Untersuchen des TensorFlow Frameworks und Erstellen eines Modells zur Schwingungsanalyse mit μControllern

T3_3101 Studienarbeit

des Studiengangs Maschinenbau, Konstruktion und Entwicklung

an der Dualen Hochschule Baden-Württemberg Stuttgart Campus Horb

von

Paul Rienäcker

02.06.2023

Matrikelnummer, Kurs:	5787298, TMB2020KE1
Ausbildungsfirma:	Hugo Kern und Liebers GmbH & Co. KG
Betreuer/Gutachter der DHBW:	Dipl.-Ing. (FH) Andreas Berndsen

I. Erklärung

Gemäß § 5 (3) der „Studien- und Prüfungsordnung DHBW-Technik“ vom 29. September 2017. Ich versichere hiermit, dass ich meine Bachelorarbeit (bzw. Projektarbeit oder Studienarbeit) mit dem Thema:

„Untersuchen des TensorFlow Frameworks und Erstellen eines Modells zur Schwingungsanalyse mit μControllern“

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Ort, Datum	Unterschrift
Horb, 02.06.2023	Paul Rienäcker

II. Inhaltsverzeichnis

III. Abbildungsverzeichnis

IV. Formelverzeichnis

V. Tabellenverzeichnis

VI. Abstract

Diese Arbeit behandelt eine Methode zur Untersuchung von Lagerschwingungen mit Tensorflow und eine anschließende Implementierung des neuralen Netzwerks auf einem Microcontroller. Es wird zuerst auf Tensorflow im Allgemeinen eingegangen und wie neurale Netzwerke in Tensorflow erstellt werden können. Anschließend wird auf den sog. *Autoencoder* genauer eingegangen und wie dieses neurale Netzwerk helfen kann, normale von anormalen Daten zu unterscheiden. In der zweiten Hälfte steht die Implementierung des Netzwerks auf einem Microcontroller im Mittelpunkt. Dabei werden Methoden aufgezeigt, um die Netzwerkgröße zu reduzieren und einen effektiven Betrieb auf einem Microcontroller zu gewährleisten.

1. Aufgabenstellung

Ziel dieser Arbeit ist es den Tensorflow Framework zu untersuchen und herauszufinden, ob dieser sich für Schwingungsanalysen eignet. Anschließend soll ein neurales Netzwerk auf einem Microcontroller implementiert werden. Mit Schwingungen sind dabei Vibrationen von Maschinen, Motoren oder ähnliches gemeint. Wünschenswert wäre dabei, Aussagen über den Gesundheitszustand eines Wälzlagers nur mittels solcher Schwingungen treffen zu können.

2. Was ist künstliche Intelligenz?

Künstliche Intelligenz oder abgekürzt KI, ist ein Überbegriff für verschiedenste Arten von Computeralgorithmen. Oft wird KI als Synonym für vielerlei Algorithmen verwendet, jedoch ist es wichtig klare Abgrenzungen zu anderen Fachgebieten zu ziehen. Für ein besseres Verständnis werden die Teilgebiete kurz beleuchtet.

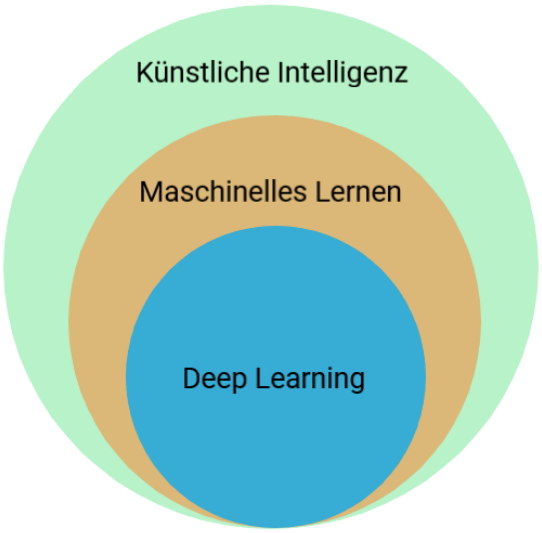

Abbildung 1: Aufteilung intelligenter Algorithmen gemäß [1]

KI wird vom Oxford Lexikon als „Die Fähigkeit von Computern oder anderen Maschinen intelligentes Verhalten zu zeigen oder zu simulieren“ [2] beschrieben. Dabei kann jede Maschine oder Algorithmus eine KI sein, wenn es oder sie intelligentes Verhalten zeigt.

Was jetzt genau intelligentes Verhalten ist, wird aktuell noch erforscht. Nach der Definition sind ein Staubsaugerroboter und ein Charakter in einem Videospiel beides eine künstliche Intelligenz. KI ist jedoch viel verzweigter als das und besteht aus einigen Untergruppen wie aus Abb. 1 ersichtlich ist.

Maschinelles lernen ist eine Untergruppe von KI und beschäftigt sich mit Algorithmen, welche anhand der vorliegenden Datensätze Lösungen entwickeln [3]. Ein Beispiel für solche Algorithmen wäre die Stütz Vektor Maschine, kurz SVM. Sie kann anhand von Eingangsdaten entweder Datengruppen voneinander trennen oder Vorhersagen treffen, wie sich die Daten zueinander verhalten. Siehe Klassifikation und Regression.

Die letzte Gruppe ist das sog. *Deep Learning* (de. Tiefes Lernen). Hier werden Aspekte des maschinellen Lernens aufgegriffen und erweitert. Deep Learning Algorithmen basieren auf dem Vorbild des biologischen Gehirns und verarbeiten Daten in hintereinander geschachtelten Netzwerken [4]. Diese Algorithmen werden meist für sehr komplexe Anwendungen wie Sprach-/ oder Bilderkennung verwendet. Durch die vielen hintereinander geschalteten Schichten sind *Deep Neural Networks* (de. Tiefe Neurale Netzwerke), kurz DNN, in der Lage aus Rohdaten wichtige Informationen zu extrahieren und Vorhersagen zu treffen. DNNs haben zwischen Eingang und Ausgang Zwischenschichten sog. *Hidden Layers* (de. Versteckte Schichten, Zwischenschichten). Sie sind für die Abstraktion verantwortlich und geben dem Netzwerk die Fähigkeit komplexe Probleme zu lernen.

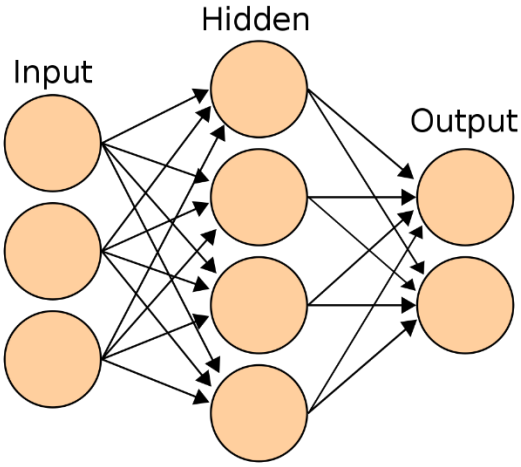

Abbildung 2: Einfaches Deep Neural Network mit einer Zwischenschicht

2.1 Aufbau neuraler Netzwerke

Jede Schicht in einem neuralen Netzwerk besteht aus Neuronen. Diese Neuronen sind, in einfachen neuralen Netzwerken, mit jedem Neuron der vorherigen und nachfolgenden Schicht verbunden. In sequenziellen Netzwerken laufen die Daten schrittweise von der Eingangsschicht bis zur Ausgangsschicht. Jedes Neuron ist eine lineare Funktion der Form:

$$f(x) = \sum(wi*xi) + b$$

Formel 1: Lineare Funktion einer Schicht

wobei w für die Gewichtung und b für die vertikale Verschiebung (oft auch *Bias* genannt) steht. Die Anzahl an Gewichtungen ist abhängig von der Anzahl an Neuronen in der vorhergehenden Schicht. In dem Netzwerk aus Abb. 2 würde die Funktion für die Ausgangsschicht folgendermaßen lauten:

$$f3(x) = w1*x1 + w2*x2 + b$$

Formel 2: Lineare Funktion an einem konkreten Beispiel

Diese Funktion ist eine simple lineare Funktion. Damit neurale Netzwerke auch nicht lineare Probleme erlenen können, braucht es sog. Aktivierungsfunktionen.

Aktivierungsfunktionen nehmen als Eingang den Funktionswert der linearen Funktion und geben den Ausgangswert gemäß der Funktion wieder:

$$y = func(f(x))$$

Formel 3: Aktivierungsfunktion

Es gibt verschiedenste Aktivierungsfunktionen, welche auch speziell für verschiedenste Probleme verwendet werden. Am bekanntesten sind die ReLu, Sigmoid, Softmax und Tanh Funktionen. Welche Funktion am besten funktioniert muss man oft durch Versuche ermitteln [5].

2.2 Trainieren von neuralen Netzen

Das Training von neuralen Netzwerken kann man drei Hauptklassen einteilen. *Supervised Learning* (de. Beaufsichtigtes Lernen), *Unsupervised Learning* (de. Unbeaufsichtigtes Lernen) und *Reinforcement Learning* (de. Bestärktes Lernen) [6].

Supervised Learning wird sehr oft eingesetzt, wenn klare, beschriftete Daten vorliegen und das Netzwerk die Korrelation zwischen den Daten und ihrer Beschriftung erlernen soll. Wenn ein Netzwerk beispielsweise anhand von Bildern entscheiden soll, ob darauf ein Hund oder eine Katze zu sehen ist, ist das ein klarer Fall von Supervised Learning. Im Training wird die Ausgabe des Netzwerks immer wieder mit den *Labels* (de. Beschriftungen) der Bilder verglichen. Dabei verändert das Netzwerk die Gewichtungen und den Bias bei jeder Iteration und erhöht so die Klassifizierungsgenauigkeit.

Bei Unsupervised Learning wird dem Netzwerk keine klare Aufgabe zugeteilt. Hier versucht das Netzwerk nützliche Informationen aus dem Datensatz zu extrahieren, Muster zu erkennen oder Anomalien zu finden.

Reinforcement Learning ist eine Mischung aus Supervised und Unsupervised Learning. Das Netzwerk versucht die Lösung für ein Problem selbstständig zu finden, indem es für richtige Entscheidungen belohnt und für Fehler bestraft wird. Diese Trainingsmethode wird gerne bei Problemen angewendet, die ein klares Ziel haben, jedoch der Weg dahin unbekannt ist. Beispiele dafür wäre das Training eines Schachroboters oder autonomes Fahren.

3. Der Tensorflow Framework

„Tensorflow ist eine […] offene Plattform für maschinelles lernen.“ [7]. Mit Tensorflow können neurale Netzwerke jeder Art erstellt, trainiert und evaluiert werden. Tensorflow wurde vom Google Brain Team entwickelt und wurde eine Zeit lang im Unternehmen für Forschungszwecke und zur einfacheren Erstellung neuraler Netze verwendet [7].

Abbildung 3: Tensorflow Logo

Der Name Tensorflow setzt sich zusammen aus *Tensor* und *Flow*. *Tensor* (de. Tensoren) sind mehrdimensionale Matrizen, mit denen eine Vielzahl an Operationen durchgeführt werden können. In Tensorflow werden Daten in solchen Tensoren gespeichert und nach Bedarf Operationen durchgeführt. *Flow* (de. Fluss) bezieht sich auf den Datenfluss und die Art und Weise wie Tensorflow Berechnungen bzw. Operationen durchführt. Operationen werden in Tensorflow in einem Flussdiagramm dargestellt und ausgeführt [8].

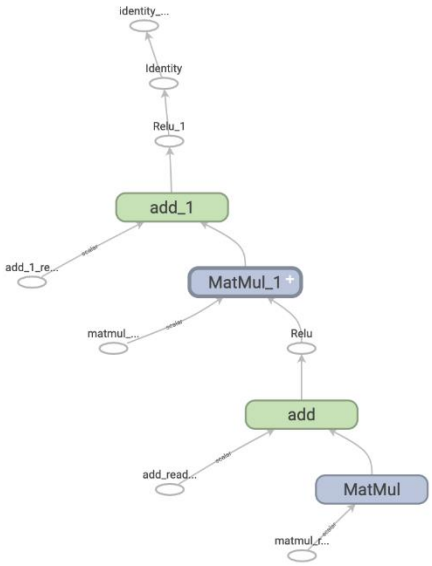

Abbildung 4: Tensorflow Flussdiagramm eines neuralen Netzwerks mit 2 Schichten

Das bietet gleich mehrere Vorteile. Zum einen ist Tensorflow dadurch flexibler und kann auf mehreren Plattformen wie C++, Python, aber auch Smartphones, eingebettete Systemen oder Server laufen [8]. Dies ist auch einer der Hauptgründe warum Tensorflow so beliebt ist. Es gibt neben Tensorflow auch viele andere Frameworks zur Erstellung neuraler Netzwerke, wie z.B. PyTorch oder Matlab. Diese haben jedoch eine limitierte Auswahl an Plattformen, weshalb sie in der Industrie eher seltener verwendet werden [9].

Zum anderen können Operationen in Tensorflow durch den flussartigen Ablauf, sehr schnell und parallelisiert ablaufen [8]. Geschwindigkeit ist unglaublich wichtig für neurale Netze, da sehr viele Rechenoperationen beim Training durchgeführt werden müssen. Gerade bei sehr großen Netzwerken kann das Training schnell viel Zeit und Geld in Anspruch nehmen. Daher ist es wichtig Operationen so effizient und schnell wie möglich auszuführen.

4. Die Keras Bibliothek

Tensorflow ist ein Framework, um verschiedenste Rechenoperationen an Tensoren durchzuführen. Es ist möglich in Tensorflow selbst, neurale Netzwerke zu erstellen und zu trainieren, jedoch ist das mit viel Aufwand verbunden. Jede einzelne Operation muss dafür vom Benutzer eigens implementiert werden.

Hier kommt die Keras Bibliothek ins Spiel. Sie wurde erstmals am 28. März 2015 veröffentlicht und ist seit 02. November 2017 ein fester Bestandteil des Tensorflow Frameworks [10]. Keras macht es durch viele seiner APIs leicht, neurale Netzwerke zu entwerfen und zu trainieren. Wie auch Tensorflow ist Keras in C++ programmiert und kann über die Python API angesteuert werden.

Um ein Deep Neural Network zu erstellen, reichten nur wenige Befehl aus, wobei alle restlichen Operationen von Keras im Hintergrund übernommen werden.

Wenn beispielsweise das Netzwerk aus Kap. 2 in Keras erstellen werden soll, sieht das folgendermaßen aus:

```
from keras.layers import Dense
from keras.models import Sequential
# Import der Bibliotheken

def DNN():                          # Definieren der Funktion
    model = Sequential()            # Art des Modells das verwendet wird
    model.add(Dense(3))             # Schicht mit 3 Neuronen
    model.add(Dense(4))             # Schicht mit 4 Neuronen
    model.add(Dense(2))             # Schicht mit 2 Neuronen
    return model

model = DNN()                       # Aufruf der Funktion
```

Neurale Netzwerke können mit Keras sehr einfach und übersichtlich gestaltet werden. Eine Schicht wird mit model.add() hinzugefügt, wobei die Art der Schicht in der Funktion definiert wird. In diesem Beispiel soll eine *Dense*-Schicht hinzugefügt werden. Eine Dense Schicht ist die Standardschicht für neurale Netzwerke, wie bereits in Kapitel 2.1 erklärt wurde. Die Zahl definiert hierbei die Anzahl an Neuronen in dieser Schicht.

5. Der Autoencoder

Um Schwingungsanalysen an Daten durchzuführen, gibt es mehrere Möglichkeiten. In den Anfängen des maschinellen Lernens wurden gerne simplere Methoden, wie beispielsweise die SVM, verwendet, um Daten zu klassifizieren. Im Laufe der Zeit finden jedoch vermehrt Deep Learning Methoden Anwendung. Grund dafür ist zum einen die gestiegene Rechenleistung von Computern und besserer Softwareunterstützung durch Tensorflow bzw. Keras und zum anderen die sehr hohe Flexibilität und Genauigkeit von Deep Learning Methoden [11].

Ein konkretes Beispiel für eine Deep Learning Methode ist der sog. Autoencoder. Autoencoder sind neurale Netzwerke, die versuchen eingehende Daten möglichst genau zu kopieren [12].

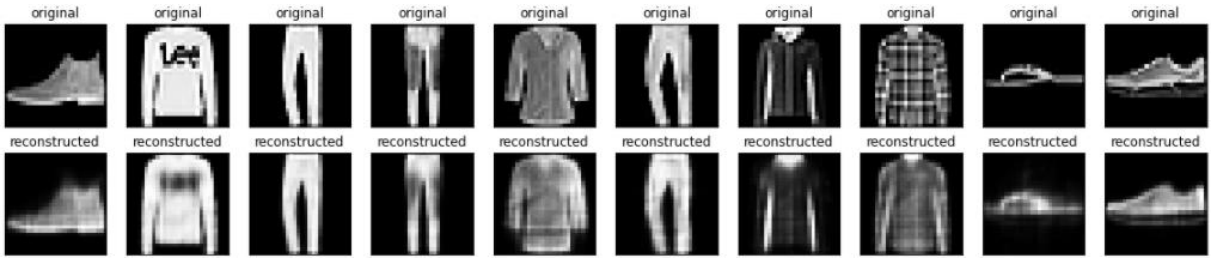

Abbildung 5: Funktionsweise eines Autoencoders

In Abb. 5 kann die Funktionsweise eines Autoencoders betrachtet werden. Man sieht deutlich, wie das Netzwerk versucht die Eingangsdaten (original) nachzubilden (reconstructed). Ein Autoencoder besteht aus einer Eingangsschicht, einer kleineren Zwischenschicht und einer Ausgangsschicht. Zwischen Eingangs-/ bzw. Ausgangsschicht können weitere Zwischenschichten eingebracht werden, um die Genauigkeit des Autoencoders zu verbessern [13].

