technische universität
dortmund

Spatial Transformer Networks zur Klassifizierung von Ziffern

Marc Ahlers
31. Juli 2019

Supervisors:
Prof. Dr.-Ing. Gernot A. Fink
Fabian Wolf, M.Sc.

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

INHALTSVERZEICHNIS

1 EINLEITUNG 3
- 1.1 Rahmen der Arbeit 4

2 GRUNDLAGEN 7
- 2.1 Die Mustererkennung 7
- 2.2 Convolutional Neural Networks 8
  - 2.2.1 Convolution-Layer 10
  - 2.2.2 Pooling-Layer 13
  - 2.2.3 Fully-Connected-Layer 17
  - 2.2.4 Aktivierungsfunktion 17
  - 2.2.5 Ausgabe und Loss 19
  - 2.2.6 Backpropagation und Optimierung 21
  - 2.2.7 Regularisierung und Overfitting 22

3 VERWANDTE ARBEITEN 25
- 3.1 Spatial Transformer Networks 26
- 3.2 Diffeomorphe Transformationen 29
- 3.3 Augmentierung 31

4 METHODIK 35
- 4.1 Architektur 35
- 4.2 Training 40

5 EXPERIMENTE UND EVALUATION 43
- 5.1 Datensätze 44
  - 5.1.1 MNIST Datensatz 45
  - 5.1.2 Rotated / Distorted MNIST 45
  - 5.1.3 SVHN Datensatz (Format 1/2) 46
  - 5.1.4 Augmented SVHN 47
- 5.2 Klassifikation einzelner Ziffern 48
- 5.3 Klassifikation mehrerer Ziffern 51
  - 5.3.1 Augmentierung und Vorverarbeitung 52
  - 5.3.2 Augmentierung und Pooling 55
  - 5.3.3 Original und Pooling 57
  - 5.3.4 Diffeomorphe Transformationen 58

6 FAZIT 61

1 EINLEITUNG

Die Klassifikation von Bildern ist ein lang bestehendes Forschungsfeld der Informatik. Dabei soll der Inhalt des Bildes maschinell verarbeitet und einer oder mehreren von vorher festgelegten Klassen zugeordnet werden.
Diese Grundidee findet beispielsweise Anwendung in der autonomen Autoindustrie (AV). Die Sensoren und die dahinterstehende Datenverarbeitung muss in der Lage sein, aus Bildinformationen semantisch relevante Merkmale — beispielsweise die Form von Objekten — zu extrahieren. So muss eine Ampel oder andere Verkehrsteilnehmer lokalisiert, Tiefeninformationen bestimmt und im Zusammenhang analysiert werden [AAD+19].
Andererseits besteht das grundlegende Klassifikationsproblem auch in der optischen sowie handschriftlichen Zeichenerkennung (OCR/HWR)[IIN17][PMR+16]: Das Transkribieren von maschinell oder handschriftlich geschriebenen Texten. Die Vielfalt an verschiedenen Handschriften für ein und dasselbe Wort erschwert dabei die Erkennung [PT15]. Ein Klassifikator, der das gegebene Problem lösen soll, muss in der Lage sein relevante Bildbereiche zu erkennen und irrelevante — bzw. nicht klassenrelevante — Bereiche zu ignorieren. Zudem soll ein Klassifikator unabhängig von einem Schreibstil (HWR), wechselnden Hintergrund oder Blickwinkel (AV) korrekt arbeiten. Hinzukommen fotometrische Unterschiede wie Farbtreue, Kontrast und Schärfe abhängig von den benutzen Sensoren. Ein konkreter Anwendungsfall wäre die Worterkennung aus natürlichen Szenen, welche zunächst versucht die geometrischen Transformationen — wie z. B. Translation, Rotation oder Skalierung — auszugleichen und daraufhin das Wort zu transkribieren [LCW+16]. Beispielhaft ist das Vorgehen der Texterkennung in natürlichen Szenen in Abbildung 1.0.1 dargestellt.

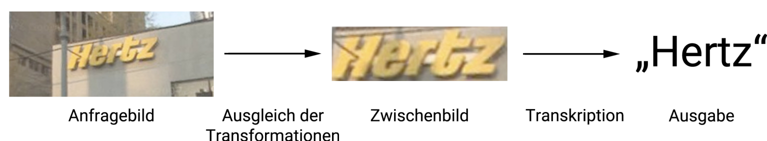

Abbildung 1.0.1: *Simplifiziertes Vorgehen der Texterkennung in natürlichen Szenen (Szenentext) aus [Wan14].*

Die Erkennung von Objekten in natürlichen Szenen hat in mehreren Bereichen des täglichen Lebens Anwendung gefunden und wird in diesen immer wichtiger. Wörter in natürlichen Szenen werden auch als Szenentext bezeichnet [LHY18]. Für Anwender ist dies hinsichtlich Echtzeit-Texterkennung ein interessantes Thema, um sich beispielsweise in einem fremden Land in Kombination mit einer Übersetzungssoftware zurecht zu finden, Informationen zu verstehen und ggf. weiterzugeben [LLY+18]. Falls die ursprüngliche Sprache nicht bekannt ist, könnte diese mithilfe von [BHYM17] bestimmt werden.
Das Klassifikationsproblem ist also in vielen verschiedenen Bereichen des täglichen Lebens vertreten und wird dort immer wichtiger. Ebenso können mehrere verschiedene Klassifikatoren gebündelt ein größeres Klassifikationsproblem lösen, wie z. B. das hier vertretene Problem der mehrstelligen Ziffernerkennung.

1.1 RAHMEN DER ARBEIT

Diese Arbeit hingegen widmet sich der Erkennung von Ziffern in gescannten Bildausschnitten und in natürlichen Szenen. Dies ist eine Untergruppe der Texterkennung und findet Anwendung in der Transkription von Hausnummern aus Bildern mit geometrischen und fotometrischen Transformationen.

Abbildung 1.1.1: *Trainingsbilder (v.l.) Nummer 30450, 19883 und 18092 aus dem SVHN Datensatz [NWC+11].*

Die Arbeit beschreibt wie aus Anfragebildern, beispielsweise dargestellt in Abbildung 1.1.1, die dementsprechenden Zahlen in korrekter Reihenfolge klassifiziert werden können. [LBBH98] vergleicht verschiedene Ansätze und es stellt sich heraus, dass sich *Convolutional Neural Networks* (CNN) für dieses Klassifikationsproblem besonders eignen. Zu erkennen sind jedoch starke Abweichungen in der relativen Zahlengröße bezogen auf die Bildgröße und verschiedene Transformationen, welche die Erkennung und Lokalisierung deutlich erschwert, wie in Kapitel 5 weiter untersucht wird.

Dazu wird ein spezielles Modul betrachtet, welches sich um den Ausgleich affiner Transformationen kümmert. Dieses *Spatial Transformer Network* (STN) kann mehrfach und an verschiedenen Stellen innerhalb eines CNNs eingesetzt werden, um die räumliche Varianz der Eingabe zu minimieren und robuster auf unbekannte Eingaben zu reagieren. Das Modul soll den relevanten Bereich der Eingabe erkennen und so verarbeiten, dass die darauffolgende Klassifikation eine geringere Fehlerrate erzielt.

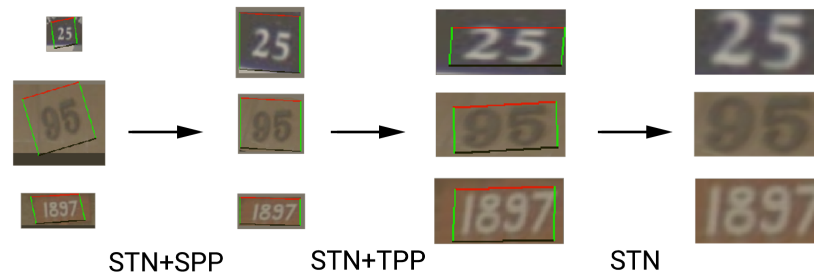

Abbildung 1.1.2: *Mehrere STNs die nacheinander die Eingabe transformieren und jedesmal versuchen den relevanten Bildbereich herauszufiltern. Die Eingabe hat dabei verschiedene Dimensionen und stammt aus dem SVHN Datensatz [NWC+11]. Das Rechteck, mit einer oberen roten Kante, stellt annähernd das Grid (vgl. Abschnitt 3.1) dar, welches benutzt wird um das Bild für den folgenden Schritt zu erzeugen.*

Zudem kann es zusammen mit verschiedenen Poolingtechniken (vgl. Abschnitt 2.2.2 und Abbildung 1.1.2) kombiniert werden um mit Eingabebildern unterschiedlicher Dimensionen umgehen zu können. Aufgrund von anlernbaren (vgl. Abschnitt 2.2.6) Bildmerkmalen (vgl. Abschnitt 2.2.1) werden die Parameter für eine Transformation (vgl. Formel 3.1.1) bestimmt, die den Fokus mehr auf die relevanten und für die Klassifizierung wichtigen Bereiche der Eingabe verschieben sollen.
Zunächst widmet sich die Arbeit in Kapitel 2 den theoretischen Grundlagen der Mustererkennung und speziell den *Convolutional Neural Networks* (Abschnitt 2.2). Daraufhin wird in Kapitel 3 Bezug auf verwandte Arbeiten genommen und die für diese Arbeit relevanten Techniken bzw. Ansätze vorgestellt. In Kapitel 4 wird das allgemeine Vorgehen eines jeden Experiment beschrieben und unter Beachtung welcher Parameter ein Netzwerk angelernt wird. Anschließend werden in Kapitel 5 mehrere Experimente mit verschiedenen Datensätzen und/oder Architekturen vorgestellt und evaluiert woraufhin in Kapitel 6 ein zusammenfassendes Fazit gezogen wird.

2 GRUNDLAGEN

Es gibt viele verschiedene Ansätze wie ein solches Klassifikationsproblem im Bereich der Mustererkennung gelöst werden kann [YD15]. In [LBBH98] werden beispielsweise verschiedene Ansätze auf dem gleichem Datensatz miteinander verglichen. Unter anderem finden sich dort k-NN oder Multiclass SVM Klassifikatoren sowie Neurale Netze die in [CSG18] detailliert verglichen werden. Es stellt sich heraus, dass *Convolutional Neural Networks* (CNN), welche speziell für Bilder als Eingabedaten angepasst wurden, besonders für ein solches Problem geeignet sind.

2.1 DIE MUSTERERKENNUNG

Die Mustererkennung — welches das Forschungsgebiet von u. a. den hier behandelten Klassifikationsproblemen ist — kann wie in [Nie83] beschrieben werden:

> Die Mustererkennung beschäftigt sich mit den mathematisch-technischen Aspekten der automatischen Verarbeitung und Auswertung von Mustern. Es wird für ein physikalisches Signal (z. B. Sprache, Bild, Messwert) eine geeignete Symbolkette (bzw. in eine formale Datenstruktur) berechnet. Dazu gehört sowohl die Klassifikation einfacher Muster als auch die Klassifikation und Analyse komplexer Muster. ([Nie83, 13-14])

Ziel ist es also die Mustererkennungsfunktion zu bestimmen, welche bei einer beliebigen Eingabe diesem ein Symbol bzw. eine Klasse (oder eine Sequenz davon) zuordnet. Viele Ansätze dieser automatischen Verarbeitung innerhalb der Mustererkennung haben zwei Grundprinzipien gemeinsam: Die Merkmalsextraktion und die Klassifizierung. Dabei kann u. a. die Eingabe noch vorverarbeitet und eine Informationsreduzierung implementiert werden. Abbildung 2.1.1 stellt dieses Grundprinzip dar.

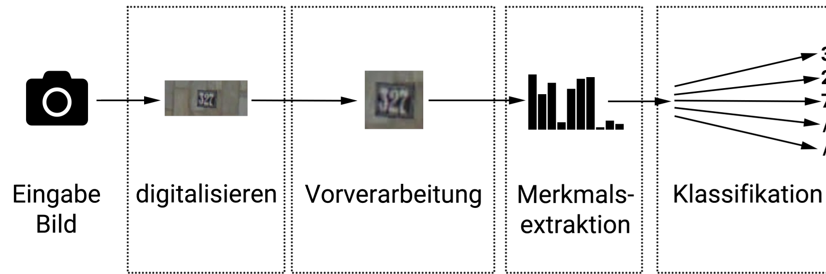

Abbildung 2.1.1: *Klassische Erkennungspipeline illustriert anhand eines Bildes des Street View House Number Datensatzen [NWC+11]. Die Kamera Vektorgrafik stammt aus [Fon19].*

Dabei existieren in jedem Schritt aus Abbildung 2.1.1 verschiedene Techniken, die es gilt passend zur Problemdomäne auszuwählen, um die Fehlerrate des Klassifikators (beispielsweise k-NN, Multiclass SVM, Softmax) zu minimieren. Dabei kann die Merkmalsextraktion beispielsweise durch HOG-Deskriptoren [LBBH98] [DT05] oder durch Aktivierungsvolumina von Faltungsschichten [LBBH98] realisiert werden. k-NN bestimmt die Klasse anhand von annotierten Trainingsdaten und ordnet unbekannten Daten der überwiegenden Klasse der nächsten k-Nachbarn im mehrdimensionalen Raum der Merkmale zu. Multiclass SVM versucht in diesem mehrdimensionalen Raum mehrere Grenzen zu definieren, die dann insgesamt einen Bereich der jeweiligen Klasse bestimmen. Softmax hingegen bestimmt ein Wahrscheinlichkeitsvektor aller Klassen, der sich auf 1 aufsummiert.

2.2 CONVOLUTIONAL NEURAL NETWORKS

Wie in [LBBH98] und [CSG18] beschrieben, eignen sich *Neural Networks* gut um einen Klassifikator zu approximieren [LBBH98]. Ein *Convolutional Neural Network* ist ein neuronale Netz, welches primär auf visuellen Daten arbeitet und bildet damit einen Spezialfall der neuronalen Netze. Dieser Klassifikator besteht aus mehreren Komponenten, die in diesem Kapitel jeweils vorgestellt, beschrieben und in Zusammenhang mit anderen Komponenten gebracht werden. Dieser soll aufgrund der Bildinformationen und davon abgeleiteten Informationen eine Klasse als Ausgabe bestimmen.
Es existieren verschiedene Ansätze wie der Lernprozess eines solchen Netzwerks aussieht [Nil96]. In dem hier behandelten *supervised learning* werden vorher innerhalb der Problemdomäne alle möglichen Klassen definiert und der Datensatz mit diesen annotiert. Dies ist im weiteren Verlauf des Lernprozesses wichtig, um die zugrundeliegende komplexe Funktion der Mustererkennung zu approximieren (vgl.

Abschnitt 2.2.6). Vorab müssen 11 unterschiedliche Klassen definiert werden, wovon 10 Klassen die Ziffern des Dezimalsystems 0-9 repräsentieren und eine zusätzliche Klasse das Fehlen einer Ziffer darstellt. Im Fall von einstelligen Anfragebildern reicht dementsprechend ein Klassifikator mit nur 10 Klassen, da a priori bekannt ist, dass es sich um genau eine Ziffer handelt. Abgleitete Informationen werden als Merkmale (*features*) bezeichnet und können zum Beispiel das Vorhandensein einer vertikalen Kante an einer bestimmten Position oder in deren direkten Umfeld anzeigen. Die Filter, welche die Merkmalsausprägung an einer bestimmten Position berechnen, müssen nicht a priori bekannt sein, sondern werden im Rahmen des Anlernen (vgl. Kapitel 2.2.6) bezogen auf die Problemdomäne schrittweise bestimmt und optimiert [Nie15]. Innerhalb eines *Convolutional Neural Network* kann es viele solcher Filter geben, die gebündelt in einer Faltungsschicht (*Convolution-Layer*, vgl. Abschnitt 2.2.1) auftreten und auf einer gemeinsamen Eingabe ihre Aktivierung, also die Ausprägung eines Merkmals, berechnen.

In diesem Abschnitt werden die Komponenten eines *Convolutional Neural Networks* (*CNN*) untersucht und beschrieben. Dieses bildet die Basis der Szenentexterkennung dieser Arbeit, wobei jedoch weiterführende Techniken in diese Basis eingesetzt werden, um die Fehlerrate eines CNNs zu verbessern und es robuster für unbekannte Daten zu gestalten. Neuronale Netze bestehen aus vielen Neuronen, wobei jedes Neuron eine Eingabe (beispielsweise die Ausgaben vorheriger Neuronen) bekommt. Die Neuronen bilden den Grundbaustein der neuronalen Netze und sind mitunter ihr wichtigster Bestandteil.

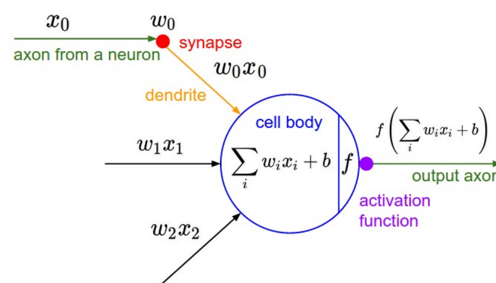

Abbildung 2.2.1: *Illustration des Grundbausteins eines neuronalen Netzes [LKJ16]: Ein Neuron. Zu erkennen sind mehrere eingehende Kanten, die jeweils aus einem Gewicht w_n und der Eingabe x_n bestehen. Die Produkte dieser beiden werden aufsummiert und anschließend zu einer Bias b hinzuaddiert. Dieser Wert wird der Aktivierungsfunktion f übergeben und dies bestimmt den Ausgabewert des Neurons.*

Primär werden Neuronen in den Faltungsschichten (vgl. Abschnitt 2.2.1) und in Fully-Connected-Layer (vgl. Abschnitt 2.2.3) eingesetzt. Ein Neuron berechnet wie in Abbildung 2.2.1 dargestellt seine Ausgabe abhängig von der Aktivierungsfunktion (vgl. Abschnitt 2.2.4) f, Gewichten w_i, Neigung (Bias) b und der Eingabe x_i mit der Formel [LKJ16]:

$$f(\sum_i w_i * x_i + b). \tag{2.2.1}$$

Eine Bündelung von mehreren Neuronen mit ggf. geteilten Gewichten wird Schicht genannt. Diese Neuronen bekommen die gleiche Eingabe, können aber abhängig von ihren Gewichten und Neigung unterschiedliche Ausgaben berechnen.

