Hochschule Kempten
University of Applied Sciences

Masterarbeit

Studiengang Angewandte Informatik
Master of Science

Theoretische Ausarbeitung und praktische Anwendung von Capsule Networks im Bereich des Deep Learning

Bonifaz Stuhr

Matrikelnummer
283166

Aufgabensteller/Prüfer	Prof. Dr. Jürgen Brauer
Arbeit vorgelegt am	28.09.2018
durchgeführt in der	Fakultät Informatik
Anschrift des Verfassers	Kulmuswiese 6, 87480 Weitnau
	boni.stuhr@hotmail.com, Tel. 08375 503

Abstract

In einem Capsule Network (kurz CapsNet) werden Capsules gruppiert und i.d.R. über Verbindungen, die einem Routing-Algorithmus unterliegen, feedforward miteinander verknüpft. Diese Capsules stellen Gruppen von Neuronen dar, welche in ihren Ausgaben verschiedene Eigenschaften derselben Entität codieren, wobei der Routing-Algorithmus dafür sorgt, dass die Entitäten in einen bedeutungsvollen Weg miteinander verbunden werden. Demnach beschreibt ein CapsNet eine Vielzahl verschiedener Entitäten und deren Beziehungen zueinander, womit typische Aufgaben des Deep Learning wie die Klassifizierung von Objekten mit vielversprechenden Ergebnissen nachvollziehbarer gelernt werden können. In dieser Arbeit findet zunächst die ausführliche theoretische Erörterung von CapsNets chronologisch und gemäß der definierten Grundidee statt, worauf die konstruktive Betrachtung der Literatur aktuelle Erkenntnisse zusammengefasst darstellt. Im praktischen Teil der Arbeit wird dann 1) durch Experimente bestätigt, dass der reine, originale Routing-Algorithmus durch Layer mit einer einzelnen Capsule ausgehebelt wird und dem entgegenwirkende Lösungsvorschläge diskutiert; 2) flache CapsNets auf den Kdef-Datensatz für Facial-Emotion-Recognition mit guten Ergebnissen angewandt und mittels vorgestellter tieferer Modelle eine Performance nahe dem State-of-the-Art erreicht, wobei die Auswirkung verschiedener Änderungen an tiefen Netzen aufgezeigt und die mehrdeutigen Ergebnisse bei Nutzung eines Deconvolution-Rekonstruktions-Modules diskutiert werden; 3) ein tiefes CapsNet mittels eines unüberwachten Lernverfahrens auf Kdef trainiert und die negativen Auswirkungen von Eingaben mit kleinen ROIs bezüglich der Performance dargelegt.

Vorwort

Die vorliegende Arbeit behandelt ausführlich ein aktuelles, theoretisches Modell des Deep Learning, welches durch die Kombination von Erkenntnissen dieses Bereichs mit weiteren neurowissenschaftlichen Errungenschaften und Hypothesen entstand und u.a. durch Maschine-Learning-Methoden in der Forschung mehrfach praktisch umgesetzt wurde. Daher gewährte mir die theoretische Erörterung dieses Modells, sowie dessen praktischer Einsatz, einen tiefen Einblick in die wissenschaftliche Vorgehensweise bei der Entwicklung neuer Modelle basierend auf vorher erarbeiteten theoretischen Ergebnissen, was eine detaillierte Sicht auf die „Bleeding Edge“ des Deep Learning Bereichs ermöglichte. Diese Einsichten stellen die initialen Gründe für die Wahl dieser Thematik dar, welche ich durch die Unterstützung von Prof. Dr. Jürgen Brauer gründlich und bedacht festlegen konnte. Für die Möglichkeit eine eigene Aufgabenstellung in einem selbstbestimmten Rahmen zu erarbeiten und für die genügend eingeräumte Zeit für Fragen und Anregungen gilt daher meinem Betreuer Prof. Dr. Jürgen Brauer mein besonderer Dank. Einen herzlichen Dank möchte ich ebenfalls meinen Korrekturlesern - insbesondere Anja Klenk - aussprechen, die reichlich Zeit und Geduld aufbrachten, um diese theoretische und daher textreiche Arbeit auf Fehler unterschiedlichster Art zu prüfen. Zuletzt möchte ich mich bei meinen Arbeits(platz)kollegen von der Stuhr GmbH für das angenehme Arbeitsklima bedanken.

Inhaltsverzeichnis

Abstract I

Vorwort II

Inhaltsverzeichnis III

1 Motivation 1

2 Einleitung 2

3 Theoretische Erörterung von Capsule Networks 3

3.1 Generalisierte Idee . . . 3
3.1.1 Capsules . . . 3
3.1.2 Topologie eines Capsule Networks . . . 6
3.1.3 Routing-Algorithmus eines Capsule Networks . . . 7
3.1.4 Weitere biologische Inspirationen und Parallelen . . . 12
3.1.5 Introspektive Experimente zur Intuition hinter Capsules . . . 18
3.2 Transforming-Autoencoder . . . 21
3.2.1 Schematischer Aufbau eines Transforming-Autoencoders . . . 21
3.2.2 Anwendung des Transforming-Autoencoders . . . 22
3.2.3 Transforming-Autoencoder in höheren Dimensionen . . . 24
3.2.4 Einige Erkenntnisse durch Transforming-Autoencoder . . . 26
3.3 Capsule Network . . . 28
3.3.1 Schematischer Aufbau einer Capsule mit Iterative Routing-by-Agreement 28
3.3.2 Dynamic Routing-by-Agreement zwischen Capsules . . . 31
3.3.3 Architektur eines Capsule Networks . . . 36
3.3.4 Loss und Training . . . 38
3.3.5 Anwendung des Capsule Networks . . . 42
3.3.6 Erkenntnisse durch Capsule Networks . . . 45
3.4 Matrix Capsules mit Expectation-Maximization-Routing . . . 46
3.4.1 Wieso Matrix Capsules? . . . 47
3.4.2 Schematischer Aufbau einer Matrix Capsule mit EM-Routing . . . 48
3.4.3 Expectation-Maximization für das Routing zwischen Capsules . . . 49
3.4.4 Architektur des Matrix Capsule Networks . . . 65
3.4.5 Loss und Training . . . 67
3.4.6 Anwendung des Matrix Capsule Networks . . . 68
3.4.7 Erkenntnisse durch Matrix Capsule Networks . . . 74

3.5 Weitere Capsule-Modelle . . . 75
3.5.1 Spectral Capsule Networks . . . 75
3.5.2 CapsuleGAN . . . 79
3.5.3 RNN-Capsule Networks . . . 82
3.5.4 SegCaps . . . 84
3.5.5 VideoCapsuleNet . . . 88
3.6 Ein weiterer Routing-Algorithmus: Eine Optimierungssicht auf das dynamische Routing zwischen Capsules . . . 89
3.7 Erweiterungen für Capsule Networks . . . 94
3.7.1 Non-Parametric Transformation Networks mit Capsule Networks . . . 94
3.7.2 Schnell konvergierendes Capsule Network mit Anwendung auf MNIST 96
3.7.3 Dichte und vielfältige Capsule Networks . . . 97
3.8 Verschiedene Klassifizierungen durch Capsule Networks . . . 99
3.8.1 Gehirntumor-Typ Klassifizierungen mit Capsule Networks . . . 99
3.8.2 Bildfusion der Fehlererkennung in Stromversorgungssystemen basierend auf Deep Learning . . . 101
3.8.3 Die Klassifizierung von UAV-Reisbildern basierend auf Capsule Networks 102
3.8.4 HSI-CapsNet . . . 104
3.8.5 Verkehrszeichenerkennung mit Capsule Networks . . . 106
3.8.6 FACSCaps: Pose-unabhängiges Facial-Action-Coding mit Capsules . . . 106
3.9 Weiteres NLP mit Capsule Networks . . . 108
3.10 Auswertung und Erklärbarkeit von Capsule Networks . . . 110
3.10.1 Capsule Network Performance für komplexe Daten . . . 110
3.10.2 Verbesserte Erklärbarkeit von Capsule Networks: Relevance-Path durch Agreement . . . 111
3.10.3 Weitere Auswertungen . . . 112
3.11 Tabellarische Zusammenstellung der restlichen gefundenen wissenschaftlichen Literatur . . . 114
3.12 Capsule Networks im Vergleich mit CNNs . . . 114

4 Praktische Anwendung von Capsule Networks 118
4.1 Aufgabenstellung . . . 118
4.2 Entwicklungsumfeld und Vorgehensweise . . . 119
4.3 Coupling-Koeffizienten-Test für originale CapsNets mit einer ClassCaps . . . 120
4.3.1 Architektur und Implementierung des CapsNets . . . 120
4.3.2 Training und Ergebnisse . . . 121
4.3.3 Diskussion und Lösungsvorschläge . . . 123

4.4 Überwachtes Lernen eines CapsNets zur Facial-Emotion-Recognition 124
4.4.1 Architektur und Implementierung dreier überwachter Ausgangs-CapsNets 125
4.4.2 Training und Ergebnisse der überwachten Ausgangs-CapsNets 130
4.4.3 Diskussion der Auswirkungen des Rekonstruktions-Moduls 132
4.4.4 Architektur und Implementierung weiterer überwachter CapsNets . . . 133
4.4.5 Training und Ergebnisse der weiteren überwachten Ausgangs-CapsNets 135
4.4.6 Diskussion der Ergebnisse 141
4.5 Unüberwachtes Lernen eines CapsNets zur Facial-Emotion-Recognition 141
4.5.1 Architektur und Implementierung des unüberwachten CapsNets 141
4.5.2 Training und Ergebnisse des unüberwachten CapsNets 144
4.5.3 Diskussion der Ergebnisse 146

5 Zusammenfassung, Diskussion und Ausblick 149

6 Anhang 153
6.1 Glossar 153
6.2 Verzeichnisse 154
6.2.1 Abbildungen 154
6.2.2 Tabellen 157
6.3 Quellen 158
6.4 Inhaltsverzeichnis der Medien CD 168
6.5 Appendix 169
6.6 Tabellarische Zusammenstellung der restlichen gefundenen wissenschaftlichen Literatur 169
6.7 Erklärung 172
6.8 Ermächtigung 172

1 Motivation

Nachdem die wichtigsten Grundlagen des Deep Learning gelernt wurden entsteht die Frage, wie nun an den aktuellen Stand der Forschung hingearbeitet und welche Teilgebiete dieses Feldes vorerst genauer betrachtet werden sollen. Diese Frage hat zahlreiche korrekte Antworten, jedoch ist dabei eine bedachte Wahl empfehlenswert. Hauptgrund dafür ist das rapide ansteigende Wachstum des bereits schnelllebigen Gebietes, wobei die Entwicklung der CNNs als ein Beispiel herangezogen werden kann. Neue vielversprechende Ansätze und Methoden sprießen hierbei so schnell wie Blumen im Frühling hervor, doch nur die meist beachteten, robustesten blühen längere Zeit und womöglich sogar bis zum nächsten Winter, der momentan weit entfernt scheint und vielleicht nicht mehr einkehrt. Um an die aktuelle Entwicklung anzuknüpfen führt selbstverständlich auch das Lernen von kurzlebigen Methoden näher ans Ziel und könnte dabei Erkenntnisse ans Tageslicht bringen, welche zu Unrecht wenig Einsatz finden. Jedoch ist für einen tiefen Einstieg eine neue, vielversprechende, von der Community beachtete Methode aus einigen Gründen von Vorteil: So erscheinen für diese in Rekordzeit Veröffentlichungen, welche es ermöglichen nach der Hinarbeit an das originale Modell, das Vorgehen und die Erkenntnisse der Weiterentwicklung zu betrachten, wobei zahlreiche altbekannte sowie neue Technologien des Machine Learning verwendet werden. Weiter ist ein solcher „Hype“ oft nicht ungerechtfertigt: Dies zeigt beispielsweise der herausragende Erfolg der CNNs seit 2012, deren Anwendung fortan in den meisten KI-Domänen erfolgreich stattfindet. Auch den in dieser Arbeit behandelten Capsule Networks könnte in Zukunft solch ein Erfolg blühen, da diese an verbesserungswürdigen Stellen der CNNs ansetzen, einige der aktuellen Forderungen an den Bereich erfüllen könnten und u.a. von Geoffrey Hinton - einer bekannten Persönlichkeit im Deep Learning - vorangetrieben werden, wobei den Modellen die dadurch entstehende Aufmerksamkeit zugutekommt, welche sich mit zahlreichen Veröffentlichungen anderer Forscher und Entwickler bemerkbar macht.

Bei der Lehre der Grundlagen des Deep Learning wird meist mit hoher Vorsicht auf die lose zu betrachtenden Analogien bezüglich des weitgehend unerklärtem menschlichen Gehirns hingewiesen. Diese Analogien zeichnen dabei oft ein Bild hinter der Entwicklung erfolgreicher Modelle, welches das Verständnis fördert und häufig zu weiteren Verbesserungen führt. Auch Capsule Networks besitzen Parallelen hinsichtlich der menschlichen Informationsverarbeitung und entstanden zudem aus daraus resultierenden Folgerungen. Weiter bilden diese Netze durch ihre Gemeinsamkeiten mit anderen Deep Learning Modellen einen wichtigen Baustein, welcher in größeren Architekturen aus vielen, im Folgenden erläuterten Gründen Verwendung findet und finden könnte. Aufgrund all dieser Argumente scheint es daher sinnvoll, aus dem Pool möglicher Antworten die Capsule Networks zu wählen und sich mittels dieser an den Forschungsstand einiger Teilgebiete des Deep Learning - insbesondere der Bildverarbeitung - hinzuarbeiten.

2 Einleitung

Die Erörterung von Capsule Networks könnte anhand oftmals verwendeter Erklärungsstrukturen von CNNs ausgeführt werden, wobei beispielsweise zu Beginn ein Überblick bezüglich des Modells sowie die folgliche, genaue Betrachtung der Grundstrukturen basierend auf diesem stattfindet. Im Anschluss wird dann auf die komplexeren State-of-the-Art-Architekturen hingearbeitet. Allerdings ist es aufgrund des aktuellen Entwicklungsstands sinnvoll vorerst ausführlich die generalisierte Idee theoretisch darzustellen und basierend auf dieser die chronologische Erläuterung der aktuellen Modelle vorzunehmen, da sich derzeit noch kein „Standardmodell“ etabliert hat. Dies wird im kommenden Theorieteil u.a. weiter diskutiert. Der theoretische Teil der Arbeit durchleuchtet daher - neben und mittels der biologischen Inspiration - erst die generalisierte Idee hinter Capsule Networks und stellt diese so dar, dass sie in kommenden Abschnitten als „Anker“ herangezogen werden kann. Folglich wird dann mittels Transforming-Autoencoder auf Capsule Networks mit dynamischem Routing-by-Agreement hingearbeitet, deren Schwächen durch die im Anschluss erörterten Matrix Capsules mit EM-Routing weiter verbessert werden sollen. Die Betrachtung der Matrix Capsules schließt dann den aktuellen Entwicklungsstand hinsichtlich der Schöpfer der Capsule Networks ab und die darauf beruhenden Erkenntnisse, Verbesserungen und neuen Modelle aus einer Vielzahl von verschiedensten Quellen werden betrachtet.

Bei der Erarbeitung der unterschiedlichen Modelle, Methoden und Erweiterungen werden neben der rein theoretischen Erörterung auch die Ergebnisse praktischer Experimente betrachtet, um ein tieferes Verständnis über die Funktionsweise und den Einsatz der unterschiedlichen daraus resultierenden neuronalen Netze zu erhalten. Vor allem Third-Party-Erkenntnisse werden bei deren Darstellung konstruktiv diskutiert und untereinander verglichen. Aus diesen Betrachtungen wird schließlich ein Vergleich zwischen CNNs und Capsule Network angestellt und die endgültige Aufgabenstellung für den praktischen Teil der Arbeit hergeleitet.

Im praktischen Teil findet die Umsetzung von drei ausgewählten Aufgaben und die Diskussion der erzielten Ergebnisse statt, wobei einige unterschiedliche Modelle gelernt und getestet werden. Sowohl überwachtes als auch unüberwachtes Training von tieferen Capsule Networks wird hier praktisch implementiert und auf einer rechenstarken Cloud durchgeführt. Das parallele Ausführen kleinerer Experimente wird durch die zeitgleiche Verwendung des eigenen Rechners möglich gemacht.

Zum Abschluss der Arbeit werden letztlich sowohl die praktischen, als auch die theoretischen Ergebnisse zusammengefasst und in ihrer Gesamtheit diskutiert, woraus einen Ausblick resultiert.

3 Theoretische Erörterung von Capsule Networks

Ein Capsule Network stellt einen neuen, auf Neuronen basierenden Deep Learning Ansatz dar, dessen Netzwerk-Modell dazu konzipiert wurde diverse Schwächen herkömmlicher CNNs auszubessern. Dieser Abschnitt durchleuchtet zuerst den Grundaufbau eines CapsNets, wobei zusätzlich der biologische Hintergrund erörtert wird. Es folgt die Erläuterung weiterer Capsule-basierende Modelle - beispielsweise Matrix Capsules - sowie anderer Verbesserungen am ursprünglichen Modell. Anschließend wird ein Vergleich zwischen CNNs und CapsNets angestellt, woraus Vor- und Nachteile der beiden Modelle resultieren und wobei die In- und Äquivariantz im Deep Learning Bereich betrachtet wird; zuletzt werden mit den Modellen erreichte Ergebnisse diskutiert und daraus die Aufgabenstellung dieser Arbeit gebildet.

3.1 Generalisierte Idee

Um den Aufbau und die Funktionsweise von CapsNets zu verstehen, wird im Folgenden die generelle Idee hinter diesem neuronalen Netzwerk erarbeitet und diese durch zwei konkrete Architekturen vertieft. Dabei werden zu Beginn die kleinsten Einheiten betrachtet und folglich das Verfahren bzw. das Modell inkrementell erweitert.

3.1.1 Capsules

Eine Capsule (engl. für Kapsel, Hülse oder Schale) stellt eine kleine Gruppe von Neuronen dar, die ihre Ergebnisse in einem Aktivitätsvektor kapseln. Dieser Vektor repräsentiert Instanziierungsparameter spezieller Entitätentypen, wie Objekte oder Objektteile, wobei die Aktivität der Neuronen innerhalb einer Capsule mit den Eigenschaften einer Entität korrespondieren. Diese Eigenschaften inkludieren verschiedene Typen von Instanziierungsparameter, wie z.B. Pose, Verformung, Geschwindigkeit, Albedo, Farbton, Textur [SS17, S. 1] [Hin11, S. 1-2]. Eine spezielle Eigenschaft ist hierbei die Existenz selbst, die beispielsweise über eine separate logistische Einheit, welche eine Existenzwahrscheinlichkeit berechnet, erreicht werden kann (Abschnitt 3.2 und 3.4) [SS17, S. 1] [Hin11, S. 4]. Alternativ kann z.B. die Länge des Aktivitätsvektor als Existenzwahrscheinlichkeit betrachtet und dessen Orientierung dazu forciert werden, die besagten Instanziierungsparameter zu repräsentieren (Abschnitt 3.3) [SS17, S. 1-2]. Ein konkretes Beispiel wäre eine DigitCapsule aus Abschnitt 3.3, deren Ausgabevektor in seiner Länge die Existenzwahrscheinlichkeit der ihr zugeordneten MNIST-Zahl enthält und dessen Orientierung u.a. Eigenschaften, wie die Höhe die Strichstärke oder Schräglage der Ziffer repräsentiert. Capsules führen daher komplizierte interne Berechnungen auf deren Eingabe durch und kapseln darauf ihre Ergebnisse in kleine Vektoren für eine sehr informative Ausgabe [Hin11, S. 2]. Hierbei kann eine Parallele zu SIFT gezogen werden, da dieser Algorithmus ebenfalls einen Vektor

als Ausgabe berechnet, der unter anderem die Pose des Objektes beinhalten kann [Hin11, S. 1].

Da [HSF18a] und [Hin11, S. 5-6] jedoch den Aktivitätsvektor (Ausgabevektor) einer Capsule durch eine Matrix ersetzen, um z.B. Informationen über die Pose in höheren Dimensionen zu erlangen (Abschnitt 3.2.3 und 3.4), könnte in der generellen Darstellung einer Capsule die Ausgabe als Matrix verallgemeinert werden. Jeder n-dimensionale Vektor wäre als $n \times 1$-dimensionale Matrix betrachtbar. Auch wenn eine Matrixdarstellung bereits sehr allgemein ist, wird in der erarbeiteten, generellen Darstellung in Abbildung 3.1 die Ausgabe in keiner Form konkretisiert. Somit folgt die Definition einer Capsule in dieser Arbeit der Definition aus [HSF18a, S. 1]: Eine Capsule ist eine Gruppe von Neuronen deren Ausgabe verschiedene Eigenschaften derselben Entität repräsentieren.

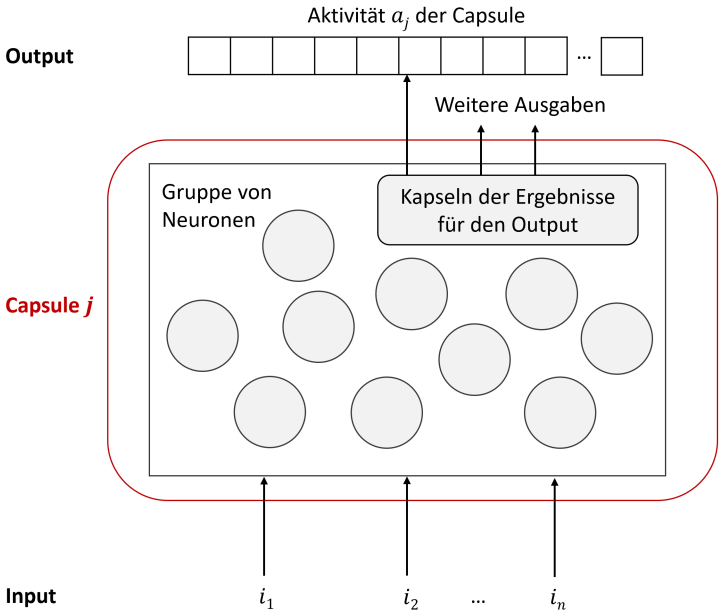


Abbildung 3.1: Lösungsneutaler Aufbau einer Capsule

Abbildung 3.1 veranschaulicht den Aufbau einer *Capsule* j lösungsneutral. Sowohl die genaue Verknüpfung der Inputs mit der *Gruppe von Neuronen*, die Verknüpfung der Neuronen untereinander, das Verfahren zum *Kapseln der Ergebnisse für den Output*, die Art der Ein- und Ausgabe (i_n und a_j), als auch die Art der Neuronen und der Verknüpfungen wird in der generalisierten Idee einer Capsule nicht vorgegeben. Zusätzlich sollte erwähnt werden, dass eine Capsule auch mehrere Ausgaben für verschiedene Zwecke besitzen kann (Abschnitt 3.3 und 3.4). Abbildung 3.1 hilft dabei, zu erkennen welche Teile einer Capsule separat betrachtbar und austauschbar sind und liefert einen groben Überblick bezüglich der Funktion einer Capsule. Folgende Abschnitte bieten hierbei zahlreiche Beispiele für die generelle Idee und vertiefen diese weiter.

Funktioniert eine Capsule korrekt, so soll die Wahrscheinlichkeit, dass eine Entität in der Eingabe präsent ist, lokal invariant sein - sie verändert sich nicht solange sich die Entität über eine Vielzahl an möglichen Erscheinungen des begrenzten, von der Capsule abgedeckten, Bereichs bewegt. Die Instanziierungsparameter hingegen sind äquivariant: Wenn sich die Betrachtungsbedingungen der Entität ändern und diese sich über ihre möglichen Erscheinungen bewegt, dann ändern sich auch die Instanziierungsparameter um einen entsprechenden Wert, da diese u.a. die intrinsischen Koordinaten der Entität im Erscheinungsbereich darstellen [Hin11, S. 2].
Bildlich und einfach formuliert: Capsules sollen mit der gleichen Wahrscheinlichkeit wissen, ob sich ein Glas Wasser im Raum befindet, solange dieses sich in ihrem Sichtfeld (rezeptiven Feld) befindet, egal ob das Glas von der Tür oder vom Arbeitsplatz aus betrachtet wird, aber Capsules müssen dabei immer ausdrücken können, ob das Glas ihnen von ihrem Blickwinkel aus z.B. klein, groß, weit weg, voll oder leer erscheint. Sie besitzen also neuronale Aktivitäten, die variieren, wenn ihr Blickwinkel variiert, anstatt zu versuchen, die Variation des Blickwinkels aus den neuronalen Aktivitäten zu eliminieren [SS17, S. 9].
Eine Parallele zum Menschen wäre hierbei beispielsweise, dass eine Sakkade eine reine Translation des Netzhautbildes bewirkt, dabei besitzt jedoch der Cortex nicht-visuellen Zugang zu Informationen über die Augenbewegungen [Hin11, S. 2-3]. Folglich variieren neuronale Aktivitäten, da die beiden Eingaben variieren und die Variation des Blickwinkels kann durch Informationen über die Augenbewegung beachtet werden.

Eine weitere Parallele zum Menschen wäre der Neocortex: Dieser scheint in Minicolumns (column engl. für Säule) strukturiert zu sein, welche sich durch seine sechs Schichten ziehen. Eine corticale Minicolum ist eine vertikale Anordnung von ca. 80-120 Neuronen; 50-100 Minicolumns gruppieren sich zu eine Macrocolumn. Neuronen in einer Minicolumn scheinen das selbe rezeptive Feld zu besitzen und die selben Features zu codieren. Minicolumns, die sich zu Marcrocolums gruppieren, verzeichnen alle das selbe rezeptive Feld, codieren jedoch unterschiedliche Features.
Die Aktivität einer Capsule repräsentiert mehrere Instanziierungsparameter - Features - einer Entität in deren Eingabe bzw. in ihrem rezeptiven Feld. So könnte eine Capsule als Macrocolumn betrachtet werden, deren Neuronen intern die Funktion von Minicolumns übernehmen. Die Analogie verstärkt sich, wenn wie in Abschnitt 3.3.3 beschrieben eine PrimaryCapsule intern mehrere Convolutional Units besitzt, welche die Neuronen der PrimaryCapsule in columnähnlicher Struktur gruppieren. Weiter wird vermutet, dass es zwischen räumlich naheliegenden Minicolumns erregende Verbindungen gibt und zwischen Minicolumns die sich weit auseinander befinden abschwächende. Wie die folgenden Abschnitte zeigen werden, gibt es Verbin-

dungen zwischen Capsules, welche sich jedoch auf Capsules zwischen Layern begrenzen und nicht weiter von der räumlichen Entfernung von Strukturen innerhalb Capsules abhängig sind. Allerdings kann nach der Erläuterung in Abschnitt 3.1.3 eine andere Parallele gezogen werden.

3.1.2 Topologie eines Capsule Networks

Das menschliche Sehen ignoriert irrelevante Details, indem es eine sorgfältig bestimmte Sequenz von Fixierungspunkten verwendet [SS17, S. 1], die durch Ausführung schneller Sakkaden erfasst werden [Zim]. Fixierungspunkte dienen dazu sicherzustellen, dass nur ein kleiner Teil des optischen Feldes mit der höchsten Auflösung verarbeitet wird. Um grob verstehen zu können, wieviel Wissen über eine Szene aus einer Sequenz von Fixierungspunkten erschlossen werden kann und wieviel Wissen Menschen von einer einzelnen Fixierung erlangen, können beispielsweise die in Abschnitten 3.2.4 und 3.1.5 beschriebenen Experimente der Introspektion (Selbstbeobachtung) genutzt werden. Für CapsNets wird angenommen, dass unser mehrschichtiges, visuelles System eine Parse-Tree-ähnliche Struktur für jede Fixierung bildet; jedoch wird das Problem, wie die aus einer Einzelfixierung entstandenen Parse Trees über mehrere Fixierungen koordiniert werden, ignoriert [SS17, S. 1].

Die Konstruktion von Parse Trees findet generell on-the-fly statt, indem dynamisch Speicher alloziert wird [SS17, S. 1]. Hinton et al. [Hin00] und [Hin79] nimmt weiter an, dass für eine einzelne Fixierung ein Parse Tree aus einem Multilayer Perceptron gemeißelt wird, wie eine Skulptur aus Gestein [SS17, S. 1].
Jeder Layer des Netzwerks wird in eine kleine Gruppe von Neuronen aufgeteilt, die wie in Abschnitt 3.1.1 beschrieben, eine Capsule bilden. Jeder Knoten im Parse Tree korrespondiert mit einer aktiven Capsule. Ein Routingprozess (Abschnitt 3.1.3) sorgt dafür, dass jede Capsule zu einem passenden Elternteil im Tree zugeordnet wird. Für die höheren Ebenen eines visuellen Systems löst dieser Prozess das Problem der Zuordnung von Teilen zum Ganzen [SS17, S. 1-2], so wie Menschen beispielsweise die Augen einem Gesicht zuordnen und sich über die dortige Position bewusst sind (ein Auge befindet sich i.d.R. nicht direkt links neben dem Mund). Hier ist das Wissen über die Teil-Ganze Beziehung viewpoint (engl. für Blickwinkel)-invariant und soll so repräsentiert werden, dass das Wissen über die Instanziierungsparameter der gerade betrachteten Entität und ihrer Teile viewpoint-äquivariant ist und durch die neuronale Aktivitäten repräsentiert wird [Hin11]. Im folgenden Abschnitt zeigt Abbildung 3.2 einen solchen Parse-Tree, der mithilfe des Routings erläutert wird. Konkrete Architekturen von Capsule Networks (kurz CapsNets) werden in den Abschnitten 3.2, 3.3, 3.4 und 3.5 vorgestellt.

3.1.3 Routing-Algorithmus eines Capsule Networks

Auch wenn nicht alle vorgestellten CapsNets (siehe z.B. Abschnitt 3.5.3) einen Routing-Algorithmus nutzen, soll auch dieser erst lösungsneutral dargestellt werden, da der Großteil der Literatur Routing-Algorithmen verwendet und da diese als Kern der in Abschnitt 3.3 und Abschnitt 3.4 erläuterten, originalen Modelle gelten:
Wie bereits in Abschnitt 3.1.2 erwähnt, sorgt der Routing-Algorithmus dafür, dass jeder Capsule für die gegebene Entität ein passender Elternteil zugeordnet wird. Ziel des Routings ist also die Zuordnung einer Capsule A, die eine einfachere Entität repräsentiert, zu einer Capsule B, die eine komplexere Entität darstellt, welche unter anderem aus der Entität der Capsule A besteht und in einer Beziehung mit dieser steht. Das Routing selbst soll dabei so konstruiert sein, dass diese Zuordnung eine entsprechende im Input vorhandene Entität bildet (z.B. ein Schiff aus Segel, Deck, usw.), welche letztlich erkannt und mit ihren, für die visuelle Aufgabe relevanten, Instanziierungsparameter beschrieben werden kann.

Da u.a. [SS17], [HSF18a] und [Bah18] das Routing mit verschiedenen Routing-By-Agreement-Algorithmen konkretisieren, wird im Folgenden die allgemeine Idee dahinter vorgestellt: Aktive Capsules auf einem Level treffen Vorhersagen für die Instanziierungsparameter von höher liegenden Capsules (über Transformationsmatrizen). Wenn für eine Capsule im höheren Level mehrere Vorhersagen von niedrigeren Capsules übereinstimmen, wird diese aktiv. Eine niedriglevelige Capsule bevorzugt das Senden ihrer Ausgabe zu höher liegenden Capsules, deren Instanziierungsparameter/Aktivität mit ihrer Vorhersage übereinstimmt. Um die Übereinstimmung zu berechnen kann beispielsweise das Skalarprodukt zwischen den Vorhersagen der unteren Capsule und dem Aktivitätsvektor der oberen Capsule gebildet werden. Je größer das Skalarprodukt ausfällt, desto stärker stimmen die Orientierungen der Vektoren überein und desto eher bewahrheitet sich die Vorhersage der unteren Capsule bezüglich des Aktivitätsvektors der oberen (Abschnitt 3.3) [SS17, S. 1]. So voten die Capsules im niedrigeren Layer - also die bereits erkannten Entitätsteile - für die Capsules im höheren Layer, welche verschiedene mögliche Zusammensetzungen der aktiven Entitätsteile aus dem vorherigen Layer darstellen. Dies ist ein leistungsfähiges Segmentierungsprinzip, welches der Kenntnis über bekannte Formen ermöglicht die Segmentierung zu steuern, anstelle nur einfach zusammenhängende Hinweise wie Nähe oder Übereinstimmung in Farbe oder Geschwindigkeit zu verwenden [HSF18a, S. 1].

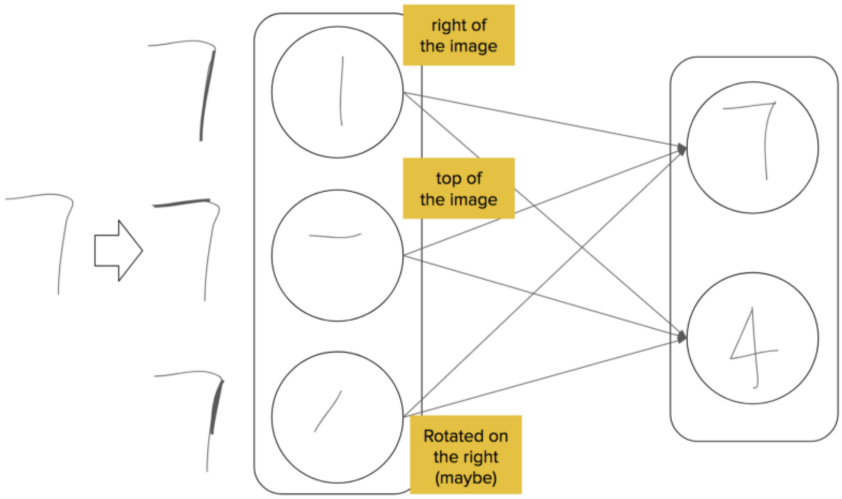


Abbildung 3.2: Routingkonzept eines Capsule Networks aus [Kot]

Abbildung 3.2 veranschaulicht dieses Prinzip und zeigt, dass das Netzwerk in der Eingabe (ein Bild einer *7*) eine vertikale und eine horizontale Linie findet, sowie ein Hinweis auf eine diagonale Linie (veranschaulicht durch *gelbe Kästchen*). Es wurden demnach Features für eine *7* und für eine *4* gefunden. Erhält die Capsule, welche die *4* repräsentiert, diese Eingaben, wird die Capsule eine niedrige Aktivierung ausgeben, da die erkannte horizontale Linie räumlich nicht mit der Horizontalen der *4* übereinstimmt und da für eine diagonale Linie nur ein schwacher Hinweis existiert [Kot]. Die entsprechenden Vorhersagen der unteren Capsules für die *4* unterscheiden sich, wie in Abbildung 3.3 dargestellt:

Abbildung 3.3: Nicht übereinstimmende Vorhersagen für die Pose der 4 aus [Kot]

Für Capsule *7* hingegen sind alle drei Linien räumlich gut platziert, weshalb ihre Aktivierung hoch ausfällt. Betrachtet folglich die Capsule der vertikalen Linie ihre Vorhersage, stellt sie fest, dass diese für eine *4* kaum übereinstimmt und die Aktivierung sehr schwach ist, für die *7* hingegen vice versa. Deshalb bevorzugt diese Capsule im Weiteren das Senden ihrer Ausgabe zur Capsule der *7* und passt sich dementsprechend an. Beispielsweise schwächt sie ihre Kopplungsstärken zu *4* ab und verstärkt die zur *7* (Abschnitt 3.3)[Kot].
Ein wichtiger Unterschied zwischen Capsules und herkömmlichen neuronalen Netzen besteht

darin, dass die Aktivierung einer Capsule auf einem Vergleich zwischen mehreren eingehenden Pose-Vorhersagen (z.B. Aktivitätsvektoren oder Matrizen) beruht, während die Aktivierung in einem normalen neuronalen Netz aus dem Vergleich eines einzelnen eingehenden Aktivitätsvektors (alle Ausgaben der vorherigen Neuronen) mit einem gelernten Gewichtsvektor hervorgeht [HSF18a, S. 1].

Der Routing-Algorithmus (das „Routing-By-Agreement“) kann auch als ein paralleler Attention-Mechanismus betrachtet werden, welcher jeder Capsule auf einem Level (z.B. durch Kopplungsstärken) erlaubt, sich einer aktiven Capsule im Level darunter zu widmen und alle anderen zu ignorieren [SS17, S. 6]. Abschnitt 3.3.5 zeigt dies anhand der Segmentierung zweier überlappender MNIST-Zahlen.

CapsNets können ebenfalls von inverser Computergrafik Perspektive aus betrachtet werden [SS17], denn Computergrafik konstruiert ein visuelles Bild aus einer internen, hierarchischen Repräsentation von geometrischen Daten. Objekte können in Arrays gespeichert werden und deren relativen Positionen und Orientierungen in Matrizen. Im Rendering wird dann aus den Daten die aktuelle Repräsentation auf den Bildschirm konvertiert. Hinton argumentiert, dass das Gehirn von den Augen stammende, visuelle Informationen in hierarchische Repräsentationen dekonstruiert, und versucht diese mit bereits gelernten Muster und Beziehungen in Verbindung zu bringen, sodass das Lernen und Wiedererkennen eines Objektes nicht vom Blickwinkel abhängt. Diese hierarchische Repräsentation, diese Beziehung zwischen den Objekten, kann in neuronalen Netzen, wie in der Computergrafik, über Pose (Translation + Rotation) zwischen Objekten modelliert werden [Ghtb] [Peca]. Im Falle der CapsNets in den Transformationsmatrizen, welche auf die Aktivitäten angewandt Vorhersagen bilden. Wenn eine Capsule lernt die Pose ihrer visuellen Entität in beispielsweise einem Vektor auszugeben, der in einer lineare Beziehung zu der „natürlichen“ Repräsentation der Pose steht, wie sie in der Computergrafik genutzt wird, dann kann ein einfacher, selektiver Test angewandt werden, um zu bestimmen, ob die von den aktiven Capsules A und B repräsentierten, visuellen Entitäten die richtige räumliche Beziehung zueinander haben, um eine höher liegende Capsule C zu aktivieren, welche eine komplexere Entität darstellt:

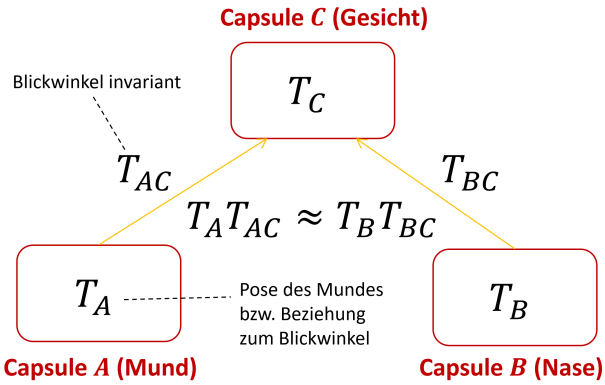


Abbildung 3.4: Blickwinkel invariante Vorhersagen von Capsule A und B für Capsule C im Bezug zu Computergrafik nach [Ghtb]

Abbildung 3.4 veranschaulicht diesen Test: Hierbei ist die Ausgabe der Pose von Capsule A durch die Matrix T_A repräsentiert, welche die Koordinatentransformation zwischen der kanonischen, visuellen Entität von A und der aktuellen Instantiierung der Entität, welche von Capsule A gefunden wurde darstellt. Wird die Pose T_A für die Teil-Ganzes-Koordinatentransformation mit T_{AC} multipliziert, welches die kanonische, visuelle Entität A mit der kanonischen, visuellen Entität C in Verbindung bringt, entsteht dabei die Vorhersage T_{AC} für T_C. Analog kann dies für T_B und T_{BC} ausgeführt werden. Wenn beide Vorhersagen gut übereinstimmen sind die von Capsule A und B gefunden Instantiierungen in der richtigen räumlichen Beziehung zueinander um C zu aktivieren. Der Durchschnitt der Vorhersagen zeigt hierbei, wie die komplexere von C repräsentierte Entität zur kanonischen, visuellen Entität C relativ transformiert ist. Repräsentiert beispielsweise A einen Mund und B eine Nase, so können beide eine Vorhersage für die Pose eines Gesichts treffen; stimmen diese überein befindet sich Nase und Mund in der richtigen räumlichen Beziehung zueinander, um ein Gesicht zu formen [Hin11, S. 2].

In Kenntnis darüber, dass für die Darstellungsformen der Pose von CapsNets (z.B. Matrix oder Vektor) nicht garantiert ist, dass diese ausschließlich Pose-Parameter beinhalten, sondern evtl. auch anderen Instanziierungsparameter einer Entität (oft gewollt z.B. Abschnitt [SS17]) oder gar Parameter die nicht zur Entität gehören, wird in der Arbeit unter einer Pose-Matrix in Zusammenhang mit CapsNets eine Instanziierungsparameter-Matrix verstanden, die durch den Lernprozess und den Aufbau von Matrix Capsules u.a. die Pose einer Entität darstellen soll.

Capsules nutzen zur Bestimmung räumlicher Beziehungen hochdimensionale Koinzidenz-Filterung: Eine vertraute Entität kann erkannt werden, indem auf die Übereinstimmung (Agreement) zwischen den Vorhersagen/Stimmen/Votes für die Instanziierungsparameter geachtet wird. Diese Votes kommen von bereits erkannten Entitätsteilen. Ein Entitätsteil kann beispielsweise

ein Vote erzeugen, indem dieser seine eigene „Pose-Matrix“ mit einer gelernten, blickwinkelinvarianten Transformationsmatrix multipliziert, welche die Beziehung zwischen dem Teil und dem Ganzen repräsentiert. Ändert sich der Blickwinkel, dann ändern sich die Pose-Matrizen der Teile und des Ganzen in einem koordinierten Weg, sodass die Übereinstimmung der Votes der verschiedenen Entitätsteile bestehen bleiben [HSF18a, S. 1].
Das Finden enger Cluster von hochdimensionalen Votes, die in einem Nebel irrelevanter Votes miteinander übereinstimmen, ist eine Möglichkeit, das Problem der Zuordnung von Teilen zum Ganzen zu lösen. Dies ist nicht trivial, da der hochdimensionale Pose-Raum nicht in der Weise gerastert werden kann, wie der niederdimensionale Translationsraum gerastert ist, um Faltungen (Convolutions) zu ermöglichen. Diese Herausforderung löst der iterative „Routing-By-Agreement“-Prozess, welcher die Wahrscheinlichkeit aktualisiert, mit der ein Entitätsteil einer ganzen Entität zugewiesen wird. Diese Aktualisierung findet basierend auf der Nähe der von diesem Entitätsteil kommenden Votes zu den Votes anderer Entitätsteile statt, welche der ganzen Entität zugeordnet sind. [HSF18a, S. 1]. In Folge der Arbeit wird dieser Prozess anhand mehrerer CapsNets konkretisiert.

Zusammenfassend kann argumentiert werden, dass Blickpunktänderungen komplizierte Effekte auf Pixelintensitäten haben, aber einfache, lineare Effekte auf die Beziehung (vor allem die Pose) zwischen einer Entität oder eines Entitätsteils und dem Betrachter. Das Ziel von Capsules ist diese zugrundeliegende Linearität sowohl für den Umgang mit Blickwinkelvariationen als auch für die Verbesserung von Segmentierungsentscheidungen zu nutzen [HSF18a, S. 1].

Um auf die biologische Inspiration hinsichtlich corticalen Columnen zurückzukommen, wird argumentiert, dass zwischen Capsules, deren Entitäten dieselbe komplexere Entität bilden, und der Capsule der besagten komplexeren Entität eine erregende Verbindung existiert, weil ihre Kopplungsstärken erhöht werden, da die Entitäten u.a. räumlich naheliegen. Zwischen Capsule der räumlich entfernten Entitäten existieren folglich abschwächende Verbindungen. Dies ist einer sehr schwache Parallele zu den bereits in Abschnitt 3.1.1 erwähnten Verbindungen der Minicolumns, bei welchen Verbindungen zwischen naheliegenden Minicolumns erregend sind und vice versa. Allerdings muss beachtet werden, dass eine Capsule eher eine Makrocolumn darstellt, da sie in ihrer Aktivität mehrere Feature einer Entität repräsentiert. Wird jedoch die Ausgabe einer Capsule als eine Zusammenfassung der Ausgaben der Minicolumns betrachtet, so greift diese Parallele. Da Minicolums in Makrocolumns mit dem selben rezeptiven Feld gruppiert werden und die Wahrscheinlichkeit erhöht wird, dass die Entität räumlich zusammenliegt, wenn sie in einem rezeptiven Feld auftritt, und falls die Features von Minicolumns, welche miteinander verbunden werden, häufig räumlich zusammenliegen sollten, würde die Parallele

zu Capsules dadurch weiter verstärkt werden. Es wird hierbei jedoch nochmal erwähnt, dass dies einen sehr losen Zusammenhang darstellt und dieser, wie so oft im Deep Learning, auf Hypothesen beruht.

3.1.4 Weitere biologische Inspirationen und Parallelen

Place- und Rate-Coding in CapsNets

Die Lageinformation einer Entität ist bei niedrigleveligen Capsules „place-coded“ und wird durch die Position der aktiven Capsules bestimmt [SS17, S. 2], da diese low-level Features für ihre kleinen rezeptiven Felder berechnen. Bewegt sich ein Objekt in der Eingabe, könnte diese „unter“ eine neue Capsule gelangen und die Lageinformation wird folglich durch eine andere Capsule dargestellt [Hin].

Je höher sich eine Capsule in der Hierarchie befindet, desto mehr Lageinformation fließt in die realen Werte der Aktivität/Ausgabe einer Capsule („rate-coded“) [SS17, S. 2], da das rezeptive Feld höherer Capsules mehrere rezeptive Felder niedrigerer Capsules umfasst. Die Information wird also in der Aktivität der Capsule, welche als Feuerrate betrachtet werden kann, codiert, beispielsweise in einem Ausgabevektor [Hin].

Da höhere Capsules größere Gebiete umfassen wird demnach niedrige „place-coded“ Äquivarianz in höhere „rate-coded“ Äquivarianz umgewandelt [Hin]. Diese Verschiebung von Place-Coding zu Rate-Coding kombiniert mit der Tatsache, dass Capsules auf höherer Ebene komplexere Einheiten mit mehr Freiheitsgraden darstellen, legt nahe, dass die Dimensionalität von Capsules zunehmen sollte, je höher sich diese in der Hierarchie befinden [SS17, S. 2].

Two-streams Hypothesis

Die Two-streams Hypothesis ist ein weit akzeptiertes Modell des neuronalen Verarbeitens für Sehen [MWE]. Hierbei wird nach Goodale et al. [Pfa]] die Sehverarbeitung in zwei in Abbildung 3.5 skizzierte Ströme eingeteilt: Der *dorsale Storm (grün)*, welcher zur Erkennung der Objekte im Raum dient, am Führen von Aktionen beteiligt ist und auch als Parietal Strom, „Wo“-Strom oder „Wie“-Strom, bezeichnet wird. Dieser „erstreckt“ sich vom primären visuellen Cortex (V1) zum *Occipitallappen*, vor zum *Parietallappen* und ist stark mit dem *ventralen Strom (lila)* verbunden, welcher mit der Objekterkennung und der Formdarstellung assoziiert und daher auch „Was“-Strom genannt wird. Dieser fließt vom primären visuellem Cortex (V1) hinunter zum *Temporallappen*.

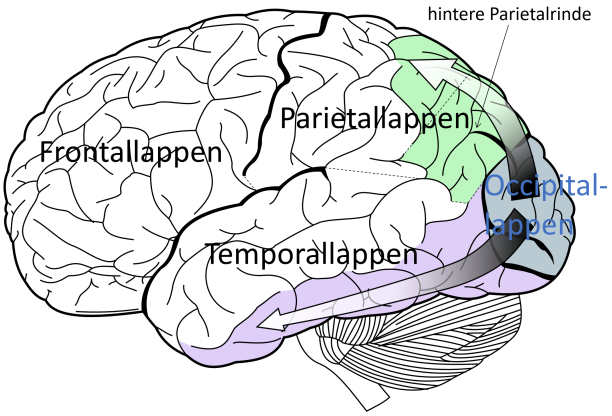


Abbildung 3.5: Dorsaler und ventraler Strom skizziert nach [Idv] und [Pfa]

Weiter unterstützen einige neurologische Störungen die Two-streams Hypothesis: Beispielsweise kann bei Läsionen (Schädigung, Verletzung oder Störung) an der dorsalen, hinteren Parietalrinde u.a. eine optische Ataxie entstehen, bei welcher der Patient keine visuell-räumliche Informationen zur Führung von Armbewegungen nutzen kann [JS92]. Der „Wo“-Strom oder der „Wie“-Strom erfüllt nicht mehr seinen Zweck. Als weiteres Beispiel dient die dorsale Simultanagnosie, welche aus bilateralen Läsionen (Läsionen an beiden Seiten des Gehirns) der Verbindung zwischen dem *Parietal- und Occipitallappen* resultiert. Dabei kann der Patient nur ein Objekt auf einmal sehen, ohne dieses als eine Komponente aus mehreren Objekten oder einem Set an Details in einem Kontext (z.B. der Wald für die Bäume) wahrzunehmen, wodurch er mit mehreren Objekten in einem Raum zusammenstößt, da er diese nicht bemerkt [JS92]. Auch hier erfüllt der „Wo“-Strom oder „Wie“-Strom nicht mehr seinen Zweck. Schäden am ventralen Strom haben hingegen zur Folge, dass Gesichter nicht mehr erkannt werden können, oder ein Gesichtsausdruck nicht mehr interpretierbar ist. Weiter resultiert ventrale Simultanagnosie aus Schäden an der linken unteren Verbindung zwischen dem *Temporal- und Occipitallappen*. Der Patient kann zwar mehrere Objekte auf einmal sehen und daher durch einen Raum navigieren ohne mit diesen zu kollidieren, jedoch ist die Erkennung (Klassifizierung) von Objekten nur stückweise möglich, oder zu einem Objekt auf einmal limitiert [JS92]. Der „Was“-Strom erfüllt nicht mehr seinen Zweck.

Die Two-streams Hypothesis gibt Aufschlüsse darüber, dass die Erkennung eines Objektes von dessen Lokalisierung getrennt werden sollte. Dies wird in CapsNets versucht, welche einerseits in ihren berechneten Existenzwahrscheinlichkeiten die Präsenzen von Objekten darstellen und in Initialisierungsparametern u.a. deren räumliche Eigenschaften. Das Routing-By-Agreement verwendet dabei Poseinformationen um ein komplexeres Objekt (höhere Capsule) aus kleineren Objekten (niedrigere Capsules) zusammenzusetzen, zu beschreiben und ggf. zu erkennen. In

Verbindung damit können Patienten mit dorsaler Simultanagnosie kein Objekt als Komponente mehrerer Objekte oder ein Set an Details in einem Kontext wahrnehmen. Da jedoch die Initialisierungsparameter neben Poseinformationen auch Formdarstellungen wie die Strichstärke einer Zahl repräsentieren können (somit „wo“ und „was“) (Abschnitt 3.3) ist die Trennung der beiden Ströme nicht zwangsläufig in jeder Capsule-Architektur strikt durchgesetzt, weil somit das ventrale System des CapsNets ebenfalls die Form eines Objektes beschreibt. Weit schwächer umgesetzt wirkt diese Trennung, wenn ein Capsule tatsächlich als ein realer Teil in einem Gehirn betrachtet wird, da dann durch eine einzelne Capsule beide Ströme fließen würden und die Ströme daher nicht wie oben beschrieben räumlich getrennt wären. Wird eine Capsule hingegen als Abstraktion eines Vorgehens betrachtet kann das Routing-By-Agreement lose als *dorsaler Storm* („wo“ und „wie“) betrachtet werden, der die räumliche Informationen liefert und stark mit der Einheit (z.b. der logistischen Einheit) verbunden ist, welche die Existenzwahrscheinlichkeit berechnet und lose als *ventraler Storm*, den „Was“-Strom bezeichnet werden könnte. Interessant wären allerdings die Ergebnisse eines Capsule-Modells, welches die Trennung des „Wo“- und „Was“-Stroms strikt durchsetzt, jedoch wurde zum Kenntnisstand dieser Arbeit kein passendes Modell in der Literatur gefunden.

Mögliches neues Neuronenmodell

In [Sar+17] führten Shira Sardi et al. neue Experimente durch, die das aktuelle Verständnis für Neuronenmodelle in Frage stellen. Dies wurde anhand der Bildung einer experimentellen Strategie erreicht, die auf den folgenden grundlegenden Schritten und Anforderungen basiert: Zunächst musste die Stimulation des Neurons aus mehreren Raumrichtungen, entweder unabhängig oder synchron, möglich gemacht werden. Diese indirekte anisotrope Stimulationen des Neurons erfordert gleichzeitig abstimmbare Stimulationstimings auf einer Sub-Millisekunden-Zeitskala; darüber hinaus muss ein Stimulierungsplan über viele Minuten hinweg stabil gehalten werden, während die kontinuierliche Aufzeichnung der neuronalen Antworten intrazellulär stattfindet. Die Erfüllung all dieser Anforderungen führte zu anisotropen extrazellulären Stimulationen, welche zusätzlich mit den intrazellulären Aufnahmen und Stimulationen synchronisiert werden musste. Somit wurden einige deutliche Signaturen in den Antworten des Neurons gefunden, welche sich erkennbar zwischen mehreren Stimulationen anisotroper Quellen und Stimulationen von einem einzigen Ort aus unterscheiden. Dementsprechend wurde eine Reihe von Experimenten entwickelt, um das neu vorgeschlagene neuronale Berechnungsschema aufzudecken und zu unterstützen. Diese Experimente werden hier jedoch nicht genauer erläutert, um nicht zu weit von Thema abzuschweifen. Die Erkenntnisse der Experimente weisen jedoch darauf hin, dass das Neuron anisotrop nach dem Ursprung der ankommenden Signale an die Membran über seine dendritischen Bäume aktiviert wird.

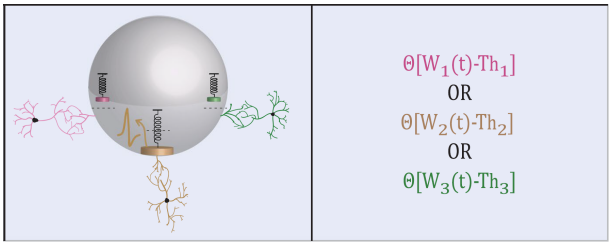


Abbildung 3.6: Schematische Darstellung des neuen Neuronenmodells [Sar+17]

Ein Neuron besteht demnach, wie Abbildung 3.6 zeigt, aus mehreren unabhängigen Threshold-Units und jede dieser subzellulären Threshold-Units summiert ihre eingehenden Signale aus einer gegebenen beschränkten Richtung mittels ihrem Schwellwert auf. Diese Threshold-Units übertragen dann die ankommenden Signale über ein einzelnes Axon an das verbundene Neuron. Die anisotropen erregbaren Stellen sind nicht identisch und sind durch unterschiedliche Spike-Wellenformen und unterschiedliche Summierungsspezifikationen gekennzeichnet. Weiter müssen laut den Autoren die anisotropen Threshold-Units von der Vorverarbeitung unterschieden werden, die mit den dendritischen Berechnungen verbunden ist und parallel lokal in jedem Dendriten und seinen Zweigen durchgeführt wird. Jede Threshold-Unit innerhalb des Neurons sammelt ihre eigenen anisotropen ankommenden Signale, weshalb keine direkte räumliche Summation zwischen ankommenden Signalen zu verschiedenen Threshold-Units existiert [Sar+17]. Laut [Dcf] wird unter „räumlicher Summation“ kurz das Aufsummieren und schließlich elektornische Ausbreiten der durch verschiedene Synapsen entstandene Potentiale in der Zelle verstanden. Aus den durchgeführten Experimenten konnte jedoch nicht die Auflösung der Anisotropie des Neurons abgeleitet werden, und es konnte auch nicht gezeigt werden, ob jede subzelluläre Threshold-Unit an einen Dendriten oder an einen Haufen benachbarter Dendriten gekoppelt ist. Es wäre jedoch möglich, dass die Anzahl der subzellulären Threshold-Units der Anzahl der hauptsächlichen anisotropen Richtungen der neuronalen Dendriten folgt, z.B. zwei, drei oder mehrere. Das Szenario, dass die Anzahl der Schwelleneinheiten, die ein Neuron bilden, unabhängig von der Anzahl der Dendriten ist, wäre ebenfalls denkbar und könnte ein neues Merkmal für die Klassifizierung von Neuronen darstellen (nach ihren Rechenfähigkeiten und ihrer räumlichen Auflösung für ankommende Signale) [Sar+17].

Zunächst könnte gefolgert werden, dass der einzige Effekt des vorgeschlagenen neuronalen Schemas darin besteht, dass ein Neuron in mehrere unabhängige traditionelle Neuronen aufgeteilt werden muss, entsprechend der Anzahl der Threshold-Units, aus welchen das Neuron besteht. Jede Threshold-Unit besitzt dabei weniger Eingaben als das gesamte Neuron und möglicherweise eine andere Schwelle, weshalb ebenfalls die räumliche Summierung modifiziert werden muss. Die Dynamik der Schwelleneinheiten ist jedoch gekoppelt, da diese das gleiche

Axon teilen und eine gemeinsame Refractory-Periode (Periode in der das Neuron keinen neues Aktionspotential/Spike generieren kann) besitzen könnten. Dies sollte laut den Autoren mit anderen möglichen „Nebeneffekten“ in Zukunft experimentell beantwortet werden. Weiter beschränkte sich die Untersuchung auf Pyramidenneuronen, welche häufig in corticalen Kulturen gefunden werden [Sar+17].

Zusammenfassend zeigen diese Ergebnisse, dass ein Neuron ein komplexeres und strukturiertes Berechnungselement darstellt als erwartet. Dies liefert interessante Implikationen auf die Funktionalität neuronaler Netzwerke; außerdem fordern diese Ergebnisse die Überprüfung der neuronalen Funktionalitäten, der fortgeschrittenen Rechenfähigkeiten und der dynamischen Eigenschaften solcher komplexer Systeme über das traditionellen Framework hinaus. Auch die Rechenkapazität auf einer Netzwerkebene des neuen Schemas sollte im Vergleich zum aktuellen Schema untersucht werden. Allerdings wird erwähnt, dass weitere Experimente durchgeführt werden müssen, um diese Erkenntnisse zu bestätigen und zu vertiefen [Sar+17].

Basierend auf dieser Zusammenfassung kann nun versucht werden, wie im oberen Abschnitt eine Capsule als eine corticale Macrocolumn betrachtet wurde, eine Capsule als ein solches „neues“ Neuron zu betrachten. Hierbei muss erwähnt werden, dass die Inspiration u.a. von Hinton von coticalen Columnen stammt und nicht von diesen kürzlich entdeckten Ergebnissen. Jedoch ist es sehr interessant diesen Vergleich hier anzustellen, um zu verstehen inwiefern sich die beiden Forschungsergebnisse abdecken, auch wenn nicht erwartet wird, dass hierbei eine eindeutige und lückenlose „Abbildung“ gefunden werden kann:
Zu Beginn fällt auf, dass eine Capsule, die aus einfachen Neuronen mit einer Aktivierungsfunktion besteht, welche einen Schwellwert (z.B. ReLu, Ramp oder Step-Funktion) besitzen, als Ansammlung von Threshold-Units betrachtet werden kann, wobei jede subzelluläre Threshold-Unit (jedes „alte“ Neuron in der Capsule) die eingehenden Signale mittels des Schwellwertes aufsummiert. Beim Kapseln der Ergebnisse der Neuronen müsste dann beachtet werden, dass hierbei keine direkte räumliche Summation zwischen ankommenden Signalen zu verschiedenen Threshold-Units stattfindet. Die Gewichte, mittels welcher ein einfaches Neuron aufsummiert, könnten als die Vorverarbeitung der dendritischen Berechnungen betrachtet werden, da diese parallel lokal in jedem Zweig eines Dendriten durchgeführt werden kann, wenn eine einfache gewichtete Verbindung eines Neurons als Dentrit oder dendritischer Zweig angesehen wird. Wenn nun die ausgegebene Aktivität der Capsule als das Signal des Axons betrachtet wird, und die Capsules so miteinander verbunden werden, dass jedes Neuron innerhalb einer Capsule seine Signale von eigenen einzelnen Dendriten oder Haufen benachbarter Dendriten sammelt, würde dieses Capsule-Schema dem neuen Neuronen-Schema sehr nahe kommen. Allerdings

wurde somit eine Capsule genau für diesen Zweck textuell konstruiert, die Capsules in der Literatur besitzen jedoch, wie oben in der generellen Idee erwähnt, Eigenschaften wie blickwinkelinvariante Transformationsmatrizen und einen Routing-Algorithmus, weshalb folglich versucht wird eine Parallele hierfür zu finden.
Die blickwinkel-invariante Transformationsmatrix zwischen zwei Capsules sorgt dafür, die Ausgabe der unteren Capsule durch eine Transformation mit der Ausgabe der oberen Capsule in Verbindung zu bringen (oft durch die Pose). Dieses „in Verbindung bringen“ wird von der unteren Capsule für alle oberen Capsules versucht und der Routing-Algorithmus bewertet meist, wie gut dies den unteren Capsules für die oberen Capsules gelingt (in Abschnitt 3.3 wird dies konkretisiert). Somit erzeugt der Routing-Algorithmus die ihm am besten vorkommende Zuordnung von oberen zu unteren Capsules, weshalb die „Summations“-Richtung einer ganzen oberen Capsule, aber nicht derer subzelluläre Elemente, beschränkt wird, da nun die unteren Capsules ihre Signale zu den am besten passenden oberen Capsules senden und nicht zu den subzelluläre Teilen einer Capsule. Weiter verbindet (kapselt) häufig die obere Capsule alle ankommenden Signale und verwendet diese für eine oder mehrere Ausgaben, jedoch schreibt das neue Neuronenschema vor, dass keine direkte räumliche Summation zwischen ankommenden Signalen zu verschiedenen Threshold-Units (hier abstrakt als subzelluläre Elemente betrachtet) stattfinden darf und dass diese ihre Ausgabe durch ein einzelnes Axon weiterleiten.
Wird allerdings das Routing noch abstrakter betrachtet, wird erkannt, dass dieser nur einige Signale unterer Capsules durch die obere Capsule passieren lässt, und somit eine Route im Netzwerk bildet. Da die Threshold-Units laut Abbildung 3.6 unabhängig voneinander das Signal zum nächsten Neuron weiterleiten können, bildet sich somit ebenfalls eine Route im neuronalen Netz, auch wenn hier offensichtlich zusätzlich noch die Zeitkomponente der Spiking-Neuronmodelle beachtet werden muss.

Zusammenfassend konnte eine Capsule textuell so konstruiert werden, dass diese dem neuen Modell nahekommt, was zeigt, dass die Idee einer Capsule sehr allgemein ist. Weiter wurden einige signifikante Unterschiede zu in der Literatur gängigen Capsules gefunden und eine abstrakte Betrachtung der Vorgänge auf Netzebene gibt mögliches ähnliches Verhalten der Modelle zu erkennen. Jedoch besitzen keine zum Kenntnisstand dieser Arbeit bekannten Capsules die Zeitkomponente der Spiking-Neuronmodelle. Da die Grundidee einer Capsule eher darin besteht mehrere Neuronen vorteilhaft zu kapseln und nicht ein neues Neuron darzustellen, wäre die Untersuchung einer Capsule die aus Neuronen des neuen Modells besteht interessant, vor allem weil die untersuchten Pyramidenneuronen häufig in corticalen Kulturen gefunden werden. Dafür sollte jedoch erst das neue Neuronen-Schema im Bereich des maschinellen Lernens einzeln entwickelt und evaluiert werden.

3.1.5 Introspektive Experimente zur Intuition hinter Capsules

Es existieren zahlreiche Experimente (siehe z.B. [Roc73]) die aufzeigen, wie und wieviel Information ein Mensch aus visuellen Reizen erlangt. Dieser Abschnitt beschreibt zwei dieser Experimente, welche wichtige Inspirationen für Capsule Networks darstellen. Andere Experimente wie das kurze Aufblitzen einer Szene geben zu erkennen, dass ein Menschen aus einer einzelnen (bzw. wenigen) Fixierungen ein breites Spektrum an Information gewinnt [Tra14]. Weiter wird in Abschnitt 3.2.4 ein Experiment durchgeführt, welches ein Limit der visuellen Informationsverarbeitung demonstriert.

Vertraute Formen und mentale Rotation

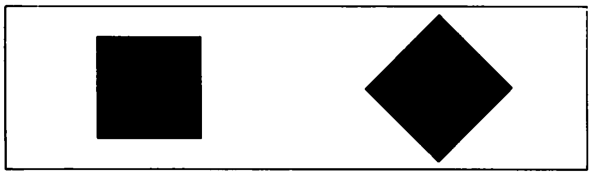


Abbildung 3.7: Rechteck und Diamant [Roc73]

Abbildung 3.7 zeigt zwei vertraute Formen: ein *Rechteck* und ein *Diamant*. Beide hier dargestellten Figuren sind identisch, doch deren Erscheinung wird als so unterschiedlich wahrgenommen, dass die eine *Rechteck* und die andere *Diamant* genannt wird. Zusätzlich erscheinen die Winkel beim *Diamanten* nicht spontan als rechtwinklig [Roc73]; beim *Rechteck* hingegen wird auf das Grad genau erkannt, ob es sich hierbei um einen *rechten* Winkel handelt [Ghtb]. Eine Rotation einer simplen Figur kann somit dafür sorgen, dass diese nicht erkannt wird; der Beobachter ist sich der Rotation nicht bewusst [Roc73] und eine vertraute Form scheint nicht mehr die selbe Erscheinung zu besitzen. Es könnte argumentiert werden, dass eine rotierte, vertraute Form so unterschiedlich erkannt wird, weil diese zuvor kaum in dieser rotierten Form gesehen wurde, jedoch sehen auch vertraute Figuren in unterschiedlichen Orientierungen anders aus, solange sie nicht mental rotiert werden (siehe z.B. [Roc73]). Weiter werden, wie Abbildung 3.8 zeigt, auch unbekannte Formen nach mentaler Rotation zu bekannten Formen [Roc73]:

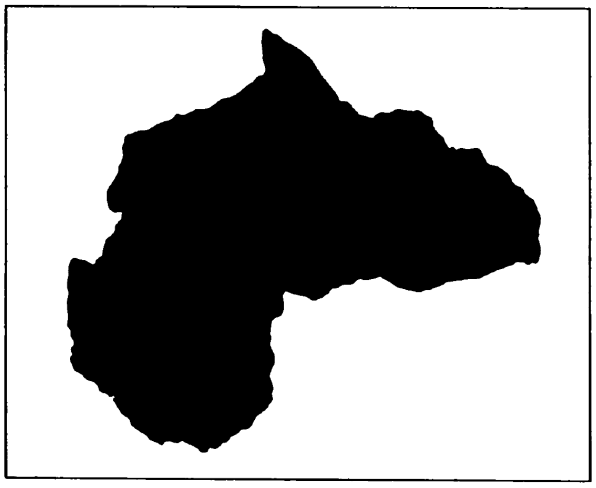


Abbildung 3.8: Unbekannte Form wird nach Mitteilung dass diese rotiert wurde durch mentale Rotation zu Afrika, aus [Roc73]

Das Nichterkennen einer vertrauten Figur in neuer Orientierung basiert auf der Veränderung ihrer wahrgenommenen Form; jedoch kann eine Figur auch ohne Effekt auf ihrer wahrgenommene Form verändert werden. Beispielsweise können Dreiecke in Größe, Farbe und auf vielen anderen Wegen variiert werden, ohne ihre wahrgenommene Form zu ändern. Hierbei kann eine Analogie zu Musik gezogen werden: der Wechsel der Transposition zu einer neuen Tonart. Alle Noten unterscheiden sich danach von der Originalform, führen jedoch zu keiner Veränderung an der Melodie. Eine Melodie entsteht daher offensichtlich aus den Relationen zwischen den Noten, welche in einer Transposition nicht geändert werden. Auf dem selben Weg ist eine visuelle Form abhängig davon, wie deren Teile zueinander in geometrischer Relation stehen. Z.B. kann ein Rechteck als vierseitige Figur mit vier rechten Winkeln und parallele gegenüberliegenden Seiten gleicher Länge beschrieben werden. Wird dieses Objekt rotiert, werden diese Eigenschaften nicht verändert, weshalb das Objekt immer noch wie ein Rechteck aussieht. Doch wieso wird es dann als Diamant wahrgenommen? Um diese Frage zu beantworten veranschaulichte Ivin Rock [Roc73] in weiteren Experimenten, dass einem Objekt verschiedene Orientierungen unter Betrachtziehen der vertikalen und horizontale Richtungen der Umwelt zugeteilt werden könnten und das diese Orientierung für die veränderte Wahrnehmung eine Figur verantwortlich sei, wenn die Figur „desorientiert“ ist [Roc73]. Menschen könnten somit rechtwinklige, im Objekten eingebettete Koordinatensysteme für das Erkennen von Formen nutzen [Ghtb].

