

Bachelorarbeit

Michael Dammann

Intelligenztests und maschinelles Lernen - Lösungsansätze
mit neuronalen Netzen für Diagrammsequenzen

Michael Dammann

Intelligenztests und maschinelles Lernen - Lösungsansätze mit neuronalen Netzen für Diagrammsequenzen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Michael Neitzke
Zweitgutachter: Prof. Dr.-Ing. Andreas Meisel

Eingereicht am: 27. Januar 2020

Michael Dammann

Thema der Arbeit

Intelligenztests und maschinelles Lernen - Lösungsansätze mit neuronalen Netzen für Diagrammsequenzen

Stichworte

Künstliche Intelligenz, Intelligenztest, Maschinelles Lernen, Neuronale Netze

Kurzzusammenfassung

In dieser Bachelorarbeit werden verschiedene Architekturen neuronaler Netze auf ihre Fähigkeit untersucht, ein Problem zu lösen, welches eine Vielzahl von Domänen vereint: Bildverarbeitung und Generierung von Bildern, Sequenzverarbeitung, Erkennung von Relationen in Daten und Dimensionsreduzierung. Dazu wird eine bestimmte Art von Intelligenztests herangezogen. Diese geben eine Abfolge von fünf Bildern vor, welche geometrische Formen beinhalten, die ihre Ausprägungen und Positionen über die Sequenz hinweg verändern. Diesen Veränderungen unterliegen bestimmte Muster, welche erkannt werden müssen, um ein sechstes Bild korrekt zu generieren. Die verwendeten Tests werden als Diagrammsequenzen bezeichnet und werden im Rahmen dieser Arbeit eigens generiert.

Ausgehend von einem geläufigen Ansatz für ein verwandtes Problem, werden insgesamt vier verschiedene Ansätze von Architekturen neuronaler Netze an diesem Problem erprobt. Die Ansätze sind dabei aufeinander aufbauend und versuchen nach Analyse des jeweiligen Vorgängers Verbesserungsmöglichkeiten aufzugreifen, womit das Problem automatisch aus verschiedenen Perspektiven betrachtet wird. Die vier verschiedenen Ansätze werden ausgewertet und verglichen, außerdem wird die Funktionsweise der Ansätze durch einen Blick in die Modelle verdeutlicht. Durch die Vielseitigkeit des betrachteten Problems bietet die Arbeit darüber hinaus einen breiten Überblick über geläufige und weniger verbreitete Konzepte des maschinellen Lernens und wie diese zur Lösung einer ausgewählten Aufgabe kombiniert werden können.

Michael Dammann

Title of Thesis

Intelligence tests and machine learning - Approaches using neural networks for the solving of diagram sequence problems

Keywords

Artificial Intelligence, Intelligence Test, Machine Learning, Neural Networks

Abstract

In this bachelor thesis, multiple neural networks are evaluated in regard to their ability to solve a problem which combines a number of domains: image processing and image generation, sequence processing, recognition of relations in data and dimensionality reduction. A certain kind of intelligence tests is chosen for this. They consist of a sequence of five images containing geometric figures which change their shape and position in each image. Certain patterns underlie these changes and the task is to recognize these patterns to be able to generate the following sixth image. These tests are called diagram sequences and their creation is also part of this work.

Starting with an approach for a related problem, in the end the solving abilities of a total of four approaches using neural networks are tested. The approaches are based on each other, each representing a possible improvement to their respective predecessor after analyzing their possible drawbacks. Consequentially the problem is viewed from a range of different perspectives. The four approaches are evaluated and compared. Also the inner workings of the used models are elucidated. Due to the versatility of the examined problem a broad overview of the field of machine learning is given, considering both common and less widespread concepts of neural networks and how they can be combined to solve a certain task.

Inhaltsverzeichnis

Abbildungsverzeichnis	viii
Tabellenverzeichnis	xi
1 Einleitung	1
1.1 Hintergrund	1
1.2 Zielsetzung	2
1.3 Aufbau der Arbeit	3
2 Grundlagen	4
2.1 Intelligenztests	4
2.2 Maschinelles Lernen	5
2.2.1 Überwachtes Lernen	6
2.2.2 Unüberwachtes Lernen	7
2.3 Neuronale Netze	9
2.3.1 Multilayer Perceptron	9
2.3.2 Convolutional Neural Networks	11
2.3.3 Recurrent Neural Networks	15
2.3.4 Autoencoder	20
2.3.5 Training neuronaler Netze	20
3 Datensatz	23
3.1 Überblick	23
3.2 Erzeugung	25
4 Technische Umsetzung	29
5 Allgemeine Modelleinstellungen und Hyperparameter	30
5.1 Normierung	30
5.2 Convolutional Layers	30

5.3	Aktivierungsfunktion	31
5.4	Batch Normalization	32
5.5	Output und Kostenfunktion	33
5.6	Optimierer, Initialisierung und Batch Size	33
5.7	Evaluation	34
6	Erster Ansatz: CNN-RNN	35
6.1	Modelle	35
6.2	Diskussion	38
7	Zweiter Ansatz: CNN-ConvLSTM	43
7.1	Convolutional LSTM	43
7.2	Modelle	45
7.3	Diskussion	47
8	Dritter Ansatz: FullyCNN	50
8.1	Dilated Convolution	51
8.2	Modelle	51
8.3	Diskussion	54
9	Vierter Ansatz: RelNet	58
9.1	Relational Networks	58
9.2	Übersicht RelNet Ansatz	60
9.3	Autoencoder zur Komprimierung der Diagramme	62
9.4	RelNetMLP	64
9.4.1	Modelle	64
9.4.2	Diskussion	66
9.5	RelNetConv	71
9.5.1	Modelle	71
9.5.2	Diskussion	74
10	Fazit	78
10.1	Zusammenfassung	78
10.2	Ausblick	80
	Literaturverzeichnis	83

A Anhang	89
A.1 Symbole, Sequenz- und Bewegungsmuster des Datensatzes	89
A.2 Verwendete Bibliotheken	90
A.3 Modelleinstellungen	91
A.3.1 Hyperparameter	91
A.3.2 Verwendete Keras Einstellungen	91
A.4 Weitere Beispielvorschläge	94
Selbstständigkeitserklärung	96

Abbildungsverzeichnis

2.1	Beispiele für die Ziffer 7 im MNIST-Datensatz [2].	5
2.2	Skizze eines gelabelten Datensatzes. Die Klassen („x“ oder „o“) eines Datensatzes sind vorher bekannt und zugeordnet.	7
2.3	Skizze eines ungelabelten Datensatzes. Mögliche Klassen sind nicht bekannt, es können jedoch durch unüberwachtes Lernen Cluster (blau und grün) ermittelt werden.	8
2.4	Graphische Darstellung eines künstlichen Neurons mit $m = 3$ Eingangswerten.	9
2.5	Graphische Darstellung einer Schicht aus 3 künstlichen Neuronen mit 2 Eingangswerten. Die Biaswerte sind aus Übersichtsgründen ausgeblendet.	10
2.6	Beispiel für ein MLP mit einem Hidden Layer.	10
2.7	Beispiel für eine Convolution eines $4 \times 4 \times 2$ Volumens mit einem 3×3 Filter.	12
2.8	Beispiel für Max Pooling mit einem 2×2 Filter und einem Stride von 2.	13
2.9	Beispiel für ein CNN mit Convolutional, Max Pooling und Fully-connected Layers.	14
2.10	Grundstruktur eines RNNs.	15
2.11	Grundstruktur einer LSTM Zelle, adaptiert nach [42].	16
2.12	Grundstruktur einer GRU Zelle, adaptiert nach [42].	18
2.13	Beispiel für einen Autoencoder.	20
3.1	Eine Diagrammsequenz aus dem Datensatz.	23
3.2	Symbole bewegen sich über den Bildrand.	24
3.3	Eine Diagrammsequenz ohne Bewegungen.	24
3.4	Ein Beispiel für ein äußeres, inneres und daraus kombiniertes Gesamtsymbol.	25
3.5	Eine Symbolsequenz mit äußerem und innerem Muster.	25
3.6	Abstraktion der Diagrammsequenz als $4 \times 4 \times 6$ -Array.	26
3.7	Das Array mit einem Platzhalter für eine bewegliche Sequenz.	26
3.8	Markierung noch freier Felder in Orange für unbewegliche Sequenzen.	27

3.9	Mit unbeweglichen Sequenzen befülltes Array.	27
3.10	Zuordnung der Symbole zu den Sequenzen.	28
3.11	Erzeugung der fertigen Diagrammsequenz.	28
5.1	Prinzip des (2,2)-UpSamplings.	31
6.1	CNN-RNN Grundarchitektur.	35
6.2	Detaillierte Darstellung der Kombination eines CNNs mit LSTM/GRU.	36
6.3	Verarbeitung einer Diagrammsequenz.	37
6.4	Beispiel einer Vorhersage von CNN-LSTM-512-64.	38
6.5	Beispiel einer Vorhersage von CNN-GRU-512-128.	39
6.6	Einige Feature Maps des ersten Layers von CNN-GRU-512-128.	40
6.7	Einige Feature Maps des vorletzten Layers von CNN-GRU-512-128.	40
6.8	Hidden State Aktivierungen von CNN-GRU-512-128 für zwei verschiedene Diagrammsequenzen.	41
7.1	Vergleich der Schnittstellen von LSTM und Convolutional LSTM (bei Verwendung eines Same-Paddings).	43
7.2	CNN-ConvLSTM Grundarchitekturen. Für nicht dargestellte Hyperparameter siehe Abb 6.1.	45
7.3	Detailliertes Prinzip der Verarbeitung mit Convolutional LSTMs.	46
7.4	Beispiel einer Vorhersage von CNN-2xConvLSTM-512.	47
7.5	Beispiel einiger Feature Maps des ersten Layers von CNN-2xConvLSTM-512.	48
7.6	Beispiel einiger Feature Maps des vorletzten Layers von CNN-2xConvLSTM-512.	48
8.1	Beispiel einer Dilated Convolution.	51
8.2	ModuleCNN Architektur.	52
8.3	SimpleCNN Architektur.	53
8.4	Beispiel einer Vorhersage von ModuleCNN-NoDil-64.	54
8.5	Beispiel einiger Feature Maps von ModuleCNN-64.	55
8.6	Beispiel einer Vorhersage von SimpleCNN-64.	56
8.7	Beispiel einiger Feature Maps von SimpleCNN-64.	57
9.1	Skizze eines Relational Networks, adaptiert nach [50].	59
9.2	Flussdiagramm: Einzelschritte des RelNet-Ansatzes.	60
9.3	Autoencoder für die Komprimierung der Diagramme.	62

9.4	Beispiele für durch den Autoencoder rekonstruierte Diagramme.	62
9.5	Beispiele für Vektorrepräsentationen.	63
9.6	Skizze der Architektur von RelNetMLP.	65
9.7	Skizze der Architektur von MLPOnly.	65
9.8	Vorhersagen MLPOnly und RelNetMLP.	66
9.9	Aktivierungen der ersten Schicht von MLPOnly-512 für zwei Beispiele. . .	67
9.10	Aktivierungen der ersten Schicht von g_θ des RelNetMLP-512-Modells. . .	68
9.11	Ausgaben (ermittelte Relationen) von g_θ des RelNetMLP-512-Modells. . .	69
9.12	Prinzip der Inputanordnung und Verwendung der CNN Filter.	71
9.13	Skizze der Architektur von RelNetConv.	72
9.14	Skizze der Architektur von ConvOnly.	73
9.15	Feature Maps der ersten Convolutional Schicht des ConvOnly-64.	75
9.16	Feature Maps der ersten Convolutional Schicht von g_θ des RelNetConv-64. .	76
A.1	Alle äußeren Symbole.	89
A.2	Alle inneren Symbole.	89
A.3	Weitere Vorhersagen von CNN-GRU-512-128.	94
A.4	Weitere Vorhersagen von CNN-ConvLSTM-2x512.	94
A.5	Weitere Vorhersagen von ModuleCNN-NoDil-64.	95
A.6	Weitere Vorhersagen von MLPOnly-512.	95

Tabellenverzeichnis

6.1	Ergebnisse CNN-LSTM Architekturen.	38
6.2	Ergebnisse CNN-GRU Architekturen.	39
7.1	Ergebnisse CNN-ConvLSTM Architekturen.	47
8.1	Ergebnisse ModuleCNN Architekturen.	54
8.2	Ergebnisse SimpleCNN Architekturen.	56
9.1	Ergebnisse RelNetMLP und MLPOnly Architekturen.	66
9.2	Ergebnisse RelNetConv und ConvOnly Architekturen.	74
10.1	Ergebnisse der besten Modelle innerhalb der vier Ansätze.	79

1 Einleitung

1.1 Hintergrund

Intelligenz beschreibt laut Duden die „Fähigkeit, [...] abstrakt und vernünftig zu denken und daraus zweckvolles Handeln abzuleiten“ [1] und ist eine Domäne, welche lange alleine dem Menschen vorbehalten war. Für lange Zeit halfen Maschinen dem Menschen insbesondere bei Aufgaben repetitiver Art, deren immer gleicher oder ähnlicher Ablauf vergleichsweise einfach technisch umgesetzt werden kann oder bei Aufgabenstellungen, welche (zu) viel körperliche Kraft beanspruchen. Seit einigen Jahren ändert sich diese Beziehung, welche die Grundvorstellung des täglichen Lebens und der Arbeitswelt prägte - denn die Maschinen sind klüger geworden.

Verschiedene Verfahren, welche Maschinen anhand von Daten eine bestimmte Aufgabe *lernen* lassen, haben sich als sehr erfolgreich erwiesen. Dieser Bereich wird als *Maschinelles Lernen* bezeichnet. So war es beispielsweise möglich, (künstliche) persönliche Assistenten wie Amazons Alexa oder Apples Siri zu erschaffen, welche auf Sprachbefehle hören können und bereits in das tägliche Leben vieler Menschen integriert sind.

Doch dabei bleibt es nicht. Es hat sich gezeigt, dass Verfahren des maschinellen Lernens häufig Muster oder Regeln erkennen, welche für den Menschen nicht ersichtlich sind - in einer Reihe von Anwendungsfällen schlagen die Maschinen also sogar den Menschen. Ein Beispiel ist die KI AlphaGo Zero, welche den besten menschlichen Go Spieler schlagen konnte, allein dadurch, dass sie unzählige Spiele gegen sich selbst spielte und sich dabei stetig verbesserte [52].

Auch in Unternehmen wird zusätzlich zu menschlicher Expertise zunehmend auf solche lernenden Verfahren zurückgegriffen, um neue Erkenntnisse zu gewinnen und beispielsweise Entscheidungsprozesse auf eine möglichst objektive Grundlage zu stellen (*Advanced Analytics*).

Weitere Felder, die durch künstliche Intelligenz in Form des maschinellen Lernens auf ein neues Level gehoben wurden, sind beispielsweise die Bildverarbeitung [10] oder die automatische Übersetzung von Texten [5].

Ein viel diskutierter Punkt ist dabei auch die Erklärbarkeit der verwendeten Modelle (*Explainable AI*). Insbesondere, wenn ein Verfahren bessere Leistungen als der Mensch zeigt, ist es ein wichtiges Ziel, die Vorgehensweise des Modells möglichst gut nachvollziehen zu können.

1.2 Zielsetzung

Im Rahmen dieser Arbeit wird eine bestimmte Art von Intelligenztests genutzt, um die Fähigkeit verschiedener neuronaler Netze zu bewerten, ein Problem zu lösen, welches eine Vielzahl von Domänen vereint: Bildverarbeitung und Generierung von Bildern, Sequenzverarbeitung, Erkennung von Relationen in Daten und Dimensionsreduzierung.

Die verwendeten Intelligenztests geben eine Sequenz von fünf Bildern vor. Diese beinhalten geometrische Formen, welche ihre Ausprägungen und Positionen über die Sequenz hinweg verändern. Diesen Veränderungen liegen bestimmte Muster zugrunde, welche erkannt werden müssen, um ein sechstes Bild korrekt zu generieren (**kein** Multiple Choice). Diese Bilder werden auch als *Diagramme* und die Tests als *Diagrammsequenzen* bezeichnet, ihre Erstellung ist ebenfalls Teil dieser Arbeit. Die Bewertung der Modelle findet nicht durch einen IQ Wert statt, sondern durch einen Soll-Ist-Vergleich der Vorhersagen auf einem Testdatensatz.

Hierbei sollen verschiedene Ansätze verglichen werden, welche aufeinander aufbauen und die Ergebnisse pro Ansatz sukzessive verbessern sollen. Basierend auf einer Analyse nach jedem Ansatz, sollen dabei schrittweise konzeptionelle Verbesserungsmöglichkeiten in jeden neuen Ansatz fließen. Es sollen Erkenntnisse gewonnen werden, wie und warum die unterschiedlichen Ansätze sich auf den Erfolg bei der Aufgabenlösung auswirken. Damit zusammenhängend sollen außerdem Einblicke in die Funktionsweisen der Modelle gewonnen werden (*Explainable AI*).

Darüber hinaus soll ein breiter Überblick über geläufige und weniger geläufige Verfahren des maschinellen Lernens gegeben werden und wie sich diese zu neuen Ansätzen kombinieren lassen, um ein ausgewähltes Problem zu lösen. Die untersuchte Problemstellung ist verwandt mit Anwendungen wie Frame Prediction [55], Visual Question Answering [3]

und Video Captioning [43], weshalb sich die Erkenntnisse dieser Arbeit möglicherweise auch auf diese Bereiche anwenden lassen.

1.3 Aufbau der Arbeit

Inklusive dieser Einleitung ist die Arbeit in insgesamt 10 Kapitel unterteilt.

In **Kapitel 2** folgen die Grundlagen, welche einen Überblick über Intelligenztests im Allgemeinen und die die grundlegenden, verwendeten Verfahren des maschinellen Lernens geben sollen.

In **Kapitel 3** wird der verwendete Intelligenztest vorgestellt und wie dieser Datensatz eigens für diese Arbeit erzeugt wurde.

In **Kapitel 4** wird die technische Umsetzung der Datensatzgenerierung und der neuronalen Netze skizziert. Es wird ein Überblick über die verwendete Soft- und Hardware gegeben.

In **Kapitel 5** werden grundlegende Modell- und Hyperparametereinstellungen, welche für alle beschriebenen Ansätze gelten, beschrieben und ihre Wahl begründet.

Kapitel 6, 7, 8 und 9 bilden den Hauptteil dieser Arbeit. Insgesamt vier Ansätze neuronaler Netze zur Lösung von Diagrammsequenzen werden vorgestellt, ausgewertet und diskutiert. Wie beschrieben sind die einzelnen Ansätze aufeinander aufbauend und greifen jeweils Verbesserungsmöglichkeiten des Vorgängers auf.

In **Kapitel 10** wird die Arbeit zusammengefasst, ein Fazit gezogen und ein Ausblick gegeben.

2 Grundlagen

2.1 Intelligenztests

„Intelligenz“ ist ein breit gefächertes Begriff und kann recht unterschiedlich ausgelegt werden. So existiert unter anderem der Begriff emotionaler Intelligenz, welcher etwas über das Empathievermögen eines Menschen aussagt. Darüber hinaus ist beispielsweise auch von politischer Intelligenz regelmäßig die Rede. Im Rahmen dieser Arbeit soll mit Intelligenz die Fähigkeit des abstrakten Denkens, der Mustererkennung und des Schlussfolgerns bezeichnet werden.

Der Versuch, Intelligenz mit einem Zahlenwert zu messen (ein Untergebiet der Psychometrie) geht zurück bis in das Jahr 1905, als Alfred Binet und Théodore Simon mit dem Binet-Simon-Test den ersten Intelligenztest entwickelten, welcher bestimmten Gütekriterien wie Objektivität und Verlässlichkeit genügen sollte [4]. Intelligenztests existieren in verschiedenartigen Ausprägungen, bei denen es z. B. darum geht Zahlenreihen zu ergänzen, das Gedächtnis zu testen oder das räumliche Denken zu proben. Über Kriterien wie Korrektheit der Antworten, Bearbeitungsgeschwindigkeit oder auch Kreativität findet die Bewertung statt. Die geläufigste Maßeinheit für Intelligenz ist der Intelligenzquotient (IQ) [4].

Zu den bekanntesten Arten von Intelligenztests gehören Ravens progressive Matrizen, benannt nach ihrem Erfinder John Raven [34]. Hierbei handelt es sich um einen Multiple-Choice-Test, bei dem acht Diagramme mit abstrakten Mustern vorgegeben werden, um auf Basis der Relationen innerhalb dieser Diagramme das richtige neunte Bild auszuwählen. Eine Absicht der Verwendung dieser sprachfreien Tests war die Nivellierung von Unterschieden bezüglich des Sprachvermögens verschiedener Testgruppen, wobei auch diese Tests nicht alle Unterschiede ausmerzen konnten. Die in dieser Arbeit verwendeten Tests ähneln Ravens progressiven Matrizen sehr, jedoch handelt es sich nicht um

Multiple-Choice-Tests, sondern um eine generative Aufgabe, d. h. das Zielbild muss erzeugt werden. Außerdem wird in dieser Arbeit als Bewertungskriterium kein IQ Wert verwendet werden. Stattdessen findet eine sog. Kostenfunktion Anwendung, welche die Ist-Ausgabe des Systems mit einer Soll-Ausgabe vergleicht und quantifiziert.

2.2 Maschinelles Lernen

Im Zusammenhang von Maschinen wird - im Allgemeinen - von einem Lernprozess gesprochen, wenn diese ihr Programm, ihre Struktur oder ihre Parameter so ändert, dass sich ihre erwartete Leistung bei einer bestimmten Aufgabe verbessert [41].

Im Bereich des maschinellen Lernens (ML) werden mathematische Modelle wie z. B. künstliche neuronale Netze (Kapitel 2.3) verwendet, die - anstelle von expliziter Programmierung - alleine anhand von Trainingsdaten lernen („*trainieren*“) sollen eine bestimmte Aufgabe zu lösen [31]. Ein solcher Ansatz wird häufig dann gewählt, wenn ein Problem nicht mehr durch menschliche Analyse und explizite Programmierung zu bewältigen ist. Wenn ein Trainingsdatensatz zur Verfügung steht, kann ein ML-Modell trainiert werden, welches im Idealfall lernt die entscheidenden Muster in den Daten zu erkennen, um die gestellte Aufgabe mit einer *akzeptablen* Leistung zu lösen. Da ML-Modelle im Allgemeinen fehlerbehaftet sind, muss dabei vom Menschen für das jeweilige Problem festgelegt werden, ab wann die Leistung eines Modells *akzeptabel* ist. Häufig geschieht dies anhand quantitativer Maße wie Precision, Recall oder Accuracy [6].

Ein Beispiel für ein Problem, bei welchem ML sich bewährt hat, ist die Erkennung (*Klassifikation*) handgeschriebener Zahlen. Der häufig genutzte MNIST-Datensatz enthält Bilder von 70000 handgeschriebenen Ziffern [2], einige Beispiele für die Ziffer 7 sind in Abb. 2.1 aufgeführt.

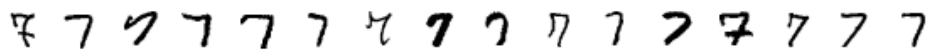


Abbildung 2.1: Beispiele für die Ziffer 7 im MNIST-Datensatz [2].

Zu sehen ist, wie stark unterschiedlich die Ziffern geschrieben werden, manche weisen auch eine große Ähnlichkeit zur Ziffer 1 auf. Es ist schwierig, für die Erkennung solcher Ziffern allgemeingültige Regeln und Programmcode aufzustellen, welche jede Ausprägung erfassen. Aus diesem Grund bieten sich lernende Verfahren an. Unter Verwendung

von Ansätzen, die sich für die Verarbeitung von Bildern besonders anbieten (Convolutional Neural Networks, Kapitel 2.3.2), ist es möglich, Modelle zu trainieren, die für die Klassifikation von MNIST-Ziffern lediglich eine Fehlerrate von 0,21 % aufweisen [47].

Insbesondere unter Verwendung neuronaler Netze hat sich maschinelles Lernen in einer Reihe von Domänen der künstlichen Intelligenz als sehr erfolgreich erwiesen. Beispielsweise bei der Klassifikation von Verkehrszeichen [10], der automatischen Übersetzung von Texten [5], der Erkennung von Objekten in Bildern [45][46] oder Empfehlungssystemen [60]. Eine wichtige Basis dieser Erfolge ist die Verfügbarkeit großer Datenmengen, die für das Trainieren verwendet werden können. Die Entwicklung erfolgreicher ML-Verfahren wird also durch das schnelle Wachstum vorhandener Daten begünstigt und in vielen Fällen überhaupt erst ermöglicht [11][12]. Außerdem ist in vielen Fällen leistungsstarke Hardware (insbesondere GPUs) notwendig, um neuronale Netze in einem vertretbaren Zeitrahmen zu trainieren. Es ist davon auszugehen, dass neuronale Netze, welche sich im Bereich des maschinellen Lernens etablieren konnten und in erster Form bereits 1943 publiziert wurden [39], in erster Linie durch die beiden Faktoren *rasantes Datenwachstum* und *steigende Hardwareleistung* in den vergangenen Jahren eine Renaissance erlebt haben. Vorher sind für Probleme aus dem Bereich der künstlichen Intelligenz häufig Support Vector Machines oder Fuzzylogic Systeme zum Einsatz gekommen [54] [32].

Bezüglich maschinellen Lernens kann unterschieden werden zwischen überwachtem und unüberwachtem Lernen [41], welche in den folgenden beiden Abschnitten 2.2.1 und 2.2.2 erläutert werden sollen. Darüber hinaus existiert das verstärkende Lernen [6], welches jedoch in dieser Arbeit keine Anwendung findet.

2.2.1 Überwachtes Lernen

Die Vorbedingung für überwachtes Lernen ist ein *gelabelter* Datensatz (s. Abb. 2.2), womit die einzelnen Samples des Datensatzes formal das Tupel (x, y) bilden. Am Beispiel des MNIST Datensatzes bilden also die Menge handgeschriebener Bilder den Definitionsbereich und die ganzen Zahlen zwischen 0 und 9 den Wertebereich. Ziel des überwachten Lernens ist es, eine Funktion zu trainieren, welcher die korrekte Abbildung aus dem Trainingsdatensatz möglichst gut gelingt, wobei die Trainingsdaten jedoch nicht auswendig gelernt werden sollen („*Overfitting*“). Stattdessen soll eine ausreichende Verallgemeine-

zung des Modells durch das Training gelingen, sodass auch ungesehene Daten möglichst korrekt zugeordnet werden. [6]

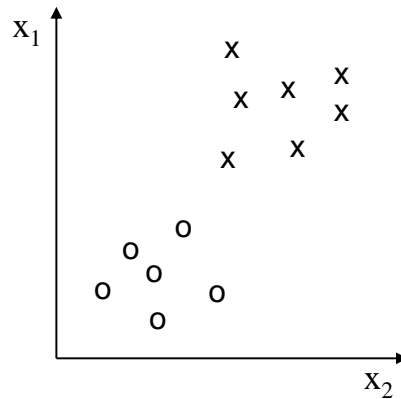


Abbildung 2.2: Skizze eines gelabelten Datensatzes. Die Klassen („x“ oder „o“) eines Datensatzes sind vorher bekannt und zugeordnet.

Im Bereich des überwachten Lernens kann wiederum unterschieden werden in Probleme der *Klassifikation* einerseits und der *Regression* andererseits. Bei der Klassifikation geht es darum, einen Input einer bestimmten *Klasse* (einem *diskreten* Wert) zuzuordnen, ein typisches Beispiel ist die bereits erwähnte Klassifikation von MNIST Ziffern. Regressionsprobleme hingegen sollen einem Input einen Wert aus einem *kontinuierlichen* Wertebereich zuordnen, ein Beispiel dafür ist die Detektion der Position, Höhe und Breite von Objekten in Bildern („*Object Detection*“) [45].

2.2.2 Unüberwachtes Lernen

Im Gegensatz zum überwachten Lernen besitzen die Daten des Trainingsdatensatzes keine *Labels*. Das Ziel des unüberwachten Lernens lautet im Allgemeinen (unbekannte) Strukturen in diesen Daten zu erkennen [6].

Ein großes Anwendungsgebiet des unüberwachten Lernens ist *Clustering*, welches die Daten in Gruppen („*Cluster*“) sortiert. Ein Algorithmus aus dieser Kategorie lautet *k-Means* [6], welcher auf Basis des euklidischen Abstands zwischen den Datenpunkten den Datensatz iterativ in eine vorher festgelegte Anzahl an Clustern aufteilt. Ein Anwendungsgebiet ist beispielsweise die Marktforschung, in welcher Clusteringverfahren genutzt werden um die Kundschaft in einzelne Zielgruppen aufzuteilen.

