technische universität
dortmund

Transformer für IMU basierte Aktivitätserkennung

Bachelorarbeit

Raphael David Spiekermann
17. März 2022

Supervisors:
Prof. Dr.-Ing. Gernot A. Fink
Philipp Oberdiek, M.Sc.

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

INHALTSVERZEICHNIS

1 EINLEITUNG

Die Aktivitätserkennung (HAR) beschäftigt sich mit Messdaten zu menschlichen Bewegungsabläufen und versucht diese in Aktivitätsklassen einzuordnen. Bei den Messdaten kann es sich um Videoaufnahmen, Puls- oder Blutdruckmessungen, Inertialmessungen und vieles mehr handeln.

Die Methoden der HAR werden bereits heute in vielen verschiedenen Bereichen des öffentlichen und privaten Lebens genutzt. Eine Beispielanwendung ist die Patientenüberwachung in Krankenhäusern, mit der Notfälle frühzeitig erkannt werden und den Patienten somit schnellstmöglich geholfen werden kann.

Seit einigen Jahren profitiert auch die Aktivitätserkennung, die lange von traditionellen Klassifikationsmethoden geprägt war, von der fortschreitenden Entwicklung tiefer neuronaler Netze. Tiefe neuronale Netze ermöglichen Ende-zu-Ende-Architekturen für die Lösung von HAR, die ohne die Notwendigkeit menschlicher Intervention in der Lage sind z.B. Inertialmessungen direkt auf Aktivitätsklassen abzubilden.

Da es sich bei den Messdaten zumeist um Zeitreihen handelt, liegt die Vermutung nahe, dass neuronale Modelle, die auf die Erkennung zeitlicher Abhängigkeiten spezialisiert sind, gegenüber allgemeineren Netzarchitekturen im Vorteil sind.

Ein solches Modell ist der sogenannte Transformer aus Vaswani et al. [2017], der ursprünglich für die Verwendung im Natural Language Processing (NLP) entwickelt wurde. Transformer zeichnen sich dadurch aus, dass sie bei der Lösung von Übersetzungsaufgaben während der Übersetzung eines Wortes nicht nur in einem begrenzten lokalen Umfeld nach zugehörigen Wörtern suchen, sondern für jedes Wort stets die gesamte Eingabesequenz (z.B. ein Satz, ein Abschnitt oder auch ein gesamter Text) hinsichtlich syntaktischer und semantischer Zusammenhänge analysieren. Dieser Ansatz bewies sich für das NLP als äußerst leistungsstark, weswegen sich Transformer schnell zu der Standardarchitektur in diesem Forschungsgebiet entwickelten.

Wie Shavit and Klein [2021] zeigen konnten, lassen sich Transformer auch für die Lösung von HAR in Bezug auf grobgranulare menschliche Aktivitäten wie „Gehen“, „Stehen“, „Sitzen“ etc. erfolgreich einsetzen. Gegenstand dieser Arbeit wird es sein, zu prüfen, ob sich dies auch auf die komplexen Aktivitäten des LARa-Datensatzes aus Niemann et al. [2020] übertragen lässt, dessen Aktivitätsklassen (z.B. „Wagen schieben“) aus mehreren Bewegungsmustern verschiedener Körperteile zusammengesetzt sind.

Um dies zu analysieren, werden neben dem in Shavit and Klein [2021] vorgestellten HAR-Modell, das von einem Transformer Gebrauch macht, weitere Konkurrenzmodelle implementiert, anhand derer die Leistung des Transformer-Modells eingeordnet werden kann. Des Weiteren werden verschiedene Konfigurationen des Transformer-Modells mit einander verglichen um einen Einblick in die Funktionsweise des Modells in Bezug auf die Lösung von HAR zu erhalten.

Ziel dieser Arbeit ist es, die Leistung von HAR-Modellen, die von Transformern Gebrauch machen, auf den IMU-Daten des LARa-Datensatzes zu bewerten. Damit die verschiedenen Modelle bzw. Modellkonfigurationen fair miteinander verglichen werden können, wird für keines der Modelle Finetuning betrieben.

Wie dies für die Lösung von HAR mit sequenziellen Aufnahmen typisch ist, wird auch hier ein Sliding-Window Ansatz verfolgt, der die Inertialmessungen in gelabelte Teilsequenzen gruppiert, die für die Klassifikation verwendet werden. Um den Einfluss der Länge der Eingabesequenzen auf die Klassifikationsleistung der Modelle messen zu können, werden verschiedene Fenstergrößen eingesetzt.

Da diese Arbeit auf den Ergebnissen von Shavit and Klein [2021] aufbaut, besteht eine erste Aufgabe darin, einen Teil der dort präsentierten Resultate zu reproduzieren. Dazu wird das vorgestellte HAR-Modell auf den Daten des MotionSense-Datensatzes aus Malekzadeh et al. [2019] ausgewertet. Im Anschluss daran werden neben dem Modell aus Shavit and Klein [2021] weitere Modelle auf den IMU-Daten des LARa-Datensatzes sowohl für die Vorhersage der Aktivitätsklassen, als auch die Vorhersage der im Datensatz enthaltenen Attributkombinationen eingesetzt.

Diese Arbeit besteht neben diesem Kapitel aus fünf weiteren Kapiteln. In Kapitel 2 wird das theoretische Grundwissen für das Verständnis der Methodik und der Experimente vermittelt. An späteren Stellen wird sich regelmäßig auf die dort vorgestellten grundlegenden Definitionen bezogen. In Kapitel 3 werden ähnliche Arbeiten aus dem Forschungsbereich der Aktivitätserkennung vorgestellt. Zusätzlich findet eine Einordnung statt, die diese Arbeit in den Kontext der verwandten Arbeiten setzt. Die hier verfolgte Methodik der Experimente wird in Kapitel 4 detailliert beschrieben. Dort werden neben der Beschreibung der Experimentumgebung auch die verschiedenen hier eingesetzten HAR-Modelle vorgestellt. Auf der Methodik aufbauend, werden in Kapitel 5 die Experimente beschrieben und analysiert. Abschließend wird in Kapitel 6 ein Fazit gezogen, das die Ergebnisse dieser Arbeit zusammenfasst. Außerdem werden weitere Ansätze für die Forschung auf diesem Gebiet angeregt, die für zukünftige Forschungsarbeiten interessant sein könnten.

2 GRUNDLAGEN

2.1 AKTIVITÄTSERKENNUNG

Unter Aktivitätserkennung oder auch Human Activity Recognition (HAR) versteht man das Problem der automatischen Klassifizierung menschlicher Aktivitäten.

Diese für Menschen gewöhnlicherweise sehr intuitive Fähigkeit wird im digitalen Zeitalter auch von Computern erwartet, weswegen in den Teilbereichen der Informatik „Computer Vision“ und „Maschinelles Lernen“ viel in diese Richtung geforscht wird [Vrigkas et al., 2015].

HAR findet in vielen verschiedenen Bereichen Anwendung [Lara and Labrador, 2012]. Im privaten Bereich können HAR-Systeme in Form von Smart-Home-Geräten oder Spielkonsolen (z. B. Microsoft Kinect) Einzug in die Haushalte halten. Im Gesundheitswesen kann die Aktivitätserkennung zur Überwachung von Demenzkranken oder Diabetikern eingesetzt werden, um diese auf ungewöhnliches Verhalten zu überwachen. Die Sicherheitsbranche kann HAR nutzen um Videoaufzeichnungen auf Straftaten zu durchsuchen oder Einbruchsversuche frühzeitig zu erkennen. In der verarbeitenden Industrie und in der Logistik kann die Überwachung von Mitarbeitern durch HAR-Systeme ebenfalls einen wesentlichen Beitrag zur Betriebsoptimierung leisten.

HAR basiert meist auf Videos oder Sensordaten in der Form von Zeitreihen, also sequenzielle Messdaten, die aus nacheinander aufgezeichneten Momentaufnahmen bestehen. Zeitreihen sind für die Lösung von HAR sinnvoll, da Aktivitäten in der Regel über längere Zeiträume stattfinden und daher alleinstehende Momentaufnahmen möglicherweise nicht genügend Informationen über die jeweilige Aktivität enthalten.

Die Mess- bzw. Aufnahmegeräte lassen sich dabei nach Vrigkas et al. [2015] in zwei Gruppen einordnen.

Externe Messgeräte wie Überwachungskameras und Smart-Home-Geräte zeichnen sich dadurch aus, dass sie an festen Orten platziert sind und Menschen aktiv mit ihnen interagieren müssen. Oft werden mehrere solcher Geräte zusammen verwendet, sodass kombinierte HAR-Systeme auch komplexe Aktivitäten erfassen können. Moderne Smart-Home Systeme können bereits erkennen ob ihre Nutzer „Essen“, „Wäsche machen“ oder „Schlafen“.

Tragbare Geräte hingegen befinden sich direkt an den Körpern der Menschen. Zu ihnen gehören Smartphones, Smartwatches, Trägheitsmessgeräte (IMUs) oder auch GPS-Empfänger. Die Qualität ihrer Messungen hängt stark von ihrer Position am Körper sowie von der jeweiligen Aufgabe ab. Sollen Bewegungen wie „Gehen“, „Sitzen“, „Treppensteigen“ klassifiziert werden, eignen sich etwa in Hosentaschen befindliche IMUs, während GPS-Daten an dieser Stelle weniger aussagekräftig sind.

Ein gängiger Ansatz zur Lösung von HAR ist der sogenannte Sliding-Window Ansatz[Reining et al., 2019]. Dabei wird ein virtuelles Fenster mit einer vordefinierten Fenstergröße schrittweise über die Zeitreihe bewegt. Die Fenstergröße gibt dabei an, wie viele aufeinanderfolgende Momentaufnahmen in einem Fenster zusammengefasst werden sollen. Wenn beispielsweise für eine Zeitreihe, die mit einer Abtastrate von 50Hz aufgezeichnet wurde, eine Fenstergröße von 100 gewählt wird, umfasst jedes Fenster Aufzeichnungen über einen Zeitraum von $\frac{1}{50} \cdot 100 = 2$ Sekunden. Neben der Fenstergröße ist die Schrittweite ein weiterer Parameter für den Sliding-Window Ansatz. Sie gibt an, um wie viele Einheiten das Fenster in einem Schritt verschoben werden soll. Nach jedem Schritt werden die aktuell im Fenster enthaltenen Sensormessungen zusammengefasst und dann extrahiert. Zusätzlich muss eine Aktivitätsklasse ausgewählt werden, die die im Fenster stattfindende Aktivität bestmöglich wiedergibt. Bei Aktivitätsübergängen innerhalb eines Fensters ist die Wahl einer Aktivitätsklasse mit einem Informationsverlust verbunden.

Die maschinelle Klassifizierung solcher Fenster erfolgt in der HAR traditionell in zwei Schritten [Grzeszick et al., 2017]. Zunächst werden dem Fenster statistische Eigenschaften (arithmetisches Mittel, Median, Min, Max, usw.), sogenannte *Features*, entnommen und zu einem Feature-Vektor zusammengefasst. Der resultierende Vektor wird daraufhin an einen Klassifikator (Entscheidungsbaum, Support Vector Machine, ...) übergeben, der anhand der Features eine Aktivitätsklasse bestimmt, in die er das Fenster einordnet. Seit einigen Jahren werden auch tiefe neuronale Netze zur Lösung von HAR eingesetzt, die den Schritt der Extraktion statistischer Merkmale überflüssig machen und somit eine direkte Ende-zu-Ende-Architektur ermöglichen.

Neben der Schwierigkeit, mit Aktivitätsübergängen umzugehen, gibt es noch viele weitere Probleme im Bereich der Aktivitätserkennung, an denen derzeit geforscht wird. So gehen beispielsweise Aktivitäten nicht nur nahtlos ineinander über, sondern es können auch mehrere Aktivitäten parallel ausgeführt werden (Multitasking), was eine Klassifizierung weiter verkompliziert [Kim et al., 2009]. Außerdem werden dieselben Aktivitäten von Person zu Person unterschiedlich ausgeführt. Das führt zu einer hohen Variabilität innerhalb einer Klasse (Intraklassenvariabilität). Darüber hinaus weisen verwandte Tätigkeiten wie „Gehen“ und „Rennen“ oft starke Ähnlichkeiten auf, was zu einer geringen Interklassenvariabilität führt. Um diesen Schwierigkeiten entge-

genzuwirken, werden große, sorgfältig annotierte Datensätze mit vielen Probanden benötigt. Solche Datensätze sind jedoch aufgrund des großen Aufwands, der mit der Datengenerierung und -annotation verbunden ist, sehr selten [Rueda and Fink, 2021].

2.1.1 *IMU basierte Aktivitätserkennung*

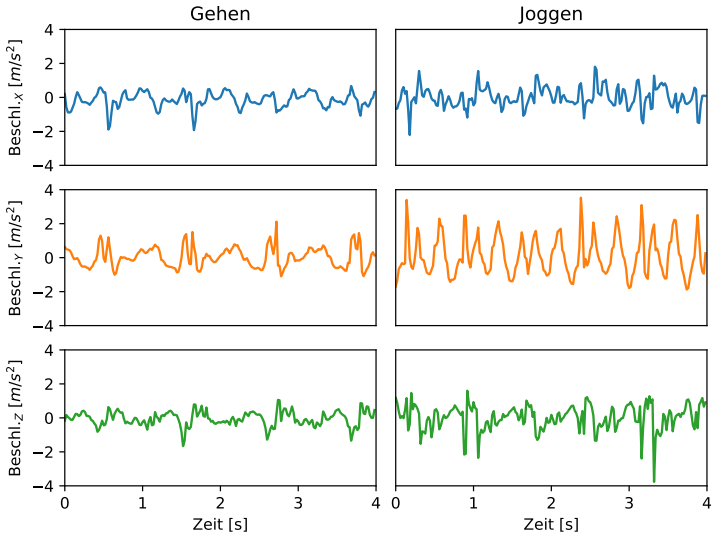

Abbildung 2.1.1: Gegenüberstellung der Aktivitätsklassen „Gehen“ und „Joggen“ mit Beschleunigungsdaten aus dem MotionSense Datensatz [Malekzadeh et al., 2019].

Inertialmessgeräte (IMUs) sind tragbare Messinstrumente, die aus mehreren Trägheitssensoren bestehen [Vec, 2022]. Eine typische Kombination für eine IMU besteht aus einem Beschleunigungsmessgerät, einem Gyroskop und einem Magnetometer. So ergibt jede Messung einen neundimensionalen Vektor, der die auf die IMU wirkende Beschleunigung in allen drei Raumdimensionen, die Winkelgeschwindigkeit pro Drehachse und die Orientierung im Magnetfeld (ebenfalls in drei Dimensionen) enthält.

Abbildung 2.1.1 zeigt die Beschleunigungsmessungen von zwei Aktivitäten („Gehen“ und „Joggen“) über einen Zeitraum von vier Sekunden. Im direkten Vergleich der Einzelbeschleunigungen (pro Raumdimension) zeigt sich insbesondere für die Beschleunigung in Richtung der Y-Achse, dass „Joggen“ die intensivere Aktivität ist. Dies lässt sich daran erkennen, dass die Intervalle zwischen den Ausschlägen deutlich

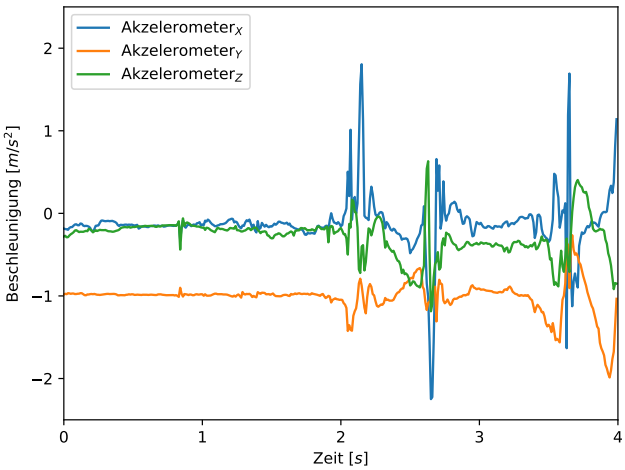

Abbildung 2.1.2: Übergang von „Stehen“ zu „Gehen“ mit Beschleunigungsdaten aus dem LARa-Datensatz [Niemann et al., 2020].

kleiner sind und die Ausschläge zudem sowohl im Positiven als auch im Negativen stärker ausfallen. Ähnliche Tendenzen sind auch bei den anderen Achsen zu erkennen, wenn auch nicht in dieser Eindeutigkeit.

Abbildung 2.1.2 zeigt einen Übergang zwischen Aktivitätsklassen, der nach etwa zwei Sekunden stattfindet. Solche Aktivitätsübergänge führen, wie bereits oben erwähnt, zu Ungenauigkeiten bei der Lösung von HAR. Nimmt man z.B. ein nach einer Sekunde beginnendes Fenster der Länge 3s [1], so enthält dieses beide Aktivitäten. Wenn eine der beiden möglichen Aktivitätsklassen diesem Fenster zugeordnet wird, geht zwangsmäßig die Information über die andere enthaltene Klasse verloren. Fenster dieser Art können das Training von Klassifikatoren erschweren, insbesondere im ungünstigsten Fall, in welchem das Fenster beide Aktivitäten zu gleichgroßen Anteilen enthält. Eine Möglichkeit, diesem Problem entgegenzuwirken, besteht darin, die Fenstergröße so klein wie möglich zu halten. Im Detail bedeutet dies, dass die kleinstmögliche Fenstergröße gewählt werden sollte, die die Erkennung von charakteristischen Mustern der Aktivitäten ermöglicht[Lara and Labrador, 2012].

1 Die Anzahl der Fensterelemente lässt sich daraus mit Kenntnis der Abtastrate der Messgeräte leicht errechnen.

2.2 MASCHINELLES LERNEN

Maschinelles Lernen ist ein Forschungszweig der künstlichen Intelligenz, der sich mit Algorithmen befasst, die den menschlichen Lernprozess nachahmen, indem sie von einer schlechten anfänglichen Problemlösung ausgehend, in einem iterativen Lernprozess, stetig bessere Lösungen erlernen[IBM, 2021].

Angenommen, es gibt ein Problem P, das vollständig durch eine unbekannte Funktion f^* gelöst wird. Im Kontext von Klassifikationsproblemen könnte dies beispielsweise bedeuten, dass f^* für jede denkbare Eingabe x immer die richtige Klassenzuordnung bestimmen kann. Algorithmen des maschinellen Lernens versuchen dann, anhand von Trainingsdaten, die die Form (Eingabe, gewünschte Ausgabe) besitzen, ein Modell f zu lernen, das die optimale Lösung f^* möglichst gut approximiert. Die Trainingsdaten sind eine Teilmenge des Problembereichs, mit bekannten Ein- und Ausgaben, an denen sich das Modell während des Lernprozesses orientieren kann.

Da f^* jedoch nicht nur für die Trainingsdaten, sondern für den gesamten Problembereich definiert ist, reicht es nicht aus, wenn die Algorithmen bloß die Trainingsdaten speichern oder auswendig lernen. Stattdessen müssen allgemeine Muster erlernt werden, die auch auf unbekannte Eingaben angewendet werden können.

Maschinelles Lernen wird heute zur Lösung zahlreicher Probleme eingesetzt. Von besonderer Bedeutung für diese Arbeit ist die Kategorie der Klassifikationsprobleme, bei denen das Ziel darin besteht, ein Modell zu entwickeln, das ihre Eingaben in vorher definierte Klassen einordnet. Ein Beispiel für ein Klassifikationsproblem ist die im vorherigen Abschnitt beschriebene Aktivitätserkennung anhand von IMU-Daten.