Mentale Rotation und Seitigkeit Das folgende Experiment und die oben dargestellten zeigen Hinton, dass die visuellen Systeme von Menschen Entitäten erkennen, indem sie in Entitäten eingebettete, rechtwinklige Koordinatensystem und möglicherweise eine Hierarchie dieser wie

in der Computergrafik aufstellen. Dies wird in den Capsule Networks umgesetzt (Abschnitt 3.1.3). Außerdem liefert das Experiment Beweise darüber, das die Relation zwischen einer Entität und dem Betrachter nicht nur von einem Neuron oder eine grob-codierten Set an Neuronen repräsentiert wird, sondern von einer ganzen Menge aktiver Neuronen [Ghtb] [Ghta]:

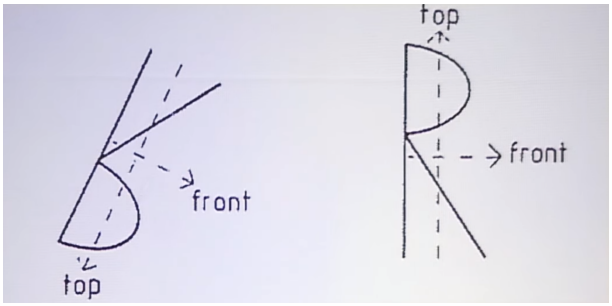


Abbildung 3.9: Ein rotiertes R aus [Ghtb] bzw. [Ghta]

Wird Abbildung 3.9 betrachtet, ist sofort ein rotiertes *R* zu erkennen; allerdings wird nicht sofort erkannt, dass dieses *R* spiegelverkehrt ist. Um zu erkennen, dass dies der Fall ist wird im Denkprozess z.B. die Spitze (*top*) des *Rs* im Uhrzeigersinn nach oben (vom Bild aus betrachtet) rotiert. Auch wenn dieser Prozess schnell abläuft, werden dafür einige Millisekunden mehr Zeit benötigt, als für das bloße Erkennen der Entität. Doch wieso muss eine solche mentale Rotation ausgeführt werden? Hinton argumentiert, dass hierbei die Beziehung zwischen Betrachter und dem *R* bereits bekannt ist - die Pose liegt somit in auf den Betrachter zentrieren Koordinaten vor. Folglich muss erschlossen werden, ob die Matrix welche das intrinsischen Koordinatensystem des *Rs* mit dem des Betrachters in Beziehung bringt rechtsseitig oder linksseitig ist (lose Veranschaulichung: Drei-Finger-Regel für Orientierung). Wenn diese Matrix wie in der Computergrafik in mehreren Nummern repräsentiert wird, muss dafür eine komplexe Rechnung (das Vorzeichen der Determinante der Matrix) ausgeführt werden. Dieses Ergebnis kann jedoch nicht berechnet werden, indem auf individuelle Einträge der Matrix geachtet wird. Hierfür werden alle Einträge auf einmal benötigt, weshalb das Ergebnis nicht direkt von der neuronalen Repräsentation berechnet wird, da neuronale Netze nicht gut bei der Berechnung dieser Art von Problemen sind, woraus weiter gefolgert werden kann, dass das menschliche Gehirn hierbei keine „Seitigkeit“ errechnet. Diese Hypothese wird hier (und von Hinton [Ghtb] [Ghta]) als ein „Beweis“ betrachtet, dass die Repräsentation des aufgestellten Koordinatensystems und die Beziehung zwischen diesem und dem Betrachtet über mehrere Nummern (bzw. Neuronen) verteilt ist, welche verschieden Aspekte dieser Beziehung erfassen. Dadurch können kontinuierliche Transformationen solange angewandt werden, bis die zwei Matrizen in alle bis auf einer Richtung übereinstimmen, was zu einer triviale Entscheidung über die „Seitigkeit“ der Matrix führt. Laut Hinton [Ghtb] [Ghta] findet dies in der mentalen Rotation statt und zeigt

darüber hinaus weshalb diese ausgeführt wird. Die mentale Rotation dient demnach nicht zum Erkennen von Objekten sondern zum Lösen von Problemen bei denen die „Seitigkeit“ unbekannt ist. Es wäre daher kein Problem zu entscheiden, ob die Matrix links- oder rechtsseitig ist, wenn individuelle Neuronen eine vollständige Pose repräsentieren, weil diese Neuronen eine „Seitigkeit“ besitzen würden, so wies es, laut dieser Hypothese, im menschlichen Gehirn der Fall sei [Ghtb] [Ghta] und auch bei Capsules, die in ihrer Aktivität u.a. Orientierung codieren.

Ergebnisse weiterer Experimente wie Schwierigkeiten beim Bilden eines Tetrahedron aus einer zerteilten, dreiseitigen Pyramide (siehe [Ghtb]), Schwierigkeiten beim Lesen rotierter Schriften und Schwierigkeiten beim Erkennen rotierte Gesichter von bekannten Persönlichkeiten (siehe [Roc73]) unterstützen diese Aussagen.

3.2 Transforming-Autoencoder

Hinton et al. [Hin11] führten zuerst das Konzept der Capsule anhand eines Transforming-Autocoders ein, welcher keine komplexe Routing-Strategie besitzt, jedoch zeigt, dass Capsules im ersten Layer so trainiert werden können, dass diese die Eigenschaften einer Entität in ihrem Ausgabevektor beinhalten. In anderen Worten: Es wurde gezeigt, dass die Capsules eines einschichtigen Transforming-Autoencoders in der Lage sind, die Sprache von Pixelinformation in die Sprache von Pose-Parametern zu übertragen [Hin11].

3.2.1 Schematischer Aufbau eines Transforming-Autoencoders

Das Netzwerk in Abbildung 3.10 erhält als Input ein Bild, sowie die gewünschte Transformation in x- und y-Richtung (Δx- und Δy) und gibt das verschobene Eingabebild aus:

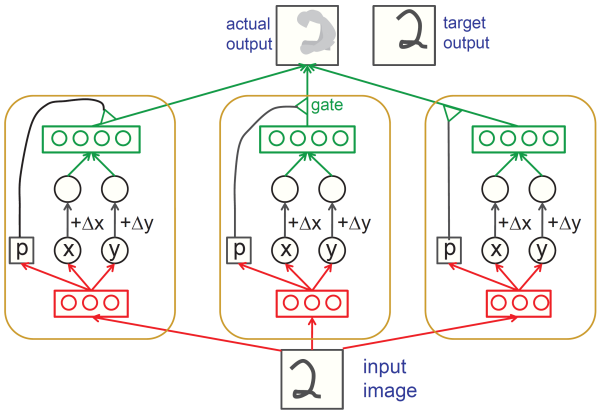


Abbildung 3.10: Schematischer Aufbau eines Transforming-Autoencoders aus [Hin11, S. 3]

Abbildung 3.10 zeigt drei Capsules (*orange*), welche jeweils aus drei *Logistic Recognition Units*

(rot) und vier *Generation Units (grün)* bestehen. Die Gewichte der Verbindungen werden durch backpropagieren der Differenz zwischen tatsächlicher und gewünschter Ausgabe erlernt.
Wird die Architektur von Abbildung 3.10 mit der erarbeiteten Darstellung der generalisierten Idee hinter Capsules (Abbildung 3.1) in Verbindung gebracht, so ist die *Gruppe von Neuronen* von Abbildung 3.1 innerhalb einer Capsule hier in zwei Gruppen unterteilt:

Jede Capsule besitzt ihre eigenen *Logistic Recognition Unit (rot)*, welche als Hidden-Layer zur Berechnung der x- und y-Position - der Poseinformation - und der Existenzwahrscheinlichkeit p einer aus dem Eingabebild visuelle Entität fungiert. Es handelt sich hierbei um die Variante bei der die Länge des Ausgabevektors für die Klassifizierung keine Rolle spielt. Außerdem besteht eine Capsule aus *Generation Units (grün)*, welche dazu verwendet werden den Beitrag der Capsule für das transformierte Bild zu berechnen. Die Eingaben der *Generation Units (grün)* sind die in den *Logistic Recognition Units* berechneten x- und y-Werte aufaddiert mit den Δx- und Δy-Werten der Eingabe des Modells.
Wird das Capsule-Modell des Transforming-Autoencoders mit dem *Kapseln der Ergebnisse für den Output* aus Abbildung 3.1 in Verbindung gebracht, besteht dies aus dem Multiplizieren des Transformationsbeitrags einer Capsule - der Ausgabe der *Generation Unit* - mit der Existenzwahrscheinlichkeit p im *Gate* der Abbildung 3.10, sodass eine inaktive Capsule - eine Capsule die keine Entität repräsentiert - keinen Beitrag zur Ausgabe leistet und daher keinen Effekt hat. Diese Capsules interagieren dann im finalen Layer miteinander, um das verschobene Bild zu produzieren [Hin11, S. 3-4]. Diese Interaktion wird im originalen Paper nicht genauer beschrieben, jedoch wird in den Implementierungen [Meh] und [Pal] eine fully-connected (fc) Layer ohne Aktivierung nach jeder *Generation Unit* eingefügt, der die Ausgabe der *Generation Unit* auf die Größe der Ausgabebilder bringt. Die Ausgabe des fc Layers jeder Capsules wird dann mit der Existenzwahrscheinlichkeit p elementweise multipliziert und als Ausgabe der Capsule behandelt. Alle Capsule-Ausgaben werden letztlich elementweise aufaddiert und einer Sigmoid-Aktivierungsfunktion überreicht.

3.2.2 Anwendung des Transforming-Autoencoders

Damit ein Transforming-Autoencoder die richtigen Ausgabebilder generiert, ist es essentiell, dass die von den *Logistic Recognition Units* jeder aktiven Capsule berechneten x- und y-Werte mit den tatsächlich x- und y-Werten der visuellen Entität korrespondieren; wobei diese Entität und deren Ursprung im Koordinatensystem im Voraus nicht bekannt sein muss.
Abbildung 3.11 veranschaulicht die Ergebnisse eines Autoencoder mit 30 Caspules welche jeweils zehn Recognition Units und 20 Generation Units besitzen und bestätigt damit die korrekte Funktionsweise.

Beim Training sieht jede Capsule das gesamte MNIST-Eingabebild. Sowohl die Eingabebilder als auch die Ausgabebilder (Label) wurden zufällig um $-2, -1, 0, 1$ oder 2 in x- oder y-Richtung verschoben und dem Transforming-Autoencoder wird die daraus folgenden Δx- und Δy-Werte der gewünschten Verschiebung zusätzlich als Input überreicht. Dieses Überreichen der Verschiebungsinformationen erscheint aus reiner Machine-Learning Perspektive unnötig, da die Verschiebung im Prinzip aus den zwei Bildern berechnet werden könnte, jedoch ist diese Art von Information meist leicht verfügbar und das Lernen wird dadurch vereinfacht [Hin11]. Außerdem sollen Transforming-Autoencoder nur zeigen, dass Instanziierungsparameter einer Entität aus Pixelinformationen extrahiert werden können, weshalb es genügt eine bekannte Transformation (hier z.B. -2 in x-Richtung) auf die Pose-Ausgabe der *Recognition Units* einer Capsule anzuwenden.

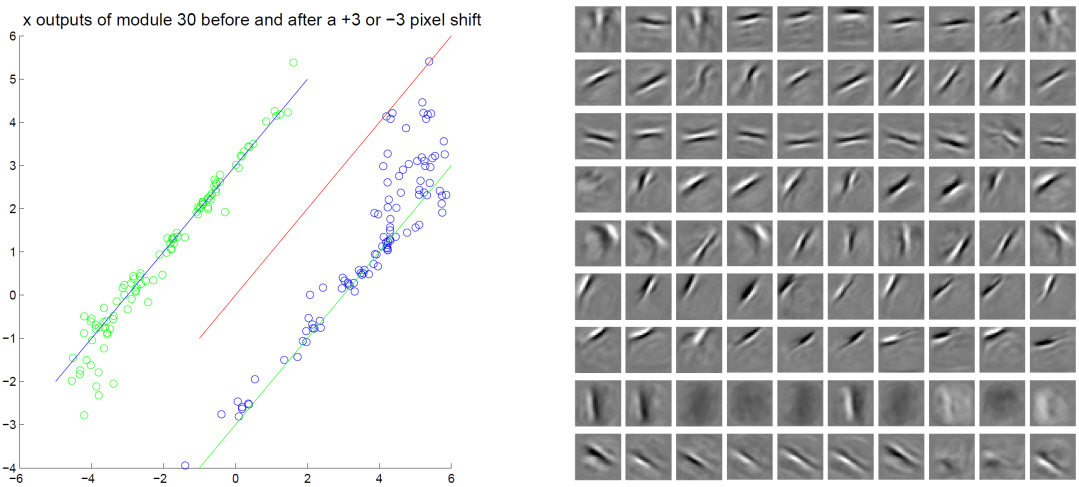


Abbildung 3.11: Links: Scatterplot mit Verschiebungsfehler der Recognition Unit einer Capsule. Rechts: die ausgehenden Gewichte verschiedener Generation Units mehrere Capsules aus [Hin11, S. 3]

Rechts zeigt Abbildung 3.11 die ausgehenden Gewichte von zehn der 20 *Generation Units* für neun Capsules; links stellt sie einen Scatterplot dar, in welchem die vertikale Achse den Output einer Capsule für jedes MNIST-Eingabebild zeigt und die horizontale Achse den x-Output der selben Capsule wenn das Eingabebild $+3$ oder -3 in der x-Richtung verschoben wird - also etwas mehr wie zuvor gelernt. Wenn sich das Originalbild bereits nahe der Grenze der x-Positionen befindet, welche die Capsule repräsentieren kann, würde eine weitere Verschiebung in dieser Richtung bewirken, dass die Capsule eine falsche Antwort erzeugt; solange die Capsule jedoch die Existenzwahrscheinlichkeit p für Daten außerhalb ihres Kompetenzbereiches auf null setzt, spielt dies keine Rolle [Hin11, S. 4].
Im Scatterplot ist zu erkennen, dass die *Recognition Units* der Capsule, tatsächlich x- und y-Werte ausgeben, welche sich genau dann verschieben, wenn das Eingangsbild verschoben wird.

Hinton et al. [Hin11] führten weiter aus, dass Capsules *Generation Units* mit projektiven Feldern lernen, die stark lokalisiert sind und dass die rezeptiven Felder der *Recognition Units* etwas mehr Rauschen verzeichnen und etwas weniger lokalisiert sind [Hin11, S. 3-4]; er begründet dies jedoch anhand der Abbildung 3.11 nicht weiter. Womöglich kann dies an der etwas stärkeren Streuung der Punkte im Scatterplott für die *Recognition Units* nach der Verschiebung und dem geringen Rauschen der ausgehenden Gewichte der *Generation Units* erkannt werden. Weiter enthalten vermutlich die ausgehenden Gewichte der *Generation Units* (bzw. der Capsules), welche in der Abbildung 3.2 aus Abschnitt 3.1.3 dargestellt werden, Informationen über verschiedene Teile von Zahlen, wie beispielsweise die Position oder Schräge einer Vier oder Sieben.

3.2.3 Transforming-Autoencoder in höheren Dimensionen

Wird die Ausgabe der *Recognition Units* als Matrix interpretiert (hier als 3x3 Matrix), so können komplexere 2D-Transformationen oder Änderungen im 3D-Blickwinkel modelliert werden. Das Training eines Transforming-Autoencoders kann beispielsweise dazu führen, dass dieser eine vollständige affine Transformation (Translation, Rotation, Skalierung und Scherung) vorhersagen kann. Dabei wurde in Analogie zu Abschnitt 3.2.2 eine bekannte Transformationsmatrix T auf den Pose-Ausgabe A der *Recognition Units* einer Capsule angewandt, um die Matrix TA zu erhalten, welche als Eingabe für die *Generation Units* verwendet wird, wenn das Zielausgabebild vorhergesagt wird. Abbildung 3.12 zeigt dies für MNIST-Bilder.

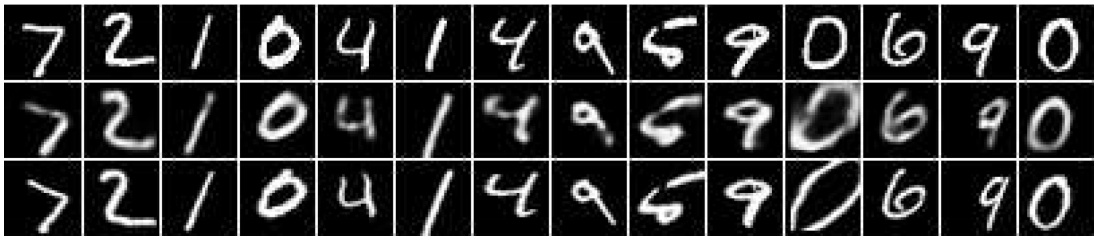


Abbildung 3.12: Vollständige Ausgabe der affin transformierten Bilder eines Transforming-Autoencoders aus [Hin11, S. 5]

Für Abbildung 3.12 wurde kurz erläutert ein Transforming-Autoencoder mit 25 Capsules verwendet, welche jeweils aus 40 *Recognition Units* und 40 *Genaration Units* bestehen. Die *obere Reihe* der Abbildung 3.12 zeigt MNIST-Eingabebilder; die *mittlere Reihe* Ausgabebilder des Transforming-Autoencoder; die *untere Reihe* gewünschte, korrekt transformierte Ausgabebilder [Hin11, S. 5]. Es ist zu erkennen, dass der Transforming-Autoencoder die Eingabebilder erfolgreich um die gewünschten Werte transformiert, jedoch dabei etwas Rauschen entsteht, welches allerdings in keinem der Ausgabebilder so stark ist, dass die Zahl unkenntlich wird.

In Abbildung 3.13 wurde ebenfalls analog ein Transforming-Autoencoder trainiert, um Stereobilder unterschiedlicher Fahrzeugtypen von verschiedenen Blickwinkeln zu generieren. Links sind Eingabe-, Ausgabe- und Zielausgabe-Paare für Trainingsdaten dargestellt, rechts die Paare für Testdaten. Auch hier konnten die gewünschten Transformationen mit etwas Rauschen durchgeführt werden.

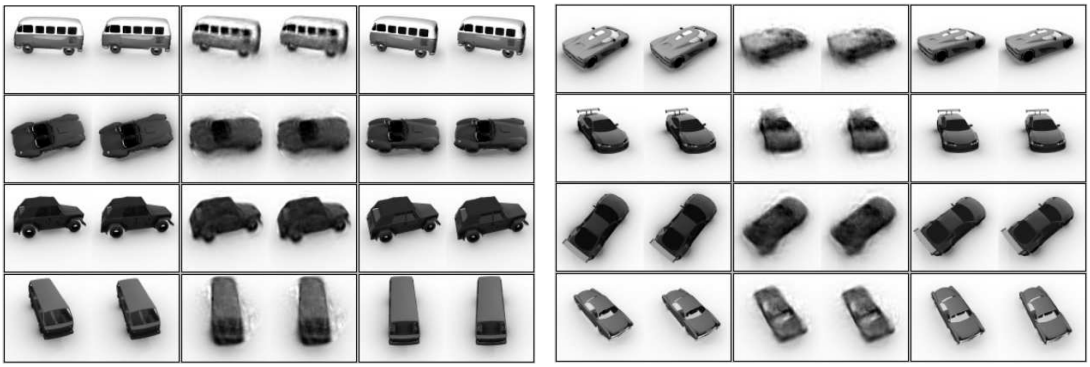


Abbildung 3.13: Vergleich der Ausgabe eines Transforming-Autoencoder für Stereobilder von verschiedener Fahrzeugtypen von verschiedenen Blickwinkeln [Hin11, S. 6]

Der Aufbau dieses Transforming-Autoencoder wurde in [Hin11, S. 5] nur kurz beschrieben: Dieser bestand aus 900 Capsules mit je zwei Layern (32, dann 64) von *Rectified Linear Recognition Units*. Die Capsules hatten 11x11 Pixel rezeptive Felder, welche auf einem 30x30-Raster (entspr. 900 Capsules) über dem 96x96-Bild angeordnet waren und einen Stride von drei Pixeln zwischen benachbarten Capsules besaßen (vermutlich waren die $30 \times 30 = 900$ rezeptiven Felder (11x11) mit einem Stride von drei in der Eingabe entfernt). Außerdem existierte kein Weight-Sharing. Jede Capsule produzierte eine 3x3 Matrixrepräsentation der 3D-Orientierung ihres Features und eine Wahrscheinlichkeit ob dieses implizit definierten Features präsent sind. Diese Matrix Pose-Repräsentation wurde dann analog zu obigem Vorgehen mit der Transformationsmatrix zwischen Eingabe und gewünschter Ausgabe multipliziert und die Ausgabe in einen Layer aus 128 *Generative Rectified Linear Units* geleitet. Weiter wurde analog zum obigem Vorgehen die Ausgabe der *Genaration Units* mit der Wahrscheinlichkeit der Features multipliziert. Dieses Ergebnis wurde dann verwendet um die Intensitäten in einem 22x22-Patch des rekonstruierten Bildes zu erhöhen, welches in der Mitte des 11x11-rezeptiven Feldes der Capsule zentriert war [Hin11, S. 5-6]. Jede Capsule füllt somit einen 22x22-Teil des transformierten Bildes, doch da die Daten aus Stereopaaren von Bildern bestanden, musste jede Capsule den 11x11-Patch beider Mitglieder des Stereopaars betrachten und einen 22x22-Patch für beide rekonstruieren [Hin11, S. 5-6]. Für mehr Details müsste die Implementierung erläutert werden, von dem abgesehen wird, da hier Transforming-Autoencoder nur als konzeptuelle Grundlage für CapsNets beschrieben werden sollen.

3.2.4 Einige Erkenntnisse durch Transforming-Autoencoder

Das Nutzen von echten Werten ist der „natürliche“ Weg Pose-Informationen zu repräsentieren [Hin81a], jedoch tritt hierbei ein Problem auf: Das einzige was diese Werte zusammenbindet ist der Fakt, dass diese sich in der Ausgabe derselben Capsule befinden. Da es sich hierbei um Instanziierungsparameter einer Entität handelt, ist es einer Capsule nicht möglich mehrere Instanzen derselben Entität darzustellen, was als ernste Schwäche angesehen werden kann. Diese Schwäche ist teilweise umgehbar, indem die niedrigleveligen Capsules nur über einen sehr begrenzten Instanziierungsparameterraum (u.a. Posenraum) operieren und nur den Capsules, die komplexeren Entitäten repräsentieren größere Regionen zugeschrieben werden. Doch egal wie klein die Region ist, das System kann immer verwirrt werden indem zwei Instanzen derselben Entität mit nur leicht veränderten Posen nebeneinander gestellt werden. Dieses Phänomen trägt den Namen „Crowding“ [Hin11, S. 6] und in [PAT08] wird anhand einer Vielzahl von Experimenten gezeigt, dass diese Verwirrung auch bei Menschen auftreten kann. Abbildung 3.14 zeigt eines dieser Experimente, welches mittels Introspektion durchführbar ist:

Abbildung 3.14: Auswirkungen des Crowdings aus [PAT08]

Hierbei müssen die Augen auf das rote Minus von Abbildung 3.14 fixiert werden während sich Gedanken zur folgenden Fragen gemacht werden soll: 1. Kann der Unterschied beider Cluster hinsichtlich Buchstabenanzahl und Position erkannt werden 2. Kann der Unterschied der Anzahl von *A*- und *B*-Buchstaben erkannt werden?
Crowding beeinträchtigt genau die Fähigkeit diese Objekteigenschaften zu beurteilen [PAT08], was durch die eigene Durchführung erkenntlich wird. Selbst wenn nur der unterste oder oberste Buchstabe eines Clusters fixiert wird, fällt es bereits schwer diese Fragen ausschließlich für dieses eine Cluster zu beantworten. Jedoch kann argumentiert werden, dass ein Mensch dieses Problem durch Ausführen mehrere Sakaden löst, da Capsule Networks allerdings wie in Abschnitt

3.1.2 beschrieben auf einer einzigen Fixierung operieren ist es nur fair dieses Problem mit einer einzigen Fixierung eines Menschen zu vergleichen. Ob Capsule Networks durch Kombination der Ergebnisse über mehrere Sakanden dieses Problem lösen können, ist zum Kenntnisstand dieser Arbeit noch nicht untersucht worden, jedoch zeigt dieses Experiment eine interessante Übereinstimmung zwischen der visuellen Informationsverarbeitung eines Menschen und eines CapsNets hinsichtlich einer Fixierung.

Weiter zeigt der Transforming-Autoencoder, dass Capsules *Recognition Units* dazu verwenden können, um präzise Koordinaten bis zu einem kleinen Bruchteil eines Pixels zu errechnen. So betrachtet kann eine Capsule mit einem Steerable Filter [FA91] verglichen werden. Steerable Filter entstehen aus einer Methode, welche Filter willkürlicher Orientierungen aus Linearkombinationen von Basisfiltern erstellt und erlaubt diese adaptiv in jede Richtung zu steuern (engl. „steer“). Analytisch betrachtet wird der Filter-Output als Funktion der Orientierung berechnet [FA91, S. 1]. Im Gegensatz zu den meisten Steerable Filter lernt eine Capsule zusätzlich rezeptive Felder für ihre *Recognition Units*, um die Genauigkeit der errechneten Koordinaten (bzw. einer Orientierung) zu optimieren. Außerdem lernt eine Capsule automatisch welche visuelle Entität sie repräsentiert.
Zusätzlich können noch weitere Parallelen gefunden werden, wie beispielsweise zu Kalman-Filtern oder zum gaußschen Rauschen [Hin11, S. 7]. Diese werden im Rahmen dieser Arbeit jedoch nicht weiter erläutert.

Ein Transforming-Autoencoder kann die Ausgabe einer Capsule dazu forcieren, jede Eigenschaft eines Bildes zu repräsentieren, welche auf einem bekannten Weg manipulierbar ist. Um beispielsweise die Beleuchtungshelligkeit zu ändern, können alle Pixelintensitäten mittels eines Helligkeits-Skalierungsfaktor erhöht werden. Wenn analog zum oben beschriebenen Verschieben einer Zahl, eine Capsule des ersten Levels einen Wert ausgibt, der zuerst mit dem Helligkeits-Skalierungsfaktor multipliziert und schließlich dazu verwendet wird die Ausgaben ihrer *Generation Units* bei der Vorhersage der helligkeits-transformierten Ausgabe zu skalieren, kann dieser Wert lernen, die Helligkeit zu repräsentieren und es der Capsule erlauben, die Existenzwahrscheinlichkeit einer Entität von der Helligkeit der Entität zu trennen. Ist weiter die Richtung des Lichtes einer Szene kontrolliert veränderbar und die betrachtete Entität komplex genug, könnte eine Capsule in diesem Fall dafür trainiert werden, die Lichtrichtung mittel zweier Zahlen aus den *Recognition Units* zu extrahieren [Hin11, S. 8].

Nachdem gezeigt wurde wie eine Capsule das Extrahieren von Instanziierungsparameter ihrer visuellen Entität durch Vorgeben der Veränderung an diesen Parametern lernt, werden in

den folgenden Abschnitten Modelle vorgestellt welche komplexere Entitäten durch Übereinstimmungen von niedrigleveligen Capsules klassifizieren und dafür keine Parametervorgabe benötigen, sondern diese direkt aus der Eingabe erlernen.

3.3 Capsule Network

2017 stellten Hinton et al. in [SS17] das erste Capsule Network mit einem dynamischen, iterativen Routing-By-Agreement-Algorithmus zwischen einzelnen Capsules vor, welches für große Resonanz sorgt und bereits vielseitig weiterentwickelt und eingesetzt wird. Der folgende Abschnitt erläutert dieses CapsNet.

3.3.1 Schematischer Aufbau einer Capsule mit Iterative Routing-by-Agreement

Um verstehen zu können, welche Berechnungen in einer Capsule stattfinden und wie das iterative Routing-By-Agreement zwischen einzelnen Capsules abläuft, wurde das in [SS17] beschriebene Verfahren in die generalisierte Idee dieser Arbeit gebracht. Hierbei ist zu erwähnen, dass wie in [SS17] das Routing als Teil einer Capsule gesehen wird, da dies der Vorstellung einer funktionsfähigen, abgeschlossenen Ansammlung an Neuronen näher kommt:

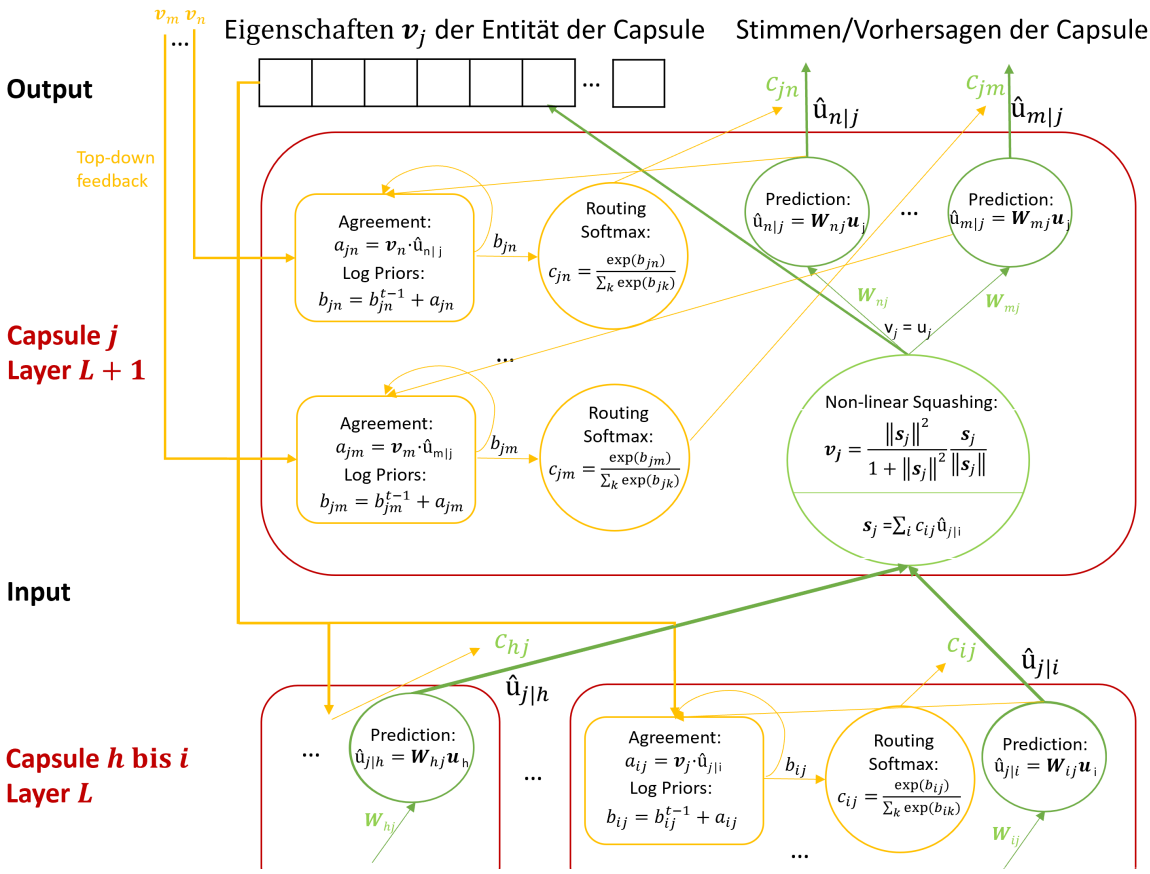


Abbildung 3.15: Schematischer Aufbau einer Capsule mit iterativem Routing-by-Agreement

Abbildung 3.15 mag erstmal überfüllt wirken, sollte jedoch nach genauer Betrachtung einen vollständigen Überblick über die Funktionsweise dieser Capsules geben. Der mit *Grün* dargestellte Teil stellt den *Feedforward-Pfad* einer Capsule dar; der mit *Gelb* gezeichnete das Anpassen sogenannter Coupling-Koeffizienten durch das *Routing-By-Agreement*, und somit einen unüberwachten Teil des Lernens. Zuerst wird der *grüne Pfad* in Abbildung 3.15 von unten nach oben beschrieben. Dabei wird angenommen, dass die Capsules $h, i, usw.$ aus dem Layer L bereits ihre Ausgaben berechnen: Eine Capsule in Layer L berechnet für jede höherlevelige Capsule (hier $L + 1$), mit der sie verbunden ist, ihre Ausgabe, folglich bekommt eine Capsule in Layer $L + 1$ von mehreren Capsules im niedrigeren Layer Eingaben. Da in [SS17] keine Layer übersprungen werden, senden in Abbildung 3.15 die Capsules $h, i, usw.$ aus Layer L ihre Outputs zu den Capsules in Layer $L + 1$ (u.a. zur Capsule j).
Um die Ausgabe zu konkretisieren wird die rechte Capsule i in Layer L nun genauer betrachtet: Diese berechnet zuerst eine Vorhersage (engl. Prediction) $\hat{u}_{j|i}$ für die Ausgabe/Aktivität der Capsule j aus Layer $L + 1$ wie folgt:

$$\hat{\mathbf{u}}_{j|i} = \mathbf{W}_{ij}\mathbf{u}_i \tag{3.1}$$

Die Vorhersage der Capsule i ergibt sich aus der Multiplikation des Output $\mathbf{u}_i$, sprich ihrer Aktivität - also der Vektor der die Instanziierungsparameter der Entität der Capsule i repräsentiert - mit einer Gewichtsmatrix $\mathbf{W}_{ij}$ [SS17, S. 2], welche über Backpropergation gelernt werden kann. Alle anderen Capsules in Layer L gehen analog vor.
Die Gewichtsmatrix $\mathbf{W}_{ij}$ kodiert wichtige räumliche und andere Beziehungen zwischen den Features aus Layer L (z.B. Auge, Nase und Mund) und den höheren Features (z.B. ein Gesicht). Wird dem Beispiel weiter gefolgt, so könnte die Matrix $\mathbf{W}_{ij}$ die Beziehung zwischen Nase und Gesicht codieren: Das Gesicht ist beispielsweise zehn mal größer also die Nase und seine Orientierung im Raum korrespondiert mit der Orientierung der Nase im Raum, weil sie auf der selben Ebene liegen. Anders ausgedrückt: Die Vorhersage der Capsule in Layer $L + 1$ repräsentiert die Lage (und andere Eigenschaften) des Gesichtes anhand der detektierten Lage (und andere Eigenschaften) der Nase. Analog dazu stellen die Gewichtsmatrizen der anderen Capsules in Layer L die Beziehung zwischen Auge-Gesicht und Mund-Gesicht dar. Sollten all diese Vorhersagen „in die selbe Richtung und zum selben Zustand“ eines Gesichtes zeigen, handelt es sich um ein Gesicht [Pecb].
Die berechneten Vorhersagen werden folglich an die Capsule j aus Layer $L + 1$ gesendet, hierbei werden diese mit Coupling-Koeffizienten gewichtet, welche von der zugehörigen Capsule in Layer L durch Routing-by-Agreement gelernt werden. Das Routing-by-Agreement wird im Abschnitt 3.3.2 beschrieben. Capsule j summiert dann die gesendeten, gewichteten Vorhersagen von Layer L wie folgt auf:

$$\mathbf{s}_j = \sum_i c_{ij} \hat{\mathbf{u}}_{j|i} \tag{3.2}$$

Der totale Input $\mathbf{s}_j$ von Capsule j ist eine gewichtete Summe über alle Vorhersagevektoren der Capsules aus dem Layer darunter [SS17, S. 2]. Die Gewichtung findet wie bereits erwähnt über die Coupling-Koeffizienten statt; da es sich bei einer Vorhersage um einen Vektor handelt ist die Multiplikation elementweise. Würden die Capsules h bis i im ersten Layer liegen, so würden diese ihre Eingaben nicht mit Coupling-Koeffizienten aufsummieren [SS17], da die Eingabe des Layers erst von Capsules in Entitäten unterteilt werden müsste und daher noch keine Orientierungen existieren, auf welche sich die Capsules einigen könnten [SS17, S. 4].

Da nun Capsule j ihre Inputs errechnet hat, wird diese weiter betrachtet:
Damit die Länge des Ausgabevektor $\mathbf{v}_j$ der Capsule j die Existenzwahrscheinlichkeit der Entität darstellt, werden die aufsummierten Vorhersagen $\mathbf{s}_j$ einer nicht-linearen Quetschfunktion (engl. Non-linear Squashing) überreicht, die wie folgt definiert ist:

$$\mathbf{v}_j = \frac{\|\mathbf{s}_j\|^2}{1 + \|\mathbf{s}_j\|^2} \frac{\mathbf{s}_j}{\|\mathbf{s}_j\|} \tag{3.3}$$

Der rechte Bruch der Gleichung skaliert den Eingabevektor so, dass er eine Einheitslänge besitzt; der linke Teil führt eine zusätzliche Skalierung durch [Pecb], damit die Ergebnislänge kurzer Vektoren zu fast null schrumpft und lange Vektoren gerade so unter der Länge eins bleiben (deshalb die $1+$ unter dem Bruch). Ein Graph dieser Funktion wird in Abbildung 3.16 dargestellt:

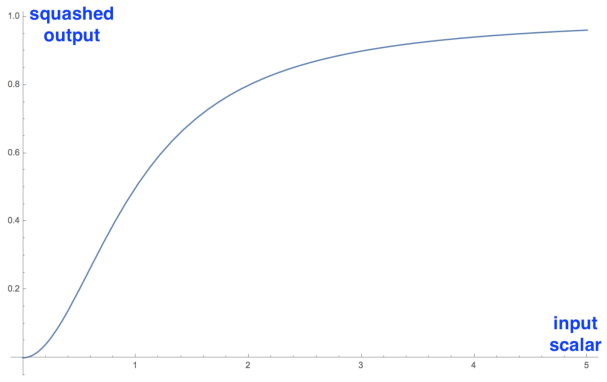


Abbildung 3.16: Graph eines Non-linear Squashing in Skalarform aus [Pecb]

Abbildung 3.16 lässt für einen eindimensionale Vektor erkennen, dass wie in [SS17, S. 2] beschrieben, die nichtliniare Squashing-Funktion die Länge kurzer Vektoren zu fast null schrumpft und lange Vektoren gerade so unter die Länge eins bringt. Der Vektor $\mathbf{v}_j$ ist i.d.R. nicht eindimensional, aber eine Visualisierung mehrdimensionale Vektoren wäre schwierig. Der Vektor $\mathbf{v}_j$

ist der Ausgabevektor der Capsule und enthält die Eigenschaften der Entität in seiner Orientierung (den Einträgen des Vektors), sowie die Existenzwahrscheinlichkeit der Entität in seiner Länge [SS17, S. 1]. Nun kann Capsule j analog zur Capsule i mit ihrem Vektor $\mathbf{v}_j$ die Vorhersagen ($\hat{\mathbf{u}}_{n|j} = \mathbf{W}_{nj}\mathbf{u}_j, \hat{\mathbf{u}}_{m|j} = \mathbf{W}_{mj}\mathbf{u}_j, ...$) treffen, welche die Vorhersagen für die Capsules $n, m, ...$ im nächsten Layer $L+2$ darstellen und wobei $\mathbf{v}_j = \mathbf{u}_j$ gilt. Handelt es sich bei Capsule j um eine Capsule in letzten Layer ist dieser Schritt nicht weiter nötig und die Länge von $\mathbf{v}_j$ kann beispielsweise als Existenzwahrscheinlichkeit der Entität mit den Existenzwahrscheinlichkeit anderer Capsules in diesem Layer verglichen werden, um eine Klassifizierung zu erzielen.

Da Capsule j nun ihre Ausgabe/Aktivität $\mathbf{v}_j$ errechnet hat, können die Capsules $h, i, ...$ aus Layer L ihre Vorhersagen prüfen, um ihre Coupling-Koeffizienten via Routing-By-Agreement zu adaptieren; dies wird im nächsten Abschnitt beschrieben.

3.3.2 Dynamic Routing-by-Agreement zwischen Capsules

Zunächst muss die Übereinstimmung (engl. Agreement) zwischen Capsule i und j hinsichtlich der Aktivität (z.B. die Pose einer Entität) berechnet werden. Hierzu wird den *gelb umrandeten Einträgen der Capsule* i in Abbildung 3.15 gefolgt, welche gerade die Aktivität $\mathbf{v}_j$ von Capsule j aus Layer $L + 1$ als Top-Down-Feedback erhalten hat.

$$a_{ij} = \mathbf{v}_j \hat{\mathbf{u}}_{j|i} \tag{3.4}$$

Die Übereinstimmung ergibt sich aus dem Skalarprodukt zwischen der Ausgabe $\mathbf{v}_j$ der Capsule j und der Vorhersage $\hat{\mathbf{u}}_{j|i}$ von Capsule i für Capsule j [SS17, S. 3]. Das Skalarprodukt zweier Vektoren ist null, wenn diese senkrecht zueinander stehen, und maximal, wenn diese die gleiche Orientierung besitzen; somit ist das Agreement a_{ij} maximal wenn die Vorhersage der Capsule i mit der Aktivität der Capsule j übereinstimmt.

Nun wird die Übereinstimmung a_{ij} als eine logarithmierte Wahrscheinlichkeit behandelt und mit einer logarithmierten a-priori-Wahrscheinlichkeit b_{ij} aufaddiert:

$$b_{ij} = b_{ij}^{t-1} + a_{ij} \tag{3.5}$$

Diese Wahrscheinlichkeit bestimmt wie „stark“ Capsule i mit Capsule j gekoppelt werden soll und kann initial auf einen Wert gesetzt werden (z.B. alle auf denselben Wert null). Alle b_{ij} werden diskriminativ zur selben Zeit wie die anderen Gewichte gelernt und hängen von dem Ort und dem Typ der zwei verbundenen Capsules ab, jedoch nicht vom Eingabebild [SS17, S. 3].

An dieser Stelle muss kurz erwähnt werden, dass die b_{ij}s im Paper gerne als Routing-Logits

bezeichnet werden. Ein Logit ist ein logarithmierter Odd (engl. für Chance), welcher angibt ob ein Ereignis eintritt (z.B. $5:4$, sprich 5 zu 4, dass ein Team ein Spiel gewinnt). Wichtig hierbei ist, dass eine Chance im strengen Sinne keine Wahrscheinlichkeit ist, da ein Odd nicht zwangsläufig zwischen 0 und 1 liegt ($5:4=\frac{5}{4}=1.25$). Werden weitere Odds $X:Y$ ausgerechnet kann erkannt werden, dass wenn $X <= Y$ alle Werte zwischen 0 und 1 liegen, für $X >= Y$ liegen die Werte jedoch zwischen 1 und ∞. Auf eine Zahlenstrahl betrachtet ist dies unsymmetrisch, was einen Vergleich von Odds erschwert, weshalb logarithmierte Odds, sprich Logits verwendet werden ($log(\frac{5}{4}) \approx 0.97$ und $log(\frac{4}{5}) = -log(\frac{5}{4}) \approx -0.97$). Werden weitere zufällige Logits errechnet, ergeben diese auf dem Zahlenstrahl eine Normalverteilung (Gaußverteilung), welche in der Statistik und somit auch im Deep Learning gerne gesehen wird. Weiter kann ein Odd mit $\frac{p}{1-p}$ aus Wahrscheinlichkeiten berechnet werden. Wenn p beispielsweise die Wahrscheinlichkeit zu gewinnen ist, ergibt sich $p=\frac{5}{9}\approx 0.56$ und der Odd wird wie folgt berechnet: $\frac{\frac{5}{9}}{1-\frac{5}{9}} = 1.25 = \frac{5}{4}$). Der Logit L ist somit durch $L = log(\frac{p}{1-p})$ berechenbar. Zusammengefasst sorgen Logits für Symmetrie, bessere Interpretierbarkeit und vereinfachen somit das statistische Vorgehen. Bezüglich des CapsNets sind die b_{ij}s nicht zwangsläufig 1 weshalb diese als Logits betrachtet werden, die folgenden Coupling-Koeffizienten c_{ij} hingegen als Wahrscheinlichkeiten [GM].

Damit nun aus b_{ij} der Coupling-Koeffizienten zwischen Capsule i und j errechnet werden kann, wird dieser einer *Routing-Softmax* übergeben, die wie folgt definiert ist:

$$c_{ij} = \frac{exp(b_{ij})}{\sum_k exp(b_{ik})} \tag{3.6}$$

Diese *Routing-Softmax* stellt sicher, dass alle k Coupling-Koeffizienten von Capsule i aufsummiert 1 ergeben und nicht 0 und ist zugleich, wie die Logistische Funktion $\frac{exp(G)}{1+exp(G)}$, eine Umkehrfunktion eines Logits G zu einer Wahrscheinlichkeit, ein weiterer Grund weshalb a_{ij} als Wahrscheinlichkeit und b_{ij} als Logit betrachtet wird.

Hiermit wurde eine Routing-Schleife zwischen Capsule i und j dargestellt. Nachdem Abbildung 3.17 das Routing-by-Agreement zusammenfasst und abstrahiert, wird das Routing für alle Capsules mittels des Pseudocodes aus Abbildung [SS17, S. 3] erläutert.

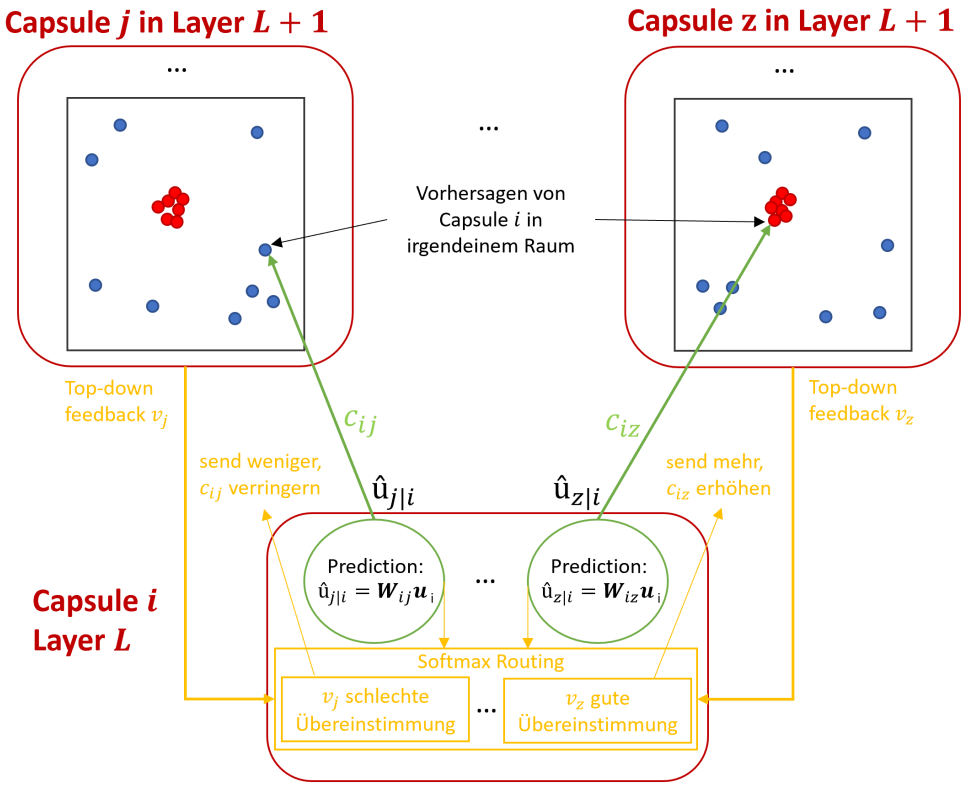


Abbildung 3.17: Eine Capsule i trifft Vorhersagen und bewertet diese (durch Routing-by-Agreement) inspiriert von [Ghtb]

Abbildung 3.17 zeigt nach [Ghtb] eine Capsule j welche entscheiden muss, zu welcher höher liegenden Capsule sie wieviel ihrer Ausgabe sendet. Diese Entscheidung wird getroffen, indem Capsule j ihre Coupling-Koeffizienten anpasst. Genau diese Entscheidung - dieses Anpassen - ist die Essenz des Routing-Algorithmus. Multipliziert Capsule i ihren Output mit den entsprechenden Gewichtsmatrizen W für die Vorhersagen $\hat{\mathbf{u}}$, so landen diese in den dargestellten Räumen der höher liegenden Capsules. Jeder *rote* und *blaue Punkt* entspricht dort der Vorhersage einer Capsule im niedrigerem Layer. Stimmen viele Vorhersagen überein bilden sich Cluster. Die Vorhersage von Capsule i landet in Abbildung 3.17 sehr nahe am Cluster von Capsule z und weit entfernt vom Cluster von Capsule j. Damit Capsule i ihre Coupling-Koeffizienten dementsprechend anpassen kann, besitzt diese einen Mechanismus der misst welche höher liegende Capsule ihre Vorhersage „gut unterbringt“ (z.B. das beschriebene Skalarprodukt mit dem Top-Down-Feedback $\mathbf{v}_j$ und $\mathbf{v}_z$). Durch diesen Mechanismus passt die Capsule automatisch ihre Coupling-Koeffizienten an, wobei in diesem Fall der Coupling-Koeffizienten zu Capsule z erhöht und der zu Capsule j erniedrigt wird.

Procedure 1 Routing algorithm.

1: **procedure** ROUTING($\hat{\boldsymbol{u}}_{j|i}$, r, l)
2: for all capsule i in layer l and capsule j in layer $(l+1)$: $b_{ij} \leftarrow 0$.
3: **for** r iterations **do**
4: for all capsule i in layer l: $\mathbf{c}_i \leftarrow \texttt{softmax}(\mathbf{b}_i)$ ▷ `softmax` computes Eq. 3
5: for all capsule j in layer $(l+1)$: $\mathbf{s}_j \leftarrow \sum_i c_{ij}\hat{\mathbf{u}}_{j|i}$
6: for all capsule j in layer $(l+1)$: $\mathbf{v}_j \leftarrow \texttt{squash}(\mathbf{s}_j)$ ▷ `squash` computes Eq. 1
7: for all capsule i in layer l and capsule j in layer $(l+1)$: $b_{ij} \leftarrow b_{ij} + \hat{\mathbf{u}}_{j|i}.\mathbf{v}_j$
return $\mathbf{v}_j$

Abbildung 3.18: Der Routing-by-Agreement-Algorithmus aus [SS17, S. 3]

Der Routing-Algorithmus 3.18 wird folglich Zeile für Zeile erläutert:
Als Eingabe dienen der Parameter $\hat{\mathbf{u}}_{j|i}$ - die Vorhersagen der Capsules aus Layer l für die Capsule $l+1$ - die Anzahl an Routing-Iterationen r und der aktuelle Layer-Index l. Für jede Routing-Iterationen r wird von allen Capsules das oben beschriebene Routing-By-Agreement ausgeführt, um genauere Übereinstimmungen zu erhalten. Im Algorithmus wird für alle Capsules i in Layer l und für alle Capsules j in Layer $l+1$, daher für alle Verbindungen zwischen den Capsules von Layer l und $l+1$, b_{ij} initial auf null gesetzt. Danach wird für alle Verbindungen für r Iterationen folgendes berechnet:

- (*Zeile 4*) Für alle Capsules i in Layer l: die Coupling-Koeffizienten $\mathbf{c_i}$ aller Verbindungen aus $\mathbf{b_i}$ unter Nutzung der Softmax Funktion und von Vektoroperationen.
- (*Zeile 5*) Für alle Capsules j in Layer $l+1$: die ungequetschte Aktivität $\mathbf{s}_j$ aus den Coupling-Koeffizienten und den Vorhersagen.
- (*Zeile 6*) Für alle Capsules j in Layer $l+1$: die gequetschte (squashed) Aktivität v_j. Also die Ausgaben der Capsules in Layer $l+1$.
- (*Zeile 7*) Für alle Verbindungen: die neuen logarithmierten a-priori-Wahrscheinlichkeit b_{ij}, indem die alte Wahrscheinlichkeit mit den Agreement aufaddiert wird.

Die Rückgabe wäre hier penibel betrachtet die Ausgabe der letzten Capsule in Layer $l+1$, Ziel war das Routing zwischen den Capsules. Alternativ und wie in den meisten Implementierungen (z.B. [Saba]) könnten auch die Aktivitäten aller Capsules aus Layer $l+1$ zurückgegeben werden. Zusätzlich muss an dieser Stelle erwähnt werden, dass der Routing-by-Agreement-Algorithmus, so wie in Abbildung 3.18 dargestellt, die Coupling-Koeffizienten der Capsules aus Layer l, jedoch zugleich auch die Outputs v_j, und somit den Feedforward-Pfad der Capsules aus Layer $l+1$ berechnet, da offensichtlich die Capsules aus Layer $l+1$ diesen erst berechnen müssen. Daher ist es sinnvoll das Routing-by-Agreement zusätzlich aus Implementierungssicht zu betrachten und nicht wie bislang aus rein theoretischer Sicht nach Hinton et al. [SS17]:

Zahlreiche Implementierungen (u.a. die originale [Saba]) betrachten die Berechnung der Coupling-Koeffizienten c_{ij} für die unteren Capsules via Routing-by-Agreement aus der Sicht des höheren Layers, da dann ebenfalls die Aktivierung der höheren Capsule v_j dort ausgerechnet wird. Auch die Votes $\hat{\mathbf{u}}_{j|i}$ der unteren Capsules können im oberen Layer berechnet werden, was zur Folge hat, dass niedrigere Capsules nur ihre Aktivität u_i bzw. v_i zu den höheren Capsules senden müssen, welche diese dann „sammeln“ und mit einer größeren Gewichtsmatrix $\mathbf{W}_{ij}$ multiplizieren (anstelle einer kleinen pro Eltern-Capsule). Diese beiden Änderungen sorgen für eine kompaktere, übersichtlichere und vermutlich auch effizientere Implementierung. Hilfreich ist auch die Einführung eines „leaky Routing“, da vom finalen Capsules-Layer nicht zwangsläufig erwartet werden kann, dass dieser das gesamten Eingabebild erklärt (es wird oft nur auf einen Subset an Klassen aus dem Eingaberaum trainiert) [SS17, S. 8].

Algorithm 1 Routing algorithm with leaky_routing_softmax from capsule in layer above.

1: $\hat{\mathbf{u}}_{j|i} = \mathbf{u}_i * \mathbf{W}_{ij}$
2: **procedure** ROUTING($\hat{\mathbf{u}}_{j|i}, r, l$)
3: for all capsule i in layer $(l-1)$ und capsule j in layer l: $b_{ij} \leftarrow 0$
4: **for** r interations **do**
5: for all capsule i in layer $(l-1)$: $\mathbf{c}_i \leftarrow$ leaky_routing_softmax($\mathbf{b}_i$)
6: for all capsule j in layer l: $\mathbf{s}_j \leftarrow \sum_i c_{ij}\hat{\mathbf{u}}_{j|i}$
7: for all capsule j in layer l: $\mathbf{v}_j \leftarrow$ squash($\mathbf{s}_j$)
8: for all capsule i in layer $(l-1)$ and capsule j in layer l: $b_{ij} \leftarrow b_{ij} + \hat{\mathbf{u}}_{j|i} \cdot \mathbf{v}_j$
return $\mathbf{v_j}$

Der Pseudocode 1 zeigt minimale Änderungen am Algorithmus aus Abbildung 3.18 und betrachtet das Routing-by-Agreement, sowie die Vorhersagenberechnung aus Sicht des höheren Layers l, wenn dieser für die i Capsules aus Layer $l-1$ die Vorhersagen $\hat{u}_{j|i}$ und die Coupling-Coeffizienten c_{ij} berechnet, sowie die Aktivierungen der j Capsules des aktuellen Layers l. Die Funktion „leaky_routing_softmax($\mathbf{b}_i$)“ ist hierbei eine Funktion, die bei guter Übereinstimmung das oben beschriebene Routing ausführt und sonst den Coupling-Koeffizienten-Vektor $\mathbf{c}_i$ für die Capsule mit zu geringer Übereinstimmung zu allen oberen Capsules so umschreibt, dass die Vorhersage in eine „none-of-the-above“ Dimension umgeleitet wird. Eine „leaky_routing_softmax“ kann kurz beschrieben z.B. realisiert werden, indem layerweise für jede Capsule eine extra Dimension an die Routing-Logits gehängt wird, was erlaubt das nun zu dieser - nicht weiter verwendeten Dimension - über die extra Routing-Logits geroutet werden kann [Saba].
Eine weitere Möglichkeit ist die Einführung einer zusätzlichen Orphan (engl. für Waise) Kategorie im letzten Capsule-Layer, welche keiner Klasse angehört, irrelevante Hintergrundinformationen einfängt und es dem Netz erlaubt Teil-Ganzes-Beziehungen effizienter zu modellieren [SS17] [WZ18, S. 3].

3.3.3 Architektur eines Capsule Networks

Um verstehen zu können, wie mehrere Capsules aus Abschnitt 3.3.1 zu einer Architektur verknüpft werden, damit Aufgaben wie das Klassifizieren von einzelnen oder überlappten MNIST-Zahlen gelernt werden können, wird im Folgenden solch eine Netzwerkarchitektur vorgestellt. Abbildung 3.19 zeigt hierbei eine sehr flache Architektur, welche lediglich zwei Convolutional-Layer (*ReLu Conv 1* und *PrimaryCaps*) und einen fc Layer (*Digit Caps*) besitzt. Die Architektur wird folglich Layer für Layer von links nach rechts erläutert:

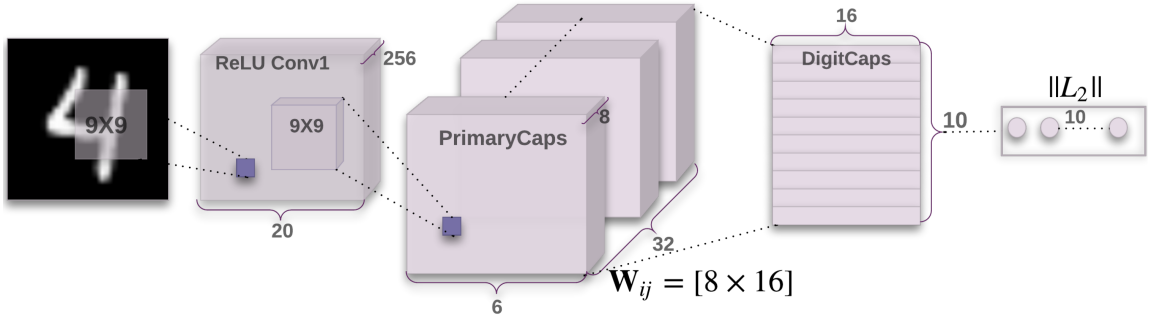


Abbildung 3.19: Einfache Capsule Network Architektur für MNIST [SS17, S. 4]

ReLu Conv 1: Die Aufgabe des conv. Layers ist das Erkennen grundlegender Features in der Eingabe [Pecc]. Als Input erhält dieser das MNIST-Eingabebild (28x28x1) und konvertiert darauf die Pixelinformationen in die Aktivität von lokalen Feature-Detektoren. Der Layer erzeugt 256 Feature-Maps über 9x9 Filter mit einem Stride von eins und besitzt daher 256 convolutional 9x9 Kernel. Der Output verzeichnet eine Dimension der Größe 20x20x256 [SS17, S. 3]. Um die Anzahl an Parametern in diesem Layer zu berechnen, muss in Betracht gezogen werden, dass jeder Kernel in einem conv. Layer einen Bias aufweist. Daher befinden sich in diesem Layer $(9 \times 9 + 1) \times 256 = 20\,992$ trainierbare Parameter [Pecc].

PrimaryCaps: Der zweite Layer ist ein Convolutional-Capsule-Layer, welcher 32 Channels von 6x6x8 Primary-Capsules besitzt. Dieser Layer kann als Convolutional-Layer mit der Squash-Funktion 3.3 als Nichtliniarität betrachtet werden, dessen Output in 32 Channels, die jeweils in einem 6x6 Grid angeordnet sind und pro Zelle im Grid einen 8D-Vektor ausgeben, eingeteilt wird [SS17, S. 3-4]. (Lose: Conv Layer mit 6x6 Ausgabegröße und $32 \times 8 = 256$ Feature-Maps).

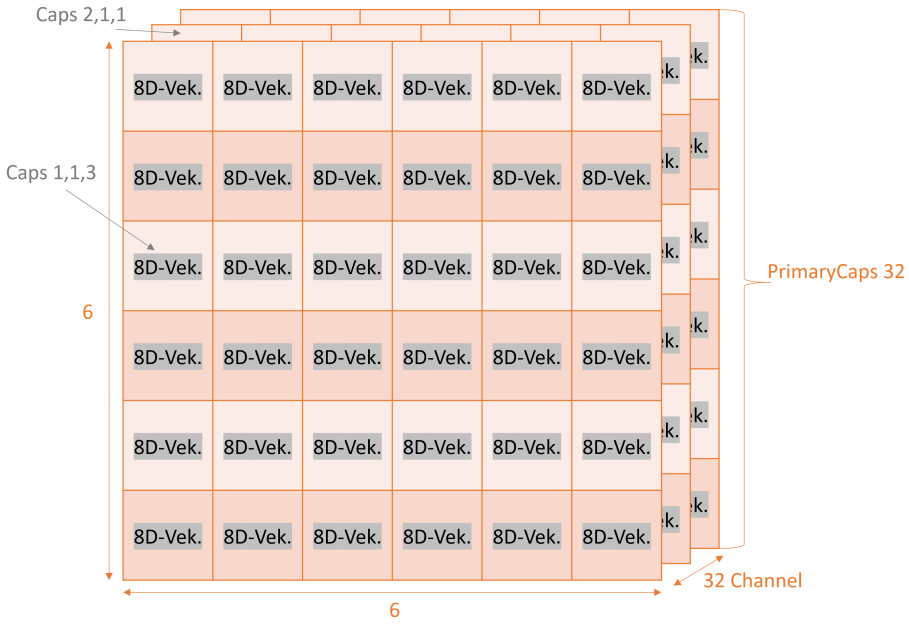


Abbildung 3.20: Die 32 6x6-Gitter der 8D-Capsules bzw. die 6x6x8 Primary-Capsules für jeden der 32 Channels

Abbildung 3.20 veranschaulicht die Struktur von Primary-Capsules und zeigt, dass eine *Primary-Capsule* eine convolutional 8D-Capsule darstellt: Jede der *32* Primary-Capsules besteht aus einem *6x6 Grid* welches pro Zelle einen *8D-Vektor* als Output berechnet (jeweils die Ausgabe einer einfachen Capsule) [SS17, S. 3-4]. Da hier die Kernel-Size auf 9x9 und der Stride auf zwei gesetzt wurde wendet jeder der *32 Primary-Capsule-Channel* 8 9x9 Filter auf die 20x20x256 Eingabe an und erzeugt somit eine 6x6x8 Ausgabe [Pecc]. Das *Grid* der Größe *6x6* entsteht durch die Input Dimension (20x20) und besitzt die Tiefe 8, da es sich um eine 8D-Capsule handelt. Folglich berechnet jeder Primary-Capsule-Channel 6x6x8 Outputs, was für alle *32 Primary-Capsules* einen Output der Größe $6 \times 6 \times 8 \times 32 = 9216$ für den Layer ergibt. Jede Capsule in einem *6x6 Grid* teilt sich dabei ihre Gewichte [SS17, S. 3-4] aufgrund der 8 Filter welche auf die Eingabe angewandt werden. Jede Primary-Capsule besitzt anders formuliert 8 Convolutional-Units, deren Kernel-Size hier auf 9x9 und deren Stride auf zwei gesetzt wurde und errechnet die Aktivitätsvektor mehrere Capsules. Pro Primary-Capsule werden je Dimension in der Ausgabe eine Convolitional-Unit benötigt, welche analog zum conv. Layer ein meist kleineres Grid aus der Eingabe erzeugen. Die 32x8 Convolitional-Units können mittels eines conv. Layers und durch Umformung des In- und Outputs wie in der Implementierung [Saba] berechnet werden. Anstelle der Multiplikation der Aktivitäten der unteren Capsules mit einfachen Gewichten für die Vorhersagen treten die Convolitional-Units, welche somit die Vorhersagen für das Routing berechnen. Ähnliche Berechnungen wie die des conv. Layer zeigen, dass der Layer $(9 \times 9 \times 256 + 1) \times 8 \times 32 = 5\,308\,672$ trainierbare Parameter besitzt [Pecc]. Die Formel für die Berechnung lautet demnach wie folgt:

$(FilterX \times FilterY \times BelowFeatureMaps + Bias) \times CapsuleAktivityVectorLenght \times Channels$

DigitCaps: Dieser Layer besteht, wie Abbildung 3.19 zeigt, aus *10 Capsules*, welche jeweils eine MNIST-Zahl in ihrem *16D-Ausgabevektor* beschreiben. Jede Capsule erhält den 8D-Input aller 6x6x32 Capsules aus dem PrimaryCaps-Layer [SS17, S. 4], dies ergibt insgesamt 1152 8D-Input-Vektoren. Gemäß der inneren Funktionsweise der Capsule erhält jeder dieser Eingabevektoren seine eigene 8x16-Gewichtsmatrix, die den 8D-Eingangsraum auf den 16D-Capsule-Ausgaberaum abbildet. Demnach existieren $6 \times 6 \times 32 = 1152$ Matrizen für jede Capsule sowie 1152 Coupling-Koeffizienten (c) und 1152 Logits (b), die in dem dynamischen Routing verwendet werden (das Agreement a muss nicht extra gespeichert oder trainiert werden).
Somit enthält der Layer $1152 \times 8 \times 16 + 1152 + 1152 = 149\,760$ trainierbare Parameter pro Capsule. Da der Layer *10 DigitCaps* (eine pro Klasse) besitzt, besteht er aus $149\,760 \times 10 = 1\,497\,600$ Parametern [Pecc]. Die Formel zur Berechnung lautet demnach wie folgt:
$BelowPrimaryCapsWidth \times BelowPrimaryCapsHeight$
$\times BelowPrimaryCapsChannels = NumMatrices = NumCouplingCoefficients$
$= NumLogits,$
$NumParameter = (NumMatrizen \times DigitCapsAktivityVectorLenght$
$\times BelowCapsuleAktivityVectorLenght + NumCouplingCoefficients + NumLogits)$
$\times NumDigitCaps.$
Die Länge des Ausgabevektors jeder *DigitCaps* indiziert die Präsenz der zugehörigen MNIST-Zahl und kann dafür benutzt werden den Loss $\|L_2\|$ der Klassifizierung zu berechnen. $\mathbf{W}_{ij}$ in Abbildung 3.19 ist die beschriebenen Gewichtsmatrix zwischen der Größe $\mathbf{u}_i, i \in (1, 32 \times 6 \times 6)$ der PrimaryCaps und der Größe $\mathbf{v}_j, j \in (1, 10)$ der *DigitCaps* für das Berechnen der Vorhersagen [SS17, S. 4].

Routing: Das Routing-by-Agreement findet nur zwischen den beiden aufeinanderfolgenden Capsule-Layer (*PrimaryCaps* und *DigitCaps*) statt. Zwischen *Conv1* und *PrimaryCaps* wird kein Routing verwendet [SS17, S. 4], da die Ausgabe (eine Feature-Map) von *Conv1* eindimensional ist und keine Orientierung in diesem Raum existiert, auf welcher Einigungen erzielt werden könnten.

3.3.4 Loss und Training

Da die Länge des Aktivitätsvektors einer Capsule die Existenzwahrscheinlichkeit repräsentieren soll, muss der Loss so definiert sein, dass die Capsule für die Klasse k nur einen langen Vektor aufweist, wenn die Zahl im Eingabebild präsent ist. Zusätzlich sollen mehrere unterschiedliche

Zahlen auf einmal erkannt und segmentiert werden, weshalb ein separater Margin-Loss L_k für jede Digit-Capsule k verwendet wird [SS17, S. 3], welcher wie folgt definiert ist:

$$L_k = T_k \; max(0, m^+ - \|\mathbf{v}_k\|)^2 + \lambda(1 - T_k) \; max(0, \|\mathbf{v}_k\| - m^-)^2 \tag{3.7}$$

Hierbei ist T_k gleich 1, wenn eine Zahl der Klasse k in der Eingabe existiert. Der Hyperparameter m^+ wird auf 0.9 gesetzt und m^- gleich 0.1. Die Herabsetzung des Loss für abwesende Ziffernklassen durch Multiplikation mit dem Gewichtungsfaktor $\lambda = 0.5$ hält das initiale Lernen davon ab, die Längen der Aktivitätsvektoren aller Capsules zu stark zu verkleinern. Der gesamte Klassifizierungs-Loss errechnet sich dann aus der Summe des Loss jeder DigitCapsule [SS17, S. 3] mit $L = \sum_k L_k$.

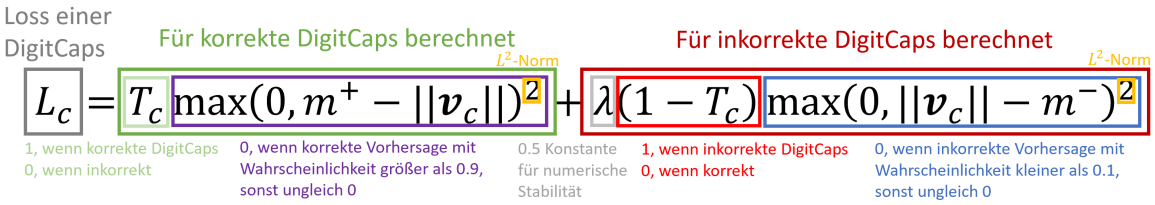


Abbildung 3.21: Der farbcodierte Margin-Loss nach [Pecc]

Die Loss-Funktion wirkt anfangs kompliziert, ist aber bei genauer Betrachtung verständlich und lässt eine Ähnlichkeit zum Loss einer SVM (Support Vector Machine) erkennen. Um die Kernidee hinter der Loss-Funktion zu verstehen muss beachtet werden, dass die Ausgabe des DigitCaps-Layer aus zehn 16D-Vektoren besteht. Abbildung 3.21 zerlegt für die weitere Erklärung die Loss-Funktion grafisch.

Während des Trainings wird für jedes Trainingsbeispiel ein Loss-Wert für jeden der zehn Vektoren mittels der obigen Formel 3.7 berechnet. Diese Loss-Werte werden für den finalen Klassifizierungs-Loss aufaddiert. Da das Lernen überwacht stattfindet, besitzt jedes Trainingsbeispiel ein korrektes Label, wobei das Label hier ein zehndimensionalen, one-hot-kodierten Vektor mit neun Nullen und einer Eins an der Stelle der korrekten Ziffer darstellt. In der Loss-Funktion bestimmt dann das korrekte Label den Wert von T_c (im *grünen* und *roten* Teil der Gleichung): Der Wert ist 1 wenn das korrekte Label der Ziffer dieser DigitCaps entspricht, sonst 0. Wäre beispielsweise die Ziffer Null das korrekte Label bedeutet dies, dass die erste DigitCaps verantwortlich dafür ist die Existenz der Ziffer Null zu codieren. Für diese DigitCaps wäre folglich die Loss-Funktion T_c gleich 1 und für die restlichen DigitCaps ergäbe die Funktion 0. Ist T_c gleich 1 wird der erste Term der Loss-Funktion (*grün*) berechnet und der zweite (*rot*) Term ergibt 0. Um in diesem Beispiel den Loss der ersten DigitCaps zu berrechnen wird der Ausgabevektor dieser DigitCaps von m^+ subtrahiert (in *lila*), welches fest auf 0.9 gesetzt ist. Folglich wird das Ergebnis nur behalten und quadriert, wenn dieses größer als 0 ist (in *lila*). Ansonsten wird 0

zurückgegeben. Anders ausgedrückt wird der Loss 0, wenn die korrekte DigitCaps das korrekte Label mit einer Wahrscheinlichkeit welche größer als 0.9 ist vorhersagt und der Loss wird ungleich 0, wenn die Wahrscheinlichkeit kleiner als 0.9 ist [Pecc].
Für DigitCaps welche nicht mit dem Label übereinstimmen ist in diesem Beispiel T_c gleich 0, weshalb der zweite Term (*rot*) evaluiert wird [Pecc], da dort $1 - T_c$ gleich 1 ergibt. In diesem Fall wird der Loss 0, wenn eine nicht passende DigitCaps ein inkorrektes Label mit einer Wahrscheinlichkeit die kleiner als 0.1 ist vorhersagt (wenn die Capsule korrekt schlussfolgert, dass ihre Ziffer nicht in der Eingabe existiert). Der Loss wird ungleich 0, wenn die DigitCaps ein inkorrektes Label mit einer Wahrscheinlichkeit größer als 0.1 vorhersagt (die Capsule ist noch zu unsicher).

