technische universität
dortmund

Quantisierung von Convolutional Neural Networks für die Aktivitätserkennung

Bachelorarbeit

Jonas Kieran Röger
29. März 2023

Supervisors:
Prof. Dr.-Ing. Gernot A. Fink
Kai Brandenbusch, M.Sc.

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

INHALTSVERZEICHNIS

1 EINLEITUNG

Die automatisierte Erkennung menschlicher Aktivitäten (HAR, *engl. Human Activity Recognition*) anhand von verschiedenen Sensordaten hat sich in den letzten Jahren zu einem wichtigen Forschungsgebiet entwickelt. Sie spielt in vielen Anwendungen, wie etwa der Gesundheitsüberwachung, Logistik oder Sicherheit eine wichtige Rolle. In der Regel werden die Bewegungsdaten für die Aktivitätserkennung über inertiale Messeinheiten, Gyroskope, und oder Videosysteme erhoben. Mit der zunehmenden Verfügbarkeit von Sensoraufnahmen und der Entwicklung von Deep-Learning-Ansätzen hat sich die Präzision deutlich verbessert.

Unter anderem haben sich CNNs (*engl. Convolutional Neural Networks*) für Aktivitätserkennung als effektiv erwiesen. Die kontinuierlich gemessenen Variablen der Sensoren werden dabei in Fenster eingeteilt, welche von dem CNN durch zeitliche Faltungen verarbeitet werden.

Die Nutzung großer leistungsfähiger Modelle ist auch mit einem erhöhtem Ressourcenverbrauch verbunden. Daher stellt sich der Einsatz effektiver Algorithmen für die Aktivitätserkennung auf mobilen Geräten als große Herausforderung dar. Ein Ansatz zur Reduktion der Ressourcennutzung ist es die Größe des verwendeten Modells durch Quantisierung zu verringern. Dadurch werden die Auflösungen der Gewichte auf weniger Bits reduziert. So kann sowohl der Speicherverbrauch, als auch der Rechenaufwand für die Erkennung der Aktivitäten verringert werden.

Mit der Zeit haben sich verschiedene Methoden zur Quantisierung von künstlichen neuronalen Netzen bewährt. In diesen Untersuchungen wird auf die PTQ (*engl. Post Training Quantization*) und das QAT (*engl. Quantization Aware Training*) zurückgegriffen [Gho+21]. Bei der PTQ werden die Gewichte ohne weitere Anpassungen quantisiert, wohingegen das QAT versucht die Gewichte auf den Effekt der Quantisierung anzupassen. Für eine erfolgreiche Quantisierung muss die Art der Quantisierungsfunktion, sowie ihre Kalibrierung passend gewählt sein.

In dieser Arbeit wird das von Moya Rueda et al. [MR+18] vorgestellte CNN-IMU als CNN zur Aktivitätserkennung verwendet. Mit dem CNN-IMU konnten die Autoren die state-of-the-art Klassifikationsleistung für die zwei etablierten Benchmark-Datensätze übertreffen. Es wird untersucht, wie sich verschiedene Techniken zur Quantisierung auf das CNN-IMU auswirken.

Um den Unterschied zwischen der state-of-the-art Klassifikationsleistung und der quantisierten Klassifikationsleistung festzustellen, werden zunächst die Ansätze von Moya Rueda et al. [MR+18] reproduziert. Dafür werden der Logistik-Datensatz LARa [Nie+20] und die zwei Benchmark-Datensätze Pamap2 [RS12] und Opportunity [Rog+10] verwendet. Die trainierten Modelle dienen dann als Grundlage für die Quantisierung auf zwei bis sieben Bits. Anschließend wird für die PTQ überprüft, welche Kalibrierungsmöglichkeiten zu den besten Resultaten führen. Später wird untersucht, ob sich die Ergebnisse durch die teilweise Quantisierung des CNN-IMUs verbessern lassen. Schlussendlich wird das QAT für das Finetuning der Gewichte verwendet. Die Ergebnisse dieser Arbeit werden in Bezug auf den aktuellen Stand der Quantisierung in der Aktivitätserkennung eingeordnet.

Für das nötige Verständnis werden in Kapitel 2 die Grundlagen zur IMU-basierten Aktivitätserkennung, künstlichen neuronalen Netzen und der Quantisierung erklärt. In Kapitel 3 werden dann zunächst verwandte Ansätze für die Quantisierung in der Aktivitätserkennung zusammengefasst. Anschließend werden in Kapitel 4 die in dieser Arbeit verwendeten Methodiken vorgestellt. Das Vorgehen für die Experimente und das Einordnen ihrer Ergebnisse wird in Kapitel 5 beschrieben. Schlussendlich wird in Kapitel 6 eine Zusammenfassung und ein Ausblick gegeben.

2 GRUNDLAGEN

2.1 IMU-BASIERTE AKTIVITÄTSERKENNUNG

Die menschliche Aktivitätserkennung (HAR, *engl. Human Activity Recognition*) ist ein relevantes Forschungsgebiet für unter anderem den Medizin- und Sicherheitssektor [JBD19]. Es ist denkbar, dass beispielsweise der Gesundheitszustand älterer Personen überwacht werden kann. Mobile Sensorik kann auch bei der Diagnose ungesunder Verhaltensmuster im Alltag helfen. Im Logistikbereich gibt es ebenfalls Anreize für die HAR, die Möglichkeiten sind hier noch zu erkunden [Nie+20]. Die in der HAR verwendete Sensorik ist breit gefächert, von Video- oder Bildaufnahmen über Motion-Capturing-Systeme bis hin zu tragbaren Sensoren, wie etwa in Smartwatches oder Smartphones.

In der IMU-basierten Aktivitätserkennung werden die zu verarbeitenden Daten von tragbaren Sensoren erhoben [LL13]. Inertiale Messeinheiten (IMUs, *engl. Inertial Measurement Units*) messen dabei Vorgänge, wie etwa eine Beschleunigung im Raum, einen Drehimpuls entlang einer Achse und/oder zusätzliche Eigenschaften der Umgebung, wie die Temperatur oder Magnetfelder. In Abbildung 2.1.1 ist eine IMU (schwarzer Chip) auf einem Breakout-Board zu erkennen.

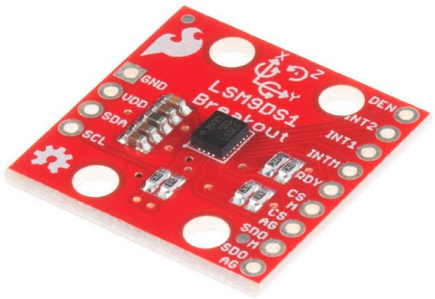

Abbildung 2.1.1: IMU *SparkFun 9DoF* [Spa] auf einem Breakout-Board.

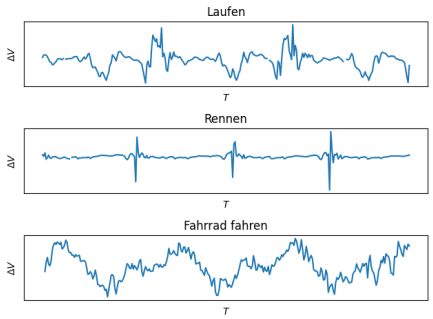

Abbildung 2.1.2: Ausgewählte Beschleunigungsdaten des Pamap2-Datensatzes.

Die IMUs werden an bestimmten Stellen am Körper angebracht. Der Pamap2-Datensatz [RS12] bietet beispielsweise Sensoraufnahmen von alltäglichen Aktivitäten, wobei IMUs am Torso und an den Hand- und Fußgelenken befestigt wurden. Die Problemstellung in der Aktivitätserkennung besteht darin, aus den erhobenen Daten Informationen über die ausgeführten Aktivitäten zu extrahieren. Abläufe wie Laufen, Rennen oder Fahrradfahren, lassen sich dabei verhältnismäßig einfach erkennen. Komplexere Aktivitäten wie das Zubereiten eines Frühstücks sind durch das Zusammenspiel vieler einzelner Abläufe aufwändiger zu erkennen. Abbildung 2.1.2 zeigt Beschleunigungsdaten einer am Fußgelenk befestigten IMU bei verschiedenen Aktivitäten.

In [VNK15] wird eine Übersicht verschiedener Aktivitätsgruppen, basierend auf ihrer Komplexität dargestellt. Auf der niedrigsten Ebene siedeln sich die Gesten, welche primitive Bewegungen repräsentieren, an. Atomare Aktivitäten sind einfache Bewegungsabläufe wie Laufen oder Rennen, welche als Teil komplexerer Aktivitäten gesehen werden können. Mensch-zu-Mensch- und Mensch-zu-Objekt-Aktivitäten beschreiben Interaktionen mit einer Umgebung. Unter Gruppenaktivitäten werden Aktivitäten von mehreren Menschen verstanden. Die abstraktere Kategorie „Verhalten“ enthält Aktivitäten, welche bestimmten psychologischen Verhalten zuzuordnen sind. In der Ereigniskategorie fasst man Aktivitäten, welche ein soziales Verhalten zwischen Menschen repräsentieren, zusammen.

In dieser Arbeit wird die IMU-basierte Aktivitätserkennung von atomaren Aktivitäten untersucht. Detaillierte Informationen über die verwendeten Datensätze sind in Abschnitt 5.2 zu finden.

2.2 MASCHINELLES LERNEN: KÜNSTLICH NEURONALE NETZE

Maschinelles Lernen ist ein Teilgebiet der künstlichen Intelligenz, das es Computern ermöglicht, aus Erfahrung zu lernen und Vorhersagen oder Entscheidungen zu treffen, ohne ausdrücklich programmiert zu werden [JM15]. Beim überwachten Lernen werden einem Computer massenhaft Beispiele für korrekte Ein-/Ausgabepaare gezeigt, sodass dieser bestenfalls Muster in den Daten erkennen kann und den zugrundeliegenden Sachverhalt approximiert.

Techniken aus dem maschinellen Lernen werden häufig dann genutzt, wenn die manuelle Programmierung eines Computers zu aufwändig oder gar unmöglich ist. Die Klassifizierung von Spam-E-Mails ist ein klassisches Problem, welches sich durch maschinelles Lernen mit sehr guter Präzision lösen lässt [AY21]. Dazu werden lediglich große Mengen von E-Mails mit der Kennung „Spam“ oder „kein Spam“ benötigt.

Ein passendes Modell für natürliche Sprache lernt dann, wie unbekannte E-Mails zu klassifizieren sind.

Neben Klassifikation eignet sich das maschinelle Lernen auch für Regressionsprobleme, dabei lernt der Computer aus Merkmalen kontinuierliche Werte vorherzusagen. Dazu gehört beispielsweise die Berechnung von erwarteten Ernteerträgen anhand von Informationen über das Wetter, die Zusammensetzung des Nährbodens, Pflanzentyp und weitere Faktoren [vKC20].

Maschinelle Lernverfahren können auch in interaktiven Umgebungen zum Einsatz kommen, der Computer lernt dabei nicht anhand eines Trainingsdatensatzes, sondern erhält in einer Feedback-Schleife Belohnungen oder Bestrafungen für ausgeführte Aktionen. So lernt er iterativ, durch welche Aktionen die maximale Belohnung erhalten wird. Diese Technik nennt sich Reinforcement Learning und wurde Beispielsweise von Programmen, wie AlphaZero genutzt um die Spiele Schach, Shogi, und Go zu meistern [Sil+18].

Insbesondere künstliche neuronale Netze (KNNs) haben sich in den letzten Jahren als Modelle für wahrnehmungsbasierte Aufgaben bewährt [Abi+19]. In diesem Unterkapitel wird zunächst der Aufbau vom einfachen Perzeptron (Abschnitt 2.2.1) und der Zusammenschluss aus mehreren Perzeptron-Schichten (Abschnitt 2.2.2) erklärt. Der Algorithmus für das Training solcher Modelle wird in Abschnitt 2.2.5 beschrieben. Anschließend wird in Unterkapitel 2.3 eine spezielle KNN Architektur für die Extraktion und Verarbeitung lokaler Muster vorgestellt.

2.2.1 *Perzeptron*

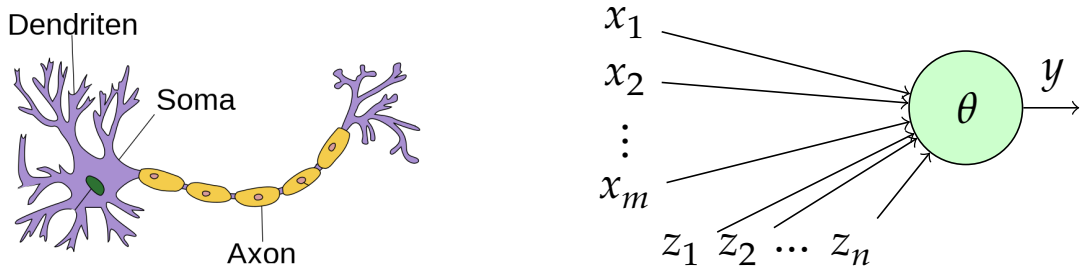

Abbildung 2.2.1: Biologisches Neuron (links) basierend auf [Jar09]; McCulloch-Pitts-Zelle (rechts)

Die Grundbausteine künstlicher neuronaler Netze basieren auf der, 1943 von McCulloch und Pitts eingeführten, mathematischen Darstellung von biologischen Neu-

ronen [MP43]. Die sogenannten McCulloch-Pitts-Zellen können beliebig viele Ausgaben anderer Zellen verknüpfen und eine binäre Ausgabe erzeugen. Dabei seien $x_1, x_2, ..., x_m \in \{0,1\}$ die Eingangssignale, $z_1, z_2, ..., z_n \in \{0,1\}$ hemmende Signale, $y \in \{0,1\}$ die Ausgabe und $\theta \in \mathbb{R}$ eine Hemmschwelle für die Aktivierung der Zelle.

Die Aktivierung der Zelle soll in loser Anlehnung an biologische Neuronen erfolgen, wenn die kumulierten Eingangssignale eine Hemmschwelle überschreiten. Des Weiteren können Zellen gehemmt werden, was ihre Ausgabe deaktiviert. Biologische Neuronen (vgl. Abbildung 2.2.1, links) bestehen aus einer Zelle (Soma), welche ihr Aktionspotential über eingehende Dendriten erhält. Überschreitet das Gesamtpotenzial eine Hemmschwelle, so liegt ein Ausgangspotential am Axon an [CS97, Kap. 2.3]. Die formale Aktivierungsregel der McCulloch-Pitts-Zelle lautet wie folgt:

$$y = \begin{cases} 0, & \text{falls } \bigvee_{i \in \{1...n\}} z_i = 1 \\ 1, & \text{falls } \theta \leq \sum_{j \in \{1...m\}} x_j \\ 0, & \text{sonst} \end{cases}$$

Die Ausgabe der Zelle ist also immer 0, wenn es mindestens ein hemmendes Signal y_i mit dem Wert 1 gibt. Andernfalls ist die Ausgabe genau dann 1, wenn die Summe aller Eingangssignale die Hemmschwelle θ überschreitet [Roj96, S. 32].

Das Perzeptron-Modell wurde 1958 von Rosenblatt erfunden. Grundsätzlich ist es ein Zusammenschluss aus einer oder mehreren McCulloch-Pitts-Zellen. Anstelle der hemmenden Signale werden die Eingangssignale individuell gewichtet. Anstatt des Schwellenwertes θ für die Aktivierung y_j, wird die Ausgabe hier mit einem „Bias" $b_j = -\theta$ addiert und mit 0 verglichen. Die Aktivierung $y_j \in \{0,1\}$ jedes Neurons wird folgendermaßen bestimmt:

$$y_j = \begin{cases} 1, & \text{falls } b_j + \sum_{i \in \{1..m\}} w_{ij} x_i \geq 0 \\ 0, & \text{sonst} \end{cases} \tag{2.2.1}$$

Die m Eingaben $x_i \in \mathbb{R}$ werden für jedes Ausgabeneuron y_j mit den Gewichten w_{ij} und dem Bias b_j linear kombiniert [Ros58]. So lässt sich Beispielsweise ein Perzeptron für die logische Funktion *UND* konstruieren. In Abbildung 2.2.2 werden die beiden Eingaben x_1 und x_2 mit w_1 und w_2 gewichtet, der Bias beträgt den Wert -2. Die Ausgabe ist hier also genau dann 1, wenn x_1 und x_2 jeweils 1 sind.

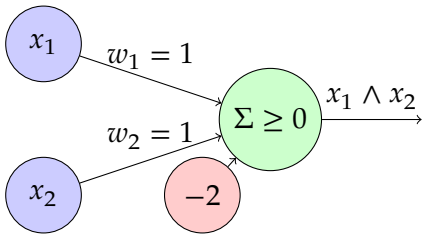

Abbildung 2.2.2: Perzeptron: Logisches UND

Perzeptron-Lernregel

Die automatische Anpassung der Gewichte eines Perzeptrons erfolgt beispielsweise über die „Standardlernregel". Dabei werden die Gewichte w_{ij} zunächst beliebig initialisiert. Anschließend wird für jeden Eingabevektor $X \overset{\text{def.}}{=} [x_1, ..., x_m]^T$ in den Trainingsdaten die erwartete Ausgabe $Y \overset{\text{def.}}{=} [y_1, ..., y_n]^T$ mit der tatsächlichen Ausgabe des Perzeptrons $\hat{Y} \overset{\text{def.}}{=} [\hat{y}_1, ..., \hat{y}_n]^T$ verglichen. Die Gewichte $w_{\star j}$ werden nur angepasst, wenn sich die erwartete Ausgabe y_j von der tatsächlichen Ausgabe $\hat{y}_j$ unterscheidet. Ist die Ausgabe 1, soll aber 0 sein, müssen die Gewichte verringert werden, andernfalls vergrößert. Mit einer Lernrate $\alpha \in \mathbb{R}^+$ werden die Gewichtsanpassungen Δw_{ij} folgendermaßen bestimmt:

$$\Delta w_{ij} = \alpha \cdot (y_j - \hat{y}_j) \cdot x_i$$

Rosenblatt konnte zeigen, dass ein Perzeptron ausreichender Größe mit dieser Lernregel für linear separierbare Trainingsdaten immer konvergiert [Ros62].

2.2.2 *Multilayer Perzeptron / Deep Learning*

Einschichtige Perzeptron-Modelle lernen essenziell linear separierbare Trainingsdaten. Sie sind daher ungeeignet, komplexere Funktionen zu approximieren. Die Idee des mehrschichtigen Perzeptron kurz MLP (*engl. Multilayer Perceptron*) als Weiterführung des einschichtigen Perzeptrons ist es, mehrere Perzeptron-Schichten über ihre Ein-/Ausgaben zu verknüpfen (vgl. Abbildung 2.2.3). Die erste Schicht wird dabei als Eingabeschicht und die letzte als Ausgabeschicht bezeichnet. Alle mittleren Schichten sind sogenannte „hidden Layer". Durch den gerichteten Informationsfluss von der Eingabe hin zur Ausgabe spricht man bei solchen Topologien von „feedforward" Netzen [TM18]. Wenn das Netz Zyklen enthält und somit Informationen mehrfach durch das Netz fließen, spricht man von rekurrenten neuronalen Netzen (RNNs).

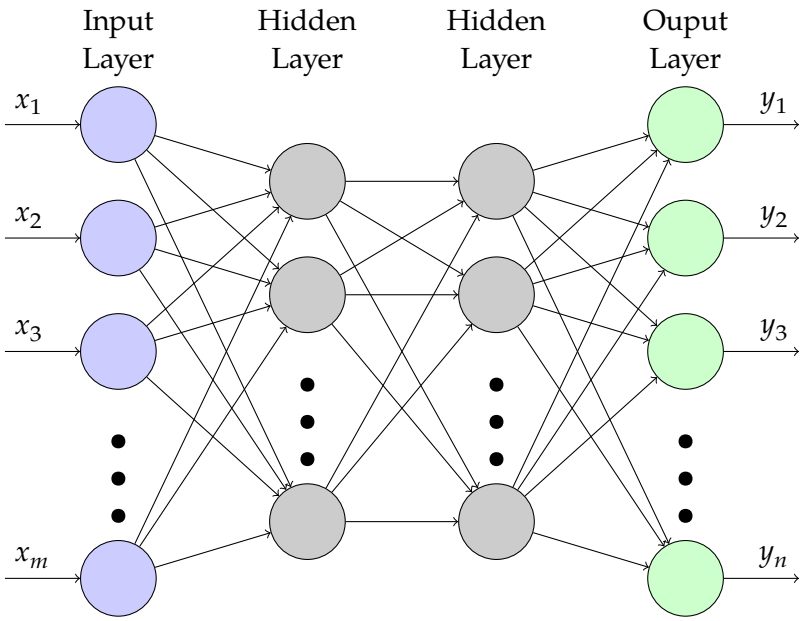

Abbildung 2.2.3: Mehrschichtiges Perzeptron mit m Eingaben, n Ausgaben und zwei versteckten Schichten.

Neben der hier gezeigten Topologie, welche aus voll verbundenen Schichten besteht, sind auch andere Schichten denkbar. Beispielsweise können gezielt Verbindungen ausgelassen werden (Abschnitt 2.2.7) oder gar grundsätzlich andere Verbindungsmuster verwendet werden (Abschnitt 2.3). In der Literatur werden KNNs mit mehreren versteckten Schichten unter dem Begriff „Deep Learning“ zusammengefasst.

Im Folgenden werden der Übersichtlichkeit halber Operationen in der Matrixdarstellung betrachtet. Vektoren werden mit einem Großbuchstaben gekennzeichnet: V. Die Elemente von Vektoren werden über den entsprechenden Kleinbuchstaben mit Index adressiert $V = [v_1, ..., v_n]^T$. Matrizen werden mit einer **dickeren** Schriftart und einem Strich gekennzeichnet: $\overline{\mathbf{M}}$. Auch hier werden die Elemente über den entsprechenden Kleinbuchstaben und Index adressiert. Skalare werden durch Kleinbuchstaben gekennzeichnet. Die Anwendung von Funktionen $f : \mathbb{R} \to \mathbb{R}$ auf einer Matrix oder einem Vektor werden als Punktoperator betrachtet, für $\overline{\mathbf{Y}}, \overline{\mathbf{X}} \in \mathbb{R}^{m \times n}$ gilt also folgendes.

$$\overline{\mathbf{Y}} = f\left(\overline{\mathbf{X}}\right) \overset{def.}{\equiv} y_{ij} = f(x_{ij}) \quad \forall (i,j) \in \{1, ..., m\} \times \{1, ..., n\}$$

Des Weiteren werden Eingaben, sofern nicht anders angegeben, als Zeilenvektor $X = [x_1, x_2, ..., x_m]$ mit m Elementen dargestellt. Analog werden Ausgaben als Zeilenvektor $Y = [y_1, y_2, ..., y_n]$ mit n Elementen dargestellt.

Im Hinblick auf die Implementierung moderner KNNs wird die Aktivierungsregel aus Gleichung 2.2.1 in zwei Schritte aufgeteilt:

$$y'_j = b_j + \sum_{i \in \{1,\dots,n\}} w_{ij} x_i \tag{2.2.2}$$
$$y_j = \Phi(y'_j) = \begin{cases} 1, & \text{falls } y'_j > 0 \\ 0, & \text{sonst} \end{cases}$$

Durch die Separation lässt sich $\Phi : \mathbb{R} \to \mathbb{R}$ als alleinstehende Aktivierungsfunktion betrachten, mehr dazu in Abschnitt 2.2.3. Des Weiteren kann die Linearkombination aus Gleichung 2.2.2 mit Matrix Operationen dargestellt werden. Dafür werden die Eingaben als Zeilenvektor $\mathrm{X} = [x_1, \dots, x_m]$ angeordnet. Die Gewichte w_{ij} für alle Eingaben x_i werden für jedes Neuron j spaltenweise in einer Matrix $\overline{\mathbf{W}} \in \mathbb{R}^{m \times n}$ angeordnet. Im Zeilenvektor $\mathrm{B} = [b_1, \dots, b_n]$ werden die Biaswerte für jedes Neuron gespeichert. In Gleichung 2.2.3 wird der Ausgabevektor $\mathrm{Y}' = [y'_1, \dots, y'_n]$ analog zu Gleichung 2.2.2 bestimmt.

$$\mathrm{Y}' = \mathrm{X} \times \overline{\mathbf{W}} + \mathrm{B} \tag{2.2.3}$$
$$\mathrm{Y} = \Phi(\mathrm{Y}')$$

Für ein MLP mit den Schichten $k \in \{1, \dots, l\}$, den zugehörigen Gewichten $\overline{\mathbf{W}}_k$, Bias B_k und Aktivierungsfunktionen Φ_k lässt sich die Ausgabe Y_l aus der Eingabe X_1 durch den Zusammenhang in Gleichung 2.2.4 errechnen. Die Schichten sind in einem MLP direkt miteinander verknüpft, deshalb gilt $\mathrm{X}_k = \mathrm{Y}_{k-1}$.

$$\mathrm{Y}_k = \Phi_k\left(\mathrm{X}_k \times \overline{\mathbf{W}}_k + \mathrm{B}_k\right) \tag{2.2.4}$$

2.2.3 *Aktivierungsfunktionen*

Für KNNs werden häufig mehrere verschiedene Aktivierungsfunktionen verwendet. In der Literatur ist es gängig die Aktivierungsfunktion in Abhängigkeit des Schichttypen und der Tiefe des Netzes zu wählen [Nwa+18]. Beispielsweise kann man für die Ausgabeschicht eine aufwändig zu berechnende Aktivierungsfunktion wählen, welche Pseudowahrscheinlichkeiten o.Ä. berechnet. Für die versteckten Schichten können dann leichter berechenbare Aktivierungsfunktionen gewählt werden.

In Abschnitt 2.2.2 wurde bereits die Einheitsschrittfunktion (Heaviside) verwendet.

$$\Gamma(x) = \begin{cases} 1, & \text{falls } x \geq 0 \\ 0, & \text{sonst} \end{cases} \tag{2.2.5}$$

Eine ähnlich einfache Aktivierungsfunktion mit weniger Informationsverlust ist die Gleichrichter-Funktion, kurz ReLU (Rectified Linear Unit).

$$\text{ReLU}(x) = \begin{cases} x, & \text{falls } x \geq 0 \\ 0, & \text{sonst} \end{cases} \tag{2.2.6}$$

Wenn die Aktivierungen in einem festen Wertebereich liegen sollen, bieten sich Sigmoidfunktionen, wie die logistische Funktion an.

$$\text{sig}(x) = \frac{1}{1 + e^{-x}} \tag{2.2.7}$$

Die SoftMax-Aktivierungsfunktion (σ) eignet sich besonders für die Ausgabe eines Klassifikators, dabei müssen die Ausgabeneuronen die jeweiligen Klassen $(c_1, ..., c_n)$ repräsentieren. Y gibt also mit jedem y_j die Zugehörigkeit zur Klasse c_j an (höher $\widehat{=}$ besser). SoftMax berechnet eine Pseudowahrscheinlichkeitsverteilung für die jeweiligen Klassenzugehörigkeiten aus der Ausgabe Y,

$$\sigma(\text{X}) = \frac{\exp(\text{X})}{\sum \exp(\text{X})} \tag{2.2.8}$$

wobei exp hier die e-Funktion als Punktoperator und $\sum$ die Summe aller Elemente darstellt.

Damit das Training der KNNs über den Backpropagation Algorithmus (Abschnitt 2.2.5) erfolgen kann, müssen alle verwendeten Aktivierungsfunktionen differenzierbar sein [Roj96]. Sigmoidähnliche Aktivierungsfunktionen haben große Bereiche, in denen der Gradient Werte nahe 0 annimmt. Durch die verkettete Anwendung dieser Funktionen kann der Gradient entsprechend schwindend gering werden. Dieses Problem ist als „vanishing gradient problem“ bekannt und kann das Training von KNNs beeinträchtigen. Eine weitere wichtige Eigenschaft von Aktivierungsfunktionen ist die Nicht-Linearität. Würden beispielsweise ausschließlich lineare Aktivierungsfunktionen in einem MLP genutzt werden, würde sich aufgrund der Assoziativität der Matrixmultiplikation ein einschichtiges Perzeptron ergeben.

2.2.4 *Zielfunktion*

Für das Training künstlich neuronaler Netze mittels Paaren von Eingaben und zugehörigen erwarteten Ausgaben wird zwangsläufig ein Gütemaß für die tatsächlichen Ausgaben des Netzes benötigt. Eine solche Funktion wird Zielfunktion oder auch Verlustfunktion (*engl. Loss Function*) genannt. Zielfunktionen bilden also eine erwartete Ausgabe Y zusammen mit der Ausgabe des Modells $\hat{Y} = f(X)$ auf einen reellen Wert ab $E : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$. Je geringer $E\big(\hat{Y}, Y\big)$, desto besser ist Ausgabe des Modells einzustufen.

