Hochschule für Technik und Wirtschaft Dresden
Fakultät Informatik/Mathematik

Long-Short-Term-Memory-Unterstützung für eingebettete Systeme

Bachelorarbeit

Autor:	Jakob Häcker
Matrikelnummer:	49687
Ort:	Dresden
Erstgutachter:	Prof. Dr.-Ing. Jörg Vogt
Zweitgutachter:	M. Sc. M. Sc. Pascal Pfeiffer
Abgabedatum:	23.08.2024

Abstract

Diese Arbeit beschäftigt sich mit der Implementierung einer Long-Short-Term-Memory-Unterstützung für das embedded AI Framework Emmi. Ziel ist es, das LSTM für Gleitkomma- (Float) und quantisierte neuronale Netze (NN) zu implementieren. Dabei wird die neuartige Quantisierungstechnik Dynamic Inference Quantization (DYINQ) verwendet. In systematischen Tests wird einerseits die Funktionsfähigkeit der Float-Implementierung nachgewiesen und andererseits die Fehleranfälligkeit der Implementierung für quantisierte Daten untersucht.

Inhaltsverzeichnis

Abkürzungsverzeichnis

BPTT	Back Propagation Through Time. 10
CEC	constant error carrousel. 11, 12
DYINQ	Dynamic Inference Quantization. 5, 18
FPU	Floating Point Unit. 5, 16
IPMS	Institut für Photonische Mikrosysteme. 1, 15
ISA	Befehlssatzarchitektur (engl. „Instruction Set Architecture“). 5
JAQ	Efficient Intege-Arithmetic-Only Inference von Benoit Jacob et. Al. 18
KI	künstliche Intelligenz. V, 1–3, 6, 7, 14, 16, 18, 19, 33
LSTM	Long Short-Term Memory. I, V, 1–3, 7, 10, 11, 13, 19–25, 27–30, 33–37
NN	neuronales Netz. 1, 8, 9, 11, 15, 28, 34
RNN	rekurrentes neuronales Netz. 11
TF	TensorFlow. 2, 3, 15, 17–19, 22–25, 27, 28, 35
TFU	Timesteps Features Units, bezeichnet in Kapitel 4 die Größe eines LSTMs, in dem die Anzahl der Timesteps, Features und Units gleich ist.. V, 30–33

Glossar

IP Core — (Intellectual Property Core) ist der Bauplan einer integrierten Schaltung in der Halbleiterindustrie. Es kann sich dabei z. B. um den Plan eines Prozessors, eines Speichermoduls oder anderer spezieller Funktionselemente wie z. B. Krypto-Module handeln. 1, 2, 35

Overflow — (zu Deutsch Überlauf), beschreibt das Phänomen, welches eintritt, wenn das Ergebnis einer arithmetischen Operation außerhalb des Wertebereiches einer Variablen liegt, in der das Ergebnis gespeichert wird. Dies führt zu unvorhersehbaren Verhalten des Programmes. V, 6, 29–35

Pass-By-Reference — bezeichnet eine Programmier Ansatz, bei dem einer Funktion ein Zeiger auf die Speicherstelle einer Variablen als Parameter übergeben wird. Dadurch ist es möglich den Wert einer Variablen in der aufgerufenen Funktion zu verändern und mehrer dieser Variablen als Ausgabe der Funktion zu deklarieren. 20

Abbildungsverzeichnis

Listings

1 Einleitung

Künstliche Intelligenz (KI) befindet sich derzeit in einer sehr dynamischen Entwicklungsphase und eröffnet sowohl im privaten als auch im gesellschaftlichen Bereich neue Möglichkeiten. Die Anwendungen reichen vom automatisierten Handel über die Diagnose von Krankheiten bis hin zur vorausschauenden Wartung von Industrieanlagen und einer optimierten Lagerhaltung zur Minimierung von Ressourcenverschwendung. [4]

Die treibenden Kräfte hinter diesem Fortschritt sind nicht nur die kontinuierliche Verbesserung der Algorithmen, sondern auch die stetige Weiterentwicklung der Hardware, auf der KI-Systeme implementiert werden. Diese Entwicklungen haben dazu geführt, dass KI-Anwendungen nicht mehr ausschließlich in großen Rechenzentren betrieben werden, sondern auch in kleineren, integrierten Geräten zum Einsatz kommen. Beispiele hierfür sind autonome Roboter, intelligente Haustechniksysteme und eingebettete Computer in einer Vielzahl von technischen Geräten. [12]

Diese lokalen KI-Systeme (auch als Edge AI bezeichnet) werden häufig mit Sensordaten oder Benutzereingaben gefüttert, die es ihnen ermöglichen, Vorhersagen zu treffen, um Entscheidungen zu treffen oder Handlungsempfehlungen zu geben.

Um die Integration von eingebetteten Systemen in Produkte verschiedener technischer Branchen zu erleichtern, hat das Fraunhofer Institut für Photonische Mikrosysteme den IP Core EMSA5 entwickelt. Der Befehlssatz des IP Cores basiert auf der RISC-V Architektur, die als Open-Source-Lösung besonders für die Entwicklung anpassungsfähiger Software geeignet ist.

Am Fraunhofer Institut für Photonische Mikrosysteme (IPMS) wurde das spezialisierte embedded AI Framework Emmi entwickelt, das für die Ausführung von neuronalen Netzen auf dem EMSA5 optimiert ist.

Ziel dieser Arbeit ist die Erweiterung des KI-Frameworks Emmi um eine LSTM-Funktionalität.

1.1 Motivation

Eingebettete Systeme, in die der EMSA5 integriert werden kann, werden in einer Vielzahl von Bereichen eingesetzt, in denen sie mit verschiedenen Arten von Daten

interagieren. Ein erheblicher Anteil dieser Daten ist zeitabhängig, was ihre Verarbeitung für viele Anwendungen unerlässlich macht. Diese Daten sind jedoch oft sehr komplex, da sie häufig von Rauschen überlagert sind und weit zurückliegende Ereignisse einen signifikanten Einfluss auf das aktuelle Systemverhalten haben können.

Die derzeit verfügbaren Funktionen des KI-Frameworks Emmi sind nicht in der Lage, solche komplexen, zeitabhängigen Muster adäquat zu verarbeiten.

Daher ist die Erweiterung um eine LSTM-Funktionalität von großer Bedeutung, um das Anwendungsspektrum des EMSA5 sowie des KI-Frameworks Emmi zu erweitern.

1.2 Anforderungen

Die zu entwickelnde Erweiterung muss folgende Kriterien erfüllen:

1. Neuronale Netze, die mit TensorFlow (TF) Version 2.15.0 entwickelt wurden und mindestens einen LSTM-Layer enthalten, müssen mit einem Modellkonverter in C-Code konvertierbar sein.
2. Die Erweiterung muss durch Unit- und Integrationstests auf Funktionalität geprüft sein.
3. Die Erweiterung muss auf dem IP Core EMSA5 lauffähig sein.

1.3 Forschungsfragen

Um die oben genannten Anforderungen zu erfüllen, müssen die folgenden Forschungsfragen beantwortet werden:

1. Wie funktioniert ein LSTM?
2. Wie kann ein LSTM im Emmi-Framework für Float-Werte implementiert werden?
3. Welche Anpassungen sind am bestehenden Modellkonverter notwendig, um TensorFlow (TF)-Modelle, die Long Short-Term Memory (LSTM)-Layer enthalten, in Emmi-kompatiblen Code zu übersetzen?
4. Welche Tests sind erforderlich, um die Funktionsfähigkeit der Erweiterung zu verifizieren?

5. Wie kann die LSTM Implementierung auf quantisierte Daten, mit einer gegbenen Quantisierungstechnik, adaptiert werden?

6. Ist die Implementierung praktisch anwendbar?

1.4 Inhaltsübersicht

Das folgende Kapitel gibt einen Überblick über die technischen und theoretischen Grundlagen der verwendeten Technologien: Zunächst erfolgt eine Einführung in die RISC-V Architektur, gefolgt von einer Erläuterung der verwendeten Quantisierungstechnik. Anschließend werden grundlegende Konzepte des maschinellen Lernens skizziert, die sich hier ausschließlich auf Inferenz konzentrieren, gefolgt von einer Betrachtung zu embedded AI sowie dem zu erweiternden Emmi-Framework und dessen Mathematikbibliothek. Das Kapitel schließt mit einer Einführung in das KI-Framework TF.

Im weiteren Verlauf der Arbeit wird detailliert auf die Implementierung der Float- und quantisierten Version des LSTM-Layers eingegangen. Außerdem werden die notwendigen Anpassungen des bestehenden Modellkonverters diskutiert.

Im vorletzten Kapitel werden die durchgeführten Tests und deren Ergebnisse vorgestellt, die die Funktionalität der Implementierungen untersuchen.

Abschließend werden die Kernaspekte der Arbeit zusammengefasst und ein Ausblick auf weiterführende Forschungsthemen gegeben.

2 Grundlagen

2.1 RISC-V

Die RISC-V Befehlssatzarchitektur, die im Jahr 2010 an der University of California, Berkeley entwickelt wurde, zählt zu den ersten Open-Source Befehlssatzarchitektur (engl. „Instruction Set Architecture“) (ISA). Diese ISA ist unter der Creative Commons Attribution 4.0 International License lizenziert, was ihre weite Verbreitung und flexible Anpassung ermöglicht. Ein für die RISC-V Architektur entworfener Chip beinhaltet standardmäßig 47 grundlegende Befehle für 32-Bit-Integer-Operationen, die im Befehlssatz „RV32I“ zusammengefasst sind. Durch zusätzliche Hardware-Erweiterungen wie z.B. eine Floating Point Unit (FPU) oder eine Einheit für vektorisierte Datenverarbeitung können Programme, die mit den entsprechenden Compiler-Flags kompiliert wurden, von diesen Erweiterungen profitieren und so ihre Leistung verbessern [2],[10].

2.2 Quantisierung

Der Begriff der Quantisierung bezeichnet den Prozess der Überführung kontinuierlicher Werte in eine diskrete Repräsentation. Es existieren verschiedene Ansätze, die Float-Werte eines spezifischen Wertebereichs F mithilfe von Zusatzinformationen auf ganzzahlige Werte (Integer) abbilden. Der in dieser Arbeit verwendete Algorithmus, bekannt als Dynamic Inference Quantization (DYINQ), wurde am Fraunhofer IPMS entwickelt [10]. Wie andere derartige Algorithmen nutzt auch dieser einen Skalierungsfaktor (engl. „scale“) s und einen Nullpunktversatz z (engl. „zero offset“), um Floats in einem b-Bit breiten Integer-Wert zu speichern.