Der Lambda-Koeffizient (mit einem festen Wert von 0.5) im linken Teil der Gleichung dient der numerischen Stabilität während des Trainings und hält, wie bereits erwähnt, das initiale Lernen davon ab die Längen der Aktivitätsvektoren aller Capsules zu sehr zu verkleinern. Die zwei Teile der Gleichung werden letztlich quadriert (*gelb*), damit die Loss-Funktion eine L^2-Norm besitzt, da diese Norm laut den Autoren bessere Ergebnisse liefert [Pecc].

Zusätzlich zum überwachten Klassifizierungs-Loss wird ein unüberwachter Rekonstruktion-Loss eingeführt, indem die Aktivität der DigitCaps, welche die korrekte Zahl repräsentiert in einen Decoder gegeben wird, mit welchem das Netzwerk versucht das gegebene 28x28 MNIST-Bild zu rekonstruieren:

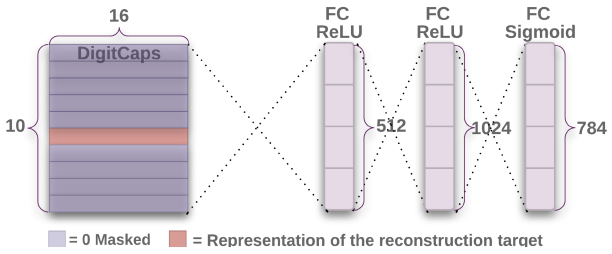


Abbildung 3.22: Decoder Struktur der Capsule-Network-Architektur für MNIST aus [SS17, S. 4]

Der Decoder besteht wie in Abbildung 3.22 dargestellt aus zwei fc ReLu-Layer mit 512 und 1024 Neuronen sowie aus einem fc Sigmoid-Layer mit 784 (28x28) Neuronen als Output-Layer. Auch wenn das Netz eine falsche Klassifizierungsentscheidung trifft, erhält der Decoder während des Trainings den Aktivitätsvektor der Capsule, welche die richtige Ziffer repräsentiert als Eingabe. Alle anderen Aktivitätsvektoren werden ignoriert (genullt). Das CapsNet soll dann im Training die Summe der quadrierten Differenzen zwischen dem Output des Sigmoid-Layers und den Pixeln Intensitäten des MNIST-Bildes minimieren. Damit dieser Loss nicht den

Klassifizierungs-Loss während des Trainings dominiert, findet eine Multiplikation mit einem Balance-Faktor von 0.0005 statt [SS17, S. 4].

Zu Beginn des Trainings werden alle Routing A-priories b_{ij} mit null initialisiert, damit der initiale Capsule-Output u_i zu allen Eltern-Capsules $v_0 ... v_9$ mit gleicher Wahrscheinlichkeit c_{ij} gesendet wird. Der Adam-Optimizer mit den Tensorflow Default-Parametern und mit exponentiell fallender Lernrate wird verwendet, um den Klassifizierungs-Loss aufaddiert mit dem skalierten Rekonstruktion-Loss zu minimieren [SS17, S. 3].
Um experimentell den Routing-Algorithmus verifizieren zu können, wurde die durchschnittliche Änderung der Routing-Logits nach jeder Routing-Iteration rechts in Abbildung 3.23 aufgezeigt [SS17, S. 3]:

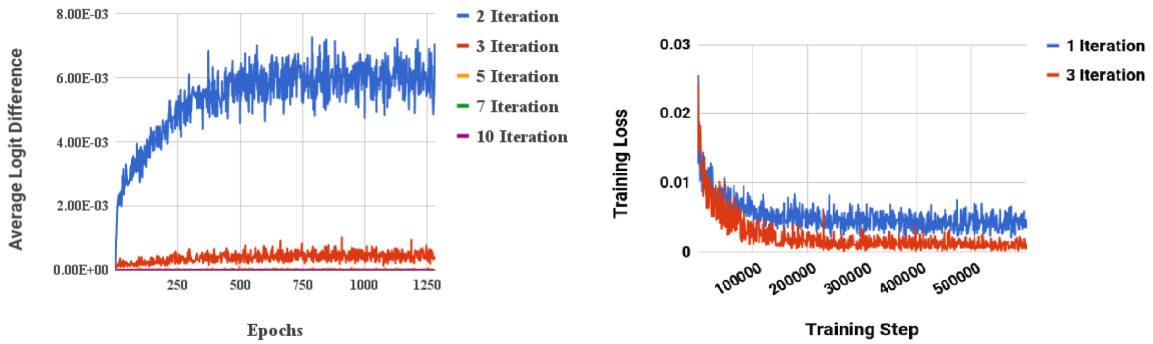


Abbildung 3.23: Links: Durchschnittliche Änderung der Routing-Logits für MNIST. Rechts: CapsNet-Loss in Abhängigkeit der Routing-Iterationen für Cifar10 aus [SS17, S. 11]

Links in Abbildung 3.23 ist zu erkennen, dass sich ab der fünften Routing-Iteration nur noch geringfügige Änderungen an den Routing-Logits ergeben, was bereits bei Trainingsbeginn beobachtet werden kann. Außerdem ist zu sehen, dass sich die durchschnittliche Änderung nach circa 500 Epochen stabilisiert [SS17, S. 11].
Rechts zeigt Abbildung 3.23 den Loss in Abhängigkeit der Routing-Iterationen anhand eines Trainings mit dem Datensatz Cifar10 und Batch-Size 128 und gibt zu erkennen, dass das Training mit drei Routing-Iterationen zu einem kleineren Loss führt und schneller konvergiert [SS17, S. 11].
Zusätzlich wurde in [SS17, S. 3] erwähnt, dass generell mehr Routing-Iterationen für erhöhte Netzwerkkapazität sorgen und daher das Netzwerk zu Overfitting tendiert.
Zusammenfassend kann aus Abbildung 3.23 geschlossen werden, dass drei Routing-Iterationen einen guten Ausgangswert darstellen [SS17, S. 11].

3.3.5 Anwendung des Capsule Networks

Ein, wie in Abschnitt 3.3.4 erläutert, auf MNIST trainiertes, flaches, 3-Layer CapsNet mit der Architektur aus Abschnitt 3.3.3, drei Routing-Iterationen und ohne Leaky-Routing erreicht bei der Klassifizierung von MNIST einen niedrigen Test-Error von 0.25%. Hierbei wurden vorweg die 28x28 MNIST-Trainingsdaten bis zu zwei Pixel in jede Richtung mit Zero-Padding verschoben; sonst wurden die Trainigsdaten nicht weiter modifiziert. Als Baseline wurde ein Standard CNN bestehend aus drei Convolutional Layer mit 256; 256; 128 Channels verwendet. Jeder Layer besitzt einen 5x5 Kernel und einen Stride von eins. Den drei Conv. Layer folgen zwei fc Layer der Größe 328 und 192. Der letzte fc Layer wurde mit Dropout zu einem 10-Klassen-Softmax-Layer mit Cross Entropy Loss verbunden. Die Baseline wurde so designed, dass sie die beste Performance hinsichtlich MNIST liefert, jedoch die Rechenkosten so nahe wie möglich am CapsNet liegen. Das CNN besitzt 35.4M Parameter, das CapsNet 8.2M und ohne das Rekonstruktions-Subnetz 6.8M. Das CNN wurde ebenfalls auf den um zwei Pixel verschobenen MNIST-Datensatz mittels des Adam-Optimizers trainiert und erreicht ein Test-Error von 0.39% [SS17, S. 5]. Ein weit komplexeres Modell von Wan et al. [Wan+13], mit Essembling, Rotation und Skalierung der Eingabe und weitere Tricks erreicht 0.21%. Das CapsNet einen etwas höheren mit, wie bereits erwähnt, 0.25%.

Nachdem kurz die Performance des beschriebenen CapsNets dargestellt wurde, findet eine leichte Veränderung einzelner Werte der Aktivitätsvektoren des DigitCaps-Layers statt (genauer um 0.05 im Intervall von -0.25 bis 0.25) bevor diese in den Decoder gegeben werden. Somit wird visuell dargestellt, welche Teile einer Entität die einzelnen Werte repräsentieren; dies zeigt Abbildung 3.24 [SS17, S. 5-7].

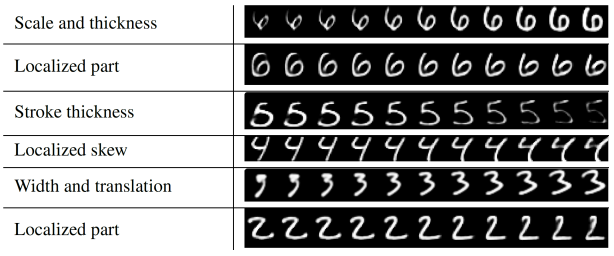


Abbildung 3.24: Beispiele decodierter Bilder nach steigenden leichten Veränderungen an den Werten der Aktivitätsvektoren aus [SS17, S. 6]

Abbildung 3.24 gibt zu erkennen, dass die Werte der Aktivitätsvektoren tatsächlich lernen einen Raum an Variationen von Instantiierungen einer Zahl aufzuspannen. Ebenfalls existieren zahlspezifische Werte, wie die Variationen der Länge des Schwanzes einer Zwei. Außerdem repräsentiert fast immer einer der 16-Werte einer DigitCaps die Breite der zugeordneten Ziffer;

einige Werte stellen Kombinationen von globalen Variationen dar, andere Werte, die Variation in einem lokalisierten Teil der Ziffer. Zum Beispiel werden unterschiedliche Werte für die Länge der Oberlänge einer Sechs und die Größe der Schleife verwendet [SS17, S. 6-7].

Wie in Abschnitt 3.1.3 erwähnt, kann das Routing als ein paralleler Attention-Mechanismus gesehen werden, welcher es jeder Capsule auf einem Level (durch die Routing-Logits) erlaubt, sich einer aktiven Capsule im Level darunter zu widmen und alle anderen zu ignorieren [SS17, S. 6].
Daher sollte ein CapsNet dazu fähig sein mehrere verschiedenen MNIST-Zahlen in einem Bild zu erkennen, auch wenn diese überlappen. Das Routing-by-Agreement sollte hierbei ermöglichen Vorkenntnisse über die Form von Objekten zu verwenden, um die Segmentierung zu unterstützen, und die Notwendigkeit beseitigen, Segmentierungsentscheidungen höherer Ebenen in der Pixeldomäne treffen zu müssen [SS17, S. 6-7]. Wird nun das CapsNet aus Abschnitt 3.3.3 mit MultiMNIST (MNIST bei dem zwei Zahlen überlappen) trainiert, wird eine höhere Genauigkeit als die des Baseline-CNN erreicht und dieselbe wie bei dem sequenziellen Attention-Modell aus Ba et al. [JB14], welches auf einer leichteren Aufgabe mit weniger Überdeckung angewandt wurde. Für die Rekonstruktion werden die zwei aktivsten Capsules als Klassifizierung ausgewählt und nacheinander in den Decoder gegeben, um eine einzelne Zahlen aus den überlappten zu generieren. Die Label-Bilder sind hierbei bekannt, da diese dazu verwendet wurden das überlappte Bild zu erstellen [SS17, S. 6-7].

Die in Abbildung 3.25 dargestellten Rekonstruktionen zeigen, dass das CapsNet dazu fähig ist, das überlappende Bild in die zwei originalen Zahlen zu segmentieren. Da diese Segmentierung nicht auf Pixelebene erfolgt, ist zu erkennen, dass das Modell in der Lage ist mit den Überlappungen (ein Pixel wird für beide orginalen Bilder benötigt) korrekt und unter Berücksichtigung aller Pixel umzugehen. Die Position und der Style eine Zahl ist wie oben gezeigt in einer DigitCaps codiert und der Decoder lernte eine Zahl aus dieser Codierung zu rekonstruieren. Die erfolgreiche Rekonstruktion mit Überlappungen zeigt, dass jede DigitCaps den Style und die Position aus den Vorhersagen, welche sie vom PrimaryCaps-Layer enthält, extrahieren kann [SS17, S. 6-7].

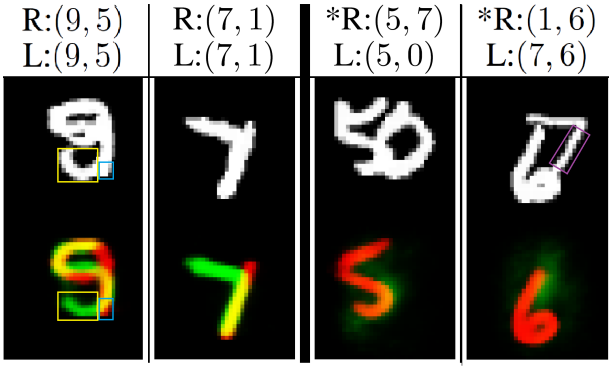


Abbildung 3.25: Beispiele von Rekonstruktionen eines CapsNet mit drei Routing-Iterationen auf MultiMNIST [SS17, S. 4]

Abbildung 3.25 stellt oben die *Eingabebilder* des Netzes und unten in grün und rot die *übereinandergelegten Rekonstruktionen* des Netzes dar. *L:*(l_1, l_2) sind die Label der zwei Zahlen und *R:*$(r_1; r_2)$ repräsentieren die zwei Zahlen, welche für die Rekonstruktion genutzt wurden. Es ist zu erkennen, dass bei korrekter Klassifizierung das Modell alle Pixel in Betracht zieht und dazu fähig ist, ein Pixel zu zwei Zahlen in sehr schwierigen Szenarios (*zweites Bild von links*) zuzuordnen [SS17, S. 8]. Wird das korrekt klassifizierte Paar *(5, 9)* links im Bild genauer betrachtet, wird festgestellt, dass das Netz den *unteren Bogen der 5 (gelbes Rechteck)* nicht zu der *9* zuordnet, da sonst keine Erklärung für den *kleinen vertikalen Strich (blaues Rechteck)* rechts neben dem unteren Bogen der 5 existiert, weshalb die Vorhersagen der Capsules nicht mit der „Pose“ einer 9 mit Schleife übereinstimmen und die „Pose“ einer 9 ohne unteren Bogen mit vertikalen Strich erzeugt, welche mehr Pixel aus der Eingabe erklärt [Sabb].
Die zwei mit *(*)* markierten Spalten zeigen absichtliche Rekonstruktionen von einer Zahl die weder das Label noch die Klassifizierung des Modells ist. Diese Spalten legen nahe, dass das Modell nicht einfach die beste Zuordnung für alle Zahlen im Bild findet, einschließlich der nicht in der Eingabe vorhandenen. So kann beispielsweise im Fall *(5, 0)* keine *7* rekonstruiert werden, das das Netz weiß, dass eine *5* und eine *0* sich am besten integrieren und alle Pixel berücksichtigen, weshalb die entsprechenden Capsules aktiviert werden [SS17, S. 8]. Auch bei *(6, 7)* könnte eine *1 (lila Rechteck)* rekonstruiert werden, da das Netz jedoch eine *7* klassifiziert hat und eine starke Erklärung für diese besitzt ist eine *1* nicht rekonstruierbar [Sabb], weshalb die Aktivität der Capsule, welche die *1* repräsentiert null oder sehr gering sein muss. Daher wird kein Pixel zu zwei Ziffern zugewiesen, wenn eine der Ziffern keine weitere Unterstützung besitzt [SS17, S. 8] [Sabb] (wenn die Vorhersagen für diese Ziffer nicht übereinstimmen).

Auch bei den Datensätzen smallNORB und einem kleinen Set von SVHN (Street View House Numbers) erreicht die exakt selbe Architektur, welche nur in der Anzahl der DigitCaps (oder hier ClassCaps) variiert wird, State-of-the-Art-Performance (2.7% und 4.3% Test Error Rate

jeweils). Ein Ensemble aus sieben dieser Architekturen erreicht ein Standard Test-Error von 10.6% auf Cifar10. Dabei wurde Leaky-Routing verwendet, da, wie bereits erwähnt, ein Nachteil von Capsules ist, dass sie, wie andere generative Modelle, dazu tendieren die vollständige Eingabe zu berücksichtigt. Dies ist bei Bildern mit variantenreichen Hintergrund wie in Cifar10 unvorteilhaft [SS17, S. 8].

3.3.6 Erkenntnisse durch Capsule Networks

Capsules unterliegen einer sehr starken Repräsentationsannahme: An jedem Ort im Bild, existiert höchstens eine Instanz des Entitätstyps, den eine Capsule repräsentiert. Diese vom Crowding (siehe Abschnitt 3.2.4) motivierte Repräsentationsannahme eliminiert das Bindungsproblem (Wie werden visuelle Teilinformationen zu einem Gesamteindruck kombiniert? [Hin81a]) und erlaubt es der Capsule verteilte Repräsentationen (den Aktivitätsvektor) zu nutzen, um Instanziierungsparameter eines Entitätstypen an einem gegebenen Ort zu codieren, solange die Einschränkung auf höchstens eine Instanz eines Entitätstyps für jedem Ort im Bild gilt. Diese verteilte Darstellung ist exponentiell effizienter als die Codierung der Instanziierungsparameter durch Aktivierung eines Punktes auf einem hochdimensionalen Gitter und mit der richtigen verteilten Darstellung können Capsules die Tatsache voll ausnutzen, dass räumliche Beziehungen durch Matrixmultiplikationen modelliert werden können - sie verwenden, wie bereits erwähnt, neuronale Aktivitäten, die variieren wenn der Blickwinkel variiert, anstelle diese Variation aus dem Aktivitäten zu eliminieren. Dies verschreibt Capsules einen Vorteil gegenüber „Normalierungs“-Methoden (z.B. Spatial-Transformer-Networks [MJ15]), da sie mit mehreren unterschiedlichen affinen Transformationen von verschiedenen Objekten oder Objektteilen zur selben Zeit umgehen können; dies wird in Abschnitt 3.12 nochmal aufgegriffen und genauer erläutert [SS17, S. 9].

Capsules sind, wie in Abschnitt 3.3.5 dargestellt, sehr gut in der Segmentierung, da ihr Aktivitätsvektor ihnen erlaubt Routing-by-Agreement zu nutzen, wodurch höher liegenden Capsules das „Wegerklären“ von irrelevanten Informationen ermöglicht wird. Die Wichtigkeit des Routings wird ebenfalls von biologisch plausiblen Modellen der invarianten Mustererkennung im visuellen Cortex unterstützt. Hinton stellte beispielsweise ein Modell in [Hin81b] vor, welches, auf kanonischen Objekten basierende, Referenzrahmen und dynamische Verbindungen nutzt, um Formbeschreibungen zu generieren, welche für die Objekterkennung verwendet werden können. Diese Verbindungen verbesserte sich durch Olshausen et al. [AO93] weiter, indem ein biologisch plausibles, position- und größen-invariantes Modell für Objektrepräsentationen entwickelt wurde [SS17, S. 9].

Hinton et al. [Hin11] stellten zuvor Transforming-Autoencoder aus Abschnitt 3.2 vor, die Instanziierungsparameter eines PrimaryCapsule-Layers erzeugen, jedoch voraussetzen, dass die Transformationsmatrizen extern berechnet und als zusätzliche Eingabe den Capsules überreicht werden. Capsule Networks hingegen formen komplette Systeme und beantworten die in Abschnitt 3.2 aufgekommene Frage, wie größere und komplexere Entitäten durch das Nutzen von Übereinstimmungen der Posen von aktiven, niedrigleveligen Capsules erkannt werden können [SS17, S. 8-9].

Da in diesem Abschnitt der Grundaufbau und die Ideen hinter dem originalen Capsule Network erörtert wurde, können in den folgenden Abschnitten dieses Teils der Arbeit, Erweiterungen, Verbesserungen und komplexere Modelle, wie Matrix Capsules, beschrieben werden.

3.4 Matrix Capsules mit Expectation-Maximization-Routing

Neuronale Netze benutzten typischerweise einfache Nichtlinearitäten indem eine nichtlineare Funktion (z.B. ReLu) auf die Ausgabe eines linearen Filters angewandt wird. Weiter können sie eine Softmax-Nichtlinearität verwenden, um einen ganzen Vektor von Logits in einen Vektor von Wahrscheinlichkeiten zu konvertieren. Capsules hingegen nutzen weitaus kompliziertere Nichtlinearitäten, um die Aktivierungswahrscheinlichkeiten und Instanziierungsparameter in einem Layer zu den Aktivierungswahrscheinlichkeiten und Instanziierungsparametern des nächsten Layers zu konvertieren [HSF18a, S. 2]. Matrix Capsules trennen hierbei (wie Transforming-Autoencoder) die Aktivierungswahrscheinlichkeit von den Instanziierungsparametern, welche in einer Pose-Matrix dargestellt werden. Dabei wird eine weit komplexere Nichtlinearität, als die der Capsules in Abschnitt 3.3 verwendet, welche einige, im folgenden Abschnitt beschriebene, Schwächen des CapsNet aus Abschnitt 3.3 ausmerzt.
Auch bei Matrix Capsules besteht jeder Layer im Netzwerk aus mehren Capsules. Diese Capsules bestehen zu einem aus einer Logistic Unit um die Präsenz einer Entität zu beschreiben und zum Anderen aus einer 4x4 Matrix welche die Beziehung (bzw. die Pose) zwischen Entität und dem Betrachter beschreiben könnte. Eine Capsule in einem Layer votet (stimmt) für die Pose-Matrix vieler verschiedener Capsules im Layer darüber, indem sie ihre eigene Pose-Matrix mit einer trainierbaren, blickwinkel-invarianten Transformationsmatrix multipliziert, welche lernen könnte die Teil-Ganzes-Beziehung zu repräsentieren. Jeder dieser Votes wird durch einen Zuweisungskoeffizient gewichtet. Diese Koeffizienten werden iterativ für jede Eingabe aktualisiert, indem ein Expectation-Maximization (engl. für Erwartung-Maximierung, kurz EM)-Routing-Algorithmus genutzt wird, sodass die Ausgabe jeder Capsule zu einer Capsule in Layer darüber geroutet wird, die ein Cluster von ähnlichen Stimmen erhält. Auch hier werden die Transformationsmatrizen durch Backpropagation über die entrollten Iterationen des Routing-

Algorithmus zwischen jedem Paar von „benachbarten“ Capsule-Layern gelernt [HSF18a, S. 1].

3.4.1 Wieso Matrix Capsules?

In Abschnitt 3.2 wurden Transforming-Autoencoder vorgestellt, welche lernen ein Stereopaar von Bildern zu einem anderen Paar in einem leicht veränderten Blickwinkel zu transformieren; die Transformationsmatrix musste allerdings extern angewandt werden. Das CapsNet mit Routing-By-Agreement in Abschnitt 3.3.2 lernt solche Matrizen und zeigte sich besonderes effizient bei der Segmentierung von stark überlappenden Ziffern; doch auch dieses System weist einige „Mängel“ auf, welche Matrix Capsules beseitigen sollen [HSF18a, S. 9]:

1. Die Länge des Aktivitätsvektors (der Pose) repräsentiert die Wahrscheinlichkeit, dass die Entität, welche durch die Capsule repräsentiert wird, präsent ist. Damit die Länge kleiner eins bleibt, wird eine „prinzipienlose“ Nicht-Liniarität (Squash) verwendet. Diese verhindert die Existenz von einer sinnvollen Zielfunktion, welche vom iterativen Routing minimiert werden kann. Hinton et al. [HSF18a] führen dies nicht weiter aus, wobei hier vermutet wird, dass unter einer sinnvollen Zielfunktion eine Funktion wie die, im folgenden Abschnitt beschriebene, „freie Energie“ verstanden wird, die auf natürlicher, physikalischer Grundlage beruht und nicht prinzipienlos erdacht wurde.

2. Es wird der Kosinus des Winkels zwischen zwei Aktivitätsvektoren genutzt, um das Agreement der Votes zu messen. Der Kosinus sättigt sich bei eins, weshalb er nicht sensitiv hinsichtlich der Unterscheidung zwischen einem guten und einem sehr guten Agreement ist. Um dies zu verhindern, nutzen Matrix Capsules eine negative, logarithmierte Varianz eines Gauß-Clusters (siehe Abschnitt 3.4.2).

3. In original CapsNets wird ein Vektor der Länge n genutzt anstelle einer Matrix mit n Elementen, um einen Pose/Aktivitätsvektor zu repräsentieren, daher besitzen die Transformationsmatrizen n^2 anstelle von lediglich n Parametern.

Die folgenden Abschnitte durchleuchten Matrix Capsules im Detail und zeigen, wie diese Verbesserungen realisiert wurden.

3.4.2 Schematischer Aufbau einer Matrix Capsule mit EM-Routing

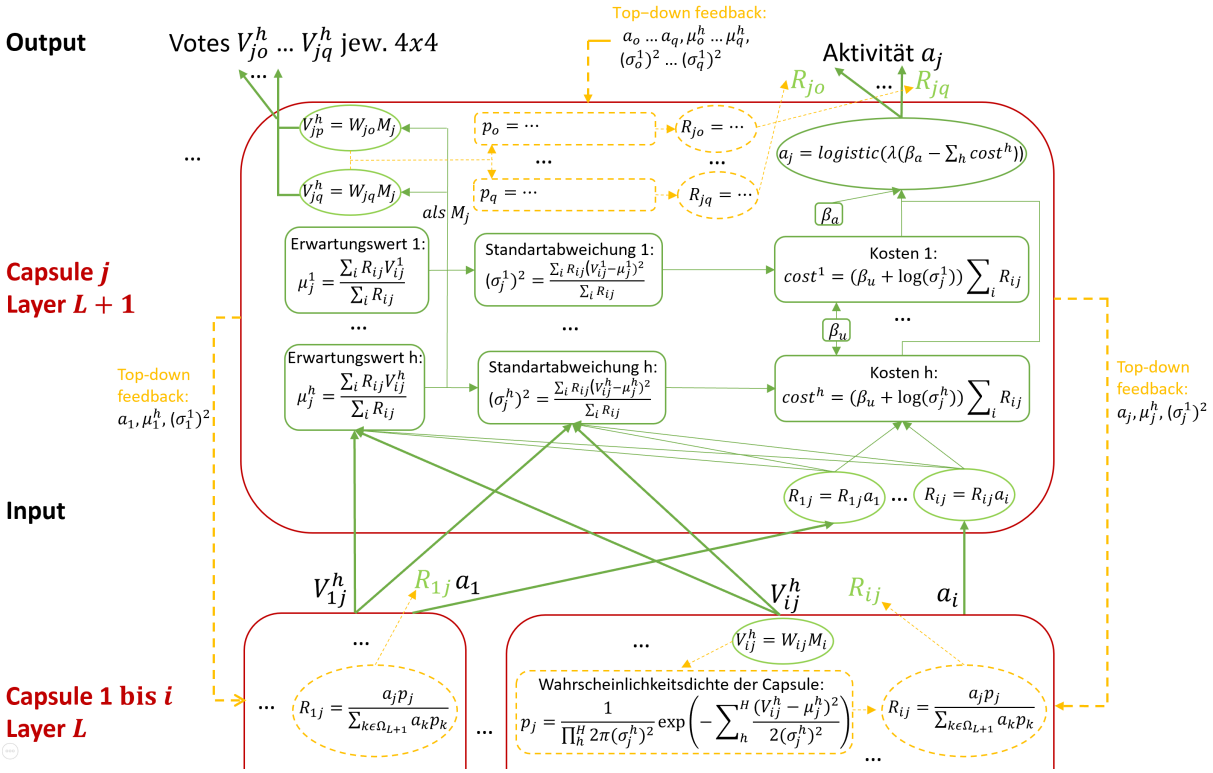


Abbildung 3.26: Schematischer Aufbau einer Matrix Capsule mit EM-Routing

Analog zum vorherigen Vorgehen, wird auch hier anhand Abbildung 3.26 ein Überblick über den Aufbau der Capsules gegeben und wie in Abbildung 3.15 die Capsule j in Layer $L+1$ in Abhängigkeit ihrer „Kind-Capsules“ betrachtet. Der *grüne* Pfad kennzeichnet hierbei alle für den Feedforward-Pfad nötigen Berechnungen und der *gelbe* Pfad die zusätzlich Schritte für die Anpassung der Coupling-Koeffizienten/Zuordnungswahrscheinlichkeiten R_{ij}. Weiter wird das in Abschnitt 3.4.3 beschriebene EM-Routing in einen E-Step und einen M-Step eingeteilt, wobei der E-Step (der Expectation-Step) durch die *durchgezogenen umrandeten Kästchen* dargestellt wird und der M-Step (der Maximisation-Step) durch die *gestrichelten*. Bevor die einzelnen Schritte für die Berechnung der Outputs in Abschnitt 3.4.3 ausführlich beschrieben werden, wird zunächst ein Überblick gegeben:

Die Menge der Capsules in Layer L werden als Ω_L bezeichnet. Jede Capsule besitzt eine 4x4 Pose-Matrix M, und eine Aktivierungswahrscheinlichkeit a. Diese können mit den Aktivitäten eines Neurons in einem herkömmlichen neuronalen Netz verglichen werden, da diese auf der aktuellen Eingabe basieren und nicht gespeichert werden. Zwischen jeder Capsule i in Layer

L und jeder Capsule j in Layer $L+1$ befindet sich eine 4x4, trainierbare Transformationsmatrix W_{ij}. Die W_{ij}s und zwei Biases pro Capsule (β_a und β_u) sind die einzigen gespeicherten Parameter und werden durch Backpropagation gelernt. Die Pose-Matrix von Capsule i wird mit W_{ij} transformiert, um eine Stimme/Vorhersage $V_{ij} = M_i W_{ij}$ für Capsule j zu erzeugen. Die Pose und Aktivierungen aller Capsules in Layer $L+1$ werden berechnet indem die nichtlineare Routing Prozedur EM-Routing genutzt wird, welche als Input V_{ij} und a_i für alle $i \epsilon \Omega_L$ und $j \epsilon \Omega_{L+1}$ erhält. Das Routing passt iterativ durch das Fitting einer Mischung von Gaußverteilungen, die Zuordnungswahrscheinlichkeiten zwischen allen Capsules $i \epsilon \Omega_L$ und $j \epsilon \Omega_{L+1}$, sowie die Mittelwerte, Varianzen und Aktivierungswahrscheinlichkeiten der Capsules in Layer $L+1$ an [HSF18a, S. 2].

3.4.3 Expectation-Maximization für das Routing zwischen Capsules

Das Routing findet zwischen den Capsules im höheren Layer und den Capsules im Layer darunter statt. Auch hier wird zuerst das Routing zwischen einem Paar von Layern vollendet, bevor das Routing für das nächsthöhere Layerpaar gestartet wird. Die Routing-Prozedur besitzt eine starke Ähnlichkeit mit dem Fitting einer Mischung von Gaußverteilungen durch das Nutzen eines EM-Algorithmus, wobei die höher liegende Capsule die Rolle der Gaußverteilungen spielt und die Mittelwerte der aktivierten niedrigeren Capsules für einen einzelnen Input die Rolle der Datenpunkte [HSF18a, S. 12]. Im Folgenden wird daher zuerst das Fitting einer Mischung von Gaußverteilungen anhand des EM-Algorithmus beschrieben, bevor das EM-Routing durch zwei Modifikationen aus diesem abgeleitet wird.

Der EM-Algorithmus und die Zielfunktion für das Fitting einer Mischung von Gaußverteilungen

Es werden nun die sinngemäßen Aussagen aus dem original Paper der Matrix Capsules ([HSF18a]) mit Präzisierungen dieser Aussagen alterniert, um ein tieferes Verständnis über die Vorgänge im EM-Routing zu erlangen.

Aussage: Das Routing hat eine starke Ähnlichkeit mit dem Fitting einer Mischung von Gaußverteilungen durch das Nutzen von EM [HSF18a, S. 12].

Präzisierungen: Der EM-Algorithmus ist ein Verfahren für unüberwachtes Clustering, welches zum Erkennen mehrerer Kategorien in einer Kollektion von Objekten dient. Das Problem ist dabei unüberwacht, da die Kategorie-Label nicht gegeben sind [Rus+10, S. 817]. Beispielsweise könnte das Spektrum von hunderttausenden Sternen aufgezeichnet werden, doch dabei tragen die Sterne noch kein bekanntes Label wie „Rote Riesen“ oder „Weiße Zwerge“. Daher wird,

um bestimmen zu können, ob das aufgezeichnete Spektrum verschiedene Typen von Sternen offenbart und um die Anzahl und Charakteristiken dieser Sternentypen zu definieren, von den Astronomen unüberwachtes Clustering angewandt [Rus+10, S. 817].

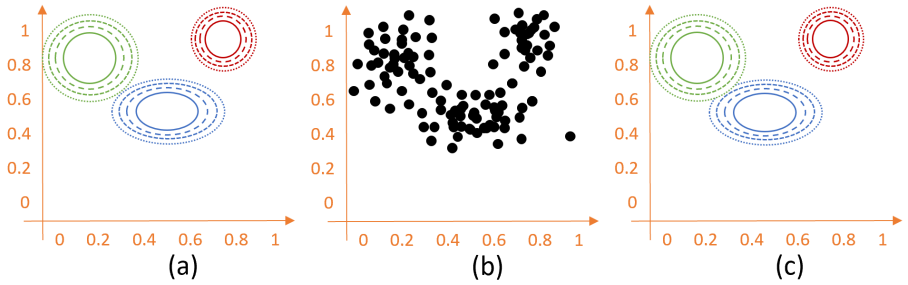


Abbildung 3.27: (a) Ein Modell einer gaußschen Mischverteilung mit drei Komponenten, (b) Sample Datenpunkte des Modells aus (a) und (c), das durch EM rekonstruierte Modell aus den Daten von (b). Nach [Rus+10, S. 818]

Abbildung 3.27 *(b)* zeigt einige Datenpunkte, wobei jeder Datenpunkt zwei kontinuierliche Attribute spezifiziert. Diese Datenpunkte könnten im Beispiel mit Sternen korrespondieren und die Attribute mit der spektralen Intensität zweier Frequenzen (x- und y-Achse). Im Clustering muss nun erschlossen werden, welche Verteilung diese Daten generiert haben könnte. Dabei wird vom Clustering angenommen, dass die Daten von einer Mischverteilung P erzeugt worden sind. Solch eine Verteilung besitzt k Komponenten, wobei jede Komponente eine eigene Verteilung darstellt. In Abbildung 3.27 *(a)*, handelt es sich beispielsweise um drei Komponenten. Sei die Zufallsvariable C die Komponente mit den Werten $1, ..., k$; dann ist die Mischverteilung gegeben durch:

$$P(\mathbf{x}) = \sum_{i=1}^{k} P(C = i)P(\mathbf{x}|C = i) \tag{3.8}$$

wobei $\mathbf{x}$ sich auf die Werte der Attribute des Datenpunktes bezieht. Für kontinuierliche Daten ist die natürliche Wahl der Verteilung die mehrdimensionale Normal- bzw. Gaußverteilung. Die Parameter eine Mischung von Gaußverteilungen sind dann $w_i = P(C = i)$ (die Gewichte der Komponente), $\boldsymbol{\mu_i}$ (der Mittelwert jeder Komponente) und Σ_i (die Kovarianz jeder Komponente). Zur Veranschaulichung zeigt Abbildung 3.27 *(a)* die Mischung der drei Gaußverteilungen, als Quelle der Daten in *(b)*. Demnach ist das Problem des unüberwachten Clusterns das Wiederherstellen des Mischmodell *(a)* aus den Rohdaten *(b)*.

Wie *(b)* zeigt, ist jedoch beim Zuordnen von Datenpunkten zu Komponenten weder bekannt, welche Komponente welchen Datenpunkt generiert hat, noch welche Parameter die einzelnen Komponenten besitzen [Rus+10, S. 817-818].

Die generelle Idee vom EM ist es so zu tun, als wären die Parameter des Modells bekannt und aus diesen die Wahrscheinlichkeit, mit der jeder Punkt zu jeder Komponente gehört, zu

schlussfolgern. Danach werden die Komponenten hinsichtlich der Daten neu angeglichen, wobei jede Komponente auf den gesamten Datensatz angepasst und jeder Datenpunkt mit der Wahrscheinlichkeit, dass er zu dieser Komponente gehört, gewichtet wird. Dieser Prozess iteriert bis zur Konvergenz. Für eine Mischung aus Gaußverteilungen können dabei die Parameter des Mischmodells willkürlich initialisiert werden; danach wird der E-Step und M-Step alternierend und iterativ ausgeführt [Rus+10, S. 817-818]. In Abbildung 3.27 sind die Ergebnis-Komponenten *(c)*, des auf den Datenpunkten *(b)* angewandten EM-Algorithmus, veranschaulicht.

Aussage: Der EM-Algorithmus für das Fitting einer Mischung von Gaußverteilungen alterniert zwischen einem E-Step und einem M-Step. Der E-Step wird dazu verwendet jedem Datenpunkt, der unter einer oder mehreren Gaußverteilungen stehen kann, die Wahrscheinlichkeit zuzuteilen, mit welcher dieser den Gaußverteilungen zugeordnet wird [HSF18a, S. 12].

Präzisierungen: E-Step: 1. Berechne die Zuordnungswahrscheinlichkeiten $p_{ij} = P(C = i|\mathbf{x}_j)$ - die Wahrscheinlichkeiten ob das Datum x_j von Komponente i generiert wurde. Durch den Satz von Bayes ergibt sich dafür $p_{ij} = \alpha P(\mathbf{x}_j|C = i)P(C = i)$. Der Term $P(\mathbf{x}_j|C = i)$ ist die Wahrscheinlichkeit von $\mathbf{x}_j$ der $i$ten Gaußverteilung, und der Term $P(C = i)$ ist der Gewichtsparameter w_i für die $i$ten Gaußverteilung [Rus+10, S. 818]. Der Parameter α dient zur Normalisierung und sorgt dafür, dass das Integral der Verteilung 1 ist [Rus+10, S. 811]. 2. Definiere $n_i = \sum_j p_{ij}$ die tatsächliche Anzahl an Datenpunkten die momentan zur Komponente i zugeordnet sind [Rus+10, S. 818].

Aussage: Diese Zuordnungswahrscheinlichkeiten agieren als ein Gewicht und der M-Step besteht für jede Gaußverteilungen darin, den Mittelwert dieser gewichteten Datenpunkte und die Varianz um diesen Mittelwert zu finden [HSF18a, S. 12].

Präzisierungen: M-Step: Berechne den Mittelwert, die (Ko)varianz und das Gewicht der Komponente mit folgenden Schritten in Sequenz [Rus+10, S. 818]:

$$\boldsymbol{\mu}_i \leftarrow \sum_j p_{ij}\mathbf{x}_j/n_i \tag{3.9}$$

$$\boldsymbol{\Sigma}_i \leftarrow \sum_j p_{ij}(\mathbf{x}_j - \boldsymbol{\mu}_i)(\mathbf{x}_j - \boldsymbol{\mu}_i)^T/n_i \tag{3.10}$$

$$w_i \leftarrow n_i/N \tag{3.11}$$

wobei N die totale Anzahl an Datenpunkten ist und sich das Gewicht der Komponente w_i direkt aus dem Anteil der zugeordneter Datenpunkte ergibt. Der M-Step findet die Werte der Parame-

ter, welche den Log-Likelihood des Modells jede Iteration maximiert [Rus+10, S. 818], also Parameter des Modells unter welchen das Vorkommen der gegebenen Datenpunkte am wahrscheinlichsten ist. Dies kann bewiesen werden (siehe z.B. [Rus+10, S. 818]), allerdings wird auf den Beweis nicht weiter eingegangen. Die Konvergenz zu einem globalem Optimum ist ebenfalls nicht garantiert [Rus+10, S. 818].
Abbildung 3.27 *(c)* zeigt ein Modell, welches der EM-Algorithmus lernen könnte [Rus+10, S. 818] und gibt zu erkennen, dass sich die Cluster überlappen können, was der Zuordnungswahrscheinlichkeit p_{ij} zu verdanken ist. Falls wie in k-Means ein Datenpunkt gänzlich zu einem Cluster zugeordnet werden soll, kann für diesen einfach die wahrscheinlichste Komponente/-Verteilung gewählt werden. Ein Wechsel von softer Zuordnung zu harter Zuordnung findet dann statt; allerdings ist im EM-Routing die Zuordnung eines Bruchteils erwünscht, was aus folgender Aussage hervorgeht:

Aussage: Findet ebenfalls das Fitting von Mischverhältnissen für jede Gaußverteilung statt, wird ein Bruchteil der Daten eines Punktes zu der Gaußverteilungen zugeordnet [HSF18a, S. 12].

Aussage: Der M-Step hält die Zuordnungswahrscheinlichkeiten konstant und passt jede Gaußverteilung so an, dass er die Summe der gewichteten, logarithmierten Wahrscheinlichkeiten maximiert, sodass die Gaußverteilung ihre zugeordneten Datenpunkte generiert.

Präzisierungen: Wie oben erläutert werden im M-Step die Werte der Parameter einer Mischung von Gaußverteilungen berechnet, dabei werden die Zuordnungswahrscheinlichkeiten zwar genutzt, aber nicht abgeändert. Der E-Step passt diese beruhend auf den Ergebnisse des M-Steps an. Der M-Step findet, wie oben erwähnt, die Werte der Parameter, welche den Log-Likelihood des Modells maximieren. Die Gaußverteilungen werden demnach so angepasst, dass diese ihren Anteil am Log-Likelihood maximieren. Da jede Komponente bezüglich des gesamten Modells mit w_i gewichtet wird, maximieren alle Gaußverteilungen die Summe der gewichteten, logarithmierten Wahrscheinlichkeiten, sodass eine Gaußverteilung die ihr zugeordneten Datenpunkte generiert.

Der EM-Algorithmus bezüglich Capsules und physikalischen Energien
Eine höher liegende Capsule wird hier als diese Gaußverteilungen betrachtet und die Mittelwerte der aktiven Capsules in niedrigen Layer als Datensatz. Für die Votes werden die Mittelwerte mit einer Transformationsmatrix multipliziert.

Aussage Die negative, logarithmierte Wahrscheinlichkeitsdichte eines Datenpunkts unter einer

Gaußverteilung kann wie die Energie eines physikalischen Systems behandelt werden, und der M-Step minimiert die erwartete Energie, indem die Erwartungen unter Verwendung der Zuweisungswahrscheinlichkeiten skaliert wird [HSF18a, S. 12].

Präzisierungen: Die Neuberechnung der Mittelwerte und Varianzen reduziert eine Energie, die den quadrierten Distanzen der Votes von den gewollten Mittelwerten, gewichtet mit den Zuordnungswahrscheinlichkeiten entspricht [HSF18b][4]. Hinton et al. [HSF18b] führt dies nicht weiter aus, jedoch kann physikalische Energie als Arbeit betrachtet werden, die mit einer gewissen Kraft über eine gewisse Distanz verrichtet wurde. Da es sich hier um eine Distanz zwischen Mittelwerten und Varianzen handelt, die mit Zuweisungswahrscheinlichkeiten (Kraft) gewichtet wird, kann dies als physikalisches System betrachtet werden. Weiter werden die quadrierten Distanzen der Votes von den gewollten Mittelwerten minimiert, weshalb folglich die Energie durch den M-Step reduziert wird.

Aussage: Der E-Step passt die Zuordnungswahrscheinlichkeiten für jeden Datenpunkt an, um eine als freie Energie bezeichnete Menge zu minimieren, die der erwarteten Energie minus der Entropie entspricht. Die erwartete Energie kann minimiert werden, indem jeder Datenpunkt mit Wahrscheinlichkeit 1 zu der Gaußverteilung, welche dem Datenpunkt die niedrigste Energie gibt (d.h. die höchste Wahrscheinlichkeitsdichte), zugewiesen wird. Die Entropie kann maximiert werden, indem jeder Datenpunkt mit gleicher Wahrscheinlichkeit zu jeder Gaußverteilung zugeordnet und dabei die Energie ignoriert wird. Der beste Kompromiss ist die Zuordnungswahrscheinlichkeiten proportional zu $exp(-E)$ zu setzen. Dies ist als Boltzmann-Verteilung in der Physik bekannt oder als A-Posteriori-Verteilung in der Statistik. Da der E-Step die freie Energie mit Respekt zu den Zuordnungswahrscheinlichkeiten minimiert und der M-Step die Entropie unverändert lässt, aber die erwartete Energie mit Respekt zu den Parametern der Gaußverteilung minimiert, ist die freie Energie eine Zielfunktion für beide Schritte [HSF18a, S. 12].

Präzisierungen: In der Physik (speziell in der Thermodynamik) wird von der inneren Energie gesprochen, welche die gesamte im System vorhandene Energie darstellt. Von dieser Energie kann jedoch nur ein Teil genutzt werden um Arbeit zu verrichten. Dieser Teil heißt freie Energie. Da die nicht nutzbare Energie oft in Form von Hitze auftritt, ist sie das Produkt aus der Temperatur und der Entropie (Maß für Unordnung bzw. spezieller, das Maß für thermale Energie die keine Arbeit verrichten kann). Daher ist die freie Energie ein Maß für die Arbeit, die ein System verrichten kann und ergibt sich aus der Differenz zwischen innerer Energie und der Energie mit der keine Arbeit verrichten werden kann, welche aus dem Produkt von Temperatur und Entropie berechnet wird [Tdf].

Der E-Step reduziert bei der Neuberechnung der Zuordnungswahrscheinlichkeiten (der Capsule-Aktivitäten) ebenfalls eine Energie. Diese freie Energie entspricht der Summe der (skalierten) Kosten, welche in der logistischen Funktion verwendet werden, minus der Entropie der Aktivierungswerte (später ausführlicher betrachtet). Da im M-Step nur die Mittelwerte und Varianzen berechnet werden, bleibt dort die Entropie unberührt und nur die oben erwähnte Energie wird minimiert. Folglich reduziert die Neuberechnung der Zuordnungswahrscheinlichkeiten die Summe der beiden Energien des E- und M-Steps minus der Entropie der Zuordnungswahrscheinlichkeiten, weshalb die freie Energie eine Zielfunktion für beide Schritte darstellt [HSF18b][4]. Intuitiv ergibt das Sinn, da wenn das Modell viele Zuordnungswahrscheinlichkeiten besitzt, welche hohe Werte aufweisen (z.B. 0.8) (entspricht hohe Entropie), eine hohe Unstimmigkeit im Netz herrscht und daher die Pfade durchs Netz „unordentlich“ erscheinen. Werden jedoch wenige Votes (Datenpunkt) mit der Wahrscheinlichkeit 1 zu einer Capsules (Gaußverteilung) zugeordnet, sind die Pfade klar ersichtlich und das Netz erscheint daher „ordentlich“. Wird weiter jeder Datenpunkt zu jeder Gaußverteilung mit gleicher Wahrscheinlichkeit zugeordnet, herrscht maximale Unordnung, und die Entscheidung des Netzes ist nicht nachvollziehbar. Jedoch muss entschieden werden welcher Datenpunkt wie zugeordnet wird. Daher schlägt Hinton et al. [HSF18a, S. 12] vor, die Zuordnungswahrscheinlichkeiten proportional zu $exp(-E)$ setzen, was als Boltzmann-Verteilung in der Physik oder der A-Posteriori-Verteilung in der Statistik bekannt ist. Kurz erläutert stellt die Boltzmann-Verteilung eine Wahrscheinlichkeitsverteilung dar, welche die Verteilung von Partikel in einem System über verschiedene Zustände beschreibt. Diese Verteilung wird durch $exp(-E/kT)$ beschrieben, wobei kT das Produkt aus der Boltzmann-Konstante und der Temperatur des Systems darstellt. Daher entspricht $exp(-E)$ hier der Boltzmann-Verteilung (mit $kT = 1$). Das Wort „System “ hat bei der Boltzmann-Verteilung eine weite Bedeutung, weshalb durch diese Verteilung viele Probleme gelöst werden können (interessanterweise auch dieses) [Bd].
Weiter besteht eine Analogie zur A-Posteriori-Verteilung der Statistik, da die Wahrscheinlichkeitsverteilung (die Softmax Funktion), welche die freie Energie minimiert, durch Verwendung von $exp(-E)$ der A-Posteriori-Verteilung der Anpassung einer Gaußverteilung auf die Eingabe entspricht [HZ94, S. 6].

Änderung 1: Eine Mischung von transformierenden Gaußverteilungen
In einer Standardmischung von Gaußverteilungen werden jeder Gaußverteilung nur eine Teilmenge der Datenpunkte zugeordnet, aber alle Gaußverteilungen sehen dieselben Daten. Wenn eine höher liegende Capsule als diese Gaußverteilungen betrachtet wird, und die Mittelwerte der aktiven Capsules im niedrigen Layer als der Datensatz, dann sieht jede Gaußverteilung einen Datensatz, bei welchem die Datenpunkte durch Transformationsmatrizen transformiert wurden.

Doch diese Matrizen sind unterschiedlich für verschiedene Gaußverteilungen: Für eine höher liegende Capsule, könnten zwei transformierte Datenpunkte näher zusammenliegen und für eine andere höher liegende Capsule könnten die selben zwei Datenpunkte in zwei Punkte weit voneinander weg transformiert worden sein. Jede Gaußverteilung hat daher einen andere Sicht auf die Daten [HSF18a, S. 13].

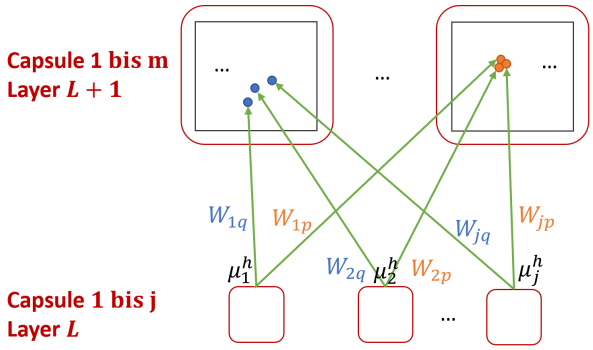


Abbildung 3.28: Capsules in Layer L als Gaußverteilung mit Sicht auf die transformierten Mittelwerte bzw. Erwartungswert der Capsules in $L + 1$

Abbildung 3.28 veranschaulicht dies abstrakt und zeigt, dass die Capsules im oberen Layer $L+1$ durch die unterschiedlichen Transformationsmatrizen (Gewichte W) verschiedene Sichten auf die Datenpunkte (Mittelwerte/Erwartungswert μ) der unteren Capsules aufweisen. Jeder transformierte Datenpunkt im Layer $L + 1$ entspricht hier abstrahiert der gesamten transformierten Pose-Matrix (μ_j^h bzw. M_j) einer niedrigeren Capsule. Wie bereits Abbildung 3.26 in Abschnitt 3.4.2 zeigt berechnet jede höhere Capsule pro Eintrag in der Pose-Matrix den Mittelwert bzw. Erwartungswert und die Varianz einer Gaußverteilung und sendet daher $4 \times 4 = 16$ Mittelwerte in Matrixform an die oberen Capsules.
Die Transformationsmatrizen sind laut Hinton et al. [HSF18a, S. 13] ein weit effektiverer Weg Symmetrie zu brechen, als einfach die Gaußverteilungen mit unterschiedlichen Mittelwerten zu initialisieren und führt generell zu weit schnellerer Konvergenz, was von Hinton et al. [HSF18a, S. 13] und auch hier nicht weiter belegt wird.

Wenn der Fitting-Prozedur (bei Matrix Capsules das EM-Routing) erlaubt wird, die Transformationsmatrizen zu modifizieren, existiert eine triviale Lösung bei welcher alle Transformationsmatrizen zu null kollabieren und daher alle transformierten Datenpunkte identisch sind. Dieses Problem wird vermieden indem die Transformationsmatrizen durch Backpropagation in der äußeren Schleife gelernt werden (wie bei den original Capsules) und das dynamische Routing auf die Änderung der Mittelwerte und Varianz der Gaußverteilungen, sowie die Wahrscheinlichkeiten mit welcher die Datenpunkte zu dieser Gaußverteilungen zugeordnet werden, beschränkt wird [HSF18a, S. 13].

Jedoch existiert noch eine weit subtilere Version des Kollabierungsproblems, welches auftritt wenn verschiedene Transformationsmatrizen unterschiedliche Determinanten besitzen: Wird angenommen, dass die Datenpunkte einer bestimmten Teilmenge in eine Gruppe von Datenpunkten im Pose-Raum der höher liegenden Capsule j transformiert werden und ebenfalls in eine unterschiedliche aber gleich enge Gruppe von Datenpunkten im Pose-Raum der höher liegenden Capsule k, könnte gefolgert werden, dass j und k gleich gute Modelle für die Teilmenge der Datenpunkte bereitstellen. Jedoch ist dies von einer Generative-Modeling-Perspektive nicht korrekt: Wenn die Transformationsmatrizen, welche die Datenpunkte in den von Capsule j genutzten Pose-Raum transformieren, größere Determinanten besitzen, dann stellt j ein besseres Modell zur Verfügung. Der Grund dafür ist, dass die Wahrscheinlichkeitsdichte der Punkte des Pose-Raums der niedrigen Capsule durch die Determinante der relevanten Transformationsmatrix gestreckt wird, wenn diese auf die Pose einer Capsule höherer Ebene abgebildet wird [HSF18a, S. 13]. Aus Sicht der linearen Algebra wird dies klarer: Wird eine Transformation von Vektoren mittels Matrixmultiplikation durchgeführt, so bestimmt die Determinate der Matrix den Faktor um welchen sich die Fläche, welche von den Vektoren aufgespannt wird, skaliert (z.B. Strecken, Stauchen). So können hier die Datenpunkte des Pose-Raums der unteren Capsule bei Multiplikation mit der Transformationsmatrix durch deren Determinaten im Pose-Raum der oberen Capsule gestreckt werden. Führt nun Modell j unter einer stärkeren Streckung zu einer gleich engen Gruppe wie Modell k, stellt dieses ein robusteres, daher besseres Modell dar. Diese Streckung der Wahrscheinlichkeitsdichte wäre ein ernsthaftes Problem, wenn die Transformationsmatrizen durch Maximierung der Wahrscheinlichkeit der beobachteten Datenpunkte gelernt werden sollen. Jedoch werden die Transformationsmatrizen durch Backproberation gelernt, weshalb dies keine Rolle spielt. Allerdings folgt daraus, dass aus der Maximierung der Wahrscheinlichkeiten der transformierten Datenpunkte durch das dynamische Routing nicht zusätzlich die Wahrscheinlichkeit der untransformierten Datenpunkte maximiert werden [HSF18a, S. 13], denn diese wurden schlichtweg vor dem Fitting durch die, vom Fitting unabhängigen, Transformationsmatrizen transformiert. Jedoch müssen streng betrachtet die untransformierten Datenpunkte für die Fehlerberrechnung der Votes ebenfalls gelernt werden.

Der offensichtliche Weg das Determinatenproblem hinsichtlich des Kollabierungsproblems zu vermeiden und den Fehler für die Votes zu berechnen, ist das Abbilden des Mittelwerts des Pose-Raums der höheren Capsule zurück zum Pose-Raum der niedrigeren Capsule unter Nutzung der inversen Transformationsmatrix. Denn ein Mittelwert in einem höher liegenden Pose-Raums kann sich allgemein zu unterschiedlichen Punkten in Pose-Räume verschiedener niedrigerer Capsules abbilden, da die Pose des Ganzen grundsätzlich unterschiedliche Vorhersagen für die Pose der verschiedenen Teile des Ganzen machen wird. Entscheidungen über das Rou-

ting können somit basierend auf einem fairen Vergleich, wie gut die top-down Vorhersage der Pose der unteren Capsule durch die obere Capsule mit der aktuellen Pose der niedrigeren Capsule übereinstimmt, getroffen werden [HSF18a, S. 13].
Doch hierbei wird die korrekte Methode (die inverse Transformationsmatrix) aus zwei Gründen nicht genutzt: Erstens muss dabei eine Matrix invertiert werden (großer Rechenaufwand). Zweitens wird immer eine neue Multiplikation mit den inversen Transformationsmatrizen benötigt, wenn das dynamische Routing den höher liegenden Mittelwert modifiziert. Deshalb wird der Fehler zwischen der aktuellen Pose der niedrigeren Capsule und der „top-down Vorhersage“ der Pose im höher gelegenen Pose-Raum gemessen. Dadurch werden Matrixinvertierungen vermieden und, weit wichtiger, die Multiplikation mit einer inversen Matrix in jeder Iteration des dynamischen Routings. Dies erlaubt mehrere Routing-Iterationen, mit denselben Kosten wie einer Vorwärtspropagierung durch die Transformationsmatrizen [HSF18a, S. 13]. Es wird schlichtweg ignoriert, dass aus der Maximierung der Wahrscheinlichkeiten der transformierten Datenpunkte durch das dynamische Routing nicht zusätzlich die Wahrscheinlichkeit der untransformierten Datenpunkte maximiert werden.

Änderung 2: Eine Mischung von austauschbaren, transformierenden Gaußverteilungen
In einer Standardmischung von Gaußverteilungen sind die modifizierbaren Parameter die Mittelwerte, die (Ko)varianzen und die Mischverhältnisse. Der Wert dieser Parameter ist das einzige, was diese verschiedenen Gaußverteilungen voneinander unterscheidet. In einer Mischung von transformierenden Gaußverteilungen unterscheiden sich diese zusätzlich durch die Transformationsmatrizen, welche die Gaußverteilungen auf die Datenpunkte anwenden. Wenn diese Transformationsmatrizen während dem Fitting der anderen Parameter fixiert sind (was sie hier sind, da diese nicht vom Routing sondern zuvor mittels Backpropagation verändert wurden), ergibt es zwar Sinn eine große Menge transformierender Gaußverteilungen parat zu haben, jedoch sollte nur die kleinste Teilmenge der geeigneten Transformationsmatrizen genutzt werden, um die vorgelegten Daten zu erklären. Die Anpassung an einen Datensatz beinhaltet dann die Entscheidung, welche der transformierenden Gaußverteilungen „eingeschaltet“ werden sollen. Deshalb wird jeder transformierenden Gaußverteilung ein zusätzlicher Aktivierungsparameter, welcher die Wahrscheinlichkeit darstellt mit welcher die Gaußverteilung für den aktuellen Datensatz eingeschaltet wird, gegeben - die Aktivierungsparameter sind hierbei keine Mischverhältnisse, da diese nicht zu 1 aufsummieren [HSF18a, S. 12-13].

Um die Aktivierungswahrscheinlichkeiten a_j für eine bestimmte höher liegende Capsule j zu errechnen, kann offensichtlich nicht einfach wie beim ersten CapsNet die Orientierung der Vektoren zueinander genutzt werden, da es sich hier um Datenpunkte handelt die unter einer Gauß-

verteilung liegen. Daher wird ein Vergleich zwischen zwei Beschreibungslängen (engl. description lengths) durchgeführt, welche auf zwei verschiedenen Wegen die Posen der aktiven durch das Routing zu j zugeordneten Capsules codieren [HSF18a, S. 13].
Um die zwei verschiedenen Wege zu erläutern wird angenommen, dass die Posen und Aktivierungen aller Capsules eines Layers bereits berechnet wurden. Nun soll entschieden werden, welche Capsule im höheren Layer aktiviert und wie jede aktive niedrigere Capsule zu eine aktiven höheren Capsule zugeordnet werden soll. Jede Capsule im höheren Layer korrespondiert, wie bereits erwähnt, mit einer Gaußverteilung und die Pose (aus Mittelwerten) jeder aktiven Capsule im Layer darunter (konvertiert zu einem Vektor) korrespondiert mit einem Datenpunkt (oder einem Bruchteil eines Datenpunktes wenn die Capsule teilweise aktiv ist) [HSF18a, S. 2]. Hierbei wird das Minimale-Beschreibungslängen-Prinzip (engl. minimum description length, kurz MDL) genutzt. Dieses Prinzip ist, kurz erläutert, eine Formalisierung von Ockhams Rasiermesser, bei der die beste Hypothese (ein Modell und seine Parameter) für eine gegebene Datenmenge diejenige ist, die zu der besten Komprimierung der Daten führt (was hier wohl bildlich wenigen aktiven Capsules mit dichten Gaußverteilungen entspricht). Wie andere statistische Methoden kann MDL zum Lernen der Parameter eines Modells mittels eines Datensatzes verwendet werden [Gru14].
Im Fall dieses CapsNets ist „Beschreibungslänge“ nur ein anderes Wort für Energie [HSF18a, S. 13]. Dies wird von Hinton et al. [HSF18a] ebenfalls nicht genauer erläutert, jedoch kann gefolgert werden, dass, da hier das Minimieren des Abstandes zu den Mittelwerten als Minimieren einer physikalische Energie betrachtet werden kann, ebenfalls das Entscheiden darüber, welche Capsules (und somit welcher Vote) im niedrigeren Layer aktiv sein sollen, die Energie beeinflusst, da die Anzahl der aktiven Votes direkt mit der Energie korrespondiert. Daher können die Beschreibungslängen, welche aus Codierungen der Posen der Capsules (der Votes) entsteht, ebenfalls als Energien betrachtet werden.
Als zweite Analogie wird in Hinton et al. [HZ94], die Verbindung zwischen der minimalen Beschreibungslänge einer Nachricht (die minimale Länge die für die Rekonstruktion durch den Empfänger benötigt wird) und der freien Energie ausführlich erläutert und auf Autoencoder angewandt. Dabei ist der Sender der Encoder und der Empfänger der Decoder.
Für Matrix Capsules finden Hinton et al. [HSF18a] unter Nutzen der MDL zwei Auswahlmöglichkeiten bei der Entscheidung ob oder ob nicht eine höher liegende Capsule aktiviert werden soll:
Wahlmöglichkeit 0: Wird die Capsule nicht aktiviert, muss ein fester Preis (Cost) von $-\beta_u$ pro Datenpunkt gezahlt werden und zwar für das bereits geschehene Beschreiben der Posen aller niedrigeren Capsule, welche zu dieser höheren Capsule zugeordnet wurden. Diese Cost $-\beta_u$ ist die negative, logarithmierte Wahrscheinlichkeitsdichte des Datenpunkts unter einem unsachge-

mäßen gleich verteilten Prior (improper uniform prior). Hierbei ist $-\beta_u$ unsachgemäß, da dies vom Backprobergation gelernt wird und daher keiner richtigen Verteilung abstammt. Für bruchstückweise Zuordnungen wird ein Teil der Kosten gezahlt [HSF18a, S. 2].
Wahlmöglichkeit 1: Wird die höhere Capsule aktiviert, muss ein fester Preis von $-\beta_a$ gezahlt werden und zwar für die Codierung der Mittelwerte und der Varianz sowie dem Fakt, dass die Capsule aktiv ist. Zusätzlich müssten Kosten anteilig der Zuordnungswahrscheinlichkeiten für die Beschreibung der Diskrepanzen zwischen der niedrigen Mittelwerten und den vorhergesagten Werten gezahlt werden, würden die Mittelwerte der höher liegende Capsule dazu verwendet werden, die Mittelwerte via der inversen Transformationsmatrix vorherzusagen. Aber wie bereits erläutert wäre der Weg über die Inverse sehr ineffizient, weshalb als einfacherer Weg für die Berechnung der Kosten für das Beschreiben eines Datenpunkts die negative, logarithmierte Wahrscheinlichkeitsdichte der Stimme (des Votes) des Datenpunkts unter einer Gaußverteilung, welche auf die zugeordnete höhere Capsule angepasst (gefittet) wurde, verwendet wird [HSF18a, S. 2].

Wird die effiziente Approximation von Wahlmöglichkeit 1 genutzt, werden die Kosten eines ganzen Datenpunkts i unter Nutzen einer aktiven Capsule j, welche eine Kovarianz-Matrix besitzt über die Gaußverteilung berechnet.

$$P_{i|j}^h = \frac{1}{\sqrt{2\pi(\sigma_j^h)^2}} exp\left(-\frac{(V_{ij}^h - \mu_j^h)^2}{2(\sigma_j^h)^2}\right) \tag{3.12}$$

$$ln(P_{i|j}^h) = -\frac{(V_{ij}^h - \mu_j^h)^2}{2(\sigma_j^h)^2} - ln(\sigma_j^h) - \frac{ln(2\pi)}{2} \tag{3.13}$$

Die Gesamtkosten des Datenpunkts ist die Summe über alle Dimensionen der Kosten für die Erklärung jeder Dimension h des Votes V_{ij}. Dabei besitzt jede Dimension h ein eigenes Gauß-Modell mit ihre eigener Varianz und eigenem Mittelwert, da es sich um eine mehrdimensionale Gaußverteilung handelt. Die Kosten einer Dimension sind dann $-ln(P_{i|j}^h)$ wobei $P_{i|j}^h$ die Wahrscheinlichkeitsdichte des h^{ten} Eintrags des vektorisierten Votes V_{ij} unter dem Gauß-Modell von Capsule j für Dimension h ist, welche die Varianz $(\sigma_j^h)^2$ und den Mittelwert μ_j^h besitzt. Dabei stellt μ_j die vektorisierte Version der Pose-Matrix M_j von Capsule j dar [HSF18a, S. 2].
Wird nun über alle Votes der niedrigen Capsules für eine einzige Dimension h von Capsule j

aufsummiert, werden folgende Kosten für Dimension h erhalten:

$$\begin{aligned} cost_j^h &= \sum_i -r_{ij} ln(P_{i|j}^h) \\ &= \frac{\sum_i r_{ij}(V_{ij}^h - \mu_j^h)^2}{2(\sigma_j^h)^2} + \left(ln(\sigma_j^h) + \frac{ln(2\pi)}{2} \right) \sum_i r_{ij} \\ &= \left(ln(\sigma_j^h) + \frac{1}{2} + \frac{ln(2\pi)}{2} \right) \sum_i r_{ij} \end{aligned} \tag{3.14}$$

Hierbei ist $\sum_i r_{ij}$ die Summe der Zuordnungswahrscheinlichkeiten (Coupling-Koeffizienten), welche die Anzahl an Daten die zu Capsule j zugeordnet werden angeben und V_{ij}^h ist ein einzelner Wert der Dimension h von V_{ij}.

Der Unterschied der beiden Kosten aus Wahlmöglichkeit 0 und 1 wird dann für alle h Dimensionen (es wird wie oben erwähnt über alle Dimensionen aufsummiert) bei jeder Iteration in eine logistische Funktion gegeben, um die Aktivierungswahrscheinlichkeit der höheren Capsule j zu erhalten [HSF18a, S. 2].

$$a_j = logistic \left(\lambda \left(\beta_a - \beta_u \sum_i r_{ij} - \sum_h cost_j^h \right) \right) \tag{3.15}$$

Hierbei ist β_a für alle Capsules gleich und λ ein inverser Temperaturparameter. Die Bias-Parameter β_a und β_u werden durch Backpropergation gelernt und λ besitzt einen festen Zeitplan. Die Werte dieses Zeitplans wurden nicht angegeben, jedoch kann λ mit der Temperatur von einem Simulated-Annealing verglichen werden. Außerdem werden die Kosten $-\beta_u$ aus Wahlmöglichkeit 0 ebenfalls mit dem Zuordungswahrscheinlichkeiten r_{ij} gewichtet.
Die logistische Funktion berechnet dabei die Verteilung, welche die freie Energie minimiert, da die Differenz der Energien der beiden Kosten das Argument der logistischen Funktion ist [HSF18a, S. 13].

Wie bereits erläutert berechnet die logistische Funktion die Aktivierungswahrscheinlichkeiten. Ebenfalls wurde bei der Verbindung zwischen freier Energie und der statistischen A-Posteriori-Verteilung erwähnt, dass die Softmax Funktion die Verteilung, welche die freie Energie minimiert, berechnet. Dies gilt hier, wenn die logarithmierten Aktivierungswahrscheinlichkeiten als negative Energien betrachtet werden, was aus der Verbindung zwischen freier Energie und der statistischen a-posteriori Verteilung hervorgeht. Daher wird die freie Energie minimiert, wenn die Softmax Funktion in der Routing-Prozedur verwendet wird, um die Zuordnungswahrscheinlichkeiten neu zu berechnen. Bei einer Neuanpassung des gaußschen Modells jeder Capsule, wird dieselbe freie Energie minimiert, vorausgesetzt die Logits der Softmax Funktion basieren auf den gleichen Energien, wie diejenigen, die beim Anpassen der Gaußverteilung optimiert

werden. Hierbei handelt es sich um dieselben freien Energien, da, wie bereits erläutert, der E- und M-Step des CapsNets dieselbe freie Energie minimieren. Jedoch muss erwähnt werden, dass die verwendeten Energien, die negativen, logarithmierten Wahrscheinlichkeiten der Votes der niedrigeren Capsules unter dem gaußschen Modell der höher liegenden Capsule sind und daher nicht die korrekten Energien darstellen, um die logarithmierten Wahrscheinlichkeiten der Daten zu maximieren, da aus oben beschriebenen Gründen nicht die inverse Matrix verwendet wird. Dies spielt jedoch für die Konvergenz keine Rolle, solange dieselben Energien für das Fitting der Gaußverteilungen und die Anpassung der Zuordnungswahrscheinlichkeiten genutzt werden [HSF18a, S. 12].
Die Zielfunktion minimiert daher Gleichung 3.16, welche sich aus allen oberen Erkenntnissen ergibt und diese gleichzeitig zusammenfasst:

$$\sum_{j \epsilon \Omega_{L+1}} a_j(-\beta_a) + a_j ln(a_j) + (1-a_j) ln(1-a_j) + \sum_h cost_j^h + \beta_u \sum_{j \epsilon \Omega_L} r_{ij} + \sum_{j \epsilon \Omega_L} a_i * r_{ij} * ln(r_{ij}) \tag{3.16}$$

Diese Gleichung besteht aus folgenden Teilen:

$$\sum_{j \epsilon \Omega_{L+1}} a_j(-\beta_a) + a_j ln(a_j) + (1-a_j) ln(1-a_j) + \sum_h cost_j^h + \beta_u \sum_{j \epsilon \Omega_L} r_{ij} + \sum_{j \epsilon \Omega_L} a_i * r_{ij} * ln(r_{ij})$$

Abbildung 3.29: Die farblich eingeteilte Zielfunktion der Matrix Capsules

- *Rot*: Die MDL Kosten $-\beta_a$, skaliert mit den Wahrscheinlichkeit der Präsenz der Capsules in Layer $L+1$ $(a_j, j \epsilon \Omega_{L+1})$ (folgt aus einem Teil von Wahlmöglichkeit 1).
- *Blau*: Die negative Entropie der Aktivierungen a_j mit $j \epsilon \Omega_{L+1}$ (folgt aus der Diskussion des Zusammenhangs mit physikalischer Energie und der austauschbaren Gaußverteilungen, sowie aus dem Zusammenhang mit der Beschreibungslänge einer Nachricht).
- *Orange* und *Gelb*: Die erwartete Energie, welche im M-Step minimiert wird, also die Summe der gewichteten, logarithmierten Wahrscheinlichkeiten $(cost_j^h)$ (folgt aus dem anderen Teil von Wahlmöglichkeit 1) und des festen Preises β_u pro Datenpunkt (folgt aus Wahlmöglichkeit 0).
- *Grün*: Die negative Entropie der Routing-Softmax-Zuordnungen (R_{ij}), skaliert mit der Präsenzwahrscheinlichkeit/Aktivierungswahrscheinlichkeit des Datenpunkts $(a_i, j \epsilon \Omega_L)$ (folgt aus der Diskussion des Zusammenhangs mit physikalischer Energie und der austauschbaren Gaußverteilungen).

