auf $d_v = d_k = d_{\text{model}}/h$ reduziert, wobei $h = 8$ gilt.

³ Bei Eintritt in den MHA-Layer haben die Query-, Key-, und Value-Vektoren eine Dimension von d_{model} , da alle Sub-Layer im Modell d_{model} -dimensionale Ausgaben produzieren [VSP⁺17].

In einem Decoder-Layer (siehe Abb. 4.2.1) existieren zwei verschiedenartige MHA-Layer: Self-Attention- und Encoder-Decoder-Attention-Layer [VSP⁺17]. Der Unterschied zwischen den beiden Layern liegt in der Herkunft der Queries, Keys und Values. Die Self-Attention-Layer befinden sich vor den Encoder-Decoder-Attention-Layern und deren Queries, Keys und Values haben dieselbe Herkunft [VSP⁺17]: sie sind entweder die Ausgabe des Vorgänger-Decoder-Layers oder werden initial durch die Sequenz y , welche in den Decoder eingegeben wird, hergeleitet. Aufgrund der Sequenz y korrespondieren somit die Queries mit den Positionen der Elemente in y . Mittels der Self-Attention-Layer können für eine Query andere Queries an unterschiedlichen Positionen betrachtet werden. Obwohl die Herkunft der Queries, Keys und Values dieselbe ist, müssen die Eingaben in die Attention-Funktion in der Definition 4.2.2 aufgrund der gelernten Projektionen nicht gleich sein. Um zu bewirken, dass im Self-Attention-Layer für eine Query an einer Position ausschließlich vorhergehende Positionen betrachtet werden, findet in der Attention-Funktion vor der Anwendung der Softmax-Funktion eine Maskierung der Eingabe statt [VSP⁺17].

Bei einem Encoder-Decoder-Attention-Layer unterscheidet sich die Herkunft der Queries von der Herkunft der Keys und Values. Die Keys und Values sind die Feature-Vektoren des Decoders [VSP⁺17] und die Queries sind die normalisierten Ausgaben des vorhergehenden MHA-Layers. Die Layer bilden die Schnittstelle zwischen dem Encoder und dem Decoder und somit eine Schnittstelle zwischen Eingabebild und Sequenz. Sie ermöglichen es dem Modell, die Queries in Anbetracht von lokalen Regionen des Eingabebildes und der bereits verarbeiteten Eingabesequenz y abzubilden.

4.2.3 Positionskodierung

Im BTTR-Modell werden die Feature-Vektoren und die Vektoren in der Eingabesequenz y positionskodiert [ZGY⁺21]. Obwohl der in Abschnitt 4.2.2 diskutierte Attention-Mechanismus unterschiedliche Positionen betrachtet, geschieht dies ohne die Berücksichtigung der Reihenfolge der Elemente in y . Daher wird die Positionskodierung eingesetzt, um dem Transformer-Modell Information über die Reihenfolge der Sequenz y und die räumliche Position der Feature-Vektoren des Decoders bereitzustellen [ZGY⁺21].

Im Bereich der mathematischen Ausdrücke ist die Berücksichtigung der Reihenfolge essentiell, da die Leserichtung mathematischer Ausdrücke von links nach rechts ist und erhebliche Auswirkungen auf die Semantik der Ausdrücke haben kann.

Für die Wort-Vektoren wird die Positionskodierung nach dem Embedding angewendet. Sei $(\mathbf{e}_1, \dots, \mathbf{e}_T)$ eine Sequenz von Wort-Vektoren und \mathbf{e}_{pos} ein Element der

Sequenz mit der Position pos . Dann wird für \mathbf{e}_{pos} ein Kodierungsvektor $\mathbf{p}_{\text{pos},d}$ wie folgt berechnet (Definition nach [ZGY⁺21]):

$$\begin{aligned}\mathbf{p}_{\text{pos},d}[2i] &= \sin(\text{pos}/10000^{2i/d}) , \\ \mathbf{p}_{\text{pos},d}[2i+1] &= \cos(\text{pos}/10000^{2i/d}) .\end{aligned}\tag{4.2.4}$$

Der Vektor $\mathbf{p}_{\text{pos},d}$ hat die gleiche Dimension d wie \mathbf{e}_{pos} . Die Komponenten i des Vektors \mathbf{e}_{pos} werden mittels sin- und cos-Funktionen unterschiedlicher Frequenzen berechnet [VSP⁺17]. Für eine Komponente mit geradem Index i wird die sin-Funktion verwendet und andernfalls die cos-Funktion. Vor der Eingabe in den Decoder-Layer wird die Summe aus \mathbf{e}_{pos} und $\mathbf{p}_{\text{pos},d}$ berechnet.

Zur Kodierung der zweidimensionalen Position (x, y) eines d_{model} -dimensionalen Feature-Vektors \mathbf{f}_i im Ausgabevolumen der Größe $H \times W \times d_{\text{model}}$ des Encoders wird zunächst die relative Position (\bar{x}, \bar{y}) von \mathbf{f}_i bestimmt:

$$(\bar{x}, \bar{y}) = \left(\frac{x}{H}, \frac{y}{W} \right) .\tag{4.2.5}$$

Für den Kodierungsvektor $\mathbf{p}_{\text{pos},d}^I$ werden danach jeweils Vektoren für die relativen Positionen \bar{x} und \bar{y} getrennt nach Definition 4.2.4 berechnet und zu einem Vektor $\mathbf{p}_{\text{pos},d}^I$ konkateniert (Definition nach [ZGY⁺21]):

$$\mathbf{p}_{\text{pos},d}^I = [\mathbf{p}_{\bar{x},d/2}; \mathbf{p}_{\bar{y},d/2}] .\tag{4.2.6}$$

Analog zu der Kodierung von Wort-Vektoren wird vor der Eingabe in die Decoder-Layer die Summe zwischen \mathbf{f}_i und $\mathbf{p}_{\text{pos},d_{\text{model}}}^I$ berechnet.

4.2.4 Sprachmodellierung und Training

Bei einer Vorhersage von Sequenzen durch ein Modell müssen bei dem Entwurf die k Klassen der Elemente in den Ausgabesequenzen festgelegt werden. Bei der Gestaltung der Klassen gibt es verschiedene Vorgehensweisen. Im Kontext der Textgenerierung ist es beispielsweise möglich, je Wort im Vokabular (Dictionary) eine Klasse festzulegen [Gra13]. Eine weitere Möglichkeit ist die Festlegung jeweils einer Klasse für jedes Zeichen [Gra13].

Bei dem BTTR-Modell hingegen werden die Klassen nach der Menge der LaTeX-Tokens im Dictionary des CROHME-Datensatzes [MZM⁺19] eingeteilt. Zu jedem Token existiert jeweils eine Klasse (insgesamt 110, siehe [Zha21]). Die Tokens gliedern sich dabei in einzelne Zeichen wie beispielsweise Ziffern 0–9, Buchstaben a–z und Strukturelemente $\{ , , ^ , _ \}$ etc. oder gliedern sich in Wörter wie $\backslash \text{frac}$, $\backslash \text{sigma}$ etc.

Um ein Token auf einen reellwertigen Vektor der Dimension d_{model} abzubilden, wird im Modell ein Embedding eingesetzt, dessen Parameter während des Trainings gelernt werden [Zha21].

Eine Besonderheit des BTTR-Modells ist, dass während des Trainings eine bidirektionale Sprachmodellierung eingesetzt wird, welche in den Experimenten der Autoren die Performanz des Modells verbessern konnte (siehe [ZGY⁺21]). Zu einem Bild x mit einer Sequenz $y = (y_1, \dots, y_T)$ im Trainingsdatensatz werden zwei Sequenzen erzeugt: eine Sequenz von links nach rechts \vec{y} (L2R) und eine Sequenz von rechts nach links \overleftarrow{y} (R2L), wobei [ZGY⁺21]

$$\begin{aligned}\vec{y} &= (\text{<SOS>}, y_1, \dots, y_T, \text{<EOS>}) , \\ \overleftarrow{y} &= (\text{<EOS>}, y_T, \dots, y_1, \text{<SOS>}) .\end{aligned}\tag{4.2.7}$$

Die Elemente <SOS> (Start Of Sequence) und <EOS> (End Of Sequence) in den Sequenzen \vec{y} und \overleftarrow{y} sind spezielle Tokens, welche den Anfang und das Ende der Sequenz markieren und ebenfalls jeweils eine Klasse bilden [ZGY⁺21].

In einem Batch bestehend aus Z Trainingsdaten $\{x^{(z)}, y^{(z)}\}_{z=1}^Z$ werden für jedes Datum $(x^{(z)}, y^{(z)})$ die L2R- und R2L-Sequenzen $\vec{y}^{(z)}$ und $\overleftarrow{y}^{(z)}$ erzeugt und die Paare $(x^{(z)}, \vec{y}^{(z)})$ und $(x^{(z)}, \overleftarrow{y}^{(z)})$ in das Modell eingegeben. Als Verlustfunktion wird ein Cross-Entropy-Loss für die Modellparameter \mathbf{w} wie folgt berechnet (Definition nach [ZGY⁺21]):

$$\bar{\mathcal{L}}_j^{(z)} = -\log \left(p(\vec{y}_j^{(z)} \mid \vec{y}_{<j}^{(z)}, x^{(z)}, \mathbf{w}) \right) ,\tag{4.2.8}$$

$$\overleftarrow{\mathcal{L}}_j^{(z)} = -\log \left(p(\overleftarrow{y}_j^{(z)} \mid \overleftarrow{y}_{<j}^{(z)}, x^{(z)}, \mathbf{w}) \right) ,\tag{4.2.9}$$

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2ZL} \sum_{z=1}^Z \sum_{j=1}^L \left(\bar{\mathcal{L}}_j^{(z)}(\mathbf{w}) + \overleftarrow{\mathcal{L}}_j^{(z)}(\mathbf{w}) \right) ,\tag{4.2.10}$$

wobei $p(\vec{y}_j^{(z)} \mid \vec{y}_{<j}^{(z)}, x^{(z)}, \mathbf{w})$ für ein z die des Modells mittels der Softmax-Funktion bestimmte Wahrscheinlichkeit für das Token an Position j , gegeben der Sequenz $\vec{y}_{<j}^{(z)}$, des Bildes $x^{(z)}$ und Modellparameter \mathbf{w} , beschreibt. $\vec{y}_{<j}^{(z)}$ ist dabei die Sequenz vor der Position j .

Da das Modell die Vorhersage eines Tokens an Position j mittels der Sequenz bereits vorhergesagter Tokens vor j trifft, wird es als autoregressiv bezeichnet [ZGY⁺21, VSP⁺17].

Während des Trainings können aufgrund der Maskierung im Self-Attention-Layer $p(\vec{y}_j^{(z)} \mid \vec{y}_{<j}^{(z)}, x^{(z)}, \mathbf{w})$ für alle j gleichzeitig berechnet werden [ZGY⁺21].

4.2.5 Inferenz

Ziel der Inferenz ist die Erzeugung einer Sequenz \hat{y} zu einem gegebenen Bild x . Dabei werden schrittweise LaTeX-Tokens vorhergesagt und eine Sequenz y erweitert, bis die Tokens, welche das Ende der Sequenz markieren vorhergesagt werden oder eine vorher definierte maximale Länge, definiert als 200 Tokens [Zha21], der Sequenz y erreicht wurde. Da für einen Dekodierungsschritt Wahrscheinlichkeiten für alle k Token-Klassen vorliegen, wird, um den Suchraum für die Sequenz y einzuschränken, der Beam-Search-Algorithmus angewendet [ZGY⁺21]. Aufgrund der bidirektionalen Sprachmodellierung ist das Modell fähig, L2R- oder R2L-Sequenzen vorherzusagen [ZGY⁺21]. Daher wird zur Verknüpfung der Vorhersagen beider Richtungen eine sogenannte Approximate-Joint-Search [LUFs16] durchgeführt.

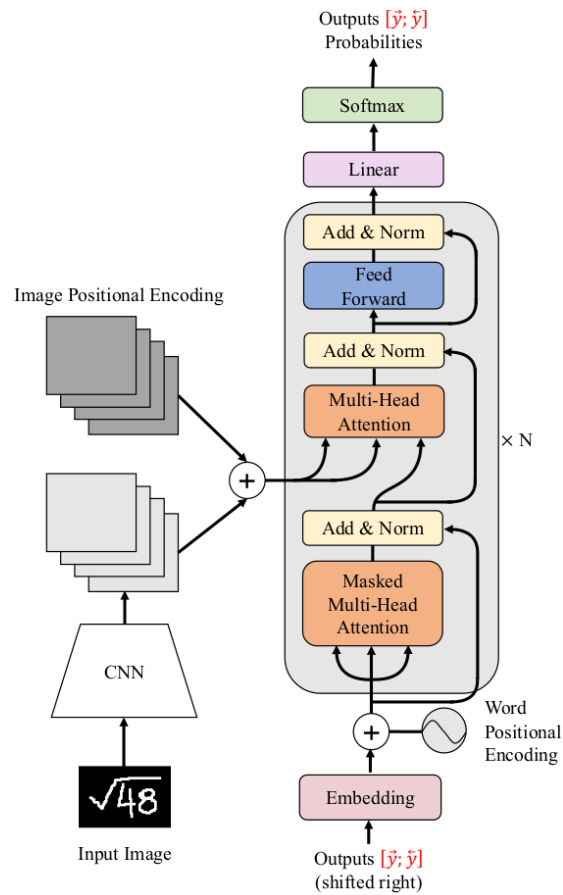


Abbildung 4.2.1: Architektur des BTTR-Modells. In der linken Bildhälfte befindet sich der CNN-Encoder und in der rechten Bildhälfte der Transformer-Decoder. Ein Decoder-Layer und dessen Komponenten sind in Grau dargestellt. Der Ausdruck $[\vec{y}; \hat{y}]$ bedeutet, dass innerhalb eines Batches die Sequenzen \vec{y} und \hat{y} getrennt eingegeben werden [ZGY⁺21]. Abbildung entnommen aus [ZGY⁺21].

Das Ziel in dieser Arbeit ist, synthetische Trainingsdaten für die Offline-Erkennung von handschriftlichen mathematischen Ausdrücken (HMA) herzustellen und anschließend die Auswirkungen auf die Performanz eines Modells, das auf den synthetischen Trainingsdaten trainiert wurde, experimentell zu untersuchen. Für die Datensynthese von mathematischen Ausdrücken wurde ein Synthesizer entworfen, welcher als Grundlage einen Font-basierten Ansatz nach [KJ16a] verwendet. Zusätzlich wird der Font-basierte Ansatz mittels handschriftlicher Symbole ähnlich wie in [DKLR17] verfeinert.