2.2.1 *Convolution-Layer*

Diese Schicht dient der Merkmalsextraktion und der Identifizierung relevanter Merkmale bezogen auf eine Problemdomäne und reiht sich damit passend in die Pipeline (vgl. Abbildung 2.1.1) ein. Wie in Abschnitt 2.2.6 beschrieben, können die Gewichte einer Schicht angelernt werden. Auf den Convoution-Layer bezogen bedeutet dies, dass Merkmalfilter angelernt werden. Ein Merkmalfilter bestimmt die Ausprägung an einer beliebigen Stelle der Eingabe, da die lokale Umgebung gewichtet zusammengefasst wird (Faltung). Die Merkmale sind demnach Positions-Invariant, da diese an jeder beliebigen Stelle gleich berechnet werden können. Dies wird sich in der Computer Vision und in CNNs zu Nutze gemacht, um mithilfe der Backpropagation (vgl. Abschnitt 2.2.6) automatisch relevante Merkmale anzulernen.
Die Idee stammt aus der Faltung (*convolution Operation*), die für einen aktuellen Zeitpunkt t, einer stetigen Messfunktion $x(t)$ und einer stetigen Gewichtungsfunktion $w(a)$, die Summe $s(t)$ für infinitesimal kleine Messpunkte (Integral) bildet [GBC16]. $w(a)$ bewertet dabei eine Messung, die a Zeiteinheiten in der Vergangenheit bzw.. für negative Werte in der Zukunft liegt.

$$s(t) = \int x(a)w(t-a)da \tag{2.2.2}$$

Da bei vielen Szenarien aber keine integrierbare, stetige Funktion sondern lediglich diskrete Messpunkte vorliegen, lässt sich 2.2.2 auf den diskreten Fall übertragen.

$$s(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a) \tag{2.2.3}$$

Diese Operation wird **Faltung** genannt und wird üblicherweise wie folgend mit einem Stern notiert [GBC16].

$$s(t) = (x * w)(t) \tag{2.2.4}$$

Die Faltung ist nun für den eindimensionalen Fall definiert, lässt sich jedoch für den zweidimensionalen Fall erweitern. Dazu betrachten wir zunächst die in einem Convolution-Layer (bzw. Faltungsschicht) auftretenden Parameter. Der Convolution-Layer untersucht ein lokales Umfeld für jede mögliche Position in der Eingabe. Die Eingabe ist dabei ein Volumen **I** und besteht anfänglich nur aus den Farb- bzw. Grauwerten des Anfragebildes. Die Dimension des Volumens setzt sich aus der Größe des Bildes in Höhe und Breite sowie Anzahl an Farbkanälen in der Tiefe zusammen. Ein 64x64 Bild mit RGB Farbkanälen würde dementsprechend ein Eingabevolumen $I(64 \times 64 \times 3)$ implizieren. Auf der Eingabe berechnet der Convolution-Layer ein Ausgabevolumen, welches abhängig von den Hyperparametern des Convolution-Layer ist. Hyperparamter sind Parameter, die sich im Lernprozess (vgl. Abschnitt 2.2.6) nicht ändern.
Der **Kernel** ist der Kern der Faltungsschicht und beschreibt einen Filter mit Gewichten w_n (vgl. Abbildung 2.2.1), die jedem Neuron der Schicht geteilt zur Verfügung stehen, da der Filter, der ein Merkmal beschreibt und dessen Ausprägung berechnet, unabhängig von der Position im Bild ist. Dieser lässt sich als Matrix mit Dimension $K \times K \times N$, *Kernel-Size* K und Eingabetiefe (beispielsweise 3 Farbkanäle) N darstellen.
Die **Anzahl der Filter** F bestimmt, wie viele verschiedene Kernel in der Faltungsschicht vorhanden sind. Sie bestimmt die Tiefe des Ausgabevolumens, da jeder Kernel die gesamte Eingabe auf eine Aktivierungsmatrix abbildet.
Die **Stride** bestimmt, wie viele Pixelkoordinaten bei der Berechnung der Ausgabe übersprungen werden. Bei einer Stride von 1, wird keine Pixelkoordinate übersprungen und die Berechnung (vgl. 2.2.7) wird an jeder möglichen Koordinate durchgeführt, um das Ausgabevolumen zu bestimmen.
Das **Padding** P bestimmt die Breite eines Randes um die Eingabe der Schicht und besteht beispielsweise aus konstanten Werten, den vorherigen und wiederholten Randwerten oder einer Spiegelung des Inneren auf den Rand [LKJ16].
Im Spezialfall von nur einem Farbkanal, lässt sich ausgehend von Formel 2.2.3, Eingabebild **I**, Kernel **K** und Messpunkt (Pixelkoordinate) (i, j) die 2D-Faltung definieren:

$$\mathbf{S}(i,j) = (\mathbf{I} * \mathbf{K})(i,j) = \sum_m \sum_n \mathbf{I}(m,n)\mathbf{K}(i-m, j-n) \tag{2.2.5}$$

Da die Operation kommutativ ist, lässt sich dies umformen [Bra00]:

$$\mathbf{S}(i,j) = (\mathbf{K} * \mathbf{I})(i,j) = \sum_m \sum_n \mathbf{I}(i-m, j-n)\mathbf{K}(m,n) \tag{2.2.6}$$

Dies lässt sich einfacher interpretieren, da für einen Punkt (i,j) die Untermatrix aus I mit Eckpunkten $(i-m, j-n)$ und (i,j) mit der gesamten Gewichtsmatrix K — Dimension $m \times n$ — elementweise multipliziert wird, was das Hadamard Produkt genannt wird [Mil07].
Formel 2.2.6 kann jetzt für einen Kernel mit Dimension $K \times K \times N$ und Eingabebild I mit Dimension $W \times H \times N$ erweitert werden:

$$\mathbf{S}(i,j) = (\mathbf{K} * \mathbf{I})(i,j) = \sum_k \sum_m \sum_n \mathbf{I}(i-m, j-n, k)\mathbf{K}(m,n,k) \tag{2.2.7}$$

Der einzige Unterschied ist, dass nun jeweils Kanäle (z. B. RGB Farbkanäle bei der Eingabe) aus dem Volumen des Bildes und der Gewichte genommen werden und anschließend, analog zu vorher, das Hadamard Produkt gebildet wird. Die Dimension des Ausgabevolumen hängt von den gewählten Hyperarametern *Kernel-Size* K, *Filteranzahl* F, *Padding* P, *Stride* S und der Dimension der Eingabe in Breite W und Höhe H ab. So finden sich die Parameter des Convolution-Layer innerhalb des Kernels, dessen Größe durch mehrere Hyperparamter bestimmt ist. Die Ausgabe hat die Dimension $W' \times H' \times F$ mit

$$W' = \frac{W-K+2P}{S+1} \text{ und } H' = \frac{H-K+2P}{S+1} \text{[LKJ16].} \tag{2.2.8}$$

Zusammenfassend lässt sich das Vorgehen für einen Streifen des Convolution-Layer wie in Abbildung 2.2.2 illustrieren. Der Kernel wird mit einer Stride von Eins über das gesamte Bild ohne Padding geschoben und berechnet nach Formel 2.2.7 die Ausgabe.

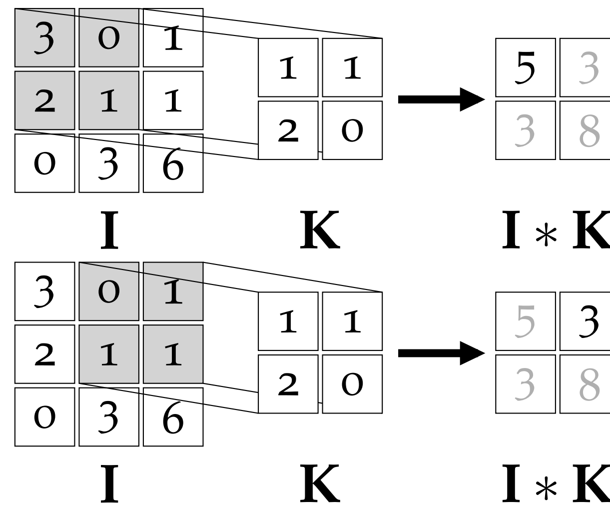

Abbildung 2.2.2: *Ein Convolution-Layer, der eine Eingabe* $\mathbf{I}(3 \times 3)$ *bekommt und Hyperparamter* $K = 2, F = 1, P = 0, S = 1$ *hat. Der Kernel der größe* $K \times K$ *wird mit Stride* S *über die Eingabe geschoben und jeweils das Hadamard Produkt berechnet. Zu sehen sind die ersten zwei Schritte und dementsprechend die Berechnung der oberen beiden Werte der Ausgabematrix der Faltung* $\mathbf{I} * \mathbf{K}$.

Ein Neuron im Ausgabevolumen ist also nach Formel 2.2.8 mit seinem lokalen Umfeld und in der kompletten Tiefe in der Eingabe verbunden. Das bedeutet für einen Filter existieren $K * K * N$ lernbare bzw.. variable Parameter. Da es F Filter gibt, besitzt ein Convolution-Layer demnach $K * K * N * F$ Parameter. Während des Trainings (vgl. Abschnitt 2.2.6) werden schrittweise die für die Problemdomäne relevantesten Merkmale durch die Filter identifiziert, woraufhin deren Ausprägungen berechnet werden können. Wie [SZ15] und [SLJ+15] zeigen, sind $K = 3$, $S = 1$, $P = 1$ sinnvolle Startwerte der Hyperparameter, doch wie [KSZQ19] darstellt, sind z. B. größere Kernel oder asymmetrische Kernel sinnvoll bei manchen Problemdomänen.

2.2.2 *Pooling-Layer*

Ein Pooling-Layer dient im allgemeinen der Informationsreduktion und der Vergrößerung der positionellen Invarianz [Nie15]. Dadurch können nachfolgende Convolution-Layer eine größere Fläche abdecken, ohne eine größere Kernel-Size zu haben, der Ressourcenaufwand wird hinsichtlich Speicherverwendung und CPU/GPU Zeit verringert und es können mit verschiedenen Techniken robust verschiedene Bildgrößen auf eine konstante Anzahl an Pooling-Ergebnissen abgebildet werden.

Ein Pooling-Layer hat analog zu einem Convolution-Layer ähnliche Hyperparameter: Eine räumliche Ausprägung von $m \times n$, Stride S und Padding P. Dem gegenüber besitzen Pooling-Layer aber keine lernbaren Parameter, sondern führen statt des Hadamard Produktes der Eingabe und der Gewichte, eine vorher festgelegte Funktion aus. So können die Werte in dem aktuellen Rechteck beispielsweise quadriert, aufsummiert und anschließend daraus die Wurzel gezogen (**L2-Norm**), der Durchschnitt aller Werte innerhalb des Rechtecks gebildet (**Avg**) oder aber das Maximum aller Werte zurückgegeben (**Max**) werden. Das Ausgabevolumen hat entsprechend der Hyperparamter die Dimension $W' \times H' \times N$ mit W', H' analog zu Formel 2.2.8 und N entsprechend der Tiefe der Eingabe, die hier unverändert bleibt [LKJ16].

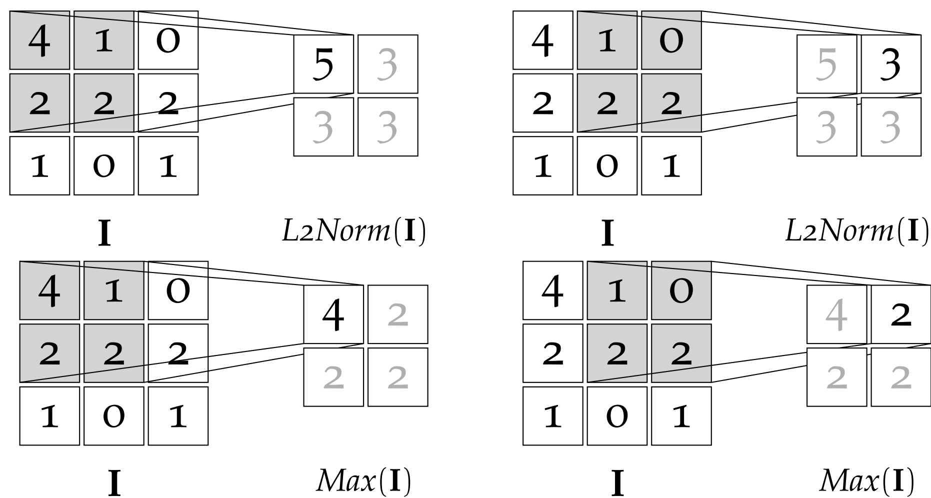

Abbildung 2.2.3: *Beispiele für eine vordefinierte Funktion eines Pooling-Layer.* ***Obere Zeile****: L2 Norm Pooling-Layer der alle Werte im Rechteck quadriert, summiert und daraus die Wurzel zieht und ausgibt.* ***Untere Zeile****: Max Pooling-Layer der das Maximum aller Werte im Rechteck ausgibt. Beide Pooling-Layer haben die Hyperparameter:* $m = n = 2, S = 1, P = 0$

Spatial Pyramid Pooling

Wie Formel 2.2.8 zeigt, hängt die Dimension des Ausgabevolumens von der Eingabe und den Hyperparametern von Convolution- bzw. Pooling-Layer ab. Wenn dies nicht erwünscht ist und die Schicht unabhängig von der Eingabegröße eine feste Ausgabegröße erzeugen soll, müssen Werte zusammengefasst oder inter- bzw. extrapoliert werden. Wir beschränken uns auf den Fall der Zusammenfassung. Das Spatial Pyramid Pooling (SPP) fasst immer x-tel Teile der Eingabe zusammen. Es unterteilt die Eingabe

$(W \times H \times N)$ in x-Teile in der Breite sowie in der Höhe, wodurch ein Raster entsteht worin jedes Rechteck eine Breite von $\frac{W}{x}$ und Höhe $\frac{H}{x}$ besitzt [HZRS14]. x ist ein Hyperparameter der Schicht und wird *Level* genannt. Alle Werte die darin enthalten sind, werden analog wie zuvor nach einer vorher festgelegten Funktion (L2-Norm, Avg, Max u. a.) zusammengefasst. Dies hat den Vorteil, dass die Ausgabe für jeden Streifen aus N, x^2 Werte berechnet, sodass ein Vektor der Länge $N \cdot x^2$ entsteht. Sollte $\frac{W}{x}$ bzw. $\frac{H}{x}$ nicht glatt Teilbar sein, um die räumliche Ausprägung $m \times n$ ganzzahlig zu bestimmen, muss an den Seiten entsprechend ein Padding von $(\mathit{Top}, \mathit{Bottom}, \mathit{Left}, \mathit{Right})$ mit

$$\begin{aligned} m &= \left\lceil \frac{W}{x} \right\rceil, & n &= \left\lceil \frac{H}{x} \right\rceil \\ \mathit{Top} &= \left\lceil \frac{nx - H}{2} \right\rceil, & \mathit{Bottom} &= \left\lfloor \frac{nx - H}{2} \right\rfloor \\ \mathit{Left} &= \left\lceil \frac{mx - W}{2} \right\rceil, & \mathit{Right} &= \left\lfloor \frac{mx - W}{2} \right\rfloor \end{aligned} \tag{2.2.9}$$

vorgenommen werden. Zu erkennen ist, dass die räumliche Ausprägung $m \times n$ abhängig von dem Hyperparameter Level ist und entsprechend des Levels aufgerundet wird. Das Padding wird danach auf den oberen und unteren bzw. linken und rechten Rand aufgeteilt, sodass die Summe garantiert $mx - W$ bzw. $nx - H$ entspricht und m bzw. n garantiert dieses mit einem Padding versehene Bild glatt teilt.
Üblicherweise werden mehrere Werte für x (Level) gewählt und diese in einen noch längeren Vektor zusammengefasst. Die Länge dieses Vektors kann in Abhängigkeit der Tiefe N der Eingabe und der Menge an Level X mit

$$\mathit{len}_{\mathrm{spp}}(N, X) = \sum_{x \in X} N x^2 \tag{2.2.10}$$

berechnet werden.

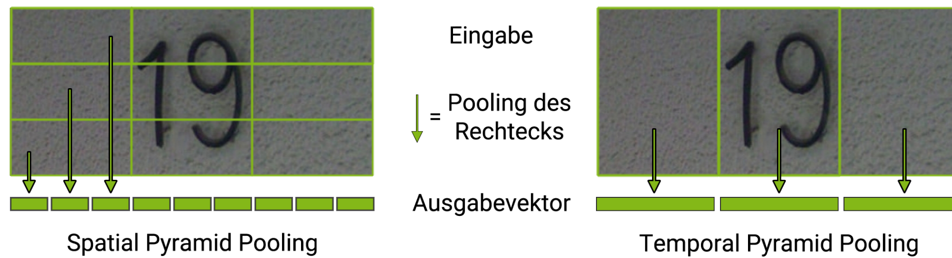

Abbildung 2.2.4: *Vorgehen von SPP und TPP mit einem Level $x = 3$ im direkten Vergleich. SPP teilt sowohl in der Länge als auch in der Breite in x Teile ein, wobei TPP dies nur in der Breite tut. Beide erzeugen einen Ausgabevektor konstanter Länge abhängig vom gewähltem Level. Das Bild der Hausnummer stammt aus dem SVHN Datensatz [NWC+11].*

Temporal Pyramid Pooling

Eine anderer Ansatz geht von der Annahme aus, dass es sich um eine Sequenz von Zeichen handelt und daher in der Breite der Eingabe mehr Informationen zu erwarten sind, als in der Höhe [SF17]. Das Temporal Pyramid Pooling (TPP) ist eine Abwandlung des Spatial Pyramid Pooling und berechnet m, n bzw. das Padding $(Top, Bottom, Left, Right)$ wie folgend:

$$\begin{aligned} m &= \left\lceil \frac{W}{x} \right\rceil, & n &= H \\ Top &= 0, & Bottom &= 0 \\ Left &= \left\lceil \frac{mx - W}{2} \right\rceil, & Right &= \left\lfloor \frac{mx - W}{2} \right\rfloor \end{aligned} \tag{2.2.11}$$

Zu erkennen ist, dass keine Einteilung und dadurch kein Padding in der Höhe der Eingabe vorgenommen wird. Jedes Rechteck bildet also einen Streifen mit Breite m und Höhe H. Auf die darin enthaltenden Werte wird erneut die zuvor festgelegte Funktion (L2-Norm, Avg, Max u. a.) angewandt. Jeder Kanal aus der Eingabe mit Tiefe N bildet daher auf x Werte ab, wodurch ein Vektor der Länge $N \cdot x$ entsteht. Auch hier werden üblicherweise mehrere verschiedene Level angewandt, sodass ein Vektor mit Länge

$$len_{tpp}(N, X) = \sum_{x \in X} Nx \tag{2.2.12}$$

entsteht und ausgegeben werden kann.

2.2.3 *Fully-Connected-Layer*

Um auf Grundlage der Merkmalsausprägungen die Klassifikation vorzunehmen, sollen alle zuvor bestimmten Informationen genutzt werden. Üblicherweise werden Fully-Connected-Layer (FC-Layer) in einem Multilayer-Percepton (MLP) eingesetzt [Hay98]. Meist werden mehrere Fully-Connected-Layer hintereinander eingesetzt um eine höhere Kombinationsmöglichkeit von Werten zu erreichen. Der letzte Fully-Connected-Layer berechnet schließlich die *Scores* aller Klassen. Diese letzte Schicht wird als Output-Layer, die erste als Input-Layer und alle Schichten innerhalb des Netzwerks werden als Hidden-Layer bezeichnet. Ein MLP bildet eine Untergruppe von neuronalen Netzen und wird innerhalb von CNNs als Klassifikator ganz am Ende der Architektur benutzt [LKJ16].
Wie in Abschnitt 2.2.1 besteht diese Schicht aus einer Menge von Neuronen. Diese sind jedoch nicht nur mit einem lokalen Teil der Eingabe verbunden und teilen sich die Gewichte, sondern sind mit allen Eingabewerten verbunden und teilen sich keine Gewichte [GBC16]. Gibt beispielsweise ein vorheriger Convolution-Layer ein Volumen von $4 \times 4 \times 92$ aus, so ist ein Neuron in einem danach folgenden Fully-Connected-Layer mit allen 1472 Neuronen des Volumens verbunden und berechnet nach Formel 2.2.1 seine Ausgabe.