2.3 KÜNSTLICHE NEURONALE NETZE

Künstliche neuronale Netze (KNNs) sind mathematische Modelle, die in der Lage sind, beliebige Funktionen zu approximieren [Goodfellow et al., 2016, S. 164 ff.]. Ihre Struktur ist lose an die des menschlichen Gehirns angelehnt und besteht aus schichtweise organisierten künstlichen Neuronen, die wie biologische Neuronen Signale (bzw. Impulse) empfangen, diese verarbeiten und bei ausreichender Erregung an benachbarte Neuronen weiterleiten [Traeger et al., 2003].

2.3.1 *Perzeptron*

Das Perzeptron wurde in den späten 1950er Jahren von Frank Rosenblatt erfunden und in Rosenblatt [1958] beschrieben. Es handelt sich dabei um ein einfaches neu-

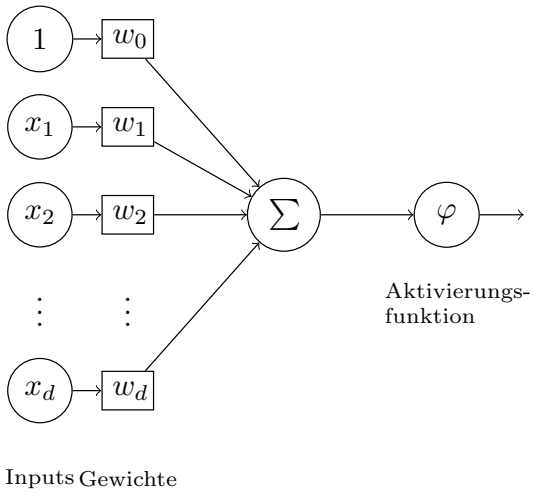

Abbildung 2.3.1: Darstellung eines einfachen Neurons mit Eingaben x_1 bis x_d und entsprechenden Gewichten w_1 bis w_d, sowie dem Bias w_0, der als konstante virtuelle Eingabe eine 1 erhält. Die Eingaben und Gewichte (auch der Bias) werden zuerst elementweise multipliziert, die Produkte dann aufsummiert und schließlich an die Aktivierungsfunktion φ übergeben, die die Ausgabe des Neurons berechnet.

ronales Netz, das nur aus einem künstlichen Neuron besteht. Im Folgenden wird eine moderne Version des Perzeptrons vorgestellt, die Rosenblatts Perzeptron erweitert und den Grundbaustein moderner künstlicher neuronaler Netze darstellt. Ein solches künstliches Neuron ist in Abbildung 2.3.1 dargestellt. Es sei jedoch darauf hingewiesen, dass es in der Literatur verschiedene, von der Funktionsweise her jedoch äquivalente, Varianten gibt, die sich in ihrer Darstellung (z.B. in der Behandlung des Bias) unterscheiden. Die Vorteile der hier beschriebenen Variante werden in späteren Abschnitten zur Optimierung von neuronalen Netzen deutlich werden.

Seien im Folgenden $w = [w_1, \dots, w_d]^T \in \mathbb{R}^d$ der Gewichtungsvektor, $b \in \mathbb{R}$ der Bias und $\varphi : \mathbb{R} \to \mathbb{R}$ die Aktivierungsfunktion eines Perzeptrons. Sei außerdem $x = [x_1, \dots, x_d]^T \in \mathbb{R}^d$ eine beliebige d-dimensionale Eingabe.

Die Aktivierung des Neurons für diese Eingabe

$$a = \sum_{i=1}^{d} w_i x_i + b = w^T x + b \tag{1}$$

errechnet sich als gewichtete Summe der Eingabeelemente, addiert mit dem skalaren Wert b. Ein Gewicht w_i bestimmt, inwieweit die entsprechende Eingabe x_i bei der

Aktivierung des Neurons erregend oder hemmend wirken soll. Der Bias b gibt intuitiv an, wie schwierig es im Allgemeinen sein sollte, ein Neuron zu aktivieren.

Den *Output* (auch Ausgabeaktivierung bzw. Ausgabe) des Neurons

$$o = \varphi(a) = \varphi\left(\sum_{i=1}^{d} w_i x_i + b\right) \tag{2}$$

erhält man durch die Anwendung der Aktivierungsfunktion φ auf die zuvor berechnete Aktivierung.

Wie bereits in Gleichung 1 zu sehen ist, kann die Notation verkürzt werden, indem die Summe durch ein Skalarprodukt ersetzt wird. Außerdem ist es für den weiteren Verlauf hilfreich, den Gewichtsvektor um $w_0 = b$ zu erweitern, und analog dazu die Eingabe x virtuell um eine 1 an nullter Stelle zu ergänzen. Dies führt zu keiner Änderung hinsichtlich der Funktionsweise des Modells, erlaubt aber die komprimierte Notation in Gl. 3.

$$o = \varphi\left(\sum_{i=0}^{d} w_i x_i\right) = \varphi\left(w^T x\right) \tag{3}$$

Das in Abbildung 2.3.1 dargestellte Perzeptron folgt ebenfalls dieser Konvention.

2.3.2 *Mehrlagiges Perzeptron*

Tiefe neuronale Netze bestehen aus mehreren Schichten $l_k, k = 0, \ldots, L$, die jeweils $M^{(k)}$ künstliche Neuronen enthalten. Es existieren drei Typen von Schichten: Die Eingabeschicht (*Input Layer*) l_0 stellt den Eingabevektor $x \in \mathbb{R}^d$ bereit, die Ausgabeschicht (*Output Layer*) l_L enthält die finalen Ergebnisse der Berechnung $\hat{y} \in \mathbb{R}^{M^{(L)}}$, und als verborgene bzw. versteckte Schichten (*Hidden Layers*) $l_k, k = 1, \ldots, L-1$ werden sämtliche weiteren Schichten bezeichnet.

Neuronen einer Schicht l_k können (ausschließlich) mit Neuronen einer anderen Schicht $l_m, m \neq k$ verbunden sein. Dann dienen die Neuronenausgaben der k-ten Schicht $o_i^{(k)}, i = 1, \ldots, M^{(k)}$ als Eingaben für die m-te Schicht.

In einem mehrlagigen Perzeptron bzw. *Multilayer Perceptron* (MLP) sind die gewichteten Verbindungen dabei so organisiert, dass jedes Neuron der Schicht $l_k, k = 0, \ldots, L-1$ mit allen Neuronen der $(k+1)$-ten Schicht durch eine Vorwärtskante verbunden ist.

Die Topologie von MLPs lässt sich durch einen gerichteten, gewichteten, kreisfreien Graphen $G = (V, E, \omega)$ mit Knoten $v \in V = V_0 \uplus V_1 \uplus \cdots \uplus V_L$, Kanten $e \in E = \{(u, v) \mid u \in V_k, v \in V_{k+1}, k = 0, \ldots, L-1\}$ und der Gewichtsfunktion $\omega : E \to \mathbb{R}$ beschreiben.

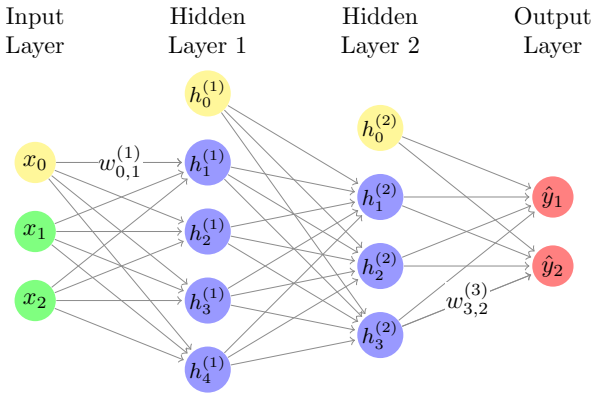

Abbildung 2.3.2: Illustration eines vierschichtigen Perzeptrons, das neben der Eingabeschicht (grün) und der Ausgabeschicht (rot) über zwei versteckte Schichten verfügt. Die Dimensionen der versteckten Schichten betragen 4 und 3. Die hier gelb markierten Knoten sind die virtuellen Neuronen, die stets 1 ausgeben und dessen ausgehende Gewichte den Bias der Folgeschicht entsprechen. Zusätzlich sind zwei Kanten beschriftet, um die Notation der Kantengewichte visuell darzustellen.

Der Semantik des Graphens lässt sich entnehmen, dass der Informationsfluss stets von der Eingabeschicht aus kommend in Richtung der Ausgabeschicht verläuft.

Da Rückkopplungen, Schleifen oder residuale (schicht-überspringende, vgl. He et al. [2016]) Verbindungen nicht existieren, bieten sich MLPs hauptsächlich für nicht-sequenzielle Daten an.

Abbildung 2.3.2 zeigt ein solches Netz, welches neben den Ein- und Ausgabeschichten auch zwei verstecke Schichten besitzt. Wie auch in bei dem einfachen Perzeptron (siehe Abschnitt 2.3.1) werden die Gewichtsvektoren pro Neuron um den Bias w_0 ergänzt, was dazu führt, dass jede Schicht $l_k, k = 0, \ldots, L-1$ ein virtuelles Neuron an nullter Stelle besitzt, das konstant den Wert 1 ausgibt. In Abbildung 2.3.2 sind diese gelb eingezeichnet.

Die Aktivierung des j-ten Neurons der k-ten Schicht lässt sich, analog zu der Berechnung der Aktivierung eines einfachen Neurons, als gewichtete Summe der Schichteingabe und somit auch als Skalarprodukt schreiben

$$a_j^{(k)} = \sum_{i=0}^{M^{(k-1)}} w_{i,j}^{(k)} o_i^{(k-1)} = w_j^{(k)T} o^{(k-1)}, \tag{4}$$

wobei $w_{i,j}^{(k)}$ das Gewicht bezeichnet, das ausgehend vom i-ten Knoten der $(k-1)$-ten Schicht in den j-ten Knoten der k-ten Schicht eingeht.

Die Ausgabe des selben Neurons

$$o_j^{(k)} = \varphi^{(k)}\left(a_j^{(k)}\right) = \varphi^{(k)}\left(\sum_{i=0}^{M^{(k-1)}} w_{i,j}^{(k)} o_i^{(k-1)}\right)$$

erhält man aus der Anwendung der Aktivierungsfunktion $\varphi^{(k)}$ (der aktuell betrachteten Schicht) auf die Aktivierung $a_j^{(k)}$.

Die Gewichte zwischen der $(k-1)$-ten und k-ten Schicht ($k = 1, \ldots, L$) lassen sich in einer Matrix $W^{(k)} \in \mathbb{R}^{(M^{(k-1)}+1)\times M^{(k)}}$ organisieren. Die Tatsache, dass die Anzahl der Zeilen der Gewichtsmatrix um 1 erhöht wird, ist auf die hier befolgte Konvention zurückzuführen, die Bias als virtuelle gewichtete Ausgaben der vorherigen Schicht zu betrachten. Die Aktivierungen der k-ten Schicht

$$a^{(k)} = W^{(k)\mathsf{T}} o^{(k-1)} \in \mathbb{R}^{M^{(k)}}$$

ergeben sich damit in vektorisierter Form als Produkt aus der transponierten Gewichtsmatrix $W^{(k)}$ und dem Ausgabevektor der vorherigen Schicht $o^{(k-1)}$.

Wendet man anschließend die Aktivierungsfunktion der k-ten Schicht $\varphi^{(k)}$ elementweise auf die Aktivierung $a^{(k)}$ an und ergänzt die virtuelle Ausgabe 1 am Index 0, so erhält man den Ausgabevektor der k-ten Schicht

$$o^{(k)} = \begin{bmatrix} 1 \\ \varphi^{(k)}\left(a_1^{(k)}\right) \\ \vdots \\ \varphi^{(k)}\left(a_{M^{(k)}}^{(k)}\right) \end{bmatrix} \in \mathbb{R}^{M^{(k)}+1}.$$

Da die Eingabeschicht lediglich die Eingabedaten bereitstellt, ist sie die einzige Schicht eines MLPs, die keine eigene Berechnung vornimmt. Dies ist in Gl. 5 festgehalten.

$$o^{(0)} = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_d \end{bmatrix} \in \mathbb{R}^{M^{(0)}+1}, \text{ mit } M^{(0)} = d \tag{5}$$

Das bedeutet auch, dass die Anzahl der künstlichen Neuronen in der Eingabeschicht der Eingabedimension entspricht. Für sehr große Eingaben kann diese Abhängigkeit zu Problemen führen, da die Anzahl der Gewichte zwischen der Eingabeschicht und der darauf folgenden Schicht dem Produkt aus $(M^{(0)} + 1)$ und $M^{(1)}$ entspricht, und eine zu große Anzahl an Netzparametern den Lernprozess stören oder zumindest verlangsamen kann.

Wie die Gleichungen 6 - 8 zeigen, lässt sich die vom MLP approximierte Funktion $f(x; W)$ als Komposition der durch die Schichten umgesetzten Teil-Funktionen $f^{(k)}\left(o^{(k-1)}; W^{(k)}\right)$ beschreiben.

$$\hat{y} = f(x; W) \tag{6}$$
$$= (f^{(L)} \circ f^{(L-1)} \circ \cdots \circ f^{(1)})(x; W) \tag{7}$$
$$= \varphi^{(L)}(W^{(L)T} \varphi^{(L-1)}(W^{(L-1)T} \dots \varphi^{(1)}(W^{(1)T} o^{(0)}) \dots)) \tag{8}$$

2.3.3 *Aktivierungsfunktionen*

Aktivierungsfunktionen $\varphi : \mathbb{R} \to \mathbb{R}$ sind zumeist nichtlineare, stetige, differenzierbare Funktionen, die für die Umwandlung der Aktivierung in die Ausgabe in künstlichen Neuronen zuständig sind. Sie sind der Grund dafür, dass neuronale Netze in der Lage sind, nicht nur lineare Transformationen, sondern auch nichtlineare Funktionen zu approximieren. Die Stetig- und Differenzierbarkeit ist relevant für die gradientenbasierte Optimierung neuronaler Netze (vgl. Abschnitt 2.3.4).

Die Wahl der Aktivierungsfunktion in der Ausgabeschicht hängt direkt von der Anwendung des Netzes ab. Im Falle einer binären Klassifikation bietet sich etwa die Sigmoidfunktion

$$\sigma(x) = \frac{e^x}{1 + e^x}$$

an, dessen Wertebereich zwischen 0 und 1 liegt. Für innere Schichten gibt es hingegen keine direkte problemabhängige Vorgabe der Aktivierungsfunktion. Stattdessen ist die Auswahl vielmehr eine von vielen Designentscheidungen, die während des Modellentwurfs getroffen werden müssen.

In den letzten Jahren hat sich insbesondere die *Rectified Linear Unit*

$$\mathrm{ReLU}(x) = \max\{x, 0\}$$

im Bereich des Deep Learnings durchgesetzt, die trotz der einfachen Definition und nicht-stetigen Ableitung einige für das Lernverfahren angenehme Eigenschaften besitzt

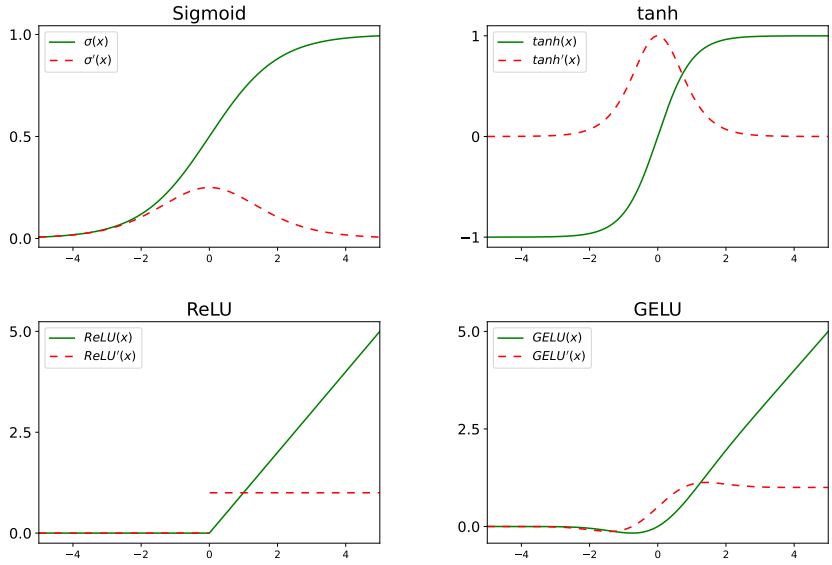

Abbildung 2.3.3: Auswahl einiger Aktivierungsfunktionen und ihrer Ableitungen.

[Ramachandran et al., 2017]. Unter dem Schlagwort *Deep Learning*, das - wie von Zhang et al. [2018] beschreiben - viele verschiedene Definitionen besitzt, werden im Folgenden mehrschichtige neuronale Netze (vgl. Abschnitt 2.3.2) oder darauf aufbauende Architekturen verstanden.

Eine Auswahl einiger Aktivierungsfunktionen und ihrer Ableitungen ist in Abbildung 2.3.3 dargestellt.

2.3.4 *Optimierung*

Ein neuronales Netz zu optimieren bedeutet, die Parameter W (also die Gewichte und Bias) des Netzes so anzupassen, dass für annotierte Eingaben (x_i, y_i) einer Stichprobe $X = \{(x_1, y_1), \ldots, (x_N, y_N)\}$ die Resultate der Berechnungen $\hat{y}_i = f(x_i; W)$ möglichst präzise die korrekten Werte y_i vorhersagen.

Das Training eines neuronalen Netzes erfordert eine Zielfunktion, deren Funktionswert optimiert werden kann. Anstatt eine Funktion zu maximieren, die direkt die Güte eines Netzes misst, wird eine gleichwertige Methode gewählt, bei der eine Funktion minimiert wird, die den Fehler des neuronalen Netzes auf den Eingabedaten quantifiziert.

Eine viel verwendete Fehlerfunktion für Regressionsprobleme (bei denen $y_i, \hat{y}_i \in \mathbb{R}$ ausnahmsweise keine Vektoren sind) ist der durchschnittliche quadratische Fehler

$$\mathrm{MSE} = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2.$$

Bei Klassifikationsproblemen wird die Kreuzentropie (*Cross-Entropy*) CE genutzt, die ebenfalls über die Eingaben gemittelt wird. Dazu seien $c \in \{1, \ldots, C\}$ die Indizes der vorherzusagenden Klassen. Sei außerdem $y \in \mathbb{R}^C$ eine vektorielle Kodierung der Klasse c, deren Einträge an den Positionen $p \in \{1, \ldots, C\}$ sich mit Gl. 9 berechnen lassen[2].

$$y_p = \begin{cases} 1 & , \text{ falls } p = c \\ 0 & , \text{ sonst} \end{cases} \tag{9}$$

Auf der Grundlage dieser sogenannten *One-Hot*-Kodierung, lässt sich die Kreuzentropie wie in Gl. 10 definieren.

$$\mathrm{CE}(\hat{y}, y) = -\sum_{c=1}^{C} y_c \log(\hat{y}_c) \tag{10}$$

Die durchschnittliche Kreuzentropie ergibt sich dann als das gewichtete arithmetische Mittel der Kreuzentropien auf den einzelnen Klassen:

$$\mathrm{CE}_{avg} = \frac{1}{N} \sum_{i=1}^{N} \mathrm{CE}(\hat{y}_i, y_i). \tag{11}$$

Gradientenabstieg

Eine Möglichkeit zur lokalen Minimierung ist das Gradientenabstiegsverfahren. Dabei handelt es sich um einen iterativen Algorithmus, bei dem ausgehend von einem zufällig gewählten Startpunkt kleine Schritte in Richtung der negativen Steigung (also dem Gradienten) gemacht werden, bis ein Abbruchkriterium erreicht ist. Dies funktioniert, weil der Gradient ∇g einer Funktion g immer in die Richtung der steilsten Steigung zeigt. Dementsprechend zeigt der negative Gradient in die Richtung des

2 Angenommen es existieren $C = 5$ Klassen, dann entspräche die kodierte Darstellung eines Elements der Klasse $c = 2$ dem Vektor $y = \begin{bmatrix} 0, 1, 0, 0, 0 \end{bmatrix}^T$.

steilsten Abstiegs, also in die Richtung eines lokalen Minimums. Für die Berechnung der Steigungen muss g differenzierbar sein. [Goodfellow et al., 2016, S. 80 ff.]

