

Masterarbeit

Computer Science

**Sensorbasierter Orientierungssinn
mit künstlichen neuronalen Netzen
und Entscheidungsbäumen**

von

Tom Dymel

Juni 2021

Betreut von

Dr. rer. nat. Marcus Venzke
Institut für Telematik, Technische Universität Hamburg

Erstprüfer | Prof. Dr. rer. nat. Volker Turau
Institut für Telematik
Technische Universität Hamburg

Zweitprüfer | Prof. Dr. sc. techn. Christian Schuster
Institut für Theoretische Elektrotechnik
Technische Universität Hamburg

Danksagung

Vorerst möchte ich mich bei dem Institut für Telematik bedanken, dass ich meine Masterarbeit zu diesem spannenden Thema verfassen durfte. Insbesondere möchte ich Dr. Marcus Venzke für die umfangreiche Betreuung bedanken, ohne dessen Beratung diese Arbeit nicht in dieser Qualität existieren würde. Zuletzt möchte ich mich bei Malte Laukötter und Jannik Friemann für ihre Korrekturen und Denkanstöße bedanken.

Eidesstattliche Erklärung

Ich, TOM DYMEL (Student im Studiengang Computer Science an der Technischen Universität Hamburg, Matr.-Nr. 21651529), versichere an Eides statt, dass ich die vorliegende Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe. Die Arbeit wurde in dieser oder ähnlicher Form noch keiner Prüfungskommission vorgelegt.

Hamburg, 1. Juni 2021

Tom Dymel

Inhaltsverzeichnis

1 Einleitung	1
2 Entscheidungsbäume	3
2.1 Scikit-Learn	3
2.2 Einzelne Entscheidungsbäume	4
2.3 Ensemble-Methoden	5
2.4 Training mit Scikit-Learn	7
2.5 Ressourcenbedarf auf dem Mikrocontroller	7
2.5.1 Ausführungszeit und Energieverbrauch	8
2.5.2 Programmgröße und RAM	9
3 Künstliche Neuronale Netze	11
3.1 Keras	13
3.2 Training von KNN	13
3.3 Lernalgorithmen	14
3.4 Aktivierungsfunktionen	17
3.5 Ressourcenbedarf auf dem Mikrocontroller	19
4 Standortbestimmung	21
4.1 Orientierungssinn von Mensch und Tier	21
4.2 Indoor- und Outdoor-Lokalisierung	22
4.3 WiFi basierte Indoor-Lokalisierung mit Transfer Lernen	23
4.4 Indoor-Lokalisierung mit Magnet- und Lichtsensoren	23
4.5 Sensorbasierter Orientierungssinn mit FFNN	24
5 Machine Learning Modelle	27
5.1 Standortenkodierung	28
5.2 Entscheidungswald	29
5.3 Feed Forward neuronales Netzwerk	30
5.4 Training der ML-Modelle	30
5.5 Anomalieerkennung	33
6 Trainings- und Testdaten	35
6.1 Sensordaten aus Simulator CoppeliaSim	35
6.2 Mit eignen Modellen ergänzte Sensordaten	37
6.2.1 Magnetfeldsensor	37
6.2.2 Temperatursensor	41
6.2.3 Geräuschsensor	41
6.2.4 WLAN-Zugangspunkte	42
6.3 Simulation von Interrupts	42
6.4 Feature-Extrahierung	43

INHALTSVERZEICHNIS

6.5	Synthetische Daten	45
6.6	Anomaliedaten	46
7	Evaluation	49
7.1	Metriken	49
7.2	Klassifizierungsgenauigkeit der Standorte	51
7.3	Klassifizierungsgenauigkeit der Anomalien	51
7.4	Signifikanz der Features	51
7.5	Benötigte Anzahl der Trainingsdaten	52
7.6	Fehlertoleranz	52
7.7	Programmspeicher	53
7.8	RAM	54
7.9	Ausführungszeit und benötigte Energie	56
8	Diskussion	59
9	Schlussfolgerungen	61
A	Inhalt des USB-Sticks	63
B	Bildanhänge	65
	Literaturverzeichnis	71

Einleitung

Als Standortbestimmung, oder *Lokalisierung*, wird der Prozess bezeichnet die Position von einem Gerät oder Nutzer in einem Koordinatensystem zu bestimmen [BHE00]. Diese Information wird von technischen Systemen genutzt, um deren Dienste anzubieten, z. B. Tracking- oder Navigationssysteme. Ein bekanntes Beispiel ist das *Global Positioning System* (GPS) [KH05]. Bei GPS berechnet das empfangende Gerät seine Position basierend auf den empfangenden Signalen der Satelliten.

In Gebäuden ist die Signalstärke der GPS-Satelliten jedoch stark eingeschränkt, sodass die Ortung ungenau wird oder überhaupt nicht funktioniert. Aus diesem Grund werden für *Indoor-Lokalisation* andere Ansätze verfolgt. Je nach Genauigkeit können Objekte mit Sendern, RFID Tags oder Barcodes markiert werden [XZYN16]. Meistens wird eine komplexe Infrastruktur benötigt, die diese Lokalisierungssysteme vergleichsweise teuer macht.

Eine weitere Art der Lokalisation ist der Orientierungssinn von Mensch und Tier. Anhand von Orientierungspunkten wird so von einem Punkt zu einem anderen Punkt navigiert. Beispielsweise Honigbienen navigieren auf Basis von gelernten Orientierungspunkten, um Nahrungsquelle und Nest zu finden [MGC⁺96].

In dieser Arbeit wird die diskrete Positionsbestimmung basierend auf Sensordaten untersucht. Dabei soll eine bestimmte Anzahl von Standorten anhand verschiedener Sensorwerte unterschieden werden. Dies ähnelt dem zuvor beschriebenen Orientierungssinn. Mit Hilfe maschinellen Lernens sollen künstliche neuronale Netze (KNN) und Entscheidungsbäume trainiert werden. Als Eingabedaten werden aus den gesammelten Sensordaten Features extrahiert, d. h. Attribute und Eigenschaften dieser Daten. Dieses System bedarf keiner Infrastruktur, muss aber für jedes Gebäude individuell trainiert werden. Damit unterscheidet sich diese Arbeit sich von anderen Arbeiten, da mehrere Sensoren zur diskreten Standorterkennung

1 EINLEITUNG

kombiniert werden für den Einsatz auf einem Mikrocontroller.

Es wird ein Evaluierungsprozess der Sensordaten erstellt, um Standorte und Anomalien mit Hilfe von ML-Modellen zu unterscheiden. Dafür werden Sensordaten mit CoppeliaSim simuliert, da Echtdaten eines Prototyps zum Zeitpunkt dieser Arbeit noch nicht Verfügbar sind. Die simulativ erfassten Sensordaten werden mit weiteren Sensordaten auf Basis vereinfachter Modelle ergänzt. Es wird die Klassifizierungsgenauigkeit der ML-Modelle untersucht im Hinblick auf Skalierbarkeit der Anzahl der Standorte, sowie Robustheit gegenüber mögliche Fehler. Außerdem wird die Klassifizierungsgenauigkeit des Anomalieerkennungsmodells untersucht. Zuletzt wird der Speicher- und Energieverbrauch eingeschätzt für den Betrieb auf einem batteriestützten Mikrocontroller.

Kapitel 2 führt Entscheidungsbäume und Ensemble-Methoden ein. Kapitel 3 führt künstliche neuronale Netze (KNN) mit dem Fokus auf Feed Forward neuronale Netze (FFNN) ein. In Kapitel 4 wird auf den Stand der Forschung für Standortbestimmung mit Hilfe von maschinellen Lernen (ML) eingegangen. Das Trainieren der ML-Modelle mit Entscheidungsbäumen und FFNN wird in Kapitel 5 erläutert. Kapitel 6 geht auf die Generierung von Trainings- und Validationsdaten auf Basis von Simulationen ein. Darauf folgt die Evaluation der Klassifizierungsgenauigkeit, Fehlertoleranz und Resourcennutzung in Kapitel 7. Kapitel 8 enthält einen kritischen Rückblick auf die Entscheidungen dieser Arbeit, bevor Kapitel 9 Schlussfolgerungen zieht.

Entscheidungsbäume

Ein Entscheidungsbaum ist ein Baum mit dem Entscheidungen getroffen werden [Qui90]. Das geschieht, indem der Baum von der Wurzel zu einem Blatt traversiert wird. Dabei bestimmt ein Test in jedem inneren Knoten, mit welchem Kindknoten fortgefahrene wird. Jedes Blatt entspricht einer Entscheidung des Entscheidungsbaums. Es wird unterschieden zwischen Bäumen, die versuchen eine der vordefinierten Klassen zu klassifizieren (Klassifizierer) und solchen, die versuchen den nächsten Wert vorherzusagen (Regressoren).

Die Konstruktion eines optimalen binären Entscheidungsbaumes ist NP-Vollständig [LR76]. Den optimalen Klassifizierer zu finden ist folglich sehr aufwendig. Aus diesem Grund werden bei der Konstruktion Heuristiken verwendet, die nur lokal die beste Entscheidung treffen. Zudem werden einzelne Entscheidungsbäume in einem Ensemble zu einen Entscheidungswald zusammengefasst, um den Fehler eines einzelnen Baumes zu reduzieren [Ent20b].

2.1 Scikit-Learn

Diese Arbeit verwendet die Python ML-Bibliothek *Scikit-Learn*. Scikit-Learn bietet verschiedene ML Algorithmen mit einem High-Level Interface an [PVG⁺11]. Zur Konstruktion der Entscheidungsbäume wird der Algorithmus von CART verwendet [Ent20a]. Die Bibliothek kann Klassifizierer und Regressoren generieren.

Nachfolgend wird nur von Klassifizierern ausgegangen, da nur diese für diese Arbeit benötigt werden. Diese bieten zahlreiche Hyperparameter an, um die Konstruktion des Entscheidungsbaumes zu steuern. In dieser Arbeit wird der Hyperparameter `max_depth` verwendet, der die maximale Baumhöhe beschränkt und somit den Programmspeicherverbrauch begrenzt.

2 ENTScheidungsbäume

Scikit-Learn bietet Ensemble-Methoden an, um Entscheidungswälder zu trainieren. Ein wichtiger Parameter dieser Methoden ist `n_estimators`. Er steuert die Größe des Ensembles bzw. die Waldgröße. Somit hat auch dieser Parameter Einfluss auf den Programmspeicherverbrauch.

2.2 Einzelne Entscheidungsbäume

Ein einzelner Entscheidungsbaum ist eine rekursive Datenstruktur um Entscheidungsregeln darzustellen [Qui90]. Jeder innere Knoten ist ein *Test*, welcher eine arbiträre Anzahl von sich gegenseitig ausschließenden Ergebnissen hat. Das Ergebnis eines Tests bestimmt mit welchem Kindknoten fortgefahrene wird. Die Blätter des Baumes stellen die Entscheidungen dar bzw. die Klassen des Entscheidungsbaumklassifizierers. Abbildung 2.1 zeigt einen binäreren Entscheidungsbaum, in dem jeder Test zwei mögliche Ergebnisse hat. Das Trainieren von



■ **Abbildung 2.1:** Beispiel eines binären Entscheidungsbaums mit 3 möglichen Ergebnissen.

Entscheidungsbäumen ist eine Art von *Supervised Learning*, d. h. aus einer beschrifteten Trainingsmenge werden Regeln abgeleitet, um das korrekte Mapping von Input zu Output abzubilden [GL09]. Die Trainingsmenge besteht aus Feature-Mengen, die mit Klassen beschriftet sind [Ste09]. Die Generalisierungsfähigkeit ist abhängig von der Trainingsmenge. Zum einen sollte die Trainingsmenge möglichst repräsentativ sein für die Aufgabe, die gelernt werden soll. Zum anderen sollten die verwendeten Features eine Partitionierung aller Klassen ermöglichen [PGP98].

Entscheidungsbäume werden heuristisch konstruiert, da die Konstruktion eines optimalen Entscheidungsbäumes NP-Vollständig ist [LR76]. Zu diesen Algorithmen gehören beispielsweise ID3 [Qui86], C4.5 [Qui14] oder CART [BFSO84]. Die Aufgabe ist durch gezielte Trennungen eine Partitionierung der Trainingsmenge zu erzeugen, sodass möglichst nur Einträge mit der gleichen Beschriftung in einer Partitionierung enthalten sind. Die Algorithmen unterscheiden sich in ihrer Strategie [Qui86].

Scikit-Learn implementiert eine optimierte Version des **CART** (Classification And Regression Trees) Algorithmus [Ent20a]. CART partitioniert die Trainingsmenge indem lokal immer die beste Teilung ausgewählt wird. Die beste Teilung ist die Teilungregel, die Einträge mit der gleichen Beschriftung möglichst gut von anderen Einträgen trennt. Dieser Vorgang wird rekursiv mit jeder Teilmenge wiederholt, bis keine weitere Teilung mehr möglich ist oder alle Einträge einer Partitionierung die gleiche Beschriftung tragen [Ste09].

2.3 Ensemble-Methoden

Ensemble-Methoden beschreiben wie mehrere Entscheidungsbäume trainiert werden, um eine möglichst hohe Diversität der einzelnen Entscheidungsbäume zu erzielen. Das Ergebnis eines Ensembles ist die Aggregation der Ergebnisse der einzelnen Entscheidungsbäume [D⁺02].

Der Wahlklassifizierer $H(x) = w_1h_1(x) + \dots + w_Kh_K(x)$ ist eine Möglichkeit die Einzelergebnisse $\{h_1, \dots, h_K\}$ gewichtet mit $\{w_1, \dots, w_K\}$ zu aggregieren [D⁺02]. Ein Ergebnis kann auf zwei Arten modelliert sein. Einerseits als eine Funktion $h_i : D^n \mapsto \mathbb{R}^m$, die einer n -dimensionalen Menge D^n jeder der m möglichen Klassen eine Wahrscheinlichkeit zuweist. Das Ergebnis ist eine Wahrscheinlichkeitsverteilung. Das diskrete Ergebnis der Klassifizierung ist die Klasse mit der höchsten Wahrscheinlichkeit in dem Ergebnis. Alternativ kann es als eine Funktion $h_i : D^n \mapsto M$ abgebildet werden, die diskret auf eine der möglichen Klassen in M verweist [Dym21]. In diesem Fall wird die Klasse ausgewählt, die am häufigsten unter allen Einzelergebnissen vorkam. In der Praxis wird die Aggregation der Wahrscheinlichkeitsverteilung genutzt [Ent20b]. Analog ist $H : D^n \mapsto \mathbb{R}^m$ oder $H : D^n \mapsto M$ definiert [D⁺02]. Für gewöhnlich hat jeder Teilnehmer einer Wahl das gleiche Gewicht.

Bagging (**Bootstrap aggregating**) konstruiert Entscheidungswälder, indem es Entscheidungsbäume mit Teilmengen der Trainingsmenge trainiert. Abbildung 2.2 illustriert die Bagging Methode für n Entscheidungsbaummodelle. Zunächst wird die Trainingsmenge in n Teilmengen aufgeteilt [Bre96]. Der Inhalt der Teilmengen wird mit der „Bootstrap sampling“ Methode bestimmt. Diese zieht aus einer Grundmenge l -mal jeweils k -Einträge [Efr92]. Mit jeder Teilmenge wird ein Entscheidungsbaum trainiert [Bre96]. Die Einzelergebnisse werden aggregiert, z. B. mit dem Wahlklassifizierer.

Random Forest erweitert die Bagging-Methode [Bre01]. Für jeden Entscheidungsbaum der trainiert werden soll, wird zusätzlich zufällig eine Teilmenge der Feature-Menge ausgewählt.

2 ENTSCHEIDUNGSBÄUME



Abbildung 2.2: Klassifizierungsprozess mit der Bagging-Methode.

Extremely Randomized Trees (ExtraTrees) verwenden ebenfalls eine Teilmenge der Feature-Menge der Trainingsmenge beim Trainieren der einzelnen Entscheidungsbäume [GEW06]. Allerdings wird für jeden Entscheidungsbau die gesamte Trainingsmenge verwendet. Bei der Konstruktion wird nicht versucht die beste Teilungsregel zu finden, sondern es werden zufällig Teilungsregeln generiert, aus denen die Beste ausgewählt wird.

Beim Boosting werden nacheinander schwache Lerner auf einer Teilmenge trainiert, die gewichtet aggregiert werden [FS97]. Dadurch entsteht ein starker Lerner. Abbildung 2.3 illustriert, wie vier schwache Lerner trainiert werden. Jeder Lerner findet eine Funktion der die trainierte Teilmenge unterteilt. Anschließend werden sie gewichtet aggregiert. Dies konstruiert einen starken Lerner, der die gesamte Trainingsmenge unterteilt. Diese Arbeit verwendet für Boosting den Algorithmus AdaBoost [FS97] von Freund und Schapire.