2.2.4 *Aktivierungsfunktion*

Bisher könnten die vorgestellten Schichten durch eine verkettete Matrixmultiplikation der jeweiligen Gewichtsmatrix die Ausgabe berechnen. Für 2 Gewichtsmatrizen (also 2 Schichten) $\mathbf{W}_1(m \times n), \mathbf{W}_2(l \times m)$ und ein Eingabevektor $\mathbf{E}(n \times 1)$ gilt

$$
\begin{aligned}
\mathbf{W}_1 \cdot \mathbf{E} = \mathbf{A}_1, \mathbf{W}_2 \cdot \mathbf{A}_1 &= \mathbf{A}_2 \\
\Rightarrow \mathbf{W}_2 \cdot \mathbf{W}_1 \cdot \mathbf{E} &= \mathbf{A}_2 \\
\Rightarrow \mathbf{W}_2 \cdot \mathbf{W}_1 &= \mathbf{W}_z \\
\Rightarrow \mathbf{W}_z \cdot \mathbf{E} &= \mathbf{A}_2 \\
\Rightarrow \mathbf{W}_2 \cdot \mathbf{W}_1 \cdot \mathbf{E} &= \mathbf{W}_z \cdot \mathbf{E} \,.
\end{aligned}
$$

Zu erkennen ist, dass $\mathbf{W}_2$ und $\mathbf{W}_1$ zu $\mathbf{W}_z(l \times n)$ zusammengefasst werden können, ohne dass sich die Ausgabe $\mathbf{A}_2(l \times 1)$ ändert. Dadurch würde diese Gewichtsmatrix eine lineare Funktion zwischen Ein- und Ausgabe beschreiben [GBC16]. Aktivie-

rungsfunktionen führen bewusst eine nicht-Linearität ein, um beliebig komplexe Mustererkennungsfunktionen zu approximieren [Nie15]. Neuronale Netze mit Aktivierungsfunktionen sind also universelle Approximatoren für beliebig komplexe Funktionen [Hor91].
Aktivierungsfunktionen bilden eine Eingabe auf eine Ausgabe mit einer vorher definierten Funktion ab. Mögliche Aktivierungsfunktionen sind z. B.:

$$\sigma(x) = \frac{1}{1+e^{-x}}, \qquad \mathit{tanh}(x) = 2\sigma(2x) - 1, \qquad \mathit{ReLU}(x) = \mathit{max}(0, x) \tag{2.2.13}$$

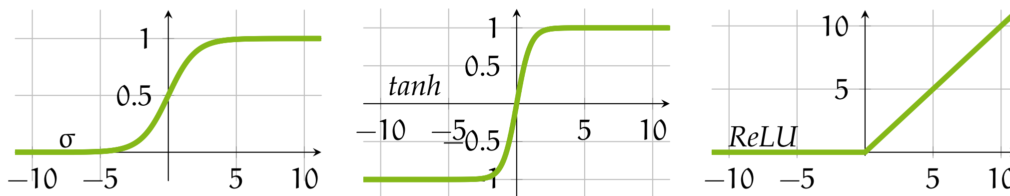

Abbildung 2.2.5: *Beispiele für Aktivierungsfunktionen. (v.l) Sigmoid, tanh und ReLU.*

Die Sigmoid Funktion $\sigma(x)$ bildet alle möglichen Eingaben auf das Intervall $[0, 1]$ ab. Zu erkennen ist, das für zwei große Werte von x beide auf Werte nahe 1 abbilden und so der Gradient zwischen diesen nahe 0 liegt. Dies ist meist unerwünscht, da der Gradient für das anlernen benötigt wird (vgl. Abschnitt 2.2.6). Die *tanh* Funktion hat den Vorteil zentriert zum Ursprung zu sein, aber hat analog zur Sigmoid Funktion für große Werte einen sehr kleinen Gradienten [LKJ16]. Dies behebt die *ReLU* Funktion, indem es die Eingabe ausgibt, wenn diese größer als 0 ist und anderenfalls immer 0 ausgibt. Zudem wird ReLU eine bessere Anlernrate zugeschrieben [KSH12].
Eine beliebte Aktivierungsfunktion für die letzte Schicht des Netzwerks ist die Softmax-Funktion [LKJ16] [Nie15]. Sie benötigt alle Ausgabewerte der Schicht, z. B. in Form eines Vektors $\vec{z}$, und normalisiert diese auf Wahrscheinlichkeiten, sodass die Summe aller normalisierten Ausgabewerte 1 ergibt. Die Softmax-Funktion ist dann für den i-ten Ausgabewert wie folgend definiert:

$$\mathit{softmax}(\vec{z})_i = \frac{e^{z_i}}{\sum_j e^{z_j}} \tag{2.2.14}$$

Der Vorteil diese Aktivierungsfunktion bei der letzten Schicht zu verwenden ist, dass die schlussendliche Ausgabe des Klassifikators als Wahrscheinlichkeitsverteilung bezogen auf die Klassen interpretiert werden kann, da die Summe aller Werte der Softmax Ausgabe 1 ergibt. Der Klassifikator schätzt also aufgrund von den angelernten

Merkmalfiltern und dessen Ausprägungen eine Klasse mit einer zugehörigen „Wahrscheinlichkeit“. Da es jedoch nur eine Art der Normalisierung der Daten ist, ist es keine wirkliche Wahrscheinlichkeit.

2.2.5 *Ausgabe und Loss*

Nun können die einzelnen Komponenten des Netzwerks zusammengefügt werden, wodurch eine vollständige Architektur die Ausgabe berechnen kann. Die Grundidee besteht darin, Convolution-Layer zur Merkmalsextraktion zu verwenden, diese Informationen in Fully-Connected-Layer zu nutzen und die Klasse abzuschätzen. Zudem können Pooling-Layer eingesetzt werden, um u. a. die Informationsmenge zu reduzieren (vgl. Abschnitt 2.2.2). Dabei können, wie in Abschnitt 2.2.4 beschrieben, mehrere Convolution-Layer hintereinander eingesetzt werden, um eine höhere nicht-Linearität zu erreichen.
Sei nun im folgenden ein Convolution-Layer als *CONV*, Fully-Connected-Layer als *FC*, ein Pooling-Layer als *POOL* und eine Aktivierungsfunktion als *ACT* gekennzeichnet. Ausgehend von obiger Überlegung könnte ein Netzwerk zur Merkmalsextraktion und Klassenabschätzung die grundlegende Struktur von

$$CONV \rightarrow ACT \rightarrow CONV \rightarrow ACT \rightarrow FC \rightarrow ACT \rightarrow FC \tag{2.2.15}$$

oder

$$CONV \rightarrow ACT \rightarrow POOL \rightarrow CONV \rightarrow ACT \rightarrow POOL \rightarrow FC \rightarrow ACT \rightarrow FC \tag{2.2.16}$$

haben. Erstere (Formel 2.2.15) enthält 2 Convolution-Layer mit dazugehöriger Aktivierungsfunktion. Daraufhin folgt ein Fully-Connected-Layer inklusive Aktivierungsfunktion und der Output-Layer. Letztere (Formel 2.2.16) fügt Pooling-Layer hinzu, um u. a. die Parameteranzahl und damit den Ressourcenaufwand zu verringern. Diese Grundstruktur kann verändert bzw. mithilfe verschiedener Techniken (vgl. Abschnitt 2.2.2, 2.2.7) ergänzt werden.
Nach Festlegung der Architektur kann der *Forward-Pass* berechnet werden. Dies bedeutet, dass eine Eingabe die definierten Berechnungen aller Schichten durchläuft, wobei die Eingabe einer Schicht die Ausgabe der vorherigen ist. Die Ausgabe der letzten Schicht (Output-Layer) wird *Ausgabe des Netzwerks* genannt.
Im *supervised learning* gibt es zu allen Partitionen des Datensatzes Annotationen welche die (meist) richtige Klasse eines jeden Elements des Datensatzes beschreibt. Da die Annotation jedoch meist händisch erfolgt, kann auch dort ein Fehler aufkommen, oder zwei verschiedene Personen würden ein Bild des Datensatzes verschieden annotieren.

Diese Annotation ist das *Soll-Ergebnis* welches das Netzwerk erreichen soll. Zu Beginn kann das Netzwerk das *Ist-Ergebnis* (die Schätzung des Netzwerks) nur zufällig bestimmen, da ohne vorherige Kenntnisse noch keinerlei Merkmale in den Filtern der Convolution-Layer angelernt sind und dementsprechend keine davon abhängige Schätzung der Klasse vorgenommen werden kann. Die Schätzung der Klasse drückt ein k-Dimensionalen Vektor (k Klassen) aus. Sei nun das Netzwerk mit der Funktion

$$y(x) = (s_1, s_2, ..., s_k) \tag{2.2.17}$$

definiert, welches eine Eingabe x auf die jeweiligen Schätzungen s_n für alle k Klassen abbildet. Analog dazu sei die Annotation auch als Funktion

$$\hat{y}(x) = (g_1, g_2, ..., g_k) \tag{2.2.18}$$

gegeben, wobei $g_n = 1$ für die korrekte Klasse und $g_n = 0$ sonst gilt.
Der *Loss*, der manchmal auch *Cost* oder *Objective* genannt mini [Nie15], bestimmt eine Art Abstand des Ist- und des Soll-Ergebnisses sodass im Rahmen der Backpropagation und Optimierung (Abschnitt 2.2.6) die Gewichte immer in Richtung der zu approximierenden Mustererkennungsfunktion verbessert werden können [LKJ16]. Der Loss wird durch eine vorher definierte Funktion, der Ausgabe des Netzwerks $y(x)$ und der Annotation $\hat{y}(x)$ bestimmt. Ziel ist es die Gewichte und Bias so zu bestimmen, dass der Loss minimiert wird. Für eine Menge an Eingaben ist der durchschnittliche Loss, gegeben einer Loss Funktion für ein Element L_i und die Anzahl an Elementen N, wie folgend definiert [LKJ16]:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i \tag{2.2.19}$$

Es gibt viele Varianten wie L_i zu wählen ist, einige Beispiele sind

$$MSE = ||y - \hat{y}||^2, CE = -\log\left(\frac{e^{y_{id(\hat{y})}}}{\sum_j e^{y_j}}\right) \tag{2.2.20}$$

wobei MSE für *Mean Squared Error*, und CE für *Cross Entropy* steht [LKJ16] [Nie15]. Man beachte, dass hier y und $\hat{y}$ in Kurzform verwendet werden und diese jeweils noch die Eingabe x übergeben bekommen und y abhängig von den momentanen Gewichten ist. $||\vec{a}||$ beschreibt die Länge eines Vektors und $id(\hat{y}(x))$ gibt den Index der korrekten Klasse bei Eingabe von x zurück.

2.2.6 *Backpropagation und Optimierung*

Formel 2.2.17 berechnet wie beschrieben die Ausgabe des Netzwerks. Dabei ist $y(x)$ von den Gewichten abhängig und im einfachsten Fall von nur einer Schicht ohne Aktivierungsfunktion ließe sich die Ausgabe durch

$$y(x, \mathbf{W}) = \mathbf{W} \cdot x \tag{2.2.21}$$

berechnen. Das Ziel ist es, *W* so zu wählen, dass der Loss (vgl. Abschnitt 2.2.5) minimal ist. Dazu werden die partiellen Ableitungen von L_i mit Respekt zu den einzelnen Variablen der Gewichtsmatrix gebildet. Dadurch ergibt sich ein Gradientenvektor (bzw. Matrix) der benutzt wird um die derzeitigen Gewichte zu optimieren. Wegen der Minimierung des Losses, wird der Gradient invertiert und mit einem Koeffzienten auf die derzeitigen Gewichte hinzugerechnet. Dieser Koeffizient, also wie weit in Richtung des Minimums korrigiert wird, definiert die *Lernrate*. Dieses Vorgehen wird im Falle von nur einem Punkt *Stochastic Gradient Descent* (SGD) und im Falle von einem Gradient über eine Menge von Punkten *Mini-Batch Gradient Descent* (MGD) genannt. Es ist unter Umständen effizienter den Gradient einer Batch zu berechnen als für jedes Element der Batch einzeln [LKJ16]. Im weiteren Verlauf der Arbeit wird auch der Mini-Batch Gradient Descent mit SGD gekennzeichnet. Formal wird der partielle Gradientenvektor

$$\nabla L = \left(\frac{\partial L}{\partial \mathbf{W}_1}, \frac{\partial L}{\partial \mathbf{W}_2}, \ldots, \frac{\partial L}{\partial \mathbf{W}_n} \right) \tag{2.2.22}$$

berechnet und daraufhin die Gewichte $\mathbf{W}$ entsprechend der Überlegung und der Lernrate λ zu $\mathbf{W}'$ verbessert [Nie15]:

$$\mathbf{W}' = \mathbf{W} - \lambda \nabla L \tag{2.2.23}$$

Im Rahmen der *Backpropagation* wird der Gradientenvektor hergeleitet, indem für jedes Neuron ein Error δ_j berechnet und davon ausgehend auf den Error der Schicht geschlossen wird [Nie15]. Mithilfe dieses Vektors kann der Fehler der vorherigen Schicht berechnet werden. Daher wird mit der letzten Schicht gestartet und schrittweise rückwärts die Error-Werte der vorherigen Schichten berechnet (Backpropagation). Mithilfe dieser Werte können die Gradienten jedes einzelnen Gewichts berechnet werden [RHW86]. Der wiederholte Vorgang der Backpropagation und der Verbesserung von $\mathbf{W}$ wird Lernprozess genannt. Eine Epoche stellt dabei die wiederholte Anwendung des SGD mithilfe der gesamten Trainingspartition (vgl. 4.2) dar. Es wird versucht, die zugrundeliegende nicht bekannte Mustererkennungsfunktion immer besser zu approximieren und diese mithilfe einer Testpartition zu evaluieren [LKJ16].

2.2.7 *Regularisierung und Overfitting*

Es existieren verschiedene Methoden um das Anlernen, die Performance und die Stabilität eines Neuronalen Netzwerks zu erhöhen. Eine Methode ist die *Batch normalization* (BN), welche in [IS15] vorgestellt wurde. Ziel darin war es, die *interne Kovariante Verschiebung* — die Änderung der Verteilung der einzelnen Eingaben jeder Schicht — zu verringern, welche einen Effekt auf die Lernrate des Netzwerks hat, jedoch wurde in [STIM18] herausgefunden, dass Batch normalization den Loss glättet. BN verringert die Größe mit der sich die internen Gewichte und Aktivierungen ändern, wodurch die Dauer des Lernprozesses durchschnittlich sinkt [IS15].
Dropout ist eine Technik um *Overfitting* zu vermeiden [SHK+14]. Overfitting bedeutet, dass das Netzwerk für bekannte Daten sehr gut Funktioniert und diese sehr genau entsprechend der Annotationen klassifiziert, jedoch bei unbekannten Daten deutlich schlechter abschneidet und dementsprechend schlecht generalisieren kann. Dies kann mit dem auswendig Lernen des Datensatzes verglichen werden: Für bekannte Daten arbeitet das Netzwerk sehr genau, kann aber nicht gut generalisieren und hat einen höheren Fehler bei unbekannten Daten.

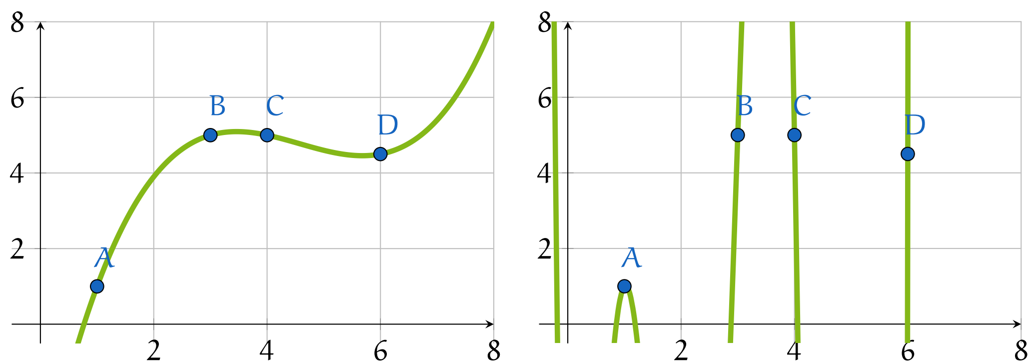

Abbildung 2.2.6: *Illustration von Overfitting im 2-Dimensionalen Raum.* ***Links****: Die Funktion berechnet sinnvoll die unbekannten Punkte auf Grundlage der gegeben Trainingspunkte (kein Overfitting).* ***Rechts****: Die Funktion berechnet zwar auf den Trainingspunkten korrekt die gewünschten Ausgaben, aber nimmt extreme und ungewünschte Werte für unbekannte Eingaben an.*

Bei Verwendung von Dropout wird zufällig in jeder Schicht ein fester Anteil der Neuronen inklusive ihrer Verbindungen zu anderen Neuronen deaktiviert. Das soll dazu führen, dass sich einzelne Neuronen nicht zu stark an einander binden. Binden bedeutet in diesem Kontext, dass ein Neuron sich im Extremfall nur auf die Werte

eines Eingabeneuron verlässt und alle anderen nicht bzw. kaum in dessen Berechnung des Ausgabewertes mit einfließt. Dies soll die Fähigkeit der Generalisierung des Netzwerks steigern [SHK+14], da ein simples auswendig Lernen nicht mehr möglich ist. Außerdem führt es dazu, dass ein Netzwerk trotz gleicher Eingabe verschiedene Ausgaben berechnet, da in jedem Durchlauf verschiedene Neuronen deaktiviert werden.

3 VERWANDTE ARBEITEN

Das Gebiet der Klassifizierung und Erkennung in der Bilddomäne ist ein andauerndes Forschungsgebiet der Informatik. Wie [DK16] darstellt, sind Convolutional Neural Networks die vorherrschenden Modelle in der Mustererkennung. Die Anfänge sind in u. a. dem LeNet-5 [LBBH98] zu finden. Hier wurde eine, mit Heute verglichen, simple Architektur mit 3 Convolution-Layer (vgl. Abschnitt 2.2.1) und rund 60 Tausend Parametern eingesetzt, welches aufgrund von leistungsschwächeren GPUs binnen Tagen statt Minuten angelernt wurde. Allein dies verdeutlicht den Sprung in Ressourcen die zum Anlernen heutzutage zur Verfügung stehen. Die größeren Mengen an Ressourcen können genutzt werden, um mehr anlernbare Parameter in einem Netzwerk zu verwenden. Zunächst war es zielführend tiefere Netzwerke mit mehr Hidden-Layer verschiedener Arten zu verwenden. Dabei war das Ziel komplexe Funktionen einfacher anzulernen [LPW+17] und größere Eingaben verarbeiten zu können. Da Faltungsschichten auch ein Volumen verkleinern können (vgl. Abschnitt 2.2.1), bedeuten tiefere Netzwerke meist kleinere Volumen, wodurch die Paramteranzahl in einem Fully-Connected-Layer sinkt. Beispielsweise verwendet das AlexNet [KSH12] 62 Millionen Parametern und musste zur Beschleunigung des Lernprozesses auf mehrere Grafikkarten parallelisiert werden. Da es mit größeren Bildern ($227 \times 277 \times 3$ statt $30 \times 30 \times 1$) und einer tieferen Architektur arbeitet, kommt dieser Sprung in der Parameterzahl zustande. Zudem wurde zum Trainieren des AlexNet auch Augmentierungsmethoden (vgl. Abschnitt 3.3) verwendet, um den Trainingsdatensatz zu vergrößern und die Fehlerrate zu senken. Den Trend, immer tiefere Netzwerke zu verwenden, setzt das VGG Net [SZ15] fort und benutzt 19 Convolution-Layer innerhalb der Architektur. Ein offensichtlicher Nachteil ist die dadurch gesteigerte Länge des Lernprozesses des Netzwerks. Dieser hat trotz einer im Vergleich zum AlexNet deutlich stärken GPU zwei bis drei Wochen gedauert. In [HZRS16] wurden tiefere Netze weiter untersucht und Möglichkeiten vorgestellt, wie tiefere Netze besser angelernt werden können. Es stellte sich heraus, dass die simple Erhöhung der Ebenenanzahl ab einen gewissen Grad kontraproduktiv hinsichtlich der Fehlerraten des Netzwerks innerhalb der gleichen Problemdomäne ist. Eine Lösung wurde u. a. in [HZRS16] oder [SIVA17] in Form des Residual Learning vorgestellt. Die Grundidee dabei ist, zu der normalen Ausgabe mehrerer sequentieller Schichten, die Eingabe hinzu zu addieren. Wenn $f(x)$ die Ausgabe mehrerer Schichten, bei Eingabe x, beschreibt und y die Ausgabe des Residual

Learning beschreibt, dann ist $y = f(x) + x$ [HZRS16]. Dadurch konnten Netzwerke mit über 2000 Schichten angelernt werden und bessere Erkennungsraten erreichen, als Netzwerke ohne diese Technik [HZRS16]. Weitere Methoden untersuchen zum Beispiel, wie eine variable Anzahl von Objekten an verschiedenen Positionen in der Eingabe erkannt werden können [Gir15]. Andere Techniken, wie die im weiteren Verlauf vorgestellten Spatial und Diffeomorphic Transformer Networks, versuchen die Eingabedaten oder Merkmalsausprägungen zu transformieren, um hinsichtlich der Erkennungsrate bessere Ergebnisse zu erzielen.

3.1 SPATIAL TRANSFORMER NETWORKS

Wie [JSZK15] darstellt, verschlechtert sich die Erkennungsrate, wenn verschiedene Transformation in der Eingabe enthalten sind oder größere nicht-klassenrelevate Varianzen im Datensatz auftreten, da z. B. die zu erkennenden Ziffern nicht immer zentriert sind. Das Netzwerk soll jedoch zu einem gewissen Grad invariant zu dieser Varianz sein. Im Vergleich zwischen zwei Fully-Connected-Networks schneidet ein Netzwerk inklusive eines Moduls zum Ausgleich der Transformationen bis zu 3.4% (5.2% zu 0.8%) besser ab. Dieses Modul ist das Spatial Transformer Network (STN), welches in eine bestehende Architektur eingesetzt werden kann und dort in gewisser Weise wie ein Pooling-Layer arbeitet, da es auch die Eingabe Downsamplen (vgl. Abschnitt 2.2.2) kann, aber nicht muss. Das Modul bildet den Kern der Arbeit und es wird untersucht welchen Effekt ein STN auf verschiedene Datensätze mit verschiedenen Vorverarbeitungs- und Augmentierungsmethoden hat.
Ein STN besteht im allgemeinen aus drei Komponenten: Ein *Localisation Network*, welches die Parameter einer Transformation bestimmen soll, eine *Grid Generator* wie in Formel 3.1.1, der die Parameter nutzt und ein Gitter erzeugt und ein *Sampler* wie in Formel 3.1.2, der auf Grundlage des Gitters die Ausgabewerte bestimmt [JSZK15]. Das Vorgehen ist in Abbildung 3.1.1 dargestellt.