Die Bitbreite definiert die Anzahl der äquidistanten Float-Werte, die innerhalb eines Intervalls gespeichert werden können. Der Skalierungsfaktor legt den Abstand zwischen diesen Werten fest. Der Nullpunktversatz verschiebt den möglicherweise nicht um den Nullpunkt zentrierten Float-Wertebereich auf den um den Nullpunkt zentrierten Integer-Bereich, der durch den Wertebereich des verwendeten Integer-Typs aufgespannt wird.

$$s = round\left(\frac{Q_{steps}}{max(F) - min(F)}\right) \tag{1}$$

$$z = Q_{min} - min(F) \cdot s \tag{2}$$

$$q = round(f \cdot s + z) \tag{3}$$

wobei

$$Q_{steps} = Q_{max} - Q_{min} = 2^b - 1 \tag{4}$$

$$Q_{max} = -2^{b-1} \tag{5}$$

$$Q_{min} = 2^{b-1} - 1 \tag{6}$$

Hier wird ein einzelner Float-Wert mit $f \in F$ bezeichnet und dessen quantisierte Entsprechung mit q (vgl. [10]).

Im Gegensatz zu anderen Quantisierungstechniken ist die Ermittlung der Metadaten, d. h. des Skalierungsfaktors und des Nullpunktversatzes, nicht bereits zum Zeitpunkt der Erstellung des neuronalen Netzes erforderlich, sondern kann während der Laufzeit berechnet werden. Diese Vorgehensweise reduziert den Aufwand für die Erstellung und Ausführung eines Modells mit quantisierten Werten, birgt jedoch ein erhöhtes Risiko für Overflows, da die Bitbreite der quantisierten Werte nach arithmetischen Operationen nicht a priori bekannt ist.

Bei der Berechnung binärer Operationen, wie beispielsweise der Matrixmultiplikation oder einer elementweisen Addition, mit quantisierten Werten ist eine Anpassung des Skalierungsfaktors und des Nullpunktversatzes der Operanden erforderlich. Des Weiteren ist die Bitbreite des Ergebnises, bei beispielsweise einer ganzzahligen Multiplikation mit quantisierten Operanden, immer größer oder gleich der maximalen Bitbreite der Operanden. Dieser Effekt, der wachsenden Bitbreite, kann zu Overflows führen.

2.3 Maschinelles Lernen

Ein wesentlicher Forschungsbereich der KI ist das maschinelle Lernen. Hierbei werden Algorithmen entwickelt, die darauf abzielen, Muster und Strukturen in Daten zu erkennen, ohne dass deren Merkmale explizit programmiert werden müssen. Der Prozess des selbstständigen Erlernens von Mustern wird als Training bezeichnet.

Die identifizierten Muster können im Anschluss verwendet werden, um unbekannte Daten zu klassifizieren oder Prognosen über deren Verlauf zu erstellen. Eine effektive Datenverarbeitung setzt voraus, dass die Algorithmen, welche auf die jeweilige Art der zu verarbeitenden Daten abgestimmt sind, in ein Modell zusammengefasst werden.

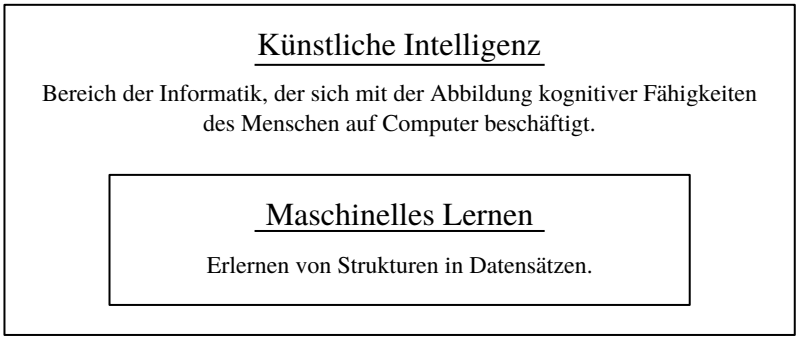

Abbildung 2.1: Hierarchie von KI und Maschinelles Lernen.

2.3.1 Neuronale Netze

Diese Arbeit befasst sich mit einer speziellen Klasse von Algorithmen, die biologische Neuronen nachahmen.

Ein künstliches Neuron wird in der vorliegenden Untersuchung wie folgt definiert: Analog zum biologischen Vorbild ist ein künstliches Neuron mit mehreren Eingängen (Synapsen) ausgestattet. Die Stärke der Eingangssignale wird durch die Gewichte modifiziert, welche während des Trainingsprozesses adaptiert werden. Die Generierung eines Ausgangssignals erfolgt durch eine Modifikation der Summe der gewichteten Eingänge mittels einer Aktivierungsfunktion, die hier mit f bezeichnet wird. Diese Funktion modifiziert das lineare Eingangssignal auf nichtlineare Weise, wodurch das Modell in der Lage ist, nichtlineare Zusammenhänge zu erlernen.

In dieser Arbeit spielen insbesondere die Sigmoid- und Tangenshyperbolicus-Funktionen eine wichtige Rolle, da diese im LSTM verwendet werden.

$$f\left(\sum w_i x_i\right) = y \tag{7}$$

Ein **Perzeptron** erweitert das Neuron um einen festen Eingabewert, den sogenannten Bias b, welcher eine Verschiebung der Aktivierungsfunktion bewirkt. Dies

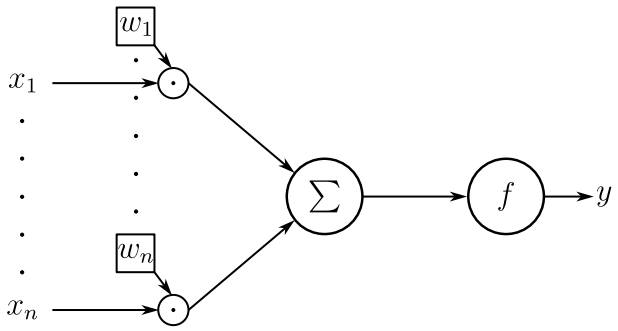

Abbildung 2.2: Ein Neuron mit x als Eingabe und w als Aktivierungsfunktion.

ermöglicht die Handhabung von nicht zentrierten Ausgabedaten, die in einer Vielzahl von Anwendungsfällen erforderlich sind.

$$f\left(\sum w_i x_i + b_i\right) = y \tag{8}$$

Ein **neuronales Netz (NN)** entsteht, wenn mehrere Neuronen in Layern, im deutschen Sprachraum auch als Schichten bezeichnet, angeordnet werden [8].

Der erste Layer, auch als Input-Layer bzw. Eingabeschicht bezeichnet, dient der Aufnahme initialer Daten, wobei deren Form festgelegt wird, ohne dass eine Veränderung der Daten selbst erfolgt. Dies gewährleistet, dass ausschließlich Daten mit den korrekten Dimensionen in das Netzwerk gelangen.

Die folgende Schicht, der (im Plural die) Hidden-Layer, und die letzte Schicht, der Output-Layer, übernehmen die schrittweise Weiterverarbeitung der Daten.

Die Anordnung der Neuronen in Layern erlaubt die Darstellung der Verarbeitung als Matrixmultiplikation. Dabei werden die Eingabe, der Bias und die Ausgabe als Vektoren definiert, während die Gewichte in einer Matrix zusammengefasst werden. Die Aktivierungsfunktionen der Neuronen eines Layers müssen identisch sein, da sie auf alle Elemente des resultierenden Vektors angewendet werden.

$$y = f(W \cdot x + b) \tag{9}$$

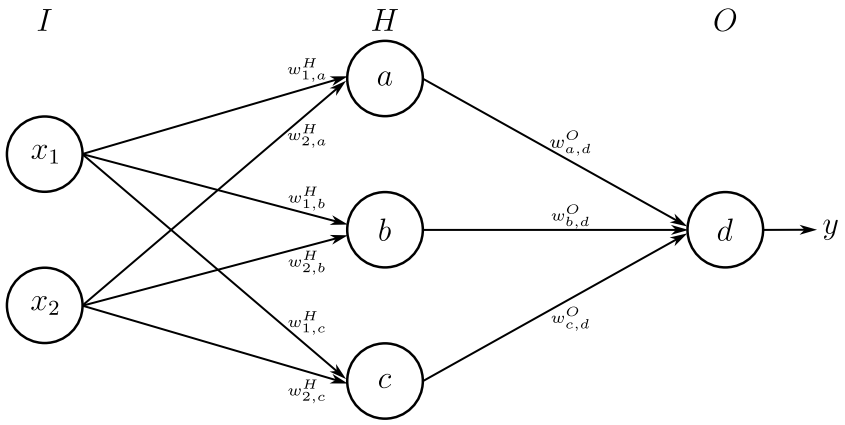

Abbildung 2.3: Ein neuronales Netz mit einem Input-Vektor $x = (x_1, x_2)^T$, einem Skalar y als Ausgabe, drei Neuronen im Hidden-Layer und einem Neuron im Output-Layer sowie den Gewichten W_H und W_O

Komplexere neuronale Netze können auch in nicht-linear angeordneten Layer strukturiert sein. Dabei wird die Eingabe auf mehrere quasi-parallele Layer aufgeteilt, deren Ausgaben zu einem späteren Zeitpunkt durch Konkatenation zusammengeführt werden. Ein Beispiel hierfür wird in Kapitel 4 diskutiert.

2.3.2 Rekurrente neuronale Netze

Im Unterschied zu konventionellen neuronalen Netzen weisen rekurrenten neuronalen Netzen Rückkopplungsverbindungen auf. Dies impliziert, dass ein Neuron eine Verbindung zu sich selbst aufweisen kann, wie in Abbildung 2.4 dargestellt.

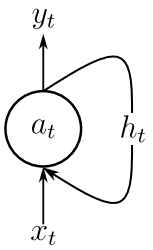

Abbildung 2.4: Ein rekurrentes Neuron zum Zeitpunkt t mit dem Input-Vektor x_t, dem skalaren Output y_t und dem skalaren rekurrenten Output h_t.

Die Speicherung und Nutzung von Informationen aus vorherigen Eingaben erfolgt mittels dieser Rückkopplungsschleife, wodurch die Verarbeitung sequenzieller Daten ermöglicht wird. Die Anordnung der Neuronen geschieht ebenfalls in Layern, wobei die Rückkopplungsverbindung durch die virtuelle Aneinanderreihung dieser Layer

über die Zeit realisiert wird. Das Eingabeformat solcher Layer ist eine Matrix, deren erste Dimension die Anzahl der Zeitschritte darstellt, welche im Weiteren mit T bezeichnet wird. Die Dimension der zweiten Variable wird durch die Anzahl der Neuronen definiert. Diese Dimension wird im Folgenden mit U bezeichnet.

Im Rahmen dieser Arbeit erfolgt eine Betrachtung der vollständig rekurrenten Architektur, insbesondere im Kontext der Back Propagation Through Time (BPTT), da deren Verbesserung das LSTM darstellt [11].