Abbildung 2.3: Klassifizierungsprozess mit der Boosting-Methode.

2.4 Training mit Scikit-Learn

Die Konstruktion eines Entscheidungsbaumes mit Scikit-Learn ist nicht deterministisch [Dym21]. Bei der Konstruktion mit CART können zwei Teilungsregeln gefunden werden, die eine gleich gute Unterteilung erzeugen. In diesem Fall wählt Scikit-Learn zufällig eine Teilung aus. Dadurch werden anschließende Teilungen beeinflusst, die in einen der Fälle womöglich bessere Teilungen hätten finden können. Dieser Zufall ist steuerbar, indem der Startwert des Zufallgenerators auf einen vordefinierten Wert gesetzt wird.

Bei identischer Trainingsmenge und Konfiguration können folglich für verschiedene Startwerte unterschiedliche Modelle erzeugt werden. Aus diesem Grund kann das Training mit einem Startwert als Monte Carlo Methode verstanden werden, d. h. wiederholtes Ausführen unter verschiedenen Startwerten erhöht die Wahrscheinlichkeit, dass das beste Modell unter der angegebenen Konfiguration gefunden wird.

Dies bedarf, dass die Klassifizierungsgenauigkeit beim Training bereits ermittelt wird, damit das beste Modell ausgewählt werden kann. Dafür wird die Trainingsmenge in zwei Mengen unterteilt. Mit einer Teilmenge werden die Entscheidungsbäume unter verschiedenen Startwerten des Zufallgenerators trainiert und mit der anderen Teilmenge werden diese untereinander verglichen. Das Ergebnis des Trainingprozesses ist das Modell, dass auf letzterer Teilmenge die höchste Klassifizierungsgenauigkeit erzielt.

2.5 Ressourcenbedarf auf dem Mikrocontroller

Zukünftig soll das Modell auf einem Mikrocontroller ausgeführt werden [Ven21]. Mikrocontroller sind stark limitiert in ihrer Rechenleistung, Speicherkapazität, RAM und werden oft zudem mit einer Batterie betrieben. Aus diesem Grund ist der Energieverbrauch zu minimieren und das Modell muss innerhalb dieser Limitierungen operieren können.

2 ENTSCHEIDUNGSBÄUME

2.5.1 Ausführungszeit und Energieverbrauch

Der Energieverbrauch korreliert mit der Ausführungszeit. Je länger die CPU ausgeschaltet ist, desto weniger Energie wird verbraucht. Kurze Ausführungszeiträume vergrößern den Zeitraum, in dem die CPU ausgeschaltet sein kann. Die Ausführungszeit ist die Zeit die benötigt wird, um alle Instruktionen auszuführen [Dym21]. Jede Instruktion bedarf eine bestimmte Anzahl an CPU-Zyklen. Die Zeit pro Zyklus ist abhängig von der Taktrate der CPU.

Die Ausführungszeit eines Entscheidungswaldes setzt sich zusammen aus der Zeit für die Feature-Extrahierung, der Evaluierung aller im Ensemble enthaltenen Entscheidungsbäume und der Aggregierungsfunktion. Im schlimmsten Fall muss die gesamte Höhe eines Entscheidungsbäumes traversiert werden, um das Ergebnis zu bestimmen. Aus diesem Grund skaliert die Ausführungszeit mit der traversierten Höhe jedes Baumes.

Um die Instruktionen zu minimieren sollten Datentypen verwendet werden, die von der CPU mit höchstens einem Wort dargestellt werden können. Eine 8-Bit CPU würde zum Laden in Register eines 32-Bit Datentypen vier mal so viele Instruktionen benötigen wie bei einem 8-Bit Datentypen. Außerdem sollten Operationen verwendet werden, die durch native Hardware-Operationen abgebildet werden können. Ist dem nicht so, muss diese Operation durch Software ersetzt werden. Dies erfordert mehr Zyklen als eine native Operation in Hardware.

Zu Beachten bei der Minimierung ist, dass Instruktionen unterschiedlich viele Zyklen benötigen und Funktionsaufrufe Overhead erzeugen. Ein Beispiel dafür ist die Optimierung *Function Inlining* [LM99]. Der Aufruf von Funktionen kann einen hohen Overhead durch den Kontextwechsel erzeugen. Aus diesem Grund verringert diese Optimierung die Ausführungszeit, erhöht aber die die Programmgröße signifikant. Im Umkehrschluss könnten durch die Verwendung von Funktionen der nutzen des Programmspeichers verringert werden, Ausführungszeit und Energieverbrauch aber erhöht werden.

2.5.2 Programmgröße und RAM

Die Programmgröße ist die Gesamtheit aller Instruktionen die für das Programm benötigt werden [Dym21]. Dabei ist der Anteil für die Entscheidungswälder integral und der Anteil für die perifären Funktionalitäten zu vernachlässigen. Die Programmgröße, die für einen Entscheidungswald benötigt wird, skaliert mit der Waldgröße und Höhe der einzelnen Entscheidungsbäume.

Die Höhe des Entscheidungsbaumes ist die Verzweigungstiefe der verschachtelten Tests. Jeder Test ist ein Vergleich mit einem Schwellenwert. Die Programmgröße für einen Vergleich setzt sich zusammen aus den Operationen, um die Operanden in die Register zu laden, und die Instruktion, um den Vergleich durchzuführen, sowie Abzweiginstruktionen. Wie in Kapitel 2.5.1 sind Instruktionen durch einen passenden Datentypen zu vermeiden.

Ein weiterer Faktor sind die Instruktionen, die zur Rückgabe des Klassifizierungsergebnis benötigt werden. In Kapitel 2.3 wurden verschiedene Möglichkeiten der Rückgabe diskutiert, die relevant bei dem Aggregierungsprozess eines Ensembles ist. Einerseits kann die Rückgabe eine Wahrscheinlichkeitsverteilung sein und andererseits eine diskrete Klasse. Bei m möglichen Klassen würde die erste Variante m -mal so viele Instruktion benötigen, wie die zweite Variante, da der Rückgabevektor zuvor mit der Wahrscheinlichkeitsverteilung gefüllt werden muss. In der Praxis werden aber weniger Instruktion benötigt, da es eine große Überschneidung der Wahrscheinlichkeitsverteilungen gibt, die zurück gegeben werden. Die Instruktionen, um den Rückgabevektor zu befüllen, können durch *Basic Blocks*, d. h. beschriftete Instruktionsblöcke, geschickt recycled werden. Zudem können Zuweisungen ausgelassen werden, die die Wahrscheinlichkeit 0 zuweisen, da der Vektor mit Nullen initialisiert wird. Dennoch werden signifikant mehr Instruktionen benötigt als bei der diskreten Variante. Aus diesem Grund wurde ein hybrider Ansatz vorgeschlagen, der im Falle eines eindeutigen Ergebnisses mit einer Toleranz von $\epsilon \in [0, 1]$ die diskrete Klasse statt der Wahrscheinlichkeitsverteilung zurück gibt.

Der benötigte RAM ist abhängig von der Größe der Feature-Menge und Anzahl zu klassifizierenden Klassen. Für die Rückgabe wird zwischen ein Byte und mQ Byte benötigt, wobei Q die Größe des verwendeten Datentypen ist, um eine Gleitkommazahl zu speichern.

2 ENTSCHEIDUNGSBÄUME

Künstliche Neuronale Netze

Das mathematische Modell von künstlichen neuronalen Netzen wurde von McCulloch und Pitts im Jahre 1943 erfunden [MP43]. Dieses Modell ist eine Abstraktion des biologischen Neuronen als logischer Mechanismus.

Das Nervensystem besteht aus einem Netz von Neuronen, die miteinander verbunden sind und über elektrische Impulse miteinander interagieren [Ros61]. Man unterscheidet beim biologischen Neuronen zwischen *Afferent*-Neuronen, *Efferent*-Neuronen und *Inter*-Neuronen. Afferent-Neuronen nehmen elektrische Signale von Organen entgegen und können als *Input* interpretiert werden. Efferent-Neuronen geben elektrische Signale an *Effektorzellen* weiter und können als *Output* interpretiert werden. Inter-Neuronen nehmen elektrische Signale von Afferent-Neuronen oder Inter-Neuronen entgegen und geben sie an Inter-Neuronen oder Efferent-Neuronen weiter. Wenn der Schwellenwert eines *Dendrite* von einem Neuronen durch ein elektrisches Signal erreicht wurde, wird ein elektrisches Signal über den *Axon* an ein anderes Neuron oder Effektorzellen übertragen.

Diese Charakteristiken werden mathematisch als ein Vergleich von einer gewichtete Summe von eingehenden Signalen mit einem Schwellenwert modelliert [HH19]. Dieser Zusammenhang wird durch (3.1) formalisiert.

$$y = \sigma\left(\sum_{i=1}^n \mathbf{w}_i \mathbf{x}_i + b\right) \quad (3.1)$$

Die Vergleichsoperation ist die *Aktivierungsfunktion* $\sigma : \mathbb{R} \mapsto \mathbb{R}$, die in diesem Fall die Stufenfunktion ist. Die Eingabe $\mathbf{x} \in \mathbb{R}^n$ wird mit $\mathbf{w} \in [0, 1]^n$ gewichtet und der *Bias* $b \in \mathbb{R}$ wird addiert. Der Bias stellt den Schwellenwert dar.

3 KÜNSTLICHE NEURONALE NETZE

Das künstliche neuronale Netz approximiert eine arbiträre Funktion f^* . Dazu findet es eine Menge von Parametern θ , wodurch $f^*(\mathbf{x}) \approx f(\mathbf{x}, \theta)$ möglichst gut von der Approximationsfunktion f abgebildet wird [BGC17].

Das KNN ist in Schichten organisiert. Analog zu den biologischen Neuronen, gibt es eine *Eingabeschicht* (engl. *Input-Layer*), *Ausgabeschicht* (engl. *Output-Layer*) und *verdeckte Schichten* (engl. *Hidden-Layer*). Dies wird in Abbildung 3.1 illustriert. Analog zur Aktivierung eines einzelnen Neuronen, dargestellt in (3.1), stellt (3.2) die Aktivierung einer Schicht dar [HH19].

$$\mathbf{a}_l = \sigma_l(\mathbf{z}_l), \quad \mathbf{z}_l := \mathbf{W}_l \mathbf{a}_{l-1} + \mathbf{b}_l \quad (3.2)$$



Abbildung 3.1: Beispiel eines FFNN mit vier Schichten und einer binären Ausgabeschicht.

Das allgemeine KNN verfügt über $L \in \mathbb{N}$ Schichten. Jede Schicht l verfügt über $n_l \in \mathbb{N}$ Neuronen. Die Aktivierungsfunktion $\sigma_l : \mathbb{R}^{n_l} \mapsto \mathbb{R}^{n_l}$ berechnet die Aktivierung mit der gewichteten Summe \mathbf{z}_l . Die gewichtete Summe setzt sich zusammen aus der Aktivierung der vorherigen Schicht \mathbf{a}_{l-1} die mit $W_i \in \mathbb{R}^{n_l \times n_{l-1}}$ gewichtet wird. Die Schwellenwerte der Neuronen werden durch die Biase $\mathbf{b}_l \in \mathbb{R}^{n_l}$ dargestellt. In (3.3) wird das allgemeine KNN mit einer rekursiven Funktion modelliert.

$$\mathbf{a}_1 := \mathbf{x}, \quad \mathbf{z}_l := \mathbf{W}_l \mathbf{a}_{l-1} + \mathbf{b}_l, \quad \mathbf{a}_l := \sigma_l(\mathbf{z}_l), \quad \mathbf{f}(\mathbf{x}) := \mathbf{a}_L \quad (3.3)$$

Diese Arbeit nutzt ausschließlich *Feed Forward neuronale Netzwerke*. Diese werden durch *dichte Schichten* (engl. *Dense-Layer*) charakterisiert, d. h. Schichten in denen alle Neuronen einer Schicht mit allen Neuronen der folgenden Schicht verbunden sind [BGC17].

3.1 Keras

Keras ist die am meisten genutzte *deep learning* API und wurde in Python geschrieben [ker21]. Dadurch ist sie kompatibel mit allen gängigen Betriebssystemen. Ihr Fokus ist eine intuitive und simple API anzubieten, sodass schnelle Iterationen im Entwicklungsprozess möglich sind. Trotzdem ist sie effizient und skalierbar, um die Kapazitäten großer Rechenverbunde auszunutzen.

Keras abstrahiert das ML System *Tensorflow*. Tensorflow implementiert ML Algorithmen, die dem Stand der Forschung entsprechen. Der Fokus ist auf effizientes Training der Modelle [ABC⁺16] gerichtet. Dafür nutzt es die Multikernarchitektur von CPUs, GPUs und spezialisierter Hardware, sogenannten TPUs (Tensor Processing Unit), aus. Es wurde als open-source Projekt veröffentlicht und ist weit verbreitet. Keras bietet die in dieser Arbeit benötigten Algorithmen an, weshalb es zum Trainieren von FFNNs verwendet wird.

3.2 Training von KNN

Um die Zielfunktion f^* zu approximieren ist eine Kostenfunktion nötig, die den Abstand von der approximierten Funktion zur approximierenden Funktion angibt [Nie]. Im Optimierungsprozess wird die Kostenfunktion minimiert. Oft wird die Kostenfunktion auch als *Verlustfunktion* (engl. *loss function*), *Fehlerfunktion* (engl. *error function*), oder *Zielfunktion* (engl. *objective function*) bezeichnet [BGC17]. Je nach Publikation können einige dieser Funktionen spezielle Bedeutungen haben, in dieser Arbeit haben sie aber die gleiche Bedeutung.

Das KNN wird für eine endliche Anzahl an Trainingsdurchläufen trainiert [Nie]. Ein Trainingsdurchlauf einer Trainingsmenge wird als Epoche bezeichnet. Als *Backpropagation* wird der Algorithmus bezeichnet, der mit Hilfe des Lernalgorithmus den Fehler der Kostenfunktion rückwärts durch die Schichten des KNN propagiert, wodurch die Parameter θ angepasst werden. Der Lernalgorithmus steuert dabei, wie in dieser Epoche die Parameter mit den Trainingsdaten aktualisiert werden. Zu den Parametern gehört die Struktur des neuronalen Netzwerks, Aktivierungsfunktionen, Gewichte und Biase.

3 KÜNSTLICHE NEURONALE NETZE

Abhängig von der Strategie des Lernalgorithmus müssen die Gradienten rückwärts, rekursiv Schicht für Schicht zurück propagiert werden. Ziel ist es den Fehler unter Berücksichtigung der Parameter zurück zu propagieren. Dafür ist es nötig die Ableitungen von der Kostenfunktion zu den Parametern zu berechnen. Gleichung 3.3 zeigt, dass das Ergebnis einer Schicht die Aktivierung der gewichteten Summe des Ergebnisses der jeweiligen vorherigen Schicht ist. Um die Ableitungen unter Berücksichtigung der Parameter zu berechnen ist die Anwendung der Kettenregel nötig.

Angefangen mit dem Gradienten der Aktivierungen der Ausgabeschicht, wird der Fehler rekursiv zurück propagiert. Dabei wird in jeder Schicht die Korrektur auf die Parameter addiert. Anschließend kann der Vorgang für nächste Epoche wiederholt werden.

3.3 Lernalgorithmen

Die Lernalgorithmen bestimmen die Strategie, mit der die Parameter θ des KNN aktualisiert werden [HH19]. In dieser Arbeit wird ADAM verwendet, da dieser Algorithmus die Vorteile von (S)GD (Stochastic Gradient Descent) mit Momentum und RMSprop vereinigt [KB14, HH19].



■ Abbildung 3.2: Verschiedene Lernalgorithmen auf der Rosenbrock-Funktion [Ros60].

SGD ist eine Approximation von *Gradient Descent* (GD) [BGC17]. GD ist ein iterativer Algorithmus, der den Gradienten in Richtung des Extremum folgt und dementsprechend die Eingabeparameter aktualisiert.

In (3.4) wird der iterative Prozess zur Minimierung im eindimensionalen Fall gezeigt, wobei $\eta > 0$ eine angemessene *Lernrate*, θ der Eingabeparameter und C die Kostenfunktion ist. Ist die Lernrate zu groß könnte keine Verbesserung beobachtet werden, da das Maxima immer übersprungen wird. Ist die Lernrate zu klein könnte die Konvergenz sehr langsam sein. Wenn ein Sattelpunkt erreicht wird ist der Gradient 0. Aus diesem Grund können mit dieser Technik globale Extremum verfehlt werden.

$$\theta_{k+1} := \theta_k - \text{sign}(C'(\theta_k))\eta \quad (3.4)$$

Im mehrdimensionalen Fall (3.5) wird für jede Komponente des Eingabevektors dieser Prozess durchgeführt, sodass für jede Komponente die Richtung des Extremum verfolgt wird. Dies impliziert, dass GD für Eingabevektoren mit hohen Dimensionen sehr aufwendig zu berechnen ist.

$$\theta_{k+1} = \theta_k - \bigtriangledown C(\theta_k)\eta \quad (3.5)$$

Diese Methode konvergiert, wenn alle Komponenten des Gradienten 0 sind. Je größer die Dimension des Eingabevektor ist, desto aufwendiger ist die Berechnung.