Je nach Problemstellung eignen sich verschiedene Zielfunktionen. Eine einfache Zielfunktion für Regressionsprobleme ist das durchschnittliche Fehlerquadrat (MSE, *engl. Mean Square Error*) [Wan+22].

$$\mathrm{MSE}\big(Y, \hat{Y}\big) = \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{n} \tag{2.2.9}$$

Der Fokus dieser Arbeit liegt auf Klassifikationsproblemen. Hier legt die Ausgabe eines Klassifikators die Zugehörigkeit einer Eingabe zu einer aus m Klassen $(c_1, ..., c_m)$ fest. Der Trainingsdatensatz enthält Paare von Eingaben und den zugehörigen Klassen (Label). Die Labels werden in der „eins-aus-k“ (*engl. One-Hot*) Kodierung dargestellt, dabei wird das Label c als Vektor Y mit $y_j = I(c = c_j)\ \forall j \in \{1, ..., m\}$ gespeichert (I bezeichnet die Indikatorfunktion). Der Vektor entsprechend zur Klasse c_j hat also genau im Eintrag j eine 1 stehen, ansonsten überall 0. Die Ausgabe $\hat{Y}$ des Modells gibt mit jedem Eintrag $\hat{y}_j$ die Zugehörigkeit zur Klasse c_j an. Bei Verwendung der SoftMax-Aktivierungsfunktion wird aus $\hat{Y}$ eine Pseudowahrscheinlichkeitsverteilung berechnet. Es bietet sich an, das Label Y ebenfalls als Wahrscheinlichkeitsverteilung mit nur einem möglichen Ereignis, der Klasse selbst, anzusehen. In diesem Fall wird üblicherweise die Kreuzentropie als Distanz, bzw. Fehler zwischen der vorhergesagten und tatsächlichen Wahrscheinlichkeitsverteilung angewandt [Wan+22, S. 195]. Formel für die Kreuzentropie:

$$H(Y, \hat{Y}) = -\sum_{j \in \{1..m\}} y_j \cdot \log \hat{y}_j \tag{2.2.10}$$

2.2.5 *Training neuronaler Netze*

Das überwachte Lernen künstlich neuronaler Netze geschieht üblicherweise mithilfe des Backpropagation Algorithmus [Ama93].

Der allgemeine Trainingsablauf im überwachten Lernen beruht auf der Idee des Gradientenabstiegs. Intuitiv lässt man das Netz eine Trainingseingabe verarbeiten und bestimmt den Fehler mithilfe der genutzten Zielfunktion. Anschließend wird die partielle Ableitung (Gradient) des Fehlers in Bezug auf die Parameter des Netzes bestimmt [Roj96]. So lässt sich der Einfluss jedes einzelnen Parameters auf den Fehler bestimmen. Die Parameter werden dann für jedes Trainingsbeispiel entsprechend angepasst, sodass iterativ ein lokales Minimum für den Fehler gefunden wird.

Seien $\theta_1, \theta_2, ..., \theta_n \in \mathbb{R}$ die Parameter eines KNNs, E ist der Fehler des Netzes für ein oder mehrere Trainingsbeispiele. ∇E ist dabei der Fehlergradient im Parameterraum $\Theta \subseteq \mathbb{R}^n$.

$$\nabla E = \left(\frac{\partial E}{\partial \theta_1}, \frac{\partial E}{\partial \theta_2}, ..., \frac{\partial E}{\partial \theta_n} \right) \tag{2.2.11}$$

Beim Gradientenabstieg wird wiederholt, bis zu einem Abbruchkriterium, der Fehlergradient berechnet. In jeder Iteration lassen sich zusammen mit einer passenden Lernrate $\alpha \in \mathbb{R}^+$ Anpassungen an den Parametern vornehmen.

$$\Delta\theta_i = -\alpha \cdot \frac{\partial E}{\partial \theta_i} \qquad \forall i \in \{1, ..., n\} \tag{2.2.12}$$

$$\theta_i^{\text{neu}} = \theta_i^{\text{alt}} + \Delta\theta_i^{\text{alt}} \qquad \forall i \in \{1, ..., n\} \tag{2.2.13}$$

Die Gleichung 2.2.12 bestimmt den Wert der Anpassung für jeden Parameter, nach Gleichung 2.2.13 werden die Parameter aktualisiert. Die Parameter werden also entgegen des Fehlergradienten mit einer Schrittweite α angepasst. Initial können die Parameter beliebig gewählt werden, üblicherweise werden sie aber mit einer bestimmten Verteilung, wie z.B. der Normalverteilung, gewählt. Mit diesem Verfahren wird sich Schrittweise einem lokalen Minimum mit $\nabla E = 0$ genähert. An dieser Stelle sei angemerkt, dass es Variationen des Gradientenabstiegs gibt, um lokale Minima zu übergehen, siehe Abschnitt 2.2.6.

Die Herausforderung bei der Anwendung des Gradientenabstiegs liegt darin, die partiellen Ableitungen des Fehlers in Bezug auf die einzelnen Parameter zu bestimmen. Der Großteil der Parameter in künstlichen neuronalen Netzen beeinflusst den Fehler nur indirekt, durch eine Vielzahl von Übertragungs- und Aktivierungsfunktionen. Der Backpropagation-Algorithmus ermöglicht es die Fehlergradienten für die Parameter jeder Übertragungsfunktion, durch geschickte Anwendung der Kettenregel, lokal zu berechnen.

Um das Konzept des Backpropagation-Algorithmus zu erklären werden KNNs zunächst, ohne Beschränkung der Allgemeinheit, als eindimensionale Funktionen betrachtet [Roj96]. Knotenpunkte im Netzwerk, werden als Komposition ihrer Übertragungs-

und Aktivierungsfunktion betrachtet, Übertragungsfunktionen führen dabei ein oder mehrere Eingaben gewichtet zusammen. Jede Funktion im Netzwerk wird so als einzelne differenzierbare Einheit betrachtet.

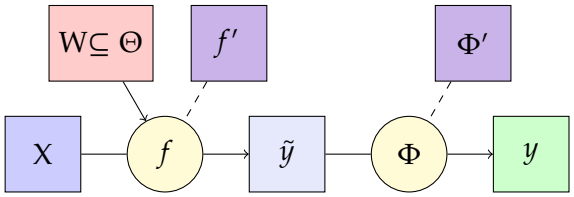

Abbildung 2.2.4: Blockschaltbild eines allgemeinen Knotenpunktes mit Übertragungsfunktion f und Aktivierungsfunktion Φ.

In Abbildung 2.2.4 ist eine allgemeine Darstellung von Knotenpunkten in einem KNN zu sehen. Die differenzierbare Übertragungsfunktionen f führt alle Eingaben X gewichtet zu einem $\tilde{y} \in \mathbb{R}$ zusammen. Für den Fall einer linearen Übertragung, wie sie bereits aus Gleichung 2.2.3 bekannt ist, würde $f(\mathrm{X}) = b + \sum_{x_i \in \mathrm{X}} x_i \cdot w_i$ gelten. Dabei wären die Parameter W die Gewichte w_i zusammen mit dem Bias Wert b. Die Aktivierungsfunktion Φ erzeugt schlussendlich die Ausgabe des Knotenpunktes.

Der Backpropagation-Algorithmus arbeitet in zwei Schritten. Als Erstes wird im sogenannten Forwardpass die Ausgabe regulär durch das Netz berechnet, dabei wird in jeder Einheit ihre Funktion f, sowie die Ableitung f' berechnet. Die Ergebnisse werden in der Einheit gespeichert, als Ausgabe wird nur das Ergebnis von f weitergegeben. Im Backwardpass werden die Ableitungen jeder Einheit von der Ausgabe hin zur Eingabe bestimmt, daher auch der Name „Backpropagation“ (dt. Rückpropagierung).

Die konkrete Backpropagation wird für die drei essenziellen Verzweigungsarten erklärt: Funktionskomposition, lineare Verknüpfungen und parallele Eingaben. Aus diesen Komponenten lassen sich alle feed-forward KNNs bilden.

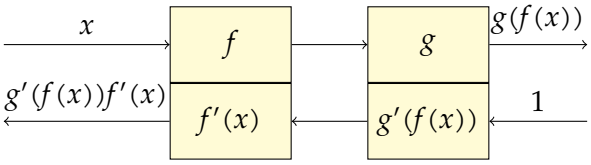

Abbildung 2.2.5: Komposition zweier Funktionen. Forwardpass oben, Backwardpass unten. Die Einheiten der Funktionen f und g sind mit gelben Rechtecken gekennzeichnet, die gespeicherte Ableitung ist jeweils im untere Teil des Rechtecks.

In Abbildung 2.2.5 ist zu sehen, wie der Forward- und Backwardpass bei einer Komposition zweier eindimensionaler Funktionen f und g berechnet wird. Die gelben Recht-

ecke kennzeichnen die Funktionseinheiten, welche im unteren Teil ihre Ableitungen speichern. Während des Forwardpasses wird die Funktion $g(f(x))$ berechnet. Für den Backwardpass multipliziert jede Einheit ihre Ableitung mit der Ableitung der nachfolgenden Einheit, im Sinne der Kettenregel. Platzhaltend ist hier eine 1, als multiplikative Identität, an Stelle der Ableitung nach g stehenden Einheit eingetragen.

In Abbildung 2.2.6 ist eine lineare Kombination zweier Knotenpunkte dargestellt. Hierbei sind a und b die Konstanten der Linearkombination, x und y sind jeweils die Ausgaben der zu kombinierenden Knotenpunkte.

Zuletzt fehlt noch die Handhabung mehrerer Eingaben für einen Knotenpunkt. Dies ist im Grunde genommen eine Verallgemeinerung der linearen Kombination. In Abbildung 2.2.7 berechnet die Einheit die Funktion f, welche von den beiden Ausgaben x und y der vorherigen Einheiten abhängt und einen Ausgabewert produziert. Im Backwardpass gibt die Einheit die partiellen Ableitungen in Bezug auf x und y entsprechend der Kettenregel weiter.

Falls die Ausgabe einer Einheit von mehreren Einheiten verwendet wird, werden beim Backwardpass mehrere Ableitungen zu dieser Einheit zurückpropagiert. Die Einheit summiert alle Ableitungen auf und multipliziert diese mit ihrer eigenen Ableitung. Zusammengefasst arbeitet der Backpropagation-Algorithmus wie folgt:

- Im **Forwardpass** berechnet jede Einheit aus der Eingabe ihre Ausgabe. Die Einheit speichert alle partiellen Ableitung in Bezug auf die Eingaben.
- Im **Backwardpass** sammelt jede Einheit alle zurückpropagierten Ableitungen der Einheiten, die ihre Ausgabe verwenden. Wurden alle Ableitungen gesammelt, wird die Summe jeweils mit den gespeicherten partiellen Ableitungen multipliziert und an die vorherigen Einheiten weitergegeben. Die Ausgabeeinheit nutzt anstelle eines zurück propagierten Wertes eine 1.

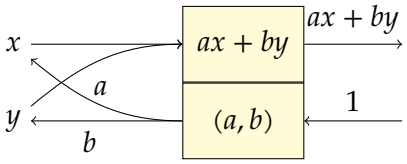

Abbildung 2.2.6: Lineare Kombination zweier Knotenpunkte mit den Ausgaben x und y. Forwardpass oben, Backwardpass unten.

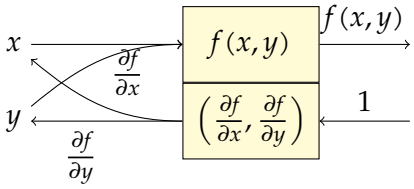

Abbildung 2.2.7: Knotenpunkt mit der mehrdimensionalen Funktion f.

Durch die Anwendung des Backpropagation-Algorithmus über die genannten Verzweigungsarten lässt sich also für alle Einheiten in einem feed-forward KNN die Ableitung als Teil der Gesamtfunktion berechnen. Für einen Korrektheitsbeweis des Algorithmus sei auf [Roj96, Proposition 7.2.1] verwiesen.

Um nun konkret den Fehlergradienten in Bezug auf die Parameter zu berechnen, wird die Zielfunktion als letzte Einheit im KNN angesehen. Bei Anwendung des Backpropagation-Algorithmus kennt folglich jede Einheit ihre eigene Ableitung als Teilfunktion von $E(X)$. In jeder parametrisierten Einheit, wie lineare Einheiten mit Gewichten, liegt also der Teil des Fehlergradienten vor, welcher sich auf die Eingaben der Einheit bezieht. Unter der Annahme einer konstanten Eingabe lässt sich der Fehlergradient in Bezug auf die Parameter der Einheit bestimmen.

Im folgenden Beispiel bezeichnet E den Fehler des Netzes $E(Y, \hat{Y})$, wobei $\hat{Y}$ die Ausgabe des Netzes und Y die erwartete Ausgabe ist. Am Anfang des Backwardpasses wird der Fehlergradient in Bezug auf die Netzausgabe berechnet. Ist die Zielfunktion beispielsweise die MSE-Funktion (Gleichung 2.2.9), ergibt sich folgender Zusammenhang:

$$\frac{\partial E}{\partial \hat{Y}} = \frac{2}{n}(\hat{Y} - Y) \tag{2.2.14}$$

Während der Backpropagation errechnet jede Einheit den Fehlergradienten in Bezug auf ihre Parameter und in Bezug auf ihre Eingabe. Zweiteres wird als Fehlergradient zurückpropagiert. Für dieses Beispiel ist die Einheit f eine lineare Übertragungsfunktion mit Eingabevektor X, Gewichten w_i und Bias b. Der durch den Backpropagation-Algorithmus zu dieser Einheit zurückpropagierte Wert δ ist der Fehlergradient in Bezug auf $f(X)$. Weil X als konstant angenommen wird, lässt sich mit der Kettenregel der Fehlergradient in Bezug auf die Parameter w_i und b bestimmen (Gleichungen 2.2.16 und 2.2.17).

$$f(X) = b + \sum_{i=1}^{m} w_i x_i \tag{2.2.15}$$

$$\frac{\partial E}{\partial w_i} = \frac{\partial f(X)}{\partial w_i} \cdot \frac{\partial E}{\partial f(X)} = x_i \cdot \delta \tag{2.2.16}$$

$$\frac{\partial E}{\partial b} = \frac{\partial f(X)}{\partial b} \cdot \frac{\partial E}{\partial f(X)} = 1 \cdot \delta \tag{2.2.17}$$

Der von der dieser Einheit an die Einheit der Eingabe x_i zurückpropagierte Wert δ_i' wird folgendermaßen bestimmt:

$$\delta_i' = \frac{\partial E}{\partial x_i} = \frac{\partial f(X)}{\partial x_i} \cdot \frac{\partial E}{\partial f(X)} = w_i \cdot \delta \tag{2.2.18}$$

Wenn die Backpropagation für eine nicht parametrisierte Einheit, wie zum Beispiel die ReLU-Funktion (Gleichung 2.2.6) ausgeführt wird, muss lediglich der Fehler in Bezug auf die Eingabe bestimmt werden. Sei Φ die ReLU-Funktion, dann ist Gleichung 2.2.19 ihre Ableitung. Gleichung 2.2.20 beschreibt, wie aus dem durch die Backpropagation erhaltenen Fehler δ, der von der Einheit zurückpropagierte Fehler δ' berechnet wird.

$$\Phi'(x) = I(x \geq 0) \tag{2.2.19}$$

$$\delta' = \frac{\partial E}{\partial x} = \frac{\partial \Phi(x)}{\partial x} \cdot \frac{\partial E}{\partial \Phi(x)} = I(x \geq 0) \cdot \delta \tag{2.2.20}$$

2.2.6 *Optimierung des Trainings*

Für das Finden eines lokalen Minimums erfordert das Training künstlicher neuronaler Netze mittels Gradientenabstieg die Berechnung des Gradienten für den gesamten Trainingsdatensatz (echter Gradient) [Rud17]. Da diese Berechnung für jede Iteration erneut ausgeführt werden muss, wird dadurch in der Praxis oft zu viel Rechenzeit in Anspruch genommen. Zudem muss der gesamte Trainingsdatensatz im Arbeitsspeicher vorgehalten werden, was unter Umständen nicht möglich ist.

Im Folgenden bezeichnet der Vektor θ_t die Parameter des Netzes im Schritt t. Der Trainingsdatensatz besteht aus Paaren (X, Y). Die Komposition aus Netz- und Zielfunktion $E(X, Y)$ berechnet den Fehler des Netzes für die Eingabe X und die erwartete Ausgabe Y.

Eine häufig genutzte Variation des Gradientenabstiegs ist der stochastische Gradientenabstieg (SGD, *engl. Stochastic Gradient Descend*) mit Mini-Batches. Hierzu wird der Gradient in jeder Iteration für eine zufällig ausgewählte Teilmenge (Mini-Batch) des Trainingsdatensatzes berechnet. Üblicherweise findet das Training in Epochen statt, in jeder Epoche werden die Mini-Batches zufällig zugeteilt und der Reihe nach für den SGD benutzt.

$$\Delta\theta = \alpha \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta E(X_i, Y_i) \tag{2.2.21}$$

In Gleichung 2.2.21 bezeichnet θ den Parametervektor des Netzes, $(X_1, Y_1), ..., (X_n, Y_n)$ sind die Trainingsbeispiele des Mini-Batches. Mit der Lernrate α wird die Parameteranpassung $\Delta\theta$ in jeder Iteration bestimmt. Der wahre Fehlergradient wird hier mit dem durchschnittlichen Fehlergradienten des Mini-Batches approximiert. Durch die Approximation wird zudem noch ermöglicht, über lokale Minima hinaus zu optimieren.

Fortan wird der durchschnittliche Fehlergradient des Mini-Batches in der Iteration t mit $\tilde{G}_t$ gekennzeichnet. Die Parameter θ_t sind die in Iteration t neu berechneten Parameter.

$$\tilde{G}_t = \frac{1}{n} \sum_{i=1}^{n} \nabla_{\theta_{t-1}} E(X_i, Y_i) \tag{2.2.22}$$

Eine in der Literatur häufig verwendete Analogie für den Gradientenabstieg ist ein Ball, welcher Schritt für Schritt die Fehlerfunktion hinab rollt. In Abbildung 2.2.8 ist der Netzfehler E für ein Trainingsbeispiel in Abhängigkeit der beiden Parameter θ_1 und θ_2 eingezeichnet. Die roten Pfeile symbolisieren den Pfad, welchen der Ball während des Gradientenabstiegs bei verschiedenen Startparametern hinabrollt. Es ist zu sehen, dass während des Gradientenabstiegs θ_1 und θ_2 zu lokalen Minima konvergieren.

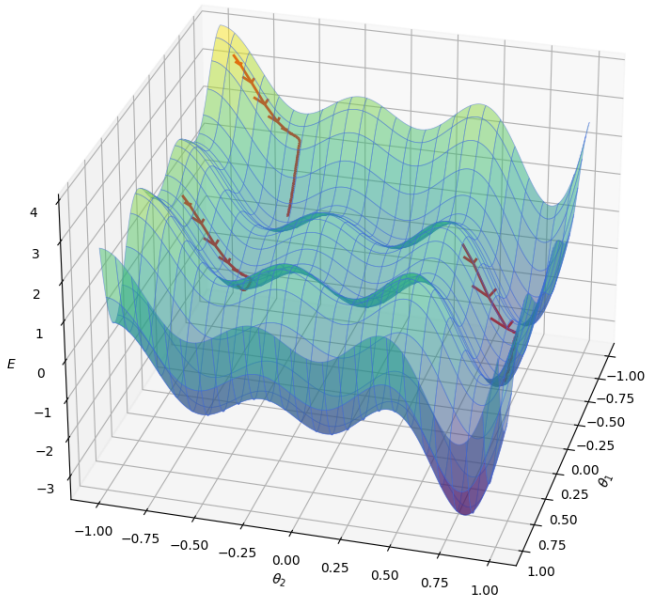

Abbildung 2.2.8: Gradientenabstieg auf der Funktion E mit eingezeichneten Schritten (rot).

Um lokale Minima zu übergehen, kann eine Variation des SGD mit Momentum genutzt werden. In der Ball-Analogie würde der Ball eine Masse haben, welche ihn durch angesammeltes Momentum, bei Umkehr des Gradienten, schwerer zu stoppen macht. Die Gleichungen in 2.2.23 beschreiben den SGD mit Momentum. V_t ist hier der Momentum-Vektor in Iteration t, initial gilt $V_0 = 0$. Mit der Konstante γ wird ein Teil

des Momentums aus dem vorherigen Zeitschritt mit in die Berechnung aufgenommen. Ein gängiger Wert für γ ist 0,9.

$$\begin{aligned} V_t &= \gamma V_{t-1} + \alpha \tilde{G}_t \\ \theta_t &= \theta_{t-1} - V_t \end{aligned} \tag{2.2.23}$$

In der Praxis benötigen die Hyperparameter, wie die Lernrate α oder die Momentum-Konstante γ, manuelle Anpassungen und zeitliche Varianz um ein möglichst gutes Konvergenzverhalten zu erzielen. Die adaptive Momentumsschätzung (Adam, *engl. Adaptive Moment Estimation*) [KB17] berechnet für jeden Parameter adaptive Lernraten. Dafür werden Schätzungen des wahren Gradienten aufgrund des ersten und zweiten statistischen Moments (Mittelwert und unzentrierete Varianz) der vorherigen Mini-Batch-Gradienten berechnet.

$$\begin{aligned} M_t &= \beta_1 M_{t-1} + (1-\beta_1)\tilde{G}_t \\ V_t &= \beta_2 V_{t-1} + (1-\beta_2)\tilde{G}_t^2 \end{aligned} \tag{2.2.24}$$

M_t ist die Schätzung des ersten statistischen Moments mittels exponentiellem Verfall, analog ist V_t die Schätzung des zweiten statistischen Moments. Die Vektoren M_0 und V_0 werden mit 0 initialisiert. Um die somit entstehenden Tendenzen zu 0 der Schätzungen M_t und V_t abzuschwächen wurden Korrekturterme eingeführt:

$$\hat{M}_t = \frac{M_t^t}{1-\beta_1^t} \qquad \hat{V}_t = \frac{V_t^t}{1-\beta_2^t} \tag{2.2.25}$$

Schlussendlich ergibt sich mit einer Lernrate α die folgende Regel für die Parameteranpassungen:

$$\theta_t = \theta_{t-1} - \alpha \hat{M}_t \frac{1}{\sqrt{\hat{V}_t} + \varepsilon} \tag{2.2.26}$$

Kingma und Ba schlagen als Standardparameter $\beta_1 = 0{,}9, \beta_2 = 0{,}999$ und $\alpha = 0{,}001$ vor, $\varepsilon = 10^8$ stellt die numerische Stabilität sicher. Diese Auswahl der Hyperparameter stellt sich als gute Wahl für die meisten Lernszenarien heraus.

2.2.7 *Overfitting und Regularisierung*

Das Ziel des maschinellen Lernens ist es, ein möglichst gutes Ergebnis für bisher ungesehene Daten zu berechnen. Mit steigender Leistungsfähigkeit der Modelle steigt auch die

Möglichkeit, dass Eigenschaften aus den Trainingsdaten gelernt werden, welche nicht repräsentativ für die Aufgabe des Modells sind, wie etwa Rauschen. Es kann so passieren, dass ein Modell die Trainingsdaten nahezu auswendig lernt und umso schlechtere Ergebnisse auf ungesehenen Daten erzielt. Dieses Phänomen ist als Overfitting (*dt. Überanpassung*) bekannt [Yin19].

Overfitting künstlicher neuronaler Netze kann mehrere zusammenhängende Ursachen haben. Grundsätzlich ist dabei zwischen Problemen mit den Trainingsdaten und dem Modell selbst zu unterscheiden. Die Trainingsdaten können beispielweise mit Rauschen behaftet sein oder unzureichend den Sachzusammenhang darstellen. Bei der Auswahl von Modellen im statistischen maschinellen Lernen ist grundsätzlich auf eine Balance zwischen Genauigkeit und Konsistenz zu achten. Bei zu großer Komplexität des Modells kann die Genauigkeit auf einem Datensatz steigen, dafür aber die Genauigkeit auf anderen Datensätzen drastisch sinken.

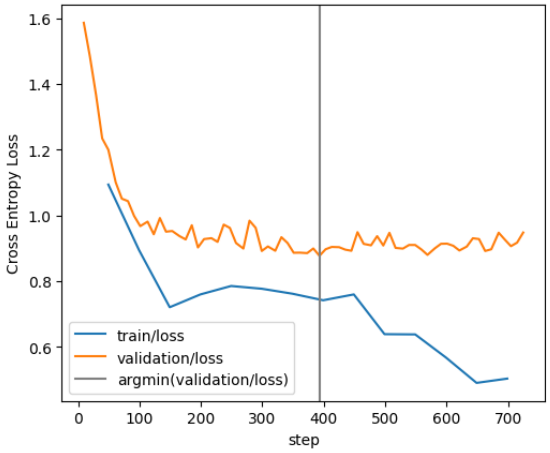

Abbildung 2.2.9: Trainingsverlauf mit Overfitting

In Abbildung 2.2.9 ist ein beispielhafter Trainingsverlauf mit Overfitting ab etwa Schritt 400 zu sehen. Zunächst verringern sich der Trainings- und Validierungsfehler simultan, danach verbessert sich nur noch der Trainingsfehler und der Validierungsfehler steigt.

Eine effektive Methode zur Erkennung und frühzeitigen Vermeidung von Overfitting ist es, den Trainingsdatensatz zu partitionieren. Üblicherweise wählt man einen Test-, Validierungs- und Trainingsdatensatz. Das Modell wird wie gehabt auf dem Trainingsdatensatz trainiert und regelmäßig auf dem Validierungsdatensatz getestet. So kann man während des Trainings erkennen, ob das Netz lernt unbekannte Daten erfolgreich

zu verarbeiten. Nach dem Training kann die Instanz des Modells, welche das beste Ergebnis auf dem Validierungsdatensatz erzielt als finales Modell auf dem Testdatensatz ausgewertet werden.

Durch Regularisierung des Modells wird versucht Overfitting zu verzögern, indem das Modell künstlich geschwächt wird. Ein Ansatz ist dabei das Verwenden von Dropout-Neuronen. Dabei hat während des Trainings jedes Neuron eine Wahrscheinlichkeit von z.B. 50% deaktiviert zu werden (durch Multiplikation mit 0). So werden bei jedem Trainingsbeispiel andere Kombinationen von Neuronen trainiert. Nach dem Training bleiben alle Neuronen aktiviert.

2.3 FALTUNGSNETZE

Eines der am meisten benutzen Arten von künstlichen neuronalen Netzen in der Mustererkennung sind Faltungsnetze (CNNs, *engl. Convolutional Neural Networks*) [AMA17]. Sie werden beispielsweise für die Bild- oder Spracherkennung verwendet.

CNNs basieren auf der diskreten Faltung (*engl. Convolution*) von Matrizen. Im Gegensatz zu den bereits betrachteten Arten von KNNs fokussieren sich CNNs auf lokale Muster in den Eingaben. Graustufenbilder werden beispielsweise als eine zweidimensionale Eingabe aus Bildpunkten aufgefasst. Die Faltungsschichten des CNNs extrahieren dann schichtweise lokale Eigenschaften des Bildes, von einfachen Strukturen wie Kanten bis hin zu komplexen Objekten.

In Abschnitt 2.3.1 wird zunächst das Konzept der diskreten Faltung erklärt. Danach werden die elementaren Einheiten der Faltungsschichten in Abschnitt 2.3.2 und 2.3.3 vorgestellt. Schlussendlich geht es in Abschnitt 2.3.4 um typische Strukturen von CNNs.