Abbildung 5.0.1 veranschaulicht die einzelnen Abläufe bei dem Prozess der Herstellung und den Experimenten. In diesem Kapitel wird die Datenextraktion und die Synthese vorgestellt. Anschließend werden in Kapitel 6 beide Methoden angewendet, um die nachfolgenden Schritte in der Synthese-Pipeline zu realisieren.

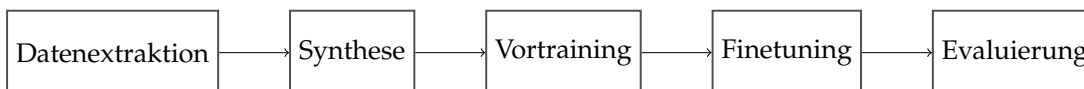


Abbildung 5.0.1: Die Schritte der Synthese-Pipeline.

5.1 DATENEXTRAKTION

Die Datenextraktion ist der erste Schritt in der Synthese-Pipeline. Sie beschäftigt sich damit, LaTeX-Strings aus unterschiedlichen Quellen zu entnehmen, in ein einheitliches Format zu bringen und in einer Datenbank abzuspeichern, sodass der Synthesizer im darauffolgenden Schritt die Ausdrücke abrufen und Trainingsbilder erzeugen kann. Die Quellen für LaTeX-Strings können beispielsweise LaTeX-Dokumente, Webseiten oder bereits vorhandene Datensätze sein. Das Entnehmen von LaTeX-Strings wird bei der Beschreibung der Datensätze in Kapitel 6 genauer vorgestellt. Im Folgenden wird angenommen, dass die LaTeX-Strings aus einer Quelle bereits in einer Liste \mathcal{L} vorliegen.

Zunächst werden alle LaTeX-Strings in \mathcal{L} , welche aus einer Folge von Zeichen bestehen, mittels eines sogenannten Lexers tokenisiert. Der Lexer nimmt dabei einen LaTeX-String entgegen und wandelt ihn in eine Sequenz von LaTeX-Tokens um. Der

Lexer wird durch die Spezifikation einer Grammatik¹ für LaTeX-Ausdrücke mit Hilfe des ANTLR-Parsergenerators [Par13] generiert. Die Tokenisierung der LaTeX-Strings ist ein unumgänglicher Vorverarbeitungsschritt. Erkennen wie in [ZDZ⁺17, DKLR17, ZGY⁺21] führen eine Klassenvorhersage auf Basis von Tokens durch und erfordern dementsprechend Datensätze, deren Annotationen für die Trainingsbilder Sequenzen von Tokens sind. Darüber hinaus erleichtert sie die Bereinigung der Ausdrücke.

Die Bereinigung ist notwendig, da in einem LaTeX-Ausdruck Tokens wie beispielsweise `\quad`, `\vbox`, `\left`, `\right`, etc. vorkommen können, welche aus Gründen der Darstellung verwendet werden und keine eigenständigen mathematischen Symbole repräsentieren. Solche Tokens werden in den Ausdrücken bereits während der Tokenisierung mittels des Lexers entfernt, weil die Erkennung primär auf die Erkennung der Symbole und Struktur in HMA abzielt und solche Tokens nicht zur Semantik des mathematischen Ausdruckes beitragen. Daraufhin findet eine Entfernung von doppelten Ausdrücken statt. An dieser Stelle erweist sich die Tokenisierung als hilfreich, da der Vergleich der Ausdrücke auf Basis von Tokens anstelle von Zeichen im Latex-String stattfinden kann.

Schließlich werden die tokenisierten und bereinigten Ausdrücke in einer Datenbank gespeichert.

5.2 SYNTHESIZER

Der Synthesizer ist ein Programm, das zur Herstellung von synthetischen Datensätzen verwendet wird. Das Programm erhält als Eingabe LaTeX-Ausdrücke und erzeugt als Ausgabe Bilder. Aufgrund der Kenntnis der Annotation lassen sich die Bilder als Trainingsdaten verwenden. Abbildung 5.2.1 veranschaulicht die schematische Funktionsweise des Synthesizers. In der regulären Arbeitsweise rendert der Synthesizer die Eingaben in Druckschrift. Zusätzlich verfügt der Synthesizer über die Fähigkeit, mittels Online-Trajektorien handschriftlich wirkende Ausdrücke zu rendern. Ersteres wird im Folgenden als Fontsynthese bezeichnet und in diesem Abschnitt vorgestellt. Letzteres hingegen wird im Folgenden als Handschriftsynthese bezeichnet und die Arbeitsweise in Abschnitt 5.3 behandelt.

Für die Verwendung des Synthesizers bei der Fontsynthese benötigt der Synthesizer Zugang zu einer Datenbank mit LaTeX-Ausdrücken \mathcal{L} und setzt voraus, dass die LaTeX-Ausdrücke in \mathcal{L} bereits tokenisiert und bereinigt wurden. Zusätzlich müssen vor der Datensynthese als Parameter die maximale Anzahl an abgerufenen Ausdrücken n , die Auflösung der Bilder in DPI d , die zu verwendenden Fonts und ein Vokabular

¹ Die Grammatik basiert auf der SymPy-Programmbibliothek [MSP⁺17].

\mathcal{V} festgelegt werden. Das Vokabular als Menge \mathcal{V} legt die zulässigen Tokens des synthetischen Datensatzes fest.

Die Synthese beginnt, indem zunächst Ausdrücke aus der Datenbank iterativ abgerufen werden. Sei \mathcal{F} eine initial leere Menge für die gefilterten Ausdrücke, A_i ein Ausdruck in der i -ten Iteration und $\text{Tokens}(x)$ die Funktion, die zu einem Ausdruck x die Menge der in x enthaltenen Tokens ausgibt. Falls der Ausdruck der i -ten Iteration A_i ausschließlich zulässige Tokens enthält, das heißt $\text{Tokens}(A_i) \subseteq \mathcal{V}$ gilt, dann wird A_i in eine Menge \mathcal{F} aufgenommen und andernfalls verworfen. Die Iteration stoppt, falls alle Ausdrücke in der Datenbank bereits abgerufen wurden oder $|\mathcal{F}| = n$ gilt.

Der Renderingprozess für druckschriftliche Trainingsbilder wird durch den LaTeX-Compiler realisiert. Dazu wird zunächst eine .tex-Datei als eine Vorlage angelegt. Danach werden die zu rendernden Ausdrücke iterativ einzeln in die Vorlage eingefügt und zunächst als .dvi-Datei kompiliert und anschließend in eine Bilddatei mit d DPI konvertiert. Falls sich ein Ausdruck erfolgreich kompilieren und konvertieren lässt, wird das Bild mit dem tokenisierten Ausdruck annotiert. Falls während der Kompilation Fehler auftreten, wird der Ausdruck ignoriert.

Die Ausdrücke werden standardmäßig in der regulären Font gerendert. Zusätzliche Fonts können durch die Parameter des Synthesizers festgelegt werden. Falls dies der Fall ist, wird jeder Ausdruck in jeweils einer Font gerendert. Die Fonts stammen alle aus der TeX-Live-Distribution [tex23] und werden mit Hilfe des „mathastext“-Paketes [Bur23] in die Vorlage eingebunden. In Abbildung 5.2.2 werden durch verschiedene Fonts gerenderte Trainingsbilder veranschaulicht.

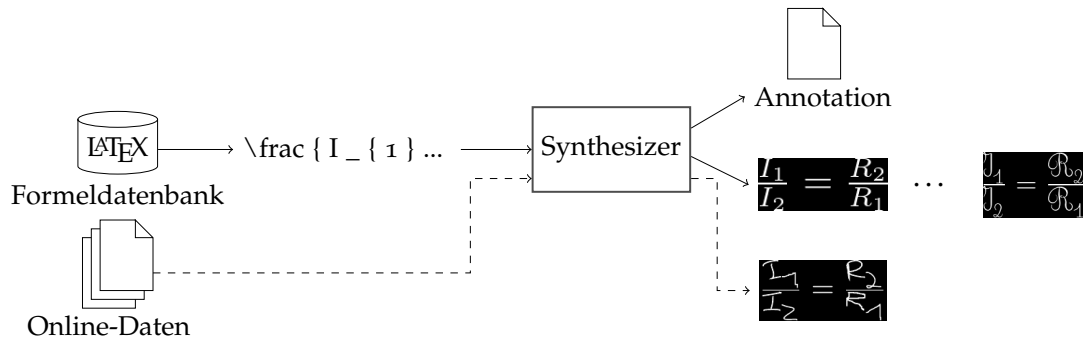


Abbildung 5.2.1: Schematische Funktionsweise des Synthesizers. Online-Symbole entnommen aus den Trainingsdaten der CROHME 2019 [MZM⁺19].

$$y = f(x) f(w \cdot x + b) \left[\sum_{i=1}^n x_i \cdot w_i \geq \theta \right] w_1, \dots, w_n$$

$$y = f\left(\sum_{i=1}^n x_i \cdot w_i + b\right) \left[\frac{2}{1+e^{-2x}} - 1 \right] x \geq 0 \quad x < 0$$

$$L + \lambda \frac{1}{2} w^T w (e_1, \dots, e_T) \sin(\text{pos}/10000^{2l/d}) \left[\frac{e^f_j}{\sum_k e^f_k} \right]$$

$$\mathcal{X} = (x^{(1)}, \dots, x^{(n)}) H(p, q) = - \sum_x p(x) \cdot \log(q(x))$$

$$L = \sum_{i=1}^N L_i \quad a_j = \sum_i z_i w_{ji}$$

Abbildung 5.2.2: Mittels der Fontsynthese gerenderte druckschriftliche Trainingsbilder von mathematischen (Teil-)Ausdrücken aus den vorherigen Kapiteln in dieser Arbeit. Die ersten beiden Ausdrücke haben die reguläre Font, welche in LaTeX-Dokumenten verwendet wird.

5.3 HANDSCHRIFTSYNTHESE

Bei der Handschriftsynthese werden handschriftlich wirkende mathematische Ausdrücke gerendert. Sie ist eine Erweiterung der zuvor vorgestellten Fontsynthese. Der Lösungsansatz ist inspiriert durch [DKLR17] und die zentrale Idee ist, dass anstelle einer Verwendung von Fonts die druckschriftlichen Symbole in einem Ausdruck jeweils durch handschriftliche Symbole ersetzt werden. Der gesamte Prozess zur Herstellung ist dabei zum bisherigen Vorgehen weitestgehend identisch. Der Unterschied ist, dass für die Handschriftsynthese neben einer Datenbank mit Ausdrücken zusätzlich segmentierte Online-Daten mit mathematischen Symbolen benötigt werden, welche vom Synthesizer verarbeitet werden. Der zusätzliche Ablauf bei der Synthese wird in Abbildung 5.3.2 dargestellt.

Die Handschriftsynthese ermöglicht es, bereits vorhandene Online-Datensätze zu verwenden, um handschriftlich wirkende Trainingsbilder auf Basis einer druckschriftlichen Struktur zu erzeugen. Falls keine Online-Daten vorhanden sind ist es möglich, Handschriftproben von einer oder mehreren Personen zusammenzutragen. Obwohl bei einem solchen Vorgehen zusätzlicher Aufwand involviert ist, besteht dieser im Kern ausschließlich darin, für jedes Token ein Beispiel zu erfassen. Ein solcher Aufwand ist dabei im Vergleich zu der manuellen Anfertigung handschriftlicher Trainingsdaten dennoch weitaus geringer. Ein Szenario, bei dem Handschriftproben von einer Per-

son zusammengetragen werden, wird bei dem finalen Experiment in Abschnitt 6.4.5 simuliert.

Die Verarbeitung der Online-Daten findet vor dem Abruf von Ausdrücken aus der Datenbank statt und beginnt, indem in jeder Datei die segmentierten Online-Symbole extrahiert und vorverarbeitet werden. Als Online-Symbol wird im Folgenden eine Sequenz S von n Trajektorien $S = (t_1, \dots, t_n)$ bezeichnet, wobei jede Trajektorie $t_i = ((x_1^{(i)}, y_1^{(i)}), \dots, (x_m^{(i)}, y_m^{(i)}))$ eine Sequenz aus zweidimensionalen Koordinaten ist. Die Länge der Trajektorien innerhalb eines Symbols S kann dabei unterschiedlich sein. Da die Online-Daten, ähnlich wie bei dem Datensatz der CROHME [MZM⁺19], mit verschiedenen Eingabegeräten und jeweils unterschiedlichen Abtastraten der Trajektorien und Skalierungen (siehe [MZM⁺19]) erfasst sein können, wird jedes Symbol bei der Vorverarbeitung normalisiert. Abbildung 5.3.1 veranschaulicht die Trajektorien eines Symbols aus den CROHME-Trainingsdaten.

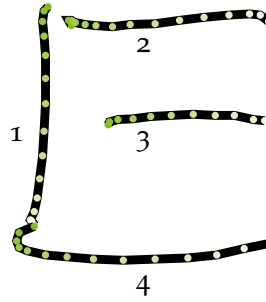


Abbildung 5.3.1: Visualisierung der Trajektorien eines Online-Symbols vom Buchstaben E. Die Zahlen geben die Reihenfolge der Trajektorien an. Die Reihenfolge der Koordinaten einer Trajektorie ist farblich gekennzeichnet. Je heller die Koordinate, desto größer ist die Position in der Reihenfolge. Online-Symbol entnommen aus den Trainingsdaten der CROHME 2019 [MZM⁺19].

Als Normalisierung wird eine dimensionsweise Min-Max-Normalisierung verwendet. Sei S ein Symbol mit Trajektorien $S = (t_1, \dots, t_n)$ und x_{\max} der größte und x_{\min} der kleinste Wert aller x -Koordinaten über alle Trajektorien in S , dann ist die Normalisierung der j -ten x -Koordinate einer Trajektorie t_i definiert als

$$\bar{x}_j^{(i)} = \frac{x_j^{(i)} - x_{\min}}{x_{\max} - x_{\min}}. \quad (5.3.1)$$

Die Definition für die y -Koordinaten einer Trajektorie t_i folgt analog. Die Normalisierung skaliert alle Koordinaten der Trajektorien, sodass nach der Normalisierung für $\bar{x}_{\min} = 0$ und für $\bar{x}_{\max} = 1$ gilt. Dies hat den Effekt, dass sich alle extrahierten

Online-Symbole im Koordinatensystem an einer einheitlichen Position befinden und zusätzlich dieselbe Höhe und Breite haben. Beide Eigenschaften erleichtern das Rendering der Symbole. Eine Schwierigkeit ist, dass Kleinbuchstaben dieselbe Höhe und Breite wie Großbuchstaben haben und daher bei manchen Schreibstilen Buchstaben wie beispielsweise *s* oder *x* nicht eindeutig unterscheidbar sind. Weiterhin werden besonders kleine Symbole wie Punkte, Kommata und `\prime` oder Symbole, bei denen die Höhe verhältnismäßig viel größer ist als die Breite wie `|` und `!` in Folge der Min-Max-Normalisierung unleserlich und daher weiterhin druckschriftlich gerendert. Außerdem werden Elemente mit variablen Breiten wie `\sqrt` oder `\frac` ebenfalls druckschriftlich gerendert.