Die Energien zur Berechnung der Zuordnungswahrscheinlichkeiten sind dieselben Energien wie die, die für das Fitting der Gaußverteilungen und zur Berechnung der Aktivierungswahrscheinlichkeiten genutzt wurden, weshalb alle drei Schritte dieselbe freie Energie minimieren, jedoch mit Respekt zu verschiedenen Parametern [HSF18a, S. 14].
Außerdem wird das Produkt der Zuordnungswahrscheinlichkeiten und Aktivierungswahrscheinlichkeiten als ein Multiplikator verwendet; sowohl für die Beschreibungslänge jedes Mittelwerts der niedrigeren Ebene, als auch für die Beschreibungslänge, die unter Nutzung des Fittings der Gaußverteilung durch eine höher liegende Capsule erhalten wird [HSF18a, S. 14]. Dies wird aus folgnder Erklärung des Pseudocodes 3.30 ersichtlich.

Procedure 1 Routing algorithm returns **activation** and **pose** of the capsules in layer $L+1$ given the **activations** and **votes** of capsules in layer L. V_{ij}^h is the h^{th} dimension of the vote from capsule i with activation a_i in layer L to capsule j in layer $L+1$. β_a, β_u are learned discriminatively and the inverse temperature λ increases at each iteration with a fixed schedule.

procedure EM ROUTING($\boldsymbol{a}$, V)
  $\forall i \in \Omega_L, j \in \Omega_{L+1}$: $R_{ij} \leftarrow 1/|\Omega_{L+1}|$
  for t iterations **do**
    $\forall j \in \Omega_{L+1}$: M-STEP($\boldsymbol{a}$, R, V, j)
    $\forall i \in \Omega_L$: E-STEP(μ, σ, $\boldsymbol{a}$, V, i)
  return $\boldsymbol{a}$, M

procedure M-STEP($\boldsymbol{a}$, R, V, j) ▷ for one higher-level capsule, j
  $\forall i \in \Omega_L$: $R_{ij} \leftarrow R_{ij} * \boldsymbol{a}_i$
  $\forall h$: $\mu_j^h \leftarrow \frac{\sum_i R_{ij} V_{ij}^h}{\sum_i R_{ij}}$
  $\forall h$: $(\sigma_j^h)^2 \leftarrow \frac{\sum_i R_{ij} (V_{ij}^h - \mu_j^h)^2}{\sum_i R_{ij}}$
  $cost^h \leftarrow \left(\beta_u + log(\sigma_j^h)\right) \sum_i R_{ij}$
  $a_j \leftarrow logistic(\lambda(\beta_a - \sum_h cost^h))$

procedure E-STEP(μ, σ, $\boldsymbol{a}$, V, i) ▷ for one lower-level capsule, i
  $\forall j \in \Omega_{L+1}$: $\boldsymbol{p}_j \leftarrow \frac{1}{\sqrt{\prod_h^H 2\pi(\sigma_j^h)^2}} \exp\left(-\sum_h^H \frac{(V_{ij}^h - \boldsymbol{\mu}_j^h)^2}{2(\sigma_j^h)^2}\right)$
  $\forall j \in \Omega_{L+1}$: $\boldsymbol{R}_{ij} \leftarrow \frac{\boldsymbol{a}_j p_j}{\sum_{k \in \Omega_{L+1}} \boldsymbol{a}_k p_k}$

Abbildung 3.30: EM-Routing Pseudocode [HSF18a, S. 3]

Im folgenden wird das EM-Routing 3.30 sowie die zu Beginn vorgestellte Abbildung zur Übersicht 3.26 Schritt für Schritt erläutert. So wird in *Zeile 1* Prozedur *EM ROUTING* als Eingabeparameter für die Berechnung der Aktivierung a und der Pose M des Layers $L+1$, die Aktivierung a und die Votes V des vorherigen Layers L gegeben. Weiter werden alle Zuordnungswahrscheinlichkeiten zwischen Layer L und $L+1$ auf $1/\Omega_{L+1}$ gesetzt, wobei Ω_{L+1} die Anzahl an Capsules in Layer $L+1$ darstellt.
Für alle Capsules i in Layer L und j in Layer $L+1$ wird dann der M- und E-Step für t Iteratio-

nen alterniert:

M-Step: Die Prozedur $M - STEP$ wird für jede Capsule aus Layer $L + 1$ ausgeführt und erhält als Eingabe die Aktivierung a und die Votes V des vorherigen Layers L, die Zuordnungswahrscheinlichkeiten R_{ij} und den Index der aktuelle Capsule j aus Layer $L + 1$. Die Votes V ergeben sich aus Multiplikation der Pose-Matrix des Layers L mit der Transformationsmatrix W_{ij}, sprich $V_{ij} = M_i W_{ij}$.
Dann werden in *Zeile 2* die Zuordnungswahrscheinlichkeiten R_{ij} mit den Aktivierungen a_i jeder unteren Capules i wie folgt multipliziert:

$$R_{ij} \leftarrow R_{ij} * \mathbf{a}_i \tag{3.17}$$

Dies bringt den Vorteil, dass gering aktive Capsules in Layer i auch eine geringere Zuordnungswahrscheinlichkeiten erhalten, was sinnvoll ist, da eine kaum aktive Capsule nicht ohne Grund kaum aktiv ist, und daher nicht durch eine hohe Zuordnung einen starken Ausschlag geben soll. So können, wie erläutert, die besseren transformierenden Gaußverteilungen stärker gewichtet werden und schlechte sogar genullt werden (sie werden austauschbar). So wird, wie bereits erwähnt, dieses Produkt weiter als ein Multiplikator verwendet; sowohl für die Beschreibungslänge jedes Mittelwerts der niedrigeren Ebene, als auch für die Beschreibungslänge, die unter Nutzung des Fittings der Gaußverteilung durch eine höher liegende Capsule erhalten wird [HSF18a, S. 14]
In *Zeile 3* wird Anhand den Votes V_{ij}^h für jede Dimension h der Mittierwert μ_j^h wie folgt errechnet:

$$\mu_j^h \leftarrow \frac{\sum_i R_{ij} V_{ij}^h}{\sum_i R_{ij}} \tag{3.18}$$

Hier sind wie erwartet die Ähnlichkeiten zu der M-Step-Gleichung 3.9 des EM-Algorithmus mit $\boldsymbol{\mu}_i \leftarrow \sum_j p_{ij} \mathbf{x}_j / n_i$ zu erkennen, wenn die Zuordnungswahrscheinlichkeiten R_{ij} als Wahrscheinlichkeit p_{ij}, dass die Komponente (die Capsule) den Datenpunkt $\mathbf{x}_j$ (den Vote V_{ij}^h) generiert hat, betrachtet wird. Die effektive Nummer der Datenpunkte n_i ist beim EM-Algorithmus durch $n_i = \sum_j p_{ij}$ definiert, was dem Teilen durch $\sum_i R_{ij}$ im EM-Routing entspricht.
Auch die in *Zeile 4* berechnete Varianz der Votes V_{ij}^h für jede Dimension h ähnelt der Berechnung der Varianz des EM-Algorithmus in Gleichung 3.9:

$$(\sigma_j^h)^2 \leftarrow \frac{\sum_i R_{ij} (V_{ij}^h - \mu_j^h)^2}{\sum_i R_{ij}} \tag{3.19}$$

Hier muss jedoch durch die Eigenschaften der Pose-Matrix (symmetrisch) keine Multiplikation von $(V_{ij}^h - \mu_j^h)$ mit deren Transponierten durchgeführt werden, da das Quadrieren von $(V_{ij}^h - \mu_j^h)$ hierbei dieselbe Wirkung hat.
In *Zeile 5* kann nun die Kosten $cost^h$ der Wahlmöglichkeit 1, wie oben beschrieben, für jede

Dimension h berechnet werden:

$$cost^h \leftarrow (\beta_u + log(\sigma_j^h)) \sum_i R_{ij} \tag{3.20}$$

Diese wird in *Zeile 6* in die ebenfalls zuvor beschriebene logistische Funktion gegeben:

$$a_j \leftarrow logistic(\lambda(\beta_a - \sum_h cost^h)) \tag{3.21}$$

Die Gleichungen 3.20 und 3.21 unterscheiden sich dabei leicht von den entsprechenden oberen Gleichungen 3.14 und 3.15. Dieser Unterschied kommt jedoch nur durch Umstellungen zwischen den beiden Gleichungen und dem Weglassen der Konstanten $\frac{1}{2} + \frac{ln(2\pi)}{2}$ in Gleichung 3.14 für $cost_j^h$.

E-Step:
Nun berechnen im E-Step die unteren Capsules i für die oberen Capsules j ihre neuen Zuordnungswahrscheinlichkeiten. Dazu wird erst die Wahrscheinlichkeit $\mathbf{p}_j$ berechnet, die angibt wie gut der Vote V_{ij} der unteren Capsule mit dem Gauß-Modell der oberen Capsule in allen h Dimensionen übereinstimmt:

$$\mathbf{p}_j \leftarrow \frac{1}{\sqrt{\prod_h^H 2\pi(\sigma_j^h)^2}} exp\left(-\sum_h^H \frac{(V_{ij}^h - \mu_j^h)^2}{2(\sigma_j^h)^2}\right) \tag{3.22}$$

Diese Gleichung ähnelt der Gleichung der Gaußverteilung, die für die Erklärung der Kosten für eine Dimension h für die höher liegende Capsule herangezogen wurde, bei welcher für die Mehrdimensionalität über alle einzelnen Kosten einer Dimension h aufsummiert wird. Der Unterschied hierbei ist, dass nun ein Modell für mehrdimensionale, austauschbare, transformierende Gaußverteilungen für die unteren Capsules benötigt wird. Austauschbar und transformierend ist das Modell bereits durch die Aktivierungswahrscheinlichkeiten und die Transformationsmatrizen zwischen den Layern, weshalb diese Gleichung eine mehrdimensionale Gaußverteilungen darstellt, welche h Dimensionen besitzt und somit den unter Capsules ermöglicht das Agreement ihres h-dimensionalen Votes für Capsule j zu ermitteln. Dies wird durch die Summation und das Produkt über die Dimensionen h in der Gleichung ersichtlich. Für eine bildliche Darstellung hilft ein einzelnes Cluster der obigen Abbildung 3.27 bei der Erklärung des EM-Algorithmus, welche eine Mischung von mehrdimensionalen Gaußverteilungen in 2D beinhaltet.
Es muss jedoch beachtet werden, dass das CapsNet durch die Aktivierungswahrscheinlichkeiten und Transformationsmatrizen weit komplexere Gauß-Modelle bildet, vor allem wenn in Betracht gezogen wird, dass die Abbildung 3.27 nur dem Modell einer höheren Capsule j ähnelt, hätte diese eine Pose-Matrix der Größe zwei. Die Pose-Matrix ist jedoch 4x4 groß und die

Anzahl der höher liegenden Capsules liegt, wie später erläutert bei bis zu 32, wobei natürlich nicht jede Capsule aktiv ist.
Letztlich wird anhand den Wahrscheinlichkeit $\mathbf{p}_j$, die neuen Zuordnungswahrscheinlichkeiten $\mathbf{R}_{ij}$ zu allen oberen Capsules j wie folgt errechnet:

$$\mathbf{R}_{ij} \leftarrow \frac{\mathbf{a}_j p_j}{\sum_{k \in \Omega_{L+1}} \mathbf{a}_k p_k} \tag{3.23}$$

Die Wahrscheinlichkeit(sdichte) $\mathbf{p}_j$, die angibt wie gut der Vote V_{ij} der unteren Capsule mit dem Gauß-Modell der oberen Capsule in allen h Dimensionen übereinstimmt, wird in die Softmax Funktion gegeben und mit der Aktivierungswahrscheinlichkeit der zugehörigen Capsule gewichtet, damit letztlich eine austauschbare, transformierte Gaußverteilung durch Beeinflussung der Zuordnungswahrscheinlichkeiten entsteht bzw. angepasst wird. Da die Softmax Funktion die Verteilung, welche die freie Energie minimiert, berechnet und dazu die logarithmierten Aktivierungswahrscheinlichkeiten als negative Energien betrachtet werden müssen, wird die Exponentialfunktion der Softmax gekürzt und es entsteht beispielsweise im Zähler $-\mathbf{a}_j(-p_j) = \mathbf{a}_j p_j$.
Oft wurde und wird $\mathbf{p}_j$, wie in Hinton et al. [HSF18a], nicht explizit als Wahrscheinlichkeitsdichte bezeichnet, was im mathematischen Sinne aus mehreren Gründen nicht korrekt ist, da z.B. die Wahrscheinlichkeitsdichtefunktion der Gaußverteilung in Gleichung 3.22, wie andere Wahrscheinlichkeitsdichtefunktionen angibt wie wahrscheinlich ein Wert (hier ein Vote) in einem gewissen Wertebereich liegt (kontinuierlich) und Wahrscheinlichkeiten hingegen angeben, mit welcher Wahrscheinlichkeit ein bestimmter Wert angenommen wird (diskret, z.B. das Würfeln einer Sechs). Dies sollte für das tiefere Verständnis im Hinterkopf behalten werden.

Das EM-Routing gibt schließlich alle Aktivitäten $\mathbf{a}_j$ und Posen M des Layers $L+1$ der letzten Routing-Iteration zurück. Die Posen M sind dabei die Mittelwerte μ_j in Matrixform.

3.4.4 Architektur des Matrix Capsule Networks

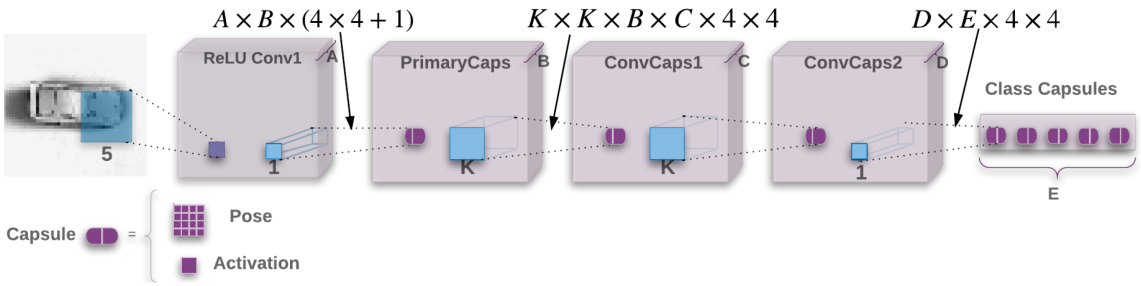


Abbildung 3.31: Architektur des Matrix Capsule Models mit EM-Routing aus [HSF18a, S. 4]

Abbildung 3.31 zeigt die generelle Architektur des im folgenden Abschnitten verwendeten Modells, auch hier werden die Layer von links nach rechts erläutert.

ReLu Conv1: Zu Beginn besitzt das Modell einen 5×5 conv. Layer mit $A = 32$ Feature-Maps, einem Stride von zwei und eine ReLu-Nichtliniarität. Alle folgenden Layer sind Capsule-Layer [HSF18a, S. 3].

PrimaryCaps: Der erste Capsule-Layer wird als *PrimaryCaps* bezeichnet und besitzt $B = 32$ Typen von Primary-Capsules. Jeder dieser B Capsules besitzt eine 4×4 Pose-Matrix $(B \times (4 \times 4 \times 1))$, welche die gelernte lineare Transformation des Outputs aller ReLus im vorherigen Layer darstellt, welche um diesen Ort zentriert sind. Die Aktivierungen der Primary-Capsules werden erzeugt, indem die Sigmoid-Funktion auf die gewichteten Summen die ReLUs des vorherigen Layers angewandt wird [HSF18a, S. 3-4] und die Kernel-Size dieses Layers ist 1×1, weshalb die Transformationsmatrizen zwischen diesem Layer und *ReLu Conv1* die Dimension $(A \times B \times (4 \times 4 \times 1))$ besitzen.

ConvCaps1 und ConvCaps2: Den Primary-Capsules folgen zwei $K \times K$ convolutional Capsule-Layer mit Kernel-Size $K = 3$. Jeder der Layer besitzt 32 Typen an Capsules, weshalb $C = D = 32$ gilt. *ConvCaps1* enthält den Input des PrimaryCaps-Layers, wodurch die Transformationsmatrizen zwischen diesen Layern eine Dimension von $(K \times K \times B \times C \times 4 \times 4)$ aufweisen. Folglich wird *ConvCaps1* mit einem Stride von zwei angewandt, *ConvCaps2* mit einem Stride von eins [HSF18a, S. 4].

Class Capsules: Der letzte conv. Capsule-Layer *ConvCaps2* ist mit dem finale Capsule-Layer *Class Capsules* verbunden, welcher eine Capsule pro Ausgabeklasse besitzt, also E Capsules. Durch diese Verbindung soll keine Information über die Lage der *ConvCaps2* verloren gehen, aber gleichzeitig soll ein Nutzen aus dem Fakt gezogen werden, dass alle Capsules desselben Typen dieselbe Entität an verschiedenen Positionen extrahieren [HSF18a, S. 4]. Capsules die denselben Typ repräsentieren entstehen dadurch, dass conv. Capsules wie die Neuronen in einem conv. Layer, aus der Eingabe ein Gitter über die Kernel-Size und den Stride bilden, welches soviel Capsules-Typen wie Channel in seiner Tiefe besitzt. Dies wurde bereits bei der oberen, originalen Capsule in Abbildung 3.20 bildlich dargestellt. Daher werden zwischen ConvCaps2 und dem ClassCaps-Layer die Transformationsmatrizen derselben Capsule-Typen zwischen verschiedenen Positionen geteilt und die Koordinate (Zeile, Spalte) des Zentrums des rezeptiven Feldes jeder Capsule (die Koordinate des Capsule-Eintrags im Ausgabe-Gitter) zu den ersten beiden Elementen der rechten Spalte ihrer Stimmmatrix addiert (Coordinate Addition).

Dies sollte die geteilten finalen Transformationen dazu anregen Werte für diese zwei Elemente zu erzeugen, welche die feine Position der Entität repräsentieren, die relativ zum Zentrum des rezeptiven Feldes der Capsule ist [HSF18a, S. 4]. Dadurch ist es nicht nötig, dass jede Class Capsule für jeden Capsule-Eintrag im Gitter von ConvCaps2 eine eigene Transformationsmatrix benötigt, um das FC-Kriterium zu erfüllen, weshalb die Transformationsmatrizen zwischen diesem Layer und *ConvCaps2* eine geringe Dimension von $(D \times E \times 4 \times 4)$ besitzen.

Routing: Das EM-Routing wird zwischen jedem benachbarten Paar an Capsule-Layern benutzt. Für die conv. Capsules sendet jede Capsule in Layer $L+1$ nur Feedback zu den Capsules in ihrem rezeptiven Feld in Layer L, daher erhält jede conv. Instanz einer Capsule in Layer L maximal $KernelSize \times KernelSize$ Feedback von jeder Capsule in Layer $L+1$. Die Instanzen näher zum Rand des Bildes erhalten weniger Feedback, wobei Capsules in den Ecken nur ein Feedback pro Capsule-Typ in Layer $L+1$ erhalten [HSF18a, S. 4].

3.4.5 Loss und Training

Damit das Training weniger sensitiv gegenüber den Initialisierungen der Hyperparameter des Modells ist wird ein „Spread Loss“ verwendet, um direkt den Spalt zwischen der Aktivierung der Zielklasse a_t und den anderen Klassen zu maximieren. Wenn die Aktivierung einer falschen Klasse a_i näher als der Margin (engl. für Spanne, Rand) m zu a_t ist, dann wird dies mit den quadratischen Abstand zum Margin bestraft [HSF18a, S. 4]:

$$L_i = (max(0, m - (a_t - a_i))^2, L = \sum_{i \neq t} L_i \tag{3.24}$$

Durch das Starten mit einer geringen Margin von 0.2 und dem linearen Erhöhen zu 0.9 während des Trainings, werden „tote“ Capsules in den früheren Schichten verhindert. Der Spread Loss ist äquivalent zu einem quadratischen Hinge Loss mit $m = 1$ [HSF18a, S. 4].

Das Training läuft folglich sehr ähnlich zu dem Training von CapsNets in Abschnitt 3.3 ab, welche weiter als original CapsNets bezeichnet werden, um diese von Matrix Capsules unterscheiden zu können. Backpropagation dient dabei zum Training der Gewichte, sowie für die Parameter β_a und β_b und die „dazwischenliegenden“ Routing-Iterationen lernen die Zuordnungswahrscheinlichkeiten und Parameter der Gaußverteilungen. Auch hier ergeben drei Routing-Iterationen einen guten Ausgangswert. Der folgende Abschnitt beschreibt das Training anhand konkreter Anwendungen genauer, allerdings wird dabei kein Rekonstruktions-Loss verwendet, weshalb die gelernten Features nicht visuell überprüft werden können. Jedoch werden in der Literatur Matrix Capsules auch mit Rekonstuktions-Loss erfolgreich angewandt. Dies wird in Abschnitt 3.5.5 ausführlicher dargestellt.

3.4.6 Anwendung des Matrix Capsule Networks

Der bereits kurz in Abschnitt 3.3.5 beschriebene smallNORB Datensatz besteht aus Graustufen-Bildern mit fünf Klassen von Spielzeugen: Flugzeugen, Autos, Trucks, Menschen und Tiere. Fünf physische Instanzen jeder Klasse werden für das Training selektiert und die anderen fünf für den Test. Jedes individuelle Spielzeug wurde von 18 verschiedenen Azimute (0-340), neun Höhen und sechs Lichtverhältnisse aufgenommen, woraus folgt, dass der Training- und Testdatensatz jeweils 24300 ($5 \times 5 \times 18 \times 9 \times 6$) von 96x96 Bildern besitzt. smallNORB fand hier als Benchmark Verwendung, da dieser Datensatz sorgsam dafür designed wurde, eine pure Formerkennungsaufgabe darzustellen, welches nicht durch Kontext oder Farbe „verwirrt“, aber näher an natürlichen Bildern liegt als MNIST [HSF18a, S. 4].

Auch hier wurde wie in Abschnitt 3.3.5 eine Verkleinerung der smallNORB-Bilder auf 48x48 Pixel vorgenommen. Eine zusätzliche Normalisierung fand statt, damit die Bilder mittelwertfrei waren und Einheitsvarianz besaßen [HSF18a, S. 5] (Dieses Preprocessing sorgt im Deep Learning Bereich i.d.R. für bessere Ergebnisse). Für das Training wurden dann zufällig 32x32 Stücke aus den Bildern geschnitten und zufällig Helligkeit und Kontrast hinzugefügt [HSF18a, S. 5]. Kein Hyperparameter des Models außer die Netzstruktur wurde in [HSF18a] explizit angegeben. Während des Tests wurden 32x32 Stücke aus der Mitte der Bilder geschnitten und somit ein Test-Error von 1.8% erreicht. Bei einer Mittlung der Klassenaktivierungen zur Testzeit über mehrere verschiedene Stücke konnte ein Test-Error von 1.4% erreicht werden. Im Vergleich zu anderen Modellen beträgt das beste berichtete Ergebnis ohne das Nutzen von Metadaten 2.56% (Ciresan et al. [Cir+11]); dies wurde erreicht indem zwei zusätzliche Bilder als Input hinzugefügt wurden, die durch das Nutzen von on-center off-surround Filter und off-center on-surround Filter erzeugt wurden (das Verfahren wird nicht weiter beschrieben). Außerdem wurden dort zusätzliche affine Verzerrungen auf die Bilder angewandt. Matrix Capsules verzeichnen weiter einen geringeren Test-Error, als die in Abschnitt 3.3 vorgestellten original Capsules, welche 2.7% erreichten. Zusätzlich wurde das Modell mit einer Fehlerrate von 2.6% auch auf NORB getestet, welches eine gejitterten Version von smallNORB mit zusätzlichem Hintergrund darstellt. Diese Fehlerrate entspricht dem State-of-the-Art von 2.7% (siehe Ciresan et al. [CMS12]) [HSF18a, S. 5].

Als Baseline für ein Experiment für das Testen der Generalisierung zu neuen Viewpoints wurde ein CNN trainiert, welches zwei conv. Layer mit 32 und 64 Kanälen besaß. Beide Layer hatten eine Kernen-Size von fünf und einen Stride von eins mit 2x2 Max Pooling. Der dritte Layer war ein fc Layer der Größe 1024 mit Dropout, welcher mit einem Softmax Ausgabelayer für fünf Klassen verbunden wurde. Alle Hidden Units verwendeten eine Relu-Nichtlinearität.

Die Eingabe des CNN durchlief wie das CapsNet die oben beschriebene Bildvorbereitung. Das dargestellte Baseline-CNN stellt das Ergebnis einer umfangreiche Hyperparameter-Suche über Filtergrößen, Anzahl der Kanäle und Lernraten dar. Bei selber Anwendung auf smallNORB wie das obere CapsNet (also ohne neue Viewpoints) wurde mit dem CNN, aus 4.2M Parametern bestehende, Baseline schließlich ein Test-Error von 5.2% erreicht. Das Netzwerk von Ciresan et al. [Cir+11] besitzt 2.7M Parameter. Durch die Nutzung von Verfahren zur Verringerung von Parameter, welche hier nicht genauer erläutert werden, konnte die Nummer der Parameter des CNNs auf 310 Tausend reduziert werden, was einem Faktor von 15 entspricht. Ein kleines CapsNet mit $A = 64$, $B = 8$, $C = D = 16$ und 68 Tausend tradierbaren Parametern erreicht einen Test-Error von 2.2%, was den State-of-the-Art übertrifft [HSF18a, S. 5].

Routing iterations	Pose structure	Loss	Coordinate Addition	Test error rate
1	Matrix	Spread	Yes	9.7%
2	Matrix	Spread	Yes	2.2%
3	Matrix	Spread	Yes	**1.8**%
5	Matrix	Spread	Yes	3.9%
3	Vector	Spread	Yes	2.9%
3	Matrix	Spread	No	2.6%
3	Vector	Spread	No	3.2%
3	Matrix	Margin[1]	Yes	3.2%
3	Matrix	CrossEnt	Yes	5.8%
Baseline CNN with 4.2M parameters				5.2%
CNN of Cireşan et al. (2011) with extra input images & deformations				2.56%
Our Best model (third row), with multiple crops during testing				**1.4**%

Abbildung 3.32: Der Effekt der Variation verschiedener Komponenten der CapsNet-Architektur für smallNORB aus [HSF18a, S. 5]

Abbildung 3.32 zeigt die Effekte der Anzahl an Routing-Iterationen, des Loss-Typs und des Nutzens von Posen-Matrizen anstelle von Posen-Vektoren, sowie die Ergebnisse des Modells von (Ciresan et al. [Cir+11] und dem Baseline-CNN zusammen [HSF18a, S. 6-7] und bestätigt, dass das anfangs beschriebene Modell am besten abschneidet.

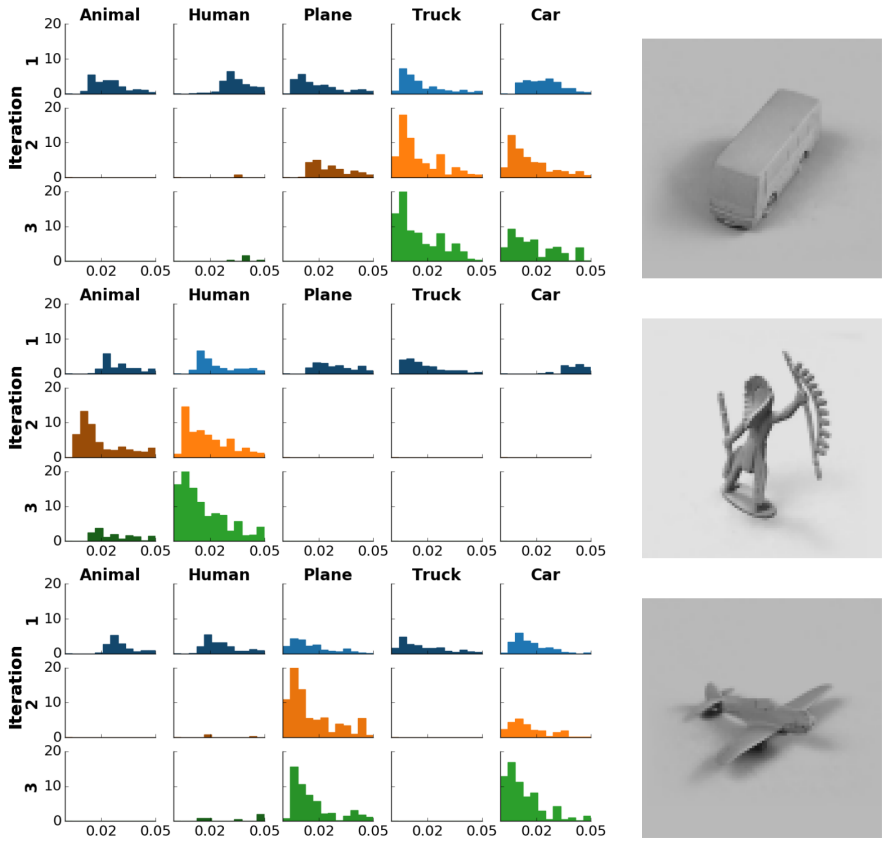


Abbildung 3.33: Histogramme von Distanzen der Votes zu den Mittelwerten jeder der fünf finalen Capsules aus [HSF18a, S. 6]

Abbildung 3.33 zeigt Histogramme von den Distanzen der Votes zu den Mittelwerten jeder der fünf finalen Capsules nach jeder Routing-Iteration. Jeder Distanzpunkt ist mit einer Zuordnungswahrscheinlichkeit gewichtet und alle drei dargestellten Bilder wurden vom smallNORB Datensatz ausgewählt. Daher ist in Abbildung 3.33 zu erkennen, wie das EM-Routing die Zuordnung der Votes und die Capsule Mittelwerte anpasst, um dichte Cluster in den Votes zu finden. Jedoch muss beachtet werden, dass die Histogramme „hineingezoomt“ sind, um nur Votes mit kleineren Abständen als 0.05 zu visualisieren. Wird nun beispielsweise den *oberen* Routing-Iterationen beim Erkennen des *Trucks* gefolgt, sind in der ersten Routing-Iteration die Votes realtiv gleichmäßig zwischen den fünf finalen Capsules verteilt. Daher erhalten alle Capsules einige Votes, welche näher als 0.05 zu ihren berechneten Mittelwert liegen. In der zweiten Iteration erhöht sich die Zuordnungswahrscheinlichkeit für übereinstimmende Votes. Daher werden die meisten Votes zu den erkannten Clustern zugewiesen, die *Tier- und Menschenklasse* in der mittleren Reihe, und die anderen Capsules erhalten nur vereinzelte Votes [HSF18a, S. 5-6]. In der dritten Iteration wird diese Zuordnung noch einmal verbessert und nun erhält auch die *Flugzeugklasse* nur noch vereinzelte Votes. Die *Truckklasse* hingegen erhält am meisten Votes und wird somit korrekt klassifiziert. Die *Autoklasse* erhält ebenfalls einige Votes, was Sinn er-

gibt da ein Truck und ein Auto viele Ähnlichkeiten untereinander besitzen.
Weiter zeigt die Abbildung, dass das EM-Routing die Votes beim Klassifizieren eines *Menschen* richtig routet und das Flugzeug-Beispiel zeigt einen seltenen Fehlerfall des Modells, bei dem das Flugzeug in der dritten Routing-Iteration mit einem Auto verwechselt wird [HSF18a, S. 6].

Die aus MDL abgeleiteten Aktivierungswahrscheinlichkeiten der Capsule berechnen eine separate Aktivierungswahrscheinlichkeit pro Capsule [HSF18a, S. 6-7], somit lernt das CapsNet streng genommen keine Mischung von Gaußverteilungen, da die Aktivierungswahrscheinlichkeit mit der Zuordnungswahrscheinlichkeit multipliziert wird und diese nicht im Layer zu eins aufsummieren. Diese Multiplikation wurde oben damit begründet, dass die transformierten Gaußverteilungen austauschbar sein sollen (was nach wie vor sinnvoll ist). Jedoch könnten anstelle der aus MDL abgeleiteten Aktivierungswahrscheinlichkeiten die Aktivierung der Capsule streng als Mischungsverhältnis einer Mischung von Gaußverteilungen gesehen und diese proportional zu der Summe der Zuordnungswahrscheinlichkeiten einer Capsule gesetzt werden, sodass die Aktivierungswahrscheinlichkeiten zu eins über alle Capsules im Layer aufsummieren. Dies erhöht allerdings den Test-Error von smallNORB auf 4.5%. Das Verfahren zum Berechnen der Aktivierungswahrscheinlichkeiten für eine strenge Mischung von Gaußverteilungen wurde nicht weiter ausgeführt, was auch nicht nötig scheint, da das Ergebnis sichtbar schlechter ausfällt.

Die gleiche CapsNet-Architektur wie in Abbildung 3.31 erreicht eine Testfehlerrate von 0.44% auf dem MNIST-Datensatz und durch die Erhöhung der Anzahl von Feature-Maps im Conv1-Layer auf 256, wurde eine Test-Error von 11.9% auf Cifar10 [HSF18a, S. 7] erreicht. Diese Werte fallen etwas schlechter aus als die der original Capsules aus Abschnitt 3.3 mit 0.25% und 10.6%. Jedoch wurde die 10.6% Test-Error auf Cifar10 von den originalen Capsules durch ein Ensemble von sieben Modellen erreicht. Hier hingegen fand nur eine Matrix-CapsNet Verwendung. Ob ein Ensemble von Matrix-CapsNets besser abschneidet als das Ensemble von original Capsules wurde zum Kenntnisstand dieser Arbeit nicht getestet und bleibt eine interessante offene Frage. Weiter muss erwähnt werden, dass das verwendete Matrix-CapsNet aus Abbildung 3.31 gerade einmal 64 Tausend Parameter besitzt, das originale CapsNet hingegen ohne Rekonstuktions-Modul 6.8M.

Generalisierung zu neuen Viewpoints

Ein schwererer Test für die Generalisierung ist das Training des CapsNets auf eine limitiere Auswahl an Viewpoints des Datensatzes und das Durchführen des Tests auf einem anderen Datensatz, der nicht trainierte Viewpoints beinhaltet. Das CapsNet und das Baseline-CNN wurden

hierfür auf einem Drittel der Trainingsdaten des smallNORB-Dateisatzes mit den Azimuten (300, 320, 340, 0, 20, 40) trainiert. Danach wurden die Modelle mit zwei Dritteln der Azimute von 60 zu 280 getestet. Weiter wurden die Netze in einem separaten Experiment auf den drei kleinere Höhen des Datensatzes trainiert und auf den sechs größeren getestet [HSF18a, S. 7].

Test set	Azimuth		Elevation	
	CNN	Capsules	CNN	Capsules
Novel viewpoints	20%	13.5%	17.8%	12.3%
Familiar viewpoints	3.7%	3.7%	4.3%	4.3%

Abbildung 3.34: Vergleich von EM-CapsNet mit Baseline-CNN aus [HSF18a, S. 7]

Jedoch traten Schwierigkeiten bei der Entscheidung auf, ob das CapsNet besser zu neuen Viewpoints generalisiert als das CNN, da das CapsNet eine bessere Testgenauigkeit bei allen Viewpoints aufweist. Um diesen verwirrenden Faktor zu eliminieren, wurde frühzeitig das Training gestoppt (Early Stopping), sobald die Performance des CapsNet mit der Performance des CNNs übereinstimmte. Folglich wurden die Modelle auf zwei Dritteln des Testdatensatzes mit den neuen Viewpoints getestet. Abbildung 3.34 zeigt die Vergleichsergebnisse und gibt zu erkennen, dass das CapsNet mit Early Stopping bei den bekannten Viewpoints in der Performance übereinstimmt. Der Test-Error für die neuen Viewpoints wird jedoch beim CapsNet für die neuen Azimute, sowie für die neuen Höhen, um etwa 30% reduziert [HSF18a, S. 7].

Robustheit gegen Adversarial Attacks

Es besteht ein zunehmendes Interesse an der Anfälligkeit neuronaler Netzwerke gegenüber Adversarial Examples - Eingaben, die von einem Angreifer geringfügig geändert wurden, um einen Klassifikator dazu zu bringen, die falsche Klassifizierung vorzunehmen. Diese Inputs können auf verschiedenen Wegen generiert werden, jedoch wurde bereits für einfache Strategien, wie FGSM (Fast Gradient Sign Method von Goodfellow et al. [GSS15]), gezeigt, dass diese drastisch die Genauigkeit von CNNs bei der Klassifizierung von Bildern verringern. Im Folgenden wird beschrieben, wie das CapsNet mit einem traditionellen CNN auf die Fähigkeit solchen Angriffen standzuhalten verglichen wurde [HSF18a, S. 7].

Kurz erläutert berechnet FGSM den Gradienten des Losses mit Respekt zu jeder Pixelintensität und ändert dann die Pixelintensität um einen festen Betrag ϵ in die Richtung, welche den Loss erhöht. So hängen die Änderungen an jedem Pixel nur von dem Vorzeichen des Gradienten ab. Dieses Prinzip kann zu einer zielgerichteten Attacke erweitert werden, indem der Input so upgedatet wird, dass die Klassifizierungsgenauigkeit einer inkorrekten Klasse maximiert wird. Da FGSM nur einen Hyperparameter (ϵ) besitzt und es mit dieser Methode leicht ist Modelle mit sehr verschiedenen Gradientenmagnituden zu vergleichen, wurden mittels FGSM Adversa-

riel Attacks für das CapsNet und das CNN generiert. Dabei wurden Adverserial Images vom Testdatensatz generiert und die beiden vollständig trainierten Modelle auf Robustheit getestet [HSF18a, S. 7] .

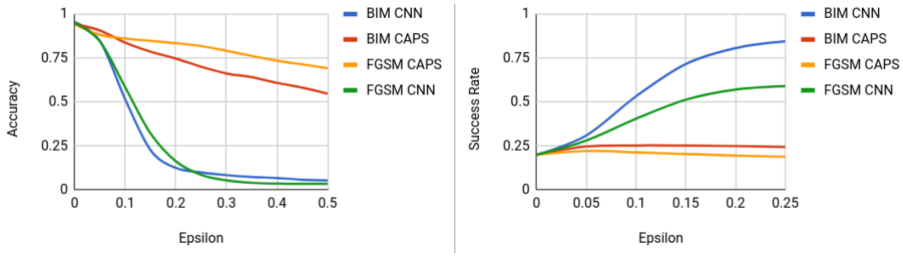


Abbildung 3.35: Links Genauigkeit gegenüber ϵ einer adversarial Attack. Rechts: Erfolgsrate nach einer adversarial Attack

Abbildung 3.35 zeigt, dass das CapsNet eine signifikant geringere Anfälligkeit für generelle und gezielte FGSM Adversarial Attacks besitzt. Bei einer gezielten Attacke versucht der Angreifer das Netz so auszutricksen, dass es anstelle der korrekten Klassen einer explizit ausgewählte Klasse klassifiziert. Bei einer generellen Attacke wird das Netz so ausgetrickst, dass es anstelle der korrekten Klassen irgendeine inkorrekte Klasse klassifiziert. Die gezielten Angriffsergebnisse wurden bewertet, indem die Erfolgsrate nach dem Angriff für jede der fünf möglichen Klassen gemittelt wurde. Die Ergebnisse zeigen, dass ein kleines ϵ dazu verwendet werden kann, die Genauigkeit des CNNs weit mehr als die des CapsNets zu verringern. Außerdem sollte erwähnt werden, dass die Genauigkeit nach keiner Attacke auf das CapsNet unter 20% fiel, während die Genauigkeit des CNNs mit einem kleinen ϵ von 0.2 signifikant unter die Wahrscheinlichkeit eines Zufallsgriffs reduziert wird [HSF18a, S. 7].
Zusätzlich wurden die Modelle mittels einer etwas anspruchsvolleren Adversarial Attack getestet: Der Basic Iterative Method (BIM) von Kurakin et al. [KGB16], welche kurz beschrieben der oben genannten Methode entspricht, nur dass mehrere kleine Schritte bei der Kreation der Adversarial Images vorgenommen werden. Hierbei ähneln die Ergebnisse stark denen des oberen FGSM-Angriffes (siehe Abbildung 3.35) [HSF18a, S. 7].
In Brendel & Bethge [NG17] wurde gezeigt, dass eine gewisse Robustheit von Modellen gegenüber Adversarial Attacks auf einfache numerische Instabilität bei der Berechnung des Gradienten zurückgeführt werden kann. Um sicherzustellen, dass dies nicht die einzige Ursache für die Robustheit des CapsNets ist, wurde der Prozentsatz der Nullwerte im Gradienten mit Bezug zum Eingabebild berechnet. Dabei wurde festgestellt, dass der Prozentsatz um eine Größenordnung von zwei kleiner ist als der des CNNs. Auch wenn die Gradienten des CapsNets kleiner sind, handelt es sich hierbei nur um eine Größenordnungen von zwei, in der Arbeit von Brendel & Bethge [NG17] fanden hingegen Größenordnungen von 16 Verwendung [HSF18a, S. 8].

Zuletzt wurde die Robustheit des Modell und des CNNs mittels Black-Box-Angriffen durch Generating Adversarial Examples, welche von einem CNN stammen, getestet. Hierbei wurde keine deutliche Verbesserung gegenüber der Performance des CNNs festgestellt.

3.4.7 Erkenntnisse durch Matrix Capsule Networks

In diesem Abschnitt wurden Matrix Capsules mit EM-Routing durchleuchtet, welche auf den Ergebnissen der Capsules von Abschnitt 3.3 aufbauten und einige Schwächen dieser ausbesserten. Matrix Capsules erhalten zusammengefasst eine Logistic Unit, welche die Präsenz der Entität darstellt und eine 4x4 Posematrix, welche die Pose der Entität repräsentiert. Das EM-Routing erlaubt den niedrigen Capsules ihren Output zu den höheren Capsules zu routen, sodass aktive Capsules ein Cluster von ähnlichen Pose-Votes erhalten und durch eine, dem EM-Algorithmus ähnelnde, aber stark erweiterte, Routing-Prozedur ihre eigene Pose ermitteln. Dabei passt das Routing das Gauß-Modell der höheren Capsules an, wodurch wiederum die unteren Capsules ihre Zuordnungswahrscheinlichkeiten aktualisieren können. Matrix Capsules erreichen somit eine bessere Genauigkeit auf dem smallNORB Datensatz, als State-of-the-Art CNNs. Weiter zeigen Matrix Capsules eine signifikant erhöhte Robustheit gegenüber White-Box Adversarial Attacks als CNNs. SmallNORB stellt hierbei einen idealen Datensatz für die inkrementelle Entwicklung von shape-recognition Modellen dar, da diesem viele Features von natürlichen Bildern fehlen. Auf größeren, natürlichen Bildern wie ImageNet wurden Matrix Capsules, sowie die Capsules aus Abschnitt 3.3 bislang jedoch nicht angewandt.
So wird beispielsweise in Vijay Badrinarayanan et al. [RL18] argumentiert, dass die originalen CapsNets bei größeren Eingaben eine Parameterexposition erleben, was das Training mit großen Eingaben ohne signifikante Änderungen unmöglich macht. So wird dort weiter ausgeführt, dass die originale CapsNet-Architektur und der dynamische Routing-Algorithmus extrem rechenintensiv sind, sowohl hinsichtlich des Speichers als auch der Laufzeit. Zusätzliche Zwischenrepräsentationen werden benötigt, um die Ausgabe von Kind-Capsules in einem gegebenen Layer zu speichern, während der dynamische Routing-Algorithmus die Koeffizienten bestimmt, mit denen diese Kinder zu den Eltern-Capsules im nächsten Layer geleitet werden. Dieses dynamische Routing findet zwischen jedem Elternteil und jedem möglichen Kind statt. So entsteht zusätzlicher Speicher, der sich multiplikativ mit der Batch-Size für einen gegebenen Layer durch seine Anzahl von Capsule-Typen erhöht. Die Anzahl der erforderlichen Parameter steigt bereits bei trivialen, kleinen Eingaben wie MNIST und Cifar10 schnell über die Kontrolle hinweg an. Wenn beispielsweise ein Satz von 32 Capsule-Typen mit 6x6, 8D-Kapseln pro Typ zu zehn 16D-Kapseln geroutet wird, beträgt die Anzahl der Parameter für diesen Layer allein $10 \times (6 \times 6 \times 32) \times 16 \times 8 = 1\,474\,560$ Parameter [BKC15]. Diese Parameterexplosion betrifft vor allem die Verbindung zwischen einem conv. Capsule-Layer (bzw. PrimaryCaps)

und einem fc ClassCaps-Layer und ist bei Verbindungen zwischen mehreren conv. Capsule-Layern durch Filtergrößen weniger ausschlaggebend, jedoch nicht zu vernachlässigen. Auch wenn zum Kenntnisstand dieser Arbeit kein Paper dieses Problem an Matrix Capsules untersucht hat, wird hier argumentiert, dass Matrix Capsules ebenfalls unter einem Performanceproblem leiden, schlichtweg aus dem Grund, dass Hinton et al. [HSF18a] planen eine effizientere Version des Matrix-CapsNets zu implementieren, um dieses auf ImageNet anwenden zu können. Dieses Performance-Problem scheint bei Matrix Capsules jedoch auf das Routing an sich zurück zu gehen und nicht auf eine Parameterexplosion zwischen ConvCaps- und ClassCaps-Layer, wie es bei den original Capsules der Fall ist. Der Grund hierfür ist, dass die Coordinate Addition der Matrix Capsules bei der Verbindung zwischen dem conv. Capsule-Layer und dem ClassCaps für eine geringe Anzahl an Transformationsmatrizen sorgt. Auch für die original Capsule existiert in Vijay Badrinarayanan et al. [RL18] bereits ein effizienter, vollständig implementierter Lösungsvorschlag der Parameterexplosion, welcher sehr gute Segmentierungsergebnisse für Lungenflügel liefert, jedoch auf die Segmentierungsaufgabe beschränkt ist, da dort das Modell kein fc Capsule-Layer enthält. Die Literatur weist außerdem noch weitere Effizienzverbesserungen der CapsNets auf die u.a. den Rechenaufwand zwischen conv. Capsule-Layern verringern. Diese und weitere, auf den Ergebnissen von Abschnitt 3.2, 3.3 und 3.4, basierende Verfahren, Modelle und Erweiterungen werden in den folgenden Abschnitten jeweils zusammengefasst vorgestellt. So werden dort auch Architekturen erläutert, welche Matrix Capsules verwenden oder erweitern. Die kommenden Abschnitte zeigen somit die aktuelle Entwicklung in diesem Bereich und stellen verschiedene Vorgehensweisen bei der Erforschung von CapsNets dar, wobei viele bislang ungeklärte Fragen aufkommen, die weitere mögliche praktische Aufgaben für diese Arbeit bilden. Die verschiedenen Paper werden hierfür in einzelne Kategorien eingeteilt, mit dem Wissen, dass diese meist mehreren Kategorien angehören. Außerdem werden viele Verfahren (z.B. die im Preprocessing) nur kurz erläutert, da sonst der Rahmen der Arbeit gesprengt werden würde und der Fokus weiter auf den CapsNets liegen soll.

3.5 Weitere Capsule-Modelle

Dieser Abschnitt beschreibt Modelle, welche aus den beiden vorgestellten CapsNets abgeleitet wurden und oft einige Verbesserungen mit sich bringen. Ebenfalls wurde ein Modell weitgehend unabhängig vom Original entwickelt (Abschnitt 3.5.3).

3.5.1 Spectral Capsule Networks

Für eine akkurate Prognose müssen Modelle im Gesundheitswesen nicht nur Risikofaktoren identifizieren, sondern auch die komplexen und hierarchisch-zeitlichen Interaktionen zwischen

Symptomen, Zuständen und Medikamenten erkennen. Auf einem Datensatz der diese Probleme widerspiegelt, wurden zwei Capsule-Modelle angewandt: Capsules mit EM-Routing aus Abschnitt 3.4 und Spectral-Capsules kurz S-Capsules. Im Kontrast zu EM-Capsules messen S-Capsules das Agreement als den Grad der Ausrichtung der Votes der unteren Capsules in einem eindimensionalen linearen Unterraum anstelle in zentralisierten Cluster. Deshalb ist in S-Capsules die Pose (Variationskomponente) der Normalenvektor eines linearen Unterraums, welcher den größten Teil der Varianz in den Votes der unteren Capsules erhält. Die Aktivierungswahrscheinlichkeit (Aktivierungskomponente) wird basierend auf dem Verhältnis der erhaltenen Varianz berechnet. Beide Modelle besitzen daher zwei Komponenten: Die Aktivierung $a \in [0, 1]$ und den Pose-Vektor $\mathbf{u} \in \mathbb{R}^d$ [Bah18]. Die Wahl eines Pose-Vektors anstelle einer Matrix ist auf die Zeitreihennatur der Merkmale zurückzuführen [Bah18], welche im Gegenzug zu einem 1D-Bild nicht statisch sind (vgl. den vektorisierten Input eines Long-Short-Term-Memory Netzes, kurz LSTMs). Abbildung 3.36 zeigt die Architektur des S-Modells und wird nun Schritt für Schritt erläutert. Für das EM-Modell ist nur Schritt 3 unterschiedlich [Bah18].

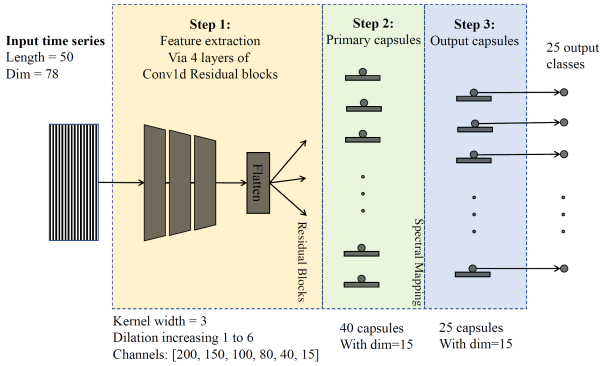


Abbildung 3.36: Details der S-Capsule Architektur aus [Bah18]

Step 1: Extrahieren der Features: Zuerst wird eine eindimensionale Convolution verwendet, um niedrigdimensionale Merkmale für die Verarbeitung durch Capsules zu extrahieren. Hierbei finden Residual-Blocks mit zunehmender Größe Einsatz. Diese sind von Van Den Oord et al. [Oor+16] inspiriert und werden hier nicht weiter erläutert. Wichtig ist, dass Residual-Blocks nicht nur das rezeptive Feld der Convolutionen erhöhen, sondern auch die Dimension des Inputs reduzieren. Die Ausgabe des Residual-Networks wird geglättet, um einen 120-dimensionalen Vektor für den Capsule-Layer zu erhalten [Bah18].

Step 2: Primary-Capsules: In beiden Architekturen werden dichte Residual-Networks pro Capsule verwendet, um die Aktivierungs- und Pose-Komponenten der Primary-Capsules zu erzeugen. Die Wahl von Residual-Blocks als Transformationsoperation anstelle der linearen Abbildung (die z.B. durch ein einfaches Gewicht zustande kommt) rührt daher, dass im Ge-

sundheitswesen keine formalen Kenntnisse über die Art der Deformationen in den Daten existiert, während in der Computervision Verzerrungen wie Rotation ausführlich untersucht sind. Residual-Bocks ermöglichen einfache nichtlineare Transformationen ohne Überparametrisierung des Netzwerks, um diesem entgegenzusetzen [Bah18] und somit zusätzlich nichtlineare Abbildungen zu ermöglichen.

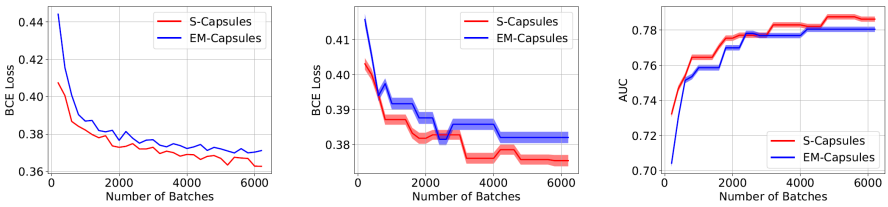
Step 3: Capsule zu Capsule Verfahren: Das EM-CapsNet verwendet das in Abschnitt 3.4.3 beschriebene EM-Routing, allerdings werden die Transformationen (oben conv. Layer oder einzelne Gewichte) durch Residual-Blocks realisiert. Um die Berechnung der S-Capsule zu beschreiben werden zwei Layer L und $L+1$ mit n_L beziehungsweise n_{L+1} Capsules herangezogen. Die j-te Capsule in Layer $L+1$ berechnet die gewichteten Votes von Layer L mit $\mathbf{y}_{j,i} = a_i R_{i,j}(\mathbf{u}_i)$ für jede Capsules i in Layer L, wobei $R_{j,i}(\cdot)$ ein dichter Residual-Block ist. Die berechneten Votes von Layer L werden dann zur Matrix $\mathbf{Y}_j \in \mathbb{R}^{n_L \times d}$ konkateniert und die Eigenwertzerlegung von $\mathbf{Y}_j^T \mathbf{Y}_j = \mathbf{V}\mathbf{\Lambda}\mathbf{Y}^T$ berechnet [Bah18]. Die Eigenwertzerlegung wird hier nicht genauer beschrieben, jedoch werden wie herkömmlich in der Matrix $\mathbf{V}$ die Eigenvektoren und in $\mathbf{\Lambda}$ die Eigenwerte gespeichert. Durch die spezielle Struktur der Matrizen (positive semi-definite) kann anstelle der herkömmlichen Invertierung transponiert werden. Das herkömmliche „A“ einer Eigenwertzerlegung wird hier als $\mathbf{Y}_j^T \mathbf{Y}_j$ geschrieben, da eine Matrix A positive semi-definite ist wenn diese als $A = \mathbf{Y}_j^T \mathbf{Y}_j$ definiert werden kann. Eigenschaften einer positive semi-definite Matrix sind unter anderem Symmetrie, die garantierte Existenz einer Eigenwertzerlegung, sowie Eigenwerte die immer positiv oder null und bei unterschiedlichen Eigenwerten paarweise orthogonal sind. Die Menge an Eigenwerten einer Matrix wird als Spektrum bezeichnet, daher der Name Spectral-Capsules [Abd].
Der Pose-Vektor ist dann der erste (dominante, d.h. maximaler Betrag) Eigenvektor $\mathbf{u}_j = \mathbf{V}[0, :]$, welcher der Normalenvektor des linearen Unterraums darstellt und die meiste Varianz in den Votes der Capsules in Layer L bewahrt. Die Aktivierung der j-ten Capsule in Layer $L+1$ wird folglich mittels den Eigenwerten λ_k des Eigenvektors berechnet [Bah18]:

$$a_j = sigmoid\left(\eta\left[\frac{\lambda_1}{\sum_{k=1}^{min(d,n_L)} \lambda_k} - b\right]\right) = sigmoid\left(\eta\left[\frac{\|\mathbf{Y}_j \mathbf{u}_j\|)_2^2}{\|\mathbf{Y}_j\|_F^2} - b\right]\right) \tag{3.25}$$

Wobei b durch Backpropergation trainiert wird und η während des Trainings einem Linear-Annealing unterliegt [Bah18]. F ist die Frobeniusnorm (auf der euklidischen Norm basierende Matrixnorm) von $\mathbf{Y}_j$. Das Verhältnis $\frac{\lambda_1}{\sum_{k=1}^{min(d,n_L)} \lambda_k}$ ist der Bruchteil der Varianz der Votes aus dem unteren Layer, die in dem von $\mathbf{u}_j$ definierten eindimensionalen Unterraum erfasst wurde, und misst die Übereinstimmung zwischen den Votes für die Pose der j-ten Capsule. Dieser Schritt benötigt nur eine symmetrische top-1 Eigenwertzerlegung, welche weit effizienter als die volle Zerlegung ist. Außerdem werden die Aktivierungen und Posen von Natur aus normalisiert,

da $||\mathbf{u}_j||_2 = 1$ und $\frac{\lambda_1}{\sum_{k=1}^{min(d,n_L)} \lambda_k} \in \left[\frac{1}{min(d,n_L)}, 1\right]$ immer gilt. Liegt eine stabile Eigenwertzerlegung vor, stabilisieren folglich die normalisierten Ausgaben das Training [Bah18]. Die Intuition hinter der Gleichung wird von den Autoren nicht erläutert und wird hier aus Platzgründen auch nicht versucht zu erahnen.

Das Modell wurde mit einem binären Cross-Entropy-Loss, Adam und eine Batch-Size von 64 trainiert. Die Lernrate wurde halbiert sobald die Validierungsgenauigkeit in ein Plateau geriet. Alle anderen Hyperparameter wurden mit dem Validierungset eingestellt, doch nicht explizit angegeben. Auch der Spread-Loss aus Abschnitt 3.4.5 wurde für das im Folgenden beschriebene Multi-Label-Setting erweitert, aber schnitt in diesen Experimenten messbar schlechter ab, als der binäre Cross-Entropy-Loss. Weiter wurde festgestellt, dass das Hinzufügen einer Skip-Verbindung von den in Schritt 1 extrahierten Features zu den Aktivierungen der letzten Capsules Layer die Leistung verbessert. Diese Experimente wurden anhand des „Lernen zu diagnostizieren“ Benchmark (Harutyunyan et al. [Har+17]) durchgeführt, welcher 78-dimensionale Zeitreihen für jeden Patienten aus dem MIMIC-III-Datensatz extrahiert. Dafür wurde dem im Benchmark angegebenen Preprocessing gefolgt, welches hier nicht weiter erläutert wird [Bah18].



(a) Training loss (b) Test loss (c) Test AUC

Abbildung 3.37: Trainingskurven von S- und EM-Capsules aus [Bah18]

Abbildung 3.37 zeigt das Konvergenzverhalten von S- und EM-Capsules im Laufe des Trainings. Aus Gründen der Fairness wurde die Lernrate für beide Algorithmen gleichgesetzt, obwohl S-Capsules mit höheren Raten lernen könnten. Die Ergebnisse bestätigen, dass S-Capsules im Vergleich zu EM-Capsules auf diesem Datensatz schneller lernen und generalisieren und mit 80.50% die Genauigkeit der Baselines übertreffen. Weiter wurde beobachtet das EM-Capsules sensitiver bei der Hyperparameterwahl (z.B. die Lernrate) sind als S-Capsules [Bah18].

Außerdem wurden hand-konstruierte medizinische Merkmale für die Eingabe erstellt und die Korrelation zwischen jeder Dimension des Pose-Vektors und den Merkmalen gemessen, um zu analysieren, ob die Pose-Vektoren der Ausgabe-Capsules die Variationen in den Eingabedaten beibehalten. Der Grund dafür liegt darin, dass das Verstehen der Muster in medizinischen Daten Fachwissen erfordert und diese nicht wie in oberen Abschnitten durch visuelle Rekon-

struktionen interpretiert werden können. 47.40% der Pose-Vektor-Elemente waren hierbei signifikant mit den Hand-Engineering-Merkmalen korreliert. Diese zeigt, dass die Pose-Vektoren eine große Menge an Variationen in den Eingabedaten beibehalten. Weiter darf die Prozentzahl nicht zu groß sein, da hand-konstruierte Merkmale i.d.R. nicht die perfekte Zusammenfassung der Eingabedaten darstellen [Bah18].

3.5.2 CapsuleGAN

Ein GAN (Generative Adversarial Network) besteht typischerweise aus zwei neuronalen Netzen; (1) ein Generator, der versucht, aus einer früheren Verteilung gezogene Abtastwerte in Abtastwerte aus einer komplexen Datenverteilung mit viel höherer Dimensionalität umzuwandeln, und (2) ein Diskriminator, der entscheidet, ob der gegebene Abtastwert reell ist oder aus der Generaltorverteilung stammt. Die beiden Komponenten werden trainiert, indem ein Adversarial-Game gespielt wird. Beispielsweise generiert der Generator Bilder von Menschen und der Diskriminator erhält entweder ein vom Generator generiertes oder ein echtes Bild und muss entscheiden ob es sich hierbei um eine echtes oder generiertes Bild eines Menschen handelt. Der Loss ist so konstruiert, dass der Generator lernt Bilder zu erzeugen, die echt aussehen [AJ18].
GANs haben sich bei der Modellierung hochkomplexer Verteilungen, welchen reale Daten (insbesondere Bilder) zugrunde liegen, als äußerst vielversprechend erwiesen. Jedoch sind sie für ihre schwierige Trainierbarkeit berüchtigt, so verzeichnen diese beispielsweise Probleme mit der Stabilität und mit Vanishing Gradients. In Ayush Jaiswal et al. [AJ18] wurde ein Generative Adversarial Capsule Network (CapsuleGAN) vorgestellt, welches im Gegensatz zu den herkömmlich eingesetzten CNNs ein CapsNet als Diskriminator verwendet. Allerdings ist hier notwendig die Anzahl der Parameter im CapsuleGAN-Diskriminator aus drei Gründen niedrig zu halten: Erstens sind CapsNets sehr mächtige Modelle und können den Generator bereits frühzeitig im Trainingsprozess bestrafen, da diese zu schnell lernen generierte Bilder zu erkennen. Dies führt beispielsweise dazu, dass der Generator komplett ausfällt. Zweites sind derzeitige Implementierungen des dynamischen Routing-Algorithmus langsam. Drittens sind conv. Diskrimatoren meist relativ flach designed und besitzen eine geringen Anzahl von relativ großen Filtern in ihren conv. Layern [AJ18].
Die Architektur des Capsule-Diskriminators ist nahezu analog zu der des original CapsNets aus Abschnitt 3.3. Der letzte Layer enthält jedoch eine einzelne Capsule, deren Länge die Wahrscheinlichkeit darstellt, ob die Eingabe ein reelles oder ein erzeugtes Bild ist [AJ18]. Die Anzahl der Routing-Iterationen oder das Nutzen eines Rekonstruktion-Moduls wird nicht explizit erwähnt, wobei die Routing-Iterationen womöglich beim Original liegen, da die Ähnlichkeit zu diesem extra erwähnt wurde. Da kein Rekonstruktion-Loss in der GAN-Zielfunktion miteinbe-

zogen wurde, wäre es sinnlos ein Rekonstruktion-Modul zu verwenden.
Das CapsuleGAN wurde mittels MNIST und Cifar10 getestet und mit einem conv. GAN sowohl qualitativ durch visuellen Vergleich der erstellten Bilder, als auch quantitativ durch Metriken verglichen [AJ18]. Auf die Metriken wird nicht weiter eingegangen, um nicht zu weit vom Thema der Arbeit abzuschweifen. Beide GAN-Modelle besaßen dabei die gleiche Architektur für ihren Generator. Für das Training des Capsule-Modells wurde der Margin-Loss aus Abschnitt 3.3.4 anstelle des für GANs üblichen Binary-Cross-Entropy-Losses in der GAN-Zielfunktion verwendet [AJ18].
Die erhaltenen Ergebnisse zeigen, dass das CapsNet besser abschneidet als das CNN und deuten darauf hin, dass CapsNets als mögliche Alternativen für Diskriminatoren in GANs und für andere schlussfolgernde Modelle verwendet werden könnten. Beim Training wurden bewusst wenige GAN-Trainingstricks verwendet, um die Ergebnisse angemessen bewerten zu können [AJ18].
Im weiten Gebiet der GANs wären weitere Experimente hinsichtlich CapsNets interessant. So könnte beispielsweise ein GAN getestet werden, welches ein CapsNet mit einem, dem Rekonstuktions-Modul ähnelnden, finalen Bilderzeuger als Generator besitzt. Darauf aufbauend könnte ein weiteres GAN entwickelt werden, welches sowohl einen Caps-Diksriminator als auch Caps-Generator besitzt.

Mathijs Pieters et al. [PW18] experimentierten ebenfalls mit einem CapsNet als Diskriminator auf MNIST und zusätzlich auf CelebA (Gesichter-Datensatz). Das Capsule-Modell ähnelt hierbei ebenfalls dem Original, besitzt jedoch im ersten Layer 128 Filter, anstelle der 256 und im finalen Layer eine einzige 16D-Capsule zur binären Klassifikation. Es wurden drei Routing-Iterationen verwendet und kein Rekonstruktion-Loss [PW18].

Da hier nun die drei Routing-Iterationen explizit erwähnt wurden, muss geklärt werden, dass dies bei einer einzelnen Capsule im höheren Layer nicht sinnvoll ist, solange keine Techniken wie Leaky-Routing oder eine Orphan-Kategorie Einsatz finden. Dies rührt daher, dass im Kern des Routings alle Capsules in unteren Layer ihre Coupling-Koeffizienten zu den oberen Capsules so einstellen, dass diese auf eins aufsummieren, sodass der Coupling-Koeffizient zu der am besten passenden oberen Capsule am größten ist. Diese Einstellung wird im Laufe der Routing-Iterationen verfeinert. Bei einer einzelnen Capsule im höheren Layer besitzt jedoch jede untere Capsule nur einen Coupling-Koeffizienten zu der einzigen oberen Capsule, der somit (und wegen Gleichung 3.6) immer eins ist, egal wieviel Routing-Iterationen verwendet werden. Experimente in Abschnitt 4.3 des praktischen Teils der Arbeit untermauern und konkretisieren dies anhand der originalen Implementierung. Bei einer einzigen oberen Capsule verlieren daher

die Coupling-Koeffizienten und somit das Routing seine Wirkung, wenn keine Maßnahmen die dem entgegenwirken getroffen werden. Nur noch der Feedforward-Pfad der Capsule und die Netzstruktur kann für Verbesserungen gegenüber anderen Modellen verantwortlich sein.

Abbildung 3.38: CelebA-Vergleich des CapsuleGAN mit einem herkömmlichen GAN aus [PW18]. Links: Ergebnisse des CapsuleGANs. Rechts: Ergebnisse des GANs

Die Experimente von Mathijs Pieters et al. [PW18] führten, wie Abbildung 3.38 zeigt, zu Bildern von geringerer Qualität im Vergleich zu einem herkömmlichen GAN. Als möglicher Grund wird angegeben, dass das CapsNet für solch eine Aufgabe schlichtweg zu klein und flach sei und das daher ein größeres vonnöten wäre [PW18]. Im obigen Artikel wurde jedoch argumentiert, dass flache Netze als Diskriminatoren oft Verwendung finden. Womöglich sind für komplexere Datensätze wie CelebA etwas größere CapsNets notwendig, da diese hinsichtlich CNNs immer noch als sehr flach gelten. Jedoch fallen hier auch die erzeugten MNIST-Bilder des CapsuleGANs sichtlich schlechter aus, als die des Baseline-Netzes und als die des oberen CapsuleGANs. Dies veranschaulicht Abbildung 3.39:

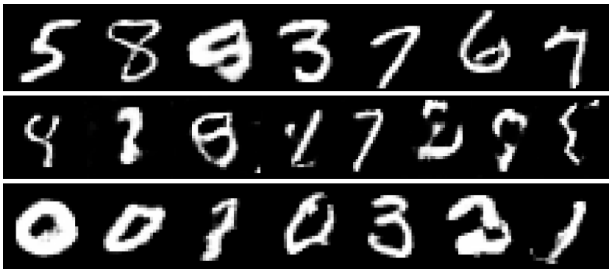


Abbildung 3.39: MNIST-Vergleich der Ergebnisse der CapsuleGANs mit einem herkömmlichen GAN. Oben: Das CapsuleGAN aus [AJ18] Mitte: Das CapsuleGAN aus [PW18]. Unten: Das GAN aus [AJ18] sehr ähnlich zu [PW18]

Die beiden Paper werfen die Frage auf, weshalb diese unterschiedlichen Ergebnisse zustande kamen. Dafür existieren zahlreichen Gründe: Womöglich sorgt der Wechsel von 256 Filter auf 128 Filter des zweiten Modells bereits für eine signifikante Verschlechterung; vielleicht liegt es auch an der bekannten Instabilität des Trainingsprozesses von GANs. Eins der beiden Modelle könnte auch Techniken wie Leaky-Routing oder eine Orphan-Kategorien unerwähnt verwendet haben, da diese als Standard angesehen wurden. Dies ist bei einer einzigen finalen Capsule aus diskutierten Gründen sinnvoll. Auch die Hyperparameter in Training der GANs könnten für die

unterschiedlichen Ergebnisse eine Rolle spielen, doch um dies beurteilen zu können müsste eine sorgfältige Experimentenreihe durchgeführt werden. Da Ayush Jaiswal etl. al [AJ18] jedoch gute Ergebnisse auf MNIST und Cifar10 zeigte scheint es so, als wären CapsNets dazu fähig als Diskrimiator zu dienen.