2.3.1 *Faltung*

Die diskrete Faltung ist ein häufig verwendetes Werkzeug in der digitalen Signalverarbeitung. Es wird dazu ein diskretes Eingabesignal mit einem sogenannten Filterkern (*engl. Kernel*) gefaltet. Das Ziel dabei ist es, gewisse Eigenschaften des Signals zu verändern, wie etwa das Rauschen zu unterdrücken oder die Kanten hervorzuheben [Wup89]. Prinzipiell wird dabei der Filterkern elementweise über die Eingabe bewegt. Aus jedem Schnittbereich wird eine gewichtete Summe gebildet, welche einem Element der Ausgabe entspricht. Formal lässt sich die diskrete Faltung $(*) : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}^{m-n+1}$

für einen eindimensionalen Eingabevektor $X = [x_1, ..., x_m]$ und einen eindimensionalen Filterkern $K = [k_1, ..., k_n]$ wie folgt definieren:

$$(X * K)_{i-n+1} = \sum_{j=1}^{n} k_j \cdot x_{i-j+1} \qquad \forall i \in \{n, ..., m\} \tag{2.3.1}$$

Es ist zu beachten, dass sich die Ausgabe durch die Filtergröße n um $n-1$ verkleinert. Eine solche Verkleinerung ist nicht immer erwünscht, häufig wird das Eingangssignal vorher um $n-1$ Werte erweitert. Beispielsweise beidseitig mit 0 aufgefüllt (padding).

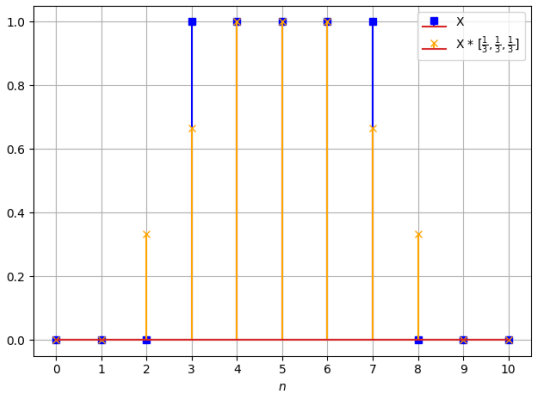

Abbildung 2.3.1: Diskrete Faltung eines Eingabevektors X mit einem Glättungskern.

In Abbildung 2.3.1 ist die Faltung eines Rechtecksignals X (blau) mit einem Glättungskern $K = \left[\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right]$ zu sehen. Das Ergebnis der Faltung ist orange dargestellt. Es ist zu erkennen, dass die Kanten des Eingangssignals durch den Filterkern der Größe 3 geglättet wurden.

Die diskrete zweidimensionale Faltung mit Matrizen $(*) : \mathbb{R}^{m \times n} \times \mathbb{R}^{o \times p} \to \mathbb{R}^{m-o+1 \times n-p+1}$ lässt sich analog definieren. Die Eingabematrix $\overline{\mathbf{X}}$ hat die Größe $m \times n$, der Filterkern $\overline{\mathbf{K}}$ die Größe $o \times p$. Für alle $(i, j) \in \{o, ..., m\} \times \{p, ..., n\}$ gilt:

$$(\overline{\mathbf{X}} * \overline{\mathbf{K}})_{i-o+1, j-p+1} = \sum_{x=1}^{o} \sum_{y=1}^{p} k_{xy} \cdot x_{i-x+1, j-y+1} \tag{2.3.2}$$

Auch hier ist die Reduktion der Größe zu beachten. In Abbildung 2.3.2 sind die ersten vier Schritte einer zweidimensionalen Faltung konzeptuell dargestellt. Die Eingabe $\overline{\mathbf{X}} \in \mathbb{R}^{5 \times 5}$ (blau) wird dort mit einem Filterkern $\overline{\mathbf{K}} \in \mathbb{R}^{3 \times 3}$ gefaltet. Zum Ausgleich der Größenveränderung wurde die Eingabe symmetrisch mit einem Padding versehen. Die Ausgabe (cyan) ist entsprechend eine 5×5 Matrix.

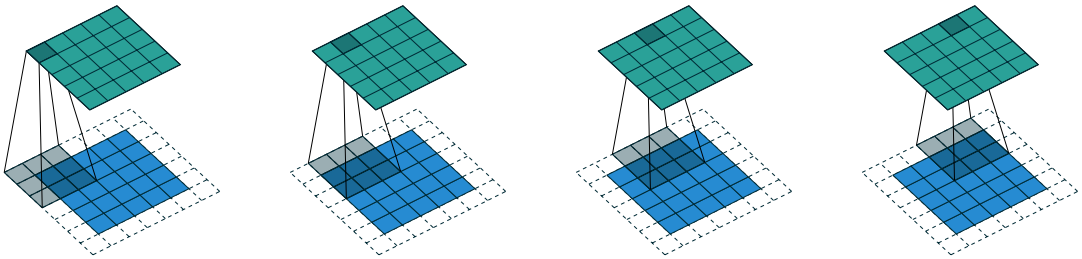

Abbildung 2.3.2: Vier Schritte einer zweidimensionalen Faltung mit Padding der Eingabe (blau) und dem Filterkern (grau). Das Ergebnis ist in Cyan dargestellt. Entnommen aus [DV18].

Neben dem Padding kann noch ein sogenannter „Stride“ (*dt. Schrittweite*) als Variation der Faltung eingeführt werden. Bisher betrug die Schrittweite implizit den Wert 1. In Abbildung 2.3.3 sind die ersten vier Schritte einer zweidimensionalen Faltung der Eingabe $\overline{\mathbf{X}} \in \mathbb{R}^{5\times5}$ und dem Filterkern $\overline{\mathbf{K}} \in \mathbb{R}^{3\times3}$ bei einer Schrittweite von 2 dargestellt. Die Eingabe wurde auch hier symmetrisch mit einem Padding versehen. Die resultierende Ausgabe ist eine 3×3 Matrix.

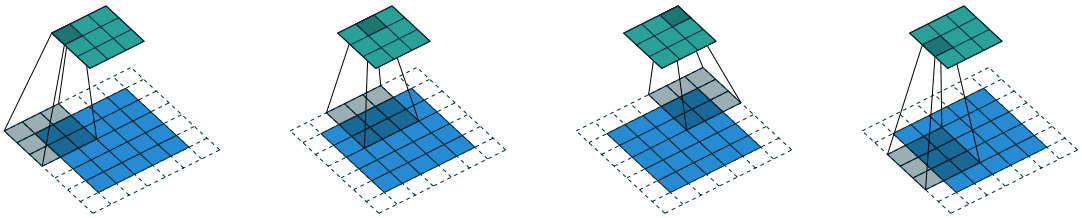

Abbildung 2.3.3: Vier Schritte einer zweidimensionalen Faltung mit Padding und Stride ($s = 2$) der Eingabe (blau) und dem Filterkern (grau). Das Ergebnis ist in Cyan dargestellt. Entnommen aus [DV18].

Für die allgemeinste Form der eindimensionalen Faltung lässt sich die Ausgabegröße o mit einem Stride s, einem beidseitigem Padding p und einer Filterkerngröße k für eine Eingabe der Größe i wie folgt berechnen:

$$o = \left\lfloor \frac{i + 2p - k}{s} \right\rfloor + 1 \tag{2.3.3}$$

Die Rundung kompensiert den Fall, dass durch den Stride der Filterkern nicht genau das Ende der Eingabe trifft [DV18]. Für den Fall mehrerer Dimensionen kann die Berechnung für jede Dimension separat durchgeführt werden.

2.3.2 *Faltungsschichten*

Angenommen es wird ein mehrschichtiges Perzeptron für die Erkennung von Bildern der Größe $(64 \times 64 \times 3)$ genutzt, wobei die letzte Dimension die Anzahl der Farbkanäle bezeichnet. Das MLP hat also $64 \cdot 64 \cdot 3 = 12288$ Eingabeneuronen, was auch bedeutet, dass pro Neuron in der nachfolgenden versteckten Schicht 12288 Parameter benötigt werden. Durch die Nutzung von Faltungsschichten lässt sich die Anzahl der Parameter für Bildeingaben drastisch verringern, indem durch parametrisierte Faltungskerne nur benachbarte Eingabeneuronen auf die nächste Schicht übertragen werden.

Eine Faltungsschicht besteht dabei aus einem oder mehreren parametrisierten Filterkernen festgelegter Größe [Gu+18]. Beispielsweise 10 Kerne der Größe (5×5). Die Anzahl der Parameter beträgt dabei $10 \cdot 5 \cdot 5 = 250$. Jeder Filterkern erzeugt aus allen Kanälen der Eingabe eine sogenannte „Featuremap“ (*dt. Merkmalskarte*), indem für jeden Filterkern die Faltung auf allen Eingabekanälen durchgeführt wird. Die Ergebnisse werden dann elementweise summiert und optional mit einem skalaren Bias addiert. Für jeden Eingabekanal $c_{in} \in \{1, ..., C_{in}\}$ und jeden Ausgabekanal $c_{out} \in \{1, ..., C_{out}\}$ gilt folgender Zusammenhang für Faltungsschichten:

$$\overline{\mathbf{Y}}_{c_{out}} = \text{bias}_{c_{out}} + \sum_{c_{in}=1}^{C_{in}} \overline{\mathbf{X}}_{c_{in}} * \overline{\mathbf{K}}_{c_{out}} \tag{2.3.4}$$

Auch hier können Variationen der Faltungsoperation mit einem Stride oder einem Padding verwendet werden.

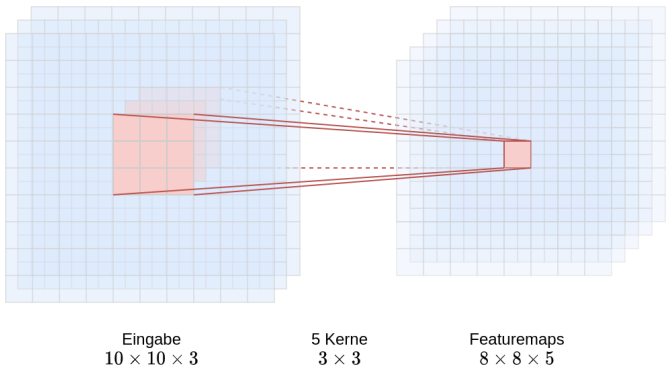

Abbildung 2.3.4: Übertragung einer Faltungsschicht für eine $10 \times 10 \times 3$-Eingabe ohne Padding mittels fünf 3×3-Filterkernen.

In Abbildung 2.3.4 ist die Anwendung einer Faltungsschicht aus fünf 3×3 Filterkernen für eine Eingabe der Größe $10 \times 10 \times 3$ ohne Padding zu sehen. Die Ausgaben sind entsprechend fünf Featuremaps der Größe 8×8.

2.3.3 *Pooling-Schichten*

Um die Anzahl der Parameter und die Komplexität der Berechnung weiter zu verringern, können neben Faltungsschichten sogenannte Pooling-Schichten verwendet werden. Diese benutzen ebenfalls ein Fenster, welches über die Eingabe geschoben wird. Anstelle einer gewichteten Summe, wie bei der Faltung, wird hier eine Pooling-Operation ausgeführt. Üblicherweise wird für das Pooling der maximale Wert im Fenster gewählt [ON15].

Durch den hohen Informationsverlust des Max-Pooling wird in der Regel nur eine Fenstergröße von 2 bei einem Stride von 2 gewählt. Eine Fenstergröße > 3 führt in den meisten Fällen zu großem Leistungsverlust des Modells. Neben dem Max-Pooling gibt es noch weitere Pooling-Operationen wie L_p-Pooling, Mixed-Pooling oder Spectral-Pooling. Für eine detailliertere Übersicht sei auf [Gu+18, Abschnitt 3.2] verwiesen.

2.3.4 *Anwendung von Faltungsnetzen*

Die Faltungs- und Pooling-Schichten alleine sind nur dafür geeignet, lokale Merkmale aus großen Eingaben auf Featuremaps zu reduzieren. Durch die Aneinanderreihung vieler Faltungs- und Pooling-Schichten, getrennt durch elementweise nicht lineare Aktivierungsfunktionen, lassen sich komplexere Merkmale über größere Bereiche der Eingabe extrahieren. Beispielsweise ist das beste Modell im ILSVRC 2015 Wettbewerb für Bilderkennung das ResNet [He+16] mit einer Tiefe von bis zu 152 Schichten. Schlussendlich werden die Featuremaps in einem CNN durch voll verbundene Schichten klassifiziert.

In Abbildung 2.3.5 ist die Struktur des LeNet-5 [Lec+98] dargestellt. Das LeNet-5 ist eines historisches CNN für die Dokumentenerkennung. Zunächst werden Graustufenbilder mit sechs Filterkernen auf entsprechend sechs Featuremaps reduziert. Mit einer Durchschnitts-Pooling-Operation (*Subsampling* in der Abbildung) werden diese um den Faktor 4 verkleinert. Die Featuremaps werden danach erneut mit 16 Filterkernen und einer Durchschnitts-Pooling-Operation verarbeitet. Mit einem dreischichtigen MLP werden die Featuremaps schlussendlich klassifiziert.

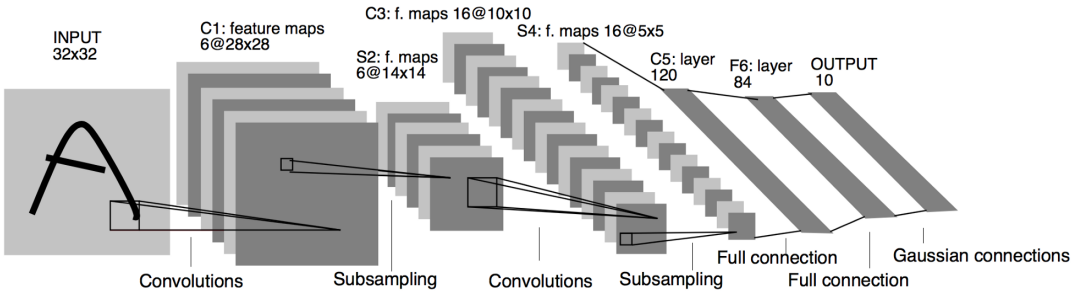

Abbildung 2.3.5: Struktur des LeNet-5 für Dokumentenerkennung, entnommen aus [Lec+98]

2.4 QUANTISIERUNG

In vielen Bereichen sind Deep-Learning-Ansätze erfolgversprechend, jedoch ist die steigende Leistung der Modelle mit erhöhter Ressourcennutzung der verfügbaren Systeme verbunden [Cho+19]. Es gibt einige Aufgabenbereiche, in denen naturgemäß nur stark beschränkte Ressourcen zur Verfügung stehen, wie etwa in der mobilen Anwendung. Es gibt verschiedene Ansätze, um aktuelle Modelle ressourcensparender zum Einsatz zu bringen.

Durch „Pruning“ lässt sich die Größe eines KNNs und somit auch der Rechenaufwand reduzieren. Es werden dafür Neuronen, welche unsensibel auf Eingaben reagieren, aus dem Netz entfernt, um insignifikante Berechnungen zu vermeiden [Gho+21]. Ein weiterer Ansatz ist das Destillieren eines KNNs. Zunächst wird ein großes KNN trainiert, ein kleineres Netz soll dann lernen die rohen Ausgaben (z.B. alle Klassenwahrscheinlichkeiten) des großen Netzes zu imitieren. Die in diesem Unterkapitel betrachtete Methode ist die Quantisierung der Gewichte (und Aktivierungen) in einem KNN. Die Auflösung der Gewichte wird so beispielsweise von 32-Bit-Gleitkommazahlen auf ganze Zahlen mit 8-Bit reduziert. Dieser Ansatz hat sich bereits in verschiedenen Szenarien als effektiv herausgestellt [PPA18].

2.4.1 *Quantisierungsfunktionen*

In dieser Arbeit wird die Menge aller Gleitkommazahlen (nach IEEE-754 [Iee]) mit einer Auflösung von 16, 32 und 64-Bit entsprechend mit $\mathbb{R}_{16}$, $\mathbb{R}_{32}$ und $\mathbb{R}_{64}$ gekennzeichnet. Die Menge der ganzen Zahlen mit n-Bit wird mit $\mathbb{Z}_n = \{-2^{n-1}, ..., 2^{n-1} - 1\} \subset \mathbb{Z}$ und die Menge der positiven ganzen Zahlen mit $\mathbb{N}_n = \{0, ..., 2^n - 1\} \subset \mathbb{N}$ bezeichnet. Die

Funktion $\lfloor \cdot \rceil : \mathbb{R} \to \mathbb{Z}$ bezeichnet die Rundungsoperation zur nächsten ganzen Zahl mit $\lfloor x \rceil = \lfloor x + 0{,}5 \rfloor$.

Grundsätzlich wird bei der Quantisierung ein Intervall der reellen Zahlen auf einen diskreten endlichen Wertebereich projiziert. In Abbildung 2.4.1 ist eine gleichmäßige symmetrische Abbildung von $[a, b] \subseteq \mathbb{R}_{32}$ auf einen reduzierten symmetrischen Bereich von $\mathbb{Z}_8$ dargestellt. Alle Werte außerhalb von $[a, b]$ werden vorher entsprechend auf a oder b abgebildet, daher wird $[a, b]$ auch als „Clipping-Range“ bezeichnet. Eine solche Quantisierungsfunktion $Q : \mathbb{R} \to \mathbb{Z}_n$ hat die folgende Form:

$$Q(x) = \left\lfloor \frac{x}{s} \right\rceil \tag{2.4.1}$$

Der Wert $s \in \mathbb{R}$ bestimmt dabei die Skalierung des Wertebereichs. Es wird hier $[a, b]$ gleichmäßig auf $\mathbb{Z}_n$ aufgeteilt:

$$s = \frac{b - a}{2^n - 1} \tag{2.4.2}$$

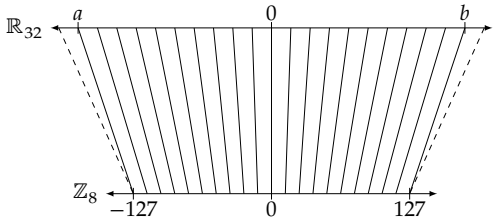

Abbildung 2.4.1: Symmetrische gleichmäßige Quantisierungsfunktion.

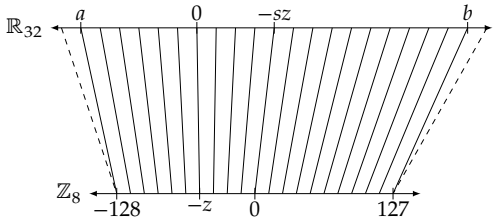

Abbildung 2.4.2: Asymmetrische gleichmäßige Quantisierungsfunktion.

Die gleichmäßige symmetrische Quantisierungsfunktion in Gleichung 2.4.1 lässt sich zudem noch mit einer Verschiebung des Nullpunktes um $z \in \mathbb{Z}$ erweitern:

$$Q(x) = \left\lfloor \frac{x}{s} \right\rceil - z \tag{2.4.3}$$

Passend dazu ist in Abbildung 2.4.2 eine gleichmäßige asymmetrische Quantisierungsfunktion dargestellt. Es ist zu erkennen, dass der Nullpunkt aus $\mathbb{R}_{32}$ auf $-z$ in $\mathbb{Z}_8$ abgebildet wird. Umgekehrt wird der Nullpunkt aus $\mathbb{Z}_8$ auf $-sz$ in $\mathbb{R}_{32}$ abgebildet. Die Dequantisierung lässt sich entsprechend über den Zusammenhang in Gleichung 2.4.4 durchführen. Wegen der Rundung in Gleichung 2.4.3 lässt sich die der Eingabewert x nicht exakt rekonstruieren.

$$\tilde{x} = s \cdot (Q(x) + z) \tag{2.4.4}$$

Quantisierungsfunktion müssen nicht immer eine gleichmäßig verteilte (affine) Abbildung sein. Wenn die zu quantisierenden Daten bekannte Verteilungen haben, können Abbildungen gewählt werden, welche häufig vorkommende Wertebereiche auf einen verhältnismäßig größeren, quantisierten Wertebereich abbilden. Die allgemeinste Form der Quantisierungsfunktion kann als stückweise konstante Funktion definiert werden.

$$Q(x) = q_i, \quad \text{wenn } x \in [\Delta_i, \Delta_{i+1}) \tag{2.4.5}$$

Jeder Wert $x \in \mathbb{R}$ wird genau dann auf den Wert $q_i \in \mathbb{Z}$ abgebildet, wenn $\Delta_i \leq x < \Delta_{i+1}$ gilt. Ist beispielsweise bekannt, dass die Daten normalverteilt auftreten, können die Intervalle in der Nähe des Erwartungswertes kleiner gewählt werden, um den Informationsverlust zu verringern. In Abbildung 2.4.3 wurden die Intervalle der Quantisierungsfunktion passend zu einer normalverteilten Eingabe mit dem Erwartungswert 0 gewählt.

Die Zentrale Problemstellung der Quantisierung von KNNs ist die Minimierung des Quantisierungsfehlers. Grundsätzlich kann der Quantisierungsfehler als Differenz zwischen der Eingabe x und der rekonstruieren Eingabe $\tilde{x}$ definiert werden:

$$Q_{\text{error}}(x) \overset{\text{def.}}{=} \tilde{x} - x \tag{2.4.6}$$

Für Eingabevektoren X kann der Quantisierungsfehler als Fehlervektor dargestellt werden [Gho+21]. Mit einer passenden p-Norm lässt sich das folgende Minimierungsproblem definieren:

$$\min_{Q} \left(\|Q_{\text{error}}(X)\|_p \right) \tag{2.4.7}$$

Wenn symmetrische gleichmäßige Quantisierungsfunktionen betrachtet werden, liefert die Lösung des Minimierungsproblems beispielsweise die optimale Clipping-Range $[a, b]$.

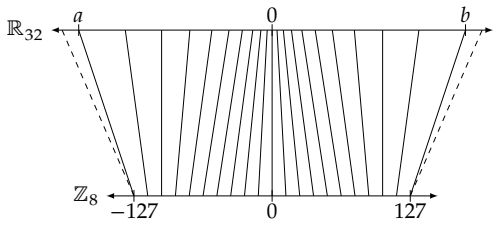

Abbildung 2.4.3: Symmetrische ungleichmäßige Quantisierungsfunktion mit höherer Präzision nahe dem Erwartungswert 0.

Das Bestimmen passender Parameter für eine Quantisierungsfunktion wird als Kalibrierung bezeichnet. Für die Clipping-Range $[a, b]$ einer gleichmäßigen Quantisierungsfunktion werden als Annäherung üblicherweise das Minimum und Maximum der zu quantisierenden Daten gewählt. Die konkrete Anwendung von Quantisierungsfunktion auf Parameter und Aktivierungen in einem KNN wird in Abschnitt 4.2 ausgeführt.

3 VERWANDTE ARBEITEN

Für die Quantisierung künstlicher neuronaler Netze im Bereich der Aktivitätserkennung gibt es bereits verschiedene Ansätze. Dieses Kapitel soll einen Überblick über gängige KNNs für die HAR bieten. Als Erstes werden die bisherigen Leistungen des CNN-IMUs zusammengefasst, welches in dieser Arbeit auf die Quantisierbarkeit untersucht wird. Anschließend werden noch drei weitere Modelle, welche erfolgreich für die HAR quantisiert wurden, vorgestellt.

3.1 CNN-IMU

Das von Moya Rueda et al. [MR+18] vorgestellte CNN-IMU ist ein CNN für die IMU-basierte Aktivitätserkennung. Die Messwerte der IMUs werden in einer $D \times T$ Matrix abgelegt, wobei T die Größe eines zeitlichen Fensters und D die Anzahl der gemessenen Variablen pro IMU angibt. Durch mehrere Faltungsschichten werden Featuremaps für jede IMU-Matrix extrahiert. Schlussendlich werden die Featuremaps durch mehrere lineare Schichten klassifiziert. Die genauere Architektur wird in Abschnitt 4.1 beschrieben. Moya Rueda et al. konnten mit verschiedenen Konfigurationen des CNN-IMUs die state-of-the-art Klassifikationsleistung für zwei etablierte Datensätze übertreffen. In dieser Arbeit werden verschiedene Techniken für die Quantisierung des CNN-IMUs untersucht.

3.2 DFTERNET

Yang et al. benutzen ein spezielles CNN zur IMU-basierten Aktivitätserkennung. Das DFTerNet (Dynamic-Fusion-Ternary(2-bit)-Convolutional-Network) [Yan+18] nutzt ähnlich wie das CNN-IMU, für jede IMU separate Featuremaps. Die Parameter der Faltungskerne werden allerdings von allen IMUs geteilt. Bevor die Featuremaps durch ein MLP klassifiziert werden, wird eine dynamische Fusion angewandt, um sie zusammen zu fassen. In Abbildung 3.2.1 ist die Architektur des DFTerNet dargestellt.

Yang et al. organisieren die Messwerte jeder IMU in einer $T \times D$ Matrix mit der Anzahl der gemessenen Variablen D und der Fenstergröße T. Die Faltungsschichten C1, C2 und C3 bestehen aus 50 (11×1), 40 (10×1) und 30 (6×1) Filterkernen, welche entlang der

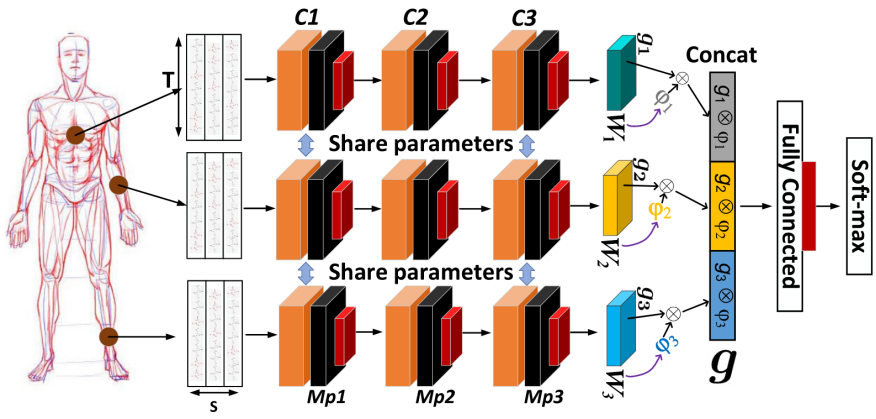

Abbildung 3.2.1: DFTerNet (Dynamic-Fusion) Architektur, entnommen aus [Yan+18].

zeitlichen Dimension bei einem Stride von 1 falten. Nach den Faltungsschichten werden die Max-Pooling-Schichten Mp1, Mp2 und Mp3 mit den Fenstergrößen (2×1), (3×1) und (1×1) verwendet. Alle Aktivierungen werden mittels Batch-Normalization [IS15] normalisiert.

$$BN(\overline{\mathbf{Y}}) = \gamma \frac{\overline{\mathbf{Y}} - \mu}{\sigma} + \beta \tag{3.2.1}$$

Gleichung 3.2.1 berechnet die Batch-Normalization für die Aktivierungen $\overline{\mathbf{Y}}$ mit dem Batch-Mittelwert μ und der Batch-Standartabweichung σ. Die Parameter γ und β sind lernbar.

Bevor das MLP die Featuremaps klassifiziert, werden sie mittels der von den Autoren neu vorgestellten Dynamic-Fusion zu einem Featurevektor zusammengefasst. Jede der Featuremaps g_1, g_2 und g_3 wird mit einer zugehörigen Maske $\varphi_1, \varphi_2, \varphi_3 \in \{0,1\}^*$ derselben Größe punktweise multipliziert, ähnlich wie beim Dropout aus Abschnitt 2.2.7 ($*$ steht hier stellvertretend für die zugehörigen Dimensionsgrößen). Die Masken werden allerdings anhand der lernbaren Parameter $W_1, W_2, W_3 \in [0,1]^*$ über die Bernoulli-Verteilung bestimmt. Jeder Wert in W_i entspricht also der Wahrscheinlichkeit, dass der entsprechende Wert in φ_i Eins annimmt.