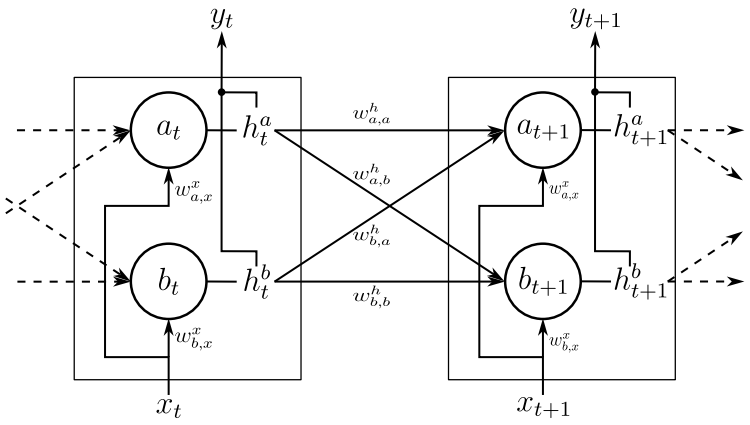

Abbildung 2.5: Zwei explizit dargestellte Zeitschritte, eines vollständig rekurrenten Netztes mit zwei rekurrenten Neuronen a, b, dem Input-Vektor x_t, dem Output-Vektor $y_t = [h_t^a, h_t^b]$, der Input Gewichtsmatrix W_x und der rekurrenten Gewichtsmatrix W_h.

Die Berechnung der Ausgabe zum Zeitpunkt t ähnelt der Berechnung in Netzen ohne rekurrente Verbindungen, wobei jedoch eine zusätzliche Gewichtsmatrix eingeführt wird, welche die Ausgabe des vorherigen Zeitschritts für den aktuellen Zeitschritt gewichtet. Die Ausgabe zum Zeitpunkt t lässt sich durch folgende Funktion darstellen:

$$h_t = f(W_x x_t + W_h h_{t-1} + b) \tag{10}$$

Dabei bezeichnet W_x die Gewichtsmatrix, welche alle Verbindungen vom Eingabevektor zu den Neuronen repräsentiert. W_h ist die Gewichtsmatrix für alle rekurrenten Verbindungen zwischen den Neuronen und den vorherigen versteckten Ausgaben.

Durch Zusammenfassung der Gewichtsmatrizen und Konkatenation der Vektoren h_{t-1} und x_t lässt sich eine allgemeine Formulierung der Ausgabe eines rekurrenten Neurons ableiten. Dies resultiert in einer Formel, die nahezu identisch mit der eines

konventionellen NNs ist.

$$y_t = f\left(W[x_t, h_{t-1}] + b\right) \tag{11}$$

In diesem Modell wird nicht zwischen einer versteckten Ausgabe und der tatsächlichen Ausgabe für den jeweiligen Zeitschritt unterschieden. Folglich gilt $y_t = h_t$.

Rekurrente neuronale Netze können sich auf zwei grundlegende Arten unterscheiden:

1. Die Ausgabe wird für jeden Zeitschritt gespeichert, wie in Abbildung 2.5 dargestellt. Dies wird als „**Sequence-Tagger**“ bezeichnet. Demnach ist die Ausgabe eine Matrix der Form $T \times U$.

2. Nur die finale Ausgabe zum Zeitpunkt T wird für die weitere Verarbeitung herangezogen. Dies wird als „**Sequence-Classifier**“ bezeichnet. Daher ist die Ausgabe ein Vektoren mit einer Länge U.

Ein grundlegendes Problem dieser Architektur besteht darin, dass beim Training die Fehlergradienten exponentiell wachsen oder gegen Null gehen können, wenn über viele Zeitschritte iteriert wird. Dies ist darauf zurückzuführen, dass die Änderung der Gewichte in diesem Fall multiplikativ in die Berechnung einfließt. Dies führt zu einem instabilen Training und kann das Lernen über längere Zeiträume hinweg erschweren oder gar unmöglich machen [11].

2.3.3 LSTM

Das Long Short-Term Memory (LSTM), zu deutsch Lang-Kurzzeit-Gedächtnis, zielt darauf ab, das Problem der verschwindenden oder explodierenden Fehlergradienten zu beheben. Um den Fehlergradienten über lange Zeiträume hinweg konstant zu halten, wird das Konzept des constant error carrousels (CECs) angewendet. Das CEC wird durch einen Cell-State umgesetzt, der linear, d. h. ohne den Einfluss einer Aktivierungsfunktion, an das gleiche Neuron im nächsten Zeitschritt weitergegeben wird. Dieser Zustand setzt sich aus dem vorherigen Cell-State C_{t-1} und einem temporären Cell-State $\tilde{C}_t$, der für jeden Zeitschritt aus den Hidden-Outputs der vorherigen Neuronen berechnet wird, zusammen.

In den nachfolgenden Formeln sind alle Variablen mit Kleinbuchstaben und C bzw. $\tilde{C}$ als Vektoren zu interpretieren. Alle W Variablen sind als Gewichtsmatrizen aufzufassen.

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t \tag{12}$$

Der Koeffizient f_t definiert dabei den Anteil des alten Cell-State, der vergessen werden soll. Daher wird dieser auch als „Forget Gate“ bezeichnet. Das Input-Gate i_t definiert den Einfluss, den der temporäre Cell-State auf den tatsächlichen Cell-State ausübt.

Um die Abhängigkeiten der Effekte des Input- und Output-Gates vom aktuellen sowie den vorherigen Zeitschritten zu erlernen, wird der Mechanismus eines normalen rekurrenten Neurons genutzt.

Aufgrund der fehlenden direkten Verbindung zum CEC manifestiert sich an dieser Stelle erneut das Problem der verschwindenden oder explodierenden Fehlergradienten. An dieser Stelle ist das Verhalten jedoch gewünscht, da auf diese Weise kurzzeitige Abhängigkeiten abgebildet werden können und die Weitergabe der Fehler begrenzt ist.

Eine Besonderheit dieser Gates besteht in der Verwendung der logistischen Sigmoid-Funktion als Aktivierungsfunktion, deren Wertebereich im Intervall $[0, 1]$ liegt. Signale, welche die Gates passieren, werden somit durchgelassen, wenn $\sigma(z) = 1$, oder unterdrückt, wenn $\sigma(z) = 0$ ist. Dabei stellt z eine allgemeine Aktivierung dar, wie in Gleichung 9. Dieses Verhalten ist vergleichbar mit einem Wasserhahn, welcher entweder offen, geschlossen oder sich in einem Zwischenzustand befindet. Durch separate Gewichte kann das Verhalten des spezifischen Gates erlernt werden.

$$i_t = \sigma(W_i \cdot [x_t, h_{t-1}] + b_i) \tag{13}$$
$$f_t = \sigma(W_f \cdot [x_t, h_{t-1}] + b_f) \tag{14}$$

Der temporäre Cell-State $\tilde{C}_t$ lässt sich gemäß folgender Formel berechnen:

$$\tilde{C} = \tanh(W_C \cdot [x_t, h_{t-1}] + b_C) \tag{15}$$

Die Aktualisierung des Hidden-Outputs erfolgt unter Zuhilfenahme des Cell-State sowie eines Output-Gates.

$$h_t = o_t \cdot \tanh(C_t) \tag{16}$$

Die Funktion des Output-Gates besteht in der Sicherung der nachfolgenden lstm-Neuronen vor Störungen des aktuellen Zeitschritts. Für weiterführende Informationen sei auf [11] verwiesen.

$$o_t = \sigma(W_o \cdot [x_t, h_{t-1}] + b_o) \tag{17}$$

Wie aus den obigen Gleichungen ersichtlich ist, sind die Gewichtsmatrizen und Bias-Vektoren nicht von der Zeitabhängigkeit betroffen und bleiben somit konstant. Die Formeln, abgesehen von der Reihenfolge der Konkatenation, stammen aus [3].

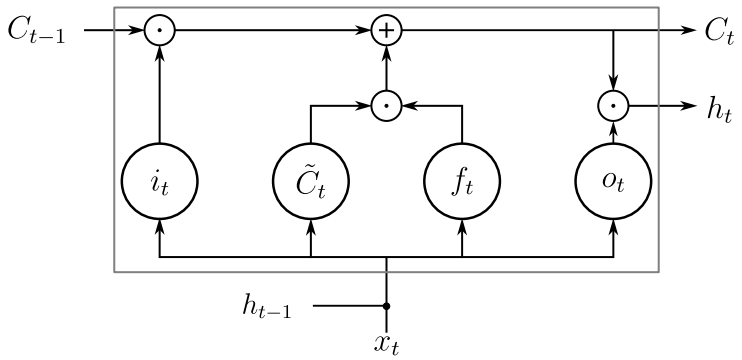

Abbildung 2.6: Schema eines LSTM-Neurons mit der Eingabe $[x_t, h_{t-1}]$

Zusammenfassung: Ein LSTM-Neuron setzt sich aus mehreren Einheiten zusammen. Diese sind dafür verantwortlich, dass bestimmte Eigenschaften der zu verarbeitenden sequentiellen Daten eine kurzzeitige Änderung der Aktivierung der Zelle auslösen können. Dies wird über die Gates realisiert. Zudem ermöglichen die Einheiten die Modellierung von Langzeitabhängigkeiten. Die additive Veränderung des Cell-States ermöglicht die Lösung des Problems der verschwindenden oder explodierenden Fehlergradienten über lange Zeiträume hinweg.

2.4 Embedded AI

Ein bedeutender Bereich der modernen KI-Forschung widmet sich der Ausführung von KI-Algorithmen auf kleineren Geräten, wie beispielsweise Mikrocontrollern. Diese Entwicklung ist jedoch mit mehrere Herausforderungen verbunden, die aus den Begrenzungen der Hardware sowie aus der Komplexität heutiger KI-Modelle resultieren.

Beschränkungen kleiner Geräte:

- Die begrenzte Speicherkapazität: Kleine Geräte verfügen oft über nur wenige Kilobyte bis Megabyte an Speicher, was die Implementierung großer Modelle erschwert [9].
- Begrenzte Rechenleistung: Die meisten Mikrocontroller besitzen nur eine geringe Anzahl von Rechenkernen, was die Verarbeitung komplexer Algorithmen verlangsamt [5].
- Energieverbrauch: Da diese Geräte oft ohne aktive Kühlung und meist mit einer Batterie versorgt, gebaut werden, muss der Energieverbrauch minimal gehalten werden, um eine übermäßige Wärmeentwicklung und Entladung der Batterie zu vermeiden [6].