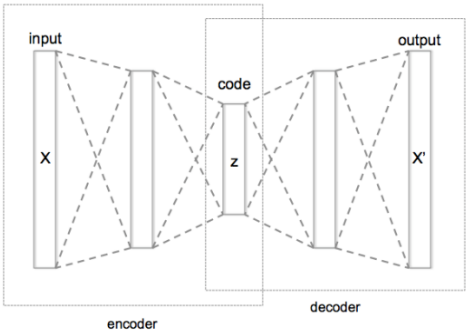

Abbildung 6: Aufbau eines Autoencoders mit mehreren Zwischenschichten

Der Autoencoder hat im Gegensatz zu traditionellen Methoden wie die SVM einen sehr großen Vorteil. Es müssen keine Merkmale aus dem Datensatz extrahiert werden mit denen das Netzwerkt trainiert [13]. SVM oder andere Algorithmen können meist nur schwer mit Rohdaten arbeiten. Damit sie gut performen muss man eine Vorauswahl durchführen, in der die Daten gesäubert und unwichtige Informationen aussortiert werden. Autoencoder führen diesen Schritt von allein durch. Durch die Architektur des Netzwerks und die Zwischenschichten werden nur die Daten weitergereicht, die für die Nachbildung wichtig sind. So sortieren Autoencoder von selbst unwichtigen Daten aus und müssen nicht mit manuell selektierten Daten trainieren [13]. In der Schwingungsanalyse können Autoencoder somit direkt mit den Schwingungen trainiert werden und man muss keinen komplexen Algorithmus entwickeln, um Daten zu säubern.

5.1 Trainieren von Autoencodern

Autoencoder werden sehr gerne für die Erkennung von Anomalien eingesetzt [11]. Dafür trainiert man sie ausschließlich mit normalen Daten. Wenn ein Autoencoder mit normalen Daten trainiert wird, versucht er diese Daten möglichst genau zu kopieren. In jeder Iteration, die das Netzwerk durchläuft, werden die Eingangsdaten mit den Ausgangsdaten verglichen und der mittlere absolute Fehler, kurz *MAE* (eng. Mean Absolute Error) ermittelt. Die Daten werden hier jeweils voneinander abgezogen und gemittelt. Das Netzwerk versucht diesen Fehler gegen null zu korrigieren.

Wenn das Netzwerk nun eine Schwingung bekommt, die es noch nie gesehen hat, beispielsweise mit anderer Frequenz, dann muss zwangsläufig der Rekonstruktionsfehler bei dieser Schwingung, im Vergleich, sehr hoch sein.

Bei Schwingungen, wie in einem Wälzlager, können wir somit den Autoencoder im Normalbetrieb trainieren und falls das Lager in Zukunft Ausfallserscheinungen zeigt, wird der Autoencoder einen Fehler anzeigen. Es braucht somit keine anormalen Schwingungen, mit denen wir das Netzwerk trainieren. Das ist vor allem von Vorteil, wenn solche Daten nicht zur Verfügung stehen, wie im Beispiel des Wälzlagers.

Die einzige Variable, die hierbei ermittelt werden muss, ist ab welchem MAE eine Schwingung als normal bzw. anormal gilt. In der Realität ist das nicht immer leicht, da solche Daten nicht zur Verfügung stehen, jedoch können sie experimentell ermittelt werden. Eine andere Möglichkeit besteht darin, jeden Fehler über dem Trainings-MAE als Fehler zu deklarieren. Dabei kann aber sein, dass auch unbedenkliche Schwingungen als Fehler erkannt werden, wobei man den MAE iterativ anpassen muss.

In diesem Fall stehen anormale Daten zur Verfügung und daher kann später genau ermittelt werden, wie gut das Netzwerk bei normalen und anormalen Daten performt.

6. Das NASA Bearing Dataset

Das *NASA Bearing Dataset* ist die Grundlage dieser Arbeit und wird im späteren Verlauf für das Training sowie die Validierung der Netzwerke verwendet [14]. Es wurde sich im Rahmen dieser Arbeit dafür entschieden, da es die Schwingungen eines Wälzlagers bis zum Ausfall dokumentiert. Dabei wurden von insgesamt vier Wälzlagern alle 10 min für eine Sekunde Schwingungen mit 20480 Hz Auflösung aufgenommen. Die Sensoren sind stationär neben den Lagern angebracht.

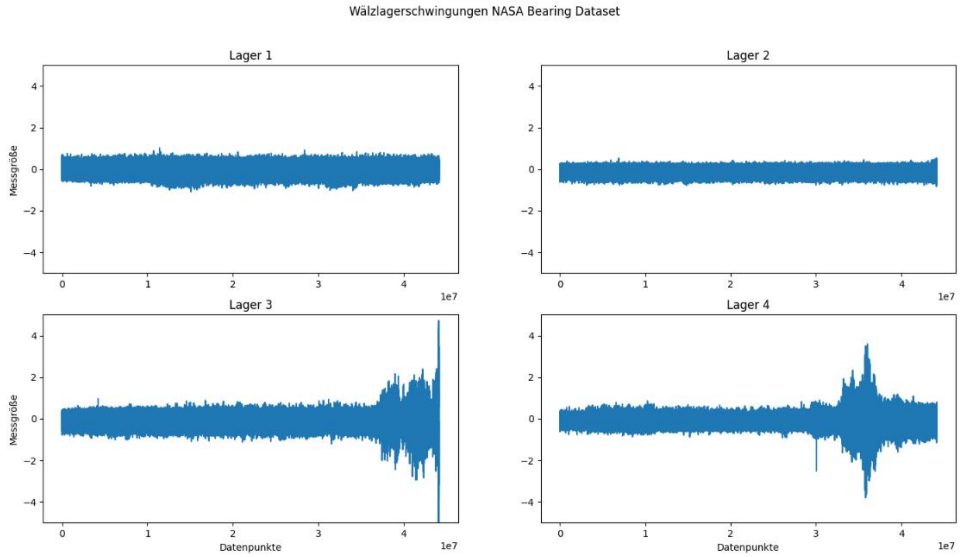

Abbildung 7: Visualisierung der Lagerschwingungen

In Abb. 7 kann man die Schwingungen von allen Lagern sehen. Es ist eindeutig, dass Lager drei und vier während des Tests stark beschädigt wurden. Die Beschleunigungswerte zeigen die starke Unwucht, die von den Lagern gegen Ende der Testphase ausging. Lager zwei hat hingegen nicht so starken Schaden genommen. Es ist nur ein leichter Anstieg der Beschleunigungswerte erkennbar. Genau das macht Lager zwei so interessant. Die Frage ist, ob der Autoencoder es schafft, auch nur sehr kleine Schwingungsänderungen als Anomalie zu identifizieren. Wenn er bei Lager zwei eine Anomalie entdeckt, sind Lager drei und vier kein Problem. Daher wird im folgenden Verlauf mit Lager zwei gearbeitet.

7. Datenvorbereitung

7.1 Zusammenfassen der Daten

Bevor ein Netzwerk erstellt und trainiert werden kann, müssen die Daten für das Netzwerk vorbereitet werden. Das NASA Bearing Dataset besteht aus vielen tausend Dateien, die sich in dem entsprechenden Ordner befinden. Sie müssen zuerst in eine einzige Datei zusammengefasst werden, bevor weitere Änderungen vorgenommen werden können. Hierfür wird schrittweise über jede einzelne Datei iteriert, der Inhalt gelesen und in eine neue Datei geschrieben.

Als neues Dateiformat soll eine CSV-Datei verwendet werden. Es stellt sich jedoch heraus, dass die Dateigröße bei CSV-Dateien dadurch sehr groß wird. Aus diesem Grund wird sich für das HDF5 Dateiformat entschieden. Es wird standardmäßig von der Pandas Bibliothek unterstützt und kann durch seine effiziente Kompression die Dateigröße relativ klein halten.

Die fertige Datei besitzt 44.154.880 Zeilen und hat eine Dateigröße von etwa 3.1 GB.

7.2 Transformieren der Daten

Damit später die Performance des Netzwerks überprüft werden kann, muss man die Daten aufteilen. In den meisten Fällen wird ein Trainings-/ und ein Testdatensatz erstellt. Mit dem Trainingsdatensatz wird das Netzwerk später trainiert und der Testdatensatz ist später zum Überprüfen der Netzwerkperformance. Als Faustregel teilt man Trainings-/ und Testdatensatz in 80/20 auf, jedoch gibt es hier keine klare Formel, sondern nur grobe Richtwerte [15]. Zusätzlich dazu muss man auch die anormalen Daten einer Gruppe zuordnen, damit später die Performance des Netzwerk ermittelt werden kann.

Die Daten befinden sich immer noch in einem unpraktischen Format und müssen daher noch weiter umgewandelt werden. Aktuell besitzen die Daten Millionen Zeilen und eine Spalte. Hier muss man sich nun auch entscheiden, wie groß später das Netzwerk werden soll. Die Dimensionsgröße der Eingangsdaten ist untrennbar mit der Netzwerkarchitektur verbunden. Dabei spielen Faktoren wie Speicherplatz und Rechenleistung eine große Rolle.

Größere Netzwerke sind bei genügend Rechenleistung ratsam, da sie Operationen meist schneller ausführen können. Wenn die Rechenleistung oder der Speicherplatz begrenzt sind, muss eine kleinere Netzwerkgröße und somit auch ein kleineres Paket an Eingangsdaten gewählt werden. Da dieses Projekt später auf einem Microcontroller implementiert werden soll, darf die Netzwerkgröße nicht zu groß sein.

Eingangsdaten für neurale Netzwerke werden i.d.R. als Vektoren, bzw. bei mehr Dimensionen als Tensoren eingegeben. Bei Schwingungen haben wir nur Beschleunigungswerte über die Zeit. Die Eingangsdaten können somit als Vektor der From [i , j] dargestellt werden, wobei i die Zeitachse und j die Anzahl der Schwingungsdatenpunkte ist.

Im Test hat sich gezeigt, dass Eingangsdaten mit einer Größe von [1,5120] und 25.8 Mio. trainierbaren Parametern, zu einer Netzwerkgröße von etwa 300 MB führt. Der EPS32 Microcontroller hat jedoch nur einen Speicher von 4 MB. Das Netzwerk ist somit viel zu groß.

Daher wird sich hier für einen Eingangsgröße von [1,128] entschieden. Das Netzwerk hat dadurch nur noch eine Anzahl von 38.700 trainierbaren Parametern und somit eine Größe von etwa 1 MB. Somit ist die gewählte Netzwerkgröße optimal für einen Microcontroller.

Wie zuvor wird der Datensatz nun geändert. Dazu wird über den gesamten Datensatz iteriert und in eine Matrix mit den Dimensionen [i,128] geändert. Die Trainings-/, Test und Anormalen Datensätze werden abgespeichert und später wieder verwendet.

7.3 Normalisieren der Daten

Im Bereich der Statistik oder des maschinellen Lernens kommt es oft vor, dass man Daten normalisiert. Normalisierung verändert die Daten so, dass sie für das Netzwerk bessere statistische Eigenschaften haben. Im besten Fall kann dadurch das Training beschleunigt und die Genauigkeit des Netzwerks verbessert werden [16].

Es gibt viele verschiedene Normalisierungsverfahren, wobei es auch hier wieder nur grobe Richtlinien für die Verwendung gibt. Da sich die Daten hier über einen Bereich gleichmäßig verteilen, wird sich an dieser Stelle für die Min-Max Normalisierung entschieden [17]. Dabei skaliert man alle Datenpunkte zwischen den Werten -1 und 1 gemäß der Formel:

$$x' = 2 * (x - x_{min}) / (x_{max} - x_{min}) - 1$$

Formel 4: Normalisierung

Hierfür wird zum ersten Mal Tensorflow selbst verwendet, da es spezielle Operatoren für diese Aufgabe zur Verfügung stellt.

Die Vorbereitung der Daten ist hiermit abgeschlossen und das Netzwerk kann nun erstellt werden.

8. Die Netzwerkarchitektur

Der wohl wichtigste Aspekt künstliche neuralen Netzwerke ist die Netzwerkarchitektur. Sie bestimmt, wie die einzelnen Schichten miteinander interagieren und hat einen sehr großen Einfluss auf die Gesamtperformance des Netzwerks. In Kapitel 5. wurde der Autoencoder bereits in seinen groben Zügen charakterisiert. Hier wird nun versucht, das eigentliche Netzwerk in Keras zu definieren.

Die Erstellung eines tiefen neuralen Netzwerks ist kein statischer Prozess. Es gibt keine klaren Regeln oder Vorschriften wie ein Netzwerk aussehen kann. Aus Erfahrung geht hervor, dass die richtige Architektur oft durch iterative bzw. empirische Prozesse ermittelt werden muss. Dabei haben Faktoren wie die Anzahl der Schichten, Anzahl der Neuronen in einer Schicht, Gewichtsinitialisierung, Aktivierungsfunktion und Optimierer einen großen Einfluss. Es ist ratsam mit kleinen Netzwerken zu beginnen und schrittweise mehr Schichten hinzuzufügen. Wenn das Netzwerk zu wenig Neuronen hat, ist es nicht in der Lage das Problem zu erlernen, *Underfitting* (de. Unteranpassung) ist die Folge. Wenn es zu viele Neuronen hat, besteht die Gefahr des *Overfitting* (de. Überanpassung). Das Netzwerk hat die Daten „auswendig“ gelernt und ist nicht in der Lage generelle Schlüsse aus den Daten zu ziehen [18]. Daher muss mit dem Testdatensatz geprüft werden, wie das Netzwerk bei unbekannten Daten performt. Bei Schwingungsanalysen handelt es sich um ein nicht sehr komplexes Problem und die grobe Architektur eines Autoencoders ist ebenfalls bekannt.

Da die Daten aus einer Matrix mit 128 Spalten bestehen, muss die erste Schicht des Autoencoders ebenfalls 128 Neuronen besitzen. Es ist generell ratsam die Größe der Zwischenschichten langsam zu reduzieren, um die Genauigkeit zu verbessern. Im Rahmen der Schwingungsuntersuchungen wird sich hier für folgende Netzwerkarchitektur entschieden:

```
input_shape = (1,128)

def AutoEncoder():
    model = Sequential()
    # Encoder ----------------------------------------------------------------
    model.add(Dense(128, activation='tanh', input_shape=input_shape))
    model.add(Dense(64, activation='tanh'))
    model.add(Dense(32, activation='tanh'))
    model.add(Dense(16, activation='tanh'))
    model.add(Dense(8, activation='tanh'))
    model.add(Dense(4, activation='tanh'))
    model.add(Dense(2, activation='tanh'))
    # Decoder ----------------------------------------------------------------
    model.add(Dense(4, activation='tanh'))
    model.add(Dense(8, activation='tanh'))
    model.add(Dense(16, activation='tanh'))
    model.add(Dense(32, activation='tanh'))
    model.add(Dense(64, activation='tanh'))
    model.add(Dense(128, activation='tanh'))
    model.summary()
    return model

model = AutoEncoder()
```

Der Encoder reduziert die Dimension von 128 auf nur 2 Neuronen. Dabei wird die Anzahl der Neuronen in jeder Schicht halbiert. Der Decoder versucht aus den 2 Neuronen die Eingangsdaten des Encoders nachzubilden. Die Modellzusammenfassung zeigt uns wie viele trainierbare Parameter das Netzwerk besitzt und wie sie verteilt sind.

Als Aktivierungsfunktion wird die Tanh bzw. der Tanges hyperbolicus verwendet. Da wir die Daten zwischen -1 und 1 normalisiert haben, ist die Tanh-Funktion bestens dafür geeignet.

9. Das Training

Der Autoencoder wird nun über 1000 Epochen trainiert. Eine Epoche enthält den kompletten Trainingsdatensatz. Nach einer Epoche wird der Testdatensatz, hier auch Validierungsdatensatz genannt, durch das Netzwerk geschickt. Dabei wird der MAE für den Trainings-/ und Testdatensatz errechnet. Im optimalen Fall sinken beide Werte im Laufe des Trainings ab. Wenn jedoch der MAE des Testdatensatzes im Training nicht mehr sinkt, sondern ansteigt, ist das ein klarer Fall von Überanpassung und das Training muss abgebrochen werden.

Das Training wird auf einer GTX 1080 Grafikkarte durchgeführt und dauert ca. 18 min. Für das Netzwerk sieht der Trainingsverlauf folgendermaßen aus:

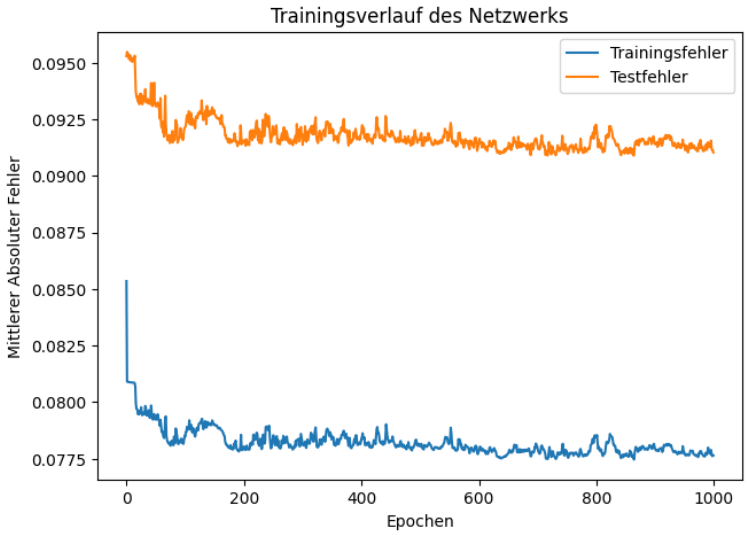

Abbildung 8: Trainingsverlauf des neuralen Netzwerks

In Abb. 8 kann man sehr gut erkennen, wie der Fehler über die Zeit abnimmt. Zudem sinkt auch der Test-/Validierungsfehler ab, was bedeutet, das Netzwerk kann auch auf unbekannten Daten gute Vorhersagen treffen.

Das Netzwerk hat somit ein hohes Maß an Generalisierung erworben und hat sich nicht Überangepasst. Somit ist das Netzwerk bereit für eine Evaluierung.

10. Die Evaluation

Nun kann anhand verschiedener Tests die Performance des Netzwerks evaluiert werden. Dafür stellt man die Orginaldaten mit den rekonstruierten Daten gegenüber, ähnlich wie in Kapitel 5. Es wird ein kurzer Abschnitt dem Testdatensatz entnommen und durch das Netzwerk geschickt. Die rekonstruierten Testdaten kann man nun mit den originalen Testdaten vergleichen. Zudem werden auch die anormalen Daten durch das Netzwerk geschickt und die anormalen Originaldaten mit der anormalen Rekonstruktion verglichen.

