technische universität
dortmund

Bachelorarbeit

Human Activity Recognition mithilfe eines Fully Convolutional Networks am Beispiel des LARa Datensets

Dennis Krön
23. Dezember 2021

Gutachter:
Prof. Dr. Gernot Fink
Wilmar Fernando Moya Rueda

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhlbezeichnung (LS-12)
http://ls12-www.cs.tu-dortmund.de

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **3**

2.1 Human Activity Recognition . . . 3

2.2 Neuronale Netze . . . 5

2.2.1 Feedforward Neural Networks . . . 6

2.2.1.1 Perzeptron . . . 6

2.2.1.2 Multilayer Perzeptron (MLP) . . . 8

2.2.1.3 Aktivierungsfunktionen . . . 9

2.2.1.4 Training von neuronalen Netzen . . . 12

2.3 Convolutional Neural Networks . . . 16

2.3.1 Convolutional Layer . . . 17

2.3.2 Pooling Layer . . . 19

2.3.3 Fully Connected Layer . . . 20

2.3.4 Temporal Convolutional Networks . . . 20

2.3.5 Fully Convolutional Networks . . . 21

2.4 Auto-Encoder-Decoder . . . 22

3 Verwandte Arbeiten **25**

3.1 Human Activity Recognition . . . 25

3.1.1 Datenerfassung . . . 26

3.1.2 Preprocessing . . . 27

3.1.3 Segmentierung . . . 27

3.1.4 Feature Extraktion . . . 28

3.1.5 Klassifikation . . . 29

3.2 Temporal Convolutional Networks in der Human Activity Recognition . . . 29

3.2.1 Deep Convolutional Neural Networks auf mehrkanaligen Zeitreihendaten . . . 30

3.2.2 Temporal Convolutional Network des LARa-Datensets . . . 32

3.3 Fully Convolutional Network in der semantischen Segmentierung 33
3.3.1 Anpassung der Klassifizierungs-CNNs 33
3.3.2 Skip-Architektur basierend auf dem VGG16-Net 34
3.4 Deep Auto-Encoder-Set Network zur Human Activity Recognition 35
3.4.1 Aufbau des Auto-Set Netzwerks 35
3.4.2 Kostenfunktion beim Training des Encoders 36
3.4.3 Ergebnisse 36
3.4.4 Diskussion 37

4 Methoden **39**
4.1 Erstellen des Fully Convolutional Networks 39
4.1.1 Aufbau des Netzes 40
4.1.2 Training 41
4.2 Erstellen des Auto-Encoder-Decoders 41
4.2.1 Aufbau des Netzes 42
4.2.2 Training 43
4.2.3 Synthetische Datengenerierung 44

5 Evaluation **45**
5.1 LARa-Datenset 45
5.1.1 Attributrepräsentation des LARa-Datensets 46
5.1.2 Klassen des LARa-Datensets 47
5.2 Performance Metrik 47
5.3 Trainingssetup 50
5.3.1 Encoder 50
5.3.1.1 Finetuning des Encoders 50
5.3.2 Auto Encoder Decoder 50
5.4 Experimente und Ergebnisse 51
5.4.1 Rekonstruktion der Baseline 51
5.4.2 Encoder 52
5.4.2.1 Anzahl der Featuremaps und Lernrate 52
5.4.2.2 Poolinglayer 53
5.4.2.3 Finetuning 56
5.4.2.4 Zusammenfassung der Ergebnisse zur Konstruktion des Encoders 56
5.4.3 Auto Encoder Decoder 60
5.4.3.1 Training mithilfe eines Decoders und dem Attributvektor als Zwischendarstellung 60

5.4.3.2 Training des Encoders mithilfe eines vortrainierten Decoders und dem Attributvektor als Zwischendarstellung . . . 62
5.4.3.3 Training mithilfe eines Decoders und geänderter Zwischendarstellung . . . 63
5.4.3.4 Zusammenfassung der Ergebnisse zum Training des Encoders mithilfe eines Decoders . . . 64
5.4.4 Visualisierung . . . 69
5.4.5 Synthetische Datengenerierung . . . 71

6 Zusammenfassung **75**

Abbildungsverzeichnis **79**

Literaturverzeichnis **86**

Kapitel 1

Einleitung

Um das Verhalten von Lebewesen zu verstehen, spielt die Beobachtung ihrer Bewegungen eine wichtige Rolle. So kann der Mensch an der Körpersprache eines Tieres erkennen, ob es sich fürchtet, oder ob es sich im Angriffsmodus befindet. Ebenfalls sind Bewegungen im Tierreich ein wichtiger Bestandteil der Kommunikation innerhalb einer Spezies wie etwa bei einem Balztanz.

Auch beim Menschen sind Bewegungen ein fester Bestandteil der zwischenmenschlichen Interaktion. Anhand dessen, wie sich ein Mensch bewegt, lassen sich Rückschlüsse auf seine Persönlichkeit, körperliche Verfassung und den psychischen Zustand ziehen.

Das Forschungsgebiet der Human Activity Recognition befasst sich mit der Aufgabe, menschliche Bewegungen mithilfe von aufgezeichneten Sensordaten maschinell zu erkennen. Maschinelle Methoden zur Erkennung von menschlichen Aktivitäten haben den Vorteil, dass sie kostengünstiger und schneller große Mengen von Bewegungsdaten verarbeiten können als Menschen. Daraus ergibt sich sowohl ein zeitlicher, als auch ein monetärer Vorteil. Allerdings besitzen Maschinen heutzutage noch nicht die Präzision eines Menschen bei der Bewegungserkennung.

In der Praxis werden unter anderem neuronale Netze verwendet, die anhand mehrerer aufgenommener Sensorkurven versuchen zu bestimmen, welche Aktivität diese darstellen. Die Netze werden vorher mit Hilfe von Trainingsdaten darauf konditioniert, bestimmte Aktivitäten zu erkennen.

Eine besondere Form von neuronalen Netzen sind die Convolutional Neural Networks (Faltungsnetze). Durch ihren Aufbau eignen sie sich gut, um Daten mit einer rasterartigen Struktur, wie etwa Bilder oder Sensorkurven, zu analysieren [22]. In der Praxis konnten besondere Convolutional Neural Networks, welche speziell zur Analyse von Zeitreihendaten entwickelt wurden, vielversprechende Ergebnisse bei der Erkennung von menschlichen Aktivitäten basierend auf Sensordaten liefern [58] [43].

Diese Arbeit befasst sich mit dem neuronalen Netz, beschrieben in [43], welches zur Erkennung von Aktivitäten in einem Lagerhaus benutzt wird. Ziel ist es, mit Erkenntnissen

zu Fully Convolutional Networks aus dem Forschungsbereich der semantischen Segmentierung [33] die Architektur des Netzes aus [43] anzupassen und damit die Performance zu verbessern.

Ein weiteres Problem bei dem Einsatz von neuronalen Netzen in der Human Activity Recognition ist die Annotation von Trainingsdaten [55]. Sie ist meist sehr aufwendig und kostet viel Zeit. Um dieses Problem zu lösen, kommen spezielle Auto-Encoder-Decoder Netzwerke zum Einsatz, welche ohne annotierte Daten eine Repräsentation der Daten erlernen können.

Zusätzlich kann ein neuronales Netz beim Training mit einem Decoder weitere Features erlernen, welche die Daten besser beschreiben [55]. Dadurch lässt sich ebenfalls die Performance eines Netzes verbessern.

Das konstruierte Fully Convolutional Network soll mithilfe eines selbst entwickelten Decoders trainiert werden. Das soll dazu führen, dass die Performance des Fully Convolutional Networks weiter gesteigert wird.

Der Vorteil eines Fully Convolutional Networks ist, dass es keine Fully Connected Layer enthält. Diese Layer eignen sich zwar sehr gut zur Analyse, verwerfen jedoch die räumliche Struktur der Daten [33]. Der Vorteil, der Beibehaltung räumlicher Strukturen im Fully Convolutional Network, soll genutzt werden, um die Eingabedaten aus der Ausgabe mittels eines Decoders zu rekonstruieren. Gelingt die Rekonstruktion, ist eine Generierung synthetischer Bewegungsdaten durch den Decoder möglich. Indem die Eingabe des Decoders wiederholt verändert wird, generiert er Sensorkurven, die in einem Datenset gesammelt werden können. Auf diese Art kann das Problem der kosten- und zeitintensiven Erstellung von Datensets umgangen werden.

Kapitel 2

Grundlagen

Dieses Kapitel erläutert die Grundlagen, auf denen die Arbeit aufbaut. In Abschnitt 2.1 wird erklärt, was Human Activity Recognition ist und wie ein Klassifizierer für menschliche Aktivitäten entwickelt werden kann. Da im Rahmen dieser Bachelorarbeit neuronale Netze zur Klassifizierung eingesetzt werden, befasst sich Abschnitt 2.2 mit den Grundlagen von neuronalen Netzen. Ausgehend von der Beschreibung eines einzelnen Neurons wird auf das Perzeptron und das Multi Layer Perzeptron eingegangen. In den letzten beiden Abschnitten werden Aktivierungsfunktionen vorgestellt und das Training von neuronalen Netzen erläutert.

In Abschnitt 2.3 wird genauer auf Convolutional Neural Networks eingegangen. Zunächst werden Convolutional Layer, Pooling Layer und Fully Connected Layer und ihre Aufgaben in Neuronalen Netzen beschrieben. Anschließend wird auf das Problem der verworfenen strukturellen Informationen durch Fully Connected Layer eingegangen, welches die Hauptmotivation für diese Arbeit darstellt.

Abschließend werden die beiden speziellen Convolutional Neural Networks beschrieben, die in dieser Arbeit verwendet werden. Zudem wird das Konzept eines Auto-Encoders erklärt.

2.1 Human Activity Recognition

Human Activity Recognition, kurz HAR, bezeichnet die Aufgabe, menschliche Bewegungen zu klassifizieren [48] und bildet einen eigenen Unterbereich der Pattern Recognition. Ziel in der HAR ist es daher, einen Klassifizierer zu entwickeln, welcher in der Lage ist, bestimmte Aktivitäten zu erkennen. Im Gegensatz zur Activity Prediction wird nicht versucht, Bewegungen vorherzusagen, sondern zu erkennen, welche Bewegungen eine Person zu einem bestimmten Zeitpunkt durchführte [29].

Human Activity Recognition kann in vielen Bereichen angewendet werden. Im Alltag benutzen Menschen Fitnesstracker, die ihnen bei der Überwachung ihrer sportlichen

Leistungen helfen [17], in der Medizin können anhand der Bewegungen eines Menschen Krankheitsverläufe und Rehabilitationen nach einem Unfall überwacht werden [20]. Aber auch in der Industrie ist Human Activity Recognition anwendbar. So können diese durch die Analyse von Bewegungsabläufen optimiert und verbessert oder die Arbeitsumgebungen angepasst werden.

Im Bereich HAR gibt es allerdings auch viele Herausforderungen zu meistern. Dazu gehören zum einen die Herausforderungen der Pattern Recognition, aber auch spezifische HAR Probleme wie etwa die Diversität von physischen Aktivitäten [3]. Da Aktivitäten in verschiedenen Umgebungen anders ausgeführt werden, gestaltet es sich schwierig, anhand der charakteristischen Eigenschaften einer Bewegung diese zu klassifizieren. Ein weiteres Problem in der Human Activity Recognition ist die Klassenimbalance [3]. Einige Bewegungen kommen bei einem Menschen häufiger vor als andere, daher ist dies bei der Erstellung eines ausgeglichenen Datensatzes zu berücksichtigen. Die Annotation von Sensordaten in der HAR ist eine aufwändige Aufgabe, da aufgezeichnete Sensorkurven nicht immer einfach zu interpretieren sind [3]. Es besteht in der Human Activity Recognition auch das Problem der Intraklassen-Variabilität und Interklassen-Ähnlichkeit [3]. Zum einen wird eine Bewegung von Mensch zu Mensch nicht immer gleich ausgeführt und zum anderen können zwei unterschiedliche Aktivitäten gleiche charakteristsche Eigenschaften in den Sensordaten aufweisen [3].

Der erste Schritt einen Klassifizierer zu entwickeln, ist die Datenerfassung. Die Daten in der Human Activity Recognition werden über unterschiedliche Sensoren und Kameras erfasst [48]. Dazu gehören Beschleunigungssensoren, Motion-Capturing Systeme, aber auch Herzschlagsensoren. Dabei gibt es jedoch keine einheitliche Vorschrift für die Anzahl und Platzierungen der Sensoren am Körper. Bei einem Experiment werden Versuchspersonen mit einem Set von Sensoren ausgestattet und überwacht. Die Personen führen bestimmte Bewegungen aus, die der Klassifizierer später einer Aktivität zuordnen soll. Die Experimente können durch Nachbildung eines realen Umfelds unter Laborbedingungen durchgeführt, oder im Alltag unter echten Bedingungen aufgezeichnet werden.

Durch die Experimente erhält man pro Versuchsperson eine Sammlung von Sensorkurven, welche die Aktivitäten während des Experiments beschreiben.

Nach der Aufnahme werden die Daten einem Preprocessing unterzogen [48]. Durch eine Vorverarbeitung der Daten können unerwünschte Einflüsse, wie etwa ein zufälliges Rauschen oder Fehlfunktionen der Sensoren, in den gesammelten Daten eliminiert werden [48]. Eine Normalisierung der Daten ist ebenfalls möglich, dadurch können die Ergebnisse in einen einheitlichen Wertebereich transformiert werden, wenn unterschiedliche Sensoren verwendet wurden.

Danach folgt die Segmentierung. Dabei werden aus den preprocessed Daten Segmente extrahiert, welche sehr wahrscheinlich eine Aktivität darstellen [48]. In der HAR wird dafür meist der sliding-window Ansatz gewählt [48]. Es wird ein ´Fenster´ mit einer festgelegten

Größe über die Sensordaten gelegt und der Inhalt extrahiert. Das Fenster wird dann mit einer bestimmten Distanz (step) weitergeschoben und der Vorgang der Extraktion wiederholt.

Nach der Segmentierung folgt die Feature Extraktion. Bei den traditionellen Ansätzen werden sogenannte Handcrafted-Features manuell bestimmt. Die Features sollen die wesentlichen Eigenschaften einer bestimmten Aktivität darstellen [48]. Nach der Reduktion der Dimensionalität der Features werden diese dazu verwendet, die Klassifizierer zu trainieren.

Die Reduktion der Featuredimensionalität dient dazu, die Genauigkeit der Klassifizierung zu verbessern. Je mehr Merkmale für die Klassifikation zur Verfügung stehen, desto schwieriger gestaltet es sich, aus diesen eine bestimmte Klasse zu ermitteln [16].

In der Pattern Recognition gibt es viele verschiedene statistische Modelle, welche als Klassifizierer dienen können. Eine oft verwendete statistische Methode im Bereich der HAR ist das Hidden Markov Model (HMM).

Mit steigender Beliebtheit werden aber auch Deeplearning Strukturen in der Human Activity Recognition eingesetzt, da sie die Featureextraktion und Klassifikation zu einem Ansatz zusammenfassen [48].

2.2 Neuronale Netze

Nicht nur in der Wetteranalyse oder beim Analysieren von Aktienkursen und Wirtschaftsdaten werden Deeplearning-Ansätze immer beliebter, auch in der Human Activity Recognition werden Deeplearning Ansätze mit großem Erfolg eingesetzt.

Um diese Deeplearning Anwendungen zu realisieren, werden in der Informatik künstliche neuronale Netze (KNNs) verwendet. Die Netze basieren auf einer abstrahierten Form der Informationsverarbeitung in biologischen neuronalen Netzen, welche die Informationsarchitektur für das menschliche Nervensystem und das Gehirn bilden. In der Biologie bestehen diese Netze aus Neuronen, welche sich wiederum aus drei Teilen zusammensetzen. Die Dendriten bilden den ersten Teil eines Neurons. Diese nehmen die Informationen auf und leiten sie in den zweiten Teil, den Zellkörper weiter. Der dritte Teil ist das Axon. Es leitet die Informationen aus dem Zellkörper an nachfolgende Zellen weiter, sobald ein gewisser Schwellenwert am Axonhügel überschritten wurde.

Das erste künstliche Neuron in der Informatik wurde im Jahr 1943 von Warren S. Mcculloch und Walter Pitts veröffentlicht [39]. McCulloch-Pitts-Neuronen sind in der Lage, eine beliebige Anzahl an binären Eingaben zu verarbeiten. Übertreffen die eingehenden Signale in der Summe den in der Zelle festgelegten Schwellenwert θ, so gibt die Zelle den Wert ´1´ aus. Falls θ nicht überschritten wird, gibt die Zelle eine ´0´ zurück. Durch genaueres Betrachten des McCulloch-Pitts-Neurons lässt sich der Aufbau eines biologischen Neurons sehr gut wiedererkennen (siehe Abbildung 2.1).

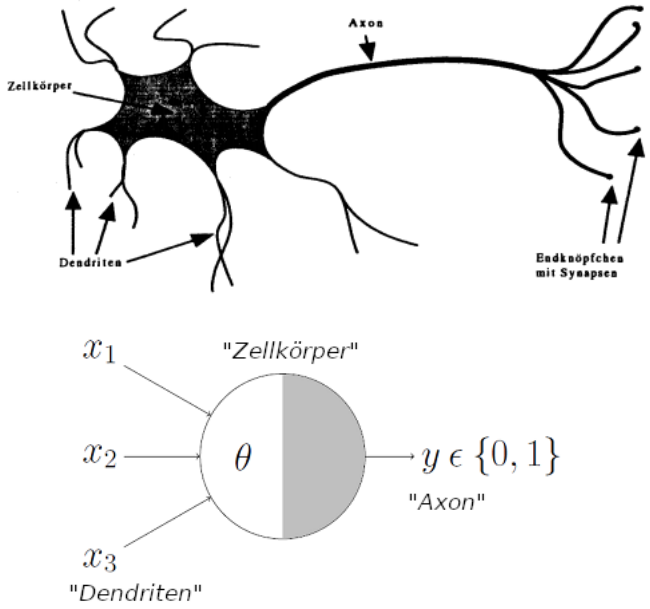

Abbildung 2.1: Oben: Aufbau eines biologischen Neurons entnommen aus[54]. Unten: McCulloch-Pitts-Neuron mit θ als Aktivierungswert entnommen aus [11]

2.2.1 Feedforward Neural Networks

Werden mehrere künstliche Neuronen miteinander vernetzt, so erhält man ein künstliches neuronales Netz. Dabei gibt es zwei verschiedene Vernetzungsarten. Werden die Neuronen wie in einem gerichteten azyklischen Graphen angeordnet, spricht man von einem feedforward Network. Bei dieser Vernetzungsart fließen die Informationen nur in eine Richtung. Neben den feedforward Networks gibt es recurrent Networks, welche sich darin unterscheiden, dass auch zyklische Verbindungen zwischen den Neuronen bestehen [22]. Recurrent Networks werden im Rahmen dieser Arbeit nicht weiter betrachtet.

In den beiden folgenden Abschnitten werden zwei feedforward Networks vorgestellt und ihre Funktionsweise erläutert.

2.2.1.1 Perzeptron

Das Perzeptron, auch single-layer Perzeptron genannt, wurde von Frank Rosenblatt entwickelt und 1958 erstmals vorgestellt [49]. Es stellt die einfachste Form eines feedforward Networks dar. Es besteht aus nur einem Layer und ist in der Lage, linear separierbare Probleme zu klassifizieren. In Abbildung 2.2 ist ein solches Perzeptron dargestellt.

Das Perzeptron erhält einen n-dimensionalen Inputvektor $x \epsilon \mathbb{R}^n$ und bildet diesen auf den binären Output $f(x) \epsilon \mathbb{B}$ ab [42]. Die Berechnung der Ausgabewerte kann mit folgender Funktion beschrieben werden:

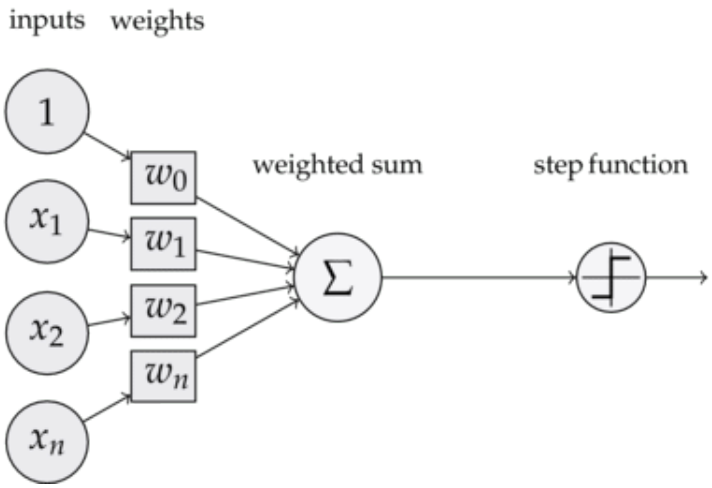

Abbildung 2.2: Bild eines single-layer Perzeptrons entnommen aus [41].

$$f(x) = \begin{cases} 1 & \text{wenn } \sum_{i=0}^{n} x_i w_i + b > 0 \\ 0 & \text{sonst} \end{cases} \tag{2.1}$$

Dabei steht x_i für die i-te Stelle im Inputvektor $x \epsilon \mathbb{R}^n$, w_i für die i-te Stelle in dem gleichgroßen Gewichtungsvektor $w \epsilon \mathbb{R}^n$ und b für einen, während des Trainings gelernten Bias, welcher unabhängig vom Eingabewert die Entscheidungsgrenze verschiebt. Überschreitet das Skalarprodukt des Inputvektors und des Gewichtungsvektors ($\sum_{i=0}^{n} x_i w_i$) zusammen mit dem Bias den Schwellenwert (in diesem Falle ´0´), gibt das Perzeptron eine ´1´ aus, andernfalls ´0´. Diese Funktion stellt die Aktivierungsfunktion des Perzeptrons dar. Im Laufe der Zeit wurden weitere Aktivierungsfunktionen entwickelt, die es einem Neuron ermöglichen, Werte zwischen ´0´ und ´1´ auszugeben. Eine genauere Betrachtung der Aktivierungsfunktionen findet in Abschnitt 2.2.1.3 statt.

Um ein Perzeptron zu trainieren, benötigt man zusätzlich eine Lernregel. Dies ist ein Algorithmus, der vorgibt welche Gewichte wie stark inkrementiert oder dekrementiert werden. Die Lernregel wird während der Trainingsphase angewandt. Ein einfaches Beispiel für eine Lernregel zum Trainieren ist die Delta-Regel. Dazu benötigt man eine Menge von Inputvektoren und die dazugehörigen Label. Die Label sind die erwarteten Ausgaben. Beim Training wird die oben beschriebene Funktion auf den ersten Inputvektor angewandt. Falls der Wert des Ergebnisses kleiner als der Wert des Labels ist, werden die Gewichte im Gewichtungsvektor w abhängig von der Größe des Unterschieds erhöht. Ist der berechntete Wert größer als der gewünschte Wert, werden die Gewichte verkleinert. Stimmt das Ergebnis mit dem Label überein, so wird dieser Schritt mit dem nächsten Input/Label Paar wiederholt. Weitere Trainingsmethoden werden im Abschnitt 2.2.1.4 behandelt.

Solange die Entscheidungsprobleme linear separierbar sind, kann ein einfaches Perzeptron zur Klassifizierung eingesetzt werden. Minsky und Papert zeigten, dass ein Perzeptron

bei nicht linear separierbaren Problemen an seine Grenzen stößt. Sie wiesen nach, dass die XOR-Funktion, bei der es sich um eine solche Funktion handelt, nicht durch ein single-layer Perzeptron berechnet werden kann [40]. Jedoch stellte sich auch heraus, dass dieses Problem durch mehrere, sequenziell aneinandergereihte Perzeptronen gelöst werden kann.

2.2.1.2 Multilayer Perzeptron (MLP)

Ein KNN aus mehreren sequenziell angeordneten Perzeptronen nennt man Multi Layer Perzeptron oder auch MLP.

Dabei werden mehrere Neuronen in einer Schicht, einem sogenannten Layer, angeordnet. Die Neuronen innerhalb eines Layers sind nicht miteinander verbunden. Jedes Neuron eines Layers ist jedoch mit jedem Neuron in der folgenden Schicht verknüpft.

Ein MLP besteht mindestens aus drei Layern, einem Inputlayer, einem Outputlayer und minimal einem Hiddenlayer.

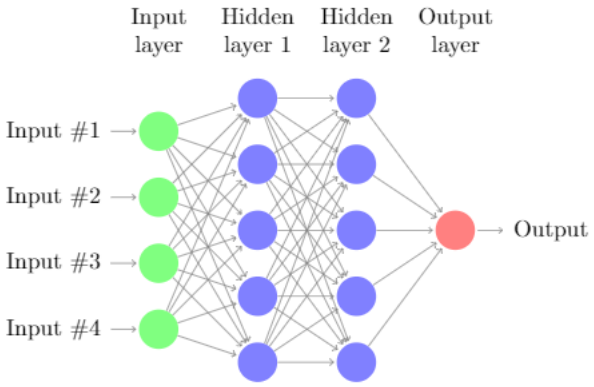

Abbildung 2.3: Bild eines Multi Layer Perzeptrons. Das MLP besitzt ein Inputlayer (grün), bestehend aus vier Neuronen, zwei Hiddenlayer (blau), bestehend aus jeweils 5 Neuronen und ein Outputlayer (rot), bestehend aus einem Neuron.

Der Inputlayer ist dafür verantwortlich, dem nachfolgenden Layer den Input über seine Neuronen bereitzustellen.

Nach dem Inputlayer folgen die Hiddenlayer. Jeder Hiddenlayer berechnet eine Zwischendarstellung des erhaltenen Inputs. Der Output eines jeden Neurons einer Schicht wird wie bei einem Perzeptron berechnet. Die Gewichtungsvektoren w_i für jedes Neuron können in einer Matrix W zusammengefasst werden, wodurch sich eine Gewichtungsmatrix für das gesamte Layer ergibt. Dabei repräsentiert die i-te Zeile in der Gewichtungsmatrix den Gewichtungsvektor des i-ten Neurons des Layers.

Durch die Erweiterung der Gewichtungsmatrix mit einer zusätzlichen Spalte, kann der Bias für jedes Neuron ebenfalls in der Gewichtungsmatrix W festgehalten werden. Zusätzlich wird der Inputvektor um einen weiteren Eintrag mit dem Wert '1' erweitert. Dadurch lässt sich die Berechnung eines Layers durch eine einfache Matrixmultiplikation mit dem Inputvektor x und der Gewichtungsmatrix W darstellen. Unter Beachtung der Aktivierungsfunktionen der Neuronen lässt sich die Berechnungsfunktion des Outputvektors eines Layers beschreiben durch:

$$f(x) = \phi(Wx) \tag{2.2}$$

Dabei bezeichnet x den Inputvektor der Schicht und ϕ die Aktivierungsfunktion der Neuronen. In dieser Arbeit wird daher zur Vereinfachung angenommen, dass der Bias ein Teil der Gewichtungsmatrix ist.

Das Ergebnis dieser Funktion dient als Inputvektor des nächsten Layers. Das letzte Layer ist das Outputlayer. Das Ergebnis der Berechnungen wird in dieser Schicht ausgegeben. Dabei lässt sich das Ergebnis des Netzes als Konkatenation aus den Berechnungsfunktionen der Layer darstellen:

$$f(x) = f_n(f_{n-1}(f_2(f_1(x)))) \tag{2.3}$$

f_i bezeichnet die Berechnungsfunktion des i-ten Layers und n die Anzahl der Layer. Diese Art von Verkettung ist die am häufigsten verwendete Struktur in neuronalen Netzen[22].

Durch die Verwendung von MLPs können, anders als mit einem single-layer Perzeptron, nicht linear separierbare Probleme approximiert werden. Dies liegt daran, dass der Output des MLPs aus Verkettung von mehreren Funktionen besteht, wodurch komplexere Funktionen dargestellt werden können [22].

2.2.1.3 Aktivierungsfunktionen

Wie bereits in Abschnitt 2.2.1.1 erwähnt, besitzt jedes Neuron eine Aktivierungsfunktion. Diese ist einer der wichtigsten Bestandteile eines Neurons, sie bestimmt, ob und wie stark ein Neuron 'feuert'. Das Perzeptron nach Rosenblatt benutzt die Heaviside-Stufenfunktion, auch Schwellenwertfunktion genannt (Abbildung 2.4).

Sie stellt die einfachste Aktivierungsfunktion dar und berechnet nur, ob das Neuron feuert oder nicht. Dieses Verhalten eignet sich gut, um binäre Klassifizierungsaufgaben zu berechnen, stößt jedoch an seine Grenzen, falls zwischen mehr als zwei Ausgaben unterschieden werden soll [53]. Die Funktion lässt sich durch folgende Formel beschreiben:

$$\phi(x) = \begin{cases} 1 & \text{wenn } x > 0 \\ 0 & \text{sonst} \end{cases} \tag{2.4}$$

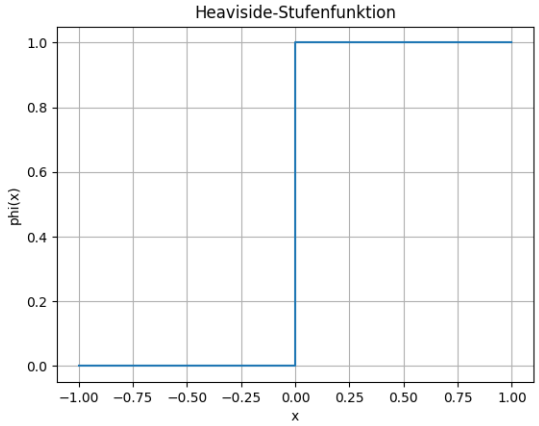

Abbildung 2.4: Die Heaviside-Stufenfunktion.