Probleme heutiger KI im Bezug auf die Anwendung in Mikrocontrollern:

- Hoher Speicherplatzbedarf: Moderne KI-Modelle benötigen oft erheblichen Speicherplatz, der auf Mikrocontrollern nicht verfügbar ist [5].
- Hohe Rechenleistung: Die Ausführung komplexer KI-Algorithmen erfordert signifikante Rechenressourcen, was auf Mikrocontrollern zu einer langsamen Verarbeitung führen kann [9].
- Energiebedarf: Vor allem während des Trainings verbrauchen KI-Modelle erhebliche Mengen an Energie, was in eingebetteten Systemen eine Herausforderung darstellt [6].

Aus den genannten Gründen fokussieren sich embedded AI Frameworks primär auf die Inferenz von KI-Modellen und bieten seltener Funktionen für die Konzeption und das Training dieser Modelle an. In vielen Fällen ist eine spezielle Optimierung bereits erstellter Modelle für die jeweils verwendete Hardware erforderlich. Als Beispiel sei

hier die Reduktion redundanter Informationen genannt, welche dazu führt, dass bei gleichbleibender Leistung der Rechenaufwand verringert werden kann.

Die in den nachfolgenden Unterkapiteln dargelegten Aussagen basieren auf der Quelle [10].

2.4.1 Emmi Softwarefamilie

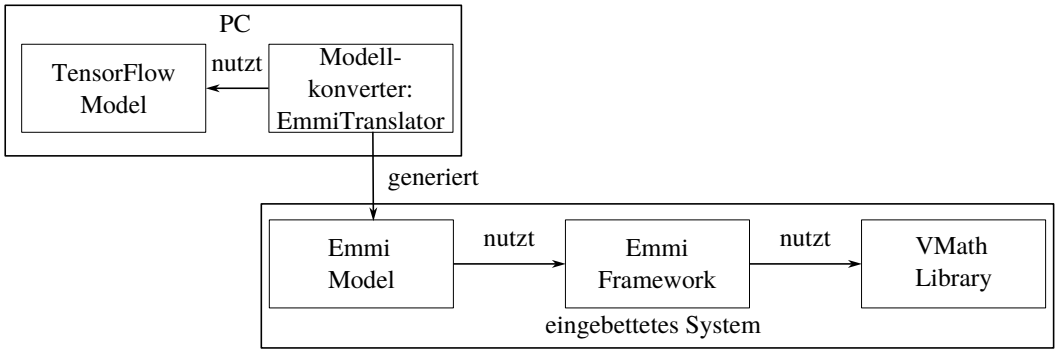

Abbildung 2.7: Architektur der Emmi Softwarekomponenten.

Das embedded AI Framework Emmi, entwickelt am Fraunhofer IPMS, ist darauf ausgelegt, trainierte neuronale Netze auf dem EMSA5 auszuführen. Funktionalitäten für das Training oder das abstrahierte Erstellen von Modellen werden nicht bereitgestellt. Die Lösung dieser Aufgaben erfolgt mittels TF, alternativ kann das Modell auch explizit im C-Code definiert werden.

Das gesamte Framework lässt sich als Softwarefamilie charakterisieren, wie in Abbildung 2.7 dargestellt. Diese besteht aus drei Komponenten:

- Dem **EmmiTranslator**, welcher verschiedene Submodule für das konvertieren der TF-Modelle in C-Code, sowie Qunatisierungsalgorithmen und Tools bereitstellt.
- Der **Emmi Core**. Dieses Modul stellt Funktionen, Typen und allgemeine Werkzeuge für NNs bereit.
- Die **VMath**-Bibliothek. Dies ist die fundamentale Mathematikbibliothek für den Emmi Core. Dort werden Rechenoperationen, Datenstrukturen und dafür benötigte Datenstruktur-Management-Funktionen definiert.

2.4.2 VMath

Die VMath-Bibliothek beinhaltet Datenstrukturen und Funktionen für numerische Berechnungen sowie das Management von ein- bis vierdimensionalen Tensoren. Die Funktionen sind für den Befehlssatz „Zve32x“ der RISC-V Architektur optimiert, sodass Operationen für ganzzahlige Vektoren variabler Länge simultan durchgeführt werden können. Die Bibliothek unterstützt zudem quantisierte Gleitkommazahlen, sodass auch Anwender der EMSA5-Versionen ohne FPU von der Vektorisierung profitieren können.

Unterstützte Datentypen der VMath-Bibliothek:

- **Float:** 32-Bit Gleitkommazahlen
- **Integer:** 32-Bit Ganzzahlen
- **Quantisierte Floats:** mit den Bitbreiten 32, 16, 8

2.4.3 Emmi Core

Das in der Programmiersprache C verfasste KI-Framework Emmi Core stellt das Kernstück der Emmi Softwarefamilie dar und lässt sich in vier voneinander unabhängige Bereiche untergliedern:

- Aktivierungsfunktionen
- Netzwerk Layer
- Benötigte Datentypen
- Hilfsfunktionen, welche in den anderen Bereichen verwendet werden.

Die implementierten Layer umfassen unter anderem ein- und zweidimensionale Convolutions, Dense Layer sowie eine Vielzahl weiterer Layertypen. Die Funktionalitäten des Softwarepakets sind kompatibel mit allen Quantisierungsdatenformaten, sowie den 32-Bit-Gleitkommazahlen, wie sie von der VMath-Bibliothek zur Verfügung gestellt werden.

2.4.4 EmmiTranslator

Das EmmiTranslator-Paket umfasst eine Reihe von Submodulen, die in logisch zusammengehörige Einheiten gegliedert sind. Diese Struktur erlaubt eine flexible Umwandlung von TF-Modellen in C-Code, welcher die Funktionen des Emmi-Frameworks nutzt. Die wesentlichen Komponenten des Pakets sind der Translator, der Model Generator und der Code Generator. Des Weiteren besteht die Option, Modelle während des Konvertierungsprozesses zu quantisieren, was durch das Modul Quantization ermöglicht wird.

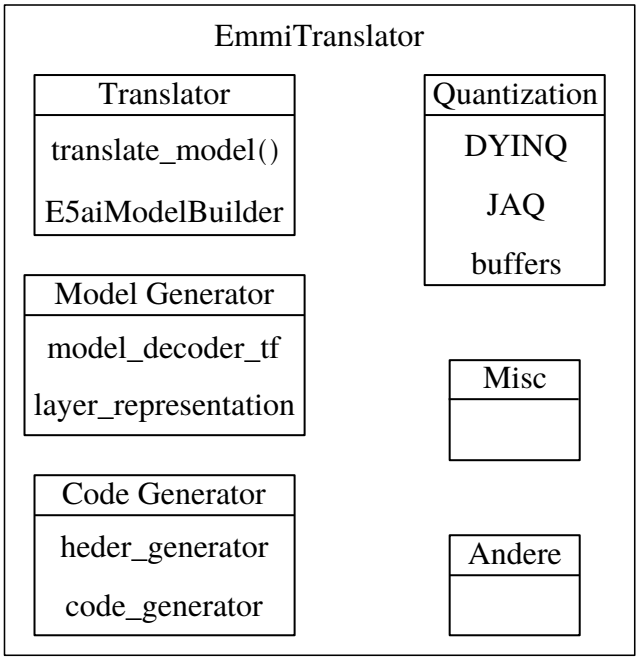

Abbildung 2.8: Logische Einheiten des EmmiTranslators.

Die Funktion *translate_model()* initiiert den Konvertierungsprozess, indem sie auf den **E5aiModelBuilder** zurückgreift. Der Builder umfasst sämtliche für eine erfolgreiche Umwandlung und gegebenenfalls Quantisierung des Modells erforderlichen Schritte.

Der EmmiTranslator verwendet eine interne Repräsentation des TF-Modells, welche unter Zuhilfenahme der Module des **Model Generators** erstellt wird. Die Modularität des Systems erlaubt die Integration neuer Layer, ohne dabei die Funktionalität der übrigen Module zu beeinträchtigen.

Die tatsächliche Erzeugung des C-Codes erfolgt durch die Klassen des **Code Generators**. Des Weiteren besteht die Möglichkeit, die Funktionen des *code_generator_tf*

zu nutzen, um die Daten für spätere Tests zu generieren.

Das **Quantization** Modul implementiert zwei Quantisierungstechniken, DYINQ und Efficient Intege-Arithmetic-Only Inference von Benoit Jacob et. Al (JAQ), welche eine identische Speicherrepräsentation für quantisierte Werte verwenden. Dies erlaubt die Verwendung eines einzigen Buffers zur Speicherung der quantisierten Werte beider Techniken. Des Weiteren implementieren beide Klassen arithmetische Operationen für ihre spezifischen Werte. Die JAQ-Quantisierung findet lediglich zum Vergleich mit DYINQ Anwendung, nicht jedoch im **Translator**.

Das Modul **Misc** dient der Zusammenfassung von Funktionen, welche von den übrigen Modulen des Pakets in gleicher Weise genutzt werden.

Die Kategorie **Andere** umfasst diverse Module für die Grafik, darunter solche, die sich mit der Analyse von TensorFlow-Gewichten, Biases und Layer-Outputs befassen. Des Weiteren werden Funktionalitätstests des EmmiTranslators durchgeführt und Code-Demonstrationen angeboten.

2.5 TensorFlow

TensorFlow (TF), eine von Google entwickelte Open-Source-Bibliothek, dient zur Durchführung numerischer Berechnungen mit einem besonderen Fokus auf maschinelles Lernen. TF unterstützt Strukturen wie ein- bis n-dimensionale Tensoren, bietet eine Vielzahl von Optimierungsalgorithmen sowie spezialisierte Algorithmen für maschinelles Lernen und tiefe neuronale Netzwerke [1].

Eine der wichtigsten Funktionen von TF ist die Integration der Keras-Subbibliothek, die eine benutzerfreundliche API bereitstellt. Diese ermöglicht das schnelle und einfache Erstellen komplexer KI-Modelle und deren automatische Optimierung für verschiedene Hardwaretypen, wie z.B. GPUs. Keras erleichtert es, tiefgehende neuronale Netze zu entwerfen und zu trainieren, ohne sich um die Details der zugrunde liegenden TF-Implementierung befassen zu müssen.

Ein wesentlicher Aspekt bei der Nutzung von TF Keras ist, dass die erste Dimension des Eingabe-Tensors die Batch-Größe repräsentiert. Diese Strukturierung ermöglicht eine effiziente Parallelisierung des Trainingsprozesses, beugt Overfitting vor und verbessert die Stabilität des Trainings durch die Mittelung des Gradienten über mehrere Datenpunkte [7].

3 Implementierung

3.1 Vorgehen

Die Vorgehensweise bei der Implementierung ist in zwei Etappen unterteilt. Die erste Etappe konzentriert sich auf die Implementierung für den Datentyp Float und die Anpassung des EmmiTranslators. Im zweiten Schritt wird die Anpassung an den Datentyp q32 implementiert.