Im Laufe der Arbeit wurden weitere Paper veröffentlicht, die original Capsules als Diskriminator verwenden. Raeid Saqur et al. [RS18] zeigt ebenfalls vielversprechende Ergebnisse beim Generieren von Bilder mit starken geometrischen Transformationen. Dabei werden u.a. ein rotiertes MNIST und SmallNORB als Datensatz verwendet und ein der original Architektur ähnelndes CapsNet, welches eine einzelne 16D-Capsule im letzten Layer besitzt. Wird der Code genauer betrachtet ist zu erkennen, dass drei Routing-Iterationen verwendet wurden. Ergebnisse unter dem Nutzen weniger Iterationen fanden keine Erwähnung, sollten jedoch aus oben diskutierten Gründen gleich ausfallen. In [YU18] stellen Yash Upadhyay et al. ebenfalls ein Caps-GAN vor, dessen Ergebnisse auf MNIST und Fashion-MNIST zeigen, dass ein CapsNet bessere und diversere Ergebnisse liefert und signifikant weniger Trainingsepochen benötigt, als andere CNN-GAN-Modelle. Dabei wurde ein finaler Capsule-Layer verwendet, welcher anstelle einer Capsule zwei nutzt (was die drei Routing-Iterationen rechtfertigt). Dabei repräsentiert eine Capsule die Unecht-Wahrscheinlichkeit und die andere die Echt-Wahrscheinlichkeit. Die Capsule mit dem länger Aktivitätsvektor gilt als Klassifizierung. Die Anzahl der Routing-Iterationen wurde allerdings nicht explizit angegeben, da jedoch dem originalen CapsNet aus Abschnitt 3.3 gefolgt wird und die Änderungen darauf basierend angegeben wurden, liegt es nahe, dass das Training auch hier drei Routing-Iterationen beinhaltete.

3.5.3 RNN-Capsule Networks

An den folgend erläuterten Capsules wurde vor der Publikation des original CapsNets gearbeitet. Daher zeigt dieses Modell, wie die Grundidee hinter Capsules unabhängig vom Original angewandt werden kann. Das Netzwerk wurde für die Sentimentanalyse von Texten konzipiert und basiert auf RNNs (Recurrent Neural Network). Für ein gegebenes Problem repräsentiert hierbei jede Capsule eine Sentimentkategorie, z.B. eine „positive“ oder „negative“ Bewertung [Wan+18b].

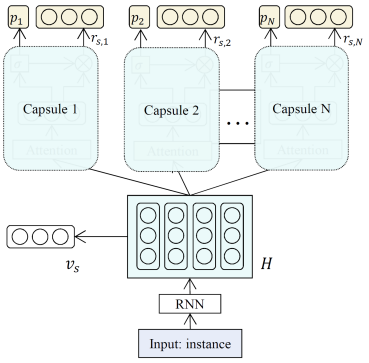


Abbildung 3.40: Architektur des RNN-Capsule Networks aus [Wan+18b]

Die Architektur der Modells wird in Abbildung 3.40 veranschaulicht: Die Nummer der Capsules N entspricht der Anzahl der Sentimentkategorien, wobei jeder Capsule mit einer Kategorie korrespondiert, weshalb eine Kategorie als ein Attribut der Capsule bezeichnet werden kann. Jede Capsule erhält dieselbe Repräsentation einer Instanz (H) als Input, welche durch ein *RNN* berechnet wird. Das *RNN* kann durch ein LSTM, eine GRU (Gated Recurrent Unit) oder deren Varianten (z.B. bidirektionales LSTM) realisiert werden. Wird dem *RNN* eine Instanz (*Input: instance*) überreicht, die beispielsweise ein Satz oder Paragraph durch einen Vektor repräsentiert, codiert das *RNN* diese Instanz und leitet den repräsentativen Hidden-Vector (H) an jede *Capsule* weiter. Die Instanz-Repräsentation des *RNN*s ist hierbei der Mittelwert aller Hidden-Vektoren. Die oberste Reihe der Abbildung 3.40 gibt weiter zu erkennen, dass darauf jede Capsule eine State-Wahrscheinlichkeit (p) durch ihr Wahrscheinlichkeits-Modul, und eine rekonstruierte Repräsentation (r) durch ihr Rekonstruktions-Modul ausgibt. Analog zu den original Capsules, wird die Capsule mit der höchsten State-Wahrscheinlichkeit aktiv, alle andere inaktiv. Ebenfalls analog, soll während des Trainings für eine Instanz die State-Wahrscheinlichkeit einer Capsule maximiert und für die restlichen minimiert werden. Durch das Label wird eine Capsule für die Rekonstruktion ausgewählt. Der Abstand zwischen der rekonstruierten Repräsentation und der Instanz-Repräsentation des *RNN*s wird minimiert und die Abstände für andere Capsules maximiert [Wan+18b].

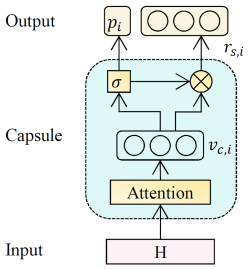


Abbildung 3.41: Aufbau einer einzelnen RNN-Capsule aus [Wan+18b]

Abbildung 3.41 zeigt eine einzelne Capsule in höherem Detail: Jede Capsule besteht aus den zwei oben genannten Rekonstruktions-Modulen ($r_{s,i}$) und Wahrscheinlichkeits-Modulen, der State-Wahrscheinlichkeit (p_i), dem Attribut, sowie einem Repräsentations-Modul ($v_{c,i}$). Die einzelnen für den Deep Learning Bereich typischen Gleichungen der Module werden nicht ausformuliert, die Berechnungen jedoch kurz textuell erläutert. Das Repräsentations-Modul nutzt einen Attention-Mechanismus um den Capsule-Repräsentations-Vektor $v_{c,i}$ zu erstellen. Der vom Attention-Layer erhaltene Vektor ist daher eine Codierung des gesamten Eingabetextes auf hohem Niveau. Der Capsule-Repräsentations-Vektor wird verwendet, um die Instanz-Repräsentation des *RNN*s zu rekonstruieren. Hierbei sorgt der Attention-Mechanismus laut den Autoren für mehr Robustheit. Das Rekonstruktion-Modul erstellt die Repräsentation $r_{s,i}$ der Eingabe-Instanz durch einfaches Multiplizieren von p_i mit $v_{c,i}$ [Wan+18b]. Das Wahrscheinlichkeits-Modul verwendet die Sigmoid-Funktion (σ), um die State-Wahrscheinlichkeit p_i der Capsule zu errechnen, wobei während des Tests die Capsule mit höchster State-Wahrscheinlichkeit aktiv ist. [Wan+18b].

Auf mehreren Datensätzen wurde letztlich gezeigt, dass die Leistung dieses Modells den State-of-the-Art bei der Sentiment-Klassifizierung erreicht. Noch wichtiger ist, dass RNN-Capsules ohne Sprachkenntnisse in der Lage sind Wörter mit Sentiment-Tendenzen auszugeben, welche die Attribute der Capsules widerspiegeln. Außerdem veranschaulichen diese Wörter gut die Domänenspezifität des Datensatzes, so ist das Modell beispielsweise in der Lage, „professionell“, „schnell“ und „fürsorglich“ als starke positive Worte in der Patientenrückmeldung an Krankenhäuser zu identifizieren [Wan+18b].

3.5.4 SegCaps

In der Objekt-Segmentierung muss, neben dem Erkennen eines Objektes, dieses zusätzlich auf Pixellevel gelabelt werden [RL18]. Dieser Abschnitt behandelt das erste - zum Kenntnisstand dieser Arbeit bislang einzige - CapsNet in der Literatur zur Bewältigung dieser Aufgabe. Dabei wurde eine Lösung für das bereits erläuterte Problem des hohen Rechenaufwands und der hohen Speicherkosten von original CapsNets gefunden. Das Parameterexplosions-Problem wurde gelöst in dem die Idee der Primary-Capsule und das dynamische Routing aus Abschnitt 3.3.2 erweitert wurde: Die Ausgaben niedriger Capsules wurden nur innerhalb eines definierten räumlich-lokalen Kernels an höher liegende Capsules weitergeleitet und die Transformationsmatrizen nur zwischen Capsules innerhalb des Gitters eines Capsule-Typen (entspricht den Channels beim Orginal) geteilt, sprich nicht über mehrere Capsule-Typen hinweg. Um den Verlust an globaler Konnektivität durch das lokal beschränkte Routing zu kompensieren, wurde das CapsNet um deconvolutional Capsules erweitert, die transponierte Convolutions verwenden und

ebenfalls durch das lokal beschränkte Routing gerouted werden. Für die Rekonstruktion des segmentierten Bildes wurde die original Methode hinsichtlich der Segmentierung erweitert. Diese Änderungen erlauben es nach den Autoren weiterhin eine diverse Menge an Capsule-Typen zu lernen und globale Kontextinformation nahezu vollständig beizubehalten. Dieser Modifikationen führe schließlich dazu, dass das neue Modell auf große (z.B. 512x512) Eingaben angewandt werden kann [RL18].

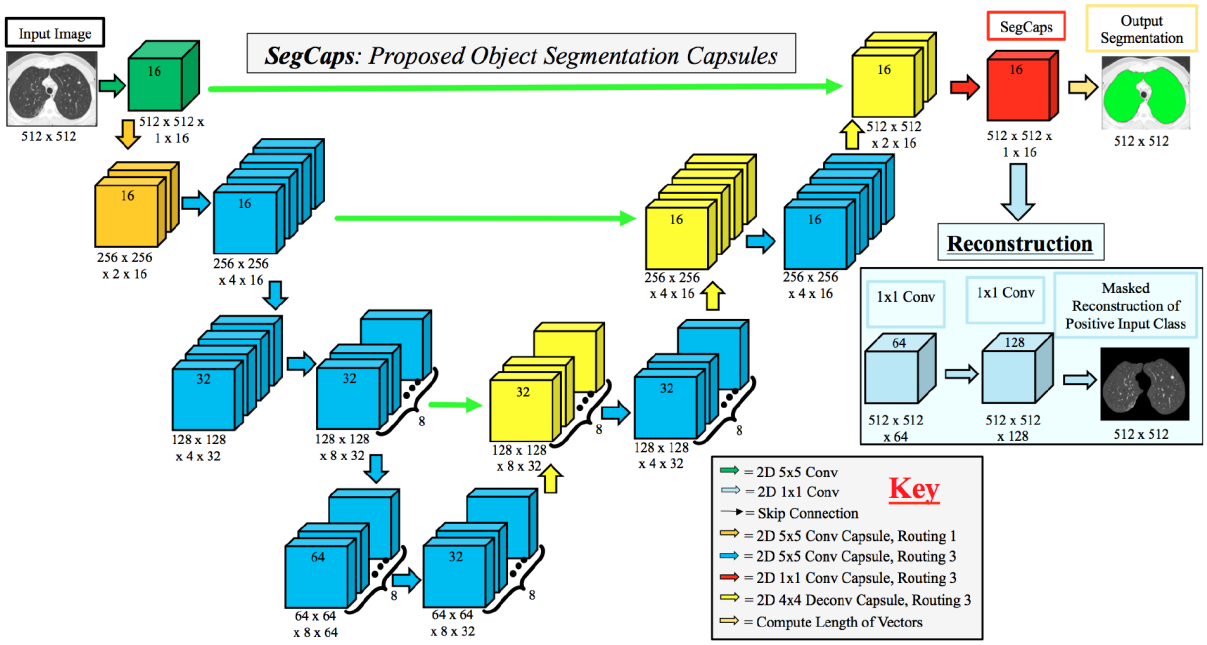


Abbildung 3.42: SegCaps Architektur für die Objekt Segmentierung aus [RL18]

Wie Abbildung 3.42 zeigt, ist der Input in das Netzwerk ein 512x512 Bild, welches einem 2D conv. Layer mit 5x5 Filtern überreicht wird, welcher 16 Feature-Maps erzeugt (*grün*). Die Ausgabe des conv. Layers wird in Caps-Layer-Form gebracht und beinhaltet folglich nur einen Capsule-Typen mit einem Gitter von 512x512, wobei die Capsule einen 16D Vektor (wegen den 16 Feature-Maps) erzeugt. Diese Ausgabe wird dem ersten neuen conv. Caps-Layer (*orange*) und einem deconv. Capsule-Layer (*gelb*) weitergegeben. Diese besitzen jeweils zwei Capsule-Typen und weiteren Werte, welche in Abbildung 3.42 ablesbar sind. Der Prozess kann wie folgt für jeden Layer l im Netz generalisiert [RL18] werden: In Layer l existiert eine Menge von Capsule-Typen $T^l = \{t_1^l, t_2^l, ..., t_n^l | n \in \mathbb{N}\}$, wobei für jedes $t_i^l \in T_l$ ein $h^l \times w^l$ Gitter von z^l-dimensionalen niedrigeren Kind-Capsules $C = \{\mathbf{c}_{11}, ..., \mathbf{c}_{1w^l}, ..., \mathbf{c}_{h^l 1}, ..., \mathbf{c}_{h^l w^l}\}$ existieren und $h^l \times w^l$ die räumlichen Dimension der Ausgabe von Layer $l-1$ ist. Im nächsten Layer $l+1$ des Netzes existiert eine Menge von Capsule-Typen $T^{l+1} = \{t_1^{l+1}, t_2^{l+1}, ..., t_n^{l+1} | m \in \mathbb{N}\}$ und für jedes $t_j^{l+1} \in T^{l+1}$ existiert ein $h^{l+1} \times w^{l+1}$ Gitter von z^{l+1}-dimensionalen nächst höheren

Eltern-Capsules $P = \{\mathbf{p}_{11}, ..., \mathbf{p}_{1w^{l+1}}, .., \mathbf{p}_{h^{l+1}1}, ..., \mathbf{p}_{h^{l+1}w^{l+1}}\}$, wobei $h^{l+1} \times w^{l+1}$ die Ausgabe des Layers l ist [RL18].

Bei conv. Capsules empfängt dann jede Eltern-Capsule $\mathbf{p}_{xy} \in P$ eine Menge von Votes, die als $\left\{\hat{\mathbf{u}}_{xy|t_1^l}, \hat{\mathbf{u}}_{xy|t_2^l}, ..., \hat{\mathbf{u}}_{xy|t_n^l}\right\}$ definiert sind - einen Vote für jeden Capsule-Typen in T^l. Diese Menge entsteht aus der Multiplikation zwischen einer gelernten Transformationsmatrix $M_{t_i^l}$ und dem unteren Gitter der Kind-Capsule-Ausgaben $U_{xy|t_i^l}$, innerhalb des benutzerdefinierten Kernels der an Position (x, y) in Layer l zentriert ist, weshalb $\hat{\mathbf{u}}_{xy|t_i^l} = M_{t_i^l} \times U_{xy|t_i^l}$ für alle $t_i^l \in T^l$ gilt. Hierbei hat jedes $U_{xy|t_i^l}$ die Form $k_h \times k_w \times z^l$ wobei $k_h \times k_w$ die Größe des benutzerdefinierten Kernel ist. Jedes $M_{t_i^l}$ hat die Form $k_h \times k_w \times z^l \times |T^{l+1}| \times z^{l+1}$ für alle Capsule-Typen T^l, wobei $|T^{l+1}|$ die Anzahl an Eltern-Capsules in Layer $l+1$ darstellt. Kein $M_{t_i^l}$ hängt von der räumlichen Position (x, y) ab, da die Transformationsmatrix über alle räumlichen Positionen innerhalb eines Capsule-Typen geteilt wird (ähnlich wie wenn conv. Kernel eine Input-Feature-Map scannen). Diese gemeinsame Nutzung von Parametern reduziert die Gesamtzahl der zu lernenden Parameter drastisch.

Die Werte der Transformationsmatrizen für jeden Capsule-Typen in einem Layer werden schließlich über Backpropagation mit einer überwachten Loss-Funktion gelernt [RL18].

Algorithm 1 Locally-Constrained Dynamic Routing.

procedure ROUTING($\hat{\boldsymbol{u}}_{xy|t_i^\ell}$, d, ℓ, k_h, k_w)
  for all capsule types t_i^ℓ within a $k_h \times k_w$ kernel centered at position (x, y) in layer ℓ and capsule xy centered at position (x, y) in layer $(\ell + 1)$: $b_{t_i^\ell|xy} \leftarrow 0$.
  for d iterations **do**
    for all capsule types t_i^ℓ in layer ℓ: $\mathbf{c}_{t_i^\ell} \leftarrow \texttt{softmax}(\mathbf{b}_{t_i^\ell})$ ▷ `softmax` computes Eq. 1
    for all capsules xy in layer $(\ell + 1)$: $\boldsymbol{p}_{xy} \leftarrow \sum_n r_{t_i^\ell|xy} \hat{\boldsymbol{u}}_{xy|t_i^\ell}$
    for all capsules xy in layer $(\ell + 1)$: $\boldsymbol{v}_{xy} \leftarrow \texttt{squash}(\boldsymbol{p}_{xy})$ ▷ `squash` computes Eq. 2
    for all capsule types t_i^ℓ in layer ℓ and capsules xy in layer $(\ell + 1)$: $b_{t_i^\ell|xy} \leftarrow b_{t_i^\ell|xy} + \hat{\boldsymbol{u}}_{xy|t_i^\ell} . \mathbf{v}_{xy}$
  return $\mathbf{v}_{xy}$

Abbildung 3.43: Pseudocode des Rounting-Algorithmus der SegCaps mit Einschränkung eines benutzerdefinierten Kernels aus [RL18]

Wie der Pseudocode von Abbildung 3.43 darstellt, wird analog zu Abschnitt 3.3 für die Ausgabe zu den Eltern-Capsules $\mathbf{p}_{xy} \in P$ die gewichtete Summe $\mathbf{p}_{xy} = \sum_n r_{t_i^l|xy} \hat{\mathbf{u}}_{xy|t_i^l}$ über die Votes berechnet, wobei $r_{t_i^l|xy}$ die Coupling-Koeffizienten darstellen [RL18] und die Berechnungen durch den benutzerdefinierten Kernel xy räumlich beschränkt sind. Die Menge von Coupling-Koeffizienten eines Layers wird analog zum Original mit $\mathbf{c}$ bezeichnet und mit einer Routing-Softmax wie folgt berechnet [RL18]:

$$r_{t_i^l|xy} = \frac{exp(b_{t_i^l|xy})}{\sum_k exp(b_{t_{ik}^l})} \tag{3.26}$$

Wobei die Logits $b_{t_i^l|xy}$ die logarithmierten a-priori-Wahrscheinlichkeit darstellen. Die Ausgabe wird dann durch die in Abschnitt 3.3 vorgestellte nichtlineare Squash-Funktion berechnet. Zuletzt wird das Agreement als Skalarprodukt $a_{t_i^l|xy} = \mathbf{v}_{xy}\hat{\mathbf{u}}_{xy|t_i^l}$ gemessen, wobei $\mathbf{v}_{xy}$ die Ausgabe der Squash-Funktion ist. So wird nur zwischen Capsules innerhalb des benutzerdefinierten Kernels geroutet. Eine finale Segmentierungsmaske wird erstellt indem die Länge der Capsule-Vekoren im letzten Layer berechnet wird und die positive Klasse denjenigen Vektoren zugeordnet wird, deren Größe über einem Schwellenwert liegt. Andernfalls erhält der Vektor die negative Klasse. Der letzte Layer (in Abbildung 3.42 (*rot*)) ist hierfür so groß wie das Eingabebild und besitzt nur einen Capsule-Typen [RL18].
Da nur die Verteilung der positiven Eingabeklasse modelliert und alle anderen Pixel als Hintergrund behandelt werden sollen, werden Ausgabe-Capsules, welche nicht zur positiven Klasse gehören, über ein Label maskiert. Somit wird eine maskierte Version des Eingabebildes ohne Hintergrund erhalten. Dabei führt, wie Abbildung 3.44 zeigt, ein dreischichtiges conv. Netz mit 1x1 Filtern die Rekonstruktion durch, wobei ein gewichteter mittlerer quadratischer Loss zwischen den positiven Eingangspixeln und der Rekonstruktion berechnet wird. Eine Beispiels-Rekonstruktionen ist in Abbildung 3.44 veranschaulicht.
Somit wurde als Regularisierungstechnik der Input rekonstruiert, um eine bessere Einbettung des Input-Raums zu erhalten. Damit wird das Netzwerk laut den Autoren nicht nur gezwungen, alle notwendigen Informationen über eine gegebene Eingabe beizubehalten, sondern auch dazu gebracht, die vollständige Verteilung des Eingangsraums besser darzustellen [RL18].

Als Beispielanwendung wurde das Modell für die Segmentierung von pathologischen Lungen von CT-Scans angewandt. Hierfür wurde der aus 878 512x512 Bildern bestehende LUNA15 Datensatz verwendet und die tiefe Architektur mit einer SegCaps-Baseline, welche mit drei Capsule-Layern näher an den original CapNets liegt, und mit anderen Architekturen verglichen. Der Datensatz wurde u.a. durch räumliche Transformationen und zufälliges Rauschen künstlich vergrößert. Das Modell wurde mit einem gewichteten Binary Cross Entropy Loss (BCE) für die Segmentierungsausgaben trainiert, die Baseline mit einer gewichteten binären Version des Margin-Loss aus Abschnitt 3.3.5. Dieser wurde nicht genauer beschrieben. Die Lernrate von Adam lag bei 0.00001 mit einem Decay Faktor von 0.05 nach $50\,000$ Iterationen und Early Stopping. Die erzielten Ergebnisse zeigen, dass der gewichtete Margin-Loss in kleinen Experimenten vergleichbar mit dem gewichteten BCE-Verlust für SegCaps zu sein scheint. Jedoch sind gründlichere Experimente erforderlich, um daraus Schlüsse zu ziehen.
Im Vergleich schneidet die vorgestellte SegCaps-Architektur mit der höchsten Genauigkeit von 98.48% ab und besitzt mit 1.4M Parameter 95.4% weniger als ein U-Net ([RFB15]), welches mit 31.0M eine Genauigkeit von 98.45% aufweist. Außerdem besitzt das tiefe CapsNet

38.4% Parameter weniger als Tiramisu ([Jég+16]) mit 2.3M und einer Genauigkeit von 98.45% [RL18].

Abbildung 3.44: SegCaps Rekonstruktionen von Lungen mit schrittweise abgeänderten Capsule-Vektor Werden aus [RL18]

Abbildung 3.44 veranschaulicht je Reihe ein ausgewähltes visuelles Attribut von verschiedenen 16D-Ausgabevektoren des letzten SegCaps-Layers. Die Werte der Vektoren wurden je Spalte verändert. Zu erkennen ist, dass Regionen mit unterschiedlichen Textureigenschaften durch die Vektoren eingefangen werden und sich deren Intensität Spalte für Spalte steigert [RL18].

3.5.5 VideoCapsuleNet

Trotz der großen Fortschritte im Bereich des Deep Learning ist die Aktionserkennung (engl. action detection), sprich das Herauslesen einer Aktion wie „Greife nach Objekt“ aus der Eingabe, eine Herausforderung [DRS18]. Ein VideoCapsuleNet wurde für solch eine pixelweise Aktionserkennung konzipiert, und mit State-of-the-Art-Performance auf Datensätze dieses Gebiets angewandt.
Hierbei stellt das Netz eine Verallgemeinerung des Matrix-CapsNets von 2D auf 3D dar, welche eine Sequenz von Video-Frames als Eingabe erhält. Die 3D-Verallgemeinerung erhöht allerdings die Anzahl der Capsules drastisch, weshalb das Routing rechenintensiv wird. Ein sogenanntes Capsule-Pooling im conv. Capsule-Layer löst dieses Problem indem ein neues Abstimmungsverfahren für den conv. Capsule-Layer Verwendung findet. Dabei wird, kurz erläutert, die Anzahl der abgegebenen Votes reduziert, indem die Transformationsmatrix nur auf den Mittelwert der Capsules im rezeptiven Feld jedes Capsule-Typen angewandt wird. Weiter werden Transformationsmatrizen zwischen Capsules desgleichen Typs geteilt (ähnlich zu Abschnitt 3.5.4), da Capsules desgleichen Typs dieselbe Entität an verschiedenen Positionen modellieren, weshalb ihre Votes nicht basierend auf ihrer Position variieren sollten. Es wurde beobachtet, dass das Routing die Aktionsdarstellungen modelliert und dass verschiedene Aktionsmerkmale von Capsules erfasst werden. Dies inspirierte die Autoren die Capsules für die Aktionslokalisierung zu verwenden und die klassenspezifischen Capsules zu nutzen um eine pixelweise Lokalisierung von Aktionen vorzunehmen [DRS18].

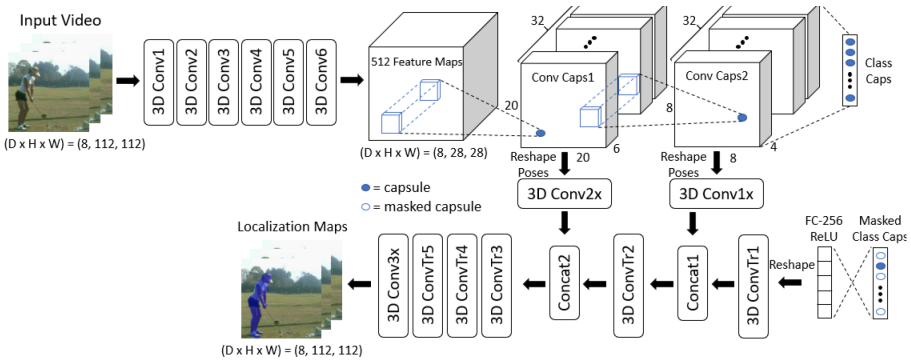


Abbildung 3.45: Architektur des VideoCapsuleNets aus [DRS18]

Abbildung 3.45 zeigt die Architektur des VideoCapsuleNet wobei vorerst aus den Video-Eingaben mithilfe von 3D conv. Layern Features extrahiert werden. Diese Merkmale formen dann den ersten Caps-Layer (*Conv Caps1*), worauf ein conv. Caps-Layer (*Conv Caps2*) und ein fc Caps-Layer (*Class Caps*) folgen. Das Decoder-Netzwerk verwendet daraufhin die Ausgabe des maskierten fc Caps-Layers und die Lokalisierung der Aktion wird durch parametrisierte Skip-Verbindungen (z.B. 3D Conv2x) mit den conv. Caps-Layern weiter verbessert. Das Netzwerk wird schließlich mittels eines Klassifizierungs- (*Masked ClassCaps*) sowie eines Lokalisierungs-Loss (*Localization Maps*) Ende-zu-Ende trainiert [DRS18]. Als Ergebnis des Modells ist z.B. die lokalisierte Aktion „Golf“ in Blau als Ausgabe unter den *Localization Maps* zu erkennen.

3.6 Ein weiterer Routing-Algorithmus: Eine Optimierungssicht auf das dynamische Routing zwischen Capsules

Schon einige der oberen Modelle nahmen Veränderungen am Routing vor oder konstruierten eine neue Version von diesem. Da im Folgenden der Fokus auf dem Routing-Algorithmus liegt und dieser diskutiert wird, wurde das entsprechende Paper in diese Kategorie eingeteilt, auch wenn es aufgrund deren engen Kopplung schwierig ist das Routing-Verfahren vom eigentlichen Modell zu trennen.

Das dynamische Routing der originalen CapsNets von Sabour et al. [SS17] stellte sich bereits für einige Probleme als effektiv heraus, jedoch fehlt eine Standardformalisierung dieser Heuristik und ihrer Implikationen. In [DW18] wird die Routing-Strategie teilweise als ein Optimierungsproblem formuliert, welches eine Kombination eines clusterbildungs-ähnlichen Losses mit einem KL-Regularisierungsterm darstellt. Dieser Regularisierungsterm liegt zwischen der aktuellen Verteilung der Coupling-Koeffizienten und den vorherigen Coupling-Koeffizienten. Dabei bezeichnet die KL-Divergenz (Kullback-Leibler-Divergenz), kurz erläutert, ein Maß für die Unterschiedlichkeit zweier Wahrscheinlichkeitsverteilungen.

Basierend auf dieser Erkenntnis wurde ein Routing-Ansatz mit einigen interessanten Eigenschaften eingeführt, welcher experimentell den originalen Routing-Algorithmus bei einer unüberwachten, perzeptuellen Gruppierungsaufgabe übertrifft. Für weitere Aufgaben wurde der folgende Algorithmus zum Kenntnisstand dieser Arbeit nicht getestet [DW18].
Sei $\hat{\mu}_{j|i} = T_{ij}\mu_i$ die Sammlung aller Ausgabevektoren der niedrig-leveligen Capsules, wobei μ_i die Ausgabe einer niedrigen Capsules i darstellt und T_{ij} die Transformationsmatrix, welche die niedrige Capsule i mit der höheren Capsule j in Verbindung bringt. Die Repräsentationen der höheren Capsules werden als $S = s_1, ..., s_k$ bezeichnet, wobei s_j im gleichen Feature-Raum wie die Ausgabe $\hat{\mu}_{j|i}$ der niedrigen Capsules liegt. Hier bezeichnet w_j die Aktivierungswahrscheinlichkeit der höher liegenden Capsules j und $C = [c_{ij}]_{i,j}$ sind die Coupling-Koeffizienten berechnet zwischen Capsule i und j. Nun kann das Routing-by-Agreement aus Abschnitt 3.3.2 wie folgt notiert werden [DW18]:

Algorithm 1 The Routing Procedure in Sabour et al. (2017)

1: for all capsule i in layer ℓ and capsule j in layer $(\ell + 1)$: $b_{ij} = 0$
2: **for** iteration t **do**
3: for all capsule i in layer ℓ: $c_{ij} = \frac{\exp(b_{ij})}{\sum_k \exp(b_{ik})}$
4: for all capsule j in layer $(\ell + 1)$: $\hat{s}_j = \sum_i c_{ij}\hat{\mu}_{j|i}, \quad s_j = \hat{s}_j/||\hat{s}_j||$.
5: for all capsule i in layer ℓ and j in layer $(\ell+1)$: $b_{ij} = b_{ij} + w_j\langle\hat{\mu}_{j|i}, s_j\rangle$, where $w_j = \frac{||\hat{s}_j||^2}{1+||\hat{s}_j||^2}$.
6: **end for**
7: **Return** $w_j s_j$

Abbildung 3.46: Das originale Routing-by-Agreement als Loss einer Clusterbildung kombiniert mit einem KL-Divergenz-Regularisierungsterm aus [DW18]

In Abbildung 3.46 repräsentiert $\langle ., . \rangle$ das Skalarprodukt und im Vergleich mit dem originalen Algorithmus aus Abbildung 3.18 sind einfache Umstellungen in Zeile *4* und *5* zu erkennen. So wird z.B. der rechte Bruch $s_j = \hat{s}_j/||\hat{s}_j||$ der Squashing-Funktion 3.3, welcher den Eingabevektor auf Einheitslänge skaliert, separat in Zeile *4* berechnet. Hierbei wurde, wie bereits erwähnt, von den Autoren beobachtet, dass das Routing-by-Agreement aus Abschnitt 3.3.2 als ein Optimierungsproblem formuliert werden kann. Dabei ähnelt der Loss dem einer Clusterbildung, welcher mit einem KL-Divergenz-Regularisierungsterm kombiniert wurde, der zwischen der alten und der aktuellen Coupling-Koeffizienten-Verteilung liegt [DW18]. Dies zeigt Gleichung 3.27, wobei $C^{old} = [c_{ij}^{old}]_{i,j}$ die Coupling-Koeffizienten des letzten Schrittes repräsentieren [DW18].:

$$\min_{C,S} \left\{ \mathcal{L}(C,S) := -\sum_{i,j} w_j c_{ij} \langle \hat{\mu}_{j|i}, s_j \rangle + \alpha KL(C||C^{old}) \right\}, \quad s.t.\ c_{ij} > 0,\ \sum_j c_{ij} = 1,\ ||s_j|| \leq 1 \tag{3.27}$$

Kurz erläutert stellt der *linke Teil* der Gleichung den clusterbildungs-ähnlichen Loss dar, welcher sich aus der aufsummierten Multiplikation der Aktivierungswahrscheinlichkeit von Capsule j mit der separaten Skalierung zur Einheitslänge jedes Votes, skaliert mit dem Coupling-Koeffizienten, berechnet. Der *rechte Teil* stellt den skalierten KL-Regularisierungsterm dar [DW18]. Ein mathematischer Beweis, dass sich Gleichung 3.27 ähnlich zum Loss des orginal CapsNets verhält wurde von den Autoren nicht angegeben, jedoch wurde textuell erläutert was für den Beweis gezeigt werden muss: Eine typischer Weg Gleichung 3.27 zu lösen, ist es Coordinate Decents (Koordinatenabstieg) zu nutzen, welches C und S alternierend optimiert. Wird angenommen, dass dabei der Koeffizient a gleich 1 ist, kann einfach gezeigt werden, dass die Aktualisierung von $S := s_j$ in Zeile *4* des original Routing-Algorithmus aus Abbildung 3.46 äquivalent zum Koordinatenabstiegs an S mit fixiertem C ist und dass die Aktualisierung von $C = c_{ij}$ (Zeile *3* und *5*) dem Koordinatenabstieg an C mit fixiertem S entspricht [DW18].

Jedoch erklärt diese Aktualisierungsregel nicht die Aktualisierung von $w_j = ||\hat{s}_j||^2/(1+||\hat{s}_j||^2$, weshalb die Autoren eine neue Variante des Routing-Verfahrens vorschlagen, welche dieses Problem anspricht und eine Reihe weiterer Verbesserungen gegenüber dem ursprünglichen Routing-Algorithmus vornimmt. Das Verfahren ist dadurch motiviert, die folgende clustering-ähnliche Zielfunktion zu lösen [DW18]:

$$\min_{C,S}\left\{\mathcal{L}(C,S) := -\sum_i\sum_j c_{ij}\langle o_{j|i}, s_j\rangle + \alpha\sum_i\sum_j c_{ij}logc_{ij}\right\}, \quad s.t.\ c_{ij} > 0,\ \sum_j c_{ij} = 1,\ ||s_j|| \leq 1 \tag{3.28}$$

Wobei $o_{j|i} = \frac{1}{||T_{ij}||_F}T_{ij}\mu_i$ und $||T_{ij}||_F$ die Frobeniusnorm von T_{ij} ist. Die Zielfunktion ist ähnlich zu der des agglomerativen Fuzzy-K-Means-Algorithmus (Li et al. [Li+08]) [DW18]. Der agglomerative Fuzzy-K-Means-Algorithmus ist eine Erweiterung des Standard-Fuzzy-K-Means-Algorithmus durch Einführen eines Strafterms in der Zielfunktion, damit der Clusterbildungsprozess nicht empfindlich gegenüber den anfänglichen Clusterzentren ist [Li+08]. Wenn die Aktualisierungen des Koordinatenabstiegs von C und S abgeleitet werden, entsteht die Aktualisierung des Algorithmus in Abbildung 3.47 [DW18]:

Algorithm 2 Our Routing Algorithm

1: **for** iteration t **do**
2: for all capsule i in layer ℓ and capsule j in layer $(\ell+1)$: $b_{ij} = \frac{1}{\alpha}\langle o_{j|i}, s_j\rangle \quad c_{ij} = \frac{\exp(b_{ij})}{\sum_k \exp(b_{ik})}$.
3: for all capsule j in layer $(\ell+1)$: $\hat{s}_j = \sum_i c_{ij} o_{j|i}, \quad s_j = \hat{s}_j / ||\hat{s}_j||$.
4: **end for**
5: for all capsule j in layer $(\ell+1)$: $w_j = \frac{||\sum_i c_{ij} o_{j|i}||}{1+\max_k ||\sum_i c_{ik} o_{k|i}||}$
6: **Return** $w_j s_j$

Abbildung 3.47: Optimierter Routing-Algorithmus aus [DW18]

An dieser Stelle könnte nun ähnlich zu Abschnitt 3.3 die Zielfunktion und der Algorithmus 3.47 Zeile für Zeile intuitiv und mit mathematischem Hintergrund erläutert werden, jedoch wird hier nur eine interessante Gemeinsamkeit mit dem EM-Routing aus Abschnitt 3.4 und mit den S-Capsules aus Abschnitt 3.5.1 aufgezeigt: Dieser aus dem originalen CapsNet entstandene Algorithmus hat Ähnlichkeit zu einer speziellen Art von K-Means, wobei K-Means wiederum Gemeinsamkeiten mit dem EM-Algorithmus besitzt, da sich dieser bei gewollter harter Zuordnung der Datenpunkte ähnlich verhält. Der EM-Algorithmus brachte wiederum die S-Capsules hervor, welche auf einem Unterraum der Cluster operieren. Weiter nutzt der Fuzzy-K-Means-Algorithmus standardmäßig - wie der EM-Algorithmus - keine harten Zuordnungen, jedoch werden für die softe Zuordnungen bei Fuzzy-K-Means Distanzen der Datenpunkte zum Mittelpunkt verwendet, anstelle von „Wahrscheinlichkeiten“ von Datenpunkten unter einer Wahrscheinlichkeitsdichtefunktion. Diese Zusammenhänge implizieren daher, dass das Routing immer auf die Clusterfindung von Votes hinausläuft und dass es womöglich sinnvoll ist, weitere Routing-Algorithmen aus verschiedenen Clusterfindungsverfahren zu entwickeln und zu testen.

Der hier betrachtete Routing-Algorithmus 3.47 bringt nach Dilin Wang et al. [DW18] folgende Verbesserungen gegenüber Algorithmus 3.46. Diese Verbesserungen erklären einige oben beschriebenen Änderungen intuitiv:

Die Aktualisierung von w_j: Verglichen mit dem original Algorithmus 3.46 wurde die Abhängigkeit der Aktivierungswahrscheinlichkeit w_j von der neuen Zielfunktion 3.28 entfernt. Stattdessen wird w_j am Ende des Routing-Verfahrens gesetzt. Auf diese Weise wird der Routing-Algorithmus als ein Optimierungsproblem formuliert; außerdem könnte dies verhindern, dass w sehr unausgewogen wird, wenn die Nummer an Iterationen wächst [DW18].

Scale-Invariant: Die Transformationsmatrix T_{ij} wird mit $o_{j|i} = \frac{1}{||T_{ij}||_F} T_{ij}\mu_i$ durch die Frobeniusnorm normalisiert, bevor sie als Eingabe der Prozedur überreicht wird. Diese Norm findet Verwendung, da sofern $\hat{\mu}_{j|i} = T_{ij}\mu_i$ und $||\mu_i|| \leq 1$ gilt, die Matrix T_{ij} regularisiert werden

muss, um den Trainingsprozess zu stabilisieren. Damit wird verhindert, dass die linke Seite der Gleichung 3.27 während des Trainings gegen negativ unendlich strebt. Die Zielfunktion 3.27 löst weiter dieses Problem nicht selbstständig; in Sabour et al. [SS17] verhindert dies stattdessen der in Abschnitt 3.3.4 vorgestellte Margin-Loss: Wird eine Entität von Capsule j repräsentiert, dann wird der Loss von $max(0, m^+ - w_j)^2$ angewandt, ansonsten $max(0, w_j - m^-)^2$, wobei $m^+ = 0.9$ und $m^- = 0.1$ ist. Dies ist äquivalent, als würde dafür gesorgt werden, dass:

$$\frac{m^-}{1 - m^-} \leq ||\sum_i c_{ij} T_{ij} \mu_i||^2 \leq \frac{m^+}{1 - m^+} \tag{3.29}$$

ist [DW18]. Sprich als würde die Matrix in ein begrenztes Intervall eingeschränkt und stabilisiert werden. Dieser Ansatz ist jedoch nicht sonderlich generell, weshalb hier ein genereller, bereits erwähnter, Lösungsansatz vorgestellt wird: Für jede Routing-Iteration wird die Gewichtsmatrix T_{ij} neu skaliert, und die Größe des Skalarprodukts zwischen $\langle \frac{1}{||T_{ij}||_F} T_{ij} \mu_i, s_j \rangle$ wird auf einen Wert unter eins beschränkt. Weiter hängt, wie bereits erwähnt, die Aktivierungswahrscheinlichkeit w_j nicht von der Größe von T_{ij} ab [DW18].

Annealing des Regularisierungsterms: Sabour et al. [SS17] nutzen eine $KL(C||C^old)$ Regularisierung mit einem fixen Koeffizient $a = 1$. Hier wurde die KL-Regularisierung durch eine Entropie-Regularisierung ersetzt, welche c_{ij} nicht mehr so stark vom vorherigen Wert abhängig macht. Daher hängt der Wert nun gleichmäßiger von der Eingabe ab, weshalb der Algorithmus stabilisiert wird. Weiter findet eine allmähliche Verringerung des Wertes von a während der Iterationen statt, da intuitiv die Votes von niedrigen Capsules in der frühen Phasen nicht vertrauenswürdig sind. Deshalb sollte zu Beginn des Trainings ein größeres a verwendet werden, damit die Entropie eine wichtigere Rolle spielt und der Routing-Prozess versucht, jede Capsule im niedrigeren Layer zu mehreren höheren Capsules auf eine einheitlichere Weise zuzuweisen. In der späten Phase des Trainings sollte folglich ein kleines a verwendet werden, damit der Routing-Prozess versucht, die Agreements zwischen niedrigen und hohen Capsules zu maximieren, worauf folgt, dass c_{ij} deterministischer wird. Die Autoren stellen weiter fest, dass der Wert von a bei vielen unüberwachten Aufgaben entscheidend für die Leistung ist und werden dies in zukünftiger Arbeit weiter diskutieren [DW18].

Die Performance des neuen Routing-Verfahrens wurde mit dem original Routing verglichen. Hierbei wurde eine unüberwachte Gruppierungsaufgabe, bei welcher drei zufällig gewählte Formen die in zufälligen Positionen in einem 28x28 Bilder liegen gruppiert werden sollen, für den Vergleich herangezogen. Dabei wurde - kurz beschrieben - ein neuronales Netz dazu trainiert die Capsule-Repräsentationen $s_1 ... s_k$ 3.3.3 in pixelweise Klassifizierungen umzuwandeln [DW18]. Das CapsNet hatte dieselbe Struktur wie das Original.

	Routing =3	Routing = 5	Routing =10	Routing =15
(Sabour et al., 2017)	0.647	0.830	0.862	0.879
Our Approach	0.816	0.901	0.911	0.914
N-EM (Greff et al., 2017)*	0.475			
RNN-EM (Greff et al., 2017)*	0.826			

Abbildung 3.48: Vergleich der Algorithmen zur Formen-Gruppierung aus [DW18]

Abbildung 3.48 zeigt, dass das Verfahren im Vergleich mit dem originalen Routing und zwei anderer neuronaler EM-Verfahren am besten abschneidet [DW18]. Interessant wäre an dieser Stelle noch ein Vergleich mit dem EM-Routing aus Abschnitt 3.4.3, jedoch war dies zur Veröffentlichung dieses Paper noch nicht bekannt. Weiter wäre eine Evaluation des vorgestellten Routings auf Klassifizierungs-Datensätze wie MNIST oder smallNORB spannend, jedoch wurde dazu, zum Kenntnisstand dieser Arbeit, keine weiteren Benchmarks veröffentlicht (laut einem Blog-Eintrag wird jedoch daran gearbeitet [Aob]) Ein weiterer Vorteil bezüglich der Rechenleistung ist, dass dieses Verfahren in Abbildung 3.48 bereits nach den ersten drei Routing-Iterationen bessere Ergebnisse zeigt als das Original. Hier würde ein tieferer Vergleich der Rechenzeiten das Ergebnis vermutlich weiter untermauern.

3.7 Erweiterungen für Capsule Networks

Als eine Erweiterung wird in diesem Abschnitt beispielsweise eine neue Aktivierungsfunktion oder das Einfügen von anderen „Deep Learning Blöcken“ in ein CapsNet verstanden. Daher handelt es sich hierbei um die Anwendung von zugeschnittenen oder losgelösten Verfahren auf Capsule-Modelle.

3.7.1 Non-Parametric Transformation Networks mit Capsule Networks

CNNs stellen sich in vielen Anwendungen als effektiv heraus, bei welchen es erforderlich ist Invarianzen für störanfällige Transformationen innerhalb einer Klasse zu lernen. Durch ihre Architektur wird jedoch nur Translations-Invarianz (und bedingte Rotationsinvarianz) erzwungen. Daher wurde in [PS18] eine neue Klasse von conv. Architekturen vorgestellt, die als Non-Parametric Transformation Networks (NPTNs) bezeichnet werden und allgemeine Invarianzen und Symmetrien direkt aus Daten lernen können. NPTNs stellen eine direkte und natürliche Verallgemeinerung von CNNs dar und können mittels Gradient Descent optimiert werden. Sie treffen keine Annahmen hinsichtlich der Struktur der Invarianzen, welche in den Daten vorhanden sind, weshalb diese als flexibel und leistungsfähig gelten. NPTNs wurden erfolgreich als Erweiterung von CapsNets angewandt.

Ein NPTN ist kurz ein Netzwerk von TN-Knoten (Transformation Network-Knoten). Abbildung 3.49 veranschaulicht hierbei die Operationen eines einzelnen TN-Knotens NPTN:

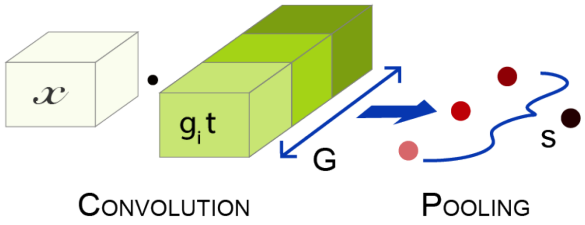


Abbildung 3.49: Operation eines Knoten des Non-Parametric Transformation Network aus [PS18]

Der Knoten in Abbildung 3.49 besitzt zwei Hauptkomponenten: (a) *Convolution* und (b) *Transformation Pooling*. Jeder Würfel zeigt eine einzelne Feature-Map bzw. Filter an. Erst wird das Skalarprodukt zwischen dem Eingabe-Patch (x) und einem Satz von $|G|$-Filtern (*grün*) berechnet. In diesem Beispiel ist $|G| = 3$. Dabei zeigt $g_i t$, dass die Transformation g_i auf den Filter t angewandt wird. Die resultierenden drei Ausgaben (*rot*) werden dann über eine Max-Pooling-Operationen gepoolt, um die finale Ausgabe s (*schwarz*) zu berechnen. Dieses Pooling steht im Gegensatz zu der einzelnen Faltungsoperation in CNNs: Der Pooling-Vorgang wird hier nicht räumlich ausgeführt, sondern über die $|G|$-Kanäle, welche nichtparametrische Transformationen codieren [PS18], wobei nichtparametrisch bedeutet, dass die Transformationen aus den Daten bestimmt werden und nicht a-priori festgelegt werden. Nichtparametrische Transformationen sind im Deep Learning demnach häufig vertreten. Nun ist die Ausgabe s invariant gegenüber der durch den Satz von Filtern G codierten Transformation.
Weiter können NPTNs als eine natürliche Verallgemeinerung von CNNs gesehen werden, da die Einstellung von $|G| = 1$ diese zu einem herkömmlichen CNN reduziert [PS18].

Auf Shifted-MNIST angewandte CapsNets werden durch die Verwendung von NPTN-Layern (anstelle von conv. Layern) verbessert und schneiden gegenüber allen im Paper verwendeten Baselines am besten ab. Der Test-Error der third-party Implementierung sinkt von 1.90% ohne NPTN-Layer auf 0.78% mit $|G| = 3$ [PS18]. In Abschnitt 3.3 besitzt jedoch das originale Modell einen niedrigeren Test-Error von 0.25%, an welchen die third-party Implementierung nicht herankommt. Zum Kenntnisstand dieser Arbeit erreichte allerdings keine third-party Implementierung diesen Test-Error. Die Autoren gehen hierbei davon aus, dass NPTN die 0.25% Test-Error der originalen Implementierung verbessern würde, jedoch muss erwähnt werden das CapsNet u.a. dazu designed wurde Max-Pooling Layer zu Gunsten der Äquivarianz abzuschaffen. Dies wirft die Frage auf, wieso der NPTN-Caps-Hyprid mit Max-Pooling die Ergebnisse weiter verbessert. Die Antwort liegt vermutlich in der Art des Max-Poolings: Capsules zie-

len auf räumliche Äquivarianz ab, das Max-Pooling in NPTNs ist jedoch, wie bereits erwähnt und im Gegensatz zu ConvNets, nicht räumlich, sondern über die $|G|$-Filter, welche nichtparametrische Transformationen codieren. Das Pooling könnte demnach so interpretiert werden, als würde es eine nichtparametrische Transformation behalten, welche aus allen nichtparametrische Transformation resultiert. Außerdem findet das Pooling innerhalb einer Primary-Capsule statt und ist daher nicht analog zum gesamten layerübergreifenden Pooling in CNNs. Somit werden die Votes der unteren Capsules mit mehreren nichtparametrischen Transformationen zur oberen Capsule transformiert (anstelle eines Gewichtes oder conv. Layer) und die - in welcher Art auch immer - aus den Transformationen resultierende Transformation wird behalten, als würde versucht werden über verschiedenen Bewegungen einen Deckel (untere Capsule) auf eine Flasche (obere Capsule) zu bekommen um schließlich daraus eine einzelne Gesamtbewegung, die sich aus allen anderen Bewegungen ergibt, zu bilden.

3.7.2 Schnell konvergierendes Capsule Network mit Anwendung auf MNIST

In [Zou+18] wird eine neue Aktivierungsfunktion für das originale CapsNet vorgestellt und ein zusätzlich Loss zur Zielfunktion hinzugefügt, wodurch die Konvergenz, die Generalisierungsfähigkeit und die Effizienz des Netzwerkes verbessert wird.
Die Autoren argumentieren, dass die originale Squashing-Funktion als eine Art Kompression und Neuverteilung der Länge des Vektors angesehen werden (der Vektor wird auf die Länge eins komprimiert und seine Werte dementsprechend umverteilt) und stellen darauf basierend eine neue, in ihren Experimenten besser abschneidende Aktivierungsfunktion vor [Zou+18]:

$$\mathbf{v}_j = (1 - exp^{-a\|\mathbf{s}_j\|^b})\frac{\mathbf{s}_j}{\|\mathbf{s}_j\|} \tag{3.30}$$

Diese Aktivierungsfunktion ist in zwei Teile unterteilt. Dabei übernimmt der linke Teil $1 - exp^{-a\|\mathbf{s}_j\|^b}$ die Rolle der zusätzlichen Skalierung der Squashing-Funktion und komprimiert die Länge des Eingabevektors $\mathbf{s}_j$ in ein Intervall von $[0.1)$. Der rechte Teil $\frac{\mathbf{s}_j}{\|\mathbf{s}_j\|}$ ist analog zur originalen Squashing-Funktion der Einheitsvektor des Eingabevektors. In der vorgestellten Aktivierungsfunktion sind a und b beide Zahlen über null, und können als Hyperparameter nach eigenen Bedürfnissen angepasst werden ($a = 10$ und $b = 1$ führt mit folgender Loss-Funktion zu 99.75% Genauigkeit auf MNIST) [Zou+18].
Weiter werden von den Autoren Vorteile dieser Aktivierungsfunktion gegenüber dem Original gezeigt: Erstens ist die Verwendung der Exponentialfunktion für die Ableitung förderlicher; zweitens führt diese Funktion zu einer schnelleren Konvergenzrate und drittens ist es einfach die Aktivierungsfunktion in das original CapsNet zu integrieren, um dadurch vermutlich eine höhere Genauigkeit zu erhalten. Hierbei ist jedoch zu erwähnen, dass nur der MNIST Datensatz getestet wurde [Zou+18].

Außerdem wurde der originale Loss um eine Weight-Attenuation-Function (in deutsch passend: Gewichtsdämpfungsfunktion) erweitert, welche wie folgt definiert ist [Zou+18]:

$$Wloss = \sum_{w} \|w\|^2 \tag{3.31}$$

Hierbei bezeichnet w eine Capsule-Gewichtsmatrix des fc Capsule-Layers. Die quadrierte Norm der Gewichtsmatrix einer Capsule wird als sogenannter Wloss aufsummiert. Der Wloss versucht, die Gewichte während des Trainings zu minimieren, sodass das Netzwerk es vorzieht, kleinere Gewichte zu erlernen und das Overfitting zu reduzieren. Die gesamte Loss-Funktion ist dann wie folgt definiert [Zou+18]:

$$loss = \alpha Mloss + \beta Rloss + (1 - \alpha - \beta)Wloss \tag{3.32}$$

Die Loss-Funktion enthält drei Teile: Mloss (Margin-Loss), Rloss (Rekonstuktion-Loss) und Wloss (Weight-Loss). Der Anteil der drei Teile kann durch die Koeffizienten α und β eingestellt werden ($\alpha = 0.5, \beta = 0.4$ für 99.75% auf MNIST). Sowohl α als auch β sind Werte im Intervall $[0, 1]$ und es gilt $0 < \alpha + \beta \leq 1$. [Zou+18].

3.7.3 Dichte und vielfältige Capsule Networks

In [SSRP18] entwickelten Sai Samarth R Phaye et al. zwei CapsNet-Modelle: das Dense Capsule Networks (DCNets) und das Diverse Capsule Networks (DCNet++). DCNets ersetzt den anfänglichen conv. Layer eines CapsNets durch ein densely connected conv. Netz. Dieses Ersetzen ist nicht mit Abschnitt 3.7.1 zu verwechseln, da dort im Gegensatz der conv. Layer jeder einzelnen Capsule mit einem NPTN ausgetauscht wird. Die vom conv. Layer des originalen CapsNet gelernten Feature-Maps verzeichnen sehr grundlegende Funktionen, die möglicherweise nicht ausreichen, um Capsules für komplexe Datensätze zu erstellen. Daher versuchten die Autoren zuerst mehrere conv. Layer in der ersten Schicht zu verwenden (einmal zwei und einmal acht) und beobachteten, dass dies zu keiner Verbesserung führte.

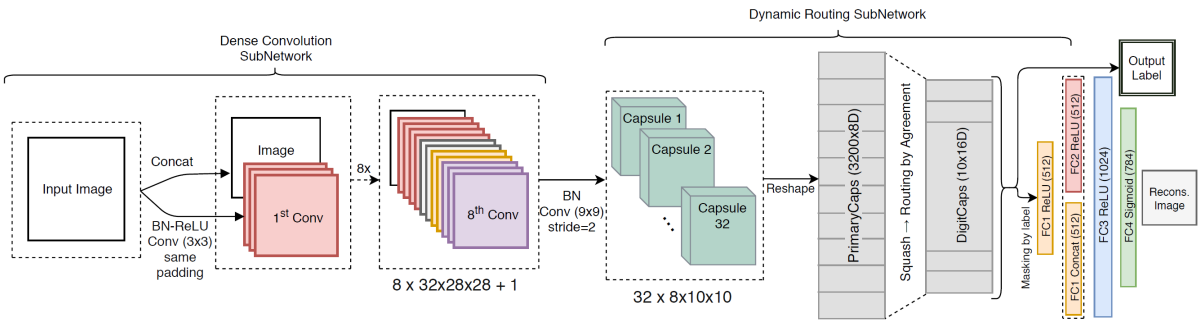


Abbildung 3.50: Die Architektur des Dense Capsule Networks (DCNets) aus [SSRP18]

Wie Abbildung 3.50 zeigt bildet DCNet eine tiefere Architektur, in welcher ein achtschichtiges, dichtes, conv. Subnetz mit Skip-Verbindungen genutzt wird (*Dense Convolution Subnetwork*). Jeder Layer wird feedforward mit allen folgenden dichten conv. Layern verkettet, bis diese folglich in einem letzten dichten conv. Layer zusammenkommen. Dies führt zu einem besseren Gradientenfluss im Vergleich zu direkt gestapelten Layern. Inspiriert von den dichten Verbindungen wurde weiter das Rekonstruktions-Modul des CapsNets modifiziert. Abbildung 3.50 veranschaulicht hierbei, dass der Decoder ein vierschichtiges Modell darstellt und eine Verkettung von Merkmalen des ersten Layers und des zweiten Layers besitzt, wodurch bessere Rekonstruktionen erzielt werden. Der Decoder übernimmt die durch das Label maskierte Ausgabe des DigitCaps-Layers als Eingabe. Die Anzahl der Neuronen im Decoder wird von 512 auf 600 und von 1024 auf 1200 geändert, wenn das Bild größer als 32x32 ist [SSRP18]. Dieses fc Modul wird ebenfalls in der Diskussion des conv. Modules in Abschnitt 4.4.6 des praktischen Teil erwähnt.

Experimente zeigen eine signifikante Steigerung der Leistungsfähigkeit bei verschiedenen Datensätzen wie MNIST, Fashion-MNIST, SVHN im Vergleich zum Original. Die Leistung von DCNet auf dem Cifar10 Datensatz gegenüber einem einfachen CapsNet-Basismodell mit denselben Parametern wurde ebenfalls gesteigert, jedoch wurde das Sieben-Ensemble-Modell des Originals aus Abschnitt 3.3.5 mit einer Genauigkeit von 89.40% nicht übertroffen. Dies liegt daran, dass die Bilder in Cifar10 im Vergleich zu MNIST komplexer sind, weshalb es für das Netzwerk nicht einfach ist die Teil-Ganzes-Beziehungen zu codieren. Aus diesem Grund erstellten die Autoren das Diverse Capsule Network (DCNet++) [SSRP18] :

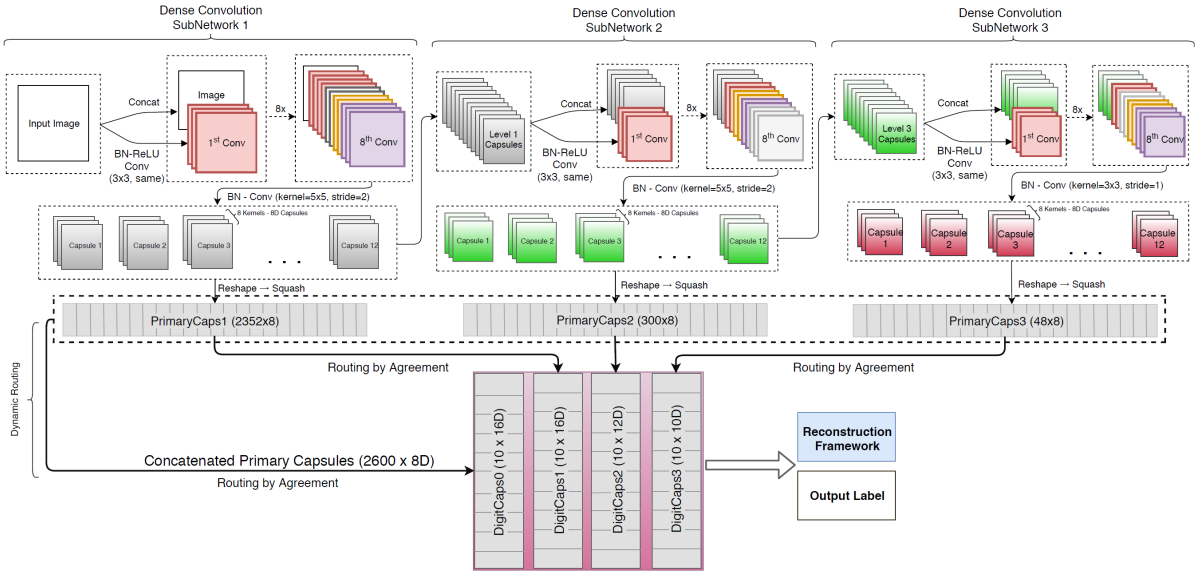


Abbildung 3.51: Die Architektur des Diverse Capsule Networks (DCNet++) aus [SSRP18]

Abbildung 3.51 zeigt hierbei, dass das DCNet++ ein hierarchisches Modell darstellt, wobei das obige DCNet-Modell verwendet wird und dessen Zwischenrepräsentation als Eingabe für das nächste DCNet dienen, welches wiederum eine Repräsentation erzeugt, welche der nächsten DCNet-Schicht zugeführt wird. Zusätzlich zu den hierarchisch entstandenen DigitCaps-Layern (*DigitCaps1-3*) wird ein weiterer DigitCaps-Ausgabelayer genutzt (*DigitCaps0*), indem die gesamte Verkettung der drei DCNet-PrimaryCaps-Layern geroutet wird (*Concatenated Primary ...*). Der Grund für das Hinzufügen dieser zusätzlichen Ebene besteht darin, dem Modell zu ermöglichen, kombinierte Merkmale aus verschiedenen Capsule-Höhen zu lernen. DCNet++ erreicht eine State-of-the-Art-Performance von 99.75% auf MNIST bei einer zwanzigfachen Verringerung der gesamten Trainings-Iterationen gegenüber dem herkömmlichen CapsNet. Bei SVHN schneidet DCNet++ mit 96.90% besser ab als das Original und übertrifft das Ensemble von sieben CapsNet-Modellen auf Cifar10 mit einer siebenfachen Verringerung der Parameteranzahl um 0.31% [SSRP18].

3.8 Verschiedene Klassifizierungen durch Capsule Networks

Die Modelle in diesem Abschnitt ähneln meist dem originalen CapsNet aus Abschnitt 3.3, weshalb hier dargestellt wird, wie mittels CapsNets anwendbare Klassifizierter entwickelt werden können, was die Intuition für die Anwendung von CapsNets weiter verstärkt.

3.8.1 Gehirntumor-Typ Klassifizierungen mit Capsule Networks

Gehirntumore gelten als eine der tödlichsten und häufigsten Formen von Krebs, sowohl bei Kindern als auch bei Erwachsenen, weshalb die Bestimmung des richtigen Gehirntumor-Typen in frühen Stadien von signifikanter Wichtigkeit für einen präzisen Behandlungsplan und für die Vorhersage der Reaktion des Patienten auf die gewählte Behandlung ist. Die Klassifizierung des Tumor-Typen bei Menschen stellt eine sehr zeitaufwändige und fehleranfällige Aufgabe dar, die stark von den Erfahrungen und Fähigkeiten des Radiologen abhängt.
Für die Bestimmung eines Typen können Magnetic Resonance Imaging (MRI)-Bilder verwendet werden, wobei das MRI durch seine harmlose Natur eine bevorzugte Methode darstellt. Daher bestand das zu klassifizierende Datenset aus 3064 MRI-Bildern von 233 Patienten die mit Tumor diagnostiziert wurden und beinhaltet sowohl die Gehirnbilder, also auch den segmentierten Tumor. Der Datensatz ist jedoch sehr klein, weshalb ein großes Interesse an Capsule Networks bestand, da diese als robust gegen Affine Transformationen und Rotationen gelten [AMP18]. Mit dem originalen CapsNet, doch mit 64 Feature-Maps anstelle von 256 im conv. Layer wurden folgende Ergebnisse erzielt [AMP18]:

1. Es wurde mit einer Genauigkeit von 86.56% auf segmentierten Bildern das Baseline-

CNN (72.13%) übertroffen. Dies spricht für die Aussage, dass CapsNets kleine Datensätze besser handhaben als CNNs. Hierbei besaß die Baseline zwei Layer mit 64x5x5 Filtern, einem Stride von eins und 2x2 Max-Pooling, worauf ein 800-800-3 MLP folgte [AMP18].

2. Die Klassifizierung auf dem gesamten Gehirnbild fällt mit 61.97% beim CNN und mit 78% beim CapsNet bei beiden Modellen schwächer aus. Grund dafür ist die Beeinträchtigung der Klassifizierung durch den variationsreichen Hintergrund. Dieses Ergebnis bestätigt die Aussage des original Papers, dass Capsules dazu tendieren alles zu modellieren [AMP18]. Im Nachhinein betrachtet und unabhängig von diesem Artikel ist dies offensichtlich, da ein Rekonstruktion-Loss des gesamten Bildes für das Training verwendet wird.

3. Das Overfitting Problem von Capsules wurde weiter untersucht, da die komplizierte Struktur und die verschiedenen lernbaren Parameter eines CapsNets schnell bei kleinen Datensätzen zu Overfitting führen. Dies wurde auch hier beobachtet weshalb als Lösung erfolgreich Early Stopping angewandt wurde [AMP18].

4. Für bessere Erklärbarkeit von CapsNets wurden analog zum Vorgehen in Abschnitt 3.3.4 die Werte der Ausgabevektoren leicht abgeändert und Bilder durch den Decoder erzeugt. Abbildung 3.52 zeigt einige Beispiele [AMP18]:

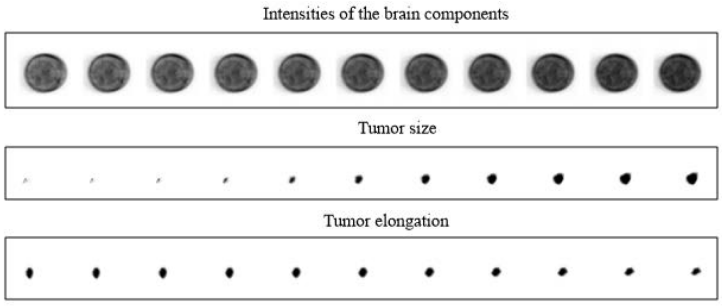


Abbildung 3.52: Gehirntumor Rekonstuktionen und Features durch leicht veränderte Ausgabevektor-Werte aus [AMP18]

In Abbildung 3.52 repräsentiert jede Spalte eine bestimmte rekonstruierte Eingabe unter Verwendung eines veränderten Ausgabevektor-Werts. Beispielsweise scheint das spezielle Feature, welches in der zweiten Spalte gelernt wurde, die Größe des Tumors zu repräsentieren, da die gezielte Abänderung dieses Features die Größe verändert. In ähnlicher Weise scheint die dritte Spalte mit der Breite des Tumors zu korrespondieren [AMP18].

3.8.2 Bildfusion der Fehlererkennung in Stromversorgungssystemen basierend auf Deep Learning

Die Überwachung der Fehler von Stromversorgungssystemen findet weite Anwendung in den Feldern Stations-Automatisierung, Katastrophenschutz und Alarm, intelligente Umspannstation, usw. Mit dem Ziel die drei Hauptprobleme von Stromversorgungssystemen - darunter Leckage (Leck in einem technischen System), hohe Temperatur und physikalische Schäden - zu untersuchen wird in diesem Paper eine Bildfusion und Fehlererkennungsmethode basierend auf CapsNets vorgestellt. Hierfür wurden drei unabhängige Datensätze erstellet die aus ultraviolettem, infrarotem oder sichtbarem Licht bestehen. Jeder Datensatz beinhaltet 400 Bilder über das Equipment eines Stromversorgungssystems und besteht hauptsächlich aus Öltransformatoren sowie Stromleitungen. Die Datensätze werden in 70% Trainingsdaten und 30% Testdaten aufgeteilt. Das sichtbare Bild wird als der Fusionsbildhintergrund verwendet, das Infrarotbild liefert die thermische Information der Ausrüstung und das Ultraviolettbild die elektrische Feldinformation auf der Außenseite der Ausrüstung. Durch jedes der Bilder können unterschiedliche Schäden erkannt werden, beispielsweise mögliche Hitzeschäden im Infrarotbild. Zusätzlich wurde durch ein aufwendiges Verfahren ein fusioniertes Bild mit der Fehlerposition als Label erstellt. Abbildung 3.53 zeigt hierbei die Pipeline der Bildverarbeitung [Li+18a]:

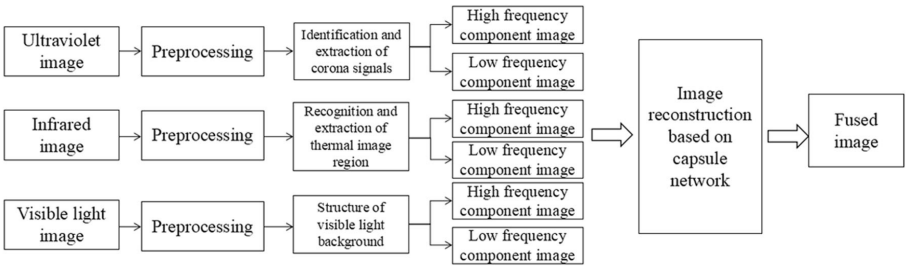


Abbildung 3.53: Pipeline der Bildverarbeitung der Fehlererkennung und Bildfusion in Stromversorgungssystemen [Li+18a]

Nach Abbildung 3.53 wird wie folgt vorgegangen [Li+18a]:

1. Das sichtbare Bild wird eingelesen und zum Preprocessing gegeben. Dieses führt kurz beschrieben hauptsächlich Bildverbesserung durch. Das verarbeitete Bild wird verwendet, um später den Hintergrund des fusionierten Bildes zu erstellen.

2. Das ultraviolette und infrarote Bild wird ebenfalls dem Preprocessing überreicht. Danach wird ein Canny-Zielerkennungs-Algorithmus dazu verwendet die Fehlerfläche der Bilder zu extrahieren.

3. Die Bilder werden in jeweils ein Hochfrequenz- und ein Niederfrequenz-Komponentenbild geteilt und separat durch ein CapsNet fusioniert.

Bei der Bildfusion werden sichtbare Licht-, Infrarot- und Ultraviolettbilder in der Form von Vektoren in einzelnen Capsules gespeichert. Folglich liegen die Bildmerkmale in einem dreidimensionalen Vektor bzw. in den drei Vektoren der PrimaryCaps vor, wobei die erste Dimension die Eigenschaften des Bildes aus sichtbarem Licht beinhaltet, die zweite die Eigenschaften des ultravioletten usw. Wie in den Abschnitten zuvor voten die PrimaryCaps für den ClassCaps-Layer über das Routing, was die Features der Bilder fusioniert. Das Fusions-Ergebnisbild entsteht aus der Rekonstruktion des CapsNets kombiniert mit dem sichtbaren Bild, genauer wird dieser Vorgang von den Autoren nicht beschrieben. Das CapsNet ähnelt jedoch stark dem Original aus Abschnitt 3.3, weshalb auf dieses nicht weiter eingegangen wird [Li+18a].
Die Autoren führen weiter aus, dass diese Methode effektiv die Datenredundanz in Systemen reduziert und dass das Fusions-Ergebnis dazu verwendet werden kann, Fehler zu erkennen und direkt im fusionierten Bild darzustellen [Li+18a]. Leider werden weder Ergebnisbilder noch Ergebnis- oder Vergleichswerte angegeben.