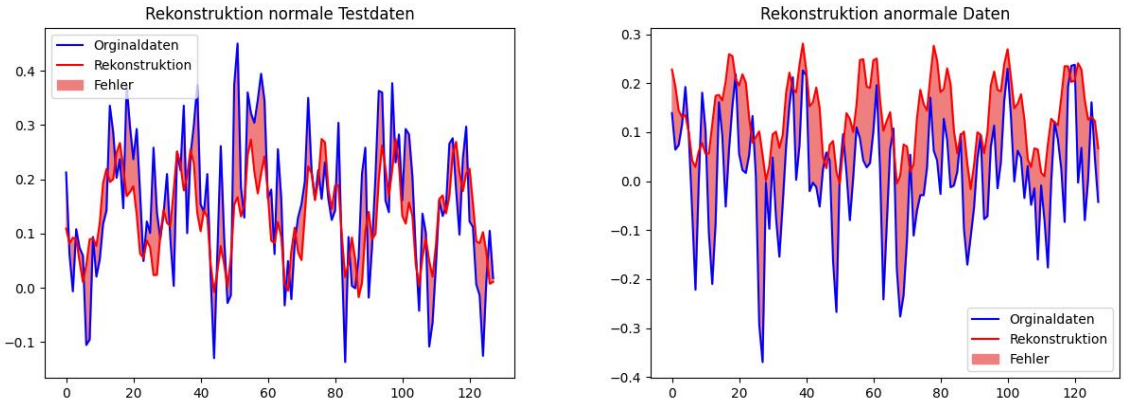

Abbildung 9: Rekonstruktionsfehler Testdaten (links) und anormale Daten (rechts)

Abb. 9 zeigt den Rekonstruktionsfehler zwischen den Originaldaten und den rekonstruierten Daten. Links befindet sich die Rekonstruktion mit normalen Testdaten. Man kann deutlich erkennen, wie das Netzwerk versucht die Daten zu imitieren. Der Fehlerbereich zwischen dem Original und der Rekonstruktion ist hier vergleichsweise gering. Auf der rechten Seite befindet sich die Rekonstruktion mit anormalen Daten. Wenn man die roten Bereiche miteinander vergleicht, fällt schnell auf, dass die anormalen Daten einen höheren Rekonstruktionsfehler besitzen als die Testdaten. Daraus lässt sich schlussfolgern, der Autoencoder produziert einen unterschiedlichen Rekonstruktionsfehler bei normalen und anormalen Daten.

Nun muss der Autoencoder auf dem gesamten Datensatz getestet werden. Hierfür rekonstruiert das Netzwerk eine [7000,128] große Matrix. Der MAE berechnet sich aus den mittleren Fehlern jeder Zeile, so ergibt sich ein 7000 Zeilen großer Vektor mit MAE-Werten. Auf den Wert von 7000 wird sich im Rahmen der Fehleruntersuchung entschieden, da der Datensatz an anormalen Daten etwas mehr als 7000 Zeilen umfasst.

Die MAE-Vektoren können nun in einem Histogramm dargestellt werden.

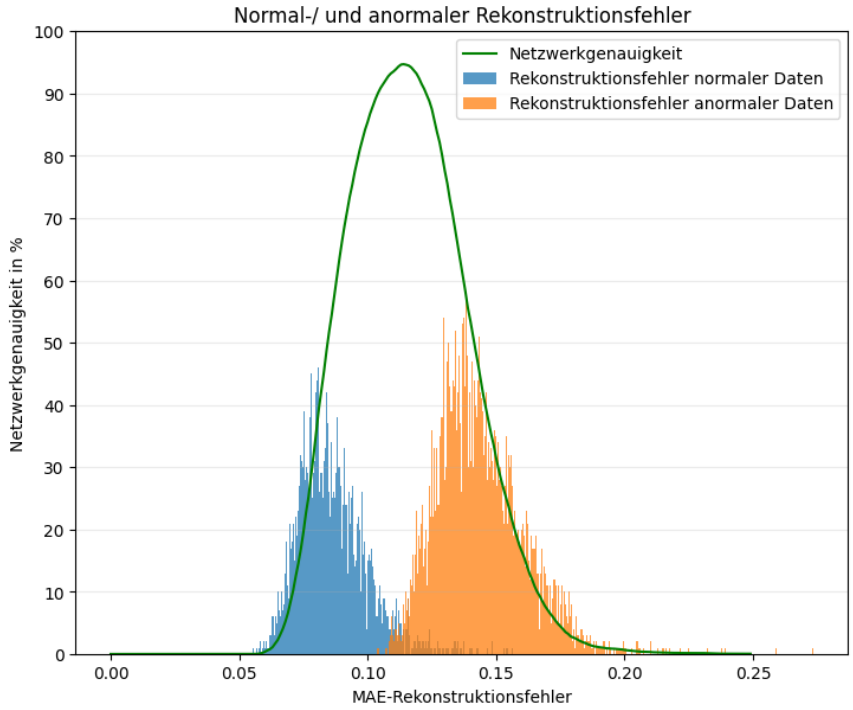

Abbildung 10: Rekonstruktionsfehler zwischen normalen und anormalen Daten mit Netzwerkgenauigkeit

Aus der Analyse geht hervor, dass das Netzwerk normale von anormalen Daten sehr gut unterscheiden kann. Es gibt nur eine kleine Überlappung zwischen den beiden Verteilungen. Man kann hier sehr gut den geforderten Grenzwert zwischen normalen und anormalen Daten ablesen. Wie bereits in Kapitel 5.1 besprochen, braucht es einen Fehlergrenzwert zwischen Rekonstruktion und Orginaldaten, ab dem das Netzwerk die Daten als anormal identifiziert. Der optimale Grenzwert liegt bei 0,114. An diesem Punkt ist der Klassifikationsfehler am geringsten.

Bei genauerer Untersuchung geht hervor, dass das Netzwerk zu 95,6% normale Daten und 98,8% anormale Daten korrekt klassifiziert. Die Gesamtgenauigkeit beläuft sich somit bei einem Grenzwert von 0,114 auf 94,7%.

Natürlich kann man nicht vernachlässigen, dass auch anormale Daten als normale und normale Daten als anormal klassifiziert werden. Es stellt sich die Frage, warum überschneiden sich die beiden Histogramme?

Wie aus der Auswahl des Datensatzes bekannt ist (Kapitel 6), wurde sich hier für das Lager zwei entschieden. Ziel war es zu testen, ob der Autoencoder auch kleine Anomalien erkennen kann. Faktoren wie die limitierte Netzwerkgröße könnten zudem eine Ursache sein.

Es ist nicht ratsam nur auf das Histogramm zu schauen. Um einen klareren Blick zu erhalten, sollte man sich vor Augen führen, was das Netzwerk eigentlich leistet. Als Eingangsdaten bekommt es 128 Datenpunkte. Die Lager, die im Rahmen des Datensatzes untersucht werden, drehen mit 2000 1/min. Wie weit sich das Lager in 128 Messungen dreht, kann nach der folgenden Formel berechnet werden:

$$U[1/min] = 2000/60/20480*128$$

Formel 5: Umdrehungen in Abfragezeit

Aus Formel 5 geht hervor, dass das Lager sich in 128 Messungen nur 0,208 Umdrehungen gedreht hat. Somit kann das Netzwerk nur anhand von einem Fünftel einer ganzen Rotation eine derart hohe Klassifizierungsgenauigkeit generieren. Das ist sehr beeindruckend!

Anhand der vorliegenden Daten lässt sich schlussfolgern, dass der Autoencoder mit hoher Sicherheit einen Lagerdefekt frühzeitig erkennen würde.

11. Convolutional Neural Networks

Im Rahmen von neuralen Netzwerken und Schwingungsanalysen mit Tensorflow gibt es eine zweite Kategorie an Netzwerken, die von Interesse sind. Die Rede ist von *Convolutional Neural Networks* (eng. Faltende neurale Netzwerke).

In einem Convolutional Neural Network, kurz CNN, kommen sog. Faltungsschichten zum Einsatz. Sie agieren ähnlich wie der visuelle Kortex bei Tieren [19]. Durch die Multiplikation mit einer Faltungsmatrix können Bildern Kanten und Formen entnommen werden. Dadurch sind CNNs in der Lage abstrakte Formen auf Bildern zu erkennen und damit zu lernen [20].

CNNs können im Gegensatz zu Netzwerken ohne Faltungsschicht besser Visuelle Probleme erlernen und sortieren besser unwichtige Informationen aus.

Diese Netzwerke sollen in der Lage sein gute Vorhersagen bei Schwingungsuntersuchungen treffen zu können [11, 21]. Im Laufe der Untersuchungen wurde sich aber gegen diesen Ansatz entschieden. Die Gründe hierfür werden nun genauer beleuchtet.

11.1 Genauigkeit der bisherigen Architektur

Die bisherige Genauigkeit des normalen Netzwerks ohne Faltungsschicht ist bereits hinreichend genau. Wie in Kapitel 10. beschrieben, ist die Netzwerkgenauigkeit auf einem akzeptablen Niveau. Eine neue Netzwerkarchitektur ist vermutlich nur schwer in der Lage eine so hohe Genauigkeit zu erzielen. Dabei kann es ebenfalls sein, dass das Netzwerk keine bessere Genauigkeit erreicht und vermutlich schlechter performt als das bisherige Netzwerk.

11.2 Aufwendige Datenvorbereitung

CNN sind visuelle Netzwerke und werden sehr gerne in der Bilderkennung eingesetzt [22]. Das bedeutet die Daten aus den NASA Bearing Dataset müssten zunächst aus der Frequenzebene in die Bildebene überführt werden. Die Methode dahinter wäre eine *Short Time Fast Furier Transformation (STFFT)* bzw. ein Spektrogramm.

Zudem müssen die Daten für das Training in 3-dimensionale Tensoren umgewandelt werden damit sie das Netzwerk lesen kann. Das alles würde sehr viel Programmieraufwand bedeuten. Durch die STFFT könnte sich die Rechenzeit auf einem Microcontroller zusätzlich erhöhen.

11.3 Lange Trainingszeit von CNN

Netzwerke mit Faltungsschichten haben eine weitaus längere Trainingsdauer als Netzwerke ohne diese Schichten. Ein Vergleich ist hierfür das Netzwerk aus [21]. Hier wird beschrieben, dass das Netzwerk in einer Zeit von 8h trainiert wurde. Im Vergleich wurde das eigens entwickelte Netzwerk nur für etwa 18 min. trainiert. Die Genauigkeit zwischen einem CNN und der Architektur aus Kapitel 8. ist trotzdem vergleichbar gut [21].

11.4 Geringeren Speicherbedarf

Das CNN Netzwerk das in [21] vorgestellt wird benötigt, nachgebaut in Keras, ca. 20 MB an Speicherbedarf. Die in diesem Bericht erarbeitete Architektur benötigt nur etwa 1 MB an Speicher. Somit kann das eigens entwickelte Netzwerk auch in speicherlimitierten Umgebungen funktionieren.

12. TensorFlow Lite

TensorFlow Lite (TF Lite) ist eine spezielle Version der TensorFlow-Bibliothek, die für mobile Geräte und eingebettete Systeme optimiert ist. Es ist eine Abkürzung für "TensorFlow Lightweight" und wird verwendet, um intelligente Algorithmen auf ressourcenbeschränkten Geräten mit begrenzter Rechenleistung und Speicherplatz zu betreiben [23].

Hierfür steht eine Reihe von Tools und APIs zur Verfügung, mit denen Entwickler Modelle auf mobile Geräte exportieren, optimieren und ausführen können. Es unterstützt sowohl CPUs als auch GPUs, einschließlich dedizierter KI-Hardware wie TPUs. TF Lite bietet auch Funktionen für die Modellkonvertierung, die Komprimierung und die Quantisierung, um die Größe und Rechenleistung von Modellen zu optimieren. Hierauf wird später etwas näher eingegangen.

12.1 TensorFlow Lite Micro

Neuronale Netzwerke auf Microcontrollern zu implementieren ist äußerst anspruchsvoll. Oft begegnet man Speicher/- und Leistungsproblemen. TensorFlow Lite Micro (TF Lite Micro) ist eine Unterbibliothek von TensorFlow Lite und wurde speziell für die Anwendung von TensorFlow Modellen auf Microcontroller entwickelt [24]. Dabei wurden weniger wichtige Funktionen entfernt und so die Gesamtgröße der Bibliothek massiv verringert. Es ist hierbei wichtig zu erwähnen, dass dadurch einige Funktionen, die sich in der normalen Version von Tensorflow befinden, nicht in der TF Lite Micro Version vorhanden sind. Deshalb muss Vorsicht geboten sein. Im späteren Verlauf der Arbeit wird darauf eingegangen, wie man verhindern kann, dass Operatoren verwendet werden, die nicht in TF Lite Micro enthalten sind.

TF Lite Mico wurde zudem schon erfolgreich auf dem ESP32 getestet. Die entsprechenden Bibliotheken kann man über die Arduino IDE herunterladen und stehen anschließend direkt zur Verfügung.

12.2 Der ESP32 Microcontroller

Der ESP32 ist ein leistungsfähiger und vielseitiger Mikrocontroller, der von der Firma Espressif Systems entwickelt wurde. Er verfügt über einen Dual-Core-Prozessor mit einer Taktrate von bis zu 240 MHz und ist mit integriertem WLAN und Bluetooth ausgestattet. Der ESP32 ist in der Lage, eine Vielzahl von Aufgaben auszuführen, von einfachen GPIO-Operationen (General Purpose Input and Output) bis hin zu komplexen Aufgaben wie maschinellem Lernen. In der Basisvariante besitzt er 4 MB Speicher und 500 kB SRAM. Der Microcontroller befindet sich oft auf Entwicklerboards, welche die Konnektivität und die Handhabung vereinfachen.

Abbildung 11: ESP32 NodeMCU auf einem Entwicklerboard

Im Rahmen dieser Arbeit wird sich für den ESP32 aus mehreren Gründen entschieden.

- Vergleichsweise hohe Leistung zu anderen Microcontrollern
- Hohe Speicherkapazität
- Vergleichsweise günstiger Preis
- Hohe Konnektivität durch Bluetooth und WLAN
- TF Lite Micro wurde bereits erfolgreich auf dem ESP32 getestet
- Nötigen Bibliotheken für TF Lite Micro auf dem ESP32 sind in der Arduino IDE vorhanden
- Fähig 32-Bit Gleitkommaoperationen auszuführen
- UART-Schnittstelle ermöglicht eine einfache Programmierung

Das verwendete Entwicklerboard ist der ESP32 NodeMCU. Es ist auch möglich andere Entwicklerboards zu verwenden, solange die Ausstattung gleichbleibt.

12.3 Der MPU-6050 Bewegungssensor

Um Bewegungsdaten auszuwerten, ist es zuerst notwendig, sie zu erfassen. Hierfür wird ein MPU-6050 Bewegungssensor verwendet. In ihm sind ein Accelerometer und ein Gyroskop verbaut. Das Accelerometer misst die translatorische Beschleunigung in allen drei Achsen und das Gyroskop misst die rotatorische Beschleunigung um alle drei Achsen. Es stehen somit 6 Achsen an Bewegungsdaten zur Verfügung. Dadurch können komplexe Dynamiken erfasst und durch das neurale Netzwerk analysiert werden.

Abbildung 12: MPU-6050 6 Achsen Bewegungssensor

Die Abtastrate des Sensors liegt bei 1 kHz für das Accelerometer und 8 kHz für das Gyroskop, wobei die Abtastrate für Gyroskop auf 1 kHz reduziert wird. Es ist somit nach dem Nyquist-Shannon-Abtasttheorem möglich, Frequenzen von bis zu 500 Hz oder 30.000 1/min zu erfassen. Der MPU-6050 kommuniziert mit dem ESP32 über eine I²C Schnittstelle. Da der Sensor bereits intern Analog/Digitalwandler besitzt können direkt die entsprechenden Werte ausgelesen werden. Es ist somit keine Wandlung auf dem ESP32 nötig. Die Vorteile werden kurz zusammengefasst:

- Beschleunigungsdaten in allen 6 Achsen messbar
- Hohe Abtastraten von bis zu 1kHz
- Interne Analog/Digitalwandlung
- Einfache Ansteuerung über I²C möglich
- Günstiger Preis

All diese Faktoren machen den MPU-6050 zu einer guten Wahl für das Projekt.

13. I²C Kommunikation

Um Bewegungsdaten aus dem MPU-6050 zu lesen, muss er über den ESP32 angesteuert werden. Das geschieht über den sog. I²C (Inter-Integrated Circuit) Datenbus. Er ermöglicht die Kommunikation zwischen verschiedenen Schaltkreisen. Hierfür werden nur zwei Leitungen, die Datenleitung (SDA) und die Taktleitung (SCL), benötigt. Ein I²C-System besteht normalerweise aus einem Master-Baustein (ESP32) und mehreren Slave-Bausteinen (MPU-6050). Der Master-Baustein initiiert die Kommunikation und kann Daten von den Slave-Bausteinen anfordern oder an sie senden. Aufgrund der einfachen Handhabung und geringem Verbindungsaufwand, ist I²C sogar in der Industrie ein anerkannter Standard.

Im Fall des ESP32 und dem MPU-6050 kann die Arduino Bibliothek „Wire.h“ verwendet werden. Sie besitzt die nötigen Funktionen um die Kommunikation zwischen ESP32 und MPU-6050 zu realisieren.

Da mit dem MPU-6050 sehr viele Daten in sehr kurzer Zeit übertragen werden sollen, muss man sich Gedanken über die Übertragungsgeschwindigkeit machen. I²C bietet hierfür unterschiedliche Übertragungsgeschwindigkeiten von 100 kBit/s (I²C Standard Mode) und 400 kBit/s (I²C Fast Mode). Die Wahl der richtigen Geschwindigkeit ist wichtig, um den Sensor nicht auszubremsen.

Alle Bewegungsdaten aus dem MPU-6050 sind 16 Bit Integer. Die I²C Kommunikation überträgt nur 8 Bit und einen Bestätigungsbit auf einmal. 16 Bit Integer müssen deshalb auf 8 Bit aufgeteilt und separat übertragen werden.

Somit wird für die Übertragung eines 16 Bit Integer und die nötigen Bestätigungsbits, 2*(8+1) = 18 Bit benötigt [25]. Da der MPU-6050 6 Achsen aufzeichnet, beläuft sich die Gesamtanzahl an Bits auf 6*(2*(8+1)) = 108 Bits.

Wenn nun die Übertragungsgeschwindigkeit (T [Bit/s]) durch die Anzahl der Bits geteilt wird, lässt sich die maximale Abtastrate ermitteln. In einer Formel ausgedrückt beläuft sich somit die maximale Abtastrate (A_{max}) auf:

$$A_{max} = \frac{T}{6 * (2 * (8 + 1))} [Hz]$$

Formel 6: Maximale Abtastrate

Wenn der I²C Standard Mode von T = 100 kBit/s eingesetzt wird, erhält man eine Abtastrate von etwa 0,925 kHz. Diese liegt unter den geforderten 1 kHz und würde somit den Sensor ausbremsen. Für den I²C Fast Mode mit T = 400 kBit/s geht eine Abtastrate von etwa 3,7 kHz hervor. Diese Liegt weit über den angepeilten 1 kHz und ist somit für den Betrieb bestens geeignet.