Yang et al. schlagen folgende Quantisierungsfunktion mit $k = 2$ (2-Bit) vor:

$$Q_k(x, \varepsilon) = Clip\left(\phi(k) \cdot \left\lfloor \frac{x \cdot \varepsilon}{\phi(k)} \right\rfloor, -1 + \phi(k), 1 - \phi(k)\right) \quad , \quad \phi(k) = 2^{1-k} \tag{3.2.2}$$

So werden reele Parameter auf ternäre Gewichte ($w \in \{-0{,}5; 0; 0{,}5\}$) abgebildet. Q_k verhält sich ähnlich, wie die symmetrische Quantisierungsfunktion aus Gleichung

2.4.1. Das Clipping wird lediglich nach der Quantisierung angewandt. Das ε_p jeder Quantisierungsfunktionen für die Gewichte W wird mit dem Hyperparameter ζ wie folgt kalibriert:

$$\varepsilon_p = \left(\frac{\zeta}{n} \sum_{i=1}^{n} |w_i| \right)^{-1} \tag{3.2.3}$$

Zunächst werden die Parameter in reeller Form trainiert. Anschließend wird der Quantisierungsfehler mittels Quantization-Aware-Training (Abschnitt 4.2.5) minimiert. Die Aktivierungen werden statisch quantisiert. In der Schicht mit den Gewichten W wird das ε_a der Quantisierungsfunktion für die Aktivierungen jede Epoche durch den folgenden Zusammenhang kalibriert:

$$\varepsilon_a = \min\left(1, \frac{1}{2^{round(\tau)}}\right) \quad , \quad \tau = \frac{1}{m} \sum_{i=1}^{m} \frac{|w_i|}{\max(w_1, ..., w_m)} \tag{3.2.4}$$

Durch die Anwendung von Bitoperatoren wurde die Inferenzzeit des DFTerNet im Vergleich zu den reellen Parametern um etwa den Faktor 9 verbessert. Der Speicherverbrauch war dabei etwa 11-Mal geringer. Die Klassifikationsleistung des DFTerNet hat sich dabei um bis zu 1% verschlechtert. Sowohl für den Pamap2-, als auch den Opportunity-Locomotion-Datensatz zeigt das DFTerNet eine Leistungsverbesserung um etwa 5% gegenüber den Baseline-Werten.

3.3 BINARIZED BLSTM RNN

Edel und Köppe nutzen kein CNN für die Aktivitätserkennung, ihr B-BLSTM-RNN (Binarized-Bidirectional-Long-Short-Term-Memory-RNN) [EK16] extrahiert zeitliche Informationen mittels LSTM-Zellen. Diese verwalten einen internen Zustand, welcher einen Kontext über längere Zeiträume bereitstellen kann [HS97]. Später wurden sie noch mit einer „Vergessensfunktion“ erweitert [GSC99]. LSTM-Zellen basieren auf einfachen linearen RNN-Schichten. Durch einen versteckten Zeilenvektor H kann die Schicht auf vorherige Zeitschritte zugreifen. Der Zusammenhang kann folgendermaßen definiert werden:

$$\mathrm{H}_t = \Phi\left(\mathrm{X}_t \times \overline{\mathbf{W}}_{xh} + \mathrm{H}_{t-1} \times \overline{\mathbf{W}}_{hh} + \mathrm{B}_h\right) \tag{3.3.1}$$

$$\mathrm{Y}_t = \mathrm{H}_t \times \overline{\mathbf{W}}_{hy} + \mathrm{B}_y \tag{3.3.2}$$

Die Gleichung 3.3.1 beschreibt die Veränderung des versteckten Vektors über die Zeit t, dazu werden der alte versteckte Vektor und die Eingabe X durch eine voll verbundene

Schicht kombiniert. Die Ausgabe wird in Gleichung 3.3.2 aus dem aktuellen versteckten Vektor errechnet. Φ bezeichnet eine Aktivierungsfunktion, $\overline{\mathbf{W}}_{xh}$, $\overline{\mathbf{W}}_{hh}$ und $\overline{\mathbf{W}}_{hy}$ sind die Gewichtsmatrizen sowie B_h und B_y die Biasvektoren für die Linearkombination.

LSTM-Zellen enthalten noch einen internen Zustandsvektor C_t, welcher durch Gatter neue Informationen aufnehmen, abrufen und löschen kann. Die folgenden Gleichungen beschreiben die Gatter I_t für die Eingabe, O_t für die Ausgabe und F_t für das Löschen im Zeitschritt t. Die punktweise Multiplikation ist hier mit $(\cdot)$ gekennzeichnet, $\sigma : \mathbb{R} \to (0, 1)$ ist ein Platzhalter für beliebige Sigmoidfunktionen.

$$\mathrm{I}_t = \sigma\Big(\mathrm{X}_t \times \overline{\mathbf{W}}_{xi} + \mathrm{H}_{t-1} \times \overline{\mathbf{W}}_{hi} + \mathrm{C}_{t-1} \cdot \mathrm{W}_{ci} + \mathrm{B}_i\Big) \tag{3.3.3}$$

$$\mathrm{O}_t = \sigma\Big(\mathrm{X}_t \times \overline{\mathbf{W}}_{xo} + \mathrm{H}_{t-1} \times \overline{\mathbf{W}}_{ho} + \mathrm{C}_{t-1} \cdot \mathrm{W}_{co} + \mathrm{B}_o\Big) \tag{3.3.4}$$

$$\mathrm{F}_t = \sigma\Big(\mathrm{X}_t \times \overline{\mathbf{W}}_{xf} + \mathrm{H}_{t-1} \times \overline{\mathbf{W}}_{hf} + \mathrm{C}_{t-1} \cdot \mathrm{W}_{cf} + \mathrm{B}_f\Big) \tag{3.3.5}$$

Die Gatter werden auf Basis der gewichteten Eingabe des letzten gewichteten versteckten Vektors und dem gewichteten letzten Zustandsvektor bestimmt. Schlussendlich können der neue Zustandsvektor und der neue versteckte Vektor mit den folgenden Gleichungen bestimmt werden:

$$\mathrm{C}_t = \mathrm{F}_t \cdot \mathrm{C}_{t-1} + \mathrm{I}_t \cdot \sigma\Big(\mathrm{X}_t \times \overline{\mathbf{W}}_{xc} + \mathrm{B}_c\Big) \tag{3.3.6}$$

$$\mathrm{H}_t = \mathrm{O}_t \cdot \sigma(\mathrm{C}_t) \tag{3.3.7}$$

Da die Eingaben für die HAR mit einer festen Fenstergröße vorliegen, kann nicht nur auf Informationen aus der Vergangenheit zugegriffen werden, sondern auch aus der Zukunft. Die Autoren nutzen deshalb Bidirektionale LSTM-Zellen (BLSTM). Für die Klassifikation werden Eingaben durch mehrere BLSTM-Zellen verarbeitet.

Edel und Köppe quantisieren die Gewichte des Netzes auf binäre Werte ($w \in \{-1, 1\}$). Als Quantisierungsfunktion benutzen sie die deterministische Signum-Funktion:

$$Q(x) = \begin{cases} 1 & \text{, falls } x \geq 0 \\ -1 & \text{, sonst} \end{cases} \tag{3.3.8}$$

Da $Q(x)$ nicht parametrisiert ist, muss auch nicht kalibriert werden. Die Aktivierungen werden entsprechend statisch quantisiert. Durch die Binarisierung ermöglicht sich die Implementierung einer linearen Schicht mit bitweisen Operatoren ($\oplus$ bezeichnet die XNOR-Funktion).

$$\hat{y}_j = b_j \sum_{i=1}^{m} Q(w_{ij}) \oplus x_i \tag{3.3.9}$$

$$y_j = Q(\hat{y}_j) \tag{3.3.10}$$

Die Gleichung 3.3.9 beschreibt die Berechnung einer gewichteten Summe der quantisierten Eingabe X, in Gleichung 3.3.10 wird die Aktivierung dann erneut quantisiert.

Auch hier wird das Netz zunächst mit reellen Parametern trainiert. Anschließend werden die Parameter mittels QAT feinjustiert. Insgesamt konnte durch die Quantisierung die Inferenzzeit für einen eigenen Datensatz um den Faktor 4 beschleunigt werden. Das nicht binarisierte BLSTM-RNN erzielt in den Auswertungen der Autoren konstant bessere Ergebnisse, als nicht rekurrente Modelle. Durch die Binarisierung des BLSTM-RNN ist die Klassifikationsleistung um etwa zwei Prozent gesunken.

3.4 LOW POWER LSTM PROCESSOR

Mazumder, Rashid und Mohsenin nutzen ebenfalls ein LSTM-basiertes Modell, welches schlussendlich auf einem FPGA (*engl. Field Programmable Gate Array*) für die 8-Bit Inferenz eingesetzt wurde [MRM20].

Die IMU-Daten (D Variablen) werden mit einer Fenstergröße von $T = 64$ segmentiert, sodass sich Eingaben der Form $D \times T$ ergeben. Das vorgestellte Modell verarbeitet ein Zeitfenster zunächst mit einer LSTM-Schicht, welche aus acht parallelen LSTM-Zellen besteht. Die Aktivierungen der LSTM-Schicht werden durch drei lineare Schichten mit 80, 32 und 13 Neuronen klassifiziert. Für die Quantisierung des trainierten Modells benutzen die Autoren die Post-Training-Quantization-Implementierung der *tensorflow-lite* Softwareumgebung. Sie spezifizieren nicht genauer, wie die Quantisierungsfunktionen kalibriert werden. Der Dokumentation [Ten] ist zu entnehmen, dass die Gewichte des Modells durch eine symmetrische affine Quantisierungsfunktion (Gleichung 2.4.1) auf 8-Bit ganze Zahlen quantisiert wird. Durch das Minimum/Maximum der Gewichte werden die Quantisierungsfunktionen kalibriert. Für die Aktivierungen wird eine dynamisch kalibrierte Quantisierungsfunktion verwendet.

Durch die Quantisierung des Modells konnten Mazumder, Rashid und Mohsenin eine Beschleunigung der Inferenzzeit um den Faktor 1,8 erzielen. Die Speichergröße des Modells wurde dabei halbiert. Aufgrund der geringen Größe übertrifft dieser Ansatz vorherige, auf FPGAs eingesetzte, Modelle in der Energieeffizienz. Durch die Quantisierung ist die Genauigkeit des Modells um etwa 2% gesunken.

4 METHODIK

Für die IMU-basierte Aktivitätserkennung stehen naturgemäß nur die beschränkten Ressourcen eines mobilen Systems zur Verfügung. Große Modelle erweisen sich daher als ungeeignet. Die Quantisierung von KNNs hat sich als universelle Methode zur Reduktion der Ressourcenanforderung bewährt [Kri18]. Für die Untersuchungen in dieser Arbeit wird das CNN-IMU als erfolgreiches CNN für die Aktivitätserkennung verwendet. Der Beitrag dieser Arbeit besteht in der Evaluierung verschiedener Quantisierungsansätze für das CNN-IMU. Im Gegensatz zum DFTerNet [Yan+18] und dem B-BLSTM-RNN [EK16] werden etwas höhere Auflösungen zwischen 2- und 7-Bit betrachtet, um zu erkunden, ob ein Kompromiss zwischen Quantisierungsstärke und Klassifikationsleistungsverlust gefunden werden kann. Ähnlich, wie in den Untersuchungen von Mazumder, Rashid und Mohsenin [MRM20] werden affine Quantisierungsfunktionen verwendet.

In den bisherigen Kapiteln wurden die notwendigen Grundlagen und verwandten Ansätze zur Quantisierung von KNNs für die Aktivitätserkennung vorgestellt. Dieses Kapitel befasst sich mit der in dieser Arbeit verwendeten Methodik. In Abschnitt 4.1 wird zunächst die Anwendung des CNN-IMUs beschrieben. Danach werden in Abschnitt 4.2 die für die Quantisierung verwendeten Methoden vorgestellt.

4.1 CNN-IMU

Als Modell für die IMU-basierte Aktivitätserkennung in dieser Arbeit wird das von Moya Rueda et al. in [MR+18] vorgestellte CNN-IMU verwendet. Wie der Name bereits verrät, handelt es sich um ein Faltungsnetz. In einem typischen HAR-Szenario kommen mehrere IMUs zum Einsatz, wobei jede IMU verschiedene Vorgänge simultan messen kann. Das CNN-IMU extrahiert für jede IMU separate Featuremaps aus all ihren Messpunkten. Schlussendlich wird die Klassifikation mit voll verbundenen Schichten durchgeführt.

Die Messungen der IMUs finden kontinuierlich statt. Da das CNN-IMU jedoch eine feste Eingabegröße erwartet, müssen die Messpunkte zeitlich segmentiert werden. Dafür wird ein Fenster mit der festen Größe T bei einer Schrittweite von s entlang der zeitlichen Dimension geschoben. Bei der Segmentierung der Trainingsdaten muss allerdings beachtet werden, dass zu einem Segment immer nur genau ein Label gehört. Norma-

lerweise ist jeder Zeitpunkt mit einem Label versehen, bei der Segmentierung gibt es nun genau T Label pro Segment. Für das Segment-Label kann der Modus als häufigstes Element der Labels gewählt werden. Die konkreten Fenstergrößen und Schrittweiten werden für jeden Datensatz unterschiedlich gewählt. Für das Training des CNN-IMUs wird sich an die von den Autoren vorgeschlagenen Werte gehalten.

Die Messpunkte eines Fensters werden für jede IMU in einer $D \times T$ Matrix abgelegt, wobei D die Anzahl der gemessenen Variablen der IMU bezeichnet. Die Faltungs- und Pooling-Operationen werden nur entlang der zeitlichen Dimension ausgeführt. Es werden dafür Blöcke aus zwei Faltungsschichten mit 64 (1×5) Filtern gefolgt von einer (1×2) Pooling-Schicht verwendet. Die Aktivierungen jeder Faltungsschicht werden mit der ReLU-Funktion bestimmt.

Für jede IMU werden B Blöcke nacheinander verknüpft. Am Ende der B Blöcke werden die Featuremaps jeder IMU mit einer eigenen linearen Schicht verarbeitet. Schlussendlich werden die Aktivierungen der linearen Schichten konkateniert und durch ein MLP klassifiziert. In Abbildung 4.1.1 ist der Gesamtaufbau des CNN-IMU dargestellt.

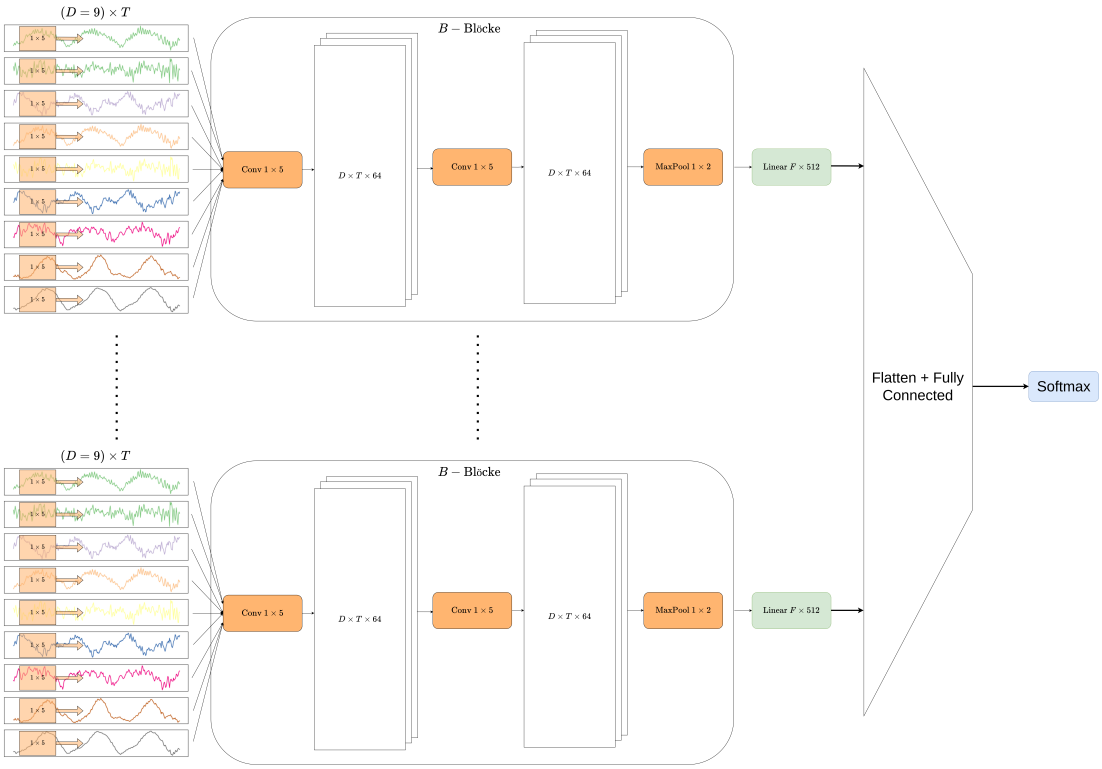

Abbildung 4.1.1: Gesamtaufbau des CNN-IMU.

4.2 QUANTISIERUNG VON KNNS

Im Grunde ist die Nutzung von diskreten Zuständen für KNNs schon seit den Anfängen in der Literatur vertreten (z.B. McCulloch-Pitts-Zellen, Abs. 2.2.1). Mit der Zeit haben sich verschiedene Techniken zur Quantisierung von modernen KNNs durchgesetzt. In dieser Arbeit wird auf die von Gholami et al. in [Gho+21] zusammengefassten Methoden zur Quantisierung zurückgegriffen. Zunächst wird in Abschnitt 4.2.1 diskutiert, nach welchen Gruppierungen die Parameter in einem Netz quantisiert werden. In den Abschnitten 4.2.2 und 4.2.3 wird das grundsätzliche Vorgehen bei der Quantisierung beschrieben. Abschließend wird in Abschnitt 4.2.5 gezeigt, wie die Parameter in Bezug auf Quantisierung angepasst werden können.

4.2.1 *Granularität*

Quantisierungsfunktionen, wie sie aus Gleichung 2.4.3 bekannt sind, werden in Bezug auf ihre Eingaben kalibriert. Je höher die Varianz der möglichen Eingabewerte ist, desto stärker ist der Informationsverlust durch die Quantisierung. Wenn die Parameter eines KNNs quantisiert werden, ist zu beachten, dass verschiedene Schichten im Netz unterschiedliche Wertebereiche haben können. Um den Wertebereich pro kalibrierter Quantisierungsfunktion gering zu halten, können die Parameter mit verschiedenen Granularitäten gruppiert werden. Beispielsweise kann schichtweise quantisiert werden, dabei wird für jede Schicht eine Quantisierungsfunktion kalibriert. Eine feinere Aufteilung für Faltungsschichten ist die kanalweise Quantisierung. Hier werden ein oder mehrere Filterkerne zusammen quantisiert. Es ist auch möglich, die Parameter noch feiner zu unterteilen. In dieser Arbeit werden schicht- und kanalweise Aufteilungen verwendet.

4.2.2 *Post Training Quantization*

Eine Form der Quantisierung künstlicher neuronaler Netze ist die Post Training Quantization (PTQ, *dt. Quantisierung nach dem Training*). Wie der Name bereits andeutet, wird ein bereits trainiertes Netz nachträglich quantisiert. Dafür werden zunächst die Parametergruppen mit für sie kalibrierten Quantisierungsfunktionen quantisiert. In Abbildung 4.2.1 ist die Quantisierung der Parametergruppe i (rot) mit ihrer Quantisierungsfunktion $Q_{p,i}$ dargestellt. $Q_{p,i}$ wird für die Parameter der Gruppe i kalibriert, um den Informationsverlust gering zu halten.

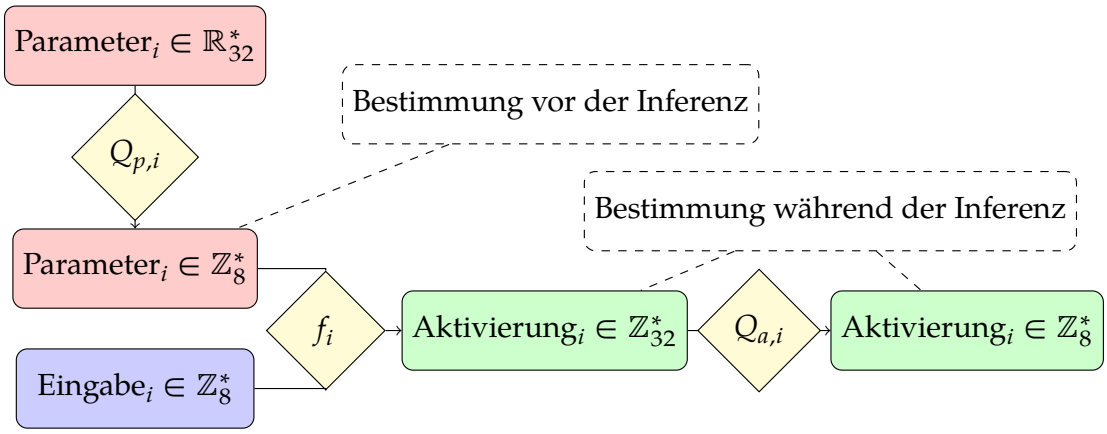

Abbildung 4.2.1: Quantisierungsfluss für die parametrisierte Übertragungsfunktion f_i.

Für die Ende-zu-Ende quantisierte Inferenz erwartet die Übertragungsfunktion f_i quantisierte Eingaben (blau) und quantisierte Parameter (rot). Eine Übertragung besteht üblicherweise aus gewichteten Summen. Handelt es sich bei f_i z. B. um eine lineare Übertragung mit einer Eingabe in $\mathbb{Z}^m_8$, dann benötigt die Ausgabe bereits $\log_2(2^8 \cdot 2^8 \cdot m) = 16 + \log_2(m)$ Bits um vollständig dargestellt werden zu können. Damit die so entstandene Aktivierung (grün) von der nächsten quantisierten Übertragungsfunktion weiter verwendet werden kann, muss sie erneut auf $\mathbb{Z}_8$ quantisiert werden. Hierfür kommt die zweite Quantisierungsfunktion $Q_{a,i}$ zum Einsatz. Im Gegensatz zu $Q_{p,i}$ hängt die Kalibrierung von den Eingaben der Übertragungsfunktion f_i ab, welche erst während der Inferenz bekannt sind. Zwei Ansätze für die Kalibrierung von $Q_{a,i}$ werden im nächsten Abschnitt erklärt.

Falls nicht Ende-zu-Ende quantisiert wird, müssen zwei weitere Fälle betrachtet werden. Liegt die Eingabe nicht quantisiert vor (beispielsweise die Netzeingabe), so wird sie ebenfalls wie eine Aktivierung behandelt. Es wird also eine Quantisierungsfunktion für die Eingabe kalibriert. Falls die Ausgabe nicht quantisiert sein muss, so kann die Ausgabe der Übertragungsfunktion direkt als reellwertig interpretiert werden.

4.2.3 *Statische / Dynamische Quantisierung*

Für die **dynamische** Quantisierung, auch *weight-only* Quantisierung genannt, werden lediglich die Parameter des Netzes nach dem Training quantisiert [Jac+17]. Die Quantisierung der Aktivierungen erfolgt während der Inferenz. Jedes Mal, wenn also eine Aktivierung quantisiert wird, wird die Quantisierungsfunktion $Q_{a,i}$ für sie kalibriert.

Um während der Inferenz den zusätzlichen Rechenaufwand für die Kalibrierung zu vermeiden, kann die **statische** Quantisierung verwendet werden. Dafür werden alle Quantisierungsfunktion $Q_{a,i}$ für einen repräsentativen Datensatz kalibriert. Es wird der Forwardpass für jedes Beispiel durchgeführt, wobei Statistiken der Aktivierungen erhoben werden. Die Kalibrierung der Quantisierungsfunktionen erfolgt auf Basis der Statistiken, so kann während der Inferenz statisch quantisiert werden. Im nächsten Abschnitt werden die verwendeten Kalibrierungsverfahren beschrieben.

4.2.4 *Kalibrierung*

In dieser Arbeit werden die Aktivierungsstatistiken mithilfe von Histogrammen, dem Minimum/Maximum und dem fortlaufenden Mittelwert aller Mini- und Maxima ermittelt [Wu+20; Quaa]. Da nur affine Quantisierungsfunktionen (Gleichung 2.4.3) verwendet werden, muss die Clipping-Range $[a, b]$ und die Verschiebung des Nullpunktes z bestimmt werden.

Für alle möglichen Eingaben X der zu kalibrierenden Quantisierungsfunktion wird die Clipping-Range mittels Minimum und Maximum durch

$$a = \min(X) \qquad b = \max(X) \tag{4.2.1}$$

bestimmt. Alternativ wird der fortlaufende Mittelwert aller Mini- und Maxima der gesehenen Eingaben X_t im Schritt t mit der Mittelungskonstante $c \in \mathbb{R}$ betrachtet. Die Clipping-Range wird so fortlaufend mit

$$a_t = (1 - c) \cdot a_{t-1} + c \cdot \min(X_t) \qquad b_t = (1 - c) \cdot b_{t-1} + c \cdot \max(X_t) \tag{4.2.2}$$

bestimmt. Initial gilt $a_0 = b_0 = 0$. Mithilfe von Histogrammen kann die Eingabeverteilung X_{bins} rekonstruiert werden. Die optimale Clipping-Range wird durch das folgende Minimierungsproblem bestimmt:

$$\min_{a,b}\left(\|Q_{\text{error}}(X_{\text{bins}})\|_2\right) \tag{4.2.3}$$

Die Verschiebung des Nullpunktes z wird in jedem Fall über die Skalierung s und den minimalen Wert des quantisierten Wertebereiches $Q_{\min}$ bestimmt:

$$z = \left\lfloor \frac{a}{s} \right\rceil - Q_{\min} \tag{4.2.4}$$

4.2.5 *Quantization Aware Training*

Der durch die Quantisierung eingeführte Fehler kann zu Leistungsverlust des Netzes führen. Um die Parameter auf den Effekt der Quantisierung abzustimmen, kann das

Quantization-Aware-Training (QAT, *dt. quantisierungsbewusstes Training*) angewendet werden. Wie auch bei der PTQ, wird beim QAT ein vortrainiertes Netz quantisiert. Während des QAT bleiben die Parameter und Gradienten in reeller Form. So wird sichergestellt, dass kleine Gradienten nicht durch die Rundung verloren gehen.

Für den Forwardpass soll der Quantisierungsfehler einen Einfluss auf den Netzfehler haben, sodass dieser minimiert werden kann. Alle Parameter werden während des Trainings reell vorgehalten, um den Effekt kleiner Gradienten nicht durch die Rundung zu neutralisieren. Für jedes Trainingsbeispiel werden die aktuellen reellen Parameter zunächst quantisiert, damit ein Ende-zu-Ende quantisierter Forwardpass ausgeführt werden kann.

Der Backwardpass wird dann vollständig in reeller Form durchgeführt. Das bedeutet, dass alle Gradienten in reeller Darstellung berechnet werden. So liegt in jeder Einheit der reelle Fehlergradient in Bezug auf die quantisierten Parameter vor. Mit der „Ableitung“ der Quantisierungsfunktion kann schlussendlich der Fehlergradient in Bezug auf die reellen Parameter bestimmt werden. So können über mehrere Trainingsbeispiele hinweg minimale Anpassungen an den Parametern vorgenommen werden, welche ggf. dazu führen, dass sich die Parameter nach der Quantisierung ändern.

In Abbildung 4.2.2 ist ein allgemeiner QAT-Schritt für die Funktion f mit den reellen Parametern θ, der quantisierten Eingabe $\tilde{X}$ und der quantisierter Ausgabe $\tilde{Y}$ dargestellt. δ bezeichnet den zurück propagierten Gradienten. Quantisierte Variablen sind hier mit einer Tilde gekennzeichnet.

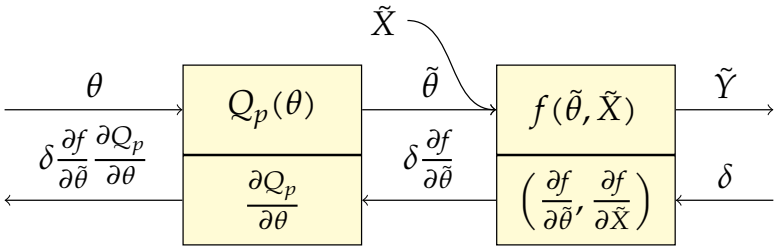

Abbildung 4.2.2: Ein allgemeiner QAT-Schritt mit der Quantisierungsfunktion Q_p als fester Bestandteil. Auf der linken Seite befinden sich die Parameter θ der Funktion f, welche für den Forwardpass quantisiert werden. Im Backwardpass wird der Gradient bis hin zu θ propagiert. Quantisierte Variablen sind mit einer Tilde gekennzeichnet.