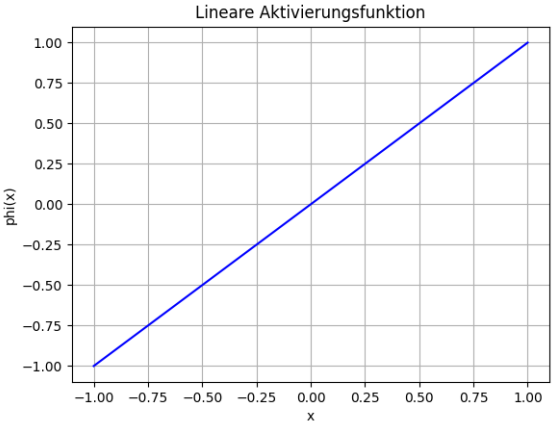

Abbildung 2.5: Die lineare Aktivierungsfunktion mit a = 1.

Ersetzt man x durch den Ausdruck $\sum_{i=0}^{n} x_i w_i + b$, welcher die Berechnung des Wertes eines Neurons ohne Aktivierungsfunktion darstellt, so erhält man die Formel 2.1.

Grundsätzlich unterscheidet man zwischen linearen und nicht-linearen Aktivierungsfunktionen. Ein Beispiel für eine lineare Aktivierungsfunktion ist die lineare Aktivierungsfunktion (Abbildung 2.5):

$$\phi(x) = ax \tag{2.5}$$

Lineare Funktionen sind jedoch dadurch beschränkt, dass eine Verknüpfung von derartigen Funktionen durch eine einzige lineare Funktion dargestellt werden kann. Daher lassen sich Netze mit mehreren linearen Schichten auf ein einschichtiges lineares Netz reduzieren. Sie eignen sich nicht, um komplexe Strukturen in Daten zu erkennen. Daher kommen sie nur bei simplen Problemen zum Einsatz [53].

Ein Beispiel für eine nicht-lineare Aktivierungsfunktion ist die Sigmoid Funktion (Abbildung 2.6). Diese Funktion ist eine Approximation der Heaviside-Stufenfunktion. Die Formel für die Sigmoid-Funktion lautet:

$$\phi(x) = \frac{1}{1 + e^{-x}} \tag{2.6}$$

Dabei steht e für die Eulersche Zahl. Im Gegensatz zur Stufenfunktion ist die Sigmoidfunktion differenzierbar, welches eine Eigenschaft ist, die für einige Trainingsalgorithmen unverzichtbar ist [14]. Die Ausgabe der Funktion liegt immer zwischen ´0´ und ´1´. Daher eignet sie sich gut, wenn es um die Berechnung von Wahrscheinlichkeiten geht. Im Laufe dieser Arbeit wird ein Netzwerk vorgestellt, welches die Sigmoidfunktion zur Approximation eines Attributvektors benutzt (siehe Abschnitt 3.2.2).

Jedoch ist die Sigmoidfunktion anfällig für das Problem der verschwindenden Gradienten beim gradientenbasierten Training von Netzen[24] [25] [14]. Dieses Problem tritt auf,

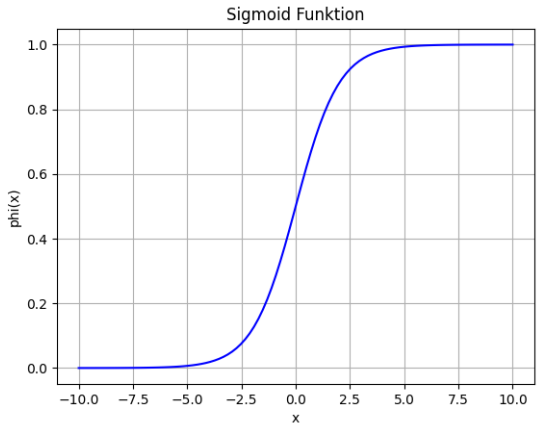

Abbildung 2.6: Die Sigmoid Funktion.

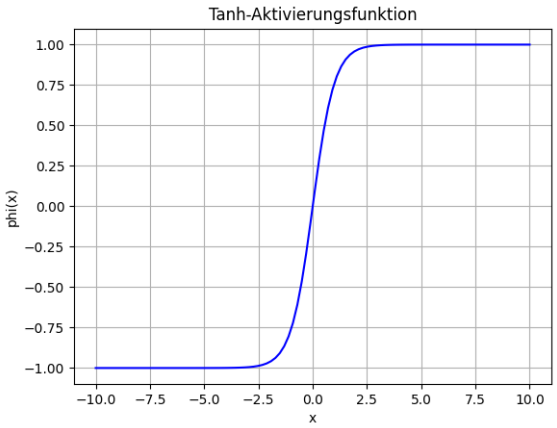

Abbildung 2.7: Die Tanh Aktivierungsfunktion.

wenn mehrere Layer eines größeren Netzes die Sigmoid Funktion als Aktivierungsfunktion verwenden. Beim gradientenbasierten Training mit dem Backpropagation-Algorithmus (genauer beschrieben in Abschnitt 2.2.1.4) wird der Gradient eines Gewichtes in einem der ersten Layer bestimmt, indem mithilfe der Kettenregel eine partielle Ableitung aus Richtung des Outputlayers berechnet wird. Da bei Benutzung der Kettenregel die einzelnen Ableitungen miteinander multipliziert werden und die Ableitung der Sigmoid Funktion für extrem große und kleine Werte fast null beträgt, werden bei der Gradientenberechnung für die ersten Layer viele sehr kleine Werte miteinander multipliziert[14]. Dadurch schrumpft der Wert des Gradienten exponentiell und er ‘verschwindet‘.

Vor allem bei tiefen Netzwerken führt dieses Problem dazu, dass ein Netz die Gewichtungen der ersten Layer sehr langsam lernt.

Ähnlich zu der Sigmoid Funktion ist die Tanh-Aktivierungsfunktion(Abbildung 2.7):

$$\phi(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.7}$$

Anders als die Sigmoid Funktion ist diese Funktion Null zentriert und besitzt einen steileren Anstieg. Durch die Null-Zentrierung ist es möglich, dass die Funktion sowohl auf positive als auch negative Zahlen abbildet, wodurch die Trainingszeit verringert werden kann [14]. Zusätzlich ist durch den steileren Anstieg die Ableitung und damit der Gradient der Funktion höher als bei der Sigmoid Funktion. Allerdings leidet die Tanh-Aktivierungsfunktion ebenfalls unter dem verschwindenden Gradienten Problem [14].

Eine weitere, sehr beliebte, nicht-lineare Aktivierungsfunktion ist die ReLU-Funktion (Abbildung 2.8). ReLU steht für Rectified Linear Unit und findet vor allem im Bereich des Deeplearnings Anwendung. Dabei handelt es sich um eine stückweise lineare Funktion, welche durch die Formel

$$\phi(x) = max(0, x) \tag{2.8}$$

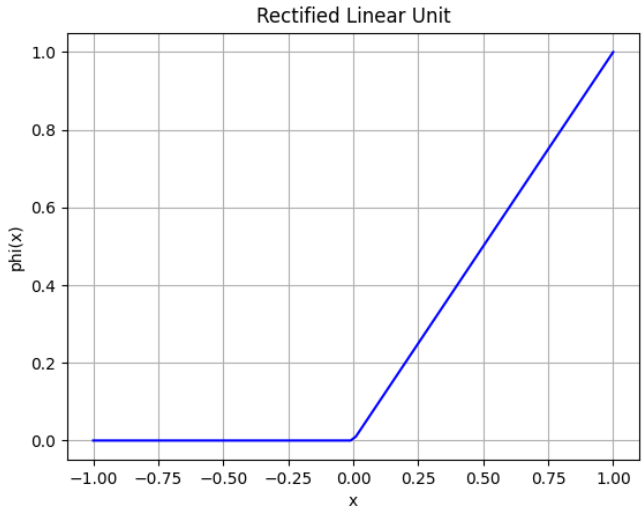

Abbildung 2.8: Die Rectified Linear Unit Aktivierungsfunktion.

beschrieben wird. Ein Vorteil gegenüber der Sigmoid-Funktion besteht darin, dass die ReLU-Funktion sich durch ihre Linearität effizienter berechnen lässt. Bei positiven Eingaben verhält sich diese Funktion wie die Identitätsfunktion.

Dadurch tritt das Problem der verschwindenden Gradienten nicht bei der ReLU-Funktion auf. Jedoch kann es beim Training zum dying ReLU Problem kommen, welches ein ähnliches Problem wie das der verschwindenden Gradienten darstellt [35]. Man spricht vom dying ReLU Problem, wenn die Gewichte eines ReLU-Neurons so angepasst werden, dass dieses jeden Input auf den Wert Null abbildet [35]. Beim Training führt dies dazu, dass die berechnete partielle Ableitung immer Null beträgt. Dadurch ist der Gradient der Gewichte des Neurons ebenfalls null und es finden keine Veränderungen der Gewichte mehr statt. Falls das Neuron in diesem Zustand angekommen ist, dann ist es schwierig, es aus diesem Zustand zu befreien [14] [38]. Dies kann dazu führen, dass das gesamte Netzwerk aufhört zu lernen [38].

Eine Möglichkeit, dies zu vermeiden, ist der Einsatz der leaky-ReLU Funktion. Im Gegensatz zur normalen ReLU Funktion bildet diese negative Werte nicht auf ´0´ ab, sondern auf einen sehr kleinen Wert abhängig vom Input. Dadurch beträgt die Ableitung der Funktion für negative Inputs nicht mehr null und somit kann auch das dying ReLU Problem nicht mehr auftreten [14] [36] [47]. Allerdings ist die Funktion aufgrund der Abbildung auf sehr kleine negative Werte anfällig für das Problem der verschwindenden Gradienten [14].

2.2.1.4 Training von neuronalen Netzen

Ziel des Trainings ist es, die Parameter eines Netzes so zu verändern, dass sie einen Input auf einen gewünschten Output abbilden können. Dazu benötigt man ein Trainingsdatenset, welches gelabelte Daten enthält [42]. Zusätzlich braucht man ein Testdatenset mit Einga-

ben, welche nicht zum Trainieren des Netzes verwendet werden. Dieser Datensatz dient dazu, die im Training gelernten Parameter anzuwenden und die Performance des Netzes zu testen [42].

Das Training an sich besteht darin, ein Optimierungsproblem zu lösen. Dazu wird eine Kostenfunktion, auch loss-function genannt, aufgestellt. Sie berechnet den Unterschied zwischen dem Output des Netzes und dem tatsächlichen Label der Inputdaten. Ziel ist es, diese Kostenfunktion zu minimieren, indem die Parameter des Netzes angepasst werden [22].

Ein einfaches Beispiel für eine Kostenfunktion ist die mean squared error Loss Funktion:

$$MSE(x, x') = \frac{1}{n}\sum_{i=1}^{n}(x_i - x'_i)^2 \tag{2.9}$$

Für einen berechneten Outputvektor x' und einen Labelvektor x errechnet diese Kostenfunktion die mittlere quadratische Abweichung. Je ähnlicher x und x' sind, desto kleiner ist der berechnete MSE-Wert. Daraus folgt, dass durch die Minimierung der Kostenfunktion, welches durch das Anpassen der Parameter des Netzwerks geschieht, x' sich immer weiter den Label annähert.

Die gewählte Kostenfunktion hängt von dem jeweiligen zu lösenden Klassifikationsproblem ab. In dieser Arbeit soll durch ein neuronales Netz ein Attributsvektor berechnet werden, dessen Einträge angeben, ob die Inputdaten ein gewisses Attribut aufweisen. Da es sich bei diesem Vorhaben um eine Multi-Label Klassifikation handelt, bei der ein Attribut entweder vorhanden ist oder nicht, bietet es sich an, die binäre Cross-Entropy-Loss Funktion als Kostenfunktion zu verwenden [13] :

$$BCE(x, x') = -\frac{1}{n}\sum_{i=0}^{n} x_i * log(x'_i) + 1 - x_i * log(1 - x'_i) \tag{2.10}$$

Auch in dieser Formel steht x' für den berechneten Outputvektor des Netzes und x für den Labelvektor.

Ein oft verwendeter Optimierungsalgorithmus zum Lösen des Optimierungsproblems ist das Gradientenabstiegsverfahren, auch gradient descent genannt. Dabei handelt es sich um einen iterativen Algorithmus, welcher ein lokales Minimum in einer differenzierbaren Funktion berechnet. Daher ist es wichtig, dass differenzierbare Aktivierungsfunktionen in den Neuronen gewählt werden.

Anhand eines einzelnen Neurons lässt sich ein einzelner Optimierungsschritt wie folgt erklären:

Gegeben sei eine Kostenfunktion $Loss(x)$, ein Input x und ein Gewichtungsvektor w eines einzelnen Neurons mit einer differenzierbaren Aktivierungsfunktion. Zuerst wird der Gradient der Kostenfunktion $\nabla loss(x)$ berechnet. Dazu bestimmt man für jedes Gewicht des Gewichtungsvektors w die partielle Ableitung:

$$\frac{\delta Loss(x)}{\delta w_i} \tag{2.11}$$

Der Gradient der Kostenfunktion $\nabla loss(x)$ ist der Vektor mit allen partiellen Ableitungen der Gewichte:

$$\nabla loss(x) = \begin{pmatrix} \frac{\delta Loss(x)}{\delta w_1} \\ \vdots \\ \frac{\delta Loss(x)}{\delta w_n} \end{pmatrix} \tag{2.12}$$

Dieser Vektor gibt die Richtung des steilsten Anstiegs für die Loss Funktion an. Nach Berechnung des Gradienten folgt das Anpassen des Gewichtungsvektors w. Die Anpassung lässt sich durch folgende Formel beschreiben:

$$w_{neu} = w_{alt} - \gamma \times \nabla loss(x) \tag{2.13}$$

Dabei steht γ für eine zusätzliche anpassbare Lernrate. Ziel ist es, die Kostenfunktion zu minimieren. Da der Gradient der Kostenfunktion in die Richtung des steilsten Anstiegs zeigt, wird dieser von den Gewichten subtrahiert. Dadurch findet ein Optimierungsschritt in die Richtung des steilsten Abstiegs der Kostenfunktion statt. Zusätzlich wird der Gradient mit einer Lernrate γ multipliziert. Dieser Faktor nimmt Einfluss auf die Größe der Optimierungsschritte. Wird die Lernrate zu groß gewählt, kann es dazu kommen, dass gute lokale Minima der Funktion in einem Optimierungsschritt übersprungen werden oder die Lernschritte anfangen zu oszilieren [46]. Bei einer zu kleinen Learningrate werden jedoch die Optimierungsschritte zu klein und es dauert sehr lange, bis man ein lokales Minimum erreicht [46]. Zusätzlich besteht das Problem, dass man ein schlechtes lokales Minimum findet und auf diesem festhängt, da die Schrittgröße des Optimierungsschritts nicht ausreicht, um ein besseres Minimum zu finden. In Abbildung 2.9 ist das Problem nochmals grafisch dargestellt.

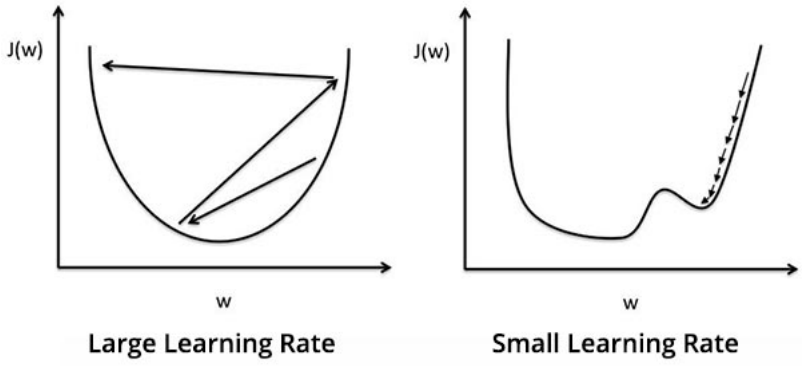

Abbildung 2.9: Links: Annäherung an ein Minimum mit einer zu großen Lernrate. Rechts: Annäherung an ein Minimum mit einer zu kleinen Lernrate. Bild entnommen aus [2]

Zusätzlich kann die Lernrate während des Trainings durch Updateregeln modifiziert werden. Eine dieser Updateregeln ist die Root Mean Square Propagation (RMSProp) [23]. Die Idee hinter RMSProp ist, die Lernrate für jedes Gewicht abhängig von dem gleitenden Durchschnitt der quadrierten Gradienten in einem Lernschritt anzupassen [6]. Dieser gleitende Durchschnitt berechnet sich für einen Lernschritt wie folgt [23] [6]:

$$E[g^2]_{neu} = \beta E[g^2]_{alt} + (1-\beta)(\nabla loss(x))^2 \tag{2.14}$$

Der standardmäßige Default-Wert für den Prameter β, welcher auch von dem Erfinder Geoff Hilton vorgeschlagen wird, beträgt ´0.9´ [23] [6]. Formel 2.13 zeigt die standardmäßige Anpassung der Gewichte beim Gradientenabstiegsverfahren ohne zusätzliche Updateregeln. Mit der RMSProp Updateregel sieht die Anpassung der Gewichte wie folgt aus:

$$w_{neu} = w_{alt} - \frac{\gamma}{\sqrt{E[g^2]_{neu}}} \times \nabla loss(x) \tag{2.15}$$

Ziel von RMSProp ist es, die Größe der Optimierungsschritte abhängig von dem Gradienten zu verändern [23]. Für große Gradienten und damit großen Steigungen in der Funktion werden die Optimierungsschritte kleiner. Dadurch lässt sich die Chance für das Überspringen von lokalen Minima senken. Mit kleiner werdenden Gradienten werden die Optimierungsschritte größer. Somit können flache Bereiche der Funktion beim Training schneller durchquert werden.

Da MLPs aus mehreren Schichten bestehen, gestaltet sich die Anwendung dieses Optimierungsalgorithmus schwieriger. Daher wird in Netzen mit mehreren Schichten zusätzlich der Backpropagation-Algorithmus beim Training angewandt. Dieser wurde erstmals im Jahre 1974 vorgestellt [56] und macht sich die Kettenregel zunutze. Ziel ist es, den Gradienten der Kostenfunktion für ein beliebiges Gewicht des Netzwerks zu bestimmen. In Bezug auf Formel 2.11 lässt sich dies darstellen durch:

$$\frac{\delta Loss(x)}{\delta w_{ij}^l} \tag{2.16}$$

Dabei steht w_{ij}^l für das i-te Gewicht des j-ten Neurons im l-ten Layer. Der Algorithmus arbeitet rekursiv vom Ergebnis aus zum Anfang des Netzes. Durch die Kettenregel lässt sich die Abhängigkeit des Ergebnisses durch das Gewicht w_{ij}^l wie folgt darstellen:

$$\frac{\delta Loss(x)}{\delta w_{ij}^l} = \frac{\delta z_j^l}{\delta w_{ij}^l} \frac{\delta a_j^l}{\delta z_j^l} \frac{\delta Loss(x)}{\delta a_j^l} \tag{2.17}$$

$$z_j^l = \sum_{k=0}^{n} a_k^{l-1} w_{kj}^l \tag{2.18}$$

$$a_j^l = \phi(z_j^l) \tag{2.19}$$

z_j^l beschreibt den Wert des j-ten Neurons im l-ten Layer und a_j^l den Output des j-ten Neurons im l-ten Layer. Der Wert des Neurons ergibt sich aus dem Skalarprodukt des Outputs des vorherigen Layers a_k^{l-1} mit den Gewichten des Neurons w_{kj}^l. Der Output a eines Neurons berechnet sich durch Anwendung der Aktivierungsfunktion $\phi(x)$ auf den Wert des Neurons.

Formel 2.17 zeigt die Berechnung der partiellen Ableitung für ein Gewicht eines Neurons im letzen Layer des Netzes. Im Folgenden wird zur Erklärungszwecken die Formel zur Berechnung einer partiellen Ableitung für ein Gewicht aus einem vorherigen Layer schrittweise aufgebaut.

Will man die partielle Ableitung für ein Gewicht aus einem vorherigen Layer bestimmen, wird zuerst die Berechnungsfunktion des Neurons z_j^l nicht mehr nach den Gewichten abgeleitet, sondern nach dem Output des vorherigen Neurons. Für den Output des Neurons k des vorherigen Layers sieht die Formel dafür wie folgt aus:

$$\frac{\delta Loss(x)}{\delta a_k^{l-1}} = \frac{\delta z_j^l}{\delta a_k^{l-1}} \frac{\delta a_j^l}{\delta z_j^l} \frac{\delta Loss(x)}{\delta a_j^l} \tag{2.20}$$

Zusätzlich muss beachtet werden, dass der Output eines Neurons aus der vorherigen Schicht mehrere Neuronen in der letzten Schicht und somit auch die Kostenfunktion mehrmals beeinflusst. Daher muss die partielle Ableitung von jedem Neuron aus dem letzen Layer in Bezug auf das Neuron aus dem Layer davor berücksichtigt werden. Dadurch ergibt sich die Formel:

$$\frac{\delta Loss(x)}{\delta a_k^{l-1}} = \sum_{j=0}^{n_l} \frac{\delta z_j^l}{\delta a_k^{l-1}} \frac{\delta a_j^l}{\delta z_j^l} \frac{\delta Loss(x)}{\delta a_j^l} \tag{2.21}$$

Dabei steht n_l für die Anzahl der Neuronen innerhalb des Layers l. Durch Anwendung der Kettenregel kann man nun aus diesem Ausdruck die Formel für die partielle Ableitung für ein Gewicht aus dem vorherigen Layer aufstellen:

$$\frac{\delta Loss(x)}{\delta w_{ik}^{l-1}} = \frac{\delta z_k^{l-1}}{\delta w_{ik}^{l-1}} \frac{\delta a_k^{l-1}}{\delta z_k^{l-1}} \frac{\delta Loss(x)}{\delta a_k^{l-1}} \tag{2.22}$$

Nach diesem Schema lassen sich die partiellen Ableitungen und somit auch die Gradienten für jedes Gewicht innerhalb eines neuronalen Netzes rekursiv vom Ergebnis aus bestimmen.

2.3 Convolutional Neural Networks

Im Jahre 1959 erforschten David H. Hubel und Torsten N. Wiesel den visuellen Kortex von Katzen[26]. Sie entdeckten zwei verschiedene Arten von Zellen, welche verschiedene Teile

zur Erkennung beitrugen. Die S-Zellen, auch Simple-cells genannt, dienten dazu, Features zu erkennen. Die C-Zellen oder auch Complex-cells sind für die Lokalisierung der Features zuständig. Basierend auf den Erkenntnissen zum Aufbau der reziptiven Feldern von Katzen entwarf Kunihiko Fukushima das ´Neocognitron´ [19]. Es beinhaltet S-Layer und C-Layer basierend auf den S- und C-Zellen von Hubel und Wiesel. Das Neocognitron stellt die Basis der Convolutional Neural Networks dar, die heutzutage verwendet werden [32].

Basierend auf früheren Arbeiten entwickelten Yann LeCun et al. im Jahre 1989 das erste Convolutional Neural Network (CNN) mit dem Namen LeNet [31]. Das Netz kann handgeschriebene Postleitzahlen erkennen und wurde mithilfe des Gradientenabstiegsverfahrens und Backward-Propagation trainiert. Dieser Ansatz wurde zur Grundlage für den Bereich der Computervision.

Als CNN werden neuronale Netze bezeichnet, welche ein sogenanntes Convolutional Layer enthalten [22]. Eine genauere Betrachtung der Convolutional Layer findet in Abschnitt 2.3.1 statt. Weiterhin enthalten die meisten CNNs zusätzliche Pooling-Layer, welche im Abschnitt 2.3.2 erläutert werden. Durch ihren Aufbau eignen Convolutional Neural Networks sich gut, um mehrdimensionale Daten mit Rasterstrukturen, wie Zeitreihendaten oder Bilder, zu verarbeiten [22].

2.3.1 Convolutional Layer

Ein Convolutional Layer erhält als Eingabe einen n-dimensionalen Input und berechnet mithilfe eines Kernels eine Featuremap. Der Kernel ist dabei eine mehrdimensionale Gewichtsmatrix, deren Gewichte ebenfalls durch ein Training des Netzes lernbar sind. Die einzelnen Elemente der Featuremap werden durch eine mathematische Faltung (convolution) des Inputs mit dem Kernel berechnet [22]. Um eine vollständige Featuremap zu berechnen, wird der Kernel wie eine Linse Schritt für Schritt über den Input geführt. Die Stelle, an der sich Kernel und Input überlappen, wird dabei rezeptives Feld genannt. In jedem Schritt wird ein Feature durch die Faltung des Inputs mit dem Kernel berechnet.

Die Berechnung eines Features für einen zweidimensionalen Input I mit einem zweidimensionalen Kernel K der Größe $m \times n$ lässt sich durch folgende Formel darstellen [22]:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{2.23}$$

In Abbildung 2.10 ist eine Featuremap mit Berechnung eines Convolutional Layers beispielhaft dargestellt.

Zusätzlich gibt es zwei Parameter für Convolutional Layer. Der erste Parameter ist der Stride. Dieser gibt an, wie groß die Schritte des Kernels über den Input sind. Durch einen höheren Stride ist es möglich, die Größe der Featuremap zu verkleinern und Überlappungen der reziptiven Felder der einzelnen Features zu verhindern.

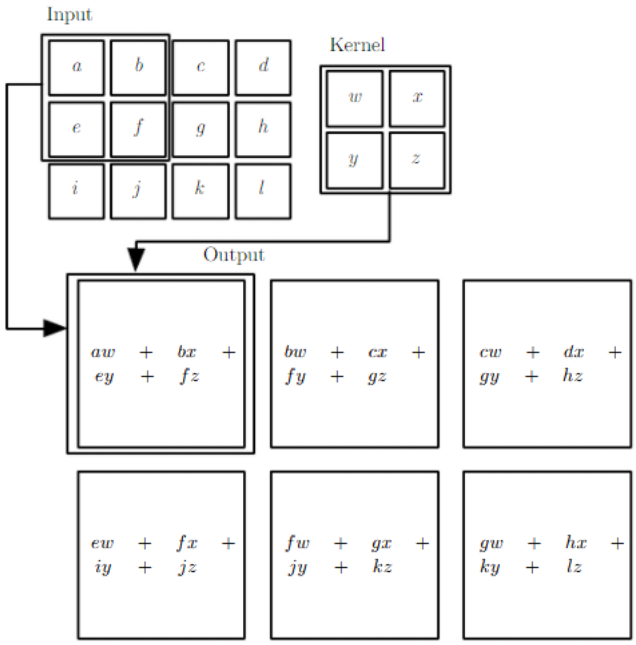

Abbildung 2.10: Beispielhafte Berechnung einer Featuremap durch Faltung. Bild entnommen aus [22].

Der zweite Parameter ist das Padding. Das Padding ermöglicht es, die Größe der untersuchten Matrix zu erhöhen, indem am Rand der zu untersuchenden Matrix neue Zeilen und Spalten eingefügt werden. So wird verhindert, dass das Ergebnis des Layers schrumpft, wenn ein Kernel größer als 1x1 verwendet wird. Würde das Ergebnis durch jeden Convolutional Layer kleiner werden, wäre die Tiefe des Netzes durch die Größe der Eingabe beschränkt.

Bei einem Kernel mit n Spalten und m Zeilen (n $>$ 1, m $>=$ 1) und einem Stride $<$ n tritt ein Informationsverlust an den Randfeldern, besonders an den Ecken der Matrix, auf. Zum Beispiel betrachtet ein 3x3 Kernel mit Stride 1 auf einem beliebigen Input ein Eckfeld nur einmal, ein Randfeld dreimal und ein mittleres Feld neunmal. Indem mittels Padding jeweils zwei Spalten und Zeilen eingefügt werden, werden auch die Ecken neunmal betrachtet.

Dabei gibt es mehrere Arten des Paddings. Ein Beispiel dafür ist das Zero-Padding. Dabei wird der Rand des Inputs mit Nullen aufgefüllt [45].

Convolutional Layer besitzen einige Vorteile gegenüber den bisher vorgestellten Fully Connected Layern. Einer der Vorteile ist die sparse connectivity der einzelnen Neuronen. In einem MLP ist jedes Neuron einer Schicht mit jedem Neuron der vorherigen Schicht verbunden. Jede Verbindung besitzt dabei ein eigenes Gewicht und wird in die Berechnung des Wertes des Neurons einbezogen. In Convolutional Layern sind die Neuronen allgemein nur teilweise miteinander verbunden (Ausnahme ist der Spezialfall, dass die Inputgröße der

Kernelgröße entspricht). Dies liegt daran, dass zur Berechnung des Outputs eines Neurons nur die Werte des rezeptiven Feldes betrachtet werden. Da dadurch weniger Parameter für die Berechnung benötigt werden, lassen sich die Outputs der einzelnen Neuronen effizienter und auch schneller als bei einem Fully Connected Layer berechnen [22].