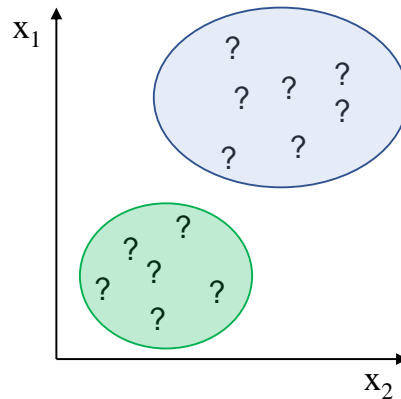


Abbildung 2.3: Skizze eines ungelabelten Datensatzes. Mögliche Klassen sind nicht bekannt, es können jedoch durch unüberwachtes Lernen Cluster (blau und grün) ermittelt werden.

Darüber hinaus bilden Verfahren der Komprimierung („*Dimensionality Reduction*“) einen weiteren Bereich des unüberwachten Lernens ab. Ein Beispiel hierfür sind sogenannte Autoencoder, welche auch in dieser Arbeit zum Einsatz kommen (s. Kapitel 2.3.4). Unter ihrer Verwendung soll erreicht werden, eine relativ niedrigdimensionale Repräsentation („*Encoding*“) hochdimensionaler Eingangsdaten zu finden. Zweck solch einer Repräsentation kann beispielsweise die Unterscheidung von Signal und Rauschen sein, da die niedrigdimensionale Repräsentation einen Flaschenhals darstellt, welcher nur die relevanten Informationen speichern kann. Damit zusammenhängend können so auch Eingangsdaten für ein weiteres Modell angefertigt werden („*Feature Engineering*“), unter der Annahme, dass die Encodings nur die relevanten Informationen enthalten und dadurch das Training des neuen Modells möglicherweise erfolgreicher abläuft als ohne solch eines Feature Engineerings.

2.3 Neuronale Netze

2.3.1 Multilayer Perceptron

Als *Multilayer Perceptrons* (MLP, deutsch: mehrschichtiges Perzeptron) wird eine Kategorie künstlicher neuronaler Netze bezeichnet, welche im Bereich des maschinellen Lernens verwendet werden [6]. Ihr Grundbaustein ist ein künstliches Neuron, welches m skalare Eingangswerte $x_1 \dots x_m$ einem skalaren Ausgangswert y nach Formel 2.1 zuordnet.

$$y = \phi \left(\sum_{i=1}^m w_i x_i + b \right) \quad (2.1)$$

$w_1 \dots w_m$ werden als Gewichte und b als Bias bezeichnet, bei ihnen handelt es sich um die trainierbaren Parameter. $\phi(\cdot)$ ist eine Aktivierungsfunktion, häufig wird hier eine nichtlineare Funktion gewählt.

Üblich ist die Darstellung als Graph wie in Abb. 2.4 abgebildet, eine netzartige Struktur wird hier bereits deutlich.

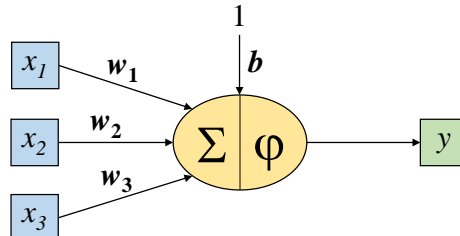


Abbildung 2.4: Graphische Darstellung eines künstlichen Neurons mit $m = 3$ Eingangswerten.

Mehrere Neuronen können zu einer Schicht (auch *Layer* genannt) zusammengefügt werden, wie in Abb. 2.5 dargestellt.

Die Inputwerte können zu $\vec{x} = (x_1, x_2)^T$ und die Outputwerte zu $\vec{y} = (y_1, y_2, y_3)^T$ zusammengefasst werden. Seien W die Matrix, welche die Gewichte beinhaltet und \vec{b} der Spaltenvektor, welche die Biaswerte enthält, so kann die Neuronenschicht ausgedrückt

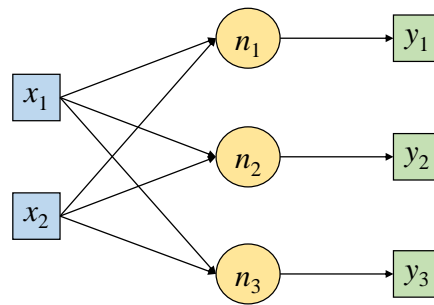


Abbildung 2.5: Graphische Darstellung einer Schicht aus 3 künstlichen Neuronen mit 2 Eingangswerten. Die Biaswerte sind aus Übersichtsgründen ausgeblendet.

werden mit:

$$\vec{y} = \phi \left(W\vec{x} + \vec{b} \right) \quad (2.2)$$

Mehrere Schichten wiederum können zu einem neuronalen Netz zusammengefügt werden, abgebildet in Abb. 2.6. Die Bezeichnung *Multilayer Perceptrons* (MLP) ist historisch bedingt, da diese Modelle ursprünglich auf dem Gedankenmodell eines Perzeptrons aufbauten [6]. Eine andere geläufige Bezeichnung ist *Fully-connected Network*. Der Eingangsvektor jeder Schicht (mit Ausnahme der ersten) ist dabei der Ausgangsvektor der vorhergehenden Schicht.

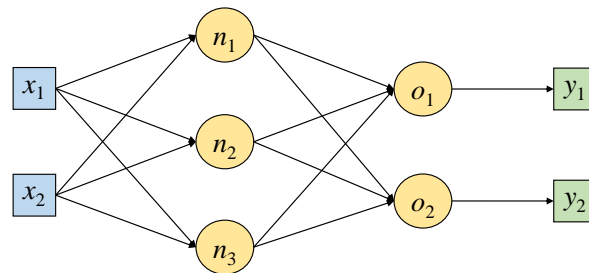


Abbildung 2.6: Beispiel für ein MLP mit einem Hidden Layer.

Die Schicht mit den Eingangswerten x_1 und x_2 wird dementsprechend als *Input Layer* bezeichnet, die Ausgangsschicht welche die Werte y_1 , y_2 und y_3 erzeugt als *Output Layer*. Alle Schichten dazwischen tragen die Bezeichnung *Hidden Layer*. Das neuronale Netz in Abb. 2.6 besitzt somit ein Hidden Layer, nämlich die Schicht mit den Neuronen n_1 , n_2 und n_3 .

Bei Wahl einer differenzierbaren Aktivierungsfunktion $\phi(\cdot)$, wie z.B. der Sigmoid-, Tangens hyperbolicus- oder Identitätsfunktion, stellt das gesamte MLP eine differenzierbare Funktion dar, womit eine notwendige Eigenschaft für das Training des neuronalen Netzes sichergestellt ist (s. Abschnitt 2.3.5).

Die Anzahl der Schichten, die Anzahl der Neuronen in einer Schicht und die Aktivierungsfunktionen werden als Hyperparameter bezeichnet, womit Parameter des Modells bezeichnet werden, die vor dem Training festgelegt werden - im Gegensatz zu den trainierbaren Parametern.

2.3.2 Convolutional Neural Networks

Sollen Bilder mit neuronalen Netzen verarbeitet werden, so müssen sie bei Verwendung eines MLPs zunächst in einen Vektor überführt werden. Häufig wird dazu zeilenweise über die Farbwerte der einzelnen Pixel iteriert, welche einzeln in einen langen, „flachen“ (*flattened*) Vektor überführt werden. Diese Vorgehensweise besitzt zum einen den Nachteil des Verlusts räumlicher Information. Zum anderen verfügt diese Methode auch über keine gute Skalierbarkeit: Bei Verwendung eines 400x400 Pixel großen Farbbildes (3 Farbkanäle) besitzt der Inputvektor eine Größe von $400 \cdot 400 \cdot 3 = 480\,000$. Soll das erste Hidden Layer des MLPs 500 Neuronen erhalten, so ergeben sich insgesamt $480\,000 \cdot 500 = 240\,000\,000$ Gewichte (und 500 Biaswerte) als trainierbare Parameter. Eine solche Parameterzahl ist unpraktikabel hoch und wirkt sich außerdem häufig negativ auf das Training aus (Overfitting).

Convolutional Neural Networks (CNNs, deutsch: Faltungsnetzwerke) haben sich als effizientere und sehr erfolgreiche Modelle für die Verarbeitung von Bildern und anderen Anwendungsfeldern mit räumlichen Eingangsdaten erwiesen. Als ihr Begründer gilt Yann LeCun [35]. Ihre Bestandteile sind Convolutional, Pooling und (in manchen Anwendungsfällen) Fully-Connected Layers, welche in den nächsten Kapiteln erläutert werden sollen.

Convolutional Layer

Ein Convolutional Layer erwartet ein Volumen als Input und liefert ebenfalls ein Volumen als Output. Bei einem 400x400 Pixel großen Farbbild mit drei Farbkanälen wäre

das Inputvolumen beispielsweise 400x400x3. Die lernbaren Parameter eines Convolutional Layers sind sogenannte Filter mit relativ niedriger Höhe und Breite (oft 3x3 oder 5x5), welche sich jedoch über die gesamte Tiefe eines Volumens erstrecken. Diese Filter „gleiten“ über das Inputvolumen und erzeugen dabei das Outputvolumen, ein Beispiel ist in Abb. 2.7 dargestellt.

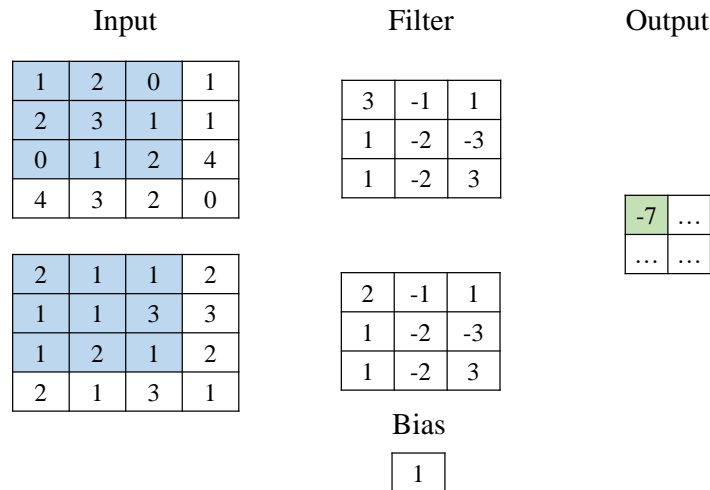


Abbildung 2.7: Beispiel für eine Convolution eines 4x4x2 Volumens mit einem 3x3 Filter.

Es ist dargestellt, wie ein einzelnes Outputneuron (grün) nur mit einem räumlich beschränkten Teil des Inputs verbunden ist (blau markiert), im Gegensatz zum MLP, bei welchem ein Neuron mit dem kompletten Input vernetzt ist („Local/Sparse Connectivity“). Aus diesem Inputausschnitt x und dem Filter f wird ein Outputwert y erzeugt. Für dieses konkrete Beispiel gilt Gleichung 2.3, die Verwandtschaft zum MLP (Gl. 2.1) ist erkennbar (in Abb. 2.7 wird die Identität als Aktivierungsfunktion verwendet).

$$y = \phi \left(\sum_{w=1}^3 \sum_{h=1}^3 \sum_{d=1}^2 f_{w,h,d} x_{w,h,d} + b \right) \quad (2.3)$$

Die Indizes w , h und d beziehen sich dabei auf die Spalten, Zeilen und Kanäle des Filters und des ausgewählten Inputbereichs (in Abb. 2.7 die blauen Werte - nicht der gesamte Input). Die Filter wandern mit einer definierbaren Schrittgröße („Stride“) durch das Bild, diese ist häufig 1 oder 2. Dies bedeutet, dass die Neuronen sich die Gewichte teilen („Shared Weights“). Während des Lernprozesses erlernen die Filter bestimmte *Eigen-*

schaften (*Features*), wie beispielsweise Kanten, zu erkennen. Bereiche der Bilder, die Kanten aufweisen, erfahren dann eine besonders hohe Aktivierung. Aus diesem Grund nennt man die einzelnen „Scheiben“ des Outputs auch *Feature Maps*. Die Anzahl der Filter und damit die Tiefe des Outputvolumens wird vor dem Training festgelegt. Außerdem kann noch ein „*Padding*“ bestimmt werden, womit ein Rahmen bezeichnet wird, welcher das Inputbild umgibt. Üblich ist das sogenannte „*Zero-Padding*“, welches aus Nullen besteht. Zweck eines solchen Paddings ist es einerseits, die Höhe und Breite des Outputvolumens auf die gewünschten Werte zu bringen (wie in Abb. 2.7 dargestellt schrumpfen Höhe und Breite bei einer Convolution ohne Padding). Andererseits würden ohne Padding Neuronen am Rand seltener in Berechnungen einbezogen werden als Neuronen in der Mitte eines Inputs, diese Ungleichheit soll durch ein Padding ausgeglichen werden.

Filtergröße, Stride, Padding und Filteranzahl sind Hyperparameter, die vor dem Training festgelegt werden.

Das Outputvolumen eines Convolutional Layers kann als Input eines folgenden Convolutional oder Pooling Layers verwendet werden, womit komplexe Architekturen gestaltet werden können.

Pooling Layer

Es ist üblich zwischen Convolutional Layers sogenannte Pooling Layers einzufügen. Pooling Layers besitzen keine trainierbaren Parameter und verringern die Höhe und Breite eines Inputvolumens, nicht jedoch die Tiefe. In Abb. 2.8 ist ein Beispiel für das sogenannte Max Pooling dargestellt.

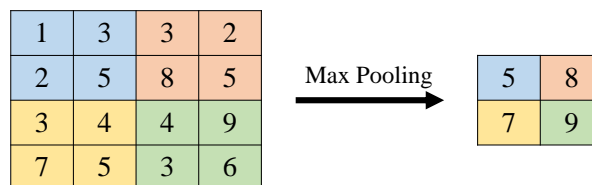


Abbildung 2.8: Beispiel für Max Pooling mit einem 2x2 Filter und einem Stride von 2.

Beim Max Pooling wandert ein Fenster (in diesem Beispiel der Größe 2x2) mit einem definierten Stride (hier 2) über das Bild, ermittelt den Maximalwert innerhalb des aktuellen

Fensters und überträgt diesen in den Output. Max Pooling wird an jeder Tiefenschicht des Volumens durchgeführt, dadurch halbieren sich in diesem Beispiel Höhe und Breite des Inputs. Durch Max Pooling sinkt die Anzahl der Parameter, was ebenso die Anzahl an Berechnungen verringert und die Gefahr des Overfittings mindert, da im Allgemeinen durch die Poolingoperation nur die relevantesten Features erhalten bleiben.

In dieser Arbeit wird Max Pooling wie in Abb. 2.8 dargestellt verwendet. Darüber hinaus existieren noch weitere Pooling Varianten wie beispielsweise Min oder Average Pooling.

Fully-connected Layer

In vielen Anwendungsfällen ist es nötig, die Struktur aus Convolutional und Pooling Layers ab einem bestimmten Punkt in eine Fully-connected Architektur zu überführen. Dies ist z. B. der Fall, wenn eine Klassifikation stattfinden soll und damit der Output einen Vektor darstellt. Bei der Klassifikation von MNIST Ziffern würde solch ein Vektor dann beispielsweise 10 Elemente enthalten. Um von einer Convolutional in eine Fully-connected Struktur überzugehen, wird über die Werte des Outputs einer Convolutional Architektur iteriert, welche in einen langen, „flachen“ Vektor überführt werden. Diese Operation wird auch als „*Flatten*“ bezeichnet und der Vektor kann dann wie der Input eines MLPs behandelt werden und mit Fully-connected Layers verknüpft werden.

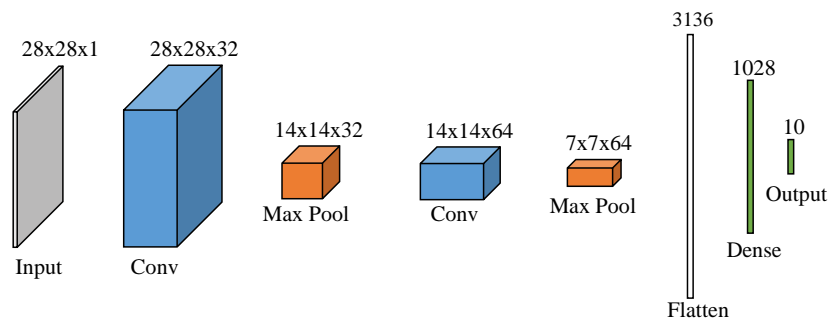


Abbildung 2.9: Beispiel für ein CNN mit Convolutional, Max Pooling und Fully-connected Layers.

In Abb. 2.9 ist eine komplette CNN Architektur dargestellt, wie sie für die Klassifikation von MNIST Zahlen verwendet werden könnte. Zu beachten ist die durchaus übliche, jedoch nicht zwingend erforderliche, abwechselnde Abfolge von Convolutional und Max Pooling Schichten, die daraus folgende Transformation des Inputs in ein Volumen

mit recht kleiner Höhe und Breite aber großer Tiefe und der Übergang zu einer Fully-connected Struktur am Ende.

2.3.3 Recurrent Neural Networks

Recurrent Neural Networks [32] (RNNs) sind geeignet für die Verarbeitung sequentieller Daten von prinzipiell beliebiger Länge. Ihre grundsätzliche Struktur ist in Abb. 2.10 abgebildet.

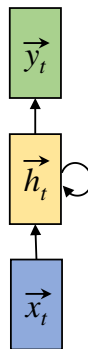


Abbildung 2.10: Grundstruktur eines RNNs.

\vec{x}_t ist der Inputvektor und \vec{y}_t der Outputvektor zu einem Zeitpunkt t , wobei der Begriff „Zeitpunkt“ nicht im streng chronologischen Sinne zu verstehen ist, sondern beispielsweise auch den Index t in einer Liste bezeichnen könnte. \vec{h}_t ist der sogenannte *Hidden State* zum Zeitpunkt t , welcher bei jedem neuen Inputvektor aus der Sequenz mittels Gl. 2.4 einem Update unterzogen wird.

$$\vec{h}_t = \tanh(W_{hh} \cdot \vec{h}_{t-1} + W_{xh} \cdot \vec{x}_t) \quad (2.4)$$

W_{hh} und W_{xh} sind hier die Gewichtsmatrizen mit den trainierbaren Parametern. Zu beachten ist, dass bei jedem Update der *Hidden State* des vorherigen Zeitpunktes \vec{h}_{t-1} miteinfließt. So kann auf frühere Informationen Bezug genommen werden, intuitiv stellt der Hidden State also eine Art Gedächtnis des RNNs dar.

Unter Verwendung von \vec{h}_t und einer weiteren Gewichtsmatrix W_{hy} wird \vec{y}_t mithilfe von Gl. 2.5 erzeugt.

$$\vec{y}_t = \tanh(W_{hy} \cdot \vec{h}_t) \quad (2.5)$$

RNNs sind aufgrund des Problems der *Vanishing* bzw. *Exploding Gradients* nicht in der Lage, unter Verwendung längerer Sequenzen gute Ergebnisse zu erzeugen [24]. Mit den sogenannten *Long Short-Term Memory* Netzen, welche in dieser Arbeit Anwendung finden und im nächsten Kapitel vorgestellt werden, kann dieses Problem gelöst werden.

Long Short-Term Memory

Long Short-Term Memory Netze (LSTMs) wurden in ihrer ersten Form 1997 von Sepp Hochreiter und Jürgen Schmidhuber vorgestellt [24]. Wie auch bei RNNs gilt bei den LSTMs ebenfalls als Grundprinzip, dass ein innerer Zustand bei jedem Zeitschritt aktualisiert wird. Die Grundstruktur einer LSTM Zelle ist in Abb. 2.11 dargestellt, zunächst ist hier zu beachten, dass zusätzlich zum Hidden State \vec{h}_t auch ein sogenannter *Cell State* \vec{c}_t aktualisiert wird. Die Länge der Vektoren \vec{h}_t und \vec{c}_t ist ein Hyperparameter, welcher als *Unit Size* bezeichnet wird, die Aktualisierung findet durch sogenannte *Gates* statt, welche Vektoren der gleichen Länge wie \vec{h}_t und \vec{c}_t und ebenfalls von t abhängig sind. Ihre Funktionsweise soll im Folgenden erläutert werden.

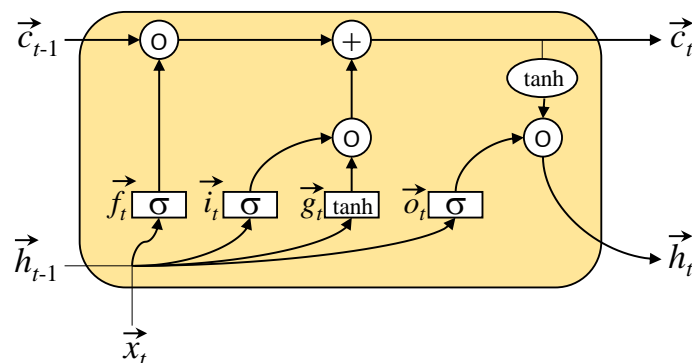


Abbildung 2.11: Grundstruktur einer LSTM Zelle, adaptiert nach [42].

Einführend eine Anmerkung zur Notation: Mit \circ wird das Hadamard Produkt gekennzeichnet und mit $[\vec{a}, \vec{b}]$ die Konkatination der Vektoren \vec{a} und \vec{b} .

Forget Gate

Zunächst wird mittels des sogenannten *Forget Gates* \vec{f}_t bestimmt, welche Werte aus \vec{c}_{t-1} gelöscht werden sollen. \vec{f}_t wird durch Gl. 2.6 ermittelt, wobei W_f die Gewichtsmatrix und \vec{b}_f den Biasvektor mit den lernbaren Parametern darstellen. Zu beachten ist, dass aufgrund der Sigmoidfunktion jeder Wert von \vec{f}_t auf $0 \dots 1$ beschränkt ist. Ein Wert nahe

1 bedeutet, dass der Wert an dieser Stelle in \vec{c}_{t-1} behalten werden soll, bei einem Wert nahe 0 soll dieser Eintrag in \vec{c}_{t-1} gelöscht werden.

$$\vec{f}_t = \sigma \left(W_f \cdot \left[\vec{h}_{t-1}, \vec{x}_t \right] + \vec{b}_f \right) \quad (2.6)$$

Input Gate und Candidate Values

Es wird außerdem entschieden, welche Informationen dem Cell State hinzugefügt werden sollen. Dazu dienen das *Input Gate* \vec{i}_t und die *Candidate Values* \vec{g}_t . Intuitiv betrachtet beschreiben die Candidate Values, was in den Cell State fließen könnte und das Input Gate, welche Kandidaten tatsächlich in den neuen Cell State fließen. \vec{i}_t und \vec{g}_t werden (analog zum Forget Gate) gemäß Gl. 2.7 bzw. Gl. 2.8 berechnet.

$$\vec{i}_t = \sigma \left(W_i \cdot \left[\vec{h}_{t-1}, \vec{x}_t \right] + \vec{b}_i \right) \quad (2.7)$$

$$\vec{g}_t = \tanh \left(W_g \cdot \left[\vec{h}_{t-1}, \vec{x}_t \right] + \vec{b}_g \right) \quad (2.8)$$

\vec{i}_t nimmt Werte zwischen 0 und 1 an und \vec{g}_t Werte zwischen -1 und 1.

Update des Cell States

Mit \vec{f}_t , \vec{i}_t und \vec{g}_t kann der Cell State nach Gl. 2.9 aktualisiert werden:

$$\vec{c}_t = \vec{f}_t \circ \vec{c}_{t-1} + \vec{i}_t \circ \vec{g}_t \quad (2.9)$$

Es werden also pro Zeitschritt zunächst mithilfe von \vec{f}_t Informationen gelöscht und anschließend mit \vec{i}_t und \vec{g}_t neue Informationen addiert.

Output Gate und Hidden State

Es wird außerdem festgelegt, welche Informationen nach außen gegeben werden sollen. Hierfür wird zunächst das *Output Gate* \vec{o}_t nach Gl. 2.11 berechnet, analog zu den anderen Gates:

$$\vec{o}_t = \sigma \left(W_o \cdot \left[\vec{h}_{t-1}, \vec{x}_t \right] + \vec{b}_o \right) \quad (2.10)$$

\vec{o}_t fließt dann gemeinsam mit dem bereits aktualisierten \vec{c}_t in das Update des Hidden States \vec{h}_t ein:

$$\vec{h}_t = \vec{o}_t \circ \tanh(\vec{c}_t) \quad (2.11)$$

Der Hidden State stellt letztendlich den Output des LSTMs zu einem Zeitpunkt t dar, womit die Bezeichnung *Hidden* nicht wirklich passend ist, jedoch als Begriff wahrscheinlich von den ursprünglichen RNNs übernommen wurde.

Ein LSTM kann - je nach Anwendungsfall - entweder zu jedem Zeitschritt oder nur bei dem letzten Zeitschritt \vec{h}_t ausgeben, dies kann als Hyperparameter festgelegt werden. Es gibt außerdem eine große Anzahl weiterer LSTM-Strukturen (andere Aktivierungsfunktionen, verschiedene Varianten der Verknüpfung von Gates und States), die jedoch laut [19] keine signifikanten Unterschiede zum hier vorgestellten LSTM bezüglich ihres Lernvermögens und ihrer Performanz aufweisen.

Gated Recurrent Unit

Neben den LSTMs konnten sich in der Sequenzverarbeitung die sog. *Gated Recurrent Units* (GRUs) etablieren, welche 2014 von Cho et al. vorgestellt wurden [7]. Die GRU Grundstruktur ist in Abb. 2.12 abgebildet und soll im Folgenden erklärt werden.

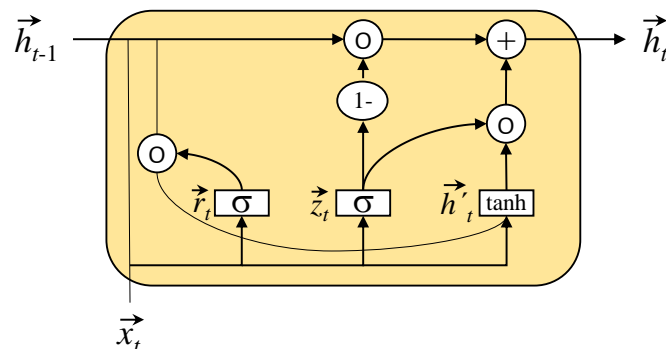


Abbildung 2.12: Grundstruktur einer GRU Zelle, adaptiert nach [42].

Zu sehen ist, dass Cell State und Hidden State zu \vec{h}_t zusammengefasst wurden. Die Zahl der Gates ist außerdem von 4 auf 3 reduziert worden.

Reset Gate

Mit dem *Reset Gate* wird ermittelt, welche Werte des Hidden States erhalten oder gelöscht werden sollen:

$$\vec{r}_t = \sigma \left(W_r \cdot \left[\vec{h}_{t-1}, \vec{x}_t \right] + \vec{b}_r \right) \quad (2.12)$$

Update Gate

Das *Update Gate* legt fest, welche neuen Informationen in den Hidden State fließen sollen:

$$\vec{z}_t = \sigma \left(W_z \cdot \left[\vec{h}_{t-1}, \vec{x}_t \right] + \vec{b}_z \right) \quad (2.13)$$

Gesamtupdate des Hidden States

Zunächst wird das Reset Gate für die Berechnung einer Art Übergangszustand des Hidden States \vec{h}'_t verwendet:

$$\vec{h}'_t = \tanh \left(W_{h'} \cdot \left[\vec{r}_t \circ \vec{h}_{t-1}, \vec{x}_t \right] + \vec{b}_{h'} \right) \quad (2.14)$$

Auf Basis dieses Übergangszustandes und des Update Gates wird anschließend das Gesamtupdate des Hidden States durchgeführt:

$$\vec{h}_t = (1 - \vec{z}_t) \circ \vec{h}_{t-1} + \vec{z}_t \circ \vec{h}'_t \quad (2.15)$$

2.3.4 Autoencoder

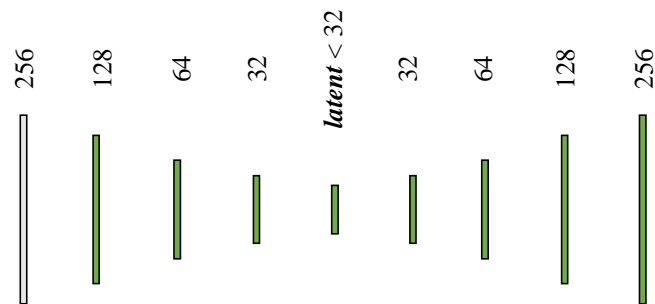


Abbildung 2.13: Beispiel für einen Autoencoder.