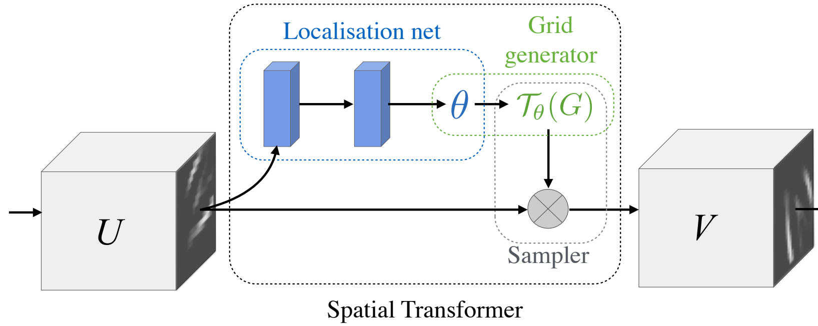

Abbildung 3.1.1: *Die Architektur eines STN. Die Eingabe* **U** *wird im Localisation Network genutzt um* θ *zu bestimmen. Ein regelmäßiges Gitter* G *auf Grundlage von* **V** *wird zu einem Sampling-Gitter* $\mathcal{T}_\theta(G)$ *nach Formel 3.1.1 transformiert. Der Sampler nutzt das transformierte Gitter und* **U***, um* **V** *zu bestimmen (vgl. Formel 3.1.2). Die Abbildung stammt aus [JSZK15].*

Die Basis bildet die affine Transformation im 2-Dimensionalen Raum. Sie kann bezogen auf einen Gitterpunkt G_i mit

$$\begin{pmatrix} x_i^s \\ y_i^s \end{pmatrix} = \mathcal{T}_\theta(G_i) = \begin{bmatrix} \theta_{11} & \theta_{12} & \theta_{13} \\ \theta_{21} & \theta_{22} & \theta_{23} \end{bmatrix} \begin{pmatrix} x_i^t \\ y_i^t \\ 1 \end{pmatrix} \tag{3.1.1}$$

definiert werden [JSZK15]. Dabei sind θ_{ij} die Parameter der Transformation, (x_i^t, y_i^t) die i-ten Koordinaten, die transformiert werden sollen und (x_i^s, y_i^s) die transformierten Koordinaten bzw. die Ausgabe. Alle Koordinaten sind dabei im Intervall $[-1, 1]$ und repräsentieren so, unabhängig von der exakten Pixelanzahl, alle Punkte innerhalb der Ein- bzw. Ausgabe. $\mathcal{T}_\theta(G_i)$ bezeichnet dabei die Transformation mit Parametern θ und Eingabekoordinate G_i aus einem Gitter G. G ist hier immer ein regelmäßiges Gitter bezogen auf die Dimension der Ausgabe, wie z. B. in Abbildung 3.1.2 (a) zu erkennen. Ein simples Beispiel wäre die Translation um t_x, t_y und Rotation um einen Winkel α eines gegeben Gitters. Dies würde

$$\theta = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & t_x \\ \sin(\alpha) & \cos(\alpha) & t_y \end{bmatrix}$$

realisieren.

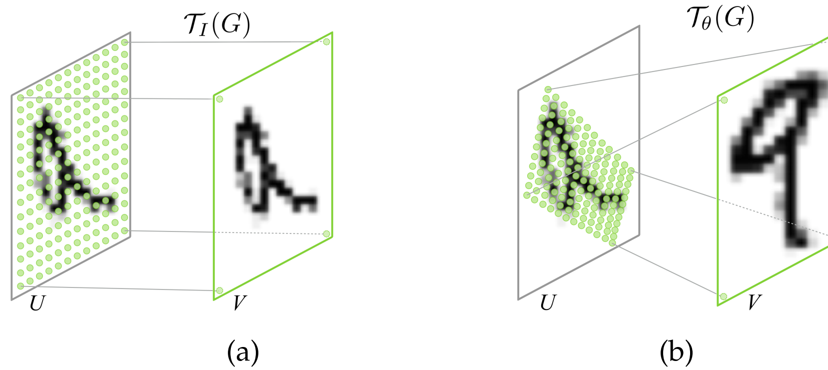

Abbildung 3.1.2: *Zwei weitere Beispiele einer affinen Transformation nach Formel 3.1.1 in der ein Grid* $T_\theta(G)$ $\mathbf{V}$ *auf* $\mathbf{U}$ *abbildet.* ***a****:* θ *ist die Identitätstransformation.* ***b****:* θ *ist eine affine Transformation mit mehreren Transformationstypen. Die Abbildung stammt aus [JSZK15]*

Falls die Ausgabe (x_i^s, y_i^s) normiert auf die Dimension der Eingabe $(m \times n)$ Gleitkommazahlen ausgibt, muss entschieden werden, welcher Quellpixel an die jeweilig korrespondierende Koordinate der Eingabe (x_i^t, y_i^t) *gesamplet* wird. Eine einfache Möglichkeit ist, den nächstliegenden Pixel zu nehmen. Für eine allgemeine Sampling Funktion k mit Parametern $\Phi_{x,y}$ gilt dann für den i-ten Wert des c-ten Kanals der Ausgabe $\mathbf{V}$ [JSZK15]:

$$\mathbf{V}_i^c = \sum_n^H \sum_m^W \mathbf{U}_{nm}^c k(x_i^s - m; \Phi_x) k(y_i^s - n; \Phi_y) \;\; \forall i \in [1 \dots H'W'] \;\; \forall c \in [1 \dots C] \quad (3.1.2)$$

Hierbei kann jeder Pixel der Eingabe $\mathbf{U}_{nm}^c$ in die Berechnung für die Ausgabe $\mathbf{V}_i^c$ mit eingehen. Die Ausgabe hat dabei die Dimension $W' \times H'$. Soll nun der nächstliegende Eingabewert kopiert werden, wird die Kronecker Delta Funktion verwendet, um zu prüfen ob die derzeitige Koordinate $\mathbf{U}_{nm}^c$ gleich der gerundeten Quellkoordinate (x_i^s, y_i^s) ist. (x_i^s, y_i^s) ist hierbei auf $[0, m]$ bzw. $[0, n]$ normiert. Die Kronecker Delta Funktion gibt 1 aus, wenn $\lfloor x_i^s + 0.5 \rfloor - m = 0$ gilt, welches nur der Fall ist, wenn $\lfloor x_i^s + 0.5 \rfloor = m$. Ansonsten ist die Funktion 0. Mithilfe der Kronecker Delta Funktion und Formel 3.1.2 lässt sich die Ausgabe wie folgt berechnen [JSZK15]:

$$\mathbf{V}_i^c = \sum_n^H \sum_m^W \mathbf{U}_{nm}^c \delta(\lfloor x_i^s + 0.5 \rfloor - m) \delta(\lfloor y_i^s + 0.5 \rfloor - n) \quad (3.1.3)$$

Alternativ wird die bilineare Interpolation mithilfe von

$$\mathbf{V}_i^c = \sum_n^H \sum_m^W \mathbf{U}_{nm}^c \max(0, 1 - |x_i^s - m|) \max(0, 1 - |y_i^s - n|) \tag{3.1.4}$$

umgesetzt [JSZK15]. Es können also beliebige Samplingfunktionen benutzt werden, solange sich die Gradienten bzw. Sub-Gradienten berechnen lassen. Analog zu Abschnitt 2.2.6 verwendet die Backpropagation diese, um die besten Parameter θ der Transformation (vgl. Formel 3.1.1) zu approximieren. Die Gradienten für bilineare Interpolation (Formel 3.1.4) lassen sich durch

$$\frac{\partial \mathbf{V}_i^c}{\partial \mathbf{U}_{nm}^c} = \sum_n^H \sum_m^W \max(0, 1 - |x_i^s - m|) \max(0, 1 - |y_i^s - n|) \tag{3.1.5}$$

$$\frac{\partial \mathbf{V}_i^c}{\partial x_i^s} = \sum_n^H \sum_m^W \mathbf{U}_{nm}^c \max(0, 1 - |y_i^s - n|) \begin{cases} 0 & \text{if } |m - x_i^s| \geqslant 1 \\ 1 & \text{if } m \geqslant x_i^s \\ -1 & \text{if } m < x_i^s \end{cases} \tag{3.1.6}$$

und

$$\frac{\partial \mathbf{V}_i^c}{\partial y_i^s} = \sum_n^H \sum_m^W \mathbf{U}_{nm}^c \max(0, 1 - |x_i^s - m|) \begin{cases} 0 & \text{if } |n - y_i^s| \geqslant 1 \\ 1 & \text{if } n \geqslant y_i^s \\ -1 & \text{if } n < y_i^s \end{cases} \tag{3.1.7}$$

berechnen [JSZK15]. Eine simple Netzwerkarchitektur, wie in Abschnitt 2.2.5, kann nun zu Beginn als *Localisation Network* (vgl. Abbildung 3.1.1) verwendet werden, um θ zu bestimmen. Durch die einfache Bestimmung des Gradienten kann das Netzwerk ressourcenarm (im Vergleich zum numerischen Annähern) optimiert werden. Des weiteren kann dieses Netzwerk als Modul in bestehende Architekturen einfach integriert werden, da es im Rahmen der Backpropagation (vgl. Abschnitt 2.2.6) durch die Gradienten mit angelernt werden kann und so kein weiterer Aufwand in der Implementierung entsteht. Der Effekt eines STN ist in Abbildung 1.1.2 dargestellt.

3.2 DIFFEOMORPHE TRANSFORMATIONEN

Ein Nachteil von STNs ist, dass bei einer zu hohen Lernrate die Parameterbestimmung irreversibel in eine falsche Richtung gelenkt werden kann und dadurch sehr starke Verzerrungen in die Ausgabe einfließen. Dies ist in Abbildung 3.2.1 veranschaulicht.

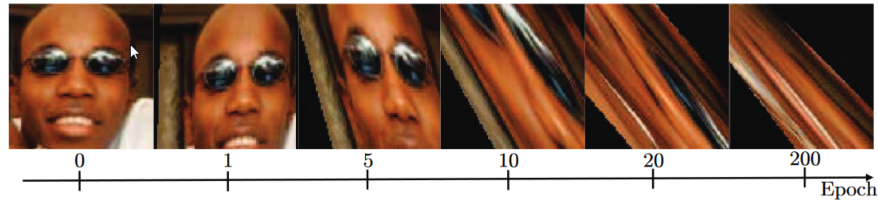

Abbildung 3.2.1: *Ein Nachteil von STNs: Bei einem zu großen falschen Optimierungsschritt kann die Transformation über die Lerndauer nicht zu einem lokalem Minimum hin optimiert werden [DFH18].*

Aus diesem Grund kann der Gradient der inversen Transformation von Vorteil sein. Er kann einen schlechten Optimierungsschritt rückgängig machen, damit eine Verzerrung wie in Abbildung 3.2.1 nicht zustande kommt [DFH18]. Eine stetige differenzierbare Abbildung (normal), dessen Umkehrabbildung auch stetig und differenzierbar ist (invers), wird diffeomorph genannt [BHWM12]. Die Basis eines *Diffeomorphic Transformer Network* (DTN) bildet die CPA-B Transformation (*Continuous Piecewise-Affine Based*), welche ein Geschwindigkeitsfeld wie in Abbildung 3.2.2 definiert.

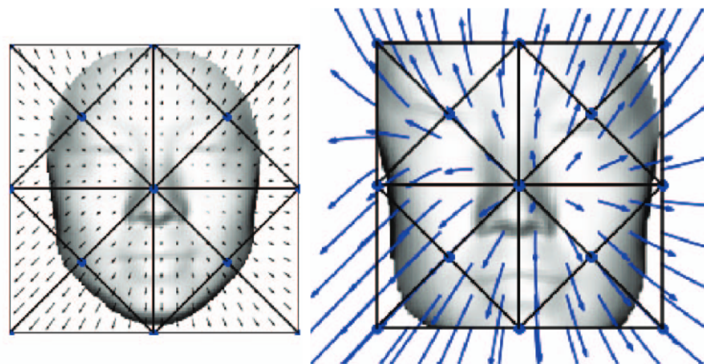

Abbildung 3.2.2: *Beispiel einer CPA-B Transformation.* ***Links****: Die schwarzen Pfeile definieren ein Geschwindigkeitsfeld welches genutzt wird um die blauen Pfade (****Rechts****) zu berechnen.* ***Rechts****: Die blauen Pfade stellen eine diffeomorphe Verformung dar [DFH18].*

Dieses Geschwindigkeitsfeld wird verwendet, um das Eingabebild entsprechend in Richtung der Pfeile des Feldes zu verformen. Es ist also im Vergleich zu einem STN keine affine Transformation, die eher etwas ausschneidet und rotiert, sondern geht eher in die Richtung einer „flüssigen“ Transformation, welches auch die Form von Objekten innerhalb einer Szene ändern kann (vgl. Abbildung 3.2.3). Ein STN kann gut relevante Bildausschnitte eingrenzen, ist jedoch bei zu großen Lernraten irreversibel

angelernt. Ein DTN ist reversibel, benutzt jedoch mehr Ressourcen [DFH18]. Daher ist die Kombination der beiden Ansätze sinnvoll, um die Vorteile beider auszunutzen. Das Vorgehen eines DTN ist in Abbildung 3.2.3 dargestellt und vergleicht ein DTN (Diffeomorphic) mit einem STN (Affine) sowie die Kombination beider. Das Ziel in diesem Anwendungsfall ist es, Gesichter von einander zu unterscheiden und zuzuordnen. Dabei konnte die Genauigkeit eines Netzwerks gesteigert werden, wenn ein DTN in Kombination mit einem STN verwendet wurde. Zu erkennen ist eine Zentrierung der Gesichter und das „flüssige“ Verformen über das gesamte Bild, wodurch die Gesichter besser voneinander unterschieden werden konnten.

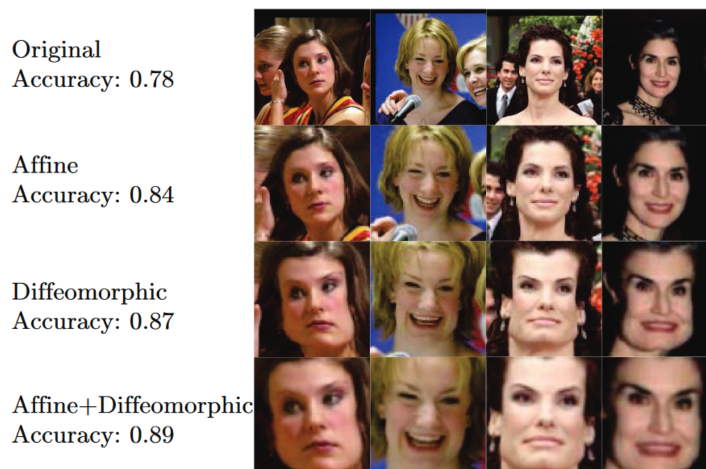

Abbildung 3.2.3: *Anwendung eines DTN bei der Erkennung von Gesichtern. Zu erkennen sind Verformungen, die eine affine Transformation nicht darstellen kann (da kein Geschwindigkeitsfeld) [DFH18].*

3.3 AUGMENTIERUNG

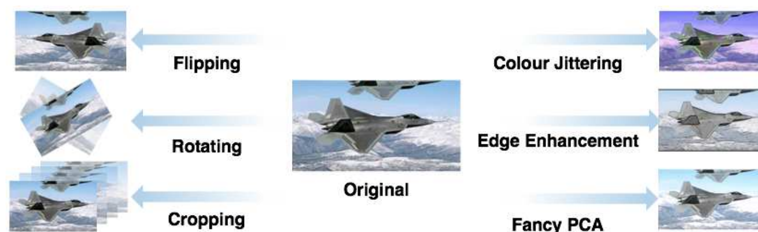

Abbildung 3.3.1: *Übersicht von verschiedenen Augmentierungsmethoden [TN17].*

Damit ein STN, DTN oder auch ganz allgemein ein CNN viele verschiedene unbekannte Eingaben mit verschiedenen u. a. positionellen Varianzen invariant verarbeitet, wird u. a. oft *Augmentierung* verwendet [TN17]. Die Augmentierung hat als Ziel den vorhandenen Datensatz maschinell zu bearbeiten und diesem wieder zuzuführen. Dadurch soll der Datensatz deutlich vergrößert werden, um die Varianz und Vielfalt innerhalb des Datensatzes zu steigern, ohne dass ein höherer manueller Annotationsaufwand entsteht. Bei den durchgeführten maschinellen Veränderung soll jedoch die Annotation beibehalten werden, sodass kein Informationsverlust hinsichtlich den Annotationen entsteht und auch keine neuen Annotation hinzugefügt werden [WGSM16]. Man kann dabei die Augmentierung in zwei Bereiche teilen: Die *geometrische Augmentierung*, welche ein Element des Datensatzes u. a. zuschneidet, verformt oder vergrößert und die *fotometrische Augmentierung*, welche u. a. die Farb- bzw. Grauwerte verändert, Rauschen hinzufügt oder Kanten intensiviert [TN17].
Ein weiteres Ziel ist es, eventuelle Einseitigkeit der Trainingspartition (vgl. Abschnitt 5.1) zu vermeiden. Sind beispielsweise in der Trainingspartition oft kleine Ziffern im Zentrum des Bildes, jedoch in der Testpartition kleine Ziffern in den Ecken der Eingabe, kann sich die Erkennungsrate verschlechtern, da das Netzwerk mit solchen Daten nicht angelernt wurde [WGSM16]. Eine Übersicht von verschiedenen geometrischen und fotometrischen Augmentierungsmethoden sind in Abbildung 3.3.1 dargestellt. In dieser Arbeit wird das *Edge Enhancement* aus [TN17] und die geometrischen Augmentierung analog zu [JSZK15] und [GBI+14] verwendet. In Kapitel 5 werden die jeweils verwendeten Augmentierungsmethoden mit den jeweiligen Hyperparamtern beschrieben, da abhängig vom Datensatz und den vorhandenen Annotationen verschiedene Techniken eingesetzt werden. So ist der SVHN (vgl. Abschnitt 5.1.3 und [NWC+11]) mit Boundingboxen um jede Ziffer annotiert, sodass ein minimales Rechteck um alle Ziffern berechnet werden kann, welches für die geometrischen Augmentierung Nutzen hat.

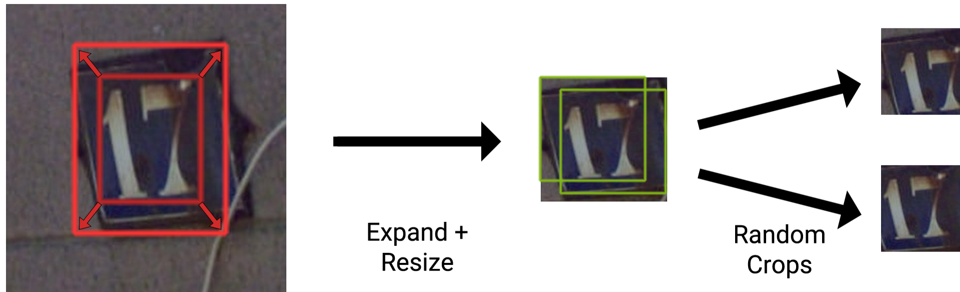

Abbildung 3.3.2: *Darstellung einer geometrischen Augmentierung. Die innere und dunkelrote Box wird um 50% vergrößert, um die äußere und hellrote Box zu erzeugen. Nach der neuen Skalierung und wird ein zufälliger Bereich fester Dimension aus diesem Zwischenbild ausgewählt (grüne Boxen). Diese stellen mögliche Elemente des Datensatzes dar. Das Bild stammt aus dem SVHN Datensatz [NWC+11].*

Diese minimale Boundingbox wird, wie in [JSZK15], [BMK15] oder [GBI+14] beschrieben, zunächst in alle Richtungen vergrößert und neu skaliert. Daraufhin wird ein kleinerer Bereich zufällig aus diesem Bild ausgeschnitten und verwendet. Dieses Vorgehen ist in Abbildung 3.3.2 dargestellt. Die Skalierung nach der Vergrößerung sowie die Dimension der zufälligen Bereiche sind dabei optionale Parameter. So kann beispielsweise immer ein konstanter Bereich zufällig ausgewählt werden, oder ein Bereich der abhängig von der Eingabedimension ist. Die Hyperparameter dieser Augmentierungsmethode sind also *Expand*, *Resize* und *Crop-Size*. Das *Edge Enhancement* hat als Hyperparameter nur *Range*, welcher einen Bereich definiert aus dem die Stärke der Kontrasterhöhung zufällig bestimmt wird.