Der zusätzliche Zeitpuffer ist zudem äußerst nützlich, da im späteren Programmablauf nicht nur die Bewegungsdaten ausgelesen werden, sondern auch die nötigen Adressen gesetzt, Integer in Beschleunigungen umgerechnet und weitere mathematische Operationen durchgeführt werden müssen.

13.1 Anschlussschema

Der MPU-6050 wird über das I²C Kommunikationsprotokoll angesprochen. Hierfür stellt der ESP32 eigene Anschlüsse zur Verfügung. Die SCL (eng. Serial Clock) und SDA (eng. Serial Data) Anschlüsse befinden sich bei dem ESP32 NodeMCU auf den Pins 21 und 22. Neben der Datenverbindung muss der Bewegungssensor auch mit Strom versorgt werden. Hierfür ist Laut dem Datenblatt eine Spannung von 3.3 Volt (V) nötig. Der ESP32 hat hierfür ebenfalls gesonderte Anschlüsse, welche die Spannung und die Erdung abdecken.

Der MPU-6050 kann nun mit dem EPS32 verbunden werden. Eine Steckplatine eignet sich dabei besonders gut, da sich der Anschlussaufwand in Grenzen hält und auch später schneller Änderungen vorgenommen werden können. Die entsprechenden Pins werden über Kabel verbunden. Die fertige Platine lässt sich in Abb. 13 betrachten.

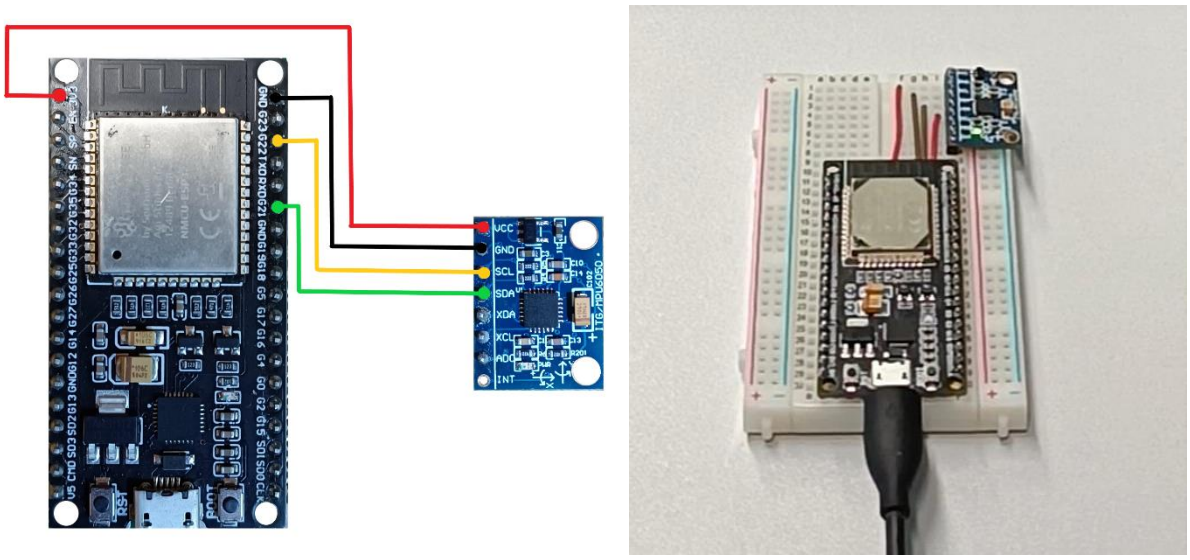

Abbildung 13: Anschlussschema ESP32 mit MPU-6050 (links) und Montage auf Steckplatine (rechts)

Der ESP32 ist nun mit dem MPU-6050 verbunden und kann über I²C mit ihm kommunizieren. Das nötige Programm wird mit der Arduino IDE (de. Integrierte Entwicklerumgebung) geschrieben und auf den ESP32 übertragen.

13.2 I²C Kommunikation mit Arduino Wire

Damit der Bewegungssensor mit dem ESP32 kommunizieren kann, ist eine entsprechende Software nötig. Die Arduino IDE bietet eine Reihe Bibliotheken an, welche die Kommunikation zwischen dem ESP32 und dem MPU-6050 vereinfachen. Zusätzlich ist die Installation solcher Bibliotheken sehr einfach, da sie in der Arduino IDE durch eine Installation sofort zur Verfügung stehen.

Mit der Wire Bibliothek ist es möglich, über I²C direkt in die Register des Sensors zu schreiben und sie zu lesen. Es wird somit nur minimaler Rechenaufwand betrieben und die Geschwindigkeit ist dementsprechend hoch.

Zuerst muss die richtige Konfiguration für den MPU-6050 vorliegen. Das beinhaltet Einstellungen wie:

- Interne Taktfrequenz | 8 MHz
- Wertebereiche für Accelerometer und Gyroskop | +- 2g | +- 250°/s
- Abtastrate für Accelerometer und Gyroskop | 1 kHz
- Deaktivieren von Filtern

Kleine Wertebereiche entsprechen der höchstmöglichen Auflösung und können so besser feine Auslenkungen registrieren. Die genaue Auflösung beläuft sich beim Accelerometer auf:

$$A_A = \frac{2 * 2g}{2^{16}} = 0{,}000061035g$$

Formel 7: Auflösung Accelerometer

Und beim Gyroskop auf:

$$A_G = \frac{2 * 250°}{2^{16}} = 0{,}0076294°$$

Formel 8: Auflösung Gyroskop

Diese Werte werden später im Programm für die Umrechnung von Binär in Dezimalzahlen benötigt.

13.3 Probleme mit Adafruit Bibliotheken

In Kapitel 13 wurde die maximale Übertragungsgeschwindigkeit von I²C Schnittstellen ermittelt. Dabei hat sich herausgestellt, dass 400 kBit/s genügen, um eine Abtastrate von theoretisch 3,7 kHz zu erreichen.

In frühen Iterationen des Programms wurden die Sensorwerte mithilfe von Adafruit Bibliotheken ausgelesen, genauer „Adafruit_Sensor.h“ und „Adafruit_MPU6050.h“. Tests haben jedoch gezeigt, dass diese Bibliotheken nicht (!) für hohe Abtastraten geeignet sind. Mit den Bibliotheken konnte im Test eine Abtastrate von nur 440 Hz erreicht werden, was weit unter den angestrebten 1 kHz liegt.

Aus diesem Grund wurde sich gegen die Verwendung von Adafruit Bibliotheken entschieden und nur die Wire Bibliothek verwendet. Nur so ist es möglich mit dem MPU-6050 eine Abtastrate von 1kHz zu erreichen.

Für Projekte mit Abtastraten unter 400 Hz sind die Bibliotheken trotzdem sehr gut geeignet, da sie den Programmieraufwand drastisch reduzieren.

14. Programmablauf

Wenn künstliche neuronale Netze auf Microcontrollern implementiert werden, unterscheidet sich die Prozedur etwas zu normalen Computern. Prinzipiell unterteilt man die Arbeit mit künstlichen neuronalen Netzen auf normalen Computern in 3 Phasen.

1. **Daten Akquirierung**
   Hier werden die nötigen Trainings-/, Test-/ und Validierungsdatensätze erstellt.
2. **Netzwerk erstellen, trainieren und testen**
   Das neuronale Netz wird entsprechend den Vorgaben und Architektur erstellt. Anschließend durchläuft es das Training und Testen. Wenn es die gewünschte Genauigkeit erzielt, ist das Netzwerk bereit für Phase 3.
3. **Implementierung**
   Das neuronale Netz wird als Modell exportiert und in das entsprechende System eingebunden. Hier erfüllt es die Aufgabe für das es trainiert wurde (Bilderkennung, Spracherkennung, etc.). Wenn ein neuronales Netz nicht mehr trainiert wird und nur noch Daten anhand des Trainings klassifiziert/rekonstruiert spricht man von *Inference* (de. Schlussfolgerung).

Wichtig hierbei ist, dass das Training oft auf anderen Computern/Servern ausgeführt wird als die Inference. Der Grund ist der hohe Rechenaufwand, der beim Training von neuronalen Netzwerken anfällt. Inference wiederum braucht, im Vergleich, sehr wenig Rechenleistung, da es sich um simple Arithmetik handelt.

Für simple Aufgaben, wie einfache Klassifizierung von Bildern, reicht oft die Rechenleistung des Computers (PC, Laptop, etc.) für das Training aus, auf dem auch später die Inference durchgeführt wird. Jedoch stößt das Prinzip schnell an Grenzen, wenn man versucht neuronale Netzwerke auf Microcontrollern zu Implementieren. Hier reicht die Leistung nicht aus, um das Modell zu trainieren.

Es muss somit auf einem anderen Computer trainiert und anschließend auf dem Microcontroller implementiert werden.

Besondere Beachtung muss man dem Exportieren der Modelle widmen. Hier bestimmt man Faktoren wie Größe, Geschwindigkeit und Genauigkeit des Netzwerks. Dieser Punkt wird später erneut aufgegriffen.

Nachdem nun alle Randbedingungen ermittelt wurden, kann man den Programmablauf erstellen. Dabei steht nicht nur das Training, sondern auch die komplette Implementierung im Vordergrund.

1. **Daten Akquirieren**

   Programmieren eines Arduino Sketches, der die Sensordaten aus dem MPU-6050 ausliest, buffert und über eine serielle Schnittstelle an den ESP32 sendet.

2. **Daten Aufzeichnen**

   Auf der Empfängerseite nimmt ein Programm die eintreffenden Daten an und speichert sie in einer CSV-Datei ab.

3. **Netzwerk erstellen, trainieren und testen**

   Das neuronale Netz wird nun mit den Daten aus der CSV-Datei trainiert und getestet.

4. **Exportieren**

   Das neuronale Netzwerk wird exportiert und in ein passendes Format für den Microcontroller umgewandelt.

5. **Inference**

   Ein Arduino Sketch übernimmt die Inference auf dem ESP32. In dieses Programm wird das neuronale Netz eingebunden. Die Abfolge ist wie folgt.

   - Sensordaten vom MPU-6050 auslesen
   - Sensordaten für Inference in das neuronale Netz geben
   - Mittlerer Absoluter Fehler für Eingabe-/ und Ausgabedaten ermitteln
   - Fehlerwert über serielle Schnittstelle ausgeben

6. **Rekonstruktionsfehler aufzeichnen**

   Empfangen des Fehlerwerts über die serielle Schnittstelle und Auswerten der Daten auf einem separaten Computer.

Die einzelnen Schritte werden nun intensiver betratet und hierfür programmatische Lösungen vorgestellt. Bei der Erklärung steht nicht nur das Programm selbst im Mittelpunkt, es wird auch versucht, nötiges Vorwissen zu vermitteln, damit die Probleme möglichst einfach gelöst werden können.

14.1 Datenakquirierung

Der erste Schritt befasst sich mit der Datenakquirierung. Hier werden die Daten gesammelt, mit denen man später das Netzwerk trainiert und testet. Dabei ist es besonders wichtig möglichst anwendungsnahe Daten zu verwenden, welche keine Anomalien enthalten. Das bedeutet, Daten, die für das Training und Testen verwendet werden, sollen später auch reale Betriebsbedingungen vermitteln. Im Beispiel des Microcontrollers darf der Bewegungssensor, während Test-/ und Trainingsdaten aufgezeichnet werden, seine Position nicht verändern. Das trifft auch dann zu, wenn bereits Daten aufgezeichnet wurden. Bei der Anomalieerkennung mit neuralen Netzwerken ist es äußerst wichtig, dass die Position des Sensors während der gesamten Lebensdauer unverändert bleibt. Kleinste Positionsveränderungen können große Auswirkungen auf die Qualität der Anomalieerkennung haben. Falls der Sensor seine Position dennoch verändern sollte, müssen unter Umständen neue Daten aufgezeichnet und das Netzwerk neu trainiert werden. Montagemethoden wie, Klebeverbindungen, Schraubverbindungen oder Kabelbinder (eher für temporäre Anwendungen) sollten für einen sicheren Halt reichen.

An dieser Stelle ist es auch wichtig sich für ein geeignetes Datenformat zu entscheiden. Speziell ist damit die Anordnung der Messwerte gemeint, wie sie auch später dem Netzwerk überführt werden soll. Da der MPU-6050 sechs mögliche Freiheitsgrade besitzt handelt es sich hier um eine multidimensionale Eingabe mit der entsprechend umgegangen werden muss.

Im Beispiel des NASA Bearing Datasets handelte es sich lediglich um einen Vektor, eine eindimensionale Eingabe. Neuronale Netzwerke erwarten als Eingabe einen Vektor, somit muss man die multidimensionale Eingabe aus dem MPU-6050 transformieren, und einen eindimensionalen Vektor generieren.

Dafür gibt es mehrere Möglichkeiten. Die Daten direkt auf dem ESP32 zu transformieren oder auf dem Computer, welcher die Daten empfängt. Die Transformierung auf dem ESP32 hat einige Vorteile. Es muss kein zusätzliches Programm geschrieben werden, um die Daten zu verarbeiten. Der Sketch, welcher die Daten aus dem MPU-6050 liest, kann sie direkt in das passende Format umwandeln. Das Programm kompakter und simpler. Daher wird sich für diese Variante entschieden.

Der Vektor an Daten, welcher auch später in das Netzwerk gegeben wird, besitzt folgenden Aufbau:

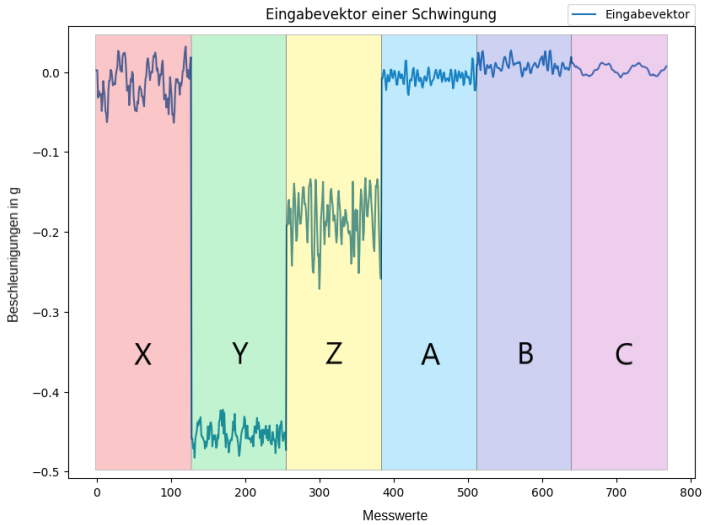

Abbildung 14: Eingabevektor in das neurale Netzwerk

In Abb. 14 ist der Aufbau deutlich zu erkennen. Jede Achse besitzt 128 Messwerte, die hintereinander angeordnet sind. Achsen X Y Z geben die translatorische Beschleunigung und A B C geben die rotatorische Beschleunigung wieder. Die Gesamtlänge beträgt somit 768 Messwerte.

Diese Größe wird gewählt aufgrund bisheriger Erfahrungen mit dem NASA Bearing Dataset und um die Gesamtgröße des Vektors zu reduzieren. Zu wenig Messwerte können verhindern, dass das Netzwerk wertvolle Dynamiken im Datensatz nicht erkennt. Gleichzeitig sorgt ein zu großer Vektor auf dem Microcontroller für Speicherprobleme, da der SRAM nur 500 KByte fasst.

Ein weiterer wichtiger Aspekt der behandelt werden muss, ist die Skalierung der Daten. Aus dem MPU-6050 werden Messwerte entnommen die zwischen +-2 bzw. +-250 liegen. Mit dieser Skalierung kann das Netzwerk nicht umgehen, da später der Tangens hyperbolicus als Aktivierungsfunktion verwendet wird. Diese Funktion besitzt einen Wertebereich zwischen -1 und 1, somit muss, für eine erfolgreiche Rekonstruktion, der Datensatz ebenfalls zwischen -1 und 1 skaliert sein. Das wird erreicht, indem man die Daten vom Accelerometer mit 0.5 und die Daten aus dem Gyroskop mit 0.004 multipliziert. In Abb. 14 ist diese Skalierung zu erkennen, da kein Messwert über +- 1 hinausgeht.

Alle wichtigen Details für die Datenakquirierung sind nun aufgelistet und können in der Arduino IDE programmatisch umgesetzt werden. Der **Data_Collection_Sketch.ino** ist das fertige Programm für die Akquirierung von Trainingsdaten.

14.2 Daten Aufzeichnen

Der Microcontroller sendet die Bewegungsdaten an den Computer, an dem er angeschlossen ist. Damit das Netzwerk mit den Daten trainieren kann, müssen sie gespeichert werden. Für diesen Zweck wird eine Python Bibliothek verwendet. PySerial [26] ist in der Lage serielle Schnittstellen am Computer zu lesen und zu verarbeiten. Hierfür wird nun ein Skript in Python erstellt, welches genau diese Funktion erfüllt.

Der Name der CSV-Datei kann im Skript frei gewählt werden. Nach einer Aufzeichnung muss man den Anfang und Ende der CSV-Datei manuell entfernen. Am Anfang wird der Header eingefügt, der neben Zahlen auch Buchstaben enthält. Die letzte Zeile ist enthält nicht alle 768 Werte des Eingabevektors.

Die nötigen Daten stehen zur Verfügung. Das eigentliche Netzwerk kann nun erstellt und trainiert werden.

14.3 Das Netzwerk erstellen

Dieses Kapitel beschäftigt sich mit dem Thema, ein neuronales Netzwerk bezogen auf die vorhandene Anwendung zu erstellen. Die Netzwerkarchitektur ähnelt hierbei dem Netzwerk für das NASA Bearing Dataset, da dieses bereits gute Ergebnisse erzielt hat.

Der Eingabevektor besteht aus 768 Elementen und muss dem Netzwerk über eine Eingabeschicht vermittelt werden. In den darauffolgenden Schichten beginnt die Dimensionsreduzierung. Hier reduzieren die Neuronen die Datenmenge, um nur relevante Informationen zu erhalten. Dies wird erreicht, indem man Schrittweise die Anzahl der Neuronen verringert. Es hat sich bewährt, systematisch die Anzahl der Neuronen in der Folgeschicht zu halbieren (siehe Kapitel 8).