Ein weiterer Vorteil von Convolutional Layern ist das sogenannte Parameter Sharing. In einem Standard MLP besitzt jede Verbindung genau ein Gewicht, welches exakt einmal zur Berechnung eines Neurons verwendet wird. In einem Convolutional Layer werden die gelernten Parameter an jeder Position des Kernels wiederverwendet. Daher wird die Parameteranzahl auf die Größe des Kernels beschränkt. Dadurch lässt sich der Speicherplatz für ein Convolutional Layer erheblich senken [22].

2.3.2 Pooling Layer

Nach einem Convolutional Layer folgt meist ein Pooling Layer. Pooling Layer werden verwendet, um die Größe der berechneten Featuremap zu reduzieren [45] und um eine robustere Erkennung bei kleinen Translationen der Features zu gewährleisten [22]. Daher spricht man beim Pooling meist von einer Downsampling Operation. Ähnlich wie bei einem Convolution Layer wird hierbei eine Linse über den Input bewegt. In den überlappenden Bereichen wird jedoch, anders als bei den Convolutional Layern, keine Faltung angewandt. Am beliebtesten ist das sogenannte Max-Pooling [45]. Dabei wird in dem überlappenden Bereich das Maximum gewählt und ausgegeben (siehe Abbildung 2.11). Die Wahl der Größe und des Strides eines Pooling Kernels kann die Performance des Netzes stark beeinflussen. In der Regel werden 2x2 Kernel mit einem Stride von 2 gewählt. Dadurch lässt sich die Größe der Featuremap um 75% verringern. Falls der Kernel größer gewählt wird als ein 3x3 Kernel, lässt sich jedoch meist beobachten, dass die Performance des Netzes stark nachlässt [45].

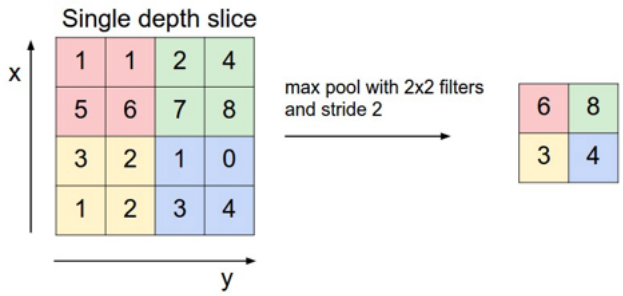

Abbildung 2.11: Beispielhafte Anwendung der Max-Pooling Operation. Abbildung entnommen aus [12].

2.3.3 Fully Connected Layer

Nach der Extraktion der Features durch Convolutional und Pooling Layer werden meist Fully Connected Layer (FC-Layer) für die Interpretation der extrahierten Features verwendet. Als Fully Connected Layer wird dabei ein Layer bezeichet, welches genau so aufgebaut ist wie ein Layer eines MLPs. Jedes Neuron der vorherigen Schicht ist mit jedem Neuron des Fully Connected Layer verbunden, daher verarbeitet jedes Neuron immer die gesamte Eingabe und berechnet auf dieser Basis die Ausgabe.

Das Ergebnis eines Fully Connected Layers ist dementsprechend ein Vektor, in dem jeder Eintrag unter Einbeziehung aller Inputwerte berechnet wurde. Da eine berechnete Featuremap eines Convolutional Layers meist mehrdimensional ist, verfällt dadurch ihre mehrdimensionale Struktur. Zusätzlich werden alle lokalen Features, welche durch vorherige Convolutional Layer extrahiert wurden, zu einem globalen Ergebnis zusammengefasst. Daher werden beim Einsatz eines Fully Connected Layers die strukturellen Informationen der Featuremaps verworfen [33].

Grundsätzlich können Fully Connected Layer auch als Spezialfälle von Convolutional Layern betrachtet werden [59]. Wählt man für ein Convolutional Layer einen Kernel mit der Größe der Inputdaten, ist es möglich, ein Convolutional Layer mit den Eigenschaften eines Fully Connected Layers zu erzeugen.

2.3.4 Temporal Convolutional Networks

Ein Temporal Convolutional Network, kurz TCN, ist eine Spezialform eines Convolutional Networks. Es eignet sich hervorragend, um Zeitreihendaten zu analysieren.

Ein TCN unterscheidet sich von einem normalen CNN in zwei Punkten. Die erste Besonderheit besteht darin, dass bei einer Inputsequenz mit einer beliebigen Länge eine Outputsequenz mit der gleichen Länge erzeugt wird [1].

Verarbeitet das TCN also eine Inputsequenz der Länge x, so besitzt die Outputsequenz ebenfalls die Länge x. In Bezug auf die Human Activity Recognition ist die Länge einer Inputsequenz definiert durch die zeitliche Dauer der zu analysierenden Sensordaten. Bei der Verarbeitung durch ein normales CNN kann die Outputsequenz eine beliebige Länge haben.

Der zweite Punkt ist, dass die Faltungen eines TCNs kausal sind. Daher gibt es keinen Informationsfluss von der Zukunft in die Vergangenheit [1]. Das bedeutet, dass zu einem gegebenen Zeitpunkt t nur die vorherigen Werte in die Featureberechnungen eingehen. Zur Verdeutlichung dient Abbildung 2.12.

In der Praxis wird daher meist ein Padding in der Größe der Kernelbreite-1 am Anfang des Inputtensors angefügt. Dadurch ist im ersten Schritt das erste Element des Inputs das letze Element im reziptiven Feld, das betrachtet wird. Weiterhin wird durch die zusätzlichen Featureberechnungen die Größe des Outputs der des Inputs angepasst.

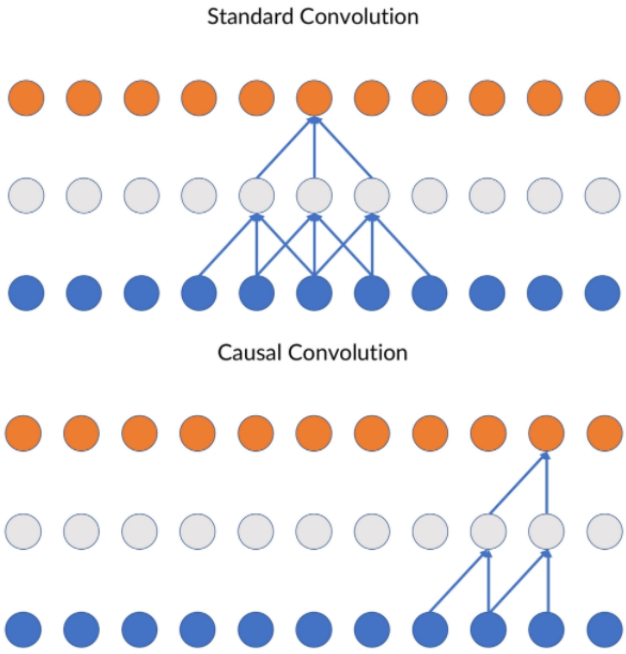

Abbildung 2.12: Oben: Standardfaltung, Unten: kausale Faltung. Bilder entnommen aus [28].

2.3.5 Fully Convolutional Networks

Ein Fully Convolutional Network, kurz FCN, ist eine weitere Spezialform eines Convolutional Networks. Die Besonderheit besteht darin, dass FCNs im Gegensatz zu CNNs nur Convolutional Layer und Pooling Layer enthalten. Daher ist ein FCN ein CNN ohne Fully Connected Layer. Anders als Convolutional Layer haben FC-Layer die Eigenschaft, räumliche Informationen zu verwerfen [33].

Zur Analyse der extrahierten Features in der Featureebene bietet es sich an, Convolutional Layer mit einem 1x1 Kernel zu verwenden. Durch Verwendung eines 1x1 Kernels besteht die Möglichleit, die Daten nur in der Featuredimension zu analysieren, ohne dabei die räumliche Struktur der Daten zu verändern [33]. So ist es möglich, ein Fully Connected Layer entlang der Featureebene zu emulieren, der für jeden, durch den Kernel betrachteten Datenpunkt, alle Features analysiert.

Zur Klassifikation der Daten in einem Fully Convolutional Network bieten sich verschiedene Outputlayer an. Zur Berechnung einer einzelnen Klasse oder mehrere Attribute wird der Output durch einen Poolinglayer erzeugt. Basierend auf der Form des Outputs werden hier mit einem Poolinglayer die berechneten Informationen aus den vorangegangenen Convolution Layern zu einem Ergebnis zusammengefasst.

Vor allem im Bereich der semantischen Segmentierung, in der es darum geht, jedes Pixel in einem Bild zu klassifizieren, werden oft Deconvolution Layer als Outputlayer verwendet. Ähnlich wie bei einem Convolution Layer wird auch hier ein Kernel über den Input geführt und das rezeptive Feld zur Berechnung des Outputs verwendet. Jedoch beträgt der Stride

eines Deconvolution Layers meist einen Wert kleiner als Eins. Dadurch wird der Output größer als der Input und es kann in einem neuronalen Netz ein Upsampling der Daten vollzogen werden. Ein weiterer Vorteil eines Fully Convolutional Networks besteht darin, dass ein Netzwerk ohne FC-Layer Inputs mit variablen Größen verarbeiten kann [33].

2.4 Auto-Encoder-Decoder

Ein Auto-Encoder-Decoder ist eine besondere Netzwerkarchitektur, welche es ermöglicht, Gewichte aus ungelabelten Daten zu lernen [55]. Wie bereits herausgestellt, ist das Gradientenabstiegsverfahren mit dem Backwardpropagation Algorithmus nur anwendbar, wenn eine Kostenfunktion berechnet werden kann. Die Berechnung einer Kostenfunktion basiert darauf, die Entfernung zwischen dem berechneten Ist-Wert und dem vorgegebenen Soll-Wert zu ermitteln. Der Soll-Wert war bis jetzt immer ein Label, welches vorher manuell ermittelt werden musste.

Ein Auto-Encoder-Decoder besteht aus zwei Teilen. Die erste Komponente ist der Encoder. Dies ist das eigentliche Netz, welches trainiert werden soll. Der Output des Encoders ist eine komprimierte Repräsentation des Inputs, welche durch die Gewichte des Encoders berechntet wurden. Die komprimierte Repräsentation dient als Input für den zweiten Teil des Netzwerks, den Decoder. Er ist ein weiteres neuronales Netz, das die Aufgabe hat, den Input aus der komprimierten Darstellung zu rekonstruieren.

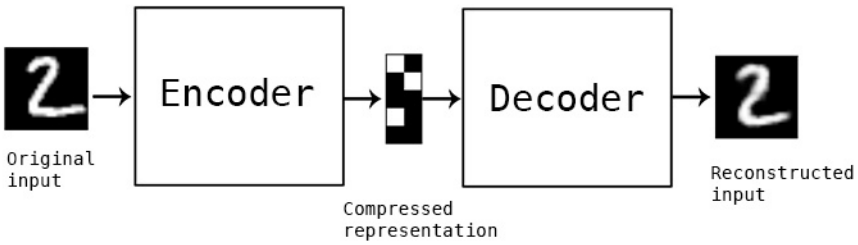

Abbildung 2.13: Darstellung eines Auto-Encoder-Decoders. Abbildung entnommen aus [18].

Ein Vorteil dieser Struktur ist, dass dadurch eine Kostenfunktion berechnet werden kann, in welcher der Soll-Wert durch den Input selbst gegeben ist. Daher benötigen die Trainingsdaten zum Lernen der Gewichte kein Label mehr, welches vorher ermittelt werden muss [55].

Um das Auto-Encoder-Decoder Netzwerk zur Klassifikation einzusetzen, wird nur der Encoder benötigt. Daher wird der Decoder nach dem Lernen der Gewichte abgekoppelt. Ein Nachteil dieser Trainingsmethode ist jedoch, dass der berechnete Output des Encoders eine komprimierte Repräsentation des Inputs ist, der für einen Menschen nicht direkt interpretierbar ist. Das Problem lässt sich allerdings durch ein nachträgliches kürzeres Trai-

ning mit gelabelten Daten für den gewünschten Output beheben, oder man verwendet die komprimierte Repräsentation für eine Klassifikation mithilfe der nearest-neighbor-search.

Kapitel 3

Verwandte Arbeiten

In diesem Kapitel werden verwandte Arbeiten vorgestellt, auf denen diese Arbeit basiert.

Zuerst wird der gesamte Vorgang der HAR detailliert dargestellt und die in anderen Arbeiten verwendeten Methoden und Metriken beschrieben.

Der nächste Abschnitt befasst sich grundlegend mit der Verwendung von Temporal Convolutional Networks als Klassifizierer in der Human Activity Recognition. Dazu wird die Arbeit von Yang et al. [57] betrachtet, in welcher ein Temporal Convolutional Network als Klassifizierer auf mehrkanaligen Zeitreihendaten einsetzt wird. Weiterhin wird ein Temporal Convolutional Network präsentiert, das zur Human Activity Recognition auf dem LARa-Datenset eingesetzt wird.

In Abschnitt 3.3 wird eine Arbeit vorgestellt, die sich mit Fully Convolutional Networks in der semantischen Segmentierung von Bildern auseinandersetzt. Diese befasst sich mit der Umwandlung von bestehenden Klassifizierungs-CNNs zu FCNs, zusätzlich wird eines der angepassten FCNs um eine Skip-Architektur erweitert.

Abschnitt 3.4 betrachtet eine Arbeit zu Auto-Encoder-Decoder Netzwerken in der Human Activity Recognition.

Im letzten Abschnitt des Kapitels werden nochmals die wichtigsten Elemente aus den genannten Arbeiten herausgestellt, die für diese Bachelorarbeit von großer Bedeutung sind.

3.1 Human Activity Recognition

Für das Erkennen von menschlichen Aktivitäten gibt es in der Forschung viele verschiedene Ansätze. In Abschnitt 2.1 wurde bereits die grundlegende Vorgehensweise zur Erstellung eines Klassifizierers beschrieben. Im Folgenden werden die Methoden jedes Schrittes von Arbeiten aus dem Bereich der Human Activity Recognition näher erläutert.

3.1.1 Datenerfassung

Zur Entwicklung eines Klassifizierers werden Daten benötigt, welche es zu analysieren gilt. Die Autoren aus [21] stellten bei ihren Experimenten fest, dass die Befestigung der Beschleunigungssensoren an der Taille und an der Brust am besten zur Erkennung von Körperhaltungen und Stürzen geeignet sind. Durch die Kombination mehrerer Sensoren an verschiedenen Körperstellen lassen sich die aufgezeichneten Daten, und damit auch die Ergebnisse des entwickelten Klassifizierers, verbessern [10]. Grundsätzlich hängt die Platzierung und die Anzahl der verwendeten Sensoren auch davon ab, welche Aktivitäten man erkennen möchte.

Eine vorgeschlagene Möglichkeit aus [34] ist, die Bewegungsdaten mithilfe eines einzelnen, an der Taille befestigten, triaxialen Beschleunigungssensor aufzuzeichnen. Die Autoren aus [37] verwenden hingegen einen einzelnen triaxialen Beschleunigungssensor am Handgelenk. Die Messwerte des Sensors werden zu einem Beschleunigungsvektor zusammengefasst, dessen Betrag zur Repräsentation der Daten verwendet wird. Sowohl [34] als auch [37] entschieden sich für ihre Sensorauswahl und Sensorposition aufgrund von Komfort und Alltagstauglichkeit mit dem Hintergrund, sportliche Aktivitäten im natürlichen Alltag aufzuzeichnen.

Die Autoren aus [15] benutzten zur Klassifikation Bewegungsdaten aus einem Motion Capturing System, bei dem die Position von Makern am Körper der Probanden im dreidimensionalen Raum visuell aufgezeichnet werden. Eine Körperposition wird kodiert, indem für jeden Marker die Rotation zum nachfolgenden Marker angegeben wird [15]. Bestimmt man einen Knoten als Wurzelknoten, kann von diesem aus die Körperposition als Menge von Rotationen beschrieben werden. Eine Bewegung definiert sich durch aufeinanderfolgende Körperpositionen in der zeitlichen Dimension. Daher entsteht für die Kodierung einer Bewegung eine Matrix, die alle Körperpositionsmengen für verschiedene Zeitpunkte enthält.

Die Marker am Körper werden aufgrund ihrer Position in fünf Bereiche unterteilt, um eine detailiertere Beschreibung der Bewegungen zu ermöglichen [15]. Bei den Bereichen handelt es sich um die vier Gliedmaßen und den Torso des menschlichen Körpers. Dadurch lassen sich die Bewegungen eines einzelnen Körperteils durch eine Teilmatrix beschreiben, welche nur die Werte der Marker enthält, die sich auf diesem Körperteil befinden [15].

Die Datenerfassung in der Human Activity Recognition ist ein aufwändiger Prozess, der viel Zeit in Anspruch nimmt. Die Daten werden daher oft in Form von Datensets der Öffentlichkeit zur Verfügung gestellt. Ein Beispiel dafür ist das LARa-Datenset aus [43], welches im Rahmen dieser Arbeit verwendet wird. Eine detailliertere Beschreibung des Datensets findet sich in Abschnitt 5.1.

3.1.2 Preprocessing

Das Preprocessing beschäftigt sich mit der Aufbereitung der erfassten Messwerte. Während der Datenerfassung kann es dazu kommen, dass Sensoren ausfallen oder ein zufälliges Rauschen in den Sensordaten auftritt. Bei der Kombination von unterschiedlichen Sensoren muss zudem beachtet werden, dass diese unterschiedliche Eigenschaften besitzen und sich Messskalen und Abtastraten unterscheiden können.

Während ausgefallene Sensoren zu unbrauchbaren Daten führen, können unterschiedliche Messbereiche durch Normalisierung auf einen gemeinsamen Wertebereich projiziert werden. Um den Signalabstand zu verbessern und das Rauschen zu eliminieren, existieren verschiedene Ansätze.

Die Autoren aus [27] benutzen einen gleitenden Mittelwert Filter, der drei aufeinanderfolgende Abtastwerte gleichzeitig betrachtet, um zufälliges Rauschen in den Sensordaten zu minimieren und das Signal zu glätten. Der durch den Filter berechnete Wert $x'(t)$ zum Zeitpunkt t des Sensors kann mit folgender Formel berechnet werden:

$$x'(t) = \frac{x(t) + x(t-1) + x(t-2)}{3} \tag{3.1}$$

Dabei steht x für das ursprüngliche Signal, auf das der Filter angewendet wird.

In [34] wird ein Tiefpassfilter von null bis fünf Hertz verwendet, um die signal-to-noise ratio (Signal-Rausch-Abstand) der aufgenommenen Daten eines triaxialen Beschleunigungssensors zu verbessern. Der Signal-Rausch-Abstand ist ein Maß für das Verhältnis der Signalstärke zur Stärke des Rauschens.

3.1.3 Segmentierung

Nach dem Preprocessing erhält man bereinigte Sensorkurven, die alle Bewegungen einer Versuchsperson über einen längeren Zeitraum beinhalten. Um die Klassifizierung vorzubereiten, müssen die Daten in Segmente aufgeteilt werden, die jeweils eine zu klassifizierende Bewegung enthalten.

Ein beliebter Ansatz zur Segmentierung ist das sliding-window Verfahren [48]. Dabei wird ein Fenster fester Größe mit einer festgelegten Schrittgröße über die Daten geschoben [4]. Die Größe des Fensters gibt an, wie lang die extrahierte Datensequenz ist und die Schrittgröße um welchen Wert das Fenster für die nächste Extraktion versetzt wird.

In der Arbeit [43] wird dieses Verfahren verwendet, um die Daten des LARa-Datensets zu segmentieren, bevor sie durch ein neuronales Netz klassifiziert werden (siehe Abschnitt 3.2.2).

3.1.4 Feature Extraktion

Aktivitäten besitzen charakteristische Eigenschaften (Features), die sie unterscheidbar machen. Diese Features werden benutzt, um die Aktivitäten zu klassifizieren. Bei der Verwendung von Deep Learning Ansätzen werden die Features automatisch mittels neuronaler Netze extrahiert. Für andere Klassifizierungsmethoden werden sie durch statistische Methoden berechnet.

In [34] werden insgesamt 19 Features verwendet, die in drei Bereiche unterteilt werden. Der erste Bereich sind die zeitlichen Features. Eines davon ist z.B. die Standardabweichung [34]. Dieses Feature soll Aufschluss über die Energieintensität einer Bewegung geben [34].

Die zweite Art von Features geht aus der Frequenz der Daten hervor. Da einige Bewegungen ein gleiches Energielevel besitzen, vergleichen die Autoren zusätzlich die Periodizität der Signale des Sensors [34]. Dazu wird die Entropie des normalisierten Wirkleistungsspektrums berrechnet [34].

Der letzte Bereich sind die räumlichen Features. Eine Orientierungsänderung des Körpers wird in der Arbeit von [34] als Veränderung der Gravitationsdaten in den drei Sensorkurven definiert. Dieses Merkmal soll dafür genutzt werden, die Stärke der Änderung der Körperhaltung innerhalb einer Aktivität zu beschreiben [34]. Andere verwendete räumliche Features betrachten die horizontalen und vertikalen Trägheitsbeschleunigungen der Sensoren [34].

Die Autoren aus [30] benutzten ein Android Handy mit einem eingebauten Beschleunigungssensor zur Datenerfassung, welches die Beschleunigungsdaten dreidimensional erfasste.

Als Features wurden die durchschnittliche Beschleunigung (getrennt für einzelne Richtungen und zusätzlich gemittelt über alle drei Dimensionen), Standardabweichung, mittlere absolute Abweichung und die Zeit zwischen den Spitzenwerten für jede der drei Achsen betrachtet [30]. Zusätzlich wurden durch Binning der Daten zu jeder Achse 30 weitere Features berechnet. Beim Binning wird in der Arbeit [30] zuerst der Wertebereich der betrachteten Daten bestimmt, welcher dann in gleich große Teile (´Bin´) aufgeteilt wird [30]. Jeder gemessene Wert wird nun einem ´Bin´ zugeordnet. Als Resultat erhält man einen Überblick, wie oft bestimmte Werte aus einem Wertebereich in den Daten enthalten sind [30]. In [30] werden für jede Aktivität pro Achse 200 Werte betrachtet, die in 10 ´Bins´ unterteilt werden.

Nach der Feature Extraktion folgt die Feature Reduktion. Je mehr Features aus den Daten extrahiert werden, desto schwieriger gestaltet es sich, aus diesen eine Klasse zu berechnen [16]. Daher hilft eine Feature Reduktion dabei, die Performance bei der Klassifikation zu verbessern.

In [34] wird zur Reduktion der Features die Hauptkomponentenanalyse verwendet. Durch Entfernen von redundanten Informationen, kann durch eine Hauptkomponenten-

analyse die Dimensionalität der Features ohne großen Informationsverlust gesenkt werden [34].

3.1.5 Klassifikation

Der letzte Schritt ist die Klassifizierung der Bewegung. Eine der einfachsten Methoden zur Klassifizierung ist der K-Nearest-neighbours-Algorithmus, bei dem die Ähnlichkeit von unbekannten Bewegungsdaten zu bekannten Bewegungen berechnet wird [7]. Hierzu wird eine Abstandsmetrik, wie etwa der euclidische Abstand, verwendet, um die ´K´ ähnlichsten bekannten Bewegungen zu ermitteln [7]. Die Bewegung, die am häufigsten unter diesen ´K´ nächsten Nachbarn zu finden ist, wird den unbekannten Bewegungsdaten zugeordnet [7]. Diese Klassifizierungsmethode wird unter anderem von den Autoren aus [5] und [52] verwendet, um menschliche Bewegungen zu klassifizieren.

Eine weitere Methode zur Klassifikation ist der Einsatz von neuronalen Netzen. Ein Vorteil dabei ist, dass die Features direkt aus den Daten erlernt und nicht per Hand bestimmt werden müssen [48].

3.2 Temporal Convolutional Networks in der Human Activity Recognition

Die Herausforderung in der Human Activity Recognition besteht darin, eine Bewegung basierend auf aufgezeichneten Bewegungsdaten, wie z.B. Sensordaten, zu erkennen. In der Forschung werden, mit wachsender Beliebtheit, deep-learning Ansätze für diese Aufgabe eingesetzt. Die dazu verwendeten neuronalen Netze unterscheidet man in ihrer Architektur und der Art, wie sie eingehende Daten analysieren.

Während mit manchen Netzen versucht wird, den Sensordaten lediglich eine einzige Aktivität zuzuordnen [57], zielen andere Ansätze darauf ab, die Sensordaten auf bestimmte Attribute zu analysieren [43], oder sogar ein Set von Aktivitäten aus den Inputdaten zu erkennen[55].

CNN Architekturen besitzen bei der Analyse von Zeitreihendaten zwei entscheidene Vorteile gegenüber anderen Strukturen [58]. Der erste Vorteil ist ihre Fähigkeit, lokale Abhängigkeiten zu erkennen. Bei der Analyse einer Sensorkurve ist dies eine wichtige Eigenschaft, da lokale Korellationen zwischen den Sensorwerten betrachtet werden können [58]. Der zweite wichtige Vorteil gegenüber anderen Strukturen ist die Skaleninvarianz. Dadurch lässt sich die Erkennung robuster gestalten gegenüber den, von Mensch zu Mensch unterschiedlichen, individuellen Ausführungsarten einer Aktivität [58]. Zusätzlich zu diesen beiden Vorteilen werden bei Temporal Convolutional Networks die Features auf der Zeitachse berechnet[43].

Durch die kausalen Faltungen eines TCNNs wird sichergestellt, dass für die Featureberechnungen zu keinem Zeitpunkt Informationen aus der Zukunft verwendet werden [1] (siehe Abbildung 2.12).

3.2.1 Deep Convolutional Neural Networks auf mehrkanaligen Zeitreihendaten

Yang et al. [57] beschreiben in ihrer Arbeit ein TCN zur Klassifikation von Zeitreihendaten in der Human Activity Recognition. Die Architektur ihres Netzes ist in Abbildung 3.1 abstrakt dargestellt.

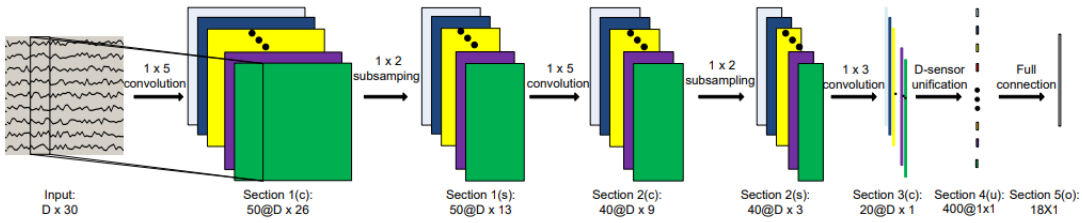

Abbildung 3.1: TCN aus der Arbeit von Yang et al. [57]. ´D´ steht für die Anzahl der Sensoren. Die Buchstaben in den Klammern stehen für die verschiedenen Operationen. ´c´ = convolution, ´s´ = subsampling/pooling, ´u´ = unification (Zusammenfassen der Featuremaps aus vorherigem Layer), ´o´ = output. Bild entnommen aus [57].

Die Autoren beschreiben die Architektur mit Hilfe von fünf Sektionen.

Die ersten beiden Sektionen bestehen aus einem Convolution Layer mit einer RELU Aktivierungsfunktion, gefolgt von einem Max-Pooling Layer mit einem Normalisierungs-Layer. Das Max-Pooling und die Faltungen der Convolution Layer werden dabei auf der zeitlichen Ebene angewandt. Die Normalisierungs-Layer sollen die Werte der berechneten Featuremaps aus dem vorherigen Layer angleichen.

Die dritteSektion des Netzes besteht aus einem Convolution Layer mit einer RELU Aktivierungsfunktion und einem Normalisierungs-Layer. In diesem Teil wird auf einen weiteren Pooling Layer verzichtet, da die zeitliche Dimension des berechneten Outputs des Convolution Layers die Größe ´1´ beträgt und somit ein Max-Pooling auf der zeitlichen Ebene keinen Einfluss mehr haben würde.

Würde ein Max-Pooling auf der zeitlichen Ebene durchgeführt werden, würde jeweils immer nur ein einzelner Wert betrachtet, da nur ein Wert auf der zeitlichen Ebene existiert. Das Maximum von nur einem Wert wäre dann trivialerweise immer der Wert selbst.

Die nächste Sektion beginnt mit einem Fully Connected Layer, welcher dazu dient, die berechneten Featuremaps aus Sektion drei zu vereinen. Auf den FC-Layer folgen eine weitere RELU-Aktivierungsfunktion und ein Normalisierungs-Layer.

Der Outputlayer in Sektion fünf ist ein Fully Connected Layer gefolgt von einer Softmax-Aktivierungsfunktion, die zur Klassifikation der Daten dienen.

Die Inputdaten des Netzes werden mithilfe des sliding-window Ansatzes auf eine feste Länge segmentiert, bevor sie analysiert werden. Diese ist abhängig von der Abtastrate, mit der die Daten aus dem jeweiligen Datenset aufgenommen wurden.

Das Netz wurde mithilfe der Cross-Entropy-Kostenfunktion und dem Backpropagation-Algorithmus sowohl auf dem Opportunity Datenset, als auch auf dem Hand Gesture Datenset trainiert und getestet [57]. Dabei beträgt die Länge der extrahierten Segmente mit dem sliding-window Ansatz beim Opportunity Datenset ´30´ Einheiten und beim Hand Gesture Datenset ´32´ Einheiten [57].