Das Verfahren wendet pro Iterationsschritt die Aktualisierungsregel 12 auf die Parameter θ an.

$$\theta^{t+1} \leftarrow \theta^t - \alpha \cdot \nabla g(\theta^t). \tag{12}$$

Nach hinreichend vielen Iterationen des Aktualisierungsschritts konvergiert das Verfahren in einem lokalen Minimum von g. Die Lernrate α ist ein sogenannter Hyperparameter und wird zur Steuerung der Schrittweite benötigt. Hyperparameter werden dem Lernprozess von außen zugeführt, um ihn zu beeinflussen. Eine adäquate Wahl von α ist notwendig, da eine zu große Schrittweite zu einem Überschießen des angestrebten Minimums oder zu aufeinanderfolgenden Iterationen führen kann, die um das Minimum oszillieren. Eine zu kleine Schrittweite hingegen führt zu einer langsameren Konvergenz und damit zu einem höheren Rechenaufwand. Ein Kompromiss zwischen Effizienz und Genauigkeit besteht darin, mit einer etwas größeren Lernrate zu beginnen und diese sukzessive (z.B. alle 5 Iterationen) um einen Faktor $0 < \beta < 1$ zu verringern.

GRADIENTENABSTIEG IN NEURONALEN NETZEN Neuronale Netze f werden auf Trainingsdaten $X = \{(x_1, y_1), \ldots, (x_N, y_N)\}$ trainiert, indem mithilfe des Gradientenverfahrens eine Fehlerfunktion $\Gamma(f, X)$ minimiert wird.

Die Aktualisierungsregel 13, die zur Aktualisierung der Netzparameter W verwendet wird, ergibt sich aus der allgemeinen Aktualisierungsregel 12.

$$W^{t+1} \leftarrow W^t - \alpha \cdot \nabla_W \Gamma(f, X), \text{ mit } \nabla_W \Gamma = \left(\frac{\partial \Gamma}{\partial w_1}, \frac{\partial \Gamma}{\partial w_2}, \ldots\right)^T \tag{13}$$

Typischerweise handelt es sich bei Γ um einen Durchschnittswert der Einzelfehler $E(\hat{y}_i, y_i)$, die sich jeweils getrennt von einander berechnen lassen[3].

$$\Gamma(f, X) = \frac{1}{N} \sum_{i=1}^{N} E(\hat{y}_i, y_i) \propto \sum_{i=1}^{N} E(\hat{y}_i, y_i) \tag{14}$$

Da Γ, wie in Gl. 14 zu sehen ist, proportional zu der Summe der Einzelfehler ist, folgt aus der Summenregel der Differenzialrechnung, dass der Gradient der Gesamtfehlerfunktion

$$\nabla_W \Gamma(f, X) \propto \sum_{i=1}^{N} \nabla_W E(\hat{y}_i, y_i) \text{ bzw. } \frac{\partial \Gamma(f, X)}{\partial w_j} \propto \sum_{i=1}^{N} \frac{\partial E(\hat{y}_i, y_i)}{\partial w_j}$$

3 Bspw. $\Gamma = CE_{avg} = \frac{1}{N} \sum_{i=1}^{N} CE(\hat{y}_i, y_i)$

proportional zu der Summe der Gradienten der Einzelfehler ist.

Da die Parameter des neuronalen Netzes die Gewichte und Bias sind, bestehen die Gradienten $\nabla_W E(\hat{y}, y)$ aus den partiellen Ableitungen $\frac{\partial E(\hat{y},y)}{\partial w_{i,j}^{(k)}}$ [4] , die sich mit dem Backpropagation-Algorithmus berechnen lassen, der weiter unter vorgestellt wird.

STOCHASTISCHER GRADIENTENABSTIEG Das im vorherigen Abschnitt beschriebene Optimierungsverfahren verwendet pro Iteration den gesamten Trainingsdatensatz X, um die Gradienten der Einzelfehler zu berechnen. Dies ist insbesondere bei großen Datensätzen sehr rechenintensiv und führt zu einem langsamen Lernprozess. Eine Alternative ist der sogenannte stochastische Gradientenabstieg (*Stochastic Gradient Descend*, SGD)[Robbins and Monro, 1951; Kiefer and Wolfowitz, 1952]. Beim SGD werden die Netzparameter in jeder Iteration anhand eines zufällig gewählten Trainingsbeispiels (x_i, y_i) aktualisiert:

$$W^{t+1} \leftarrow W^t - \alpha \cdot \nabla_W E(\hat{y}_i, y_i).$$

Nach einer Vielzahl von Iterationen konvergieren die Parameter W^*_{sgd} gegen die endgültigen Parameterwerte des nicht-stochastischen Gradientenabstiegs W^*_{gd}. Im Gegensatz zum gewöhnlichen Gradientenabstieg ist SGD jedoch anfällig für Ausreißer und Rauschen innerhalb der Trainingsdaten und nimmt daher keinen direkten Weg zum nächsten lokalen Minimum.

Moderne Optimierer implementieren eine Variante des SGDs, bei der die Gradienten pro Iteration auf sogenannten *Minibatches* gebildet und dann gemittelt werden. Minibatches B sind kleine ($|B| << |X|$) zufällig zusammengestellte Teilmengen des Trainingsdatensatzes. Der Vorteil der Verwendung von Minibatches besteht darin, dass sie einen guten Kompromiss zwischen kurzer Berechnungszeit pro Optimierungsschritt und numerischer Stabilität bieten. Die Größe der Minibatches ist somit ein weiterer Hyperparameter, der dem Lernalgorithmus von außen vorgegeben wird.

Backpropagation

Der Backpropagation-Algorithmus [Rumelhart et al., 1986] ist das Standardverfahren zur Berechnung von Gradienten in neuronalen Netzen. Dabei wird der Fehler von der Ausgabeschicht aus in Richtung der Eingabeschicht in einem sog. *Backward Pass* zurückgeführt.

Als Eingaben erhält der Algorithmus:

4 Wie oben erwähnt, sind die Bias in den Gewichtsmatrizen enthalten.

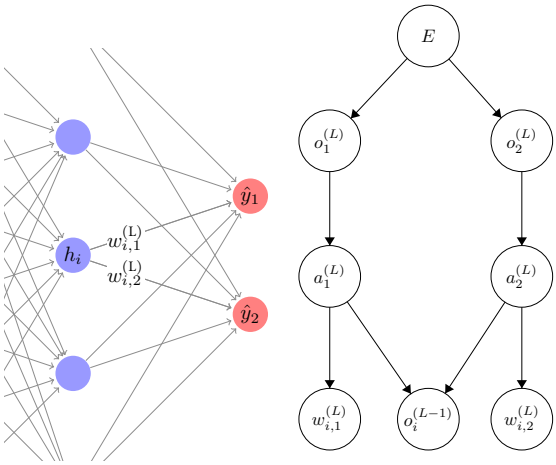

Abbildung 2.3.4: Ausschnitt der letzten zwei Schichten eines tiefen neuronalen Netzes (links), sowie der zu den beschrifteten Knoten zugehörige Teil des Abhängigkeitsgraphen (ausgehend von $E(\hat{y}, y)$). Die Ausgaben der letzten Schicht $o_j^{(L)}$ entsprechen den Vorhersagewerten $\hat{y}_j$, also den Ergebnissen des gesamten Netzes. Der Graph entsteht, indem die Berechnung des Netzes rückwärts durchlaufen wird und dabei Variablenabhängigkeiten durch Kanten markiert werden. So gilt bspw. $o_j^{(L)} = \varphi^{(L)}\left(a_j^{(L)}\right)$, weswegen $o_j^{(L)}$ direkt von $a_j^{(L)}$ abhängig ist. Hängt eine Variable von mehreren untereinander unabhängigen Variablen ab, so verzweigt der Graph unterhalb dieser.

1. Einen Datensatz $X = \{(x_1, y_1), \ldots, (x_N, y_N)\}$ mit annotierten Paaren (x_i, y_i). Da der Algorithmus immer nur auf einem Paar ausgeführt wird und der Index i im Weiteren für die Indizierung der Knoten genutzt wird, wird in diesem Abschnitt das Paar (x, y) stellvertretend für beliebige Paare (x_i, y_i) verwendet.
2. Ein MLP $f(x; W)$, dessen Parameter W optimiert werden sollen.
3. Eine Fehlerfunktion $E(\hat{y}, y)$ (abkürzend im Folgenden häufig nur E genannt), die den Fehler für ein Input-Output Paar $(x, y) \in X$ angibt.

Wie auch in vorherigen Abschnitten wird der Bias $b_j^{(k)}$ als Gewicht $w_{0,j}^{(k)}$ mit konstanter Eingabe 1 verstanden, was dazu führt, dass W nur die Gewichte $w_{i,j}^{(k)}$ enthält[5]. Es müssen daher nur die partiellen Ableitungen $\frac{\partial E}{\partial w_{i,j}^{(k)}}$ gebildet werden.

5 Auch hier stellt $w_{i,j}^{(k)}$ die Gewichtskante zwischen dem i-ten Knoten der $(k-1)$-ten Schicht und dem j-ten Knoten der k-ten Schicht dar.

Betrachtet man den in Abbildung 2.3.4 dargestellten Abhängigkeitsgraphen, so lassen sich die partiellen Ableitungen unter Verwendung der Kettenregel expandieren und vereinfachen

$$\frac{\partial E}{\partial w_{i,j}^{(k)}} = \underbrace{\frac{\partial E}{\partial o_j^{(k)}} \frac{\partial o_j^{(k)}}{\partial a_j^{(k)}}}_{\delta_j^{(k)}} \frac{\partial a_j^{(k)}}{\partial w_{i,j}^{(k)}} = \delta_j^{(k)} \frac{\partial a_j^{(k)}}{\partial w_{i,j}^{(k)}} = \delta_j^{(k)} o_i^{(k-1)}, \tag{15}$$

wobei $\delta_j^{(k)} = \frac{\partial E}{\partial o_j^{(k)}} \frac{\partial o_j^{(k)}}{\partial a_j^{(k)}}$ den Fehler des j-ten Neurons der k-ten Schicht beschreibt.

Wie in Formel 16 zu sehen, können für die Ausgabeschicht l_L die Knotenfehler $\delta_j^{(L)}$ direkt mit Hilfe der Fehlerfunktion E berechnet werden.

$$\delta_j^{(L)} = \frac{\partial E}{\partial o_j^{(L)}} \frac{\partial o_j^{(L)}}{\partial a_j^{(L)}} \overset{o_j^{(L)} = \hat{y}_j}{=} E'(\hat{y}_j, y_j) \varphi^{(L)'}\left(a_j^{(L)}\right) \tag{16}$$

Für alle weiteren Schichten $l_k, 1 \leqslant k \leqslant L-1$ ist dies nicht so einfach möglich, da E nur für die Ausgabeschicht definiert ist. An dieser Stelle kommt das Prinzip der Fehlerrückführung zum Tragen. Um dieses Konzept näher zu erläutern, wird im Folgenden anhand der Abhängigkeiten aus Abbildung 2.3.4 beispielhaft die partielle Ableitung $\frac{\partial E}{\partial o_i^{(L-1)}}$ gebildet. Wie dort zu sehen ist, existieren zwei mögliche Pfade, über die $o_i^{(L-1)}$ den Fehler E beeinflussen kann: $(o_i^{(L-1)} \to a_1^{(L)} \to o_1^{(L)} \to E)$ und $(o_i^{(L-1)} \to a_2^{(L)} \to o_2^{(L)} \to E)$. Daher müssen nach der verallgemeinerten Kettenregel beide Pfade einzeln differenziert und dann addiert werden:

$$\frac{\partial E}{\partial o_i^{(L-1)}} = \frac{\partial E}{\partial o_1^{(L)}} \frac{\partial o_1^{(L)}}{\partial a_1^{(L)}} \frac{\partial a_1^{(L)}}{\partial o_i^{(L-1)}} + \frac{\partial E}{\partial o_2^{(L)}} \frac{\partial o_2^{(L)}}{\partial a_2^{(L)}} \frac{\partial a_2^{(L)}}{\partial o_i^{(L-1)}} \tag{17}$$

$$= \delta_1^{(L)} w_{i,1}^{(L)} + \delta_2^{(L)} w_{i,2}^{(L)}. \tag{18}$$

Die Änderungsrate des Fehlers in Bezug auf eine Knotenausgabe der aktuellen Schicht hängt also direkt von den gewichteten Knotenfehlern der nächsten Schicht ab. Dieses Muster kann nun in der aus dem Beispiel (vgl. Gl. 17 und 18) abstrahierten Gleichung 19 erfasst werden.

$$\frac{\partial E}{\partial o_j^{(k)}} = \sum_{l=1}^{M^{(k+1)}} \frac{\partial E}{\partial o_l^{(k+1)}} \frac{\partial o_l^{(k+1)}}{\partial a_l^{(k+1)}} \frac{\partial a_l^{(k+1)}}{\partial o_j^{(k)}} = \sum_{l=1}^{M^{(k+1)}} \delta_l^{(k+1)} w_{j,l}^{(k+1)}. \tag{19}$$

Die Summen in Gl. 19 beginnen bei $l = 1$, da die Fehler der Bias-Knoten aus den Nachfolgeschichten $\delta_0^{(k+1)}$ nicht propagiert werden, weil Kanten der Form $w_{i,0}^{(k+1)}$ nicht existieren. Durch die Anwendung von Gl. 19 können, wie Gl. 20 - 22 zeigen, die Knotenfehler der versteckten Schichten ($k = 1, \ldots, L-1$) berechnet werden.

$$\delta_j^{(k)} = \frac{\partial E}{\partial o_j^{(k)}} \frac{\partial o_j^{(k)}}{\partial a_j^{(k)}} \tag{20}$$

$$= \sum_{l=1}^{M^{(k+1)}} \left(\frac{\partial E}{\partial o_l^{(k+1)}} \frac{\partial o_l^{(k+1)}}{\partial a_l^{(k+1)}} \frac{\partial a_l^{(k+1)}}{\partial o_j^{(k)}} \right) \varphi^{(k)'} \left(a_j^{(k)} \right) \tag{21}$$

$$= \sum_{l=1}^{M^{(k+1)}} \left(\delta_l^{(k+1)} w_{j,l}^{(k+1)} \right) \varphi^{(k)'} \left(a_j^{(k)} \right) \tag{22}$$

Der Vollständigkeit halber sei hier erwähnt, dass die Eingabeschicht von der Backpropagation-Berechnung unberührt bleibt, da sie keine interne Berechnung durchführt, sondern nur die Eingaben liefert.

Unter Verwendung der Gleichungen 16 und 22 lässt sich die Berechnung der Knotenfehler $\delta_j^{(k)}, k = 1, \ldots, L$ in der rekursiven Gleichung 23 zusammenfassen, die effizient in Computern implementiert werden kann.

$$\delta_j^{(k)} = \begin{cases} E'(\hat{y}_j, y_j) \varphi^{(k)'} \left(a_j^{(k)} \right) & , \text{ falls } k = L \\ \sum_{l=1}^{M^{(k+1)}} \left(\delta_l^{(k+1)} w_{j,l}^{(k+1)} \right) \varphi^{(k)'} \left(a_j^{(k)} \right) & , \text{ sonst} \end{cases} \tag{23}$$

Bringt man Gl. 23 in vektorisierte Form

$$\delta^{(k)} = \begin{cases} E'(\hat{y}, y) \odot \varphi^{(k)'} \left(a^{(k)} \right) & , \text{ falls } k = L \\ W^{(k+1)} \delta^{(k+1)} \odot \varphi^{(k)'} \left(a^{(k)} \right) & , \text{ sonst} \end{cases},$$

wobei $\odot$ die elementweise Multiplikation zweier Vektoren beschreibt, so ist die Ähnlichkeit zur Vorwärtsberechnung in neuronalen Netzen gut zu erkennen[6].

6 Hier wird vereinfacht angenommen, dass E' und φ' elementweise auf die Komponenten der Vektoren angewandt wird.

2.4 REKURRENTE NEURONALE NETZE

Rekurrente neuronale Netze (RNNs) sind neuronale Netze, die ihre Eingabe sequenziell abarbeiten und dabei einen sogenannten *Hidden State* verwalten, der es ihnen erlaubt Informationen vorheriger Berechnungen weiterzuverwenden. Dies ist beispielsweise bei dem Übersetzen von Texten relevant, da es nicht genügt jedes Wort individuell zu übersetzen, sondern stets der Kontext jedes Wortes von Relevanz ist. Diese Kontextinformationen werden in RNNs in dem Hidden State zwischengespeichert. Die Berechnung des Hidden States zum Zeitpunkt t lässt sich als eine Funktion

$$h^{<t>} = f(h^{<t-1>}, x^{<t>}; \theta) \tag{24}$$

betrachten, wobei θ hier verschiedene Gewichtsmatrizen und Bias zusammenfasst. Die Funktion f aus Gl. 24 bezieht sich also während der Berechnung des aktuellen Hidden States $h^{<t>}$ nicht nur auf die aktuelle Eingabe $x^{<t>}$, sondern auch auf den vorherigen Hidden State $h^{<t-1>}$.

Für eine tiefergehende Betrachtung der Funktionsweise rekurrenter Netze seien im Folgenden U, V und W Gewichtsmatrizen sowie b und c Bias-Werte eines RNNs. Dann kann die Berechnung dieses RNNs durch die Gleichungen 25 - 27 beschrieben werden [Goodfellow et al., 2016, S. 374].

$$a^{<t>} = b + Wh^{<t-1>} + Ux^{<t>} \tag{25}$$
$$h^{<t>} = \tanh\left(a^{<t>}\right) \tag{26}$$
$$o^{<t>} = c + Vh^{<t>} \tag{27}$$

Die Ausgabe des t-ten Zeitpunkts $o^{<t>}$ kann entsprechend der Anwendung des Netzes verworfen oder (direkt bzw. nach Anwendung einer Aktivierungsfunktion) ausgegeben werden. In modernen tiefen neuronalen Netzen werden teilweise mehrere aufeinander folgende RNN-Schichten übereinander gelegt. Dann werden die Ausgabesequenzen der unteren Schichten als Eingabe für die nächsthöheren Schichten verwendet. Die oberste Schicht gibt schlussendlich das endgültige Ergebnis aus.

2.5 FALTUNGSNETZE

Convolutional Neural Networks (CNNs) sind spezielle tiefe neuronale Netzwerke, die vor allem in der Computer Vision weit verbreitet sind. Ihre Stärke liegt in der Extraktion relevanter Eigenschaften oder Merkmale aus ihren Eingaben durch die Anwendung sogenannter Filter. Ihr Vorteil gegenüber MLPs ist, dass ihre Anzahl an Parametern

nicht wie bei MLPs mit der Größe der Eingabe wächst (siehe Gl. 5). Stattdessen hängt die Anzahl der Parameter von CNNs nur von der Größe und der Anzahl der Filter ab.

Wenn zum Beispiel Bilder analysiert werden, deren Pixelanzahl das Produkt aus den horizontalen und vertikalen Pixeln ist, würde dies für MLPs zu einer sehr großen Anzahl von Neuronen[7] in der Eingabeschicht führen und eine entsprechend große Gewichtsmatrix zwischen der Eingabeschicht und der nachfolgenden Schicht produzieren.

Wenn in einem CNN ausreichend viele und gut gewählte Filter eingesetzt werden, können diese dazu verwendet werden, die Eingabe in eine sinnvolle, abstrahierte Form umzuwandeln, die leichter auf Muster zu untersuchen ist als die ursprüngliche Eingabe.