An dieser Stelle wird der sog. Kompressionsfaktor K eingeführt. Er ist ein Verhältnis zwischen der Schicht mit den meisten Neuronen, zu der Schicht mit den wenigsten Neuronen. Letzteres wird auch als Vektor latenter Variablen bezeichnet, kurz latenter Vektor. In ihm sind die Daten kodiert, welche für eine erfolgreiche Rekonstruktion am wichtigsten sind [27]. Der Kompressionsfaktor betrug bei dem NASA Bearing Dataset, $K = 64$. Der latente Vektor besaß 2 Neuronen, somit auch der Vektor nur 2 Variablen. Bei einer größeren Eingabe muss auch ein größerer latenter Vektor gewählt werden, da sonst wertvolle Merkmale verloren gehen könnten. Im Beispiel des neuen Netzwerks für den ESP32 beträgt seine Größe 6 Variablen. Zieht man den Kompressionsfaktor heran, beträgt dieser nun $K = 128$ und ist somit doppelt so groß. Es hat sich herausgestellt, dass sich diese Größe gut eignet, um auch feine Anomalien zu erkennen.

Der grobe Aufbau des Netzwerks steht somit fest. Ein Encoder reduziert schrittweise die Eingabe, bis eine Schicht mit 6 Neuronen erreicht ist. Anschließend rekonstruiert der Decoder aus dem latenten Vektor, in gleicher Abfolge wie der Encoder nur umgekehrt, die Eingabedaten.

Die erste und letzte Schicht des Netzwerks können aufgrund von Speicherproblemen nicht halb so viele Neuronen wie die vorherige bzw. folgende Schicht haben. Daher wurde sich hier um einen Faktor von 8 statt 2 entschieden.

Einen umfassenden Blick auf das fertige Netzwerk kann uns das Programm *Netron* liefern [28]. Es ist speziell dafür entwickelt künstliche neuronale Netzwerke anschaulich darzustellen.

Insgesamt besitzt das Netzwerk eine Größe von 160830 trainierbaren Parametern und kann in Abb. 15 betrachtet werden.

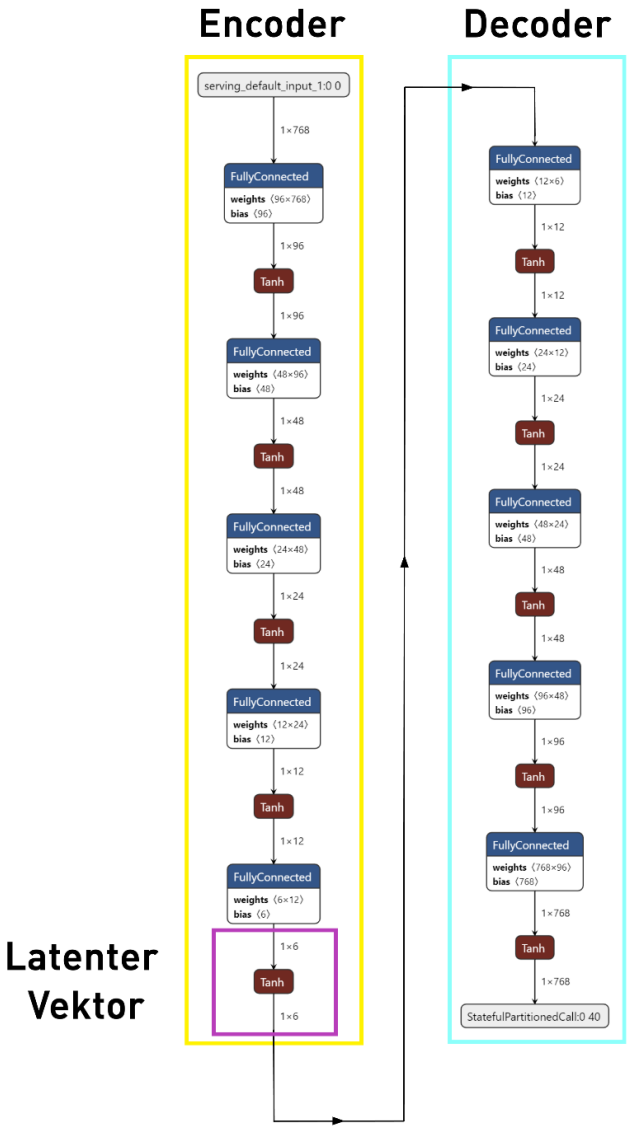

Abbildung 15: Aufbau des neuralen Netzwerks für den ESP32

14.4 Training des Netzwerks

Da dieses Netzwerk auf einem Microcontroller implementiert werden soll, muss an dieser Stelle besondere Vorsicht geboten sein. Tensorflow Lite Micro besitzt nicht die gleiche Anzahl an Operatoren wie die Standard Tensorflow Version. Das bedeutet bei exotischen Netzwerken kann es vorkommen, dass sie, trotz Einhalten der Speichergrenzen, nicht auf einem Microcontroller funktionieren werden. Falls Unsicherheiten bestehen, bietet das Tensorflow Team eine umfangreiche Liste an, auf der jeder unterstütze Operator verzeichnet ist [29].

Ein weiteres Problem, dass sich im Laufe der Forschung bemerkbar gemacht hat, ist das sog. Batch Training . Normalerweise sorgt Batch Training für eine deutliche Verkürzung der Trainingsdauer, da mehrere Eingabevektoren gleichzeitig in das Netzwerk gegeben werden [30]. Ein unschöner Nebeneffekt ist jedoch, dass Batch Training das Netzwerk komplett unbrauchbar für Anwendungen auf Microcontrollern macht. Durch Batch Training verwendet das Netzwerk Operatoren, die nicht in der Liste an unterstützen Operatoren verzeichnet ist.

Um diesen Effekt zu verhindern, ist es ratsam, in der Eingabeschicht sowie auch in der Trainingsfunktion eine Batch Größe von 1 zu wählen. So trainiert das Netzwerk nicht im Batch Modus und kann auf Microcontrollern implementiert werden.

Das neurale Netzwerk benötigt noch Daten, mit denen es trainieren kann. Im Rahmen dieser Untersuchung werden Schwingungsdaten eines Raumlüfters verwendet, welcher in Abb. 16 zu sehen ist. Die Drehzahl des Lüfters beträgt bei höchster Stufe etwa 1340 1/min. Diese geringe Drehzahl ist optimal für erste Tests, da der Sensor die Schwingung gut abbilden kann. Durch Beschweren eines Lüfterblattes mit einer Masse, können zudem sehr leicht Anomalien simuliert werden.

Abbildung 16: Raumlüfter zur Schwingungsuntersuchung

Der ESP32 mit dem MPU6050 müssen auf dem Lüfter befestigt werden. Für einen kompakteren und sicheren Aufbau ist der ESP32 und MPU-6050 auf einer Lochrasterplatine verlötet. Der MPU-6050 befindet sich auf der Hinterseite. Das fertige Paket kann an der Rückseite des Lüfters angebracht werden, wie auch in Abb. 17 zu sehen ist. Hierfür eignen sich Kabelbinder, welche den Aufbau gegen Verrutschen absichern.

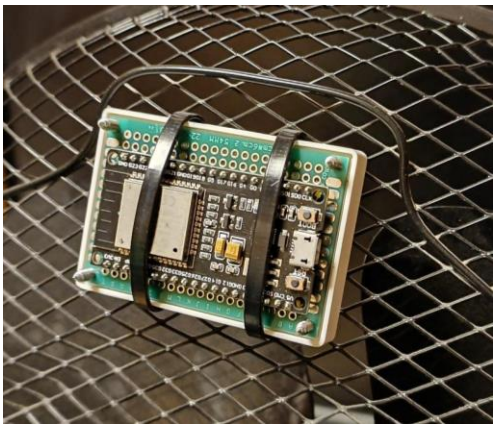

Abbildung 17: Anbringen des ESP32 und MPU6050 an dem Raumlüfter

Nach einer Einlaufphase von 20 Minuten können die Schwingungsdaten aufgezeichnet werden. Mit den beiden bisherigen Programmen empfängt ein externer Computer die Schwingungsdaten und speichert sie in einer CSV-Datei ab. Die gesamte Aufnahmezeit entspricht einer Stunde.

Der fertige Datensatz kann nun inspiziert werden.

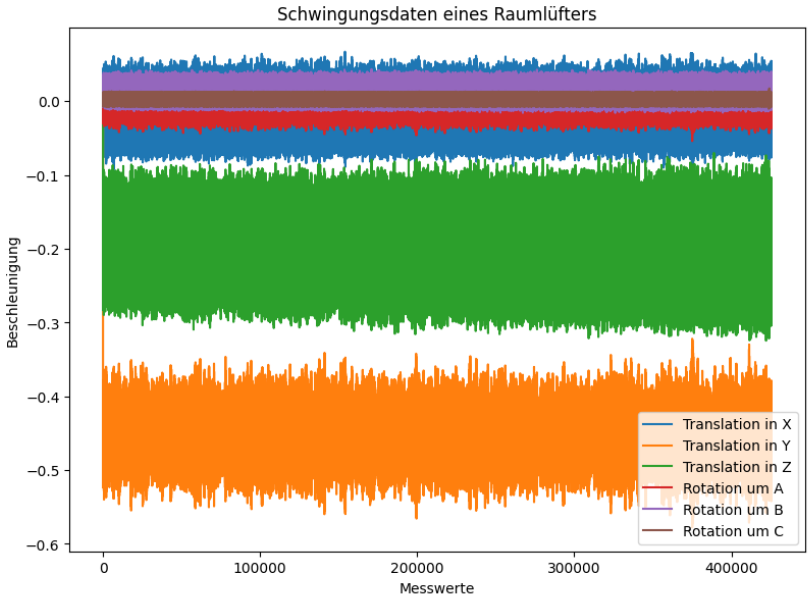

Abbildung 18: Schwingungsdaten eines Raumlüfters

In Abb. 18 ist deutlich erkennbar, wie sich die einzelnen Daten in ihren Beschleunigungswerten unterscheiden. Außerdem gibt es keine offensichtlichen Anomalien, welche die Daten unbrauchbar machen. In dieser Form eignen sich die Schwingungsdaten, um das neurale Netzwerk zu trainieren.

Das eigentliche Training kann nun beginnen. Der Datensatz besteht aus einer (3324,768) Matrix. Diese Matrix wird in 80% Trainingssatz und 20% Testsatz unterteilt. Für ein gutes Ergebnis wird ein Training mit 2000 Epochen bei einer Lernrate von 0.0001 gewählt. Die Trainingsdauer beträgt 1h 40 Minuten auf einer GTX 1080.

Nach dem Training kann der Trainingsverlauf begutachtet werden.

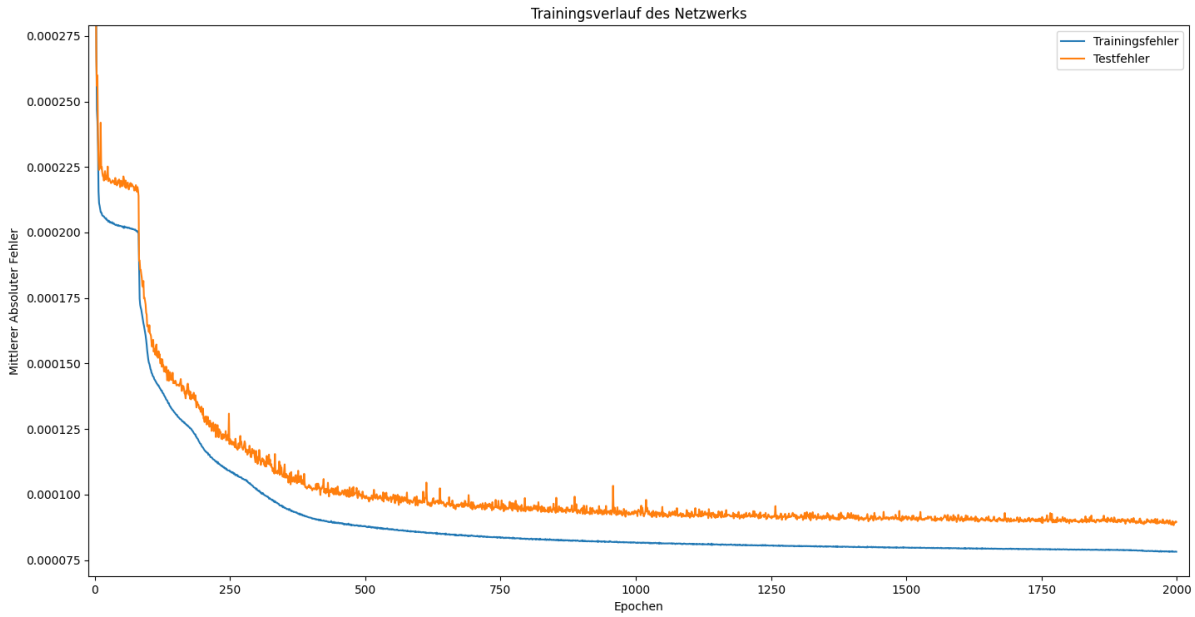

Abbildung 19: Trainingsverlauf des Netzwerks

Der Trainingsverlauf in Abb. 19 sieht sehr gut aus. Das Netzwerk konnte kontinuierlich den Rekonstruktionsfehler minimieren. Gut ist hierbei, dass nicht nur der Trainingsfehler, sondern auch der Testfehler gesunken ist. Somit kann ein Overfitting ausgeschlossen werden. Erstaunlich ist der Sprung in Epoche 80. Dort konnte das Netzwerk den Fehler sehr schnell reduzieren. Es lohnt sich somit das Training für viele Epochen durchzuführen. Ab Epoche 1000 hat sich das Training nicht sehr verbessert. Eine längere Trainingsdauer als 2000 Epochen wird daher nicht empfohlen.

Für einen besseren Überblick kann man nun die Rekonstruktion des Netzwerks betrachten. Hierfür wurden Eingabevektoren aus dem Testdatensatz in das Netzwerk gegeben. Mit den Eingabevektoren und die Ausgabevektoren kann man zusätzlich die Rekonstruktionsqualität des Netzwerks beurteilen.

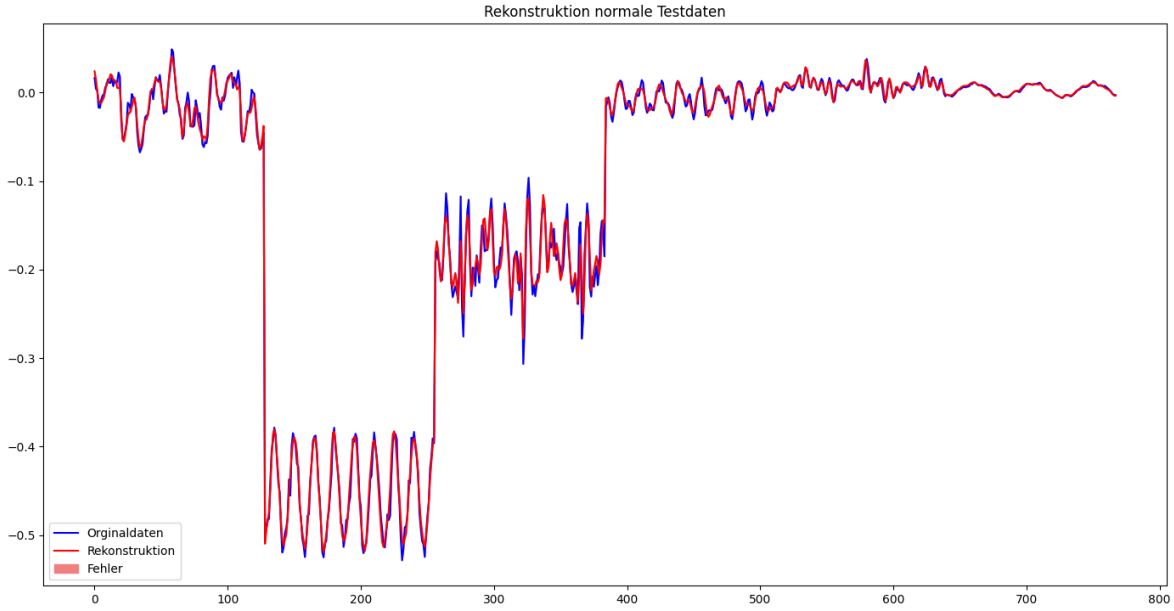

Abbildung 20: Rekonstruktion normaler Testdaten

Abb. 20 zeigt die Gegenüberstellung zwischen den Eingabevektoren und den Ausgabevektoren. Die Orginaldaten (Blau) ist der Eingabevektor und die Rekonstruktion (Rot) ist der Ausgabevektor. Es lässt sich hier sehr schön erkennen wie das Netzwerk versucht, die Daten zu imitieren. Die Schwingungen sind sehr schön ausgeprägt und es gibt nur kleinste Abweichungen. So muss eine gute Rekonstruktion aussehen.

Das Netzwerk ist mit seiner Performance sehr gut geeignet, um auf dem ESP32 implementiert zu werden. In den folgenden Kapiteln geht es um das Exportieren und Implementieren.

14.5 Exportieren des Netzwerks

Exportieren ist ein wichtiger Schritt, um neurale Netzwerke auf anderen Computern zu betreiben. In diesem Beispiel ist es besonders wichtig, da das Netzwerk später mit sehr wenig Leistung und Speicherplatz betrieben werden soll. Mit anderen Worten, Exportieren ist einer der wichtigsten Punkte, die man beachten muss.

Exportieren besteht dabei aus zwei Schritten.

1. Modell Quantifizieren
2. Modell für den Microcontroller umwandeln

Quantifizieren eines Modells legt seine Größe und Geschwindigkeit im späteren Betrieb fest. Alle Parameter im Netzwerks liegen als 32 Bit Floats (Gleitkommazahlen) vor. 32 Bit brauchen jedoch sehr viel Platz und lange in der Berechnung. Im Gegensatz zu Floats gibt es 8 Bit Integer (Ganze Zahlen). Sie brauchen wesentlich weniger Speicherplatz und sind sehr viel schneller in der Berechnung als 32 Bit Floats. Zudem ist fast jeder Microcontroller in der Lage Arithmetik mit 8 Bit Integer auszuführen. Somit wäre es sehr viel Besser, wenn die 32 Bit Floats im Netzwerk in 8 Bit Integer umgewandelt werden würden.

Genau das macht die Quantifizierung. Sie wandelt die Parameter von 32 Bit Floats in 8 Bit Integer um. Das Netzwerk wird somit 4 mal kleiner und mehr als 3 mal so schnell [31].

Natürlich verliert das Netzwerk dadurch an Genauigkeit, jedoch nur im Rahmen von 2% im Beispiel der Bilderkennung [32]. Dieser Nachteil wird jedoch schnell wieder aufgehoben. Vergleicht man quantifizierte und nicht quantifizierte Netzwerke gleicher Größe, performen letztere besser [32]. Grund hierfür ist die deutlich größere Menge an Neuronen, die ein Quantifiziertes Netzwerk besitzen kann. Daher eignet sich Quantifizierung auch für Netzwerke die eigentlich nicht in Leistung und Speicher limitiert sind.