Nach der Extraktion und Normalisierung der Online-Symbole eines Datums, werden sie in einem Dictionary gespeichert, das ein LaTeX-Symbol auf verschiedene Online-Symbole abbildet.

Schließlich wird das Rendering, wie bei der Fontsynthese, ebenfalls mittels eines LaTeX-Compilers² und einer .tex Vorlage realisiert, indem eine PDF-Datei erzeugt und in eine Bilddatei mit d-DPI konvertiert wird. Angelehnt an [DKLR17] wird jedes Token im LaTeX-Ausdruck durch ein zufällig gewähltes Online-Symbol aus dem Dictionary ersetzt.

Die Online-Symbole werden in der Vorlage mittels des TikZ-Paketes [Tan23] dargestellt. Dazu wird für jedes Online-Symbol *S* eine Grafik angefertigt und jede Trajektorie t_i in *S* gezeichnet, indem jede Koordinate $(x_j^{(i)}, y_j^{(i)})$ mit der nachfolgenden Koordinate $(x_{j+1}^{(i)}, y_{j+1}^{(i)})$ durch eine Linie verbunden wird (siehe Abbildung 5.3.1).

Die Abbildung 5.3.3 veranschaulicht verschiedene synthetische Trainingsbilder. Zu beobachten ist, dass aufgrund der zufälligen Auswahl der Online-Symbole in einem Ausdruck unterschiedliche Stile für das gleiche Symbol verwendet werden. Weiterhin ist zu beobachten, dass die Qualität eines Ausdruckes stark von der Qualität der einzelnen Symbole anhängig ist. Außerdem gibt es Symbole, wie beispielsweise *i* oder *l* im Ausdruck $\sin(\text{pos}/10000^{2i/d})$, die aufgrund des Schreibstils infolge der Normalisierung unleserlich wirken.

5.4 DATENAUGMENTIERUNGEN

Die in diesem Kapitel vorgestellten Syntheseverfahren realisieren das Rendering der Trainingsbilder auf Basis von Druckschrift, welche um handschriftliche Symbole erweitert werden kann. Dies hat jedoch zur Folge, dass die synthetischen Daten wenig Variation bezüglich der Struktur und Grundlinien [DKLR17] aufweisen. Brüche haben

² In diesem Fall: pdfTeX

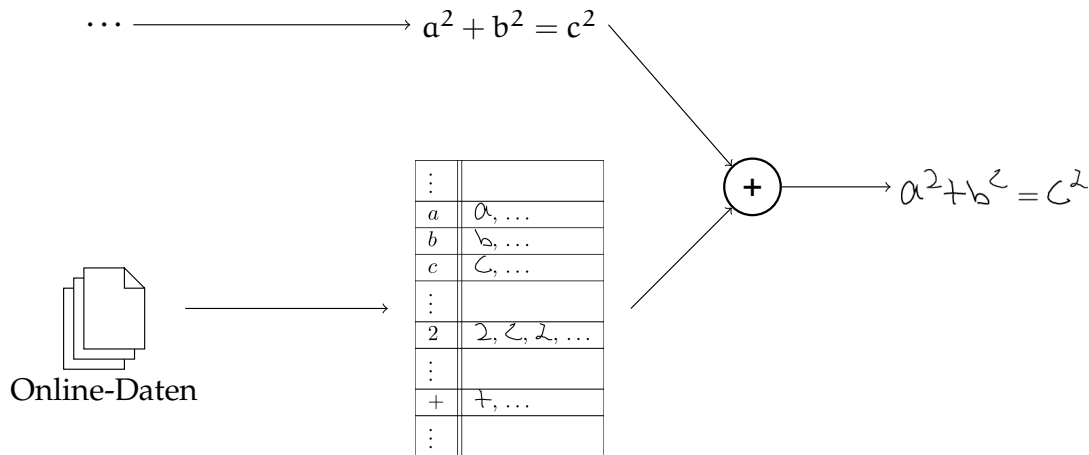


Abbildung 5.3.2: Renderingprozess bei einer Handschriftsynthese im Synthesizer. Mittels Online-Daten wird ein Dictionary (in der Grafik als Tabelle dargestellt) erzeugt und die Symbole in einem Zielausdruck ersetzt. Online-Symbole entnommen aus den Trainingsdaten der CROHME 2019 [MZM⁺19].

beispielsweise immer einen geraden Bruchstrich oder aufeinanderfolgende Symbole sind äußerst präzise horizontal angeordnet. Dies ist bei HMA typischerweise nicht der Fall.

Um solche Charakteristiken in den synthetischen Daten nachzuahmen, werden in dieser Arbeit während des Trainings zwei verschiedene Augmentierungstechniken eingesetzt: die sogenannten affinen Transformationen und die sogenannte „Random Warp Grid Distortion“ [WSD⁺17].

5.4.1 Affine Transformationen

Als affine Transformationen werden verschiedene Operationen bezeichnet, welche sich dadurch auszeichnen, dass gerade Linien nach der Anwendung einer Operation erhalten bleiben. Bei der Anwendung einer affinen Transformation werden die Positionen der Pixel in einem Bild mittels einer Transformationsmatrix verschoben [GW18, S. 100 ff.]. Im Bereich der Handschrifterkennung sind affine Transformationen als Augmentierungstechnik üblich [SF16, WSD⁺17].

$$\begin{aligned}
 & \vartheta = f(x) \quad f(w \cdot x + b) \quad \sum_{i=1}^n x_i \cdot w_i \geq \theta \quad w_1, \dots, w_n \\
 & \vartheta = f(\sum_{i=1}^n x_i \cdot w_i + b) \quad \frac{2}{1+e^{-2x}} \quad x \geq 0 \quad x < 0 \\
 & L + \lambda \frac{1}{2} W^T W \quad (e_1, \dots, e_T) \quad \sin(\text{POS} \wedge \text{ODD} \wedge \text{Z} / d) \\
 & \frac{e^{f_j}}{\sum_k e^{f_k}} \quad X = \langle x^{(1)}, \dots, x^{(n)} \rangle \\
 & H(P, q) = -\sum_x P(x) \cdot \log(q(x)) \quad L = \sum_{i=1}^N L_i \\
 & a_j = \sum_i z_i w_{ji}
 \end{aligned}$$

Abbildung 5.3.3: Aus Abbildung 5.2.2 mittels Handschriftsynthese erzeugte Ausdrücke. Reihenfolge analog zu Abbildung 5.2.2. Online-Symbole entnommen aus den Trainingsdaten der CROHME 2019 [MZM⁺19].

Sei (x, y) eine Pixelkoordinate und T eine Transformationsmatrix, dann ist die neue Koordinate (x', y') nach einer Anwendung einer affinen Transformation wie folgt definiert (Definition nach [GW18, S. 101]):

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = T \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}. \quad (5.4.1)$$

Durch eine geeignete Wahl der Werte a_{ij} in T , lassen sich durch die Operation die Eingabebilder rotieren, skalieren oder scheren [GW18, S. 101]. In dieser Arbeit werden als affine Transformationen die Skalierung (Scaling) und die Scherung (Shear) verwendet. Die Anwendung der affinen Transformationen mit verschiedenen Matrizen werden in Abbildung 5.4.1 veranschaulicht. Zu beobachten ist, dass der Bruchstrich in (d) nach der Transformation nicht mehr waagerecht ist. Im Folgenden findet die Anwendung einer Shear-Transformation sowohl horizontal als auch vertikal statt. Die Parameter für eine Transformation werden bei einer Anwendung während des Trainings jeweils zufällig aus einem vordefinierten Intervall bestimmt, wobei für Scaling $c_x, c_y \in [0, 9; 1, 1]$ und für Shear $s_h, s_v \in [-10; 10]$ gilt. Zur Veranschaulichung der Transformationen wurden in Abbildung 5.4.1 größere Werte für die Parameter gewählt.

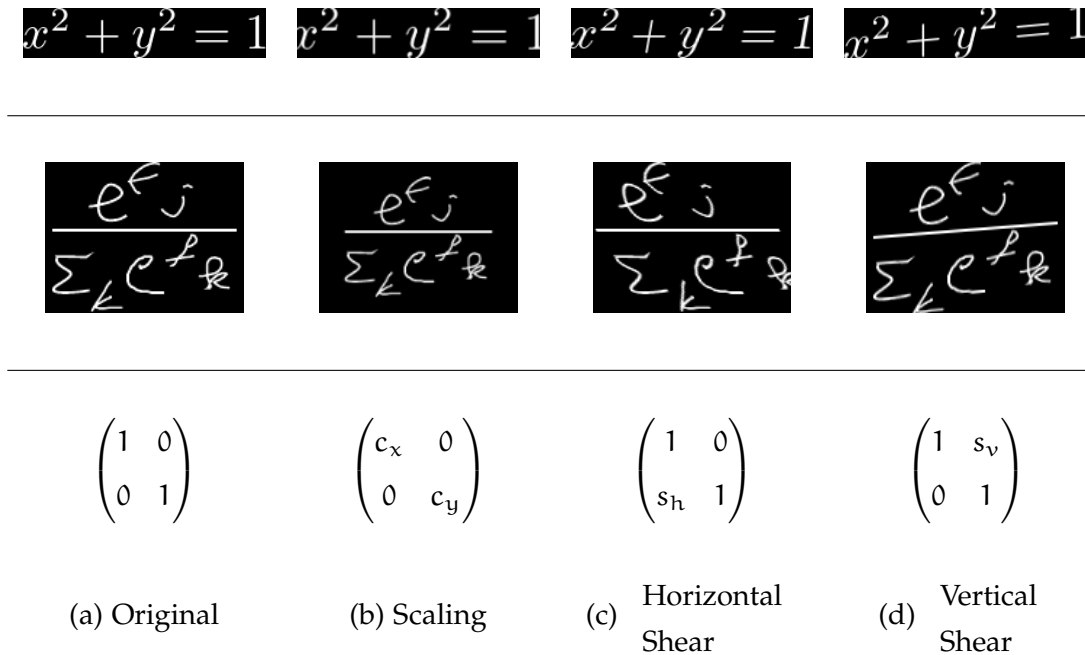
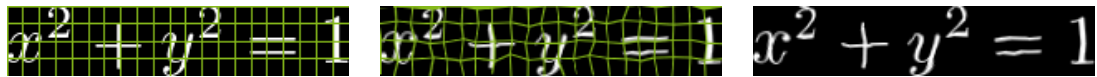


Abbildung 5.4.1: Anwendung verschiedener affiner Transformationen. Definition der Matrizen nach [GW18, S. 102].

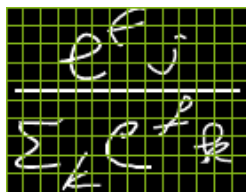
5.4.2 Random Warp Grid Distortion

Die Random Warp Grid Distortion (im Folgenden Grid Distortion) wurde von Wigginton et al. in [WSD⁺17] vorgestellt. Die Grundidee der Grid Distortion ist, dass sogenannte Control-Points in einem Gitter angeordnet werden. Die Control-Points werden danach zufällig verschoben und verzerren somit das Eingabebild. Die zufällige Verschiebung eines Control-Points wird mittels einer Normalverteilung mit einem Mittelwert von 0 und einer Standardabweichung s als Parameter bestimmt. Weiterhin wird zur Anwendung der Grid Distortion der Abstand zwischen den Punkten im Gitter festgelegt. In dieser Arbeit werden bei einer Grid Distortion die Control-Points sowohl horizontal als auch vertikal in Abständen von 10 Pixeln angeordnet und eine Standardabweichung $s = 1$ gewählt.

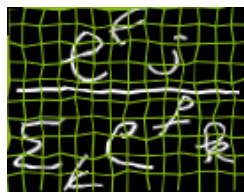
Abbildung 5.4.2 veranschaulicht die Anwendung einer Grid Distortion auf einem Bild aus der Fontsynthese und einem Bild aus der Handschriftsynthese. Zu beobachten ist, dass der Bruchstrich nach der Anwendung nicht mehr gerade ist.



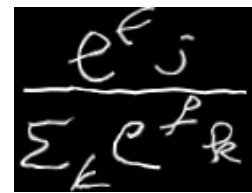
$$x^2 + y^2 = 1$$



$$\frac{e^{\epsilon_j}}{\sum_k c^{\epsilon_k}}$$



$$\frac{e^{\epsilon_j}}{\sum_k c^{\epsilon_k}}$$



$$\frac{e^{\epsilon_j}}{\sum_k c^{\epsilon_k}}$$

Abbildung 5.4.2: Visualisierung der Anwendungsschritte einer Random Warp Grid Distortion auf synthetischen Daten.

EXPERIMENTE

In diesem Kapitel wird das in dieser Arbeit vorgestellte Syntheseverfahren in Verbindung mit einem Deep-Learning-basierten Modell für die Erkennung von handschriftlichen mathematischen Ausdrücken (HMA) untersucht. Dazu werden zunächst unterschiedliche synthetische Datensätze hergestellt und anschließend für das Training des BTTR-Modells [ZGY⁺21] verwendet. Als zentraler Benchmark für die HMA-Erkennung werden in dieser Arbeit die Datensätze und Metriken der „Competition on Recognition of Online Handwritten Mathematical Expressions“ (CROHME) [MZM⁺19] verwendet.

Durch eine experimentelle Evaluierung soll die Performanz bei der HMA-Erkennung mit Hilfe der in Kapitel 5 vorgestellten Datensynthese zusammen mit dem in Abschnitt 6.3 vorgestellten Trainingsaufbau untersucht und folgende Leitfragen beantwortet werden:

- I. Können die synthetischen Trainingsdaten alleine eine ähnliche Performanz wie die echten Daten bei der Erkennung von HMA erzielen?
- II. Welche Ausdrücke sollen synthetisiert werden?
- III. Welches Syntheseverfahren eignet sich besser: Font- oder Handschriftsynthese?
- IV. Welche Augmentierungen sind hilfreich?
- V. Ist es möglich, mittels der synthetischen Daten den Bedarf an echten Trainingsdaten zu reduzieren?

Im Folgenden werden zunächst verschiedene Datensätze vorgestellt. Daraufhin folgt eine Vorstellung der Metriken, welche für die Bewertung der Performanz bei der Erkennung von HMA verwendet werden. Danach wird der allgemeine Trainingsablauf vorgestellt. Im Anschluss werden verschiedene synthetische Datensätze hergestellt und experimentell evaluiert, um die Leitfragen in diesem Kapitel zu beantworten. Die Experimente zu den synthetischen Daten in dieser Arbeit folgen einem schrittweisen Aufbau. In jedem Experiment werden dabei für die Durchführung Erkenntnisse des vorhergehenden Experiments genutzt.