Als Baseline für die Experimente dienten vier verschiedene Klassifizierungsansätze. Die ersten beiden sind eine Support Vector Machine (SVM) und ein K-Nearest-Neighbor Ansatz [57]. Beide Ansätze basieren auf der Arbeit von [9] und arbeiten auf den rohen Zeitreihendaten der Datensets. Daher findet bei diesen Ansätzen keine Vorverarbeitung der Eingabedaten statt.

Zur Klassifizierung in der dritten Baseline wird ebenfalls der K-Nearest-Neighbor Ansatz gewählt [57]. Der Unterschied ist, dass vorher die Inputdaten, wie beim vorgestellten TCN, mit dem sliding-window Ansatz segmentiert wurden. Als Inputdaten für den Klassifizierer dienen aus jedem erstellten Segment der berechnete Mittelwert und die Varianz über die Abtastwerte der Merkmale [57].

Für den letzten Klassifizierungsansatz werden die Daten ebenfalls zuerst mit dem vorgestellten sliding-window Ansatz segmentiert [57]. Über die Abtastwerte der Merkmale jedes einzelnen Segments wird der Mittelwert berechnet, welcher als Input für ein Deep Belief Network dient. Als Klassifizierer dienen sowohl ein K-Nearest-Neighbor Ansatz als auch ein Multi Layer Perzeptron [57].

Zusätzlich wendeten die Autoren in einem zweiten Schritt das Smoothing-Verfahren, beschrieben in [8], auf die Ergebnisse an, um zu sehen, ob dadurch eine weitere Verbesserung des Netzes möglich ist.

Beim Smoothing des Outputs wird für jeden Zeitpunkt ein Filter mit einer vordefinierten Länge über die Daten gelegt [57]. Dieser ermittelt das am häufigsten vorkommende Label in einem gegebenen Umkreis für jeden Zeitpunkt. Weicht das Label des Zeitpunktes von diesem Label ab, wird es an das am häufigsten vorkommende Label angepasst [8] [57].

Als Metrik dienten die Genauigkeit, der mittlere F-Score und der normalisierte F-Score.

Auf beiden Datensätzen erreichten die Autoren aus [57] mit ihrem entwickelten TCN die höchste Performance in jeder Metrik. In der Genauigkeit konnte das Netz sogar 5% mehr als alle anderen getesteten Ansätze erreichen. Durch das nachträgliche Smoothing wurden weitere kleine Verbesserungen erzielt.

3.2.2 Temporal Convolutional Network des LARa-Datensets

Basierend auf der Arbeit von Yang et al. [57] entwickelten die Autoren des Artikels [43] ein TCN zur Klassifizierung von Aktivitäten des LARa-Datensets. Im Gegensatz zu dem Netz von Yang et al. [57] besitzt dieses weder Pooling- und Normalisierungs-Layer noch einen Fully Connected Layer zum Zusammenfügen der berechneten Featuremaps.

Das TCN aus Artikel [43] besteht aus vier Convolutional Layern, welche die zeitlichen Features aus den Daten extrahieren. Darauf folgen zwei Fully Connected Layer, die Korrelationen zwischen den Features ermitteln. Abhängig von der gewünschten Repräsentation des Ergebnisses, bilden unterschiedliche Outputlayer den Abschluss des Netzes. Für die Attributrepräsentation ist dies ein Fully Connected Layer mit einer Sigmoid-Funktion und für die Klassenrepräsentation ein Fully Connected Layer mit einer Softmax-Funktion. Um die Gefahr des Overfittings zu verringern, sind nach den beiden 128-FC-Layern jeweils ein Dropoutlayer implementiert. Diese fehlen jedoch in der Abbildung 3.2.

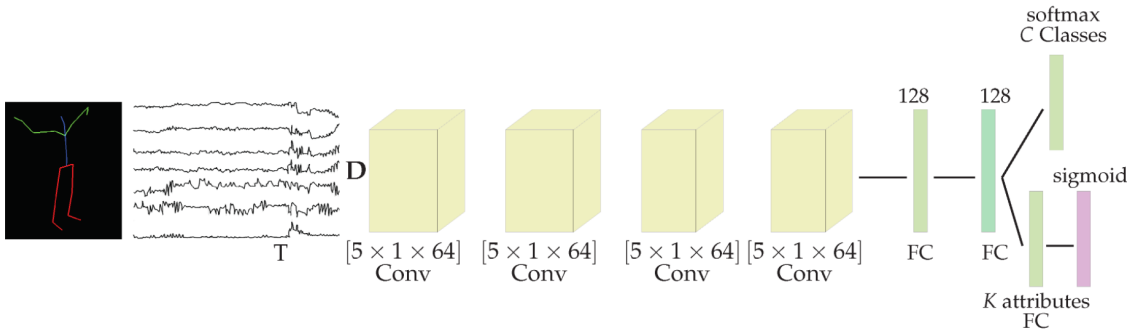

Abbildung 3.2: TCN des LARa-Datensets. Bild entnommen aus [43].

Vor dem Training wurden die Sequenzen jedes Sensors auf einen Wertebereich zwischen null und eins normalisiert und mit einem gaußschen Rauschen versehen, welches die Ungenauigkeit eines Sensors simulieren soll [43]. Als Parameter für das gaußsche Rauschen wählten die Authoren $\mu = 0$ und $\sigma = 0.01$ [43].

Die Segmente wurden mithilfe des sliding-window Verfahren extrahiert. Für die Sensordaten des verwendeten Motion Capturing Systems (MoCap) besitzt jedes Fenster eine Länge von 200 Zeiteinheiten und eine Höhe von 126 Einheiten, welche der Anzahl der Sensor channel entspricht. Die extrahierten Segmente wurden in ein Trainingsset, ein Validiationset und ein Testset unterteilt. Mithilfe des Validationsets wird während des Trainings ein ´Early Stopping´ vollzogen [43]. Dabei wird das Training nach einer definierten Zeit gestoppt und das Netz auf dem Validationset getestet. Wenn dabei bessere Ergebnisse als beim vorherigen Testen erzielt wurden, wird eine Kopie der Gewichte abgespeichert. Am Ende des Trainings wird die beste Kopie ausgegeben. Dieses Verfahren soll verhindern, dass das Netz überangepasst wird und die Trainingsdaten auswendig lernt [22].

Das Netz wurde mithilfe des Batch Gradientenabstiegsverfahrens mit der RMSProp-Updateregel und einer Learningrate von 0.00001 trainiert. Als Kostenfunktion für das Training dient die binäre Cross-Entropy-Loss Funktion [43].

Als Metrik auf dem LARa-Datenset diente die Präzision, die over-all accuracy, der gewichtete F1-Score und der Recall. Auf dem Datenset erreichte das Netz eine over-all accuracy von 75,15% mit der Attributsrepräsentation und 68,88% mit der Klassendarstellung [43]. Das TCN konnte einen gewichteten F1-Score von 73,62% mit den Attributen und 64,43% mit der Klassendarstellung erreichen [43].

3.3 Fully Convolutional Network in der semantischen Segmentierung

Bei der semantischen Segmentierung handelt es sich um ein Analyseverfahren von Bildern. Anders als bei der Objekterkennung wird in der semantischen Segmentierung jedem Pixel eines Bildes ein Wert zugeordnet, der angibt, was das Pixel darstellt. Daher ist dieser Ansatz genauer als andere Verfahren der Objekterkennung, die lediglich eine einzige Klasse für ein Bild oder Boundingboxen um erkannte Objekte in einem Bild berechnen. Dieses Verfahren wird eingesetzt, wenn präzise Bildanalysen benötigt werden wie beispielsweise beim autonomen Fahren.

Bei der semantischen Segmentierung mittels neuronaler Netzwerke werden klassischerweise CNNs verwendet, weil diese sich gut dazu eignen, mehrdimensionale Daten mit Rasterstrukturen zu verarbeiten (siehe auch Abschnitt 2.3). Die Arbeit [33] beschreibt den Einsatz von FCNs in der semantischen Segmentierung mit dem Ziel, die Erkennungsrate der einzelnen Pixel zu verbessern und Eingaben variabler Größe zu verarbeiten.

Dazu passen die Autoren herkömmliche Klassifizierungs-CNNs an und verbessern die erreichte Performance eines Netzes mithilfe einer Skip-Architektur (siehe Abschnitt 3.3.2). Am Ende demonstrieren sie ihre Ergebnisse auf dem PASCAL VOC 2011 Datensatz [33].

3.3.1 Anpassung der Klassifizierungs-CNNs

Die Umwandlung der Klassifizierungs-CNNs zu FCNs erfolgt in zwei Schritten. Zuerst werden die Fully Connected Layer ersetzt, weil sie eine feste Inputgröße benötigen. Man ersetzt sie durch Convolutional Layer, deren Kernel die Größe 1x1 besitzen. Damit wird die Funktionalität eines Fully Connected Layers für jedes Pixel entlang der Featureebene simuliert, ohne dabei die Struktur der Daten zu verändern [33]. Für jedes Pixel werden dementsprechend alle Features analysiert und aus ihnen ein neuer Wert für das Pixel berechnet.

Der Output des Layers besitzt daher die selbe Größe wie der Input, kann sich jedoch in der Anzahl der berechneten Featuremaps unterscheiden [33].

Da die Gewichte eines Convolutional Layers im Kernel gespeichert und die gleichen Gewichte für jeden Filterschritt über den Input verwendet werden (siehe Abschnitt 2.3.1), ist es ebenfalls möglich, Inputs mit verschiedenen Größen zu verarbeiten. Lediglich die Anzahl der Filterschritte über den Input, und damit auch die Größe des Outputs, ändern sich.

Als Zweites müssen die Klassifizierungslayer der CNNs angepasst werden. Das letzte Layer eines Klassifizierungsnetzes gibt aus, zu welcher Klasse das Bild gehört. Für die semantische Segmentierung ist die gewünschte Ausgabe eine Pixelkarte mit klassifizierten Pixeln.

Daher ersetzen die Autoren aus [33] das letzte Layer durch ein Convolutional Layer mit einem 1x1 Kernel und 21 Filtern gefolgt von einem Deconvolution Layer, welches den Output upsampled. Es werden 21 Filter im letzten Convolutional Layer verwendet, um für jede der 21 Klassen des PASCAL VOC 2011 Datensatzes einen Score zu berechnen.

Alle anderen Schichten können mit ihren Parametern unverändert aus den bestehenden Netzen übernommen werden [33]. Da die Netze und damit auch die Gewichte vorher auf die Klassifikation von Bildern trainiert wurden, werden sie einem Finetuning unterzogen. Dabei werden diese nochmals speziell mit Daten aus der semantischen Segmentierung trainiert, um die Parameter auf die neue Aufgabe anzupassen [33].

Mit dem besten angepassten Netz (FCN-VGG16-Net) erreichten sie nach dem Finetuning state-of-the-art Ergebnisse auf dem PASCAL VOC 2011 Datensatz[33]. Das Netz erreichte eine mean IU (mean intersection over union) von 59.4 auf einem Subset der Validierungsdaten [33].

3.3.2 Skip-Architektur basierend auf dem VGG16-Net

Basierend auf dem angepassten VGG16-Net entwickelten die Autoren eine Architektur, die es erlaubt, Informationen aus früheren Layern in die Entwicklung des Outputs einfließen zu lassen. Durch das Einbeziehen der Informationen aus vorhergehenden Schichten soll das Netz lokale Vorhersagen unter Berücksichtigung der globalen Struktur des Inputs treffen [33]. Dabei stellte sich heraus, dass die direkte Entnahme der vorhergehenden Bildinformationen aus einem früheren Pooling Layer zu einer schlechteren Performance führt. Deshalb fügten die Autoren des Papers ein 1x1 Convolution Layer nach dem Pooling Layer als Bindeglied ein, welcher zusätzliche Klassenvorhersagen, basierend auf dem Output des Pooling Layers, berechnen und somit die Performance verbessern soll [33]. Dadurch ließ sich nach dem Training des Netzes eine verbesserte mean IU erreichen.

3.4 Deep Auto-Encoder-Set Network zur Human Activity Recognition

Die Autoren des Papers [55] beschreiben eine weitere Netzarchitektur zum Lösen von HAR-Klassifizierungsproblemen. Es handelt sich dabei um ein Auto-Encoder-Decoder Netzwerk namens Auto-Set zur Erkennung von Sets bestehend aus Aktivitäten. Anders als bei einer einfachen Klassifizierung mit nur einem Ausgabewert, wird hier eine Menge von erkannten Aktivitäten ausgegeben. Das liegt daran, dass durch die Zusammenstellung der Trainingsdaten mit dem sliding-window Ansatz eine Inputsequenz mehrere aufeinanderfolgende Aktivitäten enthalten kann [55]. Der Trainingsprozess des Auto-Sets besteht aus zwei Schritten. Zuerst wird der Encoder des Netzes mithilfe eines Decoders trainiert. In einem zweiten Schritt wird der Decoder durch einen Klassifizierungskopf ersetzt und darauf trainiert, die gewünschte Darstellung zu berechnen. Durch den ersten Trainingsschritt des Decoders soll das Netzwerk selbst eine Funktion zur Berechnung von nützlichen Features finden, die mit dem zweiten Schritt verfeinert werden soll [55]. Dies soll unter anderem dazu dienen, dass für den Trainingsprozess des Netzes weniger gelabelte Trainingsdaten benötigt werden, die in der HAR sehr aufwendig zu generieren sind.

3.4.1 Aufbau des Auto-Set Netzwerks

Der Encoder des Netzwerks besteht aus einem Convolutional Neural Network, welches auf einem CNN aus der Arbeit [44] basiert. Das Netz besteht insgesamt aus vier Convolutional Layern gefolgt von zwei Fully Connected Layern und einem Outputlayer. Dabei bilden das letzte Fully Connected Layer und das Outputlayer den Klassifizierungskopf des Netzes, welcher nach dem Training mit dem Decoder zum Einsatz kommt. Der Decoder besteht aus einer gespiegelten Version des Encoders. Anders als beim Encoder bestehen die gespiegelten Convolution Layer des Decoders jedoch aus Deconvolution Layern. Das Ziel des Decoders besteht darin, den Input aus den berechneten Ergebnisdaten des Encoders zu rekonstruieren. In Abbildung 3.3 ist das gesamte Netzwerk dargestellt.

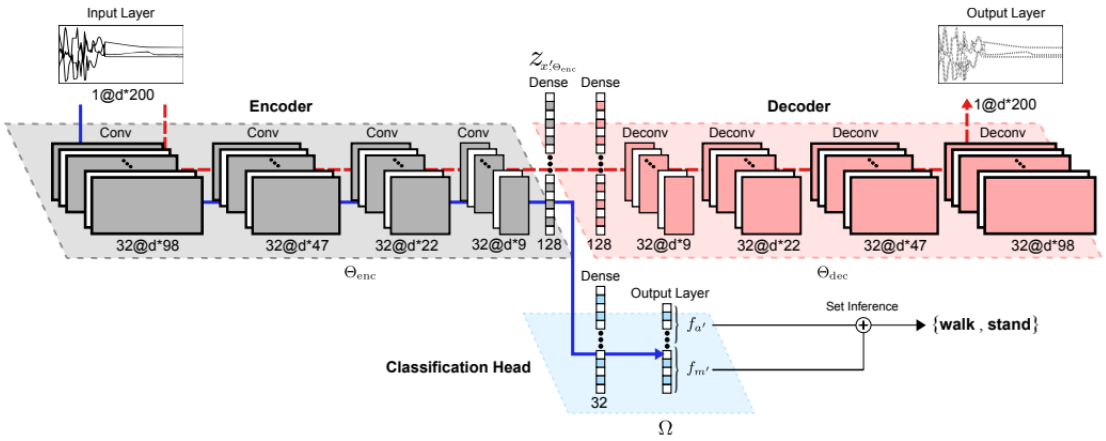

Abbildung 3.3: Aufbau der Auto-Set Netzwerkarchitektur. Grau: Encoder. Rot: Für den ersten Trainingsschritt verwendeter Decoder. Blau: Klassifizierungskopf. Bild entnommen aus [55].

3.4.2 Kostenfunktion beim Training des Encoders

Für das unsupervised Training des Encoders wird eine Kostenfunktion benötigt. Anders als beim supervised Training kann dafür kein vorher bestimmtes Label verwendet werden [55]. Die gewählte Kostenfunktion der Entwickler des Auto-Set Netzwerks besteht aus zwei Funktionen. Die erste Funktion ist die Berechnung der Zwischendarstellung z_x durch den Encoder:

$$z_x = f_{enc}(x, W_{enc}) \tag{3.2}$$

Der Input des Netzes ist durch x und die Gewichte des Encoders durch W_{enc} dargestellt. Die zweite Funktion ist die Berechnung des rekonstruierten Inputs x' durch den Decoder. Diese Funktion lässt sich darstellen durch:

$$x' = f_{dec}(z_x, W_{dec}) \tag{3.3}$$

Zusammengesetzt besteht die gewählte Kostenfunktion aus dem Vergleich des Inputs mit der Rekonstruktion des Decoders.

$$loss(x, W_{enc}, W_{dec}) = ||x - x'||^2 \tag{3.4}$$

Diese gilt es im Zuge des unsupervised Trainings mit dem Decoder zu minimieren [55].

3.4.3 Ergebnisse

Um die Überlegenheit eines Trainings mittels Decoder zu zeigen, verwendeten die Autoren insgesamt vier Netze. Das erste ist das ursprüngliche CNN aus [44] (im Folgenden Deep-BCE genannt), welches auf herkömmliche Art trainiert wurde. Das zweite Netz namens

Auto-BCE entspricht dem Deep-BCE, jedoch wurde es über einen Decoder trainiert [44]. Das dritte Netz ist das von ihnen neu entwickelte Auto-Set, welches über einen Decoder trainiert wurde. Das vierte Netz trägt den Namen Deep-Set. Es entspricht dem Auto-Set, wurde aber auf herkömmliche Art trainiert (siehe Abbildung 3.4). Unterschiede zwischen den ´BCE-Netzen´ und den ´Set-Netzen´ bestehen in der Kostenfunktion zur Bewertung des Trainings und bei der Ausgabe [44]. Während die ´BCE-Netze´ die binäre Cross-Entropy-Loss-Funktion (BCE) als Kostenfunktion und eine Multi-Label-Darstellung einer einzigen erkannten Bewegung als Ausgabe verwenden, wurde für die Set-Netze eine eigene Bewertungsfunktion entwickelt. Sie geben ein Set der erkannten Aktivitäten aus [44].

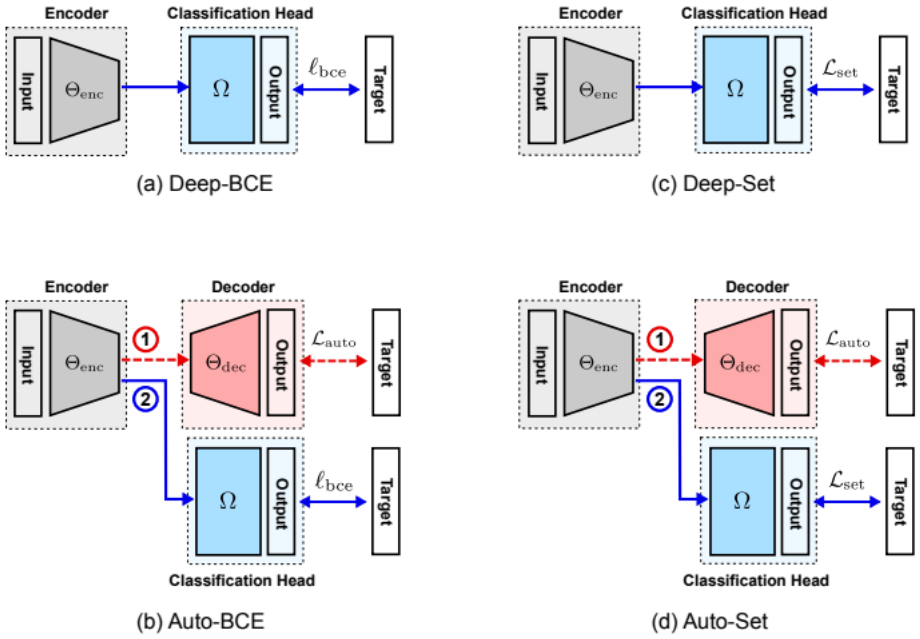

Abbildung 3.4: Überblick über die verwendeten Netzwerkarchitekturen. Bild entnommen aus [55].

Sowohl bei der exact match ratio (MR) als auch bei der Präzision, dem Recall und dem F1-Score konnten bei beiden Netzarchitekturen höhere Ergebnisse durch einen Trainingsschritt mit einem Decoder erzielt werden.

3.4.4 Diskussion

Yang et al. [57] beschreiben ein neuronales Netz, welches automatisch Features aus mehrkanaligen Zeitreihendaten extrahiert und diese Daten zu einer einzigen Klasse zuordnen. Jedoch wäre die Erkennung einer Attributrepräsentation besser geeignet, um HAR Daten zu klassifizieren. Ein großes Problem in der Human Activity Recognition sind imbalancierte Datensets, in denen bestimmte Aktivitäten überproportional oft vorkommen. Durch

Verwendung einer Attributrepräsentation, welche die Aktivitäten auf einer höheren Ebene beschreibt, kann die Darstellung eines Attributs aus oft vorkommenden Klassen erlernt und auf ähnliche Aktivitäten, welche nicht so häufig im Datenset vorkommen, übertragen werden [50]. Die Autoren des Artikels [43] verwenden für das Training ihres TCNs eine solche Attributrepräsentation, die zusätzlich auf eine einzelne Klasse reduziert werden kann. Da durch Ermittlung eines Attributvektors, der einer Klasse zugeordnet werden kann, eine bessere Klassifikation erzielt wird [50], wird das Verfahren auch in dieser Bachelorarbeit angewandt.

Für das Auto-Encoder-Set Network aus [55] wird ein Decoder verwendet, um ein Netz zur Klassifikation von Aktivitäten zu trainieren. Jedoch beschreiben die Autoren aus [55] nur eine Methode, die das Netz vorab mit einem Decoder trainiert, und erst in einem zweiten separaten Schritt mit den gelabelten Daten. In dieser Arbeit wird betrachtet, zu welchem Ergebnis ein gleichzeitiges Training mit gelabelten Daten und Decoder führt.

Kapitel 4

Methoden

Im vorherigen Kapitel wurden verwandte Arbeiten vorgestellt, auf denen diese Arbeit aufbaut. Der erste Abschnitt in diesem Kapitel handelt primär von der Erstellung eines Fully Convolutional Networks basierend auf der Arbeit [43]. Dabei soll versucht werden, mithilfe der Erkenntnisse aus der semantischen Segmentierung, die Performance des Netzes zu verbessern.

Im zweiten Abschnitt dieses Kapitels wird beschrieben, wie das Fully Convolutional Network zu einem Auto-Encoder-Decoder weiterentwickelt wurde. Der Encoder des Netzwerks soll aus der Fully Convolutional Network Architektur bestehen, welche in Abschnitt 4.1 beschrieben ist. Aufgrund der Eigenschaft, dass das konstruierte Fully Convolutional Network keine Fully Connected Layer besitzt und somit die zeitlichen Informationen der Daten nicht verliert, soll versucht werden, mit dem Decoder die Daten aus der berechneten Zwischendarstellung zu rekonstruieren. Wie in der Arbeit [55] soll der Decoder dabei einer gespiegelten Version des Encoders entsprechen.

4.1 Erstellen des Fully Convolutional Networks

Das Temporal Convolutional Network aus [43] stellt ein state-of-the-art Netzwerk zur Erkennung von Aktivitäten dar (siehe Abbildung 4.1). Dieses TCN verwendet jedoch Fully Connected Layer, die aufgrund ihres Aufbaus die räumliche Struktur der Daten verwerfen. Durch Austausch der Fully Connected Layer durch Convolutional Layer soll erreicht werden, dass die räumliche Struktur der Daten erhalten bleibt.

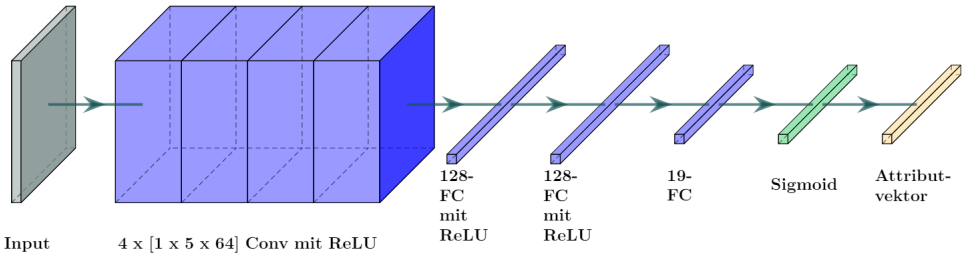

Abbildung 4.1: Aufbau der Temporal Convolutional Network Architektur aus [43].

4.1.1 Aufbau des Netzes

Die Architektur des zu entwickelnden Netzes basiert auf dem TCN aus [43]. Allgemein ist bei der Konstruktion des Netzes die Struktur der Eingabedaten zu berücksichtigen. Hier ist das die Anzahl der Sensorchannel (H) und die zeitliche Dauer der Sensorsequenzen (B). Die konkreten Angaben für das, in dieser Arbeit verwendete, LARa-Datenset können Abschnitt 5.1 entnommen werden.

Die ersten vier Convolutional Layer bleiben weiterhin zum Extrahieren von zeitlichen Features bestehen. Jedes der Convolution Layer besitzt einen 1x5-Kernel, der auf der zeitlichen Dimension Features extrahiert. Pro Convolutional Layer werden dabei 64 Featuremaps berechnet.

Als nächstes folgen zwei weitere Convolutional Layer mit einem Kernel der Größe 1x1. Diese analysieren die Daten nur auf der Feature-Ebene, ohne die Höhe und Breite des erhaltenen Tensors zu verändern. Als Vorbild dienen die 1x1 Convolution Layer aus [33], die dort erfolgreich als Ersatz für Fully Connected Layer eingesetzt wurden (siehe Abschnitt 3.3).

Durch die Verwendung von 1x1 Convolution Layern bleibt die räumliche Struktur der Daten wie in [33] bestehen. Für die verwendeten Zeitreihendaten bedeutet der Erhalt der räumlichen Struktur ein Bewahren der zeitlichen Dimension der Daten. Durch Verwendung eines 1x1 Kernels wird für jeden betrachteten Datenpunkt ein Fully Connected Layer entlang der Featureebene emuliert. Die Anzahl der Filter und damit die Anzahl der berechneten Featuremaps der beiden Convolutional Layer soll im Laufe der durchgeführten Experimenten in Abschnitt 5.4.2.1 bestimmt werden.

Die Dropoutlayer nach dem fünften und sechsten Layer bleiben bestehen, um dem Overfitting des Netzwerks vorzubeugen.

Der Attributvektor enthält 19 Attribute, daher wird durch einen weiteren 1x1 Convolution Layer die Anzahl der Featuremaps auf diese Anzahl reduziert. Für dieses Layer wird als Aktivierungsfunktion die Sigmoidfunktion verwendet, welche alle Tensorwerte in den Bereich zwischen null und eins abbildet. Der nachfolgende Average Pooling Layer betrachtet mit seinem Kernel den gesamten Tensor und bildet auf den 19-stelligen Attributsvektor ab.

In Abbildung 4.2 ist der Aufbau des Netzes dargestellt.

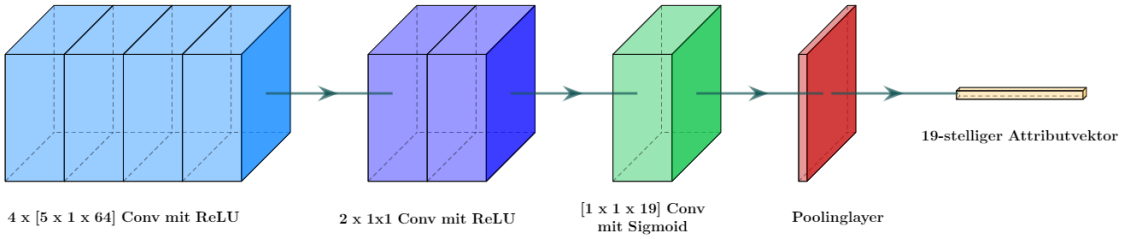

Abbildung 4.2: Aufbau der Fully Convolutional Network Architektur. Hellblau: Vier Convolutional Layer mit 5x1 Kernel und 64 berechneten Featuremaps. Dunkelblau: 1x1 Convolutional Layer. Grün: 1x1 Convolutional Layer zur Berechnung der Attribute mit Sigmoid Aktivierungsfunkton. Rot: Poolinglayer zur Berechnung eines Attributvektors. Gelb: 19-stelliger Attributvektor.

4.1.2 Training

Vor dem Training werden die Eingabedaten mit einem gaußschen Rauschen versehen, um, analog zu [43], Sensorungenauigkeiten zu simulieren. Danach findet das zweistufige Training statt. Die erste Stufe besteht aus einem Training von 20 Epochen auf einem Trainingsdatenset. Das Netz wird mit Hilfe des Batch Gradientenabstiegsverfahrens und der RMSProp-Updateregel trainiert. Eine geeignete Learningrate für das Training wird experimentell bestimmt. Zusätzlich wird ebenfalls das Early Stopping Verfahren aus [43] während des Trainings angewendet. Als Kostenfunktion dient die binäre Cross-Entropy-Loss Funktion:

$$Loss = -\frac{1}{n}\sum_{i=0}^{n} y_i * log(x_i) + 1 - y_i * log(1 - x_i) \tag{4.1}$$

Dabei steht n für die Größe des Outputvektors, x_i für das i-te Attribut im berechneten Outputvektor und y_i für das dazugehörige Element aus dem gelabelten Attributvektor der Daten.