SGD ist eine Approximation von GD. Es nutzt eine zufällige Teilmenge des Eingabevektors, den sogenannten *Mini-Batch*. Es wird angenommen, dass der Gradient des Mini-Batches ähnlich zu dem Gradienten des gesamten Eingabevektors ist. Folglich werden mit jedem Mini-Batch die Parameter des KNN aktualisiert, wodurch in jeder Epoche die Parameter mehr als einmal aktualisiert werden. Ziel dieser Approximation ist eine Laufzeitverbesserung.

(S)GD mit Momentum (3.6) versucht zu vermeiden, dass lokale Extremum gefunden werden anstatt globale Extremum, indem Momentum aus vorherigen Gradienten beibehalten wird, um aus lokalen Extremum wieder raus zu finden [HH19].

$$\mathbf{v}_{-1} = \mathbf{0}, \quad \mathbf{v}_k = \mathbf{v}_{k-1}\gamma + \bigtriangledown C(\theta_k), \quad \theta_{k+1} = \theta_k - \mathbf{v}_k\eta \quad (3.6)$$

Zur Berechnung wird ein Hilfsvektor \mathbf{v} verwendet, welcher das Momentum vergangener Gradienten darstellt. In jeder Iteration fließt ein Anteil γ , typischerweise $\gamma = 0.9$, von dem Hilfsvektor in die Berechnung der neuen Eingabeparameter ein. Der Unterschied zu GD (3.5) ist der Anteil vergangener Gradienten.

3 KÜNSTLICHE NEURONALE NETZE

RMSprop ist eine Generalisierung von *Adagrad* [MH17]. Adagrad passt die Lernrate η an, sodass Komponenten des Eingabevektors die überrepräsentiert sind eine geringere Lernrate erhalten und unterrepräsentierte Komponenten eine im Verhältnis größere Lernrate [DHS11]. In (3.7) wird Adagrad skizziert, wobei sich iterativ die Lernrate antiproportional zur kummierten Norm der Gradienten der Kostenfunktion verringert [LF19, KB14]. Dabei wird für ϵ eine kleine Zahl gewählt, um Teilen durch 0 zu vermeiden aber keinen signifikanten Einfluss auf die Berechnung zu haben.

$$\mathbf{g}_k = \nabla C_{j_k}(\boldsymbol{\theta}_k), \quad \mathbf{w}_k = \mathbf{w}_{k-1} + \mathbf{g}_k^2, \quad \boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \mathbf{g}_k \circ \frac{\eta}{\sqrt{\mathbf{w}_k + \epsilon}} \quad (3.7)$$

Das Problem an Adagrad ist, dass die Lernrate zu schnell gegen 0 konvergieren kann, wodurch das Zielextrema nicht erreicht wird [BGC17]. RMSprop [HSS12] (3.8) löst dieses Problem, indem anstatt die Gradienten zu summieren, ein exponentiell gewichteter gleitender Mittelwert verwendet wird [BGC17].

$$\begin{aligned} \mathbf{w}_{-1} &= \mathbf{0}, & \mathbf{g}_k &= \nabla C_{j_k}(\boldsymbol{\theta}_k) \\ \mathbf{w}_k &= \mathbf{w}_{k-1}\gamma + \mathbf{g}_k^2(1-\gamma), & \boldsymbol{\theta}_{k+1} &= \boldsymbol{\theta}_k - \mathbf{g}_k \circ \frac{\eta}{\sqrt{\mathbf{w}_k + \epsilon}} \end{aligned} \quad (3.8)$$

Adam (3.9) vereint RMSprop und (S)GD mit Momentum, wobei $\gamma_1 < \gamma_2 < 1$ [KB14] ist. Adam beinhaltet RMSprop und hat eine Momentum Komponente, indem das Moment der ersten Ordnung mit exponentieller Gewichtung approximiert wird [BGC17]. Zudem korrigiert Adam den Bias das Momente der ersten und zweiten Ordnung der durch die Initialisierung entsteht.

$$\begin{aligned} \mathbf{v}_{-1} &= \mathbf{w}_{-1} = \mathbf{0}, & \mathbf{g}_k &= \nabla C_{j_k}(\boldsymbol{\theta}_k) \\ \mathbf{v}_k &= (\mathbf{v}_{k-1}\gamma_1 + \mathbf{g}_k(1-\gamma_1))/(1-\gamma_1^k) \\ \mathbf{w}_k &= (\mathbf{w}_{k-1}\gamma_2 + \mathbf{g}_k^2(1-\gamma_2))/(1-\gamma_2^k) \\ \boldsymbol{\theta}_{k+1} &= \boldsymbol{\theta}_k - \mathbf{v}_k \circ \frac{\eta}{\sqrt{\mathbf{w}_k + \epsilon}} \end{aligned} \quad (3.9)$$



■ **Abbildung 3.3:** Für diese Arbeit relevante Aktivierungsfunktionen.

3.4 Aktivierungsfunktionen

Die Aktivierungsfunktion entscheidet ob ein Neuron aktiviert wird oder nicht [NIGM18]. Sie können entweder linear oder nicht-linear sein. Es ist aber nötig nicht-lineare Funktionen zu verwenden, damit jede kontinuierliche Funktion approximiert werden kann [ADIP21]. Sie unterscheiden sich in ihren Eigenschaften und Berechnungskosten, was eine besondere Rolle für Mikrocontroller spielt.

In der frühen Geschichte der neuronalen Netzwerke wurde die *Sigmoid*-Funktion (3.10) viel verwendet, da sie asymptotisch begrenzt, kontinuierlich und nicht-linear ist [ADIP21]. Heute wird sie oft in der Ausgabeschicht für binäre Klassifizierungsprobleme eingesetzt [NIGM18]. Allerdings ist sie für tiefe neuronale Netzwerke ungeeignet, da der Gradient zwischen 0 und 0.25 ist und dadurch im Backpropagation-Prozess bereits nach wenigen Schichten gegen 0 geht. Die *SoftMax*-Funktion (3.11) oder normalisierte Exponentialfunktion berechnet für einen Eingabevektor eine Wahrscheinlichkeitsverteilung. Die Einträge dieser Verteilung können für die Wahrscheinlichkeiten der einzelnen Klassen eines multivariat Klassifizierungsproblem interpretiert werden.

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}} \quad (3.10)$$

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \quad (3.11)$$

3 KÜNSTLICHE NEURONALE NETZE

Zur *ReLU*-Familie [ADIP21] gehört die *ReLU*-Funktion [GBB11, KMK14, EUD18, Alc18] und ihre Varianten [MHN13], sowie die *SoftPlus*- [DBB⁺01] und *Swish*-Funktion [RZL17]. *ReLU* (3.12) steht für „*rectified linear function*“ und wird häufig in modernen neuronalen Netzwerken verwendet [ADIP21]. Sie ist nicht differenzierbar bei 0 und die Ableitung für negative Eingaben ist 0. Dies kann zu *sterbenden Neuronen* (*engl. dying neurons*) führen, da der Bias so negativ wird, sodass das Neuron nicht mehr aktiviert wird. Zudem ist der Trainingsprozess verlangsamt, wenn der Gradient konstant 0 ist. Dafür ist die Ableitung der Funktion ansonsten 1, was den Backpropagation-Prozess vereinfacht, da der Gradient neutral zur Aktivierungsfunktion ist.

$$\text{ReLU}(x) = \max(x, 0) \quad (3.12)$$

Varianten sind beispielsweise *leaky ReLU* [MHN13] (3.13) und *ELU* (*exponential linear unit*) (3.14) [CUH15], welche versuchen die Defizite des konstanten 0 Gradienten zu lösen, indem der negative Teil der Funktion nicht 0 ist.

$$\text{leaky_ReLU}_\alpha(x) = \begin{cases} x & , \text{wenn } x \geq 0 \\ \alpha x & , \text{ansonsten } (\alpha \in \mathbb{R}_0^+) \end{cases} \quad (3.13)$$

$$\text{ELU}_\alpha(x) = \begin{cases} x & , \text{wenn } x \geq 0 \\ \alpha(e^x - 1) & , \text{ansonsten } (\alpha \in \mathbb{R}_0^+) \end{cases} \quad (3.14)$$

SoftPlus (3.15) ist analytisch, dafür im Vergleich zu *ReLU* aufwendiger zu berechnen ist [ADIP21].

$$\text{softplus}(x_i) = \ln(e^x + 1) \quad (3.15)$$

Eine weitere Variante ist *Swish* [RZL17] (3.16). Sie ist analytisch aber nicht monoton. Im Vergleich zu *ReLU* ist sie aufwendig zu berechnen. Ihre Autoren behaupten aber, dass dadurch bessere Ergebnisse erzielt werden können, ohne andere Parameter zu ändern.

$$\text{swish}(x_i) = \frac{x e^x}{e^x + 1} \quad (3.16)$$

3.5 Ressourcenbedarf auf dem Mikrocontroller

Der Speicherverbrauch eines neuronalen Netzes ist abhängig von der Anzahl der Gewichte und Biase, sowie der Größe des verwendeten Datentypen [Kub19]. Ein FFNN mit 3 Schichten der Größe n_1, n_2, n_3 hat $n_1n_2 + n_2n_3$ Gewichte und $n_2 + n_3$ Biase. Die Gewichte und Biase sind für gewöhnlich Gleitkommazahlen. Diese können mit 4 und 8 Byte dargestellt werden. Als Alternative können auch Festkommazahlen verwendet werden, die 2 Byte benötigen [Gie20].

Die Ausführungszeit des KNN ist stark abhängig von der Hardware. Das KNN kann sowohl im RAM abgelegt werden oder im Flash-Speicher [Eng18]. Wenn es im Flash-Speicher abgelegt werden muss, müssen die Gewichte bei der Ausführung in den RAM und die Register geladen werden. In Yao's Experimenten hat dies die Ausführungszeit um bis zu 74% verlangsamt [Jia17].

Außerdem wird Multiplikation und Division zur Evaluierung des KNNs benötigt [Eng18]. Wenn die Hardware keine Unterstützung dafür anbietet, müssen diese Operationen durch Software ergänzt werden. Diese sind dann signifikant komplexer zu berechnen im Vergleich zu hardwareunterstützten Operationen.

Die Ausführungszeit ist auch abhängig von der Struktur des Netzwerkes [Gie20]. Je mehr Gewichte ein KNN hat, desto mehr Multiplikationen müssen durchgeführt werden. Zudem wird in jeder Schicht eine Aktivierungsfunktion angewendet, die ebenfalls sehr aufwendige Berechnungen benötigen kann [VKK⁺20].

Es gibt verschiedene Optimierungen, die die Ausführungszeit und den Speicherbedarf verringern. Giese hat in seiner Arbeit festgestellt, dass das Löschen von Gewichten (engl. pruning) oder das Gruppieren von ähnlichen Gewichten (engl. quantization) signifikante Verbesserungen sowohl des Speicherverbrauchs als auch bei der Ausführungszeit zeigt [Gie20]. Denkbar sind auch Compiler-Optimierungen, die sich sowohl auf Ausführungszeit, Speicherbedarf und Energiebedarf auswirken können. Giese stellte fest, dass sowohl der Speicherbedarf verringert werden kann, als auch die Ausführungszeit.

Um den Energieverbrauch zu bestimmen ist ein Energiemodell nötig, das den Energieverbrauch von einzelnen Instruktionen im Zusammenhang mit Speicher und Caches genau beschreibt [RLF18]. Diese sind aber durch die Komplexität moderner Architekturen nicht verfügbar oder ungenau. Allgemein kann jedoch der Energieverbrauch proportional zur Ausführungszeit betrachtet werden [CGSS14].

3 KÜNSTLICHE NEURONALE NETZE

Standortbestimmung

Als Standortbestimmung, oder *Lokalisierung*, wird der Prozess bezeichnet die Position von einem Gerät oder Nutzer in einem Koordinatensystem zu bestimmen [BHE00]. Unterschieden wird dabei zwischen *Indoor-* und *Outdoor-Lokalisierung* [ZGL19, BHE00]. Bei Indoor-Lokalisierung wird ein Szenario innerhalb von Gebäuden betrachtet und bei Outdoor-Lokalisierung ein Szenario unter dem freien Himmel.

Weiterhin wird unterscheiden zwischen *Device-Based-* und *Device-Free-Lokalisierung* [XZYN16]. Bei Device-Based-Lokalisierung bestimmt das Gerät selbst die Position, wohingegen bei der Device-Free-Lokalisierung die Position von einer Infrastruktur bestimmt wird.

4.1 Orientierungssinn von Mensch und Tier

Lokalisierung ist aber nicht nur beschränkt auf Geräte. Menschen und Tiere haben einen Orientierungssinn, der die Navigation anhand von Orientierungspunkten ermöglicht [KSA17, MGC⁺96]. Beispielsweise hängt die Navigation von Honigbienen stark von den Orientierungspunkten ab [MGC⁺96]. Anstatt die direkte Route zu wählen, fliegen Honigbienen die Orientierungspunkte ab, um zu ihrem Ziel zu gelangen. Dabei ist die Navigation robust gegenüber leichten Veränderungen der Orientierungspunkte.

Neben Honigbienen zeigen Vögel, Insekten oder Meereslebewesen verschiedene Arten von Navigation durch Ausnutzen ihrer Sinne [ÅBLM14]. So gibt es Vögel und Insekten, die anhand der Position der Sonne und ihrer inneren Uhr navigieren. Nachtaktive Singvögel hingegen scheinen sich anhand der Sterne zu orientieren. Viele Tiere haben auch einen Sinn, um das Magnetfeld der Erde wahrzunehmen und navigieren basierend auf diesem magnetischen Kompass.

4 STANDORTBESTIMMUNG

4.2 Indoor- und Outdoor-Lokalisierung

GPS ist eine weit verbreites Standortbestimmungssystem im Outdoor-Kontext und kann für eine Vielzahl von Anwendungen eingesetzt werden, z. B. Tracking, Navigation oder Rettungsaktionen [KH05]. Es skaliert zu einer arbiträren Anzahl von Nutzern, da es einen device-based Ansatz verwendet. Die Geräte berechnen aus den empfangenden Signalen von mehreren Satelliten ihre Position. Dabei ist die Standortbestimmung bis zu 5 m genau [SS18], wobei es Varianten gibt, die noch bessere Auflösungen erzielen können [PE96]. Der Energieverbrauch ist sehr hoch [JCC⁺13], wodurch es ungeeignet für kleine batteriegestützte Systeme ist. In einem Indoor-Kontext kann GPS aber meistens nicht eingesetzt werden, da das Gebäude die benötigte Signalstärke zu den Satteliten beeinträchtigen kann [XZYN16, JLP06]. Aus diesem Grund ist es für batteriegestützte Mikrocontroller im Indoor-Bereich suboptimal.

Indoor-Lokalisierung bedarf meist einer hohen Auflösung, muss sicherheitskritische Vorgaben einhalten, energieeffizient sein, skalierbar sein und geringe Kosten haben [XZYN16]. Es gibt verschiedene Ansätze, die entweder device-based oder device-free sind.

Device-based Ansätze nutzen die Sensoren des Geräts, um die Position zu bestimmen [XZYN16]. Beispielsweise können visuelle Features mit der Kamera extrahiert werden [PH19, CPC⁺11], RSS Messungen des WiFi Netzwerkes durchgeführt werden [PZYH08], Bewegungs- oder Lichtsensoren verwendet werden [PH19, WYM18, XZYN16]. Der Vorteil sind die geringen Infrastrukturkosten [XZYN16].

Device-free Ansätze bedürfen einer Infrastruktur, um Objekte im Interessebereich wahrzunehmen [XZYN16]. Diese werden zum Beispiel für Überwachungsszenarien eingesetzt [QWZ⁺18]. Ansätze nutzen beispielsweise die bestehende Kamerainfrastruktur aus [KBP⁺19], WiFi basierte Ansätze [QWZ⁺18], Infrarot basierte Ansätze [KH10] oder RFID basierte Ansätze [YLL⁺15].

4.3 WiFi basierte Indoor-Lokalisierung mit Transfer Lernen

Pan et al. untersuchten Indoor-Lokalisierung basierend auf WiFi RSS Daten [PZYH08]. Der Empfänger misst die *Received Signal Strength* (RSS) mehrerer Sender. Dadurch steht ein Vektor von Signalstärken zur Verfügung aus dem die Position approximiert werden kann.