Etappe 1

Die Entwicklung eines ersten Prototyps in Python dient der Überprüfung des Verständnisses eines LSTM-Layers. Der Code ist im Anhang unter *scripts/lstm_noquant.py* zu finden. Durch den Vergleich der Ausgaben eines LSTM-Layers aus dem TF-Framework wird in diesem Schritt überprüft, ob der Prototyp funktioniert.

Dabei muss das Speicherformat der Gewichte und Biase aus TF analysiert und an das verwendete Format aus Kapitel 2.3.3 angepasst werden. Eine detaillierte Beschreibung hierzu findet sich in Kapitel 3.3.2.

Anschließend erfolgt die Implementierung für das Emmi-Framework in C. Diese wird durch weitere Tests, die im Anhang unter *Emmi Tests/* zu finden sind, auf Korrektheit geprüft.

Sobald sichergestellt ist, dass alle Tests für einzelne LSTM-Layer erfolgreich sind, werden Tests erstellt, deren Aufgabe es ist, die Integrierbarkeit des Layers in komplexere KI-Modelle zu verifizieren. Um Tests mit verschiedenen Konfigurationen des LSTM-Layers mit geringem Designaufwand der Testmodelle durchführen zu können, wurde zusätzlich ein Reshape-Layer für das Emmi-Framework implementiert. Dieser Layer ist nicht Gegenstand der Anforderungen und wird daher nicht weiter betrachtet. Seine Implementierung ist jedoch im Anhang unter *Emmi Core/layer_flt.c* zu finden.

Etappe 2

Wie in der ersten Etappe beginnt auch diese mit der Implementierung eines Python-Prototypen für quantisierte Daten. In einem schrittweisen Vergleich mit dem Prototyp aus Phase 1 wird dessen theoretische Funktionalität überprüft. Durch diese Vergleiche kann festgestellt werden, dass die Einführung der Quantisierung zu einem wesentlich unvorhersehbareren Verhalten führt, da es zu Überläufen und größeren Rundungsfehlern kommt. Daher wurde eine andere Teststrategie gewählt, die in Kapitel 4.2 genauer beschrieben ist.

Trotz der größtenteils nicht bestandenen Tests wurde der Prototyp an das Emmi-Framework angepasst. Diese Anpassung ist in Kapitel 3.2.2 beschrieben.

3.2 LSTM Layer

3.2.1 Emmi-Implementierung für Float

Die Implementierung des LSTMs wurde in zwei Hauptfunktionen aufgeteilt, die nach dem Prinzip des Pass-By-Reference arbeiten, um die vorinitialisierten Ausgabe-Tensoren zu modifizieren.

Die Funktion ***lstm_step_flt*** führt alle Berechnungen des Layers durch und gibt die Hidden- und Cell-State-Vektoren eines Zeitschritts zurück. Sie verwendet folgende Parameter:

- Input-Vektor
- Hidden-Vektor
- Alle Gewichtsmatrizen
- Alle Bias-Vektoren
- Alle Cell-States als Vektor zusammengefasst

Für jeden Zeitschritt werden temporäre Tensoren für die Speicherung der Gate-Werte und zwei Tensoren zum Speichern von Zwischenergebnissen angelegt. Zudem wird in der Funktion der Input-Vektor und der Hidden-Vektor konkateniert, um die Formeln aus Kapitel 2.3.3 anzuwenden. Um eine höhere Genauigkeit zu gewährleisten, werden die nicht approximierten Versionen der Sigmoid- und Tangens-Hyperbolicus-Funktionen verwendet.

Die zweite Funktion, ***lstm_flt***, dient als Aufruffunktion für den Layer im Modell. Sie prüft die Parameter des Layers und initialisiert den Cell-State- und den Hidden-Vektor mit Null. Zudem wird das Unrolling mittels einer For-Schleife durchgeführt. Dabei wird die Eingabe für den jeweiligen Zeitschritt aus dem Input-Vektor extrahiert und an ***lstm_step_flt*** übergeben.

Ein LSTM wird über die Anzahl der Neuronen, die Anzahl der Features und die Anzahl der Zeitschritte definiert, die aus den Dimensionen des Eingabevektors und einer Gewichtsmatrix rekonstruiert werden können. Ein zusätzlicher Parameter bestimmt, ob das Layer die Ausgaben aller Zeitschritte (als *Sequence-Tagger*) oder nur die letzte Ausgabe (als *Sequence-Classifier*) an den nächsten Layer bzw. den Modellausgang weitergibt.

3.2.2 Emmi-Implementierung für q32

Die Implementierung für q32 folgt dem gleichen Aufbau wie die Implementierung für Float-Werte. In beiden Funktionen wird jedoch ein zusätzlicher Parameter eingeführt, welcher die interne Bitbreite der Variablen definiert. Die betreffende Bitbreite findet bei jeder Reskalierung Anwendung. Im folgenden Kapitel 4 wird diese Bitbreite als „interne Bits“ bezeichnet. Im Gegensatz dazu werden die Bitbreiten der Gewichte, Biase und Eingaben als „main Bits“bezeichnet, da deren Werte durch die Konfiguration der Konvertierung entstehen.

Infolge der spezifischen Verarbeitung von quantisierten Werten im Rahmen binärer Operationen weichen die Berechnungen der Gates, der Cell-States sowie die Konkatenation von der Implementierung für Fließkommazahlen ab.

Konkatenation

Für eine Konkatenation muss der Skalierungsfaktor der Operanden vereinheitlicht werden. Dies wird mit der Funktion *unify_representation_q32* erreicht.

Listing 1: Quantisierte Konkatenation.

```
1    // Concatenate x and h.
2
3    // Bring x_t and h to the same scale.
4    unify_representation_q32(x_t,h);
5
6    // Set data.
7    int32_t x_h_data[x_t->m + h->m];
8    for (size_t i = 0; i < x_t->m; i++) x_h_data[i] = x_t->data[i];
```

```
9     for (size_t i = 0; i < h->m; i++) x_h_data[i + x_t->m] = h->data[i];
10
11    // Add tensor metadata.
12    Tensor_q32 x_h = {.type = QINT32,
13                      .m = x_t->m + h->m,
14                      .n = 1,
15                      .p = 1,
16                      .q = 1,
17                      .elements_mn = x_t->m + h->m,
18                      .elements_mnp = x_t->m + h->m,
19                      .elements_mnpq = x_t->m + h->m,
20                      .elements_max = x_t->m + h->m,
21                      .data = x_h_data,
22                      .zero_offset = h->zero_offset,
23                      .scale = h->scale,
24                      .n_bits = h->n_bits};
25
26    // Rescale tensors.
27    tensor_rescale_q32_q32(h,h,nbits);
28    tensor_rescale_q32_q32(&x_h,&x_h,nbits);
```

Ein ähnliches Vorgehen findet Anwendung bei der Integration des Hidden-Vektors in den Ausgabe-Tensor.

Quantisierte Gates

Im Folgenden wird ein Beispiel für ein Gate mit quantisierten Werten präsentiert.

Listing 2: Quantisiertes Gate.

```
1     // Input Gate.
2     ret |= matrix_multiply_q32_c(w_i, &x_h, &i);
3     ret |= tensor_rescale_q32_q32(&i, &i, nbits);
4     ret |= tensor_eadd_tensor_q32(&i, b_i, &i);
5     ret |= tensor_rescale_q32_q32(&i, &i, nbits);
6     ret |= sigmoid_exp_q32(&i, &i, nbits);
```

3.3 EmmiTranslator

Für eine erfolgreiche Konvertierung von TF-Modellen, die LSTM-Layer enthalten, sind spezifische Modifikationen am EmmiTranslator notwendig. Diese Änderungen werden im Folgenden erläutert.

3.3.1 Anpassung EmmiTranslator: ModelDecoderTF

Die Klasse *ModelDecoderTF* aus dem Modul *model_decoder_tf* durchläuft die Layer des TF-Modells und überträgt deren Eigenschaften in die interne Repräsentation mittels spezifischer *extract_layer_<Layer Type>* Funktionen. Für LSTM-Layer wird folgende Prozedur in der neuen Funktion *extract_layer_LSTM* umgesetzt:

1. Inkrementierung des Ausgabereferenzzählers des vorhergehenden Layers, welcher als Eingabe für den aktuellen LSTM-Layer dient.
2. Hinzufügen des Layer-Namens in die „input-Liste“, welcher die Eingabe für den aktuellen LSTM-Layer erzeugt.
3. Die Erstellung eines vorläufig leeren Layers erfolgt, sofern der Layer, welcher eine Eingabe des aktuellen LSTM-Layers liefert, noch nicht in der internen Repräsentation existiert.
4. Festlegung von Metadaten wie Layer-Name, Typ („lstm“), Datentypen sowie Dimensionen für Gewichte, Biase, Ein- und Ausgabe-Tensoren.
5. Aufnahme der Layer, die den Output des LSTM-Layers nutzen, in die „output-liste“.
6. Setzen eines Flags in der Parameterliste der LSTM Funktion des Emmi-Frameworks, das anzeigt, ob der Layer als *Sequence-Tagger* (1) oder als *Sequence-Classifier* (0) agieren soll, basierend auf dem Zustand des „return_sequences“ Flags.
7. Extraktion und Konvertierung der Gewichts- und Bias-Matrizen bzw. Vektoren in das Format, das vom Emmi-Framework genutzt wird.
8. Hinzufügen der extrahierten Gewichte und Biase zu den Metadaten.

Die letzten drei Schritte sind spezifisch für LSTM-Layer, während die übrigen Schritte hauptsächlich der korrekten Erstellung des Modell-Graphen dienen und daher auch in anderen extraktions Funktionen zu finden sind.

3.3.2 Konvertierung der Gewichte und Biase

Das TF-Format

In einem TF-Modell werden die Gewichte und Biase in einem Format gespeichert, welches sich von dem in Kapitel 2.3.3 beschriebenen Format unterscheidet. Bei der Extraktion der Gewichte und Biase eines LSTM-Layers werden drei Matrizen zurück gegeben:

1. Eine Kernel-Matrix der Dimensionen $(F \times 4U)$, die alle Gewichte der Gates vom Input-Vektor zu den LSTM-Neuronen enthält.
2. Eine rekurrente Kernel-Matrix der Dimensionen $(U \times 4U)$, die alle Gewichte des Hidden-Vektors zu den LSTM-Neuronen speichert.
3. Ein Bias-Vektor der Dimension $(4U)$, der alle Bias-Werte der Gates enthält.

Hierbei steht F für die Anzahl der Features und U für die Anzahl der LSTM-Neuronen. Die Gates werden in der Reihenfolge Input Gate, Forget Gate, Cell Gate, Output Gate gespeichert, wobei die Neuronen für das jeweilige Gate direkt hintereinander stehen.