Die zweite Stufe besteht aus dem Finetuning der letzten drei Layer des Netzes. Dazu werden die Gewichte der ersten vier Convolutional Layer, welche zum Extrahieren von zeitlichen Features dienen, während des Finetunings nicht weiter angepasst. Dadurch sollen die letzen Layer, welche zum Interpretieren der extrahierten Features dienen, eine genauere Interpretation erlernen.

4.2 Erstellen des Auto-Encoder-Decoders

Ziel des Fully Convolutional Networks ist es, die Performance des Temporal Convolutional Networks aus [43] zu verbessern, indem zusätzlich räumliche Features in den 1x1 Convolu-

tional Layern berechnet werden. Da die räumliche Struktur von Zeitreihendaten wichtige Informationen beinhalten, sollen sie nun zusätzlich auch gezielt gelernt werden.

Bislang ist jedoch das Training des Netzes nur darauf ausgelegt, bestimmte Attribute basierend auf den eingespeisten Sensordaten zu erlernen. Daher soll zusätzlich ein Decoder entwickelt werden, der die Sensorkurven basierend auf einer Zwischendarstellung des Encoders rekonstruiert. Dadurch lässt sich die Kostenfunktion erweitern, sodass sie speziell die räumliche Struktur der Daten berücksichtigt.

4.2.1 Aufbau des Netzes

Der Auto-Encoder-Decoder besteht aus zwei Teilen. Der erste Teil ist der Encoder, der basierend auf den Inputdaten eine Zwischendarstellung berechnet. Dafür wird, das in Abschnitt 4.1 beschriebene, Fully Convolutional Netzwerk eingesetzt. Als Zwischendarstellung wird die Tensorgröße 19xBxH festgelegt. Hierbei steht 'B' für die Größe der zeitlichen Dimension und 'H' für die Anzahl der Sensorchannel des Tensors, die durch den letzten 1x1 Convolution Layer berechnet werden.

Der zweite Teil des Netzwerks ist der Decoder, der in der Architektur einem gespiegelten Encoder ähnelt. Als Input erhält der Decoder den berechneten Tensor der Größe 19xBxH.

Der erste Layer bildet ein Deconvolution Layer, welches aus der Eingabe einen weiteren Tensor der Größe XxBxH berechnet. Sowohl die zeitliche Dimension 'B' als auch die Anzahl der Sensorchannel 'H' des Tensors verändern sich dabei nicht. Die Anzahl der berechneten Featuremaps 'X' ist äquivalent zu denen des vorletzten Layers des Encoders.

Der zweite Deconvolution Layer des Decoders berechnet mithilfe eines weiteren 1x1 Kernels einen neuen Tensor der Größe YxBxH. Dabei steht 'Y' für die Anzahl der berechneten Featuremaps des fünften Layers des Encoders, die Dimensionen 'B' und 'H' bleiben weiterhin gleich.

Nach der Extraktion der zeitlichen Features durch die ersten vier Convolution Layer im Encoder liegt ein Tensor der Größe 64xBxH vor. Das nächste Deconvolution Layer berechnet dementsprechend ein Tensor der selben Größe. Dazu wird ein Kernel mit der Größe 1x1 verwendet.

Die nächsten drei Layer sind weitere Deconvolution Layer, die jeweils mit einem 1x5 Kernel die Daten auf der zeitlichen Ebene entfalten. Analog dazu befinden sich im Encoder die Convolution Layer zwei, drei und vier, welche die Daten mit einem 1x5 Kernel falten und zeitliche Features extrahieren. Jedes dieser Deconvolution Layer berechnet ebenfalls 64 Featuremaps wie die äquivalenten Convolution Layer im Encoder.

Der Outputlayer des Decoders ist ein Deconvolution Layer mit einem 1x5 Kernel. Als Output wird eine Ergebnismatrix berechnet, welche die gleiche Größe besitzt, wie die der Inputdaten des Encoders.

Bis auf das Outputlayer besitzt jedes Layer des Decoders die ReLU-Aktivierungsfunktion. Ziel ist es, mithilfe des Decoders eine lineare Regression der Daten zu berechnen. Da alle Werte der Tensoren permanent im positiven Wertebereich liegen, besteht kein Unterschied, ob eine lineare Aktivierungsfunktion, oder die ReLU-Aktivierungsfunktion im Decoder gewählt wird, da die beiden Funktionen sich im positiven Wertebereich gleichen (siehe Abschnitt 2.2.1.3).

Da die Inputdaten vor der Verarbeitung normalisiert wurden, liegt jeder Wert der Inputmatrix zwischen null und eins. Daher bietet es sich für das Outputlayer an, die Sigmoid-Aktivierungsfunktion zu verwenden, die als Output für jeden Eintrag in der berechneten Outputmatrix einen Wert zwischen null und eins generiert. In Abbildung 4.3 ist der Aufbau des Decoders dargestellt.

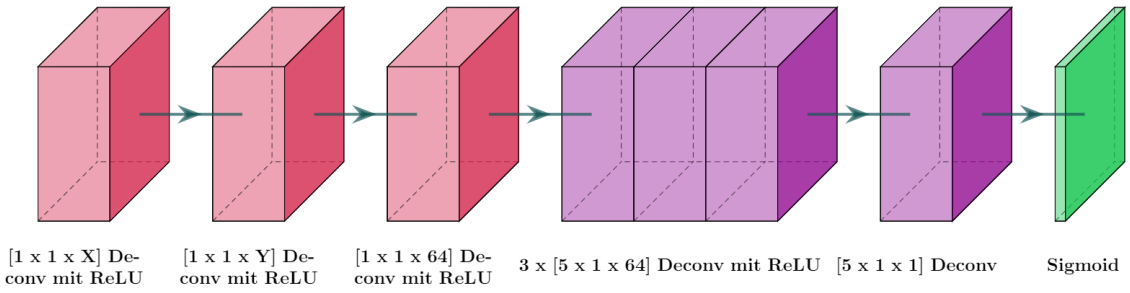

Abbildung 4.3: Aufbau des Decoders. X und Y stehen repräsentativ für die variable Anzahl der Featuremaps basierend auf den 1x1 Convolutional Layern des verwendeten Encoders aus 4.2. Rot: Deconvolutionlayer analog zu den 1x1 Convolutionlayern des Encoders. Magenta: Deconvolutionlayer analog zu den 5x1 Convolutionlayern des Encoders. Grün: Sigmoid Aktivierungsfunktion.

4.2.2 Training

Das Training des Netzwerks findet über 20 Epochen auf demselben Trainingsdatenset wie dem Fully Convolutional Network aus Abschnitt 4.1 statt. Experimentell soll zusätzlich bestimmt werden, ob das Rauschen, welches beim Training des Fully Convolutional Networks auf die Inputdaten gelegt wurde, einen Effekt auf die Rekonstruktion der Daten mittels Decoder besitzt. Das Netz wird ebenfalls, wie das Fully Convolutional Network aus dem vorherigen Abschnitt, mithilfe des Batch Gradientenabstiegsverfahrens und der RMSProp-Updateregel trainiert.

Die Lernrate für das Training hängt von dem eingesetzten Encoder ab. In Abschnitt 5.4.2.1 wird die beste Lernrate für verschiedene Encoder Architekturen bestimmt. Diese werden auch für das Training der Auto-Encoder-Decoder Architektur übernommen.

Auch beim Training des Auto-Encoder-Decoders wird das Early-Stopping Verfahren angewandt. Angestrebt wird eine hohe Genauigkeit des Attributvektors, der durch das Average-Pooling im Encoder, basierend auf der Zwischendarstellung, berechnet wird.

Die Kostenfunktion des Trainings setzt sich zusammen aus der binären Cross-Entropy-Loss Funktion (BCE) und der Mean squared error Loss Funktion (MSE):

$$Loss = BCE(z, y) + MSE(x, x') \tag{4.2}$$

Dabei steht x für den Input, x' für die Rekonstruktion des Decoders, z für die berechnete Zwischendarstellung des Encoders und y für den gelabelten Attributsvektor passend zu den Inputdaten.

4.2.3 Synthetische Datengenerierung

Der Encoder erzeugt einen 19xBxH Tensor. Jeder Datenpunkt der BxH Ebene besitzt daher 19 Features. Der Attributsvektor wird aus diesem Tensor durch einen Average-Poolinglayer berechnet, daher repräsentieren die 19 Features für jeden Datenpunkt des Tensors einen Attributsvektor. Durch Änderung der Elemente einer Featureebene ist es möglich, das dazugehörige Attribut der Bewegung zu verändern. Führt die Interpretation der geänderten Daten durch den Decoder zu einer plausiblen Ausgabe, also zu passenden Sensorkurven, kann der Decoder zur Generierung synthetischer Bewegungsdaten eingesetzt werden. Der Nachweis dazu wird erbracht durch visuellen Vergleich der grafischen Darstellungen von Input und rekonstruierten Daten des Autoencoders.

Kapitel 5

Evaluation

Im vorherigen Kapitel wurden eine Fully Convolutional Network und eine Decoder Architektur vorgestellt, die es in diesem Kapitel zu evaluieren gilt. Dafür wird ein Testdatensatz benötigt, mit dem die Architektur trainiert und getestet werden kann. Dazu wird in Abschnitt 5.1 das LARa-Datenset vorgestellt, welches im Rahmen der Experimente verwendet wird. Um die Performance der Netze vergleichen zu können, werden geeignete Metriken benötigt, die in diesem Kapitel vorgestellt werden. Weiterhin sind die Trainingssetups beschrieben. Anschließend erfolgt der Vergleich der verschiedenen Netzstrukturen anhand der Performancemessungen. Es folgt die Überprüfung der rekonstruierten Daten aus dem Autoencoder durch Visualisierung. Im letzten Abschnitt werden die Erkenntnisse aus den Experimenten genutzt, um ein Fully Convolutional Network zu entwickeln, welches in der Lage ist, durch Veränderung der Zwischendarstellung synthetische Daten zu generieren. Gelingt es, synthetische Daten zu generieren, können kostengünstig neue Daten erstellt werden. Diese können dazu eingesetzt werden, Datensets zu vergrößern oder auszugleichen (siehe Abschnitt 2.1).

5.1 LARa-Datenset

'LARa' ist ein Akronym für 'Logistic Activity Recognition Challenge'. Dieses Datenset beinhaltet insgesamt 758 Minuten an Aufzeichnungen von menschlichen Aktivitäten in einem Lagerhaus [43]. Die Sensordaten wurden gesammelt, indem 14 Versuchspersonen natürliche Aufgaben in einem künstlich aufgebauten Lagerhaus unter Laborbedingungen nachstellten.

Zur Erfassung der Bewegungen trugen die Versuchspersonen sowohl 39 Marker an ihrem Körper, welche durch ein Kamerasystem getrackt wurden, als auch sechs IMUs (Internal Measurement Units) zur Messung der triaxialen Linear- und der Winkelbeschleunigung. Die Aufzeichnungen der IMUs flossen in das MbientLab-Dataset und die Aufzeichnungen der Kamerasysteme sind im MoCap-Datenset zusammengefasst.

Für alle durchgeführten Experimente wird ausschließlich das MbientLab-Dataset verwendet. Nur bei der Visualisierung der Daten wird das MoCap-Datenset benutzt, weil es sich wesentlich besser zur 3D-Darstellung der Aktivitäten eignet.

Im LARa-Datenset besteht das Label einer Bewegung aus einer Aktivitätsklasse und einer Attributrepräsentation.

5.1.1 Attributrepräsentation des LARa-Datensets

Die Attributrepräsentation ist ein binärer Vektor mit 19 Einträgen. Jeder dieser Einträge steht für ein bestimmtes Attribut einer Bewegung wie zum Beispiel 'die linke Hand wird benutzt'. In Tabelle 5.1 wird die Bedeutung der verschiedenen Attribute und ein beispielhafter Attributvektor dargestellt. Duch die semantische Interpretation des Attributvektors lässt sich die zugehörige Aktivitätsklasse ableiten. Details dazu finden sich in dem Artikel [43].

Tabelle 5.1: Definition der Attribute und Attributrepräsentation

Attribute class	Attribute	Definition	Example vector
Legs	A	Gait Cycle	0
	B	Step	0
	C	Standing Still	1
Upper Body	A	Upwards	0
	B	Centered	0
	C	Downwards	0
	D	No Intentional Motion	1
	E	Torso Rotation	0
Hands	A	Right Hand	0
	B	Left Hand	0
	C	No Hand	1
Item	A	Bulky Unit	0
	B	Handy Unit	0
	C	Utility	0
	D	Cart	0
	E	Computer	0
	F	No Item	1
Data	A	None (data has no class)	0
	B	Error (data is corrupted)	0

5.1.2 Klassen des LARa-Datensets

Zur Klassifikation der Daten stehen acht Klassen zur Verfügung. Eine davon ist die Klasse ´none´, in die alle Bewegungen eingeornet werden, die nicht eindeutig klassifizierbar sind. Diese Klasse wird im weiteren Verlauf nicht weiter beachtet, die anderen werden in der Arbeit mit c1-c7 abgekürzt [43].

- Klasse c1 (´stehen´): Hier werden alle Daten eingeordnet, in denen die Versuchsperson steht oder kleine Schritte macht. Dabei kann die Person ein Objekt in den Händen halten.
- Klasse c2 (´gehen´): Enthalten sind alle Daten, in denen die Person geht. Optional kann auch hier ein Gegenstand getragen werden.
- Klasse c3 (´Wagen´): Hier sind alle Bewegungen gesammelt, in denen die Person einen Wagen bewegt.
- Klasse c4 (´handling oben´): Die Versuchsperson benutzt einen Gegenstand und mindestens eine Hand befindet sich oberhalb der Schulter.
- Klasse c5 (´handling center´): Es wird ein Objekt benutzt, wobei sich keine Hand über Schulterhöhe befindet und die Person weder kniet noch den Oberkörper beugt.
- Klasse c6 (´handling unten´): Die Versuchsperson benutzt einen Gegenstand in gebeugter oder knieender Haltung.
- Klasse c7 (´synchronisation´): Hier sind alle Bewegungen zusammengefasst, die zur Synchronisation der Sensoren dienen. Die Personen stehen aufrecht und winken mit beiden Händen über dem Kopf.

In der folgenden Tabelle ist dargestellt, wie groß der Anteil der Klassen im Datenset ist.

Tabelle 5.2: Anteil der Klassen im Datenset

Klassen	**c1**	**c2**	**c3**	**c4**	**c5**	**c6**	**c7**
Anteil	10.77%	11.00%	13.11%	8.34%	43.12%	7.45%	1.75%

5.2 Performance Metrik

Der Output des neuronalen Netzes ist ein Attributvektor, der in jedem Eintrag einen Wert zwischen null und eins enthält. Jeder dieser Werte gibt eine Wahrscheinlichkeit an, mit der ein bestimmtes Attribut auftritt. Zu allen gültigen Attributvektoren wird je ein Distanzvektor berechnet, der den Abstand des Outputvektors zum jeweiligen Attributvektor angibt. Der gültige Attributvektor, der die geringste Distanz zum Outputvektor des Netzes

aufweist, wird als nächster Nachbar bestimmt. Aus diesem lässt sich die Aktivitätsklasse ableiten, welche als Ergebnis der Klassifikation des Netzes ausgegeben wird. Die, durch die nearest neighbor search berechnete Klasse, wird verwendet, um die einzelnen Metriken zu berechnen. Angewandt werden die Accuracy, F1-mean Score, F1-weighted Score und die p-Metrik. Die Berechnung der Metriken findet beim Validieren auf dem gesamten Validierungs-Datenset statt, nachdem für alle Daten ein Attributvektor berechnet wurde.

- Accuracy: Die Accuracy gibt prozentual an, wie viele der berechneten Aktivitätsklassen mit den Labeln der Daten übereinstimmen. Die Formel zur Berechnung lautet:

$$Accuracy = \frac{\#correct\ \ predictions}{\#all\ \ predictions} \tag{5.1}$$

- Precision: Durch die Precision-Metrik kann berechnet werden, wie präzise das Netzwerk in Bezug auf eine Aktivitätsklasse ist. Die Formel dazu lautet:

$$Precision = \frac{\#True\ \ Positive}{\#Total\ \ predicted\ \ Positive} \tag{5.2}$$

  $\#True\ \ Positive$ ist die Anzahl der Elemente, die das Netz der untersuchten Aktivitätsklasse korrekt zugeordnet hat. $\#Total\ \ predicted\ \ Positive$ steht für die Anzahl aller Daten, die der untersuchten Aktivitätsklasse durch das Netz zugeordnet wurden. Hierbei werden auch die falschen Zuordnungen gezählt. Liegt eine hohe Precision für die untersuchte Aktivitätsklasse vor, sind die Zuordnungen zu dieser Klasse mit hoher Wahrscheinlichkeit richtig.

- Recall: Die Formel zur Berechnung des Recalls lautet:

$$Recall = \frac{\#True\ \ Positive}{\#Relevant\ \ Elements} \tag{5.3}$$

  Die Definition von $\#True\ \ Positive$ ist identisch mit der Beschreibung im vorherigen Abschnitt. $\#Relevant\ \ Elements$ steht für die Anzahl, wie viele Daten insgesamt der betrachteten Klasse hätten zugeordnet werden müssen. Dieser Wert ergibt sich aus den Labeln der Eingabedaten.

- F1 Score: Er wird über folgende Formel ermittelt:

$$F1\ \ = 2 * \frac{Recall\ \ *\ \ Precision}{Recall\ \ +\ \ Precision} \tag{5.4}$$

  Mathematisch entspricht er dem harmonischen Mittel zweier Zahlen, in diesem Falle dem Recall und der Precision. Ein optimales Ergebnis liegt vor, wenn F1, Recall und Precision identisch sind. Er wird pro Aktivitätsklasse berechnet.

- F1-mean Score: Der F1-mean Score wird über folgende Formel ermittelt:

$$F1\ \ mean = \frac{1}{n}\sum_{n}(F1_n) \tag{5.5}$$

Der F1-mean Score ist das arithmetische Mittel aller F1-Scores der Aktivitätsklassen. In der Formel steht n für die Anzahl der Klassen, der F1 Score ergibt sich aus der vorherigen Formel 5.4.

- F1-weighted Score: Der F1-weighted Score errechnet sich wie folgt:

$$F1\ \ weighted = \frac{1}{n}\sum_{n}(F1_n * Propotion\_of\_class) \tag{5.6}$$

Auf imbalancierten Datensets (darunter fällt auch das LARa-Datenset) ist der F1-mean Score nicht immer aussagekräftig, da alle Aktivitätsklassen mengenunabhängig den gleichen Einfluss auf die Bewertung haben. Das Problem wird durch den F1-weighted Score behoben, da die Gewichtung der Klassen nach ihrer Größe in die Bewertung einbezogen wird.

- p-Metrik:

Eine weitere Metrik, die in dieser Arbeit verwendet wird, ist die p-Metrik. Sie dient dazu, die Leistungsänderung von neuronalen Netzen zu messen. Dazu wird überprüft, ob die Zuordnung der Trainingsdaten das Ergebnis der Experimente beeinflusst hat. Für die Berechnung des p-Wertes werden für beide Experimente künstliche Datenmengen erzeugt, die der Anzahl der Testdaten entsprechen. Ihnen werden die Label ´erkannt´ und ´nicht erkannt´ zugeordnet. Die Verteilung der Label ergibt sich für jedes Experiment aus dem Wert der Metrik, deren Leistungsänderung gemessen werden soll. Im Rahmen dieser Arbeit wird dafür die accuracy verwendet. Im nächsten Schritt werden beide künstlichen Datenmengen miteinander permutiert. Anschließend wird die zu vergleichende Metrik (in diesem Fall die accuracy) erneut für beide Mengen berechnet. Um eine statistische Relevanz zu erreichen, wird dies sehr oft wiederholt. Dabei wird die Anzahl der Fälle ermittelt, in denen das, an der Vergleichsmetrik (accuracy) gemessene, vorher ´schlechtere´ Experiment ein besseres Ergebnis erzielen konnte, als das andere Experiment. Der p-Wert wird durch die Formel 5.7 berechnet. Ist der p-Wert nahe null, besteht ein signifikanter Unterschied zwischen den Ergebnissen beider Experimente.

$$pWert = \frac{\#different\ \ Results}{\#Permutations} \tag{5.7}$$

5.3 Trainingssetup

In den folgenden Abschnitten werden einheitliche Setups zum Training der Netzarchitekturen definiert. So ist gewährleistet, dass die Ergebnisse vergleichbar sind. Ein fester Seed verhindert, dass durch die zufällige initiale Belegung der Gewichte die Ergebnisse verfälscht werden.

5.3.1 Encoder

Für das Training der Encoder Architekturen wird das Batch Gradientenabstiegsverfahren mit einer Batch-Größe von 100 verwendet. Zur Optimierung der Gewichte während des Trainings wird der RMSprop-Algorithmus mit einem Alpha-Wert von 0.95 verwendet. In einem ersten Experiment soll die beste Lernrate für jede verwendete Netzarchitektur ermittelt werden. Es werden dabei die initialen Lernraten von 10^{-4}, 10^{-5} und 10^{-6} verwendet. Das Training jedes Netzwerks findet über 20 Epochen statt. Nach jeweils sieben und 14 Epochen wird die bestehende Lernrate mit 0.1 multipliziert, sodass sie mit zunehmender Dauer des Trainings sinkt. Ein Epoch besteht aus 914 Iterationen. Die Anzahl ergibt sich aus der Batch-Größe sowie der Anzahl der Daten für das Training. Nach jeweils 90 Iterationen findet ein Stop des Early Stopping Verfahrens statt, bei dem das bestehende Netz auf dem Validierungsdatenset getestet, und bei besserer Accuracy abgespeichert wird. Als Kostenfunktion während des Trainings wird die binäre Cross-Entropy-Loss Funktion (siehe Formel 2.10) verwendet.

5.3.1.1 Finetuning des Encoders

Nach Ermittlung der besten Architekturen und den dazugehörigen Lernraten durch die Experimente findet das Finetuning der Netze statt. Dazu werden die ersten vier Layer eingefroren, sodass sich ihre Gewichte bei einem weiteren Training nicht mehr verändern. Daher werden nur die beiden 1x1 Convolution Layer und das Fully Connected Layer neu trainiert. Das Training der Netzarchitekturen findet wie oben beschrieben statt, jedoch nur fünfmal über 10 Epochen.

5.3.2 Auto Encoder Decoder

Für das Training mit Hilfe eines Decoders, werden die besten Encoder Architekturen und die dazugehörigen Lernraten gewählt. Für jede der Architekturen wird ein symmetrischer Decoder für das Training des Encoders entwickelt. Der Aufbau des Decoders hängt stark von der Architektur des Encoders ab. Eine genauere Beschreibung der Konstruktion befindet sich in Abschnitt 4.2.1. Als Kostenfunktion wird weiterhin die binäre Cross-Entropy-Loss Funktion (BCE) verwendet, um die Berechnung des Attributvektors zu erlernen. Zusätzlich wird sie durch die Mean squared error Loss Funktion (MSE) erweitert. Dadurch

soll die Rekonstruktion der Daten erlernt werden. Formel 4.2 zeigt diese erweiterte Kostenfunktion. Die anderen Parameter des Trainings sind identisch mit denen aus Abschnitt 5.3.1.

5.4 Experimente und Ergebnisse

Der Abschnitt beschäftigt sich mit den durchgeführten Experimenten. Zuerst wird das TCN aus [43] rekonstruiert, da das Netz den Ausgangspunkt für die weiteren Experimente ist. Nach der Rekonstruktion folgt in Abschnitt 5.4.2 die schrittweise Konstruktion und Analyse eines Encoders. Die besten Encoder-Architekturen aus den Experimenten werden für die Konstruktion der Autoencoder-Architekturen in Abschnitt 5.4.3 gewählt. Durch Experimente wird ermittelt, wie unterschiedliche Trainingsmethoden mit einem Decoder die Performance beeinflussen. Zusätzlich folgt die visuelle Auswertung der rekonstruierten Daten, um festzustellen, ob grundsätzlich die Rekonstruktion von Bewegungen mithilfe eines Decoders möglich ist. Anschließend wird ein Fully Convolutional Network auf Basis der gesammelten Erkenntnisse entwickelt, welches zur synthetischen Datengenerierung eingesetzt werden soll.

5.4.1 Rekonstruktion der Baseline

Das TCN aus [43] stellt ein state-of-the-art Netzwerk zur Erkennung von menschlichen Aktivitäten auf dem LARa-Datenset dar. Daher eignet es sich sehr gut als Ausgangspunkt und Vergleichsnetzwerk für die folgenden Experimente. In einem ersten Schritt erfolgt die Rekonstruktion des TCN aus [43]. Dazu wird die Architektur des TCNs nachgebaut und mit einer Lernrate von 10^{-4} trainiert. Anschließend findet ein Finetuning der letzten Layer statt. Die Performance des rekonstruierten TCN ist in Tabelle 5.3 abgebildet. In der Arbeit [51] wird das TCN aus [43] verwendet, um die MbientLab-Daten des LARa-Datensets zu klassifizieren. In [51] wurde nur der F1 weighted Score zur Messung der Performance verwendet. Das Original-TCN erreichte dort einen Wert von 75.75% auf dem Test-Datensatz.

Tabelle 5.3: Rekonstruktion des TCNs

Netz	Metrik	Training	Finetuning	Performance auf Test-Datensatz
TCN	acc	70.52	70.93	76.28
	f1 mean	54.92	57.47	57.74
	f1 weighted	68.39	69.68	74.31

5.4.2 Encoder

Zur Umwandlung des TCNs in ein Fully Convolutional Network werden die Fully Connected Layer schrittweise ersetzt. Für jeden Schritt werden Experimente durchgeführt, um die Performanceveränderung der verschiedenen Netzarchitekturen zu analysieren. Das erste Experiment dient zur Ermittlung der besten Lernrate sowie der optimalen Anzahl von Featuremaps für die 1x1 Convolutionlayer. Ein weiteres Experiment soll Aufschluss darüber geben, ob das Hinzufügen eines Poolinglayers die Performance steigert. Im letzten Experiment wird die Auswirkung eines Finetunings auf die Netzarchitekturen untersucht.

5.4.2.1 Anzahl der Featuremaps und Lernrate

Zur Umwandlung des Temporal Convolutional Networks in ein Fully Convolutional Network mit 1x1 Convolutional Layern wurde in einem ersten Schritt die beste Anzahl der Featuremaps bestimmt. Dazu wurden dreizehn verschiedene Konfigurationen mit jeweils drei unterschiedlichen Lernraten trainiert und mit drei Metriken bewertet. Zur besseren Übersicht sind die Ergebnisse in Gruppen zusammengefasst. Die Gruppenbildung erfolgte anhand der Anzahl der Featuremaps des ersten 1x1 Convolutional Layers. Die besten Ergebnisse innerhalb jeder Gruppe sind Grün dargestellt. Die Netznamen unterliegen folgender Systematik: Encoder AxB, wobei A für die Anzahl der Featuremaps des ersten und B für die Anzahl der Featuremaps des zweiten 1x1 Convolutional Layers stehen. Die Ergebnisse sind in den Tabellen 5.4 und 5.5 dargestellt.

Tabelle 5.4: Ergebnisse Gruppe 1-2

		Gruppe 1		Gruppe 2		
Lernrate	**Metrik**	256x256	256x128	128x128	128x256	128x64
lr=10^{-4}	acc	67.50	67.95	68.34	68.49	68.43
	f1m	57.49	55.76	57.58	60.59	55.75
	f1w	68.25	68.08	68.86	69.01	68.58
lr=10^{-5}	acc	58.62	63.72	62.68	59.01	62.23
	f1m	40.87	43.01	40.80	39.90	40.69
	f1w	57.12	60.13	58.48	56.98	58.41
lr=10^{-6}	acc	55.64	54.07	53.72	55.44	49.42
	f1m	31.58	26.12	24.13	30.69	9.45
	f1w	46.03	42.83	41.58	45.46	32.72

Tabelle 5.5: Ergebnisse Gruppe 3-5

		Gruppe 3			Gruppe 4			Gruppe 5	
Lernrate	**Metrik**	64x64	64x128	64x32	32x32	32x64	32x19	19x19	19x32
lr=10^{-4}	acc	68.04	67.91	67.98	66.78	67.91	66.49	68.03	67.05
	f1m	59.53	58.20	56.24	54.00	58.60	52.05	52.87	52.26
	f1w	68.09	67.39	67.80	66.54	67.48	65.94	67.01	66.13
lr=10^{-5}	acc	62.60	62.99	61.65	61.94	62.57	59.50	60.61	62.10
	f1m	41.03	42.89	38.95	40.20	40.79	35.75	37.70	40.19
	f1w	58.51	59.35	57.09	57.84	58.29	54.49	55.99	58.00
lr=10^{-6}	acc	49.46	50.99	49.43	49.43	49.46	49.43	49.43	49.43
	f1m	9.63	16.71	9.45	9.45	9.86	9.45	9.45	9.45
	f1w	32.93	36.73	32.70	32.70	33.19	32.70	32.70	32.70

Die Experimente zeigen, dass eine Lernrate von 10^{-4} die besten Ergebnisse für alle Netzstrukturen liefert. Die Anzahl der Featuremaps haben in diesem Experiment keinen entscheidenden Einfluss auf das Ergebnis, wenn man die Ausnahme ENC 19x19 beim F1 mean Score nicht berücksichtigt.