Pan et al. stellen fest, dass bei ML Ansätzen oft zwei Annahmen getroffen werden. Zum einen wird eine *Offline*-Phase vorausgesetzt mit ausreichend beschrifteten Daten, d. h. eine Trainingsphase bevor das Modell eingesetzt wird. Zum anderen wird angenommen, dass das gelernte Modell statisch über Zeit, Raum und Geräte ist.

In der Praxis können sich Modelle jedoch über die Zeit ändern, z. B. wenn Mitarbeiter Mittags zur Kantine gehen. Es kann schwierig sein ausreichend Daten für große Gebäude zu sammeln. Verschiedene Geräte können verschiedene Sensorwerte erfassen. Dies führt zu einen erheblichen Kalibrierungsaufwand der ML Modelle.

Pan et al. schlagen Transfer Learning vor, um dieses Problem zu lösen. Transfer Learning befasst sich mit dem Problem, wenn Trainings- und Testdaten verschiedener Verteilungen folgen oder in verschiedenen Feature-Räumen repräsentiert sind. Die gewonnene Erfahrung beim Training soll dabei auf das neue, ähnliche Problem übertragen werden. Dafür müssen Beziehungen gefunden werden, die den Erfahrungstransfer ermöglichen.

Sie trainierten ein *Hidden Markov Model* (HMM), wobei die Ortserkennung als Klassifizierungsproblem diskreter Orte modelliert wurde. Auf ihren Testdaten waren sie signifikant besser als Ansätze ohne Transfer Learning.

4.4 Indoor-Lokalisierung mit Magnet- und Lichtsensoren

Wang et al. haben einen Indoor-Lokalisierungsansatz untersucht, der Magnetfelddaten und Lichtsensordaten eines Smartphones nutzt, um die Position des Gerätes zu bestimmen [WYM18]. In einem Vorverarbeitungsschritt kombinieren die Autoren Magnetfelddaten und Lichsensordaten. Damit wird ein Deep LSTM (**L**ong **S**hort **T**erm **M**emory NN) trainiert.

Die Autoren merken an, dass viele Ansätze RSS oder CSI (Channel State Information) nutzen, um Indoor-Lokalisierungsmodelle zu generieren. Diese sind aber unzerverlässig, wenn die Signalstärke schlecht ist oder nicht verfügbar, z. B. in einem Parkhaus. Dahingegen ist das Magnetfeld und Licht omnipresent. Die Magnetfelddaten weisen eine geringe Varianz auf.

4 STANDORTBESTIMMUNG

Diskrete Orte können aber durch Anomalien unterschieden werden, die durch Interferenz von Gebäuden und Geräten verursacht wird. Licht ist ebenfalls meistens vorhanden und weist Unterschiede durch Intensität und Form der Lampe, sowie Schatten und Reflektion auf. Durch die Kombination dieser Daten können eine Vielzahl von diskreten Orten unterschieden werden.

Die Autoren verglichen zwei Szenarien. Das erste Szenario ist ein Labor, welches viele Tische, Stühle und Computer enthält. Das zweite Szenario ist ein langer Korridor. In ihren Ergebnissen ist in beiden Szenarien der Fehler zum größten Teil unterhalb 0,5 m. Der größte Fehler betrug im Labor 3,7 m und im Korridor 6,5 m. Im Vergleich zu einem Modell, dass lediglich die Magnetfelddaten nutzt, erwies sich das Modell, das die Kombination nutzte, als deutlich besser.

4.5 Sensorbasierter Orientierungssinn mit FFNN

Dieser Arbeit ging die Arbeit von Mian voran, der sich zum gleichen Thema mit FFNN auseinander gesetzt hat [Mia21]. Mian nutzte den Simulator CoppeliaSim, um Daten von verschiedenen komplexen Routen zu generieren. Die Routen unterschieden sich dabei in der Anzahl verschiedener Orte und Pfade die für einen Zyklus einer Route verwendet werden können. Die aufgenommenen Daten enthalten Sensorwerte für Beschleunigung, Gyroskop, Licht und Beschriftungen für die Standorte. Dabei werden als Standorte die Teilstücke der Routen bezeichnet aus denen die Route zusammengesetzt ist.

Mian entschied sich die aufgenommen Sensordaten vorzuverarbeiten. Zunächst werden die Sensoren für fünf Stichproben über den Median geglättet. Aus den resultierenden Sensorwerten wird die Veränderung zum vorherigen Sensorwert für jeden Sensor ermittelt. Für jeden Sensor wird als Feature der Betrag dieser Differenz verwendet. Um Muster aus einer Folge von Feature-Mengen zu inferieren hat Mian ein Datenfenster eingeführt, über das hintereinander liegende Feature-Mengen zu einer Feature-Menge konkatinert werden. Zudem werden die zuletzt besuchten Standorte in Form einer exponentiell fallenenden Funktion über die Zeit als weitere Features hinzugefügt.

4.5 SENSORBASIERTER ORIENTIERUNGSSINN MIT FFNN

Mit dieser Eingabe trainierte Mian ein FFNN mit einer Rückwärtskante (FBNN) von der Ausgabe- zur Eingabeschicht, um die zuletzt bestimmten Standorte als Features nutzen zu können. Die Rückwärtskante wurde im Training simuliert, indem die Trainingsdaten in zwei Teilmengen partitioniert wurden. Mit der ersten Teilmenge wurde das FFNN mit korrekt beschrifteten Trainingsdaten trainiert. Das trainierte FFNN wurde dann genutzt, um die Standorte der zweiten Teilmenge zu bestimmen. Daraufhin wurde das FFNN mit der zweiten Teilmenge trainiert, bevor es auf einer Testmenge validiert wurde.

Mian unterscheidet vier Modellarchitekturen: FFNN, FBNN, WFFNN (FFNN mit Datenfenster) und WFBNN (FBNN mit Datenfenster). Er stellte fest, dass das FFNN nicht in der Lage war verschiedene Standorte zu unterscheiden, unabhängig von der Anzahl der verdeckten Schichten und dessen Anzahl von Neuronen.

Das FBNN hingegen konnte bei einer Route mit einem Pfad und sechs Standorten Testgenauigkeiten von bis zu 86,73% erzielen. Allerdings bedarf es dafür zwei verdeckte Schichten mit jeweils 64 Neuronen. Mit weniger Schichten oder Neuronen wurden deutlich schlechtere Ergebnisse erzielt. Bei mehr Pfaden und Standorten wurden ebenfalls schlechtere Ergebnisse erzielt.

Mit der Einführung eines Datenfensters (WFFNN und WFBNN) hat sich die Klassifizierungsgenauigkeit signifikant erhöht. Ein WFFNN mit zwei verdeckten Schichten mit jeweils 16 Neuronen und einem Datenfenster von 25 konnte eine Testgenauigkeit von 93,31% bei einem Pfad und sechs Standorten erreichen. Durch das Vergrößern des Datenfensters und mehr Neuronen pro verdeckte Schicht konnte Mian die Klassifizierungsgenauigkeit auf bis zu 95,57% erhöhen. Das WFBNN hat keine signifikante Änderung zum WFFNN gezeigt.

Mian stellte fest, dass sich die Klassifizierungsgenauigkeiten weiter verbessern ließen, wenn er eine Lichtquelle an Standorten setzt, wo sich die neuronalen Netze unsicher sind. Allerdings würden Fehler durch den Sensor oder Veränderungen der Lichtverhältnisse einen größeren Einfluss auf die Klassifizierungsgenauigkeit des Modells haben.

Mian konkludierte, dass ein Kompromiss zwischen Klassifizierungsgenauigkeit und Modellgröße geschlossen werden müsste, da die Modellgröße und Klassifizierungsgenauigkeit proportional mit der Anzahl der verdeckten Schichten und Neuronen, sowie der Datenfenstergröße zusammenhingen.

4 STANDORTBESTIMMUNG

Machine Learning Modelle

In dieser Arbeit wird die Device-Based Indoor-Lokalisation auf Basis von Sensorwerten untersucht. Der Ansatz ist inspiriert von dem Orientierungssinn von Mensch und Tier. Dabei werden diskrete Standorte unterschieden, sowie ob eine Anomalie entdeckt wurde, d. h. ob das Modell sich an einem unbekannten Standort oder auf einem unbekannten Pfad befindet.

Abbildung 5.1 zeigt die Architektur des verfolgten Ansatzes. Zunächst werden aus den Sensorwerten Features extrahiert. Die resultierende Feature-Menge wird dann vom ML-Modell genutzt, um den Standort zu klassifizieren. Zuletzt wird auf Basis historischer Daten und dem Klassifizierungsergebnis von einem weiteren ML-Modell zur Anomalieerkennung bestimmt, ob eine Anomalie vorliegt.

Mian verwendete eine ähnliche Architektur [Mia21]. Sein FFNN zur Standorterkennung hatte ebenfalls eine Rückwärtskante, um die zuletzt erkannten Standorte als Features zu verwenden. In dieser Arbeit wird zusätzlich versucht Anomalien zu erkennen. Dafür wird ein weiteres ML-Modell verwendet, welches aus dem Klassifizierungsverhalten des ML-Modells zur Standorterkennung schließt, ob eine Anomalie vorliegt. Dies motiviert die Erweiterung, um eine weitere Phase zur Feature-Extrahierung aus den zuletzt erkannten Standorten, sowie einer Phase zur Evaluierung des ML-Modells zur Anomalieerkennung aus der resultierenden Feature-Menge. Die Anomalieerkennung ist somit eine Information, die aus dem Standorterkennungsverhalten des vorangestellten ML-Modells resultiert, weshalb sowohl der Standort als auch die Einschätzung als Anomalie als Ergebnis zurückgegeben wird.



Abbildung 5.1: Architektur des verfolgten Ansatzes.

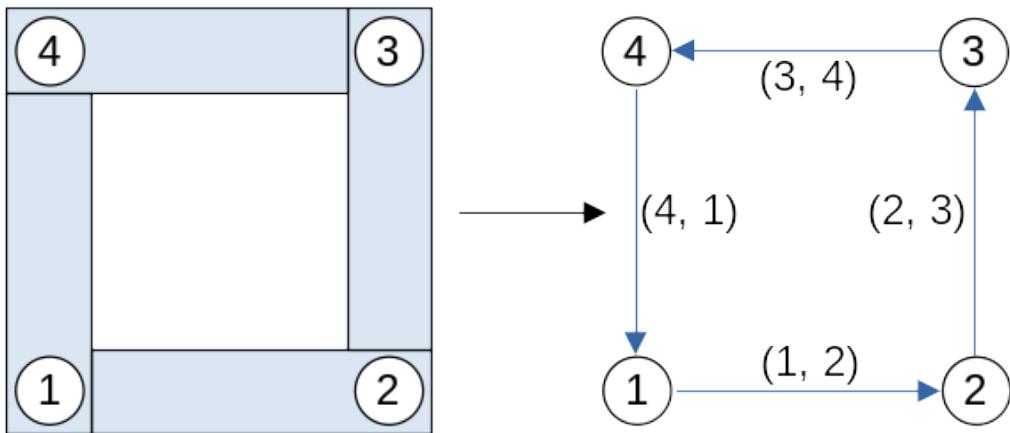
5 MACHINE LEARNING MODELLE

In dieser Arbeit werden Entscheidungsbaum basierte Klassifizierer mit KNN verglichen, insbesondere den von Mian verwendeten Ansatz mit FFNN. Entscheidungsbäume sind deutlich effizienter in der Ausführungszeit als KNN [Dym21], allerdings sind sie in ihrer Generalisierungsfähigkeit durch die berechneten Features begrenzt, wohingegen KNN komplexe Features selbst erlernen können [SLCY11].

5.1 Standortenkodierung

Als Standort wird ein einzigartiger diskreter Ort im Indoor-Szenario bezeichnet. Bei der Klassifizierung können Standorte auf verschiedene Arten im ML-Modell kodiert werden. Mian enkodierte die Pfade zwischen Punkten, die von Interesse sind, als Standorte, d. h. das Klassifizierungsergebnis ist der Pfad auf dem sich der Mikrocontroller befindet [Mia21].

In dieser Arbeit wird neben der Erkennung von Pfaden auch die Erkennung von Knoten, sowie die Erkennung von Pfaden und Knoten untersucht. Pfade und Knoten liegen auf Routen, die als zyklischen Graphen betrachtet werden können, wie in Abbildung 5.2 illustriert. Der Ansatz von Mian versucht die Kanten dieses Graphen zu klassifizieren und definiert diese als Standorte. Daneben können auch nur die Knoten als Standorte enkodiert werden und alle restlichen Datensätze als *unbekannten Standort*. Zuletzt können sowohl die Knoten und Kanten als Standorte definiert werden. Die Knoten werden durch alle Datensätze dargestellt, die innerhalb eines Umkreises von einem Punkt sind, der von Interesse ist. Folglich befinden sich alle restlichen Datensätze auf Kanten zwischen zwei Knoten oder gelten als *unbekannt*.



■ Abbildung 5.2: Standortenkodierung der Knoten und Pfade.

Daraus wird die Komplexität und Genauigkeit dieser Ansätze deutlich. Der Kantenansatz ist ein Kompromiss zwischen Genauigkeit und Komplexität. Dabei bestimmt die Anzahl der zu klassifizierenden Standorte die Komplexität. Zum einen ist gerade bei langen Pfaden eine geringe Auflösung im Vergleich zur Realposition des Objektes zu erwarten, d. h. es ist unklar, ob sich das Objekt am Anfang, Ende oder in dazwischen befindet. Zum anderen werden in einem zyklischen Graphen mindestens so viele Standorte, wie beim Knotenansatz verwendet. Der Knotenansatz benötigt am wenigsten Standorte zur Enkodierung ist aber außerhalb der Standorte sehr ungenau. Aus einer Historie von vorherigen Standorten kann aber ein möglicher Pfad inferiert werden, allerdings können auch mehrere Pfade in Frage kommen, z. B. bei einer Gabelung. Der kombinierte Ansatz encodiert soviele Standorte wie beide Ansätze zusammen, wodurch dieser Ansatz am komplexesten ist und am schlechtesten für große Routen skaliert. Dafür ist die Auflösung des kombinierten Ansatzes so gut, wie eine diskrete Enkodierung es zulässt.

Ist in den aufgenommenen Trainingsdaten die Position des Objektes zum Zeitpunkt der Aufnahme der Sensorwerte bekannt, so können die Standorte nach dem gewählten Enkodierungsansatz beliebig genau beschriftet werden. Dies kann in einem Weiterverarbeitungsschritt nach der Aufnahme der Daten mit einer Karte von den Interessepunkten geschehen.

5.2 Entscheidungswald

Entscheidungsbaum basierte Klassifizierer sind sehr effizient und können trotzdem hohe Klassifizierungsgenauigkeiten bei hohen Fehlertoleranzen erreichen [Dym21]. Entscheidungswälder erhöhen die Klassifizierungsgenauigkeit während die Varianz reduziert wird, dafür wird aber der Speicherbedarf mit jedem Baum linear erhöht. Der Klassifizierer soll zukünftig auf einem Mikrocontroller ausgeführt werden, d. h. die Größe des Entscheidungswaldes ist durch den Programmspeicher des Mikrocontrollers limitiert. Zu dem Zeitpunkt, wo diese Arbeit verfasst wird, sind die Limitierungen des Mikrocontrollers noch nicht bekannt. Für gewöhnlich ist der Programmspeicher aber auf wenige Kilo-Byte beschränkt [Dym21].

Als Ensemble-Methode wird *RandomForest* benutzt, da dieser Entscheidungsbäume auf Basis von zufälligen Teilmengen der Feature-Menge konstruiert. Dadurch ist eine erhöhte Toleranz gegenüber Fehlern, wie fehlerhafte Sensorwerte oder anderen Features zu erwarten.

5 MACHINE LEARNING MODELLE

Um den Einfluss von verschiedenen Wald- und Baumgrößen auf die Fehlertoleranz und Klassifizierungsgenauigkeit hin zu untersuchen, werden verschiedene Größen trainiert und auf den Testmengen evaluiert. Trotzdem wurden Dimensionen in einem Bereich gewählt, der für einen Mikrocontroller realistisch ist. Es wurden jeweils Bäume und Wälder der Größe 8, 16, 32, und 64 untersucht.

5.3 Feed Forward neuronales Netzwerk

Für das KNN wird ein Feed Forward neuronales Netzwerk verwendet. Das FFNN besteht aus drei bis sechs Schichten. Alle Schichten, außer der letzten, verwenden ReLU als Aktivierungsfunktion. Die letzte Schicht verwendet SoftMax.

Die Größe der Eingabeschicht ist abhängig von der Anzahl der verwendeten Features. Die Features werden in einem Vorverarbeitungsschritt im Gegensatz zum Entscheidungswald normalisiert. Die Größe der Ausgabeschicht ist die Anzahl der verschiedenen diskreten Standorte, die unterschieden werden. Je nach Konfiguration gibt es eins, zwei oder vier verdeckte Schichten, die jeweils 16, 32, 64 oder 128 Neuronen haben.