An dieser Stelle wird zum ersten Mal Tensorflow Lite verwendet. Tensorflow Lite bietet eine Vielzahl verschiedener Quantifizierungen an, die auch in Abb. 21 zu sehen sind. Zudem wird auch ein Entscheidungsbaum angeboten, welcher die Entscheidung für die richtige Quantifizierung erleichtern soll.

Von Quantifizierung, die nur die Gewichte in den Schichten anpasst (Dynamic Range Quantization) bis hin zu voller 8 Bit Integer Quantifizierung, gibt es dabei sehr viele Möglichkeiten. Es wird von Tensorflow empfohlen 8 Bit Integer Quantifizierung für Microcontroller zu verwenden, daher wird sich für diese Methode entschieden.

Technique	Benefits	Hardware
Dynamic range quantization	4x smaller, 2x-3x speedup	CPU
Full integer quantization	4x smaller, 3x+ speedup	CPU, Edge TPU, Microcontrollers
Float16 quantization	2x smaller, GPU acceleration	CPU, GPU

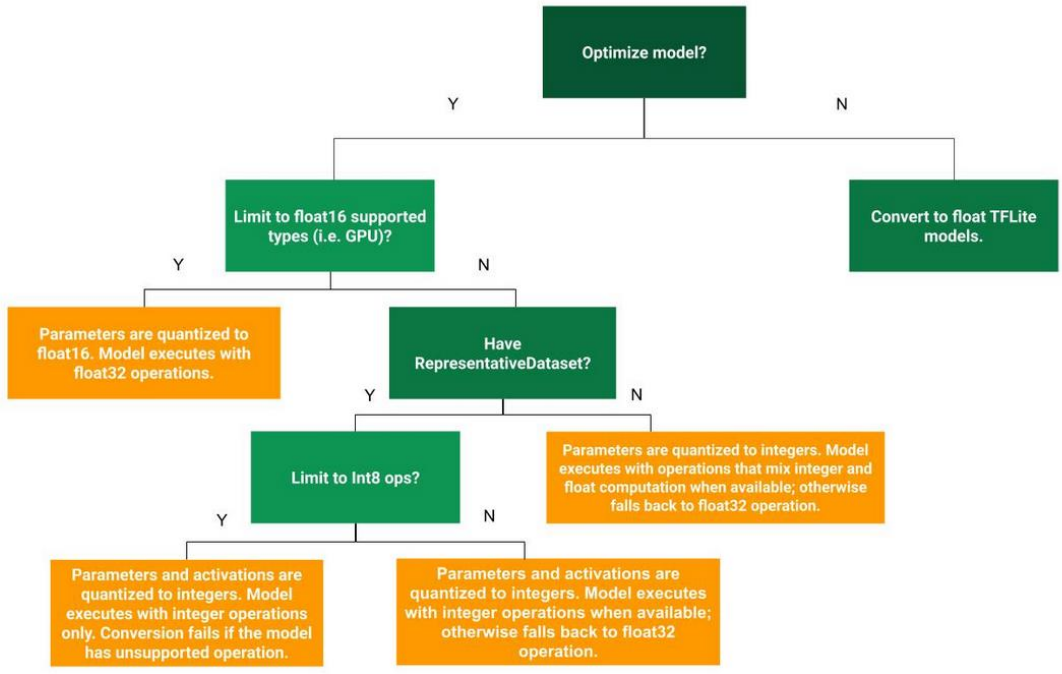

Abbildung 21: Diagramm für Quantifizierung

Diese Quantifizierung hat jedoch einen kleinen Nachteil. Das Modell darf nur 8 Bit Integer als Eingabevektor bekommen. Somit muss man die Sensorwerte zuerst auf 8 Bit Integer umrechnen. Sobald das Netzwerk die Inference abgeschlossen hat, muss der Ausgabevektor wieder auf 32 Bit Floats zurückgerechnet werden. Erst dann ist es möglich, einen Rekonstruktionsfehler zu ermitteln.

Tensorflow Lite bietet auch hier eine Formel für die Umrechnung an.

Integer in Floats:

$$Echter\_Wert = (int8\_Wert - zero\_point) * scale$$

Formel 9: Umrechnung von Integer in Floats

Floats in Integer:

$$int8\_Wert = \left(\frac{Echter\_Wert}{scale}\right) + zero\_point$$

Formel 10: Umrechnung von Floats in Integer

Zero_point und *scale* sind Variablen, die aus der Quantifizierung hervorgehen. Sie können später aus dem Modell entnommen werden.

Zudem muss für die Quantifizierung ein repräsentativer Datensatz vorhanden sein. Mit diesem Datensatz kalibriert das Modell seine Parameter für die Quantifizierung [33]. Die Größe des Datensatzes liegt meist bei mehreren hundert Eingabevektoren. Im Rahmen dieser Untersuchung wird sich für eine Größe von 500 Eingabevektoren entschieden.

Im Beispiel des Modells kann sich die Reduktion der Größe sehen lassen. Ohne Quantifizierung hat das Model mit 160830 Parametern eine Größe von 636 KByte. Nach der Quantifizierung beträgt die Größe nur noch 166 KByte, was einer fast 4-fachen Reduktion entspricht.

14.6 Umwandeln des Modells

Das quantifizierte Modell wird als eine *.tflite* Datei abgespeichert. Microcontroller sind jedoch nicht in der Lage diese Dateien zu lesen. Hierfür muss das Modell in eine binäre Repräsentation umgewandelt werden.

Dies wird mit dem Programm *XXD* erstellt, welches sog. *Hex-Dumps*, Binäre Versionen von Daten erstellen kann. Nativ ist dieses Programm auf Linux hinterlegt, auf Windows muss es jedoch separat installiert werden.

Die .tflite Datei, in dem unser Modell abgespeichert ist, muss sich im selben Ordner befinden wie das XXD-Programm. Anschließend kann man über ein PowerShell Fenster das Programm starten.

Für die Umwandlung reicht folgender Befehl aus:

```
.\xxd -i model.tflite > model.h
```

Im obigen Beispiel heißt das Modell **model** und muss bei einer anderen Namensgebung entsprechend angepasst werden. Das Zielformat ist eine .h Datei, oder auch eine sog. Header-Datei. Dieses Format eignet sich sehr gut, um das Modell später in einem C++ Sketch zu importieren.

Leider kann das umgewandelte Modell so noch nicht in einen Sketch eingebunden werden. Hierfür sind noch weitere Anpassungen nötig.

- Kodierung von 16 Bit auf 8 Bit ändern
- Einfügen der *Include-Guards*
- Deklarieren des Modells als Konstante

Die einzelnen Anpassungen werden nun im Einzelnen kurz beleuchtet.

14.6.1 Kodierung ändern

XXD erzeugt eine binäre Repräsentation des Modells. Für die einzelnen Zeichen wird das UTF (UCS Transformation Format) verwendet. Es ist das am weitesten verbreitete Kodierungsformat [34] und beschreibt, wie Buchstaben und Zahlen binär kodiert sind. XXD erzeugt eine 16 Bit Kodierung des Modells, wobei das nötige Format 8 Bit sein sollte. Die Kodierung muss somit von 16 Bit auf 8 Bit verändert werden. Dies wird über die VS-Code Umgebung erreicht, indem man das gesamte Modell in einer neuen Kodierung öffnet (Siehe Abb. 22). Dabei sollten alle Symbole, die während der Änderung auftauchen, gelöscht werden. Nur so kann später der Sketch kompilieren.

Abbildung 22: Option zum Ändern der Kodierung in VS-Code

14.6.2 Einfügen der Include Guards

Include Guards sind ein wesentlicher Bestandteil der C++ Programmiersprache. Sie verhindern mehrfache Deklarierung in Programmen und tragen so maßgeblich zur Fehlervorbeugung bei. Sie werden generell am Anfang von Header-Dateien eingefügt und bilden eine If-Schleife, in welcher das gesamte Modell eingebunden ist.

14.6.3 Deklarieren des Modells als Konstante

Konstanten haben in C++ einen besonderen Stellenwert. Sie sind unveränderbare Variablen, die man im Programmablauf nur lesen und nicht beschreiben kann. Zudem benötigen diese Variablen besonders wenig Speicher. Daher eignen sie sich besonders gut, um das Modell zu deklarieren. Wird das Modell nicht als Konstante deklariert, kann dies zu einem Kompilierungsfehler führen.

All diese Änderungen sind nötig, um für einen sicheren und reibungsfreien Betrieb des Modells auf einem Microcontroller zu gewährleisten. Das Modell ist nun in einem passenden Format und alle wichtigen Anpassungen wurden durchgeführt. Nun ist das Netzwerk bereit für die Einbindung auf dem Microcontroller.

14.7 Inference

Das Modell ist nun bereit für die Inference. Hier wird das Modell auf dem Microcontroller implementiert und es erfüllt seinen Zweck für das es trainiert wurde. Dazu wird nun ein Arduino Sketch erstellt, in den das Modell eingebunden ist. Die genaue Abfolge des **ESP32_Inference_Sketch.ino** wird nun genauer erklärt. Inspiration des Sketches basiert auf einem Beispiel des Tensorflow Lite Micro Teams [35].

Ablauf des **ESP32_Inference_Sketch.ino**.

1. **Bibliotheken importieren und nötige Einstellungen tätigen**
   Genau wie bei der Datenakquirierung müssen auch hier entsprechende Bibliotheken importiert und Einstellungen getätigt werden. Einstellungen umfassen den MPU-6050, benötigter Speicher für das Modell, Namespaces, deklarieren der Operatoren, deklarieren des Modells, erstellen der Eingabe-/ und Ausgabepointer.

2. **Datenakquirierung**
   Für die Inference muss das Programm einen Eingabevektor generieren. Dafür wird der MPU-6050, wie bei der Datenakquirierung, abgefragt. Sobald genug Werte vorhanden sind, springt das Programm weiter.

3. **Umwandeln in 8 Bit Integer**
   Wie bereits erwähnt müssen die Eingabevektoren ein geeignetes Datenformat besitzen. Die Umwandlung findet gemäß der Formel in Kapitel 14.5 statt. Nach der Umwandlung werden diese Werte direkt in den Eingabepointer des Modells gelegt.

4. **Inference durchführen**

   Die Daten aus dem Eingabepointer laufen durch das Netzwerk. Der Ausgabepointer enthält das Ergebnis des Netzwerks.

5. **Umrechnen in Floats und Fehler ermitteln**

   Die Ausgabewerte aus dem Netzwerk müssen wieder in Floats umgewandelt werden. Anschließend kann man sie Summieren und von den Eingabewerten abziehen. Der Gesamtfehler zwischen Eingabewerten und Ausgabewerten ergibt den Rekonstruktionsfehler. Dieser wird über die serielle Schnittstelle ausgegeben und die Schleife startet erneut.

Der Rekonstruktionsfehler wird, genau wie bei der Datenakquirierung, auf einem separaten Computer aufgezeichnet und ausgewertet. Bei einer unbekannten Schwingung muss der Rekonstruktionsfehler größer sein als bei einer normalen Schwingung.

14.8 Rekonstruktionsfehler aufzeichnen

Der ESP32 sendet kontinuierlich den Rekonstruktionsfehler über die serielle Schnittstelle nach außen. Die Daten müssen nun auf einem externen Computer aufgezeichnet werden. Hierfür eignet sich das Python Programm aus Kapitel 14.2 sehr gut, da es die gleiche Aufgabe erfüllen muss. Die Rekonstruktionsdaten werden dabei wieder in einer CSV-Datei abgespeichert. Anschließend ist eine Visualisierung mit Python möglich.

15. Testen des Netzwerks unter realen Bedingungen

Das Programm ist nun bereit für einen Test mit realen Schwingungen. Dazu wird wieder der Raumlüfter verwendet, auf dem der ESP32 unverändert montiert ist. Nachdem das Programm auf den ESP32 hochgeladen und an einen externen Computer angeschlossen wurde, kann der Test beginnen. Der Computer zeichnet für eine Stunde den Rekonstruktionsfehler des Netzwerks auf. Dabei wird der Test in zwei Phasen unterteilt.

In der ersten Phase wird der Raumlüfter nicht eingeschaltet und erzeugt auch keinerlei Schwingungen. Ziel dieser Phase ist es zu sehen, ob das Netzwerk auch ein Stehenbleiben des Lüfters erkennen kann. Wenn dieser Test gut ausfällt, deutet das auf eine sehr hohe Netzwerkgenauigkeit hin. Das Netzwerk hat somit die Charakteristik der Schwingung erlernt und gibt nicht nur Pseudowerte aus.

In der zweiten Phase wird der Lüfter aktiviert und läuft auf der gleichen Drehzahl wie bei der Datenakquirierung. Hier soll das Netzwerk im laufenden Betrieb getestet werden. Es ist wichtig, dass sich die beiden Phasen voneinander trennen lassen.

Die gesamte Testzeit beträgt eine Stunde. Nach 10 Minuten wird der Lüfter aktiviert und dreht auf seiner Standarddrehzahl.

Die Ergebnisse werden im folgenden Diagramm visualisiert.

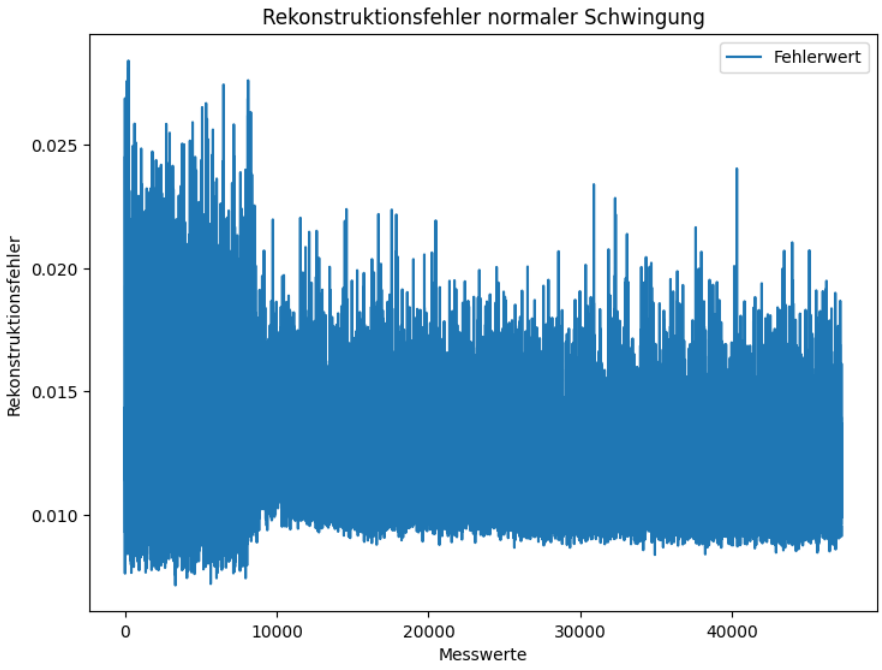

Abbildung 23: Rekonstruktionsfehler normaler Schwingung

Die zwei Phasen sind in dem Diagramm (Abb. 23) sehr schön sichtbar. Phase 1 erstreckt sich von 0 bis ca. 8000 und Phase 2 von 8000 bis 47000 Messwerte.

In der ersten Phase schlägt der Fehler weit aus und die Varianz ist sehr hoch. Das Netzwerk hat somit die Charakteristik der Schwingung erlernt und kann auch einen Stillstand der Lüfterblätter erkennen.

Anschließend wird der Lüfter eingeschaltet. Die Varianz ist kleiner als in Phase 1. Interessant hierbei ist auch, dass der Durchschnitt über die Zeit weiter absinkt. Ein möglicher Grund, könnte das Einlaufen des Lüfters sein. Die Trainingsdaten wurden zu einem Zeitpunkt aufgenommen, ab dem der Lüfter bereits eine Weile lief. Dies ist eine äußerst interessante Erkenntnis und auch ein Beweis für die Empfindlichkeit solcher Netzwerke.

Für die beiden vorliegen Phasen werden die wichtigsten statistischen Daten ermittelt und in einer Tabelle festgehalten.

Phase 1	**Phase 2**
Durchschnitt: 0.0124695	Durchschnitt: 0.0116875
Varianz: 0.0029439	Varianz: 0.0017856
Min: 0.00712	Min: 0.00836
Max: 0.02842	Max: 0.02404

Tabelle 1: Statistische Werte zu Phase 1 (Lüfterstillstand) und Phase 2 (Normalbetrieb)

Nun wird das Netzwerk auf dynamische Anomalien getestet. Hierfür wird eine kleine Masse in Form eines Klebestreifens an einem Lüfterblatt befestigt. Das Gewicht der Masse ist sehr gering und beträgt lediglich 0,1g.

Abbildung 24: Anbringen einer exzentrischen Masse an dem Raumlüfter

In Abb. 24 lässt sich der Klebestreifen erkennen. Durch eine exzentrische Masse wird das Lüfterblatt in radiale Richtung stärker beschleunigt. Der Lüfter läuft dadurch unrund. Das hier präsentierte System soll diese Schwingungen erkennen.

Als normale Daten werden die Daten aus dem ersten Test verwendet, wobei der Stillstand und die Eilaufphase nicht verwendet werden.

Die anomalen Daten mit Klebestreifen zeichnet der Computer nach einer Einlaufphase von 20 Minuten auf. Anschließend werden die normalen und anormalen Daten in einem Diagramm dargestellt.

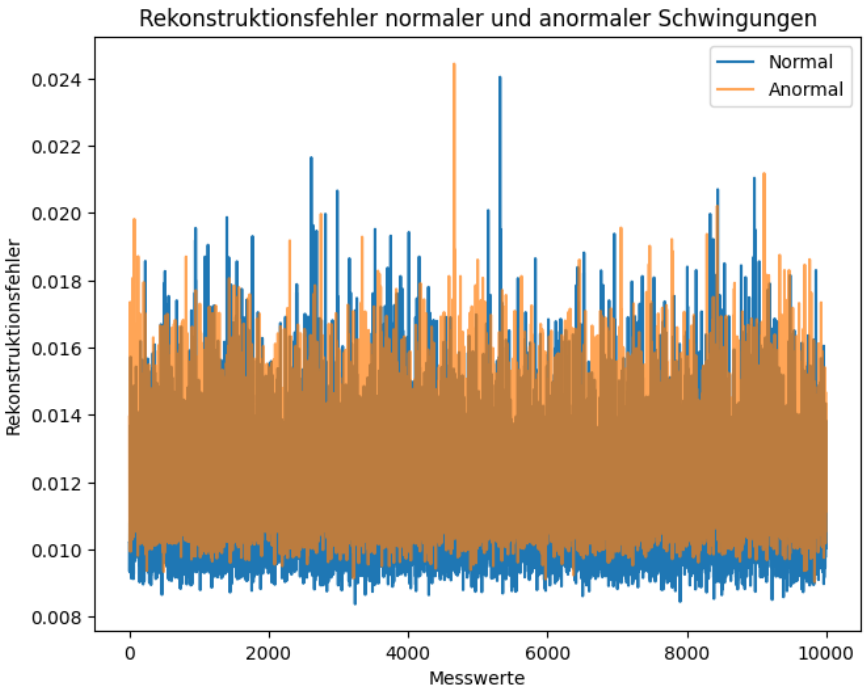

Abbildung 25: Rekonstruktionsfehler normaler und Anormaler Schwingungen

In Abb. 25 kann man einen deutlichen Unterschied zwischen normaler und Anormaler Schwingung erkennen. Beide Messreihen werden daraufhin auf ihre statistischen Charakteristiken untersucht. Dabei spielen Durchschnitt, Varianz, Min und Max Werte die wichtigsten Rollen.