Autoencoder (AE) stellen Verfahren aus dem unüberwachten Lernen für die Dimensionsreduktion dar [23]. Sie werden darauf trainiert ihren Input zu reproduzieren. In Abb. 2.13 ist ein Beispiel für einen einfachen (MLP-)Autoencoder abgebildet. Es ist zu sehen, dass der AE sich in einen *Encoder* (linker Teil), den *Latent Vector* und einen *Decoder* (rechter Teil) aufteilt. Der *Latent Vector* $\in \mathbb{R}^{\text{latent}}$ ist die gesuchte Vektorrepräsentation, welche die Eingangsdaten komprimiert. Diese kann auch als „Flaschenhals“ betrachtet werden, welche nur die relevanten Informationen erhält, um das Bild möglichst gut zu rekonstruieren. Im Allgemeinen arbeitet ein Autoencoder jedoch nicht vollkommen verlustfrei.

2.3.5 Training neuronaler Netze

Wie bereits vorgestellt, besitzen neuronale Netze lernbare Parameter (Gewichte und Biases), die während des Trainings angepasst werden, um die Leistung des Modells stetig zu verbessern.

Da die Anzahl dieser Parameter i.d.R. sehr hoch ist - in dieser Arbeit besitzen manche neuronale Netze mehrere Millionen Parameter - kann nicht der gesamte Parameterraum nach der optimalen Lösung für den Datensatz durchsucht werden. Stattdessen wird ein Verfahren gewählt, das im Vergleich rechnerisch deutlich günstiger ist und welches sich einer möglichst guten Lösung schrittweise annähert. Dieses Verfahren nennt sich „*Mini-batch Stochastic Gradient Descent*“ (MBSGD) und soll im Folgenden kurz skizziert werden [6].

Zunächst wird der Gesamtdatensatz in einen Trainings-, Validierungs- und Testdatensatz (oft ca. 80 %, 10 %, 10 % von Gesamtdatensatz) aufgeteilt. Die Parameter werden

zufällig initialisiert (*Anmerkung*: Es werden i. d. R. komplexere Initialisierungsmethoden verwendet), womit das Modell vor Beginn des Trainings noch keine „Intelligenz“ besitzt. Der Trainingsdatensatz wird in kleinere *Minibatches* (im Folgenden kurz *Batches* genannt) mit einer bestimmten *Batch Size* aufgeteilt, in dieser Arbeit beträgt die *Batch Size* beispielsweise immer 32.

Pro Batch werden die Eingabemuster in das Netz gegeben und die durchschnittliche Performanz des Modells ermittelt, dies geschieht mithilfe einer sogenannten Kostenfunktion (*Cost Function*), welche die Abweichung des tatsächlichen Modelloutputs mit den Sollwerten vergleicht und einen Zahlenwert für die Abweichung (genannt *Kosten*) liefert. Die Art der Kostenfunktion ist ein Hyperparameter des zu trainierenden Modells. Sei \vec{w} der Vektor, der alle trainierbaren Gewichte beinhaltet. Ziel des MBSGD Verfahrens ist es nun die Kostenfunktion $E(\vec{w})$, welche abhängig von allen lernbaren Parametern ist, zu minimieren (im Allgemeinen wird ein lokales Minimum gefunden). Hierzu wird der Gradient der Kostenfunktion unter Verwendung des Durchschnittfehlers des ausgewählten Batches ermittelt. Dies geschieht durch das *Backpropagation* Verfahren [6], wobei durch Anwendung der Kettenregel die Kosten auf die einzelnen Gewichte zurückgeführt werden. Pro Batch werden die einzelnen Gewichte beim standardmäßigen MBSGD Verfahren dann eine definierte Schrittgröße η (*Lernrate*) angepasst, sodass der entstandene Fehler etwas kleiner wird:

$$\vec{w}_{neu} = \vec{w}_{alt} - \eta \frac{\partial E}{\partial \vec{w}_{alt}} \quad (2.16)$$

Nachdem dieser Prozess mit jedem Batch durchlaufen wurde, wird von einer vollendeten *Epoche* gesprochen. MBSGD baut auf dem *Gradient Descent* Verfahren auf, bei welchem der gesamte Datensatz für eine Anpassung der Gewichte durchlaufen werden muss. Aufgrund der Größe der Datensätze, welche im Bereich des maschinellen Lernens verwendet werden, ist dieses Verfahren jedoch keine gute Wahl, da das Training zu viel Zeit in Anspruch nehmen würde. Stattdessen wird angenommen, dass die Gradienten, welche durch die einzelnen Batches ermittelt werden, in ihrer Gesamtheit eine gute Annäherung an den tatsächlichen Gradienten des Gesamtdatensatzes darstellen.

Das trainierte Modell wird dann mit dem Validierungsdatensatz getestet, wobei die Performanz sich ungefähr im Rahmen der Leistung auf dem Trainingsdatensatz bewegen sollte - ist das Ergebnis mit dem Trainingsdatensatz signifikant besser, wird von *Overfitting* gesprochen. I. d. R. wird mehrere (manchmal Hunderte) Epochen trainiert und der Prozess gestoppt, wenn ein (zu großes) *Overfitting* festgestellt wird oder sich die

Leistung auf dem Validierungsdatensatz mit steigender Epochenzahl wieder verschlechtert. Dann wird das Modell der Epoche, welche auf dem Validierungsdatensatz die beste Performanz zeigte, endgültig auf dem Testdatensatz getestet und das Experiment damit beendet.

Im Rahmen dieser Arbeit wird das *Adam* Verfahren Anwendung finden, welches eine Erweiterung von MBSGD darstellt.

3 Datensatz

3.1 Überblick

Der verwendete Datensatz, welcher als Teil dieser Arbeit eigens generiert wurde, besteht aus insgesamt 30.000 einzelnen Tests, welche als *Diagrammsequenzen* bezeichnet werden. Eine einzelne Diagrammsequenz besteht dabei aus einer Abfolge von 6 Bildern (*Diagrammen*), in denen geometrische Formen in einem 4x4 Raster zu sehen sind. Wie eingangs erwähnt, liegen diesen geometrischen Formen über die gesamte Diagrammsequenz hinweg bestimmte Ablaufmuster zugrunde. Diese müssen in den ersten fünf Bildern erkannt werden, um das sechste Bild zu generieren. Ein Beispieltest ist in Abb. 3.1 abgebildet.



Abbildung 3.1: Eine Diagrammsequenz aus dem Datensatz.

Hier ist zu erkennen, dass sich unten links immer ein hellgrauer Kreis und ein weißes Fünfeck abwechseln. Die Formen besitzen außerdem eine bestimmte Abfolge von Füllungen, welches hier daraus besteht, dass sich eine vertikale und horizontale Linie bei jedem neuen Bild abwechseln. Also muss im sechsten Bild an dieser Stelle ein weißes Fünfeck mit einer horizontalen Linie zu sehen sein. Darüber hinaus sind Bewegungen zentraler Bestandteil der Diagrammsequenzen. In der Mitte des Beispiels ist zu sehen, wie Symbole zwischen zwei Positionen oszillieren.



Abbildung 3.2: Symbole bewegen sich über den Bildrand.

In Abb. 3.2 ist eine Diagrammsequenz abgebildet, bei welcher Symbole sich über den Bildrand hinaus bewegen und ihre Bewegung am gegenüberliegenden Bildrand fortsetzen (rot umrandet). Es sind Bewegungen horizontaler, vertikaler und diagonaler Art möglich.



Abbildung 3.3: Eine Diagrammsequenz ohne Bewegungen.

Abb. 3.3 zeigt eine Diagrammsequenz ganz ohne Bewegungen, dafür mit einer recht hohen Anzahl von Symbolen.

Durch eine Vielzahl möglicher Symbole und Bewegungen kann so eine große Anzahl komplexer Diagrammsequenzen erzeugt werden. Wie erwähnt umfasst der Datensatz insgesamt 30.000 Tests (also insgesamt 180.000 einzelne Bilder in Grautönen), wobei die Aufteilung in Trainings-, Validierungs- und Testdatensatz 24.000/3.000/3.000 beträgt.

3.2 Erzeugung

Die Erzeugung einer Diagrammsequenz soll im Folgenden Schritt für Schritt erläutert werden.

Vorarbeit: Symbole und Sequenzmuster festlegen

Zunächst werden die Symbole erstellt, welche Bestandteile der Diagramme sein sollen. Es wird dabei zwischen äußeren Symbolen und inneren Symbolen unterschieden. Äußere Symbole sind beispielsweise ein Kreis oder ein Quadrat. Innere Symbole sind Formen wie etwa eine horizontale oder vertikale Linie, welche innerhalb der äußeren Symbole auftreten. So entstehen letztendlich Gesamtsymbole wie ein Fünfeck mit einer vertikalen Linie (Abb. 3.4). Die Gesamtheit aller Symbole ist im Anhang unter A.1 zu finden.

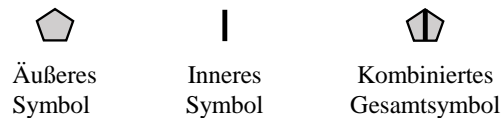


Abbildung 3.4: Ein Beispiel für ein äußeres, inneres und daraus kombiniertes Gesamtsymbol.

Es werden außerdem verschiedene Sequenzmuster festgelegt, die in einem späteren Schritt mit normalen und inneren Symbolen gefüllt werden. Ein Beispiel für ein Sequenzmuster ist $[1, 2, 1, 2, 1, 2]$: Hier wechseln sich zwei Symbole (Platzhalter 1 und 2) bei jedem neuen Bild ab. Innere Symbole, äußere Symbole und Sequenzmuster werden in einem späteren Schritt verwendet, um Symbolsequenzen zu bilden. Ein Beispiel ist in Abb. 3.5 dargestellt. Alle verwendeten Sequenzmuster sind im Anhang unter A.1 aufgelistet.







Symbolsequenz						
Äußeres Sequenzmuster	1	2	1	2	1	2
Inneres Sequenzmuster	1	1	2	1	1	2

Abbildung 3.5: Eine Symbolsequenz mit äußerem und innerem Muster.

Es gibt ein äußeres und ein inneres Sequenzmuster. Hier wurde bei dem äußeren Sequenzmuster die 1 mit einem hellgrauen Fünfeck belegt und die 2 mit einem weißen Quadrat belegt. Bei dem inneren Sequenzmuster wird die 1 mit einer vertikalen Linie

und die 2 mit gar keinem inneren Symbol belegt. Daraus ergibt sich die abgebildete Symbolsequenz über insgesamt sechs Diagramme.

Darüber hinaus werden Bewegungsmuster erstellt, denn die Symbole eines Teils der Sequenzmuster, die in den Diagrammen dargestellt werden, sollen außerdem auch ihre Position verändern. Ein Beispiel für ein Bewegungsmuster ist $[(1, 0), (1, 0), (1, 0), (1, 0), (1, 0)]$, hier wird die Bewegung zwischen zwei Diagrammen als ein Tupel (x -Richtung, y -Richtung) dargestellt. Bei diesem Beispiel bewegt sich das Symbol also immer einen Schritt nach rechts. Auch für die Bewegungsmuster findet sich ein Überblick im Anhang unter A.1.

1. Schritt: Bildstruktur abstrahieren

Ein einzelnes Diagramm kann maximal 16 Symbole enthalten, welche in einem Gitter von 4 Reihen und 4 Spalten angeordnet sind. Dies bedeutet, dass ein Diagramm zunächst auch als 4x4-Array abstrahiert werden kann. Bei einer Sequenzlänge von 6 entsteht so zunächst ein dreidimensionales Array mit den Maßen 4x4x6, s. Abb. 3.6

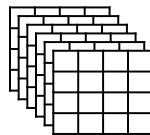


Abbildung 3.6: Abstraktion der Diagrammsequenz als 4x4x6-Array.

2. Platzhalter für bewegliche Sequenzen festlegen

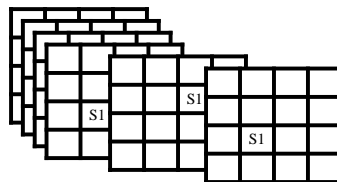


Abbildung 3.7: Das Array mit einem Platzhalter für eine bewegliche Sequenz.

Im nächsten Schritt wird das 4x4x6-Array mit Platzhaltern für die beweglichen Symbolsequenzen belegt. Hierzu werden die Bewegungsmuster verwendet. Ein Test kann zwischen 0 und 4 bewegliche Sequenzen enthalten. Wenn sich ein Symbol über eine Position am Bildrand hinaus bewegt, wird die Bewegung an der Position am gegenüberliegenden

Bildrand fortgesetzt. Im folgenden Beispiel werden Plätze im Array mit S1 belegt (s. Abb. 3.7), diese werden in Schritt 4 mit einer Symbolsequenz belegt.

3. Platzhalter für unbewegliche Sequenzen festlegen

Anschließend werden die Felder ermittelt, welche über alle sechs Diagramme hinweg noch von keinem Symbol belegt sind (farblich markiert), dargestellt in Abb. 3.8.

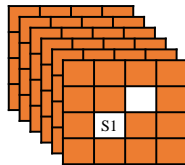


Abbildung 3.8: Markierung noch freier Felder in Orange für unbewegliche Sequenzen.

Jede dieser Positionen wird dann mit einer Wahrscheinlichkeit von 40 % mit einer Sequenz befüllt, s. Abb. 3.9.

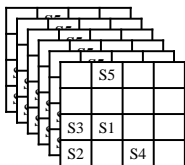


Abbildung 3.9: Mit unbeweglichen Sequenzen befülltes Array.

4. Platzhalter mit Symbolen füllen

Die Platzhalter (S1, S2,...) werden nun mit Mustern und Symbolen befüllt. Hierzu wird jedem Platzhalter zunächst ein inneres und äußeres Sequenzmuster zugeordnet, wie beispielsweise:

S1.außenMuster = 121212

S2.innenMuster = 112233

Anschließend werden die Ziffern der Muster zufällig mit Symbolen belegt, dargestellt in Abb. 3.10.

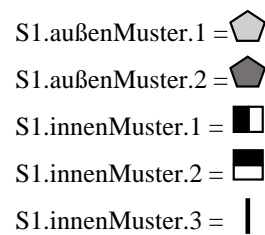


Abbildung 3.10: Zuordnung der Symbole zu den Sequenzen.

5. In Bilddateien umwandeln

Mit dem belegten 4x4x6 Array und den zugewiesenen Symbolen können dann die einzelnen Bilder erzeugt werden, s. Abb. 3.11.

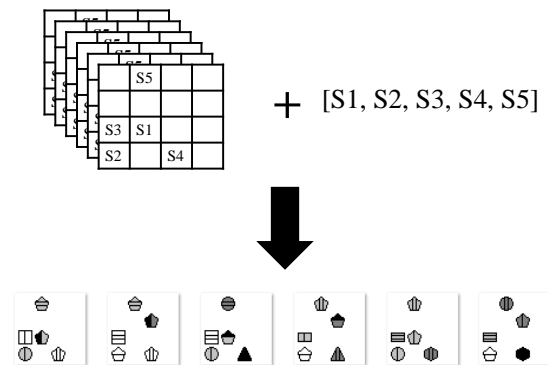


Abbildung 3.11: Erzeugung der fertigen Diagrammsequenz.

4 Technische Umsetzung

Die gesamte Implementierung erfolgt in Python 3, einer Programmiersprache, welche dynamische Typisierung bietet und den Anspruch verfolgt, intuitiv verständlich zu sein [44]. Durch eine große Auswahl bestehender - und weiterhin gewarteter und weiterentwickelter - Bibliotheken, welche sich für die Bereiche wissenschaftliches Rechnen, Datenanalyse und maschinelles Lernen eignen und gegenseitig ergänzen, konnte sich Python insbesondere in diesen Domänen etablieren. Alle verwendeten Bibliotheken sind im Anhang unter A.2 aufgeführt.

Für das Erstellen und Trainieren der Modelle wird Keras¹ mit Tensorflow² als Backend verwendet. Tensorflow ist ein Framework, welches Funktionen für effiziente Rechenoperationen in Graphen (wie beispielsweise neuronalen Netzen) bereitstellt und insbesondere Unterstützung für die Verwendung von GPUs (*Graphical Processing Units*) liefert. Dies ist wichtig, da die auf Arrays beruhenden Berechnungen während des Trainings eines neuronalen Netzes damit stark beschleunigt und in vielen Fällen überhaupt erst in einem vertretbaren Zeitraum durchgeführt werden können.

Das Framework Keras kann als *Wrapper* für ML-Bibliotheken wie Tensorflow bezeichnet werden. Es stellt einfach zugängliche Funktionen für die typischen Bestandteile eines neuronalen Netzes zur Verfügung und stellt damit eine Abstraktion des Backends dar, welches sich sozusagen näher an der Mathematik und Technik befindet. Bedingt dadurch geht ein Teil an Flexibilität bei der Erstellung von Modellen verloren, für jedes Modell im Rahmen dieser Arbeit ist Keras jedoch vollkommen ausreichend. Die verwendeten Layer- und Optimierereinstellungen finden sich im Anhang unter A.3.2.

Als Umgebung wird das JupyterHub³ der HAW Hamburg verwendet⁴, welches eine NVIDIA Tesla V100 GPU mit 16 GB RAM für das Training zur Verfügung stellt.

¹keras.io

²tensorflow.org

³jupyter.org

⁴jupyter.icc.informatik.haw-hamburg.de

5 Allgemeine Modelleinstellungen und Hyperparameter

Im Rahmen dieser Arbeit wird eine Vielzahl an Modellen untersucht. Es ist sinnvoll, möglichst viele Modell- und Hyperparametereinstellungen für alle Modelle im Voraus weitestgehend festzuhalten und nicht zu variieren, um die Anzahl der Experimente überschaubar zu halten (*Fluch der Dimensionalität*).

5.1 Normierung

Die Grauwerte der Pixel der Diagramme werden auf Werte zwischen 0,0 (Grauwert 0) und 1,0 (Grauwert 255) normiert.

5.2 Convolutional Layers

Zur Lösung der Tests sollten die Modelle sowohl einen verarbeitenden als auch einen generierenden Teil besitzen. Deshalb wird zunächst ein Convolutional Autoencoder herangezogen, welcher in der Lage ist die einzelnen Diagramme unter möglichst geringem Informationsverlust wiederherzustellen (d. h. alle wichtigen Linien sind erkennbar, wenig Rauschen). In der Annahme, dass sich diese Architektur im Allgemeinen gut für die Verarbeitung von Diagrammen eignet, findet sich ihr Aufbau in den einzelnen Ansätzen wieder.

Bei Verwendung eines Autoencoders muss das Bild nach erfolgtem *Encoding* wieder von einer geringen Höhe und Breite in die Originaldimensionen überführt werden. Hierzu wird UpSampling verwendet, dargestellt in Abb. 5.1. Es kann als Umkehrung von Max Pooling verstanden werden: Beim verwendeten (2,2)-UpSampling wird die Höhe

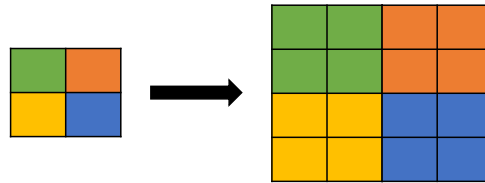


Abbildung 5.1: Prinzip des (2,2)-Upsamplings.

und Breite eines Volumens verdoppelt, die Tiefe bleibt jedoch gleich. Eine Alternative wäre Transposed Convolution [13]. Bei stichprobenartigen Tests zeigten Upsampling und Transposed Convolution vergleichbare Ergebnisse. Da Upsampling keine lernbaren Parameter benötigt, wurde sich für diese Methode entschieden, da die verwendete Parameteranzahl der Modelle ebenfalls ein Bewertungskriterium darstellt und möglichst niedrig gehalten werden soll.

Um sich auf die Merkmale fokussieren zu können, welche die einzelnen Ansätze besonders und voneinander verschieden machen, wird die CNN Struktur des Convolutional Autoencoders also für die untersuchten Modelle zu einem großen Teil festgehalten. Die Gewichte werden jedoch nicht übernommen (**kein** *Transfer Learning* [56]), sondern immer neu trainiert.

5.3 Aktivierungsfunktion

Als Aktivierungsfunktion nach jeder Schicht wird ReLU verwendet [16]:

$$\text{ReLU}(x) = \max(0, x) \quad (5.1)$$

Glorot, Bordes und Bengio [16] beschreiben, dass die Verwendung von ReLU *Sparsity* („Sparsamkeit“) begünstigt, womit Aktivierungen gemeint sind, von denen viele Einträge (nahe) null sind und einige wenige einen Wert ungleich null besitzen. Diese sind dafür jedoch mit einer großen Aussagekraft belegt. Eng damit zusammenhängend ist das sogenannte *Disentanglement* („Entflechtung/Entwirrung“) einer Aktivierung. Hierbei geht es darum, dass einzelne Elemente der Aktivierung eine ganz bestimmte Bedeutung besitzen, für welche auch nur diese Elemente verantwortlich sind. Als einfaches Beispiel könnte ein Element immer (und nur) dann hoch aktiviert sein, falls im Diagramm oben

links ein Quadrat auftritt. Häufig hilft dies sehr bei der Erklärbarkeit eines Modells (*Explainable AI*), was ebenfalls Teil dieser Arbeit sein soll und ein Grund für die Wahl von ReLU ist. Sowohl Sparsity als auch Disentanglement werden laut [16] durch die Verwendung von ReLU begünstigt, in erster Linie, da durch ihre spezielle Konstruktion Werte unter null „hart abgeschnitten werden“ (*hard zeroes*) und so im Mittel die Hälfte der Aktivierungen exakt null sind.

Ein weiterer Punkt ist die gute Eignung von ReLU für sehr tiefe Netze, wie sie auch in dieser Arbeit verwendet werden. Die Ableitungen von Aktivierungsfunktionen wie der Sigmoid- oder Tangens Hyperbolicus Funktion sind in weiten Bereichen praktisch null, wodurch insbesondere bei Netzen mit vielen Schichten (tief/*deep*) das Auftreten des *Vanishing Gradient Problems* begünstigt wird. Im Mittel sind 50 % der Ableitungen der Elemente einer ReLU-aktivierten Schicht über null, wodurch diese Problematik laut [16] mit deutlich geringerer Wahrscheinlichkeit auftritt.

5.4 Batch Normalization

Batch Normalization [25] (auch *Batch Norm* oder kurz *BN* genannt) ist ein Verfahren, bei welchem pro Batch in den Schichten innerhalb des Netzes eine Standardisierung der Aktivierungen durchgeführt wird, wobei der Grad dieser Standardisierung durch lernbare Parameter während des Trainingsprozesses angepasst werden kann. Durch dieses Verfahren soll der *Internal Covariate Shift* (ICS) ausgeglichen werden, eine Bezeichnung für das Phänomen, bei welchem standardisierte Eingangsdaten durch die Verrechnung mit Gewichtsparametern über viele Schichten hinweg ihre Standardisierung verlieren.

In dieser Arbeit erfolgt nach jeder ReLU Aktivierung ein Batch Normalization Layer. Es herrscht in der Fachwelt keine eindeutige Klarheit darüber, ob BN vor oder nach einer nicht-linearen Aktivierungsfunktion die größere Wirkung zeigt, laut der Originalpublikation [25] ist es vor der Aktivierung. Interessanterweise existiert jedoch ein Beitrag von Francois Chollet (Entwickler des Keras Frameworks, Kollege von Szegedy, einem Autor der BN Publikation) in einem Fachforum¹, welchem zufolge Szegedy in Experimenten BN nach der Aktivierungsfunktion verwendet. In stichprobenartigen Tests im Rahmen dieser Arbeit hat sich BN nach der Aktivierungsfunktion als wirkungsvoller gezeigt (schnelleres Training, bessere Vorhersagen), weshalb diese Variante verwendet werden wird.

¹github.com/keras-team/keras/issues/1802#issuecomment-187966878, abgerufen am 15.12.2019.

5.5 Output und Kostenfunktion

Die Erzeugung des sechsten Bildes der Diagrammsequenzen wird als Regressionsproblem definiert (Vorhersage der Grauwerte aus einem kontinuierlichen Wertebereich zwischen 0 und 1). Die Aktivierungsfunktion der Outputschicht ist linear. Als Kostenfunktion wird der mittlere absolute Fehler verwendet (*Mean Absolute Error*, MAE [49]):

$$MAE = \frac{1}{m} \sum_{i=1}^m |\hat{y}_i - y_i| \quad (5.2)$$

Bei \hat{y}_i handelt es sich dabei um den Ist-Wert und bei y_i um den Soll-Wert. Eine weitere Option wäre die Verwendung von MSE [49] (*Mean Squared Error*), in einzelnen Tests zeigte MAE jedoch die besseren Ergebnisse und insbesondere klarere Symbole, wenn auch teilweise falsch. Möglicherweise ist dies darauf zurückzuführen, dass der Fehler im Gegensatz zu MSE nicht quadriert wird und das Modell so ein „höheres Risiko“ bei der Vorhersage eingeht.

Eine weitere Option wäre die Verwendung einer Sigmoid Aktivierungsfunktion, um die Ausgangswerte zwischen 0 und 1 zu normieren, in Verbindung mit einer *Binary Cross Entropy* Kostenfunktion [48]. Aber auch diese Variante zeigte schlechtere Ergebnisse.

5.6 Optimierer, Initialisierung und Batch Size

Zur Optimierung wird Adam verwendet [29], eine Erweiterung des Minibatch Stochastic Gradient Descent Verfahrens. Hierbei wird das Verfahren um eine adaptive Lernrate und einen Momentumterm erweitert, womit Oszillationen der Parameter verringert werden, Plateaus leichter überwunden und ein lokales Minimum in vielen Anwendungsfällen in weniger Iterationen gefunden wird [29]. Die genauen Einstellungen finden sich im Anhang unter A.3.2. Die Gewichte werden mit der *Glorot uniform* Methode (auch *Xaver uniform* genannt) initialisiert, einem Verfahren, welches die Häufigkeit des Auftretens des Vanishing Gradient Problems reduzieren konnte [15]. Die verwendete Batch Size ist 32.

5.7 Evaluation

Die trainierten Modelle werden auf dem Testdatensatz evaluiert. Hierbei wird der mittlere absolute Fehler (MAE) für die Bewertung verwendet. Damit findet eine Auswertung auf Pixelebene statt. Eine Alternative wäre gewesen, die Anzahl richtiger Symbole explizit zu ermitteln. Da sich jedoch zeigte, dass die Symbole häufig verschwommen und undeutlich sind, ist es schwierig zu bestimmen, ab welcher Deutlichkeit ein Symbol als richtig generiert gilt. Dieser Umstand findet bei einem Vergleich der einzelnen Pixel größere Beachtung, mit welchem möglichst große Objektivität erreicht werden soll. Außerdem werden einige Vorhersagen der Modelle abgebildet, wodurch die reinen Zahlenwerte besser eingeordnet und weitere Erkenntnisse erlangt werden können.

Zum Verständnis ihrer Funktionsweise und für eine bessere Interpretierbarkeit der Ergebnisse werden darüber hinaus einige Aktivierungen und/oder Feature Maps innerhalb der Modelle visualisiert. (Anmerkung: In den Visualisierungen sind hohe Aktivierungen stets hell gekennzeichnet.)

6 Erster Ansatz: CNN-RNN

Die beschriebene Aufgabe ist stark verwandt mit *Frame Prediction*, bei welcher es ebenfalls darum geht aus einer Folge von Einzelbildern eines Videos (Frames) das folgende Bild zu generieren [55][8][37]. Ein häufig gewählter Ansatz ist dabei die Kombination von CNNs mit LSTMs oder GRUs, welcher im Folgenden als erster in dieser Arbeit erprobt werden soll.

6.1 Modelle

Die verwendete Architektur ist in Abb. 6.1 dargestellt. Wie beschrieben (vgl. Abschnitt 5.2) handelte es sich hierbei ursprünglich um einen Autoencoder, welcher um eine rekurrente Einheit erweitert wurde.