Die Standorte sind kategorische Daten, d. h. ihr Wert haben keine Aussage über die Beziehung der Standorte zueinander. Aus diesem Grund werden sie kategorisch encodiert, d. h. aus einem Wert i aus N möglichen Werten wird eine Liste der Größe N generiert, die überall 0 ist außer an der Stelle i , die 1 ist. Folglich wird als Kostenfunktion *kategoriale Crossentropy* verwendet. Als Lernalgorithmus wird Adam verwendet mit einer Batch-Größe von 50. Trainiert wird über 75 Epochen.

5.4 Training der ML-Modelle

Typischerweise haben weder Entscheidungsbaum basierte Klassifizierer noch FFNN Rückwärtskanten. Neuronale Netze mit Rückwärtskanten werden als *rekurrente Netze* (RNN) bezeichnet. Abbildung 5.1 zeigt, dass die Rückwärtskante genutzt wird, um das Klassifizierungsergebnis, also den letzten Standort, bei der Feature-Extrahierung zu nutzen. Das Klassifizierungsergebnis ist aber nicht immer korrekt, wodurch fehlerhafte Features im Zusammenhang mit dem Klassifizierungsergebnis als Eingabe in das ML-Modell verwendet werden können. Damit das ML-Modell lernt mit diesem Fehler umzugehen, ist es notwendig, dass das ML-Modell Trainingsbeispiele mit Features auf Basis eigener Klassifizierungsbeispiele zur Verfügung hat.

Abbildung 5.3 illustriert den Trainingsablauf. Die simulierten Daten der aufgenommenen Routen sind unterteilt in Partitionen basierend auf deren Zyklusbeschriftung, damit in jeder Partition alle Standorte vorhanden sind. Der Zyklus ist ein Umlauf einer Route, bevor sie wiederholt wird. Insgesamt besteht die Datenmenge aus 20 Zyklen. Die ersten fünf Zyklen werden zum „Aufwärmen“ verwendet, d. h. das ML-Modell wird mit korrekt beschrifteten Daten trainiert, welche es nicht selbst beschriftet hat. In den folgenden zehn Zyklen werden weitere Partitionen zur Trainingsmenge hinzugefügt, die mit quadratisch steigendem Anteil von dem ML-Modell selbst beschriftet sind. Zunächst werden 50% der Partition i vom ML-Modell beschriftet, bis beim 13. Zyklus schließlich 100% beschriftet wird. Die Elemente aus der Partition, die beschriftet werden sollen, ist zufällig, damit der Klassifizierungsfehler auf allen Teilstücken der Route gelernt werden kann.



■ Abbildung 5.3: Der Trainingsablauf des verfolgten Ansatzes.

5 MACHINE LEARNING MODELLE

Die erste Trainingsphase ist abgeschlossen, nachdem das ML-Modell mit einer Trainingsmenge von 15 Zyklen trainiert wurde. Anschließend wird einmalig eine Feature-Auswahl betrieben, in der insignifikante Features aus der Feature-Menge entfernt werden. Insignifikante Features sind Feature, die eine geringe Permutationswichtigkeit aufweisen, d. h. Feature die einen sehr geringen Klassifizierungsfehler erzeugen, wenn ihre Einträge in der Validationsmenge permutiert werden [Bre01]. Dies ermöglicht kleinere ML-Modelle zu verwenden und verringert die Dimensionen des Suchraumes, wodurch das Trainieren erleichtert wird. Außerdem müssen Modelle individuell für verschiedene Einsatzgebiete trainiert werden, bei denen möglicherweise einige Sensoren bzw. Features nicht genutzt werden. In Abbildung 5.3 ist die Feature-Auswahl nur einmalig vorgesehen. Denkbar wäre aber auch eine iterative Eliminierung der Features oder Optimierung durch ein evolutionären Algorithmus. Anschließend wird das ML-Modell erneut auf den Partitionen trainiert, bis es validiert und evaluiert werden kann.

5.5 Anomalieerkennung

Als Anomalieerkennung wird in dieser Arbeit das Problem bezeichnet, zu erkennen, dass die Sensorenbox sich an einem unbekannten Standort oder auf einem unbekannten Pfad befindet. Abbildung 5.1 zeigt, dass die Anomalieerkennung ein eigener Schritt bei der Evaluierung der Sensordaten ist. Die Eingabe sind Features, die auf historischen Daten und dem momentanen Standort basieren.

Es wird ein zweites ML-Modell trainiert, anstatt dem ML-Modell zur Standorterkennung einen *Anomaliestandort* lernen zu lassen. Dies ist begründet auf der Schwierigkeit Trainingsdaten für Anomalien basierend auf den Sensordaten zu entwickeln, da es unendlich viele Szenarien geben könnte, die als Anomalie zu bezeichnen sind. Stattdessen werden Features genutzt, die eine Abweichung von der Normalität ausdrücken, d. h. das ML-Modell lernt nicht explizite Anomaliepfade, sondern das Verhalten des Standortklassifizierungsmodells einzuordnen.

Dafür werden analog zu Kapitel 5.2 und 5.3 Entscheidungswälder und FFNN trainiert. Allerdings bedarf dieses Modell keine Rückwärtskante und das FFNN kann für binäre Klassifizierung vereinfacht werden. Die letzte Schicht des FFNN hat nur ein Neuron und nutzt die Sigmoid-Funktion, anstatt der SoftMax-Funktion. Außerdem wird für die Kostenfunktion *binäre Crossentropy* verwendet, anstatt kategorische Crossentropy. Binäre Crossentropy bedarf keine kategorische Enkodierung im Gegensatz zur kategorischen Crossentropy.

Die ML-Modelle zur Anomalieerkennung können separat von den ML-Modellen zur Standorterkennung trainiert werden. Sie werden im Gegensatz zu den ML-Modellen zur Standorterkennung nur einmalig trainiert mit vorbereiteten Trainingsdaten. Die Trainingsdaten werden aus den Anomaliedatenmengen und Datenmengen für die verschiedenen Routen generiert. Dafür werden die Klassifizierungsergebnisse auf diesen Datenmengen von den zuvor trainierten ML-Modellen zur Standorterkennung genutzt. Daraus werden beschriftete Features extrahiert, die in Kapitel 6.6 detailliert erläutert werden.

5 MACHINE LEARNING MODELLE

Alternativ können nicht ML-Modelle als Vergleichsmodelle genutzt werden. Die trivialen Modelle sind das *Coin-Toss*-, *immer Falsch*- und *immer Wahr*-Modell. Ein komplexeres Modell nutzt die Topologie des Systems aus, um eine Anomalie zu indizieren, wenn die Sensorenbox nicht dem Pfad folgt. In einer lockeren Variante könnte nur erwartet werden, dass einem Pfad kontinuierlich gefolgt wird, sodass übersprungene Standorte nicht dauerhaft eine Anomalie indizieren. Abbildung 5.1 skizziert dieses Modell. Wenn der Standort und der vorherige Standort nicht der unbekannte Standort ist und der vorherige Standort ungleich dem erwarteten Standort ist, dann wird ein Alarm ausgelöst. Dieser Alarm indiziert für eine konstante Dauer, dass eine Anomalie vorliegt. Dies ist motiviert aus dem Zusammenhang, dass wenn eine Anomalie vorliegt, es wahrscheinlich ist, dass sie kurz danach immernoch vorliegt.

```
if standort > 0 and vorheriger_standort > 0 and ↴
    topologie[standort].vorheriger_standort != vorheriger_standort:
        alarm_zahler = 0

if alarm_dauer > alarm_zahler:
    alarm_zahler++
    return true
return false
```

■ **Listing 5.1:** Skizze des Modells zur Anomalieerkennung auf Basis der Topologie.

Trainings- und Testdaten

Zu dem Zeitpunkt, zu dem diese Arbeit verfasst wurde, existierte noch keine Möglichkeit Echtdaten in einem realistischen Szenario mit einem Mikrocontroller aufzunehmen, der über alle nötigen Sensoren verfügt. Aus diesem Grund ist es nötig Daten simulativ zu erfassen.

Zur Datenerfassung wird der allzweck Robotersimulator CoppeliaSim verwendet [RSF13]. Mit diesem Simulator werden verschiedene Fabrikszenarien simuliert und dabei verschiedene Sensorwerte erfasst. Diese werden dann in einem Vorverarbeitungsschritt gefiltert und mit Sensordaten ergänzt, die in CoppeliaSim nicht verfügbar sind. Zuletzt werden Features extrahiert und die resultierende Datenmenge in Trainings-, Validations- und Testdaten unterteilt.

6.1 Sensordaten aus Simulator CoppeliaSim

Insgesamt wurden vier Routen über 20 Zyklen jeweils zwei mal erfasst. Einmal für Trainingsdaten und einmal für Testdaten, wobei die letzten fünf Zyklen der Trainingsdaten als Validationsmenge genutzt werden, die außerdem zur Feature-Auswahl genutzt wird. Dabei wurden alle 50 ms die xyz-, Koordinaten, Accelerometerdaten und Gyroskopdaten erfasst, sowie Lichtintensität und Metadaten. Zu den Metadaten gehören Zeitstempel, Beschriftung des Routenabschnitts und Beschriftung des derzeitigen Zyklusses. Ein Zyklus ist der vollständige Umlauf einer Route.

6 TRAININGS- UND TESTDATEN

Abbildung 6.1 zeigt eine der vier Routen „simple_square“. Jede Route ist mit Markierungen für Zyklen und Standorte ausgestattet. Die Zyklusmarkierung wird genutzt, um die Datensätze mit dem derzeitigen Zyklus zu beschriften. Jedes mal, wenn die Sensorenbox diese Markierung überschreitet, wird der Zähler für den Zyklus inkrementiert. Die Standortmarkierung wird genutzt, um die Datensätze mit dem derzeitigen Routenabschnitt zu beschriften. Jedes mal, wenn die Sensorenbox diese Markierung überschreitet, wird der derzeitige Wert für den Routenabschnitt auf den Wert der Markierung gesetzt. Dabei werden alle aufgenommenen Datensätze immer mit dem derzeitigen Wert für den Routenabschnitt markiert.



■ Abbildung 6.1: Modell der Route „simple_square“ in CoppeliaSim mit Beschriftungen.

6.2 MIT EIGNENEN MODELLEN ERGÄNZTE SENSORDATEN

Neben „simple_square“ gibt es noch drei weitere Routen (siehe Abbildungen B.1, B.3 und B.5). Die Route „long_rectangle“ weist lange Pfade mit wenig Änderungen auf. Die Route „rectangle_with_ramp“ besitzt zusätzlich zwei Rampen, wodurch Höhenunterschiede simuliert werden. Die Route „many_corners“ ist sehr komplex und hat viele verschiedene Standorte. Die Förderbänder können verschiedene Geschwindigkeiten haben mit sowohl abrupten Übergangen, als auch fließenden Übergängen zueinander.

Je nach Enkodierungsart (siehe Kapitel 5.1) müssen die Knoten und Kanten, dieses zyklischen Graphen, als Standorte enkodiert werden. Als Knoten wird die Menge der Datensätze bezeichnet, die sich in einem Umkreis des ersten Datensatzes befinden, der mit einem Standort in der Simulation markiert wurde. Dadurch gibt es Datensätze, die zwar in der Simulation mit diesem Standort markiert wurden, aber nicht innerhalb des Umkreises von dem ersten Datensatz liegen und somit zu diesen Knoten gehören. Die Knoten werden mit dem Wert der Standortbeschriftung beschriftet. Die übrigen Datensätze werden entweder als unbekannten Standort beschriftet oder erhalten einen Standortwert, der die Beziehung der Kante zwischen zwei Knoten enkodiert, aber nicht in der Simulation markiert wurde.

6.2 Mit eignen Modelle ergänzte Sensordaten

Im Projekt ist die Sensorenbox potentiell mit einem Accelerometer, Gyroskop, Lichtsensor, Magnetfeldsensor, Temperatursensor und Geräuschsensor ausgestattet, sowie einen Empfänger zur Erfassung von WLAN-Zugangspunkten. Mit Coppeliasim ist es allerdings nur möglich Sensorwerte des Accelerometers, Gyroskops und Lichtsensors zu erfassen. Aus diesem Grund werden die fehlenden Sensoren durch das Ausnutzen vereinfachter Modelle ergänzt.

Abbildung 6.2 zeigt die Route „simple_square“ mit eingezeichneten Objekten, die Einfluss auf die modellierten Sensoren nehmen können. Die Routen befinden sich im gleichen Koordinatensystem, weswegen alle Routen diese Verteilung teilen. Abbildungen B.2, B.4 und B.6 zeigen eine analoge Karte für die restlichen Routen.

6.2.1 Magnetfeldsensor

Magnetfeldsensoren messen das Magnetfeld auf Basis von Effekten die dadurch induziert werden, z. B. die Lorentzkraft oder der Hall-Effekt [TH09]. Eine Anwendung dieses Sensors ist der Kompass. Dieser richtet sich nach dem Magnetfeld der Erde aus und wird daher traditionell zur Navigation verwendet.

6 TRAININGS- UND TESTDATEN

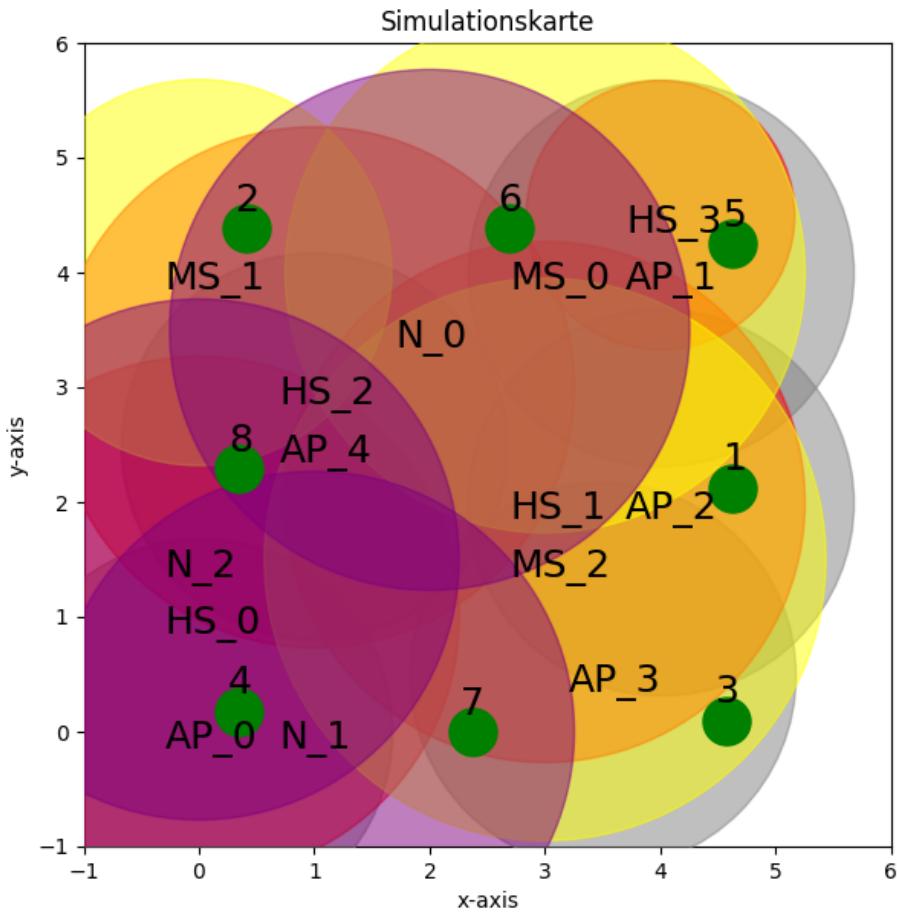


Abbildung 6.2: Karte der Route „simple_square“ mit eingezeichneten Einflussbereichen der Objekte, die Einfluss auf modellierte Sensoren haben. *Magnetic Source* (Gelb), *Noise Source* (Lila), *Access Point* (Grau), *Heat Source* (Rot) und Standorte (Grün).

Das vereinfachte Modell des simulierten Magnetfeldsensors fungiert wie ein Kompass. Die Ausgabe ist der relative Richtungsunterschied von der Sensorenbox zum magnetischen Nordpol. Dabei können starke magnetische Objekte in der Umgebung Einfluss auf den Sensor haben, sodass sich der magnetische Nordpol für den simulierten Sensor ändern kann. Die Ausgabewerte sind zwischen 0 und 359, d. h. 0 ist Norden, 90 Osten, 180 Süden und 270 Westen.