Beispiel: Betrachtet wird ein LSTM-Layer mit $U = 2$ und $F = 2$. Die Gewichte des rekurrenten Kernels sind dem entsprechend wie folgt angeordnet:

$$Kernel_{rekur} = \begin{pmatrix} i_{u_1,u_1} & i_{u_1,u_2} & f_{u_1,u_1} & f_{u_1,u_2} & c_{u_1,u_1} & c_{u_1,u_2} & o_{u_1,u_1} & o_{u_1,u_2} \\ i_{u_2,u_1} & i_{u_2,u_2} & f_{u_2,u_1} & f_{u_2,u_2} & c_{u_2,u_1} & c_{u_2,u_2} & o_{u_2,u_1} & o_{u_2,u_2} \end{pmatrix} \quad (18)$$

Das Emmi-Format

Im Gegensatz zum TF-Format erfordert die Emmi-Implementierung die Angabe von Gewichten und Biase, die nach Gates separiert sind. Die Umformatierung der Gewichte und Biase erfolgt innerhalb des Moduls *model_decoder_tf*.

Die Dimensionen der Gewichtsmatrizen sind jeweils $(U \times (F + U))$. Es sei angemerkt, dass die Elemente der Dimension $F + U$ nicht vertauschbar sind, da im konkatenierten Input-Vektor zunächst die Features des Zeitschrittes und anschließend der Hidden-Vektor des vorhergehenden Zeitschrittes stehen. Folglich sind in den Matrizen pro Zeile zunächst die Gewichte für die Features und sodann jene

für den Hidden-Output aufgeführt. Der Bias aus dem TF-Format wird an den entsprechenden Stellen getrennt, sodass eine Kompatibilität mit dem Emmi-Format gewährleistet ist.

Listing 3: Auszug aus der Gewichts-Konvertierungs-Funktion des *model_decoder_tf* Moduls.

```
    kernel = lstm_weights[0]
    kernel_rekur = lstm_weights[1]
    b = lstm_weights[2]

    units = kernel_rekur.shape[0]
    features = kernel.shape[0]
    b_i = b[0:units]
    b_f = b[units:2*units]
    b_c = b[2*units:3*units]
    b_o = b[3*units:]

    w_i = np.zeros((units,features+units))
    w_f = np.zeros((units,features+units))
    w_c = np.zeros((units,features+units))
    w_o = np.zeros((units,features+units))

    for n in range(units):
            i_gateK = kernel[:,n]
            i_gateR = kernel_rekur[:,n]
            mitmit  w_i[n,:] = np.hstack((i_gateK.flatten(), i_gateR))

            f_gateK = kernel[:,n+units]
            f_gateR = kernel_rekur[:,n+units]
            w_f[n,:] = np.hstack((f_gateK.flatten(), f_gateR))

            c_gateK = kernel[:,n+units*2]
            c_gateR = kernel_rekur[:,n+units*2]
            w_c[n,:] = np.hstack((c_gateK.flatten(), c_gateR))

            o_gateK = kernel[:,n+units*3]
            o_gateR = kernel_rekur[:,n+units*3]
            w_o[n,:] = np.hstack((o_gateK.flatten(), o_gateR))
```

Anpassungen EmmiTranslator: quantisiertes LSTM

Aufgrund der strukturellen Trennung zwischen Modell- und Code-Erzeugung im EmmiTranslators muss lediglich das LSTM in ein Tuple von Layern integriet werden, die mit quantisierte Datentypen kompatibel sind.

4 Evaluation

Um sicherzustellen, dass die Float-Implemenetierung funktioniert wurden Unit- und Intgrationstests erstellt. Während der Entwicklung des quantisierten Layers wurde festgestellt, dass dieser Ansatz nicht direkt übertragbar ist. Daher wurde für den quantisierten Layer eine umfassendere Analysestrategie gewählt.

Im Allgemeinen erfolgen die Tests durch ein Vergleich der Ausgabewerte der Implementierungen mit jenen der Referenzimplementierung aus TF, da die Modelle letztlich mit TF erstellt und trainiert werden. Aus diesem Grund wird kein Wert auf die Interpretierbarkeit der Ausgaben gelegt. Stattdessen werden nur zufällig generierte Eingaben, Gewichte und Biase verwendet.

Die Bewertung der Güte der Ausgaben erfolgt durch eine Funktion, welche die Ausgabewerte elementweise mit den Referenzwerten vergleicht und eine Abweichung innerhalb einer definierten Toleranz als Kriterium heranzieht. Liegt mindestens einer der Ausgabewerte außerhalb der Toleranz, ist der Test nicht bestanden. Die Definition der Toleranz erfolgt als Parameter der Funktion durch eine Prozentangabe.

Ein Test bezeichnet hier den Vergleich eines Referenzwertes, welcher meist in Form eines Tensors gegeben ist, mit der Ausgabe der erstellten Implementierung.

4.1 Teststrategie der Float-Implementierung

Während der Entwicklungsphase wurden die Ausgaben des Python-Prototyps mit denen des TF LSTMs verglichen. Nach erfolgreicher Verifikation des Prototyps wurden Tests für den Layer des Emmi-Frameworks in Form von Unit Tests erstellt und anschließend Modelle für Integrationstests entwickelt.

4.1.1 Unit Tests

In den Unit Tests wird die Funktionalität des LSTM-Layers überprüft. Dazu werden Layer unterschiedlicher Größe und mit zufällig generierten Eingabedaten und Gewichten getestet. Ein Testdaten- und Testfallgenerator wurde in Python erstellt, welcher im Anhang unter *Emmi Tests/layer tests* zu finden ist.

Der Ablauf der Layer Tests wurde wie folgt gestaltet:

1. Testdaten und Testfälle erstellen.
2. Tests durchführen.
3. Fehlersuche und -behebung, falls die Ausgaben nicht in einer 10%-Umgebung um den Referenzwert liegen.

4.1.2 Integrationstests

Ziel der Integrationstests ist es zu überprüfen, ob der LSTM-Layer erfolgreich in komplexere NNs integriert werden kann.

Die folgenden Schritte sind notwendig, um die Integrationstests durchzuführen:

1. Modell in TF erstellen.
2. Modell konvertieren.
3. Testfälle erstellen.
4. Tests durchführen.
5. Fehlersuche und -behebung, falls die Ausgaben nicht in einer 10%-Umgebung um den Referenzwert liegen.

Für die Tests wurden drei Modelle entwickelt. Die ersten beiden sind sequentiell strukturiert, während das dritte ein funktionales Modell darstellt. Dieses funktionale Modell wurde speziell entwickelt, um zu überprüfen, ob auch nicht sequentielle Modelle korrekt übersetzt werden und funktionsfähig sind.

Die für das Training und die anschließenden Tests verwendeten Daten stammen aus einem bereits existierenden Zeitreihendatensatz, der mehrere verschiedene eindimensionale, künstlich erzeugte Funktionen enthält. Bei den Tests wurde jedoch nicht die Genauigkeit der Vorhersage zur Bewertung des Testergebnisses herangezogen, sondern die Übereinstimmung mit den Referenzwerten des TF-Modells.

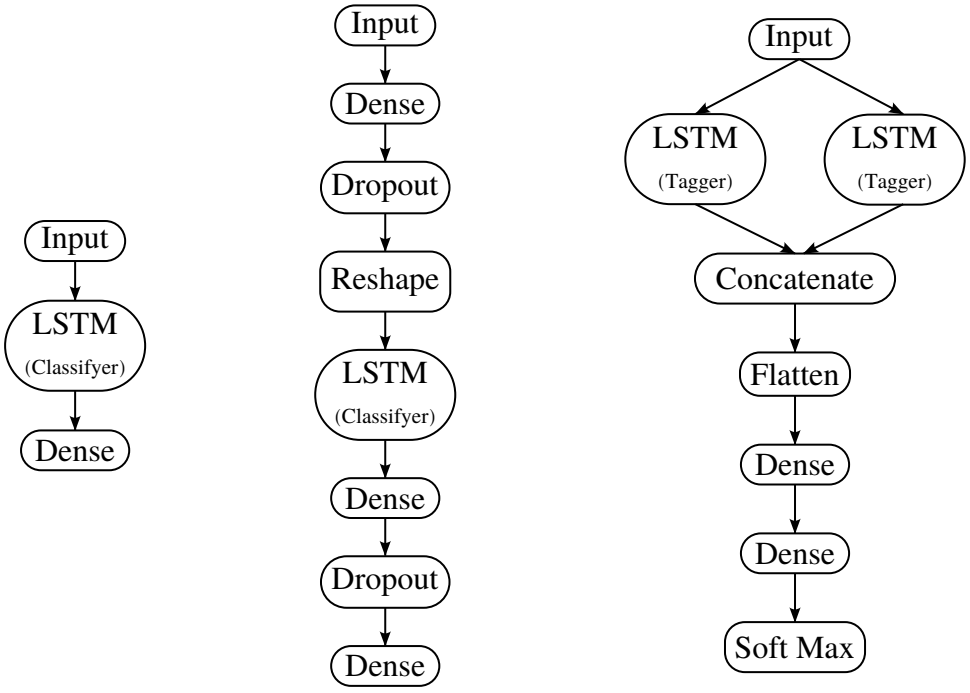

Abbildung 4.1: Getestete Float-Modelle.

Wie in der Abbildung zu sehen ist, enthält eines der sequentiellen Modelle Dropout-Layer. Diese Layer sind während der Inferenz nicht relevant und können daher durch Setzen eines Flags beim Aufruf der Translate-Funktion des EmmiTranslators entfernt werden. Dies hat Auswirkungen auf die Struktur des generierten Modells. Daher wurde das Modell einmal mit und einmal ohne Dropout-Layer generiert, um zu überprüfen ob diese Änderung mit dem neuen LSTM-Layer korrekt zusammenarbeitet.

4.1.3 Ergebnisse

Diese Tests haben gezeigt, dass die Ergebnisse der Float-Implementierung innerhalb einer Toleranz von 10% liegen. Dies bestätigt die Funktionsfähigkeit der Implementierung.

4.2 Teststrategie der quantisierten Implementierung

Bei der Entwicklung des Prototyps hat sich gezeigt, dass bei sehr kleinen LSTMs schnell Overflows auftreten können. Wenn keine Overflows auftraten, waren die Abweichungen oft sehr groß. Um besser zu verstehen, unter welchen Bedingungen Over-

flows oder Rundungsfehler auftreten, wurden Testdaten für den Prototyp entwickelt. Python bietet den Vorteil einer eingebauten Overflow-Warnung, die genutzt wurde, um diese Probleme zu identifizieren. Für die Emmi-Implementierung wurden ähnliche Unit Tests durchgeführt, die jedoch aufgrund des fehlenden Overflow-Handlings nicht gut analysiert werden konnten.

Um Overflow-Warnungen abzufangen, wurde der LSTM-Prototyp leicht modifiziert. Diese Modifikation ist als separater Layer im Anhang unter *scripts/lstm_quant.py* zu finden.