5.4.2.2 Poolinglayer

Ein zweites Experiment soll zeigen, ob die Performance des Encoders durch einen zusätzlichen Poolinglayer gesteigert werden kann. In der Human Activity Recognition wird nach der Featureextraktion die Anzahl der Features reduziert, um die Performance der Klassifikatoren zu steigern (siehe Abschnitt 3.1.4). Je mehr Features zur Interpretation für eine Klasse gegeben sind, desto schwieriger gestaltet sich die Berechnung [16]. Poolinglayer reduzieren in einem neuronalen Netz die Anzahl der Parameter eines Tensors, daher soll durch Verwendung eines solchen versucht werden, die Klassifikation des Netzes zu verbessern. Dieser wird zwischen dem letzten 1x1 Convolution Layer und dem Fully Connected Layer eingefügt. Abbildung 5.1 zeigt die veränderte Architektur des Encoders.

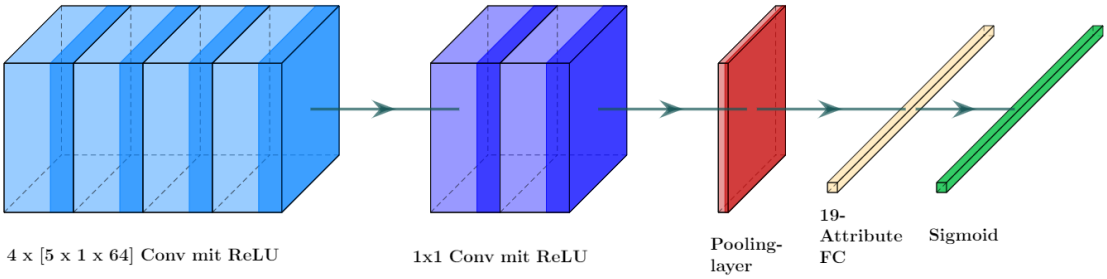

Abbildung 5.1: Aufbau der Encoder Architektur mit einem zusätzlichen Poolinglayer. Hellblau: Vier Convolutional Layer mit 5x1 Kernel und 64 berechneten Featuremaps. Dunkelblau: 1x1 Convolutional Layer. Rot: Zusätzliches Poolinglayer. Gelb: Fully Connected Layer mit 19 Neuronen. Grün: Sigmoid Aktivierungsfunktion.

Zur Überprüfung dieser These werden die besten Netze des ersten Experiments (siehe 5.4.2.1) ausgewählt und modifiziert. Es werden sowohl Max-Pooling, als auch Average-Pooling separat über jede Dimension der Daten (Sensor, Zeit und Feature) ausgeführt. Die Sensorebene der Daten wird von fünf IMUs bereitgestellt, die jeweils drei Kanäle für die Linear- und drei Kanäle für die Winkelbeschleunigung besitzen. Deshalb wird für das Pooling ein 1x3 Kernel mit einem vertikalen Stride von drei verwendet, der jeweils die linearen Beschleunigungswerte und die Winkelbeschleunigungen auf einen Wert abbildet. Die Ergebnisse des Sensor-Poolings befinden sich in Tabelle 5.6.

Tabelle 5.6: Poolinglayer: Ergebnisse Sensor-Pooling

Pooling	**Metrik**	**19x19**	**32x64**	**64x64**	**128x256**	**256x256**
AVERAGE	accuracy	62.21	65.00	65.37	68.05	68.05
	f1 mean	37.30	45.73	47.42	53.28	52.04
	f1 weighted	56.52	62.25	63.11	67.11	66.69
MAX	accuracy	62.79	60.27	60.58	63.91	63.55
	f1 mean	50.92	50.35	49.64	55.38	53.91
	f1 weighted	63.43	61.89	61.67	65.09	64.73
OHNE POOL.	accuracy	68.03	67.91	68.04	68.49	67.50
	f1 mean	52.87	58.60	59.53	60.59	57.49
	f1 weighted	67.01	67.48	68.09	69.01	68.50

Die zeitliche Ebene spielt für die Analyse der Daten eine wichtige Rolle. Da Pooling die Daten generell destruktiv bearbeitet, wurde ein möglichst kleiner Kernel der Größe 2x1 mit einem Stride von eins ausgewählt.

Tabelle 5.7: Experiment Poolinglayer: Zeit-Pooling

Pooling	Metrik	19x19	32x64	64x64	128x256	256x256
AVERAGE	accuracy	67.57	68.02	68.77	68.35	68.16
	f1 mean	50.83	56.32	58.87	59.32	56.90
	f1 weighted	65.90	67.82	68.80	68.96	68.81
MAX	accuracy	66.80	66.58	68.14	68.21	67.24
	f1 mean	51.41	58.15	58.66	58.94	56.48
	f1 weighted	65.93	67.48	68.74	68.57	68.02
OHNE POOL.	accuracy	68.03	67.91	68.04	68.49	67.50
	f1 mean	52.87	58.60	59.53	60.59	57.49
	f1 weighted	67.01	67.48	68.09	69.01	68.50

Beim Pooling über die Feature-Dimension wird ein 2x1x1 Kernel mit einem Stride von zwei verwendet. Die Kernelgröße wurde möglichst klein gewählt, weil dadurch der Informationsverlust durch das Pooling minimiert wird. Das ENC 19x19 besitzt eine ungerade Anzahl an Featuremaps, daher werden in einem der Schritte drei Features zusammengefasst. Tabelle 5.8 zeigt die Ergebnisse dieses Experiments.

Tabelle 5.8: Experiment Poolinglayer: Feature-Pooling

Pooling	Metrik	19x19	32x64	64x64	128x256	256x256
AVERAGE	accuracy	63.35	66.62	68.10	67.28	67.45
	f1 mean	43.37	49.92	52.34	54.94	54.63
	f1 weighted	60.34	64.93	66.79	67.40	67.55
MAX	accuracy	66.44	66.93	66.49	67.62	67.07
	f1 mean	48.20	52.44	50.73	54.91	54.39
	f1 weighted	63.85	66.39	65.40	67.92	67.56
OHNE POOL.	accuracy	68.03	67.91	68.04	68.49	67.50
	f1 mean	52.87	58.60	59.53	60.59	57.49
	f1 weighted	67.01	67.48	68.09	69.01	68.50

Sowohl beim Zeit-Pooling als auch beim Sensor-Pooling ist zu beobachten, dass ein Average-Pooling Layer eine bessere Performance liefert als ein Max-Pooling Layer. Beim Feature-Pooling sind die Ergebnisse gemischt. Es ist zu erkennen, dass ein Feature-Pooling die Performance des Netzes gegenüber einem Netz ohne Pooling Layer verschlechtert.

Beim Sensor-Pooling ist eine ähnliche Tendenz zu sehen, jedoch scheint ein Average-Pooling über die Sensorebene bei dem FCN 256x256 die Performance leicht zu verbessern.

Mit einem Average-Pooling über die Zeitebene konnten die besten Ergebnisse erzielt werden. Die Performance des FCN 32x64, FCN 64x64 und FCN 256x256 konnte in den Experimenten mithilfe eines Zeit-Poolings verbessert werden. Die größte Verbesserung konnte beim FCN 64x64 mit einer Steigerung von 0.73% in der Accuracy und 0,71% beim F1-Score erreicht werden. Jedoch liefern das FCN 19x19 und FCN 128x256 die beste Performance ohne Pooling Layer.

5.4.2.3 Finetuning

Während des Trainings neuronaler Netze verändern sich die Gewichte der Schichten ständig. Dadurch verändert sich die Featurerepräsentation der Daten ebenfalls. Das erschwert die Interpretation der Features durch die nachfolgenden Schichten. Das Finetuning ist der Versuch diesem Problem entgegen zu wirken. Durch einen zweiten Trainingsschritt, bei dem die Gewichte der ersten vier Convolutionlayer eingefroren sind, soll eine bessere Interpretation durch die Konstanz der Features erlernt werden.

Versuche mit dem TCN aus [43] zeigten, das dieses Verfahren zu einer verbesserten Performance führte (siehe Tabelle 5.3). Die hier verwendeten Architekturen basieren auf der Struktur des TCN, daher wird untersucht, ob auch für die Encoder durch Finetuning eine Performanceverbesserung erreichbar ist. Die Ergebnisse des Finetunings sind in Tabelle 5.9 aufgelistet. Es ist zu erkennen, dass das Finetuning überraschend die Performance aller getesteten Netze verschlechtert hat. Auf eine Klärung dieses Verhaltens wird im Rahmen dieser Arbeit verzichtet, weil der Grund nicht relevant für die synthetische Datengenerierung ist.

Tabelle 5.9: Experiment Finetuning: Ergebnisse mit und ohne Finetuning

Finetuning	**Metrik**	**19x19 ohne Pooling**	**32x64 Zeit-Pooling**	**64x64 Zeit-Pooling**	**128x256 ohne Pooling**	**256x256 Zeit-Pooling**
MIT	accuracy	67.51	67.47	67.95	67.78	67.96
	f1 mean	52.64	54.10	57.15	56.61	56.49
	f1 weighted	66.64	67.22	68.33	68.29	68.46
OHNE	accuracy	68.03	68.02	68.77	68.49	68.16
	f1 mean	52.87	56.32	58.87	60.59	56.90
	f1 weighted	67.01	67.82	68.80	69.01	68.81

5.4.2.4 Zusammenfassung der Ergebnisse zur Konstruktion des Encoders

Zunächst wird die Performance des TCN aus 5.3 mit den fünf besten Encodern der durchgeführten Experimente verglichen. Das Ergebnis stellt Tabelle 5.10 dar.

Tabelle 5.10: Die besten Encoder Architekturen aus den durchgeführten Experimenten

Netz	**Metrik**	**Ergebnis**
256x256 mit Zeit-Pooling	accuracy	68.16
	f1 mean	56.90
	f1 weighted	68.81
128x256 ohne Pooling	accuracy	68.49
	f1 mean	60.59
	f1 weighted	69.01
64x64 mit Zeit-Pooling	accuracy	68.77
	f1 mean	58.87
	f1 weighted	68.80
32x64 mit Zeit-Pooling	accuracy	68.02
	f1 mean	56.32
	f1 weighted	67.82
19x19 ohne Pooling	accuracy	68.03
	f1 mean	52.87
	f1 weighted	67.01
Rekonstruiertes TCN	accuracy	70.93
	f1 mean	57.47
	f1 weighted	69.68

Zu erkennen ist, dass die Performance der konstruierten Encoder auf dem Validierungs-Datensatz in der Accuracy ca. 2-3% und beim F1-Score ca. 1-2% schlechter ist als beim rekonstruierten TCN aus [43]. Zur weiteren Analyse werden im Folgenden die beiden besten Encoder und das rekonstruierte TCN auf einem Test-Datensatz getestet. Die Ergebnisse sind in den Abbildungen 5.2 bis 5.4 dargestellt.

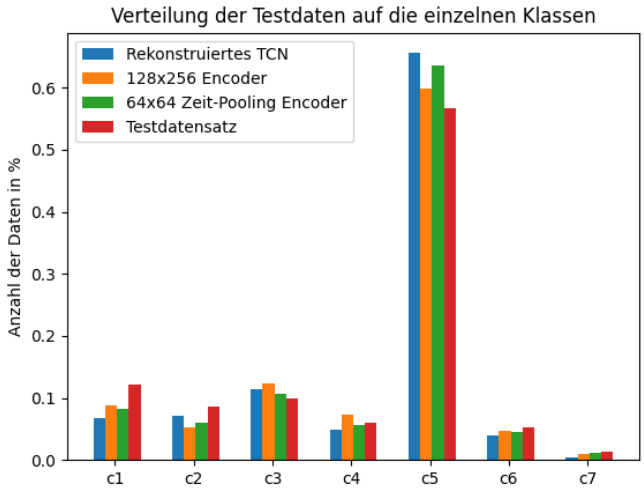

Abbildung 5.2: Verteilung der Daten bezogen auf die einzelnen Klassen. Rot: Tatsächliche Verteilung der Klassen im Test-Datenset. Blau/Orange/Grün: Durch neuronale Netze berechnete Ergebnisse.

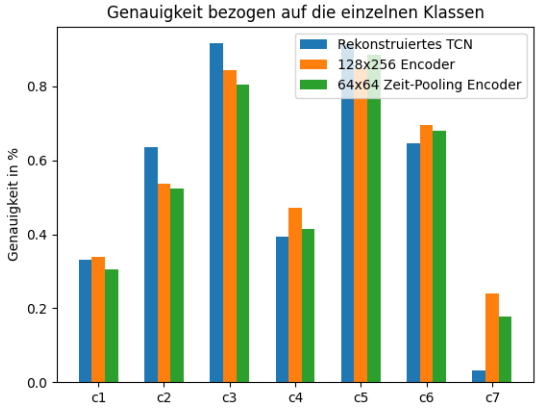

Abbildung 5.3: Genauigkeit der Klassifikation für die einzelnen Aktivitäten auf dem Test-Datensatz

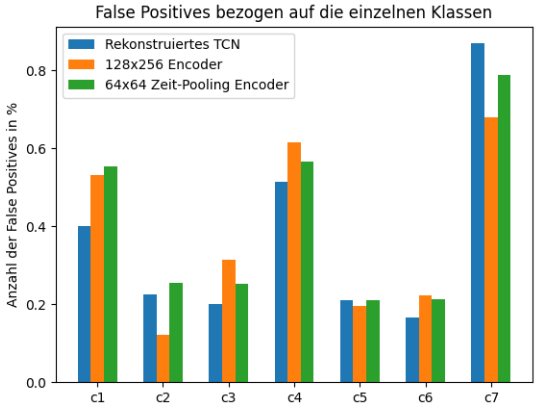

Abbildung 5.4: Anzahl der False Positives in Bezug auf die einzelnen Klassen auf dem Test-Datensatz

Grundsätzlich ist zu erkennen, dass alle getesteten Netze Schwierigkeiten haben, die Klassen ´stehen´, ´Handling oben´ und ´synchronisation´ (c1, c4 und c7) zu erkennen. In den Klassen ´Handling oben´, ´Handling unten´ und ´synchronisation´ schneiden die Encoder besser ab als das TCN, in den anderen Klassen sind sie unterlegen.

Tabelle 5.11: Werte der Confusionsmatrix des 128x256 Encoders (dünn) und des TCNs (**fett**) im Vergleich

Klassen	c1	c2	c3	c4	c5	c6	c7
c1	2105/ **2155**	242/ **316**	399/ **230**	277/ **85**	2908/ **3429**	130/ **68**	230/ **108**
c2	994/ **611**	2443/ **2893**	538/ **252**	115/ **39**	410/ **730**	30/ **12**	24/ **17**
c3	124/ **41**	12/ **45**	4424/ **4799**	2/ **0**	666/ **351**	5/ **1**	4/ **0**
c4	109/ **65**	24/ **18**	79/ **63**	1479/ **1214**	1412/ **1745**	6/ **4**	35/ **11**
c5	1107/ **581**	450/ **424**	960/ **647**	1542/ **644**	25326/ **27254**	375/ **264**	64/ **10**
c6	113/ **110**	16/ **36**	33/ **7**	9/ **3**	650/ **808**	1902/ **1766**	7/ **0**
c7	2/ **1**	0/ **0**	5/ **0**	418/ **539**	113/ **147**	0/ **0**	171/ **22**

Betrachtet man die Confusionsmatrizen (Tabelle 5.11), stellt man fest, dass die Encoder die drei Handling-Klassen (c4, c5 und c6) besser voneinander differenzieren können als das TCN. Eine Verbesserung in der Unterscheidung der Klassen ´Handling oben´ (c4) und ´synchronisation´ (c7) lässt sich ebenfalls feststellen. Während das TCN 55.5% der Daten mit dem Label ´Handling oben´ (c4) der Klasse ´Handling centered´ (c5) zuordnet, sind dies beim 128x256 Encoder nur 45% der Daten. Für die Daten mit dem Label ´Handling unten´ (c6) sind es 30% beim TCN und 24% beim Encoder, die der Klasse ´Handling centered´ (c5) zugeordnet werden.

Bei der Betrachtung der Ergebnisse fällt auf, dass allen Netzen die korrekte Zuordnung einiger Klassenpaare besonders schwer fällt. Die folgende Liste beschreibt diese Paare und enthält einen Erklärungsversuch anhand der Ähnlichkeit der Bewegungen unter Bezug auf die Klassendefinitionen des LARa-Datensets (siehe Abschnitt 5.1.2).

- ´synchronisation´ (c7) und ´Handling oben´ (c4): Der Großteil der falsch zugeordneten Daten der Klasse ´synchronisation´ (c7) wird der Klasse ´Handling oben´ (c4) zugeordnet. Beim TCN sind dies 76% und beim 128x256 Encoder 59% der Daten. Bei der Synchronisation winkt die Versuchsperson mit beiden Händen über dem Kopf und bei der Aktivität ´handling oben´ wird ein Objekt oberhalb der Schultern bewegt. Der Unterschied dieser beiden Bewegungen ist marginal. Neuronale Netze haben die Eigenschaft, schlecht unterscheidbare Daten tendenziell der häufigeren Klasse zuzuordnen. 1.75% der Daten im gesamten Datenset haben das Label ´synchronisation´ (c7) und 8.3% der Daten tragen das Label ´Handling oben´ (c4), wodurch die falsche Zuordnung erklärt werden kann.

- ´Handling oben´ (c4) und ´Handling centered´ (c5): Der Unterschied der Aktivität besteht lediglich darin, dass die Versuchspersonen ein Objekt oberhalb oder unterhalb der Schultern verwenden. Während die Aktivität ´Handling oben´ (c4) 8.3% der Daten im gesamten Datenset ausmacht, wird die Klasse ´Handling centered´ (c5) durch 43.1% der Daten repräsentiert.

- ´stehen´ (c1) und ´Handling centered´ (c5): Die Klasse ´stehen´ (c1) umfasst auch das Stehen mit einem Objekt in der Hand. Die Sensorkurven unterscheiden sich daher nur sehr gering von denen der Klasse ´Handling centered´ (c5). Auch hier ist die ungleiche Verteilung der Daten gegeben, die Klasse c1 hat nur einen Anteil von 10.8% gegenüber dem Anteil von 43.1% der Klasse c5.

- ´Handling unten´ (c6) und ´Handling centered´ (c5): Hier ist die Anzahl der falschen Zuordnungen geringer, aber dennoch erwähnenswert. Die Ursache liegt hier weniger in der Gleichheit der Bewegungen, sondern eher in der ungleichen Verteilung. Während Klasse ´Handling unten´ (c6) nur 7.5% umfasst, besitzt ´Handling centered´ (c5) einen Anteil von 43.1%.

In Abbildung 5.2 ist dieses Verhalten für die betroffenen Klassen sehr gut zu erkennen. Die Klassen, deren Bewegungen den Aktivitäten der Klasse ´Handling centered´ (c5) sehr ähnlich sind (c1, c4, c6), bekommen durch das TCN weniger Daten zugeordnet.

Durch Anwendung der p-Metrik zeigt sich, dass die Performance der Encoder signifikant schlechter ist als die des TCN. Für alle Encoder die mit dem TCN verglichen wurden, betragen die p-Werte 0.0. Daraus folgt, dass das Austauschen der Fully Connected Layer nicht ausreicht, die Performance des Netzwerks zu verbessern. Trotz der schlechteren Gesamtperformance konnte durch den Austausch eine bessere Differenzierung zwischen den unterschiedlichen Klassen erreicht werden, wie man an der Confusionsmatrix (siehe Tabelle 5.11) erkennen kann.

5.4.3 Auto Encoder Decoder

Ziel der folgenden Experimente ist es, die Performance der entwickelten Encoder aus den vorherigen Experimenten zu verbessern, indem nun zusätzlich gezielt räumliche Features erlernt werden.

5.4.3.1 Training mithilfe eines Decoders und dem Attributvektor als Zwischendarstellung

Im ersten Experiment wird der Decoder an das Outputlayer des Encoders gekoppelt. Der Aufbau des gesamten Netzes ist in Abbildung 5.5 vereinfacht dargestellt.

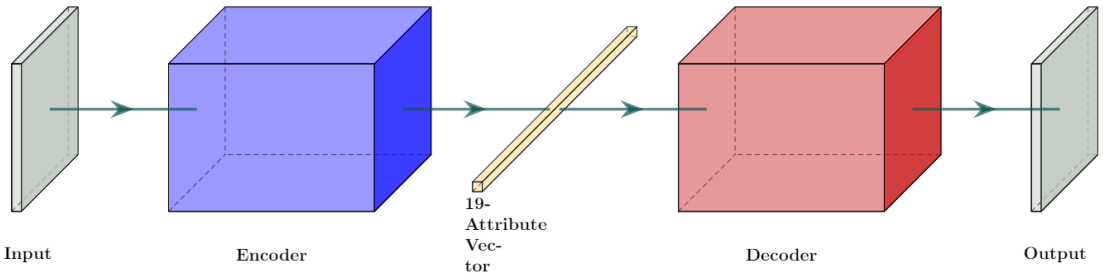

Abbildung 5.5: Vereinfachter Aufbau der Auto-Encoder Network Architektur des Experiments.

Der vom Encoder berechnete Attributvektor stellt die Zwischendarstellung des Netzwerks dar. Der Decoder versucht, die Eingabedaten aus dem 19-stelligen Attributvektor zu rekonstruieren. Sowohl die Klassifikation als auch die Bestimmung der Metriken zur Performancemessung finden unverändert über den Attributvektor statt (siehe Abschnitt 5.2). Die erreichte Performance dieser Netzarchitektur ist in Tabelle 5.12 zu finden.

Tabelle 5.12: AutoEncoder: Ergebnisse des Trainings mit einem Attributvektor als Zwischendarstellung

Netz	accuracy	f1 mean	f1 weighted
512x512 Zeit-Pooling	68.05	56.84	68.67
256x256 Zeit-Pooling	68.53	57.04	69.06
128x256 ohne Pooling	67.96	58.11	68.50
64x64 Zeit-Pooling	67.84	59.15	68.17
32x64 Zeit-Pooling	68.09	56.83	68.31
19x19 ohne Pooling	66.11	48.58	63.76

Der Performancevergleich zeigt, dass ein Training mit Decoder im Vergleich zu einem Training ohne Decoder zu schlechteren Ergebnissen führt. Lediglich die Performance der 256x256 und der 32x64 Netzarchitektur konnte geringfügig verbessert werden, wobei die marginale Verbesserung der 32x64 Netzarchitektur auch zufällig sein kann. Bei den fünf getesteten Netzarchitekturen ist der Trend zu beobachten, dass größere Netze eine bessere Performance lieferten. Daher wurde zusätzlich eine 512x512 Netzarchitektur mit Zeit-Pooling getestet. Das 512x512 Netz weist jedoch eine schlechtere Performance auf als das 256x256 Netz. Einer Verbesserung der Performance durch einfache Vergrößerung des Netzes sind daher Grenzen gesetzt.

5.4.3.2 Training des Encoders mithilfe eines vortrainierten Decoders und dem Attributvektor als Zwischendarstellung

Im Gegensatz zum Experiment im Abschnitt 5.4.3.1, bei dem Encoder und Decoder zusammen trainiert wurden, wird nun versucht, die Performance zu verbessern, indem der Decoder vorab separat trainiert wird. Das Vortraining des Decoders findet mittels der 19-stelligen Attributvektoren statt, die im LARa-Datenset enthalten sind. Als Label dienen die dazugehörigen Sensordaten des Datensets, die durch den Decoder aus den Attributvektoren rekonstruiert werden sollen. Da nur der Decoder vortrainiert wird, ist die Kostenfunktion für das Training der MSE-Loss zwischen den rekonstruierten Daten und den Sensordaten des Datensets.

Nach dem Vortraining des Decoders wird das gesamte Netz, wie in Abschnitt 5.3.2 beschrieben, trainiert. Dazu werden die vortrainierten Gewichte des Decoders vor dem Training geladen, bleiben jedoch beim Training veränderbar. Die Klassifikation der Daten sowie die Berechnung der Performancemetriken finden weiterhin über den berechneten Attributvektor des Encoders statt (siehe 5.2). Die Ergebnisse sind in Tabelle 5.13 dargestellt.

Die Motivation für diese Anpassung liegt in der Vermutung, dass bei dem Experiment aus Abschnitt 5.4.3.1 zum Trainingsbeginn ´schlechte´ Attributvektoren durch den Encoder errechnet werden. Das führt dazu, dass der Decoder zum Trainingsbeginn ´falsche´ Gewichtungen erlernt und ´gute´ Attributvektoren im späteren Verlauf des Trainings zu einer schlechten Rekonstruktion führen, was wiederum die Kostenfunktion negativ beeinflusst.

Tabelle 5.13: Autoencoder: Ergebnisse des Trainings mithilfe eines vortrainierten Decoders

Netz	accuracy	f1 mean	f1 weighted
512x512 Pooling	68.20	57.04	68.82
256x256 Pooling	68.57	57.20	69.16
128x256 ohne Pooling	68.01	58.03	68.55
64x64 Pooling	67.68	59.48	68.23
32x64 Pooling	67.99	56.56	68.28
19x19 ohne Pooling	66.52	49.32	64.12

Gegenüber dem ersten Experiment aus 5.4.3.1 konnten keine wesentlichen Verbesserungen erzielt werden. Die größte Abweichung weist der 19x19 Encoder im f1 mean Score auf. Trotz dieser Veränderung kann dieses Netz die Erkennungsraten der anderen Netze nicht erreichen. Die Veränderungen der anderen Netze liegen im Bereich von 0.3%, was als unwesentlich zu betrachten ist.

5.4.3.3 Training mithilfe eines Decoders und geänderter Zwischendarstellung

In den ersten beiden Experimenten wurde versucht, die Rekonstruktion der Daten durch den Decoder anhand des Attributvektors durchzuführen, der vom Encoder berechnet wurde. Der letzte Tensor des Encoders hat eine Größe von 47880 bis hin zu 645120 Einträgen, die durch das letzte Layer auf 19 Attribute herunter gerechnet werden. Dabei entsteht ein erheblicher Informationsverlust. Daher wird in diesem Experiment geprüft, ob durch Verwendung einer anderen Zwischendarstellung mit mehr Parametern, die Performance des Netzes verbessert werden kann. Als Zwischendarstellung wird die Ausgabe des letzten 1x1 Convolutionlayers des Encoders gewählt. Durch den höheren Informationsgehalt in der Zwischendarstellung, ist die plausible Rekonstruktion der Bewegungsdaten eher möglich. Dies ist wichtig für die angestrebte synthetische Datengenerierung.

Dazu wird die Architektur des Auto-Encoder-Decoder Netzes so angepasst, dass der Decoder den berechneten Tensor des letzten 1x1 Convolution Layers aus dem Encoder als Zwischendarstellung erhält. Der Aufbau des Netzes ist in Abbildung 5.6 vereinfacht dargestellt.

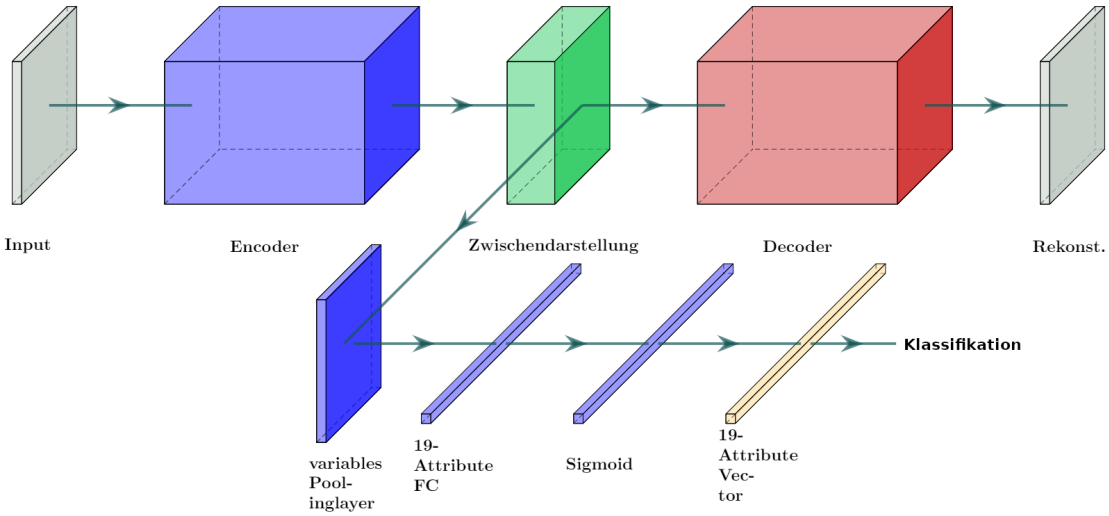

Abbildung 5.6: Vereinfachter Aufbau des Auto-Encoders für das zweite Experiment.

Zur Realisierung dieser Architektur wird im Decoder der erste Deconvolution Layer entfernt, da nun die Auflösung des Attributvektors nicht mehr notwendig ist. Er bezieht seine Eingabe direkt aus der Ausgabe des letzten 1x1 Convolutional Layer des Encoders. Das Fully Connected Layer zur Berechnung des Attributvektors und das variable Poolinglayer des Encoders bleiben jedoch weiterhin für die Klassifikation parallel bestehen. Dies ist notwendig, um weiterhin die Features des Attributvektors zu erlernen und um die bekannten Performancemetriken beizubehalten. Denn die Klassifikation der Daten sowie die Berechnung der Performancemetriken finden weiterhin über den berechneten Attri-

butvektor des Encoders statt (siehe 5.2). Die Performance der verschiedenen getesteten Netzarchitekturen ist in Tabelle 5.14 zu finden.