Es wird angenommen, dass der magnetische Nordpol der Erde weit genug weg ist, sodass sich im Fabrikszenario die Richtung nur ändern kann, wenn die Ausrichtung der Sensorenbox geändert wird oder, wenn ein magnetisches Objekt in der Umgebung Einfluss ausübt. Außerdem wird angenommen, dass sich die Ausrichtung des Objektes in einem Zyklus nicht ändert, da Fließbandsysteme für gewöhnlich nicht rund sind, sondern Kante auf Kante aufeinander

6.2 MIT EIGNENEN MODELLEN ERGÄNZTE SENSORDATEN

übergehen. Allerdings wird für jeden Zyklus eine neue zufällige Ausrichtung zwischen 0 und 359 gewählt, da das Objekt mit verschiedenen Ausrichtungen auf das Fließband gelegt werden könnte. Zudem wird keine Interferenz der magnetischen Objekte zueinander angenommen. Sollten sie sich überschneiden wird das magnetische Objekt mit dem meisten Einfluss gewählt.

Starke magnetische Objekte sind strategisch in der Umgebung des Fließbandsystems plaziert. Ihre Stärke wird dabei durch die Einflussreichweite definiert, wobei der Einfluss quadratisch mit der Distanz abnimmt. Ist der Einfluss bei 100%, so wird für den magnetischen Nordpol die Position des magnetischen Objekts angenommen. Wenn der Einfluss geringer ist, dann wird ein magnetischer Nordpol zwischen dem magnetischen Nordpol der Erde und des magnetischen Objektes Anteilweise gewählt. Die magnetischen Objekte können unterschiedliche Einflussreichweiten haben.

Abbildung 6.3 illustriert diese Situation. Um die Ausrichtung der Sensorenbox relativ zum magnetischen Nordpol der Erde bei 0 und dem Einfluss des magnetischen Objekts zu berechnen, ist es nötig die Winkel zwischen dem magnetischen Nordpol der Erde und dem magnetischen Objekt, sowie den Winkel zur Sensorenbox zu wissen. Der Winkel β vom magnetischen Nordpol der Erde zur Sensorenbox p_o ist bekannt. Der Winkel γ vom magnetischen Nordpol der Erde zum magnetischen Objekt p_m wird aus dem Winkel α innerhalb des eigenen Quadranten und der Anzahl der Quadranten, zum Quadranten in dem sich p_m befindet, berechnet. Dabei werden die Quadranten im Uhrzeigersinn gezählt. Je nach Quadrant können für a und b die x - bzw. y -Koordinate von p_m gewählt werden. Da der Einfluss des magnetischen Objekts p_m abhängig von der Position der Sensorenbox p_o ist, müssen a und b mit p_o als Ursprung transformiert werden. Die Berechnung von α folgt dann Formel 6.1.

$$\alpha = \arcsin\left(\frac{|p_{o_a} - p_{m_a}|}{\sqrt{(p_{o_a} - p_{m_a})^2 + (p_{o_b} - p_{m_b})^2}}\right) \quad (6.1)$$

Mit den Winkeln γ und β kann dann das Koordinatensystem zum magnetischen Objekt hin rotiert werden, sodass die Ausrichtung von p_o zum beeinflussten magnetischen Nordpol aus Formel 6.2 berechnet werden kann.

$$\gamma' = (\gamma + (360 - \beta)) \mod 360 \quad (6.2)$$

6 TRAININGS- UND TESTDATEN



Abbildung 6.3: Ausrichtung der Sensorenbox relativ zum magnetischen Objekt und magnetischen Nordpol der Erde.

Zu beachten ist schließlich der Einfluss η , der abhängig von der Distanz von p_m zu p_o ist. Formel 6.3 modelliert den Einfluss als quadratisch abfallende Funktion mit zunehmender Distanz d und maximaler Einflussdistanz d_{\max} , wobei der maximale Einfluss 1 ist und der minimale Einfluss 0 ist.

$$\eta(d) = \min(0, 1 - \frac{d^2}{d_{\max}^2}) \quad (6.3)$$

Der Einfluss wirkt sich proportional auf den Winkel von p_m zum magnetischen Nordpol der Erde aus. Dabei ist aber die Ausrichtung von p_m zum magnetischen Nordpol der Erde zu beachten, denn bei den Polen des Magnetfeldes ändert sich die Richtung des Magnetfeldes. Formel 6.4 zeigt die Berechnung von γ' abhängig von der Ausrichtung von p_m zum magnetischen Nordpol der Erde, wobei $d = \sqrt{(p_{o_a} - p_{m_a})^2 + (p_{o_b} - p_{m_b})^2}$.

$$\begin{aligned} \gamma'_L &= (\gamma + (360(1 + \eta(d)) - \beta(1 - \eta(d)))) \mod 360 \\ \gamma'_R &= (\gamma + (360 - \beta(1 - \eta(d)))) \mod 360 \end{aligned} \quad (6.4)$$

6.2.2 Temperatursensor

Das vereinfachte Modell für diesen Sensor geht von einer konstanten Umgebungstemperatur aus. Im Raum sind Wärmequellen strategisch verteilt, die eine Temperatur unterhalb und oberhalb der Umgebungstemperatur haben können. Je näher sich die Sensorenbox an einer der Wärmequellen befindet, desto mehr nähert sich die gemessene Temperatur der Temperatur der Wärmequelle T_{\max} an.

Jede Wärmequelle hat einen Einflussbereich, in der sie sich auf die Umgebungstemperatur T_U auswirken kann. Sollten sich zwei Wärmequellen überschneiden wird die Temperatur ausgewählt, die den größten Unterschied zur Umgebungstemperatur verursacht. Formel 6.5 zeigt die Berechnung der resultierenden Temperatur T' abhängig von der Distanz d der Wärmequelle zur Sensorenbox und der maximalen Einflussdistanz d_{\max} .

$$T' = \begin{cases} T_U & , \text{ wenn } d > d_{\max} \\ \frac{(T_{\max} - T_U)d^2}{d_{\max}} + T_U & , \text{ ansonsten} \end{cases} \quad (6.5)$$

6.2.3 Geräuschsensor

Es gibt verschiedene Arten von Geräuschsensoren, z. B. Geräuschrichtungssensoren [TDS⁺14] oder spezifische Anwendungen, wie Herzgeräuschsensoren [ZLGZ16]. Das Modell ist einem Akustiksensor [Ses91], oder einem Mikrophone am ähnlichsten, dass die Lautstärke in einem Frequenzbereich misst.

In diesem Modell werden Frequenzbereiche vernachlässigt. Es gibt ein Hintergrundrauschen V_H , auf das für jeden Datensatz ein zufälliges Rauschen addiert wird. Daneben gibt es Lautstärkequellen, die entweder periodisch oder konstant sind und strategisch im Raum verteilt sind. Die Interferenz von verschiedenen Lautstärkequellen wird vernachlässigt. Sollten sich zwei Lautstärkequellen überschneiden, so wird die maximale Lautstärke ausgewählt.

Die Lautstärke V_{\max} die von einer Quelle ausgeht nimmt quadratisch mit der Distanz d ab. Dabei wird der Einflussbereich von der maximalen Einflussdistanz d_{\max} bestimmt. Formel 6.6 zeigt die Berechnung für die resultierende Lautstärke durch eine konstante Lautstärkequelle.

$$V' = \begin{cases} V_H & , \text{ wenn } d > d_{\max} \\ \frac{(V_{\max} - V_H)d^2}{d_{\max}} + V_H & , \text{ ansonsten} \end{cases} \quad (6.6)$$

6 TRAININGS- UND TESTDATEN

Periodische Lautstärkequellen emittieren ein Geräusch in regelmäßigen Abständen. Dies wird modelliert durch eine quadratisch verringerte Lautstärke über 500 ms, wenn die Lautstärke gemessen wird nachdem das Geräusch stattgefunden hat, d. h. nach 500 ms, ist das Geräusch nicht mehr wahrzunehmen. Formel 6.7 beschreibt diesen Zusammenhang, wobei t der momentane Zeitpunkt ist und t_n das Intervall des periodischen Geräusches ist.

$$V' = \begin{cases} V_H & , \text{ wenn } d > d_{\max} \vee t \bmod t_n \leq 0,5 \\ \max(V_H, (\frac{(V_{\max}-V_H)d^2}{d_{\max}} + V_H)(1 - 4(t \bmod t_n)^2)) & , \text{ ansonsten} \end{cases} \quad (6.7)$$

6.2.4 WLAN-Zugangspunkte

Die Detektierung von WLAN-Zugangspunkten ist eine Konsequenz aus der Detektierung von RSSI-Werten bzw. MAC-Adressen von WLAN-Zugangspunkten. Das Modell vereinfacht dies, indem es lediglich aussagt, ob ein WLAN-Zugangspunkt in Reichweite ist oder nicht.

Im Raum sind strategisch WLAN-Zugangspunkte verteilt. Diese können innerhalb einer maximalen Reichweite empfangen werden. Ist die Distanz der Sensorenbox zum WLAN-Zugangspunkt innerhalb der Reichweite, gilt der WLAN-Zugangspunkt als empfangen. Interferenzen und Reflektionen werden vernachlässigt.

6.3 Simulation von Interrupts

Die Simulationsdaten, die mit CoppeliaSim aufgenommen wurden, enthalten alle 50 ms Einträge für die aufgenommenen Sensoren. Unter realen Bedingungen wäre solch eine Abtastrate aber nicht mit den Limitierungen der Batterielaufzeit zu vereinbaren. Aus diesem Grund führen in solchen Systemen Sensoren *Interrupts* aus, wenn eine signifikante Änderung festgestellt oder ein Schwellenwert überschritten wurde. Interrupts sind Benachrichtigungen an die CPU, dass der Sensor ausgelesen werden, wodurch die CPU in der Zwischenzeit schlafen und somit Energie preservieren kann.

Um dieses Verhalten nachzustellen werden diese Interrupts simuliert, wodurch die Datenmenge gefiltert wird. In dieser Arbeit wird die Änderung zu dem letzten Interrupt eines Sensors modelliert, d. h. jeder Sensor merkt sich seinen Sensorwert, wenn es einen Interrupt auslöst und führt das nächste mal nur einen Interrupt aus, wenn sich der Sensorwert um einen bestimmten Prozentsatz zu dem gemerkten Sensorwert unterscheidet.

6.4 FEATURE-EXTRAHIERUNG

Dadurch wird einerseits ein realistischeres Szenario dargestellt, denn von einer Sensorenbox die still steht würde auch keine Aktivität erwartet werden. Andererseits verringert sich die Datenmenge, wodurch die Trainingszeit verringert wird.

Unterschieden werden drei Ansätze, um Interrupts zu erzeugen. Der erste Ansatz vergleicht den Betrag der Differenz mit einem festen Schwellenwert. Der zweite Ansatz erfordert, dass die Änderung einen Prozentsatz des vorherigen Wertes ausmacht. Der dritte Ansatz erfordert nur eine Änderung.

Die verwendeten Ansätze und Schwellenwerte sind Tabelle 6.1 zu entnehmen. Für Sensorwerte des Accelerometers und Gyroskops, sowie des Magnetfeldsensors wurde der erste Ansatz verwendet, da signifikante Änderungswerte nicht relativ zu dem vorherigen Sensorwert sind, sondern lediglich eine absolute Änderung erzeugen. Für Sensorwerte des Temperatur-, Licht- und Geräuschsensors wurde der zweite Ansatz verwendet, da diese ein allgegenwärtiges Rauschen aufnehmen, das je nach Umgebung variiieren kann. Für die WLAN-Zugangspunkte wird der dritte Ansatz verwendet, da die Sensorwerte binär sind. Die Schwellenwerte wurden so gewählt, dass jeder Standort Interrupts auslöst und nicht mehr als 50% der Ursprungsdaten genutzt werden. Je nach Route werden zwischen 15% und 50% verwendet.

Sensordaten	Fester Schwellenwert	Variabler Schwellenwert	Änderung
Accelerometer	0.1	-	-
Gyroskop	0.1	-	-
Magnetfeld	8	-	-
Temperatur	-	0.12	-
Licht	-	0.12	-
Geräusch	-	0.16	-
WLAN-Zugangspunkte	-	-	1

■ **Tabelle 6.1:** Schwellenwerte der simulierten Interrupts.

6.4 Feature-Extrahierung

Die Feature-Extrahierung ist der Prozess, in dem Feature aus den Rohdaten der Datenmenge extrahiert werden. In dieser Arbeit findet dieser Schritt nach der Filterung durch künstliche Interrupts statt. Features sind berechnete Attribute und Eigenschaften von einem oder mehreren Sensorwerten der Rohdaten.

6 TRAININGS- UND TESTDATEN

Durch die Feature-Extrahierung müssen die ML-Modelle diese Features nicht selbständig lernen, sondern lediglich darauf abstrahieren. Dies erleichtert das Training, kann aber gerade für tiefe NN die Generalisierungsfähigkeit einschränken, wenn die Features nicht manuell konstruiert wurden [SLCY11]. Einerseits benötigt das Entscheidungsbaum ML-Modell einen solchen Prozess, da Features das Rauschen der Rohdaten verringern und dadurch eine Partitionierung vereinfachen. Andererseits kann das FFNN durch die Limitierungen der Hardware möglicherweise nicht groß genug sein, um diese Features zu erlernen.

Mian hat in seiner Arbeit ein Datenfenster verwendet, um die Sensorwerte zu glätten [Mia21]. Als Datenfenster werden die letzten N Einträge der Sensordaten bezeichnet, wobei N die Fenstergröße ist. Mian hat eine hohe Abtastrate verwendet, wodurch die Unterschiede zu hintereinander liegenden Datensätzen gering ist und Rauschen eine große Auswirkung hat. Durch die künstlichen Interrupts werden nur Datensätze verwendet, die signifikante Änderungen enthalten, wodurch große Datenfenster in dieser Arbeit nicht benötigt werden. Es wird dennoch ein Datenfenster verwendet, da dadurch mehr Features konstruiert werden können, die eine bessere Generalisierung ermöglichen. Das Datenfenster ist aber signifikant kleiner, da durch die künstlichen Interrupts eine im Mittel geringe Abtastrate zu erwarten ist.

Tabelle 6.2 listet alle verwendeten Features auf. Zunächst werden aus jedem Sensor, sofern der Sensor dies erlaubt, die gleiche Menge von Features extrahiert. Aus den Datenfenstern der Sensordaten werden die Standardabweichung, Minimum, Maximum und der Durchschnitt für jeden Sensor berechnet. Zusätzlich wird der momentane Wert jedes Sensors als Feature verwendet. Da die Werte des Accelerometers und Gyroskops abhängig von der Ausrichtung der Sensorenbox ist, wird von diesen nur der Betrag der Summe der x-, y- und z-Komponenten verwendet. Für die Detektionsdaten der WLAN-Zugangspunkte erschien die Standardabweichung, Minimum, Maximum und Durchschnitt nicht sinnvoll, da die Daten nur binäre Werte annehmen.

Daneben wurden noch drei weitere Features aus den Metadaten extrahiert. Das erste Feature ist der zuletzt bestimmte Standort des ML-Modells. Das zweite Feature ist der zuletzt bestimmte Standort des ML-Modells, der nicht als unbekannter Standort gilt und nicht der aktuelle Standort ist. Mian hat ein Feature für jeden zu klassifizierenden Standort als Eingabe benutzt, dass auf eins gesetzt wird, wenn es erkannt wurde und ansonsten exponentiell abfällt [Mia21]. In dieser Arbeit wurde sich dagegen entschieden, da dies schlecht mit der steigenden Anzahl von zu klassifizierenden Standorten skaliert und eine Abhängigkeit zu dem zuletzt bestimmten Standort für die Robustheit vermieden werden sollte. Das dritte Feature ist die Standardabweichung über die Zeit des Datenfensters. Alle anderen Features sind Zeitunabhängig, da sie sich auf Zeitunabhängige Sensorwerte im Datenfenster beziehen.

Sensordaten	Standardabweichung	Minimum	Maximum	Durchschnitt	Wert
Accelerometer	X	X	X	X	X
Gyroskop	X	X	X	X	X
Magnetfeld	X	X	X	X	X
Temperatur	X	X	X	X	X
Licht	X	X	X	X	X
Geräusch	X	X	X	X	X
WLAN-Zugangspunkte	-	-	-	-	X
Letzter Standort	-	-	-	-	X
Letzter unterschiedlicher Standort	-	-	-	-	X
Zeit	X	-	-	-	-

■ **Tabelle 6.2:** Extrihierte Features aus verfügbaren Sensordaten.

Die Signifikanz der einzelnen Features ist abhängig von der Einsatzumgebung, weshalb in Kapitel 5.4 ein Feature-Auswahl Schritt in dem Trainingsprozess vorgeschlagen wurde, um die Anzahl der Features zu verringern. Dabei wird die Signifikanz, oder Wichtigkeit, über die Permutationswichtigkeit bestimmt. Aus diesem Grund muss individuell für jedes Einsatzgebiet abgewogen werden, welche Sensoren und Features am meisten Nutzen, im Vergleich zu deren Kosten und Energieverbrauch, bringen.