Diese Modifikation ermöglicht es, dem Ergebnis eines Tests (erfolgreich oder nicht erfolgreich) hinzuzufügen ob ein Overflow aufgetreten ist oder nicht. Wenn ein Overflow während des Tests auftritt, wird der Test abgebrochen und als nicht bestanden markiert. Tritt kein Overflow auf, ist der Test entweder bestanden oder das Ergebnis liegt außerhalb des Toleranzbereichs. Der Toleranzbereich ist im Gegensatz zu den Tests der Float-Implementierung auf 20% gesetzt, um den Fehler, der durch die Qunatisierung entsteht, auszugleichen.

Es wurde beobachtet, dass die Wahrscheinlichkeit eines Overflows oder eines fehlgeschlagenen Tests von den internen Bits, den main Bits und der Größe des LSTMs, die durch Timesteps T, Features F und Units U bestimmt wird, beeinflusst wird. Bemerkenswert ist, dass die Overflows fast ausschließlich im Skalierungsfaktor s der quantisierten Werte auftraten.

Zur Untersuchung des Verhaltens wurden mehrere LSTMs getestet, bei denen die Größe $T = F = U$ festgelegt wurde, um den Raum der möglichen Konfigurationen einzuschränken. Im Folgenden wird die Größe eines LSTMs mit $T = F = U$ als TFU bezeichnet.

Für jeden TFU-Wert wurde jeweils einer der Parameter (main Bits bzw. interne Bits) variiert, während der andere auf einem festen Wert belassen wurde. Dieser feste Wert beträgt für beide Parameter acht. Die Obergrenze von 14-Bit für die main bzw. internen Bits ergibt sich aus [10] Kapitel 2.4.5.

Für jede dieser Konfigurationen wurden verschiedene Eingangsdaten, Gewichte und Biase getestet. Die folgenden Diagramme zeigen diese Konfigurationen eines LSTM-Layers. Die Farbe gibt an, wie viele Tests der gleichen LSTM-Konfiguration im Verhältnis zur Anzahl der mit dieser Konfiguration durchgeführten Tests fehlgeschlagen sind.

4.2.1 Messungen

Hier werden die Messergebnisse grafisch dargestellt.

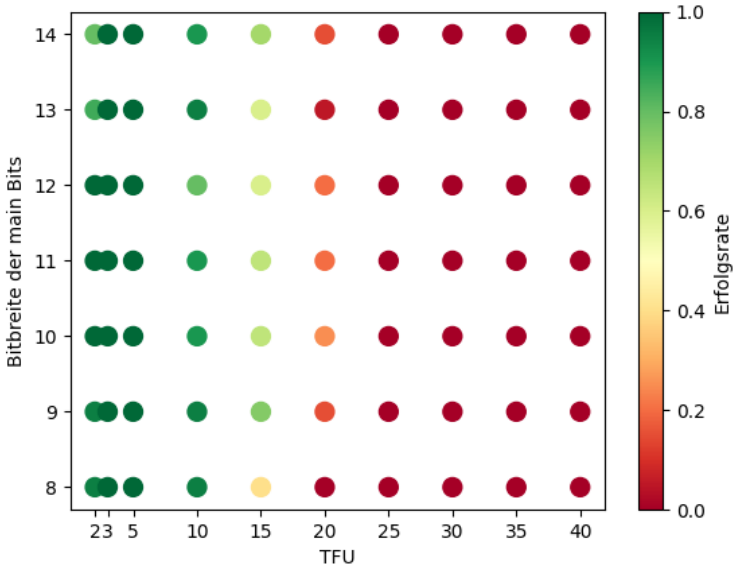

Abbildung 4.2: Erfolgsraten mit veränderlicher TFU und variablen main Bits.

In Abbildung 4.2 sind nur geringe Unterschiede zwischen den Farben einer Spalte für TFU $\in \{2, 10, 15, 20\}$ zu erkennen. Dies deutet darauf hin, dass die main Bits nur einen geringen Einfluss auf das Fehlverhalten haben. Außerdem wird deutlich, dass ab einem TFU-Wert von ca. 20 eine Grenze für das Bestehen der Tests existiert.

In der Grafik 4.3 gibt es eine Verschiebung des Übergangs zu den ausschließlich roten Punkten einer Reihe. Bei den Bitbreiten 13 und 14 stagniert diese Verschiebung. Daher eine Bitbreite von 12-Bit als optimaler Wert für die internen Bits in Bezug zu dem Standertwert der main Bits sein. Für die Stellen TFU $\in \{2, 3\}$ gibt es interessanterweise bei den oberen internen Bits viele fehlgeschlagene Tests. Interessanterweise gibt es bei den oberen internen Bits an der Stelle TFU $\in \{2, 3\}$ viele fehlgeschlagene Tests.

Es ist wichtig zu beachten, dass in diesen Diagrammen nicht zwischen Tests unterschieden wird, die aufgrund eines Overflows fehlgeschlagen sind, und solchen, die wegen eines wachsenden Fehlers versagt haben.

Werden die Fälle eliminiert, in denen das Fehlschlagen eines Tests nicht auf das Auftreten eines Overflows zurückzuführen ist, so zeigt sich, dass nur ein kleiner Teil der fehlgeschlagenen Tests auf einen Overflow zurückzuführen ist.

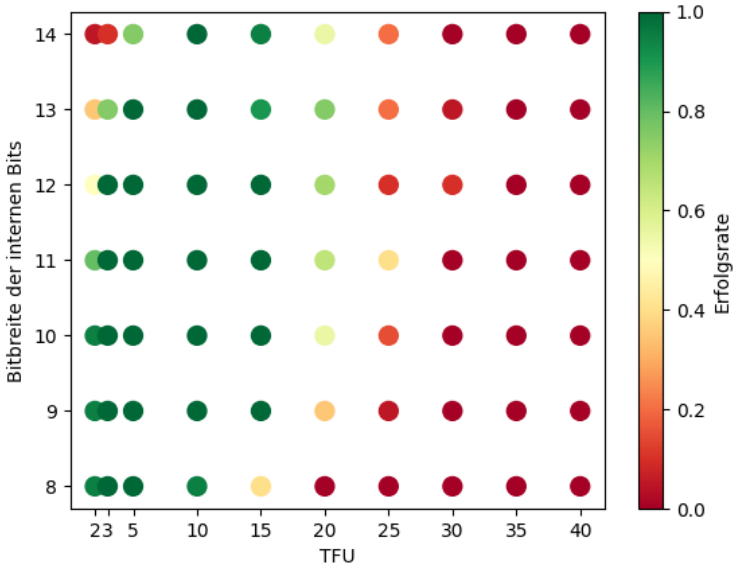

Abbildung 4.3: Erfolgsraten mit veränderlicher TFU und variablen intern Bits.

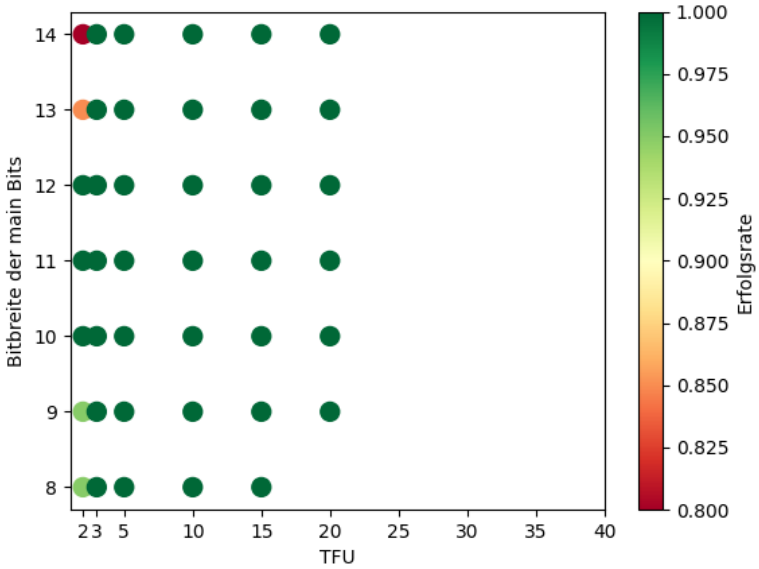

Abbildung 4.4: Erfolgsraten, bei denen ein Overflow zu Fehlschlägen führt, mit veränderlicher TFU und variablen main Bits.

Im Vergleich zu Abbildung 4.2 sind im linken oberen Bereich des Diagramms mehr rötliche Punkte zu sehen, was auf eine höhere Anzahl von Overflow-bedingten Fehlern hinweist. Ab TFU= 3 sind nur noch grüne Punkte sichtbar, was darauf schließen lässt, dass die Verschiebung ins Rote in Abbildung 4.2 ausschließlich auf Rundungsfehler zurückzuführen ist.

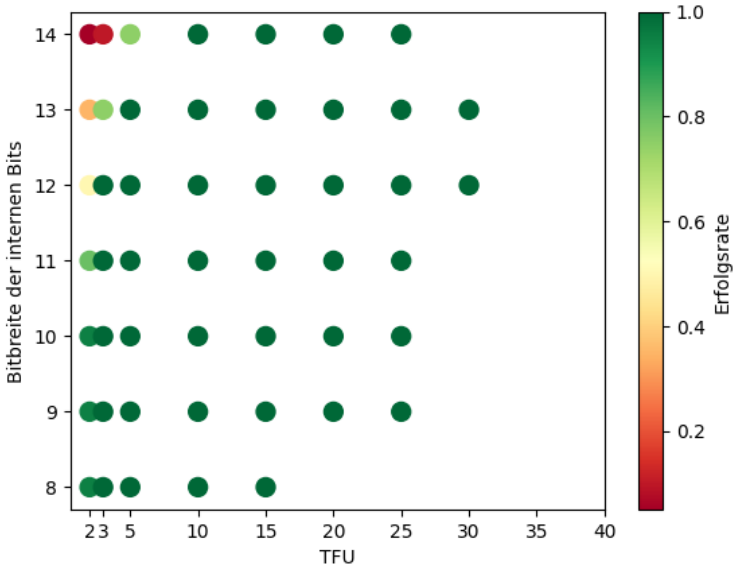

Abbildung 4.5: Erfolgsraten, bei denen ein Overflow zu Fehlschlägen führt, mit veränderlicher TFU und variablen internen Bits.

In Abbildung 4.5 kann eine ähnliche Beobachtung wie in Abbildung 4.4 gemacht werden. Allerdings treten hier auch Overflows für TFU= 3 auf. Das Besondere an diesem Diagramm ist, dass die Tests auch für TFU $\in \{25, 30\}$ bestanden werden.

Overflows treten vor allem bei kleinen TFU-Werten und bevorzugt bei hohen Bitbreiten auf.

Aufgrund der hohen Fehlerrate wurden keine weiteren Tests für ganze KI-Modelle durchgeführt.