Normale Daten	**Anormale Daten**
Durchschnitt: 0.0114023	Durchschnitt: 0.012321
Varianz: 0.0017707	Varianz: 0.0016341
Min: 0.00838	Min: 0.00904
Max: 0.02404	Max: 0.02443

Tabelle 2: Statistische Unterschiede normaler und anormale Schwingung Raumlüfter

Aus Tabelle 2 lässt sich ableiten, dass die anormalen Daten einen höheren durchschnittlichen Fehlerwert bei gleichzeitig niedriger Varianz haben. Das ist äußerst interessant da dies bedeutet, dass nicht nur der durchschnittliche Fehlerwert relevant ist, sondern auch die Varianz betrachtet werden muss. In prozentualen Verhältnissen ausgedrückt befinden sich der durchschnitt anormaler Daten 8,057% über und die Varianz anormaler Daten 8,359% unter dem Niveau der normalen Daten.

Zusammenfassend lässt sich sagen, dass das neurale Netzwerk in der Lage ist verschiedene Anomalien zu erkennen, darunter stehende Lüfterblätter als auch eine Unwucht am Lüfterblatt.

16. Schwingungsuntersuchung an einem Unwuchtmotor

Im Rahmen weiterer Untersuchungen soll das Netzwerk auf einem Unwuchtmotor getestet werden. Der Aufbau besteht aus einer Welle, welche hängend an einem Gestell gelagert ist. Fest auf der Welle befinden sich Scheiben, auf die man nach Bedarf exzentrisch Gewichte anbringen kann. Die Welle wird von einem Elektromotor über einen Riemen angetrieben. Der gesamte Versuchsaufbau ist in Abb. 26 zu sehen.

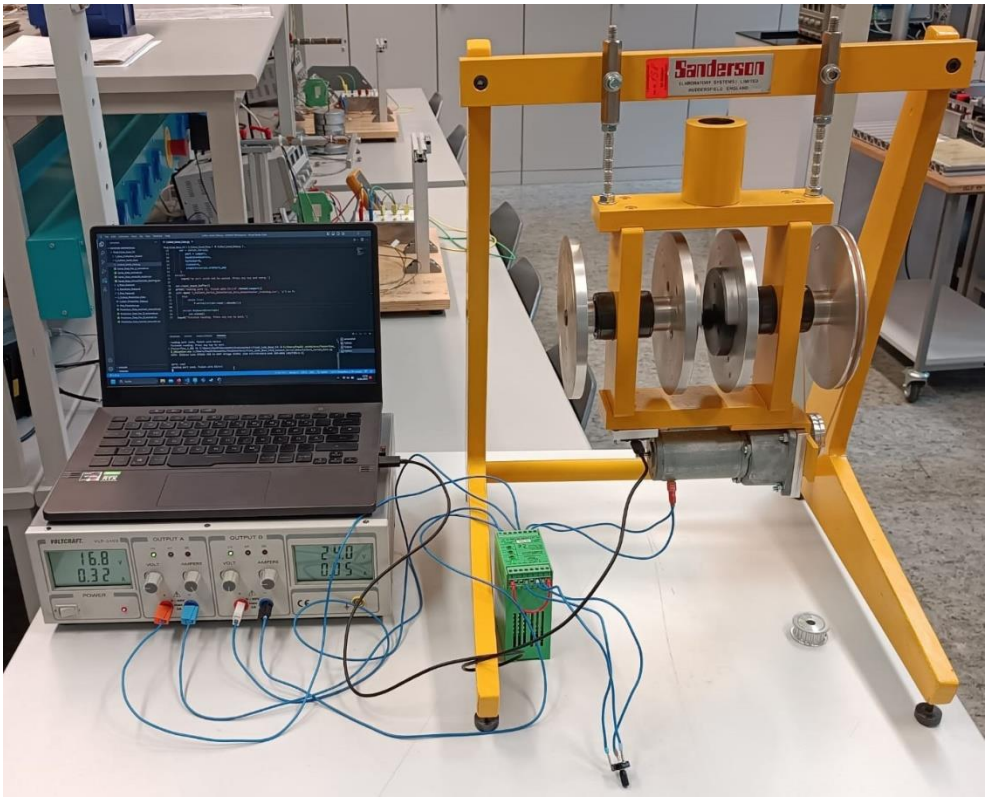

Abbildung 26: Versuchsaufbau Unwuchtmotor

Mit dem Unwuchtmotor ist es möglich, gezielte Schwingungen zu erzeugen. Die Gewichte und Drehzahl sind nach Bedarf anpassbar und bieten sehr große Variationsmöglichkeiten. Außerdem ist dieser Aufbau besonders realitätsnah, da es sich hier um einen technischen Aufbau mit Bauteilen wie Lagern, Gestell, Elektromotor und Steuerung handelt.

Die Konstruktion ist zudem besonders Steif ausgelegt und lässt nur wenig Spiel zu. Bei der geringen Drehzahl von 125 1/min muss der Sensor somit sehr kleine Schwingungen erfassen können.

Der ESP32 wird an der Unterseite der Konstruktion, neben dem Elektromotor angebracht. Zur Befestigung dient doppelseitiges Klebeband.

Die Zeit, in der die Daten aufgezeichnet und das Netzwerk trainiert wird, ist sehr viel kürzer als im vorherigen Test. Im Rahmen der Untersuchung soll gezeigt werden, dass neurale Netzwerke mit sehr wenig Daten und kurzer Trainingsdauer gute Ergebnisse liefern können.

Die Aufnahmezeit für die Datenakquirierung beträgt nur 15 min statt 60 min und die Trainingsepochen reduzieren sich auf lediglich 100 Epochen, was einer Trainingsdauer von nur 5 min entspricht. Dieser Test simuliert extreme Bedingungen und es ist nicht ratsam echte Anwendungen auf dieselbe Weise zu betreiben.

Das Vorgehen ist identisch zum ersten Test. Die Daten müssen zunächst aufgezeichnet und abgespeichert werden. Dabei befinden sich keine Gewichte an der Welle. Die Schwingungsdaten kann man in ein Diagramm anschaulich darstellen.

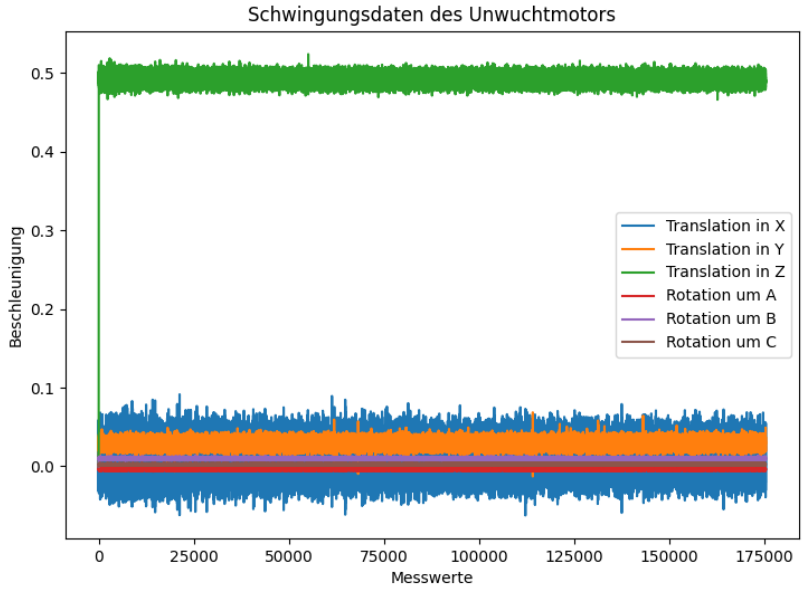

Abbildung 27: Schwingungsdaten des Unwuchtmotors

Das Schwingungsbild des Unwuchtmotors in Abb. 27 unterscheidet sich deutlich zu dem des Raumlüfters. Aufgrund der höheren Steifigkeit und geringeren Drehzahl sind die Amplituden deutlich kleiner. Die Achsen haben zudem eine andere Anordnung, da der Sensor in einer anderen Position montiert ist.

Das Training wird, anders wie im ersten Test, mit einem Laptop durchgeführt und dauert nur 5 Minuten. Der Trainingsverlauf lässt sich in einem Diagramm darstellen.

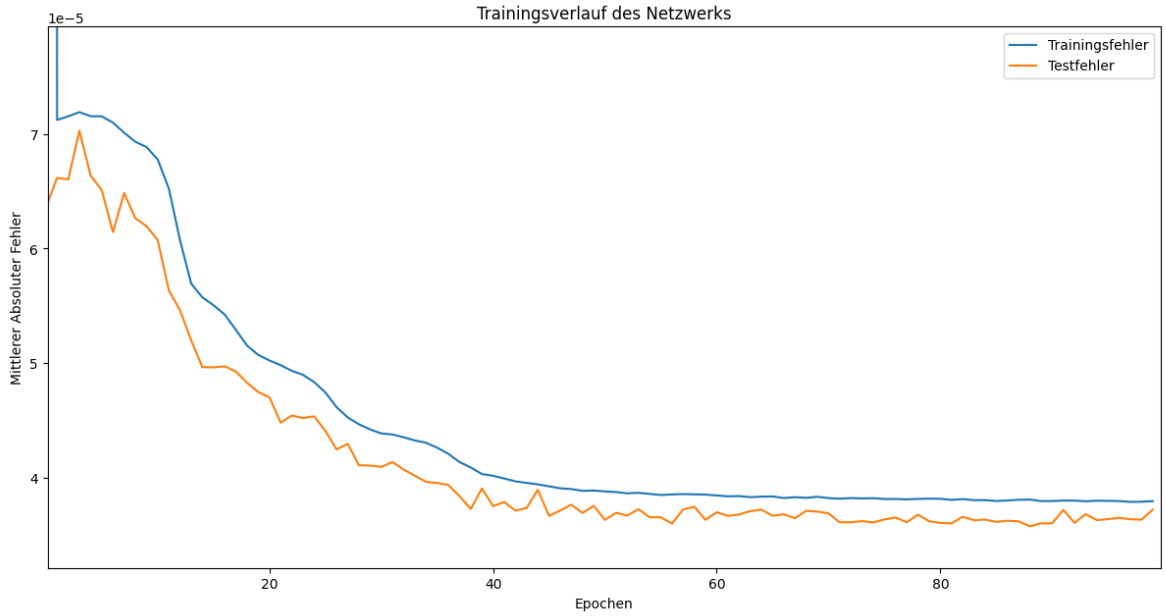

Abbildung 28: Trainingsverlauf Unwuchtmotor

Laut Abb. 28 konnte das Netzwerk gut, den Trainings-/ und Testfehler zu reduzieren. Der Testfehler steigt gegen Ende nicht an, somit liegt kein Overfitting vor. Dieser Verlauf deutet auf ein gutes Training hin.

Anschließend wird der Eingabevektor (Orginaldaten) mit dem Ausgabevektor (Rekonstruktion) verglichen, um die Rekonstruktionsgenauigkeit zu überprüfen. Die Rekonstruktion sollte nah an den Orginaldaten liegen.

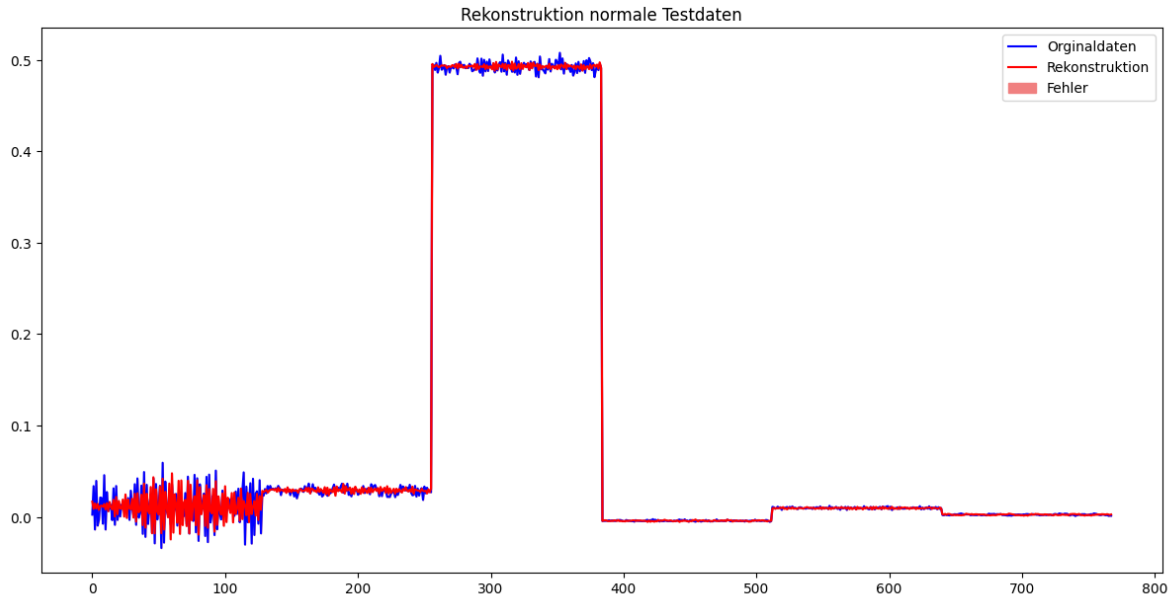

Abbildung 29: Rekonstruktion normaler Testdaten Unwuchtmotor

In Abb. 29 erkennt man den Unterschied zwischen den Orginaldaten und der Rekonstruktion. Die Güte der Rekonstruktion ist nicht so hoch wie beim ersten Test. In Abschnitt 0 bis 128 unterscheiden sich die Orginaldaten teilweise deutlich von der Rekonstruktion. Dennoch ist die Gesamtgenauigkeit zufriedenstellend. Das Netzwerk versucht die Daten zu rekonstruieren und es gibt keine Anzeichen von groben Abweichungen.

Das Netzwerk kann daher auf den Microcontroller übertragen werden. Anschließend zeichnet ein Computer den Rekonstruktionsfehler auf und speichert ihn ab.

Bei gleicher Drehzahl und ohne Gewichte wird der Rekonstruktionsfehler für 15 Minuten aufgezeichnet. Das entspricht dem Normalbetrieb und auch den Parametern, bei denen die Trainingsdaten aufgezeichnet wurden.

Die Ergebnisse lassen sich wieder in einem Diagramm darstellen.

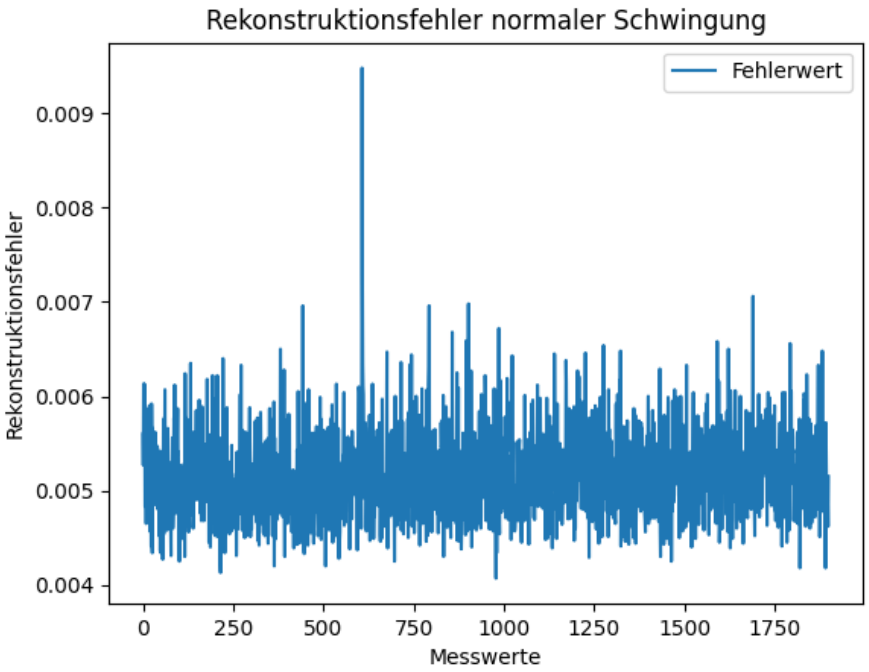

Abbildung 30: Rekonstruktionsfehler normaler Schwingung Unwuchtmotor

In Abb. 30 ist der Rekonstruktionsfehler der normalen Schwingung erkennbar. Sie hat große Ähnlichkeit mit der Rekonstruktion aus dem ersten Test. Verblüffend hierbei ist, dass der Rekonstruktionsfehler im Durchschnitt um mehr als den Faktor 10 niedriger ist als im Test mit dem Raumlüfter. Beim Raumlüfter lag der Fehler im Durchschnitt bei ca. 0,011 und hier bei 0,005. Dieser Fakt lässt darauf schließen, dass der Fehlerwert nicht pauschalisiert werden darf, sondern für jedes System einzigartig ist.

In den Scheiben des Unwuchtmotors wird nun eine Masse befestigt. Die Masse beschleunigt den Aufbau in radiale Richtung und erzeugt somit eine Schwingung, die das Netzwerk registrieren soll. In Abb. 31 kann man die Masse erkennen. Das Gewicht der Masse beträgt 60g.