Eine typische CNN-basierte Netzarchitektur besteht aus (mehreren) konsekutiven Faltungsschichten und Pooling-Schichten, sowie einem MLP am Ende des Modells. Durch die Anwendung von Faltungen und Pooling erzeugen CNNs mit zunehmender Netztiefe eine immer abstraktere Darstellung der Eingabe, die schließlich von einem MLP entsprechend der Zielsetzung weiterverarbeitet wird. Das Ziel beim Lernen eines CNN ist es also, möglichst effektive Filter mit guten Abstraktionsfähigkeiten zu finden.

2.5.1 *Faltungsschichten*

Convolutional Layer oder Faltungsschichten bestehen aus lernbaren Matrizen mit meist nur wenigen Einträgen. Diese Matrizen, genannt *Kernel* K, fungieren als Filter.

Informell ausgedrückt, erhält man das Ergebnis einer Faltung, indem man einen Kernel schrittweise über die Eingabe gleiten lässt und in jedem Schritt das Skalarprodukt aus dem Kernel und den abgedeckten Eingabeelementen bildet. Das Ergebnis einer Faltung wird *Feature Map* genannt. Oft wird zusätzlich der sogenannte *Stride*-Parameter s verwendet, um die Schrittweite festzulegen, mit der der Kernel über die Eingabe bewegt wird. Ein Stride $s = 2$ bedeutet so etwa, dass die Matrix pro Schritt um zwei Felder verschoben wird.

Handelt es sich um eine mehrdimensionale Eingabe, wie z.B. ein Graustufenbild $P \in \mathbb{R}^{H \times W}$ (mit Höhe H und Breite W), steht der Kernel (genauer gesagt die linke obere Ecke des Kernels) am Anfang an der Position $(0, 0)$ und wird dann zunächst so lange über die Spalten (d.h. horizontal) geschoben ($(0, 0) \to (0, 1s) \to (0, 2s), \to (0, 3s) \ldots$), bis dies nicht mehr möglich ist. Erst wenn das Ende der 0-ten Zeile erreicht wird, springt der Kernel an Position $(1s, 0)$ von der aus wieder horizontal fortgeschritten wird.

7 Ein 1920×1080 Bild würde für $1920 \cdot 1080 = 2073600$ Neuronen in der Eingabeschicht eines MLP sorgen.

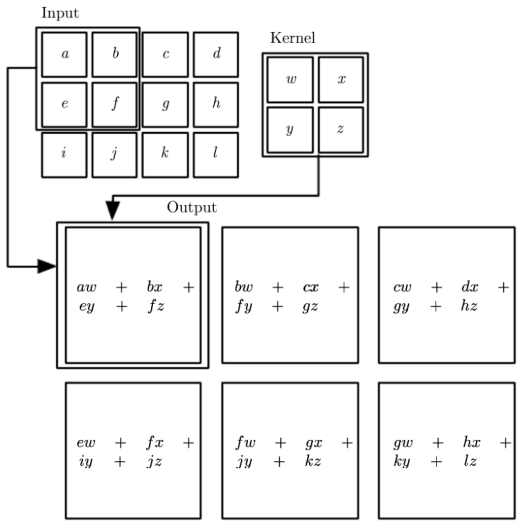

Abbildung 2.5.1: Darstellung einer 2D-Faltung mit stride $s = 1$ [Goodfellow et al., 2016, S. 330].

Neben dem Stride-Parameter gibt es weitere Parameter, die an eine Faltung übergeben werden können, wie z.B. das *Padding*, das angibt, wie die Eingabe an den Rändern erweitert werden soll, bevor gefaltet wird. Die Verwendung von Padding kann sicherstellen, dass alle Eingabeelemente an der gleichen Anzahl von Faltungen teilnehmen.

Betrachtet man einen Kernel $K \in \mathbb{R}^{N \times M}$ und eine Eingabe $I \in \mathbb{R}^{H \times W}$, so ergeben sich die Elemente des Outputs $O \in \mathbb{R}^{H' \times W'}$ einer 2D-Faltung mit folgender Rechenvorschrift[8]:

$$O[i,j] = \sum_{k=0}^{N-1} \sum_{l=0}^{M-1} I[i \cdot s + k, j \cdot s + l] \cdot K[k,l].$$

Abbildung 2.5.1 veranschaulicht die Berechnungen einer 2D-Faltung mit dem Stride $s = 1$.

In der Realität werden mehrere Kernel pro Faltungsschicht verwendet. Formal betrachtet lässt sich die n-dimensionale CNN-Faltung einer m-dimensionalen Eingabe als Funktion

$$conv : \mathbb{R}^{D_1 \times \cdots \times D_m \times d} \to \mathbb{R}^{D'_1 \times \cdots \times D'_m \times k}, \text{ mit } m \geqslant n, \tag{28}$$

8 Die Ergebnisdimensionen hängen von verschiedenen Faktoren (Stride, Padding, etc.) ab und sollen hier nicht genauer betrachtet werden.

definieren, wobei d die Anzahl der Eingabekanäle angibt und k die Anzahl der Filter ist. Alle Dimensionen D_i mit $i = 1, \ldots, m - n$ bleiben von der Faltung unberührt, daher gilt für diese $D_i = D'_i$. Im konkreten Fall eines (20×20) RGB Bildes, das mit 32 Filtern der Größe (2×2) und Stride $s = 1$ gefaltet wird, wäre $conv$ eine Abbildung von $\mathbb{R}^{20\times20\times3}$ nach $\mathbb{R}^{19\times19\times32}$.

2.5.2 *Pooling-Schichten*

Pooling-Schichten werden für das *Downsampling* benutzt. Downsampling bezeichnet die Reduzierung der Eingabegröße unter Inkaufnahme von Informationsverlust. Dies wird in CNNs benötigt um die Sensitivität der Feature Maps gegenüber den genauen Positionen der Eingabeelemente zu reduzieren. Zu diesem Zweck wird, ähnlich wie bei den faltenden Schichten, ein Kernel über die Eingabe bewegt, der diejenigen Eingabeelemente markiert, die kombiniert werden sollen. Der Unterschied zu den Faltungsschichten besteht darin, dass der Kernel selbst hier keine Einträge enthält und daher nicht gelernt wird, sondern nur zur Markierung der zu aggregierenden Bereiche benötigt wird. Auf den markierten Bereichen werden einfache Aggregierungsfunktionen wie die Maximumsbildung (*Max-Pooling*), oder die Berechnung des arithmetischen Mittels (*Avg-Pooling*) angewandt. In der Regel werden quadratische Kernel für das Pooling verwendet, und der Stride-Parameter wird gewöhnlicherweise an die Kernelgröße angepasst. Verwendet man z.B. einen (2×2) Filter, so wird normalerweise der Stride auf den Wert 2 gesetzt und somit sichergestellt, dass jedes Element (höchstens) einmal in einer Pooling-Operation vorkommt.

2.6 LAYER NORMALIZATION

Ba et al. [2016] stellen ein Verfahren für die Normalisierung der Schichtausgaben von neuronalen Netzen vor. Bei der *Layer Normalization* wird der Ausgabevektor der k-ten Schicht $o^{(k)} \in \mathbb{R}^{M^{(k)}}$ unter Verwendung des arithmetischen Mittels $\mu^{(k)} = \frac{1}{M^{(k)}} \sum_{i=1}^{M^{(k)}} o_i^{(k)}$ und der Standardabweichung $\sigma^{(k)} = \sqrt{\frac{1}{M^{(k)}} \sum_{i=1}^{M^{(k)}} \left(o_i^{(k)} - \mu^{(k)}\right)^2}$ normalisiert. Zusätzlich werden zwei lernbare Parameter $g, b \in \mathbb{R}^{M^{(k)}}$ verwendet, die die gleiche Länge wie $o^{(k)}$ besitzen. Die Formel für die Schichtnormalisierung steht in

Gleichung 29, wobei die Subtraktion im Zähler des Bruches für jedes Element von $o^{(k)}$ separat durchgeführt wird.

$$o^{(k)}_{\text{normalized}} = \text{LN}(o^{(k)}) = g \odot \frac{o^{(k)} - \mu^{(k)}}{\sigma^{(k)}} + b \tag{29}$$

2.7 SELF-ATTENTION

Durch die Verwendung von *Attention* können Modelle des maschinellen Lernens bei der Lösung eines Problems einzelnen Problembereichen Relevanzen zuweisen. Dies ahmt die menschliche Wahrnehmung nach, die zwischen wichtigen und unwichtigen Informationen unterscheidet. Im Bereich der Bilderkennung beispielsweise könnte es für neuronale Modelle hilfreich sein, zwischen Vorder- und Hintergrund zu unterscheiden und die Aufmerksamkeit je nach Problemstellung auf den entsprechenden Bereich zu richten

Self-Attention ist eine spezielle Form der Attention, bei der innerhalb einer Sequenz nach Korrelationen gesucht wird [Vaswani et al., 2017]. Dieser Ansatz kann auch auf mehrdimensionale Eingaben wie Bilder ausgeweitet werden. Eine wichtige Eigenschaft von HAR-Modellen, also künstlichen neuronalen Netzwerken für die Lösung des HAR-Problems, ist es, zeitliche Abhängigkeiten innerhalb der Messdaten (z.B. IMU-Sequenzen) zu erkennen. Daher ist HAR ein Anwendungsbereich, in dem Self-Attention eingesetzt werden kann.

Sei $S = [s_1, \ldots, s_n] \in \mathbb{R}^{n \times d}$ eine Sequenz, die aus n Elementen $s_i \in \mathbb{R}^d$ besteht und mit Hilfe des Self-Attention-Mechanismus auf Abhängigkeiten untersucht werden soll.

Für die Berechnung der Self-Attention wird S zuerst mit drei lernbaren Gewichtsmatrizen $W^Q, W^K, W^V \in \mathbb{R}^{d \times d'}$ projiziert. Die Projektionsergebnisse

$$Q = S \cdot W^Q \in \mathbb{R}^{n \times d'},\ K = S \cdot W^K \in \mathbb{R}^{n \times d'} \text{ und } V = S \cdot W^V \in \mathbb{R}^{n \times d'}$$

sind drei abstrakte Repräsentationen von S, dessen Elemente nun die Dimension d' besitzen[9]. Die Namen der Projektionen leiten sich dabei von den Begriffen *Queries*, *Keys* und *Values* ab.

Die Self-Attention-Berechnung beginnt indem die Skalarprodukte von Q und K

$$Q \cdot K^T \in \mathbb{R}^{n \times n}$$

gebildet werden, was in einer $n \times n$ Matrix resultiert, deren Einträge $q_i \cdot k_j^T$ die Skalarprodukte der Queries $q_i = s_i \cdot W^Q$ und Keys $k_j = s_j \cdot W^K$ sind[10].

9 Die Dimension d' kann hier (noch) beliebig gewählt sein.

10 Formal betrachtet handelt es sich bei q_i und k_j um Zeilenvektoren.

Die Matrix $\left(Q \cdot K^T\right)$ wird im Anschluss daran mit dem Faktor $\frac{1}{\sqrt{d}}$ normalisiert[11] und durch die zeilenweise Anwendung der Softmax-Funktion

$$\text{Softmax}(x) = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \tag{30}$$

derart transformiert, dass die Zeilensummen stets 1 ergeben. Das Ergebnis von

$$\text{Softmax}\left(\frac{Q \cdot K^T}{\sqrt{d}}\right) \in \mathbb{R}^{n \times n} \tag{31}$$

ist somit eine Art Korrelationsmatrix, deren Einträge die Attention-Scores $att_{i,j}$ sind. Ein *Attention-Score* $att_{i,j}$ ist ein Maß dafür, wie wichtig das j-te Sequenzelement für das i-te Element ist.

Wechselt man in den Kontext der Übersetzung natürlicher Sprache, wo S eine Folge von Wörtern (z.B. ein Satz) wäre, lassen sich diese Abhängigkeiten intuitiv erklären. Ein großer Wert von $att_{i,j}$ würde in diesem Zusammenhang nämlich bedeuten, dass bei der Übersetzung des Wortes an i-ter Position die Bedeutung des j-ten Wortes wichtig ist. Wenn beispielsweise ein Relativsatz übersetzt wird, wäre es von Vorteil, sich bei der Übersetzung des Relativpronomens auf das zugehörige Substantiv zu konzentrieren, d.h. dem Substantiv (aus der Perspektive des Relativpronomens) einen hohen Attention-Score zuzuweisen.

Multipliziert man die Matrix der Attention-Scores 31 mit den Values $V \in \mathbb{R}^{n \times d'}$, erhält man eine aktualisierte Repräsentation der ursprünglichen Sequenz

$$S' = \text{Self-Attention}(Q, K, V) = \text{Softmax}\left(\frac{Q \cdot K^T}{\sqrt{d}}\right) \cdot V \in \mathbb{R}^{n \times d'}, \tag{32}$$

deren Elemente $s'_i = v_i \cdot (att_{i,1} + \cdots + att_{i,n}) \in \mathbb{R}^{d'}$ gewichtete Aggregierungen der ursprünglichen Elemente $s_i \in S$ darstellen, basierend auf der relativen Wichtigkeit aller Elemente der Eingabesequenz[Shavit and Klein, 2021].

2.7.1 *Self-Multi-Head-Attention*

In Transformern wird die Berechnung der Self-Attention mehrfach parallel durchgeführt, indem mehrere sogenannte *Attention-Heads* eingesetzt werden[Vaswani et al., 2017]. Jeder dieser Köpfe führt die Berechnung aus Gl. 32 durch, wobei jeder Kopf seine eigenen Projektionsmatrizen $W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d \times d'}$ besitzt, die jeweils von d nach

11 Nach Vaswani et al. [2017] sorgt dies im Trainingsprozess für stabilere Gradienten.

$d' = \frac{d}{h}$ abbilden. Damit die Berechnung dimensionserhaltend ausgeführt werden kann, muss die Anzahl der Attention-Heads h ein Teiler der ursprünglichen Dimension d sein.

Seien die projizierten Repräsentationen der Eingabesequenz $S \in \mathbb{R}^{n \times d}$

$$Q_i = S \cdot W_i^Q \in \mathbb{R}^{n \times d'},\ K_i = S \cdot W_i^K \in \mathbb{R}^{n \times d'} \text{ und } V_i = S \cdot W_i^V \in \mathbb{R}^{n \times d'}$$

nun jeweils einem Self-Attention-Head zugeteilt und sei zusätzlich $W_o \in \mathbb{R}^{(h \cdot \frac{d}{h}) \times d}$ eine weitere lernbare Matrix, dann lässt sich die Berechnung der *Self-Multi-Head-Attention* definieren als

$$\text{sMHA}(S) = \underbrace{\text{Concat}(\text{head}_1, \ldots, \text{head}_h)}_{\in \mathbb{R}^{n \times (h \cdot \frac{d}{h})}} \cdot W_o \in \mathbb{R}^{n \times d}\ , \tag{33}$$

$$\text{mit } \text{head}_i = \text{Self-Attention}(Q_i, K_i, V_i).$$

3 VERWANDTE ARBEITEN

Shavit and Klein [2021] stellen für die Lösung des Problems der Aktivitätserkennung ein neuartiges, auf Transformern (Vaswani et al. [2017]) basierendes tiefes neuronales Modell vor. Sowohl für Human Activity Recognition (vgl. Kapitel 2.1), als auch für Smartphone Location Recognition (SLR) konnte das Transformer-basierte Modell auf drei verschiedenen Datensätzen eine CNN-basierte Baselinearchitektur übertreffen. SLR ist eng mit HAR verwandt, unterscheidet sich jedoch darin, dass es nicht darum geht die Aktivitäten der Personen direkt zu identifizieren, sondern die Position (Tasche, Hand, etc.) des Smartphones zu bestimmen, was jedoch auch Aufschluss auf eine mögliche Aktivität der Person geben kann.

Die vorgeschlagene Ende-zu-Ende-Modellarchitektur empfängt Eingabesequenzen (z.B. Inertialmessungen) von konstanter Länge und weist ihnen eine Wahrscheinlichkeitsverteilung über die Aktivitätsklassen zu, anhand welcher eine Klassifizierung vorgenommen werden kann.

Das Transformer-basierte Modell besteht aus vier wesentlichen Komponenten: Einem CNN zur Einbettung der Eingabesequenz; einem lernbaren Tensor, der Positionsinformationen der Sequenzelemente enthält; einem Transformer-Encoder, der die eingebettete Sequenz weiter aggregiert; und schließlich einem MLP, das die endgültige Klassenverteilung erzeugt.

Als Baseline nutzen Shavit and Klein [2021] ein Convolutional Neural Network, das aus zwei Faltungsschichten, einer Max-Pooling-Schicht und einem MLP besteht.

Das vorgeschlagene Modell wurde auf drei Datensätzen evaluiert. Für diese Arbeit sind besonders die Resultate auf dem MotionSense-Datensatz aus Malekzadeh et al. [2019] relevant, da die anderen in Shavit and Klein [2021] verwendeten Datensätze keine bzw. keine reinen HAR-Datensätze sind und daher nicht Gegenstand dieser Arbeit sein werden. Der MotionSense-Datensatz enthält annotierte IMU-Aufnahmen über sechs verschiedene alltägliche Aktivitäten, die von 24 Testpersonen ausgeführt wurden.

Dieser Datensatz wurde für die Evaluation der Modelle in gelabelte IMU-Fenster der Größe 50 (entspricht einer Sekunde) segmentiert. Das Transfomer-basierte Modell erreichte unter den beschriebenen Bedingungen eine Genauigkeit (Accuracy) von 89.6% auf den MotionSense-Daten, während die CNN-Baseline mit 86.2% um 3.4 Prozentpunkte schlechter abschnitt.

Die Autoren aus Niemann et al. [2020] entwickelten mit dem LARa-Datensatz einen HAR-Datensatz im Kontext logistischer Lagerhausarbeiten, der in Kapitel 5.2.1 genauer beschrieben wird.

Auf den IMU-Daten des LARa-Datensatzes, konnten Rueda and Fink [2021] Genauigkeiten von $75.75 \pm 0.4\%$ mit dem von Yang et al. [2015] vorgeschlagenen tCNN, sowie $75.65 \pm 0.4\%$ mit dem IMU-tCNN aus Grzeszick et al. [2017] erreichen. Temporale Convolutional Neural Networks (tCNNs) sind CNNs, die entlang der zeitlichen Dimension falten und so in ihrer Eingabesequenz lokale temporäre Abhängigkeiten erkennen können.

Ordóñez and Roggen [2016] stellen ein tiefes neuronales Netz für die Lösung von IMU-basierter HAR vor, das aus einer Kombination von faltenden und rekurrenten LSTM (Long Short Term Memory) Schichten besteht. Bei den alltäglichen Aktivitäten des OPPORTUNITY-Datensatzes ([Chavarriaga et al., 2013]) übertraf diese Architektur bisherige Modelle im Durchschnitt um 4 Prozentpunkte. Auf dem Skoda-Datensatz aus Stiefmeier et al. [2008] konnte sogar eine Steigerung von 6 Prozentpunkten gegenüber den besten zuvor veröffentlichten Resultaten erzielt werden. Die Autoren konnten feststellen, dass die rekurrenten LSTM-Zellen von grundlegender Bedeutung für die Unterscheidung nah verwandter Gesten (wie z.B. „Tür öffnen“ vs. „Tür schließen“) sind, die sich nur durch die Reihenfolge der Sensordaten unterscheiden.

Diese Bachelorarbeit soll mit einer Vielzahl an Experimenten prüfen, ob die Ergebnisse von Shavit and Klein [2021] auf dem LARa-Datensatz reproduzierbar sind. Dazu gilt es herauszufinden, ob die Transformer-basierte Architektur in der Lage ist, besser zu klassifizierende Repräsentationen der Sequenzen zu erlernen als die Vergleichsarchitekturen. Neben tCNNs wird zusätzlich eine zu Ordóñez and Roggen [2016] ähnliche Architektur getestet, bei der der Transformer-Encoder, des Modells aus Shavit and Klein [2021], durch ein LSTM ersetzt wird. Dies ermöglicht es, den direkten Einfluss des Transformer-Encoders auf die Klassifizierungsleistung einzuordnen und gibt einen tieferen Einblick in die Funktionsweise des Modells.