3.8.3 Die Klassifizierung von UAV-Reisbildern basierend auf Capsule Networks

Um das Wachstum von Reis überwachen zu können, um Krankheiten sowie Schädlinge zu verhindern, für eine effiziente Kontrolle, präzise Düngung, rationale Verteilung von Wasserressourcen, Ernteerkennung, Nährwertanalyse und schließlich gesteigerte landwirtschaftliche Erträge, ist es wichtig hochauflösenden Reisbilder, die durch ein unbemanntes Fluggerät (engl. Unmanned Aerial Vehicle kurz UAV) aufgenommen wurden, zu klassifizieren. Diese Bilder verzeichnen unterschiedliche Reis-Wachstumsstadien (Sämlingsstadium, Bestockungsstadium, Rispeln Differenzierungsstadium und Kornfüllstadium). Außerdem zeigen ein paar Bilder Krankheiten oder Schädlinge. Der Datensatz beinhaltet ebenfalls negative Proben, die im Allgemeinen Unkrautbilder darstellen. Die 500 Bilder besitzen eine Größe von 450x300 davon werden 300 für das Training und 200 für den Test ausgewählt [Li+18b].

Während des Aufnahmeprozesses des UAV bewegt sich die Kamera mit hoher Geschwindigkeit relativ zum Boden. Da dabei Fahrzeugvibrationen und Luftströmungsstörungen auftreten, verschwimmen die UAV-Bilder, was die Bildqualität verringert und die Identifikation von Details erschwert. Um diesen entgegenzuwirken werden im Preprocessing hauptsächlich zwei Methoden verwendet: Die Histogrammäqualisation verbessert den Bildkontrast und die Entfernung des Rauschens, wobei die Form des Reisblattes markanter und klarer wird, was wiederum dem CapsNet ermöglicht die gesamten Merkmale des Reisbildes zu extrahieren. Der Superpixel-Algorithmus (hier Simple Linear Iterative Clustering kurz SLIC [Ach+10]) unterteilt das Bild in Superpixel um den Rechenaufwand der nachfolgenden Verarbeitung (CapsNet) zu reduzieren. Das Graustufenbild der Histogrammäqualisation und das Superpixel-Segmentierungsergebnis

wird dann als Input für das, in Abbildung 3.54 dargestellte, CapsNet verwendet. Die Autoren erläutern hierbei, dass beide *Conv1-Layer* einen Input der Größe 21x21x1 erhalten. Wie genau der Input zu diesen Größen skaliert wird, findet keine Erwähnung [Li+18b].

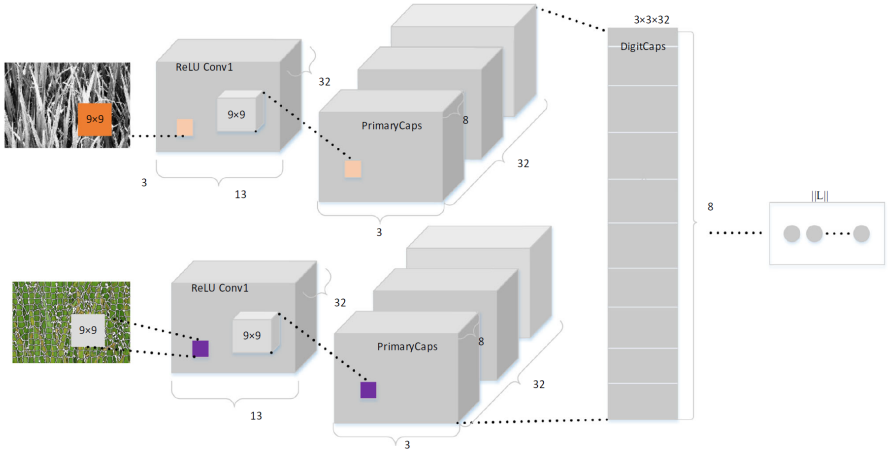


Abbildung 3.54: Architektur des Capsule Networks für Reiserkennung aus [Li+18b]

Abbildung 3.54 zeigt, dass das Graustufenbild der Histogrammäqualisation und das Superpixel-Segmentierungsergebnis in jeweils ein Teil-CapsNets gegeben werden. Einen ClassCaps-Layer (im Paper etwas unpassend als *DigitClass* bezeichnet) vereinigt letztlich die Ergebnisse der beiden Teil-CapsNets [Li+18b]. Auf die Hyperparameter des Modells wird nicht weiter eingegangen, da hier der Fokus auf der Haupterkenntnis, sprich dem Preprocessing und der Architektur, liegen soll. Wichtig war hier, dass durch mehrfache Überlappung von Reis entstandene Identifikationsprobleme zu lösen. Der ClassCaps-Layer wird verwendet um zu bestimmen ob es sich um Reis handelt und im welchen Wachstumsstadium dieser sich befindet [Li+18b]. Genau genommen kommt hier das in Abschnitt 3.2.4 angesprochene „Crowding“ ins Spiel, jedoch klassifiziert das Netz nicht einzelne Reis-Halme, sondern das gesamte Eingabebild, weshalb das CapsNet Features vermutlich nicht streng in einzelne Halme unterteilt. Der Trainingsprozess an sich wird von den Autoren nicht weiter beschrieben und die Architektur ähnelt zu stark der originalen um das „Crowding“ zu vermeiden. Weiter bleibt unklar ob die geringe Anzahl an Bildern für Overfitting sorgten und wie dieses, falls es auftrat, verhindert wurde. Auch die gelernten Features wurden nicht näher untersucht, aber die Vergleichsergebnisse mit einem CNN und SVM wurden zu Gunsten des CapsNet, jedoch leider zahlenlos, in folgender Tabelle dargestellt [Li+18b]:

Method	High speed	Overlapping image
SVM	Poor	–
CNNs	Well	Poor
CapsNet	Well	Well

Abbildung 3.55: Vergleich des Capsule Networks mit CNN und SVM aus [Li+18b]

3.8.4 HSI-CapsNet

Hyperspektrale Bilder (engl. Hyperspectral Images kurz HSI) werden von Flugzeugen mit speziellen Fernsensoren aufgezeichnet und von Bodenobjekten reflektierende, spektrale Daten für einem bestimmten Bereich der Erde gesammelt. Die wichtigste Eigenschaft des HSI stellt die Kombination von Bildgebungs- und Spektraldetektions-Techniken dar. Während der Abbildung der räumlichen Merkmale des Ziels werden Dutzende oder sogar Hunderte schmaler Bänder für jedes räumliche Pixel gestreut, um eine kontinuierliche spektrale Abdeckung zu erhalten. So werden Positions- und Verteilungsinformation des Ziels sowie das Spektrum (die Reflexionsintensität jedes Pixels in verschiedenen Wellenlängenbändern) erhalten. Hyperspektrale Bilder besitzen daher eine Fülle von Informationen, weshalb diese in vielen Gebieten verwendet werden, wie beispielsweise in geologischen und hydrologischen Wissenschaften oder in präziser Landwirtschaft. Die damit geformten Daten können visuell als 3D-Datenblock beschrieben werden; Abbildung 3.56 (a) zeigt ein Beispiel [Luo+18].

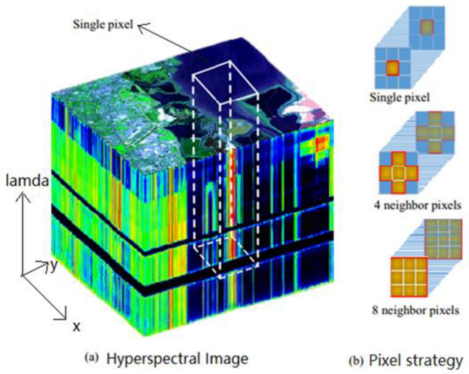


Abbildung 3.56: Links: die Rohdaten von HSI (a); Rechts: Würfeldaten mit verschiedenen Nachbarschaften (b) aus [Luo+18]

In Abbildung 3.56 (a) stellen x und y zweidimensionale, planare Pixelinformations-Koordinatenachsen dar und die dritte Dimension (λ-Achse) eine Wellenlängeninformations-Achse. Einzelne Pixel mit ihren Spektrumsbändern werden als Kategorien für das Training gelabelt [Luo+18]. Die Korrelation zwischen der Nachbarschaft eines Pixels und sich selbst ist für Pixel mit gleichen oder ähnliche Eigenschaften hoch. Leng et al. [Len+16] schlugen daher einen Ansatz vor,

einen Spektralwürfel mit verschiedenen räumlicher Strategien zu extrahieren. Dabei wurden verschiedene, in Abbildung 3.56 (*b*) dargestellte Nachbarschaften (Einzelpixel, 4rer, und 8ter) untersucht, wobei das Pixel im Center zu klassifizieren ist (z.B. Klasse Wald oder Gebäude). Da die 8ter-Nachbarschaft sich beim Extrahieren von spektral-räumlichen Merkmalen als beste Wahl herausstellte und sich die Klassifizierungsleistung dadurch verbesserte, wurden diese als Eingabe für das Modell verwendet [Luo+18].

Das HSI-CapsNet verzeichnet ebenfalls die gleiche Architektur wie das originale CapsNet aus Abschnitt 3.3, jedoch wurden einige Hyperparameter geändert: Der erste conv. Layer besitzt 96 1x24 Filter, ein Stride von eins und eine Sigmoid-Aktivierung (anstelle ReLu). Der nächste PrimaryCaps-Layer besitzt zwölf Kanäle von 8D Capsules. Dort variieren die Filtergrößen je Datensatz damit dieser Layer immer ein Ausgabe-Gitter der Größe [12, 5, 5] aufweist. Jede Ausgabe ist hierbei der 8D-Vektor einer Capsule. Der letzte Layer (ClassCaps) besitzt eine 32D-Ausgabe (anstelle 16D) für jede HSI-Pixelklasse und die Inputs werden mit drei fc Layern rekonstruiert (keine genauere Angabe vorhanden). Die Summe der quadratischen Abstände zum Input wird skaliert mit 0.005 zum Loss addiert. Alle Experimente wurden mit einer Lernrate von 0.1, einer Decay-Rate von 0.09 und einer Batch-Size von 100 durchgeführt [Luo+18].

Method / Datasets	CCS	HSI-CapsNet	HSI-CNN
KSC	0.9871	0.9515 ± 0.0139	**0.9928 ± 0.0040**
IP	0.9781	0.8949 ± 0.0073	**0.9909 ± 0.0022**
PU	0.9945	0.9605 ± 0.0017	**0.9952 ± 0.0015**
SA	0.9504	0.9575 ± 0.0063	**0.9895 ± 0.0006**

Abbildung 3.57: Gesamtgenauigkeit der Modelle für vier nicht weiter beschriebene HSI-Datensätze: Kennedy Space Center (KSC), Indian Pines (IP), Pavia University scene (PU) und Salinas scene (SA). Aus [Luo+18]

Abbildung 3.57 zeigt, dass die HSI-CapsNet Performance geringer ausfällt als die des HSI-CNNs. Die Autoren vermuten, dass vielleicht das HSI-CapsNet die Beziehung zwischen verschiedenen Bändern, die in einem Pixel weit voneinander entfernt sind, nicht gelernt hat [Luo+18]. Die Architektur des 7-Layer HSI-CNNs wird hier nicht weiter erläutert, jedoch wurde dieses u.a. durch einen Reshape-Layer, der nach den ersten beiden conv. Layern die 1D-Eingabe in 2D umwandelt, erweitert [Luo+18, S. 3]. Interessant wäre diesen Reshape auch für das CapsNet durchzuführen, denn dieser hilft laut den Autoren, die in den Originaldaten verborgenen spektralen und räumlichen Informationen vollständiger zu nutzen [Luo+18, S. 3]. Da liegt es nahe, dass solch ein Reshape-Layer auch einem CapsNet von Nutzen sein könnte.

3.8.5 Verkehrszeichenerkennung mit Capsule Networks

Amara Dinesh Kumar et al. [Kum18] wandten das original CapsNets aus Abschnitt 3.3 auf den German Traffic Sign Recognition Benchmark Datensatz (GTSRB) [Sta+12] an. Das Vorgehen war hierbei analog zum Original, allerdings besaß der ClassCaps-Layer 43 32D-Capsules, wobei jede Capsule ein Verkehrszeichen repräsentierte. Weiter fand im conv. Layer ein Dropout von 0.7 Verwendung. Der Datensatz bestand aus $51\,840$ 32x32 Bilder von Verkehrszeichen auf welchem mit einer Batch-Size von 50 eine Genauigkeit von 97.6% erreicht wurde. Als Vergleich wurde u.a. die beste menschliche Performance angegeben, jedoch ist hier die Abbildung im Paper mit 99.46% und der Text mit 98.8% widersprüchlich. Wird der Quelle gefolgt so beträgt die durchschnittliche menschliche Performance 98.84%, wobei 99.22% die Performance des besten Individuums darstellt [Sta+12]. Ein Komitee von CNNs erreicht auf dem Datensatz 99.46% [Mat+13]. Werden nun die verschiedenen Werte miteinander verglichen, so erreicht ein einzelnes herkömmliches CapsNet keine Spitzenperformance, allerdings könnte durch Einsetzten verschiedener Techniken die Leistung womöglich noch gesteigert werden, da jedes Spitzenmodell aus [Mat+13] ein weit komplexeres Vorgehen nutzt. Als Beispiel hierfür kann der in Abschnitt 3.3.5 erwähnte Performancegewinn durch das Nutzen von Ensembles bei der Klassifizierung von Cifar10 herangezogen werden.

3.8.6 FACSCaps: Pose-unabhängiges Facial-Action-Coding mit Capsules

FACSCaps ist ein CapsNet, welches dazu trainiert wurde mulit-view und mulit-label Facial-Action-Units zu erkennen und dabei auf neue Blickwinkeln zu generalisieren. Facial-Action-Units, kurz AUs, können in ihrer Kombination fast alle möglichen Gesichtsausdrücke beschreiben [OEJC18]. So sind beispielsweise beim Ausdruck der Freude AU6 (angehobenen Wange) und AU12 (Anheben der Mundwinkel) aktiv. Um Verwirrung zu vermeiden muss erwähnt werden, dass im Abstract des Artikels von Matrix Capsules die Rede ist, jedoch beschreiben und verwenden die Autoren im weiteren Verlauf original CapsNets. Das CapsNet ähnelt stark dem Original und unterscheidet sich hierbei wie folgt: Der conv. Layer verwendet einen Stride von vier anstelle von eins um das 128x128 Eingabebild eines Gesichtes mit einer bestimmten Emotion zu verkleinern. Anstelle von 32 Primary-Capsules wurden vier verwendet und der ClassCaps-Layer bestand aus zehn 8D-AUCaps für jede ausgewählte AU des Datensatzes (das gesamte Facial Action Coding System enthält 46 AUs). Der Aufbau des Rekonstuktions-Modul blieb ebenfalls unverändert, da dieses jedoch eine 128x128 Eingabe rekonstruiert enthält der letzte fc Layer $16\,384$ Neuronen [OEJC18], was die Parameteranzahl des Moduls auf $56\times512+512+512\times1024+1024+1024\times16\,384+16\,384 = 17\,348\,096$ stark erhöht. Die Größe des Rekonstruktions-Moduls beträgt somit abgerundet das 19-fache des eigentlichen, restlichen CapsNets. Diese Explosion der Parameter im Rekonstruktions-Modul wird in Abschnitt 4.4 des

praktischen Teils der Arbeit etwas genauer betrachtet. Der Balance-Faktor für den Loss wurde von 0.005 auf 0.0005 erniedrigt und der Decay-Rate betrug 0.9 bei der Standard Lernrate von 0.001. Während des Trainings wurden AU-Capsules so maskiert, dass nur die Capsules, deren AUs im Bild vorhanden sind, zur Rekonstruktion des Bilds verwendet wurden [OEJC18].
Die Auswertung von FACSCaps fand mit dem FERA 2017-Gesichtsausdrucksdatensatz statt, der spontane Gesichtsausdrücke in einer Vielzahl von Kopforientierungen umfasst. FACSCaps übertrifft hierbei State-of-the-Art CNNs und sogar deren Erweiterungen mit zeitlichen Modulen wie LSTMs [OEJC18].

Da davon ausgegangen wird, dass Capsules ihre internen Repräsentationen unabhängig vom Blickwinkel lernen, ist zu erwarten, dass das Modell AUs in einem Blickwinkel erkennt, welche zuvor nicht gesehen wurden. Um dies zu überprüfen, führten die Autoren Cross-Pose-Experimente durch, bei denen das Modell mit acht der neun Posen des Trainingssatzes trainiert und mit der verbleibenden Pose getestet wurde. Dabei wurde angenommen und gezeigt, dass die Erkennung von AUs in nicht trainierten, extremeren Posen wie Pose 1 (-40 Grad Yaw, -40 Grad Pitch) oder Pose 3 (0 Grad Yaw, -40 Grad Pitch) schwierig ist, da diese nicht so einfach von anderen Posen interpoliert werden können. Andererseits sind Posen die sich in der Mitte von zwei Posen befinden, wie Pose 2 (-20 Grad Yaw, -40 Grad Abstand) unter Verwendung von Pose 1 (-40 Grad Yaw, -40 Grad Pitch) und Pose 3 (0 Grad Yaw, -40 Grad Pitch) einfacher zu interpolieren. Ein Vergleich mit den CNNs wurde nicht durchgeführt, da diese in der bislangen Literatur keine Cross-Pose-Testergebnisse aufweisen [OEJC18].

Um die gelernte Repräsentation der AUCaps zu verstehen wurden analog zu Abschnitt 3.3.5 Werte des Aktivitätsvektor leicht abgeändert und die Bilder in Abbildung 3.58 daraus rekonstruiert:

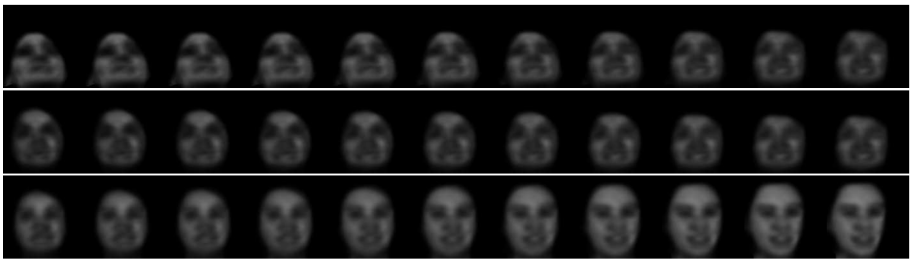


Abbildung 3.58: Synthetisierte Bilder, erhalten durch Abänderung einzelner Dimensionen des Aktivitätsvektors von AU12 aus [OEJC18]

Abbildung 3.58 stellt als Beispiel die Veränderung einiger Werte an AU12 (Anheben der Mundwinkel wie beim Lächeln oder Lachen) dar. In der ersten Zeile ändert sich die Pose von 1 zu 3 in FERA2017, und der Mund öffnet sich, wenn die Intensität von AU12 zunimmt. Gesichter

in der zweiten Reihe von AU12 sind hauptsächlich frontal mit zunehmender Lach-Intensität. In der dritten Reihe von AU12 ändert sich die Größe des Kopfes und des Yaw-Winkels. Ähnliche Ergebnisse wurden hierbei auch für andere AUs dargestellt, jedoch führen beispielsweise Änderungen an Werten von AU7 (Anspannen des unteren eventuell auch des oberen Augenlids) zu keinen signifikanten Änderungen in der Rekonstruktion [OEJC18].

Abbildung 3.59: Okklusionssensitivitätskarten überlagern mit der Eingabe für die Posen 1-9 von AU7 und AU12 aus [OEJC18]

Weiter wurden Occlusion Sensitivity Maps (M. D. Zeiler et al. [ZF13]) für verschiedene Posen und AUs erstellt. Das Verfahren wird hier nicht erläutert, jedoch kann aus den Maps geschlossen werden, dass die FACSCaps-Architektur korrekt lernt, wo im Bild welche AU zu finden ist. Da der Yaw-Winkel in diesem Datensatz nur in einer Richtung variiert, sind die Karten im Allgemeinen nicht symmetrisch. Abbildung 3.59 zeigt hierbei, dass die Capsule nach einem „Lachen“ (AU12) in Regionen von Mund- und Lippenecken sucht. Da AU7 nicht zu einer offensichtlichen Änderung der Rekonstruktion führte, konnte das Modell nicht lernen, wo sich AU7 in der Eingabe befindet, weshalb dort Karten uneindeutig sind. Da jedoch das Endergebnis für AU7 hoch ist, kann gefolgert werden, dass das Modell bei der Klassifizierung von AU7 andere visuelle Änderungen im Gesicht berücksichtigt, die durch AUs verursacht werden, und die mit AU7 zusammen auftreten [OEJC18].

Zusammenfassend folgern die Autoren, dass die Verwendung von CapsNets für die AU-Detektion vielversprechende Ergebnisse liefert und erhöhte Interpretierbarkeit mit sich bringt. Zukünftig könnte beispielsweise die Architektur für eine AU-Intensitätsschätzung (wie aktiv ist die AU) erweitert werden. Darüber hinaus könnte untersucht werden, ob ein mit einem großen Datensatz von Gesichtern verschiedener Blickwinkel trainiertes CapsNet zu besseren Ergebnissen führt, als anderen State-of-the-Art-Architekturen [OEJC18].

3.9 Weiteres NLP mit Capsule Networks

Das RNN-Capsule Network aus Abschnitt 3.5.3 führte bereits NLP (Natural Language Processing) durch. Hier wird noch ein weiteres Modell aus dieser Kategorie vorgestellt. Weitere gefundene Paper die CapsNets in diesem Bereich behandeln werden aus Platzgründen in der

Tabelle 6.6 des Appendix zusammengefasst.

Die Such-Personalisierung zielt darauf ab, Suchergebnisse für jeden spezifischen Benutzer basierend auf seinen persönlichen Interessen und Präferenzen (dem Benutzerprofil) anzupassen. Neuere Forschungsansätze modellieren hierfür die mögliche 3-Wege-Beziehung zwischen der gesendeten Anfrage, dem Benutzer und den Suchergebnissen (dem Dokument). Anschließend wird diese Beziehung verwendet, um die Suchergebnisse für den Benutzer zu personalisieren. Nguyen et al. [Ngu+18] stellen hierzu ein neuartiges auf CapsNets basierendes Embedding-Modell vor, welches die 3-Wege-Beziehungen für die Such-Personalisierung modelliert.
In diesem Modell wird der Benutzer, die gesendete Abfrage und das zurückgegebenes Dokument durch jeweils einen Vektor im selben Vektorraum eingebettet; das Embedding-Verfahren wird hier nicht weiter erläutert. Die 3-Wege-Beziehung wird als ein Tripel (Abfrage, Benutzer, Dokument) beschrieben, welches als 3-Spalten-Matrix modelliert wird, welche die drei Embedding-Vektoren enthält. Danach wird die Matrix in das Modell eingespeist, um die von einem Basis-Ranker zurückgegebenen Suchergebnisse neu zu ordnen.
Die Architektur des Modells umfasst drei Hauptschritte: Im ersten Schritt wird ein conv. Layer verwendet, um Feature-Maps aus dem Embedding-Tripel zu extrahieren. Im nächsten Schritt wird jede Feature-Map als eine Capsule betrachtet und das originale Routing zu nur einer einzigen finalen Capsule wird durchgeführt. Schließlich ist die Länge der Vektorausgabe der letzten Capsule die gemessene Punktzahl für das Tripel (Abfrage, Benutzer, Dokument) [Ngu+18], womit die Ergebnisse neu geordnet werden können. Eine Orphan-Kategorie oder ein Leak-Routing wurden nicht erwähnt und auch hier besteht der letzte Caps-Layer aus nur einer Capsule, womit der einzige Coupling-Koeffizient jeder unteren Capsule immer eins ist und somit die verwendeten drei Routing-Iterationen überflüssig werden. Durch eine Orphan-Kategorie oder Leak-Routing würde den unteren Capsules die Wahl freigestellt, ob diese ihre Werte im Bezug auf die anderen Capsules im Layer als wichtig betrachten, was vermutlich das Modell weiter verbessern würde.
Ungeachtet dessen zeigen die experimentellen Ergebnisse, dass das Modell die starken Baselines auf den Abfrageprotokollen einer kommerziellen Web-Suchmaschine übertrifft. Ein Grund dafür ist, dass das Modell nicht nur reichere relationale Merkmale innerhalb des Triple erfassen kann, indem es eine tiefere Architektur verwendet, sondern das dieses zusätzlich die Übergangsbeziehungen zwischen Einbettungen von Benutzeranfragen und relevanten Dokumenten für Benutzerprofile verallgemeinert [Ngu+18]. Hier muss jedoch erwähnt werden, dass das verwendete Modell stark dem Original entspricht und nicht wirklich tief ist, die Baselines waren hierbei nur sehr flach. Ein tiefes Deep Learning Modell, wie ein CNN, befand sich bei dieser speziellen Aufgabe nicht unter den Vergleichsmodellen.

3.10 Auswertung und Erklärbarkeit von Capsule Networks

Hier werden Artikel erläutert, deren Fokus darauf liegt, CapsNet-Modelle anhand verschiedener Verfahren auszuwerten, oder CapsNets und deren typisches Verhalten genauer zu erklären (z.B. den Pfad des Routings).

3.10.1 Capsule Network Performance für komplexe Daten

Edgar Xi et al. [EX17] testeten das originale CapsNet aus Abschnitt 3.3 auf „komplexere Daten“, wobei im Kontext dieses Papers von Cifar10 mit seinen 32x32 Bildern die Rede ist. Vorweg muss erwähnt werden, dass im originalen CapsNet Paper, wie in Abschnitt 3.3.5 erläutert, ein CapsNet bereits auf Cifar10 trainiert und ein Test-Error von 10.6% erzielt wurde. Dieser entspricht ungefähr dem Test-Error von CNNs, als diese das erste Mal auf Cifar10 angewandt wurden. Dieses Paper erhält allerdings Ergebnisse einiger verschiedener Modelle, wobei jedoch keines den Test-Error von 10.6% erreicht. Folgende Gründe könnten dazu beigetragen haben: 1. Die Rekonstruktionen im Paper sind Graustufenbilder, Cifar10 besteht jedoch aus RGB-Bildern, evtl. konnte das CapsNet des original Papers die Farbinformation im Aktivitätsvektor nutzen. 2. Im originalen Paper werden beim Training und Test zufällig 24x24 Stücke aus den Bildern geschnitten, in diesem Paper werden keine präzisen Angaben zum Trainingsprozess gegeben. Womöglich wurde daher mit den 32x32 Bildern getestet und trainiert. 3. Das im original Paper beschriebene Modell wurde nicht verwendet. 4. Die Rechenperformance welche zur Verfügung stand war limitiert, weshalb die komplexeren Modelle nicht vollständig trainiert werden konnten. Dennoch sind die getesteten Modelle sehr vielfältig und die Ergebnisse in Abbildung 3.60 zeigen den Einfluss von Änderungen an der CapsNet-Architektur auf den Test-Error.

	Validation Accuracy	
Models	25 Epochs	50 Epochs
MNIST Model Baseline	67.51%	68.93%
64 Capsule Layers	60.54%	64.67%
4-Model Ensemble (4 Ensemble)	68.97%	70.78%
2-Convolution Layers (2 Conv)	68.14%	69.34%
4 Ensemble + 2 Conv	70.34%	71.50%
7 Ensemble + 2 Conv	70.50%	_____
4 Ensemble + 2 Conv + 0.0001 Reconstruction Scaling	69.21%	_____
Stack Additional Capsule Layer	10.11%	_____

Abbildung 3.60: Ergebnisse verschiedener dem Original ähnelnden CapsNet-Modelle auf Cifar10 aus [EX17]

In Abbildung 3.60 ist zu erkennen, dass die beste Genauigkeit erreicht wurde, indem ein Essemble Modell aus vier CapsNets mir zwei conv. Layern genutzt wurde. Dies stellt laut den Autoren eine 2.57% Verbesserung gegenüber dem Baseline-Modell im original Paper da. Hierbei handelt es sich jedoch nicht um das vorgestellte 7-Essemble Modell mit 10.6% Test-Error, sondern

um das in Abschnitt 3.3.5 beschriebene, auf MNIST angewendete einzelne Modell [EX17]. Interessant an diesem Ergebniss ist, dass der zweite conv. Layer im Gegenzug zu der Entwicklung der DCNets in Abschnitt 3.7.3 Verbesserungen bringt. Um den Grund hierfür zu finden, müssten die Modelle einheitlich getestet werden.
Das 7-Essembel Modell dieses Papers ist am vielversprechendsten, jedoch konnte dieses durch limitierte Rechenleistung nicht 50 Epochen lang trainiert werden. Beim 4-Modell Essemble bringt ein zweiter conv. Layer ebenfalls bessere Genauigkeit.
Am interessantesten ist das Ergebnis eines Modells mit einem weiteren CapsLayer (es ist nicht eindeutig welcher, vermutlich ein conv. CapsLayer). Hierbei wurde eine sehr geringe Genauigkeit von 10.11% erreicht, jedoch wird beispielsweise in den Abschnitten 3.5.4 und 3.5.5 eine höhere Genauigkeiten durch das Hinzufügen mehrere Capsule-Layer erzielt. In Abschnitt 4.4.2 des praktischen Teils der Arbeit werden ebenfalls einige conv. CapsLayer gestapelt und gute Ergebnisse damit erzielt. Ein Unterschied ist, dass das hier vorgestellte Netz mit mehreren CapsLayer eine eigene Aktivierungsfunktion nutzt. Allerdings wird bei genauerer Betrachtung erkannt, dass diese äquivalent zur Gleichung 3.30 aus Abschnitt 3.7.2 ist, wenn $a = b = 1$, weshalb dies vermutlich nicht der Grund für die niedrige Genauigkeit darstellt. Außerdem lässt sich aus dem Paper nicht eindeutig erschließen, ob die Aktivierungsfunktion auch bei den anderen Modellen Verwendung fand und da weiter nicht beschrieben wurde wie genau der zweite CapsLayer eingefügt wurde, bleibt unklar weshalb diese niedrige Performance zustande kam.

3.10.2 Verbesserte Erklärbarkeit von Capsule Networks: Relevance-Path durch Agreement

In kritischen Domänen wie medizinische Applikationen oder autonomes Fahren ist keine einzige falsche Entscheidung akzeptierbar, da diese katastrophale Auswirkungen haben kann. Um die Zuverlässigkeit eines Machine-Learning-Modells zu gewährleisten, ist es daher von entscheidender Bedeutung rationale Gründe für die Entscheidungen dieser Modelle analysieren und verstehen zu können. Mit anderen Worten, die Black-Box muss geöffnet werden. Dies bringt viele Vorteile mit sich, so können beispielsweise die für den Fehler verantwortlichen Module gefunden werden. Aus diesen Gründen wurden Strukturen und Verhaltensweisen der CapsNets analysiert und mögliche Erklärbarkeitseigenschaften aufgezeigt. Weiter wurde auf die Möglichkeit verwiesen, dass Deep Learning Architekturen durch das Einfügen von Capsules in verschiedenen Layern anstelle von conv. Layern erklärbarer gemacht werden können. Insbesondere wurde festgestellt, dass CapsNets automatisch ein Verification Framework aus Xiaowei Huang et al. [Hua+16] formen [SMP18]. In solch einem Verification Framework wird angenommen, dass es in jeder Eingabe eine Region gibt, die spezifisch die Klassenkategorie bestimmt. Wenn eine Vorhersage diese Kategorie anzeigt, sollte ihre Eingabe wichtige Punkte dieser Regi-

on enthalten. Die Analyse dieser Regionen soll dann in tiefere Layer propagiert werden können [Hua+16, S. 23]. CapsNets lernen demnach Regions of Interest, welche die Klassenkategorie bestimmen und somit die Vertrauenswürdigkeit und Erklärbarkeit von tiefen Netzen verbessern [SMP18]. Diese sind beispielsweise in den Occlusion Sensitivity Maps aus Abschnitt 3.8.6 für einzelne Capsules zu erkennen. Weiter erstellen CapsNets von Natur aus den Relevance-Path (eingeführt in Ribeiro et al. [RSG16]), welcher beim Propagieren der Analyse der Regionen in tiefere Layer hilft und als Relevance-Path-by-Agreement bezeichnet wird, da dieser ein Nebenprodukt des Routing-by-Agreement darstellt. Capsules formen intrinsisch diesen Pfad, da die Coupling-Koeffizienten von nicht zusammenhängenden Capsules gegen null gehen, während die Coupling-Koeffizienten zusammenhängender Capsules zunehmen. Dadurch ist die Notwendigkeit eines Rückwärtsprozesses zur Konstruktion des Relevance-Path beseitigt. Der Grund, dass der Relevance-Path-by-Agreement existiert, rührt somit daher, dass CapsNets dynamisches Routing anstelle von herkömmlichen Pooling-Methoden verwenden. Für den Pfad können dann zwei Variablem verwendet werden: die Aktivierungswahrscheinlichkeit, zur Erklärung der Existenz eines Features und die Werte des Ausgabevektors, welche die Konsistenz zwischen den Layern erklärt. Beispielsweise ist im unteren Layer die Aktivierungswahrscheinlichkeit einzelner Gesichts-Teile sehr hoch, die Netzwerkentscheidung besteht jedoch darin, dass in der Eingabe kein Gesicht vorhanden ist, da die Aktivierungswahrscheinlichkeit der Gesichts-Capsule im oberen Layer relativ gering ist. Dies kann durch die Inkonsistenz zwischen den Werten des Ausgabevektors der unteren Capsules erklärt werden, da die Votes dieser kein Gesicht ergeben. Weiter wurde das Erklären von Klassifizierungsfehlern anhand der Rekonstruktionen analog zum Abschnitt 3.3.5 erläutert [SMP18].

3.10.3 Weitere Auswertungen

In [Hoc18] wurde festgestellt, dass das Hinzufügen eines CapsNets mit einem einzigen Capsule-Layer aus zehn 16D-Capsules mit eindimensionalen Convolutions zu den meisten bekannten sequenziellen Modellen (wie LSTM oder GRU) die Genauigkeit steigern, während die Rechenzeit nicht erheblich erhöht wird. Weiter wird erwähnt das die Forschung zu diesem Paper noch im Gange ist und weitere Ideen zum Nutzen von RNNs mit CapsNets untersucht werden. So wird beispielsweise theoretische diskutiert, wie das EM-Routing mit RNNs kombiniert werden kann, was hier nicht weiter erläutert wird.

In [RM18] evaluieren Rinat Mukhometzianov et al. die Leistung des originalen CapsNets im Vergleich zu drei bekannten Klassifikatoren (Fisherfaces, LeNet und ResNet). Hierbei wurde die Klassifizierungsgenauigkeit für vier Datensätze getestet. Darunter Bilder von Gesichtern, Verkehrszeichen und alltäglichen Objekten. Die Evaluierungsergebnisse zeigen, dass selbst für

einfache Architekturen das Training des CapsNet erhebliche Rechenressourcen erfordert und die Klassifizierungsleistung unter die durchschnittlichen Genauigkeitswerte der anderen drei Klassifikatoren fällt. Weiter wird jedoch argumentiert, dass CapsNets eine vielversprechende neue Technik für die Bildklassifizierung zu sein scheinen, und weitere Experimente mit robusteren Rechenressourcen und verfeinerten CapsNet-Architekturen zu besseren Ergebnissen führen könnten. Für alle Experimente wurde die Struktur der originalen Architektur aus Abschnitt 3.3.3 verwendet und keine Essemble-Techniken angewandt.

In [YL18] stellen Yujian Li et al. ein einheitliches, mathematisches Framework für Deep Learning mit CapsNets vor, welches die Beschreibung von existierenden tiefen, neuronale Netzen vereinfachen und eine theoretische Basis für Grafikdesign und Programmiertechniken für Deep Learning Modelle darstellen könnte. Laut den Autoren hätte dies für den Fortschritt des Deep Learning große Bedeutung. In Zukunft soll weiter ein industrieller Standard definiert und eine grafische Plattform für die Fortschritte in CapsNets implementiert werden, gefolgt von einer Erweiterung für RNNs.

David Rawlinson et al. zeigen in [RAK18], dass das unüberwachte Lernen von Capsules ausschließlich über den Rekonstuktion-Loss und ohne Label-Masking des finalen Capsule-Layers zum Verlust der Äquivarianz und anderer wünschenswerten Capsule-Qualitäten führt. Dies impliziert, dass überwachtes Lernen nicht auf tiefere CapsNets verlustlos angewandt werden kann [RAK18], vermutlich, da die niedrigeren Layer keinem Masking unterliegen und dies sich mit zunehmender Tiefe auf die Capsule-Qualitäten auswirkt. Untersuchungen dazu sollen folgen; ab wann ein CapsNet zu tief ist bleibt ungewiss. Der praktische Teil der Arbeit trainiert jedoch erfolgreich ein CapsNet mit vier Capsule-Layern.
Im weiteren Verlauf stellen die Autoren unüberwacht (nur durch den Rekonstuktion-Loss) mittels unkontrolliertem Sparsing der Capsule-Aktivitäten diese Qualitäten wieder her. Außerdem scheint diese Methode besser zu generalisieren als überwachtes Masking, was potentiell tiefere CapsNets ermöglicht. Um dies aufzuzeigen wurde ein spärliches, unüberwachtes CapsNet mit gleicher Architektur wie das Original trainiert und die Klassifizierungsrate auf MNIST und affNIST getestet. Dabei wurden die Capsule-Aktivitäten vom letzten Capsule-Layer des unüberwachten Netzes als encodierter Input einer SVM verwendet, welche überwacht trainiert wurde. Bei affNIST konnte somit die Genauigkeit von 79% auf 89% gegenüber dem originalen CapsNets gesteigert werden. Die bereits sehr hohe Klassifizierungsrate von MNIST wurde nicht weiter durch den Wechsel von überwachten zu spärlichem unüberwachten Lernen verbessert. Außerdem wurde darauf aufmerksam gemacht, dass bei originalen Capsules und Matrix Capsules zwar das Routing an sich unüberwacht ist und nicht direkt Gewichte trainiert werden, jedoch

das Ergebnis des überwachten Trainings durch das Anpassen der Capsule-Ausgaben über das Agreement beeinflusst. Das unüberwachte Lernen des Papers findet im praktischen Teil der Arbeit Verwendung und wird dort weiter erläutert.

3.11 Tabellarische Zusammenstellung der restlichen gefundenen wissenschaftlichen Literatur

Da die Literatur hinsichtlich CapsNets im Laufe der Arbeit förmlich explodierte, wurden die restlichen gelesenen Paper in Tabelle 6.1 des Appendix zusammengestellt und mit Kategorien, einer Bewertung der CapsNets von den Autoren und einigen Stichwörtern hinsichtlich der Erkenntnisse versehen. Viele dieser Artikel behandeln Capsules in sehr interessanten Gebieten, wie beispielsweise One-Shot-Learning, konnten jedoch leider aus Platz- und Zeitgründen nicht weiter ausformuliert werden, weshalb diese dort Erwähnung finden. Da CapsNets noch sehr jung sind, handelt es sich hierbei trotz der Explosion an Publikationen - zusammen mit der obigen Literatur - um die gesamte Literatur, die bis zum 12.06.2018 unter anderem bei Springer, Google Scholar, arXiv, OpenReview und unter bester Kenntnis gefunden wurde. In der Zwischenzeit sind jedoch viele neue Erkenntnisse publiziert worden.

3.12 Capsule Networks im Vergleich mit CNNs

CNNs basieren auf der einfachen Tatsache, dass ein Bildverarbeitungssystem dasselbe Wissen an allen Stellen im Bild verwenden soll. Dies wird erreicht, indem die Gewichte der Feature-Detektoren so verknüpft werden, dass an einem Ort erlernte Features an anderen Orten verfügbar sind. Convolutionale Capsules erweitern das Teilen von Informationen zwischen verschiedenen Orte, um das Wissen über Teil-Ganzes-Beziehungen, welche vertraute Formen kennzeichnen [HSF18a, S. 1]. Anders formuliert stellen CNNs keine räumliche Hierarchie zwischen Objektteilen dar, weshalb diese, wenn die Existenz von Teilen erfüllt ist, zwar die Existenz eines ganzen Objekts ausgeben, jedoch dabei die räumliche Orientierung zwischen den einzelnen Teilen ignorieren, was schließlich zu False-Positives führen kann. Darüber hinaus besitzen CNNs nur sehr geringe Rotationsinvarianz, weshalb dasselbe Objekt, wenn dieses in einer anderen Orientierung betrachtet wird, als ein anderes Objekt erkannt werden kann. Dies führt zu False-Negatives, da die Pose hierbei in CNNs nicht unabhängig von der internen Repräsentation eines Objekts ist [OEJC18].
Aufgrund der skalaren und additiven Natur der Neuronen in CNNs sind dort Neuronen in jedem gegebenen Layer eines Netzwerks ambivalent zu den räumlichen Beziehungen von Neuronen innerhalb ihres Kernels des vorherigen Layers und somit innerhalb ihres effektiven rezeptiven Feldes der gegebenen Eingabe. Dem hingegen formen Capsules über ihren Routing-

Algorithmus, welcher das Agreement zwischen Ausgabevektoren berücksichtigt, bedeutungsvolle Teil-Ganzes-Beziehungen zwischen den Layern, und somit widerspruchsfreie Zwischen-Layer-Beziehungen. So weisen CNNs trotz ihrer bemerkenswerten Flexibilität und Leistungsfähigkeit in einer Vielzahl von Computer-Bildverarbeitungsaufgaben ihre eigenen Schwächen auf [BKC15].
Jedoch ist genau diese hohe Flexibilität und Leistungsfähigkeit von CNN derzeit ausschlaggebend. Auch wenn in der für die Arbeit verwendeten Literatur das Feedback hinsichtlich Capsules größtenteils positiv ausfällt und diese erfolgreich in verschiedensten Domänen mit State-of-the-Art-Performance angewandt wurden und werden, ist dort bislang immer entweder die Eingabe klein bzw. verkleinert worden, oder das aktuelle Modell für größere Eingaben problemspezifisch erweitert worden. Diese problemspezifischen Erweiterungen sollen zwar oft auch allgemeine Vorteile (für andere Datensätze) bringen, wurden bislang jedoch meist sehr eingeschränkt getestet. Somit existiert derzeit nicht „das CapsNet“ welches auf beliebige Eingaben angewandt werden kann und sich als Standard etabliert hat. Natürlich werden auch CNNs fortgehend für verschiedene Aufgaben erweitert, jedoch existiert für diese ein Standardmodell: Das Modell, welches zum Einstieg gelehrt wird. Dieses Standardmodell ist darüber hinaus, und im Gegenzug zum originalen CapsNet sowie zu Matrix Capsules, hoch skalierbar und flexibel. So können einfache CNNs aus Performance-Sicht beispielsweise auf dem riesigen ImageNet-Datensatz trainiert werden, auch wenn ohne weitere Tricks die Ergebnisse nicht herausragend sind. Ein CapsNet kann hingegen zum Kenntnisstand dieser Arbeit nicht effizient auf ImageNet angewandt werden, so reichten nicht einmal eine Reihe leistungsstarker GPUs von Cloud-Anbietern dafür aus.

Trotz der fehlenden Skalierbarkeit und den daraus resultierenden Einbussen an Flexibilität - welche Schwächen darstellen die für den Erfolg von CapsNet unabdingbar ausgebessert werden müssen - muss erwähnt werden, dass der heutige, aktuelle Stand der CapsNet-Forschung mit dem Entwicklungsstand von CNNs um 1998 vergleichbar ist [RM18]. Dies bedeutet schlichtweg, dass CapsNets im Bereich des Deep Learning noch Neulinge darstellen und sich in einem frühen Forschungsstadium befinden, weshalb erst eine Vielzahl an Forschung, Experimenten und Tests durchgeführt werden müssen, um das volle Potenzial dieser Methode aufzeigen zu können. Hierbei zeigt die Literatur, dass die Deep Learning Community zuversichtlich zu sein scheint und allgemein erwartet wird, dass starke Capsule-Modelle entwickelt werden. Verschiedene in dieser Arbeit aufgezeigten Quellen führten hierbei bereits Schritte in Richtung Skalierbarkeit und Flexibilität durch, welche implizieren, dass effiziente CapsNets möglich sind. Weiter kann den CapsNets ihre Äquivarianz in Zukunft zugutekommen, da diese wie zu Beginn diskutiert den Capsules ermöglichen, ihre invariantes Wissen um Teil-Ganzes-Beziehungen zu

erweitern, welches ebenfalls an jedem Ort in der Eingabe zur Verfügung steht. Langfristig betrachtet könnte dies dafür sorgen, dass somit Rechenleistung eingespart werden kann. Jedoch ist hier der Routing-Algorithmus mit seinen Routing-Iterationen ausschlaggebend, bei welchem bislang drei Routing-Iterationen für vernünftige Ergebnisse vonnöten sind. Dies bedeutet kurz, dass der Feedforward-Pfad dreimal in einer Schleife ausgeführt werden muss und zusätzliche Performance für die Berechnung der Zuweisungen zwischen den Capsules in Anspruch genommen wird. Bei CNNs hingegen wird der Feedforward-Pfad i.d.R. nur einmal durchlaufen.
So bleibt abzuwarten, wie sich CapsNets aus Seiten der Performance entwickeln; womöglich findet sogar einen ähnlichen Ablauf wie bei CNNs statt, welche zu ihrem Beginn für die damalige Zeit eine zu hohe Rechenleistung und Datenmenge benötigten und somit in den Hintergrund gerieten. Als die Rechenleistung und die Datenmengen verfügbar wurden, erlebten diese ein schlagartiges Aufblühen. Ob den CapsNet ein ähnliches Schicksal blüht bleibt abzuwarten, dabei scheint eins jedoch sicher: zu wenige Daten werden nicht deren limitierender Faktor, da die Literatur zeigt, dass bereits das original CapsNet aufgrund typischer Capsule-Vorteile (wie das Modellieren der Teil-Ganzes-Beziehung) auf weniger Trainingsdaten oft besser abschneidet als herkömmliche CNNs.

Doch nicht nur die Performance ist ein wichtiger Faktor: So bringen Capsules im Vergleich mit CNNs sichtlich erhöhte Erklärbarkeit, was durch verschiedene Paper in Abschnitt 3.10 aufgezeigt wurde. Dieser Gewinn an Durchschaubarkeit und Determinierbarkeit ist von großer Bedeutung, vor allem in Anbetracht aktueller Entwicklungen im Bereich des Deep Learning, welche für kritischen Domänen, wie beispielsweise Medizin oder autonomes Fahren hoch durchschaubare Ergebnisse fordern. Eine Forderung, die für dieses Black-Box-reiche Gebiet nicht einfach zu erfüllen ist. So bringt beispielsweise die Interpretierung der Entscheidungen eines tiefen CNNs einen hohen Aufwand mit sich und fordert das Anwenden von komplexen Techniken der Feature-Visualisierung. Diese Techniken (siehe z.B. in Olah et al. [OMS17] für Feature-Visualisierung auf ImageNet) führen zwar zu einer erhöhten Erklärbarkeit, scheinen jedoch nicht an die Erklärungen von CapsNets heran zu kommen, welche die Modelle von Natur aus und ohne aufwändige Erweiterungen mit sich bringen. Wird dabei zusätzlich in Betracht gezogen, dass sich CapsNets in einer sehr frühen Entwicklungsphase befinden, so sind in Zukunft im Bereich der auf CapsNets basierenden Feature-Visualisierung interessante Ergebnisse zu erwarten.
Darüber hinaus stellen sich CNNs als sehr anfällig gegenüber White-Box Adversarial Attacks und der sogenannten „Fast-Gradient-Sign-Methode“ heraus [RM18]. Dies führt sogar soweit, dass diese in Gamaleldin et al. [GFE18] über Adversarial Attacks neu programmiert werden konnten: So wurden beispielsweise bekannte Inception- und ResNet-Modelle dazu gebracht, an-

stelle ImageNet-Klassen zu klassifizieren, Rechtecke im Bild zu zählen. Das Matrix-CapsNet hingegen zeigt gegen White-Box Adversarial Attacks hohe Resistenz (siehe Abschnitt 3.4.6) und könnte womöglich diese sicherheitstechnischen Nachteile überwinden [RM18]. Auch wenn dies ein Fortschritt darstellt, wurde bei einem komplexeren Black-Box-Angriff durch Generativ-Adversarial-Examples mittels eines CNNs keine bedeutende Verbesserung erreicht.

Zusammengefasst wird sich auch hier der in der Literatur stark vertretenen Meinung angeschlossen: CapsNets stellen ein vielversprechendes Modell dar, welches aktuell geforderte Verbesserungen hinsichtlich des Deep Learnig angeht und bereits in vielen Domänen mit State-of-the-Art-Performance erfolgreich angewandt wurde. CapsNets sind momentan jedoch genau das: vielversprechend. Daher sind diese - solange ihre Entwicklung nicht die Performancegrenzen überwindet und allgemein voranschreitet - nicht der neue, bessere „CNN-Ersatz“ und werden es vermutlich auch nie sein. Dies zeigt beispielsweise der Literaturteil der Arbeit: So wurden bereits einige Capsule-Modelle vorgestellt welche mit mehreren conv. (ähnlichen) Layern größere Eingaben verkleinern, damit das Modell effizient bleibt. Dies führte zu guten Ergebnissen, jedoch wird dabei das CNN nicht ersetzt - wovon so oft gesprochen wird - sondern die vorgestellten Modelle nutzen diese, was klar zeigt, dass CapsNets CNNs oder CNN-Vorgehensweisen inkorporieren. So ist z.B. der erste Layer von allen vorgestellten CapsNets immer ein CNN oder eine aus CNNs abgeleitetes Modell und die conv. Capsules nutzen ebenfalls conv. Strukturen. Auch wenn zukünftige effiziente Modelle, vielleicht den/die ersten CNN-Layer ersetzen, bleiben sehr wahrscheinlich bekannte CNN-Vorgehensweisen in den Capsule-Layern erhalten. So wird in Zukunft möglicherweise nicht mehr ein „CNN“ eingesetzt um ein Problem zu zu lösen, sondern ein „CapsNet“, jedoch wird dieses CapsNet mit hoher Wahrscheinlichkeit Strukturen die auf CNNs basieren in sich tragen, weshalb daher wohl nicht davon gesprochen werden sollte, dass CapsNets CNNs ersetzen, sondern das CapsNets auf CNNs aufbauen und diese verbessern.

4 Praktische Anwendung von Capsule Networks

Im praktischen Teil der Arbeit wird zuerst die Aufgabenstellung festgelegt, nach welcher eine kurze Beschreibung des Entwicklungsumfeldes sowie der Vorgehensweise folgt. Daraufhin werden jeweils die für die einzelnen Aufgaben entwickelten Modelle und deren Ergebnisse vorgestellt und abschließend diskutiert.

4.1 Aufgabenstellung

Wie der theoretische Teil der Arbeit an vielen Stellen zeigt, existieren zahlreiche offene Fragestellungen und nicht untersuchte Gebiete des Deep Learning hinsichtlich CapsNets, woraus eine Vielzahl von Aufgabenstellungen resultieren. So könnte beispielsweise ein GAN erstellt werden, dessen Diskriminator und Generator aus Capsules besteht; es könnte ein Routing-Algorithmus mit anderen Clustering-Verfahren entwickelt werden (interessant wäre ein SOM-Routing); verschiedene vorgestellte Routing-Algorithmen und Erweiterungen könnten breiter und tiefer untersucht und verglichen werden; Capsules für die Segmentierung mehrere Objekte (anstelle von einem) könnten erstellt werden; Capsules könnten auf Regressions-Aufgaben angewandt werden (wie z.B. Objektposition); es könnten neue Aktivierungsfunktionen entwickelt und evaluiert werden, verschiedene Capsule-Modelle verglichen werden, usw ...
In dieser Arbeit wurde sich basierend auf eigenen Ideen sowie auf der verbleibenden Zeit für drei konkrete Anwendungen entschieden: Zuerst soll die theoretische Erkenntnis, dass das Routing-by-Agreement in Layern mit einer original Capsule überflüssig ist, auch praktisch bewiesen werden. Dabei soll zusätzlich aufgezeigt werden, dass Strategien wie Leaky-Routing mögliche Lösungen hierfür darstellen.
Zweitens soll ein CapsNet dazu trainiert werden Emotionen von menschlichen Gesichtern zu erkennen. Diese Aufgabe unterscheidet sich von der Anwendung von FACSCaps aus Abschnitt 3.8.6, da das zu erstellende Netz nicht auf AUs arbeitet und Emotionen somit direkt und ohne Zwischenschritt aus der Eingabe liest. Außerdem wurden bei FACSCaps keine Datensätze für die direkte Erkennung von Emotionen getestet. Womöglich erlernt das zu entwickelnde CapsNet sogar automatisch AUs zu detektieren und mittels dieser Emotionen zu klassifizieren. Weiter sollen hier auch tiefere CapsNets getestet werden, was im Gegensatz zu einer Vielzahl von Klassifizierungen aus dem Theorieteil steht. Das Ziel hierbei ist nicht durch exzessive Parametersuche die maximale Klassifizierungsrate zu finden, sondern lediglich eine hohe. Basierend auf diesem Modell kann dann untersucht werden, welche Änderungen am Modell welche Auswirkungen mit sich bringen.
Zuletzt wird in der dritten Aufgabe dann das beste CapsNet aus Aufgabe zwei unüberwacht trainiert, sodass ein kompaktes, vielsagendes Encoding durch die Aktivitäten der Capsule in

der letzten Schicht entsteht. Auf solch einem Encoding kann folglich weiter gearbeitet werden; z.B. wird ein SVM für das Klassifizieren von Emotionen über das Encoding herangezogen, um zu testen wie eindeutig das vom CapsNet erstellte Encoding ist. Somit kann untersucht werden, ob das Verfahren zum unüberwachten Lernen von CapsNets aus Abschnitt 3.10.3 auch für Eingaben, deren ROIs (kurz für Region of Interest) nur einen kleinen Teil der Werte ausmacht, funktioniert. Hierbei wird vermutet, dass dieses Verfahren dabei keine guten Ergebnisse liefert, da die Capsules durch den fehlenden Klassifizierung-Loss nicht auf die ROIs hingewiesen wird. Als Basis für alle Aufgaben dient das originale CapsNet aus Abschnitt 3.3, da für Matrix Capsules und deren Erweiterungen zum Kenntnisstand dieser Arbeit keine vollständige und zugängliche Implementierung existiert. Ein Grund dafür ist u.a. der Annealing-Schedule der Matrix Capsules, welcher von den Autoren nicht explizit angegeben wurde. Für die originalen CapsNets hingegen ist sogar die originale Implementierung der Autoren zugänglich, was mögliche Third-Party-Implementierungsfehler eliminiert. Folglich wäre für das Testen eigener Ideen zuerst eine eigenständige Implementierung der Matrix Capsules unter fehlenden Informationen vonnöten gewesen, was vermutlich den gesamten praktischen Teil der Arbeit abgedeckt hätte. Daher wird aus der gegebenen Implementierung des originalen CapsNets eine kleine Bibliothek erstellt mit welcher die eigenen Capsule-Modelle implementiert werden.

4.2 Entwicklungsumfeld und Vorgehensweise

Die TensorFlow-1.6-Implementierung in Python 3.6 und das Training von kleinen Modellen auf kleinen Datensätzen findet auf einem HP Spectre x360 Convertible 15-bl1XX statt. Es handelt sich hierbei um das eigene Ultrabook, welches mit 16GB RAM, sowie einer NVIDIA GeForce MX150 mit 2048 MB GDDR5 Speicher und einem Kerntakt von 1468 - 1532 (Boost) MHz gerade so dieser Aufgabe gerecht wird. So wird beispielsweise der Coupling-Koeffizienten-Test aus Abschnitt 4.3 auf diesem ausgeführt. Parallel dazu werden die Implementierungen der komplexeren Modelle auf der Cloud trainiert.
Nach einer Recherche wurde sich für Google's Colaboratory (kurz Colab) [Col] als Cloud-Service entschieden. Colab ist eine auf Jupitery-Notebook [Jn] beruhende Web-Anwendung zum Erstellen und Teilen von Dokumenten (Notebooks), welche u.a. Live-Code enthalten.
In Sessions, die auf 12 Stunden begrenzt sind, wird gratis eine Tesla K80-GPU zur Verfügung gestellt, welche eine Single-Precision-Leistung von bis zu 8.73 TeraFLOPS aufweist und von welcher circa 12 GB der GPU-RAM über die Session hinweg verwendet werden kann. Da andere Anbieter für ähnlich starke GPUs oft per Stunde oder Monat abrechnen, ist die 12-Stunden-Begrenzung verkraftbar.
Der Workflow ist dann wie folgt: Zuerst wird auf dem Ultrabook der Code entwickelt und geprüft, welcher zugleich für die spätere Ausführung mittles eines Notbooks auf Colab aufgerufen

werden kann. Hierfür und zur Versionsverwaltung wird der erstellte Code auf Git gepushed und schließlich mit einem Script des Colab-Notebooks eine Verbindung von Colab zu Git hergestellt, worauf das Projekt auf den Cloud-Rechner gecloned wird. Danach werden weitere Scripte ausgeführt um eine Verbindung zu Google-Drive herzustellen und die dort hochgeladenen Datensätze auf den GPU-Cloud-Rechner zu laden. Ein Script stellt folglich für die Überwachung des Trainings einen remote TensorBoard-Tunnel zum Ultrabook her. Letztlich wird auf der Cloud-Seite die Konfiguration des Trainings vorgenommen und über eine 12-Stunden-Session trainiert (sofern keine Verbindungsfehler auftreten). Je größer das Modell, desto mehr dieser Sessions werden benötigt. Eine Sicherung von Model-Checkpoints auf Google-Drive findent über das gesamte Training alle 500 Schritte statt. Somit kann bei einem Fehlerfall oder nach dem Ende einer Session das Training wieder aufgenommen werden. Nach erfolgreichem Training, Evaluierung und Test kann letztlich das Modell von Google-Drive heruntergeladen und weiter getestet werden, beispielsweise durch den bildlichen Vergleich von Rekonstruktionen.

4.3 Coupling-Koeffizienten-Test für originale CapsNets mit einer ClassCaps

Der Coupling-Koeffizienten-Test soll für das originale Routing-by-Agreement mit einer Capsule im letzten Layer und ohne Verwendung von Techniken, die das Routing beeinflussen (wie Leaky-Routing) praktisch zeigen, dass hierbei alle Coupling-Koeffizienten immer bei eins liegen und daher das Routing nicht sinnvoll ist, da dieses seine Stärken so nicht ausspielen kann. Außerdem wird somit in diesem Abschnitt die erlernte Theorie praktisch geprüft.

4.3.1 Architektur und Implementierung des CapsNets

Um die Coupling-Koeffizienten eines CapsNets mit einer einzelnen letzten Capsule vergleichen zu können, werden vier Modelle implementiert. Alle Modelle unterscheiden sich lediglich im ClassCaps-Layer vom Original, um so nahe wie möglich an den in der Literatur verwendeten CapsNets mit einer Ausgabe-Capsule zu bleiben. Das erste Modell *a* besitzt nur eine Capsule im letzten Layer und wird mit einer Routing-Iteration trainiert. Modell *b* entspricht Modell *a*, nur dass mit drei Routing-Iterationen trainiert wurde, um zu zeigen, dass nach wie vor alle Coupling-Koeffizienten bei eins liegen. Modell *c* entspricht ebenfalls Modell *a*, nur dass das Training mit drei Leaky-Routing-Iterationen stattfindet, weswegen die Coupling-Koeffizienten von eins abweichen sollten. Letztlich wird in Modell *d* das Modell *a* um eine Capsule im letzten Layer erweitert und mit drei Routing-Itrationen trainiert, damit dargestellt werden kann, dass das Routing in einem CapsNet mit mehr als einer Capsule sinnvoll ist.

4.3.2 Training und Ergebnisse

Alle vier Modelle werden für eine einfache MNIST-Aufgabe trainiert, da die Klassifizierung von Datensätzen hierbei nicht im Vordergrund steht. Nur das Verhalten der Coupling-Koeffizienten hinsichtlich des letzten Layers soll untersucht werden. Daher besteht die Aufgabe darin, für zufällig gezeigten MNIST-Zahlen zu erkennen, ob es sich um eine Null handelt oder nicht. Bei den Modellen mit einer Capsule wird das erreicht, indem die Länge des Aktivitätsvektors der einzelnen Capsule herangezogen wird. Ist dieser länger als 0.9, so ist das Netz ausreichend sicher, dass die Eingabe eine Null darstellt. Ansonsten wird die Entscheidung des Netzes vice versa interpretiert und es handelt sich in der Eingabe um eine der anderen Ziffern 1-9. Für das Modell mit zwei Capsules entspricht die erste Capsule der Klasse der Null und die zweite Capsule der Rest-Klasse für die Ziffern 1-9. Die Capsule mit dem längsten Aktivitätsvektor wird als Entscheidung des CapsNets betrachtet. Die Loss-Funktion ist analog zum Original der Margin-Loss aus Gleichung 3.7 und die Hyper-Parameter des Trainings entsprechen ebenfalls denen des originalen, auf MNIST trainierten CapsNet aus Abschnitt 3.3.4. Nur die Batch-Size wurde auf 32 heruntergesetzt. Während des Trainings werden die Coupling-Koeffizienten der Modelle für jede Iteration geplottet:

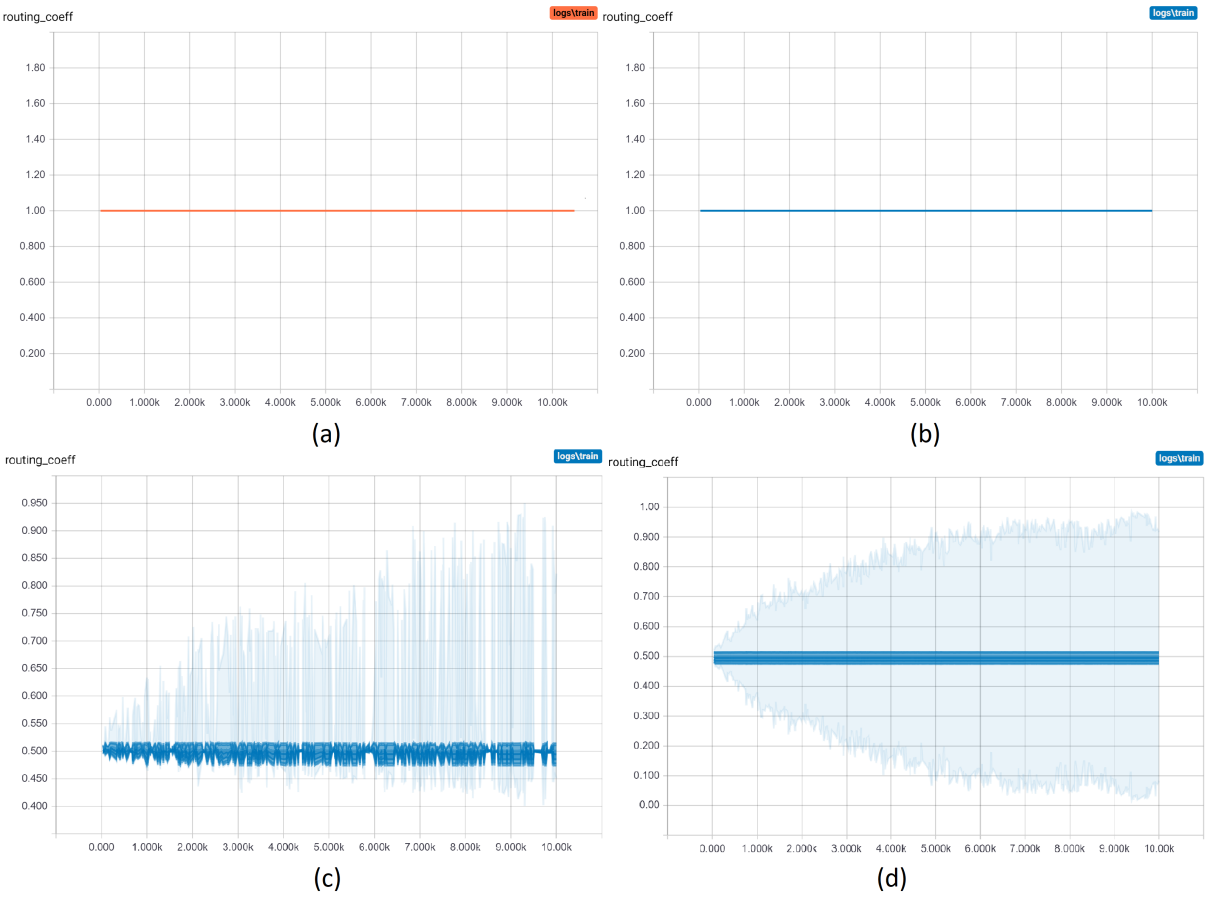


Abbildung 4.1: Coupling-Koeffizienten von den Modellen mit (a) einer Capsule und einer Routing-Iteration, (b) einer Capsule und drei Routing-Iterationen, (c) einer Capsule und drei Leaky-Routing-Iteration, (d) zwei Capsules und drei Routing-Iterationen im letzten Layer

Abbildung 4.1 stellt die Verteilung der Coupling-Koeffizienten aller vier Modelle innerhalb 10 000 Trainings-Iterationen dar, wobei eine Iteration dem Training mit einem Batch entspricht. Hier wurde das Training nach 10 000 Iterationen abgebrochen, da die Ergebnisse eindeutig sind. Somit wurde Zeit gespart, die für andere Aufgaben genutzt werden konnte. Das Alpha der Farbe (*Orange* oder *Blau*) mit welcher die Coupling-Koeffizienten dargestellt werden, entspricht der Häufigkeit des Aufkommens der entsprechenden Coupling-Koeffizienten-Werte auf der *Y-Achse*. Je sichtbarer die Farbe dabei ist, desto häufiger kommt der entsprechende Coupling-Koeffizienten-Wert zwischen dem PrimaryCaps- und ClassCaps-Layer des Modells vor. Die Abbildung 4.1 zeigt dabei klar, dass die Coupling-Koeffizienten bei lediglich einer Capsule im letzten Layer, unabhängig von den Routing-Iterationen (siehe *(a)* für eine und *(b)* für drei Iterationen) immer bei eins liegen. Wird Leaky-Routing verwendet *(c)*, zeigt das Routing seine Wirkung und die Coupling-Koeffizienten verteilen sich. Bei dem Nutzen zweier Capsules

(siehe *(d)*), welche beide direkt vom Loss beeinflusst werden, scheint diese Verteilung gleichmäßiger zu sein. Dies könnte daran liegen, dass die PrimaryCapsules über die Rest-Capsule-Features der von Null unterschiedlichen Ziffern lernen, weshalb zu der Rest-Capsule große Coupling-Koeffizienten existieren, wenn eine Ziffern 1-9 in der Eingabe vorhanden ist. Dem hingegen lernt bei Leaky-Routing das CapsNet hauptsächlich Features der Null, da nur die ClassCaps der Null und eine Leaky-Dimension - von welcher kein direkter Gradient ausgeht - den PrimaryCapules zur Verfügung stehen. Daher gibt es keinen direkten Gradienten-Fluss für die Features anderer Ziffern außer der Null, sondern nur den indirekten über die Null, wenn der Loss die Capsule bestraft wird sobald eine 1-9 als Null interpretiert wird. Daher sind alle Coupling-Koeffizienten nahe null, sobald eine Ziffer, die wenig gemeinsame Features mit der Null teilt, die Eingaben des Netzes darstellt, woraus das unregelmäßige Muster der Coupling-Koeffizienten in Abbildung 4.1 *(c)* resultiert. Da die Aufgabenstellung mit den oberen Ergebnissen erfüllt wurde und nun aus Zeitgründen zu der nächsten Aufgabe übergegangen werden sollte, wird das Verhalten von Coupling-Koeffizienten für verschiedene Routing-Techniken und Erweiterungen in dieser Arbeit nicht näher untersucht.

4.3.3 Diskussion und Lösungsvorschläge

Es wurde gezeigt, dass das originale Routing-by-Agreement nur seine Wirkung ausüben kann, wenn mehr als eine Capsule im letzten Layer verwendet werden, oder Techniken wie Leaky-Routing im CapsNet Einsatz finden. Dies lässt sich verallgemeinern: Wird in einem CapsNet mit dynamischen Routing-by-Agreement ein Layer mit einer einzelnen Capsule eingesetzt und keine weiteren Routing-Techniken wie beispielsweise Leaky-Routing verwendet, so wird das Routing zwischen dem Layer mit einer Capsule und dem Layer darunter ausgehebelt.
Als mögliche Lösungsvorschläge wurden weiter die Coupling-Koeffizienten für ein CapsNet mit Leaky-Routing und für ein CapsNet mit zwei Capsules dargestellt. Allerdings existieren noch andere Verfahren zur Lösung dieses „Problems“. So kann beispielsweise eine in Abschnitt 3.3.1 erläuterte Orphan-Kategorie verwendet werden, oder ein „umgedrehtes Routing“, wie es von Jingjing Gong et al. in [Gon+18] vorgestellt wurde. Der einzige Unterschied zum originalen Routing ist hierbei, dass die oberen Capsules über ihre Coupling-Koeffizienten zu den unteren Capsules entscheiden und diese auf eins aufsummieren. Daher ist dort das Routing immer sinnvoll, solange sich mehr als eine Capsule im unteren Layer befinden.
Welcher der vier Lösungsvorschläge den besten darstellt, kann ohne weitere, ausführliche Experimente nicht entschieden werden. Womöglich würden diese Experimente sogar zeigen, dass der beste Lösungsvorschlag abhängig von der Aufgabenstellung oder des Datensatzes ist. Jedoch können die vier Lösungen theoretisch diskutiert werden:
Abschnitt 3.5.2 zeigt, dass ein GAN mit zwei Ausgabe-Capsules (eine für die Fake- und die

andere für die Echt-Wahrscheinlichkeit) gute Ergebnisse bringt. Weiter ist das Leaky-Routing eigentlich dazu gedacht einen variierenden Hintergrund auszugleichen, weshalb dies, wenn es als Lösung für dieses Problem eingesetzt wird, evtl. bei komplizierteren Datensätzen in diesem Kontext unvorhergesehene Auswirkungen haben könnte. Auch zu beachten ist, dass durch die Leaky-Dimension kein direkter Gradient des Losses durch das CapsNet fließt, welcher von der Rest-Klasse stammt, sondern nur indirekt durch den Gradient der Capsule, welche die zu erkennende Klasse darstellt und bei falscher Klassifizierung bestraft wird. Dies könnte evtl. Nachteile gegenüber der Lösung mit einer zweiten Capsule bringen, da beispielsweise das CapsNet bei GANs nicht direkt lernt, was ein echtes Bild ausmacht, sondern nur was ein Generator-Bild ausmacht. Sollen allerdings nur Features einer Klasse im Netz gelernt werden, könnte dies wiederum als Vorteil gesehen werden, da somit beispielsweise mehr „Platz“ existiert, um diese Features in ihrer Breite zu perfektionieren. Auch bei einer Orphan-Kategorie fließt über die Orphan-Capsule kein direkter Gradient, was auch hier ähnliche Folgen haben könnte. Der Unterschied zu Leaky-Routing ist, dass hierbei nicht jeder Capsule eine Dimension zur Verfügung gestellt wird, in welche gerootet werden kann, wenn keine obere Capsule gut genug übereinstimmt, sondern eine einzige komplette obere Orphan-Capsule, zu der die unteren Capsules routen können. Dies scheint mehr Performance zu benötigen, da zusätzlich die üblichen Capsule-Berechnungen für die Orphan-Capsule ausgeführt werden müssen.
Das umgedrehte Routing von Jingjing Gong et al. [Gon+18] wurde, kurz erläutert, dazu entwickelt einen Text über ein original CapsNet in der Ausgabe zu codieren. Jedoch wurden dabei drei Ausgabe-Capsules verwendet [Gon+18], was den Vergleich im Kontext der Aufgabenstellung schwierig macht. Für die drei Ausgabe-Capsules schnitt das umgedrehte Routing jedoch gegen den originalen Routing-Algorithmus etwas schlechter ab [Gon+18]. Da der Unterschied jedoch gering ist, kann vermutet werden, dass das umgedrehte Routing bei einer Capsule im Ausgabe-Layer besser abschneidet, da das originale Routing durch die alleinstehende Capsule im oberen Layer ausgehebelt wird. Allerdings gilt auch hier, wie für die anderen drei Lösungsvorschläge, dass ausführlich Experimente angestellt werden müssen, um dies vernünftig bewerten zu können. Nichtsdestotrotz ist jeder der Vorschläge besser, als das Routing wirkungslos zu lassen solange mit dem original CapsNet gearbeitet wird, da das Routing dessen Kern darstellt.