4 METHODIK

Die verschiedenen Schichttypen sollen nun innerhalb eines Netzwerks die liegende Funktion der Mustererkennung approximieren. Verschiedene Architekturen erweisen sich dabei besser als andere bezogen auf ein spezielles Erkennungsproblem (vgl. Abschnitt 5). Die Architektur soll die unbekannte Zuordnung (Klassifizierung) zwischen einem Eingabebild und den dazugehörigen Klassen bzw. der einzelnen Klasse lernen. Insbesondere soll der Einfluss eines Transformer Networks (vgl. Abschnitt 3.1 und 3.2) untersucht werden. Dazu muss zunächst eine Architektur definiert (vgl. Abschnitt 4.1) und eine Trainingsmethode festgelegt werden (vgl. Abschnitt 4.2). Dann kann in eine bestehende Basis ein STN hinzugefügt bzw. entfernt werden, um den Einfluss bei sonst gleichbleibender Architektur zu untersuchen.

4.1 ARCHITEKTUR

Die folgenden Architekturen lassen sich grob in drei Abschnitte aufteilen. *Faltung* beschreibt dabei alle Convolution-Layer und Pooling-Layer (vgl. Abschnitt 2.2.1 und 2.2.2), wobei dort auch STNs bzw. DTNs auftreten können. *FC* fasst ggf. mehrere Fully-Connected-Layer zusammen (vgl. Abschnitt 2.2.3). Anschließend wird mithilfe von einem letztem Fully-Connected-Layer die *Ausgabe* berechnet, welche jedoch von der Problemdomäne abhängt. Ein STN gibt 6 Parameter für θ, eine MNIST Architektur die geschätzte Klasse aus den 10 möglichen und eine SVHN Architektur gibt für alle 5 Stellen eine von 11 Klassen und die Länge aus.
Jede Architektur lässt sich durch verfolgen der jeweiligen Spalte ablesen. Alle darin durchquerten Zellen sind Teil der Architektur und beschreiben jeweils eine Schicht. *conv*$(F, d = D, (K, S, P))$ kennzeichnet einen Convolution-Layer mit Filteranzahl F. Dabei können die jeweiligen Standardwerte der Architekturen durch $d = D$ und (K, S, P) überschrieben werden. D beschreibt den verwendeten Dropout, K die Kernel-Size, S die Stride und P das verwendete Padding (vgl. 2.2.1). *maxpool* markiert ein Pooling-Layer mithilfe der Maximum Funktion und $K = 2$ bzw. $S = 2$ (vgl. 2.2.2). *STN*(X) betitelt dabei ein Spatial Transformer Network mit Architektur X (vgl. Tabelle 4.1.1 und Abschnitt 3.1). *fc*(N, D) bezeichnet einen Fully-Connected-Layer mit N Neuronen, die mit allen Neuronen der vorherigen Schicht verbunden sind und ein Dropout D

STN Architekturen

Name	$\mathbf{A}_{tpp}$	$\mathbf{A}_{spp}$	**A**	**B**	$\mathbf{B}_{spp}$	**M**	$\mathbf{FC}_1$	$\mathbf{FC}_2$
Faltung	conv(32)			conv(32)		conv(20, p=0)		
	maxpool							
	conv(32)			conv(64)		conv(20, p=0)		
						maxpool		
				conv(64)				
PP	TPP	SPP			SPP			
FC	fc(32)			fc(256)			fc(32)	fc(48)
	fc(32)			fc(128)		fc(24)	fc(32)	fc(48)
$FC - \theta$	fc(6) ohne Akt. Funktion							
Gitter	96×64	78×44	$B \times H$		$128 \times H'$	$B \times H$		

Tabelle 4.1.1: *Übersicht der verwendeten Architekturen der Spatial Transformer Networks. Ein* ***conv(X)*** *steht standardmäßig für* ***conv***$(X, d = 0, (5, 1, 2))$ *und* ***fc***(N) *für* ***fc***$(N, 0)$*. Das Pyramid Pooling (vgl. Abbildung 2.2.4) wird gesondert am Ende der Faltungsschichten in* ***PP*** *betrachtet und besteht aus den Leveln* $X = \{17, 11, 7, 5, 3, 1\}$*. Das Gitter bestimmt die Dimension von* G *(vgl. Formel 3.1.1), wobei* B *und* H *Platzhalter für die Dimension der Eingabe sind und* H' *die angepasste (mit Respekt zum Seitenverhältnis) Höhe ausgehend von* H *bezeichnet, wenn eine konstante Breite angegeben ist.*

verwendet. Wenn nicht anders spezifiziert, benutzt jede Ebene anschließend eine *ReLU* Aktivierungsfunktion und Convolution-Layer zusätzlich jeweils *Batch Normalization* (BN) (vgl. Abschnitt 2.2.7 und 2.2.4).

Zunächst sind in Tabelle 4.1.1 die verwendeten Architekturen der STNs dargestellt. Ein STN kann, wie bereits beschrieben, unabhängig in ein existierendes CNN eingesetzt und gleichzeitig mit allen Parametern angelernt werden. Daher bilden sie eine weitere Art von Schicht, die dann in den folgenden Architekturen für verschiedene Datensätze eingesetzt wird. Die Basis der STNs ist an [JSZK15] angelehnt. So ist A genau nach den Spezifikationen von [JSZK15] konstruiert, wobei noch zusätzlich BN verwendet wird. B behält die Grundstruktur bei, ist jedoch tiefer und die Schichten haben stellenweise mehr Filter als A. M ist ein STN, welches für den MNIST Datensatz und dessen Variationen eingesetzt wird und orientiert sich auch an [JSZK15]. $FC_{1/2}$ sind ebenfalls aus selbigem Paper, jedoch ist FC_2 eine größere Version von FC_1, um den

MNIST Architekturen

Name	$\mathbf{N_1}$	$\mathbf{N_2}$	**O**
Faltung / STN	STN(M)	STN(A)	
	conv(32)		
	maxpool		
	STN(M_{mini})		
	conv(64)		
	maxpool		
FC	fc(128, 0.05)		
Ausgabe	softmax fc(10)		

Tabelle 4.1.2: *Übersicht der verwendeten Architekturen der CNNs für den MNIST Datensatz und dessen Variationen. Ein **conv(X)** steht standardmäßig für **conv**($X, d = 0.2, (5, 1, 0)$). M_{mini} ist hierbei eine Abwandlung von M aus Tabelle 4.1.1, die aber keinen zweiten Convolution-Layer und auch keinen zweiten Pooling-Layer benutzt. Jedes Netz gibt mithilfe der Softmax-Aktivierungsfunktion eine Art Wahrscheinlichkeitsverteilung aller Klassen aus.*

Unterschied in der Genauigkeit der Klassifikation weiter zu untersuchen. Es wurde sich sehr genau an [JSZK15] gehalten, um einerseits die Implementierung in PyTorch [PGC+17] zu verifizieren und andererseits um auf dieser Grundlage aufzubauen und neue Experimente mit weiteren Techniken wie SPP oder TPP zu evaluieren. A_{spp} und A_{tpp} geben jeweils ein Volumen mit fester Breite und Höhe aus, sowie die gleiche Tiefe (Merkmalskarten) wie die Eingabe. Daraus ergibt sich, dass ab diesem STN garantiert werden kann, welche Dimension ein Volumen innerhalb der Architektur hat. Dadurch können Bilder verschiedener Größen ohne vorheriges Skalieren in das Netzwerk eingegeben werden und FC-Schichten mit einer konstanten Anzahl an Gewichten (und damit Verbindungen zu vorherigen Neuronen) benutzt werden. Im allgemeinen besteht auch ein STN aus den zwei grundlegenden Teilen der Merkmalsidentifizierung und der darauffolgenden „Klassifizierung“, wobei ein STN keine Klasse schätzt, sondern, abhängig von den Merkmalsausprägungen, die Parameter der Transformation (vgl. Formel 3.1.1) bestimmt.

Die in Tabelle 4.1.2 dargestellten Architekturen sollen den MNIST Datensatz (vgl. Abschnitt 5.1.1) klassifizieren und die zugehörige Mustererkennungsfunktion appro-

ximieren und sind auf Basis von [JSZK15] konstruiert. Die Ausgabe eines solchen Netzwerks ist eine Art Wahrscheinlichkeitsverteilung für alle möglichen 10 Klassen (Ziffern $0-9$) und wird mithilfe einer Softmax Aktivierungsfunktion (vgl. Abschnitt 2.2.4) realisiert. Bezogen auf die eingesetzten Convolution-Layer sind alle Architekturen gleich. Sie unterscheiden sich nur in der Verwendung von STNs und den jeweiligen STN Architekturen (vgl. Tabelle 4.1.1). Jedoch benutzen die Convolution-Layer eine Kernel-Size von 5, welches sich zu [JSZK15] unterscheidet (9 und 7). Darüber hinaus wird Dropout und BN benutzt, welches beides in [JSZK15] nicht verwendet wird. Das verdeutlicht, wie in Kapitel 5 näher thematisiert, dass verschiedene Architekturen eine ähnliche Genauigkeit haben können.

In Tabelle 4.1.3 sind die verwendeten Architekturen für die verschiedenen Variationen des SVHN Datensatzes (vgl. Abschnitt 5.1.3 und folgende) dargestellt. Gegenüber zu den MNIST Netzwerken gibt ein Netzwerk für den SVHN Datensatz insgesamt 6 Wahrscheinlichkeitsverteilungen aus. Alle einzelnen Fully-Connected-Layer auf Schicht der Ausgabe verwenden die Softmax-Funktion und bekommen als Eingabe die Ausgabe des FC Teils. Alle 6 verwerten die selbe Eingabe und leiten daraus verschiedene Wahrscheinlichkeitsschätzungen ab. Die ersten 5 bestimmen jeweils die Verteilung über 11 Klassen (Ziffern 0-9 und „leer"). Der übrige Fully-Connected-Layer soll die Länge der gesamten Zahl bestimmen und gibt daher eine Wahrscheinlichkeitsverteilung über 6 Klassen (Zahlenlänge 0-5) aus. N und M sind aus [JSZK15] übernommen und O ist die gleiche Architektur wie N nur ohne vorherigen STN. M_{pool} und M_{wide} sind für beliebige Eingaben mit verschiedenen Dimensionen konstruiert und basieren auf M. M_{wide} untersucht, welchen Effekt es hat, wenn ein STN ein Gitter im Seitenverhältnis 21:9 ausgibt, statt eines Quadratischen, da es sich um Sequenzen von Ziffern handelt, die dementsprechend in die Breite statt in der Höhe skalieren, je länger eine solche Sequenz ist. M_{pool} versucht auf den Originalbildern, ohne jegliche Vorverarbeitung und Augmentierungstechniken, die Zahl zu finden und zu erkennen und verwendet dafür mehrere STNs. Die Grundidee besteht darin, zunächst die Ziffer zu begradigen (A_{spp}) und daraufhin zu zentrieren (A_{tpp} und folgende) während immer mehr Merkmalsausprägungen berechnet werden. Jedoch kann man dies nicht direkt steuern, da alle STNs automatisch im Rahmen der Backpropagation (vgl. Abschnitt 2.2.6) angelernt werden und nur versuchen den Loss zu minimieren.

SVHN Architekturen

<table>
<tr><th>Name</th><th>N</th><th>O</th><th>M</th><th>M_{pool}</th><th>M_{wide}</th></tr>
<tr><td rowspan="22">Faltung / STN</td><td>STN(A)</td><td></td><td>STN(FC1)</td><td>STN(A_spp)</td><td>STN(B_spp)</td></tr>
<tr><td colspan="3"></td><td>conv(12)</td><td></td></tr>
<tr><td colspan="3"></td><td>maxpool</td><td></td></tr>
<tr><td colspan="3"></td><td>STN(A_tpp)</td><td></td></tr>
<tr><td colspan="3">conv(48, d=0)</td><td>conv(48)</td><td>conv(64, d=0)</td></tr>
<tr><td colspan="5">maxpool</td></tr>
<tr><td colspan="2"></td><td>STN(FC1)</td><td>STN(FC2)</td><td>STN(FC1)</td></tr>
<tr><td colspan="4">conv(64)</td><td>conv(92, d=0.2)</td></tr>
<tr><td colspan="4"></td><td>conv(92, d=0, (7,2,3))</td></tr>
<tr><td colspan="2"></td><td>STN(FC1)</td><td>STN(A)</td><td>STN(FC2)</td></tr>
<tr><td colspan="4">conv(128)</td><td>conv(128, d=0.4)</td></tr>
<tr><td colspan="5">maxpool</td></tr>
<tr><td colspan="2"></td><td>STN(FC1)</td><td></td><td>STN(FC1)</td></tr>
<tr><td colspan="3">conv(160)</td><td></td><td>conv(160, d=0.4)</td></tr>
<tr><td colspan="3"></td><td>STN(FC2)</td><td></td></tr>
<tr><td colspan="3">conv(192)</td><td></td><td>conv(192, d=0.4)</td></tr>
<tr><td colspan="5">maxpool</td></tr>
<tr><td colspan="3">conv(192)</td><td></td><td>conv(192, d=0.4)</td></tr>
<tr><td colspan="3"></td><td>STN(FC2)</td><td></td></tr>
<tr><td colspan="3">conv(192)</td><td>conv(212)</td><td>conv(212, d=0.4)</td></tr>
<tr><td colspan="5">maxpool</td></tr>
<tr><td colspan="3">conv(192)</td><td>conv(256)</td><td>conv(256, d=0.4)</td></tr>
<tr><td rowspan="3">FC</td><td colspan="5">fc(3072, 0.5)</td></tr>
<tr><td colspan="5">fc(3072, 0.5)</td></tr>
<tr><td colspan="5">fc(3072, 0.5)</td></tr>
<tr><td>Ausgabe</td><td colspan="5">Parallel+softmax: 5*fc(11, 0.0)+fc(6, 0.0)</td></tr>
</table>

Tabelle 4.1.3: *Übersicht der verwendeten Architekturen der Spatial Transformer Networks. Ein **conv(X)** steht standardmäßig für **conv**$(X, d = 0.5, (5, 1, 2))$. Jedes Netz klassifiziert parallel 5 Ziffern und die Länge der gesamten Zahl, indem die Softmax-Funktion die Wahrscheinlichkeitsverteilung jeweils über alle möglichen Klassen berechnet.*

4.2 TRAINING

Alle Netzwerke werden mithilfe des Stochastic Gradient Descent (vgl. Abschnitt 2.2.6) trainiert. Bei Datensätzen mit konstanter Dimension der Eingabe wird der Mini-Batch Gradient Descent mit einer Batchsize von 128 (SVHN) und 256 (MNIST) verwendet (vgl. Abschnitt 5.1 für die Datensätze). Ein Optimierungsschritt einer Batch wird Iteration genannt. Die Lernrate ist anfänglich auf 10^{-2} festgelegt und wird je nach Datensatz zu verschiedenen Zeitpunkten des Trainings um einen festen Faktor von 10 verringert. Die Lernrate für STNs innerhalb der Architekturen ist anfänglich zwischen 10^{-3} und 10^{-5}, um dem Effekt aus Abbildung 3.2.1 entgegen zu wirken. Auch diese Lernrate wird zu gleichen Zeitpunkten des Trainings und um selbigen Faktor verringert. Eine Epoche bezeichnet einen Durchlauf des gesamten Datensatzes zum trainieren. Experimente des MNIST Datensatzes wurden für 645 Epochen trainiert, um die in [JSZK15] beschriebenen 150.000 Iterationen zu erreichen ($645 * \frac{60000}{256} \approx 151.000$) (vgl. Abschnitt 5.1.1). Für Experimente des SHVH Datensatzes wurden die Netzwerke jedoch nur für 360 Epochen und damit für ungefähr 100.000 Iterationen trainiert, was deutlich weniger ist als die 400.000 aus [JSZK15]. Jedoch haben die Graphen der Genauigkeit der Klassifikation in Bezug zur Epoche gezeigt, dass die Netzwerke bereits früh konvergieren (vgl. Abbildung 4.2.1). Die Lernrate wurde analog zu [JSZK15] bei jedem Drittel (215 Epochen) des gesamten Trainings verringert (MNIST) bzw. jedes Fünftel (72 Epochen) (SVHN). Die Parameter der letzten Schicht eines STN (Output

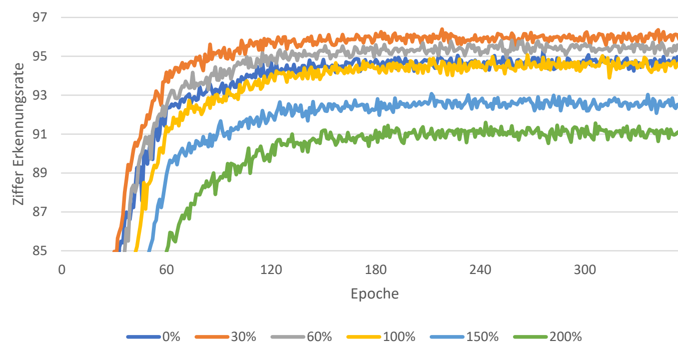

Abbildung 4.2.1: *Vergleich der Genauigkeit über die Trainingsdauer hinweg einer gleichen Architektur für den SVHN Datensatz und verschiedenen Augmentierungsmethoden ($0 - 200\%$) welches in Kapitel 5 weiter untersucht wird (vgl. Tabelle 5.3.3). Alle Netzwerke haben innehalb von 250 Epochen ein lokales Maximum erreicht und zeigen keine deutlichen Verbesserungen mehr bei weiteren Epochen.*

Layer) werden mit der Identitätstransformation und alle anderen wie in [HZRS16] beschrieben, initialisiert. Trainiert werden die Netzwerke auf verschiedenen, aber einzelnen, GPUs (NVIDIA Titan X, 1080, 1070 Ti).