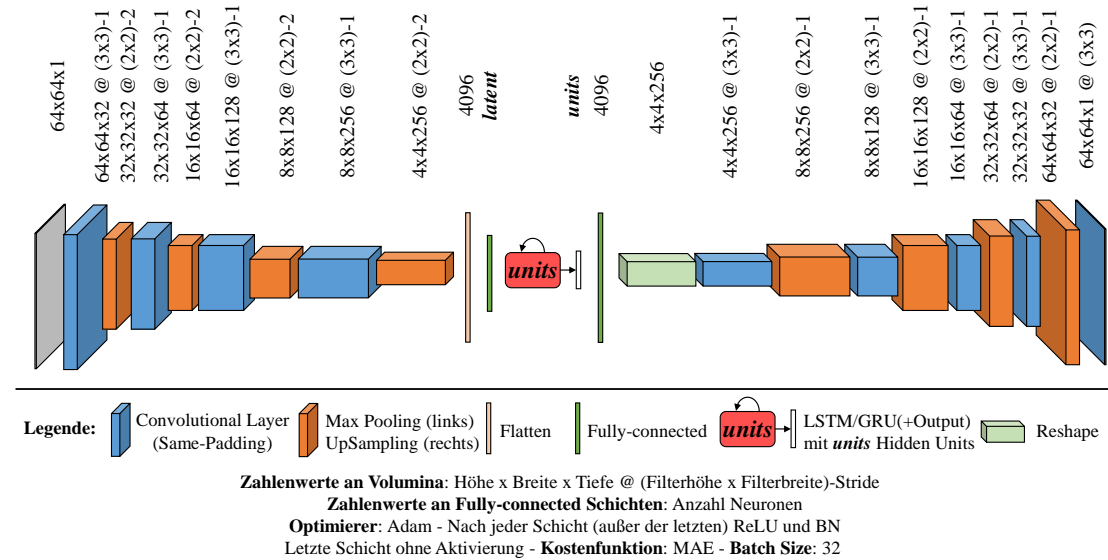


Abbildung 6.1: CNN-RNN Grundarchitektur.

Die Convolutional Layers verwenden ein *Same*-Padding, d. h. das Outputvolumen hat die gleiche Höhe und Breite wie der Input. Zunächst durchlaufen die fünf Eingangsbilder jeweils einzeln mehrere Convolutional und Max Pooling Layer bis eine Höhe und Breite von $4 \cdot 4$ erreicht wurde, was mit der Anzahl möglicher Symbole pro Spalte bzw. Zeile korrespondiert. Die folgende Verarbeitung unter Verwendung eines LSTMs/einer GRU wird in Abb. 6.2 detailliert dargestellt.

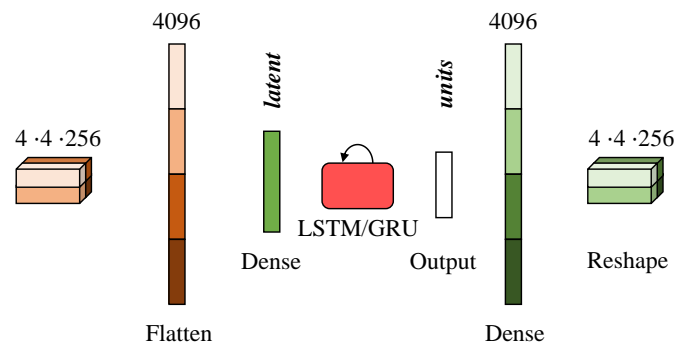


Abbildung 6.2: Detaillierte Darstellung der Kombination eines CNNs mit LSTM/GRU.

Das Volumen wird einer Flatten-Operation unterzogen. Wie schon in Abschnitt 2.3.2 beschrieben, werden hierbei schlicht alle Reihen des Volumens zu einem langen Vektor aneinandergereiht (bei diesem Modell ergibt sich damit ein Vektor mit 4096 Dimensionen) und anschließend *Fully-connected* in einen Vektor mit *latent* < 4096 Neuronen überführt. Dies ist notwendig, da ein LSTM/GRU einen Vektor als Eingabe erwartet und das Beibehalten von 4096 Elementen die Anzahl der Parameter zu sehr in die Höhe treiben würde. Außerdem stellt die Fully-connected Schicht eine weitere Feature Extraction für die rekurrente Einheit dar. Wie die Bezeichnung der Neuronenanzahl vermuten lässt, handelt es sich hierbei um den *Latent Vector* des ursprünglichen Autoencoders. Dieser wird dann in eine rekurrente Einheit (LSTM/GRU) mit der Unit Size *units* gegeben.

Nach dem fünften Bild soll das sechste erzeugt werden. Hierzu wird zunächst der Output der rekurrenten Einheit - ein Vektor mit *units* Elementen, s. Grundlagenteil LSTM und GRU - mithilfe einer weiteren Fully-connected Schicht wieder in eine Länge von 4096 Elementen überführt, um dann einer *Reshape*-Operation unterzogen zu werden. Diese stellt quasi die Umkehrung der Flatten-Operation dar (keine lernbaren Gewichte) und erzeugt aus dem Vektor ein Volumen durch eine schlichte Umsortierung der Elemente. Dieses kann dann als Input eines Convolutional Layers verwendet werden.

Nach dem fünften Bild wird das sechste Bild mit einer „auffaltenden“ Struktur erzeugt, welche den linken Teil der Architektur spiegelt. Statt Max Pooling („DownSampling“) wird wie beschrieben UpSampling verwendet (vgl. Abschnitt 5.2).

Das Verarbeitungsprinzip einer gesamten Sequenz von fünf Diagrammen ist abschließend in Abb. 6.3 dargestellt. Es ist zu sehen, wie von dem linken, faltenden CNN-Teil die einzelnen Diagramme entgegengenommen und verarbeitet werden. Anschließend fließt der beschriebene Vektor pro Diagramm/Zeitschritt in das LSTM/GRU. Nach dem fünften Diagramm wird die Vorhersage erzeugt.

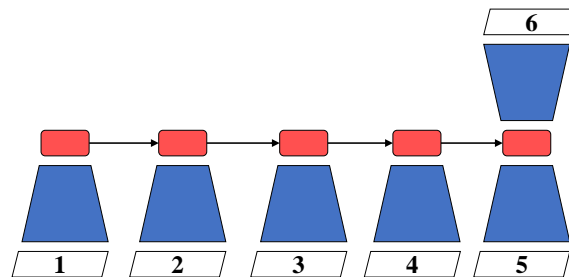


Abbildung 6.3: Verarbeitung einer Diagrammsequenz.

Mit verschiedenen Werten für *latent* werden verschiedenen „Flaschenhalsgrößen“ der Informationen, die der rekurrenten Einheit übergeben werden, erprobt. Durch die Variation von *units* soll das Verhalten der LSTMs und GRUs bei verschiedenen Unit Sizes erprobt werden. Die Modelle tragen die Namen „CNN-(LSTM|GRU)-*units-latent*“.

6.2 Diskussion

Ergebnisse CNN-LSTM

Zunächst sollen die CNN-LSTM Modelle betrachtet werden, die Evaluierungsergebnisse sind in Tabelle 6.1 aufgeführt.

Name	<i>units</i>	<i>latent</i>	Anzahl Parameter	MAE
CNN-LSTM-64-64	64	64	1.930.817	$9,645 \cdot 10^{-2}$
CNN-LSTM-128-64	128	64	2.258.753	$8,394 \cdot 10^{-2}$
CNN-LSTM-256-64	256	64	3.012.929	$7,288 \cdot 10^{-2}$
CNN-LSTM-512-64	512	64	4.914.577	$7,279 \cdot 10^{-2}$
CNN-LSTM-1024-64	1024	64	10.290.497	$7,357 \cdot 10^{-2}$
CNN-LSTM-512-16	512	16	4.619.537	$7,385 \cdot 10^{-2}$
CNN-LSTM-512-32	512	32	4.717.857	$7,315 \cdot 10^{-2}$
CNN-LSTM-512-128	512	128	5.307.777	$8,435 \cdot 10^{-2}$
CNN-LSTM-512-256	512	256	6.094.337	$9,072 \cdot 10^{-2}$

Tabelle 6.1: Ergebnisse CNN-LSTM Architekturen.

Die besten Ergebnisse erzielt CNN-LSTM-512-64, es scheint ein Plateau erreicht, da sich bei Erhöhung der *units* und *latent* Größe die Ergebnisse wieder verschlechtern. Durch die sich ergebende hohe Parameterzahl unter Verwendung größerer *units* und *latent* Werte wird ein Overfitting hier möglicherweise begünstigt.



Abbildung 6.4: Beispiel einer Vorhersage von CNN-LSTM-512-64.

Ein typisches Beispiel einer Vorhersage wird in Abb. 6.4 gegeben. Es ist zu sehen, dass die Positionen der vorherzusagenden Symbole zwar richtig erkannt werden, die Symbole selbst jedoch verschwommen sind.

Ergebnisse CNN-GRU

Es folgen die Evaluierungsergebnisse der CNN-GRU Modelle, sie sind in Tabelle 6.2 aufgelistet.

Name	<i>units</i>	<i>latent</i>	Anzahl Parameter	MAE
CNN-GRU-64-64	64	64	1.922.561	$9,909 \cdot 10^{-2}$
CNN-GRU-128-64	128	64	2.234.049	$9,856 \cdot 10^{-2}$
CNN-GRU-256-64	256	64	2.930.753	$7,205 \cdot 10^{-2}$
CNN-GRU-512-64	512	64	4.619.073	$7,318 \cdot 10^{-2}$
CNN-GRU-1024-64	1024	64	9.217.149	$7,246 \cdot 10^{-2}$
CNN-GRU-512-16	512	16	4.348.689	$7,362 \cdot 10^{-2}$
CNN-GRU-512-32	512	32	4.438.817	$7,221 \cdot 10^{-2}$
CNN-GRU-512-128	512	128	4.979.585	$7,080 \cdot 10^{-2}$
CNN-GRU-512-256	512	256	6.094.337	$10,177 \cdot 10^{-2}$

Tabelle 6.2: Ergebnisse CNN-GRU Architekturen.

Im Wesentlichen spiegeln die Ergebnisse die des CNN-LSTMs wider. Das beste Modell ist CNN-GRU-512-128, welches eine leicht bessere Performanz als das CNN-LSTM-512-64 zeigt und damit das beste Modell für den CNN-RNN Ansatz darstellt. In Abb. 6.5 ist zu erkennen, dass auch bei diesem Modell die Positionen erkannt werden, die Symbole jedoch nicht.



Abbildung 6.5: Beispiel einer Vorhersage von CNN-GRU-512-128.

Im Folgenden soll versucht werden, die innere Funktionsweise des CNN-GRU-512-128-Modells zu interpretieren. Dazu werden Feature Maps und Hidden State Aktivierungen herangezogen.

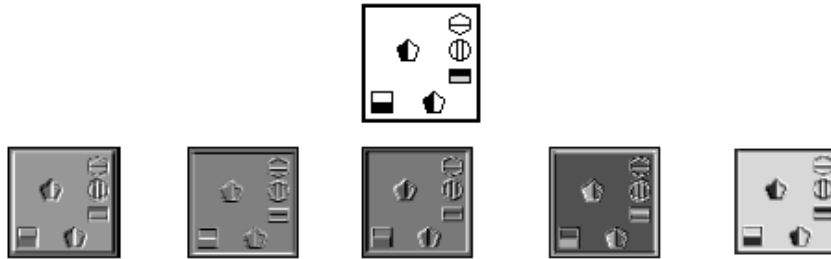


Abbildung 6.6: Einige Feature Maps des ersten Layers von CNN-GRU-512-128.

Abb. 6.6 zeigt, dass für CNNs „typische“ Filter erlernt werden, wie beispielsweise für die Erkennung dunkler oder heller Bereiche, Hell-dunkel-Übergänge oder Linien einer bestimmten Ausrichtung.

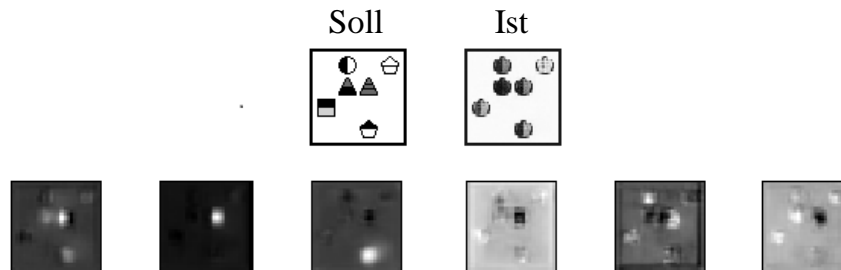


Abbildung 6.7: Einige Feature Maps des vorletzten Layers von CNN-GRU-512-128.

In Abb. 6.7 sind die Feature Maps des vorletzten Convolutional Layers dargestellt, also dem Teil des Modells, welches für die Generierung des sechsten Bildes zuständig ist (die letzte Schicht besitzt nur eine „Feature Map“ - die Vorhersage selbst). Es ist zu sehen, dass zwischen Filtern unterschieden werden kann, welche einerseits einzelne Symbole generieren oder andererseits mehrere Symbole bzw. größere Bereiche erfassen. Insbesondere ist jedoch zu erkennen, dass die Aktivierungen sehr unscharf sind, was sich in den unklaren Symbolen der generierten Bilder widerspiegelt.

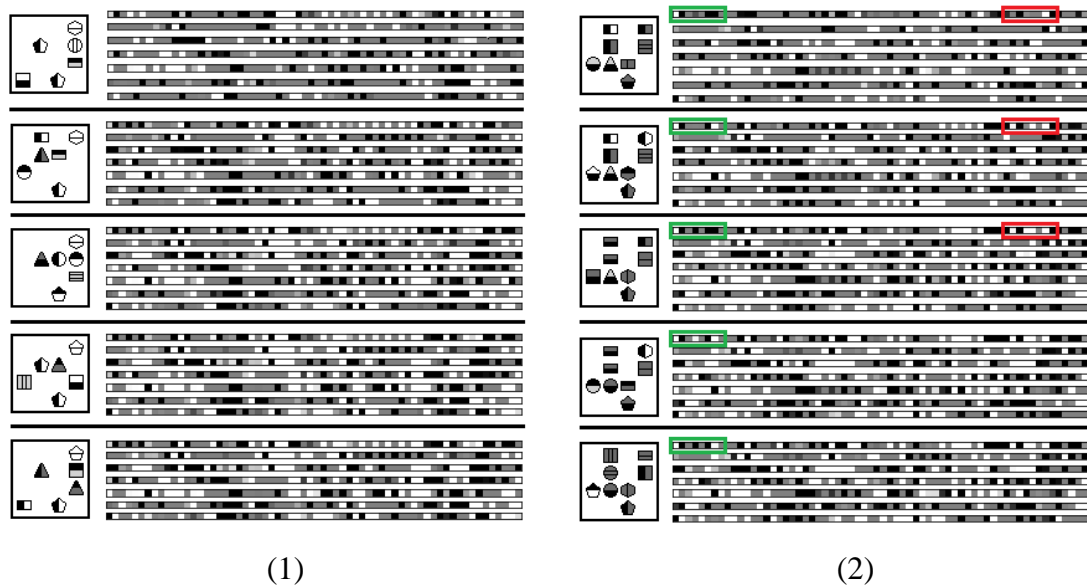


Abbildung 6.8: Hidden State Aktivierungen von CNN-GRU-512-128 für zwei verschiedene Diagrammsequenzen.

Abb. 6.8 stellt die Hidden State Aktivierungen des CNN-GRU-512-128-Modells dar. Ein Hidden State besitzt 512 Elemente, welche hier in 7 Zeilen mit 64 Einträgen dargestellt sind. Hohe Aktivierungen sind hell visualisiert. Es ist zu sehen, wie die Aktivierungen über die Zeitschritte eher mehr werden (z. B. rote Markierung) und nur wenig Information wieder gelöscht wird. Einige Aktivierungen werden jedoch über die Zeitschritte auch weniger oder „oszillieren“ (grüne Markierung), möglicherweise zurückzuführen auf die verwendeten Muster, die alle gemein haben, dass sich Symbole nach spätestens dem dritten Schritt wiederholen. Es existieren hoch aktivierte und inaktive Bereiche, je nach Diagramm unterschiedlich - eventuell ein Hinweis auf Disentanglement (vgl. Abschnitt 5.3).

Schlussfolgerung

Im Rahmen der Literaturrecherche dieser Arbeit konnte keine allgemeingültige Aussage darüber gefunden werden, ob GRUs bessere Ergebnisse als LSTMs liefern oder umgekehrt [58][9][14]. Es scheint auf den Anwendungsfall anzukommen und häufig zeigen beide Ansätze eine ähnlich Performanz, wie es auch bei diesem Ansatz zu beobachten ist.

Bei beiden Ansätzen, LSTM und GRU, werden die Positionen der vorherzusagenden Symbole richtig erkannt. Die Symbole selbst sind jedoch undeutlich und wirken eher wie

der Mittelwert aller möglichen Symbole. Ein möglicher Erklärungsansatz wäre, dass es für die Minimierung der Fehlerfunktion wichtiger ist, die grundlegende Position (größerer Fehler) eines eher groben Symbols vorherzusagen als die richtige Generierung feiner Linien (kleinerer Fehler). Unter Umständen könnte auch ein größerer Datensatz die Ergebnisse verbessern.

In den folgenden Ansätzen sollen mögliche Probleme der CNN-RNN-Modelle aufgegriffen und verbessert werden.

7 Zweiter Ansatz: CNN-ConvLSTM

Die Ergebnisse der ersten Ansatzes sind noch verbesserungswürdig. Bei den CNN-(LSTM/GRU)-Modellen wird vor dem Eingang in die rekurrente Einheit einer Flatten-Operation unterzogen. Hierbei gehen räumliche Informationen wie Nachbarschaftsbeziehungen verloren, obwohl CNNs ursprünglich anstelle von MLPs verwendet wurden, um ebendiese zu erhalten. Möglicherweise stellt dies einen Nachteil für das Training dar. Mit dem *Convolutional LSTM* schlagen Shi et al. ein rekurrentes Modul vor, welches räumliche Informationen erhalten kann [51].

7.1 Convolutional LSTM

Die Schnittstelle eines Convolutional LSTMs [51] (ConvLSTM) erwartet ein Volumen und gibt ebenfalls ein Volumen zurück (s. Abb. 7.1).

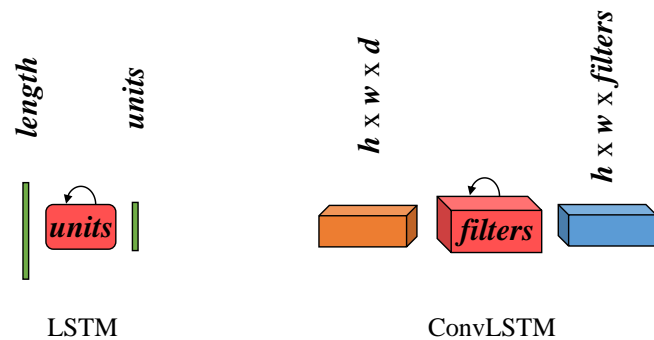


Abbildung 7.1: Vergleich der Schnittstellen von LSTM und Convolutional LSTM (bei Verwendung eines Same-Paddings).

Der Input zu einem Zeitpunkt t , X_t , ist somit ein 3D Tensor, ebenso der Cell State C_t und der Hidden State H_t . Analog zum LSTM werden ein Input Gate I_t , ein Forget Gate

F_t und ein Candidate Gate G_t definiert - allesamt 3D Tensoren:

$$I_t = \sigma(W_{xi} * X_t + W_{hi} * H_{t-1} + W_{ci} \circ C_{t-1} + b_i) \quad (7.1)$$

$$F_t = \sigma(W_{xf} * X_t + W_{hf} * H_{t-1} + W_{cf} \circ C_{t-1} + b_f) \quad (7.2)$$

$$G_t = \tanh(W_{xg} * X_t + W_{hg} * H_{t-1} + b_g) \quad (7.3)$$

Mit W werden auch hier lernbare Gewichte bezeichnet. Im Gegensatz zur LSTM Variante, welche im Grundlagenteil vorgestellt wurde (Abschnitt 2.3.3), fließt bei I_t und F_t auch der Cell State in die Berechnung der Gates mit ein - ein Ansatz, welcher ursprünglich von Alex Graves stammt [18]. Der Input X_t und der Hidden State H_{t-1} werden einer Faltung $*$ mit lernbaren Filtern unterzogen, hierbei handelt es sich um eine Convolution, wie sie in Kap. 2.3.2 beschrieben wird. Es wird ein *Same*-Padding verwendet. Die Tiefe von H_t und C_t und den Gates entspricht der Anzahl der Filter, welche als Hyperparameter festgelegt werden.

Ebenso wird der Cell State analog zum LSTM aktualisiert:

$$C_t = F_t \circ C_{t-1} + I_t \circ G_t \quad (7.4)$$

Unter Verwendung des neuen Cell States wird das Output Gate O_t (ebenfalls ein 3D Tensor und mithilfe von C_t) aktualisiert und unter dessen Verwendung letztendlich der Hidden State H_t , welcher als Output dient:

$$O_t = \sigma(W_{xo} * X_t + W_{ho} * H_{t-1} + W_{co} \circ C_t + b_o) \quad (7.5)$$

$$H_t = O_t \circ \tanh(C_t) \quad (7.6)$$

7.2 Modelle

Die Struktur des „faltenden“ und „auffaltenden“ CNNs wurden von Ansatz 1 übernommen. Auf die Flatten-, Fully-connected und Reshape Schichten kann verzichtet werden und die komplette Architektur (mit Ausnahme der Hadamard-Produkte innerhalb des ConvLSTMs) ist damit *fully-convolutional*.

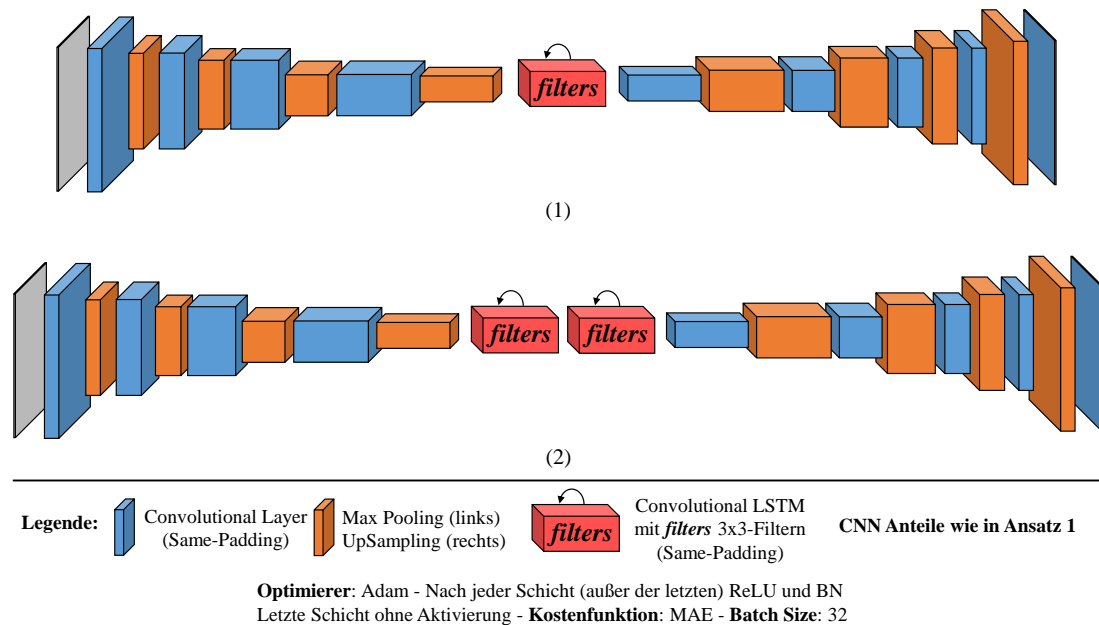


Abbildung 7.2: CNN-ConvLSTM Grundarchitekturen. Für nicht dargestellte Hyperparameter siehe Abb 6.1.

Die erprobten Architekturen sind in Abb. 7.2 abgebildet. Die ConvLSTMs verwenden stets **3x3-Filter**. Es existieren zwei Grundarchitekturen, der erste Ansatz verwendet ein einzelnes ConvLSTM, der andere zwei gestapelte (*stacked*) ConvLSTMs. Bei beiden wird die Anzahl der Filter (*filters*) variiert.

Abb. 7.3 (1) zeigt im Detail, wie der Output des ConvLSTMs (weiß) von den weiteren CNN Schichten entgegengenommen wird. Die detaillierte Verarbeitung über zwei ConvLSTMs ist ebenfalls in Abb. 7.3 (2) dargestellt. Das Volumen des ersten ConvLSTMs, welches *filters1* Feature Maps erzeugt geht bei jedem einzelnen Zeitschritt direkt in das zweite über. Nach dem fünften Diagramm beginnt dieses das Bild zu generieren, indem es *filters2* Feature Maps ausgibt, welche anschließend vom auffaltenden CNN-Teil verarbeitet werden. Für alle verwendeten Modelle gilt *filters1* = *filters2*. Ein zweites

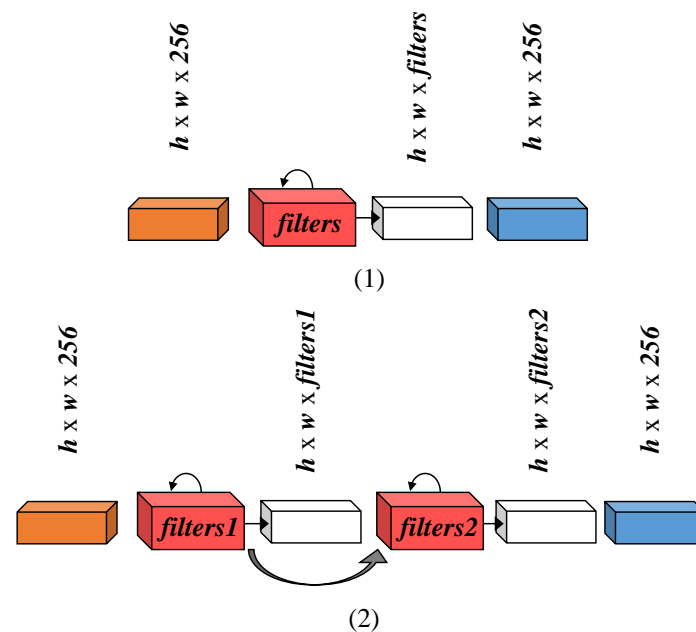


Abbildung 7.3: Detailliertes Prinzip der Verarbeitung mit Convolutional LSTMs.

ConvLSTM vergrößert in erster Linie die Kapazität des rekurrenten Abschnittes deutlich. Es ist darüber hinaus vorstellbar, dass eine Aufgabenteilung zwischen den ConvLSTMs stattfindet, das erste könnte Informationen der einzelnen Diagramme komprimieren und das zweite ist möglicherweise eher für die Generierung des sechsten Diagrammes zuständig. Für beide Varianten, einzelnes ConvLSTM und zwei gestapelte ConvLSTMs, gilt das gleiche Verarbeitungsprinzip wie es in Abb. 6.3 dargestellt ist.

Es werden Modelle mit $filters = 256$ und 512 erprobt, sie tragen die Bezeichnungen CNN-ConvLSTM-*filters* bzw. CNN-2xConvLSTM-*filters* für die Variante mit zwei gestapelten ConvLSTMs.

7.3 Diskussion

Ergebnisse - CNN-(2x)ConvLSTM

Die Ergebnisse der CNN-(2x)ConvLSTM Modelle sind in Tabelle 7.1 aufgelistet.

Name	<i>filters</i>	Anzahl Parameter	MAE
CNN-ConvLSTM-256	256	6.088.961	$4,016 \cdot 10^{-2}$
CNN-ConvLSTM-512	512	16.116.993	$3,384 \cdot 10^{-2}$
CNN-2xConvLSTM-256	2x256	10.808.577	$3,837 \cdot 10^{-2}$
CNN-2xConvLSTM-512	2x512	34.993.409	$3,181 \cdot 10^{-2}$

Tabelle 7.1: Ergebnisse CNN-ConvLSTM Architekturen.

Im Vergleich zum CNN-RNN Ansatz konnten die Ergebnisse deutlich verbessert werden. CNN-2xConvLSTM-512 ist das beste Modell, wobei diese Performanz mit über 30 Millionen Parametern teuer erkaufte wird. Jedoch kann auch CNN-ConvLSTM-256 mit einer Parameterzahl in der Größenordnung der CNN-RNNs deutlich bessere Ergebnisse erzielen.



Abbildung 7.4: Beispiel einer Vorhersage von CNN-2xConvLSTM-512.

In Abb. 7.4 ist zu sehen, dass die Symbole deutlich schärfer sind und zum Teil richtig erkannt werden.