4.4 Überwachtes Lernen eines CapsNets zur Facial-Emotion-Recognition

Dieser Abschnitt beschreibt das überwachte Training einiger Modelle, um Emotionen anhand von Gesichtern zu klassifizieren. Hierfür werden zu Beginn ein tieferes und zwei flache CapsNets trainiert und basierend auf deren Ergebnissen eine zweite Version des tiefen CapsNets erstellt. Weiter wird die zweite Version als Basis verwendet, von welcher aus einzelne Än-

derungen vorgenommen und deren Auswirkungen auf die Klassifizierung diskutiert werden. Ebenfalls wird ein neues Rekonstuktions-Modul hergeleitet und eingesetzt, welches weit weniger Parameter aufweist, als das originale fc Modul und dessen Ergebnisse verglichen. Das am besten abschneidende CapsNet wird dann in Abschnitt 4.5 unüberwacht trainiert.

4.4.1 Architektur und Implementierung dreier überwachter Ausgangs-CapsNets

In diesem Teil werden die Modelle der drei CapsNets dargestellt und relevante Architektur-Entscheidungen erläutert.

Architektur des tiefen CapsNets

Zuerst soll lediglich mit den theoretisch erlangten Kenntnissen und ohne dem Folgen einer speziellen, bereits angewandten Architektur ein tiefes CapsNet entwickelt werden. Abbildung 4.2 zeigt hierbei die allgemeine Struktur des Modells. Diese generelle Struktur gilt auch für die meisten weiteren entwickelten, tiefen CapsNets.

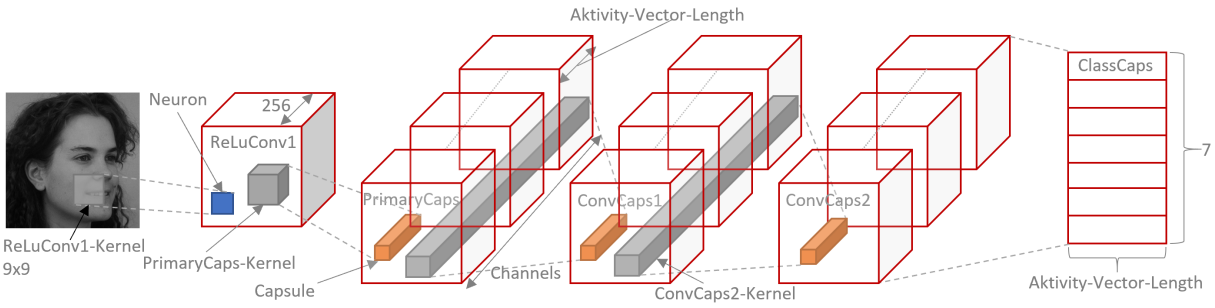


Abbildung 4.2: Allgemeine Struktur der entwickelten tiefen CapsNets

Wie in Abbildung 4.2 dargestellt wird in dieser Arbeit unter einem „tiefen“ CapsNet ein Netzwerk mit einem conv. Layer und vier Capsule-Layern verstanden. Wie im originalen CapsNet werden im *ReLuConv1*-Layer aus der Eingabe über 9x9 Kernel und eine ReLu-Aktivierungsfunktion 256 Feature-Maps erstellt, auf welchem der *PrimaryCaps*-Layer operiert. Daher findet wie im Original kein Routing zwischen *ReLuConv1* und *PrimaryCaps* statt. Zwischen allen anderen Layern wird das Routing-by-Agreement mit drei Routing-Iterationen ausgeführt (später auch Leaky). Der *ConvCaps1*-Layer ist gleich definiert wie der *PrimaryCaps*-Layer, wandelt jedoch nicht primär die Eingabe-Features in Capsule-Repräsentationen um, sondern arbeitet bereits auf den Capsule-Repräsentationen des *PrimaryCaps*-Layers, weshalb dieser als *ConvCaps1* bezeichnet wird. Dabei operiert ein Capsule-Channel des *ConvCaps1*-Layers über den Kernel (meist 9x9) auf der gesamten *PrimaryCaps*-Ausgabe. Dies ist mit bedachter Wahl der Capsule-Anzahl und der Ausgabevektor-Länge aus Performancesicht möglich, da es sich

hierbei um conv. und nicht um fc Capsule-Layer handelt. Anstelle auf den 256 Featur-Maps des *ReLuConv1*-Layers arbeitet *ConvCaps1* also auf den $Channels \times AktivityVectorLength$ Feature-Maps des *ConvCaps1*-Layers. Analog operiert der *ConvCaps2*-Layer auf dem *ConvCaps1*-Layer und schließlich wird, wie im originalen CapsNet über einen fc *ClassCaps*-Layer die Ausgabe des *ConvCaps2*-Layers in Repräsentationen der sieben Klassen des Datensatzes umgewandelt. Hierbei zeigt die Abbildung 4.2 links bereits eine Beispieleingabe, jedoch wird der Datensatz später genauer erläutert.

Da die allgemeine Struktur des tiefen CapsNets nun dargestellt wurde, können die einzelnen Hyperparameter, welche in den kommenden Modellen variieren, für das erste Ausgangs-Modell (*KdefDeepCapsNet*) Layer für Layer erläutert werden:

ReLuConv1: Dieser Layer erhält als Input ein 64x64 Bild welches aus einem 70x70 Bild zufällig herausgeschnitten wurde. Dieses Herausschneiden entspricht der erläuterten Anwendung des originalen CapsNets aus Abschnitt 3.3 auf Cifar10 und sorgt dafür, dass ähnlich wie zum Verschieben der MNIST-Zahlen die Eingabe um ein paar Pixel in ihrer Position variiert, damit das Training der räumlichen Äquivarainz stattfinden kann. Außerdem wird der Datensatz dadurch künstlich erweitert und enthält somit mehrere „unterschiedliche“ Trainingsdaten. Aus der 64x64 schwarzweiß Eingabe erstellt der Layer dann mittels einer ReLu-Aktivierungsfunktion über 9x9 Kernel ohne Stride und mit Valid-Padding 256 Feature-Maps.

PrimaryCaps: Der Layer besitzt acht Capsule-Channel, welche Capsules mit einer Ausgabevektor-Länge von vier besitzen. Das conv. Gitter (Höhe, Breite) jedes Channels wird erstellt in dem die Capsules mit einem 9x9 Kernel unter einem Stride von zwei über die Eingabe mit Valid-Padding analog zu Abschnitt 3.3 angewandt werden.

ConvCaps1: Dieser Layer besitzt 16 Capsule Channel, jedoch wird die Ausgabevektor-Länge auf sechs erhöht, da höher liegende Capsules, wie in Abschnitt 3.1.4 diskutiert, größere Gebiete umfassen, weshalb niedrige „place-coded“ Äquivarianz in höhere „rate-coded“ Äquivarianz umgewandelt wird und diese Verschiebung von Place-Coding zu Rate-Coding kombiniert mit der Tatsache, dass Capsules auf höherer Ebene komplexere Einheiten mit mehr Freiheitsgraden darstellen, nahe legt, dass die Dimensionalität von Capsules zunehmen sollte, je höher sich diese in der Hierarchie befinden. Daher wächst auch die Länge des Aktivitätsvektors in folgenden Layern. Weiter besitzt auch dieser Layer einen 9x9 Kernel, einen Stride von zwei und Valid-Padding. Routing zwischen *PrimaryCaps* und *ConvCaps1* findet mit drei Iterationen statt.

ConvCaps2: Hier finden 32 Capsule-Channel mit einer Ausgabevektor-Länge von acht, durch einen 9x9 Kernel, einen Stride von eins und einem Same-Padding Verwendung. Das Routing wird in drei Iterationen zwischen *ConvCaps1* und *ConvCaps2* ausgeführt.

ClassCaps: Da der Datensatz sieben Klassen besitzt, enthält dieser Layer sieben fc Capsules. Die Länge des Ausgabevektors beträgt 32, weshalb die gesamte Ausgabe die Größe $7 \times 32 = 224$ besitzt. Auch zwischen diesem Layer und *ConvCaps2* werden drei Routing-Iterationen verwendet. Analog zum Original wird mittels einer Label-Maske die Ausgabe des Layers maskiert und zum Rekonstruktions-Modul weitergeleitet. Da die Eingabe zum Rekonstuktions-Modul immer gleich groß sein muss, werden dabei (wie im Original) alle Ausgabevektoren zusammengehängt und nur der Ausgabevektor der Capsule, welcher die Klasse repräsentiert, wird nicht genullt.

Rekonstuktions-Modul: Aus in Abschnitt 3.8.6 diskutierten Gründen, enthält ein Rekonstuktions-Modul, welches ausschließlich aus fc Layern besteht, eine hohe Parameteranzahl. Um diese Parameteranzahl auch für große Eingaben gering zu halten, können womöglich andere Rekonstruktions-Techniken in Frage kommen. So wird in dieser Arbeit basierend auf dem Wissen über die Deconvolution (umgekehrte Convolution) ein conv. Rekonstuktions-Modul erstellt. Hierbei ist jedoch ein Problem der reinen Deconvolution zu beachten, wobei zu Artefakten führende Überlappungen auftreten können, insbesondere wenn die Kernel-Size nicht restlos durch den Stride teilbar ist, wie es Abbildung 4.3 veranschaulicht [OD16].

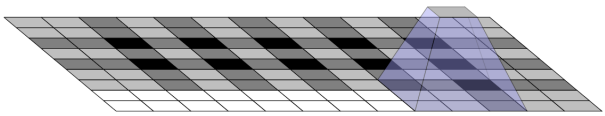


Abbildung 4.3: Durch Deconvolution auftretende Überlappungen im zweidimensionalen Raum aus [OD16]

Abbildung 4.3 zeigt weiter, dass sich im zweidimensionalen Raum die Überlappungen der beiden Achsen miteinander multiplizieren und ein charakteristisches schachbrettartiges Muster erzeugen. Theoretisch könnte das Netzwerk Gewichte lernen, um genau dieses zu vermeiden; praktisch haben die Netze Schwierigkeiten damit diese Muster zu verhindern. Allerdings verwenden neuronale Netze typischerweise mehrere Ebenen der Deconvolution beim Erstellen von Bildern, um iterativ ein größeres Bild aus einer Reihe von kleineren Beschreibungen zu erstellen. Diese gestapelten Deconvolutions wären ebenfalls fähig die Artefakte aufzuheben, jedoch verbinden diese sich oft und erzeugen Artefakte verschiedener Größen. [OD16].
Stride-1-Deconvolutions werden meist im letzten Layer von erfolgreichen Modellen eingesetzt

und sind sehr effektiv bei der Dämpfung von Artefakten. Allerdings zeigen viele neuere Modelle, dass Artefakte auch dabei durchaus noch durchdringen können [OD16]. Eine andere Vorgehensweise ist es, sicherzustellen, dass Kernel-Sizes verwendet werden, die durch den Stride restlos teilbar sind, um das Problem der Überlappung zu vermeiden. Dies verbessert zwar das Ergebnis ein wenig, jedoch ist es für die Deconvolution dennoch leicht, Artefakte zu erzeugen. Aus diesen Gründen wird eine Alternative zur regulären Deconvolution benötigt, welche im Gegensatz zur Deconvolution beim Upsampling keine Artefakte als Standardverhalten aufweist. Im Idealfall würde der Ansatz sogar weiter gehen und gegen solche Artefakte vorgehen. Eine Lösung besteht darin, das Upsampling auf eine höhere Auflösung und die Feature-Berechnung der Convolution zu separieren. Beispielsweise kann das Bild durch Nearest-Neighbor-Interpolation vergrößert und darauffolgend eine conv. Layer für die Features eingefügt werden [OD16]. Dies führt laut Odena et al. [OD16] zu guten Ergebnissen. Für die Implementierung können Resize-Convolution-Layer in TensorFlow einfach mit tf.image.resize_images() und tf.nn.conv2d() erstellt und vor einer Convolution (tf.nn.conv2d()) die Funktion tf.pad() verwendet werden, um Randartefakte zu vermeiden [OD16].
Nach diesem Schema wird ein Rekonstruktions-Modul für eine (noch relativ kleine) 64x64 Eingabe wie folgt aufgebaut:

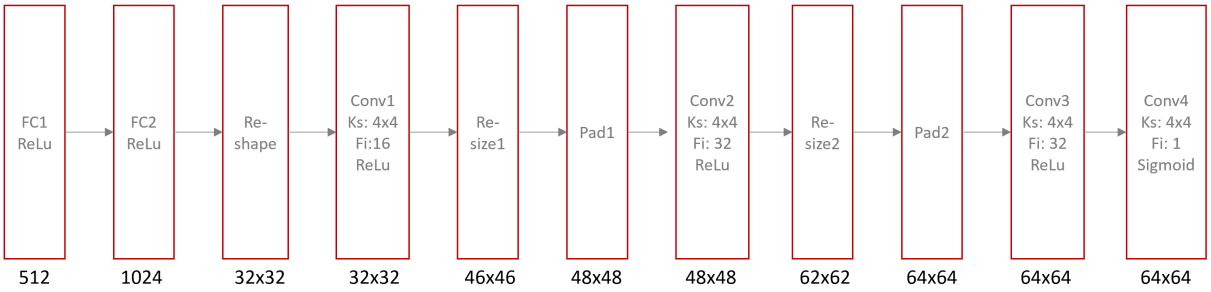


Abbildung 4.4: Rekonstruktions-Modul aus fc, convolutional, resize und pad Layern

Abbildung 4.4 zeigt, dass wie im originalen CapsNet, die Features des maskierten Capsule-Layers zuerst zwei fc Layer (*FC1* und *FC2*) passieren. Diese Layer besitzen 512 und 1024 Neuronen und sorgen dafür, dass der folgende Layer eine Eingabe erhält, die groß genug ist, um diese im Zweidimensionalen auf eine brauchbare höhere Auflösung zu skalieren. Daraufhin folgt der Reshape auf 2D und der erste conv Layer (*Conv1*); beim originalen CapsNet würde hingegen ein weiterer fc Layer folgen. Dies ist der Punkt an dem die Parameter Einsparung stattfindet. Würde es sich um einen weiteren fc Layer handeln, befänden sich allein zwischen *FC2* und diesem Layer $1024 \times 64 \times 64 + 64 \times 64 = 4\,198\,400$ Parameter. Wird die Eingabegröße auf 128x128 verdoppelt, wären es bereits $16\,793\,600$ Parameter. Hier wird allerdings nach dem Reshape auf 2D, welches im originalen Rekonstruktions-Modul zum Schluss statt-

findet, eine zweifache Abfolge von conv., resize und padding Layern verwendet, bevor durch *Conv4* die endgültige Rekonstruktion erstellt wird. Dabei verwendet *Resize1* und *Resize2* eine Nearest-Neighbor-Interpolation (tf.image.ResizeMethod.NEAREST_NEIGHBOR). Wäre hierbei das zu rekonstruierende Ziel größer, können weitere dieser Abfolgen eingefügt werden. Dieses Vorgehen führt dazu, dass das gesamte Rekonstuktions-Modul für eine 64x64 Eingabe mit sieben ClassCaps und 16D-Ausgabevektor lediglich $608\,593$ Parameter benötigt; das originale Rekonstuktions-Modul würde hierfür aus $4\,781\,568$ Parameter bestehen, was zu einer Einsparung von 87.27% führt. Bei doppelter Eingabegröße und dem selben Modul mit Resize-Werten von 62 und 126 anstelle von 46 und 62, hätte das Modul ebenfalls $608\,593$ Parameter, das originale Modul jedoch $17\,327\,768$, was eine Einsparung von 96.48% ergibt.
Allerdings könnte die Grundstruktur des 64x64 Rekonstuktions-Moduls für 128x128 Ausgaben nicht ausreichen. Um dem entgegen zu wirken, wird ein Modul erstellt, dessen Struktur wie folgt definiert ist: Nach den 512 und 1024 fc Layern folgt der Reshape auf 32x32 und auf diesen ein conv. Layer, der analog zu *Conv1* aus der Abbildung 4.4 erstellt wird. *Conv1* folgt wiederum ein Resize auf 46x46 und das anschließende Padding auf 48x48. Diese Conv-Resize-Pad-Struktur wird dann vier mal mit den Resize Werten 62, 86, 110, 126 und den conv. Filteranzahlen von 32, 32 ,64, 64 wiederholt. Zuletzt erstellt ein weiter conv. Layer analog zu *Conv4* aus der Abbildung 4.4 die Rekonstruktion. Dieses Rekonsuktions-Modul besitzt dann $711\,649$ Parameter und führt zu einer Einsparung von 95.89% gegenüber dem Original.

Architektur der flachen CapsNets

Das erste flache Netz ist das FACSCaps-Modell von David Rawlinson et al. aus Abschnitt 3.8.6, welches wiederum stark dem originalen CapsNet ähnelt. Da dieses in Abschnitt 3.8.6 bereits beschrieben wurde, wird das Modell hier nicht erneut erläutert. Jedoch wurde das Modell angewandt um AUs aus Bildern von Gesichtern zu erkennen und mehre ClassCaps anstelle einer zur Rekonstruktion des Bildes verwendet. Da eine Emotion mehreren AUs entspricht und da die Länge des Ausgabevektors in höheren Layen zunehmen soll, wird im zweiten, flachen Modell die Ausgabevektorlänge des FACSCaps-Netzes von 8 auf 16 erhöht, damit einzelne Capsules genügend Platz besitzen, um die für eine Emotion nötigen Parameter in ihrer Ausgabe zu codieren. Womöglich lernt eine Capsule hierbei sogar über das Erkennen von Emotionen indirekt AUs. Außerdem wird mit der aus Performancegründen maximal möglichen Batch-Size von 64 das originale CapsNet mit exakt gleichen Parametern, wie die im original Paper (siehe Abschnitt 3.3.4), trainiert. Das Training aller andren CapsNets - einschließlich der tiefen - kann, wie im folgenden Abschnitt beschrieben, mit einer Batch-Size von 128 ausgeführt werden.

4.4.2 Training und Ergebnisse der überwachten Ausgangs-CapsNets

Die Modelle werden auf dem Kdef (Karolinska Directed Emotional Faces)-Datensatz trainiert, welcher 4900 512x512 Farbbilder von menschlichen Gesichtern enthält. Die Gesichter wurden aus fünf verschiedenen Posen aufgenommen: Frontal, circa 45 Grad nach links und rechts, sowie seitwärts links und rechts, sprich 90 Grad, gedreht. Jede Person imitierte hierbei sieben verschiedene Emotionen: ängstlich, wütend, angeekelt, traurig, fröhlich, überrascht und neutral. Daher dass die Emotionen geschauspielert sind, entsprechen diese eher dem Idealbild, weshalb mögliche unterbewusste Gesichtsregungen, welche bei ungespielten Emotionen zustande kommen, im Datensatz nicht vorhanden sind. Jedoch ist es schwierig solch einen ungespielten, gelabelten Datensatz überhaupt zu finden und dadurch, dass die Bilder aus Performancegründen komprimiert werden, ist es fraglich ob diese Regungen überhaupt erkennbar wären.
Der Datensatz beinhaltet trotz der kontrollierten Umgebung fehlerhafte Bilder (nur schwarz oder überbelichtet) welche aussortiert werden, wodurch der Datensatz auf 4820 Bilder reduziert wird. Das Aussortieren fehlerhafter Bilder wurde auch von anderen Verfahren die auf diesem Datensatz arbeiten durchgeführt. Weiter wird jedes der Bilder vertikal gespiegelt, um den Datensatz künstlich auf 9640 Bilder zu erweitern. Nach dem zufälligen Mischen werden schließlich 1328 Bilder für den Test, 6650 Bilder für das Training und die verbleibenden 1620 Bilder für die Evaluation ausgewählt. Die leicht variierenden Größen des Evaluierungs- und Testdatensatzes entstand durch das Einlesen, Spiegeln und Aussortieren der Daten. Außerdem ist das Mischen hierbei nicht ganz sinnvoll, da dadurch zufälligerweise keine komplett neuen Gesichter im Testdatensatz auftreten könnten, sondern nur bekannte Gesichter mit anderen Emotionen oder aus anderem Winkel. Da jedoch jedes Verfahren, mit welchen das beste CapsNets zum Schluss verglichen wird, die Daten zufällig mischt, wird aus Gründen der Vergleichbarkeit hier ebenfalls so vorgegangen. Außerdem wird mittels eines Seeds gemischt, damit hier jedes Modell auf denselben Daten arbeitet.
Die Höhe der Bilder wird um 140 Pixel von oben und 90 von unten verkleinert, um den irrelevanten Hintergrund zu reduzieren. Für die beiden flachen CapsNets wird die Eingabe auf 140x140 komprimiert und zu schwarzweiß konvertiert. Folglich wird ein zufälliger 128x128 Ausschnitt dem Netz überreicht, wobei das vorherige Reduzieren des Hintergrundes zu besseren Details im Gesicht der Eingabe führt. Für das erste tiefe CapsNet hingegen ist der Input ein 64x64 Ausschnitt des Bildes, welches zuvor auf 70x70 skaliert wurde. Hierbei werden diese unterschiedlichen Größen verwendet, da das tiefe CapsNet vor der Kenntnis über FaceCaps und somit vor den flachen Modellen entwickelt wurde. Dabei verwendet das tiefe Netz keinen Stride in ersten conv. Layer, FaceCaps hingegen einen Stride von vier, weshalb eine 128x128 Eingabe mit einer Batch-Size von 128 aus Seiten der Performance überhaupt erst möglich wird. Im späteren Verlauf werden jedoch basierend auf den hier erzielten Ergebnissen tiefe CapsNets

auf einer 128x128 Eingabe trainiert.
Alle Modelle verwenden die originale Kombination aus Margin- und Rekonstruktion-Loss. Bei FaceCaps und dem daraus resultierenden Modell (KdefCaps) wird hierbei, wie in Abschnitt 3.8.6, die Decay-Rate von 0.96 auf 0.9 und der Balance-Faktor zwischen Rekonstuktion-Loss und Margin-Loss von 0.005 auf 0.0005 geändert. Die Lernrate der Modelle für dem Adam-Optimizer beträgt 0.001 und der Decay findet alle 2000 Schritte statt. Die Parameter des original CapsNets bleiben dem hingegen unverändert.

Modell	**Train**	**Eval**	**Test**	**Parameteranzahl**
OriginalCaps	84.32%	70.48%	69.40%	$11\,780\,976 + 17\,376\,768 = 29\,157\,744$
FaceCaps	96.29%	80.16%	79.70%	$901\,464 + 17\,348\,096 = 18\,249\,560$
KdefCaps	**97.43%**	**82.25%**	**81.88%**	$1\,118\,352 + 17\,376\,768 = 18\,495\,120$
KdefDeepCaps	97.31%	80.74%	79.94%	$1\,190\,336 + 665\,937 = \mathbf{1\,856\,273}$

Tabelle 4.1: Trainings-, Evaluations- und Testergebnisse der Modelle

Tabelle 4.1 zeigt die Trainings-, Evaluations- und Testergebnisse der vier Modelle und gibt zu erkennen, dass das flache *KdefCaps*-Modell mit einem 16D-Ausgabevektor die höchste Genauigkeit aufweist. Grund dafür kann dabei nur, wie bereits oben vermutet, der größere Aktivitätsvektor sein, welcher vermutlich hilft vielsagende Features über Gesichtszüge zu kodieren. Das originale CapsNet schneidet nach fast dreifach so vielen Trainings-Iterationen ($150\,000$) am schlechtesten ab, da vermutlich das Modell zwar noch weiter lernen könnte, aber sich die Trainingsgeschwindigkeit durch die hohe Parameteranzahl und der Größe der Eingabe viel zu stark verringert hat und daher fast der Konvergenz gleicht. Somit muss das Training abgebrochen werden, weil dies zeitlich nicht mehr berechenbar ist. Tabelle 4.1 zeigt weiter, dass das erarbeitete tiefe *KdefDeepCaps*-Modell das FaceCaps-Modell schlägt und außerdem unter 9.96-facher Parametereinsparung (mit nur 10.04% der Parameter) gegenüber *KdefCaps* lediglich eine 1.94% geringere Testgenauigkeit aufweist. Die Gründe dafür, dass das erarbeitete tiefe Modell etwas schlechter ausfällt als das ebenfalls erarbeitet *KdefCaps*-Modell sind zahlreich: So ist vermutlich die kleinere 64x64 Eingabe von *KdefDeepCaps* am ausschlaggebendsten. Weiter wird in diesem ein eigens kreiertes Rekonstuktions-Modul verwendet, welches ebenfalls die Klassifizierung verschlechtern könnte. Womöglich ist sogar die wesentlich höhere Capsule-Anzahl oder die Ausgabevektor-Größe in den einzelnen Layern für die etwas schlechteren Ergebnisse verantwortlich. Um diesem nachzugehen werden im folgenden Abschnitt vorerst die verschiedenen Rekonstuktions-Module untersucht und im weiteren Verlauf eine Vielzahl verschiedener tiefer Modelle getestet. Wie Abschnitt 4.4.5 zeigen wird, sind die daraus entstehenden Ergebnisse nahe dem State-of-the-Art.

4.4.3 Diskussion der Auswirkungen des Rekonstruktions-Moduls

Um die Auswirkungen der kreierten Rekonstuktions-Module zu testen, wird das flache *KdefCaps*-Modell mit diesen trainiert und die Ergebnisse mit demselben Modell, welches mit einem fc Modul gelernt wurde, verglichen. Dabei werden beide beschriebenen Rekonstruktions-Module für die Größe 128x128 verwendet. Außerdem wird in Abschnitt 4.4.5 ein bildlicher Vergleich der Rekonstruktionen von CapsNets mit unterschiedlichen Rekonstuktions-Modulen vorgenommen.

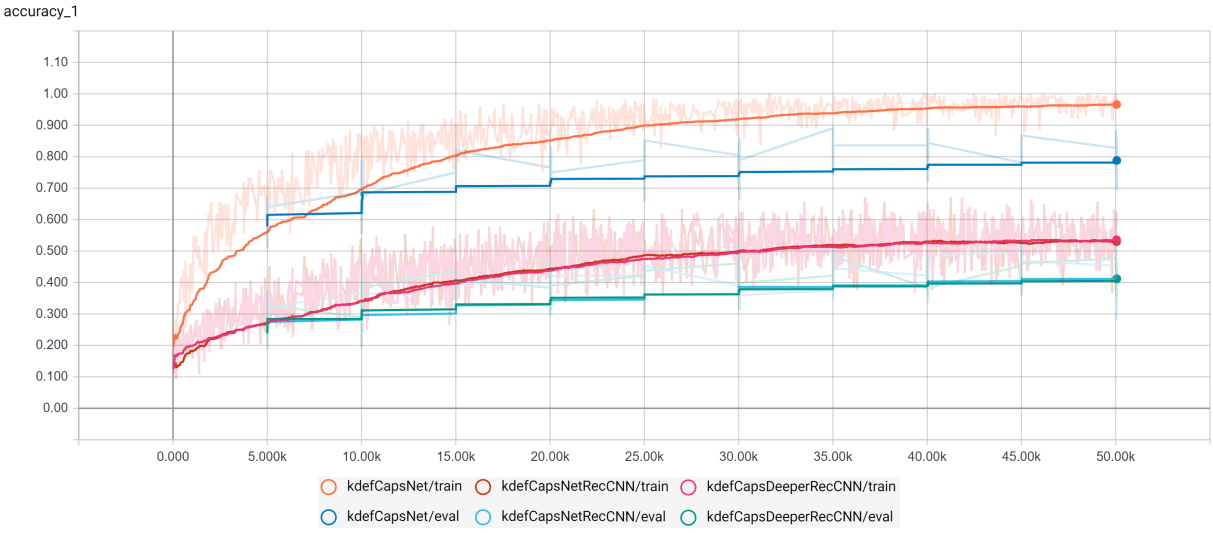


Abbildung 4.5: Trainings- und Evaluations-Klassifizierungsgenauigkeit für das vorgestellte flache KdefCapsNet mit jeweils einem der drei Rekonstruktions-Module nach 50 000 Trainings-Iterationen

Abbildung 4.5 zeigt die Trainings- und Evaluations-Klassifizierungsgenauigkeit nach 50 000 Trainings-Iterationen für das flache KdefCapsNet mit dem originalen Modul (*orange* und *blau*), dem flachen Deconvolution-Modul (*rot* und *hellblau*) und dem tiefen Deconvolution-Modul (*pink* und *grün*). Dabei stellen jeweils die Graphen mit hohem Alpha die geglättete Version der unbearbeiteten Graphen mit niedrigem Alpha dar. Das „Treppenmuster“ der Evaluations-Kurve kam zu Stande, da alle 5000 Schritte die Evaluations-Klassifizierungsgenauigkeit für den gesamten Evaluations-Datensatz batchweise berechnet und nacheinander die einzelnen Ergebnisse des Batchs geplottet wurden, anstelle diese zu mitteln. Dies ist weniger übersichtlich als das Nutzen des Mittelwerts, hilf jedoch die „Unsicherheit“ (bzw. die Schwankungen) des Netzes besser darzustellen, während der Zweck der Evaluation beibehalten wird. Eindeutig ist zu erkennen, dass die hier vorgestellten Module beim vorgestellten flachen CapsNets die Klassifizierungsgenauigkeit stark verringern. Dabei zeigen die Graphen für den Loss das exakt selbe Verhalten, weshalb diese hier nicht dargestellt wurden. Außerdem führt die tiefere Version des

Deconvolution-Modul zu einem beinahe identischen Ergebnis wie die flache, was u.a. vermuten lässt, dass bereits die flache Version das „Maximum“ aus dieser Kombination herausholt. Womöglich ist es schwieriger für ein flaches Netz Capsule-Aktivitäten zu lernen, welche mittels dieser Deconvolution zu einer Ausgabe höherer Auflösung skaliert werden können. Allerdings wäre es ebenfalls denkbar, dass die hier vorgestellte Struktur des Rekonstuktions-Moduls im Gegenzug zur originalen nicht in der Lage ist auf den Features von CapsNets gut genug zu arbeiten und der daraus resultierende Rekonstruktions-Loss das CapsNet beim Lernen hindert. Die beiden Möglichkeiten klingen sehr ähnlich, jedoch zeigt das erste tiefere trainierte Modell, dass die hier vorgestellte Struktur des Rekonstuktions-Moduls durchaus einem CapsNet erlaubt, eine höhere Klassifizierungsleistung unter massiver Parametereinsparung nach ähnlich vielen Trainings-Iterationen zu erlangen. Allerdings ist um behaupten zu können, dass solch ein Modul ein komplexeres, tieferes CapsNet für gute Ergebnisse benötigt noch ein wichtiger Punkt zu beachten: Die rekonstruierten Bilder des tiefen CapsNets besitzen eine Auflösung von 64x64, die hier verwendeten Bilder eine Größe von 128x128. Weiter könnten ebenfalls die Module für die größeren Eingaben fehl-designed sein, weshalb beispielsweise die Anzahl an Filter in den ersten Layern erhöht werden könnte. Diese stellen momentan womöglich den Bottleneck dar, welcher die sehr ähnlichen Ergebnisse der beiden vorgestellten Deconvolution-Module erklären würde. Um dies Beantworten zu können müssten weitere Experimente durchgeführt werden, jedoch soll vorerst die weit wichtigere Frage geklärt werden, ob solch ein tiefes CapsNet überhaupt eine höhere Genauigkeit als das beste vorgestellte flache Netz erreichen kann.

4.4.4 Architektur und Implementierung weiterer überwachter CapsNets

Die oberen Ergebnisse legen nahe, ein tiefes CapsNets zu erstellen, welches hinsichtlich Hyperparametern dem KdefCaps-Modell ähnelt, da dieses bislang die besten Ergebnisse aufweist. Außerdem werden basierend auf diesem zweiten tiefen Netz (*KdefDeepCapsV2*), weitere Modelle durch einseitige Abwandlungen erstellt (z.B. nur die Capsule-Anzahl erhöhen), um Auswirkungen von Änderungen an tiefen CapsNets darzustellen. Die meisten Modelle folgen hierbei der vorgestellten Struktur aus Abbildung 4.2. Weiter verwenden alle folgenden CapsNets das originale Rekonstruktions-Modul, um, wie oben diskutiert, mögliche Nebenwirkungen anderer Module auszuschließen.

Modell	**Parameter der Layer**				
	Conv1	PrimaryCaps	ConvCaps1	ConvCaps2	ClassCaps
KdefDeepCapsV2	Ks: 9x9 Fi: 256 St: 4x4 Pa: Valid	Ks: 9x9 Cn: 2 Al: 4 St: 2x2 Pa: Valid	Ks: 9x9 Cn: 4 Al: 6 St: 1x1 Pa: Same	Ks: 9x9 Cn: 4 Al: 8 St: 1x1 Pa: Same	Cn: 7 Al: 16
KdefDeepCapsV3	Ks: 9x9 Fi: 256 St: 4x4 Pa: Valid	Ks: 3x3 Cn: 2 Al: 4 St: 2x2 Pa: Same	Ks: 6x6 Cn: 4 Al: 6 St: 2x2 Pa: Same	Ks: 9x9 Cn: 4 Al: 8 St: 1x1 Pa: Same	Cn: 7 Al: 16
KdefDeepCapsV4	Ks: 9x9 Fi: 256 St: 2x2 Pa: Valid	Ks: 9x9 Cn: 2 Al: 4 St: 2x2 Pa: Same	Ks: 9x9 Cn: 4 Al: 6 St: 2x2 Pa: Valid	Ks: 9x9 Cn: 4 Al: 8 St: 1x1 Pa: Same	Cn: 7 Al: 16
KdefDeepCapsV5	Ks: 9x9 Fi: 256 St: 4x4 Pa: Valid	Ks: 9x9 Cn: 4 Al: 4 St: 2x2 Pa: Valid	Ks: 9x9 Cn: 8 Al: 6 St: 2x2 Pa: Same	Ks: 9x9 Cn: 8 Al: 8 St: 1x1 Pa: Same	Cn: 7 Al: 16
KdefDeepCapsV6	Ks: 9x9 Fi: 256 St: 4x4 Pa: Valid	Ks: 9x9 Cn: 2 Al: 6 St: 2x2 Pa: Valid	Ks: 9x9 Cn: 4 Al: 8 St: 1x1 Pa: Same	Ks: 9x9 Cn: 4 Al: 16 St: 1x1 Pa: Same	Cn: 7 Al: 16
KdefDeepCapsV7	Ks: 9x9 Fi: 256 St: 2x2 Pa: Valid	Ks: 9x9 Cn: 4 Al: 4 St: 2x2 Pa: Same	Ks: 9x9 Cn: 8 Al: 6 St: 2x2 Pa: Valid	Ks: 9x9 Cn: 8 Al: 8 St: 1x1 Pa: Same	Cn: 7 Al: 16

Tabelle 4.2: Modell-Parameter der überwachten trainierten Modelle für den Datensatz Kdef: Kernel-Size (Ks), Anzahl der Filter (Fi), Anzahl der Capsules (Cn), Aktivitätsvektor-Länge (Al), Stride (St) und Padding (Pa)

4.4.5 Training und Ergebnisse der weiteren überwachten Ausgangs-CapsNets

Die Modelle werden allesamt mittels denselben Trainingsparametern wie *KdefCaps* aus Abschnitt 4.4.2 bis zur Konvergenz trainiert.

Modell	**Train**	**Eval**	**Test**
FaceCaps	96.29%	80.16%	79.70%
KdefCaps	97.43%	82.25%	81.88%
KdefDeepCaps	97.31%	80.74%	79.94%
KdefDeepCapsV2	98.36%	83.13%	80.64%
KdefDeepCapsV3	96.70%	81.56%	80.37%
KdefDeepCapsV4	98.34%	86.54%	85.59%
KdefDeepCapsV5	99.29%	85.14%	82.27%
KdefDeepCapsV6	99.12%	82.38%	80.58%
KdefDeepCapsV7	99.14%	88.15%	87.39%
KdefDeepCapsV7Leaky	99.43%	**89.19%**	**88.39%**
KdefDeepCapsV8Leaky	**99.55%**	87.26%	84.48%

Tabelle 4.3: Trainings-, Evaluations- und Testergebnisse der Modelle aus Abschnitt 4.4.1 und der Tabelle 4.2

Die Testergebnisse in Tabelle 4.3 zeigen, dass die zweite Version des tiefen CapsNets (*KdefDeepCapsV2*) bereits die flachen Netze und die vorgängige Version übertrifft (*KdefDeepCapsV1*). Dies spricht dafür, dass ein tieferes CapsNet eine bessere Leistung als eine ähnliche flache Version erbringen kann. Folglich wurde *KdefDeepCapsV2* als Basis für die Modelle *V3*-*V6* verwendet, welche aus einseitigen Abänderungen an dieser entstanden und mittels welchen untersucht wird, welche Auswirkungen die Parameteränderung an der tiefen Capsule-Struktur auf die Ergebnisse haben. So wurde für *V3* wie Tabelle 4.2 zeigt die Kernel-Size des PrimaryCaps-Layers auf 3x3 verkleinert und folglich inkrementell Layer für Layer um drei vergrößert. Grund für diese Entscheidung ist, dass sich kleinere Filter bei CNNs für viele Probleme als vorteilhaft herausgestellt haben. Weiter rührt die inkrementelle Steigung der Filter daher, dass das rezeptive Feld einer Capsule im höheren Layer womöglich zunehmen sollte, da diese komplexere Entitäten, welche aus mehreren Capsules im unteren Gitter entstehen verkörpert. Diese Verkleinerung der Filteranzahl verschlechtert jedoch das Klassifizierungsergebnis. Vielleicht benötigen die niedrigeren Capsules bereits ein größeres rezeptives Feld für den conv. Layer, um ihre Entitäten komplex genug abbilden zu können oder die Vergrößerung des rezeptiven Feldes nach oben ist zu extrem, da ebenfalls das conv. Gitter in höheren Layern durch das Padding und den Stride verkleinert wird.

Im Modell *V4* wird untersucht, ob die Features des conv. Layer das Testergebnis limitieren. Der conv. Layer besitzt einen großen Stride von 4x4 um die Eingabe zu verkleinern, damit die Performance des Netzes akzeptabel bleibt. Nun wird der Stride auf 2x2 gesetzt und somit eine größere Features-Map des conv. Layers erzeugt. Weiter wird der Stride und das Padding in den kommenden Capsule-Layern angepasst, um die entstehenden Capsule-Gitter Layer für Layer so früh wie möglich zu verkleinern. Dadurch wird das Routing beschleunigt und die Features werden in den Capsule-Aktivitäten codiert. Die Ergebnisse zeigen, dass dies die Klassifizierungsleistung um circa 3-5% in Evaluierungs- und Testdatensatz steigert. Allerdings wird die Batch-Verarbeitungsgeschwindigkeit fast gedrittelt, dem jedoch wiederum eine circa doppelt so schnelle Konvergenz etwas entgegen wirkt.
Das Modell *V5* beinhaltet eine erhöhte Anzahl von Capsule-Channel in den Primary und conv. Capsule-Layern. Durch diese Änderungen wird erhofft, dass das Modell mehrere verschiedene Capsule-Entitäten lernt und somit leichter eine Emotion klassifizieren kann, da die Capsule-Features von *V2*, welche u.a. von der Entitätenanzahl abhängen, womöglich noch nicht ausreichen. Diese Vergrößerung des Netzes führt zu einer Verbesserung der Leistung um etwa 2% gegenüber *V2*. Dabei wird die Rechenperformance im Vergleich zu *V2* kaum beeinträchtigt.
Die Änderung in Modell *V6* erhöht die Größe der Ausgabevektoren der Primary und conv. Capsule-Layer, wodurch sich jedoch das Ergebnis minimal verschlechtert. Womöglich führte diese Veränderung zu einem erhöhten Overfitting, da die Trainingsgenauigkeit leicht gestiegen ist. Allerdings muss hier erwähnt werden, dass aus zeitlichen Gründen und wegen limitiert verfügbaren Rechenressourcen jedes Modell nur einmal trainiert wurde, weshalb auch ein leicht unterschiedlich ablaufender Trainingsprozess für diese minimale Änderung verantwortlich sein könnte. Nichtsdestotrotz scheint hier eine Vergrößerung der Aktivitätsvektoren keinen großen positiven Einfluss mit sich zu bringen.
Für *KdefDeepCapsV7* werden die vielversprechendsten Änderungen von Modell *V4* und *V5* auf *V2* angewandt. Mit dem daraus resultierenden Modell wird somit die höchste Genauigkeit erreicht, welche um circa 5-7% im Evaluierungs- und Testdatensatz gegenüber *V2* gestiegen ist. Dies zeigt, dass zumindest hier die beiden Änderungen kombinierbar sind und sich sogar ihr jeweiliger Gewinn an Klassifizierungsgenauigkeit (2% und 3-5%) „aufaddiert“. Im weiteren Verlauf wird *KdefDeepCapsV7* dann mittels Leaky-Routing trainiert und ein weiterer Anstieg von 1% Genauigkeit beim Evaluierungs- und Testdatensatz erzielt. Ein Grund dafür könnte sein, dass nun Capsules mehr Hintergrundinformationen, die nichts mit der Klassifizierungsaufgabe zu tun haben, ignorieren können. Genauer könnten Capsule beispielsweise gelernt haben, die für diese Aufgabe unnützlichen Haare, welche einen großen Teil der Eingabe einnehmen, nicht mehr allzu stark zu beachten und sich folglich mehr auf die Gesichtsausdrücke zu spezialisieren.

Zu guter Letzt wird in *KdefDeepCapsV8Leaky* ein weitere Capsule-Layer direkt vor dem Class-Caps-Layer von *KdefDeepCapsV7Leaky* eingefügt. Dieser Capsule-Layer entspricht dem letzten ConvCaps2-Layer von *KdefDeepCapsV7Leaky*, nur dass die Aktivitätsvektorgröße von acht auf zehn erhöht wurde, da höhere Capsules, wie bereits erläutert, größere Ausgabevektoren besitzen sollten. Die Ergebnisse des zusätzlichen Layers zeigen dabei, ob hier ein tieferes Netz die Performance noch weiter verbessert wird. Allerdings sinken somit die Evaluierungs- und Testergebnisse um 2% und 4% hinsichtlich *KdefDeepCapsV7Leaky*. Auch hier könnte das verbesserte Klassifizierungsergebnis auf dem Trainingsdatensatz ein Indikator für erhöhtes Overfitting sein, was dann der Grund für die schlechteren Ergebnisse wäre. *KdefDeepCapsV8Leaky* zeigt zwar, dass dieser spezielle Layer bei diesem Datensatz keine weiter Verbesserung bringt, schließt jedoch keineswegs aus, dass generell noch tiefere Netze keine besseren Leistung bringen. So könnten möglicherweise Techniken gegen Overfitting oder schlankere Layer dafür sorgen, dass ein tieferes Netz als *KdefDeepCapsV7Leaky* bessere Performance liefert. Weiter könnten *KdefDeepCapsV7Leaky* um ein Layer erniedrigt werden, um zu testen ob ein etwas flacheres Netz noch besser abschneiden würde, was dann ein Indikator dafür wäre, dass flachere CapsNets zu besseren Ergebnissen tendieren. Jedoch wurde bereits gezeigt dass ein tiefes Modell besser abschneiden kann, als ein flaches (vgl. z.B. die Performance von *KdefDeepCapsV7Leaky* und *KdefCaps*). Auch wenn dies nicht ausschließt, dass ein Netz mit einem Layer weniger als *KdefDeepCapsV7Leaky* evtl. aufgrund von reduziertem Overfitting noch bessere Leistung bringt, wird hier nun ein Schlussstrich gezogen, da die Zeit verlangt sich kommenden Aufgaben zu widmen und das Training weiterer Modelle vermutlich immer weitere Fragen hervorbringen werden, weshalb dieser Teil der Arbeit so oder so mit offenen Fragen ein Ende findet. Für den kommenden Vergleich der Rekonstruktionen und um die erarbeiteten Deconvolution-Rekonstruktions-Module genauer zu testen, könnte das beste Modell (*KdefDeepCapsV7Leaky*) mit dem tiefen Deconvolution-Modul für 128x128 Rekonstruktionen trainiert werden. Dies würde womöglich die Frage klären weshalb die flachen Modelle in Abschnitt 4.4.3 durch dieses Modul stark eingeschränkt werden, jedoch erlaubt die Zeit nur noch das Training eines weiteren Modells. Da das unüberwachte Training bereits implementiert wurde und die Vermutung bezüglich Aufgabe 3 noch überprüft werden soll, wird daher das überwachte Training eingestellt und *KdefDeepCapsV7Leaky* nicht mit dem Deconvolution-Modul trainiert.

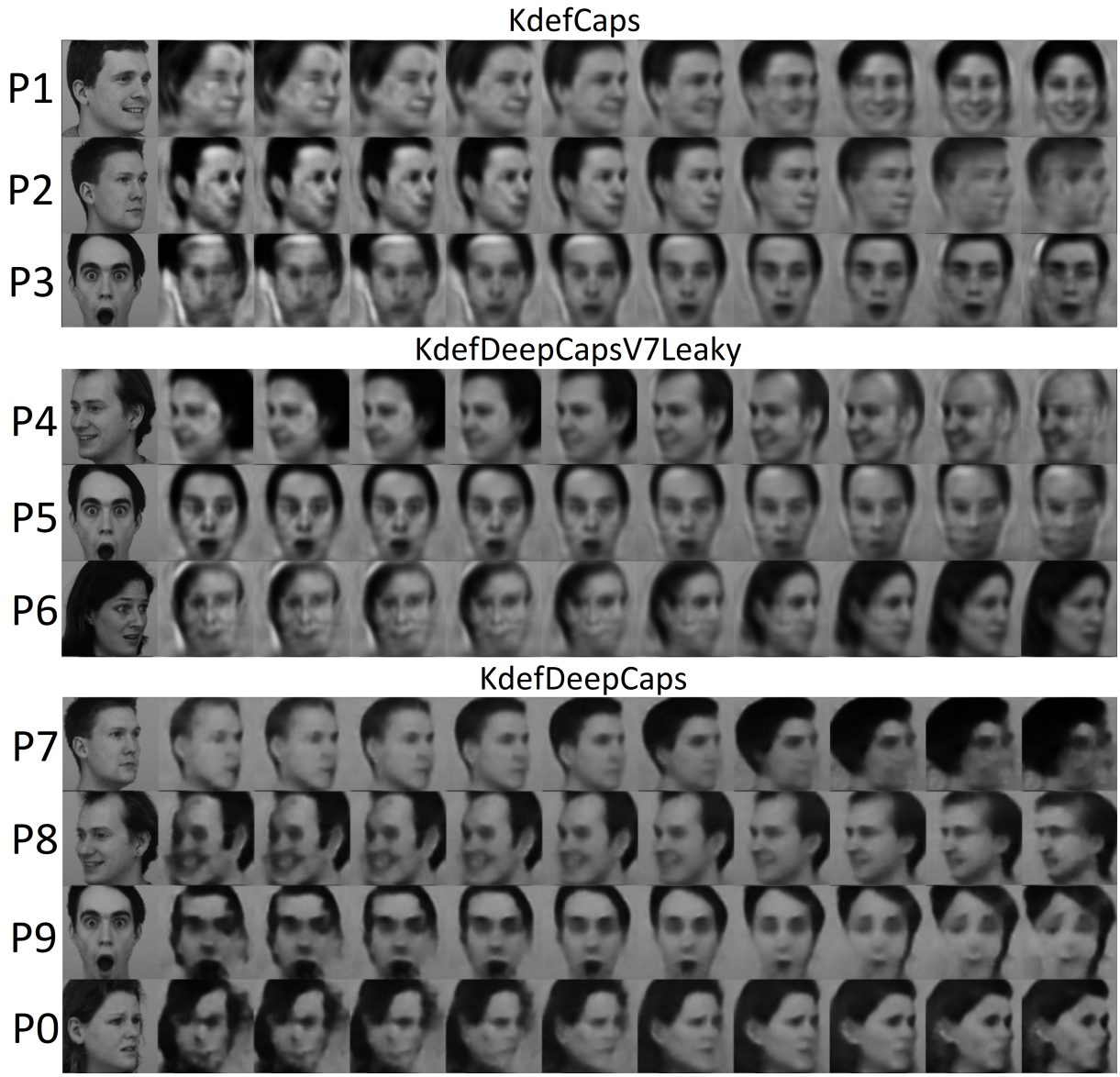


Abbildung 4.6: Rekonstruktionen der Modelle KdefCaps, KdefDeepCapsV7Leaky und KdefDeepCaps, wenn einzelne Aktivitätsvektorwerte schrittweise abgeändert werden

Um erkennen zu können, welche Features die einzelnen Aktivitätsvektorwerte verkörpern und um die Rekonstruktionen einiger Modelle zu vergleichen, wird in Abbildung 4.6 je Kategorie (flach, tief, vorgestelltes Rekonstruktions-Modul) das CapsNets gewählt, welches das beste Klassifizierungsergebnis aufweist, sprich *KdefCaps*, *KdefDeepCapsV7Leaky* und *KdefDeepCaps*. Folglich wird analog zum original CapsNet, wie in Abschnitt 3.3.5 beschrieben, jeder einzelne Wert eines Aktivitätsvektors des letzten Capsule-Layers schrittweise im Intervall -0.25 bis 0.25 um 0.05 abgeändert. Daraufhin werden die Aktivitätsvektoren wie zuvor in das Rekonstruktions-Modul gegeben, welches das rekonstruierte Bild basierend auf den abgeänder-

ten Werten erstellt.
Werden die Rekonstruktionen in Abbildung 4.6 betrachtet fällt auf, dass alle Modelle lernen verschiedene AUs in ihren Capsule-Aktivitäten zu codieren. So korrespondieren die Rekonstruktionen *P3, P5* und *P9* mit AU27 (dem weiten Öffnen des Mundes) und die Rekonstruktionen *P1* und *P8* mit AU12 (dem Anheben der Mundwinkels wie beim Lächeln oder Lachen). Diese AUs werden von Modell *KdefDeepCaps* darüber hinaus für ungewöhnlich Aktivitätsvektor-Werte (die ganz links bzw. rechts) überspitzt dargestellt und somit über ihr natürliche Auftreten hinaus gelernt (siehe *P8* und *P9*). Außerdem lernen die Modelle zwischen den verschiedenen Betrachtungswinkel zu interpolieren, was durch die Rekonstruktionen *P1, P2, P6* und *P0* verbildlicht wird. *KdefDeepCaps* fällt dabei das Lernen solcher Interpolationen am schwersten, dies wurde durch das seltenen Auftreten dieser bemerkt. Im Modell *KdefCaps* wird bei Rekonstruktion *P1* die Interpolation des Blickwinkels mit dem Grad des Lachens vermischt und der entsprechende Aktivitätsvektor-Wert codiert beide Informationen. An den Rekonstruktionen *P2, P4* und *P7* ist zu erkennen, dass die Aktivitätsvektoren Informationen über Haare codieren: so bestimmt der Aktivitätsvektor-Wert von *P4* und *P7* die Länge der Haare und der Wert von *P2* die Haarfarbe. Auch wenn das Leaky-Routing von Modell *KdefDeepCapsV7Leaky* die Ergebnisse verbessert, genügt dies laut Rekonstruktion *P4* nicht dazu die für diese Aufgabe unnötigen Information über Haare zu ignorieren, wobei dies in solchem Ausmaß nicht erwartet wurde, da die Haare eine sehr große Region der Eingabe ausmachen. Um die Qualität der erfolgreich gelernten Aktivitätsvektor-Werte untereinander quantitativ zu bewerten kann argumentiert werden, dass die 64x64 Rekonstruktionen des vorgestellten Deconvoltion-Moduls das Lernen von vielsagenden Aktivitätsvektor-Werten nicht stark negativ beeinflusst: So lernt das Modell zwar seltener Interpolationen, aber dabei muss beachtet werden, dass dieses Modell 32 Aktivitätsvektor-Werte pro Klasse aufweist anstelle von 16, was dazu führen könnte, dass solch eine Interpolation nicht stark genug durch die Einschränkung an möglichen Aktivitätsvektor-Werten forciert wurde. Darüber hinaus lernt das Modell sogar das Konzept der AUs überspitzt - ein klares Indiz dafür, dass die Aktivitätsvektor-Werte korrekt gelernt wurden. Weiter unterscheiden sich die Art der Artefakte in den Rekonstruktionen der verschiedenen Module insofern, dass die des Deconvolution-Modul „kantiger“ gegenüber den kontinuierlichen Artefakten der Modelle mit fc Modulen wirken. Allerdings treten diese Artefakte nicht häufiger auf. Durch die höhere Auflösung von 128x128 sind die Rekonstruktionen der beiden anderen Modelle schärfer, was an den rekonstruierten Augen gut erkennbar ist, da dort teilweise die Iris mit rekonstruiert wurde. Wird den Rekonstruktionen zuletzt quantitativ ein Rang zugeschrieben, dann folgt dieser der Klassifizierungsgenauigkeit des jeweiligen Modells. So scheinen die Rekonstruktionen von *KdefDeepCapsV7Leaky* am detailreichsten, gefolgt von *KdefCaps* und schließlich von *KdefDeepCaps*.

Im Folgenden wird *KdefDeepCapsV7Leaky* mit einigen aktuellen Modellen auf dem Kdef-Datensatz verglichen. Dieser Vergleich ist nicht ganz einfach, da diese Modelle in der Literatur oft unter weglassen gewisser Blickwinkel (z.B. nur frontal) trainiert werden und da für Kdef kein Evaluierungs- und Testdatensatz einheitlich spezifiziert ist, was zu unterschiedlichsten Trainings- und Teststrategien führt.

Modell	**Test**	**Kommentar**
VGG-16 (2017, [AS17])	71.4%	80-10-10 Split
ResNet50 (2017, [AS17])	73.8%	80-10-10 Split
VGG-16+ResNet50 (2017, [AS17])	75.8%	80-10-10 Split
CNN (2018, [TW18])	82.5%	3900 Trainings-, 1000 Testbilder
CNN-WAV2 (2018, [TW18])	86.1%	Ensemble: 3900 Trainings-, 1000 Testbilder
KdefDeepCapsV7Leaky	88,39%	6650 Trainings-, 1328 Testbilder
CNN-WAV4 (2018, [TW18])	93.9%	Ensemble: 3900 Trainings-, 1000 Testbilder

Tabelle 4.4: KdefDeepCapsV7Leaky im Vergleich mit dem State-of-the-Art

Tabelle 4.4 zeigt hierbei einige aktuelle Modelle, deren Training, Test und Evaluation ähnlich zu dem hier vorgestellten Vorgehen stattfand. Jedoch muss nochmal erwähnt werden, dass der Vergleich wegen geringer Unterschiede bei den Trainings- und Teststrategien nicht ganz fair ist, und dass die oberen Modelle nur einen Auszug aus der Vielzahl möglicher auf Kdef trainierter Modelle darstellt, welcher beruhend auf der Ähnlichkeit der Trainings- und Teststrategien gewählt wurde. Besonders interessant dabei ist, dass das vorgestellte Modell besser als ResNet50 und VGG-16 abschneidet und sogar deren Ensemble-Kombination schlägt. Jedoch wird das Modell um 5.5% von einem vierer Ensemble des CNN-WAV-Modells von Williams et al. [TW18] übertroffen. Da es sich hierbei jedoch um ein vierer Ensemble handelt, ist dieses Ergebnis nicht zu streng zu interpretieren, insbesondere da ein zweier Ensemble dieses CNN-WAV-Modells bereits schlechter als das tiefe CapsNet ausfällt. Weiter schlägt das CapsNet ein im Jahr 2018 trainiertes CNN um etwa 5.9%. Trotz der guten Ergebnisse wird hier weiter argumentiert, dass analog zu den Ergebnissen aus dem Theorieteil derzeit komplexere und tiefere Modelle wie ResNet-19 besser als das tiefe CapsNet abschneiden. So wird beispielsweise in Chen et al. [Che+18] im Jahr 2018 ein ResNet-19 auf Kdef mit 93.72% Testgenauigkeit gelernt. Dieses Modell wurde nicht in die obige Tabelle aufgenommen, da bei den Trainings- und Teststrategien zu viele Unterschiede existieren, welche allerdings für das ResNet-19 sprechen.

4.4.6 Diskussion der Ergebnisse

Zusammengefasst wurden sowohl tiefe als auch flache CapsNet-Modelle mit guten Ergebnissen auf Kdef trainiert, wobei durch kontrollierte Änderungen jeweils die entsprechenden Auswirkungen auf die Genauigkeit dargestellt werden konnten. Mit diesem Vorgehen wurde ein tiefes Modell trainiert, welches mit 88.39% nahe am State-of-the-Art liegt, auch wenn keine exzessive Parametersuche herangezogen wurde um die Ergebnisse noch weiter zu verbessern. So wäre es in Zukunft interessant, solch ein tiefes Modell mittels einer automatisierten Parametersuche zu untersuchen und dabei die Auswirkungen auf die Netzstruktur sowie auf die Klassifizierungsgenauigkeit darzustellen. Dabei könnten beispielsweise evolutionäre Algorithmen verwendet werden, wofür jedoch zuerst die Trainigsgeschwindigkeit dieser tiefen Modelle (und der CapsNets im Allgemeinen) erhöht werden muss.
Zusätzlich wurde für diese Aufgabe ein Deconvolution-Rekonstruktions-Modul entwickelt, welches zu einer Parametereinsparung von 87.27% bei 64x64 Eingaben und zu einer Genauigkeit von 79.94% führt, wobei nur ein einziges tiefes CapsNet trainiert und getestet wurde. Dies lässt das Modul vielversprechend erscheinen, da dieses womöglich die Genauigkeit nicht zu stark negativ beeinflusst, jedoch konnte mit dem vorgestellten Deconvolution-Rekonstruktions-Modul kein flaches CapsNet auf größere 128x128 Eingaben ohne eine starke Verringerung der Genauigkeit trainiert werden. Diese Verringerung kann viele Gründe haben und weitere Modelle müssen trainiert werden um den wahren Nutzen dieser Module zu bestimmen. So wäre es interessant in einer Reihe von Experimenten eine Vielzahl verschiedener Rekonstruktions-Module zu testen, wobei ebenfalls andere Verfahren der „Deconvolution“, sowie andere fc Rekonstruktions-Module (z.B. das „dichte“ Modul aus Abschnitt 3.7.3) untereinander verglichen werden.

4.5 Unüberwachtes Lernen eines CapsNets zur Facial-Emotion-Recognition

In diesem Abschnitt wird das *KdefDeepCapsV7Leaky*-Modell unüberwacht gelernt und hierfür vorerst das Vorgehen des unüberwachten Trainings erläutert. Schließlich wird das durch das Netz erstelle Encoding über eine SVM getestet, wobei die Capsule-Aktivitäten des letzten CapsNet-Layers das Encoding der Eingabe bilden.

4.5.1 Architektur und Implementierung des unüberwachten CapsNets

Um ein CapsNet unüberwacht zu trainieren liegt es nahe, das Training zuerst lediglich über den Rekonstuktions-Loss, also durch Weglassen des Klassifizierungs-Loss und damit der Maske für die Capsule-Aktivitäten zu bewerkstelligen. David Rawlinson et al. [RAK18] stellten hierbei fest, dass dieses Vorgehen das Netzwerk in einen Autoencoder mit einem guten Rekonstruktions-

Loss verwandelt, jedoch ausschlaggebende Capsule-Qualitäten, wie Äquivarianz, dabei verloren gehen. Damit das originale Capsule-Verhalten wieder hergestellt werden kann, ist es daher notwendig den Capsules das Spezialisieren zu erlauben, was durch das Wegfallen der Label-Maske beim unüberwachten Training über den Rekonstuktions-Loss zuvor nicht möglich war. Jede der sogenannten LatentCapsules (der früheren ClassCapsules), sollte hierbei eine Teilmenge des Eingangsbereichs entdecken, welche sie effizient modellieren und parametrisieren kann. Dies ist eine andere Art zu sagen, dass latente Capsules innerhalb eines Layers spärlich (sparsely) aktiv sein sollten. Damit die Ausgabe des latenten Capsule-Layers Spärlichkeit verzeichnet, wird dem Routing erlaubt zu bestimmen welche LatentCapsules die größten eingehenden Coupling-Koeffizienten c_{ijk} aufweisen. Hierbei ist c_{ij} wie im Original aus Abschnitt 3.3 definiert und k ist ein Index über den aktuellen Batch. Da der PrimaryCapsule-Layer - dessen Coupling-Koeffizienten in den latenten Capsule-Layer eingehen - convolutional ist, besitzt die Matrix von Coupling-Koeffizienten des PrimaryCapsule-Layers die Dimension $[K, W, H, P]$, wobei K die Batch-Size darstellt, W und H die Breite und Höhe des Capsule-Gitters und P die Anzahl der Channels, d.h. die Tiefe des PrimaryCapsule-Gitters in Capsules. Für den Routing-Support ψ_{jk} jeder LatentCapsule j in jedem Batch k besteht lediglich Interesse an den größten Coupling-Koeffizienten an jedem Platz im Gitter des PrimaryCapsule-Layers, weshalb hier das Maximum über P berechnet und schließlich über W und H aufsummiert wird. Folglich wird der Routing-Support ψ_{jk} mit einem später erläuterten Boosting-Wert $\mathbf{g}_j$ multipliziert um z_{jk} zu erhalten [RAK18]:

$$z_{jk} = \psi_{jk} \cdot \mathbf{g}_j \tag{4.1}$$

Um aus dem geboosteten Routing-Support z_{jk} eine Sparse-Mask m_{jk} für jede Capsule j zu generieren, wird dieser mit absteigendem Rang geordnet (der maximale Support erhält Rang 0)

$$r_{jk} = rank(z_k, j) \tag{4.2}$$

Danach wird die Sparse-Mask m_{jk} über die Exponentialfunktion wie folgt berechnet [RAK18]:

$$m_{jk} = e^{\gamma \frac{r_{jk}}{L-1}} \tag{4.3}$$

Hierbei ist L die Anzahl an latenten Capsules und γ bestimmt die Spärlichkeit der Ausgabe. Beispielsweise ist für $L = 16$ und $\gamma = 12$ gleichzeitig eine Capsule vollkommen aktiv und zwei andere teilweise aktiv [RAK18]. Weiter werden Mask-Werte die kleiner als 0.01 für nummerische Stabilität genullt und die Maske m elementweise mit der Capsule-Ausgabe v multipliziert, womit diese angewandt wird [RAK18]:

$$v' = v \circ m \tag{4.4}$$

v' ersetzt dann die Ausgabe v der latenten Capsules [RAK18].

Die Analyse dieses Algorithmus ohne Boosting-Faktor $\mathbf{g}_j$ zeigte, dass nur wenige Capsules für

alle Eingaben konsistent aktiv sind, wodurch die Spezialisierung auf Äquivarianz verloren geht. Um dem entgegenzuwirken, wird ein „Lifetime-Sparsity-Constraint“ hinzugefügt (Makhzani et. al [MF15]), damit sichergestellt werden kann, dass alle Capsules gelegentlich an der Ausgabe teilnehmen. Zur Vereinfachung der Implementierung wird ein Ad-hoc-Online-Boosting-System verwendet, um die Ausgabe von selten verwendeten Capsules zu erhöhen und die Ausgabe von überlasteten Capsules zu reduzieren, wenn die Rank-Zero-Frequency außerhalb eines spezifizierten Bereichs liegt [RAK18]. Die Häufigkeit jeder latenten Capsule j, welche den Rang 0 erreicht (Rank-Zero-Frequency), wird hierbei innerhalb jedes Batchs berechnet und als ein exponentiell gewichteter, gleitender Durchschnitt über viele Batchs mit einem Abfall von $\alpha = 0.99$ gemessen [RAK18]. Die Häufigkeit ϵ_j einer latenten Capsule j, welche den Rang 0 erreicht ergibt sich dann innerhalb eines Batchs aus:

$$\epsilon_j = \frac{1}{K \cdot J} \cdot \sum_k \begin{cases} 1, & \text{if } r_{jk} = 0 \\ 0, & \text{otherwise} \end{cases} \tag{4.5}$$

Wobei r_{jk} den Rang darstellt, welcher als 1 über den Batch aufsummiert wird, sofern es sich um einen Rang 0 handelt. Der aufsummierte Wert wird dann mit $\frac{1}{J \cdot K}$ normiert, wobei J die Anzahl der latenten Capsules ist. Aus ϵ_j wird schließlich ein gewichteter gleitender Durchschnitt μ'_j errechnet:

$$\mu'_j = \alpha \cdot \mu_j + (1 - \alpha) \cdot \epsilon_j \tag{4.6}$$

Hierbei ist α der bereits erwähnte Abfall mit 0.99 und μ_j der gewichtete gleitende Durchschnitt des vorherigen Schrittes mit Initialwert 0. Jüngere Samples besitzen somit einen größeren Einfluss. Nun kann der Boosting-Faktor $\mathbf{g}_j$ wie folgt berechnet werden:

Algorithm 1 Post-batch capsule boost update. n is the current batch count.

$d = 0.1$
$N = 50$
$n \leftarrow n + 1$
if $(n \bmod N) <> 0$ **then**
  return
end if
for `each latent capsule` j **do**
  if $\mu_j < \mu_{min}$ **then**
    $\mathbf{g}_j = \mathbf{g}_j + d$
  end if
  if $\mu_j > \mu_{max}$ **then**
    $\mathbf{g}_j = \max(1, \mathbf{g}_j - d)$
  end if
end for

Abbildung 4.7: Das Anwenden des Boosting-Faktors aus [RAK18]

Abbildung 4.7 zeigt die Anwendung des Boosting-Faktors in Pseudocode und gibt zu erkennen, dass alle $N = 50$ Schritte ein Boosting-Update stattfindet (*Zeile 2 bis 6*). Hierbei wird für jede latente Capsules j zunächst überprüft ob der durch Gleichung 4.6 errechnete Durchschnitt μ_j kleiner als μ_{min} oder größer als μ_{max} ist. Ist μ_j kleiner als μ_{min} wird auf den Boosting-Faktor $d = 0.1$ addiert (boost). Ist er größer als μ_{max} wird $d = 0.1$ von $\mathbf{g}_j$ abgezogen und $\mathbf{g}_j$ schließlich als das Maximum zwischen $\mathbf{g}_j - d$ und 1 gesetzt (deboost). Initial ist $\mathbf{g}_j$ gleich 1. Dabei liefert $\mu_{min} = 0.03$, $\mu_{max} = 0.12$, wenn d gleich 0.1 ist, gute Ergebnisse.
Diese Parameter wurden in [RAK18] abgeleitet, wobei eine einheitliche Frequenz als ein Mittelpunkt berechnet und dann empirisch die Grenzen optimiert werden. Genauere Angaben veröffentlichten die Autoren hierbei nicht. In Zukunft sollte es allerdings möglich sein, den Frequenzbereich abhängig von der Anzahl der Capsules zu gestalten, oder eine Online-Anpassung zu verwenden, die eine gute Ressourcennutzung sicherstellt. Mit den errechneten Werten wurde in [RAK18] folglich auf MNIST-Datensätzen gute, äquivariante Ergebnisse erzielt.
Das Boosting der latenten Capsules und das Maskieren wird hierbei nur auf die Rekonstruktions-Ausgabe angewendet, was lediglich das Training der Capsules beeinflusst. Es ist jedoch möglich, dass das zusätzliche Boosting von Coupling-Coeffizienten während der Routing-Iterationen für das Training von Vorteil wäre [RAK18].

Durch Erweiterungen am Code des besten oberen überwacht trainierten Modells wird dieses im folgenden Abschnitt wie beschrieben unüberwacht trainiert. Spannend ist hierbei, ob das Modell die Features der Gesichtsausdrücke ausreichend gut erlernt, da die Features von MNIST-Zahlen weit deutlicher sind und einen größeren Teil der Eingabe abdecken, wohingegen Features von Emotionen meist nur an speziellen Orten in der Eingabe zu finden sind und diese daher womöglich zu wenig Einfluss auf den Rekonstuktions-Loss ausüben könnten.