Um den Fehlergradienten in Bezug auf die reellen Parameter zu bestimmen, werden allerdings die Ableitungen der Quantisierungsfunktionen benötigt. Allgemein sind Quantisierungsfunktionen, wie in Gleichung 2.4.5, nicht differenzierbar. Selbst wenn

die Sprungstellen ignoriert werden, handelt es sich noch um eine stückweise konstante Funktion, dessen Ableitung fast überall 0 beträgt. Üblicherweise werden affine Quantisierungsfunktionen, wie in Gleichung 2.4.3, mit einer linearen Funktion approximiert (STE, *engl. Straight Through Estimator*) [BLC13]. Man ignoriert also die Rundungsoperation. So lässt sich eine Ableitung bestimmen, welche als Approximation der tatsächlichen Ableitung genügt:

$$Q_{\text{STE}}(x) = \frac{x}{s} - z \qquad \frac{d}{dx} Q_{\text{STE}}(x) = \frac{1}{s} \tag{4.2.5}$$

Für den Backwardpass bei einer linearen Übertragung, wie sie aus Gleichung 2.2.16 bekannt ist, muss die Quantisierungsfunktion der Parameter und Aktivierungen mit einbezogen werden. Im folgenden Beispiel werden die affinen Quantisierungsfunktionen Q_p und Q_a benutzt, des Weiteren werden quantisierte Variablen mit einer Tilde gekennzeichnet. Die Gleichung 4.2.6 beschreibt den Forwardpass mit den Parametern $b, w_i \in \mathbb{R}$, welche durch Q_p quantisiert werden. In Gleichung 4.2.7 ist die Quantisierung der Ausgabe zu finden. Die Gleichungen 4.2.8 und 4.2.9 beschreiben, wie der Fehlergradient in Bezug auf die reellen Parameter bestimmt wird. Mithilfe des STE können hier die Ableitungen mit $Q_a' = \frac{1}{s_a}$ und $Q_p' = \frac{1}{s_p}$ abgeschätzt werden. δ ist der rückwärts propagierte Fehler in Bezug auf $\tilde{Y}$.

$$f(\tilde{X}) = Q_p(b) + \sum_{i=1}^{m} Q_p(w_i)\tilde{x}_i \overset{\text{def.}}{=} \tilde{b} + \sum_{i=1}^{m} \tilde{w}_i \tilde{x}_i \tag{4.2.6}$$

$$\tilde{Y} = Q_a(f(\tilde{X})) \tag{4.2.7}$$

$$\frac{\partial E}{\partial w_i} = \frac{\partial \tilde{w}_i}{\partial w_i} \cdot \frac{\partial f(\tilde{X})}{\partial \tilde{w}_i} \cdot \frac{\partial \tilde{Y}}{\partial f(\tilde{X})} \cdot \frac{\partial E}{\partial \tilde{Y}} \overset{\text{STE}}{\approx} \frac{1}{s_p} \cdot \tilde{x}_i \cdot \frac{1}{s_a} \cdot \delta \tag{4.2.8}$$

$$\frac{\partial E}{\partial b} = \frac{\partial \tilde{b}}{\partial b} \cdot \frac{\partial f(\tilde{X})}{\partial \tilde{b}} \cdot \frac{\partial \tilde{Y}}{\partial f(\tilde{X})} \cdot \frac{\partial E}{\partial \tilde{Y}} \overset{\text{STE}}{\approx} \frac{1}{s_p} \cdot 1 \cdot \frac{1}{s_a} \cdot \delta \tag{4.2.9}$$

Nachdem das QAT abgeschlossen ist, werden die Parameter mit den entsprechenden kalibrierten Quantisierungsfunktionen quantisiert. Die Aktivierungen können statisch oder dynamisch quantisiert werden.

5 EXPERIMENTE

In diesem Kapitel geht es um die Umsetzung und Evaluierung der Methodik (Kapitel 4). Zunächst werden die verwendeten Metriken für die Evaluierung in Abschnitt 5.1 erklärt. Danach geht es in Abschnitt 5.2 um die Aufbereitung der verschiedenen Datensätze. Nachdem das Vorgehen der Experimente in Abschnitt 5.4 beschrieben wurde, werden die Ergebnisse der einzelnen Schritte in den Abschnitten 5.5, 5.6, 5.7 und 5.8 evaluiert. Abschließend werden die Ergebnisse dieser Arbeit mit den verwandten Arbeiten verglichen (Abschnitt 5.9).

5.1 METRIKEN

5.1.1 *Klassifikationsleistung*

In dieser Arbeit werden unter anderem die Leistungen von nicht-binären Klassifikatoren untersucht. Als Metriken bieten sich die allgemeine Genauigkeit und Variationen des F_1-Scores an [GBV20]. Ein Testbeispiel gilt als korrekt klassifiziert, wenn die höchste errechnete Klassenwahrscheinlichkeit die des erwarteten Labels ist. Die Genauigkeit ist der relative Anteil der korrekt klassifizierten Testbeispiele.

$$\text{acc} = \frac{\#\text{Korrekt}}{\#\text{Testbeispiele}} \tag{5.1.1}$$

Der F_1-Score bezieht sich erst einmal auf binäre (positiv/negativ) Klassifikatoren. Für alle Testbeispiele lassen sich die Ergebnisse in die Kategorien *true positive* (TP), *true negative* (TN), *false positive* (FP) und *false nevative* (FN) einordnen. TP und TN sind korrekt klassifizierte positive und negative Beispiele. FN und FP sind falsch klassifizierte positive und negative Beispiele (vgl. Abbildung 5.1.1). Aus diesen Kategorien lassen sich Precision (P, *dt. Genauigkeit*) und Recall (R, *dt. Trefferquote*) bestimmen. Unter Precision versteht man den relativen Anteil der richtig positiv klassifizierten Beispiele unter allen positiv vorhergesagten Beispielen. Der Recall gibt den relativen Anteil der richtig positiv klassifizierten Beispiele unter den Beispielen mit positivem Label an.

$$P = \frac{\#TP}{\#TP + \#FP} \qquad R = \frac{\#TP}{\#TP + \#FN} \tag{5.1.2}$$

		Vorhersage	
		positiv	negativ
Label	positiv	TP	FN
	negativ	FP	TN

Abbildung 5.1.1: Konfusionsmatrix eines binären Klassifikators.

		Vorhersage			
	Klasse	1	2	3	4
Label	1	TN	FP	TN	TN
	2	FN	TP	FN	FN
	3	TN	FP	TN	TN
	4	TN	FP	TN	TN

Abbildung 5.1.2: Konfusionsmatrix für die Klasse $c = 2$ eines nichtbinären Klassifikators.

Der F_1-Score ist das harmonische Mittel zwischen Precision und Recall. Er nimmt somit Werte zwischen 0 und 1 an, je höher der Wert, desto besser ist die Leistung des Klassifikators.

$$F_1 = \frac{2}{\frac{1}{P} + \frac{1}{R}} = 2\frac{P \cdot R}{P + R} \tag{5.1.3}$$

Für die Bewertung eines Klassifikators mit mehreren Klassen wird eine erweiterte Definition von TP, TN, FP und FN benötigt. Man betrachtet dafür alle Testbeispiele gruppiert nach Klassenlabel c. Für jede Gruppe geben TP_c und FP_c alle Beispiele an, welche richtig oder falsch als c klassifiziert wurden. Analog geben TN_c und FN_c alle Beispiele an, welche richtig oder falsch als *nicht* c klassifiziert wurden (vgl. Abbildung 5.1.2). Mit dieser Definition können auch Precision und Recall für ein Mehrklassen-Szenario erweitert werden.

$$P_c = \frac{\#TP_c}{\#TP_c + \#FP_c} \qquad R_c = \frac{\#TP_c}{\#TP_c + \#FN_c} \tag{5.1.4}$$

Entsprechend ergibt sich für den F_1-Score ergibt sich Folgendes:

$$F_{1_c} = \frac{2}{\frac{1}{P_c} + \frac{1}{R_c}} = 2\frac{P_c \cdot R_c}{P_c + R_c} \tag{5.1.5}$$

Um die F_1-Scores jeder Klasse zusammenzuführen wird üblicherweise ein gewichteter Mittelwert berechnet. Die Gewichte entsprechen den relativen Häufigkeiten der jeweiligen Klasse im Datensatz.

$$wF_1 = \sum_{c \in \{1,\dots,n\}} F_{1_c} \frac{\#\text{Beispiele mit Label } c}{\#\text{Beispiele}} \tag{5.1.6}$$

5.1.2 *Komplexität*

Zur Bewertung, der durch die Quantisierung eines KNN entstehenden Vorteile, werden der Rechenaufwand und der Speicherbedarf betrachtet.

Der theoretische Speicherbedarf der Parameter ist relativ simpel zu bestimmen. Er ergibt sich aus der Addition der benötigten Bits aller n Parameter θ_i im Netz.

$$\#\text{Bits} = \sum_{i=1}^{n} \#\text{Bits}(\theta_i) \tag{5.1.7}$$

Für die tatsächliche Speichernutzung während der Inferenz muss allerdings die Speichernutzung der Aktivierungen mit einberechnet werden. Durch Hard- und Softwareunterschiede kann diese stark variieren. Für einen plattformunabhängigen Vergleich wird in dieser Arbeit auf den theoretischen Speicherbedarf der Parameter zurückgegriffen.

Der Rechenaufwand für die Inferenz lässt sich ebenfalls auf unterschiedliche Art und Weise bestimmen. Einerseits kann die theoretische Anzahl der MAC-Operationen (*engl. multiply-accumulate*) für ein Batch abgeschätzt werden. Andererseits kann die tatsächliche Inferenzzeit gemessen werden. Eine MAC-Operation ist eine Akkumulation eines Produkts in einem Register:

$$\text{accum} \leftarrow \text{accum} + (a \cdot b) \tag{5.1.8}$$

Die theoretische Anzahl der MAC-Operationen verändert sich durch die Quantisierung eines KNN nicht. Es werden lediglich ganzzahlige Operationen anstelle von Gleitkommaoperationen (FLOPs, *engl. floating-point operations*) ausgeführt. Dadurch kann ein signifikanter Zeitvorteil entstehen, da ganzzahlige Operationen deutlich schneller berechnet werden. Hier wird die Inferenzzeit als Metrik für den Rechenaufwand genutzt, so kann ein Vergleich zwischen dem quantisierten und dem nicht-quantisierten KNN stattfinden. Auch hier besteht eine Abhängigkeit zu der genutzten Hard- und Software, allerdings kann der relative Zeitgewinn mit anderen Arbeiten verglichen werden.

5.2 DATENSÄTZE

In dieser Arbeit werden drei verschiedene Datensätze für die Experimente verwendet. Dazu gehören zwei Benchmark-Datensätze für eine Vergleichsbasis mit den verwandten Arbeiten und ein spezieller Datensatz aus dem Bereich der Logistik. Dieser Abschnitt soll einen Überblick über die verwendeten Datensätze und die zugehörige Vorverarbeitung bieten.

5.2.1 *Pamap2*

Im Rahmen des PAMAP-Projektes (*Physical Activity Monitoring for Aging People*) wurde 2012 von Reiss und Stricker der Pamap2-Datensatz veröffentlicht [RS12]. Er beinhaltet IMU-Aufzeichnungen von Aktivitäten aus dem täglichen Leben. Es wurden von neun Probanden insgesamt über zehn Stunden Sensordaten aufgezeichnet, welche sich in 18 Klassen wie z. B. *Laufen*, *Treppensteigen* oder *Bügeln* einteilen lassen. Die Daten wurden von drei am Körper befestigen IMUs und einem Herzfrequenzsensor erhoben. Jede IMU misst auf 3-Achsen mit je zwei Akzelerometern, einem Gyroskop und einem Magnetometer. Platziert wurden die IMUs je an der dominanten Hand, dem dominanten Fußgelenk und am Torso. Für das CNN-IMU wird der Herzfrequenzsensor als eine separate IMU mit einer gemessenen Variable angesehen.

Zum Aufnahmeprotokoll gehören die zwölf Aktivitäten *Liegen, Sitzen, Stehen, Laufen, Fahrradfahren, Nordic Walking, Treppenhinaufgehen, Treppenhinabgehen, Bügeln, Staubsaugen und Seilspringen*. Des Weiteren wurden noch optionale Tätigkeiten, wie etwa *Fußball spielen* ausgeführt. Für das Training des CNN-IMU und die Vorverarbeitung wird Moya Rueda et al. [MR+18] gefolgt, die Aufzeichnungen von Person 5 und 6 werden entsprechend für die Validierung und das Testen verwendet. Mit den restlichen Aufzeichnungen wird das Modell trainiert.

Für die Vorverarbeitung werden zunächst die Messungen des Herzfrequenzsensors aufgrund der geringeren Abtastrate linear interpoliert. Anschließend werden alle Zeitpunkte, bei welchen ein Messpunkt fehlt (*NaN*), entfernt. In dieser Arbeit werden elf[1] Aktivitäten des Aufnahmeprotokolls betrachtet, deshalb wird jeder Messpunkt, welcher nicht zu einer dieser Klassen gehört, entfernt. Die Messfrequenz wird danach von 100Hz auf 30Hz mittels Downsampling reduziert. Zuletzt werden die Messpunkte jeder Sensordimension auf einen Mittelwert von $\frac{1}{2}$ und eine Varianz von 1 normiert.

Die vorverarbeiteten Daten werden für das Training segmentiert. Es wird eine Fenstergröße von $T = 100$ (3 Sekunden) bei einem Stride von $s = 33$ (1 Sekunde) gewählt. Die entstehenden Segmente werden dynamisch mit einem additiven gaußschem Rauschen ($\mu = 0, \sigma = 0{,}01$) versehen. In Abbildung 5.2.1 ist die Klassenverteilung unter allen Segmenten nach der Vorverarbeitung dargestellt.

1 Aktivitäten des Aufnahmeprotokolls abzüglich der Klasse *Bügeln*

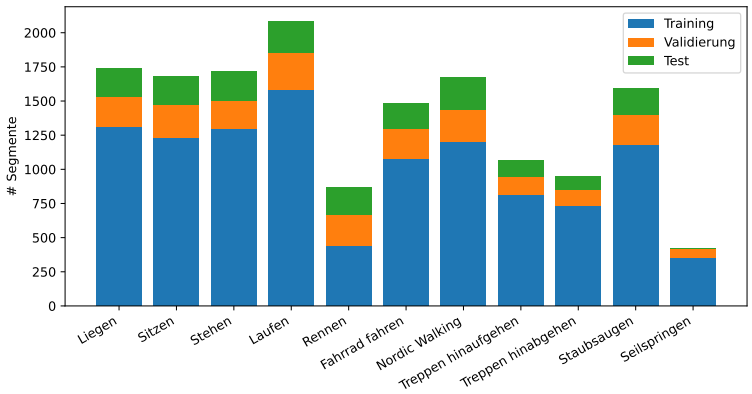

Abbildung 5.2.1: Verteilung der Aktivitätsklassen des Pamap2-Datensatzes nach der Vorverarbeitung.

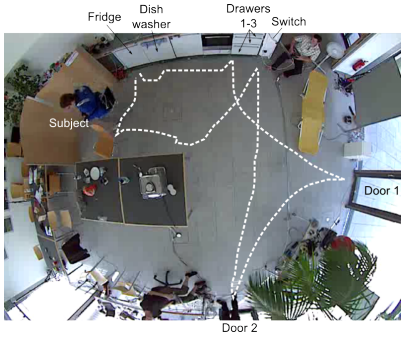

Abbildung 5.2.2: Aufnahmeumgebung des Opportunity-Datensatzes mit den üblichen Laufwegen. Entnommen aus [Rog+10].

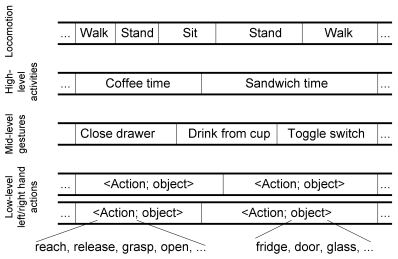

Abbildung 5.2.3: Kategorien der Aktivitätsklassen im Opportunity-Datensatz. Entnommen aus [Rog+10].

5.2.2 *Opportunity*

Als Teil des EU-Projektes *OPPORTUNITY* wurde von Roggen et al. ein Datensatz mit IMU-Aufzeichnungen von menschlichen Aktivitäten in einer sensorreichen Umgebung veröffentlicht [Rog+10]. Dafür wurden 72 Sensoren sowohl an den Probanden als auch in der Umgebung angebracht. Insgesamt wurden von zwölf Probanden etwa 25 Stunden an Sensordaten erhoben. Sie wurden dazu aufgefordert, eine feste Routine aus abstrakten Aufgaben, wie z. B. *Kaffee zubereiten, Aufräumen* oder *etwas Essen*, auszuführen, die genaue Ausführung blieb den Probanden überlassen. In Abbildung 5.2.2 ist die Testumgebung mit den üblichen Laufwegen für die ausgeführten Aktivitäten zu erkennen.

Die Annotationen der Daten sind in verschiedene Kategorien von einfachen Bewegungen bis hin zu komplexen Aufgaben aufgeteilt (Abbildung 5.2.3). Die Aktivitätsklassen lassen sich unter anderem in Bewegungs- und Mensch-zu-Objekt-Aktivitäten einteilen. In dieser Arbeit werden die Bewegungsaktivitäten (*Opportunity-Locomotion*) verwendet.

Auch hier wird sich für die Vorverarbeitung der Daten für das CNN-IMU an Moya Rueda et al. [MR+18] gehalten. Für die Validierung werden die Aufzeichnungen $ADL3$ der Probanden zwei und drei verwendet, zum Testen die Aufzeichnungen $ADL4$ und $ADL5$ der Probanden zwei und drei. Die restlichen Aufnahmen werden für das Training verwendet. Zunächst werden ungültige Spalten der Sensordaten in den Aufzeichnungen auf 0 gesetzt, damit nicht zu viele Zeitpunkte mit *NaN* Einträgen gelöscht werden müssen. Eine Spalte wird in dieser Arbeit als ungültig angesehen, wenn über 50% aller Messpunkte *NaN* ist. Anschließend werden alle Zeitpunkte mit mindestens einem fehlenden Eintrag (*NaN*) gelöscht. Da nur die Bewegungsaktivitäten betrachtet werden, werden auch nur die Aufzeichnungen von an Menschen befestigten IMUs benutzt. Auch hier werden die Messpunkte jeder Sensordimension auf einen Mittelwert von $\frac{1}{2}$ und eine Varianz von 1 normiert.

Die Abtastrate der Sensoren beträgt 30Hz, segmentiert wird mit einer Fenstergröße von $T = 24$ (800ms) und bei einem Stride von $s = 12$ (400ms). Auch hier werden die entstehenden Segmente dynamisch mit einem additiven gaußschem Rauschen ($\mu = 0, \sigma = 0{,}01$) versehen. Die Verteilung der Aktivitätsklassen aller Segmente ist in Abbildung 5.2.4 dargestellt.

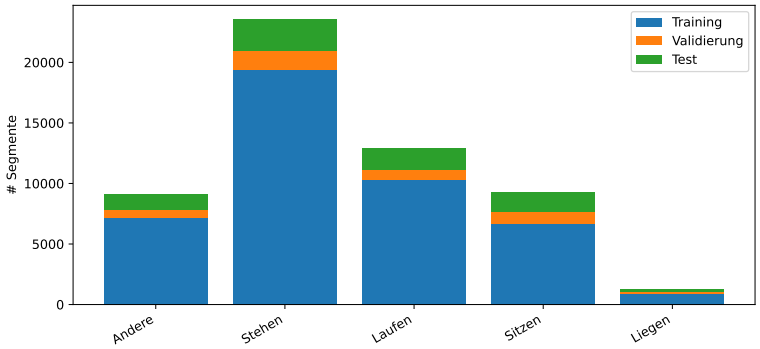

Abbildung 5.2.4: Verteilung der Aktivitätsklassen des Opportunity-Locomotion-Datensatzes nach der Vorverarbeitung.

5.2.3 *LARa*

2020 wurde von Niemann et al. der LARa-Datensatz für die Aktivitätserkennung in der Logistik veröffentlicht [Nie+20]. Die Daten wurden in drei verschiedenen Logistik-Szenarien aufgezeichnet. Die Probanden haben dafür beispielsweise Waren in Paketen verpackt, diese etikettiert und mit Wägen durch die Lagerhalle transportiert. In Abbildung 5.2.5 sind die Aufnahmeumgebungen des zweiten und dritten Logistik-Szenarios dargestellt.

Abbildung 5.2.5: Umgebung des zweiten (links) und dritten (rechts) Logistik-Szenarios für die Aufnahme des LARa-Dazensatzes. Entnommen aus [Nie+20].

Insgesamt umfasst der LARa-Datensatz knapp 13 Stunden annotierte Sensoraufnahmen von 14 Probanden. Es wurden verschiedene IMUs und ein Motion-Capturing-System für die Aufzeichnung der Aktivitäten verwendet. Die Aktivitäten lassen sich in 8 Klassen mit 19 zusätzlichen Attributen unterteilen, wobei eine Klasse nicht erkennbare Aktivitäten enthält. In dieser Arbeit werden die aufgezeichneten Daten der sechs am Körper befestigen IMUs verwendet, um die Aktivitätsklassen vorherzusagen.

Für die Aufteilung in Trainings-, Validierungs- und Testmengen wird Niemann et al. [Nie+20] gefolgt. Es werden der Aufzeichnungen der Probanden $\{1, 2, 3, 4, 7, 8, 9, 10\}$ für das Training, $\{5, 11, 12\}$ für die Validierung und $\{6, 13, 15\}$ für das Testen verwendet. Da die IMU-Aufzeichnungen auch mit einer Abtastrate von 100Hz aufgenommen wurden, wird die Vorverarbeitung analog zu dem Pamap2-Datensatz ausgeführt. Es werden zunächst alle Zeitpunkte mit fehlenden (NaN) Messpunkten entfernt. Anschließend wird mittels Downsampling die Abtastrate auf 30Hz reduziert. Die Messpunkte aller Sensordimensionen werden auf einen Mittelwert von $\frac{1}{2}$ und eine Varianz von 1 normiert. Segmentiert wird mit einer Fenstergröße von $T = 100$ (3s) bei einem Stride von $s = 12$ (400ms). Die entstehenden Segmente werden dynamisch mit einem additiven gaußschem Rauschen $(\mu = 0, \sigma = 0{,}01)$ versehen. In Abbildung 5.2.6 ist die Verteilung der Aktivitätsklassen unter den Segmenten dargestellt.

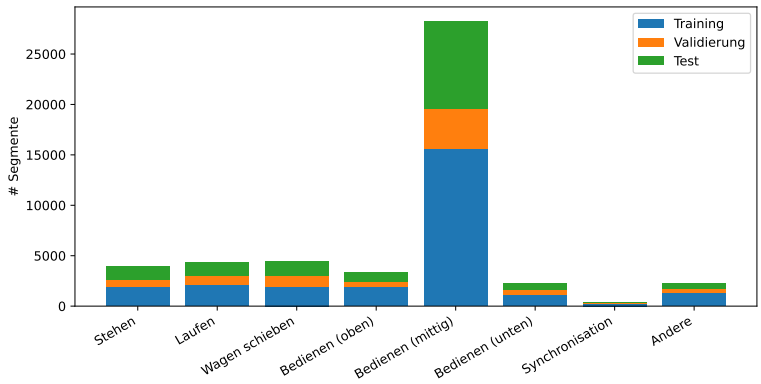

Abbildung 5.2.6: Verteilung der Aktivitätsklassen des LARa-Datensatzes nach der Vorverarbeitung.

5.3 VERWENDETE SOFTWARE

Für die Durchführung der Experimente wird das Deep-Learning-Framework *PyTorch* [Pas+19] verwendet. Es bietet eine Schnittstelle für diverse tensorbasierte Rechenoperationen mit automatischer Differenziation. Zudem werden einige Module für die Komposition von KNNs zur Verfügung gestellt. Neben der Unterstützung von Grafikkarten für die effiziente Berechnung der Operationen können auch Backends für quantisierte Tensoren verwendet werden [Quac]. Zum Zeitpunkt dieser Arbeit werden drei verschiedene Backends für die Quantisierung zum Teil als Beta- und Prototypen-Version unterstützt [Quab].

In dieser Arbeit wird das *FBGEMM*-Backend (*Facebook-General-Matrix-Multiplication*) aufgrund seiner Anpassungsmöglichkeiten verwendet. Es bietet stark optimierte Implementationen für quantisierte Matrix-Operationen [Khu+21]. Durch zur Laufzeit generierten Maschinencode können die Operationen auf bestimmte Matrix-Größen zugeschnitten werden. Das Backend unterstützt ganze Zahlen mit einer Auflösung von 8-Bit, die Gewichte werden dabei mit Vorzeichen und die Aktivierungen ohne Vorzeichen repräsentiert. Alle Operationen können Ende-zu-Ende quantisiert durchgeführt werden, FBGEMM quantisiert dafür die resultierenden Aktivierungen.

Es werden noch zwei weitere Backends unterstützt, welche in dieser Arbeit jedoch nicht verwendet werden. Das *QNNPACK*-Backend (*Quantized Neural Network PACKage*) bietet quantisierte Matrix-Operationen, welche für ARM-Prozessoren optimiert wurden [DWL18]. Es ist daher besonders für den Einsatz auf mobilen Geräten geeignet. Unterstützt werden ganze Zahlen mit einer Auflösung von 8-Bit. Da für die Experimente ein x86_64-Prozessor verwendetet wird, eignet sich dieses Backend nicht.

Das *TensorRT*-Backend ermöglicht die Beschleunigung von quantisierten Operationen durch Grafikkarten. Es erlaubt neben der Inferenz auch das optimierte QAT mit 8-Bit ganzen Zahlen. Die PyTorch-Anbindung ist allerdings noch experimentell und erlaubt wenige Anpassungen in Bezug auf die Quantisierung, weshalb in dieser Arbeit auf FBGEMM zurückgegriffen wird.

5.4 GLIEDERUNG DER EXPERIMENTE

Die zentrale Fragestellung der Experimente in dieser Arbeit bezieht sich auf das Verhalten des CNN-IMU bei verschiedenen Quantisierungsmodi. Gibt es bestimmte Konfigurationen, welche sich besonders eignen, um das CNN-IMU mit überschaubarem Leistungsverlust zu quantisieren? Um dieser Frage auf den Grund zu gehen, werden die Untersuchungen in vier Teile gegliedert:

I Das CNN-IMU wird in reeller-Form trainiert, dabei werden verschiedene Konfigurationen ausprobiert. Die besten Modelle für jeden Datensatz dienen als Grundlage für die Quantisierung.

II Es wird die statische PTQ für verschiedene Auflösungen angewandt, dabei werden die besten Kalibrierungsverfahren gesucht.

III Anschließend wird der Effekt der partiellen PTQ untersucht. Ist es sinnvoll Teile des Netzes *nicht* zu quantisieren?

IV Zuletzt wird überprüft, ob die vorherigen Quantisierungsmodi durch das QAT verbessert werden können.

5.5 ERGEBNISSE DES CNN-IMU TRAININGS

Die ersten Untersuchungen bestehen im Training des CNN-IMUs für die drei verwendeten Datensätze. In Tabelle 5.5.1 sind die verschiedenen Einstellungen der Hyperparameter aufgelistet. Für das Training werden die Gewichtsmatrizen orthogonal initialisiert [ACB19]. Nach je 20% aller Trainingsbeispiele in einer Epoche wird die Leistung der aktuellen Parameter für den gesamten Validierungsdatensatz ausgewertet. Wenn sich der Kreuzentropie-Fehler (Gleichung 2.2.10) nach 32 Validierungsschritten (etwa sechs Epochen) nicht verbessert hat, wird das Training gestoppt. Die Parameter, welche den besten wF_1-Score während der Validierung erzielt haben, werden für das finale Modell ausgewählt. Schlussendlich wird das finale Modell jeder Hyperparameter-Konfiguration auf dem Testdatensatz ausgewertet. Das Modell mit dem besten wF_1-Score wird als Grundlage für die Quantisierung in den späteren Experimenten genutzt.

Anzahl der Faltungsblöcke pro IMU	$2, 3$
Breite der voll-verbundenen Schichten	$256, 512$
Optimierer	Adam $\alpha = 10^{-3}$ $\beta_1 = 0,9$ $\beta_2 = 0{,}999$

Tabelle 5.5.1: Betrachtete Hyperparameter für das Training des CNN-IMU.

In Abbildung 5.5.1 sind die Trainingsverläufe des besten Modells für jeden Datensatz dargestellt. Die Kreuzentropie-Fehler während des Trainings und der Validierung sind in Blau und Orange eingezeichnet. Der Fehler während des Trainings wurde für die

Übersichtlichkeit als Mittelwert von je 25 Iterationen dargestellt. In Grün ist der wF_1-Score für die Validierungsmenge dargestellt, welcher als Auswahlkriterium der besten Parameter genutzt wird. Die rote vertikale Linie kennzeichnet die Iteration des besten Modells bezüglich des Auswahlkriteriums. Es ist zu erkennen, dass Overfitting im Sinne des Fehlers bereits ab etwa dem 500. Trainings-Batch auftritt.