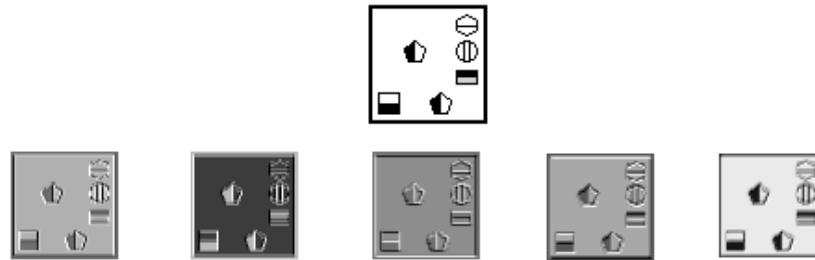


Abbildung 7.5: Beispiel einiger Feature Maps des ersten Layers von CNN-2xConvLSTM-512.

Die Feature Maps, welche im ersten Layer des CNN-2xConvLSTM-512-Modells erzeugt werden (siehe Abb. 7.5), ähneln stark denen des besten CNN-RNN-Modells, CNN-GRU-512-128 (Hell-Filter, Dunkel-Filter, Hell-dunkel-Übergänge in verschiedene Richtungen, feine Linien).

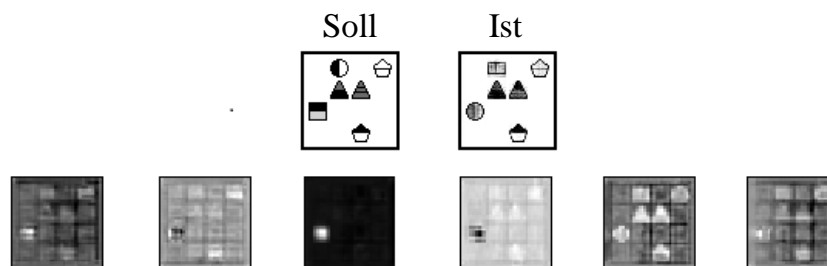


Abbildung 7.6: Beispiel einiger Feature Maps des vorletzten Layers von CNN-2xConvLSTM-512.

Auch für dieses Modell wurden die Feature Maps aus dem generierendem Modellteil untersucht, Abb 7.6 zeigt die Feature Maps aus dem vorletzten Convolutional Layer. Zu sehen sind deutlich schärfere Aktivierungen, bei denen Symbole recht klar erkennbar sind, was im Einklang mit den besseren Ergebnissen steht.

Schlussfolgerung

Die Ergebnisse konnten im Vergleich zum CNN-RNN Ansatz deutlich verbessert werden. Da die CNN-Struktur unverändert bleibt, ist es naheliegend, dass tatsächlich der Erhalt räumlicher Information für diese Verbesserung verantwortlich ist. Diese Verbesserung geht konform mit den Ergebnissen der ConvLSTM Originalpublikation [51], in welcher ebenfalls eine Verbesserung vom LSTM zum ConvLSTM festgestellt werden konnte.

Das beste Modell CNN-2xConvLSTM-512 besitzt jedoch eine sehr hohe Parameterzahl, welche für die Art des betrachteten Problem es unverhältnismäßig hoch wirkt.

8 Dritter Ansatz: FullyCNN

Bei den vorherigen Ansätzen ist zu sehen, dass die Bewegungen gut erkannt werden, es aber Probleme bei der Generierung der einzelnen Symbole gibt. Deshalb sollen die Erkennung der Bewegungen einerseits und die Verarbeitung der Symbole andererseits in jeweils einzelne Module ausgelagert werden, in der Hoffnung, dass die Symbole dadurch besser verarbeitet und anschließend generiert werden.

Ein weiterer Punkt, welcher bei dem folgenden Ansatz aufgegriffen werden soll, ist, dass die Verwendung einer rekurrenten Einheit insbesondere dann Sinn ergibt, wenn die Sequenzlänge variabel ist, wie z. B. bei der Verarbeitung verschieden langer Texte. Bei den Diagrammsequenzen ist die Länge jedoch immer gleich. Möglicherweise hebt die Verwendung einer rekurrenten Einheit das Modell also auch auf ein unnötig komplexes Niveau - grundsätzlich kann auf diese Einheit auch verzichtet werden, wenn das Problem anders modelliert wird.

Beispielsweise wählten Mnih et al. [40] für das Erlernen von Atari-Spielen (*Deep Reinforcement Learning*) CNN-Architekturen, welche als Input die neuesten vier Frames des laufenden Spieles gestapelt (*stacked*) erhalten. Der Input besitzt damit vier Kanäle und die Frames werden gemeinsam verarbeitet, im Gegensatz zu einem Modell mit rekurrenter Einheit, welches die Frames einzeln verarbeitet (wie in den Ansätzen 1 und 2). Der sequenzielle Charakter der Daten wird also nicht mittels eines LSTMs o. Ä. explizit in das Modell integriert, sondern stattdessen in die Eingabedaten eines weniger spezifischen Modells.

8.1 Dilated Convolution

Um ein Teilnetz des verwendeten Modells speziell die Bewegungen einer Sequenz erlernen zu lassen, wird *Dilated Convolution* verwendet [59].

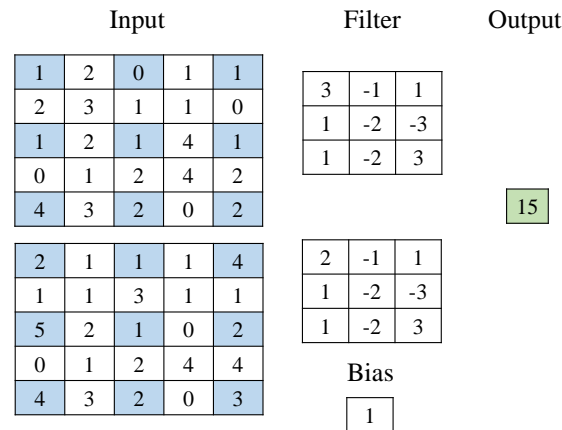


Abbildung 8.1: Beispiel einer Dilated Convolution.

Ein Beispiel für eine Dilated Convolution (*geweitete Faltung*) ist in Abb. 8.1 gegeben. Es ist zu sehen, dass nur jeder zweite Eingangswert (in blau) in die Berechnung einfließt, es wird hierbei von einer *Dilation Rate* von 2 gesprochen. Dadurch vergrößert sich der betrachtete Ausschnitt des Bildes pro Neuron, bei gleichbleibender Anzahl Parameter. Das „rezeptive Feld“ vergrößert sich. Jedoch ist die Information natürlich „größer“, da jeder zweite Bildpunkt übersprungen wird.

Dilated Convolutions konnten sich beispielsweise im Bereich *Image Segmentation* beweisen [20][61].

8.2 Modelle

ModuleCNN

Der entwickelte Ansatz, welcher in diesem Kapitel vorgestellt werden soll, nennt sich ModuleCNN und ist in Abb. 8.2 dargestellt. Es werden zwei Module verwendet, welche beide die vollständige Inputsequenz, bestehend aus 5 Diagrammen, gestapelt entgegennehmen (zu beachten ist also jeweils das Inputvolumen von $64 \cdot 64 \cdot 5$). Zum einen existiert das Bewegungsmodul, dessen Filter durch die Verwendung von Dilated Convolution einen

eher größeren, groben Überblick über die Eingabebilder erhalten und damit insbesondere die Bewegungen erfassen sollen. Zum anderen das Symbolmodul, welches normale Convolution (Dilation Rate von 1) verwendet und damit detailliert die Symbole erfassen soll. Abgesehen von der Dilation Rate verwenden beide Module die gleichen Hyperparameter. Ihre Outputvolumina werden konkateniert (\oplus) und in den auffaltenden Teil gegeben.

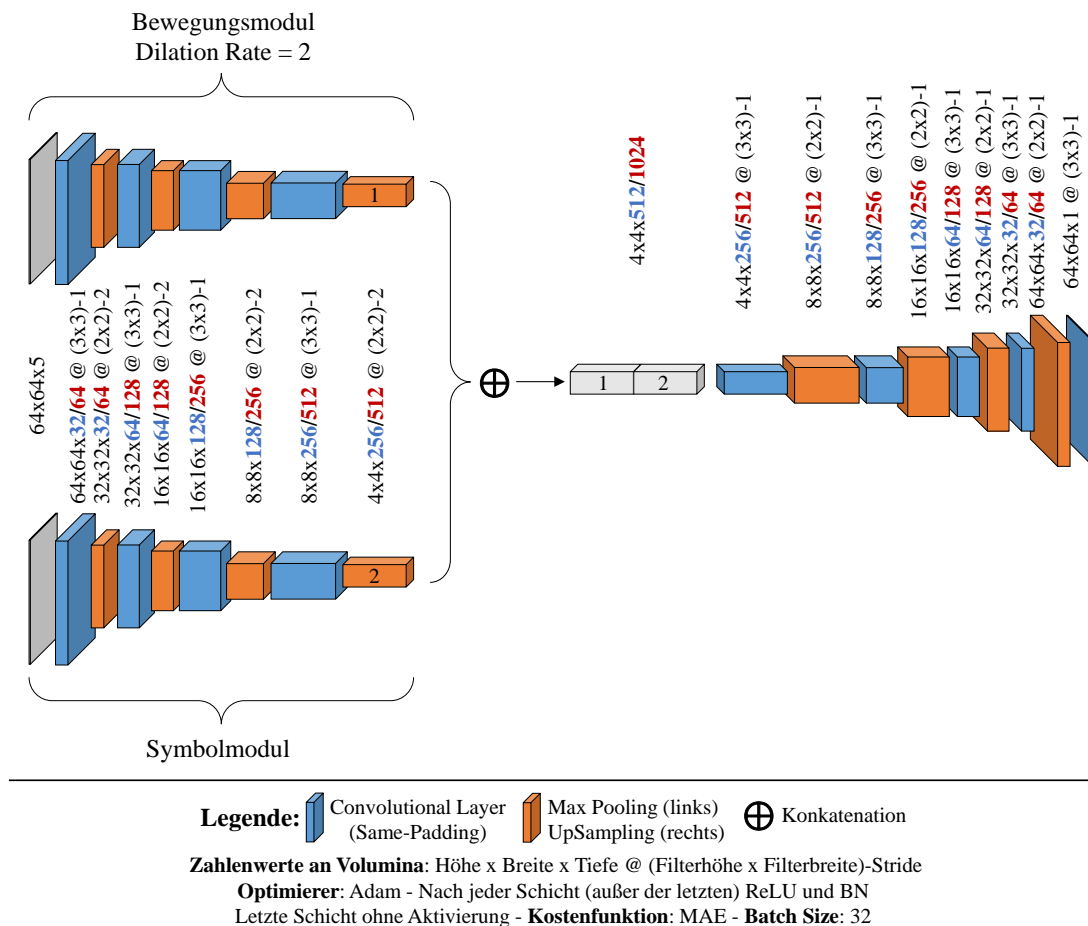


Abbildung 8.2: ModuleCNN Architektur.

Es werden zwei Module erprobt, das ModuleCNN-32 (blaue Zahlen) und das ModuleCNN-64 (rote Zahlen). Außerdem werden zum Vergleich beide Modelle auch ohne Dilated Convolution trainiert (ModuleCNN-NoDil).

SimpleCNN

Zum Vergleich soll auch ein möglichst „unvoreingenommener“, einfacher Ansatz erprobt werden: Ein schlichtes, faltendes und anschließend auffaltendes CNN. Hierbei fließen keine Annahmen über die sequenzielle Eigenschaft des Problems oder die Eigenheiten der Bilder (Bewegungs-, Symbolmodul) in das Modell ein. Diese Architektur, genannt SimpleCNN, entspricht grundsätzlich der CNN Architektur aus den Ansätzen 1 und 2, jedoch ohne eine rekurrente Einheit, siehe Abb. 8.3. Zu beachten ist auch hier die Eingangsschicht, welche die ersten fünf Diagramme gestapelt entgegennimmt. Es werden zwei Modelle erprobt, eines mit weniger Filtern, genannt SimpleCNN-32 (blaue Zahlen) und eines mit mehr Filtern, genannt SimpleCNN-64 (rote Zahlen).

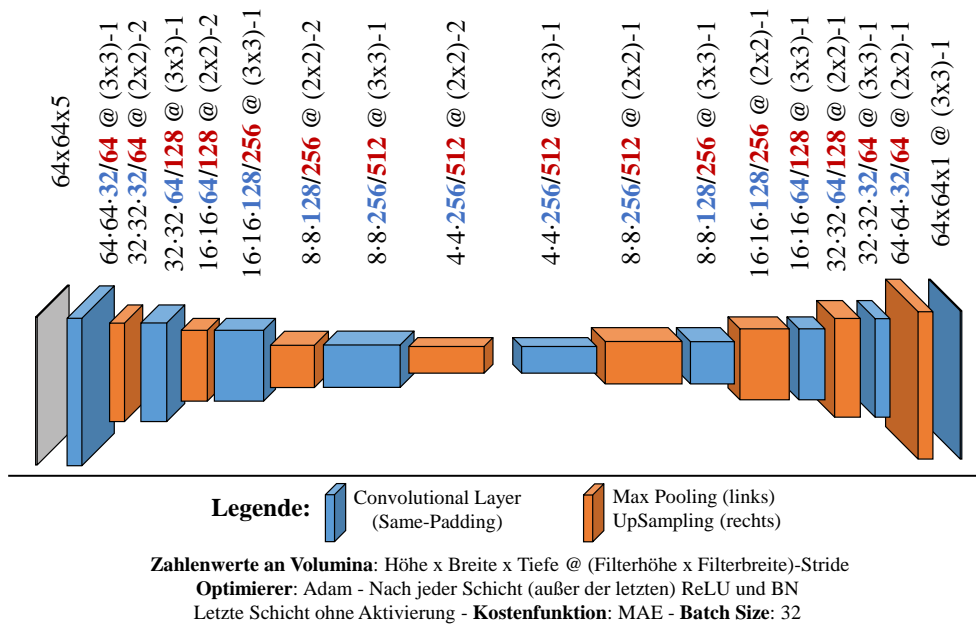


Abbildung 8.3: SimpleCNN Architektur.

8.3 Diskussion

Ergebnisse ModuleCNN

Tabelle 8.1 zeigt die ModuleCNN-Ergebnisse.

Name	Anzahl Parameter	MAE
ModuleCNN-32	2.351.233	$2,820 \cdot 10^{-2}$
ModuleCNN-NoDil-32	2.351.233	$2,642 \cdot 10^{-2}$
ModuleCNN-64	9.384.193	$2,138 \cdot 10^{-2}$
ModuleCNN-NoDil-64	9.384.193	$2,134 \cdot 10^{-2}$

Tabelle 8.1: Ergebnisse ModuleCNN Architekturen.

Zu sehen ist, dass jede einzelne ModuleCNN-Modellvariante alle vorherigen Ansätze schlägt. Die Modelle ohne Dilation können sogar eine minimal bessere Leistung als ihre Pendanten mit Dilation aufweisen, wobei der Unterschied so klein ist, dass es sich eher um Rauschen handelt. Eine Beispielvorhersage von ModuleCNN-NoDil-64 ist in Abb. 8.4 dargestellt. Es ist zu sehen, dass alle Symbole bis auf eines richtig erkannt wurden. In stichprobenartigen Tests zeigte sich, dass in den meisten Tests nur zwischen 0 und 3 Symbole falsch erkannt wurden - häufig auch nur teilweise falsch, wie beispielsweise ein verfälschtes inneres Symbol bei richtigem äußeren Symbol. Viele Tests können richtig gelöst werden.

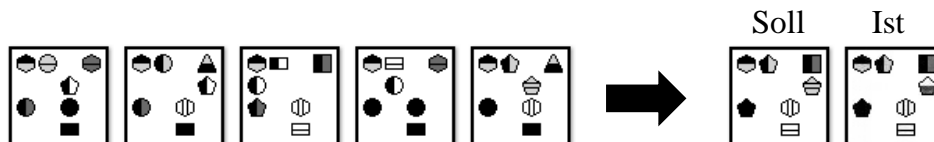


Abbildung 8.4: Beispiel einer Vorhersage von ModuleCNN-NoDil-64.

Im Folgenden soll durch Untersuchung der Feature Maps ermittelt werden, ob Bewegungs- und Symbolmodul tatsächlich die ihnen zugeschriebenen Aufgaben erlernt haben.

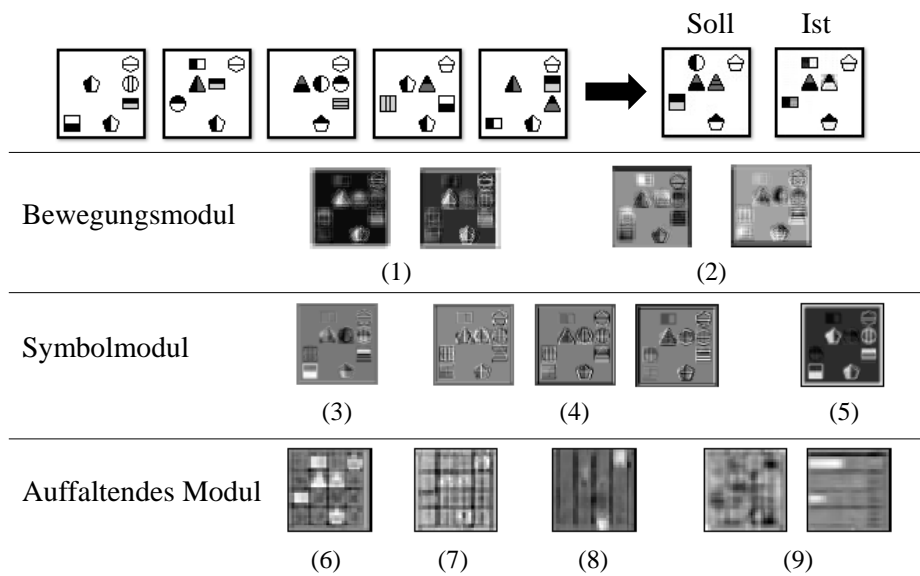


Abbildung 8.5: Beispiel einiger Feature Maps von ModuleCNN-64.

Abb. 8.5 zeigt, dass das Bewegungsmodul (oben) tatsächlich eher globale Eigenschaften erkennt, wie beispielsweise die unbeweglichen Symbole (1) oder Teile der Bilder, in denen Bewegungen stattfinden (2). Die Filter des Symbolmoduls (Mitte) erkennen einzelne Symbole (3), horizontale, vertikale und diagonale Linien (4) und extrahieren Einzelbilder der Gesamtsequenz (5). Es wurden außerdem die Feature Maps des letzten Convolutional Layers vor der finalen Vorhersage des auffaltenden Moduls untersucht (unten). Zu sehen ist beispielsweise, dass Filter vorhanden sind, welche die äußeren Symbole (6), Symbole mit einer Spitze (7) und Fünfecke (8) vorhersagen. Es existieren außerdem Feature Maps, welche nicht wirklich interpretierbar sind (9), möglicherweise enthalten diese weitere encodierte Informationen, insbesondere über die inneren Symbole.

SimpleCNN

In Tabelle 8.2 sind die SimpleCNN Ergebnisse aufgeführt.

Name	Anzahl Parameter	MAE
SimpleCNN-32	1.370.497	$3,171 \cdot 10^{-2}$
SimpleCNN-64	5.468.929	$2,390 \cdot 10^{-2}$

Tabelle 8.2: Ergebnisse SimpleCNN Architekturen.

Die Ergebnisse bewegen sich im Rahmen des Module-CNNs. Bemerkenswert ist, dass das SimpleCNN-32 als Modell mit der bisher niedrigsten Parameteranzahl das CNN-2xConvLSTM-512 schlägt, obwohl dieses mehr als 24-mal so viele Parameter besitzt. Mit einer Erhöhung der Parameter zum SimpleCNN-64 kann die Performanz nochmals verbessert werden, an die besten ModuleCNN Ergebnisse reicht es jedoch nicht ganz heran, möglicherweise auch einfach aufgrund der geringeren Parameteranzahl.

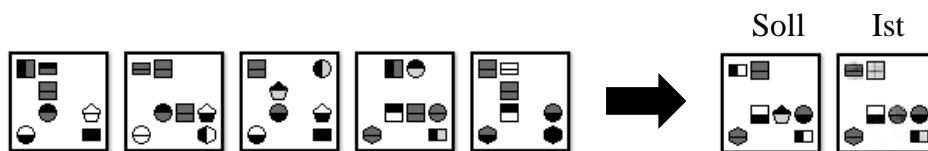


Abbildung 8.6: Beispiel einer Vorhersage von SimpleCNN-64.

Wie in Abb. 8.6 erkennbar ist, sagt auch das SimpleCNN-64 viele Symbole richtig vorher.

Um die Funktionsweise auch dieses Modells zumindest in Teilen nachvollziehen zu können, werden einige Feature Maps in Abb. 8.7 visualisiert.

Alle dargestellten Feature Maps - bis auf (2) - wurden aus dem ersten Convolutional Layer extrahiert. Bei (1) ist zu sehen, dass einzelne Filter sich auf die Extraktion eines einzelnen Diagrammes aus der Gesamtsequenz spezialisiert haben. (2) stammt aus dem vorletzten Convolutional Layer und zeigt an, wie die äußeren Symbole gebildet werden. Die gitterartige Struktur wird erkannt. In (3) ist die Erkennung feiner Linien sichtbar. (4) scheint für die Erkennung von unbeweglichen Symbolen verantwortlich zu sein. In (5) sind eher diffuse Aktivierungen im Bereich von Symbolen im rechten Bildbereich zu sehen, welche möglicherweise Bewegungen erfassen.

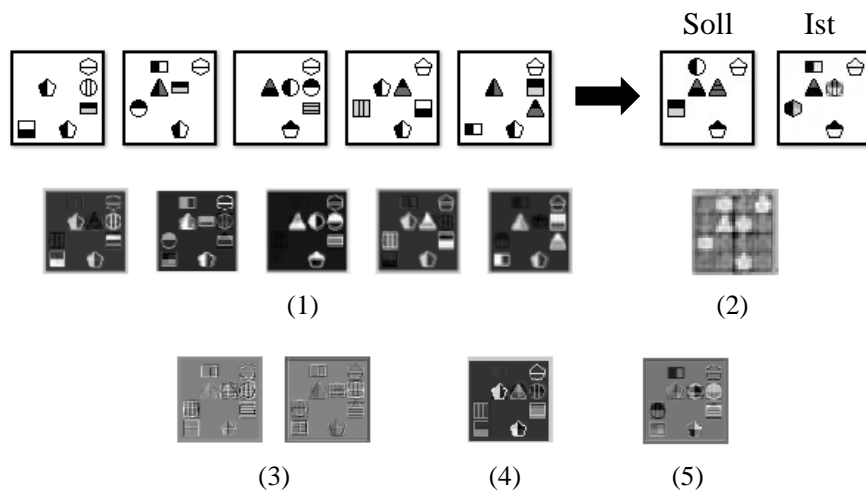


Abbildung 8.7: Beispiel einiger Feature Maps von SimpleCNN-64.

Schlussfolgerung

Die große Verbesserung im Vergleich zu den Ansätzen 1 und 2 ist möglicherweise auf die sehr direkte Vorgehensweise der FullyCNN-Modelle zurückzuführen. Die Diagramme können durch ihre Stapelung direkt bei Eingang in das Modell miteinander verglichen werden. Bei den rekurrenten Ansätzen werden „Kodierungen“ bzw. extrahierte Features erst nach dem Durchlaufen mehrerer Convolutional und Max Pooling Layer in der rekurrenten Einheit in Beziehung gesetzt. Dies stellt einen sehr indirekten Weg dar, bei welchem Informationen verloren gehen oder möglicherweise nur schwer erfasst werden können - insbesondere die sehr feinen Details, welche Bestandteile der Tests sind, wie beispielsweise sehr dünne Linien. Es wird angenommen, dass dieser „Umweg“ mit einer hohen Zahl an Parametern erkauft werden muss.

Außerdem wurde gezeigt, dass es möglich ist das Modell verschiedene Eigenschaften der Daten (Bewegungen und Symbole) in eigenen, expliziten Modulen lernen zu lassen. Notwendig und ausschlaggebend für den Modellerfolg ist dieses jedoch anscheinend nicht, da die Modelle ohne Dilated Convolution genau so gute Leistungen zeigen wie die Dilation-Varianten. Die Stapelung der Diagramme und das nicht komplexe, direkte Modell scheinen die wichtigeren Faktoren zu sein.

9 Vierter Ansatz: RelNet

Die untersuchten Diagrammsequenzen beruhen auf einer, insgesamt betrachtet, überschaubaren Menge an Mustern und Bewegungen. Mit den vorgestellten Modellen konnten die Tests z.T. zufriedenstellend gelöst werden, dies ist jedoch stets mit einer recht hohen Parameterzahl einhergegangen. Wird die Komplexität der Tests betrachtet, ist es denkbar, dass die Diagrammsequenzen auch mit weniger komplexen Modellen gelöst werden könnten. Dieser Gedanke soll in diesem Abschnitt verfolgt werden, indem die Diagrammsequenzen zunächst komprimiert und dann in - verglichen mit den vorherigen Ansätzen - schlankere Modelle gegeben werden.

Im Zuge dessen soll auch ein anderer Fokus bei der Modellbildung gesetzt werden: Bei der Verwendung von CNNs floss in das Modell die Annahme ein, dass die räumliche Anordnung der Daten eine wichtige Rolle für den Lernprozess darstellt. Das Gleiche gilt für Sequenzen unter Verwendung von RNNs. Das Konzept des *Relational Networks* (RelNet) soll den Fokus auf die Relationen innerhalb der Daten lenken. Es wurde von Santoro et al. (DeepMind) vorgestellt [50] und soll in diesem Kapitel Anwendung finden.

9.1 Relational Networks

In der Grundform wird ein RelNet wie folgt definiert [50]:

$$\text{RN}(O) = f_{\phi} \left(\sum_{i=1}^n \sum_{j=1}^n g_{\theta}(\vec{o}_i, \vec{o}_j) \right) \quad (9.1)$$

$O = \{\vec{o}_1, \vec{o}_2, \dots, \vec{o}_n\}$, $\vec{o}_i \in \mathbb{R}^m$ wird als eine Menge von Objekten bezeichnet, wobei jedes Objekt durch einen Vektor mit m Elementen beschrieben wird. Die Funktionen f_{ϕ} und g_{θ} werden als neuronale Netze (in dieser Grundvariante MLPs) mit den Mengen lernbarer Parameter ϕ bzw θ formuliert. g_{θ} nimmt die Konkatenation von je einem Paar von

Objekten \vec{o}_i und \vec{o}_j entgegen. Der Output von g_θ , ein Vektor, wird als Relation dieser zwei Objekte \vec{o}_i und \vec{o}_j bezeichnet. Die Aufgabe von g_θ soll es damit sein, festzustellen, ob eine Beziehung - und wenn ja, welche - vorliegt. f_ϕ erhält die (elementweise) Summe aller Relationen und die genaue Aufgabe von f_ϕ hängt vom gewählten Anwendungsfall ab. Der Output ist hier ebenfalls ein Vektor. Beide Funktionen werden Ende-zu-Ende trainiert, wodurch es sich bei den Relationsvektoren, die g_θ ausgibt, um interne Repräsentationen innerhalb des Netzes handelt.

Ein einfaches Beispiel ist in Abb. 9.1 gegeben. g_θ erhält wie erwähnt die Konkatenation von je zwei Objekten als Eingabe, zur besseren Übersicht sind nur 3 Kombinationen von Objekten dargestellt. Zu beachten ist, dass für alle Objektkombinationen das gleiche g_θ -MLP angewandt wird, d. h. bei allen Kombinationen werden die gleichen Gewichte verwendet.

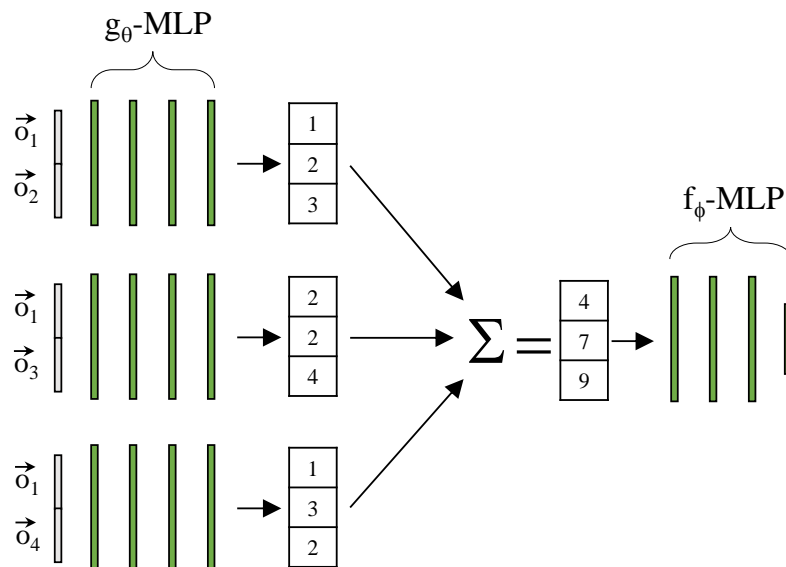


Abbildung 9.1: Skizze eines Relational Networks, adaptiert nach [50].