6.1 DATENSÄTZE

Für die Experimente werden verschiedene Datensätze eingesetzt. Diese teilen sich in zwei Kategorien auf: Evaluierung und Ausdrucksextraktion. Der Datensatz zur Evaluierung ist ein Datensatz mit HMA, welcher für das Training und Testen der Performanz auf handschriftlichen Daten eingesetzt wird. Die Datensätze für die Ausdrucksextraktion umfassen LaTeX-Ausdrücke, welche die Datengrundlage für die Datensynthese bilden.

6.1.1 Evaluierung

Für das Training und die Evaluierung auf echten handschriftlichen Daten werden die Datensätze der CROHME aus den Jahren 2014 [MVZG14], 2016 [MVZG16] und 2019 [MZM⁺19] verwendet. Die HMA wurden als Online-Daten von verschiedenen Personen mit einem Stift oder mit dem Finger auf unterschiedlichen Eingabegeräten wie Grafiktablets, Whiteboards und Touchscreengeräten gezeichnet [MVZG14, MZGV16, MVZG16, MZM⁺19]. Die mathematischen Ausdrücke stammen aus unterschiedlichen Quellen, wie Wikipediaartikeln [MVZG14] und arXiv Papern [MZM⁺19] oder wurden zufällig generiert [MZGV16].

Der Trainingsdatensatz der CROHME2014 besteht aus 8836 Trainingsdaten und blieb in den darauffolgenden Wiederholungen weitestgehend¹ unverändert. Die Testdatensätze der Jahre 2014, 2016 und 2019 umfassen jeweils 986, 1149 und 1199 Ausdrücke. Die Datensätze beinhalten insgesamt 101 mathematische Symbolklassen² [MZM⁺19], wobei das BTTR-Modell aufgrund der Formulierung der Ausgaben in LaTeX-Tokens insgesamt 110 Token-Klassen umfasst [Zha21].

Ein Vorteil des Datensatzes ist, dass die Trainingsdaten segmentiert sind und keine Ligaturen beinhalten, sodass jede Trajektorie zu genau einem Symbol zugeordnet ist [MZGV16]. Dies ermöglicht die Nutzung der Trainingsdaten für die Herstellung mittels der Handschriftsynthese bei den Experimenten in Abschnitt 6.4.3 und 6.4.5.

Da der Datensatz Online vorliegt, wird für die Eingabe in das BTTR-Modell eine gerenderte Offline-Version, welche von den Autoren des Modells in [Zha21] bereitgestellt wird, verwendet. Die gerenderten Daten sind in Abbildung 6.1.1 dargestellt.

¹ Bei der vergangenen CROHME2019 wurden die 8836 Trainingsdaten um die Testdaten aus den Jahren 2012 [MVK⁺12] und 2013 [MVZ⁺13] erweitert. Die Erweiterung wird in dieser Arbeit aufgrund der Vergleichbarkeit zu Zhao et al. [ZGY⁺21] nicht verwendet.

² Eine Übersicht über die Symbolklassen befindet sich in [MZGV16].

The image displays a collection of mathematical expressions and symbols arranged in a collage-like fashion. The formulas include:

- $2x^5 - x + 1 = 0$
- $P(x) = \sum_{i=0}^n a_j x^j$
- ΔH_s
- $c^2 = a^2 + b^2 - 2ab \cos C$
- $M-F$
- $\dots - \ell$
- 2.4
- $((177-72) \times (104 \times 164)) \div (79 \times (113 \times 18)) = 11.15$
- $\frac{ab}{2}$
- $\sqrt{1 + \sqrt{2 + \sqrt{3 + \sqrt{4}}}}$
- $\frac{100!}{97! \cdot 6!}$
- $\int_a^b f(x) dx = F(b) - F(a)$
- $a + b = c^2$
- $\frac{\tan \alpha - \tan \beta}{1 + \tan \alpha \tan \beta}$
- $\frac{g+g}{x-y}$
- $\frac{A}{i}$

Abbildung 6.1.1: Offline gerenderte Trainingsdaten des CROHME-Datensatzes [MZM⁺19].
Bereitgestellt von [ZGY⁺21].

6.1.2 Ausdrucksextraktion

NTCIR12 Wikipedia

Der Wikipedia-Korpus des NTCIR12-Datensatz beinhaltet eine Sammlung von mathematischen Ausdrücken. Der Datensatz wurde für die „Math Information Retrieval“-Aufgabe der 12. NTCIR³ Konferenz [ZAK⁺16] präsentiert und umfasst über 590.000 mathematische Ausdrücke aus englischen Wikipediaartikeln.

Bei der CROHME2016 [MVZG16] wurde der Datensatz ebenfalls bereitgestellt, damit die teilnehmenden Systeme mit den Ausdrücken Sprachmodelle wie probabilistische Grammatiken trainieren konnten.

Aus dem Datensatz werden die LaTeX-Ausdrücke aus Tags innerhalb von .html-Dateien entnommen.

HME100k

Der HME100k-Datensatz [YLD⁺22] ist ein Datensatz für die Offline-HMA-Erkennung und beinhaltet auf Papier aufgeschriebene HMA-Bilder. Der Datensatz umfasst insgesamt ungefähr 100.000 Trainingsdaten, welche in 74.502 Trainings- und 24.607 Testdaten aufgeteilt sind. Die Bilder enthalten unterschiedliche Variationen in Farbe, Hintergrund und Illumination. Daher weist der Datensatz im Vergleich zu dem in

³ „NII Testbeds and Community for Information Access Research“

dieser Arbeit vorgestellten Syntheseverfahren eine unterschiedliche Bilddomäne auf und aus diesem Grund werden ausschließlich die Annotationen in LaTeX für die Extraktion der Ausdrücke verwendet.

6.2 METRIKEN

Um die Performanz eines Modells bei der Klassifikation von HMA zu messen wird die von der CROHME verwendete Expression Recognition Rate [MVK⁺12] (im Folgenden ExpRate) eingesetzt. Die ExpRate ähnelt der im Kontext des maschinellen Lernens bekannten Accuracy und ist definiert als

$$\text{Exp} = \frac{\text{Anzahl korrekt erkannter Ausdrücke}}{\text{Anzahl aller Ausdrücke}}. \quad (6.2.1)$$

Die Berechnung der Anzahl an korrekt erkannten Ausdrücken wird dabei basierend auf einer sogenannten Symbol-Label-Graph-Repräsentation [MZM⁺19] realisiert. Der Symbol-Label-Graph eines mathematischen Ausdrucks repräsentiert die im Ausdruck enthaltenen Symbole und die Anordnungsbeziehungen. Ein Ausdruck gilt dabei als korrekt erkannt, wenn dessen Symbol Label Graph mit dem Graphen der Annotation übereinstimmt. Die Repräsentation der mathematischen Ausdrücke als Graphen ermöglicht es, falsch klassifizierte Symbole, Anordnungsbeziehungen oder Strukturen zu ermitteln [MZM⁺19].

Die ExpRate wird zusätzlich für Ausdrücke mit höchstens einem und höchstens zwei Fehlern berechnet. Die Tolerierung der Fehler gewährt insbesondere bei den anfänglichen Experimenten in Abschnitt 6.4.1 und 6.4.2 einen präziseren Einblick in die Arbeitsweise der Modelle. Im Folgenden wird die ExpRate ohne Fehler als E_0 , mit höchstens einem Fehler als E_1 und mit höchstens zwei Fehlern als E_2 bezeichnet.

Neben der ExpRate wird zusätzlich die sogenannte Structure Expression Rate (im Folgenden StructRate oder Str) [MZGV16] berechnet, welche analog zu 6.2.1 definiert ist. Ein Ausdruck gilt dabei als korrekt erkannt, wenn die Struktur unabhängig von den erkannten Symbolen mit der Struktur der Annotation übereinstimmt. Bei der StructRate stimmt beispielsweise die Vorhersage des Ausdrucks $x^2 - 1$ mit einer Annotation $2^a + b$ überein [MZGV16].

Für die finale Berechnung der Metriken mit den LaTeX-Ausdrücken als Ausgabe des BTTR-Modells werden die bereitgestellten offiziellen Evaluierungs-Tools der CROHME2019 [MZM⁺19] verwendet. Beide Metriken werden im Folgenden in den Tabellen und Plots jeweils in Prozent angegeben.

6.3 TRAININGSAUFBAU

In dieser Arbeit wird die PyTorch-Implementierung des BTTR-Modells aus [ZGY⁺21, Zha21] verwendet. Als Werte für die Modellparameter, welche bereits in Kapitel 4 vorgestellt wurden, werden dieselben wie in der Arbeit von Zhao et al. [ZGY⁺21] verwendet und bleiben während des gesamten Trainings unverändert.

Die Trainingsparameter orientieren sich ebenfalls an [ZGY⁺21] und das Training des Modells findet mit einer Batch-Size 8 auf einer NVIDIA Tesla P100 GPU statt. Als Verlustfunktion wird der Cross-Entropy-Loss aus Abschnitt 4.2.4 verwendet und die Anpassungsregel während des Trainings mittels des Gradient Descent ist der ADADELTA-Algorithmus [Zei12]. Weiterhin wurde ein Dropout mit einer Wahrscheinlichkeit von 0,3 angewendet [ZGY⁺21].

Bei dem Einsatz von synthetischen Trainingsdaten setzt sich ein Training, angelehnt an [KJ16b], aus zwei Schritten zusammen: Vortraining und Fine-Tuning. Bei dem Vortraining wird ein Modell auf synthetischen Daten trainiert. Anschließend findet im Fine-Tuning-Schritt ein sogenanntes Transfer-Learning aus der synthetischen Domäne in eine handschriftliche Domäne statt [KJ16b]. Dabei wird das Training eines Modells auf handschriftlichen Daten fortgesetzt, indem es mit den Gewichten aus dem Vortraining initialisiert wird.

Während des Trainings wird in regelmäßigen Abständen zwischen den Epochen eine Validierung durchgeführt und eine E_0 -Rate aus technischen Gründen auf Basis der LaTeX-Tokens berechnet. Als Validierungsdaten werden, ähnlich wie bei der CROHME in den Jahren 2016 [MVZG16] und 2019 [MZM⁺19], die Testdaten aus dem Jahr 2014 verwendet.

Die Learning Rate hat zum Anfang eines Trainings bei einer zufälligen Initialisierung der Gewichte (synthetisches Vortraining oder Baseline-Experiment) den Wert $\epsilon = 1$ und wird bei dem Standardvorgehen nach [Zha21] während des Trainings mittels eines Schedulers dynamisch um Faktor 10 verringert, sobald nach einer festgelegten Anzahl an Epochen keine Verbesserung der E_0 -Rate erreicht wird. Um nach [KJ16b] ein Verlernen der Gewichte des Vortrainings zu verhindern, wird für das Fine-Tuning die initiale Learning-Rate auf den Wert $\epsilon = 0,1$ verringert.

Die Anzahl der Epochen in einem Vortraining ist abhängig von der Größe des Datensatzes. Bei den Experimenten wurde während des Vortrainings auf eine ungefähr gleichlange Trainingsdauer geachtet. Bei einem Fine-Tuning werden alle Modelle aus Gründen der Vergleichbarkeit der unterschiedlichen Synthesen immer mit 100 Epochen trainiert. Modelle ohne synthetisches Vortraining hingegen werden auf handschriftlichen Daten mit mehr Epochen bzw. Iterationen trainiert.

Am Ende eines Trainings werden als Gewichte diejenigen gewählt, welche während der Validierung die beste Performanz erzielten. Nach dem Training wird dasselbe Modell auf den drei unterschiedlichen Testdatensätzen evaluiert.

6.4 ERGEBNISSE

6.4.1 Baseline

Das erste Experiment in dieser Arbeit ist ein sogenanntes Baseline-Experiment, bei dem zunächst noch keine synthetischen Trainingsdaten verwendet werden und das Training ausschließlich auf handschriftlichen Daten stattfindet. Es dient dazu, die Ergebnisse aus der Arbeit von Zhao et al. [ZGY⁺21] zu reproduzieren und eine Grundlage für die nachfolgenden Experimente zu bilden, damit bei einer Nutzung von synthetischen Daten die Performanz verglichen werden kann.

Tabelle 6.4.1 veranschaulicht die Resultate des Baseline-Experiments. Zu beobachten ist, dass die drei Testdatensätze die ExpRates E_1 , E_2 und die StructRate Str sowohl positive als auch negative Abweichungen aufweisen. Bei dem 2016er- und 2019er-Datensatz sind die negativen Abweichungen mit 1,83% und 1,33% am größten und deuten auf die Schwierigkeit bei der Optimierung der ExpRate E_0 hin.

Modell	2014				2016				2019			
	E_0	E_1	E_2	Str	E_0	E_1	E_2	Str	E_0	E_1	E_2	Str
Zhao et al.	53,96	66,02	70,28	71,40	52,31	63,90	68,61	69,40	52,96	65,97	69,14	70,06
Baseline	54,36	65,92	70,28	70,69	50,48	63,64	69,22	69,92	51,63	66,14	69,73	70,23

Tabelle 6.4.1: Baseline-Experiment. Der Eintrag Zhao et al. stammt aus [ZGY⁺21]. Der Eintrag Baseline sind die Resultate bei dem Versuch, die Werte von Zhao et al. zu reproduzieren. Alle Angaben in Prozent.

6.4.2 Ausdrucksquellen

Das Ausdrucksquellen-Experiment ist das erste Experiment, bei dem synthetische Trainingsdaten in einem Vortraining eingesetzt werden. Das Ziel des Experiments ist es zu untersuchen, welche Wahl der synthetisierten Ausdrücke im Vortraining einen positiven Einfluss auf die Performanz der Modelle hat. Dazu werden drei syntheti-

sche Datensätze, welche jeweils unterschiedliche Quellen für die LaTeX-Ausdrücke verwenden, hergestellt.

Bei der Datenextraktion wurden als Quellen für die LaTeX-Ausdrücke die Datensätze NTCIR12, HME100k und CROHME verwendet. Für den Vergleich der unterschiedlichen Quellen wird als Syntheseverfahren für die drei Datensätze eine Fontsynthese mit der standardmäßigen LaTeX-Font verwendet, weil dies die grundlegendste Form der in dieser Arbeit vorgestellten Syntheseverfahren darstellt. Das Vokabular wurde für den weiteren Verlauf aller Experimente auf die Token-Klassen des BTTR-Modells festgelegt. Eine Schwierigkeit bei der Herstellung war die Wahl der DPI, da die gerenderten originalen CROHME-Daten (siehe 6.1.1) in der Höhe und Breite variieren. Nach anfänglichen Renderings wurde die DPI bei der Synthese für den weiteren Verlauf aller Experimente auf den Wert 300 festgelegt. Bei der Synthese der Datensätze wurde versucht, möglichst viele der verfügbaren LaTeX-Ausdrücke herzustellen.