6.5 Synthetische Daten

Es kann passieren, dass Sensoren über die Zeit kaputt gehen oder vorübergehend gestört werden. Ist die Abhängigkeit des ML-Modells zu groß zu diesen Sensoren, dann wird die Klassifizierungsfähigkeit des ML-Modells stark eingeschränkt, wenn dieser Fall eintritt. Um die ML-Modelle robust gegenüber diesen Szenarien zu machen, werden synthetische Daten auf Basis existierenden Daten erzeugt, die zusätzlich im Training benutzt werden.

Mögliche Fehler der Sensoren sind temporärer oder permanenter Ausfall oder leichte Abweichungen zum Originalwert. Zudem können Standorte nicht erfasst werden, weil kein Interrupt ausgelöst wurde als die Sensorenbox sich an diesem Standort befand oder die Sensorbox hat auf unbekannten Wege einen Standorte übersprungen.

6 TRAININGS- UND TESTDATEN

Fehlerhafte Sensoren werden durch die Duplizierung von Routen dargestellt, in denen die Sensoren genutzt werden bzw. ein zufälliges Rauschen addiert wird. Verpasste Standorte werden zum einen durch permutierte duplizierte Routen dargestellt. Zum anderen werden sie durch duplizierte Routen dargestellt, in denen der zuletzt besuchte nicht unbekannte Standort übersprungen wurde. Um keinen Bias zu erzeugen, werden von den synthetischen Routen 10% von jedem Standort zur Trainingsmenge hinzugefügt.

6.6 Anomaliedaten

Für Anomaliedaten werden die bekannten Routen modifiziert, indem Umleitungen eingebaut werden, die teilweise Standorte überspringen und nicht in den Trainingsdaten vorhanden sind. Dadurch wird ein Szenario simuliert, in der das Klassifizierungsverhalten von dem ML-Modell zur Standorterkennung auf unbekannten Wegen beobachtet werden kann. Aus den Klassifizierungsergebnissen des ML-Modells zur Standorterkennung werden schließlich Features extrahiert, die von dem ML-Modell zur Anomalieerkennung verwendet werden.

Wenn eine Anomalie vorliegt, wird erwartet, dass das ML-Modell unsicherer wird und stärker fluktuiert, wodurch drei Features motiviert sind. Das erste Feature ist abgeleitet aus der Anzahl der Standortänderungen. Zum einen wird die durchschnittliche Anzahl der Standortänderungen in einem Datenfenster bestimmt. Zum anderen wird die durchschnittliche Anzahl der Standortänderungen bestimmt, wenn keine Anomalie vorliegt. Das Feature ist der Betrag der Differenz von diesen beiden Komponenten.

6.6 ANOMALIEDATEN

Das zweite Feature ist analog zu dem ersten Feature konstruiert. Dieses nutzt ansatt der Anzahl der Standortänderungen die summierte Wahrscheinlichkeit der erkannten Standorte. Das ML-Modell zur Standorterkennung hat keine diskrete Ausgabe, sondern gibt einen Vektor von Wahrscheinlichkeiten aus, der für jeden Standort die Klassifizierungswahrscheinlichkeit angibt. Dabei gilt der klassifizierte Standort als der Eintrag im Vektor mit der höchsten Wahrscheinlichkeit. Diese Wahrscheinlichkeit wird dann analog zu den Standortänderungen summiert. Das dritte Feature ist die Standardabweichung der ersten fünf Einträge dieses Vektors, der absteigend sortiert ist.

Zusätzlich wird das Ergebnis des Modells auf Basis der Topologie als Feature verwendet. Dieses Feature indiziert, dass das ML-Modell zur Standorterkennung ein Fehler gemacht haben muss oder dass tatsächlich eine Anomalie vorliegt, da die Topologie verletzt wurde.

Daneben wurde ebenfalls eine Rückwärtskante in Betracht gezogen, da es wahrscheinlich ist, dass wenn zuvor eine Anomalie vorlag, danach immer noch eine Anomalie vorliegt. Im Training wurden damit zwar sehr gute Ergebnisse erreicht, in der Praxis war das Modell aber equivalent zu dem *Immer-Falsch* Modell.

6 TRAININGS- UND TESTDATEN

Evaluation

In der Evaluation werden drei Aspekte betrachtet: Klassifizierungsgenauigkeit, Robustheit und Resourcennutzung. Bei der Klassifizierungsgenauigkeit wird einerseits die Standorterkennung und andererseits die Anomalieerkennung evaluiert. Dabei wird sowohl auf verschiedene Größen von FFNN und Entscheidungswälder eingegangen, als auch auf verschieden viele zu unterscheidenden Standorte.

Bei der Robustheit wird auf den Fehler des besten FFNN und Entscheidungswald bei verschiedene Fehlerszenarien eingegangen. Diese bestehen aus fehlerhaften Sensordaten durch Rauschen oder ausgefallenen Sensoren, Routen mit permutierten Teilstücken und Routen, bei denen simuliert wird, dass der letzte Standort übersprungen wurde.

Bei der Resourcennutzung wird auf die Programmgröße und die Ausführungszeit der besten ML-Modelle eingegangen. Außerdem wird der Energieverbrauch für verschiedene Szenarien eingeschätzt.

Unterschieden werden zwei Varianten der Testmengen. Diese unterscheiden sich in der Art, wie die Features für den vorherigen Standort bestimmt werden. In der ersten Variante sind alle vorherigen Standorte korrekt. In der zweiten Variante ist der erste vorherige Standort 0 und alle folgenden vorherigen Standorte werden iterativ durch das ML-Modell bestimmt, d. h. in dieser Variante wird der propagierte Fehler durch die Rückwärtskante des ML-Modells betrachtet.

7.1 Metriken

In dieser Arbeit werden verschiedene Metriken zur Ermittlung der Klassifizierungsgenauigkeit ermittelt. Zunächst die übliche Klassifizierungsgenauigkeit (7.1), in der die Anzahl der korrekt

7 EVALUATION

klassifizierten Standorte mit der Gesamtanzahl verglichen werden. Diese Metrik wird auch von Mian verwendet und ist für den direkten Vergleich erforderlich.

$$P(A) := \frac{\text{Anzahl korrekter Klassifizierungen}}{\text{Gesamtanzahl}} \quad (7.1)$$

Die zweite Metrik (7.2) betrachtet die Klassifizierungsgenauigkeit unter Tolerierung, dass ein Standort fünf bzw. zehn Klassifizierungen kontinuierlich zu früh oder zu spät verlassen wurde, d. h. Fehlklassifizierungen werden vernachlässigt, wenn kontinuierlich der letzte korrekte Standort bzw. der nächste korrekte Standort klassifiziert wird mit einer Gesamttoleranz von fünf bzw. zehn Klassifizierungen. Diese Metrik ist besonders gut für den Vergleich geeignet, da, entgegen der Beschriftung in der Testmenge, harte Übergänge zwischen zwei benachbarten Standorten nicht der Wirklichkeit entsprechen. Im Übergang können Sensorwerte möglicherweise mehrdeutig sein und dies sollte bei der Standorterkennung beachtet werden, solange sie konsequent ist. Dadurch wird im Vergleich das Rauschen in der Klassifizierungsgenauigkeit reduziert, dass durch diese Mehrdeutigkeit im Übergang entsteht, wodurch die ML-Modelle besser vergleichbar sind.

$$\epsilon \in \{5, 10\}$$

L := Menge von dem ML-Modell klassifizierten Standorte.

K := Menge von den wirklichen Standorten.

$\Phi(i)$:= Index von dem nächsten Standort.

$\Psi(i)$:= Index von dem vorherigen Standort.

$$\Omega(i) := \Phi(i) - i \leq \epsilon \wedge \bigwedge_{i \leq q \leq \min(\#K, \Phi(i))} L_q = K_{\Phi(i)}$$

$$\Theta(i) := i - \Psi(i) \leq \epsilon \wedge \bigwedge_{\max(0, \Psi(i)) \leq q \leq i} L_q = K_{\Psi(i)}$$

$$P(B) := \frac{\#\{L_i | L_i = K_i \vee \Omega(i) \vee \Theta(i) \text{ für } i \in \{0, 1, \dots, \#L - 1\}\}}{\#K} \quad (7.2)$$

Zuletzt zwei Metriken bei denen die Klassifizierungsgenauigkeit bestimmt wird, unter der Bedingung, dass der vorherige Standort korrekt bzw. inkorrekt (7.3) war. Diese Metriken sind besonders relevant bei der Beurteilung der Stabilität der ML-Modelle, insbesondere wenn das ML-Modell einen Fehler gemacht hat.

$$P(C) \text{ bzw. } P(D) := \frac{\text{Anzahl korrekter Klassifizierungen, wenn vorheriger Standort (in)korrekt war}}{\text{Alle Klassifizierungen, wenn vorheriger Standort (in)korrekt war}} \quad (7.3)$$

7.2 Klassifizierungsgenauigkeit der Standorte

- Wie groß ist die Wahrscheinlichkeit, dass die Orte korrekt erkannt werden?
- Entscheidungsbaum vs KNN
- Skalierung mit Anzahl der Orte

7.3 Klassifizierungsgenauigkeit der Anomalien

- Was ist das?
- Zusammenhang zu Enkodierungsansatz
- Schwierig zu trainieren, da man einerseits Robust sein will und andererseits nicht weiß was trainiert werden soll
- Post Processing
- Metriken: Location Change Frequency, Accumulated Confidence, Fraction Zero
- Sag Motivation, warum diese Metriken (Indikatoren)
- Beispiele
- Wie zuverlässig können Anomalien erkannt werden?
- Vergleich mit Coin-Toss, True und False
- Vergleiche Enkodierungsansätze, ob einer besser als der andere ist

7.4 Signifikanz der Features

- Signifikanz der Features
- Einfluss von einzelnen Features für die Klassifizierungsgenauigkeit(?) => Last Distinct location möglicherweise schlecht, da die eigentliche Last Distinct Location oft durch fehlende Interrupts übersprungen wird
- Welche Feature werden genutzt? => Abhängig von Feature Importance und wie günstig zu berechnen
- Ist es sinnvoll für jeden Ort ein Feature zu haben, dass auf eins gesetzt wird, wenn der Ort erkannt wurde und ansonsten exponentiell abfällt. Wie schnell sollte es fallen, wenn ja?

7 EVALUATION

7.5 Benötigte Anzahl der Trainingsdaten

- Werden mehr Trainingsdaten benötigt mit steigender Ort Anzahl? Wenn ja wie viel?
(Hier oder bei ML-Modell Training)
- Wie viele Trainingsdaten werden benötigt?
 - ◆ Um KNN zu trainieren?
 - ◆ Um Entscheidungsbaum zu trainieren?
 - ◆ Ggf. Unterschiede klären
 - ◆ (Gehört das schon in eine Evaluation, oder ist das hier okay?)

7.6 Fehlertoleranz

- Irgendwas über Robustheit
- Metriken => Insbesondere continued_predict hier erwähnen, i.e. dass es dadurch instabiler wird, weil der Fehler propagiert wird.
- Wenn falscher Ort erkannt wurde, wie lange dauert es um wieder den korrekten Ort zu finden? Was ist die Wahrscheinlichkeit dafür? Wovon hängt es ab, dass wieder der richtige Ort erkannt wurde?
- Was passiert wenn Sensoren ausfallen? Wie robust ist es dagegen?
- Transportbox nicht dem trainierten Pfad folgt? Wie Robust ist sie dagegen? => Sezenerien enumrieren, wo sowas passieren kann, dann Testszenario erklären.
- Änderungen der Fabrik, e.g. Licht, Wärme, Magnet, Schnelligkeit der Fließbänder
- Einfluss von einzelnen Features für die Fehlertoleranz(?)
 - In welchen Fällen ist die Ortung unzuverlässig? => Welche Fälle sind schwierig?
- Schlussfolgerung: Wie Fehlertolerant ist die Ortung? => Zusammenfassende Metriken über alle Testsets?
- Diskutieren, dass an fast jedem Standort mehrere Sensoren verfügbar sind, weswegen eine gewisse Robustheit möglich sein sollte.

7.7 Programmspeicher

Der Großteil des Programmspeichers wird für das ML-Modell benötigt. Aus diesem Grund wird der Anteil des Programmspeichers X in der Evaluation vernachlässigt, der für die restlichen Funktionen und für die Feature-Extrahierung benötigt wird. Zudem ist der benötigte Programmspeicher dieses Anteils konstant und skaliert nicht mit der Größe, wie die ML-Modelle.

Zur Estimierung des Programmspeichers der Entscheidungsbäume wird der hybride Ansatz mit einer Toleranz von $\epsilon = 0$ angenommen, d. h. es werden für eindeutige Ergebnisse diskrete Rückgaben zurückgegeben, anstatt der Wahrscheinlichkeitsverteilung. Als Datentyp für die Vergleiche und allen Features wird angenommen, dass ein vier Byte Datentyp verwendet wird. Für einen Vergleich werden fünf Instruktionen benötigt [Dym21]. Für eine Rückgabe werden zwischen zwei und $2(N + 1)$ Instruktionen und zwischen 0 und $2N$ Parameter benötigt, wobei N die Anzahl der möglichen Standorte ist. Die Größe einer Instruktion ist 4 Byte, da eine 32-Bit CPU angenommen wird. Die Größe des zusammenfassenden Klassifizierers Z wird vernachlässigt.

Für die FFNNs wird ebenfalls ein vier Byte Datentyp für die Biase und die Gewichte angenommen. Die Größe des Algorithmus zur Ausführung des KNN ist unbekannt und wird als konstanter Wert Y angenommen, liegt aber, den Zahlen in Giese's Arbeit nach zu urteilen, zwischen 6 und 7 KB [Gie20].

Tabelle 7.1 zeigt Estimierungen des benötigten Programmspeichers der verschiedenen Konfigurationen der ML-Modelle, wobei die konstanten Anteile vernachlässigt werden. Potentielle Optimierungen, z. B. durch den Compiler, wurden dabei nicht betrachtet, sowie potentielle Optimierungen des FFNN, wie Giese sie vorgeschlagen hat [Gie20]. Giese hat mit dem CSC-MA-Bit Format die Programmgröße um 39% reduzieren können. Kompilierung mit der Optimierungsstufe $O2$ konnte experimentell generierten C-Code eines Entscheidungswaldes um bis zu 21,3% reduzieren.

Der benötigte Programmspeicher beider ML-Modelle skaliert mit der Anzahl der zu klassifizierenden Standorte, der Anzahl der Schichten bzw. Bäume und der Anzahl der Neuronen pro Schicht bzw. der maximalen Baumhöhe. Der von den Entscheidungswäldern benötigte Programmspeicher ist für fast alle Fälle zu viel für die Limitierungen eines kleinen eingebetteten Systems. Die FFNNs hingegen benötigen deutlich weniger Programmspeicher und könnten innerhalb der Limitierungen der kleinen eingebetteten Systeme passen.

7 EVALUATION

TODO: Braucht man mehr Neuronen/Hidden Layer mit steigender Ort Anzahl? (Hier oder bei ML-Modell FFNN)

TODO: Wie viel Speicherverbrauch spart man durch die Feature-Selection zusätzlich ein?

Größe in KB über Standorte	9	16	17	25	32	48	52	102
Entscheidungswälder								
Waldgröße	Max. Baumgröße							
16	8	79.9	83.2	117.9	169.6	147.2	204.1	254.9
16	16	192.2	199.0	277.4	512.7	371.1	750.4	914.4
16	32	185.7	197.6	287.2	550.2	394.5	875.6	1016.7
16	64	185.7	197.6	287.2	543.0	394.5	875.6	943.6
8	32	94.7	108.1	136.7	252.7	192.9	436.8	465.6
16	32	185.7	197.6	287.2	550.2	394.5	875.6	1016.7
32	32	364.8	401.7	575.7	1055.9	803.9	1701.0	1962.6
64	32	776.1	842.8	1173.8	2132.7	1584.7	3455.2	3987.8
32	64	364.8	401.7	575.7	1055.9	803.9	1701.0	1962.6
								3215.9
Feed Forward neuronale Netzwerke								
#Schichten	#Neuronen							
1	16	2.8	3.2	3.3	3.8	4.2	5.2	5.5
1	32	5.5	6.4	6.5	7.6	8.4	10.5	11.0
1	64	11.0	12.8	13.1	15.1	16.9	21.0	22.0
1	128	22.0	25.6	26.1	30.2	33.8	42.0	44.0
2	32	9.6	10.5	10.6	11.6	12.5	14.6	15.1
4	32	17.8	18.7	18.8	19.8	20.7	22.8	23.3
8	32	34.2	35.1	35.2	36.2	37.1	39.2	39.7
4	64	60.2	62.0	62.2	64.3	66.0	70.1	71.2
								84.0

■ **Tabelle 7.1:** Größe in KB über Standorte und verschiedenen Konfigurationen der ML-Modelle zur Standorterkennung.