5 EXPERIMENTE UND EVALUATION

Nun werden die vorgestellten Architekturen mit der beschriebenen Methodik angelernt. Um daraufhin die Genauigkeit mehrerer Architekturen auf gleichen Datensätzen bzw. gleiche Architekturen auf verschiedenen Datensätzen zu vergleichen, werden folgende Metriken benötigt. Diese Metrik beschreibt im einfachsten Fall, wie viel Prozent der Testpartition (vgl. Abschnitt 5.1) korrekt klassifiziert wurden oder wie gut die Länge einer aus ggf. mehreren Ziffern bestehenden Zahl bestimmt werden kann. Formal lässt sich durch diese Überlegung die *allgemeine Genauigkeit* (allgemeine *Accuracy*) der Klassifikation definieren. Dabei beschreibt X die Menge an Eingaben, K die Menge an parallelen Klassifikatoren (für Ziffern), y_i stellt ein Klassifikator des Netzwerks und $\hat{y}_i$ die Annotationen für eine Eingabe (vgl. Formel 2.2.17 und 2.2.18). max_i gibt den Index des maximalen Elements eines Vektors zurück und bestimmt damit einerseits die Schätzung des Netzwerks sowie den Index der „korrekten" Klasse, basierend auf den Annotationen. Zunächst kann die Menge an korrekt klassifizierten Elementen C bestimmt werden, wodurch der Anteil von der zu testenden Menge bestimmt wird:

$$
\begin{aligned}
C_a &= \{x \in X \mid \forall i \in K \ \max_i(y_i(x)) == \max_i(\hat{y}_i(x))\} \\
P_{all} &= \frac{|C_a|}{|X|}
\end{aligned}
\tag{5.0.1}
$$

Analog lässt sich dadurch die *Genauigkeit der Ziffernschätzung* bestimmen, welche beschreibt, wie viele einzelne Ziffern korrekt klassifiziert wurden. A priori wurde in Abschnitt 4.1 festgelegt, wie viele Klassifikatoren für die Ziffern benötigt werden. Für den MNIST Datensatz und SVHN Format 2 ist dies nur ein Klassifikator, jedoch sind es für den SVHN Format 1 und Augmented Datensatz 5 Klassifikatoren ($K = \{1, 2, 3, 4, 5\}$) (vgl. Abschnitt 5.1). Dadurch lässt sich die Gesamtzahl an zu klassifizierenden Objekten durch das Produkt der Klassifikatoren und der Anzahl an Elementen des Datensatzes

bestimmen. Auch wenn keine Ziffer an einer Stelle vorkommt, muss die Klasse „leer“ korrekt bestimmt werden und geht damit in die Genauigkeit mit ein.

$$\begin{aligned} C_z &= \{(x, i) \in X \times K \mid \max_i(y_i(x)) == \max_i(\hat{y}_i(x))\} \\ P_{digit} &= \frac{|C_z|}{|X \times K|} \end{aligned} \tag{5.0.2}$$

Für mehrstellige Ziffern lässt sich darüber hinaus noch die *Genauigkeit der Längenschätzung* definieren, welche den Anteil an richtigen Vorhersagen der Länge aus einer Menge an Eingaben darstellt. Die Länge ist dabei der Index des ersten Vorkommnisses der Klasse „leer“ oder die maximale Länge von 5. Sei dieser Klassifikator durch l und die durch die Annotationen definierte Länge mit $\hat{l}$ gegeben, dann gilt für die *Genauigkeit der Längenschätzung*

$$\begin{aligned} C_l &= \{x \in X \mid l(x) == \hat{l}(x)\} \\ P_{len} &= \frac{|C_l|}{|X|}. \end{aligned} \tag{5.0.3}$$

Ausgehend von einer Metrik lässt sich die zugehörige *Fehlerrate* (Error) bestimmen, die hier zum Vergleich mit verwandten Arbeiten verwendet wird:

$$\begin{aligned} E_{all} &= 1 - P_{all} \\ E_{digit} &= 1 - P_{digit} \\ E_{len} &= 1 - P_{len} \end{aligned} \tag{5.0.4}$$

5.1 DATENSÄTZE

Die Datensätze bestehen grundlegend aus zwei Partitionen. Eine Partition wird zum Trainieren (vgl. Abschnitt 2.2.6) des Netzwerks benutzt und die andere zur Evaluation. Dies ist notwendig, um ein realistisches Bild der Genauigkeit zu bekommen. Der Testdatensatz simuliert unbekannte Daten und prüft die Generalisierungsfähigkeit des Netzwerks. Würde das Netzwerk nur auf einer Partition angelernt und getestet werden, könnte sich das Netzwerk mit genügend Parametern und Zeit den Datensatz annähernd vollständig merken (vgl. Abschnitt 2.2.7) und so ein verzerrtes Bild der Genauigkeit des Netzwerks liefern. Darüber hinaus sind die Datensätze *annotiert*. Diese Annotationen sollen die korrekte Klasse eines Elements des Datensatzes beschreiben. Es kann jedoch mehrere Annotationen für ein Element des Datensatzes geben, so ist z. B. der SVHN Datensatz (vgl. Abschnitt 5.1.3) zusätzlich neben den Klassen für alle Ziffern im Bild noch mit Positionsinformationen und Dimensionsinformationen einer

jeden Ziffer annotiert.
Zudem können auf die vorhanden Datensätze unterschiedliche Techniken aus den Bereichen der Vorverarbeitung und der Augmentation angewandt werden, um einerseits andere Testszenarien zu schaffen und andererseits die Robustheit des Netzwerks zu steigern (vgl. Abschnitt 3.3).

5.1.1 *MNIST Datensatz*

Der **M**odified **N**ational **I**nstitute of **S**tandards and **T**echnology Datensatz besteht aus 60000 handgeschriebenen Ziffern in der Trainingspartition und 10000 in der Testpartition [LC10]. Alle Ziffern wurden zuvor größen-normalisiert und mit dem Massezentrum der Pixelwerte in einem leerem 28x28 Bild zentriert (vgl. Abbildung 5.1.1). Dieser Datensatz bildet eine Basis um mithilfe von verschiedenen Transformationen neue Elemente zu generieren. Diese neuen Elemente bilden damit Partitionen eines neuen, von der Basis abgeleiteten, Datensatzes, die ggf. schwieriger für bisherige Architekturen sind. Schwieriger bedeutet in dem Sinne eine verschlechterte Genauigkeit bei gleichbleibender Architektur.

5.1.2 *Rotated / Distorted MNIST*

Um den Effekt von Transformationen innerhalb des Datensatzes zu untersuchen und um den Einfluss von STNs einschätzen zu können, werden auf dem MNIST Datensatz verschiedene Transformationen angewandt um dadurch Varianten des Datensatzes zu erzeugen. Die beschriebenen Transformationen sind analog zu denen, die in [JSZK15] verwendet wurden. Alle drei Varianten des MNIST Datensatzes sind in Abbildung 5.1.1 dargestellt. Bei jeder Abfrage eines Elements einer Partition des *Rotated MNIST* Datensatzes wird vor der Ausgabe eine zufällige Rotation um $\pm 45°$ durchgeführt. Da es sich um ein quadratisches Ausgangsbild mit Breite $b = 28$ handelt, kann dies zu einer neuen Breite von $\sqrt{2}b$ bei einer Rotation um $45°$ führen. Daher können bei dieser Transformation Informationen verloren gehen. Darauf aufbauend wird im *Distorted MNIST* Datensatz neben der Rotation auch Translation und Skalierung auf den zugrundeliegenden MNIST Datensatz (5.1.1) angewendet. Das Ausgangsbild mit Seitenlänge b_1 wird zunächst zufällig auf eine Skalierung zwischen 80% und 120% verkleinert bzw. vergrößert. Daraufhin wird erneut eine zufällige Rotation um $\pm 45°$ angewendet, jedoch wird, im Gegensatz zu *Rotated MNIST*, das rotierte Bild wenn nötig vergrößert, damit keine Informationen des Ausgangsbildes verloren gehen. Zuletzt wird dieses Bild zufällig in ein größeres, quadratisches und leeres Bild mit

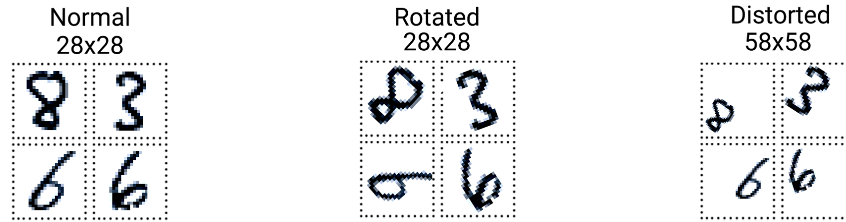

Abbildung 5.1.1: *Beispielbilder aus den verschiedenen Variationen des MNIST Datensatzes [LC10]. Die „normalen" Ziffern haben geringe Unterschiede im Raum der affinen Transformationen, sodass die gleiche Ziffer z. B. an einer leicht anderen Position und/oder anders gedreht erscheinen kann. Auf diesen Bildern werden affine Transformationen angewandt, um den Rotated und Distorted MNIST Datensatz zu erzeugen. Nur der Distorted MNIST Datensatz hat eine größere Dimension von* 58×58.

Seitenlänge b_2 platziert. Ausgehend von der maximalen Rotation (wenn diese $\geqslant 45°$, sonst ändert sich der Faktor $\sqrt{2}$), Skalierung s_{max} und Translation t_{max}, lässt sich b_2 wie folgt berechnen.

$$b_2 = \lceil s_{max} * b_1 * \sqrt{2} + t_{max} \rceil$$

Konkret ergibt sich für $b_1 = 28$, $s_{max} = 120\%$, $t_{min} = 10$: $b_2 = 58$. So wird auf der Basis von dem 28x28 MNIST Datensatz ein 58x58 Distorted MNIST Datensatz erzeugt.

5.1.3 *SVHN Datensatz (Format 1/2)*

Der **S**treet **V**iew **H**ouse **N**umbers Datensatz besteht aus 3 Partitionen. Jede davon enthält Bilder mit drei Farbkanälen (RGB) und bildet eine natürliche Szene einer Hausnummer inklusive Umgebung ab. Die Daten stammen aus den Google StreetView Aufnahmen[NWC+11]. Die Trainingspartition umfasst 33402 Hausnummer mit jeweils einer Länge zwischen $1 - 6$ Ziffern und insgesamt 73257 Ziffern. Die Testpartition besteht wiederum aus 13068 Hausnummern mit gleicher maximalen Länge und insgesamt 26032 Ziffern. Zudem existiert eine zusätzliche *extra* Partition, die vergleichsweise schwierigere Bilder mit mehr und/oder anderen Transformationen und mehr Hintergrund enthält. Diese besteht aus 202353 Hausnummern gleicher variabler Länge und insgesamt 531131 Ziffern[NWC+11]. Format 1 besteht aus den originalen Bildern, sodass die Hausnummer nur einen Bruchteil des Bildes ausmachen kann oder es aber vollständig einnimmt (vgl. Abbildung 1.1.1 und 5.1.2). Darüber hinaus sind die Bilder von verschiedener Größe, sodass Poolingmethoden (vgl. Abschnitt 2.2.2) oder vorherige Vorverarbeitung ggf. inklusive Augmentation angewendet werden müssen

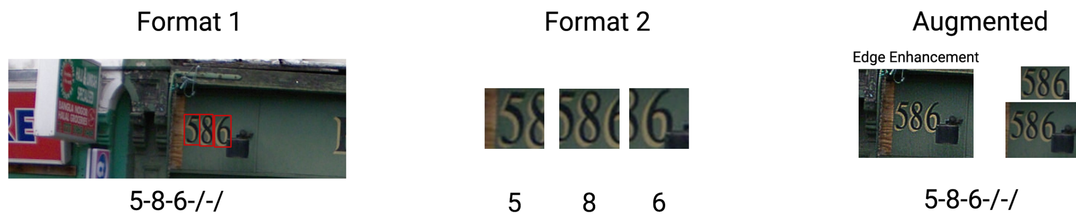

Abbildung 5.1.2: *Beispielbilder aus den verschiedenen Variationen des SVHN Datensatzes [NWC+11] inklusive der annotierten „richtigen" Klassen. Format 1 beinhaltet die Originalbilder und Annotationen bezüglich den Boundingboxen einer Ziffer. Format 2 schneidet jeweils die einzelnen Ziffern aus und skaliert diese auf eine feste Größe. Beispielhaft ist hier die Augmentierung mit Edge Enhancement und zwei verschiedenen Hyperparametern hinsichtlich Random-Crop und Expand in der Spalte Augmented dargestellt. Resize ist in diesem Beispiel 0.*

(vgl. Abschnitt 3.3). Der Datensatz enthält Annotationen über die Labels (0-9 oder *leer*) der Ziffern sowie Informationen der Boundingbox um jede einzelne Ziffer. Im Gegensatz dazu ist Format 2 bereits Vorverarbeitet und schneidet jede Ziffer aus den originalen Bildern mithilfe der annotierten Boundingboxen aus, vergrößert dieses Rechteck zu dem dazugehörigen minimalen Quadrat und skaliert diese auf ein 32x32 Bild. Dies führt dazu, dass neben der eigentlichen Zahl auch andere Zahlen vorkommen können[NWC+11] (vgl. Abbildung 5.1.2). Dementsprechend hat dieser Datensatz so viele Elemente wie es Ziffern im SVHN Datensatz gibt.

5.1.4 *Augmented SVHN*

Auf Grundlage von 5.1.3 (Format 1) können wie in 3.3 beschrieben verschiedene geometrische Augmentierungsmethoden angewandt werden. Zuerst kann bestimmt werden, um wie viel Prozent die minimale Boundingbox um alle Ziffern vergrößert werden soll (*Expand*). Dann kann eine Skalierung ohne Beachtung der Seitenverhältnisse folgen (*Resize*). Anschließend kann ein zufälliger Bereich fester Größe aus diesem Bild ausgeschnitten werden (*Random-Crop*). Zudem kann die fotometrische Augmentierung *Edge Enhance* zu einem festgelegten zufälligen maximalen Anteil angewendet werden. Dies resultiert in 4 Parameter (*Expand*, *Resize*, *Random-Crop*, *Edge Enhance*) welche den Datensatz verändern. So kann beispielsweise ein Parameter variiert und die anderen konstant gewählt werden um die Auswirkung dieses Parameters zu untersuchen (vgl. Abbildung 5.1.2). Da die Veränderung der Hyperparamter deutlich unterschiedliche

Eingabebilder erzeugen, ist dies als eigenständiger Datensatz aufgeführt. Im allgemeinen wird die Augmentierung als zusätzliches Mittel zur Verbesserung der Robustheit eingesetzt.

5.2 KLASSIFIKATION EINZELNER ZIFFERN

MNIST E_{all}	Normal	Rotated	Distorted
N_1	0.64	0.64	0.56
N_2	**0.61**	**0.57**	**0.5**
O	0.73	0.92	1.2
DM-FCN [JSZK15]		1.2*	0.8
DM-CNN [JSZK15]		0.7*	0.5
NN-DropConnect [WZZ+13]	**0.52**		**0.21**

Tabelle 5.2.1: *Vergleich der vorgestellten MNIST Architekturen (vgl. Tabelle 4.1.2) zwischen den verschiedenen Varianten des Datensatzes. DM-FCN und DM-CNN stammen aus [JSZK15]. NN-DropConnect wurde in [WZZ+13] vorgestellt und nutzt eine abgewandelte Art des Dropouts. *: Bei Google Deepmind [JSZK15] wurde bei der Rotated MNIST Variante eine zufällige Rotation zwischen $-90° \leqslant x \leqslant +90°$ angewandt, statt wie hier $-45° \leqslant x \leqslant +45°$. Dies wurde verändert um eine bessere Vergleichbarkeit zwischen der Rotated und der Distorted Variante zu schaffen. Distorted führt dadurch nur neue Transformationen ein, ändert aber keine bisherigen. Ferner kann bei einer Rotation um 90° eine 6 nicht von einer 9 unterschieden werden, wodurch die Genauigkeit sinkt.*

Das Ziel ist es, zu untersuchen in wie weit ein STN die Genauigkeit bzw. die Fehlerrate beeinflusst, wenn verschiedene Transformationen im Datensatz auftreten. Dazu werden die in Tabelle 4.1.2 und 4.1.1 eingeführten Architekturen verwendet und wie in Abschnitt 4.2 beschrieben angelernt. Zunächst handelt es sich um ein Erkennungsproblem einer einzelnen Ziffer, die entweder zentriert in einer Eingabe oder durch affine Transformationen in einem Teil des Bildes platziert ist, wodurch $P_{all} = P_{digit}$ gilt und somit nur P_{all} bzw. E_{all} betrachtet wird. Hierfür werden alle Variationen des MNIST Datensatzes (vgl. Abschnitt 5.1.1) und der SVHN Format 2 Datensatz (vgl. Abschnitt 5.1.3) miteinander verglichen.

In Tabelle 5.2.1 werden die verschiedenen Fehlerraten der vorgestellten Netzwerke verglichen und den Netzwerken von Google Deepmind ([JSZK15]) gegenübergestellt. Darüber hinaus ist auch das derzeitige state-of-the-art Netzwerk NN-DropConnect

Abbildung 5.2.1: *Einfluss eines STN innerhalb der* N_2 *Architektur. Das STN bestimmt Gitter über der Eingabe (Durch die Mehrfarbige Box gekennzeichnet), womit die Ausgabe dieser Schicht berechnet wird. Auffällig ist, dass gleiche Zahlen oft ähnlich transformiert werden, obwohl dessen Eingabe mit zufälligen Transformationen versehen ist. Die* 5 *wird oft leicht nach links gedreht dargestellt und die* 1 *sowie* 4 *oft leicht nach rechts. Dies könnte einen Einfluss auf die Genauigkeit des Klassifikators haben.*

aus [WZZ+13] in der Tabelle dargestellt. Die Experimente auf den MNIST Varianten haben gezeigt, dass ein STN erst einen Effekt hat, wenn überhaupt gut voneinander unterscheidbare Transformationen vorliegen. Die Ergebnisse des Distorted MNIST Datensatzes übertreffen sogar die des zugrunde liegenden Datensatzes. Man könnte dieses Ergebnis so interpretieren, dass die STNs einige Zahlen anders transformieren als andere, also abhängig von der Klasse eine Zahl z. B. eher immer leicht nach rechts oder links verschieben bzw. drehen, wodurch der Klassifikator mithilfe dieser durch den STN erhöhten klassenrelevanten Varianz eine bessere Genauigkeit erreicht. Das würde bedeuten, dass die nicht-klassenrelevante Varianz ausgenutzt wird um klassenrelevante Unterschiede hervorzuheben. Die Visualisierung (vgl. Abbildung 5.2.1) einiger Zwischenbilder der Testpartition des Netzwerks N_2 stützt die Hypothese. Generell wird die Robustheit des Netzwerks gesteigert, da eine Zahl zufällig im Bild platziert, skaliert und rotiert wird und dennoch korrekt erkannt werden kann. Weiterhin ist, wie erwartet, die Fehlerrate bei dem Netzwerk O ohne die Verwendung eines STN gestiegen, je mehr Transformationen auf dem Datensatz angewandt wurden. DM-FCN ist dabei das von Google **D**eep**M**ind veröffentliche Netzwerk, welches nur Fully-Connected-Layer und keine Faltungsschichten nutzt. DM-CNN ist mit $N_{1/2}$ vergleichbar (vgl. Abschnitt 4.1) und erzielt eine ähnliche Genauigkeit wie die Implementierung dieser Arbeit.
Jetzt werden die gleichen Netzwerke auf den SVHN Format 2 Datensatz getestet und

SVHN E_{all}	Format 2
N_1	**4.97**
N_2	5.07
O	7.06

Tabelle 5.2.2: *Vergleich der vorgestellten MNIST Architekturen (vgl. Tabelle 4.1.2) auf Grundlage des SVHN Format 2 (vgl. Abschnitt 5.1.3) Datensatzes.*

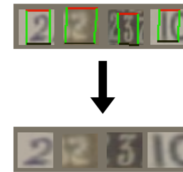

Abbildung 5.2.2: *Einfluss eines STN innerhalb der N_2 Architektur und auf dem SVHN Format 2 Datensatz.*

verglichen. Da der Datensatz mehr nicht-klassenrelevante Varianz, wie z. B. wechselnde Hintergrundfarben, Teile von anderen Ziffern links und rechts sowie unterschiedliche Farben der Ziffer selbst, enthält, sollte hier die Erkennungsrate sinken. Die Ergebnisse, welche in Tabelle 5.2.2 dargestellt sind, unterstützen die Vermutung.
Trotz der vorherigen Zentrierung einer Ziffer in einem Element des Datensatzes, verbessern STNs die Genauigkeit. In der Visualisierung ist zu erkennen, dass meist eine Boundingbox um die zentrierte Ziffer bestimmt wird, sodass keine Teile von anderen Ziffern im weiteren Verlauf des Netzwerks vorhanden sind. N_1 und N_2 liegen sehr nah beieinander, was im Rahmen des Fehlers sein könnte und keine weitere Aussagen darauf basierend getroffen werden können. Jedoch scheint die generelle Verwendung eines STN zielführend für das Erkennungsproblem zu sein.
Zusammenfassend lässt sich über die Verwendung von STNs bei der Erkennung von einzelnen Ziffern also sagen, dass diese sinnvoll sind, wenn die Problemdomäne Verzerrungen der zu erkennenden Objekten beinhaltet. Sind diese bereits zentriert bzw. haben eine geringe nicht-klassenrelevante Varianz, so hilft ein STN nur geringfügig, falls überhaupt.