Bei dem NTCIR12-Datensatz (NTCIRSYN) ist es mittels der Fontsynthese gelungen, 97.547 einzigartige synthetische Daten aus den 590.000 LaTeX-Ausdrücken herzustellen. Für das Vortraining wurden die Daten in Trainings- und Validierungsdaten in einem Verhältnis von 90%/10% zufällig aufgeteilt. Weiterhin wurden jeweils fontsynthetische Versionen des HME100k- und des CROHME-Datensatzes hergestellt. Die Synthese der Ausdrücke aus dem CROHME-Datensatz ist hilfreich, um die Performanz auf Ausdrücken, die repräsentativ für die Testdaten sind, zu untersuchen.

Ogleich die echten Datensätze für die HMA-Erkennung bereits in tokenisierter Form vorliegen, mussten beide Datensätze jeweils die Datenextraktion (siehe Abschnitt 5.1) durchlaufen, da vereinzelt unterschiedliche Bilder in den originalen Datensätzen gleiche Labels haben. Ein solcher Effekt ist bei der Fontsynthese zu vermeiden, da zu einem vorgegebenen Label und Font immer das gleiche Bild erzeugt wird.

Der fontsynthetische HME100k-Datensatz (HMESYN) enthält 37.197 Trainings- und 15.010 Testdaten. Die originale Aufteilung der Daten wurde beibehalten, wobei die Testdaten während des Vortrainings zur Validierung verwendet werden. Analog zum HMESYN enthält der fontsynthetische CROHME-Datensatz (CROHMESYN) 4.832 Trainings- und 973 Testdaten. Als Validierungsdaten wurde der Testdatensatz der CROHME 2014 synthetisiert.

Tabelle 6.4.2 vergleicht die Performanz der Modelle, welche auf den unterschiedlichen synthetischen Datensätzen (vor-) trainiert wurden. Dabei wurde direkt nach dem Vortraining eine Evaluierung auf den handschriftlichen Testdaten durchgeführt. Im Anschluss fand ein Fine-Tuning derselben Modelle und eine erneute Evaluierung statt.

Zu beobachten ist, dass die Modelle, welche ausschließlich auf synthetischen Daten trainiert wurden, für keinen Datensatz eine vergleichbare Performanz nahe Baseline erzielen konnten. Somit stellt in dieser Arbeit die Adaption des Modells an die

	Datensatz	2014				2016				2019			
		E ₀	E ₁	E ₂	Str	E ₀	E ₁	E ₂	Str	E ₀	E ₁	E ₂	Str
Ohne Fine-Tuning	ntcirsyn	0,20	1,52	6,39	12,17	0,09	0,96	3,49	10,03	0,08	0,42	1,67	3,84
	hmesyn	0,30	1,83	5,58	9,94	0,00	0,35	2,53	5,49	0,08	0,33	0,58	1,58
	crohmesyn	0,20	1,52	4,67	9,33	0,09	0,79	2,44	5,67	0,08	0,08	0,33	1,58
Mit Fine-Tuning	ntcirsyn	52,33	67,55	72,62	73,73	51,53	65,56	70,71	71,84	52,21	67,39	72,39	73,23
	hmesyn	53,25	66,63	70,28	71,40	52,49	65,21	70,79	71,49	51,21	66,47	70,64	71,48
	crohmesyn	52,13	65,31	69,88	70,59	50,22	63,99	69,05	69,92	49,04	65,72	70,31	71,14
Baseline		54,36	65,92	70,28	70,69	50,48	63,64	69,22	69,92	51,63	66,14	69,73	70,23

Tabelle 6.4.2: Auf synthetischen Datensätzen trainierte Modelle mit jeweils unterschiedlicher Datenquelle. Oben: Evaluierung ohne Fine-Tuning. Mitte: Evaluierung nach dem Fine-Tuning. Unten: Baseline aus Tabelle 6.4.1. Alle Angaben in Prozent.

handschriftliche Domäne während des Fine-Tunings einen essentiellen Schritt in der Synthese-Pipeline dar.

Weiterhin ist zu sehen, dass durch das Fine-Tuning mit echten Trainingsdaten bereits die einfache Fontsynthese mit ausschließlich der Standard-Font bei den Datensätzen HMESYN und NTCIRSYN erste Verbesserungen in der Performanz gegenüber der Baseline erzielen konnte. Über die verschiedenen Metriken und Testjahre weist dabei das Vortraining mit dem NTCIRSYN die meisten Verbesserungen auf. Insbesondere bei den 2019er-Testdaten, ist die Performanz sowohl bei den ExpRates als auch bei der StructRate besser als mit HMESYN oder CROHMESYN.

Während der HMESYN-Datensatz leichte Verbesserungen gegenüber der Baseline aufweist, ist die Performanz bei dem CROHMESYN ähnlich zu der Baseline geblieben oder hat sich verschlechtert. Ein möglicher Grund dafür ist, dass der Datensatz im Vergleich zu den anderen beiden die wenigsten Trainingsdaten enthält. Da der NTCIRSYN mehr als doppelt so viele Trainingsdaten wie der HMESYN Datensatz umfasst, kommt daher die unterschiedliche Größe ebenfalls als Grund für die Verbesserung der Performanz in Betracht. Weiterhin wurde beobachtet, dass der NTCIRSYN-Datensatz bezüglich der LaTeX-Ausdrücke eine größere Schnittmenge mit den LaTeX-Ausdrücken der CROHME-Testdatensätze aufweist als der HMESYN.

Auffällig ist, dass die synthetischen Daten bei der Erkennung der Struktur der HMA helfen, da sich die StructRate bei den synthetischen Datensätzen (bis auf geringe Abweichungen im CROHMESYN) gegenüber der Baseline verbessert hat. Eine Erklärungsmöglichkeit ist, dass die synthetischen Daten aufgrund der einheitlichen Struktur infolge der Nutzung von Druckschrift, eine geringe Varianz bezüglich der Positionen der Symbole aufweisen. Eine solche geringe Varianz in der synthetischen Domäne kann bei der Adaption an die handschriftliche Domäne hilfreich gewesen sein.

Zusammenfassend lässt sich feststellen, dass die synthetischen Daten für die HMA-Erkennung in dieser Arbeit alleine nicht ausreichend sind, da keine vergleichbare Performanz nahe der Baseline zu erzielt werden konnte (Leitfrage I). Weiterhin ist es aufgrund der Anzahl der LaTeX-Ausdrücke hilfreich, für den weiteren Verlauf im Vortraining die LaTeX-Ausdrücke des NTCIR12-Datensatzes zu verwenden (Leitfrage II).

6.4.3 Syntheseverfahren

In diesem Experiment werden die in dieser Arbeit vorgestellten Syntheseverfahren Fontsynthese und Handschriftsynthese miteinander verglichen. Aufbauend auf dem vorhergehenden Experiment werden die LaTeX-Ausdrücke aus dem NTCIR12-Datensatz als Datengrundlage verwendet und zwei Datensätze hergestellt.

Bei dem NTF-Datensatz wird jeder der 97.547 LaTeX-Ausdrücke aus dem NTCIR12-Datensatz mittels der Fontsynthese in 8 verschiedenen Fonts gerendert, sodass der Datensatz insgesamt 780.376 Trainingsbilder umfasst. Anders als bei den bisher vorgestellten synthetischen Datensätzen haben bei diesem Datensatz unterschiedliche Bilder dieselben Labels. Die Daten werden, aufgrund der Vergrößerung um Faktor 8, in einem Verhältnis von 99%/1% in Trainings- und Validierungsdaten aufgeteilt.

Bei dem NTHW-Datensatz werden dieselben LaTeX-Ausdrücke mittels der Handschriftsynthese genau ein mal gerendert. Als Eingabe für die Online-Daten werden die originalen Online-Trainingsdaten des CROHME-Datensatzes verwendet. Insgesamt werden über 85.000 segmentierte Online-Symbole aus den Trainingsdaten extrahiert.

Wegen der großen Anzahl an Trainingsdaten im NTF8-Datensatz wurde für beide Vortrainings anstelle der dynamischen Verringerung der Learning-Rate auf Basis der Validierung zwischen Epochen, die Learning-Rate statisch nach einer festen Anzahl an Iterationen um Faktor 10 verringert.

Tabelle 6.4.3 zeigt die Performanz der beiden Syntheseverfahren. Sowohl mit der Fontsynthese als auch mit der Handschriftsynthese ist es gelungen, die Performanz

Verfahren	2014				2016				2019			
	E ₀	E ₁	E ₂	Str	E ₀	E ₁	E ₂	Str	E ₀	E ₁	E ₂	Str
NTF	53,96	68,26	73,33	74,04	54,58	68,18	73,76	74,80	53,96	69,98	73,81	74,73
NTHW	56,90	69,27	73,63	74,44	56,50	68,27	73,50	74,46	55,21	68,64	71,98	72,56
Baseline	54,36	65,92	70,28	70,69	50,48	63,64	69,22	69,92	51,63	66,14	69,73	70,23

Tabelle 6.4.3: Vergleich der in dieser Arbeit vorgestellten Syntheseverfahren im Vortraining: Fontsynthese mit 8 Fonts (NTF) und Handschriftsynthese (NTHW). Als Datengrundlage wurden die LaTeX-Ausdrücke des NTCIR12 Datensatzes verwendet. Baseline aus Tabelle 6.4.1. Alle Angaben in Prozent.

aus dem vorhergehenden Experiment 6.4.2 weitestgehend zu verbessern. Als Grund für die Verbesserung deutet die Zunahme der Varianz, welche sowohl durch die verschiedenen Fonts als auch durch den Einsatz von Online-Symbolen entsteht, hin.

Obwohl bei dem Trainingsaufbau die Fontsynthese und die Handschriftsynthese eine ähnliche Performanz bei den ExpRates E_1 , E_2 und der StructRate erzielen, hebt sich zwischen den beiden Verfahren die Handschriftsynthese mit einer Verbesserung von mindestens 1,25% bezüglich der ExpRate E_0 gegenüber der Fontsynthese für alle Testdatensätze hervor. Die Zunahme in der Performanz kann damit begründet werden, dass sich die gerenderten Online-Symbole bezüglich Charakteristiken, wie Strichdicke und Form, insgesamt näher an den gerenderten CROHME-Daten orientieren, als die druckschriftlichen Symbole in den unterschiedlichen Fonts.

Insgesamt lässt sich feststellen, dass sich die beiden Syntheseverfahren für den Trainingsaufbau in dieser Arbeit eignen, um die Performanz zu verbessern, wobei sich die Handschriftsynthese besser eignet (Leitfrage III), da im Vergleich zu der Fontsynthese mit weniger synthetischen Trainingsdaten eine bessere Performanz bei der fehlerfreien Erkennung erzielt werden konnte.

6.4.4 *Augmentierungen*

Nun wird in diesem Experiment die Nutzung von verschiedenen Augmentierungstechniken, welche in Kapitel 5 vorgestellt wurden, in Verbindung mit den synthetischen Trainingsdaten untersucht. Als synthetischer Datensatz wird für das Vortraining der gleiche Trainingsaufbau und handschriftsynthetische NTCIR12-Datensatz (NTHW)

wie im vorhergehenden Experiment 6.4.3 eingesetzt. Für alle Kombinationen aus (Random Warp) Grid Distortion, Scaling und Shear wird jeweils ein Modell trainiert.

Die Augmentierungen werden ausschließlich während des Vortrainings auf den synthetischen Daten angewendet. Dabei wird während einer Epoche für ein Datum mit einer Wahrscheinlichkeit von 0,5 zufällig entschieden, ob es augmentiert wird. Bei einer Anwendung mehrerer Augmentierungen wird folgende Reihenfolge beachtet: Grid Distortion, Shear und Scaling.

Augment.	2014				2016				2019			
	E ₀	E ₁	E ₂	Str	E ₀	E ₁	E ₂	Str	E ₀	E ₁	E ₂	Str
None	56,90	69,27	73,63	74,44	56,50	68,27	73,50	74,46	55,21	68,64	71,98	72,56
Grid	56,19	69,57	73,02	74,14	56,23	68,70	73,67	74,37	55,13	69,14	73,31	74,15
Scaling	56,59	69,57	73,33	73,53	55,10	67,22	72,45	72,71	54,88	69,14	73,23	73,89
Shear	56,39	69,68	73,83	75,25	55,97	69,14	73,32	73,84	56,46	70,89	73,81	74,40
Grid, Shear	57,20	70,79	74,04	75,05	54,58	67,48	72,89	72,97	57,72	69,89	73,39	74,31
Grid, Scaling	57,61	70,18	74,14	75,05	55,89	69,40	74,37	74,80	56,13	70,06	73,65	74,40
Shear, Scaling	56,59	68,86	72,62	73,33	56,41	68,96	73,93	74,98	56,71	69,64	72,64	73,31
All	57,00	69,47	73,43	74,14	57,28	67,48	72,97	73,50	57,47	69,89	72,98	73,48

Tabelle 6.4.4: Anwendung verschiedener Kombinationen von Augmentierungstechniken während des Vortrainings auf einem handschriftsynthetischen Datensatz (NTHW). Der Eintrag None bezeichnet keine Anwendung und wurde aus Tabelle 6.4.3 übernommen. All bezeichnet die Anwendung aller vorgestellten Augmentierungen. Die Grid Distortion wurde mit „Grid“ abgekürzt. Alle Angaben in Prozent.

Die Ergebnisse der verschiedenen Kombinationen von Augmentierungstechniken werden in Tabelle 6.4.4 präsentiert. Zu beobachten ist, dass über alle Metriken und Testdatensätze sowohl Verbesserungen als auch Verschlechterungen in der Performanz festgestellt werden können. Entgegen der Erwartung gibt es keine Kombination, welche sich beispielsweise durch eine Verbesserung ExpRate E₀ auf allen Testdatensätzen stark hervorhebt. Ab dem Eintrag „Shear“ bis „All“ lassen sich jedoch Verbesserungen der Metriken für die 2019er-Testdaten beobachten.

Insgesamt betrachtet sorgen die Augmentierungstechniken bei dem Trainingsaufbau nicht für eine starke Verschlechterung der Performanz. Da jedoch nicht eindeutig bestimmt werden konnte, welche Kombination besonders hilfreich ist und die Kombi-

nation aus allen drei Augmentierungen (Eintrag „All“) im Vergleich zu den restlichen Kombinationen ebenfalls keine starke negative Abweichung in der ExpRate E_0 aufweist, wird im Folgenden die Kombination aus allen Augmentierungen eingesetzt (Leitfrage IV).