Tabelle 5.14: Autoencoder: Ergebnisse des Trainings mit einer geänderten Zwischendarstellung

Netz	accuracy	f1 mean	f1 weighted
512x512 Zeit-Pooling	68.33	56.67	68.81
256x256 Zeit-Pooling	68.69	56.61	68.97
128x256 ohne Pooling	68.28	57.26	68.58
64x64 Zeit-Pooling	67.30	58.30	68.10
32x64 Zeit-Pooling	67.80	57.57	68.14
19x19 ohne Pooling	66.94	51.76	65.88

Die Kostenfunktion für das Training entspricht der Formel 4.2. Die berechnete Rekonstruktion des Decoders beeinflusst das Training über den MSE-Loss und der berechnete Attributvektor über den BCE-Loss Teil der Formel. Auch bei diesem Experiment ist in den fünf getesteten FCNs die Tendenz zu erkennen, dass größere Netze eine bessere Performance liefern. Daher wurde auch an dieser Stelle der 512x512 Encoder mit einem Zeit-Pooling noch einmal getestet, jedoch konnte er wieder keine bessere Performance als der 256x256 Encoder erreichen. Mit dieser Trainingsmethode wurde die Performance der drei größten Netze leicht verbessert. Allerdings sank die Performance der kleineren Netze mit Ausnahme des 19x19 Encoders. Überraschenderweise ist der größte Anstieg der Performance beim 19x19 Encoder zu erkennen. Hier verbesserte sich die Accuracy um ca. 0.9% und der F1-Score um ca. 2.1%.

Auch im Vergleich zum Experiment aus Abschnitt 5.4.3.3 zeigen sich keine wesentlichen Änderungen. Alle Netze haben sich positiv im Bezug auf den F1 mean Score entwickelt. Die größten Abweichungen liegen im Bereich von 1%.

5.4.3.4 Zusammenfassung der Ergebnisse zum Training des Encoders mithilfe eines Decoders

Im Durchschnitt lieferte der 256x256 Autoencoder die besten Ergebnisse in den durchgeführten Versuchen. Daher wird dieser als Grundlage für die folgende Analyse verwendet. Die Benennung der Netze erfolgt nach folgendem Schema:

$$Autoenc(Zwischendarstellung, Vortraining) \tag{5.8}$$

Zwischendarstellung nimmt die Werte ´ATT´ für Attributvektor oder ´EZD´ für erweiterte Zwischendarstellung an. Der Parameter *Vortraining* gibt an, ob ein vortrainierter Decoder verwendet wurde (´V´) oder nicht (´NV´). In Tabelle 5.15 sind die Performan-

cedaten des 256x256 Autoencoders der Experimente aus den Kapiteln 5.4.3.1 bis 5.4.3.3 zusammengefasst.

Tabelle 5.15: Performancevergleich 256x256 Autoencoder aus den durchgeführten Experimenten auf Basis des Validierungs-Datensets

Netz	accuracy	f1 mean	f1 weighted
256x256 Autoenc(ATT,NV)	68.53	57.04	69.06
256x256 Autoenc(EZD,NV)	68.69	56.61	68.97
256x256 Autoenc(ATT,V)	68.57	57.20	69.16

Anhand des Testdatensatzes wird die Genauigkeit und die Anzahl der False Positives der einzelnen Klassen ermittelt. Die Ergebnisse zeigen, dass die Klassen ´stehen´, ´Handling oben´ und ´synchronisation´ (c1, c4 und c7) allgemein schlecht von den Netzen klassifiziert werden können. Der 256x256 Autoenc(ATT,V) liefert die besten Ergebnisse, mit Ausnahme der Klassen ´Handling oben´ und ´Handling center´ (c4 und c5). Bei diesem Netz entspricht die Zuordnung der Bewegungen zu den Klassen am ehesten der Verteilung der Daten im Testdatenset, wie Grafik 5.7 zeigt.

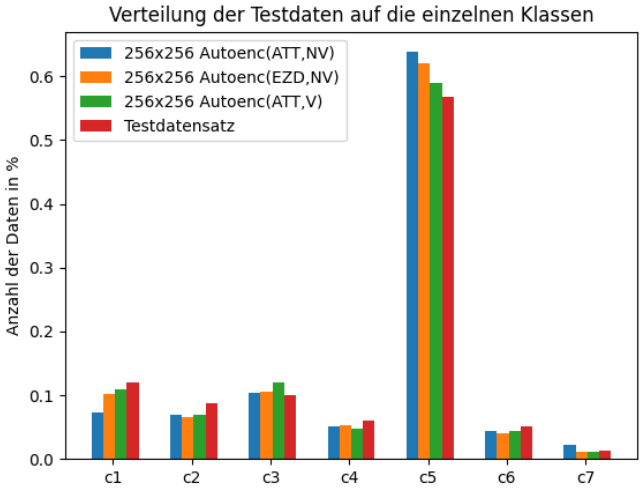

Abbildung 5.7: Verteilung der Daten bezogen auf die einzelnen Klassen. Rot: Tatsächliche Verteilung der Klassen auf dem Test-Datenset. Blau/Orange/Grün: Durch neuronale Netze berechnete Ergebnisse.

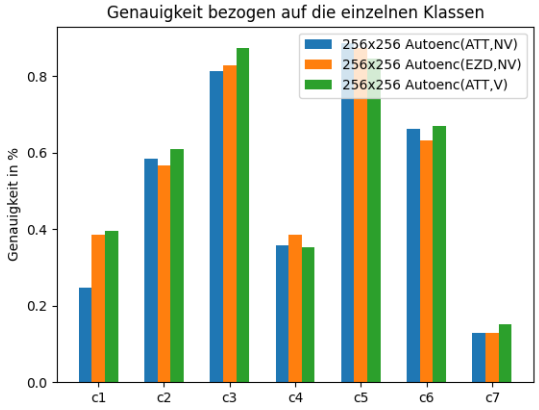

Abbildung 5.8: Genauigkeit der Klassifikation für die einzelnen Aktivitäten auf dem Test-Datensatz

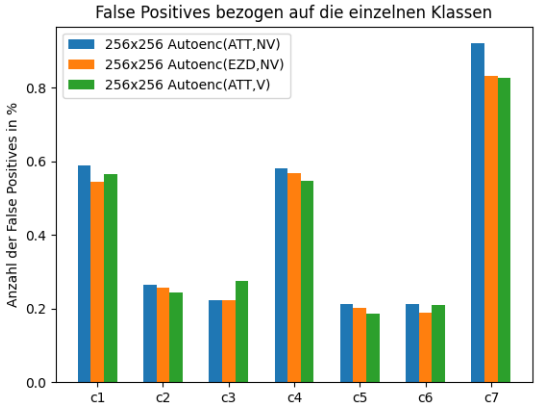

Abbildung 5.9: Anzahl der False Positives in Bezug auf die einzelnen Klassen auf dem Test-Datensatz

Für die weitere Analyse wird der 256x256 Autoenc(ATT,V) mit dem rekonstruierten TCN aus Abschnitt 5.4.1 und dem 128x256 Encoder ohne Pooling aus Abschnitt 5.4.3, welches ohne Decoder trainiert wurde, auf dem Testdatensatz verglichen. In Tabelle 5.16 sind die Netze und ihre Performance auf dem Validierungsdatenset nochmals zur Übersicht aufgeführt.

Tabelle 5.16: Performancevergleich rekonstruiertes TCN, 128x256 Encoder und 256x256 Autoenc(ATT,V) auf Basis des Validierungs-Datensets

Netz	accuracy	f1 mean	f1 weighted
rekonstruiertes TCN (Baseline)	70.93	57.47	69.68
128x256 ohne Pooling	68.49	60.59	69.01
256x256 Autoenc(ATT,V)	68.57	57.20	69.16

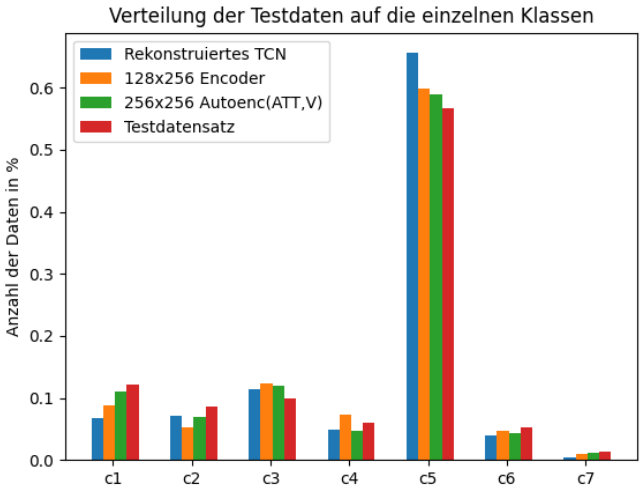

Abbildung 5.10: Verteilung der Daten bezogen auf die einzelnen Klassen. Rot: Tatsächliche Verteilung der Klassen auf dem Test-Datenset. Blau/Orange/Grün: Durch neuronale Netze berechnete Ergebnisse.

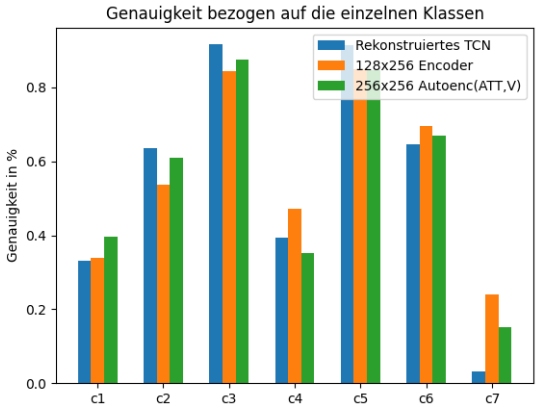

Abbildung 5.11: Genauigkeit der Klassifikation für die einzelnen Aktivitäten auf dem Test-Datensatz

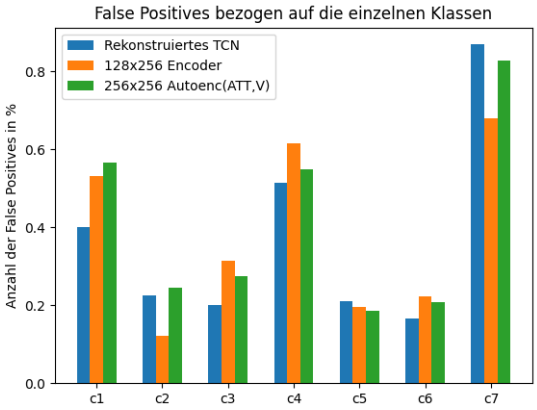

Abbildung 5.12: Anzahl der False Positives in Bezug auf die einzelnen Klassen auf dem Test-Datensatz

Vergleicht man die Performance anhand der berechneten Metriken auf dem Validierungsdatenset (Tabelle 5.16), so erkennt man, dass das TCN weiterhin die beste Performance liefert. Die p-Metrik zeigt, dass das TCN signifikant besser performt als der 256x256 Autoenc(ATT,V). Der p-Wert zwischen den beiden Netzen beträgt auch hier 0.0. Betrachtet man die Anzahl der richtig zugeordneten Klassen (Abbildung 5.11), ist zu beobachten, dass das TCN für die Klassen ´gehen´, ´Wagen´ und ´Handling centered´ (c2,c3 und c5) die meisten richtigen Zuordnungen liefert. Der 256x256 Autoenc(ATT,V) erkennt die Klassen ´stehen´, ´gehen´ und ´Wagen´ (c1, c2 und c3) besser als der 128x256 Encoder. Das

Training mit einem Decoder verbessert die Erkennung in diesen Klassen. Besonders bei der Aktivität ´stehen´ (c1) liefert das Netz die besten Ergebnisse. Bei den Klassen ´Handling oben´, ´Handling unten´ und ´syncronisation´ (c4, c6 und c7) liefert der 256x256 Autoenc(ATT,V) schlechtere Ergebnisse als der 128x256 Encoder, bis auf die Klasse ´Handling oben´ (c4) liegen die Ergebnisse über dem TCN. Aus Abbildung 5.10 geht hervor, dass der 256x256 Autoenc(ATT,V) in Bezug auf die Klassen ´stehen´ (c1) und ´Handling centered´ (c5) die besten Ergebnisse liefert, weil er der Verteilung im Testdatenset am nächsten kommt.

Tabelle 5.17: Confusionsmatrix des 256x256 Autoenc(ATT,V)

Klassen	**c1**	**c2**	**c3**	**c4**	**c5**	**c6**	**c7**
c1	2505	405	306	109	2452	113	451
c2	903	2778	383	66	368	26	30
c3	190	29	4581	0	431	5	1
c4	239	47	95	1109	1631	6	17
c5	1756	747	981	764	25233	332	74
c6	171	24	37	5	660	1828	5
c7	5	1	3	404	187	1	108

Vergleicht man die Confusionsmatrix des 256x256 Autoenc(ATT,V) (Tabelle 5.17) mit denen der anderen Netze (Tabelle 5.11), so erkennt man, dass sich die Differenzierung der Klassen ´Handling oben´, ´Handling centered´ und ´Handling unten´ (c4, c5 und c6), im Gegensatz zum 128x256 Encoder, verschlechtert hat. Grundsätzlich treten die gleichen Zuordnungsprobleme auf, wie sie in Abschnitt 5.4.2.4 bereits dargestellt wurden.

256x256 Autoenc(ATT,V) ordnet die Klassen ´stehen´ und ´Handling centered´ (c1 und c5) besser zu als die anderen Netze. Die Anzahl der falsch klassifizierten Daten aus der Klasse ´stehen´ zur Klasse ´Handling centered´ verringert sich von 54.3% auf 38.66%. Trotzdem erhöht sich die Erkennungsrate für die Aktivität ´stehen´ (c1) nur um 5%.

Durch die hohe Imbalance der Daten scheint der 256x256 Autoenc(ATT,V) auf die gleichen Probleme zu stoßen wie das TCN, obwohl er explizit die räumlichen Features mittels Decoder gelernt hat.

Im Gegensatz zum TCN ordnet der 256x256 Autoenc(ATT,V) nicht alles der Klasse c5 zu, sondern versucht es auf die anderen Klassen zu verteilen. Durch die hohe Anzahl an Bewegungen aus c5 scheint das Netz die räumlichen Features dieser Klasse gelernt zu haben. Es kann die Daten jedoch nicht der richtigen Klasse zuordnen, weil die Features der anderen Bewegungen, aufgrund der Imbalance im Datensatz, nicht ausreichend erlernt wurden.

Aus diesen Erkenntnissen lässt sich folgern, dass der 128x256 Encoder der beste Ansatz zur Bewegungserkennung ist, weil insgesamt die beste Differenzierung der Daten stattfin-

det. Das TCN erreicht die höchste Punktzahl in den Bewertungsmetriken, aber die Confusionsmatrix zeigt, dass die Klassenzuordnung weniger differenziert ist. Die gute Performance wird im Wesentlichen erreicht, weil die schlecht erkannten Klassen einen zu geringen Anteil am Gesamtvolumen haben, sodass Fehlzuordnungen die Performance nicht deutlich beeinflussen. Der 256x256 Autoenc(ATT,V) scheint ebenfalls ein vielversprechender Ansatz zu sein. Jedoch ergeben sich Probleme aufgrund der hohen Imbalance des Datensets.

5.4.4 Visualisierung

Ziel ist es einen Decoder zu entwickeln, der synthetische Daten generieren kann. Dies setzt eine genaue Rekonstruktion der Inputdaten basierend auf einer Zwischendarstellung voraus. Das ist jedoch ohne zeitliche Informationen in der Zwischendarstellung nicht möglich. Für den Encoder wurden daher zum Erhalt der zeitlichen Informationen in Experiment 5.4.2.1 die Fully Connected Layer durch 1x1 Convolutionlayer ersetzt. Dies entspricht dem Vorgehen der Autoren aus [33], die bestehende Netze zum Erhalt der räumlichen Informationen für die semantische Segmentierung anpassten (siehe auch Abschnitt 3.3.1). In Abschnitt 5.4.3 wurde der Einfluss durch ein Training mit einem Decoder analysiert, bei dem gezielt strukturelle Features durch Rekonstruktion der Daten gelernt wurden. Die Architektur des Decoders entstammt der Arbeit [55]. Dort konnte durch die zusätzliche Rekonstruktion der Daten während des Trainings eine Verbesserung in der Featureberechnung erreicht werden, was sich positiv auf die Klassifikation auswirkte.

Weil durch die Berechnung der Zwischendarstellung durch ein Fully Connected Layer die zeitlichen und sensorischen Informationen der Daten verloren gehen, wird der 256x256 Autoencoder aus dem Experiment 5.4.3.3 gewählt. Dieser verwendet eine Zwischendarstellung, die wesentlich mehr Informationen enthält als der Attributvektor.

Um das Ergebnis der Rekonstruktion mit dem Input zu vergleichen, werden diese visuell miteinander verglichen. Weiterhin wird das Netz für die Visualisierung mit dem MoCap-Daten des LARa-Datensets trainiert. Jede der Bewegungen ist mit insgesamt 128 Sensorchanneln beschrieben, wodurch sie genauer aufgezeichnet wurden als beim MbientLab-Datenset.

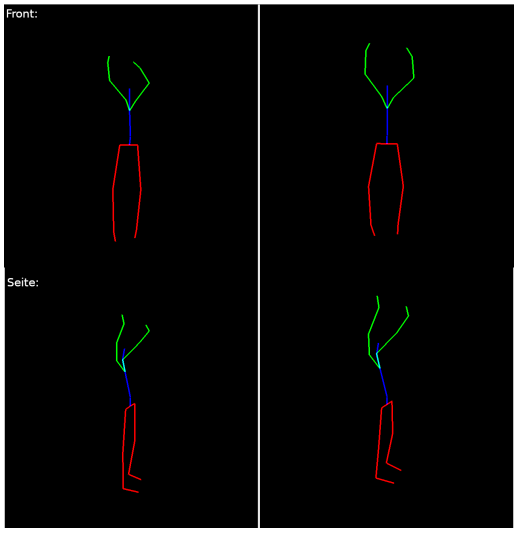

Abbildung 5.13: Bewegung: Synchronisation (Hände ausgestreckt über dem Kopf). Links: Rekonstruktion. Rechts: Original.

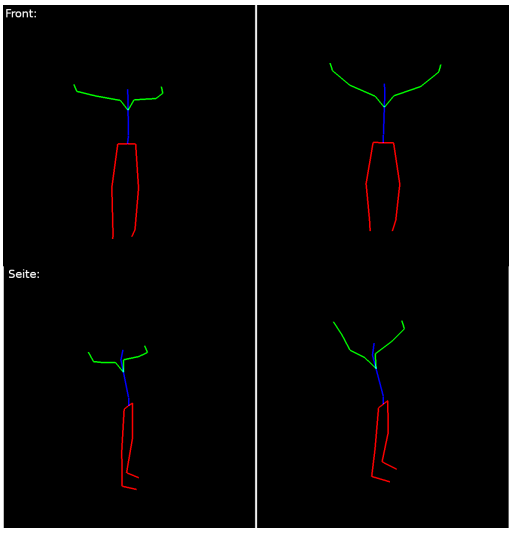

Abbildung 5.14: Bewegung: Synchronisation (Hände ausgestreckt zur Seite). Links: Rekonstruktion. Rechts: Original.

Abbildungen 5.13 und 5.14 zeigen zwei verschiedene Zeitpunkte aus der gleichen Bewegung (Synchronisation) von der Front und von der Seite. Es ist deutlich zu erkennen, dass das neuronale Netz die Inputdaten sehr gut rekonstruiert hat. Die größten Unterschiede sind leichte Veränderungen in der Ausrichtung der Arme, jedoch bleibt die ausgeführte Aktivität sehr deutlich erkennbar.

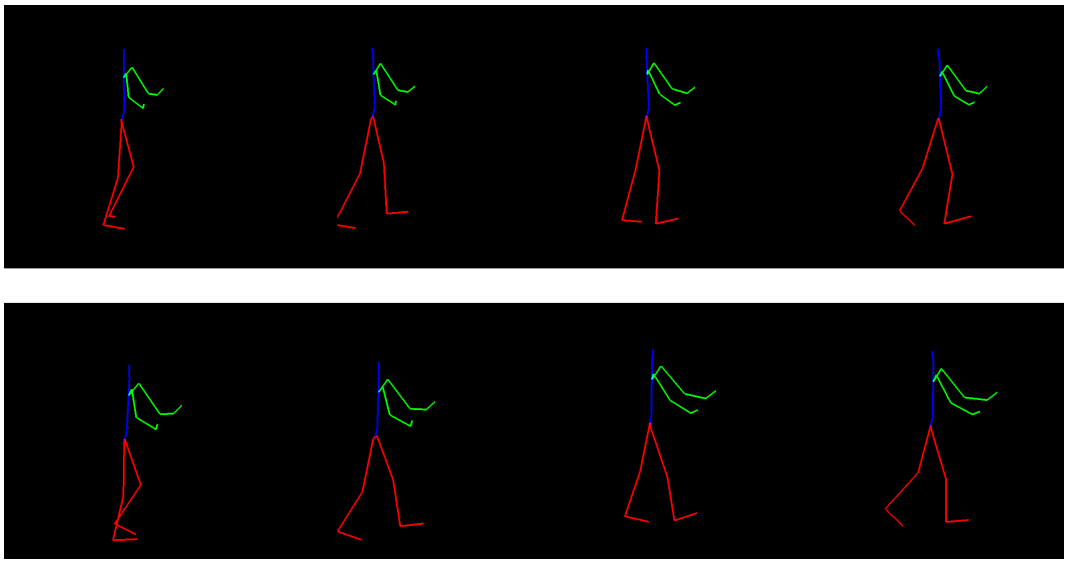

Abbildung 5.15: Bewegungsablauf: Gehen. Oben: Rekonstruktion. Unten: Original.

Abbildung 5.15 zeigt zeitlich aufeinanderfolgende Frames der Aktivität ´Gehen´. Auch hier sind die rekonstruierten Daten des Autoencoders kaum unterscheidbar von den Origi-

naldaten. Aus dem visuellen Vergleich der Daten lässt sich schließen, dass der Autoencoder sehr plausible Daten aus der Zwischendarstellung berechnet.

5.4.5 Synthetische Datengenerierung

Mit den Erkenntnissen aus den vorherigen Experimenten soll ein Fully Convolutional Network entwickelt werden, das zur synthetischen Datengenerierung eingesetzt werden kann. Wie in Abschnitt 5.4.4 bereits gezeigt, ist es möglich, mit einem Decoder aus einer Zwischendarstellung eine gute Rekonstruktion der Daten zu berechnen. Ziel ist es nun, eine Zwischendarstellung zu generieren, die auf den semantischen Attributen einer Bewegung basiert und für den Menschen interpretierbar ist. Berechnet man den Attributvektor wie in den vorherigen Experimenten mit einem Fully Connected Layer, bleibt die Zwischendarstellung weiterhin uninterpretierbar. Ziel ist es daher ein Poolinglayer zur Bestimmung des Attributvektors zu benutzen, damit für die Zwischendarstellung ein interpretierbarer Zustand erlernt wird. Dadurch erhält man strukturell ein Fully Convolutional Network und es wird möglich, die Rekonstruktion der Daten logisch zu beeinflussen, indem ein Attribut in der Zwischendarstellung verändert wird. Bei den berechneten Rekonstruktionen der manuell geänderten Zwischendarstellung handelt es sich um synthetische Daten. Die Netzarchitektur des Fully Convolutional Networks ist in Abbildung 5.16 dargestellt.

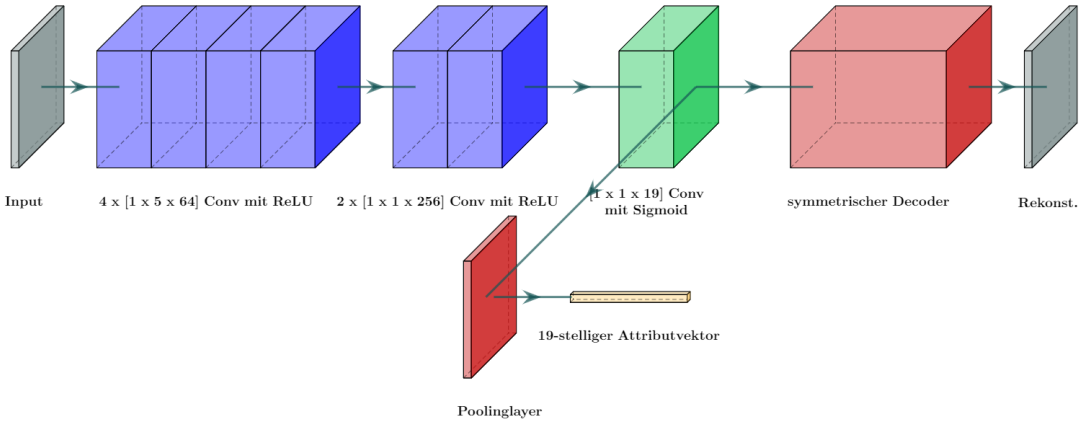

Abbildung 5.16: Aufbau des Fully Convolutional Networks zur synthetischen Datengenerierung.

Der Encoder enthält vier Convolutional Layer zum Extrahieren der zeitlichen Features und zwei 1x1 Convolution Layer mit 256 Featuremaps, um die Daten tiefer zu analysieren. Diese Struktur ergibt sich aus den vorherigen Experimenten, weil damit die besten Versuchsergebnisse erzielt wurden.

Darauf folgt ein 1x1 Convolution Layer mit insgesamt 19 Featuremaps. Dieses Layer soll für jeden Datenpunkt einen 19-stelligen Attributvektor in der Featureebene berechnen. Der Output dient als Zwischendarstellung des Netzwerks, aus welcher der symmetrisch konstruierte Decoder die Rekonstruktion der Daten erlernen soll.

Parallel zur Rekonstruktion des Decoders wird basierend auf der Zwischendarstellung ein Attributvektor berechnet. Dieser wird durch ein Pooling über die gesamte Zwischendarstellung generiert. Das ist notwendig, um die Attribute in der Zwischendarstellung zu erlernen.

Beim Training des Fully Convolutional Networks zeigte sich, dass diese Netzarchitektur keine geeignete Attributrepräsentation erlernen konnte. Für alle Daten wurde der gleiche Attributvektor berechnet, welcher der Klasse ´handling centered´ entspricht. Nur dadurch erreichte das Netz eine Accuracy von ca. 40%, was dem Anteil der Klasse ´handling centered´ im Datensatz entspricht. Es besteht die Vermutung, dass es aufgrund der Destruktivität des Poolinglayers nicht möglich ist, während des Trainings geeignete Gradienten zur Optimierung der Gewichte der Layer zu berrechnen. Dadurch erlernt das Netz weder die Repräsentation der Attribute in der Zwischendarstellung, noch andere Features, die zur Unterscheidung der Daten notwendig wären.

Im Rahmen des Experiments 5.4.3.3 wurde ein Netzwerk getestet, das eine Zwischendarstellung mit 19 Featuremaps verwendet. Jedoch wurde der Attributvektor dort mit einem Fully Connected Layer basierend auf der Zwischendarstellung berechnet. Da in dem Experiment 5.4.3.3 eine höhere Genauigkeit in der Klassifikation als beim Fully Convolutional Network erzielt werden konnte, lässt sich schließen, dass ein Poolinglayer nicht zur Berechnung eines Attributvektors geeignet ist. Dementsprechend ist auch die erlernte Zwischendarstellung des Fully Convolutional Networks für die synthetische Datengenerierung unbrauchbar.

Da das Fully Convolutional Network nur mit einem Poolinglayer keine Berechnung von Attributvektoren erlernen kann, wird im Folgenden das Netzwerk um einen ´Klassifikationskopf´ erweitert. Er besteht aus einem Zeitpooling und einem Fully Connected Layer, welches zusätzlich zur Berechnung der Attributvektoren eingesetzt wird. Dieser wird am zweiten 1x1 Convolutionlayer des Encoders integriert. Die Architektur der Erweiterung ergibt sich ebenfalls aus den vorherigen Experimenten aufgrund der erreichten Performance.

Weiterhin wird die Zwischendarstellung vergrößert, indem die Ausgaben der letzen beiden 1x1 Convolution Layer miteinander konkateniert werden. Dadurch enthält die Zwischendarstellung für jeden Datenpunkt 256 Features und 19 Attribute.

Zusätzlich findet das Training des Netzes in zwei Stufen statt. Zuerst werden die Layer des Encoders, die zur Ermittlung der Features dienen, mit dem Klassifikationskopf vortrainiert. In einem zweiten Schritt, wird das letzte 1x1 Convolution Layer mit 19 Featuremaps gezielt darauf trainiert die Attribute für die Datenpunkte zu berechnen. Anders als beim Fully Convolutional Network werden die, durch den Klassifikationskopf und das Pooling berechneten, Attributvektoren für den zweiten Trainingsschritt konkateniert und mit einem Average-Pooling zu einem Gesamtattributvektor zusammengefasst. Dieser Attributvektor wird für die Klassifikation des Netzwerks verwendet und dient als Ausgangspunkt zur Berechnung der Metriken. Die erweiterte Netzarchitektur ist in Abbildung 5.17 zu finden.