Die Verwendung von RelNets bietet sich aufgrund der zahlreichen Beziehungen an, welche in einer Diagrammsequenz auftreten: Relationen der äußeren Symbole, der inneren Symbole und der Bewegungen innerhalb einer Sequenz.

9.2 Übersicht RelNet Ansatz

Abb. 9.2 zeigt die Einzelschritte, welche bei beiden folgenden Ansätzen Anwendung finden sollen.

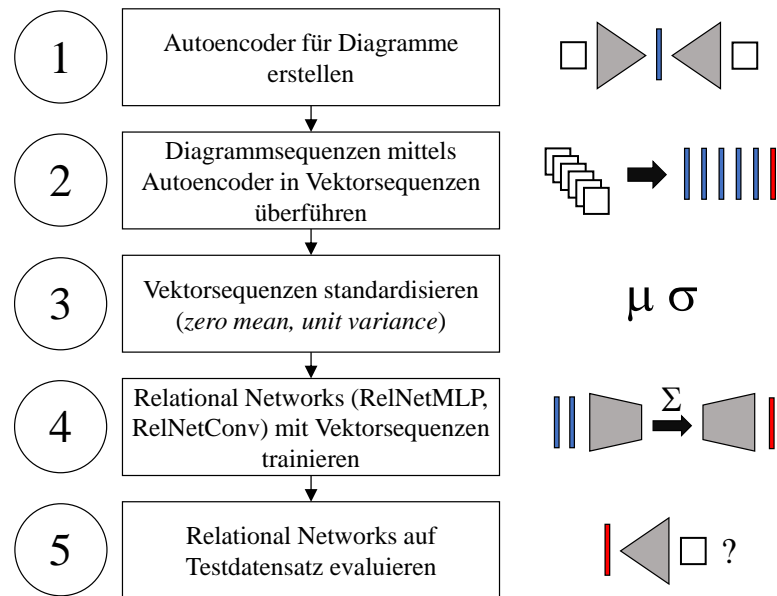


Abbildung 9.2: Flussdiagramm: Einzelschritte des RelNet-Ansatzes.

In Schritt 1 wird zunächst ein Autoencoder trainiert, welcher einzelne Diagramme in niedrigdimensionale Vektoren überführt. Unter Verwendung dieses Autoencoders können die Diagrammsequenzen dann in Schritt 2 in Vektorsequenzen überführt werden.

Es hat sich als gute Praxis erwiesen, die Eingangsdaten eines neuronalen Netzes zu standardisieren, d.h. dass Mittelwertfreiheit und eine Varianz von 1 (annähernd) erreicht werden (*zero mean, unit variance*). So wird u.a. vermieden, dass möglicherweise stark unterschiedliche Maßstäbe der Features das Training beeinträchtigen. Bewegt sich beispielsweise ein Element im fünfstelligen Bereich und ein anderes im einstelligen, ist ersichtlich, dass ohne Standardisierung die fünfstellige Dimension den Optimierungsprozess dominieren könnte - alleine aufgrund des großen Zahlenwertes, obwohl dieser möglicherweise gar keine tiefere Bedeutung besitzt. Aus diesem Grund werden im dritten Schritt die Vektoren elementweise standardisiert. Sei x ein Element des Vektors, μ der Mittelwert und σ die Varianz (jeweils ermittelt elementweise pro betrachtetem Datensatz: Training, Validierung oder Test), dann kann x' , der standardisierte Wert für x ,

ermittelt werden mit:

$$x' = \frac{x - \mu}{\sigma} \tag{9.2}$$

Mit den erhaltenen Vektorsequenzen werden im vierten Schritt die RelNets trainiert, welche in den nächsten beiden Kapiteln vorgestellt werden. Abschließend, in Schritt 5, wird die Performanz der RelNets auf dem Testdatensatz getestet, dazu wird der vorhergesagte Vektor „destandardisiert“, in den Decoder des Autoencoders gegeben und das entstandene Bild mit dem Soll-Diagramm verglichen. Dazu wird auch hier die MAE Kostenfunktion verwendet.

9.3 Autoencoder zur Komprimierung der Diagramme

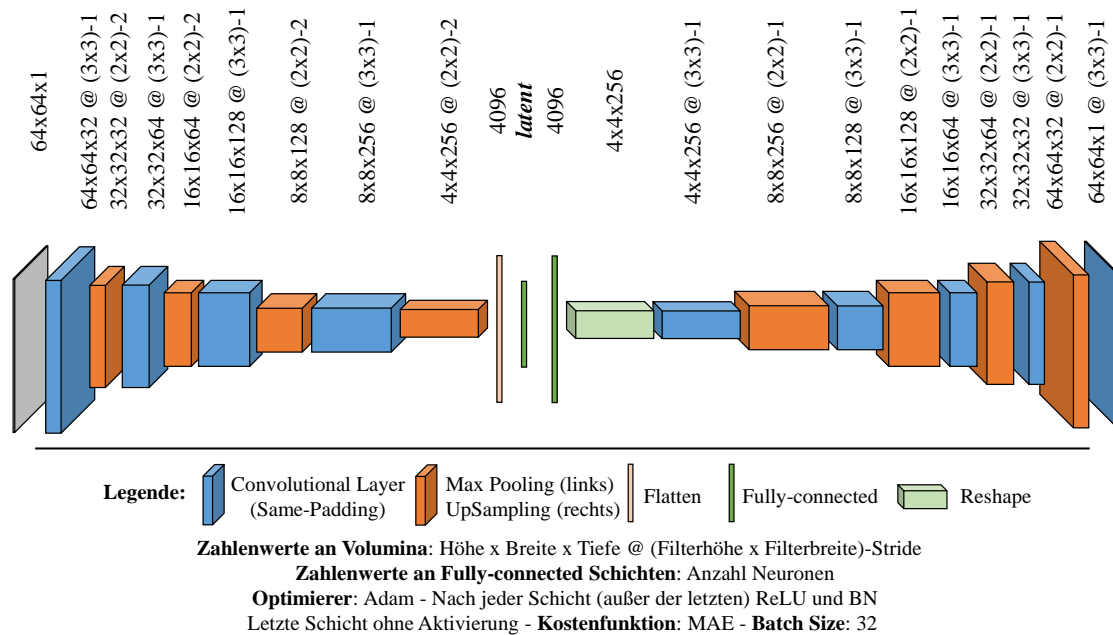


Abbildung 9.3: Autoencoder für die Komprimierung der Diagramme.

Wie in Abschnitt 9.2, Schritt 1 beschrieben, sollen unter Verwendung eines Autoencoders die einzelnen Diagramme in niedrigdimensionale Vektorrepräsentationen überführt werden, um in einem Relational Network verwendet werden zu können. Da den Tests ohnehin eine symbolische Grundstruktur zugrundeliegt, ist es eventuell ein aussichtsreicher Ansatz, von den Diagrammen in eine niedrigdimensionale Repräsentation überzugehen.

Diese Übersetzung einzelner Diagramme in niedrigdimensionale Vektorrepräsentationen soll von einem Autoencoder erlernt werden. Das verwendete Modell ist in Abb. 9.3 abgebildet, die Anzahl der Elemente des *Latent Vectors* ist *latent* = 64.



Abbildung 9.4: Beispiele für durch den Autoencoder rekonstruierte Diagramme.

Einige Beispiele für rekonstruierte Diagramme sind in Abb. 9.4 gegeben (links jeweils das Original, rechts die Rekonstruktion). Es ist zu sehen, dass die Bilder nahezu verlustfrei wiederhergestellt werden und auch dünne Linien der inneren Symbole erkennbar sind.

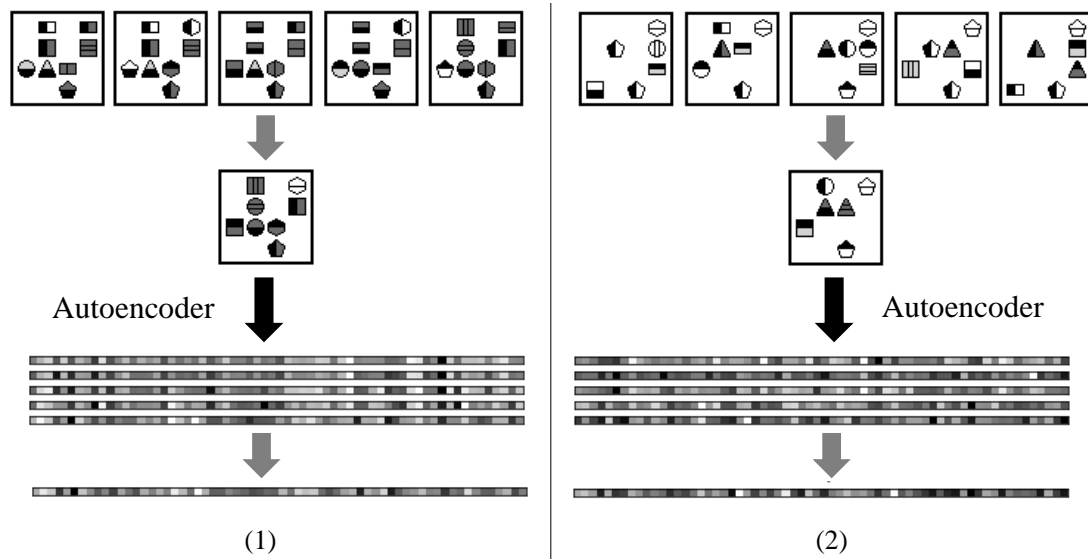


Abbildung 9.5: Beispiele für Vektorrepräsentationen.

Abb. 9.5 zeigt die Vektorrepräsentationen zweier Diagrammsequenzen. Bei (1) handelt es sich um eine Sequenz ohne Bewegungen. Es ist zu sehen, dass die Vektoren alle recht ähnliche Aktivierungen besitzen. Möglicherweise repräsentieren einzelne Elemente also explizit bestimmte Positionen. Unterstützt wird diese Annahme durch die Vektorrepräsentationen von (2), einer Sequenz, welche Bewegungen enthält. Die Aktivierungen unterscheiden sich hier deutlicher voneinander.

9.4 RelNetMLP

9.4.1 Modelle

Mithilfe des trainierten Autoencoders wird eine Diagrammsequenz in die Vektorsequenz $(\vec{d}_0, \vec{d}_1, \vec{d}_2, \vec{d}_3, \vec{d}_4, \vec{d}_5)$ überführt (vgl. Abschnitt 9.2, Schritt 2). Die Vektoren werden standardisiert (*zero mean, unit variance*, vgl. Abschnitt 9.2, Schritt 3) und die erstellten Modelle anschließend mit diesen trainiert (vgl. Abschnitt 9.2, Schritt 4). g_θ soll verwendet werden, um die Relation zwischen je zwei Vektorrepräsentationen zu ermitteln. Als Eingangswert für das MLP, welches g_θ modelliert, wird die Konkatenation \oplus der beiden betrachteten Vektoren und ihrer Indizes verwendet, um die Reihenfolgeinformation der Eingangsbilder zu erhalten. Der Input von g_θ für jeweils paarweise betrachtete Vektorrepräsentationen d_i und d_j ist damit definiert als:

$$\vec{x}_{i,j} = \vec{d}_i \oplus i' \oplus \vec{d}_j \oplus j' \quad (9.3)$$

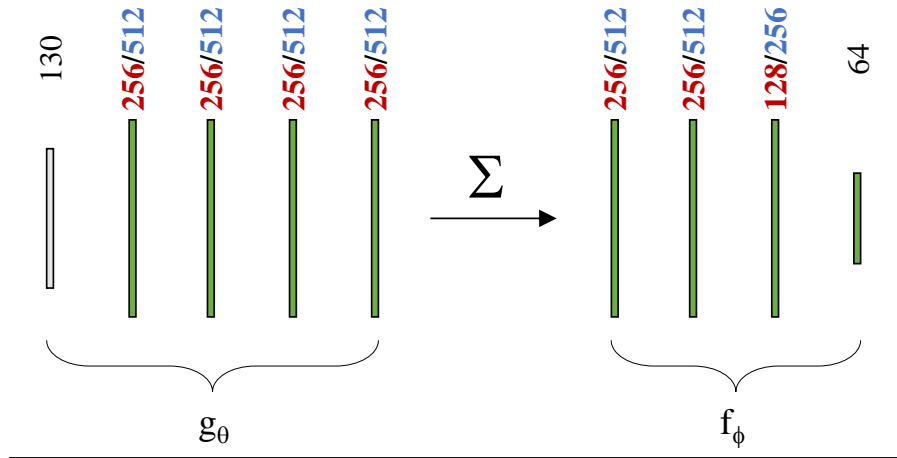
Mit $i' = \frac{i}{4}$ (j' analog). Die Länge des Inputvektors ist damit $64 + 1 + 64 + 1 = 130$.

f_ϕ soll dann unter Verwendung dieser Relationen die (standardisierte) Vektorrepräsentation des sechsten Diagrammes $\vec{y} \in \mathbb{R}^{64}$ vorhersagen:

$$\vec{y} = f_\phi \left(\sum_{i=0}^4 \sum_{j=i}^4 g_\theta(\vec{x}_{i,j}) \right) \quad (9.4)$$

f_ϕ wird ebenfalls als MLP modelliert und mit g_θ Ende-zu-Ende trainiert. Die verwendete Architektur ist in Abb. 9.6 dargestellt und soll in den Varianten RelNetMLP-256 (rote Zahlen) und RelNetMLP-512 (blauen Zahlen) erprobt werden. Die Vorhersage der sechsten Vektorrepräsentation wird wie die Vorhersage der Bilder auch als Regressionsproblem formuliert, die Outputschicht ist linear und es wird der MAE Loss verwendet.

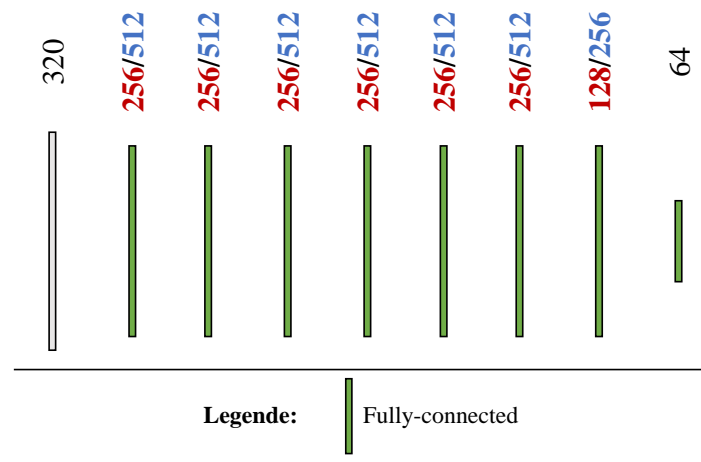
Zum Vergleich soll außerdem ein reines MLP Modell verwendet werden, welches die gesamte Vektorsequenz konkateniert als Input erhält (Inputgröße: $5 \cdot 64 = 320$) und daraus die sechste Vektorrepräsentation vorhersagt. Dieses Modell ist in Abb. 9.7 dargestellt und wird in den Varianten MLPOnly-256 (rote Zahlen) und MLPOnly-512 (blaue Zahlen) erprobt.



Legende:  Fully-connected

Zahlenwerte an Fully-connected Schichten: Anzahl Neuronen
Optimierer: Adam - Nach jeder Schicht (außer der letzten) ReLU und BN
 Letzte Schicht ohne Aktivierung - **Kostenfunktion:** MAE - **Batch Size:** 32

Abbildung 9.6: Skizze der Architektur von RelNetMLP.



Legende:  Fully-connected

Zahlenwerte an Fully-connected Schichten: Anzahl Neuronen
Optimierer: Adam - Nach jeder Schicht (außer der letzten) ReLU und BN
 Letzte Schicht ohne Aktivierung - **Kostenfunktion:** MAE - **Batch Size:** 32

Abbildung 9.7: Skizze der Architektur von MLPOnly.

9.4.2 Diskussion

Ergebnisse RelNetMLP

Tabelle 9.1 zeigt die Ergebnisse der RelNetMLP und MLPOnly Architekturen. Die vorhergesagten Vektoren werden durch den Decoder wieder in Diagramme überführt und mit den Soll-Vorhersagen verglichen (vgl. Abschnitt 9.2, Schritt 5).

Name	Anzahl Parameter	MAE
MLPOnly-256	458.944	$7,803 \cdot 10^{-2}$
MLPOnly-512	1.638.720	$7,578 \cdot 10^{-2}$
RelNetMLP-256	410,304	$12,057 \cdot 10^{-2}$
RelNetMLP-512	1.541.440	$11,758 \cdot 10^{-2}$

Tabelle 9.1: Ergebnisse RelNetMLP und MLPOnly Architekturen.

Der reine MLP Ansatz, MLPOnly-512, zeigt die beste Leistung und funktioniert deutlich besser als sein RelNet verwendendes Pendant, RelNetMLP-512. Eine Beispielvorhersage ist in Abb. 9.8 gegeben.

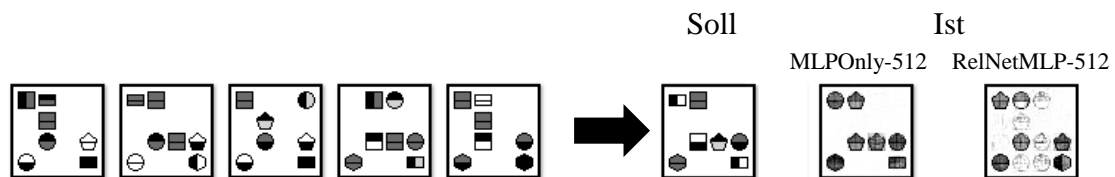


Abbildung 9.8: Vorhersagen MLPOnly und RelNetMLP.

Der Fehler von MLPOnly-512 bewegt sich im Bereich des ersten Ansatzes, jedoch bei deutlich niedrigerer Parameterzahl. Es konnte also gezeigt werden, dass trotz Komprimierung der Daten und Modelle die Vorhersagequalität vorheriger, umfangreicherer Ansätze erreicht werden kann. An das beste Modell (Ansatz 3) reicht die Performanz dieses Ansatzes jedoch nicht heran.

Auch die Art der Vorhersagen ähnelt denen der CNN-RNN Modelle: Die Modelle erkennen die Bewegungen der Symbole, haben jedoch Probleme bei der Generierung der einzelnen Symbole. Bei beiden Ansätzen, RelNet und CNN-RNN, werden die Diagramme durch einen Flaschenhals geführt, welcher nicht *Convolutional* ist. Es hat den Anschein, dass dieses im Allgemeinen zu Problemen der detaillierten Generierung der Symbole

führt. Da die einzelnen Linien eines Symbols Feinheiten der Diagramme darstellen, unterscheiden sich eventuell die Vektorrepräsentationen für zwei verschiedene Symbole auch nur sehr geringfügig. Möglicherweise ist nur der trainierte Autoencoder in der Lage diese feinen Unterschiede zu erkennen und aufzulösen - einem neuen Modell könnte dieses schwerfallen.

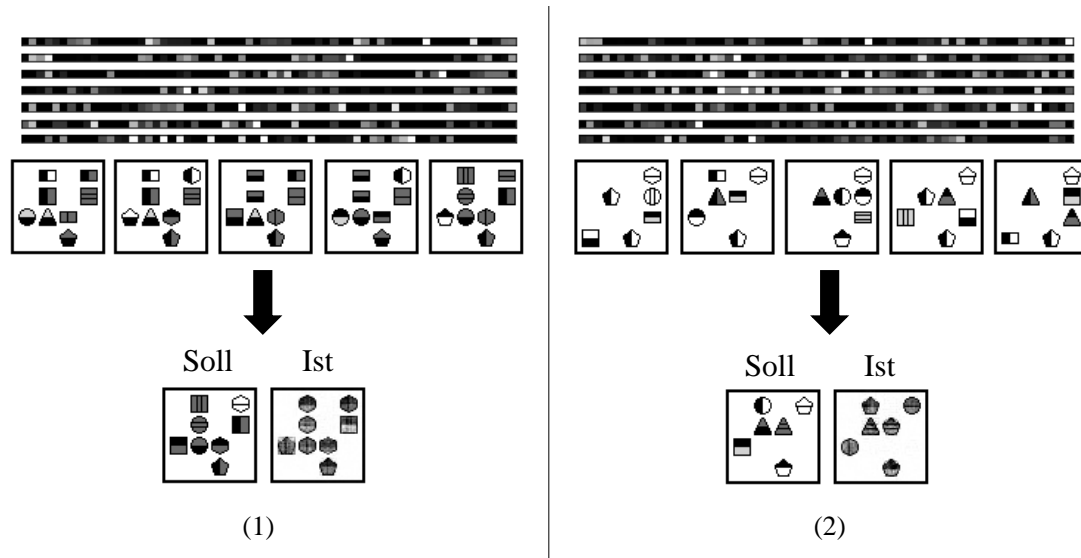


Abbildung 9.9: Aktivierungen der ersten Schicht von MLPOnly-512 für zwei Beispiele.

Abb. 9.9 zeigt die Aktivierungen der ersten Schicht des MLPOnly-512 Modelles für einen Test mit komplett unbeweglichen Symbolen (1) und einen mit beweglichen (2). Die erste Schicht besitzt 512 Neuronen, dargestellt in 7 Zeilen der Länge 64. Es ist zu sehen, dass die Aktivierungen ausgesprochen *sparse* ausfallen. Möglicherweise stellt die relativ große Anzahl an Neuronen (512) genug Platz zur Verfügung, um für jede Art von Position und Symbol recht explizite Positionen in der Aktivierung zu erzeugen.

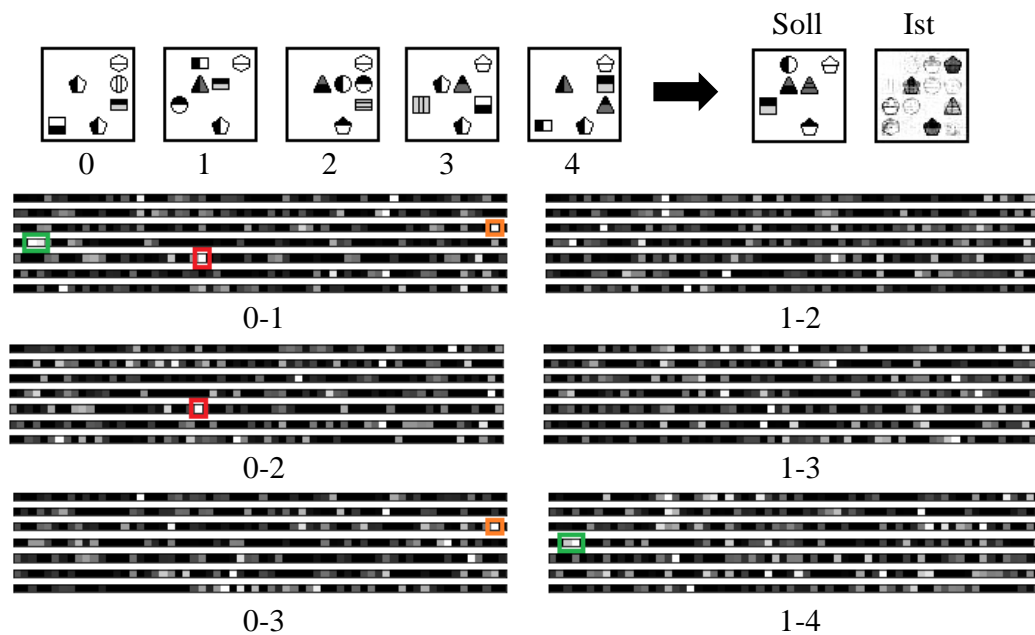


Abbildung 9.10: Aktivierungen der ersten Schicht von g_θ des RelNetMLP-512-Modells.

In Abb. 9.10 sind die Aktivierungen der ersten Schicht von g_θ des RelNetMLP-512-Modells für eine Diagrammsequenz aus dem Testdatensatz dargestellt. Wie beschrieben, nimmt g_θ die Diagramme jeweils paarweise entgegen, gezeigt sind die Aktivierungen einiger Beispielpaare. Auch hier sind die insgesamt 512 Neuronen in 7 Zeilen der Länge 64 aufgeführt. „0-1“ zeigt beispielsweise die Aktivierung bei Eingabe des ersten (Index 0) und zweiten (Index 1) Diagrammes. Es ist zu sehen, dass trotz unterschiedlicher Abstände zwischen den betrachteten Diagrammen gleiche Elemente aktiviert sind (siehe rote, grüne und orange Markierung). Dies ist möglicherweise ein Hinweis darauf, dass bestimmte Bereiche der Aktivierung verantwortlich für bestimmte Symbole und Muster sind, unabhängig davon, in welchem (und wievielm) Diagramm(paar) diese auftreten.

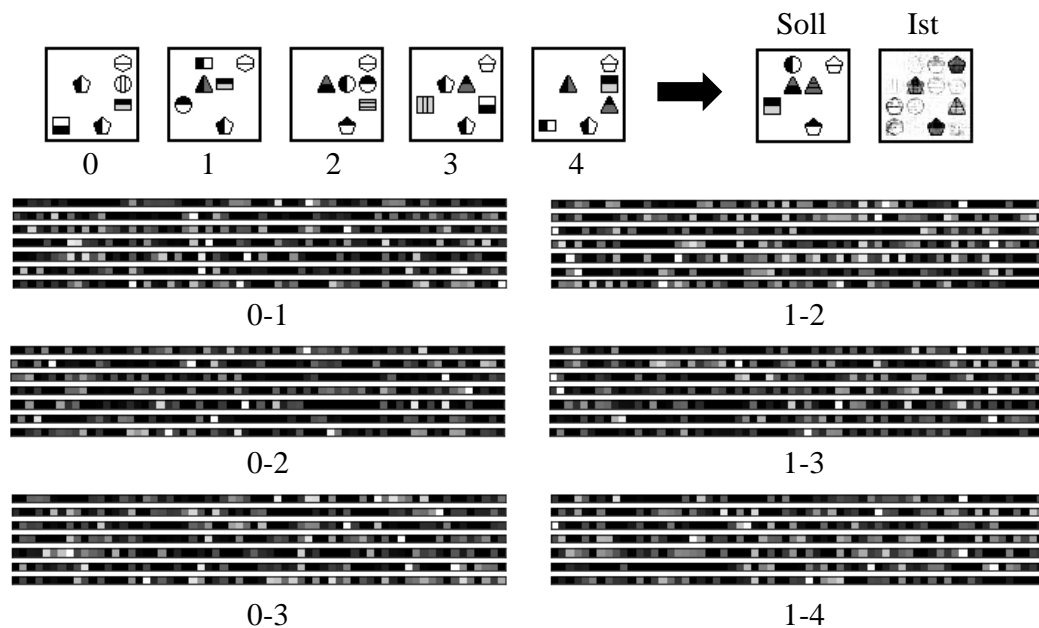


Abbildung 9.11: Ausgaben (ermittelte Relationen) von g_θ des RelNetMLP-512-Modells.

Abb. 9.11 zeigt außerdem die Ausgaben von g_θ für die gleichen ausgewählten Diagrammpaare wie in Abb. 9.10. Hierbei handelt es sich also um die ermittelten „Relationen“. Eine Interpretation fällt schwierig, da es sich um die vierte Fully-connected Schicht des Modells handelt und bis zu dieser Schicht bereits sehr viele Berechnungen stattgefunden haben, welche ihrerseits auf ebenfalls schwierig interpretierbaren Vektorrepräsentationen eines Autoencoders basieren. Im Allgemeinen fällt auch hier die hohe Sparsity der Aktivierungen auf.

Schlussfolgerung

Ein Problem könnte die grundsätzliche Formulierung eines RelNets sein. Durch die elementweise Aufsummierung der Ausgaben von g_θ könnten Informationen „verwischen“ bzw. ihre Herkunft eventuell nicht mehr nachvollzogen werden. Wenn beispielsweise ein Diagrammpaar in einem Bereich der Ausgabe von g_θ eine hohe Aktivierung auslöst und ein anderes eine niedrige und diese dann aufsummiert werden, so ist für f_ϕ nicht ersichtlich, woher die hohe Aktivierung stammen könnte. Die undeutlichen Positionen der RelNetMLP-512-Vorhersagen könnten so erklärt werden. Möglicherweise wäre eine Konkatination anstelle der Summe aussichtsreicher.