4 METHODIK

In dieser Arbeit wird eine neuartige Ende-zu-Ende-Architektur zur Lösung von HAR, d.h. zur Erkennung menschlicher Aktivitäten, implementiert und ihre Vorhersagequalität bewertet. Ende-zu-Ende bedeutet in diesem Zusammenhang, dass der gesamte Ablauf der HAR, ausgehend von den rohen IMU-Daten bis hin zu der Klassifizierung in die Aktivitätsklassen, an einem Stück und vollkommen automatisch stattfindet. In den nächsten Abschnitten werden auftretende Probleme bei der Lösung von HAR genauer analysiert und geeignete Lösungsansätze vorgestellt. Des Weiteren werden verschiedene HAR-Modelle präsentiert, die in den Experimenten eingesetzt werden.

4.1 KLASSIFIKATIONSTYPEN

Einige HAR-Datensätze, wie der LARa-Datensatz aus Niemann et al. [2020], enthalten neben den Aktivitätsklassen ergänzende Informationen, die die jeweiligen Bewegungsabläufe genauer charakterisieren. So liefert der LARa-Datensatz für jeden Datenpunkt 19 binäre Attribute, die mit neuronalen Netzen, genauso wie auch die jeweilige Aktivitätsklasse, vorhergesagt werden können.

Die Klassifizierung von Aktivitätsklassen ist ein Sequence-to-One-Problem, also ein Problem, dessen Eingabe eine Sequenz von IMU-Messdaten über einen bestimmten Zeitraum ist, und die Ausgabe die Klasse ist, der das neuronale Modell die höchste Pseudo-Wahrscheinlichkeit zuweist.

Die Klassifizierung von Attributen hingegen ist ein Sequence-to-Many-Problem, da die Eingaben nicht mehr jeweils nur einer Klasse, sondern einer großen Anzahl von Attributen zugeordnet werden. Im Falle von binären Attributvektoren kann das Problem praktischerweise mit einem einzigen neuronalen Modell gelöst werden, dessen Ausgabe ein Vektor mit einem Element pro Attribut ist.

Der Hauptunterschied im Design der HAR-Modelle, die zu den beiden Klassifizierungstypen gehören, liegt in der Ausgabeschicht. Bei der Klassifizierung in einzelne Aktivitätsklassen verfügt die Ausgabeschicht über ein Neuron pro Klasse und verwendet die Softmax-Aktivierungsfunktion (vgl. Gl. 30). Sollen dagegen binäre Attributvektoren vorhergesagt werden, besitzt die Ausgabeschicht ein Sigmoid-Neuron pro Attribut.

4.2 PROBLEMSTELLUNGEN DER AKTIVITÄTSERKENNUNG

Die erste Teilaufgabe für die Lösung von HAR besteht darin, die Datensätze, auf denen die Modelle gelernt werden, vorzubereiten. Ausgehend von den Rohdaten des zugrunde liegenden Datensatzes umfasst dies: Die Entfernung fehlerhafter Datenpunkte; die Normalisierung der Daten; die Implementierung des Sliding-Window Ansatzes; und die Einteilung der erzeugten Fenster in Trainings-, Validierungs- und Testdatensätze.

Das Entfernen fehlerhafter Daten sorgt für eine Fragmentierung der Messdaten und kann bei zeitabhängigen Daten zu unerwünschtem Verhalten führen. Der hier verfolgte Umgang mit fehlerhaften Daten besteht darin, diese aus den Datensätzen zu entfernen und die entstehenden Lücken jeweils mit einer Markierung zu versehen. Anhand der markierten Stellen kann dann bei der Extraktion der Fenster überprüft werden, ob die Fenster tatsächlich aus zusammenhängenden IMU-Messungen bestehen.

IMU-Messdaten sind oft mit einem hohen Maß an Rauschen behaftet, weshalb für die Normalisierung eine Methode gewählt werden sollte, die robust gegenüber Ausreißern ist. Zu diesem Zweck werden die Aufnahmekanäle hier mit der Z-Score-Normalisierung

$$x \leftarrow \frac{x - \mu}{\sigma}$$

normalisiert, die diese gewünschte Eigenschaft besitzt. Außerdem wird den Daten Gaußsches Rauschen mit einem Mittelwert $\mu = 0$ und einer Standardabweichung $\sigma = 0,01$ hinzugefügt, um die Robustheit der HAR-Modelle zu fördern.

Bei der Umsetzung des Sliding-Window Ansatzes werden aufeinanderfolgende IMU-Messungen zu gelabelten Fenstern zusammengefasst. Für jedes Fenster muss ein Label gewählt werden, das die Bewegungsabfolge so gut wie möglich wiedergibt. In nichttrivialen Fällen, z.B. wenn ein Aktivitätswechsel innerhalb des Fensters stattfindet, ist die Wahl des Labels eine Implementierungsentscheidung, für die es mehrere mögliche Lösungen gibt. Hier wurde sich dazu entschieden, jedem Fenster das Label des (linken) mittleren IMU-Datenpunkts zuzuweisen. Da Fenster bei den in der Praxis gewählten Fenstergrößen normalerweise höchstens einen Aktivitätsübergang haben können, entspricht die Wahl eines mittleren Labels dem Modus aller vorkommenden Label innerhalb des Fensters. Während der Entnahme der Fenster wird außerdem stets darauf geachtet, dass nur Fenster erhalten bleiben, die auch in der Realität zu finden wären. Das heißt, dass die Fenster keine Lücken enthalten dürfen und von derselben Testperson und Aufnahme stammen müssen.

Die Wahl der Fenstergröße und der Schrittgröße, d.h. der Anzahl der übersprungenen Datenpunkte nach jeder Extraktion eines Fensters, sind weitere Parameter, die bei

der Implementierung des Sliding-Window Ansatzes berücksichtigt werden müssen. Die Schrittgröße bestimmt einerseits, ob bzw. wie stark sich die einzelnen Fenster überschneiden, und hat andererseits einen direkten Einfluss auf die Gesamtzahl der extrahierten Fenster. Betrachtet man einen Datensatz mit insgesamt N Datenpunkten, dann beträgt die Anzahl der extrahierten Fenster mit der Schrittgröße s etwa N/s.

Für die Klassifizierung der Fenster werden die Testpersonen in drei Gruppen eingeteilt, auf deren Grundlage die gelabelten IMU-Fenster in Trainings-, Validierungs- und Testdaten eingeteilt werden. Die Aufteilung der Daten nach Probanden ist für HAR geeignet, weil sie sicherstellt, dass die Modelle keine personenspezifischen Merkmale lernen, sondern allgemeine menschliche Bewegungsmuster erkennen.

Um HAR-Modelle mit unterschiedlichen Eigenschaften vergleichen zu können, muss sichergestellt werden, dass das Trainingsverfahren kein Modell bevorzugt. Ein Problem bei Modellen mit vielen Parametern ist das sogenannte *Overfitting*[Hawkins, 2004], das auftritt, wenn Modelle so viele Parameter haben, dass sie die Trainingsdaten auswendig lernen können. Dies führt zwar zu einer außergewöhnlich guten Klassifizierungsgenauigkeit auf den Trainingsdaten, aber zu schwachen Ergebnissen bei den unbekannten Daten des Testdatensatzes.

Die Modelle werden epochenweise auf den Trainingsdaten gelernt. Pro Epoche werden alle Trainingsdaten durchlaufen, die zuvor randomisiert in Minibatches eingeteilt wurden.

Zusätzlich zur Optimierung auf den Trainingsdaten werden die aktuellen Modellparameter in jeder Epoche auf den Validierungsdaten bewertet. Nach einer ausreichend großen Anzahl von Epochen bricht der Lernprozess ab und die Modellparameter, die zu der Epoche mit dem geringsten Verlust auf den Validierungsdaten gehören, werden auf den Testdaten evaluiert. Dadurch wird sichergestellt, dass auch Modelle, die nach einer gewissen Zeit beginnen Overfitting zu betreiben, fair behandelt werden. Eine weitere Maßnahme, die in den Experimenten verwendet wird, um Overfitting entgegenzuwirken, ist die Verwendung von *Dropout* (Hinton et al. [2012]), d.h. die Einführung einer Wahrscheinlichkeit darüber, ob Parameter in einem Aktualisierungsschritt ignoriert werden. In den Experimenten wird diese Wahrscheinlichkeit auf 0.1 gesetzt, was bedeutet, dass pro Optimierungsschritt nur 90% der Parameter tatsächlich gemäß der berechneten Gradienten aktualisiert werden.

Eine weitere Schwierigkeit von HAR besteht darin, dass die zugrunde liegenden Datensätze (und damit die extrahierten Fenster) in der Regel sehr unbalanciert sind. Dies kann dazu führen, dass einige Modelle nur die am häufigsten vertretenen Klassen lernen. Ebenso könnte die Qualität eines Modells überschätzt werden, das bei der Klassifizierung zu den häufigen Klassen tendiert. Daher sollte auch die Verteilung der Klassen bei der Interpretation der Modellergebnisse berücksichtigt werden.

Schließlich ist bei der Einordnung (bzgl. der Klassifikationsleistung) der Modelle zu beachten, dass neuronale Netze stochastisch sind, da die Trainingsverfahren auf dem stochastischen Gradientenabstieg beruhen und die Modellparameter zu Beginn zufällig initialisiert werden. Um dennoch generelle Aussagen über die Modelle tätigen zu können, müssen die Experimente mehrfach wiederholt werden. Bei jeder Wiederholung wird der Seed, d.h. der Anfangswert eines Zufallszahlengenerators, verändert, was die Initialisierung der Modellparameter beeinflusst. Das führt dazu, dass die Modelle stets von verschiedenen Startpunkten aus optimiert werden, was den Einfluss einzelner schlechter lokaler Minima während des Gradientenabstieges reduziert.

Die Modelle und das Rahmenprogramm wurden mit Hilfe des in Paszke et al. [2019] beschriebenen Python-Frameworks Pytorch implementiert.

4.3 MODELLARCHITEKTUREN

In diesem Abschnitt wird die Funktionsweise des Transformer-Modells beschrieben. Zudem werden die Vergleichsmodelle vorgestellt, mit denen das Transformer-Modell in den Experimenten verglichen wird.

4.3.1 *Das Transformer-Modell*

Der Transformer wurde erstmals in Vaswani et al. [2017] als neuartige Encoder-Decoder Architektur vorgestellt, die vollständig auf dem Self-Attention-Mechanismus basiert und keine rekurrenten oder faltenden Komponenten enthält. In dieser Arbeit soll der Transformer-Encoder als Komponente einer HAR-Architektur implementiert werden, die mit Hilfe des Transformer-Encoders aussagekräftige zeitliche Beziehungen innerhalb der IMU-Sequenzen erkennen soll.

Obwohl nur der Encoder, d.h. nur eine Hälfte des eigentlichen Transformers, verwendet wird, wird das Modell von nun an als *Transformer-Modell* bezeichnet. Das Modell stammt aus Shavit and Klein [2021] und ist in Abbildung 4.3.1 dargestellt.

Als Eingabe erhält das Transformer-Modell eine Sequenz $S \in \mathbb{R}^{k \times d_{input}}$, wobei k die gewählte Fenstergröße beschreibt und d_{input} die Anzahl der Eingabekanäle ist. Durch die Anwendung vierer Faltungen entlang der Zeitachse, jede mit d Filtern der Größe 1, kombiniert mit GELU-Nichtlinearität, wird S in eine abstrakte, höherdimensionale Repräsentation $E_S \in \mathbb{R}^{k \times d}$ eingebettet. Danach wird E_S an einen lernbaren Vektor $C \in \mathbb{R}^d$ (das sog. *Classtoken*) konkateniert, in den der Transformer-Encoder später die endgültige Repräsentation $Y_c \in \mathbb{R}^d$ der Sequenz schreibt. Bevor dies geschieht, wird $[C, E_S] \in \mathbb{R}^{(k+1) \times d}$ mit einer ebenfalls erlernten Positionseinbettung $E_P \in \mathbb{R}^{(k+1) \times d}$

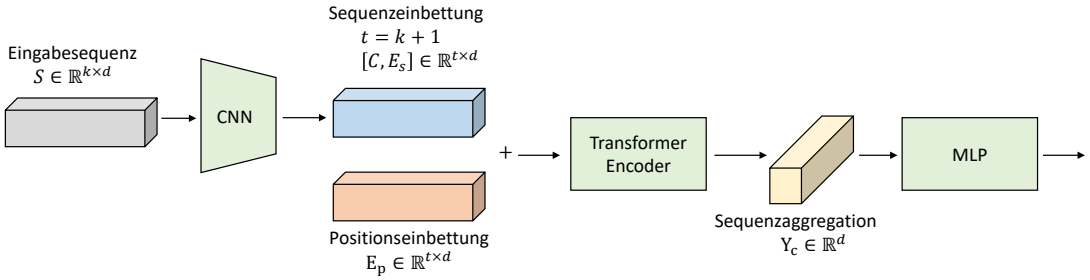

Abbildung 4.3.1: Aufbau des von Shavit and Klein [2021] vorgeschlagenen HAR-Modells, das in dieser Arbeit „Transformer-Default“ genannt wird. Aufbauend auf dieser Architektur entstehen die weiteren in dieser Arbeit implementierten Transformer-Modelle, indem einzelnen Komponenten ausgelassen oder modifiziert werden.

addiert, die es dem Transformer-Encoder ermöglichen soll, die Positionen und Abstände der Sequenzelemente in die Kodierung einzubeziehen. Insgesamt erhält der Transformer-Encoder somit die eingebettete Sequenz

$$Z_0 = [C, E_S] + E_P \in \mathbb{R}^{t \times d} \text{ , mit } t = k + 1$$

als Eingabe und aggregiert diese, durch die Anwendung mehrerer übereinander befindlichen Kodierungsschichten, weiter. Jede Encoderschicht $l = 1, \ldots, L$ des Transformer-Encoders führt auf ihrer Eingabe Z_{l-1} die Self-Multi-Head-Attention Berechnung $sMHA(Z_{l-1})$ (vgl. Formel 33) durch, addiert die residuale Verbindung Z_{l-1} dazu und normalisiert die Summe mit Hilfe der Layer Norm (siehe Formel 29)

$$Z'_l = LN(sMHA(Z_{l-1}) + Z_{l-1}) \in \mathbb{R}^{t \times d}. \tag{34}$$

Das Zwischenergebnis aus Gleichung 34 wird daraufhin an das zweischichtige MLP (mit Hidden-Dimension 2d und GELU-Nichtlinearität) übergeben, dessen Ausgaben wieder mit einer residualen Verbindung Z'_l addiert und normalisiert werden:

$$Z_l = LN(MLP(Z'_l) + Z'_l) \in \mathbb{R}^{t \times d}.$$

Die Ausgabe des gesamten Encoder-Stacks

$$Y_C = Z_L[0] \in \mathbb{R}^d$$

befindet sich am Index 0 der obersten Encoderschicht.

Im letzten Teil des Modells wird Y_C an ein zweischichtiges MLP übergeben, das zunächst die Dimension dieses Vektors auf $d/4$ und dann auf #Klassen bzw. #Attribute reduziert.

Tabelle 4.3.1 zeigt die vier verschiedenen Variationen des Transformer-Modells. Das Modell mit dem Namen „Transformer-Default“ entspricht der soeben vorgestellten Modellarchitektur. Die anderen Modelle variieren verschiedene Komponenten des Transformer-Modells oder lassen diese aus. Das Modell „Transformer-NoPos“ enthält so zum Beispiel keine Positionseinbettung.[1].

Model	Transformer Dimension	sMHA Köpfe	Encoder Schichten	Embedding Schichten	Positions-einbettung
Transformer-Default	64	8	6	4	True
Transformer-Big	128	16	8	4	True
Transformer-NoPos	64	8	6	4	False
Transformer-NoCNN	30	6	6	0	True

Tabelle 4.3.1: Transformer Modelle, die in dieser Arbeit getestet werden. Dass das Transformer-NoCNN-Modell nur 6 (statt 8) Attention-Köpfe besitzt, liegt daran, dass die Anzahl der sMHA-Köpfe ein Teiler der Transformer-Dimension sein muss.

4.3.2 *Vergleichsmodelle*

Als Vergleichsmodelle werden temporale CNNs (tCNNs) und LSTMs verwendet. Temporale CNNs zeichnen sich dadurch aus, dass sie ihre Eingabe entlang der Zeitachse falten. Die tCNN-Modelle werden in zwei Gruppen unterteilt: tCNN_1D mit 1D-Faltungskernel und tCNN_2D, die zweidimensionale Kernel besitzen. Zweidimensionale Kernel können sich als nützlich erweisen, wenn die Eingangsdaten von mehreren synchronisierten IMUs stammen. Dann können die Signale der einzelnen IMUs jeweils entlang der Zeitachse gefaltet und anschließend entweder konkateniert oder gepoolt werden. Das Grundgerüst der tCNNs ist für beide Gruppen identisch und besteht

1 Im Folgenden werden die Anführungszeichen weggelassen.

aus mehreren Faltungsschichten mit ReLU-Aktivierung, gefolgt von einer optionalen Max-Pooling-Schicht und einem zweischichtigen MLP.

Model	Faltungsschichten	Filter	Kernel	Pooling	MLP-Dimension
Baseline	2	64	1	Max	64
tCNN_1D	4	64	5	None	128
tCNN_2D	4	64	(1,5)	None	128

Tabelle 4.3.2: tCNN Modelle, die in dieser Arbeit getestet werden.

LSTMs (Long Short Term Memories) [Hochreiter and Schmidhuber, 1997] sind spezielle rekurrente neuronale Netze, die im Gegensatz zu gewöhnlichen RNNs in der Lage sind, auch weit entfernte Sequenzelemente miteinander in Verbindung zu setzen. LSTMs bestehen, wie auch Transformer-Encoder, aus $L \geqslant 1$ übereinanderliegenden Schichten. Für die Experimente dieser Arbeit sind LSTMs praktisch, weil sie die gleiche Rolle wie Transformer-Encoder einnehmen können. Dadurch kann die Rahmenarchitektur des Transformer-Encoders auch für die LSTMs verwendet werden.

Es werden zwei LSTM-Varianten implementiert, die sich dadurch unterscheiden, dass ihre Ausgaben auf unterschiedliche Art und Weise ausgelesen werden. In einer Variante wird der letzte Hidden State der obersten LSTM-Schicht als Ausgabe verwendet. In der anderen Variante hingegen wird analog zur Transformer-Architektur ein Classtoken C hinzugefügt, diesmal jedoch am Ende der Sequenz. Diese Änderung ist auf die Funktionsweise von LSTMs zurückzuführen, da diese im Gegensatz zu Transformer-Encodern ihre Eingaben sequenziell verarbeiten. Daher kann eine aggregierte Darstellung der Eingabesequenz von einem LSTM nur in ein Token geschrieben werden, das sich am Ende der Sequenz befindet. Die übrige Struktur des LSTM-Modells entspricht genau der des Transformer-Modells.

Model	LSTM Dimension	LSTM Schichten	LSTM Output	Embedding Schichten	Positions-einbettung
LSTM-Hidden	128	2	Hidden	4	True
LSTM-Token	128	2	Token	4	True

Tabelle 4.3.3: LSTM Modelle, die in dieser Arbeit getestet werden.

Die Tabellen 4.3.2 und 4.3.3 zeigen die Modelle, mit denen die Transformer-Architektur experimentell verglichen wird.

4.4 METRIKEN

Die Qualität von Klassifikatoren lässt sich durch geeignete Metriken auf den Testdaten messen. Sei $X = \{(x_1, y_1), \ldots, (x_N, y_N)\}$ der Testdatensatz mit N Eingabe-Ausgabe-Paaren, auf dem der Klassifikator f bewertet wird.

Eine simple Metrik ist die *Accuracy*

$$\text{Accuracy} = \frac{|\{(x_i, y_i) \in X \mid f(x_i) = y_i\}|}{N},$$

die das Verhältnis zwischen der Anzahl korrekter Klassifikationen und der Größe des Testdatensatzes angibt.