Wenn nun DTNs (vgl. Abschnitt 3.2) statt STNs oder in Kombination mit STNs verwendet werden, verändert sich die erzielte Genauigkeit. Zum untersuchen dieses Effekts wurde ein DTN jeweils nach jedem in N_2 vorkommenden STN eingesetzt (N_{2+dtn}) bzw. ersetzt ($N_{2,dtnonly}$). Daraufhin werden diese beiden Netzwerke analog auf dem Rotated und Distorted MNIST Datensatz trainiert. Die Kombination beider Ansätze zeigt, dass der STN keine Transformationen mehr erlernt und der DTN diesen nahezu vollständig ersetzt. Dazu wird die jeweilige Ziffer „flüssig" transformiert, sodass vorherige Drehungen rückgängig gemacht werden. Die könnte daran liegen, dass ein DTN direkt nach ein STN eingesetzt wird und mit einer höheren Lernrate trai-

niert wird, wodurch dieser schneller merkbare Transformationen erlernt, wodurch der STN ausgebremst wird, da dadurch keine weiteren starken Transformationen benötigt werden. Die Vorgehensweise der ersten beiden einzelnen Transformer Networks ist in Abbildung 5.2.3 dargestellt und zeigt, dass ein STN in N_{2+dtn} keinerlei Transformation erlernt hat und ein DTN in dem Experiment ein STN vollständig ersetzt. Interessanterweise tritt dieser Effekt verstärkt auf dem Distorted MNIST Datensatz auf, wobei dort die STNs immer noch keinerlei Transformation vornehmen, jedoch auch die DTNs in diesem Fall keinen großen Effekt haben. Ohne weitere Änderungen werden die im Datensatz vorkommenden Transformationen also nicht rückgängig gemacht, trotzdem erreicht N_{2+dtn} eine Fehlerrate von 0.54% und ist damit nur knapp hinter dem Ergebnis von N_2 (0.5%). Die DTNs zentrieren die Ziffer annähernd, jedoch werden die vorkommenden Rotationen oftmals nicht gut im Vergleich zu N_2 ausgeglichen. Die verschiedenen Skalierungen hingegen werden jedoch gut korrigiert, sodass alle Ziffern nach dem ersten DTN in etwa die selbe Größe haben. $N_{2,dtnonly}$ erreicht eine Fehlerrate von 0.64% und ist dadurch interessanterweise schlechter als die Kombination beider Ansätze. Es sind die gleichen Effekte an Zentrierung, fehlender Rotation und ausgleich der Skalierung zu erkennen. Dadurch wird zwar bestätigt, dass ein DTN in N_{2+dtn} den vorherigen STN nahezu vollständig in seiner Funktion ersetzt, jedoch nicht komplett. Der STN hat in der Kombination immer noch einen nutzen, auch wenn dieser aus der Visualisierung nicht direkt offensichtlich ist.

5.3 KLASSIFIKATION MEHRERER ZIFFERN

Erneut wird untersucht, welchen Einfluss ein STN innerhalb von CNNs hat. Jedoch handelt es sich nun um Zahlen die, aus mehreren Ziffern bestehen und unterschied-

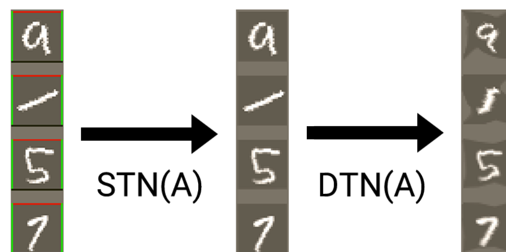

Abbildung 5.2.3: *Die ersten beiden Transformer Networks aus der N_{2+dtn} Architektur. Der STN bestimmt nun keine von der Identität abweichenden Transformationen, jedoch übernimmt der DTN dies.*

liche Länge haben. Da der normale SVHN Datensatz viele Variationen an Zahlen enthält, die verschieden groß relativ zur Bildgröße sind und diese selbst auch variiert, werden verschiedene Vorverarbeitungsmethoden verwendet, um ein besseres Bild des Effekts von STNs zu bekommen. Dazu werden verschiedene Testszenarien mithilfe verschiedener Vorverarbeitungen und Augmentierungen erzeugt. Zunächst wird untersucht, wie sich Netzwerke mit STN und ohne auf einem vorher augmentierten und vorverarbeiteten Datensatz verhält. Daraufhin wird die Vorverarbeitung entfernt, wodurch spezielle Pooling-Layer (vgl. Abschnitt 2.2.2) verwendet werden müssen. Dann wird der originale Datensatz untersucht und wie ein Netzwerk auf diesem die Zahl noch lokalisieren kann. Schließlich wird untersucht, in wie weit ein DTN die Genauigkeit beeinflusst, wenn diese in Kombination mit STNs eingesetzt werden.

5.3.1 *Augmentierung und Vorverarbeitung*

SVHN (Resize)	E_{all}	E_{digit}	E_{len}
N	68.3	22.42	26.91
M	**66.11**	**21.61**	**26.29**

Tabelle 5.3.1: *Vergleich zweier SVHN Architekturen bei Anwendung des naiven Ansatzes der Neuskalierung. Die Eingabe wurde immer auf* 64×64 *neu skaliert, egal welche Eingabegröße vorlag.*

Um Eingaben verschiedener Größe verarbeiten zu können, müssen entweder Pooling-Layer eingesetzt werden, oder die Eingaben vorverarbeitet werden. Naiv könnte man alle Bilder des SVHN Datensatzes neu skalieren. Dies hat jedoch zur Folge, dass kleinere Bilder interpoliert werden müssen und große Bilder Informationen verlieren. Die Neuskalierung ignoriert vorherige Seitenverhältnisse der Eingabe und skaliert ein Bild beispielsweise immer auf 128×128. Große Bilder, in denen die Zahl nur einen kleinen Teil einnimmt, verlieren dadurch wichtige Informationen. Wenn eine bereits kleine Ziffer noch weiter herunter skaliert und eventuell wegen den Seitenverhältnissen verzerrt wird, gehen Informationen der Ziffer verloren. Dem gegenüber würde ein kleines Bild, welches eine Zahl fast vollständig ausfüllt, hoch skaliert werden und Informationen müssten durch Interpolation hinzugefügt werden. Dadurch können Zahlen sehr klein erscheinen oder aber sehr groß, wodurch es erschwert wird, einheitliche Merkmalsfilter anzulernen, die auf allen Elementen des Datensatzes eine gute Genauigkeit erreichen. Die Ergebnisse dieser Methode sind in

SVHN (Aug)	E_{all}	E_{digit}	E_{len}
O	22.72	6.11	3.55
N	15.56	4.13	3.19
M	17.16	4.53	3
CNN [JSZK15]		4.0	
ST-CNN Single [JSZK15]		3.7	
ST-CNN Multi [JSZK15]		**3.6**	
Maxout CNN [GBI+14]		4.0	
DRAM [BMK15]		3.9	

Tabelle 5.3.2: *Vergleich zwischen* O, N, M *und Netzwerken verwandter Arbeiten.* ***Aug****: Die Augmentierung wurde mit den Hyperparamtern Expand* $= 30\%$, *Resize* $= (64 \times 64)$, *Crop-Size* $= (54 \times 54)$, *Edge-Enhancement* $= (0, 0)$ *durchgeführt.*

Tabelle 5.3.1 dargestellt. Trotz der Verwendung von STNs ist die Erkennungsrate nicht sehr gut und beträgt 78.39% (Fehlerrate von 22.61%). Augmentierung würde diesen wechselnden Unterschied zwischen der Zahlengröße und der Bildgröße eliminieren, da immer die minimale Boundingbox um die Zahl um einen festen Prozentsatz vergrößert wird. Daher ist das Verhältnis zwischen der Größe der Zahl und des augmentierten Eingabebildes immer sehr ähnlich.

Zunächst wird das Vorgehen aus [JSZK15], [GBI+14] und [BMK15] verwendet und in PyTorch ([PGC+17]) bzw. Python implementiert. Dazu wird wie in Abschnitt 3.3 beschrieben die Augmentierung realisiert und die minimale Boundingbox mithilfe der Annotationen des Datensatzes berechnet und um einen festen Prozentsatz erweitert. Hier wurde analog zu [GBI+14] eine Erweiterung um 30% in der Breite und Höhe vorgenommen. Anschließend wird dieses resultierende Bild auf eine feste Dimension von 64×64 neu skaliert, woraufhin zufällige 54×54 Regionen aus diesem ausgeschnitten werden. Es ist jedoch nicht weiter beschrieben ob *Crop-Size* oder *Resize* auf 64×64 gesetzt wird. Weitere Experimente haben gezeigt, dass *Resize* $= (92 \times 92)$ und *Crop-Size* $= (78 \times 78)$ sehr ähnliche Ergebnisse erzielen und auch die Fehlerrate 3.6% erreichen (vgl. Tabelle 5.3.3). Es wird hier kein *Edge Enhancement* verwendet, da in [JSZK15] auch keine weiteren fotometrischen Augmentierungen vorgenommen wurden. Damit lassen sich die hier in der Arbeit verwendeten Architekturen mit den aus [JSZK15] vergleichen. Die Ergebnisse davon sind in Tabelle 5.3.2 abgebildet. Da in [JSZK15] nur die Fehlerraten für die Erkennung einzelner Ziffern beschrieben

SVHN (Aug)	M_{wide}			O		
Expand	E_{all}	E_{digit}	E_{len}	E_{all}	E_{digit}	E_{len}
0%	17.31	4.71	3.67	24.73	6.92	6.1
30%	**13.06**	**3.6**	2.58	21.12	5.83	4.42
60%	15.14	4.18	**2.43**	24.55	6.77	4.45
100%	18.6	4.93	2.55	35.16	9.54	4.69
150%	26.5	6.92	2.52	58.2	16.32	6.01
200%	31.67	8.4	3.28	74.91	21.61	6.16

Tabelle 5.3.3: *Gegenüberstellung einer Architektur (M_{wide} bzw. O) bei wechselndem Hyperparamter Expand der Augmentierung.* ***Aug****: Die Augmentierung wurde mit den Hyperparamtern Resize* $= (92 \times 92)$, *Crop-Size* $= (78 \times 78)$, *Edge-Enhancement* $= (0, 0.6)$ *durchgeführt.*

sind, sind die anderen Fehlerraten leer. Die Augmentierung hat, wie erwartet, die Fehlerrate stark senken können, da die Zahl immer eine feste relative Größe bezogen auf das Eingabebild hat und dadurch das Lokalisationsproblem und das Anlernen der relevanten Filter einfacher ist.
Nun wird untersucht, wie die Netzwerke sich bei verändernder „Genauigkeit" der Lokalisierung, also das Variieren des *Expand* Hyperparameters, verhält. Dementsprechend verändert sich auch immer die relative Größe einer Zahl in Bezug zum Eingabebild. Ist z. B. *Expand* auf 200% gesetzt wobei die anderen Hyperparameter gleich bleiben, enthält ein solches Eingabebild deutlich kleinere Ziffern, als bei einer Vergrößerung um nur 50%. Untersucht werden verschiedene Vergrößerungsstufen zwischen 0% und 200%. Diesmal wurde jedoch *Edge Enhancement* verwendet, um eine weitere Ebene an Variation hinzuzufügen, welches zu einer robusteren Genauigkeit führen soll. Zudem wird hier jeweils ausschließlich die SVHN Architektur M_{wide} bzw. O verwendet, da M_{wide} bei Testläufen bessere Ergebnisse als N und M erzielt hat und O zum direkten Vergleich dient. Die Ergebnisse sind in Tabelle 5.3.3 dargestellt. Die Ergebnisse sind ähnlich gut wie die in [JSZK15] veröffentlichten, wenn *Expand* $= 30\%$ gewählt wird. Wie zu erwarten, verschlechtert sich die Erkennungsrate, je höher die Vergrößerung ist. Dies könnte damit zusammenhängen, dass die Bilder nach der Vergrößerung immer auf den festen Wert des *Resize* Hyperparameter neu skaliert werden, wodurch die Zahl einen immer kleineren Teil im Bild einnimmt je höher die Vergrößerung ist. Da die Architektur konstant bleibt, ist die *Kernel-Size* der Faltungsschichten immer konstant. Daher werden die Merkmalsfilter auch immer für eine gleich große lokale Umgebung

angelernt, die dadurch bei sehr kleinen Zahlen oder zu großen Zahlen keine gute Genauigkeit erreichen. Dies könnte auch erklären, warum die Genauigkeit bei einer Augmentierung ohne jede Vergrößerung schlechter ist, als bei einer Vergrößerung um 30%. Die Länge der gesamten Zahl kann hingegen bei fast allen Testszenarien gleich gut bestimmt werden. Der Effekt eines STN wird hier besonders bei größeren Werten für den Hyperparameter *Expand* deutlich. Es scheint, wenn die Ziffer in einer Eingabe nur einen geringen Teil einnimmt, ist es umso wichtiger ein STN einzusetzen. Bei geringen Vergrößerungen der Boundingbox haben die beiden Netzwerke noch eine Genauigkeit in einer gleichen Größenordnung, jedoch wird dieser Unterschied immer größer, je höher *Expand* gewählt wird. Im Falle von 200% beträgt der absolute Unterschied 13.21% bzw. relativ rund 157%, wohingegen dieser Unterschied im Falle von 30% auf absolut 2.23% bzw. relativ 61.94% sinkt.

5.3.2 *Augmentierung und Pooling*

SVHN (Aug)	*Expand* = 60%			*Expand* = 100%		
	E_{all}	E_{digit}	E_{len}	E_{all}	E_{digit}	E_{len}
M_{wide}	15.14	4.18	**2.43**	18.6	4.93	**2.55**
N (SPP-Even)	17.17	4.4	3.17	18.26	4.81	5.38
N (SPP-Prim)	**15.05**	**3.91**	2.95	**18.09**	**4.67**	3.64

Tabelle 5.3.4: *Vergleich zwischen der Verwendung von Vorverarbeitung und ohne jegliche Vorverarbeitung, jedoch Pooling-Layer mit zwei verschiedenen Levelmengen.* ***Aug:*** *Die Augmentierung wurde mit den Hyperparamtern für* M_{wide} *mit Resize* = (92 × 92), *Crop-Size* = (78 × 78), *Edge-Enhancement* = (0, 0.6) *und für* N *mit Resize* = /, *Crop-Size* = /, *Edge-Enhancement* = (0, 0.6) *durchgeführt*

Da bei dem vorherigen Vorgehen abhängig vom *Resize* Hyperparameter ein Element des Datensatzes neu skaliert wird und dadurch eventuell Informationen verloren gehen und Verzerrungen hinzugefügt werden, wird nun untersucht, wie sich ein Netzwerk verhält, wenn man diese Vorverarbeitung nicht vornimmt. Dies bedeutet dann, dass zwar die Augmentierung nach dem Hyperparameter *Expand* durchgeführt wird, jedoch keine zufälligen Bereiche ausgeschnitten werden und keine Neuskalierung stattfindet. Damit ein Netzwerk diese variablen Eingaben verarbeiten kann, müssen spezielle Pooling-Layer (vgl. Abschnitt 2.2.2) eingesetzt werden. Konkret wird die SVHN Architektur N untersucht, wobei STN(A_{spp}) in der ersten Schicht verwendet

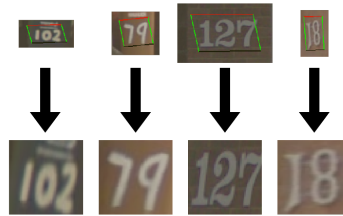

Abbildung 5.3.1: *Die obere Reihe stellt die Eingabe des Netzwerks* N *(SPP-Prim) dar und die untere die Ausgabe der ersten STN Schicht, welche hier mit Spatial Pyramid Pooling kombiniert ist.*

wird, statt $\mathrm{STN}(A)$. Dies führt dazu, dass die erste Schicht immer ein Bild der Größe (54×54) ausgibt, egal welche Dimension die Eingabe besitzt. Zudem wird untersucht, wie sich unterschiedliche Level des Spatial Pyramid Poolings auswirken. Zum einen werden Vielfache von 2 als Level verwendet (SPP-Even $= \{16, 12, 8, 4, 2, 1\}$) und zum Anderen ausschließlich Primzahlen (SPP-Prim $= \{17, 11, 7, 5, 3, 1\}$). Es wurde darauf geachtet, dass die dadurch resultierenden Vektoren eine ähnliche Länge haben (485 bzw. 494). Die Intention ist zu untersuchen, ob Vielfache von 2 eine schlechtere Fehlerrate erzielen, da z. B. ein Spatial Pyramid Pooling mit Level 4 eine feinere Auflösung des Level 2 ist. Daher könnte nur das Level 16 verwendet werden und durch mehrmaliges zusammenfassen von einzelnen Regionen dadurch auf die restlichen Level geschlossen werden (12 bildet dabei die einzige Ausnahme). Demgegenüber tritt dieses Problem bei der Verwendung von Primzahlen als verschiedene Level nicht auf. Dort könnte nur jeweils das Level 1 durch jeweils alle anderen ausgedrückt werden. Ein weitere Problematik im Verlauf des Lernprozesses bei Netzwerken, die mit beliebigen Dimensionen umgehen können, ist, dass ein Forward-Pass nicht in Batches berechnet werden kann, da in PyTorch die Eingaben in eine große 4-Dimensionale Matrix (Tensor) lädt, welche eine feste Größe besitzen muss ($B \times K \times W \times H$ mit Batchsize B, (Farb-)Kanälen K, Breite W und Höhe H). Ändert sich innerhalb einer Batch also die Größe eines Elements, kann dies mit PyTorch nicht als Batch berechnet werden. Daher wird die Batchsize auf 1 gesetzt, wodurch die Parallelisierung innerhalb der Grafikkarte nicht optimal ausgenutzt werden kann und der Lernprozess sich verlängert. Die Ergebnisse sind in Tabelle 5.3.4 dargestellt und zeigen, dass die Vermutung, zumindest in diesem kleinen Umfang, bestätigt wird. Bemerkenswert ist auch, dass das Netzwerk N (SPP-Prim) das größere Netzwerk M_{wide} übertrifft. Zudem erreicht N (SPP-Prim) eine bessere Genauigkeit als N ohne Pooling und mit nur 30% *Expand* (vgl. Tabelle 5.3.2). Die Verarbeitung einiger Bilder verschiedener Dimension mithilfe des N (SPP-Prim) Netzwerks ist in Abbildung 5.3.1 visualisiert.

5.3.3 *Original und Pooling*

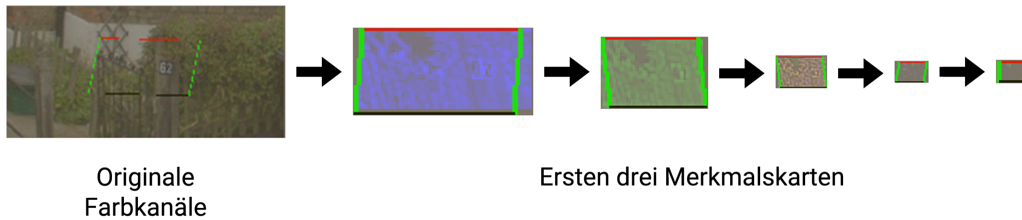

Abbildung 5.3.2: *Eine Beispieleingabe in das M_{pool} Netzwerk. Es wird jeweils vor jeder Transformation eines STN das derzeitige Bild visualisiert. Zunächst stehen 3 Farbkanäle zur Verfügung, die im weiteren Verlauf durch die ersten 3 Merkmalskarten ausgestauscht werden. Zu erkennen sind die immer grobauflösenden Bilder, da im Laufe des Architektur immer mehr Poolingschichten angewandt werden.*

Jetzt wird neben der Vorverarbeitung auch die Augmentierung entfernt und das Netzwerk wird auf den grundlegenden Daten des SVHN Datensatzes angelernt bzw. getestet. Hier ist zu erwarten, dass ein solches Netzwerk schlechtere Ergebnisse erzielt, da keinerlei Vorverarbeitung durchgeführt wird und eine Zahl deutlich unterschiedlich viel Platz im Eingabebild einnimmt. Auch hier ist analog zu den Netzwerken des vorherigen Abschnitts die Batchsize auf 1 gesetzt, da die Elemente des Datensatzes unterschiedliche Dimensionen besitzen. Die Intention hierbei ist über mehrere Faltungsschichten hinweg Spatial Transformer Networks kombiniert mit Poolingsmethoden einzusetzen um die Ziffer in den Originalbildern zu lokalisieren und die korrekte Boundingbox um die Zahl zu bestimmen. M_{pool} verwendet dafür einerseits das Spatial Pyramid Pooling sowie das Temporal Pyramid Pooling (vgl. Abschnitt 2.2.2). Die Evaluation hat gezeigt, dass die erste STN-SPP Schicht die Zahl zunächst grob lokalisiert. Die darauffolgenden STNs lokalisieren die Zahl nun auf Grundlage von Merkmalskarten immer feiner und bestimmen dadurch immer besser die Boundingbox. 2 Beispielbilder der Testpartition und dessen weiteren Verarbeitung sind in Abbildung 5.3.2 abgebildet. Wie zu erkennen, muss der erste STN bereits sehr gute Ergebnisse erzielen, da die darauffolgenden bei diesen Beispielen keine großen Transformationen vornehmen. Dies ist jedoch bei anderen Eingabebildern anders, sodass bei dem 5. und 6. STN noch auffällige Transformationen zu erkennen sind. Die Ergebnisse sind in Tabelle 5.3.5 dargestellt und zeigen eine mittelmäßige Genauigkeit im Vergleich zu den vorherigen Experimenten. Dies ist jedoch zu erwarten, da die

SVHN (Original)	E_{all}	E_{digit}	E_{len}
M_{pool}	35.78	12.27	17.75

Tabelle 5.3.5: *Ergebnisse des M_{pool} Netzwerks, welches mit den originalen Bildern des SVHN Datensatzes angelernt und getestet wurde.*

Bilder viel mehr Hintergrund und stellenweise sogar mehrere mehrstellige Zahlen an verschiedenen Orten enthalten. Dennoch zeigt dies eine vielversprechende Basis, auf der aufgebaut werden kann, um die Genauigkeit noch weiter zu steigern.