Die schlechte Performanz der relationalen Modelle ist möglicherweise auch auf die zu große „Zerstückelung“ des Problems zurückzuführen: Zunächst gehen die Diagramme durch einen Encoder, dann jeweils paarweise in ein RelNet, um dann durch den Decoder mit den Soll-Bildern verglichen zu werden. Dies bringt unter Umständen zu große Komplexität in die Modelle und es gibt zu viele Stellen, an denen Informationen verloren gehen können. Von den bisherigen Ansätzen konnte sich das konzeptionell einfachste mit den wenigsten Annahmen - FullyCNN - durchsetzen und dieser Trend wurde auch mit den RelNetMLP Ansätzen nicht gebrochen.

Die Annahme der zu großen „Zerstückelung“ wird dadurch unterstützt, dass Santoro et al. [50] Erfolge bei der Verwendung von RelNets mit Architekturen zeigen konnten, die Ende-zu-Ende trainiert wurden - vom Eingabebild in ein CNN direkt in ein RelNet. Dies könnte sich auch bei Diagrammsequenzen als nützlich erweisen, jedoch ginge dann die gewünschte Dimensionsreduzierung verloren. Außerdem sind die von Santoro et al. [50] betrachteten Probleme nicht generativer Art sondern stellen i. d. R. eine Form der Klassifikation dar, d. h. ab einem gewissen Punkt muss das CNN ohnehin in eine MLP Architektur überführt werden. Hier können MLP-RelNets möglicherweise eher ihre Stärke ausspielen als bei generativen Aufgaben, bei denen wieder in ein CNN übergegangen werden muss und damit die gleichen Probleme des Verlusts räumlicher Informationen entstehen könnten, wie sie schon in Kapitel 7 besprochen wurden.

Soll hingegen der Dimensionsreduzierungsansatz mit Autoencoder und RelNetMLP weiter verfolgt werden, könnte eine Verbesserung durch einen weniger komplexen Autoencoder unter Verwendung eines *Latent Spaces* mit mehr Dimensionen erzielt werden. Die Idee hierbei wäre, die „Intelligenz“ von dem En- und Decoder in das Encoding zu verschieben - wenn der Autoencoder weniger Parameter zum Erkennen der Feinheiten hat, könnten die Feinheiten klarer/gröber strukturiert in dem Encoding repräsentiert werden. Ergänzend könnte auch L1 Regularisierung innerhalb des *Latent Vectors* hilfreich sein, da diese *Sparsity* und damit zusammenhängend *Disentanglement* begünstigen [38][53] (vgl. Abschnitt 5.3, Erklärung *Sparsity* und *Disentanglement*), also ebenfalls Eigenschaften, welche zu aussagekräftigen Repräsentationen führen können.

9.5 RelNetConv

9.5.1 Modelle

In den folgenden Ansatz soll die Annahme fließen, dass die einzelnen Einträge der erlernten Vektorrepräsentationen jeweils eine bestimmte, interne Bedeutung für den Autoencoder besitzen (*Disentanglement*). Gleichzeitig soll diese Annahme durch die Ergebnisse rückblickend überprüft werden. Vereinfacht ausgedrückt, wäre es beispielsweise vorstellbar, dass der erste Eintrag eines Vektors immer das Symbol links oben beschreibt. In den Architekturen der RelNetMLPs wird auf diesen Umstand nicht weiter Wert gelegt, die Vektoren werden einfach konkateniert in das Modell gegeben. Im folgenden Modell, genannt *RelNetConv*, soll auf diese Annahme ein größerer Fokus gelegt werden, indem die Eigenschaften von CNNs ausgenutzt werden.

Die jeweils paarweise betrachteten Vektoren $\vec{d}_i \oplus i'$ und $\vec{d}_j \oplus j'$, die als Input von g_θ dienen, werden nicht konkateniert, sondern „übereinandergelegt“, womit ein Tensor der Dimension $1 \times 65 \times 2$ entsteht. Zur Modellierung von f_ϕ und g_θ werden dann ausschließlich Convolutional Layers mit einer Filterbreite und -höhe von 1 verwendet, womit explizit die Relation zwischen zwei zusammengehörigen Einträgen ermittelt werden soll. In Abb. 9.12 wird dieses Prinzip nochmals zum besseren Verständnis visualisiert.

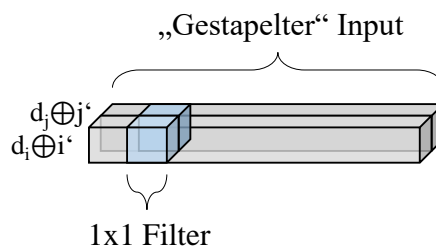


Abbildung 9.12: Prinzip der Inputanordnung und Verwendung der CNN Filter.

Möglicherweise werden so Filter gelernt, die für das Erkennen ganz bestimmter Arten von Relationen zwischen zwei betrachteten Einträgen verantwortlich sind - und zwar ganz unabhängig davon, an welchem Index sich diese befinden (*Shared Weights*). Außerdem bleibt die räumliche Anordnung der Vektoreinträge über das gesamte Modell gleich. Es ist vorstellbar, dass dies einen Vorteil bei der Vorhersage der sechsten Vektorrepräsentation darstellen könnte.

Die verwendete Architektur wird als *RelNetConv* bezeichnet und ist in Abb. 9.13 dargestellt.

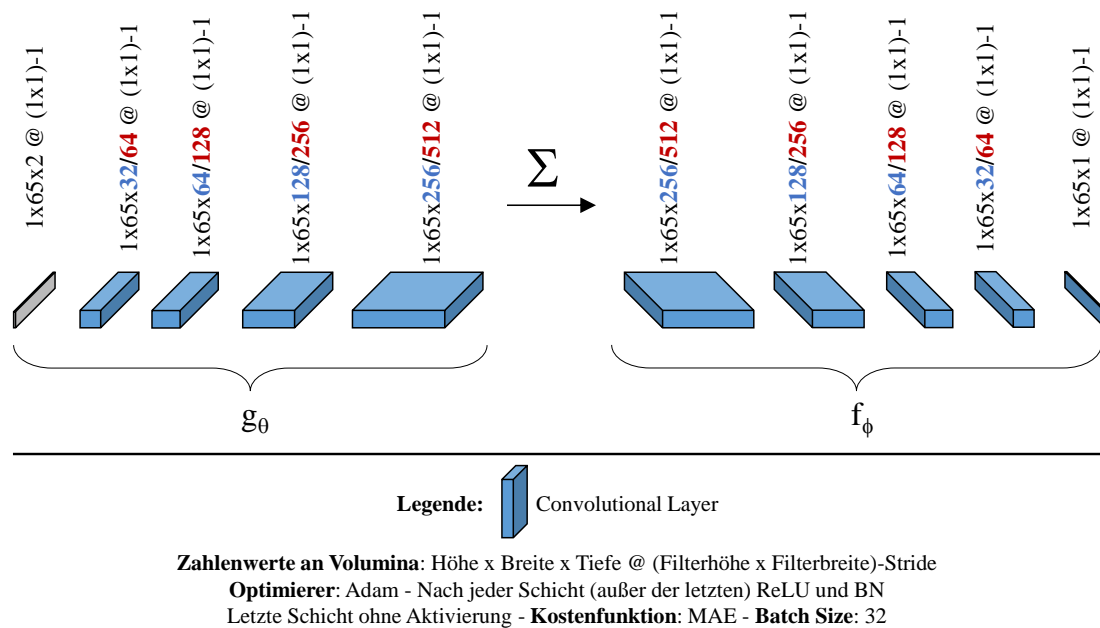
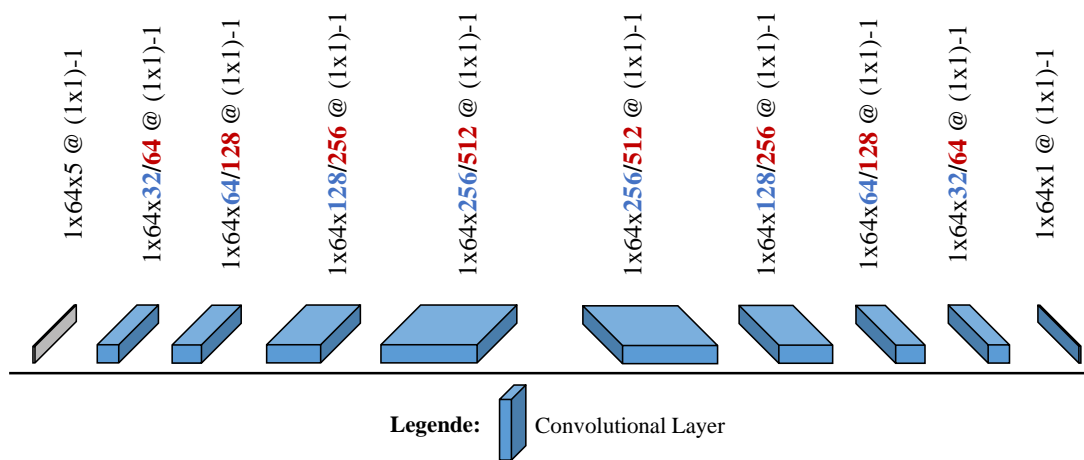


Abbildung 9.13: Skizze der Architektur von RelNetConv.

Sie wird in zwei Varianten erprobt, RelNetConv-32 (blaue Zahlen) und RelNetConv-64 (rote Zahlen), zu beachten ist der Input von $1 \times 65 \times 2$. Es werden die gleichen Kombinationen wie im vorherigen Ansatz verwendet. In jeder Schicht kommen 1×1 -Filter zum Einsatz. Hier werden ebenfalls werden die Ergebnisse von g_θ elementweise aufsummiert und f_ϕ übergeben. Die Architektur ist fully-convolutional und wird Ende-zu-Ende trainiert.

Auch hier soll zum Vergleich ein Ansatz ohne RelNet dienen. Das verwendete ConvOnly-32 (blaue Zahlen) bzw. ConvOnly-64 (rote Zahlen) ist in Abb. 9.14 abgebildet. Auf die Konkatination des Index kann verzichtet werden, woraus sich eine Inputdimension von $1 \times 64 \times 5$ ergibt.



Zahlenwerte an Volumina: Höhe x Breite x Tiefe @ (Filterhöhe x Filterbreite)-Stride

Optimierer: Adam - Nach jeder Schicht (außer der letzten) ReLU und BN

Letzte Schicht ohne Aktivierung - **Kostenfunktion:** MAE - **Batch Size:** 32

Abbildung 9.14: Skizze der Architektur von ConvOnly.

9.5.2 Diskussion

In Tabelle 9.2 sind die RelNetConv Ergebnisse aufgelistet. Auch hier werden die vorhergesagten Vektoren unter Verwendung des Decoders in Diagramme überführt und mit den Soll-Diagrammen verglichen.

Name	Anzahl Parameter	MAE
ConvOnly-32	154.369	$8,610 \cdot 10^{-2}$
ConvOnly-64	607.745	$8,600 \cdot 10^{-2}$
RelNetConv-32	154.273	$13,127 \cdot 10^{-2}$
RelNetConv-64	607.553	$12,764 \cdot 10^{-2}$

Tabelle 9.2: Ergebnisse RelNetConv und ConvOnly Architekturen.

Die Ergebnisse sind insgesamt schlechter als die der RelNetMLP Modelle. Auch hier ist zu sehen, dass die Nicht-RelNet-Modelle (ConvOnly) eine bessere Performanz zeigen als die RelNet-Varianten. Möglicherweise spielt hier ebenfalls die zu große „Zerstückelung“ des Problems eine Rolle.

Dass die Ergebnisse der ConvOnly Varianten eine schlechtere Leistung als die MLPOnly zeigen, ist möglicherweise ein Hinweis darauf, dass die Annahme der „jeweils bestimmten, internen Bedeutung“ pro Eintrag des Vektors zumindest teilweise falsch war. Durch die Verwendung von 1x1 Filtern werden durch das gesamte Modell hinweg stets nur die Einträge am gleichen Index verglichen. Unter Umständen bestehen jedoch eine Reihe von Querbeziehungen, welche durch die Fully-connected RelNetMLP/MLPOnly Architekturen besser erfasst werden.

Im Folgenden soll auch für die RelNetConv- und ConvOnly-Modelle eine Untersuchung ihrer Funktionsweise stattfinden.

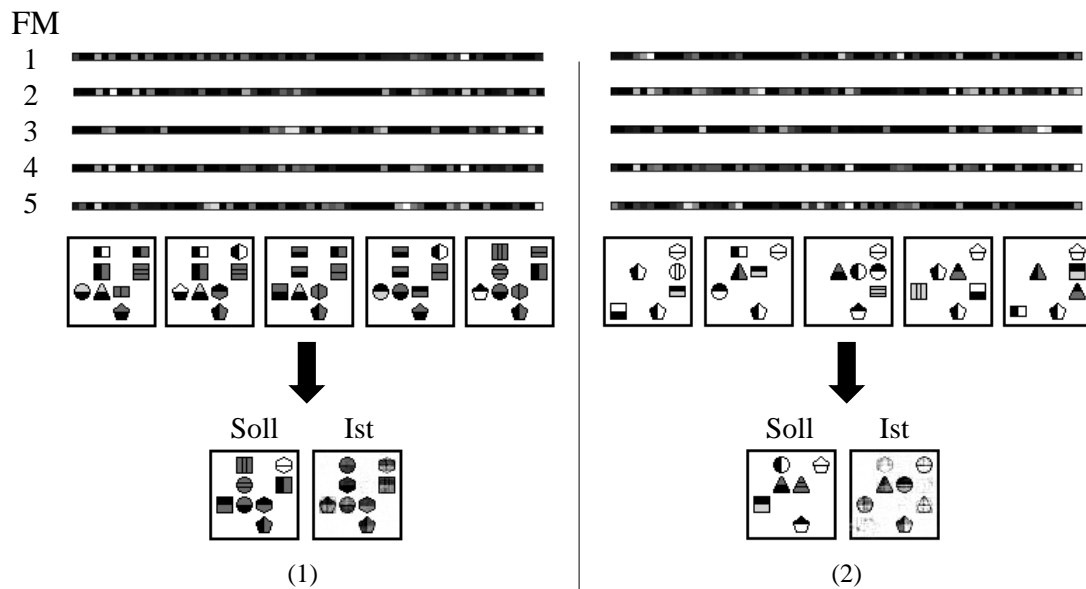


Abbildung 9.15: Feature Maps der ersten Convolutional Schicht des ConvOnly-64.

In Abb. 9.15 sind stichprobenartig fünf der insgesamt 64 Feature Maps der ersten Convolutional Schicht des ConvOnly-64 Modells abgebildet, zusammen mit der jeweiligen Vorhersage. Es handelt sich um jeweils 64 Elemente lange Vektoren. Es ist zu sehen, dass die Aktivierungen z. T. sehr *sparse* sind und bei den betrachteten Beispielen sehr verschieden. Dies könnte ein Hinweis darauf sein, dass die einzelnen Filter tatsächlich auf bestimmte erlernte Relationen zwischen zwei Einträgen reagieren, unabhängig davon, an welchem Index sie auftreten.

Darüber hinaus wirkt die Vorhersage bei der unbeweglichen Diagrammsequenz (1) verhältnismäßig gut, dem Modell scheinen diese Tests deutlich leichter zu fallen. Möglicherweise ist dies zurückzuführen auf die sehr ähnlichen Vektorrepräsentationen von Diagrammen mit Symbolen an gleichen Positionen (vgl. Abschnitt 9.3). Bei diesen treten Aktivierungen an ähnlichen Stellen auf, wodurch 1x1-Filter besonders effektiv sein könnten.

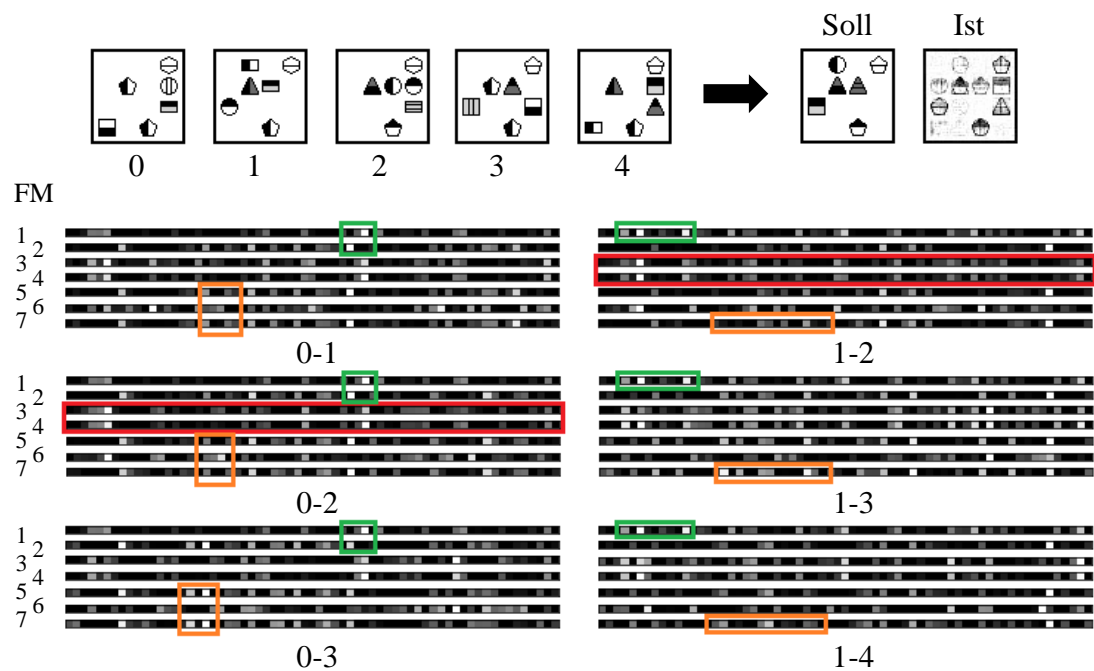


Abbildung 9.16: Feature Maps der ersten Convolutional Schicht von g_θ des RelNetConv-64.

Analog zu Abb. 9.10 zeigt Abb. 9.16 die ersten 7 Feature Maps der ersten Convolutional Schicht von g_θ des Modells RelNetConv-64 für verschiedene Diagrammpaare. Es ist zu sehen, dass für alle Paare mit Diagramm 0 (linke Spalte) einige Filter immer die gleichen Aktivierungen hervorrufen (grüne Markierungen). Das Gleiche ist in der rechten Spalte mit Diagramm 1 zu beobachten (ebenfalls grün markiert). Ebenso gibt es Filter, welche mit variiertem zweitem Diagramm jeweils andere Aktivierungen liefern (orange umrandet). Anscheinend reagieren also einige Filter explizit auf die erste Vektorrepräsentation, andere explizit auf die zweite. Außerdem sieht es danach aus, dass einige Filter doppelt bzw. sehr ähnlich gelernt (rot umrandet) werden, ein Verhalten, welches relativ typisch für CNNs ist.

Schlussfolgerung

Die Ergebnisse deuten darauf hin, dass die Repräsentationen nicht (vollständig) *disentangled* sind. Die bereits besprochenen Verbesserungsansätze für den Autoencoder gelten auch für diesen Teil, ebenso das besprochene mögliche Additionsproblem des RelNet Ansatzes im Allgemeinen (vgl. Abschnitt 9.4.2). Es könnten breitere Filter erprobt werden, um möglicherweise mehr Querverbindungen zu erfassen. Auch gänzlich andere Ansätze

aus dem Bereich der Autoencoder bieten sich an, wie beispielsweise der β -Variational Autoencoder [22], welcher gezielt für aussagekräftige Encodings und Disentanglement konstruiert wurde.

10 Fazit

10.1 Zusammenfassung

Im Rahmen dieser Arbeit wurden die Fähigkeiten verschiedenartiger neuronaler Netze untersucht und bewertet, sogenannte Diagrammsequenzen zu lösen - ein Problem welches die Bereiche Bildverarbeitung und Generierung von Bildern, Sequenzverarbeitung, Erkennung von Relationen in Daten und Dimensionsreduzierung vereint. Der Begriff „Diagrammsequenz“ wurde im Rahmen dieser Arbeit für eine Sequenz von insgesamt 6 Bildern („Diagrammen“) definiert, welche bestimmte Symbole (kombinierte geometrische Formen) beinhalten. Diese verändern, bestimmten Mustern unterliegend, ihr Aussehen und/oder ihre Position über die Sequenz hinweg. Die neuronalen Netze bekommen die ersten fünf Bilder als Eingabe und sollen mithilfe dieser das sechste Bild generieren. Ein Datensatz für Training, Validierung und abschließendem Test mit insgesamt 30.000 Diagrammsequenzen wurde eigens für diese Arbeit generiert. Es wurden insgesamt vier verschiedene Ansätze erprobt.

Im **ersten Ansatz** wurden Kombinationen aus CNNs und LSTMs/GRUs erprobt. Diese waren zwar in der Lage die richtigen Positionen der Formen des sechsten Diagrammes zu erzeugen, die einzelnen erzeugten Symbole waren jedoch von verschwommener Gestalt.

In der Annahme, dass durch eine Umformung innerhalb der CNN-RNN Modelle wichtige räumliche Informationen verloren gehen, wurde das bestehende Modell im **zweiten Ansatz** um das Konzept der Convolutional LSTMs erweitert, unter dessen Verwendung diese Umformung nicht nötig ist. Die Ergebnisse konnten signifikant verbessert werden und die Symbole waren deutlich schärfer. Einher ging diese Verbesserung jedoch mit einer drastischen Erhöhung der Parameter.

Der darauf folgende **dritte Ansatz** basiert auf der Beobachtung, dass die Positionen gut erkannt werden, es jedoch weiterhin Probleme mit der Generierung der detaillierten Zielsymbole gibt. Aus diesem Grund sollen die Bewegungs- und Symbolerkennung in

jeweils eigene Module ausgelagert werden („ModuleCNN“), wozu Dilated Convolution verwendet wurde. Außerdem entfiel bei diesem Ansatz die rekurrente Einheit und die Form der Eingabedaten wurde entsprechend umgeformt („FullyCNN“). Dieser Ansatz konnte alle vorhergehenden Ansätze übertrumpfen, obwohl die Parameterzahl deutlich niedriger als die der Convolutional LSTM Modelle war. Viele Tests konnten richtig gelöst werden. Es konnte ebenfalls gezeigt werden, dass die einzelnen Module tatsächlich die ihnen zugewiesenen Aufgaben erfüllten. Die besonders gute Performanz dieses Ansatzes wurde jedoch nicht auf diese Aufgabenteilung zurückgeführt, sondern auf die sehr direkte Modellierung des Problems ohne viele Annahmen (kein RNN für Sequenzverarbeitung) und den direkten Vergleich aller Diagramme beim Eingang in das Modell.

Im finalen **vierten Ansatz** sollte zum einen Wert darauf gelegt werden, dass die Komplexität der Tests insgesamt überschaubar ist und deswegen eine Komprimierung sowohl der Daten als auch des Modells möglich sein sollte. Zum anderen sollten sogenannte Relational Networks (RelNets) erprobt werden, welche mit dem Vorhaben konstruiert wurden, Beziehungen innerhalb von Daten besonders gut zu erkennen. Die Diagrammsequenzen wurden unter Verwendung eines Autoencoders zu niedrigdimensionaleren Vektorsequenzen komprimiert. Es wurden zwei RelNet Ansätze erprobt, der erste verwendet ein MLP, der zweite ein CNN. Jeweils beide Ansätze zeigten eine schlechtere Leistung als ebenfalls erstellte Vergleichsmodelle, die ohne RelNet, jedoch ebenfalls den komprimierten Eingangsdaten arbeiten. Das MLPOnly-Vergleichsmodell konnte tatsächlich eine Leistung auf dem Niveau des ersten Ansatzes erzielen, bei deutlich geringerer Parameteranzahl, grundsätzlich ist dieser Ansatz also aussichtsreich. Durch Untersuchungen der Feature Maps und Aktivierungen der Modelle konnten darüber hinaus Rückschlüsse über die Art der Komprimierung (verbesserungswürdiges *Disentanglement*) und die Funktionsweise der RelNets gezogen werden.

Ein Vergleich der jeweils besten Modelle über die Ansätze hinweg ist in Tabelle 10.1 gegeben. Weitere Beispielvorschläge befinden sich im Anhang unter A.4.

Ansatz	Name	Anzahl Parameter	MAE
CNN-RNN	CNN-GRU-512-128	4.979.585	$7,080 \cdot 10^{-2}$
CNN-ConvLSTM	CNN-ConvLSTM-2x512	34.993.409	$3,181 \cdot 10^{-2}$
FullyCNN	ModuleCNN-NoDil-64	9.384.193	$2,134 \cdot 10^{-2}$
RelNet	MLPOnly-512	1.541.440	$7,578 \cdot 10^{-2}$

Tabelle 10.1: Ergebnisse der besten Modelle innerhalb der vier Ansätze.

Gemessen an der Zielsetzung (vgl. Abschnitt 1.2) wurden Modelle gefunden, welche in der Lage waren viele Diagrammsequenzen zu lösen. Durch Visualisierungen von Aktivierungen innerhalb der Modelle für bestimmte Eingaben konnten die Arbeitsweisen der Modelle in Teilen nachvollzogen sowie ihre Performanz begründeter beurteilt und verglichen werden. Dabei wurde ein Überblick über geläufige und eher exotische Verfahren des maschinellen Lernens gegeben, und wie diese kombiniert werden können, um ein Problem, welches viele Domänen vereint, zu lösen.

10.2 Ausblick

Im Folgenden soll ein Ausblick auf mögliche Erweiterungen dieser Arbeit gegeben werden.

Hyperparameter

In dieser Arbeit wurden viele Hyperparameter festgehalten, um die Anzahl der Experimente nicht zu groß werden zu lassen. Dementsprechend gibt es natürlich noch viele Möglichkeiten, diese zu variieren. Die Unit Size der erprobten RNNs (LSTM, GRU) könnte noch weiter modifiziert werden, ebenso die Anzahl der Elemente des eingehenden Vektors. Insbesondere bei den CNNs gibt es viele Optionen, welche noch erprobt werden könnten (Filtergröße, Filteranzahl). Darüber hinaus könnten verschiedene Optimierer oder andere Aktivierungsfunktionen erprobt werden.

Statt eines Trial-and-Error Ansatzes bei den CNNs wäre auch die Verwendung von *Residual Networks* [21] (ResNets) eine Alternative. Hierbei handelt es sich um sehr tiefe CNNs, welche - vereinfacht ausgedrückt - Abkürzungen zwischen den Schichten ermöglichen. Welcher Weg durch das Netz gewählt wird, ist Teil des Lernprozesses, während dem die optimale Anzahl an durchlaufenen Schichten im Prinzip selbst erlernt wird. ResNets konnten große Erfolge aufweisen und eine Reihe an Wettbewerben im Bereich der Bildverarbeitung gewinnen [21].

Explainable AI

Durch die Visualisierung von Feature Maps und Aktivierungen innerhalb der Modelle konnten die Funktionsweisen dieser grundlegend nachvollzogen werden. Nichtsdestotrotz könnte in dieses Gebiet noch tiefer eingestiegen werden.

In die bestehenden Modelle könnten spezielle, konstruierte, Diagrammsequenzen gegeben werden, welche ausschließlich ein einzelnes bestimmtes Muster enthalten. Damit könnten möglicherweise - über Betrachtung der Aktivierungen - Teile des Modells ausgemacht werden, die explizit für ebendiese Muster zuständig sind.

Eine weitere Idee wäre die Verwendung eines Attention-Mechanismus. Diese haben große Erfolge im Bereich des maschinellen Lernens bringen können [5]. Es gibt eine Reihe verschiedener Ansätze in diesem Bereich [57][33], häufig besteht die Aufgabe solch eines Mechanismus jedoch darin, das neuronale Netz für die gewählte Aufgabe explizit auf bestimmte Teile des Inputs „achten“ zu lassen. I. d. R. ist dieser Mechanismus absichtlich so gestaltet, dass er für den Menschen recht gut interpretierbar ist.

Weitere Modellansätze

Es gibt eine Reihe weiterführender Modellansätze, welche für die Lösung der Diagrammsequenzen erprobt werden könnten.