4.5.2 Training und Ergebnisse des unüberwachten CapsNets

Das Vorgehen und die Hyperparameter, bleiben dieselben wie beim überwachten Training aus Abschnitt 4.4.4, jedoch wird aufgrund des unüberwachten Trainings weder ein Balance-Faktor noch ein Margin-Loss benötigt. Nur der Rekonstruktions-Loss findet mit den oben erläuterten unüberwachten Techniken Verwendung. Außerdem wird der ClassCaps-Layer von 7 auf 16 Capsules erhöht, da David Rawlinson et al. [RAK18] für MNIST und affNIST (mit 40x40 Bildern) diesen Layer von 10 auf 16 erhöht, weshalb diese etwas stärkere Erhöhung von 7 auf 16 Capsules für die größere Eingabe sinnvoll scheint. Da Parameter wie μ_{min} in Abhängigkeit von der Capsule-Anzahl berechnet und das Vorgehen dabei nicht genau beschrieben wurde ergibt sich daraus ein weiterer Vorteil: Findet die Erhöhung auf 16 LatentCapsules statt, wird somit die gleiche Anzahl an Capsules verwendet, weshalb die Neuberechnung der Parameter

für eine andere Capsule-Anzahl nicht nötig ist. Weiter besagt hierbei das MDL-Prinzip das Capsule-Aktivitäten-Encoding für gute Ergebnisse hinsichtlich des anschließenden Trainings der SVM so klein wie möglich aber so groß wie nötig zu halten, wobei eine Encoding-Länge von $16 \times 16 = 256$ bereits gute Ergebnisse lieferte (siehe David Rawlinson et al. [RAK18]) und aufgrund der niedrigeren Klassenanzahl jedoch größeren Eingaben auch hier sinnvoll scheint.
Nachdem das CapsNet unüberwacht trainiert wurde, wird ein Encoding des Kdef-Datensatzes für Training, Evaluation und Test erstellt. Dabei werden die Aktivitäten der Ausgabevektoren für jedes Kdef-Bild mit dem zugehörigen Label des Bildes gespeichert, wobei nicht das gesamte Kdef-Bild codiert wird, sondern nur der zufällige, durch das Netz verwendete Ausschnitt des Bildes, dessen Zustandekommen bereits in Abschnitt 4.4.2 erläutert wurde. Da dieser Ausschnitt aus einer abgegrenzten Region der Eingabe zufällig entsteht, kann der gesamte Encoding-Datensatz durch mehrfaches Encodieren des Kdef-Datensatzes ebenfalls künstlich erweitert werden. Hierbei wird der erweiterte Kdef-Datensatz der Größe 9640 insgesamt 30 mal durchlaufen, was zu einem Encoding-Datensatz der Größe $289\,200$ führt. Die Aufteilung in Trainings-, Evaluations- und Testdatensatz ist hierbei für einen fairen Vergleich die exakt gleiche wie in Abschnitt 4.4.2, nur dass diese jeweils 30-fach codiert werden.
Im Anschluss wird eine SVM auf dem Encoding trainiert, wobei die Parameter dieselben bleiben wie in David Rawlinson et al. [RAK18], da die Kdef-Encoding-Größe der des MNIST-Encoding entspricht.

Modell	**Train**	**Test**
KdefDeepCapsV7Leaky	**99.43%**	**88.39%**
KdefDeepCapsV7LeakyUnsup	76.97%	39.04%

Tabelle 4.5: Trainings- und Testergebnisse einer SVM auf dem Encoding des unüberwachten Modells im Vergleich

Der Vergleich der Klassifizierungsergebnisse auf dem Encoding mit dem besten überwachten Modell in Tabelle 4.5 zeigen klar, dass wie vermutet das überwacht trainierte CapsNet besser abschneidet. Dabei konnte, um bestimmen zu können wann das unüberwachte Training beendet werden soll, nur unter großem Aufwand die Klassifizierungsergebnisse herangezogen werden, da für jede Evaluation das gesamte Encoding erstellt und eine SVM trainiert werden muss. Daher wurden Model-Checkpoints stichprobenartig während des Trainings vom Cloud-Service heruntergeladen und die Erstellung des Encoding sowie das Training der SVM lokal durchgeführt solange bis die Testgenauigkeit durch Overfitting kontinuierlich abstieg. Folglich pendelten sich weitere Stichproben auf $68\,000$ Trainings-Iterationen ein und eine maximale Testgenauigkeit von 39.04% bei einer Trainingsgenauigkeit von 76.97% wurde erreicht. Die Distanz

zwischen Trainings- und Testergebnissen fällt somit im Vergleich zu *KdefDeepCapsV7Leaky* weit größer aus, was ein klares Indiz für erhöhtes und verfrüht eintretendes Overfitting darstellt. Dieses Overfitting könnte auf die kleinen ROIs der Eingabe zurückgeführt werden, was nach dem Vergleich der Rekonstruktionen genauer diskutiert wird.

Abbildung 4.8: Rekonstruktionen des Modelles KdefDeepCapsV7LeakyUnsup, wenn einzelne Aktivitätsvektor-Werte schrittweise abgeändert werden

Abbildung 4.8 stellt die Rekonstruktionen des unüberwachten Modells dar, wenn analog zu Abschnitt 4.4.5 die Aktivitätsvektor-Werte schrittweise abgeändert werden. Hierbei muss vorerst erwähnt werden, dass es durch die geringe Klassifizierungsgenauigkeit schwierig ist, auf den Testdaten vielsagende Aktivitätsvektor-Werte zu finden. Allerdings wurden beispielsweise mittels der Rekonstruktionen *U1* und *U2* zwei Werte gefunden, welche AUs codieren (für *U1* AU27, das weite Öffnen des Mundes, und für *U2* AU9, das Rümpfen der Nase, sowie AU15, das Herabziehen der Mundwinkel). Jedoch wurden Aktivitätsvektor-Werte für Interpolationen zwischen den Blickwinkeln nicht gefunden. Ein möglicher Grund hierfür ist, dass dem Rekonstuktions-Modul beim unüberwachten Training die Aktivitätsvektoren jeder Ausgabe-Capsule zur Verfügung stehen, anstelle nur der Vektor der korrekten Klasse. Zwar werden niedrige Aktivitätsvektor-Werte für numerische Stabilität genullt, dennoch ist nicht zwangsläufig ein Aktivitätsvektor-Wert vonnöten, der die Blickwinkel-Information codiert, da das Nullen der Werte die Aktivitätsvektor-Werte nicht daran hindert nur groß zu werden, wenn eine gewisser Blickwinkel vorliegt. Somit würde die Blickwinkel-Information nicht in den Werten kodiert werden, sondern anhand der Position der aktiven Aktivitätsvektor-Werte im Layer. Dementgegen gelang die Interpolation der Position der Ziffer bei MNIST und affNIST in David Rawlinson et al. [RAK18], jedoch ist diese weit einfacher als die grobe Interpolation für die wenigen verschiedenen Blickwinkel in der weit größeren Eingabe des Kdef-Datensatzes.

4.5.3 Diskussion der Ergebnisse

Wie bereits zu Beginn vermutet fallen die Ergebnisse des unüberwachten Trainings auf Kdef schlechter aus als die des überwachten CapsNets. Da diese Methode mit derselben SVM auf MNIST und affNiST jedoch sehr gute Ergebnisse liefert und das Encoding in etwa die gleiche

Größe wie das Kdef-Encoding besitzt, kann gefolgert werden, dass dies mit der Art der Eingabe zusammenhängt. Der Unterschied zwischen Kdef und MNIST ist einerseits offensichtlich die Größe der Bilder, welche sicher bereits dem unüberwachten Training erschwert, ausschlaggebende Capsule-Entitäten zu lernen welche über den Großteil des Datensatzes in verschiedenen Eingaben genutzt werden können und nicht zu stark auf einzelne Bilder bzw. Personen spezialisiert sind. Der zweite, nicht ganz triviale Unterschied, ist das Aufkommen der Features in der Eingabe an sich. So existieren die für die Klassifizierung ausschlaggebenden ROIs einer Emotion nur in einem sehr kleinen Teil der Eingabe, wohingegen sich eine Zahl über den kompletten Eingaberaum erstreckt. Dadurch, dass das unüberwachte CapsNet hauptsächlich auf dem Rekonstuktions-Loss arbeitet, wird das Netz daher nicht dazu geleitet diese ROIs zu entdecken, da es nicht stark genug von der Zielfunktion bestraft wird, wenn dieses beispielsweise den Mund, welcher einen sehr kleinen Teil der Eingabe ausmacht, nicht richtig rekonstruiert. Dabei zeigen die überwacht gelernten Netze, dass bereits das Einbeziehen des Labels in die Zielfunktion dem CapsNet ausreicht, um diese Regionen zu finden. Hier hingegen kann das Netz erst in viel späteren Trainigs-Iterationen die gesamte Eingabe gut genug rekonstruieren, damit das unpräzise Darstellen einer kritischen ROI (z.B. der Mund) ausschlaggebend bestraft wird. Jedoch ist dabei das Netz sehr wahrscheinlich bereits auf andere Features (wie die Haare) zu stark overfittet. Sicher kann durch eine andere Trainingsstrategie oder Zielfunktion das Problem bewältigt werden, allerdings wird dies in dieser Arbeit aus Zeitgründen nicht mehr untersucht.

Unabhängig von der Lösung dieses Problems entstand während der Arbeit eine Idee hinsichtlich der Erweiterung des Encodings. Diese Methode wird kurz am Rande erwähnt, da die Experimente hierbei noch keine eindeutigen Ergebnisse lieferten und die Zeit nicht erlaubt im Rahmen dieser Arbeit sich tiefer und länger damit zu beschäftigen. Um das Encoding des obigen unüberwacht gelehrten CapsNets zu erweitern können die x höchsten Coupling-Koeffizienten der Routing-Verbindungen miteinbezogen werden. Bildlich wird hierbei das „Aufleuchten“ des CapsNets genutzt, welches vermutlich nützliche Informationen enthält. Diese Informationen befinden sich zwar bereits indirekt in dem Aktivitätsvektoren, aber könnten womöglich - und vor allem bei tieferen CapsNets - über die Layer hinweg in den Aktivitätsvektor-Werten zu sehr verschwimmen.

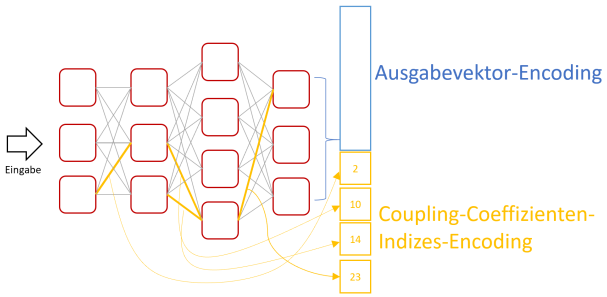


Abbildung 4.9: Skizze: Erweiterung des Encodings um die höchsten Coupling-Coeffizienten-Indizes

Abbildung 4.9 skizziert die Erweiterung des Encodings, wobei schlichtweg die Indizes der x höchsten Coupling-Koeffizienten an das im oberen Abschnitt 4.5 beschriebenen Encoding gehängt werden. Hierbei handelt es sich nur um ein Konzept, da bei Primary- und ConvCapsules die Verbindungsmuster eine andere Struktur besitzen, welche jedoch keine negativen Auswirkungen haben sollte. Für das Erstellen der Coupling-Koeffizienten-Indizes wird eine eindeutig fortgehende Nummerierung der Coupling-Koeffizienten vorgenommen. Somit wird einen Nutzen daraus gezogen, dass die Coupling-Koeffizienten nachvollziehbare Pfade durch das CapsNet bilden. Sollten diese Pfade für unterschiedliche Eingaben derselben Klasse ähnlich sein, könnte das Encoding von dem daraus entstehenden Muster profitieren. Das zusätzliche Anhängen der entsprechenden Coupling-Koeffizienten-Werte führt evtl. zu weiteren Performance-Verbesserungen, da somit der Grad des „Leuchtens“ mit einbezogen wird.
Erste Experimente zeigen hierbei, dass die Nutzung der jeweils 16, 64 und 128 höchsten Coupling-Koeffizienten auf MNIST und Kdef zu erhöhten Trainings- aber leicht niedrigeren Testergebnissen führt, da vermutlich verschiedene Coupling-Koeffizienten-Konstelationen durch ihre Eindeutigkeit und ihr geringes Auftreten für stärkeres Overfitting sorgen. Hierbei wurde beispielsweise auf dem 30-fach codierten Kdef-Datensatz die Trainingsgenauigkeit beim Nutzen der 128 höchsten Coupling-Koeffizienten auf 81.35% erhöht, wobei der Testerror um 0.49% auf 38.45% sank. Werden weniger Coupling-Koeffizienten verwendet so steigt die Trainingsgenauigkeit nicht allzu stark an und die Testgenauigkeit sinkt weniger (z.B. 77.06% und 39.01% bei 16 Koeffizienten). Das Hinzuziehen des Grades des „Leuchtens“ führte zu keinen ausschlaggebenden Veränderungen. Jedoch handelt es sich hierbei nur um erste Ergebnisse und weit ausführlichere Experimente sind für eine klare Aussage vonnöten. So könnten im weiteren Verlauf beispielsweise die Coupling-Koeffizienten niedrigerer Layer nicht im Encoding mit einbezogen werden, da diese verschiedenen einfachen Entitäten zugeordnet sind, welche vermutlich oft und nicht spezifisch in der Eingabe auftreten. Durch dieses Vorgehen wird ausschließlich das „Aufleuchten“ des generelleren Teils des Pfades einer zu klassifizierenden Entität gelernt und womöglich das Overfitting reduziert.

5 Zusammenfassung, Diskussion und Ausblick

Zu Beginn der Arbeit wurde die Motivation dargelegt und mit dieser die aktuelle Lage im Bereich des Deep Learning kurz diskutiert, woraus einige Gründe für die Erörterung und Anwendung von Capsule Networks hervorkamen. Die CapsNets befanden sich - und befinden sich nach wie vor - jedoch in einem frühen Entwicklungsstadium, weshalb noch kein Standardmodell existiert. Daher fand in der darauf folgenden theoretischen Erörterung zuerst die Erarbeitung der generalisierten Idee anhand der bis zu diesem Kenntnisstand gesamten Literatur statt, wobei zuerst die Capsules, die Struktur des CapsNets, sowie der Routing-Algorithmus einzeln und in Gesamtheit betrachtet wurden. In diesem Part wurde ebenfalls die Darstellung der neurowissenschaftlichen Inspirationen und Parallelen vorgenommen; genauer wurde hierbei die corticalen Minicolums und Makrocolumns, die Two-streams Hypothesis, sowie das Place- und Rate-Coding erläutert und die generalisierte Idee mit diesen verglichen. Außerdem fand ein Vergleich mit einer interessanten neu aufkommenden Entwicklung bezüglich des Neuronenmodells und die Darstellung einer Parallele hinsichtlich einer Computer-Grafik-Perspektive statt. Weiter wurden einige introspektive Experimente verbildlicht, welche einige Funktionsweisen und Grenzen des menschlichen Sehvermögens darlegten und ebenfalls als Intuition hinter CapsNets gelten.
Nachdem mittels der generalisierten Idee ein einheitlicher Rahmen für die chronologische Erklärung aller Capsule-Modelle geschaffen wurde, konnte zunächst der Transforming-Autoencoder durchleuchtet werden. Hierbei wurde - wie bei den Folgemodellen - der schematische Aufbau in den zuvor erstellten einheitlichen Rahmen gebracht und das Netz anhand diesem erläutert. Nach der Erörterung verschiedener Anwendungen dieses Modells wurden in der Diskussion einige Schwächen dargelegt: So wird beispielsweise die Anwendung der Transformationsmatrizen extern vorgenommen und das Ergebnis als zusätzliche Eingabe den Capsules überreicht.
Das daraufhin ausführlich erörterte CapsNet mit dynamischem Routing-by-Agreement löste dieses Problem und stellt bislang - besonders wegen der Klarheit der Hyperparameter und der Zugänglichkeit zum original Code - das in der Literatur am häufigsten verwendete Modell dar. Auch hier wurde der schematische Aufbau durchleuchtet. Da dieses Modell das bereits in Transforming-Autoencoders angekündigte Routing umsetzt, fand daraufhin die Darstellung des Routing-Algorithmus statt. Nachdem die Architektur und das Routing mit genauer Parameterangabe dargelegt wurde, konnte die Anwendung auf MNIST erläutert werden. Dabei wurde gezeigt, dass das CapsNet State-of-the-Art-Performance bei dessen Klassifizierung erreicht und herausragende Ergebnisse bei der Segmentierung zweier überlappender MNIST-Zahlen aufweist. Weiter wurde dargelegt, dass auch die Ergebnisse auf smallNORB, einem kleinen Set von SVHN sowie Cifar10 für den aktuellen Entwicklungsstand beachtlich sind und dem

State-of-the-Art entsprechen. In anderen Teilen der Arbeit stellte sich dann heraus, dass diese Architektur häufig in der Literatur nahezu eins zu eins übernommen wird und dadurch für die meisten Problemstellungen mit kleinen Eingaben gute bis sehr gute Ergebnisse erzielt wurden. Auch hier fand letztlich die Diskussion der Ergebnisse des Modells statt, wobei u.a. festgestellt wurde, dass dieses CapsNet ebenfalls das Crowding, welches bei introspektiven Experimenten erläutert wurde, nicht löst und nicht zu lösen versucht. Auch keines der Folgemodelle nimmt sich diesem Problem an, wobei vermutet wird, dass das kluge Zusammenschließen mehrerer CapsNets eine Lösung darstellt.
Basierend auf den Ergebnissen des Modells mit dynamischem Routing-by-Agreement wurden folglich von den Schöpfern der CapsNets die komplexeren Matrix Capsules entwickelt, welche ebenfalls im selben Schema erörtert wurden. So wurde zu Beginn deren Erörterung u.a. dargelegt, dass das Nutzen einer negativen, logarithmierten Varianz eines Gauß-Clusters beim Berechnen der Votes, im Gegenzug zum Nutzen des Kosinus bei original CapsNets, zu einer feineren Berechnung des Agreements führt. Folglich wurde das EM-Routing mittels zweier Abänderungen am EM-Algorithmus erläutert und dabei beispielsweise die Parallelen zur freien Energie und Boltzmann-Verteilung in der Physik dargestellt. Bei der genauen Betrachtung der Architektur wurde jedoch festgestellt, dass nicht alle Hyperparameter und keine Referenzimplementierung der Autoren preisgegeben wurden - ein Grund weshalb die komplexen Matrix Capsules bislang wenig Einsatz fanden. Der Einsatz des Modells von Autorenseite auf smallNorb zeigte jedoch vielversprechende Ergebnisse. Allerdings lag bei der folgenden Diskussion das Hauptaugenmerk auf der Effizienz von CapsNets, welche verbesserungswürdig und für den langfristigen Erfolg dieser Modelle unabdingbar ist.
Beruhend auf diesen First-Party-Erkenntnissen konnten dann zahlreiche Ergebnisse anderer Quellen bezüglich CapsNets erörtert werden. Hierbei wurden kurz weitere Capsule-Modelle (darunter RNN-, GAN-, Segmentierungs- und Video-CapsNets), weitere und erweiterte Routing-Algorithmen, andere Erweiterungen der Modelle, verschiedene Klassifizierungen, NLP mittels CapsNets und einige Auswertungen und Diskussionen der Erklärbarkeit von CapsNets erläutert. Durch konstruktive Diskussion und durch die Durchleuchtung dieser Veröffentlichungen kamen zahlreiche offene Fragen und Probleme ans Tageslicht. Eine dieser Fragen - ob der original Routing-Algorithmus bei einer Capsule im letzten Layer sinnvoll ist - wurde folglich im praktischen Teil beantwortet. Außerdem wurde festgestellt, dass einige Modelle bereits das Effizienz-Problem spezifisch lösen. Im weiteren Verlauf der Arbeit explodierte jedoch die Anzahl an Publikationen, weshalb einige gelesene Veröffentlichungen tabellarisch noch weiter zusammengefasst wurden und die Betrachtung der Literatur zu einem bestimmten Zeitpunkt beendet werden musste, um sich dem praktischen Teil zu widmen.
Mit einem Vergleich von CapsNets und CNNs, in welchem u.a. geschlussfolgert wurde, dass

CapsNets das CNN-Modell nicht ersetzen, sondern auf diesem aufbauen, sowie das CapsNet das Effizienz-Problem für ihren Erfolg in einem Standardmodell lösen sollte, folgte schließlich der praktische Teil der Arbeit.

Im praktischen Teil der Arbeit wurden zunächst die drei Aufgabenstellungen festgelegt und das Entwicklungsumfeld sowie die Vorgehensweise dargestellt. Hierbei konnten u.a. einige Verbindungen zu einem Cloud-Service aufgebaut werden, der das Training tiefer CapsNet-Modelle ermöglichte.
Nachdem der Coupling-Koeffizienten-Test anhand einer einfachen Klassifizierungsaufgabe zeigte, dass, wie vorweg theoretisch erklärt, das Routing mit einer Capsule in letzten Layer nicht sinnvoll ist, wurden für die Klassifizierung von Emotionen menschlicher Gesichter sowohl flache, als auch tiefe CapsNet-Modelle erarbeitet und implementiert. Dabei fand eine einstige Abänderung eines tiefen CapsNet-Modells statt, wodurch die Auswirkungen hinsichtlich der Klassifizierungsgenauigkeit auf dem Kdef-Datensatz betrachtet werden konnten. Mit diesem Vorgehen wurde dann Schritt für Schritt ein tiefes Modell erarbeitet, welches weit besser als die vorgestellten flachen Modelle abschneidet und auch das originale CapsNet hinsichtlich der Performance weit übertrifft. Hierbei wurde ebenfalls ein neues Rekonstruktions-Modul entwickelt, welches zu einer Parametereinsparung von 87.27% bei 64x64 Eingaben führt und gute Ergebnisse liefert, allerdings aus zu gering untersuchten Gründen bei 128x128 Eingabe unter Verwendung flacher Modelle die Genauigkeit zu stark verringert.
Das beste tiefe CapsNet wurde in der letzten Aufgabenstellung auf demselben Datensatz unüberwacht trainiert, wobei die Capsule-Aktivitäten der Layer (insbesondere des letzten Layers) nur unter Verwendung des Rekonstruktion-Loss und ohne den leitenden Margin-Loss spärlich gehalten werden mussten. Dies wurde zwar erreicht, jedoch fiel die Klassifizierungsgenauigkeit aufgrund der Eigenschaften des Kdef-Datensatzes, wie zuvor vermutet, nicht gut aus. Hauptproblem hierbei ist vermutlich, dass Kdef im Gegenzug zu MNIST sehr kleine ROIs besitzt.
Zusätzlich wurden erste Ergebnisse hinsichtlich der Erweiterung des vom unüberwachten CapsNet erstellen Encodings um das „Aufleuchten“ des Netzes dargestellt. Hierbei wurden die Indizes der x höchsten Coupling-Koeffizienten an das Encoding der Ausgabevektoren gehängt und die Auswirkungen der Klassifizierung einer SVM auf dem erstellten Encoding betrachtet. Dabei zeigen erste Experimente noch keine eindeutigen Ergebnisse und Hinweise auf Overfitting, da jedoch die Trainingsergebnisse auf dem MNIST- und Kdef-Datensatz dadurch verbessert wurden, scheint es sinnvoll das Verfahren zu erweitern und weiter zu testen. Diese Folgerung schloss dann, wie fast jeder Abschnitt der Arbeit, den praktischen Teil mit vielen offenen Fragen.

Der Ausblick bezüglich CapsNet scheint im Rahmen dieser Arbeit bereits offensichtlich: So werden bestimmt einige der offenen Fragen von der Community geklärt und die Arbeit am Effizienz-Problem wird weiter ihren Lauf nehmen. Wird dieses gelöst, stellen die CapsNets ein bemerkenswertes neues Modell im Deep Learning dar, welche in einem breiten Feld eingesetzt werden können und einige aktuelle Forderungen an den Bereich durch die Struktur großteils erfüllen. Jedoch muss erwähnt werden, dass nicht alle Forderungen durch diese Struktur bereits erfüllt wurden, so zeigen CapsNets beispielsweise immer noch große Schwächen hinsichtlich Black-Box-Angriffen durch Generativ-Adversarial-Examples mittels eines CNNs.
Wird das Effizienzproblem nicht gelöst, sind CapsNets jedoch immer noch bemerkenswert: Gründe dafür sind die Gedankengänge, sowie das Vorgehen und die Erkenntnisse hinter deren Entwicklung. Allerdings können CapsNets dann, bis auf ein paar spezielle Ausnahmen, nur für kleine Eingaben Verwendung finden. Dies könnte deren Anwendung als Bausteine in Layern großer Hyrid-Modelle nicht im Wege stehen, doch deren Etablierung als eigenständiges Modell: So ist es in diesem Fall nicht möglich CapsNets auf Datensätzen wie ImageNet zu trainieren, was die Anwendung für aktuelle Bereiche und Problemstellungen mit großen Datensätzen - wie beispielsweise autonomes Fahren - kaum möglich macht oder stark erschwert. Ob oder ob nicht das Effizienz-Problem allgemein gelöst wird scheint daher die derzeit spannendste Frage, welche von den Schöpfern der CapsNets bereits angegangen wird, da dies in der Veröffentlichung der Matrix Capsules als nächster Schritt gilt. Womöglich widerfährt den CapsNets dabei ein ähnliches Schicksal wie den CNNs, deren Erfolg erst viele Jahre nach deren „Entdeckung“ stattfand, nachdem die Rechenleistung und die benötigte Datenmenge aufholte. Jedoch liefern CapsNet im Gegenzug zur CNN-Geschichte bereits sehr gute Ergebnisse auf kleinen Datensätzen mit kleinen Eingaben, weshalb die heutigen Datensatzgrößen die CapsNets vermutlich nicht limitieren. Weiter existieren in der Literatur bereits effiziente Modelle für einzelne Aufgaben, wodurch die Lösung des Effizienzproblems in naher Zukunft möglich scheint.

Die Zusammenfassung und der Ausblick zeigen, dass die Forschung und Entwicklung der CapsNets noch lange nicht vorbei ist. Viele Fronten finden Interesse an diesem Modell und den bereits zahlreichen Veröffentlichungen schließen sich viele weitere an - ein typisches Muster im Deep Learning. So kann bereits zu diesem Zeitpunk nur mit hohem Zeitaufwand jeder Erkenntnis hinsichtlich CapsNets gefolgt werden, was weiter das starke Interesse an diesem Modellen sowie am Bereich des Deep Learning zeigt. Abschließend kann daher geschlussfolgert werden, dass nach wie vor eine Vielzahl an aufschlussreichen Erkenntnissen aus diesem Gebiet zu erwarten sind, wobei das Schicksal der CapsNets sprichwörtlich eine Frage der Zeit ist.

6 Anhang

6.1 Glossar

AU	Facial-Action-Unit
BCE	Binary-Cross-Entropy-Loss
CapsNet	Capsule Network
CNN	Convolutional Neural Network
Colab	Google's Colaboratory
EM	Expectation-Maximization engl. für Erwartung-Maximierung
EW	Erweiterung
FC	Fully Connected engl. für voll verbunden
FGSM	Fast Gradient Sign Method
GAN	Generative Adversarial Network
GRU	Gated Recurrent Unit
GTSRB	German Traffic Sign Recognition Benchmark
HSI	Hyperspectral Image engl. für Hyperspektrale Bilder
IP	Indian Pines
KA	Klassifizierung
Kdef	Karolinska Directed Emotional Faces
KSC	Kennedy Space Center
LSTM	Long-Short-Term-Memory
MDL	Minimum Description length, engl. für minimale Beschreibungslänge
MLP	Multilayer Perceptron
MNIST	Modified National Institute of Standards and Technology

MRI	Magnetic Resonance Imagin
NLP	Natural Language Processing
NM	Neues Modell
NPTN	Non-Parametric Transformation Network
PU	Pavia University scene
RA	Routing-Algorithmus
RNN	Recurrent Neural Network engl. für rekurrentes neuronales Netz
ROI	Region of Interest
SA	Salinas scene
SIFT	Scale-invariant feature transform
SLIC	Simplelinear Iterative Clustering
SOTA	State-of-the-Art
SVHN	Street View House Numbers
SVM	Support Vector Machine
TN	Transformation Network
UAV	Unmanned Aerial Vehicle engl. für unbemanntes Fluggerät

6.2 Verzeichnisse

6.2.1 Abbildungen

3.1 Lösungsneutaler Aufbau einer Capsule 4
3.2 Routingkonzept eines Capsule Networks aus [Kot] 8
3.3 Nicht übereinstimmende Vorhersagen für die Pose der 4 aus [Kot] 8
3.4 Blickwinkel invariante Vorhersagen von Capsule A und B für Capsule C im Bezug zu Computergrafik nach [Ghtb] 10
3.5 Dorsaler und ventraler Strom skizziert nach [Idv] und [Pfa] 13
3.6 Schematische Darstellung des neuen Neuronenmodells [Sar+17] 15
3.7 Rechteck und Diamant [Roc73] 18

3.8 Unbekannte Form wird nach Mitteilung dass diese rotiert wurde durch mentale Rotation zu Afrika, aus [Roc73] 19
3.9 Ein rotiertes R aus [Ghtb] bzw. [Ghta] 20
3.10 Schematischer Aufbau eines Transforming-Autoencoders aus [Hin11, S. 3] . . 21
3.11 Links: Scatterplot mit Verschiebungsfehler der Recognition Unit einer Capsule. Rechts: die ausgehenden Gewichte verschiedener Generation Units mehrere Capsules aus [Hin11, S. 3] 23
3.12 Vollständige Ausgabe der affin transformierten Bilder eines Transforming-Autoencoders aus [Hin11, S. 5] 24
3.13 Vergleich der Ausgabe eines Transforming-Autoencoder für Stereobilder von verschiedener Fahrzeugtypen von verschiedenen Blickwinkeln [Hin11, S. 6] . . 25
3.14 Auswirkungen des Crowdings aus [PAT08] 26
3.15 Schematischer Aufbau einer Capsule mit iterativem Routing-by-Agreement . . 28
3.16 Graph eines Non-linear Squashing in Skalarform aus [Pecb] 30
3.17 Eine Capsule i trifft Vorhersagen und bewertet diese (durch Routing-by-Agreement) inspiriert von [Ghtb] 33
3.18 Der Routing-by-Agreement-Algorithmus aus [SS17, S. 3] 34
3.19 Einfache Capsule Network Architektur für MNIST [SS17, S. 4] 36
3.20 Die 32 6x6-Gitter der 8D-Capsules bzw. die 6x6x8 Primary-Capsules für jeden der 32 Channels 37
3.21 Der farbcodierte Margin-Loss nach [Pecc] 39
3.22 Decoder Struktur der Capsule-Network-Architektur für MNIST aus [SS17, S. 4] 40
3.23 Links: Durchschnittliche Änderung der Routing-Logits für MNIST. Rechts: CapsNet-Loss in Abhängigkeit der Routing-Iterationen für Cifar10 aus [SS17, S. 11] 41
3.24 Beispiele decodierter Bilder nach steigenden leichten Veränderungen an den Werten der Aktivitätsvektoren aus [SS17, S. 6] 42
3.25 Beispiele von Rekonstruktionen eines CapsNet mit drei Routing-Iterationen auf MultiMNIST [SS17, S. 4] 44
3.26 Schematischer Aufbau einer Matrix Capsule mit EM-Routing 48
3.27 (a) Ein Modell einer gaußschen Mischverteilung mit drei Komponenten, (b) Sample Datenpunkte des Modells aus (a) und (c), das durch EM rekonstruierte Modell aus den Daten von (b). Nach [Rus+10, S. 818] 50
3.28 Capsules in Layer L als Gaußverteilung mit Sicht auf die transformierten Mittelwerte bzw. Erwartungswert der Capsules in $L + 1$ 55
3.29 Die farblich eingeteilte Zielfunktion der Matrix Capsules 61

3.30 EM-Routing Pseudocode [HSF18a, S. 3] 62
3.31 Architektur des Matrix Capsule Models mit EM-Routing aus [HSF18a, S. 4] . . 65
3.32 Der Effekt der Variation verschiedener Komponenten der CapsNet-Architektur für smallNORB aus [HSF18a, S. 5] 69
3.33 Histogramme von Distanzen der Votes zu den Mittelwerten jeder der fünf finalen Capsules aus [HSF18a, S. 6] 70
3.34 Vergleich von EM-CapsNet mit Baseline-CNN aus [HSF18a, S. 7] 72
3.35 Links Genauigkeit gegenüber ϵ einer adversarial Attack. Rechts: Erfolgsrate nach einer adversarial Attack 73
3.36 Details der S-Capsule Architektur aus [Bah18] 76
3.37 Trainingskurven von S- und EM-Capsules aus [Bah18] 78
3.38 CelebA-Vergleich des CapsuleGAN mit einem herkömmlichen GAN aus [PW18]. Links: Ergebnisse des CapsuleGANs. Rechts: Ergebnisse des GANs 81
3.39 MNIST-Vergleich der Ergebnisse der CapsuleGANs mit einem herkömmlichen GAN. Oben: Das CapsuleGAN aus [AJ18] Mitte: Das CapsuleGAN aus [PW18]. Unten: Das GAN aus [AJ18] sehr ähnlich zu [PW18] 81
3.40 Architektur des RNN-Capsule Networks aus [Wan+18b] 83
3.41 Aufbau einer einzelnen RNN-Capsule aus [Wan+18b] 83
3.42 SegCaps Architektur für die Objekt Segmentierung aus [RL18] 85
3.43 Pseudocode des Rounting-Algorithmus der SegCaps mit Einschränkung eines benutzerdefinierten Kernels aus [RL18] 86
3.44 SegCaps Rekonstruktionen von Lungen mit schrittweise abgeänderten Capsule-Vektor Werden aus [RL18] 88
3.45 Architektur des VideoCapsuleNets aus [DRS18] 89
3.46 Das originale Routing-by-Agreement als Loss einer Clusterbildung kombiniert mit einem KL-Divergenz-Regularisierungsterm aus [DW18] 90
3.47 Optimierter Routing-Algorithmus aus [DW18] 92
3.48 Vergleich der Algorithmen zur Formen-Gruppierung aus [DW18] 94
3.49 Operation eines Knoten des Non-Parametric Transformation Network aus [PS18] 95
3.50 Die Architektur des Dense Capsule Networks (DCNets) aus [SSRP18] 97
3.51 Die Architektur des Diverse Capsule Networks (DCNet++) aus [SSRP18] . . . 98
3.52 Gehirntumor Rekonstuktionen und Features durch leicht veränderte Ausgabevektor-Werte aus [AMP18] 100
3.53 Pipeline der Bildverarbeitung der Fehlererkennung und Bildfusion in Stromversorgungssystemen [Li+18a] 101
3.54 Architektur des Capsule Networks für Reiserkennung aus [Li+18b] 103

3.55 Vergleich des Capsule Networks mit CNN und SVM aus [Li+18b] 104
3.56 Links: die Rohdaten von HSI (a); Rechts: Würfeldaten mit verschiedenen Nachbarschaften (b) aus [Luo+18] 104
3.57 Gesamtgenauigkeit der Modelle für vier nicht weiter beschriebene HSI-Datensätze: Kennedy Space Center (KSC), Indian Pines (IP), Pavia University scene (PU) und Salinas scene (SA). Aus [Luo+18] 105
3.58 Synthetisierte Bilder, erhalten durch Abänderung einzelner Dimensionen des Aktivitätsvektors von AU12 aus [OEJC18] 107
3.59 Okklusionssensitivitätskarten überlagern mit der Eingabe für die Posen 1-9 von AU7 und AU12 aus [OEJC18] 108
3.60 Ergebnisse verschiedener dem Original ähnelnden CapsNet-Modelle auf Cifar10 aus [EX17] 110
4.1 Coupling-Koeffizienten von den Modellen mit (a) einer Capsule und einer Routing-Iteration, (b) einer Capsule und drei Routing-Iterationen, (c) einer Capsule und drei Leaky-Routing-Iteration, (d) zwei Capsules und drei Routing-Iterationen im letzten Layer 122
4.2 Allgemeine Struktur der entwickelten tiefen CapsNets 125
4.3 Durch Deconvolution auftretende Überlappungen im zweidimensionalen Raum aus [OD16] 127
4.4 Rekonstruktions-Modul aus fc, convolutional, resize und pad Layern 128
4.5 Trainings- und Evaluations-Klassifizierungsgenauigkeit für das vorgestellte flache KdefCapsNet mit jeweils einem der drei Rekonstruktions-Module nach $50\,000$ Trainings-Iterationen 132
4.6 Rekonstruktionen der Modelle KdefCaps, KdefDeepCapsV7Leaky und KdefDeepCaps, wenn einzelne Aktivitätsvektorwerte schrittweise abgeändert werden 138
4.7 Das Anwenden des Boosting-Faktors aus [RAK18] 143
4.8 Rekonstruktionen des Modelles KdefDeepCapsV7LeakyUnsup, wenn einzelne Aktivitätsvektor-Werte schrittweise abgeändert werden 146
4.9 Skizze: Erweiterung des Encodings um die höchsten Coupling-Coeffizienten-Indizes 148

6.2.2 Tabellen

4.1 Trainings-, Evaluations- und Testergebnisse der Modelle 131
4.2 Modell-Parameter der überwachten trainierten Modelle für den Datensatz Kdef: Kernel-Size (Ks), Anzahl der Filter (Fi), Anzahl der Capsules (Cn), Aktivitätsvektor-Länge (Al), Stride (St) und Padding (Pa) 134

4.3 Trainings-, Evaluations- und Testergebnisse der Modelle aus Abschnitt 4.4.1 und der Tabelle 4.2 135
4.4 KdefDeepCapsV7Leaky im Vergleich mit dem State-of-the-Art 140
4.5 Trainings- und Testergebnisse einer SVM auf dem Encoding des unüberwachten Modells im Vergleich 145
6.1 Tabellarische Zusammenstellung der restlichen gefundenen wissenschaftlichen Literatur. Hinzukommende Akronyme sind wie folgt definiert: neues Modell (NM), Klassifizierung (KA), Erweiterung (EW), Routing-Algorithmus (RA) und State-of-the-Art (SOTA) 171

6.3 Quellen

[Abd] Hervé Abdi. „The Eigen-Decomposition: Eigenvalues and Eigenvectors“. In: *The University of Texas at Dallas* (). Online erhältlich unter https://www.utdallas.edu/~herve/Abdi-EVD2007-pretty.pdf; zuletzt abgerufen am 03.09.2018.

[Ach+10] Radhakrishna Achanta u. a. „SLIC Superpixels“. In: (2010), S. 15.

[AJ18] Yue Wu Premkumar Natarajan Ayush Jaiswal Wael AbdAlmageed. „CapsuleGAN: Generative Adversarial Capsule Network“. In: (2018). Online erhältlich unter https://arxiv.org/abs/1802.06167; zuletzt abgerufen am 23.05.2018. arXiv: 1803.09093. URL: https://arxiv.org/abs/1802.06167.

[AMP18] Parnian Afshar, Arash Mohammadi und Konstantinos N. Plataniotis. „Brain Tumor Type Classification via Capsule Networks“. In: *CoRR* abs/1802.10200 (2018). arXiv: 1802.10200. URL: http://arxiv.org/abs/1802.10200.

[AO93] H Anderson C C Van Essen D A Olshausen B. „A neurobiological model of visual attention and invariant pattern recognition based on dynamic routing of information“. In: 13 (Dez. 1993), S. 4700–19.

[Aob] „An Optimization View on Dynamic Routing Between Capsules“. In: *OpenReview* (). Online erhältlich unter https://openreview.net/forum?id=HJjtFYJDf; abgerufen am 18.07.2018.

[AS17] James Wong Alexandru Savoiu. „Recognizing Facial Expressions Using Deep Learning“. In: (2017). Online erhältlich unter https://pdfs.semanticscholar.org/59b5/0e396e657627cc503a0747c5b84bd17c5468.pdf; zuletzt abgerufen am 03.09.2018.

[Bah18] Mohammad Taha Bahadori. „Spectral Capsule Networks“. In: *Workshop track - ICLR 2018* (2018). https://openreview.net/forum?id=HJuMvYPaM.

[Bd] *Boltzmann distribution*. Online erhältlich unter https://en.wikipedia.org/wiki/Boltzmann_distribution; zuletzt abgerufen am 03.09.2018.

[BKC15] Vijay Badrinarayanan, Alex Kendall und Roberto Cipolla. „SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation“. In: *CoRR* abs/1511.00561 (2015). arXiv: 1511.00561. URL: http://arxiv.org/abs/1511.00561.

[Che+18] Yunhua Chen u. a. „Robust Expression Recognition Using ResNet with a Biologically-Plausible Activation Function“. In: *Image and Video Technology*. Hrsg. von Shin'ichi Satoh. Cham: Springer International Publishing, 2018, S. 426–438. ISBN: 978-3-319-92753-4.

[Chi+18] Zhixiang Chi u. a. „Single Image Reflection Removal Using Deep Encoder-Decoder Network“. In: *CoRR* abs/1802.00094 (2018). arXiv: 1802.00094. URL: http://arxiv.org/abs/1802.00094.

[Cir+11] Dan C. Ciresan u. a. „High-Performance Neural Networks for Visual Object Classification“. In: *CoRR* abs/1102.0183 (2011). arXiv: 1102.0183. URL: http://arxiv.org/abs/1102.0183.

[CMS12] Dan C. Ciresan, Ueli Meier und Jürgen Schmidhuber. „Multi-column Deep Neural Networks for Image Classification“. In: *CoRR* abs/1202.2745 (2012). arXiv: 1202.2745. URL: http://arxiv.org/abs/1202.2745.

[Col] *Hello, Colaboratory*. Online erhältlich unter https://colab.research.google.com/notebooks/welcome.ipynb; zuletzt abgerufen am 03.09.2018.

[Dcf] „Summation“. In: *DocCheck Flexikon* (). Online erhältlich unter http://flexikon.doccheck.com/de/Summation; zuletzt abgerufen am 03.09.2018.

[DRS18] Kevin Duarte, Yogesh Singh Rawat und Mubarak Shah. „VideoCapsuleNet: A Simplified Network for Action Detection“. In: *CoRR* abs/1805.08162 (2018).

[DW18] Qiang Liu Dilin Wang. „An Optimization View on Dynamic Routing Between Capsules“. In: *Workshop track - ICLR 2018* (2018). Online erhältlich unter https://openreview.net/pdf?id=HJjtFYJDf; zuletzt abgerufen am 22.05.2018.

[EX17] Yang Jin Edgar Xi Selina Bing. „Capsule Network Performance on Complex Data“. In: (2017). URL: https://arxiv.org/abs/1712.034802.

[FA91] W. T. Freeman und E. H. Adelson. „The design and use of steerable filters“. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 13.9 (1991), S. 891–906. ISSN: 0162-8828. DOI: 10.1109/34.93808.

[GFE18] Jascha Sohl-Dickstein Gamaleldin F. Elsayed Ian Goodfellow. „Adversarial Reprogramming of Neural Networks“. In: (2018). URL: https://arxiv.org/abs/1806.11146.

[Ghta] *Geoffrey Hinton: "Does the Brain do Inverse Graphics?"* Online erhältlich unter https://www.youtube.com/watch?v=TFIMqt0yT2I; zuletzt abgerufen am 03.09.2018.

[Ghtb] *Geoffrey Hinton talk "What is wrong with convolutional neural nets?"* Online erhältlich unter https://www.youtube.com/watch?v=rTawFwUvnLE&feature=youtu.be; zuletzt abgerufen am 03.09.2018.

[GK18] Andrew Gritsevskiy und Maksym Korablyov. „Capsule networks for low-data transfer learning“. In: *CoRR* abs/1804.10172 (2018). arXiv: 1804.10172. URL: http://arxiv.org/abs/1804.10172.

[GM] Karen Grace-Martin. „What is a Logit Function and Why Use Logistic Regression?“ In: (). Online erhältlich unter https://www.theanalysisfactor.com/what-is-logit-function/; zuletzt abgerufen am 03.09.2018.

[Gon+18] Jingjing Gong u. a. „Information Aggregation via Dynamic Routing for Sequence Encoding“. In: 2018. URL: https://www.semanticscholar.org/paper/Information-Aggregation-via-Dynamic-Routing-for-Gong-Qiu/80cdde16e086fadacbd3bee37a98bfc664d79a4a.

[Gru14] Peter Grunwald. „A tutorial introduction to the minimum description length principle“. In: (2014). Online erhältlich unter https://arxiv.org/abs/math/0406077; zuletzt abgerufen am 03.09.2018.

[GSS15] Ian Goodfellow, Jonathon Shlens und Christian Szegedy. *Explaining and harnessing adversarial examples*. Jan. 2015.

[Har+17] Hrayr Harutyunyan u. a. „Multitask Learning and Benchmarking with Clinical Time Series Data“. In: (März 2017).

[Hin] Geoffrey Hinton. *What is wrong with convolutional neural nets?* Online erhältlich unter https://www.youtube.com/watch?v=Jv1VDdI4vy4; zuletzt abgerufen am 03.09.2018.

[Hin00] Ghahramani Zoubin Teh Yee Whye Hinton Geoffrey E. „Learning to Parse Images“. In: *Advances in Neural Information Processing Systems 12*. Hrsg. von S. A. Solla, T. K. Leen und K. Müller. MIT Press, 2000, S. 463–469. URL: http://papers.nips.cc/paper/1710-learning-to-parse-images.pdf.

[Hin11] Krizhevsky Alex Wang Sida D. Hinton Geoffrey E. „Transforming Auto-Encoders“. In: *Artificial Neural Networks and Machine Learning – ICANN 2011*. Hrsg. von Duch Włodzisław Girolami Mark Kaski Samuel Honkela Timo. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, S. 44–51. ISBN: 978-3-642-21735-7.

[Hin79] Geoffrey Hinton. „Some Demonstrations of the Effects of Structural Descriptions in Mental Imagery*“. In: 3 (Juli 1979), S. 231 –250.

[Hin81a] Geoffrey Hinton. „Shape representation in parallel systems“. In: 2 (1981).

[Hin81b] Geoffrey F. Hinton. „A Parallel Computation That Assigns Canonical Object-based Frames of Reference“. In: *Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 2*. IJCAI'81. Vancouver, BC, Canada: Morgan Kaufmann Publishers Inc., 1981, S. 683–685. URL: http://dl.acm.org/citation.cfm?id=1623264.1623282.

[Hoc18] Phileas Hocquard. „Exploring the Use of Capsules for Sequential Tasks“. In: (Mai 2018). Online erhältlich unter http://www.phileas.me/static/exploring_sequential_capsules.pdf; zuletzt abgerufen am 23.05.2018.

[HSF18a] Geoffrey E Hinton, Sara Sabour und Nicholas Frosst. „Matrix capsules with EM routing“. In: *International Conference on Learning Representations*. 2018. URL: https://openreview.net/forum?id=HJWLfGWRb.

[HSF18b] Geoffrey E Hinton, Sara Sabour und Nicholas Frosst. „Matrix capsules with EM routing Version 1“. In: *International Conference on Learning Representations*. 2018. URL: https://openreview.net/forum?id=HJWLfGWRb.

[Hua+16] Xiaowei Huang u. a. „Safety Verification of Deep Neural Networks“. In: *CoRR* abs/1610.06940 (2016). arXiv: 1610.06940. URL: http://arxiv.org/abs/1610.06940.

[HZ94] Geoffrey E Hinton und Richard S. Zemel. „Autoencoders, Minimum Description Length and Helmholtz Free Energy“. In: *Advances in Neural Information Processing Systems 6*. Hrsg. von J. D. Cowan, G. Tesauro und J. Alspector. Morgan-Kaufmann, 1994, S. 3–10. URL: http://papers.nips.cc/paper/798-

autoencoders-minimum-description-length-and-helmholtz-free-energy.pdf.

[IA18] Tomas Iesmantas und Robertas Alzbutas. „Convolutional capsule network for classification of breast cancer histology images“. In: (2018). URL: https://arxiv.org/abs/1804.08376.

[Idv] *Image showing dorsal stream (green) and ventral stream (purple) in the human brain visual system*. Online erhältlich unter https://en.wikipedia.org/wiki/Two-streams_hypothesis; zuletzt abgerufen am 03.09.2018.

[JB14] Koray Kavukcuoglu Jimmy Ba. Volodymyr Mnih. „Multiple Object Recognition with Visual Attention“. In: *CoRR* abs/1412.7755 (2014). arXiv: 1412.7755. URL: http://arxiv.org/abs/1412.7755.

[Jn] *Jupyter*. Online erhältlich unter http://jupyter.org/; zuletzt abgerufen am 03.09.2018.

[JS92] Ria De Bleser Josephine Semmes. „Visual Agnosia: A Case of Reduced Attentional “Spotlight”?“ In: *Cortex* 28.4 (1992), S. 601 –621. ISSN: 0010-9452. DOI: https://doi.org/10.1016/S0010-9452(13)80230-4. URL: http://www.sciencedirect.com/science/article/pii/S0010945213802304.

[Jég+16] Simon Jégou u. a. „The One Hundred Layers Tiramisu: Fully Convolutional DenseNets for Semantic Segmentation“. In: *CoRR* abs/1611.09326 (2016). arXiv: 1611.09326. URL: http://arxiv.org/abs/1611.09326.

[KGB16] Alexey Kurakin, Ian J. Goodfellow und Samy Bengio. „Adversarial examples in the physical world“. In: *CoRR* abs/1607.02533 (2016). arXiv: 1607.02533. URL: http://arxiv.org/abs/1607.02533.

[Kot] Tanay Kothar. *Uncovering the Intuition behind Capsule Networks and Inverse Graphics: Part I*. Online erhältlich unter https://hackernoon.com/uncovering-the-intuition-behind-capsule-networks-and-inverse-graphics-part-i-7412d121798d; zuletzt abgerufen am 03.09.2018.

[Kum18] Amara Dinesh Kumar. „Novel Deep Learning Model for Traffic Sign Detection Using Capsule Networks“. In: (2018). Online erhältlich unter https://arxiv.org/abs/1805.04424; zuletzt abgerufen am 22.05.2018.

[Len+16] Jiabing Leng u. a. „Cube-CNN-SVM: A Novel Hyperspectral Image Classification Method“. In: *2016 IEEE 28th International Conference on Tools with Artificial Intelligence (ICTAI)* (2016), S. 1027–1034.

[Li+08] M. J. Li u. a. „Agglomerative Fuzzy K-Means Clustering Algorithm with Selection of Number of Clusters“. In: *IEEE Transactions on Knowledge and Data Engineering* 20.11 (2008), S. 1519–1534. ISSN: 1041-4347.

[Li+18a] Yu Li u. a. „Image fusion of fault detection in power system based on deep learning“. In: *Cluster Computing* (2018). ISSN: 1573-7543. DOI: 10.1007/s10586-018-2264-2. URL: https://doi.org/10.1007/s10586-018-2264-2.

[Li+18b] Yu Li u. a. „The recognition of rice images by UAV based on capsule network“. In: (März 2018).

[Luo+18] Yanan Luo u. a. „HSI-CNN: A Novel Convolution Neural Network for Hyperspectral Image“. In: *CoRR* abs/1802.10478 (2018). arXiv: 1802.10478. URL: http://arxiv.org/abs/1802.10478.

[Mat+13] M. Mathias u. a. „Traffic sign recognition - How far are we from the solution?“ In: *The 2013 International Joint Conference on Neural Networks (IJCNN)*. 2013, S. 1–8. DOI: 10.1109/IJCNN.2013.6707049.

[Meh] Nikhil Mehta. *Transforming-Autoencoder-TF*. Online erhältlich unter https://github.com/nikhil-dce/Transforming-Autoencoder-TF; zuletzt abgerufen am 03.09.2018.

[MF15] Alireza Makhzani und Brendan J Frey. „Winner-Take-All Autoencoders“. In: *Advances in Neural Information Processing Systems 28*. Hrsg. von C. Cortes u. a. Curran Associates, Inc., 2015, S. 2791–2799. URL: http://papers.nips.cc/paper/5783-winner-take-all-autoencoders.pdf.

[MJ15] Andrew Zisserman Koray Kavukcuoglu Max Jaderberg Karen Simonyan. „Spatial Transformer Networks“. In: *CoRR* abs/1506.02025 (2015). arXiv: 1506.02025. URL: http://arxiv.org/abs/1506.02025.

[MWE] Mark T. Keane Michael W. Eysenck. *Cognitive Psychology: A Student's Handbook*. Taylor und Francis Ltd. ISBN: 1848724160. URL: https://books.google.de/books?id=G-3KrQEACAAJ.

[NG17] Aran Nayebi und Surya Ganguli. „Biologically inspired protection of deep networks from adversarial attacks“. In: (März 2017).

[Ngu+18] Dai Quoc Nguyen u. a. „A Capsule Network-based Embedding Model for Search Personalization“. In: *CoRR* abs/1804.04266 (2018). arXiv: 1804.04266. URL: http://arxiv.org/abs/1804.04266.

[O'N18] James O'Neill. „Siamese Capsule Networks“. In: (Mai 2018). URL: https://www.researchgate.net/publication/325262901_Siamese_Capsule_Networks.

[OD16] Augustus Odena und Chris Dumoulin Vincent and. „Deconvolution and Checkerboard Artifacts“. In: *Distill* (2016). Online erhältlich unter http://distill.pub/2016/deconv-checkerboard; zuletzt abgerufen am 03.09.2018. DOI: 10.23915/distill.00003.

[OEJC18] Itir Onal Ertugrul, Laszlo A. Jeni und Jeffrey F. Cohn. „FACSCaps: Pose-Independent Facial Action Coding With Capsules“. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*. 2018.

[OMS17] Chris Olah, Alexander Mordvintsev und Ludwig Schubert. „Feature Visualization“. In: *Distill* (2017). Online erhältlich unter http://distill.pub/2016/deconv-checkerboard; zuletzt abgerufen am 03.09.2018. DOI: 10.23915/distill.00007.

[Oor+16] Aäron van den Oord u. a. „WaveNet: A Generative Model for Raw Audio“. In: *CoRR* abs/1609.03499 (2016). arXiv: 1609.03499. URL: http://arxiv.org/abs/1609.03499.

[Pal] Andrea Palazzi. *Transforming-Autoencoders*. Online erhältlich unter https://github.com/ndrplz/transforming-autoencoders; zuletzt abgerufen am 03.09.2018.

[PAT08] Denis Pelli und Katharine A Tillman. „The Uncrowded Window of Object Recognition“. In: 11 (Nov. 2008), S. 1129–1135.

[Peca] Max Pechyonkin. *Understanding Hinton's Capsule Networks. Part I: Intuition*. Online erhältlich unter https://medium.com/ai^3-theory-practice-business/understanding-hintons-capsule-networks-part-i-intuition-b4b559d1159b; zuletzt abgerufen am 04.09.2018.

[Pecb] Max Pechyonkin. *Understanding Hinton's Capsule Networks. Part II: How Capsules Work*. Online erhältlich unter https://medium.com/ai^3-theory-practice-business/understanding-hintons-capsule-networks-part-ii-how-capsules-work-153b6ade9f66; zuletzt abgerufen am 04.09.2018.

[Pecc] Max Pechyonkin. *Understanding Hinton's Capsule Networks. Part IV: CapsNet Architecture*. Online erhältlich unter https://medium.com/@pechyonkin/part-iv-capsnet-architecture-6a64422f7dce; zuletzt abgerufen am 03.09.2018.

[Pfa] *Principal fissures and lobes of the cerebrum viewed laterally*. Online erhältlich unter https://commons.wikimedia.org/wiki/File:Gray728.svg; zuletzt abgerufen am 03.09.2018.

[PS18] Dipan K. Pal und Marios Savvides. „Non-Parametric Transformation Networks". In: *CoRR* abs/1801.04520 (2018). arXiv: 1801.04520. URL: http://arxiv.org/abs/1801.04520.

[PW18] Mathijs Pieters und Marco Wiering. „Comparing Generative Adversarial Network Techniques for Image Creation and Modification". In: *CoRR* abs/1803.09093 (2018). arXiv: 1803.09093. URL: http://arxiv.org/abs/1803.09093.

[Qia+18] Kai Qiao u. a. „Accurate reconstruction of image stimuli from human fMRI based on the decoding model with capsule network architecture". In: *CoRR* abs/1801.00602 (2018). arXiv: 1801.00602. URL: http://arxiv.org/abs/1801.00602.

[RAK18] David Rawlinson, Abdelrahman Ahmed und Gideon Kowadlo. „Sparse Unsupervised Capsules Generalize Better". In: *CoRR* abs/1804.06094 (2018). arXiv: 1804.06094. URL: http://arxiv.org/abs/1804.06094.

[Rei+01] E. Reinhard u. a. „Color transfer between images". In: *IEEE Computer Graphics and Applications* 21.5 (2001), S. 34–41. ISSN: 0272-1716.

[RFB15] Olaf Ronneberger, Philipp Fischer und Thomas Brox. „U-Net: Convolutional Networks for Biomedical Image Segmentation". In: *CoRR* abs/1505.04597 (2015). arXiv: 1505.04597. URL: http://arxiv.org/abs/1505.04597.

[Rh18] Vincent Renkens und Hugo Van hamme. „Capsule Networks for Low Resource Spoken Language Understanding". In: (2018). URL: https://arxiv.org/abs/1805.02922.

[RL18] Ulas Bagci Rodney LaLonde. „Capsules for Object Segmentation". In: (2018). Online erhältlich unter https://arxiv.org/abs/1804.04241; zuletzt abgerufen am 23.05.2018.

[RM18] Juan Carrillo Rinat Mukhometzianov. „CapsNet comparative performance evaluation for image classification". In: (Mai 2018). URL: https://arxiv.org/ftp/arxiv/papers/1805/1805.11195.pdf.

[Roc73] I. Rock. *Orientation and form*. Academic Press, 1973. ISBN: 9780125912501. URL: https://books.google.de/books?id=hgQEAQAAIAAJ.

[RS18] Sal Vivona Raeid Saqur. „CapsGAN: Using Dynamic Routing for Generative Adversarial Networks“. In: (Juni 2018). URL: https://arxiv.org/pdf/1806.03968.pdf.

[RSG16] Marco Túlio Ribeiro, Sameer Singh und Carlos Guestrin. „"Why Should I Trust You?": Explaining the Predictions of Any Classifier“. In: *CoRR* abs/1602.04938 (2016). arXiv: 1602.04938. URL: http://arxiv.org/abs/1602.04938.

[Rus+10] S.J. Russell u. a. *Artificial Intelligence: A Modern Approach*. Prentice Hall series in artificial intelligence. Prentice Hall, 2010. ISBN: 9780136042594. URL: https://books.google.de/books?id=8jZBksh-bUMC.

[Saba] Sara Sabour. *Code for Capsule model used in the following paper: "Dynamic Routing between Capsules"by Sara Sabour, Nickolas Frosst, Geoffrey E. Hinton*. Online erhältlich unter https://github.com/Sarasra/models/tree/master/research/capsules; zuletzt abgerufen am 03.09.2018.

[Sabb] Sara Sabour. *Dynamic routing between capsules*. Online erhältlich unter https://www.youtube.com/watch?v=gq-7HgzfDBM; zuletzt abgerufen am 03.09.2018.

[Sar+17] Shira Sardi u. a. „New Types of Experiments Reveal that a Neuron Functions as Multiple Independent Threshold Units“. In: *Scientific Reports*. Online erhältlich unter https://www.nature.com/articles/s41598-017-18363-1; zuletzt abgerufen am 03.09.2018. 2017.

[Sie18] Miller KD Jemal A. Siegel RL. „Cancer statistics, 2018“. In: *CA Cancer J Clin. 68:7- 30* (2018). Online erhältlich unter https://onlinelibrary.wiley.com/doi/full/10.3322/caac.21442; zuletzt abgerufen am 03.09.2018.

[SMP18] Atefeh Shahroudnejad, Arash Mohammadi und Konstantinos N. Plataniotis. „Improved Explainability of Capsule Networks: Relevance Path by Agreement“. In: *CoRR* abs/1802.10204 (2018). arXiv: 1802.10204. URL: http://arxiv.org/abs/1802.10204.

[SS17] Geoffrey E. Hinton Sara Sabour Nicholas Frosst. „Dynamic Routing Between Capsules“. In: *CoRR* abs/1710.09829 (2017). arXiv: 1710.09829. URL: http://arxiv.org/abs/1710.09829.

[SSRP18] Abhinav Dhall Deepti Bathula Sai Samarth R Phaye Apoorva Sikka. „Dense and Diverse Capsule Networks: Making the Capsules Learn Better“. In: (Mai 2018). URL: https://arxiv.org/pdf/1805.04001.pdf.

[Sta+12] J. Stallkamp u. a. „Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition“. In: *Neural Networks* 32 (2012). Selected Papers from IJCNN 2011, S. 323 –332. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2012.02.016. URL: http://www.sciencedirect.com/science/article/pii/S0893608012000457.

[SV18] Zhi-Li Zhang Saurabh Verma. „Graph Capsule Convolutional Neural Networks“. In: (Apr. 2018). URL: https://arxiv.org/pdf/1805.08090.pdf.

[Tdf] *Thermodynamic free energy*. Online erhältlich unter https://en.wikipedia.org/wiki/Thermodynamic_free_energy; zuletzt abgerufen am 03.09.2018.

[Tra14] Anne Trafton. „In the blink of an eye“. In: *MIT News* (Jan. 2014). Online erhältlich unter http://news.mit.edu/2014/in-the-blink-of-an-eye-0116; zuletzt abgerufen am 03.09.2018.

[TW18] Robert Li Travis Williams. „An Ensemble of Convolutional Neural Networks Using Wavelets for Image Classification“. In: *Journal of Software Engineering and Applications* (2018). Online erhältlich unter https://www.researchgate.net/publication/322957424_An_Ensemble_of_Convolutional_Neural_Networks_Using_Wavelets_for_Image_Classification; zuletzt abgerufen am 03.09.2018.

[Wan+13] Li Wan u. a. „Regularization of Neural Networks using DropConnect“. In: *Proceedings of the 30th International Conference on Machine Learning*. Hrsg. von Sanjoy Dasgupta und David McAllester. Bd. 28. Proceedings of Machine Learning Research 3. Atlanta, Georgia, USA: PMLR, 2013, S. 1058–1066. URL: http://proceedings.mlr.press/v28/wan13.html.

[Wan+18a] Qi Wang u. a. „An attention-based Bi-GRU-CapsNet model for hypernymy detection between compound entities“. In: *CoRR* abs/1805.04827 (2018). arXiv: 1805.04827. URL: http://arxiv.org/abs/1805.04827.

[Wan+18b] Yequan Wang u. a. „Sentiment Analysis by Capsules“. In: (Apr. 2018).

[WZ18] Min Yang Zeyang Lei Suofei Zhang Zhou Zhao Wei Zhao Jianbo Ye. „Investigating Capsule Networks with Dynamic Routing for Text Classification“. In: 2 (Apr. 2018). URL: https://arxiv.org/pdf/1804.00538.pdf.

[YL18] Chuanhui Shan Yujian Li. „A Unified Framework of Deep Neural Networks by Capsules“. In: (Mai 2018). URL: https://arxiv.org/pdf/1805.03551.pdf.

[YU18] Paul Schrater Yash Upadhyay. „Generative Adversarial Network Architectures For Image Synthesis Using Capsule Networks“. In: (Juni 2018). URL: https://arxiv.org/pdf/1806.03796.pdf.

[ZEQ18] Liheng Zhang, Marzieh Edraki und Guo-Jun Qi. „CapProNet: Deep Feature Learning via Orthogonal Projections onto Capsule Subspaces“. In: *CoRR* abs/1805.07621 (2018). arXiv: 1805.07621. URL: http://arxiv.org/abs/1805.07621.

[ZF13] Matthew D. Zeiler und Rob Fergus. „Visualizing and Understanding Convolutional Networks“. In: *CoRR* abs/1311.2901 (2013). arXiv: 1311.2901. URL: http://arxiv.org/abs/1311.2901.

[Zha+18a] Suofei Zhang u. a. „Fast Dynamic Routing Based on Weighted Kernel Density Estimation“. In: *CoRR* abs/1805.10807 (2018). arXiv: 1805.10807. URL: http://arxiv.org/abs/1805.10807.

[Zha+18b] Wei Zhao u. a. „Investigating Capsule Networks with Dynamic Routing for Text Classification“. In: *CoRR* abs/1804.00538 (2018). arXiv: 1804.00538. URL: http://arxiv.org/abs/1804.00538.

[Zha11] B. Zhang. „Breast cancer diagnosis from biopsy images by serial fusion of Random Subspace ensembles“. In: *2011 4th International Conference on Biomedical Engineering and Informatics (BMEI)*. Bd. 1. 2011, S. 180–186. DOI: 10.1109/BMEI.2011.6098229.

[Zim] Urs Zimmermann. *Definition: Sakkaden*. Online erhältlich unter https://eyetracking.ch/glossar-sakkade/; zuletzt abgerufen am 03.09.2018.

[Zou+18] Xianli Zou u. a. „Fast Convergent Capsule Network with Applications in MNIST“. In: *Advances in Neural Networks –ISNN 2018*. Hrsg. von Tingwen Huang u. a. Cham: Springer International Publishing, 2018, S. 3–10. ISBN: 978-3-319-92537-0.

6.4 Inhaltsverzeichnis der Medien CD

/Masterarbeit in PDF

Die vorliegende Masterarbeit im PDF-Format.

/Grafiken
Sowohl die selbst erstellten, als auch die von Quellen entnommenen Grafiken.

/LaTEX
Die LaTEX-Dokumente der vorliegenden Masterarbeit.

/Quellcode
Die TensorFlow-1.6-Implementierung in Python 3.6 für alle ausgeführten Experimente, sowie die erstellte CapsNet-Bibliothek. Das Python-Notebook für den Cloud-Service Collaboratory und der Code von Experimenten, die aus verschiedenen Gründen in der Arbeit keine Erwähnung fanden, sind dort ebenfalls enthalten.

/Quellen
Sämtliche in der Arbeit verwendeten Paper zu CapsNets, sowie zusätzliche neuere Veröffentlichungen. Außerdem finden sich dort sämtliche weitere frei verfügbare Literatur, z.B. zur Facial Emotion Recognition, und die PDF-Version der unter 6.2.2 aufgeführten Internetsquellen.

6.5 Appendix

6.6 Tabellarische Zusammenstellung der restlichen gefundenen wissenschaftlichen Literatur

Kategorien	Stichwörter	Wertung	Quelle
CapProNet: Deep Feature Learning via Orthogonal Projections onto Capsule Subspaces			
NM, KA	Capsule-Projection-Net, Capsule-Subspace-Group-Learning, Caps+ResNet, outperforms SOTA, Cifar10, Cifa1100, SVHN.	Positiv	[ZEQ18]
Graph Capsule Convolutional Neural Networks			
NM, KA	Graph-Klassifizierung, outperforms SOTA, 6 Bioinformatik Datensätze, sehr mathematisch	Positiv	[SV18]
Siamese Capsule Networks			
EW, NM, KA	Original, MNIST, Cifar10, smallNOR, Face-Verification, Pairwise-Learning, Few-Shot-Learning, AMSGrad-Optimization, performs well	Positiv	[O'N18]
Information Aggregation via Dynamic Routing for Sequence Encoding			

RA, KA	Sequence-Encoding, reversed dynamic Routing, Softmax anders angewandt, RNN/CNN-Encoding-Layer, 5 Text-KA-Task	Positiv	[Gon+18]
Fast Dynamic Routing Based on Weighted Kernel Density Estimation			
RA, KA	multivariate Gaußverteilung, EM-Routing Ähnlichkeit zu Matrix-Caps, 40% Routing Zeiteffizienz, Hybrid conv. & caps. Layer, 64x64 Pixels, smallNORB, parallel Performance, KDE, mean shift	Positiv	[Zha+18a]
Capsule networks for low-data transfer learning			
NM, KA	Generalisierung durch wenige Beispiele, 25 mal schneller als ähnliche CNNs, multiMNIST, 1-100 Bilder der fehlenden Zahl nach Training nötig, Injektion neuer Informationen hilft Netz seine Informationen verm. besser zu sortieren alles wenn es alle bekommt, memo	Neutral	[GK18]
Accurate reconstruction of image stimuli from human fMRI based on the decoding model with capsule network architecture			
KA	Bücke zw. menschl. fmri und image stimuli, Mapping von CapsNet-Featurs zu fmri, MNIST, Structural-Similarity-Index, original CapsNet, outperforms SOTA	Positiv	[Qia+18]
Convolutional capsule network for classification of breast cancer histology images			
KA	Brustkrebs-KA mit 256x256 Brusthistologie-Mikroskopie-Bildern, ähnelt original CapsNet mit anfangs fünf conv. Layern, keine Rekonstruktionen, t-SNE Visualisierung	Neutral	[Qia+18]
An attention-based Bi-GRU-CapsNet model for hypernymy detection between compound entities			
NLP, KA, EW	compound entities: englisch und chinesische Symptom- und Krankheitspaare, Hypernymy-Detection, Hypernymy-Relationship existiert oder nicht, original CapsNet, outperforms SOTA	Positiv	[Wan+18a]
Capsule Networks for Low Resource Spoken Language Understanding			

NLP, NM, KA	Spoken-Language-Understanding für command-and-control Applikationen, lernen von Nutzerdemonstrationen, Attention- und Sequenz-Erw. vom Original, wenig Trainigsdaten, outperforms SOTA	Positiv	[Rh18]
Investigating Capsule Networks with Dynamic Routing for Text Classification			
NLP, KA, EW, RA	Studie, drei Strategien um Routing zu stabilisieren um nicht gut trainierte CapsNets (wegen Hintergrund) zu verringern: Orphan, Leaky, Coefficients-Amendment, auf sechs Benchmarks trainiert, multi-label Stärke von CapsNets, N-gram conv. Layer, tieferes original CapsNet, schlägt Baselines	Positiv	[Zha+18b]

Tabelle 6.1: Tabellarische Zusammenstellung der restlichen gefundenen wissenschaftlichen Literatur. Hinzukommende Akronyme sind wie folgt definiert: neues Modell (NM), Klassifizierung (KA), Erweiterung (EW), Routing-Algorithmus (RA) und State-of-the-Art (SOTA)

6.7 Erklärung

Ich versichere, dass ich diese Masterarbeit selbstständig angefertigt, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben, sowie wörtliche und sinngemäße Zitate gekennzeichnet habe.

Kempten, den ....

Unterschrift

6.8 Ermächtigung

Hiermit ermächtige ich die Hochschule Kempten zur Veröffentlichung der Kurzzusammenfassung (Abstract) meiner Arbeit, z. Bsp. auf gedruckten Medien oder auf einer Internetseite.

Kempten, den ....

Unterschrift