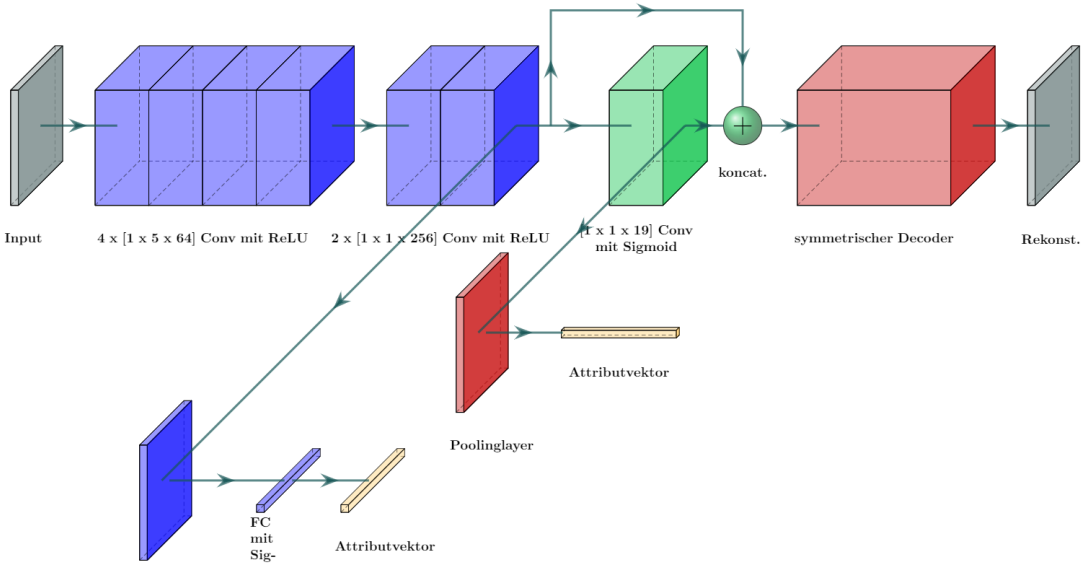

Abbildung 5.17: Aufbau des angepassten Netzwerks zur synthetischen Datengenerierung.

Tabelle 5.18: Performances nach dem ersten und zweitem Trainingsschritt basierend auf dem Testdatenset

Trainingsschritt	accuracy	f1 mean	f1 weighted
Training mit Klassifikationskopf	70.98	62.25	70.71
Training mit Klassifikationskopf und Pooling	68.20	60.26	68.38

Die Performance des Netzes bei der Klassifikation der Daten (siehe Tabelle 5.18) lässt darauf schließen, dass im Gegensatz zum Fully Convolutional Network das Netz erlernt hat, Features aus den Daten zu extrahieren und Attributvektoren zu berechnen. Jedoch fällt bei genauerer Betrachtung der berechneten Zwischendarstellungen auf, dass auch hier das 1x1 Convolutional Layer mit 19 Featuremaps des Encoders keine geeigneten Attributvektoren für die Datenpunkte erlernt. Stattdessen beträgt jeder Wert des berechneten Tensors, der Attributvektoren enthalten sollte, ca. 0.5. Dies entspricht einer Wahrscheinlichkeit von 50% für jedes Attribut, dass es in dem Attributvektor auftritt. Daraus lässt sich schließen, dass auch dieses Netz keine veränderbare Zwischendarstellung basierend auf den semantischen Attributen der Bewegungen erlernt hat.

Die These wurde anhand eines Beispiels überprüft. Die Zwischendarstellung einer Synchronisationsbewegung wurde so verändert, dass sie einer anderen Klasse ohne Armbewegungen entsprechen sollte. Jedoch zeigt das Ergebnis in Abbildung 5.18, dass in den rekonstruierten Daten dennoch Armbewegungen vorhanden sind. Ebenfalls ist zu erken-

nen, dass die Veränderung der Attribute einen ungewollten Einfluss auf die Körperposition hat.

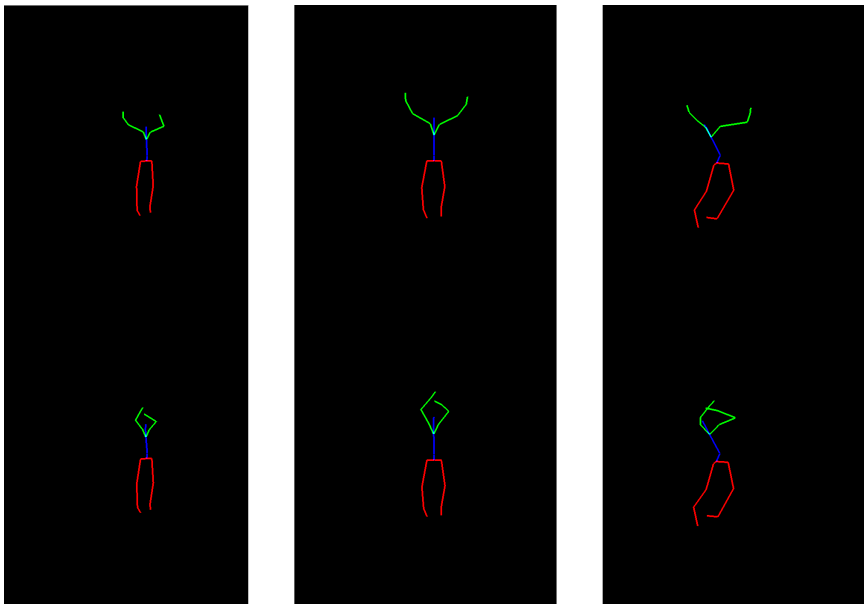

Abbildung 5.18: Bewegung: Synchronisation. Links: Rekonstruktion. Mitte: Original. Rechts: Rekonstruktion nach Anpassung der Attribute

Auch dieses Ergebnis lässt darauf schließen, dass das Netz keine attributgesteuerte Rekonstruktion der Daten erlernt hat.

In diesem Abschnitt wurde eine weitere Netzarchitektur erstellt, die Bewegungen plausibel rekonstruieren kann. Allerdings konnte keine, auf den semantischen Attributen basierende, Zwischendarstellung der Daten generiert werden, die zur Erzeugung plausibler synthetischer Daten benutzt werden kann. Die Experimente in diesem Abschnitt zeigen, dass das Problem vermutlich an dem eingesetzten Poolinglayer zur Berechnung der Attributvektoren liegt. Durch die Größe des Kernels und die damit einhergehende Destruktivität auf den Daten scheinen beim Training keine geeigneten Gradienten zur Anpassung der Gewichte berechnet zu werden. Jedoch kann aus zeitlichen Gründen das Problem in dieser Arbeit nicht weiter verfolgt werden.

Kapitel 6

Zusammenfassung

Ziel der Bachelorarbeit war es, ein Fully Convolutional Network auf Basis eines Autoencoders zu entwickeln, mit dem synthetische HAR-Daten generiert werden können.

Dazu wurden sowohl Experimente zur Struktur des Encoders durchgeführt, um die bestmögliche Featureextraktion zu gewährleisten, als auch Experimente zum Training mithilfe eines Decoders, um den Einfluss eines solchen auf das Training zu analysieren. Dabei konnte herausgestellt werden, dass die imbalancierten Testdaten einen schlechten Einfluss auf die Performance der Netzarchitekturen genommen hat. Dadurch konnten die Netze einige Klassen nur schwer erkennen. Ein Lösungsansatz ist die Verwendung gleichverteilter Trainingsbatches, in denen alle Bewegungsklassen zu gleichen Teilen vertreten sind. Eine andere Möglichkeit den Einfluss der imbalancierten Daten zu minimieren, ist die Berechnung der Gewichte abhängig von der Verteilung der Daten im Trainingsdatenset zu gestalten. So beeinflussen häufig vorkommende Klassen die Gewichte wenig, seltene Klassen hingegen mehr.

Mit dem rekonstruierten TCN aus Abschnitt 5.4.1 konnten state-of-the-art Ergebnisse erreicht werden. Trotz der negativen Einflüsse, basierend auf der hohen Imbalance der Trainingsdaten, war es möglich, die Inputdaten durch einen Autoencoder sehr gut zu rekonstruieren. Einen wichtigen Teil trug dazu der Erhalt der räumlichen Features bei, die durch das Entfernen der Fully Connected Layer in die Berechnung der Zwischendarstellung eingingen. Dieser Schritt ist ein wichtiger Teilerfolg auf dem Weg synthetische Bewegungsdaten zu generieren. Allerdings war es nicht möglich, ein reines Fully Convolutional Network zu entwickeln, mit dem synthetische Daten generiert werden konnten. Während des Trainings erlernte die Fully Convolutional Network Architektur weder eine Differenzierung der Daten, noch konnte eine Zwischendarstellung basierend auf den semantischen Attributen einer Bewegung erlernt werden. Durch Erweiterung der Architektur um einen zusätzlichen, unterstützenden Fully Connected Layer zur Berechnung eines Attributvektors, erlernte das Netz zwar Features zur Differenzierung der Daten, jedoch weiterhin keine attributgesteuerte Zwischendarstellung. Durch manuelle Veränderung der Zwischendarstellung konnten die

Bewegungsabläufe nicht gezielt verändert werden. Allerdings entstand dadurch ein Rauschen, was durchaus zur synthetischen Datengenerierung eingesetzt werden kann.

Trotz des Misserfolgs, synthetische Daten durch eine attributgesteuerte Zwischendarstellung mit einem Fully Convolution Netzwerk zu erzeugen, können zukünftige Arbeiten die Ergebnisse zur Rekonstruktion der Daten durch einen Autoencoder verwenden. So gilt es herauszufinden, ob durch zufälliges Rauschen auf der Zwischendarstellung oder Vertauschen der Sensorchannel in den Eingabedaten, plausible synthetische Daten generiert werden können.

Abbildungsverzeichnis

2.1 Oben: Aufbau eines biologischen Neurons entnommen aus[54]. Unten: McCulloch-Pitts-Neuron mit θ als Aktivierungswert entnommen aus [11] 6
2.2 Bild eines single-layer Perzeptrons entnommen aus [41]. 7
2.3 Bild eines Multi Layer Perzeptrons. Das MLP besitzt ein Inputlayer (grün), bestehend aus vier Neuronen, zwei Hiddenlayer (blau), bestehend aus jeweils 5 Neuronen und ein Outputlayer (rot), bestehend aus einem Neuron. 8
2.4 Die Heaviside-Stufenfunktion. 10
2.5 Die lineare Aktivierungsfunktion mit a = 1. 10
2.6 Die Sigmoid Funktion. 11
2.7 Die Tanh Aktivierungsfunktion. 11
2.8 Die Rectified Linear Unit Aktivierungsfunktion. 12
2.9 Links: Annäherung an ein Minimum mit einer zu großen Lernrate. Rechts: Annäherung an ein Minimum mit einer zu kleinen Lernrate. Bild entnommen aus [2] 14
2.10 Beispielhafte Berechnung einer Featuremap durch Faltung. Bild entnommen aus [22]. 18
2.11 Beispielhafte Anwendung der Max-Pooling Operation. Abbildung entnommen aus [12]. 19
2.12 Oben: Standardfaltung, Unten: kausale Faltung. Bilder entnommen aus [28]. 21
2.13 Darstellung eines Auto-Encoder-Decoders. Abbildung entnommen aus [18]. 22

3.1 TCN aus der Arbeit von Yang et al. [57]. ´D´ steht für die Anzahl der Sensoren. Die Buchstaben in den Klammern stehen für die verschiedenen Operationen. ´c´ = convolution, ´s´ = subsampling/pooling, ´u´ = unification (Zusammenfassen der Featuremaps aus vorherigem Layer), ´o´ = output. Bild entnommen aus [57]. 30
3.2 TCN des LARa-Datensets. Bild entnommen aus [43]. 32
3.3 Aufbau der Auto-Set Netzwerkarchitektur. Grau: Encoder. Rot: Für den ersten Trainingsschritt verwendeter Decoder. Blau: Klassifizierungskopf. Bild entnommen aus [55]. 36

3.4 Überblick über die verwendeten Netzwerkarchitekturen. Bild entnommen aus [55]. 37

4.1 Aufbau der Temporal Convolutional Network Architektur aus [43]. 40
4.2 Aufbau der Fully Convolutional Network Architektur. Hellblau: Vier Convolutional Layer mit 5x1 Kernel und 64 berechneten Featuremaps. Dunkelblau: 1x1 Convolutional Layer. Grün: 1x1 Convolutional Layer zur Berechnung der Attribute mit Sigmoid Aktivierungsfunkton. Rot: Poolinglayer zur Berechnung eines Attributvektors. Gelb: 19-stelliger Attributvektor. 41
4.3 Aufbau des Decoders. X und Y stehen repräsentativ für die variable Anzahl der Featuremaps basierend auf den 1x1 Convolutional Layern des verwendeten Encoders aus 4.2. Rot: Deconvolutionlayer analog zu den 1x1 Convolutionlayern des Encoders. Magenta: Deconvolutionlayer analog zu den 5x1 Convolutionlayern des Encoders. Grün: Sigmoid Aktivierungsfunktion. . . . 43

5.1 Aufbau der Encoder Architektur mit einem zusätzlichen Poolinglayer. Hellblau: Vier Convolutional Layer mit 5x1 Kernel und 64 berechneten Featuremaps. Dunkelblau: 1x1 Convolutional Layer. Rot: Zusätzliches Poolinglayer. Gelb: Fully Connected Layer mit 19 Neuronen. Grün: Sigmoid Aktivierungsfunktion. 54
5.2 Verteilung der Daten bezogen auf die einzelnen Klassen. Rot: Tatsächliche Verteilung der Klassen im Test-Datenset. Blau/Orange/Grün: Durch neuronale Netze berechnete Ergebnisse. 58
5.3 Genauigkeit der Klassifikation für die einzelnen Aktivitäten auf dem Test-Datensatz 58
5.4 Anzahl der False Positives in Bezug auf die einzelnen Klassen auf dem Test-Datensatz 58
5.5 Vereinfachter Aufbau der Auto-Encoder Network Architektur des Experiments. 61
5.6 Vereinfachter Aufbau des Auto-Encoders für das zweite Experiment. 63
5.7 Verteilung der Daten bezogen auf die einzelnen Klassen. Rot: Tatsächliche Verteilung der Klassen auf dem Test-Datenset. Blau/Orange/Grün: Durch neuronale Netze berechnete Ergebnisse. 65
5.8 Genauigkeit der Klassifikation für die einzelnen Aktivitäten auf dem Test-Datensatz 66
5.9 Anzahl der False Positives in Bezug auf die einzelnen Klassen auf dem Test-Datensatz 66
5.10 Verteilung der Daten bezogen auf die einzelnen Klassen. Rot: Tatsächliche Verteilung der Klassen auf dem Test-Datenset. Blau/Orange/Grün: Durch neuronale Netze berechnete Ergebnisse. 67

5.11 Genauigkeit der Klassifikation für die einzelnen Aktivitäten auf dem Test-Datensatz 67
5.12 Anzahl der False Positives in Bezug auf die einzelnen Klassen auf dem Test-Datensatz 67
5.13 Bewegung: Synchronisation (Hände ausgestreckt über dem Kopf). Links: Rekonstruktion. Rechts: Original. 70
5.14 Bewegung: Synchronisation (Hände ausgestreckt zur Seite). Links: Rekonstruktion. Rechts: Original. 70
5.15 Bewegungsablauf: Gehen. Oben: Rekonstruktion. Unten: Original. 70
5.16 Aufbau des Fully Convolutional Networks zur synthetischen Datengenerierung. 71
5.17 Aufbau des angepassten Netzwerks zur synthetischen Datengenerierung. . . 73
5.18 Bewegung: Synchronisation. Links: Rekonstruktion. Mitte: Original. Rechts: Rekonstruktion nach Anpassung der Attribute 74

Literaturverzeichnis

[1] Bai, Shaojie, J. Zico Kolter und Vladlen Koltun: *An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling*, 2018.

[2] Bhattarai, Saugat: *LEARNING-RATE-GRADIENT-DESCENT*. 2018. `https://saugatbhattarai.com.np/what-is-gradient-descent-in-machine-learning/learning-rate-gradient-descent/`.

[3] Bulling, Andreas, Ulf Blanke und Bernt Schiele: *A Tutorial on Human Activity Recognition Using Body-Worn Inertial Sensors*. ACM Computing Surveys, 46, 01 2013.

[4] Bulling, Andreas, Ulf Blanke und Bernt Schiele: *A Tutorial on Human Activity Recognition Using Body-Worn Inertial Sensors*. ACM Comput. Surv., 46(3), Januar 2014.

[5] Bulling, Andreas, Ulf Blanke und Bernt Schiele: *A tutorial on human activity recognition using body-worn inertial sensors*. ACM Computing Surveys (CSUR), 46(3):1–33, 2014.

[6] Bushaev, Vitaly: *Understanding RMSprop — faster neural network learning*. 2018. `https://towardsdatascience.com/understanding-rmsprop-faster-neural-network-learning-62e116fcf29a`.

[7] Bustoni, Isna Alfi, Indra Hidayatulloh, A Ningtyas, A Purwaningsih und S Azhari: *Classification methods performance on human activity recognition*. Journal of Physics: Conference Series, 1456:012027, 01 2020.

[8] Cao, Hong, Minh Nhut Nguyen, Clifton Phua, Shonali Krishnaswamy und Xiao-Li Li: *An Integrated Framework for Human Activity Classification*. In: *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, Seite 331–340, New York, NY, USA, 2012. Association for Computing Machinery.

[9] Cao, Hong, Minh Nhut Nguyen, Clifton Phua, Shonali Krishnaswamy und Xiao-Li Li: *An Integrated Framework for Human Activity Recognition*. In: *Proceedings*

of the 2012 ACM Conference on Ubiquitous Computing, UbiComp '12, Seite 621–622, New York, NY, USA, 2012. Association for Computing Machinery.

[10] Chamroukhi, F., S. Mohammed, D. Trabelsi, L. Oukhellou und Y. Amirat: *Joint segmentation of multivariate time series with hidden process regression for human activity recognition.* Neurocomputing, 120:633–644, 2013. Image Feature Detection and Description.

[11] Chandra, Akshay L: *McCulloch-Pitts Neuron — Mankind's First Mathematical Model Of A Biological Neuron.* 2018. `https://towardsdatascience.com/mcculloch-pitts-model-5fdf65ac5dd1`.

[12] Cornelisse, Daphne: *An intuitive guide to Convolutional Neural Networks.* 2018. `https://www.freecodecamp.org/news/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050/`.

[13] Creswell, Antonia, Kai Arulkumaran und Anil A. Bharath: *On denoising autoencoders trained to minimise binary cross-entropy.* 2017.

[14] Datta, Leonid: *A Survey on Activation Functions and their relation with Xavier and He Normal Initialization*, 2020.

[15] Deng, Liqun, Howard Leung, Naijie Gu und Yang Yang: *Generalized Model-Based Human Motion Recognition with Body Partition Index Maps.* In: *Computer Graphics Forum*, Band 31, Seiten 202–215. Wiley Online Library, 2012.

[16] El Moudden, Ismail, Benyacoub und Souad El bernoussi: *Modeling Human Activity Recognition by Dimensionality Reduction Approach.* 05 2016.

[17] Faria, Igor Lopes de und Vaninha Vieira: *A Comparative Study on Fitness Activity Recognition.* In: *Proceedings of the 24th Brazilian Symposium on Multimedia and the Web*, WebMedia '18, Seite 327–330, New York, NY, USA, 2018. Association for Computing Machinery.

[18] Francois Chollet: *Building Autoencoders in Keras.* 2016. [Online; accessed Juni 28, 2021].

[19] Fukushima, K.: *Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position.* Biological Cybernetics, 36:193–202, 2004.

[20] Gao, Yongbin, Xuehao Xiang, Naixue Xiong, Huang Bo, Hyo Jong Lee, Rad Alrifai, Xiaoyan Jiang und Zhijun Fang: *Human Action Monitoring for Healthcare Based on Deep Learning.* IEEE Access, 6:1–1, 09 2018.

[21] Gjoreski, Hristijan, Mitja Lustrek und Matjaz Gams: *Accelerometer Placement for Posture Recognition and Fall Detection.* In: *2011 Seventh International Conference on Intelligent Environments*, Seiten 47–54, 2011.

[22] Goodfellow, Ian, Yoshua Bengio und Aaron Courville: *Deep Learning.* MIT Press, 2016. `http://www.deeplearningbook.org`.

[23] Hinton, Geoffrey, Nitish Srivastava und Kevin Swersky: *Neural networks for machine learning lecture 6a overview of mini-batch gradient descent.* Cited on, 14(8):2, 2012.

[24] Hochreiter, S., Y. Bengio, P. Frasconi und J. Schmidhuber: *Gradient flow in recurrent nets: the difficulty of learning long-term dependencies.* In: Kremer, S. C. und J. F. Kolen (Herausgeber): *A Field Guide to Dynamical Recurrent Neural Networks.* IEEE Press, 2001.

[25] Hochreiter, Sepp: *The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions.* International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems, 6:107–116, 04 1998.

[26] Hubel, David H und Torsten N Wiesel: *Receptive fields of single neurones in the cat's striate cortex.* The Journal of physiology, 148(3):574–591, 1959.

[27] Khan, A. M., Y.-K. Lee, S. Y. Lee und T.-S. Kim: *Human Activity Recognition via an Accelerometer-Enabled-Smartphone Using Kernel Discriminant Analysis.* In: *2010 5th International Conference on Future Information Technology*, Seiten 1–6, 2010.

[28] Klaas, Jannes: *Machine Learning for Finance.* Packt, 2019.

[29] Kong, Yu und Yun Fu: *Human Action Recognition and Prediction: A Survey*, 2018.

[30] Kwapisz, Jennifer R., Gary M. Weiss und Samuel A. Moore: *Activity Recognition Using Cell Phone Accelerometers.* SIGKDD Explor. Newsl., 12(2):74–82, März 2011.

[31] LeCun, Y., B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard und L. D. Jackel: *Backpropagation Applied to Handwritten Zip Code Recognition.* Neural Computation, 1(4):541–551, 12 1989.

[32] LeCun, Yann, Y. Bengio und Geoffrey Hinton: *Deep Learning.* Nature, 521:436–44, 05 2015.

[33] Long, Jonathan, Evan Shelhamer und Trevor Darrell: *Fully Convolutional Networks for Semantic Segmentation*, 2015.

[34] Long, Xi, Bin Yin und Ronald M Aarts: *Single-accelerometer-based daily physical activity classification.* In: *2009 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, Seiten 6107–6110. IEEE, 2009.

[35] Lu, Lu: *Dying ReLU and Initialization: Theory and Numerical Examples.* Communications in Computational Physics, 28(5):1671–1706, Jun 2020.

[36] Maas, Andrew L, Awni Y Hannun, Andrew Y Ng et al.: *Rectifier nonlinearities improve neural network acoustic models.* Citeseer.

[37] Margarito, Jenny, Rim Helaoui, Anna M. Bianchi, Francesco Sartor und Alberto G. Bonomi: *User-Independent Recognition of Sports Activities From a Single Wrist-Worn Accelerometer: A Template-Matching-Based Approach.* IEEE Transactions on Biomedical Engineering, 63(4):788–796, 2016.

[38] Mastromichalakis, Stamatis: *ALReLU: A different approach on Leaky ReLU activation function to improve Neural Networks Performance*, 2021.

[39] Mcculloch, Warren und Walter Pitts: *A Logical Calculus of Ideas Immanent in Nervous Activity.* Bulletin of Mathematical Biophysics, 5:127–147, 1943.

[40] Minsky, Marvin und Seymour A Papert: *Perceptrons: An introduction to computational geometry.* MIT press, 2017.

[41] Müller, Fabian: *Das Rosenblatt Perzeptron – die frühen Anfänge des Deep Learnings.* 2017. `https://www.statworx.com/de/blog/das-rosenblatt-perzeptron-die-fruehen-anfaenge-des-deep-learnings/`.

[42] Nielsen, Michael A: *Neural networks and deep learning*, Band 25. Determination press San Francisco, CA, 2015.

[43] Niemann, Friedrich, Christopher Reining, Fernando Moya Rueda, Nilah Ravi Nair, Janine Anika Steffens, Gernot A. Fink und Michael ten Hompel: *LARa: Creating a Dataset for Human Activity Recognition in Logistics Using Semantic Attributes.* Sensors, 20(15), 2020.

[44] Ordóñez, Francisco Javier und Daniel Roggen: *Deep Convolutional and LSTM Recurrent Neural Networks for Multimodal Wearable Activity Recognition.* Sensors, 16(1), 2016.

[45] O'Shea, Keiron und Ryan Nash: *An Introduction to Convolutional Neural Networks*, 2015.

[46] Patterson, Josh und Adam Gibson: *Deep learning: A practitioner's approach.* Ö'Reilly Media, Inc.", 2017.

[47] Pedamonti, Dabal: *Comparison of non-linear activation functions for deep neural networks on MNIST classification task*, 2018.

[48] Reining, Christopher, Friedrich Niemann, Fernando Moya Rueda, Gernot A. Fink und Michael ten Hompel: *Human Activity Recognition for Production and Logistics—A Systematic Literature Review.* Information, 10(8), 2019.

[49] Rosenblatt, Frank: *The perceptron: a probabilistic model for information storage and organization in the brain.* Psychological review, 65(6):386, 1958.

[50] Rueda, Fernando Moya und Gernot A. Fink: *Learning Attribute Representation for Human Activity Recognition*, 2018.

[51] Rueda, Fernando Moya und Gernot A. Fink: *From Human Pose to On-Body Devices for Human-Activity Recognition.* In: *2020 25th International Conference on Pattern Recognition (ICPR)*, Seiten 10066–10073, 2021.

[52] Scheurer, Sebastian, Salvatore Tedesco, Kenneth N Brown und Brendan O'Flynn: *Human activity recognition for emergency first responders via body-worn inertial sensors.* In: *2017 IEEE 14th International Conference on Wearable and Implantable Body Sensor Networks (BSN)*, Seiten 5–8. IEEE, 2017.

[53] Sharma, Sagar und Simone Sharma: *Activation functions in neural networks.* Towards Data Science, 6(12):310–316, 2017.

[54] Strecker, Stefan: *Künstliche Neuronale Netze: Aufbau und Funktionsweise.* Universitätsbibliothek, 2004.

[55] Varamin, Alireza Abedin, Ehsan Abbasnejad, Qinfeng Shi, Damith Ranasinghe und Hamid Rezatofighi: *Deep Auto-Set: A Deep Auto-Encoder-Set Network for Activity Recognition Using Wearables*, 2018.

[56] Werbos, Paul und Paul John: *Beyond regression : new tools for prediction and analysis in the behavioral sciences /.* 01 1974.

[57] Yang, Jianbo, Minh Nhut Nguyen, Phyo Phyo San, Xiao Li Li und Shonali Krishnaswamy: *Deep convolutional neural networks on multichannel time series for human activity recognition.* In: *Twenty-fourth international joint conference on artificial intelligence*, 2015.

[58] Zeng, Ming, Le T Nguyen, Bo Yu, Ole J Mengshoel, Jiang Zhu, Pang Wu und Joy Zhang: *Convolutional neural networks for human activity recognition using mobile sensors.* In: *6th international conference on mobile computing, applications and services*, Seiten 197–205. IEEE, 2014.

[59] Zhou, Shuchang, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen und Yuheng Zou: *DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients*, 2018.

Eidesstattliche Versicherung

(Affidavit)

Krön, Dennis
Name, Vorname
(surname, first name)

192456
Matrikelnummer
(student ID number)

☒ Bachelorarbeit
(Bachelor's thesis)

☐ Masterarbeit
(Master's thesis)

Titel
(Title)

Human Activity Recognition mithilfe eines Fully Convolutional Networks am Beispiel des LARa Datensets

Ich versichere hiermit an Eides statt, dass ich die vorliegende Abschlussarbeit mit dem oben genannten Titel selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

I declare in lieu of oath that I have completed the present thesis with the above-mentioned title independently and without any unauthorized assistance. I have not used any other sources or aids than the ones listed and have documented quotations and paraphrases as such. The thesis in its current or similar version has not been submitted to an auditing institution before.

Iserlohn, 21.12.2021
Ort, Datum
(place, date)

D. Krön
Unterschrift
(signature)

Belehrung:
Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -).

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird ggf. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin") zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Official notification:
Any person who intentionally breaches any regulation of university examination regulations relating to deception in examination performance is acting improperly. This offense can be punished with a fine of up to EUR 50,000.00. The competent administrative authority for the pursuit and prosecution of offenses of this type is the Chancellor of TU Dortmund University. In the case of multiple or other serious attempts at deception, the examinee can also be unenrolled, Section 63 (5) North Rhine-Westphalia Higher Education Act (*Hochschulgesetz, HG*).

The submission of a false affidavit will be punished with a prison sentence of up to three years or a fine.

As may be necessary, TU Dortmund University will make use of electronic plagiarism-prevention tools (e.g. the "turnitin" service) in order to monitor violations during the examination procedures.

I have taken note of the above official notification:*

Iserlohn, 21.12.2021
Ort, Datum
(place, date)

D. Krön
Unterschrift
(signature)

***Please be aware that solely the German version of the affidavit ("Eidesstattliche Versicherung") for the Bachelor's/ Master's thesis is the official and legally binding version.**