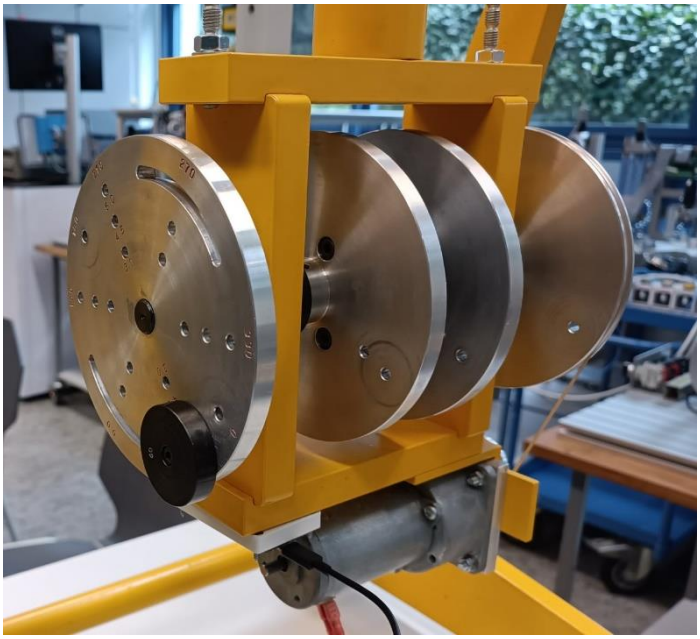

Abbildung 31: Massebehafteter Unwuchtmotor

Gemäß dem ersten Test mit dem Raumlüfter wird auch hier der Rekonstruktionsfehler der anormalen Schwingung aufgezeichnet. Die normale Schwingung kann nun mit der anormalen in einem Diagramm verglichen werden.

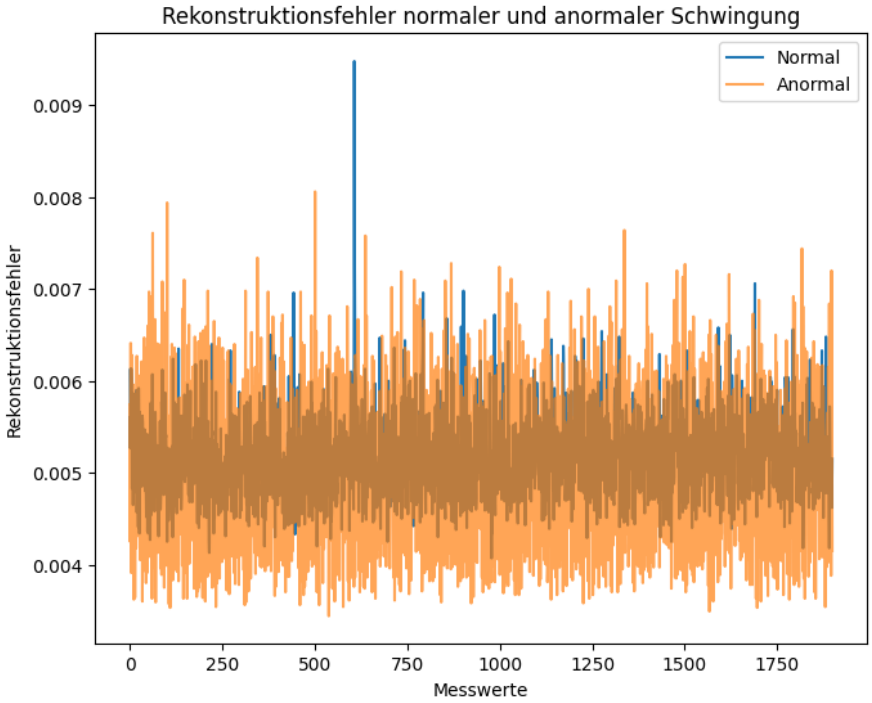

Abbildung 32: Rekonstruktionsfehler normaler und anormaler Schwingung

In Abb. 32 kann sich sehr schön der Unterschied zwischen normaler und anormaler Schwingung erkennen. Die anormale Schwingung (orange) besitzt eine viel höhere Varianz als die normale Schwingung (blau). Hier greift auch dieselbe Thematik wie im ersten Test. Der durchschnittliche Fehlerwert reicht nicht für die Unterscheidung von normaler und anormaler Schwingung aus. Für eine zuverlässige Unterscheidung muss zusätzlich die Varianz verglichen werden.

Die statistischen Werte in Tabelle 3 bestärken diesen Fakt.

Normale Rekonstruktion	Anormale Rekonstruktion
Durchschnitt: 0.0051690	Durchschnitt: 0.0050558
Varianz: 0.0004595	Varianz: 0.0008739
Min: 0.00407	Min: 0.00344
Max: 0.00948	Max: 0.00806

Tabelle 3: Statistische Werte normale und anormale Schwingungen Unwuchtmotor

Aus der Untersuchung geht hervor (Tabelle 3), dass bei der anormalen Rekonstruktion der Durchschnitt um 2,19% unter und die Varianz 90,18% über dem Niveau der normalen Rekonstruktion liegt. Die anormalen Daten haben somit eine eindeutig erkennbare Charakteristik und sind durch die hohe Varianz sehr leicht von den normalen Daten zu trennen.

Zusammenfassend lässt sich sagen, dass ein Test mit nur 15 Minuten Datenakquirierung und 5 Minuten Training eine sehr gute Unterscheidung zwischen normalen und anormalen Schwingungen erzielt hat.

17. Vorteile neuraler Netzwerke auf Microcontroller

Oft gelten neurale Netzwerke bzw. künstliche Intelligenz als sehr Speicher-/ und Leistungshungrig. Beispielsweise kostet der Betrieb von ChatGPT, der wohl bekanntesten KI unserer Zeit, 700.000$ pro Tag [36]. Die hier präsentierte Methode ist zwar nicht in der Lage Gedichte zu schreiben, dennoch hat sie etwas das große Modelle nicht bieten können. Sie ist günstig. Microcontroller und andere Edge Geräte (Microcontroller, Smartwatches, Smartphones, etc.) brauchen sehr wenig Strom für ihre Funktion. Ein ESP32 mit dem MPU-6050 benötigt im Betrieb nur maximal 0.78 Watt [37] elektrische Energie. Die Anschaffungskosten der Geräte sind ähnlich gering und belaufen sich auf unter 10€.

Neurale Netzwerke auf Microcontroller brauchen keine Verbindung zum Internet und können so auch an abgelegenen Orten verwendet werden. Zudem spart die lokale Berechnung in vielen Fällen Zeit, da man die Daten nicht zuerst auf einen externen Server schicken muss. Die Kosten für eine Cloud würden somit ebenfalls entfallen.

Kunden und Benutzer müssen sich zudem weniger Gedanken um ihre Sicherheit machen, da die Berechnungen und die Auswertung nur Lokal auf dem jeweiligen Gerät stattfinden.

Natürlich können sehr große Netzwerke, welche oft für Bildverarbeitung oder Textgenerierung verwendet werden, nicht auf einem Microcontroller funktionieren. Dennoch wurde in dieser Arbeit gezeigt, dass selbst sehr feine Auswertung von Schwingungen mit Microcontrollern gelöst werden können.

18. Fazit

In der vorliegenden Arbeit wurde zunächst der Tensorflow Framework untersucht, um herauszufinden, ob dieser sich für eine Schwingungsuntersuchung eignet. Dazu verwendete man reale Daten aus dem NASA Bearing Dataset und trainierte ein entsprechendes Autoencoder Netzwerk. Das Netzwerk war in der Lage, selbst kleinste Anomalien in den Daten zu identifizieren, wobei damit der erste Teil erfolgreich abgeschlossen wurde. Im zweiten Teil sollte das neurale Netzwerk auf einem ESP32 Microcontroller implementiert werden und Schwingungen aus einem MPU-6050 in Echtzeit verarbeiten und auswerten. Dabei wurden alle wichtigen Punkte von der Datenakquirierung bis hin zur eigentlichen Inference ausführlich dokumentiert und das Vorgehen beschrieben. Zwei anschließende Tests mit einem Raumlüfter und einem Unwuchtmotor konnten die Funktion des Netzwerkes bestätigen, normalen von anormalen Schwingungen zu unterscheiden.

19. Ausblick

Diese Arbeit soll einen Grundstein zukünftiger Arbeiten in Richtung künstlicher Intelligenz auf Edge Geräten legen. KI auf Edge Geräten nimmt eine immer größere Rolle ein. Gerade in Verbindung mit IoT-Geräten (eng. Internet of Things) schlummert hier ein großes, bisher unberührtes Potential. Durch Edge KI können Daten gezielt am Entstehungsort verarbeitet und bei guter Umsetzung viele Ressourcen gespart werden. Da unsere Welt zunehmend in der Digitalisierung voranstrebt, ist es nur sinnvoll, und auch nötig, sich mit solchen Dingen zu beschäftigen. Vor allem in der Industrie wird zukünftig der Einsatz von KI, angesichts Industrie 4.0 und Digitalisierung, eine Schlüsselrolle spielen. Daher ist es wichtig das nötige Wissen zu besitzen, um diese Technologie in die richtige Bahn zu lenken und verantwortungsvoll mit ihr umzugehen.

A. Bildquellen

[Abb. 2] Einfaches Deep Neural Network mit einer Zwischenschicht

https://upload.wikimedia.org/wikipedia/commons/thumb/e/e4/Artificial_neural_network.svg/1024px-Artificial_neural_network.svg.png

(zuletzt eingesehen am 07.01.23)

[Abb. 3] Tensorflow Logo

https://www.gstatic.com/devrel-devsite/prod/vdbc400b97a86c8815ab6ee057e8dc91626aee8cf89b10f7d89037e5a33539f53/tensorflow/images/lockup.svg

(zuletzt eingesehen am 07.01.23)

[Abb. 4] Tensorflow Flussdiagramm eines neuralen Netzwerks mit 2 Schichten

https://raw.githubusercontent.com/tensorflow/docs/master/site/en/guide/images/intro_to_graphs/two-layer-network.png

(zuletzt eingesehen am 07.01.23)

[Abb. 5] Funktionsweise eines Autoencoders

https://www.tensorflow.org/static/tutorials/generative/images/intro_autoencoder_result.png

(zuletzt eingesehen am 07.01.23)

[Abb. 6] Aufbau eines Autoencoders mit mehreren Zwischenschichten

https://en.wikipedia.org/wiki/Autoencoder#/media/File:Autoencoder_structure.png

(zuletzt eingesehen am 07.01.23)

[Abb. 11] ESP32 NodeMCU auf einem Entwicklerboard
https://www.berrybase.de/media/image/4d/b3/8a/ID_207304_orig.jpg
(zuletzt eingesehen am 01.06.2023)

[Abb. 12] MPU-6050 6 Achsen Bewegungssensor
https://tse1.mm.bing.net/th?id=OIP.zns3bkSrMvVkTalWBCcj3AHaHa&pid=Api
(zuletzt eingesehen am 01.06.2023)

[Abb. 21] Diagramm für Quantifizierung
https://www.tensorflow.org/static/lite/performance/images/optimization.jpg
(zuletzt eingesehen am 01.06.2023)

(Alle hier nicht genannten Abbildungen sind Eigenanfertigungen)

B. Quellcode und Tutorien

Quellcode auf GitHub:
https://github.com/Palettenbrett/Schwingungsuntersuchung_mit_Microcontrollern

Erklärvideo für die richtige Verwendung der Software:
https://www.youtube.com/watch?v=GsNp37Y7YcQ

C. Literatur

[1] S. V, „AN EMPIRICAL SCIENCE RESEARCH ON BIOINFORMATICS IN MACHINE LEARNING,“ *JMCMS*, Jg. spl7, Nr. 1, 2020, doi: 10.26782/jmcms.spl.7/2020.02.00006.

[2] „artificial intelligence, n. : Oxford English Dictionary.” https://www.oed.com/viewdictionaryentry/Entry/271625 (Zugriff am: 2. Januar 2023).

[3] News Center Microsoft Deutschland. „Microsoft erklärt: Was ist Machine Learning? Definition & Funktionen von ML | News Center Microsoft.” https://news.microsoft.com/de-de/microsoft-erklaert-was-ist-machine-learning-definition-funktionen-von-ml/ (Zugriff am: 2. Januar 2023).

[4] A. Brahme, Hg. *Comprehensive biomedical physics*. Amsterdam: Elsevier, 2014. [Online]. Verfügbar unter: https://www.sciencedirect.com/science/referenceworks/9780444536334

[5] „What are Neural Networks? | IBM.” https://www.ibm.com/topics/neural-networks (Zugriff am: 3. Januar 2023).

[6] I. Salian. „NVIDIA Blog: Supervised Vs. Unsupervised Learning.” https://blogs.nvidia.com/blog/2018/08/02/supervised-unsupervised-learning/ (Zugriff am: 3. Januar 2023).

[7] GitHub. „tensorflow/tensorflow: An Open Source Machine Learning Framework for Everyone.” https://github.com/tensorflow/tensorflow (Zugriff am: 2. Januar 2023).

[8] „Introduction to graphs and tf.function | TensorFlow Core.” https://www.tensorflow.org/guide/intro_to_graphs (Zugriff am: 2. Januar 2023).

[9] „Case Studies and Mentions | TensorFlow.” https://www.tensorflow.org/about/case-studies?filter=all (Zugriff am: 2. Januar 2023).

[10] GitHub. „Release TensorFlow 1.4.0 · tensorflow/tensorflow.” https://github.com/tensorflow/tensorflow/releases/tag/v1.4.0 (Zugriff am: 3. Januar 2023).

[11] Maarten Meire und Peter Karsmakers. „Comparison of Deep Autoencoder Arcitectures for Real-time Acoustic Based Anomaly Detection in Assets.”

[12] TensorFlow. „Intro to Autoencoders | TensorFlow Core.” https://www.tensorflow.org/tutorials/generative/autoencoder (Zugriff am: 3. Januar 2023).

[13] M. A. Kramer, „Nonlinear principal component analysis using autoassociative neural networks,“ *AIChE J.*, Jg. 37, Nr. 2, S. 233–243, 1991. doi: 10.1002/aic.690370209. [Online]. Verfügbar unter: https://aiche.onlinelibrary.wiley.com/doi/10.1002/aic.690370209

[14] V. Tyagi. „NASA Bearing Dataset.” https://www.kaggle.com/datasets/vinayak123tyagi/bearing-dataset (Zugriff am: 4. Januar 2023).

[15] T. Shah, „About Train, Validation and Test Sets in Machine Learning," *Towards Data Science*, 06. Dezember 2017. https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7 (Zugriff am: 4. Januar 2023).
[16] S. Ioffe und C. Szegedy, „Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," Feb. 2015. [Online]. Verfügbar unter: https://arxiv.org/pdf/1502.03167
[17] Google Developers. „Normalisierung | Machine Learning | Google Developers." https://developers.google.com/machine-learning/data-prep/transform/normalization (Zugriff am: 5. Januar 2023).
[18] „What is Underfitting? | IBM." https://www.ibm.com/topics/underfitting (Zugriff am: 5. Januar 2023).
[19] D. H. Hubel und T. N. Wiesel, „Receptive fields and functional architecture of monkey striate cortex," *The Journal of Physiology*, Jg. 195, Nr. 1, S. 215–243, 1968. doi: 10.1113/jphysiol.1968.sp008455. [Online]. Verfügbar unter: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1557912/
[20] P. Mahapattanakul, „From Human Vision to Computer Vision — Convolutional Neural Network(Part3/4)," *Becoming Human: Artificial Intelligence Magazine*, 10. November 2019. https://becominghuman.ai/from-human-vision-to-computer-vision-convolutional-neural-network-part3-4-24b55ffa7045 (Zugriff am: 6. Januar 2023).
[21] D. Y. Oh und I. D. Yun, „Residual Error Based Anomaly Detection Using Auto-Encoder in SMD Machine Sound," *Sensors (Basel, Switzerland)*, Early Access. doi: 10.3390/s18051308.
[22] M. V. Valueva, N. N. Nagornov, P. A. Lyakhov, G. V. Valuev und N. I. Chervyakov, „Application of the residue number system to reduce hardware costs of the convolutional neural network implementation," *Mathematics and Computers in Simulation*, Jg. 177, S. 232–243, 2020. doi: 10.1016/j.matcom.2020.04.031. [Online]. Verfügbar unter: https://www.sciencedirect.com/science/article/pii/S0378475420301580
[23] TensorFlow. „TensorFlow Lite | ML for Mobile and Edge Devices." https://www.tensorflow.org/lite (Zugriff am: 1. Juni 2023).
[24] „TensorFlow Lite for Microcontrollers," https://www.tensorflow.org/lite/microcontrollers (Zugriff am: 1. Juni 2023).
[25] A. Meroth und P. Sora, *Sensornetzwerke in Theorie und Praxis: Embedded Systems-Projekte erfolgreich realisieren,* 2. Aufl. Wiesbaden: Springer Fachmedien Wiesbaden GmbH; Springer Vieweg, 2021.
[26] „pySerial — pySerial 3.4 documentation." https://pyserial.readthedocs.io/en/latest/pyserial.html (Zugriff am: 1. Juni 2023).
[27] Y. Dodge und D. Cox, Hg. *The Oxford dictionary of statistical terms,* 2006. Aufl. Oxford: Oxford University Press, 2006.

[28] GitHub. „GitHub - lutzroeder/netron: Visualizer for neural network, deep learning, and machine learning models.” https://github.com/lutzroeder/netron (Zugriff am: 1. Juni 2023).
[29] „TensorFlow Lite and TensorFlow operator compatibility,” https://www.tensorflow.org/lite/guide/ops_compatibility (Zugriff am: 1. Juni 2023).
[30] D. Masters und C. Luschi, „Revisiting Small Batch Training for Deep Neural Networks,“ Apr. 2018. [Online]. Verfügbar unter: http://arxiv.org/pdf/1804.07612v1
[31] „Post-training quantization,” https://www.tensorflow.org/lite/performance/post_training_quantization (Zugriff am: 28. Mai 2023).
[32] B. Jacob *et al.,* „Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference,“ Dez. 2017. [Online]. Verfügbar unter: http://arxiv.org/pdf/1712.05877v1
[33] TensorFlow. „tf.lite.RepresentativeDataset | TensorFlow Lite.” https://www.tensorflow.org/lite/api_docs/python/tf/lite/RepresentativeDataset (Zugriff am: 1. Juni 2023).
[34] „Historical trends in the usage statistics of character encodings for websites, June 2023.” https://w3techs.com/technologies/history_overview/character_encoding (Zugriff am: 1. Juni 2023).
[35] GitHub. „tflite-micro/tensorflow/lite/micro/examples/hello_world at main · tensorflow/tflite-micro.” https://github.com/tensorflow/tflite-micro/tree/main/tensorflow/lite/micro/examples/hello_world (Zugriff am: 1. Juni 2023).
[36] A. Mok, „How much does ChatGPT cost to run? $700K/day, per analyst,“ *Insider*, 20. April 2023. https://www.businessinsider.com/how-much-chatgpt-costs-openai-to-run-estimate-report-2023-4 (Zugriff am: 1. Juni 2023).
[37] Armin Vogt. „Forum für Camper Selbstausbauer.” https://forum.camper-bauen.de/viewtopic.php?t=2770 (Zugriff am: 1. Juni 2023).