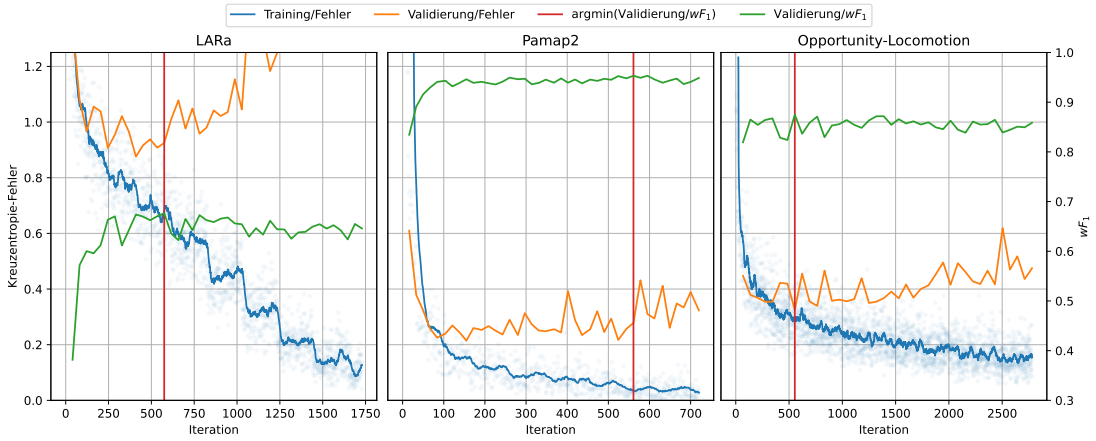

Abbildung 5.5.1: Die besten Trainingsverläufe des reellen CNN-IMUs bei einer Batch-Größe von 128. Der Trainingsfehler wurde für je 25 Iterationen gemittelt.

Die Ergebnisse dieses Auswahlverfahrens sind in Tabelle 5.5.2 dargestellt. In den Spalten *B* und *FC* stehen die Anzahl der CNN-IMU-Blöcke und die Anzahl der Features in den linearen Schichten. *#IMU* bezeichnet die Anzahl der IMU-Zweige im CNN-IMU bei einer Fenstergröße von *T*. Die Spalte *Batch-Time* enthält den Mittelwert und die Standardabweichung aller gemessenen Inferenzzeiten für die Test-Batches der Größe 128. Alle Zeitmessungen wurden auf einem isolierten System[2] durchgeführt.

Durch diese Ergebnisse lässt sich bereits erkennen, dass die Fenstergröße *T* und die Anzahl der Features in den linearen Schichten *FC* maßgebende Variablen für die Inferenzzeit sind.

2 OS: `Linux-5.10.0-21-amd64-x86_64-with-glibc2.31`
CPU: `Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz`
PyTorch: `2.0.0.dev20230203+cu117`

	#IMU	T	B	FC	wF_1[%]	Acc [%]	Batch-Time [ms]
LARa	5	100	2	512	72,21	73,34	$226{,}8 \pm 1{,}8$
Pamap2	4	100	2	512	95,45	95,13	$293{,}6 \pm 4{,}6$
Opportuntiy	19	24	2	256	85,67	85,85	$217{,}8 \pm 8$

Tabelle 5.5.2: Die besten Ergebnisse des reellen CNN-IMU-Trainings.

5.6 ERGEBNISSE DER PTQ

PyTorch ermöglicht die Quantisierung mittels Observern (*dt. Beobachter*). Ein Observer wird an einer bestimmten Stelle in einem KNN eingesetzt, um Statistiken für die Kalibrierung einer Quantisierungsfunktion zu sammeln.

Die Observer unterstützen durch manuelles Überschreiben des quantisierten Wertebereiches $[Q_{\text{min}}, Q_{\text{max}}]$ eine virtuelle Verkleinerung der Auflösung. In dieser Arbeit wird so zusätzlich der Effekt von kleineren Auflösungen als sieben Bits überprüft. Die Kalibrierungsverfahren der Observer sind in Tabelle 5.6.1 gelistet. Für eine genauere Beschreibung sei auf Abschnitt 4.2.4 verwiesen.

Observer	Clipping-Range $[a, b]$	Nullpunkt z
MinMax	$a = \min(\mathrm{X}), b = \max(\mathrm{X})$	$\lfloor \frac{a}{s} \rceil - Q_{\text{min}}$
MovingAverageMinMax	$a_t = (1-c)a_{t-1} + c \cdot \min(\mathrm{X}_i)$ $b_t = (1-c)b_{t-1} + c \cdot \max(\mathrm{X}_i)$	$\lfloor \frac{a}{s} \rceil - Q_{\text{min}}$
Histogram	$a, b \widehat{=} \min_{a,b}\left(\lVert Q_{\text{error}}(\mathrm{X}_{\text{bins}}) \rVert_2\right)$	$\lfloor \frac{a}{s} \rceil - Q_{\text{min}}$

Tabelle 5.6.1: Bestimmung der Parameter einer affinen Quantisierungsfunktion durch die Observer.

Für jeden Datensatz wird das beste Modell zunächst mittels PTQ quantisiert. Dabei werden verschiedene Quantisierungsmodi (Tabelle 5.6.2) verwendet. Es werden Auflösungen zwischen zwei und sieben Bits betrachtet, wobei die Wertebereiche der Parameter *voll*, *symmetrisch* oder *halb* gewählt werden. Die Granularität der Quantisierung kann pro Tensor, also pro linearer Schicht oder pro Faltungsschicht gewählt werden. Alternativ können Parameter pro Kanal quantisiert werden, was bei linearen Schichten die Gewichte der einzelnen Ausgabeneuronen und bei Faltungsschichten die einzelnen Kerne umfasst.

	Aktivierungen	Parameter
Auflösung (n-Bit)	2-7	2-7
Wertebereich	Voll $[0, 2^n - 1]$	Voll $[-2^{n-1}, 2^{n-1} - 1]$ Symmetrisch $[1 - 2^{n-1}, 2^{n-1} - 1]$ Halb $[0, 2^{n-1} - 1]$
Kalibrierung	MinMax MovingAverageMinMax Histogram	MinMax
Granularität	Pro Tensor	Pro Tensor Pro Kanal

Tabelle 5.6.2: Betrachtete Konfigurationsmöglichkeiten der Quantisierungsfunktionen für Parameter und Aktivierungen.

Die Aktivierungen werden ausschließlich in voller Darstellung quantisiert. Die Parameter werden anhand der Mini- und Maxima kalibriert. Für die Aktivierungen werden zudem noch die fortlaufenden Mittlewerte der Mini- und Maxima und Histogramme verwendet (vgl. Tabelle 5.6.1). Als Kalibrierungsdatensatz werden die jeweiligen Trainingsmengen der Datensätze verwendet. Nach genau einer Kalibrierungsepoche wurden entsprechend alle Trainingsbeispiele in die Statistiken für die Kalibrierung aufgenommen. Da die Quantisierung deterministisch ist, werden alle Kombinationen der Quantisierungsmodi genau einmal angewendet.

Zunächst wurde für eine Auflösung von sieben Bits überprüft, inwiefern sich der Wertebereich, die Kalibrierung und die Granularität auf die Klassifikationsleistung des quantisierten Modells und die Kalibrierungsdauer auswirken. Durch eine Betrachtung der besten zehn Konfigurationen (Abbildung 5.6.1) lässt sich kein Zusammenhang zwischen Kalibrierungsdauer und Klassifikationsleistung erkennen. Zudem ist die Ausnutzung des Wertebereichs der quantisierten Gewichte ebenfalls nicht signifikant für die Klassifikationsleistung[3]. Für die restlichen Experimente mit kleineren Auflösungen als sieben Bits wird stets der volle Wertebereich verwendet.

3 Tabelle A.5.1

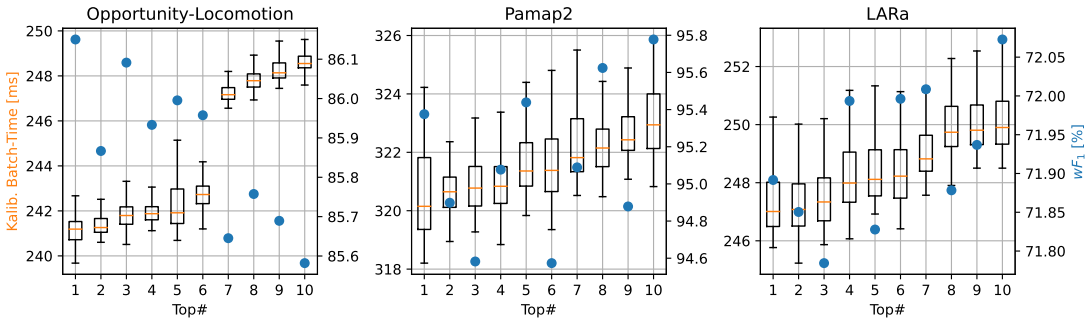

Abbildung 5.6.1: Die besten 10 PTQ Konfigurationen in Bezug den Median der Batch-Time während der Kalibrierung mit den zugeordneten wF_1-Scores (blau). Ein Vergleich aller Konfigurationen ist in Abbildung A.5.1 zu finden.

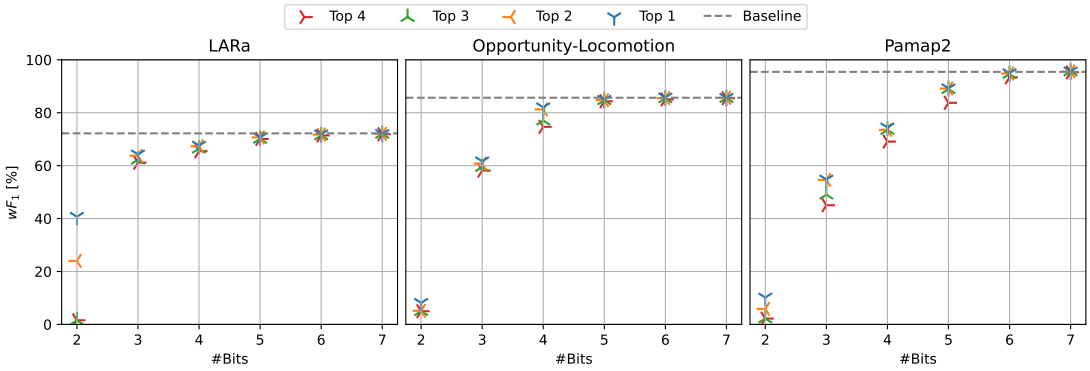

Abbildung 5.6.2: Klassifikationsleistung der vier besten PTQ-Konfigurationen pro Datensatz im Vergleich zur Baseline.

5.6.1 *Leistungsverlust*

Im Folgenden wird unter Baseline, sofern nicht anders angegeben, immer das nicht quantisierte CNN-IMU verstanden. In Abbildung 5.6.2 sind die Klassifikationsleistungen der vier besten Konfigurationen pro Auflösung dargestellt. Es ist zu erkennen, dass die Ende-zu-Ende-Quantisierung auf sechs und sieben Bits nahezu keinen Leistungsverlust mit sich bringt. Die Verluste des wF_1-Scores im Vergleich zur Baseline liegen für den LARa-, Opportunity-Locomotion- und Pamap2-Datensatz entsprechend in den Bereichen $[0{,}29\%;\ 1{,}12\%]$, $[-0{,}16\%;\ 0{,}71\%]$ und $[-0{,}22\%;\ 2{,}16\%]$. Auflösungen von zwei und drei Bits führen fast immer zu einer signifikanten Verringerung des wF_1-Scores. Hier liegen die Verluste entsprechend in den Bereichen $[11{,}04\%;\ 97{,}85\%]$,

[28,04%; 94,18%] und [42,60%; 97,69%]. Die Ergebnisse deuten an, dass bei einer guten Baseline der Effekt geringerer Auflösungen stärker ins Gewicht fällt.

Unter den besten vier Konfigurationen[4] für jeden Datensatz und jede Auflösung lässt sich kein Muster bezüglich der Granularität und der verwendeten Kalibrierungsstatistik erkennen. Für die weiteren Experimente wird deshalb die Annahme getroffen, dass durch eine kanalweise MinMax Quantisierung der Gewichte sowie der MovingAverageMinMax Quantisierung der Aktivierung kein signifikanter Nachteil gegenüber den anderen Konfigurationen entsteht.

5.6.2 *Zeitgewinn*

Durch die Ende-zu-Ende-Quatisierung des CNN-IMUs kann eine Verbesserung der Inferenzzeit erzielt werden. In Abbildung 5.6.3 sind die Verteilungen der gemessenen Inferenzzeiten pro Test-Batch der Größe 128 dargestellt. Auf der linken Seite sind die Zeiten der nicht quantisierten Modelle eingezeichnet, rechts die Zeiten des quantisierten Modelle mit einer eigenen Skala.

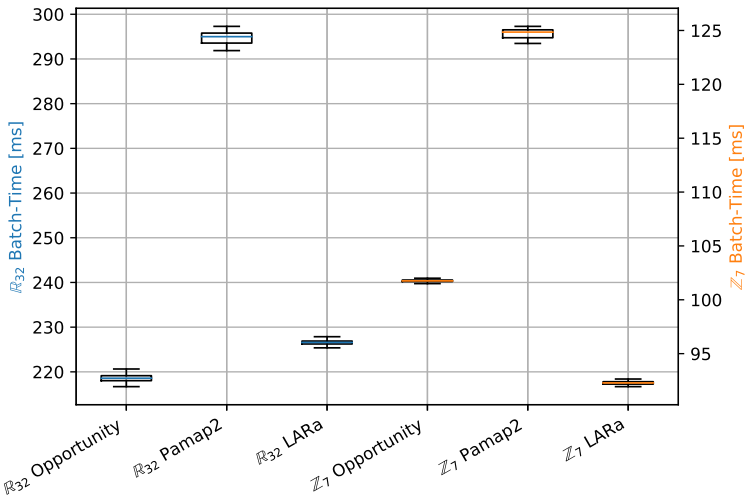

Abbildung 5.6.3: Inferenzzeiten für Batches der Größe 128 des quantisierten (orange) und nicht quantisierten (blau) CNN-IMUs.

Für den LARa-Datensatz ergibt sich so eine Verbesserung des Medians um 53,45% ($\Delta = -116{,}82$ms), für den Pamap2-Datensatz um 57,68% ($\Delta = -170{,}14$ms) und für den

4 LARa: Tabelle A.5.2, Opportunity-Locomotion: Tabelle A.5.3, Pamap2: Tabelle A.5.4

Opportunity-Locomotion-Datensatz um 59,27% ($\Delta = -134{,}3\text{ms}$). Es sei an dieser Stelle noch einmal angemerkt, dass mit dem FBGEMM-Backend die Auflösungen unter sieben Bits nur simuliert werden und somit keinen Effekt auf die Inferenzzeit haben.

Der Grafik ist zu entnehmen, dass sich die relative Lage der Zeiten unter den Datensätzen durch die Quantisierung verändert. Dies lässt sich darauf zurückführen, dass durch die unterschiedlichen Fenstergrößen und IMU-Anzahlen das Verhältnis von Features in den Faltungsschichten zu den Features in linearen Schichten verändert. Die Quantisierung beeinflusst den Inferenzzeitgewinn somit unterschiedlich stark.

5.7 ERGEBNISSE DER PARTIELLEN QUANTISIERUNG

Die weiteren Untersuchungen beziehen sich auf die partielle Quantisierung des CNN-IMUs. Durch die partielle Quantisierung wird versuch einen Kompromiss zwischen Inferenzzeitgewinn und Klassifikationsleistungsverlust für das CNN-IMU zu finden. Das CNN-IMU wird dafür in fünf Subnetze unterteilt, welche selektiv quantisiert oder nicht quantisiert werden. In Tabelle 5.7.1 sind die verschiedenen Subnetze mit ihren Bezeichnern aufgezählt. Durch Softwarelimitierungen werden die Aktivierungen nach einer quantisierten Schicht immer quantisiert. Wenn sie danach in reeller Form benötigt werden, werden sie rekonstruiert (Gleichung 2.4.4). Die Gewichte werden kanalweise mit der MinMax-Statistik kalibriert. Die Aktivierungen werden mit der MovingAverageMinMax-Statistik kalibriert.

Bezeichnung	Subnetz im CNN-IMU
Eingabe	Der erste Faltungsblock jeder IMU
Conv	Die Restlichen $B - 1$ Faltungsblöcke jeder IMU
IMU-FC	Die voll-verbundene Schicht am Ende der Faltungsblöcke jeder IMU
FC	Die ersten $N - 1$ voll-verbundenen Schichten des Klassifizierers
Ausgabe	Die letzte voll-verbundene Schicht des Klassifizierers

Tabelle 5.7.1: Subnetze des CNN-IMUs, welche selektiv quantisiert oder nicht-quantisiert werden.

In Abbildung 5.7.1 sind die Klassifikationsleistungen bei selektivem Auslassen der Quantisierung der einzelnen Subnetze dargestellt. Es lässt sich erkennen, dass das nicht Quantisieren der Eingabe- oder Ausgabe-Subnetze zum Teil deutlich bessere Klassifikationsleistungen für niedrigere Auflösungen erzielt. Bezüglich der Inferenzzeit

(Abbildung 5.7.2), macht es nahezu keinen Unterschied, wenn das AUSGABE-Subnetz nicht quantisiert ist. Ist allerdings das EINGABE-Subnetz nicht quantisiert, dann erhöht sich die Inferenzzeit drastisch. Allgemein ergibt sich durch die Quantisierung der linearen Schichten ein geringer Zeitvorteil. Durch die Quantisierung der Faltungsschichten wird die Inferenzzeit allerdings stark beschleunigt.

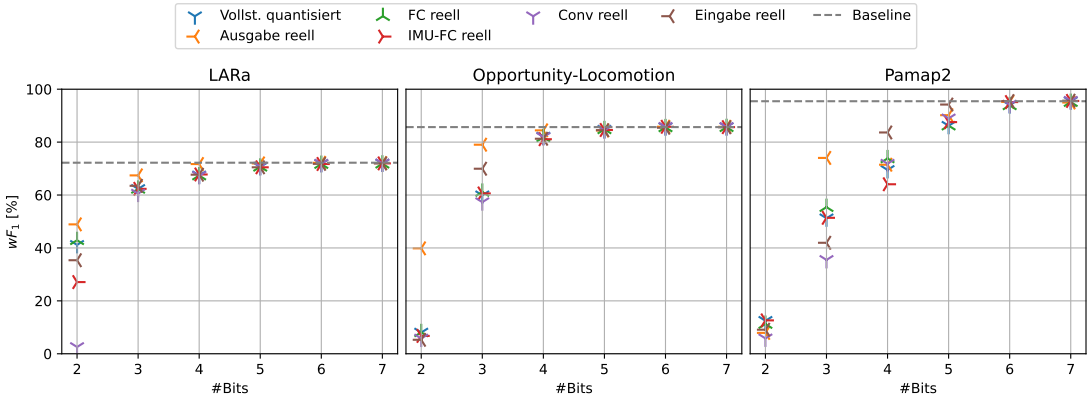

Abbildung 5.7.1: Klassifikationsleistung nach dem selektiven Auslassen der Quantisierung einzelner Subnetze des CNN-IMU.

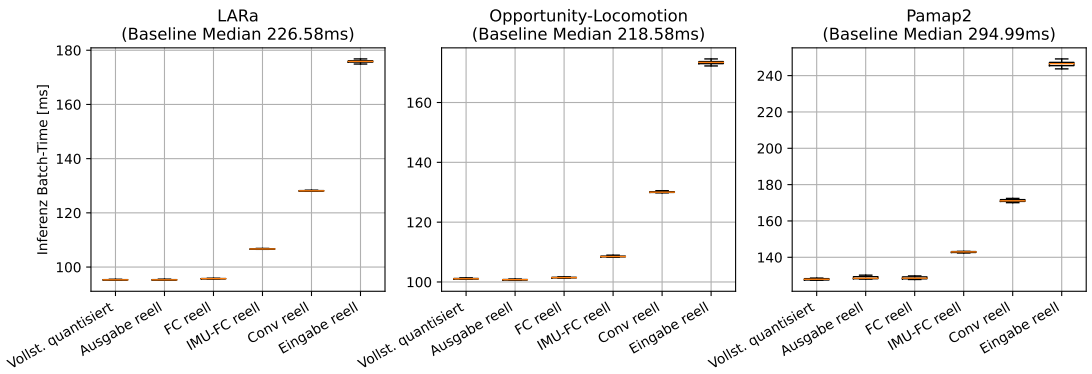

Abbildung 5.7.2: Inferenzzeit für Batches der Größe 128 nach dem selektiven Auslassen der Quantisierung einzelner Subnetze des CNN-IMUs.

Die Kombination der selektiven Quantisierung für EINGABE- und AUSGABE-Subnetze ist in Abbildung 5.7.3 dargestellt. Durch das nicht Quantisieren beider Subnetze lassen sich für niedrigere Auflösungen zum Teil Klassifikationsleistungen nahe der Baseline erzielen. Um weiterhin von dem Inferenzzeitgewinn der quantisierten Faltungsblöcke zu profitieren, wird nun die partielle Quantisierung der linearen Schichten genauer

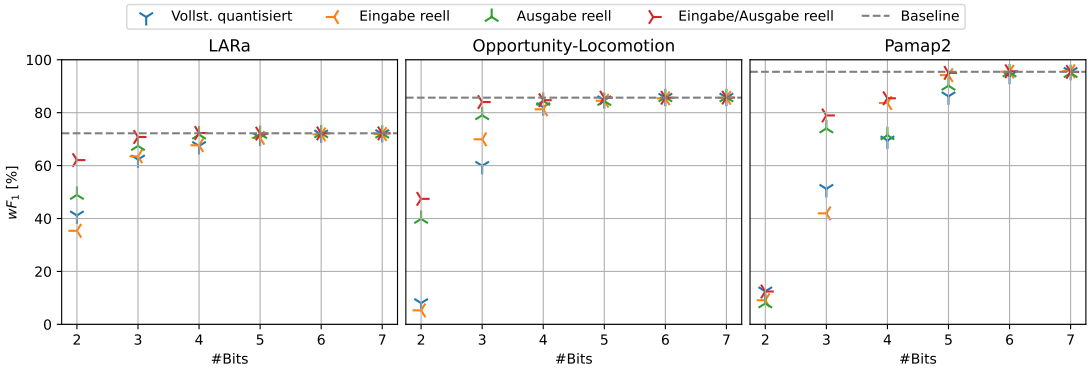

Abbildung 5.7.3: Klassifikationsleistungen nach dem selektiven Auslassen der Quantisierung der Eingabe- und Ausgabe-Subnetze des CNN-IMUs. Die Zeiten sind in Abbildung A.5.2 zu finden.

betrachtet. In Abbildung 5.7.4 sind die Klassifikationsleistungen bei der selektiven Quantisierung der Ausgabe-, FC- und IMU-FC-Subnetze dargestellt. Der Grafik ist zu entnehmen, dass es keinen signifikanten Vorteil mit sich bringt die FC- und IMU-FC-Subnetze zusätzlich zu dem Ausgabe-Subnetz nicht zu quantisieren. Im Anhang[5] sind die Ergebnisse weiterer Konfigurationen zu finden.

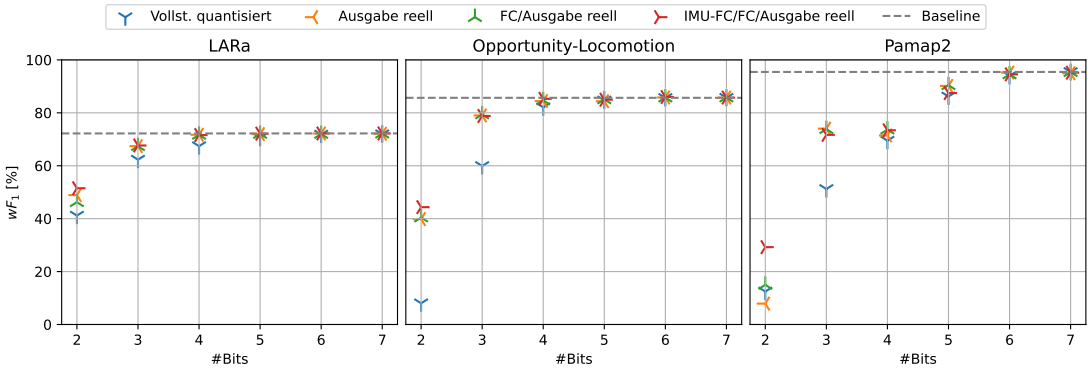

Abbildung 5.7.4: Klassifikationsleistungen nach dem selektiven Auslassen der Quantisierung von linearen Schichten im CNN-IMU. Die Zeiten sind in Abbildung A.5.3 zu finden.

Allgemein lässt sich hier festhalten, dass die Quantisierung aller Faltungsblöcke einen deutlichen Inferenzzeitgewinn mit sich bringt, wobei der Leistungsverlust überschau-

5 Vergleich 1: Abbildungen A.5.4, A.5.5; Vergleich 2: Abbildungen A.5.6, A.5.7

bar bleibt. Ist die Ausgabe nicht quantisiert, lässt sich der Leistungsverlust merkbar einschränken, wobei der Inferenzzeitverlust nicht signifikant ist.

5.7.1 *Speicherbedarf*

An dieser Stelle können bereits die Einsparungen bezüglich des Speicherbedarfes ausgewertet werden. Die einzigen zwei Hyperparameter, welche einen Einfluss auf den theoretischen Speicherbedarf (Gleichung 5.1.7) haben, sind die Auflösung und die Kombination der partiellen Quantisierung. In Abbildung 5.7.5 ist die Speichereinsparung in Abhängigkeit dieser beiden Hyperparameter dargestellt. Für die Darstellung wurde bewusst die Konfiguration *IMU-FC/FC/Output reell* für den Erhalt der Skala ausgelassen. Da der Großteil der Parameter des CNN-IMU im IMU-FC-Subnetz liegt, sind die quantisierten Netze dieser Konfiguration (7-Bit) für den LARa-, Opportunity-Locomotion- und Pamap2-Datensatz entsprechend 104MB, 58MB und 136MB groß. Durch diese Konfiguration entsteht kaum ein Vorteil für den theoretischen Speicherbedarf in Bezug auf die Baseline.

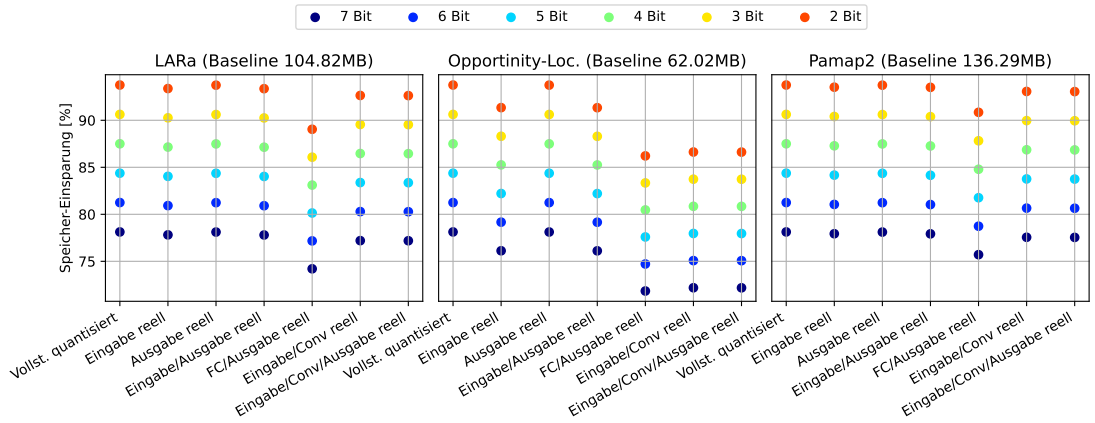

Abbildung 5.7.5: Theoretischer Speicherbedarf ausgewählter Kombinationen der partiellen Quantisierung für das CNN-IMU.

Bei allen Datensätzen ist zu erkennen, dass das Auslassen der Quantisierung des Ausgabe-Subnetzes nahezu keinen Unterschied im Vergleich zum vollständig quantisieren Netz macht. Zudem kann man der Grafik entnehmen, dass CNN-IMUs mit vielen IMU-Zweigen, wie z. B. für den Opportunity-Locomotion-Datensatz, merklich davon profitieren, wenn die Faltungsblöcke quantisiert sind. Zusammenfassend lässt

sich sagen, dass das IMU-FC-Subnetz in jedem Fall quantisiert werden sollte, wenn der Speicherbedarf eine wichtige Rolle spielt.