Da es sich um eine generierende Aufgabe handelt, könnten sich *Generative Adversarial Networks* [17] (GANs) als nützlich erweisen, ein Ansatz des unüberwachten Lernens von Ian Goodfellow. Hierbei handelt es sich um generierende Modelle, welche aus einem Paar von Netzen bestehen: Dem Generator, welcher Bilder erzeugt und dem Diskriminator, welcher „Fälschungen“ des Generators und echte Beispiele aus einem Datensatz unterscheiden soll. Dieser Ansatz hat sich als sehr erfolgreich erwiesen bei der Generierung von Bildern, stellt die Grundlage breiter Forschung in diesem Bereich dar [27][28][26] und könnte möglicherweise auch bei der Lösung von Diagrammsequenzen Erfolg zeigen.

Wie bereits in den Kapiteln 9.4.2 und 9.5.2 angeschnitten, könnte der RelNet Ansatz durch „bessere“ Vektorrepräsentationen größere Erfolge zeigen. Damit sind insbesondere Repräsentationen gemeint, welche hohe *Sparsity* und *Disentanglement* aufweisen. *Variational Autoencoder* (VAE) [30], β -VAE [22] und der halbüberwacht lernende *Disentangled VAE* [36] konnten Erfolge beim Erreichen dieser Ziele zeigen und könnten ebenfalls im Kontext der RelNets eine Verbesserung bringen.

Diagrammsequenzen

Die aktuell verwendeten Diagramme bestehen aus einem 4·4 Raster und die Sequenz besitzt eine Länge von 6. Sie könnten auf vielfältige Art und Weise erweitert werden. Die möglichen Positionen könnten beispielsweise stärker variiert werden. Auch die Anzahl der

Symbole könnte erhöht werden, genau wie die Anzahl der möglichen Sequenz- und Bewegungsmuster. Anstelle von Graubildern könnten Farben verwendet werden. Interessant wäre ebenfalls die Verwendung verschieden langer Sequenzen, um zu sehen, ob die Verwendung von RNNs hier einen Vorteil bringen könnte. Darüber hinaus könnte der Ansatz verfolgt werden, die Vorhersagen zusätzlich wie eine Klassifikation zu bewerten („Wie viele Symbole wurden richtig erkannt“), wodurch die Testergebniswerte wahrscheinlich verständlicher wären als nur unter Beurteilung mithilfe eines MAE-Werts.

Literaturverzeichnis

- [1] *Duden Online - Intelligenz.* <https://www.duden.de/rechtschreibung/Intelligenz>. – Abrufdatum: 15.10.2019
- [2] *MNIST-Datensatz.* <http://yann.lecun.com/exdb/mnist/>. – Abrufdatum: 15.11.2019
- [3] AGRAWAL, Aishwarya ; LU, Jiasen ; ANTOL, Stanislaw ; MITCHELL, Margaret ; ZITNICK, C. L. ; BATRA, Dhruv ; PARIKH, Devi: *VQA: Visual Question Answering.* 2015
- [4] AMELANG, M.: *Differentielle Psychologie und Persönlichkeitsforschung.* Kohlhammer, 2006 (Kohlhammer Standards Psychologie). – URL https://books.google.de/books?id=j_22jqy6xJEC. – ISBN 9783170186408
- [5] BAHDANAU, Dzmitry ; CHO, Kyunghyun ; BENGIO, Yoshua: *Neural Machine Translation by Jointly Learning to Align and Translate.* 2014
- [6] BIBEL, Wolfgang ; ERTEL, Wolfgang ; KRUSE, Rudolf: *Grundkurs Künstliche Intelligenz : Eine praxisorientierte Einführung.* 2013. – URL http://www.worldcat.org/search?qt=worldcat_org_all&q=9783834821577
- [7] CHO, Kyunghyun ; MERRIENBOER, Bart van ; GULCEHRE, Caglar ; BAHDANAU, Dzmitry ; BOUGARES, Fethi ; SCHWENK, Holger ; BENGIO, Yoshua: *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation.* 2014
- [8] CHOI, Hyomin ; BAJIC, Ivan V.: *Deep Frame Prediction for Video Coding.* 2018
- [9] CHUNG, Junyoung ; GULCEHRE, Caglar ; CHO, KyungHyun ; BENGIO, Yoshua: *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling.* 2014

- [10] CIRESAN, Dan C. ; MEIER, Ueli ; MASCI, Jonathan ; SCHMIDHUBER, Jürgen: Multi-column deep neural network for traffic sign classification. In: *Neural Networks* 32 (2012), S. 333–338. – URL <http://dblp.uni-trier.de/db/journals/nn/nn32.html#CiresanMMS12>
- [11] COMSENSE: *10 Key Marketing Trends for 2017*. URL: http://comsense.consulting/wp-content/uploads/2017/03/10_Key_Marketing_Trends_for_2017_and_Ideas_for_Exceeding_Customer_Expectations.pdf. – Abrufdatum: 15.11.2019
- [12] COUGHLIN, Tom ; FORBES.COM (Hrsg.): *175 Zettabytes By 2025*. URL: <https://www.forbes.com/sites/tomcoughlin/2018/11/27/175-zettabytes-by-2025/>. November 2017. – Abrufdatum: 15.11.2019
- [13] DUMOULIN, Vincent ; VISIN, Francesco: *A guide to convolution arithmetic for deep learning*. 2016
- [14] FU, Rui ; ZHANG, Zuo ; LI, Li: Using LSTM and GRU neural network methods for traffic flow prediction, 11 2016, S. 324–328
- [15] GLOROT, Xavier ; BENGIO, Yoshua: Understanding the difficulty of training deep feedforward neural networks. In: *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS10)*. Society for Artificial Intelligence and Statistics, 2010
- [16] GLOROT, Xavier ; BORDES, Antoine ; BENGIO, Yoshua: Deep Sparse Rectifier Neural Networks. In: GORDON, Geoffrey (Hrsg.) ; DUNSON, David (Hrsg.) ; DUDÍK, Miroslav (Hrsg.): *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics* Bd. 15. Fort Lauderdale, FL, USA : PMLR, 11–13 Apr 2011, S. 315–323. – URL <http://proceedings.mlr.press/v15/glorot11a.html>
- [17] GOODFELLOW, Ian J. ; POUGET-ABADIE, Jean ; MIRZA, Mehdi ; XU, Bing ; WARDE-FARLEY, David ; OZAI, Sherjil ; COURVILLE, Aaron ; BENGIO, Yoshua: *Generative Adversarial Networks*. 2014
- [18] GRAVES, Alex: *Generating Sequences With Recurrent Neural Networks*. 2013
- [19] GREFF, Klaus ; SRIVASTAVA, Rupesh K. ; KOUTNIK, Jan ; STEUNEBRINK, Bas R. ; SCHMIDHUBER, Jürgen: LSTM: A Search Space Odyssey. In: *IEEE Transactions on*

- Neural Networks and Learning Systems* 28 (2017), Oct, Nr. 10, S. 2222–2232. – URL <http://dx.doi.org/10.1109/TNNLS.2016.2582924>. – ISSN 2162-2388
- [20] HAMAGUCHI, Ryuhei ; FUJITA, Aito ; NEMOTO, Keisuke ; IMAIZUMI, Tomoyuki ; HIKOSAKA, Shuhei: *Effective Use of Dilated Convolutions for Segmenting Small Object Instances in Remote Sensing Imagery*. 2017
- [21] HE, Kaiming ; ZHANG, Xiangyu ; REN, Shaoqing ; SUN, Jian: *Deep Residual Learning for Image Recognition*. 2015
- [22] HIGGINS, Irina ; MATTHEY, Loïc ; PAL, Arka ; BURGESS, Christopher ; GLOROT, Xavier ; BOTVINICK, Matthew M. ; MOHAMED, Shakir ; LERCHNER, Alexander: beta-VAE: Learning Basic Visual Concepts with a Constrained Variational Framework. In: *ICLR*, 2017
- [23] HINTON, G. E. ; SALAKHUTDINOV, R. R.: Reducing the Dimensionality of Data with Neural Networks. In: *Science* 313 (2006), Nr. 5786, S. 504–507. – URL <https://science.sciencemag.org/content/313/5786/504>. – ISSN 0036-8075
- [24] HOCHREITER, Sepp ; SCHMIDHUBER, Jürgen: Long Short-term Memory. In: *Neural computation* 9 (1997), 12, S. 1735–80
- [25] IOFFE, Sergey ; SZEGEDY, Christian: *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015
- [26] ISOLA, Phillip ; ZHU, Jun-Yan ; ZHOU, Tinghui ; EFROS, Alexei A.: *Image-to-Image Translation with Conditional Adversarial Networks*. 2016
- [27] KARRAS, Tero ; LAINE, Samuli ; AILA, Timo: *A Style-Based Generator Architecture for Generative Adversarial Networks*. 2018
- [28] KIM, Taeksoo ; CHA, Moon-su ; KIM, Hyunsoo ; LEE, Jung K. ; KIM, Jiwon: *Learning to Discover Cross-Domain Relations with Generative Adversarial Networks*. 2017
- [29] KINGMA, Diederik P. ; BA, Jimmy: *Adam: A Method for Stochastic Optimization*. 2014
- [30] KINGMA, Diederik P. ; WELING, Max: *Auto-Encoding Variational Bayes*. 2013
- [31] KOZA, John R. ; BENNETT, Forrest H. ; ANDRE, David ; KEANE, Martin A.: *Automated Design of Both the Topology and Sizing of Analog Electrical Circuits Using Genetic Programming*. S. 151–170. In: GERO, John S. (Hrsg.) ; SUDWEEKS, Fay (Hrsg.): *Artificial Intelligence in Design '96*. Dordrecht : Springer Netherlands,

1996. – URL https://doi.org/10.1007/978-94-009-0279-4_9. – ISBN 978-94-009-0279-4
- [32] KRUSE, Rudolf ; BORGELT, Christian ; BRAUNE, Christian ; KLAWONN, Frank ; MOEWES, Christian ; STEINBRECHER, Matthias: *Computational Intelligence Eine methodische Einführung in Künstliche Neuronale Netze, Evolutionäre Algorithmen, Fuzzy-Systeme und Bayes-Netze*. 2. Wiesbaden : Springer Vieweg, 2015 (Computational Intelligence). – ISBN 978-3-658-10903-5
- [33] LAI, Qiuxia ; WANG, Wenguan ; KHAN, Salman ; SHEN, Jianbing ; SUN, Hanqiu ; SHAO, Ling: *Human vs Machine Attention in Neural Networks: A Comparative Study*. 2019
- [34] LEAVITT, Victoria M.: *Standard Progressive Matrices*. S. 2368–2368. In: KREUTZER, Jeffrey S. (Hrsg.) ; DELUCA, John (Hrsg.) ; CAPLAN, Bruce (Hrsg.): *Encyclopedia of Clinical Neuropsychology*. New York, NY : Springer New York, 2011. – URL https://doi.org/10.1007/978-0-387-79948-3_1068. – ISBN 978-0-387-79948-3
- [35] LECUN, Yann ; BOTTOU, Léon ; BENGIO, Yoshua ; HAFFNER, Patrick: Gradient-Based Learning Applied to Document Recognition. In: *Proceedings of the IEEE* Bd. 86, URL <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.7665>, 1998, S. 2278–2324
- [36] LI, Yang ; PAN, Quan ; WANG, Suhang ; PENG, Haiyun ; YANG, Tao ; CAMBRIA, Erik: *Disentangled Variational Auto-Encoder for Semi-supervised Learning*. 2017
- [37] LU, Yiwei ; REDDY, Mahesh Kumar K. ; NABAVI, Seyed shahabeddin ; WANG, Yang: *Future Frame Prediction Using Convolutional VRNN for Anomaly Detection*. 2019
- [38] MAKHZANI, Alireza ; FREY, Brendan: *k-Sparse Autoencoders*. 2013
- [39] MCCULLOCH, Warren S. ; PITTS, Walter: A logical calculus of the ideas immanent in nervous activity. In: *The bulletin of mathematical biophysics* 5 (1943), Dec, Nr. 4, S. 115–133. – URL <https://doi.org/10.1007/BF02478259>. – ISSN 1522-9602
- [40] MNIH, Volodymyr ; KAVUKCUOGLU, Koray ; SILVER, David ; GRAVES, Alex ; ANTONOGLU, Ioannis ; WIERSTRA, Daan ; RIEDMILLER, Martin: *Playing Atari with Deep Reinforcement Learning*. 2013

- [41] NILSSON, Nils J.: *Introduction to Machine Learning. An early draft of a proposed textbook*. 1996
- [42] OLAH, Christopher: *Understanding LSTM Networks*. URL: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>. – Abgerufen am 26.01.2020
- [43] OLIVASTRI, Silvio ; SINGH, Gurkirt ; CUZZOLIN, Fabio: *End-to-End Video Captioning*. 2019
- [44] O.V.: *Python 3 FAQ*. URL: docs.python.org/3/faq/general.html. – Aburfdatum: 27.12.2019
- [45] REDMON, Joseph ; FARHADI, Ali: *YOLOv3: An Incremental Improvement*. 2018
- [46] REN, Shaoqing ; HE, Kaiming ; GIRSHICK, Ross ; SUN, Jian: *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*. 2015
- [47] ROMANUKE, V.V.: Training Data Expansion and Boosting of Convolutional Neural Networks for Reducing the MNIST Dataset Error Rate. (2016), 12, S. 29–34
- [48] RUBINSTEIN, Reuven Y. ; KROESE, Dirk P.: *The Cross Entropy Method: A Unified Approach To Combinatorial Optimization, Monte-Carlo Simulation (Information Science and Statistics)*. Berlin, Heidelberg : Springer-Verlag, 2004. – ISBN 038721240X
- [49] SAMMUT, Claude (Hrsg.) ; WEBB, Geoffrey I. (Hrsg.): *Mean Absolute Error, Mean Squared Error*. In: SAMMUT, Claude (Hrsg.) ; WEBB, Geoffrey I. (Hrsg.): *Encyclopedia of Machine Learning*. Boston, MA : Springer US, 2010. – URL https://doi.org/10.1007/978-0-387-30164-8_525. – ISBN 978-0-387-30164-8
- [50] SANTORO, Adam ; RAPOSO, David ; BARRETT, David G. T. ; MALINOWSKI, Mateusz ; PASCANU, Razvan ; BATTAGLIA, Peter ; LILICRAP, Timothy: *A simple neural network module for relational reasoning*. 2017
- [51] SHI, Xingjian ; CHEN, Zhourong ; WANG, Hao ; YEUNG, Dit-Yan ; WONG, Wai kin ; WOO, Wang chun: *Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting*. 2015

- [52] SILVER, David ; SCHRIITWIESER, Julian ; SIMONYAN, Karen ; ANTONOGLU, Ioannis ; HUANG, Aja ; GUEZ, Arthur ; HUBERT, Thomas ; BAKER, Lucas ; LAI, Matthew ; BOLTON, Adrian ; CHEN, Yutian ; LILLICRAP, Timothy ; HUI, Fan ; SIFRE, Laurent ; DRIESSCHE, George van den ; GRAEPEL, Thore ; HASSABIS, Demis: Mastering the game of Go without human knowledge. In: *Nature* 550 (2017), Oktober, S. 354–. – URL <http://dx.doi.org/10.1038/nature24270>
- [53] SMITH, Virginia ; FORTE, Simone ; JORDAN, Michael I. ; JAGGI, Martin: *L1-Regularized Distributed Optimization: A Communication-Efficient Primal-Dual Framework*. 2015
- [54] STEINWART, Ingo ; CHRISTMANN, Andreas: *Support Vector Machines*. 1st. Springer Publishing Company, Incorporated, 2008. – ISBN 0387772413
- [55] SULUN, Serkan: *Deep Learned Frame Prediction for Video Compression*. 2018
- [56] TAN, Chuanqi ; SUN, Fuchun ; KONG, Tao ; ZHANG, Wenchang ; YANG, Chao ; LIU, Chunfang: *A Survey on Deep Transfer Learning*. 2018
- [57] VASWANI, Ashish ; SHAZEER, Noam ; PARMAR, Niki ; USZKOREIT, Jakob ; JONES, Llion ; GOMEZ, Aidan N. ; KAISER, Lukasz ; POLOSUKHIN, Illia: *Attention Is All You Need*. 2017
- [58] YIN, Wenpeng ; KANN, Katharina ; YU, Mo ; SCHÜTZE, Hinrich: *Comparative Study of CNN and RNN for Natural Language Processing*. 2017
- [59] YU, Fisher ; KOLTUN, Vladlen: *Multi-Scale Context Aggregation by Dilated Convolutions*. 2015
- [60] ZHANG, Shuai ; YAO, Lina ; SUN, Aixin ; TAY, Yi: Deep Learning Based Recommender System. In: *ACM Computing Surveys* 52 (2019), Feb, Nr. 1, S. 138. – URL <http://dx.doi.org/10.1145/3285029>. – ISSN 0360-0300
- [61] ZIEGLER, Thomas ; FRITSCHKE, Manuel ; KUHN, Lorenz ; DONHAUSER, Konstantin: *Efficient Smoothing of Dilated Convolutions for Image Segmentation*. 2019

A Anhang

A.1 Symbole, Sequenz- und Bewegungsmuster des Datensatzes

Äußere Symbole



Abbildung A.1: Alle äußeren Symbole.

Innere Symbole



Abbildung A.2: Alle inneren Symbole.

Sequenzmuster

- [1, 1, 1, 1, 1, 1]
- [1, 2, 1, 2, 1, 2]
- ["r", 1, "r", 1, "r", 1]
- [1, 2, 2, 1, 2, 2]
- [1, 2, 3, 1, 2, 3]
- [1, 1, 2, 1, 1, 2]
- [1, 1, 1, 2, 2, 2]
- [1, 1, 2, 2, 3, 3]

- ["r", 1, 1, "r", 1, 1]

Erläuterung: 1, 2 und 3 stellen Platzhalter für bestimmte, später ausgewählte Symbole dar. „r“ bezeichnet ein zufälliges Symbol.

Bewegungsmuster

- [(1, 0), (1, 0), (1, 0), (1, 0), (1, 0)]
- [(1, 0), (-1, 0), (1, 0), (-1, 0), (1, 0)]
- [(1, 1), (1, 1), (1, 1), (1, 1), (1, 1)]
- [(1, 1), (-1, -1), (1, 1), (-1, -1), (1, 1)]

Erläuterung: Die Tupel stellen einen Schritt zwischen zwei Diagrammen dar und tragen die Bedeutung (*x-Richtung*, *y-Richtung*). Jedes Muster kann auch gedreht und/oder gespiegelt auftreten.

A.2 Verwendete Bibliotheken

- NumPy
- OpenCV
- Pickle
- Keras
- Tensorflow
- Matplotlib
- Scikit-image

A.3 Modelleinstellungen

A.3.1 Hyperparameter

Für alle Modelle gilt:

Nach jeder Schicht folgt eine ReLU Aktivierung und Batch Normalization.

Ausnahme: Letzte Schicht, diese immer ohne Aktivierung und ohne Batch Normalization.

Kostenfunktion: MAE

Optimierer: Adam

Batch Size: 32

Training: Solange, bis es 10 Epochen lang keine Verbesserung auf dem Validierungsdatensatz gab.

A.3.2 Verwendete Keras Einstellungen

Schichten

```
keras.layers.Conv2D(filters , kernel_size , strides=(1, 1),
padding='valid' , data_format=None, dilation_rate=(1, 1),
activation=None, use_bias=True,
kernel_initializer='glorot_uniform' ,
bias_initializer='zeros' , kernel_regularizer=None,
bias_regularizer=None, activity_regularizer=None,
kernel_constraint=None, bias_constraint=None)
```

```
keras.layers.MaxPooling2D(pool_size=(2, 2), strides=None,
padding='valid' , data_format=None)
```

```
keras.layers.UpSampling2D(size=(2, 2),
data_format=None, interpolation='nearest')
```

```
keras.layers.Dense(units , activation=None, use_bias=True,
kernel_initializer='glorot_uniform' , bias_initializer='zeros' ,
kernel_regularizer=None, bias_regularizer=None,
```

```
activity_regularizer=None, kernel_constraint=None,  
bias_constraint=None)
```

```
keras.layers.ConvLSTM2D(filters, kernel_size, strides=(1, 1),  
padding='valid', data_format=None, dilation_rate=(1, 1),  
activation='tanh', recurrent_activation='hard_sigmoid',  
use_bias=True, kernel_initializer='glorot_uniform',  
recurrent_initializer='orthogonal', bias_initializer='zeros',  
unit_forget_bias=True, kernel_regularizer=None,  
recurrent_regularizer=None, bias_regularizer=None,  
activity_regularizer=None, kernel_constraint=None,  
recurrent_constraint=None, bias_constraint=None,  
return_sequences=False, go_backwards=False,  
stateful=False, dropout=0.0, recurrent_dropout=0.0)
```

```
keras.layers.GRU(units, activation='tanh',  
recurrent_activation='sigmoid', use_bias=True,  
kernel_initializer='glorot_uniform',  
recurrent_initializer='orthogonal', bias_initializer='zeros',  
kernel_regularizer=None, recurrent_regularizer=None,  
bias_regularizer=None, activity_regularizer=None,  
kernel_constraint=None, recurrent_constraint=None,  
bias_constraint=None, dropout=0.0, recurrent_dropout=0.0,  
implementation=2, return_sequences=False, return_state=False,  
go_backwards=False, stateful=False, unroll=False, reset_after=False)
```

```
keras.layers.LSTM(units, activation='tanh', recurrent_activation='sigmoid',  
use_bias=True, kernel_initializer='glorot_uniform',  
recurrent_initializer='orthogonal', bias_initializer='zeros',  
unit_forget_bias=True, kernel_regularizer=None,  
recurrent_regularizer=None, bias_regularizer=None,  
activity_regularizer=None, kernel_constraint=None,  
recurrent_constraint=None, bias_constraint=None,  
dropout=0.0, recurrent_dropout=0.0, implementation=2,  
return_sequences=False, return_state=False, go_backwards=False,  
stateful=False, unroll=False)
```

Optimierer

```
keras.optimizers.Adam(learning_rate=0.001, beta_1=0.9,  
beta_2=0.999, amsgrad=False)
```


A.4 Weitere Beispielvorhersagen

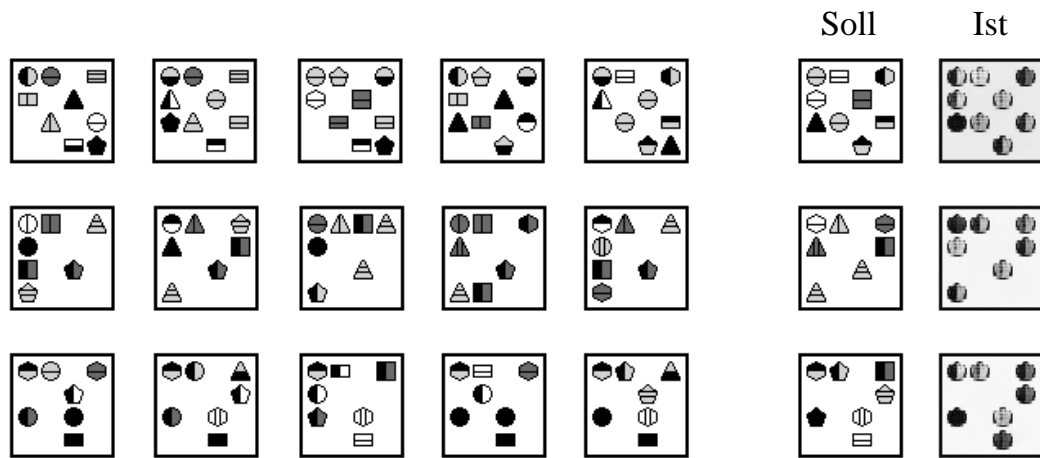


Abbildung A.3: Weitere Vorhersagen von CNN-GRU-512-128.

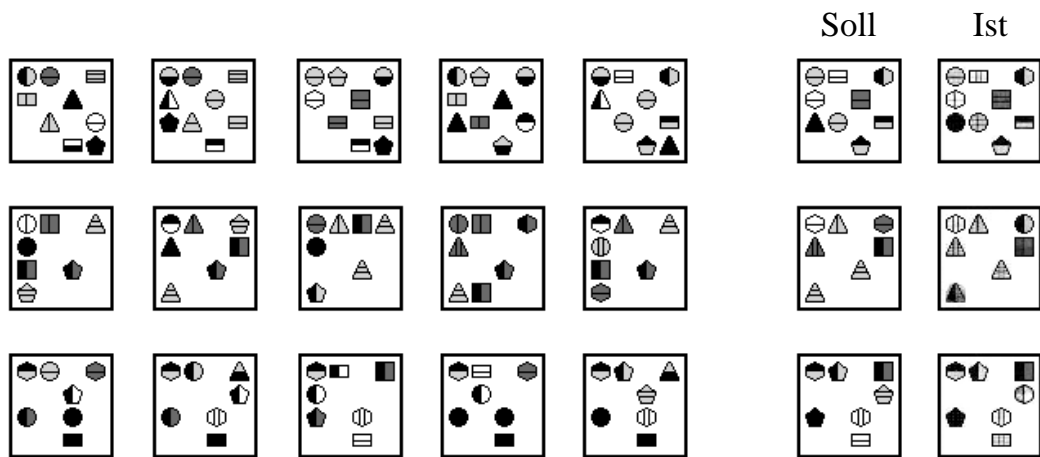


Abbildung A.4: Weitere Vorhersagen von CNN-ConvLSTM-2x512.

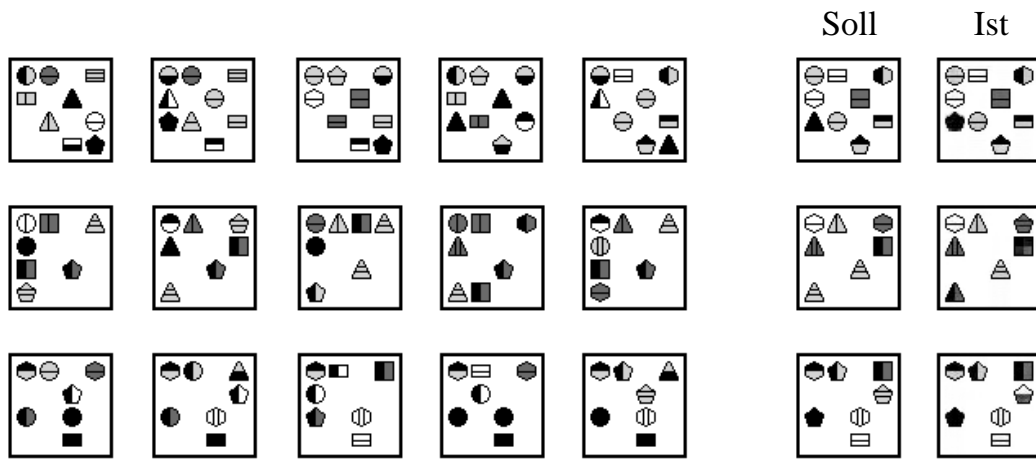


Abbildung A.5: Weitere Vorhersagen von ModuleCNN-NoDil-64.

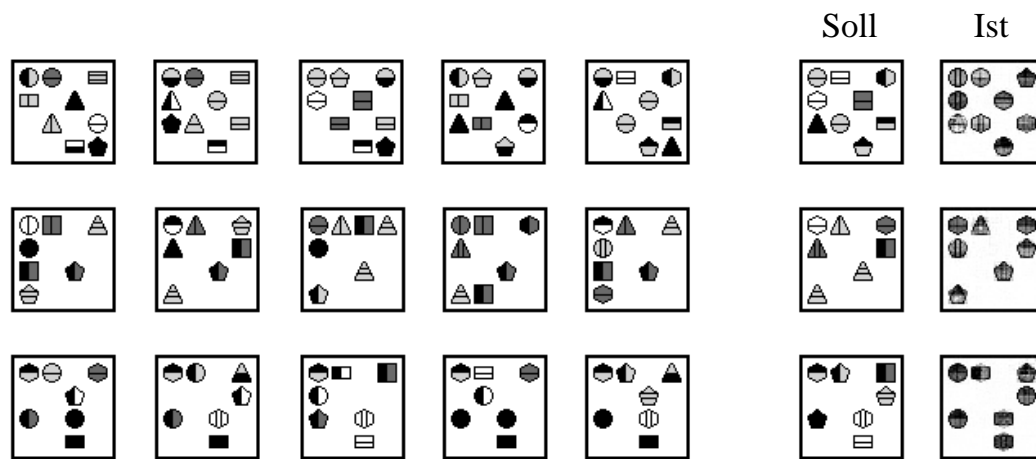


Abbildung A.6: Weitere Vorhersagen von MLPOnly-512.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „- bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] - ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen.“

Quelle: §16 Abs. 5 APSO-TI-BM bzw. §15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Intelligenztests und maschinelles Lernen - Lösungsansätze mit neuronalen Netzen für Diagrammsequenzen

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort

Datum

Unterschrift im Original