Geht man eine Ebene tiefer und betrachtet die Klassifikationen bezüglich einer einzelnen Klasse c (gilt genauso auch für einzelne Attribute), so lassen sich die Genauigkeit (*Precision*) und die Trefferquote (*Recall*) messen. Die Genauigkeit

$$\text{Precision}_c = \frac{|\{(x_i, y_i) \in X \mid f(x_i) = c \wedge f(x_i) = y_i\}|}{|\{(x_i, y_i) \in X \mid f(x_i) = c\}|}$$

misst für alle Fälle in denen c von dem Klassifikator vorhergesagt wurde, wie häufig es sich tatsächlich um diese Klasse handelt.

Des Weiteren lässt sich die Trefferquote

$$\text{Recall}_c = \frac{|\{(x_i, y_i) \in X \mid y_i = c \wedge f(x_i) = y_i\}|}{|\{(x_i, y_i) \in X \mid y_i = c\}|}$$

bestimmen, die die Wahrscheinlichkeit angibt, dass eine vorliegende Klasse c korrekt klassifiziert wird. Betrachtet man die Teilmenge des Testdatensatzes, die nur Elemente der Klasse c enthält, kann der Recall von c als die Accuracy auf dieser Teilmenge interpretiert werden. Daher gilt auch die Beziehung

$$\text{Accuracy} = \sum_c \frac{N_c}{N} \cdot \text{Recall}_c \ ,$$

wobei $N_c = |\{(x_i, y_i) \in X \mid y_i = c\}|$, d.h. die absolute Häufigkeit der Klasse c, ist. Die Accuracy ergibt sich somit als gewichtetes arithmetisches Mittel der Klassen-Trefferquoten.

Eine Kombination aus Genauigkeit und Trefferquote ist der *F1-Score*

$$F1_c = 2 \cdot \frac{\text{Precision}_c \cdot \text{Recall}_c}{\text{Precision}_c + \text{Recall}_c} \ ,$$

der das harmonische Mittel beider Metriken bildet. Mittelt man den F1-Score der einzelnen Klassen (gewichtet), so erhält man den gewichteten F1-Score

$$wF1 = \sum_{c} \frac{N_c}{N} \cdot F1_c \ ,$$

dessen Wert bei stark unbalancierten Datensätzen, wie es bei HAR zumeist der Fall ist, ein robusteres Qualitätsmaß als die Accuracy ist.

In den Experimenten wird die Accuracy sowohl für die Aktivitätsklassenzuordnung als auch bei der Attributvorhersage verwendet. Zusätzlich wird der gewichtete F1 Score als sekundäre Metrik bei der Klassifizierung von Aktivitätsklassen hinzugezogen.

5 EXPERIMENTE

5.1 MOTIONSENSE

In dem ersten Experiment geht es darum, die Resultate von Shavit and Klein [2021] auf dem MotionSense-Datensatz ([Malekzadeh et al., 2019]) zu reproduzieren und zusätzlich weitere Fenstergrößen auszuprobieren. Von den in Abschnitt 4.3 präsentierten Modellen wird hier die von Shavit and Klein [2021] vorgeschlagene Transformer-Architektur (Transformer-Default) mit der Baseline verglichen, die ebenfalls aus Shavit and Klein [2021] stammt. Beide Modelle werden mit variierenden Fenstergrößen $k = 14, 26, 38, 50$ trainiert und getestet. Jede Konfiguration [HAR-Modell, Fenstergröße] wird mit fünf verschiedenen Seeds wiederholt. Die Seeds beeinflussen die anfänglichen Gewichte der Modelle und die Einteilung der Trainingsdaten in die Minibatches. Für die Analyse einer Konfiguration werden im Folgenden immer das arithmetische Mittel und die Standardabweichung für die Metriken der einzelnen Wiederholungen angegeben. Insgesamt besteht das Experiment aus $2 \cdot 4 \cdot 5 = 40$ (#Modelle · #Fenstergrößen · #Wiederholungen) Durchläufen.

5.1.1 *Datensatz*

Der MotionSense-Datensatz enthält iPhone 6s-Aufnahmen, die entstanden, als Testpersonen das Smartphone, bei der Ausführung unterschiedlicher Aktivitäten, in ihrer Hosentasche trugen. Die 24 Probanden mit heterogenen körperlichen Merkmalen (Geschlecht, Alter, Größe, Gewicht etc.) führten sechs verschiedene Aktionen („Treppe hinab“, „Treppe hinauf“, „Gehen“, „Joggen“, „Stehen“ und „Sitzen “) aus. Insgesamt enthält der Datensatz 1412865 annotierte IMU-Messungen, die mit einer Abtastrate von 50Hz aufgenommen wurden.

Abbildung 5.1.1 zeigt die Verteilung der Aktivitätsklassen (die für einen HAR-Datensatz recht ausgewogen ist).

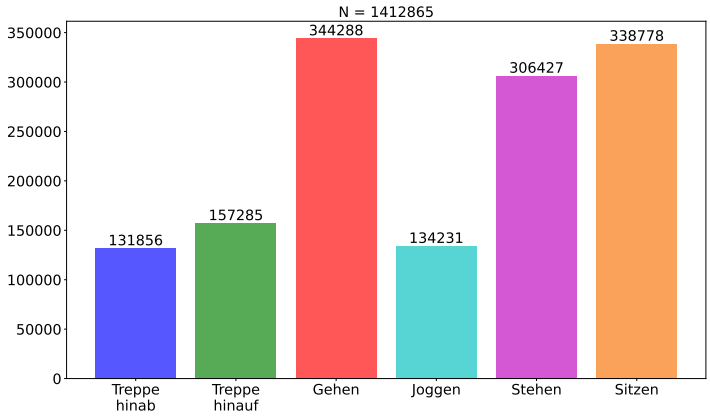

Abbildung 5.1.1: Klassenverteilung des MotionSense-Datensatzes [Malekzadeh et al., 2019].

5.1.2 *Durchführung*

Der MotionSense-Datensatz wird wie in Abschnitt 4.2 der Methodik beschrieben vorverarbeitet. Dabei werden die gelabelten Fenster mit der Schrittgröße $s = 10$ und der zur Konfiguration gehörenden Fenstergröße k extrahiert und dann in Trainings-, Validierungs- und Testdaten unterteilt. Die genaue Anzahl der Fenster hängt von k ab, liegt jedoch in dem Intervall $[139487, 140798]$.

Der Trainingsdatensatz enthält alle Fenster der Testpersonen $[1, 2, 3, 4, 5, 6, 9, 10, 12, 13, 14, 16, 17, 19, 22]$, der Validierungsdatensatz besteht aus den Probanden $[8, 18, 21, 23, 24]$ und der Testdatensatz aus $[7, 11, 15, 20]$. Die Partitionierung der Probanden wurde im Voraus randomisiert, da in Shavit and Klein [2021] kein Hinweis darauf gegeben wurde, wie die Unterteilung der Fenster erfolgt ist. Der Testdatensatz enthält etwa 15% aller Fenster. Von den verbleibenden Fenstern, die für das Training verwendet werden, werden ca. 25% für die Validierung des Trainingsfortschritts verwendet.

Während des Trainings wird der Trainingsdatensatz epochenweise in Minibatches der Größe 128 eingeteilt. Da es sich um ein Mehrklassenproblem handelt, wird die Kreuzentropie (vgl. Formel 11) als Verlustfunktion verwendet. Die Optimierung erfolgt mit dem Adam-Optimierer aus Kingma and Ba [2017] mit den Parametern [weight decay=$1e-4$, $\epsilon = 1e-10$, $\beta_1 = 0.9, \beta_2 = 0.99$)]. Die initiale Lernrate beträgt $1e-4$ und wird alle fünf Epochen halbiert. Während des gesamten Trainings werden die Modellparameter der Epoche mit dem bisher geringsten Verlust auf den Validierungsdaten zwischengespeichert. Da sich recht schnell herausstellte, dass die Modelle in der

Regel sehr früh ihr Optimum finden (oft nach weniger als fünf Epochen), aber dennoch alle Modelle genügend Zeit zur Optimierung haben sollten, endet der Lernprozess nach 30 Epochen.

Die Modellparameter der erfolgreichsten Epoche werden nach dem Training auf den Testdaten ausgewertet. Anhand der Vorhersagen des Modells werden anschließend die für die Analyse ausgewählten Metriken pro Konfiguration berechnet. Nachdem alle Durchläufe abgeschlossen sind, werden die ermittelten Metriken pro Konfiguration zusammengefasst.

5.1.3 *Ergebnisse*

Die gemittelten Accuracy-Werte (und die zugehörigen Standardabweichungen) der Konfigurationen sind in Tabelle 5.1.1 dargestellt. Betrachtet man zunächst die Resultate mit der Fenstergröße $k = 50$, so deckt sich die grundlegende Tendenz, dass das Transformer-Modell dem tCNN-Modell überlegen ist, gut mit den Werten aus Shavit and Klein [2021]. Dort wurden unter Verwendung der gleichen Modellearchitekturen Accuracy-Werte von 86.2% für das CNN und 89.6% für den Transformer ermittelt. Dass die hier reproduzierten Werte von 83.19 ± 1.39 (tCNN) und 89.18 ± 1.96 (Transformer) nicht genau den Werten aus Shavit and Klein [2021] entsprechen, kann verschiedene Ursachen haben. Ein Problem bei der Reproduktion war, dass die Beschreibung der Methodik der Experimente nicht ganz vollständig war. So mussten hier bestimmte Annahmen bezüglich der Vorverarbeitung (Schrittgröße für die Fensterextraktion, sowie das Einteilungsverfahren der Trainings-, Validierungs- und Testdaten) getroffen werden. Ansonsten wurde stets darauf geachtet, alle gegebenen Informationen aus Shavit and Klein [2021] sowie aus der dazugehörigen Codebasis zu übernehmen. In Anbetracht dieser Tatsachen zeigt die Reproduktion eine gute Übereinstimmung mit den dort präsentierten Ergebnissen. Interessanterweise zeigen die Accuracy-Werte für die Fenstergröße $k = 50$ der Reproduktion einen noch stärkeren Leistungsgewinn (6 Prozentpunkte) des Transformer-Modells als in Shavit and Klein [2021] (3.4 Prozentpunkte).

Auch die Resultate mit den anderen Fenstergrößen $k = 14, 26, 38$ zeigen, sowohl für die Accuracy (vgl. Tabelle 5.1.1) als auch für die gemittelten F1-Scores (vgl. Tabelle 5.1.2), ein ähnliches Verhalten. Wie zu erwarten, steigt die Genauigkeit der Klassifizierungen mit zunehmender Fenstergröße für beide Modelle. Es ist jedoch nicht zu erkennen, dass der Unterschied zwischen den beiden Modellen signifikant zunimmt oder abnimmt. Die Fähigkeit des Transformer-Encoders, weit entfernte Sequenzele-

Modell	Accuracy [%]			
	k = 14	k = 26	k = 38	k = 50
Transformer-Default	**85.22 ± 0.86**	**88.61 ± 0.85**	**88.53 ± 2.21**	**89.18 ± 1.96**
Baseline	80.52 ± 1.22	82.31 ± 1.37	81.95 ± 1.05	83.19 ± 1.37

Tabelle 5.1.1: Die gemittelten Accuracies der Konfigurationen auf den Aktivitätsklassen des MotionSense-Datensatzes.

Modell	wF1-Score [%]			
	k = 14	k = 26	k = 38	k = 50
Transformer-Default	**85.71 ± 0.89**	**88.81 ± 0.83**	**88.73 ± 2.21**	**89.40 ± 1.85**
Baseline	81.28 ± 1.20	83.08 ± 1.22	82.73 ± 1.15	83.68 ± 1.27

Tabelle 5.1.2: Die gemittelten wF1-Scores der Konfigurationen auf den Aktivitätsklassen des MotionSense-Datensatzes.

mente miteinander in Verbindung zu setzen, scheint hier daher nicht ausschlaggebend zu sein.

Insgesamt lässt sich feststellen, dass das Transformer-Modell auf den MotionSense-Daten in der Lage ist, deutlich bessere interne Repräsentationen der Eingabefenster zu erzeugen, was eine höhere Klassifizierungsgenauigkeit ermöglicht.

5.2 LARA

In den zwei folgenden Teilexperimenten soll geprüft werden, ob sich die Resultate des vorherigen Experimentes auch auf die IMU-Daten des LARa-Datensatzes aus Niemann et al. [2020] übertragen lassen. Dazu werden sowohl die Aktivitätsklassen (Unterexperiment 1) als auch die Attributvektoren (Unterexperiment 2) mit verschiedenen Modellen gelernt. Darauf aufbauend wird analysiert, wie gut die einzelnen Modelle (insbesondere die Transformer-Modelle) hier abschneiden.

5.2.1 *Datensatz*

Der LARa-Datensatz enthält Messdaten von 14 Personen, die Kommissioniertätigkeiten unter (simulierten) Realbedingungen durchgeführt haben. Abbildung 5.2.1 zeigt das Szenario, in dem der Datensatz aufgenommen wurde. Die Arbeitsschritte können in sieben Aktivitätsklassen eingeteilt werden („Stehen“, „Gehen“, „Wagen schieben“, „Handling (nach oben)“, „Handling (zentral)“, „Handling (nach unten)“, „Synchroni-

Abbildung 5.2.1: Szenario in dem die Daten für den LARa-Datensatz aufgenommen wurden[Reining et al., 2019].

sation“). Darüber hinaus gibt es die Dummy-Klasse „None“, die Aufnahmen markiert, denen keine Aktivität zugeordnet werden konnte. Nach Niemann et al. [2020] bezieht sich die Aktivität „Handling“ auf Arm- und Handbewegungen der Versuchspersonen, während sie etwas an einem Gegenstand, einer Kiste oder einem Werkzeug bearbeiten. Des Weiteren enthält der Datensatz 19 feingranulare Attribute, die den Aufnahmen zugeordnet sind und ebenfalls klassifiziert werden können. Die Attribute geben einen detaillierteren Einblick in den aktuellen Bewegungsablauf, indem sie z.B. signalisieren, mit welcher Hand die Versuchsperson gerade arbeitet[1]. Zusätzlich zu den hier betrachteten Sensordaten der IMUs enthält der Datensatz auch OmoCap- und RGB-Kameraaufnahmen, die jedoch für diese Arbeit nicht von Bedeutung sind, so dass sich im Folgenden ausschließlich auf die IMU-Daten bezogen wird.

Die IMU-Aufnahmen setzen sich aus den Messungen von fünf synchronisierten IMUs zusammen, die jeweils aus einem Beschleunigungsmesser und einem Gyroskop bestehen. Sie befinden sich an (beiden) Armen und Beinen, sowie an dem Bauch der Testpersonen. Insgesamt enthält somit jeder Datenpunkt einen Vektor der Länge $5 \cdot 6 = 30$, den Index der zugehörigen Klasse, sowie die 19 binären Attribute.

Abbildung 5.2.2 zeigt die Verteilung der Klassen des Datensatzes. Wie dort gut zu sehen ist, ist der LARa-Datensatz stark unbalanciert, da die Aktivität „Handling(zentral)“ mehr als die Hälfte aller Datenpunkte ausmacht.

1 Die zugehörigen Attribute dazu wären: a_9 = Linke Hand, a_{10} = Rechte Hand, a_{11} = Keine Hand.

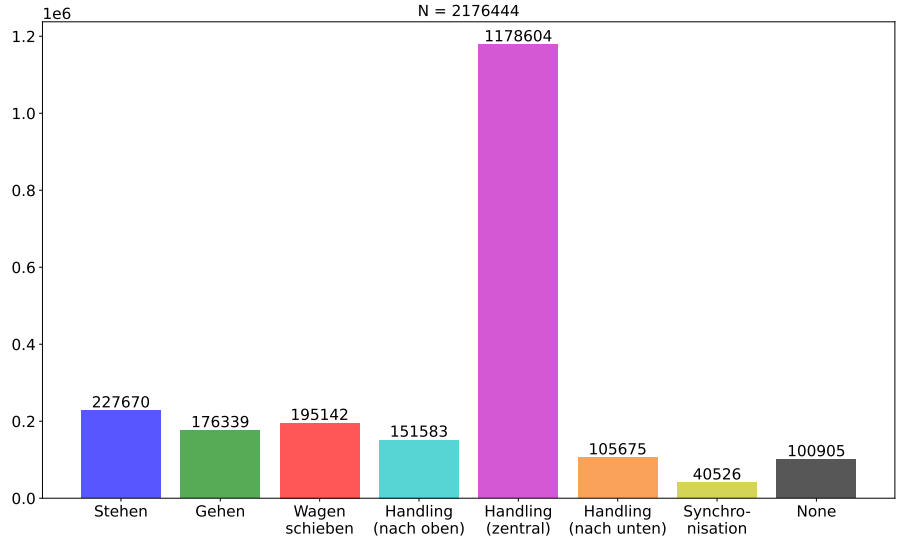

Abbildung 5.2.2: Klassenverteilung des LARa-Datensatzes [Reining et al., 2019].

5.2.2 *Aktivitätsklassen-Vorhersage*

Das erste Teilexperiment beschäftigt sich mit der Klassifizierung von IMU-Sequenzen in die Aktivitätsklassen des LARa-Datensatzes. Auf den gelabelten IMU-Daten werden die verschiedenen Transformer- und Vergleichsmodelle implementiert, die zum einen einer genaueren Einordnung (bzgl. der Klassifizierungsqualität) des Transformer-Modells dienen und zum anderen einen tieferen Einblick in die Funktionsweise des Transformer-Modells geben sollen. Jedes Modell wird mit den Fenstergrößen $k = 30, 40, 50, 60, 70, 80, 90, 100$ jeweils 5 mal trainiert und getestet. Da die Aufzeichnungsrate der LARa-IMUs 100Hz beträgt, entspricht ein Fenster der Größe 100 der gleichen Zeitspanne wie eine Fenstergröße 50 auf den MotionSense-Daten. Die Fenster werden mit einer Schrittgröße $s = 20$ extrahiert, sodass je nach Fenstergröße insgesamt zwischen 102647 und 103441 Fenster extrahiert werden. Die Schrittgröße $s = 20$ ist hier angemessen, da dies dem gleichen zeitlichen Sprung entspricht, mit dem die Fenster aus den MotionSense-Daten extrahiert wurden. Niemann et al. [2020] nutzten für die OMOCap-Daten des LARa-Datensatzes, die mehr Probanden enthalten und daher insgesamt aus wesentlich mehr Aufnahmen bestehen, eine sehr ähnliche Schrittgröße von

$s = 25$. Die für dieses Experiment kleiner gewählte Schrittgröße gleicht den Nachteil des kleineren IMU-Datensatzes etwas aus.

Sämtliche Modelle, die in den Tabellen 4.3.1-4.3.3 dargestellt sind, werden hier für die Aktivitätserkennung eingesetzt. Daher ergeben sich insgesamt $9 \cdot 8 \cdot 5 = 360$ (#Modelle · #Fenstergrößen · #Wiederholungen) Durchläufe.

Durchführung

Wie im vorangegangenen MotionSense-Experiment (vgl. Abschnitt 5.1) werden die LARa-Daten zu Beginn eines jeden Durchlaufs deterministisch in Trainings-, Validierungs- und Testdatenfenster unterteilt. Die Unterteilung sieht dabei folgendermaßen aus: Die Trainingsdaten bestehen aus den Fenstern der Testpersonen $[7, 8, 9, 10]$, die Validierungsdaten bestehen aus den Fenstern der Personen $[11, 12]$ und die Testdaten entstammen den Personen $[13, 14]$. Die hier gewählte Unterteilung stimmt mit dem Vorschlag von Rueda and Fink [2021] überein.