5.8 ERGEBNISSE DES QAT

Zuletzt wird untersucht, ob sich die Klassifikationsleistungen aus den vorherigen Experimenten durch QAT verbessern lassen. Da bei Auflösungen von sechs und sieben Bits nahe zu keine Verbesserungen zur Baseline möglich sind, werden sie nicht für das QAT in Betracht gezogen. Zudem kann durch das QAT nur die Klassifikationsleistung verbessert werden, weshalb die Speichernutzung und Inferenzzeit nicht erneut betrachtet werden.

Die reellen Modelle wurden mit dem Adam-Optimierer trainiert, weswegen die adaptiven Lernraten dazu beigetragen haben sich einem lokalen Minimum zu nähern. Wenn das Training (mittels QAT) ohne Beachtung der adaptiven Lernraten weitergeführt wird, entfernen sich die Parameter durch zu große Schritte vom lokalen Minimum. Aus diesem Grund werden die Schätzer der Momente (Gleichung 2.2.24) aus dem reellen Training für die Initialisierung des Adam-Optimierers im QAT verwendet.

Das QAT wird mit der Trainingsmenge der jeweiligen Datensätze durchgeführt. Nach 33% jeder Epoche wird die Leistung der aktuellen Parameter mit dem gesamten Validierungsdatensatz getestet. Dabei werden die Parameter, welche den besten wF_1-Score während der Validierung erzielt haben, für das finale Modell verwendet. Wenn sich der Validierungsfehler (Kreuzentropie) nach zehn Validierungsschritten (etwa drei Epochen) nicht verbessert hat, wird das Training abgebrochen. Für ein repräsentatives Ergebnis werden für jede Konfiguration $N = 5$ Durchläufe gestartet.

In den vorherigen Experimenten haben sich bereits die MovingAverageMinMax-Kalibrierung für Aktivierungen und die MinMax-Kalibrierung für Parameter als sinnvoll erwiesen. Zudem kann es neben der Ende-zu-Ende-Quantisierung in Betracht gezogen werden, das *Ausgabe*-Subnetz, die linearen Schichten und die Faltungsblöcke nicht zu quantisieren. Für diese Konfigurationen ist das Ergebnis des QAT in Abbildung 5.8.1 dargestellt. Mit den Kreuzen werden die vorherigen Ergebnisse der PTQ markiert, die Sterne in den jeweiligen Farben markieren den durchschnittlichen wF_1-Score aller QAT-Durchläufe. Dabei fällt auf, dass für den Opportunity-Locomotion-Datensatz und den LARa-Datensatz nur zum Teil Verbesserungen erzielt werden konnten. Für den Pamap2-Datensatz wurden fast ausschließlich Verbesserungen des wF_1-Scores erzielt. Diese Ergebnisse lassen vermuten, dass der Lernerfolg des QAT mit dem Lernerfolg des nicht quantisierten Modells positiv korreliert.

Weitere QAT-Ergebnisse zu anderen Konfigurationen sind im Anhang[6] zu finden. Auch dort konnten nur teilweise Erfolge durch das QAT erzielt werden. Allgemein ist aber zu erkennen, dass wenn eine Verbesserung erzielt wurde, diese bei niedrigeren Auflösungen größer ist.

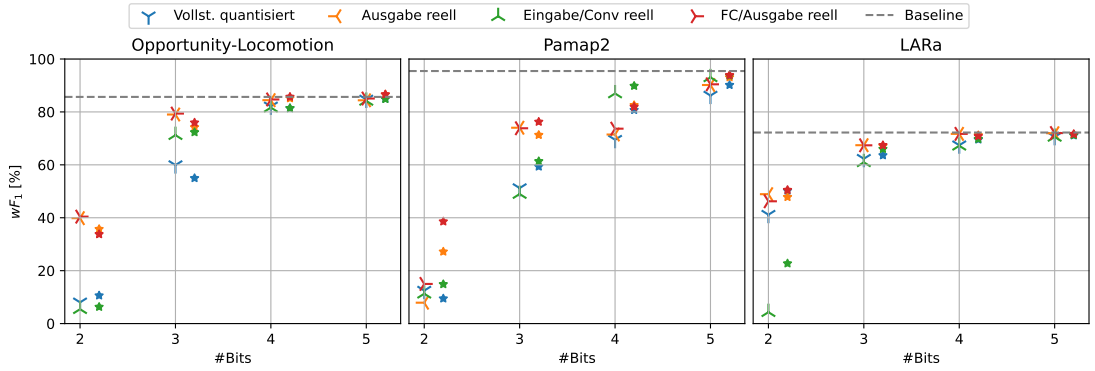

Abbildung 5.8.1: QAT-Vorteil gegenüber der PTQ bei ausgewählten Kombinationen der partiellen Quantisierung. Die Sterne markieren den durchschnittlichen wF_1-Score nach dem QAT.

5.9 ZUSAMMENFASSUNG UND EINORDNUNG DER ERGEBNISSE

Aus den Ergebnissen dieser Arbeit geht hervor, dass sich das CNN-IMU ohne weiteres auf sechs bis sieben Bits Ende-zu-Ende-Quantisieren lässt. Es konnte dabei kein Einfluss der verwendeten Kalibrierungsstatistiken (MinMax, MovingMinMax und Histogram) auf die Klassifikationsleistung festgestellt werden. Durch die symmetrische oder halbe Ausnutzung des Wertebereichs für die Parameter konnte ebenfalls keine Verbesserung festgestellt werden.

Ausgehend von einem vollständig quantisierten Modell ist es für niedrigere Auflösungen wichtig die Ausgabeschicht des CNN-IMUs nicht zu quantisieren, da so starke Leistungsverbesserungen bei praktisch keinen Kosten erzielt werden. Die Quantisierung der Faltungsschichten bringt eine starke Beschleunigung der Inferenzzeit mit sich, wobei der theoretische Speicherbedarf und die Klassifikationsleistung leicht verringert werden. Durch die Quantisierung der inneren linearen Schichten entsteht nahezu kein Leistungsverlust, jedoch wird Inferenzzeit eingespart. Bezüglich der theoretischen Speichernutzung ist es wichtig die linearen Schichten jedes IMU-Zweigs zu quantisieren, da dort der Großteil aller Parameter liegt.

6 Abbildung A.5.8, Abbildung A.5.9,Abbildung A.5.10

Durch das QAT konnten teilweise Leistungsverbesserungen für niedrigere Auflösungen erzielt werden. Allerdings lassen die Ergebnisse vermuten, dass das Training stark von dem Trainingserfolg des nicht quantisierten Modells abhängt.

Für einen Vergleich mit den verwandten Arbeiten wird eine Auswahl repräsentativer quantisierter Modelle zusammengestellt. Es werden geeignete Modelle für drei, fünf und sieben Bits gewählt. Im Folgenden werden sie mit CNN-IMU-Q3 CNN-IMU-Q5 und CNN-IMU-Q7 bezeichnet. Das CNN-IMU-Q7 ist vollständig quantisiert und wurde mit der MinMax-Statistik für die Parameter und der MovingAverageMinMax-Statistik für die Aktivierungen kalibriert. Das CNN-IMU-Q5 wurde mit denselben Statistiken kalibriert, die Ausgabeschicht wird jedoch nicht quantisiert. Das CNN-IMU-Q3 benutzt ebenfalls die Statistiken der Vorgänger, quantisiert jedoch neben der Ausgabeschicht auch die Faltungsblöcke nicht, um den Verlust bei einer Auflösung von drei Bits zu kompensieren. Zudem wurde die Leistung des CNN-IMU-Q3 durch QAT verbessert.

Modell	Datensatz	wF_1 [%]	Acc [%]	Z+	S+	BT [ms]	Sp. [MB]
B-BLSTM-RNN [EK16]	Opportunity	-	76	4,03	-	-	-
	Pamap2	-	92	4,03	-	-	-
DFTerNet [Yan+18]	Opportunity	90,9	-	~ 4	~ 11	-	0,034
	Pamap2	90,1	-	~ 4	~ 11	-	0,017
LP-LSTM [MRM20]	Pamap2	-	87,9	1,8	-	-	-
CNN-IMU	LARa	72,21	74,39	-	-	226,75 ± 1,78	104,82
	Opportunity	85,67	85,85	-	-	217,84 ± 8,06	62,02
	Pamap2	95,45	95,13	-	-	293,64 ± 4,58	136,29
CNN-IMU-Q7	LARa	71,91	73,00	2,38	4,57	95,37 ± 0,58	22,93
	Opportunity	85,68	85,88	2,16	4,6	100,79 ± 3,21	13,57
	Pamap2	95,42	95,13	2,28	4,57	129,00 ± 7,05	29,81
CNN-IMU-Q5	LARa	71,95	73,01	2,37	6,4	95,76 ± 0,56	16,39
	Opportunity	84,93	84,65	2,15	6,4	101,15 ± 3,27	9,69
	Pamap2	90,17	89,32	2,29	6,4	127,95 ± 2,20	21,31
CNN-IMU-Q3	LARa	71,81	73,86	1,04	9,56	218,77 ± 1,46	10,97
	Opportunity	86,27	86,42	1,02	6,15	214,57 ± 7,72	10,09
	Pamap2	91,60	91,42	1,03	9,95	283,32 ± 5,20	13,7

Tabelle 5.9.1: Gegenüberstellung der Ergebnisse mit eingeschränkter Vergleichbarkeit. Die roten Spalten beinhalten die relativen Verbesserungen durch die Quantisierung des jeweiligen Modells.

In Tabelle 5.9.1 sind die Ergebnisse der ausgewählten Modelle inklusive der Baseline (CNN-IMU) gelistet. Die Spalten Z+ und S+ stehen entsprechend für den Inferenzzeitgewinn und die Speichereinsparung, beide werden als Faktor angegeben. In der BT

Spalte stehen die Mittelwerte und Standardabweichungen der gemessenen Inferenzzeiten für ein Batch der Größe 128. Die Sp.-Spalte gibt den theoretischen Speicherbedarf (Gleichung 5.1.7) des Modells an. Die hervorgehobenen Zellen beinhalten jeweils den besten Wert der Spalte für die einzelnen Datensätze.

Die Vergleiche sind nur eingeschränkt möglich, weil die Datensätze unterschiedlich benutzt wurden. Für das B-BLSTM-RNN [EK16] wurden alle Klassen des Pamap2- und Opportunity-Datensatzes genutzt. Die Autoren haben eigene Aufteilungen für Training-, Validierungs- und Testmengen benutzt. Es wurde die Aufteilung C2 für den Vergleich ausgewählt, weil sie am nächsten an der Aufteilung in dieser Arbeit liegt. Für das DFTerNet [Yan+18] wurden ähnlich wie in dieser Arbeit die Bewegungsklassen des Opportunity-Datensatzes und die Aktivitätsklassen des Pamap2-Datensatzes verwendet. In dieser Arbeit wurde lediglich die Klasse *Bügeln* (Pamap2) nicht betrachtet. Unter LP-LSTM wird hier das Modell von Mazumder, Rashid und Mohsenin [MRM20] verstanden. Ihre Experimente fanden auf dem gesamten Pamap2-Datensatz, mit einer unbekannten Aufteilung (80/10/10) statt.

Die mittlere (rote) Spalte beinhaltet Vergleiche, welche größtenteils datensatz- und hardwareunabhängig sind, da nur eine relative Verbesserung durch die Quantisierung angegeben wird. Es ist zu erkennen, dass der Inferenzzeitgewinn der beiden CNN-IMUs (Q5 und Q7) mit den Ergebnissen der verwandten Arbeiten mithalten kann. Im Vergleich zu den stark quantisierten Modellen ist der theoretische Speicherbedarf deutlich größer, allerdings konnte mit dem CNN-IMU-Q3 durch die Quantisierung ähnlich viel Speicher eingespart werden. Die Inferenzzeit hat sich jedoch durch die ausschließliche Quantisierung der inneren linearen Schichten kaum verbessert. Je nachdem, welche Quantisierungskonfiguration für das CNN-IMU benutzt wird, kann eine vergleichbare Inferenzzeit- oder Speichereinsparung erzielt werden.

Aufgrund der unterschiedlichen Verwendungen der Datensätze sind die Vergleiche der Klassifikationsleistungen nicht ohne Weiteres möglich. Es deutet sich allerdings an, dass die betrachteten quantisierten CNN-IMUs aufgrund einer Auflösung von mehr als zwei Bits auf Kosten der Speichergröße und Inferenzzeit etwas bessere Klassifikationsleistungen erzielen.

6 FAZIT

In dieser Arbeit wurde die Quantisierung von CNNs für die IMU-basierte Aktivitätserkennung untersucht. Als Modell wurde auf das von Moya Rueda et al. [MR+18] vorgestellte CNN-IMU zurückgegriffen. Dieses wurde in den Experimenten mit affinen Quantisierungsfunktionen bei Auflösungen von zwei bis sieben Bits quantisiert. Dabei wurden verschiedene Verfahren zur Kalibrierung der Quantisierungsfunktionen evaluiert. Zudem wurde der Effekt der partiellen Quantisierung des CNN-IMUs untersucht. Die Quantisierung wurde mittels statischer Post-Training-Quantization und dem Quantization-Aware-Training durchgeführt.

Durch die partielle Quantisierung des CNN-IMUs konnten Subnetze im CNN-IMU identifiziert werden, welche robuster gegenüber der Quantisierung sind als andere Subnetze. Insbesondere die Ausgabeschicht hat sich als besonders sensibel für Auflösungen mit weniger als sechs Bits herausgestellt. Die inneren linearen Schichten waren im Gegensatz dazu sehr robust gegenüber der Quantisierung bei niedrigeren Auflösungen. Die Quantisierung der Faltungsschichten hat einen starken Inferenzzeitgewinn auf Kosten der Klassifikationsleistung erzielen können.

In dieser Arbeit konnten keine konsistenten Verbesserungen durch das QAT erzielt werden, allerdings konnten in einzelnen Fällen der Leistungsverlust erkennbar eingeschränkt werden. Besonders für den Pamap2-Datensatz [RS12] konnten so Verbesserungen festgestellt werden, was darauf hindeutet, dass ein erfolgreiches Training des reellen Modells ausschlaggebend für den Lernerfolg im QAT ist.

Um die besten Ergebnisse dieser Arbeit mit den verwandten Arbeiten vergleichen zu können, wurden neben dem Logistik-Datensatz LARa [Nie+20] auch die zwei Benchmark-Datensätze Pamap2 [RS12] und Opportunity-Locomotion [Rog+10] verwendet. Insbesondere das CNN-IMU-Q7 konnte im Hinblick auf die Klassifikationsleistung vergleichbare Werte zu den verwandten Arbeiten erzielen. Die partiell quantisierten CNN-IMUs (Q3 und Q5) konnten bei einer niedrigeren Auflösung durch das Auslassen der Quantisierung für bestimmte Schichten immer noch zum reellen Modell vergleichbare Klassifikationsleistungen erzielen. Insgesamt konnte durch die Quantisierung eine Verbesserung der Inferenzzeit erzielt werden, welche vergleichbar mit den verwandten Ansätzen ist. Die Reduzierung des theoretischen Speicherbedarfs ist durch die etwas höhere Auflösung nicht so stark, wie etwa bei dem DFTerNet [Yan+18]. Im Vergleich zu den stark quantisierten Modellen aus den verwandten Arbeiten, stellen die in dieser Ar-

beit betrachteten Ansätze einen Kompromiss zwischen Klassifikationsleistungsverlust und Inferenzzeitgewinn dar.

Die Ansätze in dieser Arbeit haben ein breites Spektrum an Anpassungsmöglichkeit für die Quantisierung abgedeckt. Dabei hat insbesondere die partielle Quantisierung bei niedrigeren Auflösungen erfolgversprechende Ansätze geboten. Es wurden allerdings stets Konfigurationen mit einheitlichen Auflösungen betrachtet. Deshalb könnte sich das Mischen verschiedener Auflösungen im selben Netz als interessanter Ansatz für weitere Untersuchungen herausstellen. Gerade die Faltungsschichten und die Ausgabeschicht können hier bei sechs bis sieben Bits quantisiert werden, wobei die robusten linearen Schichten stärker quantisiert werden können. Des Weiteren könnte die Binarisierung der inneren linearen Schichten zu interessanten Ergebnissen führen.

Da sich die Softwarelandschaft für die Quantisierung von KNNs noch in der Entwicklungsphase befindet, lohnt es sich die diskutierten Ansätze zu einem späteren Zeitpunkt mit effizienteren oder umfangreicheren Implementationen zu erkunden. Gerade die dynamische Quantisierung könnte weitere Möglichkeiten zur Eingrenzung des Klassifikationsleistungsverlustes bieten.

A

ANHANG

Top #	Opportunity-Locomotion	Pamap2	LARa
1	A: MinMax P: MinMax B: halb G: Kanal	A: MovingMinMax P: MinMax B: voll G: Tensor	A: MovingMinMax P: MinMax B: symm. G: Tensor
2	A: MinMax P: MinMax B: symm. G: Kanal	A: Histogram P: MinMax B: voll G: Tensor	A: Histogram P: MinMax B: symm. G: Tensor
3	A: MinMax P: MinMax B: symm. G: Tensor	A: Histogram P: MinMax B: symm. G: Kanal	A: MovingMinMax P: MinMax B: symm. G: Kanal
4	A: MinMax P: MinMax B: halb G: Tensor	A: Histogram P: MinMax B: halb G: Tensor	A: MinMax P: MinMax B: voll G: Tensor
5	A: MinMax P: MinMax B: voll G: Kanal	A: MovingMinMax P: MinMax B: halb G: Tensor	A: MinMax P: MinMax B: symm. G: Tensor
6	A: MinMax P: MinMax B: voll G: Tensor	A: Histogram P: MinMax B: voll G: Kanal	A: Histogram P: MinMax B: voll G: Tensor
7	A: Histogram P: MinMax B: voll G: Kanal	A: Histogram P: MinMax B: halb G: Kanal	A: MovingMinMax P: MinMax B: halb G: Tensor
8	A: MovingMinMax P: MinMax B: symm. G: Tensor	A: MovingMinMax P: MinMax B: symm. G: Tensor	A: MovingMinMax P: MinMax B: voll G: Kanal
9	A: Histogram P: MinMax B: symm. G: Kanal	A: MovingMinMax P: MinMax B: halb G: Kanal	A: MovingMinMax P: MinMax B: voll G: Tensor
10	A: Histogram P: MinMax B: halb G: Tensor	A: MinMax P: MinMax B: halb G: Tensor	A: MinMax P: MinMax B: halb G: Tensor

Tabelle A.5.1: Die besten zehn Konfigurationen für die PTQ bei 7-Bit. (A = Aktivierung, P = Parameter, B = Parameter Wertebereich, G = Parameter Granularität)

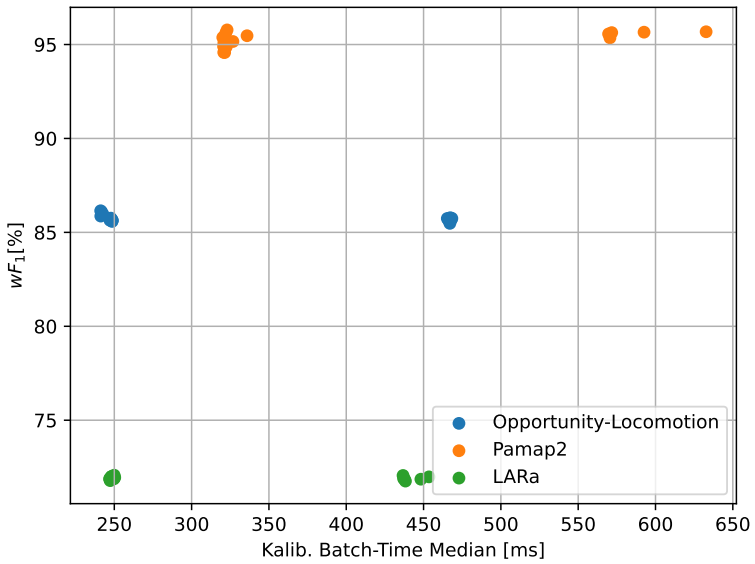

Abbildung A.5.1: Alle PTQ Konfigurationen als Punkt mit dem Median der Batch-Time und dem wF_1-Score als Dimensionen. Die weiter rechts liegenden Gruppierungen stammen von den Experimenten mit Histogram-Kalibrierung.

#Bits	Top 1 (LARa)	Top 2 (LARa)	Top 3 (LARa)	Top 4 (LARa)
7	A: MovingMinMax P: MinMax G: Kanal	A: Histogram P: MinMax G: Tensor	A: MovingMinMax P: MinMax G: Tensor	A: Histogram P: MinMax G: Kanal
6	A: MovingMinMax P: MinMax G: Tensor	A: MovingMinMax P: MinMax G: Kanal	A: Histogram P: MinMax G: Kanal	A: Histogram P: MinMax G: Tensor
5	A: MovingMinMax P: MinMax G: Kanal	A: MovingMinMax P: MinMax G: Tensor	A: Histogram P: MinMax G: Tensor	A: Histogram P: MinMax G: Kanal
4	A: MovingMinMax P: MinMax G: Kanal	A: MovingMinMax P: MinMax G: Tensor	A: Histogram P: MinMax G: Tensor	A: Histogram P: MinMax G: Kanal
3	A: Histogram P: MinMax G: Tensor	A: MovingMinMax P: MinMax G: Tensor	A: MovingMinMax P: MinMax G: Kanal	A: Histogram P: MinMax G: Kanal
2	A: MovingMinMax P: MinMax G: Kanal	A: Histogram P: MinMax G: Kanal	A: Histogram P: MinMax G: Tensor	A: MovingMinMax P: MinMax G: Tensor

Tabelle A.5.2: Die besten vier Konfigurationen für die PTQ mit dem LARa-Datensatz. (A = Aktivierung, P = Parameter, G = Parameter Granularität)

#Bits	Top 1 (Opp.-Loc.)	Top 2 (Opp.-Loc.)	Top 3 (Opp.-Loc.)	Top 4 (Opp.-Loc.)
7	A: MovingMinMax P: MinMax G: Kanal	A: Histogram P: MinMax G: Tensor	A: Histogram P: MinMax G: Kanal	A: MovingMinMax P: MinMax G: Tensor
6	A: MovingMinMax P: MinMax G: Tensor	A: MovingMinMax P: MinMax G: Kanal	A: Histogram P: MinMax G: Tensor	A: Histogram P: MinMax G: Kanal
5	A: MovingMinMax P: MinMax G: Tensor	A: MovingMinMax P: MinMax G: Kanal	A: Histogram P: MinMax G: Kanal	A: Histogram P: MinMax G: Tensor
4	A: MovingMinMax P: MinMax G: Kanal	A: MovingMinMax P: MinMax G: Tensor	A: Histogram P: MinMax G: Kanal	A: Histogram P: MinMax G: Tensor
3	A: MovingMinMax P: MinMax G: Tensor	A: MovingMinMax P: MinMax G: Kanal	A: Histogram P: MinMax G: Tensor	A: Histogram P: MinMax G: Kanal
2	A: MovingMinMax P: MinMax G: Kanal	A: Histogram P: MinMax G: Kanal	A: Histogram P: MinMax G: Tensor	A: MovingMinMax P: MinMax G: Tensor

Tabelle A.5.3: Die besten vier Konfigurationen für die PTQ mit dem Oportunity-Locomotion-Datensatz. (A = Aktivierung, P = Parameter, G = Parameter Granularität)

#Bits	Top 1 (Pamap2)	Top 2 (Pamap2)	Top 3 (Pamap2)	Top 4 (Pamap2)
7	A: Histogram P: MinMax G: Tensor	A: MovingMinMax P: MinMax G: Tensor	A: Histogram P: MinMax G: Kanal	A: MovingMinMax P: MinMax G: Kanal
6	A: Histogram P: MinMax G: Kanal	A: Histogram P: MinMax G: Tensor	A: MovingMinMax P: MinMax G: Tensor	A: MovingMinMax P: MinMax G: Kanal
5	A: Histogram P: MinMax G: Kanal	A: Histogram P: MinMax G: Tensor	A: MovingMinMax P: MinMax G: Tensor	A: MovingMinMax P: MinMax G: Kanal
4	A: MovingMinMax P: MinMax G: Tensor	A: Histogram P: MinMax G: Tensor	A: Histogram P: MinMax G: Kanal	A: MovingMinMax P: MinMax G: Kanal
3	A: Histogram P: MinMax G: Kanal	A: MovingMinMax P: MinMax G: Kanal	A: Histogram P: MinMax G: Tensor	A: MovingMinMax P: MinMax G: Tensor
2	A: MovingMinMax P: MinMax G: Kanal	A: Histogram P: MinMax G: Kanal	A: Histogram P: MinMax G: Tensor	A: MovingMinMax P: MinMax G: Tensor

Tabelle A.5.4: Die besten vier Konfigurationen für die PTQ mit dem Pamap2-Datensatz. (A = Aktivierung, P = Parameter, G = Parameter Granularität)

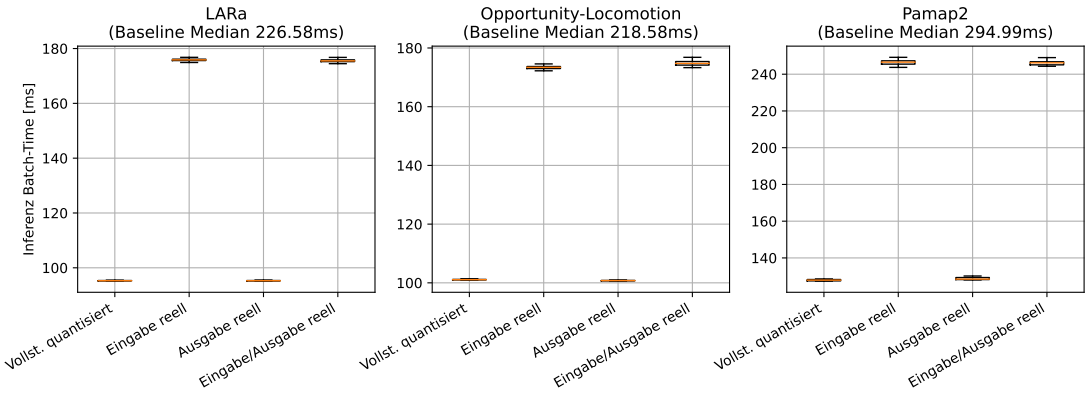

Abbildung A.5.2: Inferenzzeiten ausgewählter Kombinationen der partiellen Quantisierung.

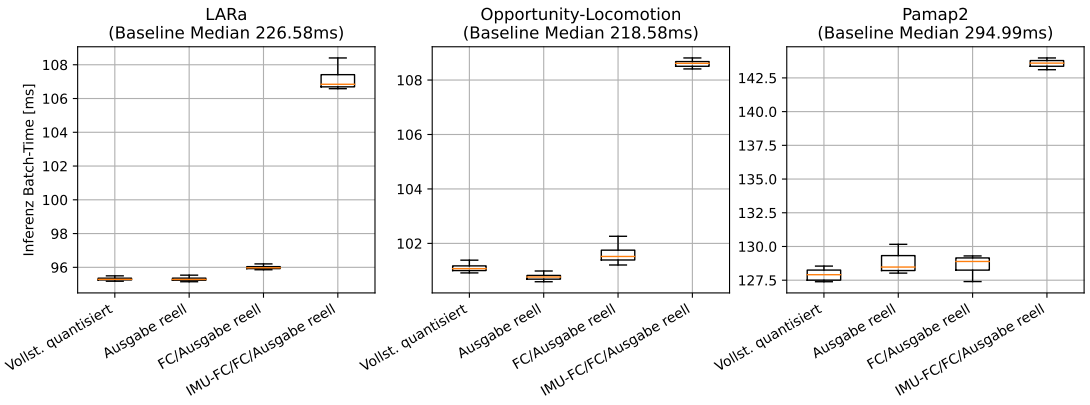

Abbildung A.5.3: Inferenzzeiten ausgewählter Kombinationen der partiellen Quantisierung.