6.4.5 Reduktion der Trainingsdaten

In dem finalen Experiment in dieser Arbeit wird nun ein Szenario simuliert, bei dem nur wenige Trainingsdaten verfügbar sind. Dazu werden die Erkenntnisse aus den vorhergehenden Experimenten zusammengefasst und untersucht, inwieweit es mittels der synthetischen Daten möglich ist, den Bedarf an echten handschriftlichen Daten zu decken. Dazu werden die bereits vorhandenen echten Trainingsdaten reduziert und jeweils zwei Modelle trainiert: ein Modell, das ausschließlich auf den reduzierten handschriftlichen Daten trainiert wird und ein Modell, das ein synthetisches Vortraining durchläuft. Im Anschluss werden beide Modelle auf die Performanz untersucht.

Bei einem Vortraining wird als Syntheseverfahren die Handschriftsynthese mit dem NTCIR12-Datensatz als Grundlage für die LaTeX-Ausdrücke eingesetzt. Anders als im vorhergehenden Experiment wird für die Online-Symbole nicht der gesamte CROHME-Trainingsdatensatz verwendet. Stattdessen wird für jedes ersetzbare LaTeX-Token aus der Handschriftsynthese genau ein Online-Symbol aus dem CROHME-Trainingsdatensatz extrahiert. Dies hat einerseits den Grund, dass ein Modell, das mit synthetischen Daten trainiert wird, bei der Reduktion von echten Trainingsdaten keinen Vorteil bezüglich der Information über die Symbole des gesamten Trainingsdatensatzes erhält. Andererseits soll damit das Szenario simuliert werden, bei dem keine Online-Daten vorhanden sind und somit Handschriftproben von einer fiktiven Person für die Handschriftsynthese zusammengetragen werden.

Die Wahl der Symbole fand manuell statt und es wurde darauf geachtet, dass sie nach der Normalisierung (siehe Abschnitt 5.3) leserlich sind. Abbildung 6.4.1 veranschaulicht die zusammengetragene fiktive Handschriftprobe, welche für die Handschriftsynthese eingesetzt wird. Als Augmentierungen wird nach Abschnitt 6.4.4 die Kombination aus allen in dieser Arbeit vorgestellten Augmentierungen eingesetzt.

Für die handschriftlichen Daten wurde zunächst der CROHME-Trainingsdatensatz zufällig in 10 gleich große Teile aufgeteilt. Danach wurden die Trainingsdaten schrittweise kumuliert, bis die ursprüngliche Größe erreicht wurde. Während des Trainings wurde ein Modell ohne synthetisches Vortraining auf den echten Trainingsdaten länger trainiert, als ein Modell mit synthetischem Vortraining.

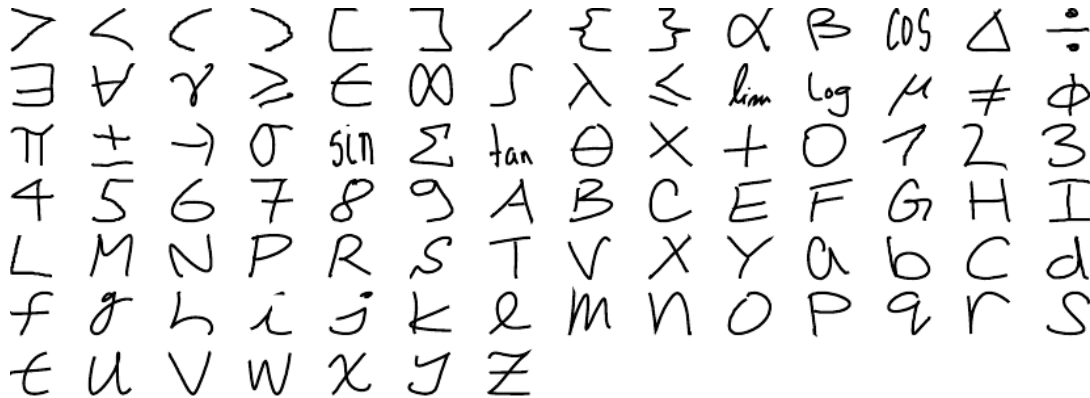


Abbildung 6.4.1: Fiktive Handschriftprobe für die Datensynthese. Die Symbole wurden nach den jeweiligen LaTeX-Tokens lexikographisch angeordnet. Online-Symbole entnommen aus den Trainingsdaten der CROHME 2019 [MZM⁺19].

Abbildung 6.4.2 veranschaulicht die Performanz der Modelle, welche auf unterschiedlich großen Teilen des CROHME-Datensatzes trainiert wurden, für die Metriken ExpRate E_0 (obere Reihe) und StructRate Str (untere Reihe). Die Werte für 100% der Trainingsdaten wurden aus dem Baseline-Experiment 6.4.1 übernommen und durch eine horizontale Linie kenntlich gemacht. Zu beobachten ist, dass die Modelle mit einem synthetischen Vortraining für beide Metriken (bis auf eine geringe Abweichung bei der E_0 für 100% der Trainingsdaten im 2014er Testjahr) eine durchgehend bessere Performanz aufweisen als die Modelle, welche direkt mit echten Trainingsdaten trainiert wurden. Die Verbesserung der Performanz bei der ExpRate E_0 nimmt bei abfallender Anzahl an echten Trainingsdaten zu und ist für den Fall, dass nur 10% der echten Trainingsdaten vorhanden sind, mit einer Differenz von 13,19%, 16,57% und 14,18% bei den Testjahren 2014, 2016 und 2019 am größten⁴. Insgesamt ist zu beobachten, dass sich mit Hilfe der synthetischen Trainingsdaten die Anzahl an echten Trainingsdaten mit vereinzelt geringen Einbußen bei der ExpRate E_0 auf circa 60% verringern lässt (Leitfrage V).

⁴ Die exakten Werte des Experiments und Plots für die ExpRates E_1 und E_2 befinden sich im Anhang (siehe Tabelle A.0.1 und Abbildung A.0.1).

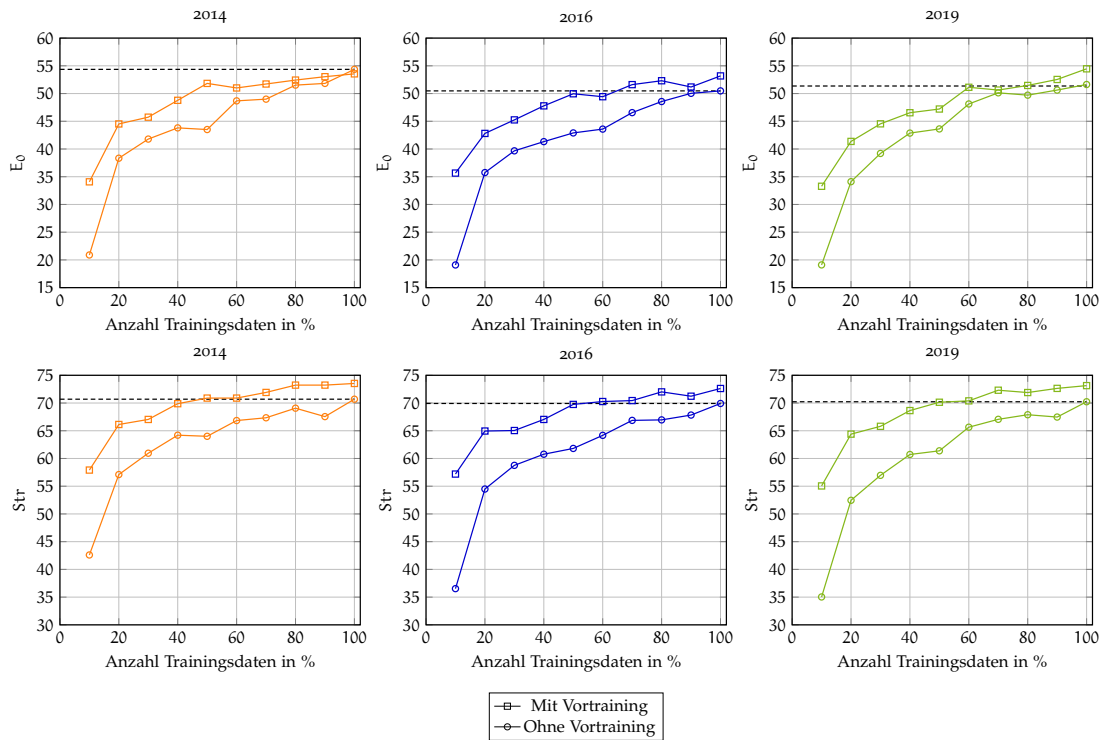


Abbildung 6.4.2: Vergleich der Performanz von Modellen, welche auf einem Teil der echten handschriftlichen CROHME-Trainingsdaten trainiert wurden, jeweils mit und ohne synthetisches Vortraining. Die Werte für 100% ohne synthetisches Vortraining sind die Baseline aus 6.4.1 und wurden durch eine horizontale Linie hervorgehoben. Die exakten Werte und die Plots für die ExpRates E_1 , E_2 befinden sich in Tabelle A.0.1 und Abbildung A.0.1. Alle Angaben in Prozent.

FAZIT

In dieser Arbeit wurden die Auswirkungen eines Trainings auf synthetischen Offline-Daten bei der Erkennung von handschriftlichen mathematischen Ausdrücken (HMA) untersucht. Dazu wurde ein Synthesizer vorgestellt, der fähig ist, Bilder von synthetischen mathematischen Ausdrücken zu erzeugen. Der Synthesizer verfügt dabei über zwei Syntheseverfahren: die Fontsynthese und die Handschriftsynthese. Bei einer Fontsynthese können druckschriftliche Ausdrücke in verschiedenen Fonts gerendert werden. Die Handschriftsynthese erweitert das fontsynthetische Verfahren, indem durch die Eingabe von Online-Daten handschriftlich wirkende mathematische Ausdrücke auf Basis einer druckschriftlichen Struktur gerendert werden.

Anschließend wurde die Datensynthese schrittweise mittels des in der Forschungsgemeinde etablierten CROHME-Benchmarks [MZM⁺19] und dem Deep-Learning-basierten BTTR-Modell [ZGY⁺21] experimentell evaluiert. Für die Durchführung der Experimente wurden verschiedene Leitfragen formuliert, welche innerhalb der Experimente beantwortet wurden.

Zunächst wurde durch ein Experiment, bei dem einfache fontsynthetische Datensätze mit der Standard-Font und Ausdrücken aus verschiedenen Quellen hergestellt wurden, festgestellt, dass bereits erste Verbesserungen der Performanz im Vergleich zu einem puren Training mit echten Daten erzielt werden konnten. Weiterhin wurde beobachtet, dass die verschiedenen Ausdrucksquellen unterschiedliche Performanzen aufgewiesen haben, wobei mit einer zunehmenden Größe eine höhere Performanz einherging. Außerdem konnte festgestellt werden, dass das Fine-Tuning [KJ16a] mittels echter Daten einen unabdingbaren Schritt in der Synthese-Pipeline (siehe Abbildung 5.0.1) darstellt, da andernfalls ein alleiniges Training mit synthetischen Daten eine schlechte Performanz bei der Erkennung von handschriftlichen Daten aufweist.

In dem darauffolgenden Experiment wurden dann die Fontsynthese und die Handschriftsynthese miteinander verglichen, indem aus derselben Ausdrucksquelle jeweils zwei Datensätze hergestellt wurden. Sowohl durch die Zunahme von Fonts bei der Fontsynthese als auch durch die Zugabe von Online-Daten bei der Handschriftsynthese konnten erneut Verbesserungen in der Performanz erzielt werden. In dem direkten Vergleich der beiden Syntheseverfahren konnte sich dabei die Handschriftsynthese hervorheben.

Im Anschluss wurden in einem weiteren Experiment die Auswirkungen verschiedener Kombinationen von Augmentierungstechniken auf einem handschriftsynthetischen Datensatz untersucht. Dabei hatte sich keine Kombination durch eine weitestgehende Verbesserung der Performanz hervorgehoben.

In einem finalen Experiment wurden schließlich die Erkenntnisse aus den vorherigen Experimenten genutzt, um die Auswirkungen der Datensynthese bei der Reduktion der echten Trainingsdaten zu untersuchen. Zusätzlich wurde ein Szenario simuliert, bei dem Handschriftproben von einer fiktiven Person für die Handschriftsynthese zusammengetragen wurden. Insgesamt ließ sich mit Hilfe der synthetischen Daten die Anzahl der echten Trainingsdaten ohne starke Einbußen in der Performanz auf 60% verringern.

Zusammenfassend ist festzustellen, dass eine Datensynthese für den in dieser Arbeit diskutierten Bereich der HMA-Erkennung sowohl bei der Verbesserung der Performanz als auch bei der Verringerung der echten Trainingsdaten eingesetzt werden kann.

In der kommenden CROHME in diesem Jahr¹ wurde angekündigt, dass es neben Offline-gerenderten Ausdrücken, zusätzlich auch gescannte Bilder von handschriftlichen Ausdrücken geben wird. In einer weiteren Arbeit könnte somit versucht werden, durch eine Erweiterung der Syntheseverfahren um zusätzliche Nachbearbeitungsschritte (ähnlich wie in [KJ16a]) die Charakteristiken der gescannten Daten, wie beispielsweise die Pixelverteilung der Vorder- und Hintergründe von den echten Daten, nachzuahmen.

¹ <https://crohme2023.ltu-ai.dev/>

ANHANG

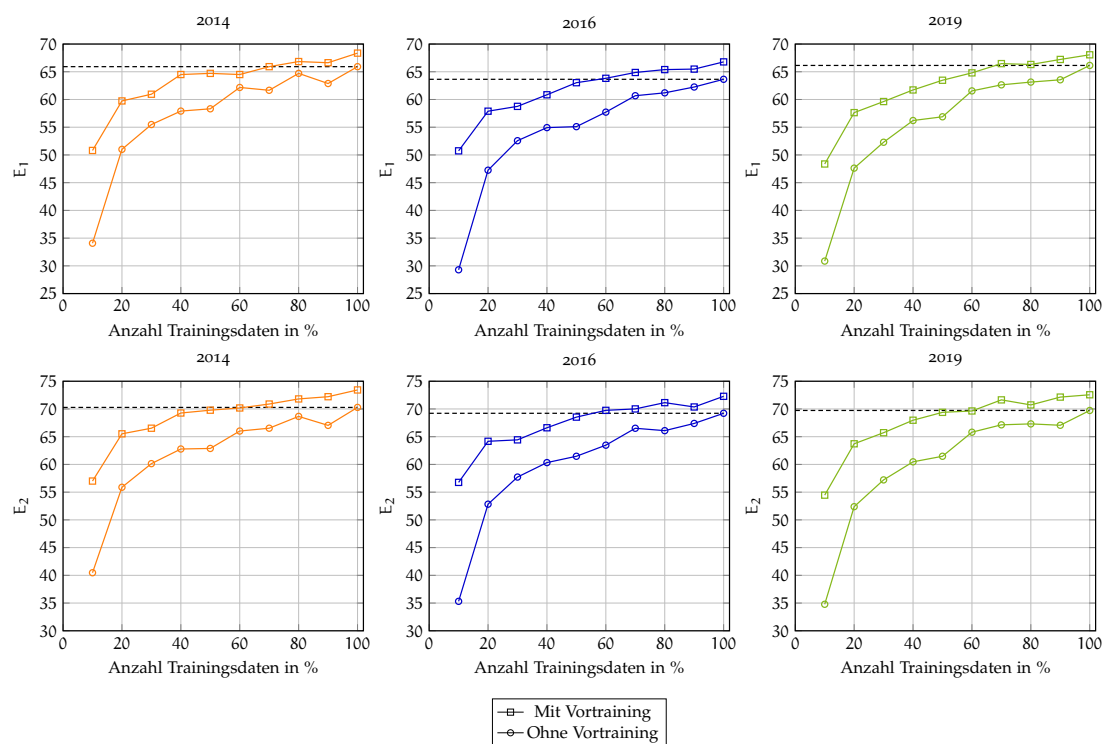


Abbildung A.0.1: Ergänzende Plots für die ExpRates E_1 , E_2 des Experiments 6.4.5 zur Reduktion der Trainingsdaten. Alle Angaben in Prozent.