Auch hier werden die Modelle über 30 Epochen hinweg auf Minibatches der Größe 128 mit dem Adam-Optimierer (Hyperparameter wie in Abschnitt 5.1.2 beschrieben) trainiert und anschließend die Modell-Parameter der erfolgreichsten Epoche (bzgl. der Validierungsdaten) gespeichert und auf den Testfenstern evaluiert.

Die auf den Testfenstern errechneten Metriken werden je Konfiguration (auch hier [HAR-Modell, Fenstergröße]) für die weitere Analyse zusammengefasst.

Auswahl der Vergleichsmodelle

Die große Anzahl von Konfigurationen erschwert die Auswertung des Experiments. Daher wird im Folgenden für jeden Modelltyp (tCNN, Transformer, LSTM) eine Vorauswahl getroffen. Auf der Grundlage dieser Vorauswahl werden die Modelltypen dann miteinander verglichen.

TCNN-MODELLE Vergleicht man die Accuracies aus Tabelle 5.2.1 mit den *w*F1-Scores aus der Tabelle 5.2.2, liegen die Accuracies für die tCNN-Modelle bei allen Fenstergrößen über den *w*F1-Scores. Im Hinblick auf die Definition des *w*F1-Scores und den Zusammenhang von Accuracy und Recall (vgl. Abschnitt 4.4), bedeutet dies, dass die durchschnittliche Precision der Modelle stets unterhalb des durchschnittlichen Recalls liegt. Dies wiederum deutet darauf hin, dass die tCNN-Modelle bestimmte (häufige) Klassen bevorzugen, indem sie diese vermehrt vorhersagen. Diese Eigenschaft trifft am stärksten auf das tCNN-2D-Modell zu, dessen Accuracy-Werte die *w*F1-Scores in der Regel um etwa 2 Prozentpunkte übertreffen.

Modell	**Accuracy [%]**							
	k = 30	k = 40	k = 50	k = 60	k = 70	k = 80	k = 90	k = 100
Baseline	71.96 ±0.36	72.38 ±0.46	**72.68 ±0.40**	72.67 ±0.28	72.77 ±0.38	73.05 ±0.57	73.54 ±0.42	73.64 ±0.40
tCNN-1D	71.51 ±0.43	**72.45 ±0.20**	72.43 ±0.57	**72.73 ±0.34**	**73.00 ±0.63**	**73.20 ±0.63**	**73.63 ±0.60**	73.81 ±0.58
tCNN-2D	**71.97 ±0.44**	71.98 ±0.40	72.15 ±0.53	72.15 ±0.52	72.97 ±0.80	73.14 ±0.31	73.33 ±0.50	**74.22 ±0.26**

Tabelle 5.2.1: Die gemittelten Accuracies der tCNN-Konfigurationen auf den Aktivitätsklassen des LARa-Datensatzes.

Modell	**wF1-Score [%]**							
	k = 30	k = 40	k = 50	k = 60	k = 70	k = 80	k = 90	k = 100
Baseline	70.21 ±0.51	70.71 ±0.54	**71.24 ±0.22**	71.29 ±0.13	70.94 ±0.73	**71.69 ±0.53**	72.06 ±0.42	72.23 ±0.36
tCNN-1D	**70.22 ±0.61**	**70.77 ±0.76**	71.12 ±0.48	**71.36 ±0.68**	**71.43 ±0.78**	71.48 ±0.97	**72.30 ±0.75**	**72.60 ±0.49**
tCNN-2D	69.76 ±0.49	70.04 ±0.67	70.21 ±1.18	69.44 ±1.35	70.87 ±1.05	71.33 ±0.63	71.37 ±0.82	71.95 ±0.72

Tabelle 5.2.2: Die gemittelten wF1-Scores der tCNN-Konfigurationen auf den Aktivitätsklassen des LARa-Datensatzes.

Bei beiden Metriken ist die Klassifizierungsgenauigkeit der tCNNs mit eindimensionalem Kernel insgesamt besser als die Genauigkeit des tCNN-2D-Modells. Nur in Bezug auf die Accuracy gibt es zwei Fenstergrößen ($k = 30, 100$), in denen das tCNN-2D-Modell am besten abschneidet. Vergleicht man die beiden tCNNs mit eindimensionalem Kernel, so übertrifft zwar insgesamt das tCNN-1D die Baseline, dennoch liegen die beiden Modelle insgesamt beinahe gleich auf.

Dass das tCNN-1D nur geringfügig bessere Ergebnisse als die Baseline liefert, ist ein erstes interessantes Zwischenergebnis, da die Baseline nur zwei Faltungsschichten mit einem Kernel der Größe 1 besitzt, während das tCNN-1D aus vier Faltungsschichten mit einem Kernel der Größe 5 besteht. Das bedeutet, dass mehr als zwei Faltungsschichten nicht zwingend zu einer eindeutig besseren Leistung führen. Außerdem bekräftigt es die Aussagekraft des vorherigen Experiments, bei dem die Baseline als Vergleichsmodell für das Transformer-Modell verwendet wurde.

Da das tCNN-1D insgesamt am besten abschneidet, wird es als Vertreter der tCNN-Modelle für die weitere Auswertung ausgewählt.

LSTM-MODELLE Bei den LSTM-Modellen zeigen die Accuracies und F1-Scores ein sehr ähnliches Bild, wie in Tabellen 5.2.3 und 5.2.4 zu sehen ist.

Modell	**Accuracy [%]**							
	$k = 30$	$k = 40$	$k = 50$	$k = 60$	$k = 70$	$k = 80$	$k = 90$	$k = 100$
LSTM-Hidden	72.12 ±0.71	72.71 ±0.28	**73.31 ±0.52**	73.42 ±0.51	**73.70 ±0.63**	**74.22 ±0.28**	73.37 ±0.91	73.69 ±0.67
LSTM-Token	**72.89 ±0.59**	**73.07 ±0.50**	72.83 ±0.50	**73.82 ±0.53**	73.35 ±0.41	73.33 ±1.04	**74.20 ±0.29**	**74.07 ±0.36**

Tabelle 5.2.3: Die gemittelten Accuracies der LSTM-Konfigurationen auf den Aktivitätsklassen des LARa-Datensatzes.

Modell	**wF1-Score [%]**							
	$k = 30$	$k = 40$	$k = 50$	$k = 60$	$k = 70$	$k = 80$	$k = 90$	$k = 100$
LSTM-Hidden	70.22 ±0.74	71.34 ±0.31	**71.89 ±0.65**	71.92 ±0.65	**72.27 ±0.91**	**72.75 ±0.45**	71.80 ±1.61	72.18 ±0.81
LSTM-Token	**71.28 ±0.83**	**71.48 ±0.56**	71.20 ±0.84	**72.24 ±0.60**	71.63 ±0.59	71.64 ±1.63	**72.82 ±0.43**	**72.52 ±0.25**

Tabelle 5.2.4: Die gemittelten *w*F1-Scores der LSTM-Konfigurationen auf den Aktivitätsklassen des LARa-Datensatzes.

Auch hier ist, genau wie für die tCNNs zu erkennen, dass beide LSTM-Modelle durchgehend höhere Accuracies als *w*F1-Scores produzieren.

Für einen Großteil der Fenstergrößen ($k = 30, 40, 60, 90, 100$) übertrifft die Klassifikationsleistung des LSTM-Token-Modells die Leistung des LSTM-Modells, dessen Ausgabe aus dem finalen Hidden State ausgelesen wird. Aus diesem Grund wird das LSTM-Token-Modell repräsentativ für die LSTM-Modelle verwendet.

TRANSFORMER-MODELLE Wenn man die Klassifizierungsergebnisse der Transformer-Modelle in Abbildung 5.2.3 (bzw. in den entsprechenden Zeilen der Tabellen A.0.1 und A.0.2) betrachtet, so ist erkennbar, dass mit Ausnahme des Transformer-NoCNN, d.h. des Modells, das die Eingabefenster nicht einbettet, sondern sie direkt an den Transformer-Encoder weitergibt, alle anderen Modelle ähnlich gut abschneiden.

Wie auch bei allen anderen Modelltypen, übertreffen die Accuracies aller Modelle stets die *w*F1-Scores. Dies scheint also eine generelle Eigenschaft von neuronalen Modellen zu sein, die auf den Daten des LARa-Datensatzes gelernt werden.

Die Verwendung eines CNNs zur Einbettung der Eingabefenster scheint in dieser Trainingsumgebung notwendig zu sein, um mit den anderen Modellen mithalten zu können. Dies ist nicht unbedingt nur darauf zurückzuführen, dass das CNN die Eingabesequenz bereits aggregiert, sondern kann auch damit zusammenhängen, dass die Ausgabedimension der Einbettung einen direkten Einfluss auf den Transformer-Encoder hat. Das liegt daran, dass die interne Dimension des Transformers der Dimension der eingebetteten Sequenz entspricht. Findet keine Einbettung statt, so entspricht die Dimension des Transformer der Dimension der Rohdaten. Im Falle des LARa-Datensatzes erhält der Transformer-Encoder daher die interne Dimension 30. Eine zu geringe interne Dimensionierung kann einen negativen Einfluss auf die Leistung des Encoders nehmen. Außerdem hat der Transformer-NoCNN nur 6 statt 8 Attention-Heads, da die Anzahl der Attention-Heads ein Divisor der Transformer-Dimension sein muss. Die tatsächliche(n) Ursache(n) für die schlechte Leistung des CNN-freien Modells können daher durch dieses Experiment nicht eindeutig bestimmt werden. Dennoch lässt sich die Erkenntnis gewinnen, dass die Verwendung einer Einbettungsschicht in den Transformer-Modellen von Vorteil ist.

Unter den drei konkurrenzfähigen Modellen (Transformer-Default, Transformer-Big, Transformer-NoPos) fällt das Modell ohne Positionseinbettung am weitesten ab, da es bei beiden Metriken jeweils fünf mal das zweitschlechteste Modell ist, was nahelegt, dass die Verwendung einer erlernten Positionseinbettung durchaus eine wichtige Komponente für auf Transformer(-Encoder) basierende HAR-Modelle ist. Da hier die Verwendung der Positionseinbettung keinerlei Einfluss auf die restliche Architektur hat und somit der Vergleichbarkeit der Modelle nichts im Wege steht, lässt sich dieses Zwischenergebnis bereits festhalten. Interessanterweise steigt mit zunehmender Fenstergröße die Klassifizierungsgenauigkeit für das Modell ohne Positionseinbettung in ähnlicher Weise wie für die Modelle mit Positionseinbettung.

Für fünf der acht getesteten Fenstergrößen (gilt für beide Metriken, aber mit unterschiedlichen Fenstergrößen) funktioniert der Transformer-Default, d.h. das von Shavit and Klein [2021] vorgeschlagene Modell, am besten. Auch der Transformer-Big, ausgestattet mit zusätzlichen Aufmerksamkeitsköpfen und Kodierschichten sowie der

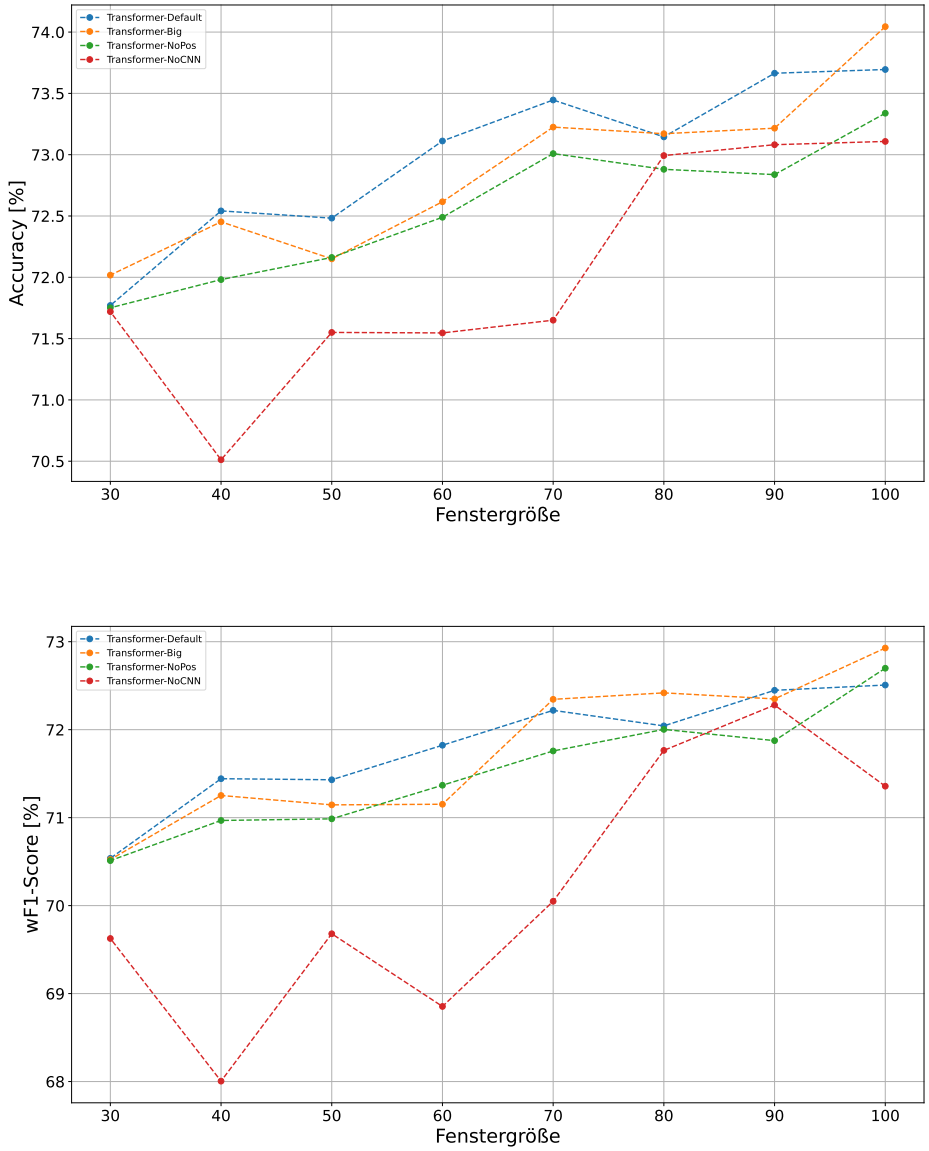

Abbildung 5.2.3: Entwicklung der gemittelten Accuracies und *w*F1-Scores der Transformer-Modelle auf den Aktivitätsklassen des LARa-Datensatzes mit zunehmender Fenstergröße.

doppelten Transformer-Dimension, scheint nicht in der Lage zu sein, weitere sinnvolle Abstraktionen zu lernen. Möglicherweise sorgen die vielen zusätzlichen Parameter bereits früh für starkes Overfitting, was der Qualität des Modells schaden könnte. Aus diesem Grund wird der Transformer-Default für die weitere Auswertung des Experiments verwendet.

Ergebnisse

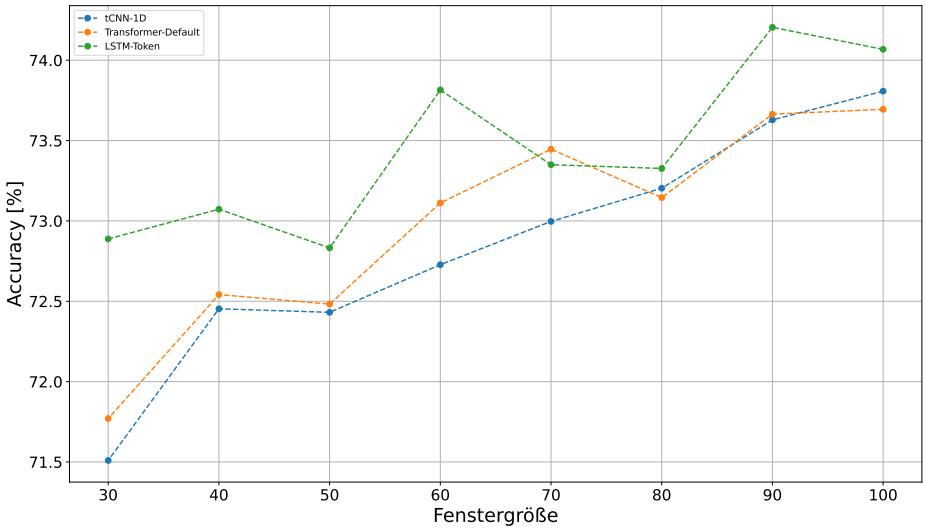

Abbildung 5.2.4: Entwicklung der gemittelten Accuracies der ausgewählten Modelle auf den Aktivitätsklassen des LARa-Datensatzes mit zunehmender Fenstergröße.

Wie Abbildung 5.2.4, sowie die zu den drei ausgewählten Modellen (tCNN-1D, Transformer-Default, LSTM-Token) zugehörigen Zeilen der Tabelle A.0.1 zeigen, ist das LSTM-Modell seinen Konkurrenten, gemessen an der Accuracy, für alle Fenstergrößen außer $k = 70$ überlegen. Bei den anderen beiden Modellen ist zwar zu erkennen, dass die Accuracy-Werte des Transformer-Modells meist über den Accuracies des tCNN-2D-Modells liegen, aber die Werte sind oft ($k = 40, 50, 80, 90$) sehr nahe beieinander. Dies lässt keine eindeutige Schlussfolgerung zu, da bei solch geringen Unterschieden die statistische Ungenauigkeit berücksichtigt werden muss. Aus der Verteilung der Accuracies kann daher nur geschlossen werden, dass das LSTM-Token, gemessen

an dieser Metrik, den beiden anderen Modellen und damit insbesondere auch dem Transformer-basierten Modell überlegen ist.

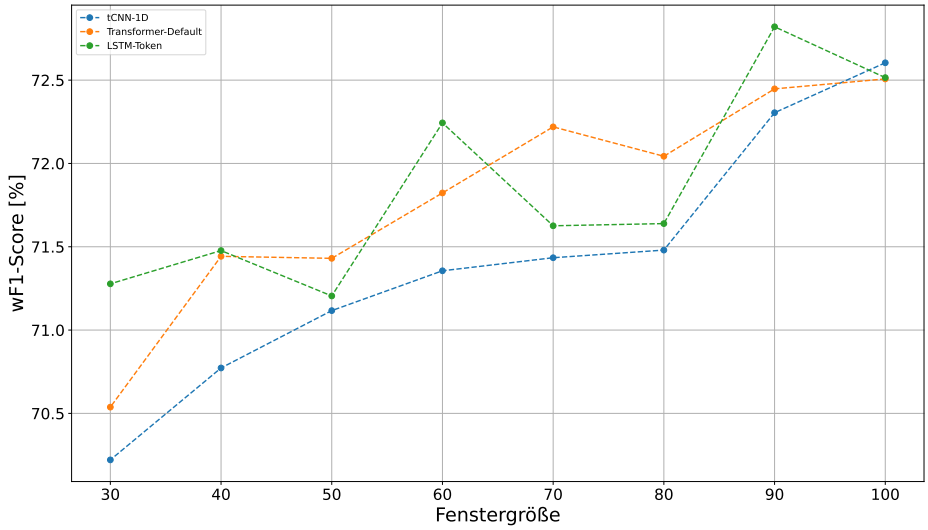

Abbildung 5.2.5: Entwicklung der gemittelten *w*F1-Scores der ausgewählten Modelle auf den Aktivitätsklassen des LARa-Datensatzes mit zunehmender Fenstergröße.

Betrachtet man die Entwicklung der *w*F1-Scores in Abbildung 5.2.5 (bzw. entsprechenden Zeilen der Tabelle A.0.2), so ist nun nicht mehr zu erkennen, dass das LSTM-Modell besser als das Transformer-Modell ist. Stattdessen wechseln sich mit zunehmender Fenstergröße die beiden Modelle regelrecht darin ab, welches aktuell den höchsten *w*F1-Score besitzt. Das tCNN-1D-Modell hingegen liefert recht eindeutig die schwächsten *w*F1-Scores.