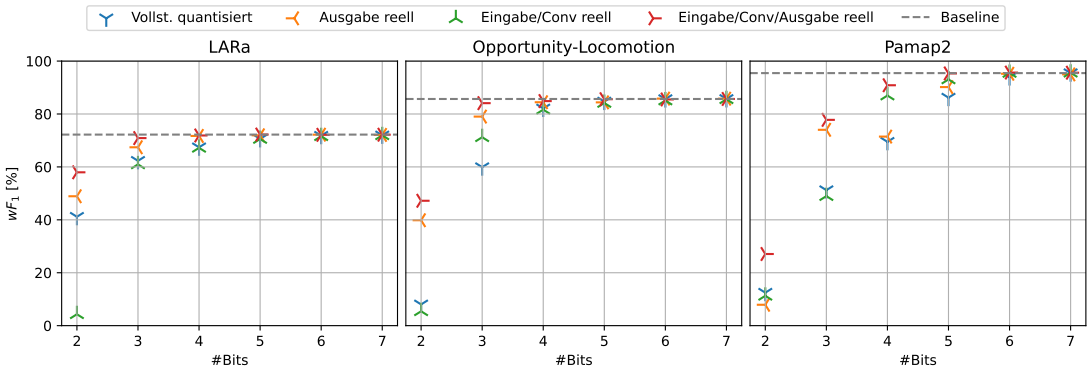

Abbildung A.5.4: Klassifikationsleistungen ausgewählter Kombinationen der partiellen Quantisierung.

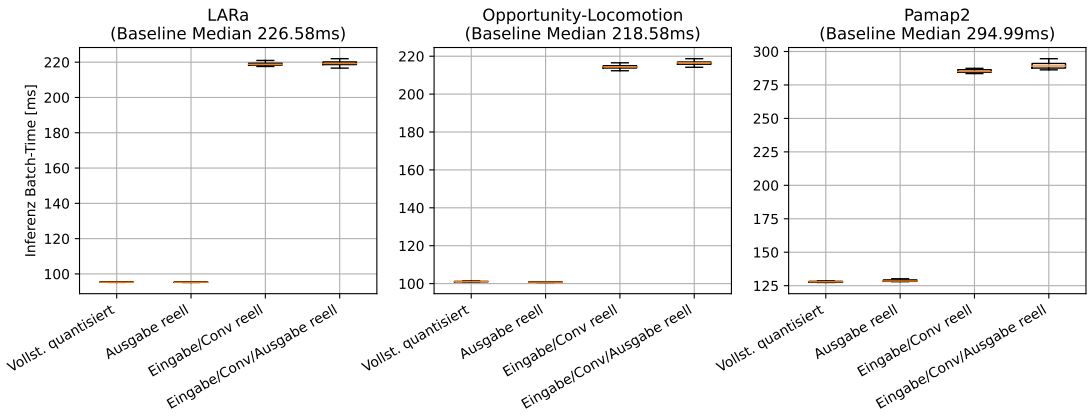

Abbildung A.5.5: Inferenzzeiten ausgewählter Kombinationen der partiellen Quantisierung.

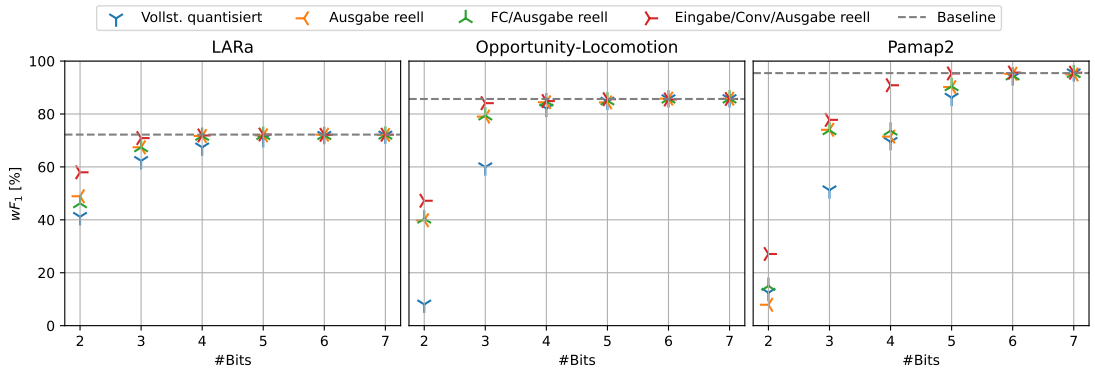

Abbildung A.5.6: Klassifikationsleistungen ausgewählter Kombinationen der partiellen Quantisierung.

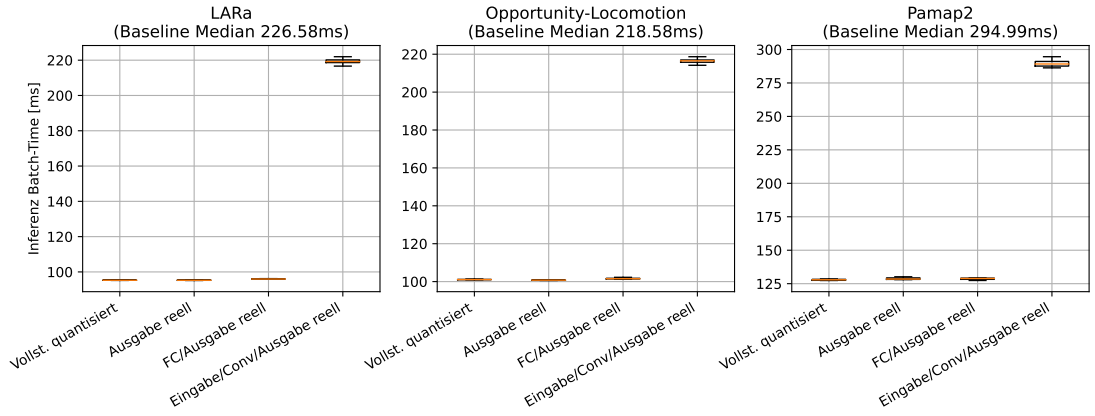

Abbildung A.5.7: Inferenzzeiten ausgewählter Kombinationen der partiellen Quantisierung.

#Bits	Voll quantisiert	Eingabe reell	Ausgabe reell	Eingabe+ Ausgabe reell	FC+Ausgabe reell	IMU-FC+FC+ Ausgabe reell	Conv reell	Conv+ Ausgabe reell
7	L: 22.93MB O: 13.57MB P: 29.81MB	L: 23.26MB O: 14.81MB P: 30.08MB	L: 22.94MB O: 13.57MB P: 29.83MB	L: 23.27MB O: 14.81MB P: 30.09MB	L: 27.04MB O: 17.46MB P: 33.11MB	L: 103.85MB O: 58.34MB P: 135.52MB	L: 23.90MB O: 17.25MB P: 30.59MB	L: 23.91MB O: 17.25MB P: 30.61MB
6	L: 19.65MB O: 11.63MB P: 25.55MB	L: 19.99MB O: 12.92MB P: 25.83MB	L: 19.67MB O: 11.63MB P: 25.57MB	L: 20.01MB O: 12.92MB P: 25.85MB	L: 23.93MB O: 15.68MB P: 28.98MB	L: 103.81MB O: 58.19MB P: 135.49MB	L: 20.66MB O: 15.46MB P: 26.36MB	L: 20.67MB O: 15.46MB P: 26.38MB
5	L: 16.38MB O: 9.69MB P: 21.30MB	L: 16.73MB O: 11.03MB P: 21.58MB	L: 16.39MB O: 9.69MB P: 21.31MB	L: 16.74MB O: 11.04MB P: 21.60MB	L: 20.82MB O: 13.90MB P: 24.86MB	L: 103.77MB O: 58.04MB P: 135.45MB	L: 17.42MB O: 13.67MB P: 22.13MB	L: 17.44MB O: 13.67MB P: 22.15MB
4	L: 13.10MB O: 7.75MB P: 17.04MB	L: 13.47MB O: 9.14MB P: 17.33MB	L: 13.12MB O: 7.76MB P: 17.06MB	L: 13.48MB O: 9.15MB P: 17.35MB	L: 17.71MB O: 12.12MB P: 20.73MB	L: 103.73MB O: 57.89MB P: 135.42MB	L: 14.19MB O: 11.88MB P: 17.90MB	L: 14.20MB O: 11.88MB P: 17.92MB
3	L: 9.83MB O: 5.81MB P: 12.78MB	L: 10.21MB O: 7.26MB P: 13.08MB	L: 9.84MB O: 5.82MB P: 12.80MB	L: 10.22MB O: 7.26MB P: 13.10MB	L: 14.59MB O: 10.33MB P: 16.60MB	L: 103.69MB O: 57.75MB P: 135.39MB	L: 10.95MB O: 10.09MB P: 13.68MB	L: 10.97MB O: 10.09MB P: 13.70MB
2	L: 6.55MB O: 3.88MB P: 8.52MB	L: 6.94MB O: 5.37MB P: 8.83MB	L: 6.57MB O: 3.88MB P: 8.54MB	L: 6.96MB O: 5.37MB P: 8.85MB	L: 11.48MB O: 8.55MB P: 12.47MB	L: 103.65MB O: 57.60MB P: 135.36MB	L: 7.71MB O: 8.29MB P: 9.45MB	L: 7.73MB O: 8.30MB P: 9.47MB

Tabelle A.5.5: Theoretischer Speicherbedarf ausgewählter Konfigurationen der partiellen Quantisierung. (L = LARa, O = Opportunity-Locomotion, P = Pamap2)

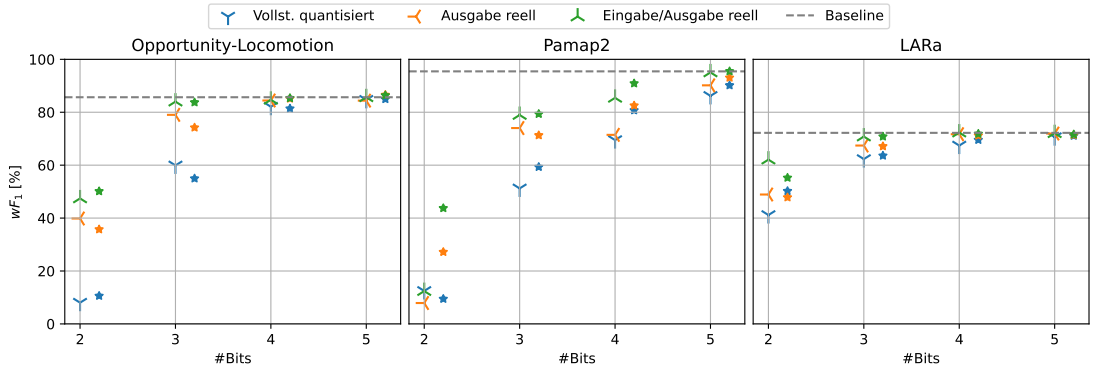

Abbildung A.5.8: QAT-Vorteil gegenüber der PTQ bei ausgewählten Kombinationen der partiellen Quantisierung. Die Sterne markieren den durchschnittlichen wF_1-Score nach dem QAT.

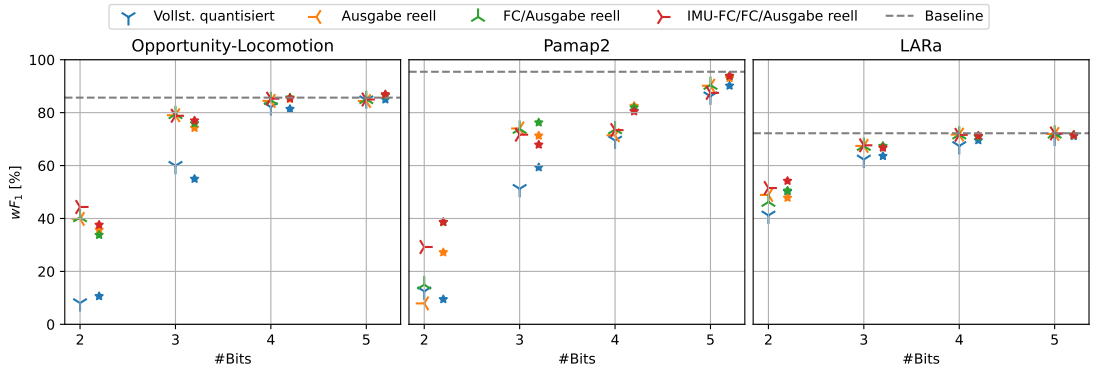

Abbildung A.5.9: QAT-Vorteil gegenüber der PTQ bei ausgewählten Kombinationen der partiellen Quantisierung. Die Sterne markieren den durchschnittlichen wF_1-Score nach dem QAT.

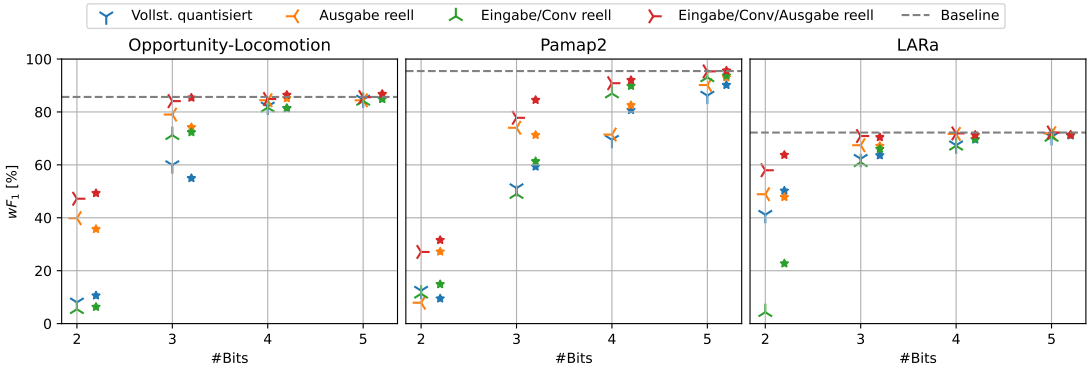

Abbildung A.5.10: QAT-Vorteil gegenüber der PTQ bei ausgewählten Kombinationen der partiellen Quantisierung. Die Sterne markieren den durchschnittlichen wF_1-Score nach dem QAT.

[Abi+19] Oludare Isaac Abiodun et al. „Comprehensive Review of Artificial Neural Network Applications to Pattern Recognition“. In: *IEEE Access* 7 (2019), S. 158820–158846. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2945545.

[ACB19] Devansh Arpit, Víctor Campos und Yoshua Bengio. „How to Initialize Your Network? Robust Initialization for WeightNorm & ResNets“. In: *Advances in Neural Information Processing Systems*. Bd. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper/2019/hash/e520f70ac3930490458892665cda6620-Abstract.html (besucht am 18. 03. 2023).

[AMA17] Saad Albawi, Tareq Abed Mohammed und Saad Al-Zawi. „Understanding of a Convolutional Neural Network“. In: *2017 International Conference on Engineering and Technology (ICET)*. Aug. 2017, S. 1–6. DOI: 10.1109/ICEngTechnol.2017.8308186.

[Ama93] Shun-ichi Amari. „Backpropagation and Stochastic Gradient Descent Method“. In: *Neurocomputing* 5.4 (Juni 1993), S. 185–196. ISSN: 0925-2312. DOI: 10.1016/0925-2312(93)90006-O. (Besucht am 22. 02. 2023).

[AY21] Isra'a AbdulNabi und Qussai Yaseen. „Spam Email Detection Using Deep Learning Techniques“. In: *Procedia Computer Science*. The 12th International Conference on Ambient Systems, Networks and Technologies (ANT) / The 4th International Conference on Emerging Data and Industry 4.0 (EDI40) / Affiliated Workshops 184 (Jan. 2021), S. 853–858. ISSN: 1877-0509. DOI: 10.1016/j.procs.2021.03.107. (Besucht am 17. 02. 2023).

[BLC13] Yoshua Bengio, Nicholas Léonard und Aaron Courville. *Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation*. Aug. 2013. DOI: 10.48550/arXiv.1308.3432. arXiv: arXiv:1308.3432. (Besucht am 10. 03. 2023).

[Cho+19] Yoni Choukroun et al. „Low-Bit Quantization of Neural Networks for Efficient Inference“. In: *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*. Okt. 2019, S. 3009–3018. DOI: 10.1109/ICCVW.2019.00363.

[CS97] Patricia S. Churchland und Terrence J. Sejnowski. „Grundlagen zur Neuroinformatik und Neurobiologie“. In: Wiesbaden: Vieweg+Teubner Verlag, 1997. ISBN: 978-3-322-86822-0 978-3-322-86821-3. DOI: 10.1007/978-3-322-86821-3. (Besucht am 16. 02. 2023).

[DV18] Vincent Dumoulin und Francesco Visin. *A Guide to Convolution Arithmetic for Deep Learning*. Jan. 2018. DOI: 10.48550/arXiv.1603.07285. arXiv: arXiv:1603.07285. (Besucht am 23. 02. 2023).

[DWL18] Marat Dukhan, Yiming Wu und Hao Lu. *QNNPACK: Open Source Library for Optimized Mobile Deep Learning*. Okt. 2018. URL: https://engineering.fb.com/2018/10/29/ml-applications/qnnpack/ (besucht am 15. 03. 2023).

[EK16] Marcus Edel und Enrico Köppe. „Binarized-BLSTM-RNN Based Human Activity Recognition“. In: *2016 International Conference on Indoor Positioning and Indoor Navigation (IPIN)*. 2016, S. 1–7. DOI: 10.1109/IPIN.2016.7743581.

[GBV20] Margherita Grandini, Enrico Bagli und Giorgio Visani. *Metrics for Multi-Class Classification: An Overview*. 2020. DOI: 10.48550/ARXIV.2008.05756.

[Gho+21] Amir Gholami et al. *A Survey of Quantization Methods for Efficient Neural Network Inference*. 2021. DOI: 10.48550/ARXIV.2103.13630.

[GSC99] F.A. Gers, J. Schmidhuber und F. Cummins. „Learning to Forget: Continual Prediction with LSTM“. In: *1999 Ninth International Conference on Artificial Neural Networks ICANN 99. (Conf. Publ. No. 470)*. Bd. 2. Sep. 1999, 850–855 vol.2. DOI: 10.1049/cp:19991218.

[Gu+18] Jiuxiang Gu et al. „Recent Advances in Convolutional Neural Networks“. In: *Pattern Recognition* 77 (Mai 2018), S. 354–377. ISSN: 0031-3203. DOI: 10.1016/j.patcog.2017.10.013. (Besucht am 23. 02. 2023).

[He+16] Kaiming He et al. „Deep Residual Learning for Image Recognition“. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, S. 770–778. URL: https://openaccess.thecvf.com/content_cvpr_2016/html/He_Deep_Residual_Learning_CVPR_2016_paper.html (besucht am 24. 02. 2023).

[HS97] Sepp Hochreiter und Jürgen Schmidhuber. „Long Short-Term Memory“. In: *Neural Computation* 9.8 (Nov. 1997), S. 1735–1780. ISSN: 0899-7667, 1530-888X. DOI: 10.1162/neco.1997.9.8.1735. (Besucht am 08. 03. 2023).

[Iee] „IEEE Standard for Floating-Point Arithmetic“. In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (Juli 2019), S. 1–84. DOI: 10.1109/IEEESTD.2019.8766229.

[IS15] Sergey Ioffe und Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. März 2015. DOI: 10.48550/arXiv.1502.03167. arXiv: arXiv:1502.03167. (Besucht am 08. 03. 2023).

[Jac+17] Benoit Jacob et al. *Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference*. Dez. 2017. DOI: 10.48550/arXiv.1712.05877. arXiv: arXiv:1712.05877. (Besucht am 10. 03. 2023).

[Jar09] Quasar Jarosz. *Neuron Hand-tuned*. Aug. 2009. URL: https://de.wikipedia.org/wiki/Datei:Neuron_Hand-tuned.svg (besucht am 16. 02. 2023).

[JBD19] Charmi Jobanputra, Jatna Bavishi und Nishant Doshi. „Human Activity Recognition: A Survey“. In: *Procedia Computer Science* 155 (2019), S. 698–703. ISSN: 1877-0509. DOI: 10.1016/j.procs.2019.08.100.

[JM15] M. I. Jordan und T. M. Mitchell. „Machine Learning: Trends, Perspectives, and Prospects“. In: *Science* 349.6245 (Juli 2015), S. 255–260. DOI: 10.1126/science.aaa8415. (Besucht am 15. 02. 2023).

[KB17] Diederik P. Kingma und Jimmy Ba. *Adam: A Method for Stochastic Optimization*. Jan. 2017. DOI: 10.48550/arXiv.1412.6980. arXiv: arXiv:1412.6980. (Besucht am 22. 02. 2023).

[Khu+21] Daya Khudia et al. *FBGEMM: Enabling High-Performance Low-Precision Deep Learning Inference*. Jan. 2021. DOI: 10.48550/arXiv.2101.05615. arXiv: arXiv:2101.05615. (Besucht am 26. 01. 2023).

[Kri18] Raghuraman Krishnamoorthi. *Quantizing Deep Convolutional Networks for Efficient Inference: A Whitepaper*. Juni 2018. DOI: 10.48550/arXiv.1806.08342. arXiv: arXiv:1806.08342. (Besucht am 10. 03. 2023).

[Lec+98] Y. Lecun et al. „Gradient-Based Learning Applied to Document Recognition“. In: *Proceedings of the IEEE* 86.11 (Nov. 1998), S. 2278–2324. ISSN: 1558-2256. DOI: 10.1109/5.726791.

[LL13] Oscar D. Lara und Miguel A. Labrador. „A Survey on Human Activity Recognition Using Wearable Sensors“. In: *IEEE Communications Surveys & Tutorials* 15.3 (2013), S. 1192–1209. ISSN: 1553-877X. DOI: 10.1109/SURV.2012.110112.00192.

[MP43] Warren S. McCulloch und Walter Pitts. „A Logical Calculus of the Ideas Immanent in Nervous Activity“. In: *The bulletin of mathematical biophysics* 5.4 (Dez. 1943), S. 115–133. ISSN: 1522-9602. DOI: 10.1007/BF02478259. (Besucht am 27. 01. 2023).

[MR+18] Fernando Moya Rueda et al. „Convolutional Neural Networks for Human Activity Recognition Using Body-Worn Sensors“. In: *Informatics* 5.2 (2018). ISSN: 2227-9709. DOI: 10.3390/informatics5020026.

[MRM20] Arnab Neelim Mazumder, Hasib-Al Rashid und Tinoosh Mohsenin. „An Energy-Efficient Low Power LSTM Processor for Human Activity Monitoring“. In: *2020 IEEE 33rd International System-on-Chip Conference (SOCC)*. Sep. 2020, S. 54–59. DOI: 10.1109/SOCC49529.2020.9524796.

[Nie+20] Friedrich Niemann et al. „LARa: Creating a Dataset for Human Activity Recognition in Logistics Using Semantic Attributes“. In: *Sensors* 20.15 (2020). ISSN: 1424-8220. DOI: 10.3390/s20154083.

[Nwa+18] Chigozie Nwankpa et al. *Activation Functions: Comparison of Trends in Practice and Research for Deep Learning*. Nov. 2018. DOI: 10.48550/arXiv.1811.03378. arXiv: arXiv:1811.03378. (Besucht am 28. 02. 2023).

[ON15] Keiron O'Shea und Ryan Nash. *An Introduction to Convolutional Neural Networks*. Dez. 2015. DOI: 10.48550/arXiv.1511.08458. arXiv: arXiv:1511.08458. (Besucht am 23. 02. 2023).

[Pas+19] Adam Paszke et al. „PyTorch: An Imperative Style, High-Performance Deep Learning Library“. In: *Advances in Neural Information Processing Systems 32*. Hrsg. von H. Wallach et al. Curran Associates, Inc., 2019, S. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[PPA18] Antonio Polino, Razvan Pascanu und Dan Alistarh. *Model Compression via Distillation and Quantization*. Feb. 2018. DOI: 10.48550/arXiv.1802.05668. arXiv: arXiv:1802.05668. (Besucht am 25. 02. 2023).

[Quaa] *Quantization API Observer — PyTorch 1.13 Documentation*. URL: https://pytorch.org/docs/stable/quantization-support.html#torch-ao-quantization-observer (besucht am 14. 03. 2023).

[Quab] *Quantization Backend Support — PyTorch 1.13 Documentation*. URL: https://pytorch.org/docs/stable/quantization.html#backend-hardware-support (besucht am 14. 03. 2023).

[Quac] *Quantization — PyTorch 1.13 Documentation*. URL: https://pytorch.org/docs/stable/quantization.html (besucht am 14. 03. 2023).

[Rog+10] Daniel Roggen et al. „Collecting Complex Activity Datasets in Highly Rich Networked Sensor Environments“. In: *2010 Seventh International Conference on Networked Sensing Systems (INSS)*. Juni 2010, S. 233–240. DOI: 10.1109/INSS.2010.5573462.

[Roj96] Raúl Rojas. *Neural Networks*. Berlin, Heidelberg: Springer, 1996. ISBN: 978-3-540-60505-8 978-3-642-61068-4. DOI: 10.1007/978-3-642-61068-4. (Besucht am 27. 01. 2023).

[Ros58] F. Rosenblatt. „The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain.“ In: *Psychological Review* 65.6 (1958), S. 386–408. ISSN: 1939-1471, 0033-295X. DOI: 10.1037/h0042519. (Besucht am 27. 01. 2023).

[Ros62] Frank Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan Books, 1962.

[RS12] Attila Reiss und Didier Stricker. „Introducing a New Benchmarked Dataset for Activity Monitoring“. In: *2012 16th International Symposium on Wearable Computers*. Juni 2012, S. 108–109. DOI: 10.1109/ISWC.2012.13.

[Rud17] Sebastian Ruder. *An Overview of Gradient Descent Optimization Algorithms*. Juni 2017. DOI: 10.48550/arXiv.1609.04747. arXiv: arXiv:1609.04747. (Besucht am 20. 02. 2023).

[Sil+18] David Silver et al. „A General Reinforcement Learning Algorithm That Masters Chess, Shogi, and Go through Self-Play“. In: *Science* 362.6419 (Dez. 2018), S. 1140–1144. DOI: 10.1126/science.aar6404. (Besucht am 17. 02. 2023).

[Spa] SparkFun. *9DoF-IMU-Breakout LSM9DS1 13284-02*. URL: https://www.sparkfun.com/products/retired/13284.

[Ten] *TensorFlow Lite 8-Bit Quantization Specification*. URL: https://www.tensorflow.org/lite/performance/quantization_spec (besucht am 09. 03. 2023).

[TM18] H. Taud und J.F. Mas. „Multilayer Perceptron (MLP)“. In: *Geomatic Approaches for Modeling Land Change Scenarios*. Hrsg. von María Teresa Camacho Olmedo et al. Lecture Notes in Geoinformation and Cartography. Cham: Springer International Publishing, 2018, S. 451–455. ISBN: 978-3-319-60801-3. DOI: 10.1007/978-3-319-60801-3_27. (Besucht am 10. 02. 2023).

[vKC20] Thomas van Klompenburg, Ayalew Kassahun und Cagatay Catal. „Crop Yield Prediction Using Machine Learning: A Systematic Literature Review“. In: *Computers and Electronics in Agriculture* 177 (Okt. 2020), S. 105709. ISSN: 0168-1699. DOI: 10.1016/j.compag.2020.105709. (Besucht am 17. 02. 2023).

[VNK15] Michalis Vrigkas, Christophoros Nikou und Ioannis A. Kakadiaris. „A Review of Human Activity Recognition Methods“. In: *Frontiers in Robotics and AI* 2 (2015). ISSN: 2296-9144. URL: https://www.frontiersin.org/articles/10.3389/frobt.2015.00028 (besucht am 26. 02. 2023).

[Wan+22] Qi Wang et al. „A Comprehensive Survey of Loss Functions in Machine Learning“. In: *Annals of Data Science* 9.2 (Apr. 2022), S. 187–212. ISSN: 2198-5812. DOI: 10.1007/s40745-020-00253-5. (Besucht am 15. 02. 2023).

[Wu+20] Hao Wu et al. *Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation*. Apr. 2020. DOI: 10.48550/arXiv.2004.09602. arXiv: arXiv:2004.09602. (Besucht am 10. 03. 2023).

[Wup89] Horst Wupper. *Einführung in Die Digitale Signalverarbeitung*. ELTEX: Studientexte Elektronik. Hüthig, 1989. ISBN: 978-3-7785-1442-9.

[Yan+18] Zhan Yang et al. „DFTerNet: Towards 2-Bit Dynamic Fusion Networks for Accurate Human Activity Recognition“. In: *IEEE Access* 6 (2018), S. 56750–56764. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2873315.

[Yin19] Xue Ying. „An Overview of Overfitting and Its Solutions“. In: *Journal of Physics: Conference Series* 1168.2 (Feb. 2019), S. 022022. ISSN: 1742-6596. DOI: 10.1088/1742-6596/1168/2/022022. (Besucht am 22. 02. 2023).