	%	2014				2016				2019			
		E ₀	E ₁	E ₂	Str	E ₀	E ₁	E ₂	Str	E ₀	E ₁	E ₂	Str
Mit Vortraining	10	34,08	50,81	57,00	57,91	35,66	50,74	56,76	57,19	33,28	48,37	54,46	55,05
	20	44,52	59,74	65,52	66,13	42,81	57,89	64,17	64,95	41,37	57,63	63,72	64,39
	30	45,74	60,95	66,53	67,04	45,25	58,76	64,43	65,04	44,54	59,63	65,72	65,80
	40	48,78	64,50	69,27	69,88	47,78	60,85	66,61	67,04	46,54	61,72	67,97	68,64
	50	51,83	64,71	69,78	70,89	49,96	63,03	68,53	69,75	47,21	63,47	69,39	70,14
	60	51,01	64,50	70,18	70,89	49,43	63,82	69,75	70,27	51,13	64,80	69,64	70,39
	70	51,72	65,92	70,89	71,91	51,61	64,87	70,01	70,44	50,63	66,47	71,64	72,31
	80	52,43	66,84	71,81	73,23	52,31	65,39	71,14	72,01	51,46	66,31	70,73	71,89
	90	53,04	66,63	72,21	73,23	51,18	65,48	70,36	71,23	52,54	67,22	72,14	72,64
	100	53,55	68,36	73,43	73,53	53,18	66,78	72,28	72,62	54,46	68,06	72,56	73,14
Ohne Vortraining	10	20,89	34,08	40,47	42,60	19,09	29,29	35,31	36,53	19,10	30,86	34,78	35,03
	20	38,34	51,01	55,88	57,10	35,75	47,25	52,83	54,49	34,11	47,62	52,38	52,46
	30	41,79	55,48	60,14	60,95	39,67	52,57	57,72	58,76	39,20	52,29	57,21	56,96
	40	43,81	57,91	62,78	64,20	41,33	54,93	60,33	60,77	42,87	56,21	60,47	60,72
	50	43,51	58,32	62,88	64,00	42,90	55,10	61,47	61,81	43,62	56,88	61,47	61,38
	60	48,68	62,17	66,02	66,84	43,59	57,72	63,47	64,17	48,12	61,55	65,81	65,64
	70	48,99	61,66	66,53	67,34	46,56	60,68	66,52	66,87	50,13	62,64	67,14	67,06
	80	51,52	64,71	68,66	69,07	48,56	61,20	66,09	66,96	49,71	63,14	67,31	67,89
	90	51,83	62,88	67,04	67,55	50,04	62,25	67,39	67,83	50,63	63,55	67,06	67,47
	Baseline	54,36	65,92	70,28	70,69	50,48	63,64	69,22	69,92	51,63	66,14	69,73	70,23

Tabelle A.o.1: Die exakten Werte des Experiments 6.4.5 zur Reduktion der Trainingsdaten. Die Einträge bezeichnen Modelle, welche auf unterschiedlich großen Teilen der echten handschriftlichen Trainingsdaten trainiert wurden. Oben: Modelle mit einem synthetischen Vortraining. Unten: ausschließlich auf handschriftlichen Daten trainierte Modelle. Die Werte werden in den Plots der Abbildungen 6.4.2 und A.o.1 visualisiert. Der Eintrag Baseline wurde aus Tabelle 6.4.1 übernommen. Alle Angaben in Prozent.

LITERATURVERZEICHNIS

- [And67] ANDERSON, Robert H.: Syntax-directed recognition of hand-printed two-dimensional mathematics. In: *Proc. ACM Symposium on Interactive Systems for Experimental Applied Mathematics*. Washington, D.C., USA, 1967, S. 436–459
- [BCB15] BAHDANAU, Dzmitry ; CHO, Kyunghyun ; BENGIO, Yoshua: Neural Machine Translation by Jointly Learning to Align and Translate. In: *Proc. Int. Conf. on Learning Representations (ICLR)*. San Diego, CA, USA, 2015
- [Bis95] BISHOP, Christopher M.: *Neural networks for pattern recognition*. Oxford, United Kingdom : Clarendon Press, 1995. – ISBN 9780198538646
- [Bis06] BISHOP, Christopher M.: *Pattern Recognition and Machine Learning*. New York, NY, USA : Springer, 2006. – ISBN 9780387310732
- [BKH16] BA, Lei J. ; KIROS, Jamie R. ; HINTON, Geoffrey E.: Layer Normalization. In: *CoRR abs/1607.06450* (2016). <http://arxiv.org/abs/1607.06450>
- [Bur23] BURNOL, Jean-François: *Paket mathastext*. <https://ctan.org/pkg/mathastext>. Version: 2023. – Letzter Zugriff: 09.02.2023
- [CGCB14] CHUNG, Junyoung ; GÜLÇEHRE, Çağlar ; CHO, KyungHyun ; BENGIO, Yoshua: Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. In: *CoRR abs/1412.3555* (2014)
- [CMG⁺14] CHO, Kyunghyun ; MERRIENBOER, Bart van ; GÜLÇEHRE, Çağlar ; BAHDANAU, Dzmitry ; BOUGARES, Fethi ; SCHWENK, Holger ; BENGIO, Yoshua: Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. In: *Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar, 2014, S. 1724–1734
- [CMS⁺20] CARION, Nicolas ; MASSA, Francisco ; SYNNAEVE, Gabriel ; USUNIER, Nicolas ; KIRILLOV, Alexander ; ZAGORUYKO, Sergey: End-to-End Object Detection with Transformers. In: *Proc. European Conf. on Computer Vision (ECCV)*. Glasgow, United Kingdom, 2020, S. 213–229

- [CY00] CHAN, Kam-Fai ; YEUNG, Dit-Yan: Mathematical expression recognition: a survey. In: *Int. Journal on Document Analysis and Recognition (IJ DAR)* 3 (2000), Nr. 1, S. 3–15
- [DCLT19] DEVLIN, Jacob ; CHANG, Ming-Wei ; LEE, Kenton ; TOUTANOVA, Kristina: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In: *Proc. Conf. of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*. Minneapolis, MN, USA, 2019, S. 4171–4186
- [DKLR17] DENG, Yuntian ; KANERVISTO, Anssi ; LING, Jeffrey ; RUSH, Alexander M.: Image-to-Markup Generation with Coarse-to-Fine Attention. In: *Proc. Int. Conf. on Machine Learning (ICML)*. Sydney, Australia, 2017, S. 980–989
- [FDK⁺18] FRID-ADAR, Maayan ; DIAMANT, Idit ; KLANG, Eyal ; AMITAI, Michal ; GOLDBERGER, Jacob ; GREENSPAN, Hayit: GAN-based synthetic medical image augmentation for increased CNN performance in liver lesion classification. In: *Neurocomputing* 321 (2018), S. 321–331
- [FM82] FUKUSHIMA, Kunihiko ; MIYAKE, Sei: Neocognitron: A new algorithm for pattern recognition tolerant of deformations and shifts in position. In: *Pattern Recognition* 15 (1982), Nr. 6, S. 455–469
- [GBB11] GLOT, Xavier ; BORDES, Antoine ; BENGIO, Yoshua: Deep Sparse Rectifier Neural Networks. In: *Proc. Int. Conf. on Artificial Intelligence and Statistics (AISTATS)*. Fort Lauderdale, FL, USA, 2011, S. 315–323
- [GBC16] GOODFELLOW, Ian ; BENGIO, Yoshua ; COURVILLE, Aaron: *Deep Learning*. MIT Press, 2016. – <http://www.deeplearningbook.org>
- [GFGSo6] GRAVES, Alex ; FERNÁNDEZ, Santiago ; GOMEZ, Faustino J. ; SCHMIDHUBER, Jürgen: Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In: *Proc. Int. Conf. on Machine Learning (ICML)*. Pittsburgh, PA, USA, 2006, S. 369–376
- [GPM⁺20] GOODFELLOW, Ian J. ; POUGET-ABADIE, Jean ; MIRZA, Mehdi ; XU, Bing ; WARDE-FARLEY, David ; OZAIR, Sherjil ; COURVILLE, Aaron C. ; BENGIO, Yoshua: Generative adversarial networks. In: *Communications of the ACM* 63 (2020), Nr. 11, S. 139–144
- [Gra13] GRAVES, Alex: Generating Sequences With Recurrent Neural Networks. In: *CoRR abs/1308.0850* (2013). <http://arxiv.org/abs/1308.0850>

- [GW18] GONZALEZ, Rafael C. ; WOODS, Richard E.: *Digital image processing*. Fourth edition, global edition. New York, NY, USA : Pearson, 2018. – ISBN 9781292223049
- [HLMW17] HUANG, Gao ; LIU, Zhuang ; MAATEN, Laurens van d. ; WEINBERGER, Kilian Q.: Densely Connected Convolutional Networks. In: *Proc. IEEE Comp. Soc. Conf. on Computer Vision and Pattern Recognition (CVPR)*. Honolulu, HI, USA : IEEE Computer Society, 2017, S. 2261–2269
- [HM15] HIRSCHBERG, Julia ; MANNING, Christopher D.: Advances in natural language processing. In: *Science* 349 (2015), Nr. 6245, S. 261–266
- [HS97] HOCHREITER, Sepp ; SCHMIDHUBER, Jürgen: Long Short-Term Memory. In: *Neural Computation* 9 (1997), Nr. 8, S. 1735–1780
- [HSW89] HORNIK, Kurt ; STINCHCOMBE, Maxwell B. ; WHITE, Halbert: Multilayer feedforward networks are universal approximators. In: *Neural Networks* 2 (1989), Nr. 5, S. 359–366
- [HZRS16] HE, Kaiming ; ZHANG, Xiangyu ; REN, Shaoqing ; SUN, Jian: Deep Residual Learning for Image Recognition. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, IEEE Computer Society, 2016, S. 770–778
- [JSVZ14] JADERBERG, Max ; SIMONYAN, Karen ; VEDALDI, Andrea ; ZISSERMAN, Andrew: Synthetic Data and Artificial Neural Networks for Natural Scene Text Recognition. In: *CoRR* abs/1406.2227 (2014). <http://arxiv.org/abs/1406.2227>
- [KB15] KINGMA, Diederik P. ; BA, Jimmy: Adam: A Method for Stochastic Optimization. In: *Proc. Int. Conf. on Learning Representations (ICLR)*. San Diego, CA, USA, 2015
- [KBB⁺15] KRUSE, R. ; BORGELT, C. ; BRAUNE, C. ; KLAWONN, F. ; MOEWES, C. ; STEINBRECHER, M.: *Computational Intelligence: Eine methodische Einführung in Künstliche Neuronale Netze, Evolutionäre Algorithmen, Fuzzy-Systeme und Bayes-Netze*. Springer Fachmedien Wiesbaden, 2015. – ISBN 9783658109042
- [KJ16a] KRISHNAN, Praveen ; JAWAHAR, C. V.: Generating Synthetic Data for Text Recognition. In: *CoRR* abs/1608.04224 (2016). <http://arxiv.org/abs/1608.04224>

- [KJ16b] KRISHNAN, Praveen ; JAWAHAR, C. V.: Matching Handwritten Document Images. In: *Proc. European Conf. on Computer Vision (ECCV)*. Amsterdam, Netherlands, 2016, S. 766–782
- [KSH12] KRIZHEVSKY, Alex ; SUTSKEVER, Ilya ; HINTON, Geoffrey E.: ImageNet Classification with Deep Convolutional Neural Networks. In: *Advances in Neural Information Processing Systems (NIPS)*. Lake Tahoe, NV, USA, 2012, S. 1106–1114
- [KUNP19] KHUONG, Vu Tran M. ; UNG, Quang H. ; NAKAGAWA, Masaki ; PHAN, Khanh M.: Generating Synthetic Handwritten Mathematical Expressions from a LaTeX Sequence or a MathML Script. In: *Proc. Int. Conf. on Document Analysis and Recognition (ICDAR)*. Sydney, Australia, 2019, S. 922–927
- [Lam94] LAMPORT, Leslie: *LATEX: a document preparation system: user's guide and reference manual*. 2nd ed. Reading, MA, USA : Addison-Wesley, 1994. – ISBN 9780201529838
- [LB05] LIWICKI, Marcus ; BUNKE, Horst: IAM-OnDB - an On-Line English Sentence Database Acquired from Handwritten Text on a Whiteboard. In: *Proc. Int. Conf. on Document Analysis and Recognition (ICDAR)*. Seoul, Korea, 2005, S. 956–961
- [LBBH98] LECUN, Yann ; BOTTOU, Léon ; BENGIO, Yoshua ; HAFNER, Patrick: Gradient-based learning applied to document recognition. In: *Proc. of the IEEE* 86 (1998), Nr. 11, S. 2278–2324
- [LBH15] LECUN, Yann ; BENGIO, Yoshua ; HINTON, Geoffrey E.: Deep learning. In: *Nature* 521 (2015), Nr. 7553, S. 436–444
- [LBOM12] LECUN, Yann ; BOTTOU, Léon ; ORR, Genevieve B. ; MÜLLER, Klaus-Robert: Efficient BackProp. In: *Neural Networks: Tricks of the Trade*. Second Edition. Springer, 2012, S. 9–48
- [LSD15] LONG, Jonathan ; SHELHAMER, Evan ; DARRELL, Trevor: Fully convolutional networks for semantic segmentation. In: *Proc. IEEE Comp. Soc. Conf. on Computer Vision and Pattern Recognition (CVPR)*. Boston, MA, USA, 2015, S. 3431–3440
- [LT20] LIN, Hong ; TAN, Jun: Application of Deep Learning in Handwritten Mathematical Expressions Recognition. In: *Proc. Int. Conf. on Pattern*