Kombiniert man beide Metriken, lässt sich eine grobe Reihenfolge der Klassifizierungsgüte der Modelle erkennen, die besagt, dass das LSTM-Token besser abschneidet als der Transformer-Default, der wiederum das tCNN-1D übertrifft. Diese Reihenfolge ist jedoch mit einer gewissen Vorsicht zu betrachten, da es sich um dicht beieinanderliegende Durchschnittswerte mit einer gewissen Streuung handelt.

Andererseits ist ein wichtiges grundlegendes Ergebnis, dass weder die LSTM- noch die Transformer-Modelle in der Lage sind, signifikant bessere Genauigkeiten auf den IMU-Fenstern des LARa-Datensatzes zu erzielen als die rein faltungsbasierten temporalen CNNs. Signifikant soll hier heißen, dass die Unterschiede zwischen den

Modellen konstant groß genug dafür sind, dass die Standardabweichungen der gemittelten Metriken keine Überschneidungen mehr besitzen, wie dies (vgl. Tabellen A.0.1 und A.0.2) für die Auswertungen auf den LARa-Fenstern noch zumeist der Fall ist. Dies ist ein interessanter Befund angesichts der Ergebnisse des vorangegangenen Experiments 5.1, bei dem das Transformer-Modell die Baseline durchweg um mehrere Prozentpunkte übertraf. Dies bestätigt, zumindest auf den hier getesteten Fenstern des LARa-Datensatzes, die in Shavit and Klein [2021] formulierte These, dass LSTMs ihren internen Sequenzrepräsentationen nicht notwendigerweise informative zeitliche Informationen hinzufügen können, was sich an dieser Stelle auch für den Transformer-Encoder sagen lässt.

5.2.3 *Attribut-Vorhersage*

Im zweiten Unterexperiment wird auf dem LARa-Datensatz die vorherige Aktivitätsklassenvorhersage auf die Attributvorhersage erweitert. Zu diesem Zweck werden die drei zuvor ausgewählten Modelle (tCNN-1D, Transformer-Default und LSTM-Token) mit den Fenstergrößen $k = 50, 100$ für die Attributvorhersage verwendet. Das Training der Modelle erfolgt mit denselben Hyperparametern wie in den vorherigen Experimenten. Dies ermöglicht eine experimentübergreifende Vergleichbarkeit der Resultate, insbesondere mit den Ergebnissen aus dem vorherigen Unterexperiment.

Ergebnisse

Auf der Ebene der reinen Attributvorhersage liegt diesmal, gemessen an den Accuracy-Werten aus Tabelle 5.2.5, auf beiden Fenstergrößen das tCNN-Modell vorne.

Modell	**Accuracy [%]**	
	$k = 50$	$k = 100$
tCNN-1D	$\mathbf{26.66 \pm 0.48}$	$\mathbf{27.97 \pm 1.23}$
Transformer-Default	21.40 ± 3.16	25.15 ± 1.98
LSTM-Token	24.30 ± 0.71	23.99 ± 0.97

Tabelle 5.2.5: Accuracy-Werte der ausgewählten Modelle für die Vorhersage der Attribute des LARa-Datensatzes.

Da die Klassifizierung mehrerer Attribute, von denen einige gleichzeitig auftreten können, ein viel komplexeres Problem ist als die Klassifizierung einzelner Aktivitätsklassen, sind die Genauigkeitswerte hier deutlich niedriger als die Werte in den vorherigen Experimenten.

Betrachtet man beispielsweise die Klassifizierungsergebnisse des tCNN-1D-Modells mit der Fenstergröße $k = 100$ für die einzelnen Attribute in Tabelle 5.2.6, so kann man feststellen, dass dieses Modell bei den meisten Attributen a_i recht akzeptable Genauigkeiten erzielt. Dies lässt sich auch für die anderen Modelle beobachten.

Metrik	Attribute																		
	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{17}	a_{18}	a_{19}
Precision [%]	80.3	69.9	77.3	62.8	81.6	87.2	78.8	0	85.6	72.6	87.6	55	47.7	49.6	62.4	70.9	82.1	0	0
Recall [%]	76.8	56.2	90.1	43.9	82.9	46.2	75.4	0	97.7	90.6	52.1	36	56.8	46.6	82.6	72.2	58	0	0

Tabelle 5.2.6: Durchschnittliche Precision- und Recall-Werte des tCNN-1D-Modells auf den Attributen des LARa-Datensatzes bei einer Fenstergröße von $k = 100$.

Da jede vorkommende Attributkombination $[a_1, \ldots, a_{19}]$ nur zu genau einer LARa-Klasse c gehört, können diese Zuordnungen ($[a_1, \ldots, a_{19}] \mapsto c$) dazu verwendet werden, die Leistungen auf den Attributen in den Kontext der Klassenvorhersage zu bringen. Dazu kann für jede Attributvorhersage die tatsächliche Attributkombination und die vorhergesagte Kombination auf die jeweils entsprechende Klasse des LARa-Datensatzes abgebildet werden. Wird dies getan, ergeben sich die Accuracies und *w*F1-Scores in der Tabelle 5.2.7. Wie dort zu sehen ist, liegen beide Metriken insgesamt nur wenige Prozentpunkte unter den entsprechenden Werten der direkten Klassenvorhersage (vgl. Tabellen A.0.1 und A.0.2).

Modell	Accuracy [%] / wF1 [%]	
	$k = 50$	$k = 100$
tCNN-1D	71.70 ± 0.45/**70.84 ± 0.62**	**72.96 ± 0.67/71.89 ± 0.55**
Transformer-Default	68.57 ± 2.04/64.88 ± 3.06	70.85 ± 1.62/68.32 ± 1.99
LSTM-Token	**72.19 ± 0.50**/70.56 ± 0.44	72.91 ± 0.44/71.24 ± 0.52

Tabelle 5.2.7: Accuracies und *w*F1-Scores der ausgewählten Modelle, nachdem die Attributvektoren auf die Klassen des LARa-Datensatzes übertragen wurden und somit die Attributvorhersage im Kontext der Aktivitätsklassenvorhersage analysiert werden kann.

Auch hier zeichnet sich ab, dass weder die Transformer- noch die LSTM-Modelle in der Lage sind, besser zu klassifizierende innere Repräsentationen der Eingabefenster zu erzeugen, als dies temporale CNNs können.

6 FAZIT

In dieser Arbeit wurde die Aktivitätserkennung anhand annotierter IMU-Sensordaten untersucht. Dabei wurde eine neuartige HAR-Architektur implementiert, die einen Transformer-Encoder verwendet, um zeitliche Abhängigkeiten in den IMU-Sequenzen zu erkennen.

Das von Shavit and Klein [2021] vorgeschlagene Transformer-Modell wurde dazu zum einen mit anderen Modelltypen (tCNN- und LSTM-Modelle) verglichen und zum anderen wurde es durch die Änderung oder dem Weglassen einzelner Komponenten modifiziert. So konnte einerseits gemessen werden, wie die Klassifikationsleistung der Transformer-Modelle im Allgemeinen im Vergleich zu anderen State-of-the-Art-Modellen ist, und andererseits konnte untersucht werden, wie die einzelnen Modellkomponenten die Leistung des Transformer-Modells beeinflussen.

Für die Experimente wurden der MotionSense-Datensatz ([Malekzadeh et al., 2019]) und der LARa-Datensatz ([Niemann et al., 2020]) verwendet. Während der MotionSense-Datensatz alltägliche Aktivitäten enthält, besteht der LARa-Datensatz aus Arbeitsschritten, die im Rahmen von Kommissioniertätigkeiten in einem Lagerhaus stattfanden.

Die Robustheit der Ergebnisse wurde durch die Verwendung vieler verschiedener Experimentparameter, wie z.B. der Fenstergröße des Sliding-Windows, den neun unterschiedlichen HAR-Modellen, sowie der mehrfachen Experimentwiederholung sichergestellt. Insgesamt wurden im Rahmen dieser Arbeit 96 verschiedene Experimente mit jeweils 5 Wiederholungen (d.h. insgesamt 480 Durchläufe) durchgeführt und ausgewertet.

Für den MotionSense-Datensatz bestätigten die Experimente die Ergebnisse von Shavit and Klein [2021], die zeigen, dass bei diesen Aktivitäten die Verwendung eines Transformer-Encoders zusätzliche Informationen zur internen Repräsentation der IMU-Sequenzen hinzufügen kann, was die Klassifizierungsgenauigkeit im Gegensatz zu einer CNN-basierten Baseline signifikant verbessert.

Beim LARa-Datensatz zeigte der Vergleich der verschiedenen Transformer-Modell-Konfigurationen, dass sowohl die Verwendung einer Positionseinbettung als auch die Verwendung der Einbettungsschicht, bestehend aus mehreren Faltungen, die Klassifikationsleistung des Transformer-Modells positiv beeinflusste. Darüber hinaus zeigten die Experimente, dass eine Vergrößerung des Transformer-Modells keinen

Vorteil bringt. Im Vergleich mit weiteren HAR-Modellen zeigte sich auf dem LARa-Datensatz, dass alle Modelltypen sehr ähnlich gut funktionieren, was bedeutet, dass weder Transformer-Encoder noch LSTMs zusätzliche relevante Eigenschaften in den Eingabesequenzen erkennen konnten.

Die Resultate dieser Arbeit geben einen allgemeinen Überblick über die Einordnung der Leistungsstärke von Transformern für die Lösung von HAR. Während die Evaluation der Experimente hier stets auf einer recht abstrakten Ebene stattfand, wäre ein weiterer sinnvoller Ansatz für die Forschung auf diesem Gebiet, die Anzahl der Experimentkonfigurationen deutlich zu reduzieren, was eine detailliertere Analyse einzelner Konfigurationen erlauben würde. Auf der Ebene der Vorhersage einzelner Klassen/Attribute könnten zum Beispiel weitere Schlussfolgerungen darüber gezogen werden, mit welchen Arten von Aktivitäten die verschiedenen Modelle besser zurechtkommen und bei welchen Aktivitäten sie Probleme bei der Klassifizierung haben.

Des Weiteren existieren noch viele andere Möglichkeiten zur Modifikation des Transformer-Modells, die eventuell verbesserte Resultate erzielen könnten. Eine hier nicht geprüfte Modifikation wäre beispielsweise ein verkleinertes Transformer-Modell, das dennoch Gebrauch von einem CNN als Einbettungsschicht macht. Da hier kein Finetuning der Modelle vorgenommen wurde, um die Vergleichbarkeit der Modelle so hoch wie möglich zu halten, könnte eine Anpassung der Hyperparameter des Modells oder der Trainingseinstellungen weitere interessante Ergebnisse liefern.

A

LARA-AKTIVITÄTSKLASSEN

Modell	**Accuracy [%]**							
	k = 30	k = 40	k = 50	k = 60	k = 70	k = 80	k = 90	k = 100
Baseline	71.96 ±0.36	72.38 ±0.46	72.68 ±0.40	72.67 ±0.28	72.77 ±0.38	73.05 ±0.57	73.54 ±0.42	73.64 ±0.40
tCNN-1D	71.51 ±0.43	72.45 ±0.20	72.43 ±0.57	72.73 ±0.34	73.00 ±0.63	73.20 ±0.63	73.63 ±0.60	73.81 ±0.58
tCNN-2D	71.97 ±0.44	71.98 ±0.40	72.15 ±0.53	72.15 ±0.52	72.97 ±0.80	73.14 ±0.31	73.33 ±0.50	74.22 ±0.26
Transformer-Default	71.77 ±0.58	72.54 ±0.63	72.48 ±0.66	73.11 ±0.46	73.45 ±0.58	73.15 ±0.88	73.66 ±0.53	73.69 ±0.41
Transformer-Big	72.02 ±0.61	72.45 ±0.82	72.15 ±0.59	72.62 ±0.30	73.22 ±0.55	73.17 ±0.56	73.22 ±0.55	74.04 ±0.99
Transformer-NoPos	71.75 ±0.52	71.98 ±0.49	72.16 ±0.57	72.49 ±0.80	73.01 ±0.77	72.88 ±0.53	72.84 ±0.95	73.34 ±0.70
Transformer-NoCNN	71.72 ±0.42	70.51 ±2.16	71.55 ±1.28	71.55 ±1.07	71.65 ±1.99	72.99 ±0.43	73.08 ±0.51	73.11 ±0.76
LSTM-Hidden	72.12 ±0.71	72.71 ±0.28	73.31 ±0.52	73.42 ±0.51	73.70 ±0.63	74.22 ±0.28	73.37 ±0.91	73.69 ±0.67
LSTM-Token	72.89 ±0.59	73.07 ±0.50	72.83 ±0.50	73.82 ±0.53	73.35 ±0.41	73.33 ±1.04	74.20 ±0.29	74.07 ±0.36

Tabelle A.0.1: Übersicht über die gemittelten Accuracies aller Modelle, die auf dem LARa-Datensatz für die Aktvitätsklassenvorhersage verwendet wurden. Jedes Experiment wurde fünf mal wiederholt. Zu jedem Durchschnittswert wird auch die Streuung der Accuracies anhand der Standardabweichung angegeben.

Modell	**wF1-Score [%]**							
	k = 30	k = 40	k = 50	k = 60	k = 70	k = 80	k = 90	k = 100
Baseline	70.21 ±0.51	70.71 ±0.54	71.24 ±0.22	71.29 ±0.13	70.94 ±0.73	71.69 ±0.53	72.06 ±0.42	72.23 ±0.36
tCNN-1D	70.22 ±0.61	70.77 ±0.76	71.12 ±0.48	71.36 ±0.68	71.43 ±0.78	71.48 ±0.97	72.30 ±0.75	72.60 ±0.49
tCNN-2D	69.76 ±0.49	70.04 ±0.67	70.21 ±1.18	69.44 ±1.35	70.87 ±1.05	71.33 ±0.63	71.37 ±0.82	71.95 ±0.72
Transformer-Default	70.54 ±0.32	71.44 ±0.55	71.43 ±0.67	71.82 ±0.32	72.22 ±0.85	72.04 ±0.64	72.45 ±0.94	72.51 ±0.64
Transformer-Big	70.52 ±0.73	71.25 ±0.97	71.15 ±0.54	71.15 ±0.36	72.34 ±0.43	72.42 ±0.46	72.35 ±0.86	72.93 ±1.16
Transformer-NoPos	70.51 ±0.61	70.97 ±0.65	70.99 ±0.62	71.37 ±0.76	71.76 ±1.05	72.00 ±0.47	71.88 ±0.96	72.70 ±0.85
Transformer-NoCNN	69.63 ±1.70	68.00 ±3.00	69.68 ±2.34	68.85 ±1.97	70.05 ±2.87	71.76 ±0.59	72.28 ±0.56	71.36 ±1.37
LSTM-Hidden	70.22 ±0.74	71.34 ±0.31	71.89 ±0.65	71.92 ±0.65	72.27 ±0.91	72.75 ±0.45	71.80 ±1.61	72.18 ±0.81
LSTM-Token	71.28 ±0.83	71.48 ±0.56	71.20 ±0.84	72.24 ±0.60	71.63 ±0.59	71.64 ±1.63	72.82 ±0.43	72.52 ±0.25

Tabelle A.0.2: Übersicht über die gemittelten *w*F1-Scores aller Modelle, die auf dem LARa-Datensatz für die Aktvitätsklassenvorhersage verwendet wurden. Jedes Experiment wurde fünf mal wiederholt. Zu jedem Durchschnittswert wird auch die Streuung der *w*F1-Scores anhand der Standardabweichung angegeben.

LITERATURVERZEICHNIS

What is an inertial measurement unit? https://www.vectornav.com/resources/inertial-navigation-articles/what-is-an-inertial-measurement-unit-imu, 2022. [Online, Zugriff 22. Januar 2022].

J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization, 2016.

R. Chavarriaga, H. Sagha, A. Calatroni, S. T. Digumarti, G. Tröster, J. del R. Millán, and D. Roggen. The opportunity challenge: A benchmark database for on-body sensor-based activity recognition. *Pattern Recognition Letters*, 34(15):2033–2042, 2013. ISSN 0167-8655. doi: https://doi.org/10.1016/j.patrec.2012.12.014. URL https://www.sciencedirect.com/science/article/pii/S0167865512004205. Smart Approaches for Human Action Recognition.

I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

R. Grzeszick, J. M. Lenk, F. M. Rueda, G. A. Fink, S. Feldhorst, and M. ten Hompel. Deep neural network based human activity recognition for the order picking process. In *Proceedings of the 4th international Workshop on Sensor-based Activity Recognition and Interaction*, pages 1–6, 2017.

D. M. Hawkins. The problem of overfitting. *Journal of chemical information and computer sciences*, 44(1):1–12, 2004.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580, 2012. URL http://arxiv.org/abs/1207.0580.

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

IBM. Machine learning. https://www.ibm.com/cloud/learn/machine-learning, 2021. [Online, Zugriff 08. Februar 2022].

J. Kiefer and J. Wolfowitz. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, pages 462–466, 1952.

E. Kim, S. Helal, and D. Cook. Human activity recognition and pattern discovery. *IEEE pervasive computing*, 9(1):48–53, 2009.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization, 2017.

O. D. Lara and M. A. Labrador. A survey on human activity recognition using wearable sensors. *IEEE communications surveys & tutorials*, 15(3):1192–1209, 2012.

M. Malekzadeh, R. G. Clegg, A. Cavallaro, and H. Haddadi. Mobile sensor data anonymization. In *Proceedings of the International Conference on Internet of Things Design and Implementation*, IoTDI '19, pages 49–58, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6283-2. doi: 10.1145/3302505.3310068. URL http://doi.acm.org/10.1145/3302505.3310068.

F. Niemann, C. Reining, F. Moya Rueda, N. R. Nair, J. A. Steffens, G. A. Fink, and M. Ten Hompel. Lara: Creating a dataset for human activity recognition in logistics using semantic attributes. *Sensors*, 20(15):4083, 2020.

F. J. Ordóñez and D. Roggen. Deep convolutional and lstm recurrent neural networks for multimodal wearable activity recognition. *Sensors*, 16(1), 2016. ISSN 1424-8220. doi: 10.3390/s16010115. URL https://www.mdpi.com/1424-8220/16/1/115.

A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Z. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. *CoRR*, abs/1912.01703, 2019. URL http://arxiv.org/abs/1912.01703.

P. Ramachandran, B. Zoph, and Q. V. Le. Searching for activation functions, 2017.

C. Reining, F. Niemann, F. Moya Rueda, G. A. Fink, and M. ten Hompel. Human activity recognition for production and logistics—a systematic literature review. *Information*, 10(8):245, 2019.

H. Robbins and S. Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

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

F. M. Rueda and G. A. Fink. From human pose to on-body devices for human-activity recognition. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pages 10066–10073. IEEE, 2021.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

Y. Shavit and I. Klein. Boosting inertial-based human activity recognition with transformers. *IEEE Access*, 9:53540–53547, 2021.

T. Stiefmeier, D. Roggen, G. Ogris, P. Lukowicz, and G. Tröster. Wearable activity tracking in car manufacturing. *IEEE Pervasive Computing*, 7(2):42–50, 2008. doi: 10.1109/MPRV.2008.40.

M. Traeger, A. Eberhart, G. Geldner, A. Morin, C. Putzke, H. Wulf, and L. Eberhart. Künstliche neuronale netze. *Der Anaesthesist*, 52(11):1055–1061, 2003.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

M. Vrigkas, C. Nikou, and I. A. Kakadiaris. A review of human activity recognition methods. *Frontiers in Robotics and AI*, 2:28, 2015.

J. Yang, M. N. Nguyen, P. P. San, X. L. Li, and S. Krishnaswamy. Deep convolutional neural networks on multichannel time series for human activity recognition. In *Twenty-fourth international joint conference on artificial intelligence*, 2015.

W. Zhang, G. Yang, Y. Lin, C. Ji, and M. M. Gupta. On definition of deep learning. In *2018 World Automation Congress (WAC)*, pages 1–5, 2018. doi: 10.23919/WAC.2018.8430387.