5.3.4 *Diffeomorphe Transformationen*

Abbildung 5.3.3: *2 Beispielhafte Diffeomorphe Transformationen, die von dem DTN nach dem 2. STN in der Architektur M_{wide} bestimmt wurde.*

Schließlich wird untersucht, welchen Effekt DTNs in Kombination mit STNs haben (vgl. Abschnitt 3.2). Wie [DFH18] darstellt, ist die Kombination von DTN und STN besser als die isolierte Nutzung einer der beiden Ansätze. Der Augmented SVHN Datensatz mit den Hyperparamtern *Resize* $= (92 \times 92)$, *Crop-Size* $= (78 \times 78)$, *Edge-Enhancement* $= (0, 0.6)$ und wechselndem *Expand* dient dem Anlernen und Testen des Netzwerks. Es wird nach allen STNs in M_{wide} ein weiteres DTN hinzugefügt, wodurch die Architektur $M_{wide+dtn}$ definiert wird. Die Ergebnisse, welche in Tabelle 5.3.6 abgetragen sind, bestätigen zum Teil die Ergebnisse aus [DFH18].

SVHN (Aug)	M_{wide}			$M_{wide+dtn}$		
Expand	E_{all}	E_{digit}	E_{len}	E_{all}	E_{digit}	E_{len}
0%	17.31	4.71	3.67	17.82	4.79	3.49
30%	**13.06**	**3.6**	2.58	13.73	3.85	2.7
60%	15.14	4.18	**2.43**	13.46	3.71	2.46
100%	18.6	4.93	2.55	16.29	4.38	2.25
150%	26.5	6.92	2.52	22.96	6.01	2.4
200%	31.67	8.4	3.28	28.43	7.56	3.09

Tabelle 5.3.6: *Gegenüberstellung zweier Architekturen (M_{wide} bzw. $M_{wide+dtn}$) unter wechselndem Hyperparamter Expand bei der Augmentierung.* ***Aug:*** *Die Augmentierung wurde mit den Hyperparamtern Resize* $= (92 \times 92)$, *Crop-Size* $= (78 \times 78)$, *Edge-Enhancement* $= (0, 0.6)$ *durchgeführt.*

Beispielhaft ist der Effekt einiger DTNs innerhalb des Netzwerks in Abbildung 5.3.3 abgebildet. Zu erkennen sind verschiedene Transformationen, die über die Klasse der affinen Transformationen hinaus geht. So kann die Zahl z. B. in der Mitte „zusammengepresst“ und an den Seiten „auseinandergezogen“ werden. Zudem unterliegt das Netzwerk mit DTN in manchen Fällen dem Netzwerk ohne DTN. Je höher jedoch die Menge an Hintergrund, und damit auch der Transformationsaufwand, im Bild ist, desto wertvoller scheint ein DTN zu sein, da ab einer Vergrößerung der Boundingbox um mindestens 60% $M_{wide+dtn}$ immer eine bessere Genauigkeit erzielt als M_{wide}. Analog zu den Versuchen mit DTNs in Abschnitt 5.2 übernimmt auch hier ein DTN annähernd die gesamte Transformation und der STN bestimmt meist nur minimale Transformationen. Es ist daher auch zu erwarten, dass ein Netzwerk welches nur DTNs statt STNs erneut leicht schlechtere Ergebnisse erreicht.

6 FAZIT

Convolutional Neural Networks sind derzeit relativ unangefochten die state-of-the-art Architektur, um einige Probleme der Mustererkennung gut annähernd zu lösen. CNNs sind Ende zu Ende trainierbar, wodurch eine gewisse Eingabe auf eine gewünschte Ausgabe (Klasse) abgebildet wird und unbekannte Eingaben generalisiert werden. Ferner kann darauf aufgebaut und weiterführende Module verwendet werden, um z. B. auf den Daten Transformationen durchzuführen oder mit unterschiedlichen Längen der Eingabe umgehen zu können. Dadurch können die Grundidee der Merkmalsidentifikation und dessen Ausprägungskarte (Ausprägung eines Merkmals an einer bestimmten Stelle in der Eingabe) mit daraus resultierenden Berechnungen kombiniert werden. So nutzen Spatial und Diffeomorphe Transformer Networks diese Informationen, um die Parameter einer Transformation zu bestimmen oder die Informationen werden mithilfe von Spatial bzw. Temproal Pyramid Pooling zu einer konstanten Repräsentation reduziert. Abseits davon werden CNNs abgewandelt, um unter Anderem Residual CNNs zu definieren (vgl. Abschnitt 3). Sie bilden also eine grundlegende Struktur und Idee auf der aufgebaut sowie welche abgewandelt werden kann, um bezüglich der Problemdomäne bessere Architekturen zu definieren.
Spatial Transformer Networks bieten eine einfache Möglichkeit in ein bestehendes Netzwerk Transformationen auf den zu verarbeitenden Daten anzuwenden, die wegen ihrer Struktur auch parallel zum Rest des Netzwerks angelernt werden kann. Sie sind eine Erweiterung der CNN Architektur und können allein nicht angelernt werden und werden deswegen immer in bestehende Netzwerke eingesetzt. Ohne weitere Änderungen können diese jedoch nur affine Transformationen durchführen, welche nicht immer die optimale Wahl für das vorliegende Problem, beispielsweise die Gesichtserkennung (vgl. Abschnitt 3.2), sind. STNs erlauben also die explizite Annahmen an Transformationen in die Modellarchitektur zu integrieren mit dem Ziel diese auszugleichen. Diffeomorphe Transformer Networks bauen auf diese Idee auf, führen aber eine CPA-B Transformation durch. Dadurch können Eingaben „flüssig“ verformt werden, wodurch in Kombination mit anderen Transformationen oftmals bessere Ergebnisse erzielt als nur eine Technik isoliert. Jedoch sind diese nicht in allen Situationen pauschal zielführend. So kann, wie in Tabelle 5.3.6 dargestellt, die Kombination beider schlechter performen.
Allgemein lässt sich aus Tabelle 5.2.1 folgern, dass Transformer Networks erst einen

positiven Effekt haben, wenn auch Verzerrungen im Datensatz auftauchen, welche rückgängig gemacht werden sollen. Daher muss bezüglich der Problemdomäne evaluiert werden, welche — und ob — Transformer Networks verwendet werden können. Die Experimente haben gezeigt, dass die Erkennung einer einzelnen Ziffer mit sehr geringen Fehlerraten möglich ist, diese jedoch höher wird, wenn mehrere Ziffern erkannt werden soll, welche in einem ggf. augmentierten Bild nicht zentriert oder in immer wechselnde Verhältnisse in Bezug zur gesamten Eingabebildgröße vorkommen. Zudem kann mithilfe von Poolingtechniken auf den orignalen Bildern mit sehr viel Hintergrund trotzdem noch im Vergleich zu dem naiven Ansatz bessere Ergebnisse erzielt werden. Im Allgemeinen muss der Datensatz betrachtet und eingeschätzt werden und dementsprechend eine Architektur bestimmt werden. Auf dessen Grundlage sollte schrittweise der Nutze verschiedener weiterführender Techniken getestet werden. Augmentierung hat sich in diesem Zusammenhang als sinnvolle Technik erwiesen um die positionelle Varianz der Eingabedaten zu erhöhen, damit das Netzwerk robuster gegenüber diesen wird und auf unbekannten Daten bessere Genauigkeiten erzielt. Zunächst wurde gezeigt, dass ein CNN inklusive zweier Spatial Transformer Networks sehr gute Ergebnisse auf dem MNIST Datensatz erreicht, welche sogar besser wurden, wenn der Datensatz verschiedene Transformationen enthält. Danach wurde die gleiche Architektur auf dem SVHN Format 2 Datensatz trainiert, wobei dort im direkten Vergleich deutlich schlechtere Fehlerraten erreicht wurden, da hier Teile von anderen Zahlen und mehrere Farbkanäle in der Eingabe vorhanden sind, sodass es unter Umständen selbst für eine Person nicht immer eindeutig ist, welche Ziffer nun die „korrekte“ Annotation ist. Daraufhin wurde untersucht, wie eine Architektur aussehen könnte, die auf dem SVHN Format 1 Datensatz mit ggf. Vorverarbeitung und Augmentierung trainiert wird und auch bisherige state-of-the-art Performance aus [JSZK15] erreicht. Dabei wurde der Effekt der Augmentierung und Vorverarbeitung untersucht und diese immer weiter vereinfacht, bis schließlich nur der originale Datensatz übrig war. Dabei erwies sich eine Augmentierung mit einer Bildvergrößerung der minimalen Boundingbox um 30% als bester Fall. Bei höheren Werten ist ein Netzwerk, welches eine Kombination aus Spatial und Diffeomorphen Transformer verwendet auf diesem Datensatz besser als eines, das nur STNs verwendet. Bei geringen Werten scheint dieser Effekt nicht aufzutreten und erreicht nur ähnliche bzw. leicht schlechtere Ergebnisse. Abschließend lässt sich sagen, dass CNNs einen geeigneten Anfang in die Mustererkennung im Bereich von Bilddaten bieten, da sie sehr ausbaufähig sind und auf simplen Datensätzen bereits sehr gute Ergebnisse erzielen. STNs und DTNs sind beides vielversprechende Module die bei passenden Problemdomänen durchaus die Fehlerrate verbessern.

[AAD+19] Arnold, E. ; Al-Jarrah, O. Y. ; Dianati, M. ; Fallah, S. ; Oxtoby, D. ; Mouzakitis, A.: A Survey on 3D Object Detection Methods for Autonomous Driving Applications. In: *IEEE Transactions on Intelligent Transportation Systems* (2019), S. 1–14. – ISSN 1524–9050

[BHWM12] Burg, Klemens ; Haf, Herbert ; Wille, Friedrich ; Meister, Andreas: *Vektoranalysis*. 2. 2012. – 115 S. – ISBN 978–3–8348–1851–5

[BHYM17] Bartz, Christian ; Herold, Tom ; Yang, Haojin ; Meinel, Christoph: Language Identification Using Deep Convolutional Recurrent Neural Networks. In: *Neural Information Processing - 24th International Conference, ICONIP 2017, Guangzhou, China, November 14-18, 2017, Proceedings, Part VI*, 2017, S. 880–889

[BMK15] Ba, Jimmy ; Mnih, Volodymyr ; Kavukcuoglu, Koray: Multiple Object Recognition with Visual Attention. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015

[Bra00] Bracewell, Ronald: The Fourier Transform and Its Applications (3rd ed.). (2000), S. 26–27. ISBN 0–07–303938–1

[CSG18] Cohen, Gilad ; Sapiro, Guillermo ; Giryes, Raja: DNN or k-NN: That is the Generalize vs. Memorize Question. In: *CoRR* abs/1805.06822 (2018)

[DFH18] Detlefsen, Nicki S. ; Freifeld, Oren ; Hauberg, Søren: Deep Diffeomorphic Transformer Networks. In: *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, 2018, S. 4403–4412

[DK16] Druzhkov, P. N. ; Kustikova, V. D.: A survey of deep learning methods and software tools for image classification and object detection. In: *Pattern Recognition and Image Analysis* 26 (2016), Jan, Nr. 1, S. 9–15. – ISSN 1555–6212

[DT05] DALAL, Navneet ; TRIGGS, Bill: Histograms of Oriented Gradients for Human Detection. In: *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. Washington, DC, USA : IEEE Computer Society, 2005. – ISBN 0–7695–2372–2, S. 886–893

[Fon19] FONTICONS, Inc.: *Font Awesome Free*. https://fontawesome.com/license. Version: Jun 2019

[GBC16] GOODFELLOW, Ian J. ; BENGIO, Yoshua ; COURVILLE, Aaron C.: *Deep Learning*. MIT Press, 2016 (Adaptive computation and machine learning). http://www.deeplearningbook.org/. – ISBN 978–0–262–03561–3

[GBI+14] GOODFELLOW, Ian J. ; BULATOV, Yaroslav ; IBARZ, Julian ; ARNOUD, Sacha ; SHET, Vinay D.: Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks. In: *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014

[Gir15] GIRSHICK, Ross B.: Fast R-CNN. In: *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015*, 2015, S. 1440–1448

[Hay98] HAYKIN, Simon: *Neural Networks: A Comprehensive Foundation*. 2nd. Upper Saddle River, NJ, USA : Prentice Hall PTR, 1998. – ISBN 0132733501

[Hor91] HORNIK, Kurt: Approximation capabilities of multilayer feedforward networks. In: *Neural Networks* 4 (1991), Nr. 2, S. 251–257

[HZRS14] HE, Kaiming ; ZHANG, Xiangyu ; REN, Shaoqing ; SUN, Jian: Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition. In: *Computer Vision - ECCV 2014 - 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part III*, 2014, S. 346–361

[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*, 2016, S. 770–778

[IIN17] ISLAM, Noman ; ISLAM, Zeeshan ; NOOR, Nazia: A Survey on Optical Character Recognition System. In: *CoRR* abs/1710.05703 (2017)

[IS15] Ioffe, Sergey ; Szegedy, Christian: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In: *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, 2015, S. 448–456

[JSZK15] Jaderberg, Max ; Simonyan, Karen ; Zisserman, Andrew ; Kavukcuoglu, Koray: Spatial Transformer Networks. In: *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, 2015, S. 2017–2025

[KSH12] Krizhevsky, Alex ; Sutskever, Ilya ; Hinton, Geoffrey E.: ImageNet Classification with Deep Convolutional Neural Networks. In: *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States.*, 2012, S. 1106–1114

[KSZQ19] Khan, Asifullah ; Sohail, Anabia ; Zahoora, Umme ; Qureshi, Aqsa S.: A Survey of the Recent Architectures of Deep Convolutional Neural Networks. In: *CoRR* abs/1901.06032 (2019)

[LBBH98] LeCun, Y. ; Bottou, L. ; Bengio, Y. ; Haffner, P.: Gradient-Based Learning Applied to Document Recognition. In: *Proceedings of the IEEE* 86 (1998), November, Nr. 11, S. 2278–2324

[LC10] LeCun, Yann ; Cortes, Corinna: MNIST handwritten digit database. (2010). http://yann.lecun.com/exdb/mnist/

[LCW+16] Liu, Wei ; Chen, Chaofeng ; Wong, Kwan-Yee K. ; Su, Zhizhong ; Han, Junyu: STAR-Net: A SpaTial Attention Residue Network for Scene Text Recognition. In: *Proceedings of the British Machine Vision Conference 2016, BMVC 2016, York, UK, September 19-22, 2016*, 2016

[LHY18] Long, Shangbang ; He, Xin ; Yao, Cong: Scene Text Detection and Recognition: The Deep Learning Era. In: *CoRR* abs/1811.04256 (2018)

[LKJ16] Ly, Fei-Fei ; Karpathy, Andrej ; Johnson, Justin: *CS231n: Convolutional Neural Networks for Visual Recognition*. http://cs231n.stanford.edu/, 2016

[LLY+18] Liu, Xuebo ; Liang, Ding ; Yan, Shi ; Chen, Dagui ; Qiao, Yu ; Yan, Junjie: FOTS: Fast Oriented Text Spotting With a Unified Network. In: *2018 IEEE*

Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018, 2018, S. 5676–5685

[LPW+17] Lu, Zhou ; Pu, Hongming ; Wang, Feicheng ; Hu, Zhiqiang ; Wang, Liwei: The Expressive Power of Neural Networks: A View from the Width. In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, 2017, S. 6231–6239

[Mil07] Million, Elizabeth: The Hadamard Product. (2007), Apr. http://buzzard.ups.edu/courses/2007spring/projects/million-paper.pdf

[Nie83] Niemann, H.: Klassifikation von Mustern. (1983), S. 13–14. ISBN 3–540–12642–2

[Nie15] Nielsen, Michael A.: Neural Networks and Deep Learning. (2015)

[Nil96] Nilsson, Nils J.: *Introduction to Machine Learning. An early draft of a proposed textbook*. 1996

[NWC+11] Netzer, Yuval ; Wang, Tao ; Coates, Adam ; Bissacco, Alessandro ; Wu, Bo ; Ng, Andrew Y.: Reading Digits in Natural Images with Unsupervised Feature Learning. In: *NIPS Workshop on Deep Learning and Unsupervised Feature Learning* (2011)

[PGC+17] Paszke, Adam ; Gross, Sam ; Chintala, Soumith ; Chanan, Gregory ; Yang, Edward ; DeVito, Zachary ; Lin, Zeming ; Desmaison, Alban ; Antiga, Luca ; Lerer, Adam: Automatic Differentiation in PyTorch. In: *NIPS Autodiff Workshop*, 2017

[PMR+16] Priya, A. ; Mishra, S. ; Raj, S. ; Mandal, S. ; Datta, S.: Online and offline character recognition: A survey. In: *2016 International Conference on Communication and Signal Processing (ICCSP)*, 2016, S. 0967–0970

[PT15] Patel, Monica ; Thakkar, Shital P.: Handwritten character recognition in english: a survey. In: *International Journal of Advanced Research in Computer and Communication Engineering* Bd. 4, 2015, S. 345–350

[RHW86] Rummelhart, David E. ; Hinton, Geoffrey E. ; Williams, Ronald J.: Learning representations by backpropagating errors. In: *Nature 323* (1986), S. 553–536

[SF17] SUDHOLT, Sebastian ; FINK, Gernot A.: Evaluating Word String Embeddings and Loss Functions for CNN-Based Word Spotting. In: *14th IAPR International Conference on Document Analysis and Recognition, ICDAR 2017, Kyoto, Japan, November 9-15, 2017*, 2017, S. 493–498

[SHK+14] SRIVASTAVA, Nitish ; HINTON, Geoffrey ; KRIZHEVSKY, Alex ; SUTSKEVER, Ilya ; SALAKHUTDINOV, Ruslan: Dropout: A Simple Way to Prevent Neural Networks from Overfitting. In: *Journal of Machine Learning Research* 15 (2014), 1929–1958. http://jmlr.org/papers/v15/srivastava14a.html

[SIVA17] SZEGEDY, Christian ; IOFFE, Sergey ; VANHOUCKE, Vincent ; ALEMI, Alexander A.: Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, 2017, S. 4278–4284

[SLJ+15] SZEGEDY, Christian ; LIU, Wei ; JIA, Yangqing ; SERMANET, Pierre ; REED, Scott E. ; ANGUELOV, Dragomir ; ERHAN, Dumitru ; VANHOUCKE, Vincent ; RABINOVICH, Andrew: Going deeper with convolutions. In: *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, 2015, S. 1–9

[STIM18] SANTURKAR, Shibani ; TSIPRAS, Dimitris ; ILYAS, Andrew ; MADRY, Aleksander: How Does Batch Normalization Help Optimization? In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada.*, 2018, S. 2488–2498

[SZ15] SIMONYAN, Karen ; ZISSERMAN, Andrew: Very Deep Convolutional Networks for Large-Scale Image Recognition. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015

[TN17] TAYLOR, Luke ; NITSCHKE, Geoff: Improving Deep Learning using Generic Data Augmentation. In: *CoRR* abs/1708.06020 (2017)

[Wan14] WANG, Kai: The Street View Text Dataset (SVT). (2014). http://tc11.cvc.uab.es/datasets/SVT_1

[WGSM16] WONG, Sebastien C. ; GATT, Adam ; STAMATESCU, Victor ; MCDONNELL, Mark D.: Understanding Data Augmentation for Classification: When

to Warp? In: *2016 International Conference on Digital Image Computing: Techniques and Applications, DICTA 2016, Gold Coast, Australia, November 30 - December 2, 2016*, 2016, S. 1–6

[WZZ+13] Wan, Li ; Zeiler, Matthew D. ; Zhang, Sixin ; LeCun, Yann ; Fergus, Rob: Regularization of Neural Networks using DropConnect. In: *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, 2013, S. 1058–1066

[YD15] Ye, Q. ; Doermann, D.: Text Detection and Recognition in Imagery: A Survey. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 37 (2015), July, Nr. 7, S. 1480–1500. – ISSN 0162–8828