- Recognition and Artificial Intelligence (ICPRAI)*. Zhongshan, China, 2020, S. 137–147
- [LUF^S16] LIU, Lemao ; UTIYAMA, Masao ; FINCH, Andrew M. ; SUMITA, Eiichiro: Agreement on Target-bidirectional Neural Machine Translation. In: *Proc. North American Chapter of the Association for Computational Linguistics*. San Diego, CA, USA, 2016, S. 411–416
- [LWG] LI, Fei-Fei ; WU, Jiajun ; GAO, Ruohan: *Stanford University CS231n: Deep Learning for Computer Vision*. <http://cs231n.stanford.edu/>. – Letzter Zugriff: 30.01.2023
- [Mit97] MITCHELL, Tom M.: *Machine learning*. International Edition. McGraw-Hill, 1997. – ISBN 9780070428072
- [MP43] MCCULLOCH, Warren S. ; PITTS, Walter: A logical calculus of the ideas immanent in nervous activity. In: *The Bulletin of Mathematical Biophysics* 5 (1943), Nr. 4, S. 115–133
- [MSP⁺17] MEURER, Aaron ; SMITH, Christopher P. ; PAPROCKI, Mateusz ; ČERTÍK, Ondřej ; KIRPICHEV, Sergey B. ; ROCKLIN, Matthew ; KUMAR, Amit ; IVANOV, Sergiu ; MOORE, Jason K. ; SINGH, Sartaj ; RATHNAYAKE, Thilina ; VIG, Sean ; GRANGER, Brian E. ; MULLER, Richard P. ; BONAZZI, Francesco ; GUPTA, Harsh ; VATS, Shivam ; JOHANSSON, Fredrik ; PEDREGOSA, Fabian ; CURRY, Matthew J. ; TERREL, Andy R. ; ROUČKA, Štěpán ; SABOO, Ashutosh ; FERNANDO, Isuru ; KULAL, Sumith ; CIMRMAN, Robert ; SCOPATZ, Anthony: SymPy: symbolic computing in Python. In: *PeerJ Computer Science* 3 (2017), S. e103
- [Mur12] MURPHY, Kevin P.: *Machine learning: a probabilistic perspective*. Cambridge, MA, USA : MIT Press, 2012. – ISBN 9780262018029
- [MVK⁺11] MOUCHÈRE, Harold ; VIARD-GAUDIN, Christian ; KIM, Dae H. ; KIM, Jin H. ; GARAIN, Utpal: CROHME2011: Competition on Recognition of Online Handwritten Mathematical Expressions. In: *Proc. Int. Conf. on Document Analysis and Recognition (ICDAR)*. Beijing, China, 2011, S. 1497–1500
- [MVK⁺12] MOUCHÈRE, Harold ; VIARD-GAUDIN, Christian ; KIM, Dae H. ; KIM, Jin H. ; GARAIN, Utpal: ICFHR 2012 Competition on Recognition of On-Line Mathematical Expressions (CROHME 2012). In: *Proc. Int. Conf. on Frontiers in Handwriting Recognition (ICFHR)*. Bari, Italy, 2012, S. 811–816

- [MVZ⁺₁₃] MOUCHÈRE, Harold ; VIARD-GAUDIN, Christian ; ZANIBBI, Richard ; GARAIN, Utpal ; KIM, Dae H.: ICDAR 2013 CROHME: Third International Competition on Recognition of Online Handwritten Mathematical Expressions. In: *Proc. Int. Conf. on Document Analysis and Recognition (ICDAR)*. Washington, D.C., USA, 2013, S. 1428–1432
- [MVZG₁₄] MOUCHÈRE, Harold ; VIARD-GAUDIN, Christian ; ZANIBBI, Richard ; GARAIN, Utpal: ICFHR 2014 Competition on Recognition of On-Line Handwritten Mathematical Expressions (CROHME 2014). In: *Proc. Int. Conf. on Frontiers in Handwriting Recognition (ICFHR)*. Crete, Greece, 2014, S. 791–796
- [MVZG₁₆] MOUCHÈRE, Harold ; VIARD-GAUDIN, Christian ; ZANIBBI, Richard ; GARAIN, Utpal: ICFHR2016 CROHME: Competition on Recognition of Online Handwritten Mathematical Expressions. In: *Proc. Int. Conf. on Frontiers in Handwriting Recognition (ICFHR)*. Shenzhen, China, 2016, S. 607–612
- [MZGV₁₆] MOUCHÈRE, Harold ; ZANIBBI, Richard ; GARAIN, Utpal ; VIARD-GAUDIN, Christian: Advancing the state of the art for handwritten math recognition: the CROHME competitions, 2011-2014. In: *Int. Journal on Document Analysis and Recognition (IJ DAR)* 19 (2016), Nr. 2, S. 173–189
- [MZM⁺₁₉] MAHDAVI, Mahshad ; ZANIBBI, Richard ; MOUCHÈRE, Harold ; VIARD-GAUDIN, Christian ; GARAIN, Utpal: ICDAR 2019 CROHME + TFD: Competition on Recognition of Handwritten Mathematical Expressions and Typeset Formula Detection. In: *Proc. Int. Conf. on Document Analysis and Recognition (ICDAR)*. Sydney, Australia : IEEE, 2019, S. 1533–1538
- [Nie₁₅] NIELSEN, Michael A.: *Neural Networks and Deep Learning*. Determination Press, 2015
- [Nik₁₉] NIKOLENKO, Sergey I.: Synthetic Data for Deep Learning. In: *CoRR* abs/1909.11512 (2019). <http://arxiv.org/abs/1909.11512>
- [Par₁₃] PARR, Terence: *The Definitive ANTLR 4 Reference*. Second edition. Dallas, TX, USA : Pragmatic Bookshelf, 2013. – ISBN 9781934356999
- [PMB₁₃] PASCANU, Razvan ; MIKOLOV, Tomás ; BENGIO, Yoshua: On the difficulty of training recurrent neural networks. In: *Proc. Int. Conf. on Machine Learning (ICML)*. Atlanta, GA, USA, 2013, S. 1310–1318

- [PSoo] PLAMONDON, Réjean ; SRIHARI, Sargur N.: On-Line and Off-Line Handwriting Recognition: A Comprehensive Survey. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 22 (2000), Nr. 1, S. 63–84
- [PVU⁺18] PARMAR, Niki ; VASWANI, Ashish ; USZKOREIT, Jakob ; KAISER, Lukasz ; SHAZEER, Noam ; KU, Alexander ; TRAN, Dustin: Image Transformer. In: *Proc. Int. Conf. on Machine Learning (ICML)*. Stockholm, Sweden, 2018, S. 4052–4061
- [Qui86] QUINLAN, J. R.: Induction of decision trees. In: *Machine Learning* 1 (1986), Nr. 1, S. 81–106
- [RDS⁺15] RUSSAKOVSKY, Olga ; DENG, Jia ; SU, Hao ; KRAUSE, Jonathan ; SATHEESH, Sanjeev ; MA, Sean ; HUANG, Zhiheng ; KARPATHY, Andrej ; KHOSLA, Aditya ; BERNSTEIN, Michael ; BERG, Alexander C. ; FEI-FEI, Li: ImageNet Large Scale Visual Recognition Challenge. In: *Int. Journal of Computer Vision (IJCV)* 115 (2015), Nr. 3, S. 211–252
- [RHW86] RUMELHART, David E. ; HINTON, Geoffrey E. ; WILLIAMS, Ronald J.: Learning representations by back-propagating errors. In: *Nature* 323 (1986), Nr. 6088, S. 533–536
- [RN21] RUSSELL, Stuart ; NORVIG, Peter: *Artificial Intelligence, Global Edition A Modern Approach*. Pearson Deutschland, 2021. – 1168 S. – ISBN 9781292401133
- [Roj96] ROJAS, Raul: *Neural Networks - A Systematic Introduction*. Berlin, Germany : Springer-Verlag, 1996 <https://page.mi.fu-berlin.de/rojas/neural/>
- [Ros58] ROSENBLATT, F.: The perceptron: A probabilistic model for information storage and organization in the brain. In: *Psychological Review* 65 (1958), S. 386–408
- [RSM⁺16] ROS, Germán ; SELLART, Laura ; MATERZYNSKA, Joanna ; VÁZQUEZ, David ; LÓPEZ, Antonio M.: The SYNTHIA Dataset: A Large Collection of Synthetic Images for Semantic Segmentation of Urban Scenes. In: *Proc. IEEE Comp. Soc. Conf. on Computer Vision and Pattern Recognition (CVPR)*. Las Vegas, NV, USA, 2016, S. 3234–3243
- [Sch14] SCHMIDHUBER, Jürgen: Deep Learning in Neural Networks: An Overview. In: *CoRR abs/1404.7828* (2014). <http://arxiv.org/abs/1404.7828>

- [SF16] SUDHOLT, Sebastian ; FINK, Gernot A.: PHOCNet: A Deep Convolutional Neural Network for Word Spotting in Handwritten Documents. In: *Proc. Int. Conf. on Frontiers in Handwriting Recognition (ICFHR)*. Shenzhen, China, 2016, S. 277–282
- [SHK⁺14] SRIVASTAVA, Nitish ; HINTON, Geoffrey E. ; KRIZHEVSKY, Alex ; SUTSKEVER, Ilya ; SALAKHUTDINOV, Ruslan: Dropout: a simple way to prevent neural networks from overfitting. In: *Journal of Machine Learning Research (JMLR)* 15 (2014), Nr. 1, S. 1929–1958
- [SME21] SPRINGSTEIN, Matthias ; MÜLLER-BUDACK, Eric ; EWERTH, Ralph: Un-supervised Training Data Generation of Handwritten Formulas using Generative Adversarial Networks with Self-Attention. In: *Proc. Workshop on Multi-Modal Pre-Training for Multimedia Understanding*. Taipei, Taiwan, 2021, S. 46–54
- [SZ15] SIMONYAN, Karen ; ZISSERMAN, Andrew: Very Deep Convolutional Networks for Large-Scale Image Recognition. In: *Proc. Int. Conf. on Learning Representations (ICLR)*. San Diego, CA, USA, 2015
- [Tan23] TANTAU, Till: *Paket TikZ*. <https://www.ctan.org/pkg/pgf>. Version: 2023. – Letzter Zugriff: 26.02.2023
- [tex23] *TeX Live - TeX Users Group*. <https://www.tug.org/texlive/>. Version: 2023. – Letzter Zugriff: 09.02.2023
- [TLL⁺16] TU, Zhaopeng ; LU, Zhengdong ; LIU, Yang ; LIU, Xiaohua ; LI, Hang: Modeling Coverage for Neural Machine Translation. In: *Proc. Annual Meeting of the Association for Computational Linguistics (ACL)*. Berlin, Germany, 2016, S. 76–85
- [Vap10] VAPNIK, Vladimir N.: *The nature of statistical learning theory*. Second edition. New York, NY, USA : Springer, 2010 (Statistics for engineering and information science). – ISBN 9781441931603
- [Vel23] VELIČKOVIĆ, Petar: *Github Repository TikZ (MIT Lizenz)*. <https://github.com/PetarV-/TikZ>. Version: 2023. – Letzter Zugriff: 08.01.2023 (Commit 86aee04)
- [VSP⁺17] VASWANI, Ashish ; SHAZEER, Noam ; PARMAR, Niki ; USZKOREIT, Jakob ; JONES, Llion ; GOMEZ, Aidan N. ; KAISER, Lukasz ; POLOSUKHIN, Illia:

- Attention is All you Need. In: *Advances in Neural Information Processing Systems (NIPS)*. Long Beach, CA, USA, 2017, S. 5998–6008
- [WSD⁺17] WIGINGTON, Curtis ; STEWART, Seth ; DAVIS, Brian L. ; BARRETT, Bill ; PRICE, Brian L. ; COHEN, Scott: Data Augmentation for Recognition of Handwritten Words and Lines Using a CNN-LSTM Network. In: *Proc. Int. Conf. on Document Analysis and Recognition (ICDAR)*. Kyoto, Japan, 2017, S. 639–645
- [YLD⁺22] YUAN, Ye ; LIU, Xiao ; DIKUBAB, Wondimu ; LIU, Hui ; JI, Zhilong ; WU, Zhongqin ; BAI, Xiang: Syntax-Aware Network for Handwritten Mathematical Expression Recognition. In: *Proc. IEEE Comp. Soc. Conf. on Computer Vision and Pattern Recognition (CVPR)*. New Orleans, LA, USA, 2022, S. 4543–4552
- [ZAK⁺16] ZANIBBI, Richard ; AIZAWA, Akiko ; KOHLHASE, Michael ; OUNIS, Iadh ; TOPIC, Goran ; DAVILA, Kenny: NTCIR-12 MathIR Task Overview. In: *Proc. NTCIR Conf. on Evaluation of Information Access Technologies (NTCIR)*. Tokyo, Japan, 2016, S. 299–308
- [ZDD18] ZHANG, Jianshu ; DU, Jun ; DAI, Lirong: Multi-Scale Attention with Dense Encoder for Handwritten Mathematical Expression Recognition. In: *Proc. Int. Conf. on Pattern Recognition (ICPR)*. Beijing, China, 2018, S. 2245–2250
- [ZDZ⁺17] ZHANG, Jianshu ; DU, Jun ; ZHANG, Shiliang ; LIU, Dan ; HU, Yulong ; HU, Jin-Shui ; WEI, Si ; DAI, Li-Rong: Watch, attend and parse: An end-to-end neural network based approach to handwritten mathematical expression recognition. In: *Pattern Recognition* 71 (2017), S. 196–206
- [Zei12] ZEILER, Matthew D.: ADADELTA: An Adaptive Learning Rate Method. In: *CoRR* abs/1212.5701 (2012). <http://arxiv.org/abs/1212.5701>
- [ZG22] ZHAO, Wenqi ; GAO, Liangcai: CoMER: Modeling Coverage for Transformer-Based Handwritten Mathematical Expression Recognition. In: *Proc. European Conf. on Computer Vision (ECCV)*. Tel Aviv, Israel, 2022, S. 392–408
- [ZGY⁺21] ZHAO, Wenqi ; GAO, Liangcai ; YAN, Zuoyu ; PENG, Shuai ; DU, Lin ; ZHANG, Ziyin: Handwritten Mathematical Expression Recognition with Bidirectionally Trained Transformer. In: *Proc. Int. Conf. on Document Analysis and Recognition (ICDAR)*. Lausanne, Switzerland, 2021, S. 570–584

- [Zha21] ZHAO, Wenqi: *GitHub - BTTR*. <https://github.com/Green-Wood/BTTR>.
Version: 2021. – Letzter Zugriff: 29.01.2023 (Commit a3f544b)