4.2.2 Ergebnisse

Im Allgemeinen hat die Größe der LSTMs den größten Einfluss auf das Bestehen der Tests, da ab einem TFU-Wert von etwa 20 mehr Tests fehlschlagen als bestanden werden.

Die Parameter interne Bits und main Bits haben ebenfalls einen nachweisbaren Einfluss auf das Fehlverhalten. Dabei ist der Einfluss der main Bits weniger stark aus. Ein Grund dafür könnte sein, dass die Eingabe, die Gewichte und die Biase, welche die main Bits als Bitbreite verwenden, während der Berechnung des LSTMs konstant bleiben. Im Gegensatz dazu werden die internen Bits für variable Werte verwendet und haben daher einen größeren Einfluss auf die Genauigkeit des Ergebnisses.

Für die main Bits sollte ein Wert zwischen 10-Bit und 12-Bit verwendet werden, da in diesem Bereich keine Fehler durch Overflows auftreten.

Es ist wichtig zu beachten, dass die Diagramme nur dazu dienen, einen groben Überblick über den Verlauf der fehlgeschlagenen Tests zu geben. Es ist möglich, dass bessere Ergebnisse erzielt werden, wenn die Standardparameter optimaler gewählt werden. Beispielsweise wurde festgestellt, dass acht nicht die optimale Bitbreite ist, obwohl dies der Standardwert für diese Analyse ist. Daher könnten bessere Ergebnisse erzielt werden, wenn beide Bitbreiten gleichzeitig auf einen höheren Wert gesetzt würden.

Um gute Ergebnisse zu erzielen, muss daher für jeden Anwendungsfall zunächst mit der Float-Implementierung geprüft werden, welche Größe des LSTMs ausreichend ist, um dann ein Optimum für die Bitbreiten zu ermitteln.

Aus diesem Grund wird von einer unüberlegten Verwendung der quantisierten Implementierung abgeraten, da sie für allgemeine Fälle nicht geeignet ist. Die Verwendung dieser Implementierung in komplexen NN wird nicht empfohlen, da das Verhalten in dieser Arbeit nicht untersucht wurde.

5 Zusammenfassung und Ausblick

5.1 Zusammenfassung

Die vorliegende Arbeit hat sich mit der Erweiterung des Emmi-Frameworks um die Unterstützung von LSTM-Layern beschäftigt. Besonderes Augenmerk wurde dabei auf die Implementierung sowohl für den Float- als auch für den quantisierten Datentyp q32 gelegt. Dies ermöglichte die Erweiterung der Funktionalität des Emmi-Frameworks für eingebettete Systeme, insbesondere auf dem RISC-V basierten EMSA5 IP Core.

Wesentliche Bestandteile der Arbeit waren die Entwicklung und Validierung von Prototypen in Python, die anschließende Implementierung im Emmi-Framework und die notwendigen Anpassungen im EmmiTranslator, um TF-Modelle in C-Code zu übersetzen. Die Ergebnisse der Tests mit Float-Daten zeigten eine hohe Übereinstimmung mit den TF-Referenzwerten, wobei die Implementierung eine Toleranzgrenze von 10% einhielt.

Die Herausforderungen bei der Implementierung für quantisierte Daten wurden eingehend untersucht, wobei die Tests zeigten, dass die Größe eines LSTM-Layers sowie die internen und main Bits einen signifikanten Einfluss auf das Verhalten des quantisierten Layers haben. Insbesondere wurden die Gefahr von Overflows und die Genauigkeit der Berechnungen bei der Verwendung von quantisierten Werten behandelt.

5.2 Anforderungen und Forschungsfragen

Die in Kapitel 1 formulierten Anforderungen wurden durch die Beantwortung der Forschungsfragen erfolgreich erfüllt. Der Zusammenhang zwischen den Forschungsfragen und den entsprechenden Kapiteln, in denen sie behandelt wurden, ergibt sich wie folgt:

- Forschungsfrage eins wurde in Kapitel 2 adressiert.
- Die Fragen zwei, drei und fünf wurden in Kapitel 3 behandelt.
- Die Antwort der Fragen vier und sechs finden sich in Kapitel 4.

5.3 Ausblick

Für die Zukunft sind verschiedene Erweiterungen und Verbesserungen denkbar. Beispielsweise könnte die Robustheit der q32-Implementierung durch verbesserte Überlaufkontrollmechanismen erhöht werden. Eine Anpassung der internen und main Bitbreiten könnte zu einer effizienteren Ausführung und weniger Quantisierungsfehlern führen.

Ein weiterer interessanter Ansatz ist die Erweiterung des Frameworks um die Unterstützung anderer Typen rekurrenter neuronaler Netze sowie die Optimierung der bestehenden Implementierung, um die Ausführungsgeschwindigkeit und den Speicherverbrauch zu reduzieren.

Schließlich ist es sinnvoll, das LSTM in realen Anwendungsszenarien zu testen, um seine Leistungsfähigkeit und Praxistauglichkeit zu evaluieren. Solche Studien könnten wertvolle Einblicke in die Stärken und Grenzen der aktuellen Implementierung liefern und zukünftige Forschungs- und Entwicklungsarbeiten anleiten.

A Anhang

Der Anhang besteht aus einem digitalen Archiv, das den während der Arbeit entwickelten Quellcode enthält. Die Struktur des Archivs wird im Folgenden näher erläutert.

Das Archiv ist in vier Bereiche unterteilt:

- **Emmi Core:** Hier finden sich die Implementierungen des LSTM-Layers für den Emmi Core.
- **EmmiTranslator:** In diesem Verzeichnis werden die Codeausschnitte des *model_decoder_tf* präsentiert. Es finden sich hier die Konvertierungsfunktion aus Listing 3.3.2 sowie der in Kapitel 3.3.1 beschreiben *extract_layer_LSTM* Funktion.
- **Emmi Tests:** Dieser Ordner enthält Skripte, die Modelle und Layer Tests für das Emmi-Framework generieren. Zum anderen wird hier der generierte und manuell erstellte C-Code bereitgestellt, der für die Tests benötigt wird.
- **scritps:** Dies ist eine Sammlung der Prototypen in Python und der verwendeten Aktivierungsfunktionen. In einem Unterverzeichnis befinden sich die Skripte, die zur Erzeugung der Diagramme aus Kapitel 4.2 verwendet wurden. Zusätzlich sind auch die generierten Daten für diese Plots hinterlegt.

Es ist wichtig zu erwähnen, dass das Archiv nur Codeausschnitte enthält. Daher sind die meisten Dateien nicht ausführbar und die Pfadangaben sind nicht korrekt. Eine Ausnahme bilden die Skripte, welche die Diagramme aus Kapitel 4.2 erzeugen.

Literatur

[1] Abadi, Martín ; Barham, Paul ; Chen, Jianmin ; Chen, Zhifeng ; Davis, Andy ; Dean, Jeffrey ; Devin, Matthieu ; Ghemawat, Sanjay ; Irving, Geoffrey ; Isard, Michael ; Kudlur, Manjunath ; Levenberg, Josh ; Monga, Rajat ; Moore, Sherry ; Murray, Derek G. ; Steiner, Benoit ; Tucker, Paul ; Vasudevan, Vijay ; Warden, Pete ; Wicke, Martin ; Yu, Yuan ; Zheng, Xiaoqiang: *TensorFlow: A system for large-scale machine learning*. https://arxiv.org/abs/1605.08695. Version: 2016

[2] Andrew Waterman, Krste A.: The RISC-V Instruction Set Manual. (2019), Dezember

[3] Dominik, Andreas: *Vorlesung: LSTM - Long Short-term Memory*. https://youtu.be/9feFTgU5S88?si=3u9QOREwDfRjwuF-, Abruf: 21.07.2024

[4] Faishal, M. ; Mathew, Saju ; Neikha, Kelengol ; Pusa, Khriemenuo ; Zhimomi, Tonoli: The future of work: AI, automation, and the changing dynamics of developed economies. In: *World Journal of Advanced Research and Reviews* (2023). http://dx.doi.org/10.30574/wjarr.2023.18.3.1086. – DOI 10.30574/wjarr.2023.18.3.1086

[5] Falbo, Vincenzo ; Apicella, T. ; Aurioso, Daniele ; Danese, Luisa ; Bellotti, F. ; Berta, Riccardo ; Gloria, A. D.: Analyzing Machine Learning on Mainstream Microcontrollers. (2019), S. 103–108. http://dx.doi.org/10.1007/978-3-030-37277-4_12. – DOI 10.1007/978–3–030–37277–4_12

[6] Ferrag, M. ; Chekkai, Nassira ; Constantine, M. ; Nafa, M.: Securing Embedded Systems: Cyberattacks, Countermeasures, and Challenges. (2015). http://dx.doi.org/10.1201/b19311-12. – DOI 10.1201/b19311–12

[7] Goodfellow, Ian ; Bengio, Yoshua ; Courville, Aaron: *Deep Learning*. MIT Press, 2016. – http://www.deeplearningbook.org

[8] McCurley, E. R. ; Miller, K. R. ; Shonkwiler, R.: Classification power of multiple-layer artificial neural networks. 1294 (1990). http://dx.doi.org/10.1117/12.21208. – DOI 10.1117/12.21208

[9] Moin, Armin ; Challenger, Moharram ; Badii, A. ; Gunnemann, Stephan: Supporting AI Engineering on the IoT Edge through Model-Driven TinyML. In: *2022 IEEE 46th Annual Computers, Software, and Applications*

Conference (COMPSAC) (2021), S. 884–893. http://dx.doi.org/10.1109/COMPSAC54236.2022.00140. – DOI 10.1109/COMPSAC54236.2022.00140

[10] Pfeiffer, Pascal: Development of a Machine Learning Framework for Quantized Neural Networks on Embedded RISC-V Systems. (2023), September

[11] Sepp Hochreiter, Jürgen S.: Long Short-Term Memory. (1997), Dezember

[12] Zhang, Zhaoyun ; Li, Jingpeng: A Review of Artificial Intelligence in Embedded Systems. In: *Micromachines* 14 (2023). http://dx.doi.org/10.3390/mi14050897. – DOI 10.3390/mi14050897

Erklärung

Der Verfasser erklärt, dass er die vorliegende Arbeit selbständig, ohne fremde Hilfe und ohne Benutzung anderer als die angegebenen Hilfsmittel angefertigt hat. Die aus fremden Quellen (einschließlich elektronischer Quellen) direkt oder indirekt übernommenen Gedanken sind ausnahmslos als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden.Zudem bestätigt der Verfasser, dass er den Lehrstuhlleitfaden in der jeweiligen geltenden Fassung gelesen und verstanden hat und sich daher über die gestellten Anforderungen und Bewertungsmaßstäbe im Klaren ist.

23.08.2024, Dresden

Datum, Ort

Jakob Häcker