7.8 RAM

Für den benötigten RAM muss neben dem Anteil der ML-Modelle, die Historie der Sensorwerte und die berechneten Features betrachtet werden. Wichtig ist dabei der größte akkumulierte

benötigte RAM, der zu einem Zeitpunkt benötigt werden kann.

Für jeden Sensorwert, bis auf der Detektion von WLAN-Zugangspunkten, wird ein vier Byte Datentyp angenommen. Für die Detektion der Zugangspunkte wird ein ein Byte Daten- typ angenommen. Insgesamt beträgt der benötigte RAM für einen Vektor von Sensorwerten damit 61 Byte. Dies setzt sich zusammen aus dem Zeitstempel, der xyz-Komponente von Accelerometer und Gyroskop, dem Lichtsensor, der Temperatursensor, der Magnetfeldsensor, der Geräuschsensor und fünf möglichen WLAN-Zugangspunkten. Bei einem Datenfenster von drei Einträgen wird damit 183 Byte für Sensorwerte benötigt.

Der Anteil der Features ist abhängig von den Features die für ein bestimmtes Szenario eingesetzt werden. Insgesamt werden aber 34 Features verwendet, die vereinfacht alle als 4 Byte Datentyp angenommen werden. Zur Evaluierung des ML-Modells wird nur die aktuelle Feature-Menge benötigt. Damit wird für die Feature-Menge insgesamt 136 Byte benötigt, wenn alle Features verwendet werden.

Zur Ausführung eines Entscheidungswaldes wird für die Rückgabe der Wahrscheinlichkeitsverteilung für jeden Standort vier Byte benötigt. Je nach Implementierung würde dieser Vektor mehrmals benötigt werden, z. B. bei der parallelen Evaluierung der Entscheidungsbäume skaliert dies mit der Anzahl der Prozessor. In diesem Fall wird keine Nebenläufigkeit angenommen. In dieser Arbeit wurden zwischen 9 und 102 Standorte untersucht, d. h. es wurden zwischen 36 und 408 Byte benötigt. Die Anzahl der Standorte ist aber abhängig von dem Einsatzszenario. Die anschließende Evaluierung eines Entscheidungswaldes zur Anomalieerkennung kann vernachlässigt werden, da dieser ein diskretes Ergebnis zurückgeben kann und die benötigte Feature-Menge deutlich kleiner ist. Damit wird für N Standorte und K Features mit einem Entscheidungswald als ML-Modell zu einem Zeitpunkt ca. $183 + 4(N + K)$ Byte benötigt, d. h. bei 102 Standorten und 34 Features ca. 727 Byte.

Zur Ausführung eines KNN können nur wenige Byte verwendet werden, um die nötigen Multiplikationen eines Neuronen durchzuführen. Dies würde die Ausführungszeit, und den Energiebedarf, aber signifikant erhöhen, da die benötigten Gewichte ständig aus dem Programmspeicher geladen werden müssen. Das heißt, es müssen mindestens die Zwischenergebnisse einer Schicht im RAM gehalten werden, sowie ein Gewicht und ein Bias. Damit benötigt ein FFNN, dessen größte Schicht M Neuronen hat, mindestens $4(M + 2)$ Byte. Maximal wird $4M$ Byte, zuzüglich der Größe aller Gewichte und Biase benötigt. Der maximale RAM, der zu einem Zeitpunkt benötigt wird mit einem FFNN, beträgt damit mindestens $183 + 4(M + K)$, wobei M die Schicht mit den meisten Neuronen von entweder dem ML-Modell zur Standort- oder Anomalieerkennung ist.

7.9 Ausführungszeit und benötigte Energie

Es ist sehr problematisch eine sinnvolle Estimation für die benötigte Ausführungszeit und Energie anzugeben. Ausführungszeit und die benötigte Energie sind abhängig von dem verwendeten Mikrocontroller. Vergleichbare 32-Bit Mikrocontroller mit FPU (Floating Point Unit), zu den Microcontrollern die Dymel verwendet hat [Dym21], sind aus der AVR C Serie [Cor12]. Leider ist deren Datenblätter keine Information über die Ausführungszeit von Gleitkommazahlinstruktionen und kein Energiemodell zu entnehmen. Es ist aber anzunehmen, dass deutlich weniger Cyclen benötigt werden für hardwareunterstützte Gleitkommazahloperationen, als Software basierte Alternativen. Aus diesem Grund wird die Ausführungszeit in Gleitkommazahl- Vergleichen, Multiplikationen, Division, Additionen und Wurzel angegeben, da diese die integralen Bestandteile der Feature-Extrahierung und Evaluation der ML-Modelle sind.

Hier werden die Konfigurationen (TODO) betrachtet, da diese die beste Klassifizierungsgenauigkeit erzielt in Relation zu ihrer Größe erzielt haben. Die in dieser Arbeit vorgeschlagene Architektur (5.1) hat fünf Bestandteile, die jeweils zur Gesamtausführungszeit beitragen. Die Aufnahme der Sensorwerte wird als konstanter Energieverbrauch angenommen und in dieser Rechnung vernachlässigt. In der ersten Feature-Extrahierung werden 34 Features aus dem Datenfenster extrahiert. Tabelle 7.2 zeigt die estmierte Anzahl der Operationen, die pro Art des Features benötigt werden.

Art des Features	Vergleich	Multiplikation	Division	Addition	Wurzel
Standardabweichung (7)	0	3	2	7	1
Minimum (6) / Maximum (6)	2	0	0	0	0
Durchschnitt (6)	0	0	1	2	0
Wert (9)	0	0	0	0	0

■ **Tabelle 7.2:** Estimierte Anzahl der Operationen pro Art des Features bei einer Datenfenstergröße von 3. Fettgedruckte Zahl zeigt die Verwendungsanzahl in der Feature-Menge an.

Tabelle (TODO) zeigt die estmierte Anzahl der Operationen, die jeweils für den besten Entscheidungswäldern und FFNNs zur Standorterkennung benötigt werden. Die Anzahl der Vergleichsoperationen und Multiplikationen hängen nicht mit der Anzahl der Standorte zusammen, jedoch haben sich verschieden komplexe Strukturen für diese Fälle als Beste erwiesen.

Bei der Feature-Extrahierung für das ML-Modell zur Anomalieerkennung werden nur vier Features extrahiert. Tabelle 7.3 zeigt die estmierte Anzahl der Operationen, die für die einzelnen

7.9 AUSFÜHRUNGSZEIT UND BENÖTIGTE ENERGIE

Features benötigt werden. Der Entscheidungswald zur Anomalieerkennung benötigt (TODO) Vergleiche und das FFNN zur Anomalieerkennung benötigt (TODO) Multiplikationen.

Feature	Vergleich	Multiplikation	Division	Addition	Wurzel
Abweichung zum ØStandortänderungen	4	0	2	5	0
Abweichung zum ØKlassierungswahrscheinlichkeit	4	0	2	5	0
Topologieverletzung	5	1	0	1	0
Standardabweichung Top 5 Klassifizierungen	0	5	2	13	1

■ **Tabelle 7.3:** Estimierte Anzahl der Operationen pro Feature der Anomalieerkennung.

Tabelle (TODO) fasst die Anzahl der Operationen für eine Konfiguration mit ausschließlich Entscheidungswäldern und FFNNs zusammen. Es ist zu erwarten, dass die Entscheidungswälder weniger Ausführungszeit benötigen als die FFNNs, da deutlich weniger Operationen benötigt werden, wodurch sich ein Entscheidungsbaum basierter Klasifizierer in Hinsicht auf die benötigte Energie besser eignet. Bei konstanter Bewegung wurde in den Testszenarien der Mikrocontroller alle (TODO)s aufgeweckt. Dies ist aber unrealistisch, da die Sensorenbox auch für lange Zeit an einem Ort verbleiben kann, was sich positiv auf den Energiebedarf auswirkt.

7 EVALUATION

Diskussion

Ein Großteil dieser Arbeit ist die Generierung von Sensordaten. Dazu wurde der Ansatz von Mian aufgegriffen, wodurch verschiedene Routen in CoppeliaSim simuliert wurden. Der Nachteil dieses Ansatzes ist die schlechte Skalierbarkeit, wenn es um die Untersuchung von Pfaden und Standorten geht. Um weitere Pfade und Standorte zu untersuchen, mussten entweder komplexere Routen in CoppeliaSim erstellt werden, Routen synthetisch kombiniert werden oder Routen unabhängig von einander trainiert werden.

Der erste Ansatz ist sehr aufwändig und der zweite Ansatz unterscheidet sich nicht stark von dem dritten Ansatz, da die Routen nur an einem Punkt kombiniert werden. Ein besserer Ansatz wäre es gewesen, wenn man lediglich abgeschlossene Teilstücke simuliert hätte, z. B. eine Kurve oder eine Gerade nach links laufend. Jedes dieser Teilstücke hätte eine feste Länge und könnte sogar transformiert werden in sowohl der Länge, als auch den aufgenommen Sensordaten. Durch geschickte Kombination, könnten dann beliebig komplexe Routen mit beliebig vielen Pfaden und Standorten generiert werden, was die Untersuchung erheblich erleichtert hätte.

8 DISKUSSION

Schlussfolgerungen

- Kurze, knappe und gut formulierte Schlussfolgerung
 - ◆ Was ist besser Entscheidungsbaum oder Entscheidungswald? Welche Vor- und Nachteile?
 - ◆ Sollten Entscheidungswälder verwendet werden? => Vor und Nachteile Besprechen
 - ◆ Sollten KNNs verwendet werden? => Vor und Nachteile Besprechen
 - ◆ Is die Realzeit relevant für die Ortung(?)
 - ◆ Wie Fehlertolerant ist die Ortung?
 - ◆ Wie gut können Anomalien erkannt werden?
- Kurze Zusammenfassung wichtigster Dinge

Zukünftige Arbeiten sollten drei Bereiche weiter untersuchen. Zunächst sollten diese ML-Modelle mit Echtdaten untersucht werden. Dafür müsste ein Prototyp der Sensorenbox konstruiert werden und in einem geeigneten Szenario ausreichend Daten aufgenommen werden. Möglicherweise könnte ein Arduino Board mit handelsüblichen, vergleichbaren Sensoren ausreichen.

Weitere Optimierungen der ML-Modelle sollten untersucht werden. Zunächst könnte die Anzahl der Features reduziert werden durch eine sorgfältige Auswahl mit Hilfe eines Optimierungsverfahrens. Dort sollte die Klassifizierungsgenauigkeit und Ausführungszeit maximiert werden unter Einhaltung der Hardwarelimitierungen. Zur Optimierung von KNN könnten die Ansätze von Giese aufgegriffen werden [Gie20].

Ein Energiemodell zur Einschätzung des Energieverbrauchs der verschiedenen ML-Modelle könnte erstellt werden. Dieses könnte für verschiedene Batteriegrößen und Mikrocontroller,

9 SCHLUSSFOLGERUNGEN

Einschätzungen für obere Schranken der Größe von verschiedenen ML-Ansätzen angeben.
Damit können dann gezielter weitere Ansätze untersucht werden.

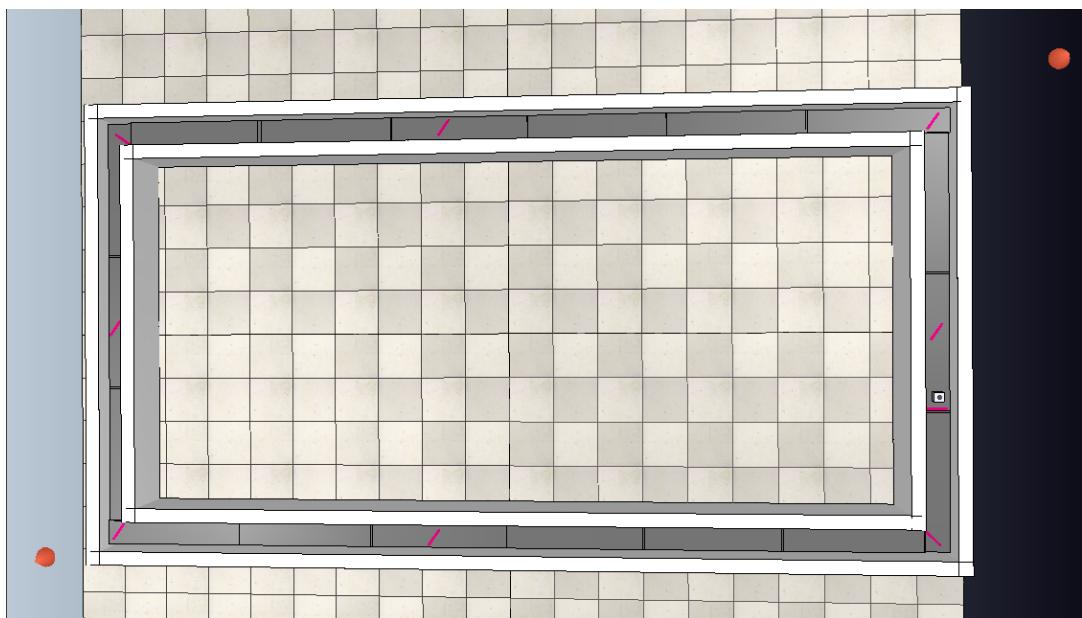
Inhalt des USB-Sticks

- PDF und Quelldateien für diese Arbeit (/thesis/thesis.pdf)
- PDF und Quelldateien für den Antritts-, Interem- und Abschlussvortrag (/antrittsvortrag, /intermediate_vortrag, /abschlussvortrag)
- Quelldateien für die Szenen zur Generierung der Sensordaten in CoppeliaSim (/scenes)
- ZIP mit den simulierten Sensordaten aus CoppeliaSim (/bin_data.zip)
- Ergebnisse der Evaluation von verschiedenen ML-Modellen in Rohform (/bin)
- Quelldateien zur Verarbeitung der Sensordaten und Generierung der Modelle (/sources)

Weitere Informationen können dem README.md entnommen werden.

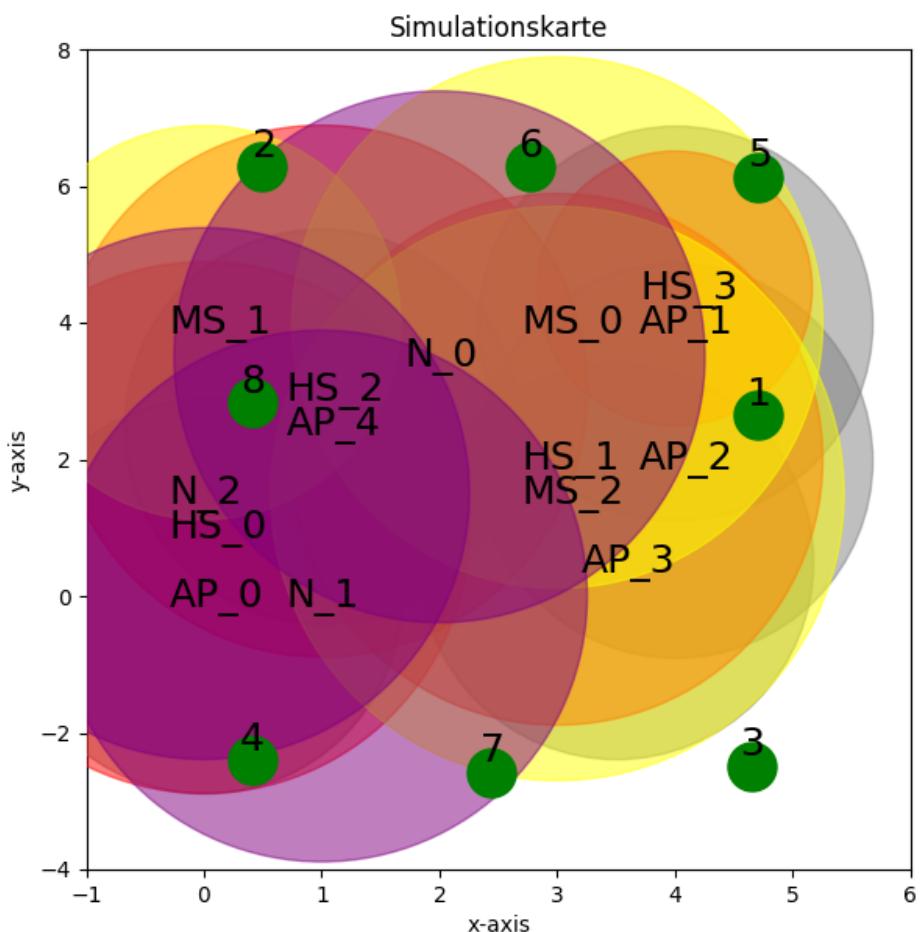
A INHALT DES USB-STICKS

Bildanhänge

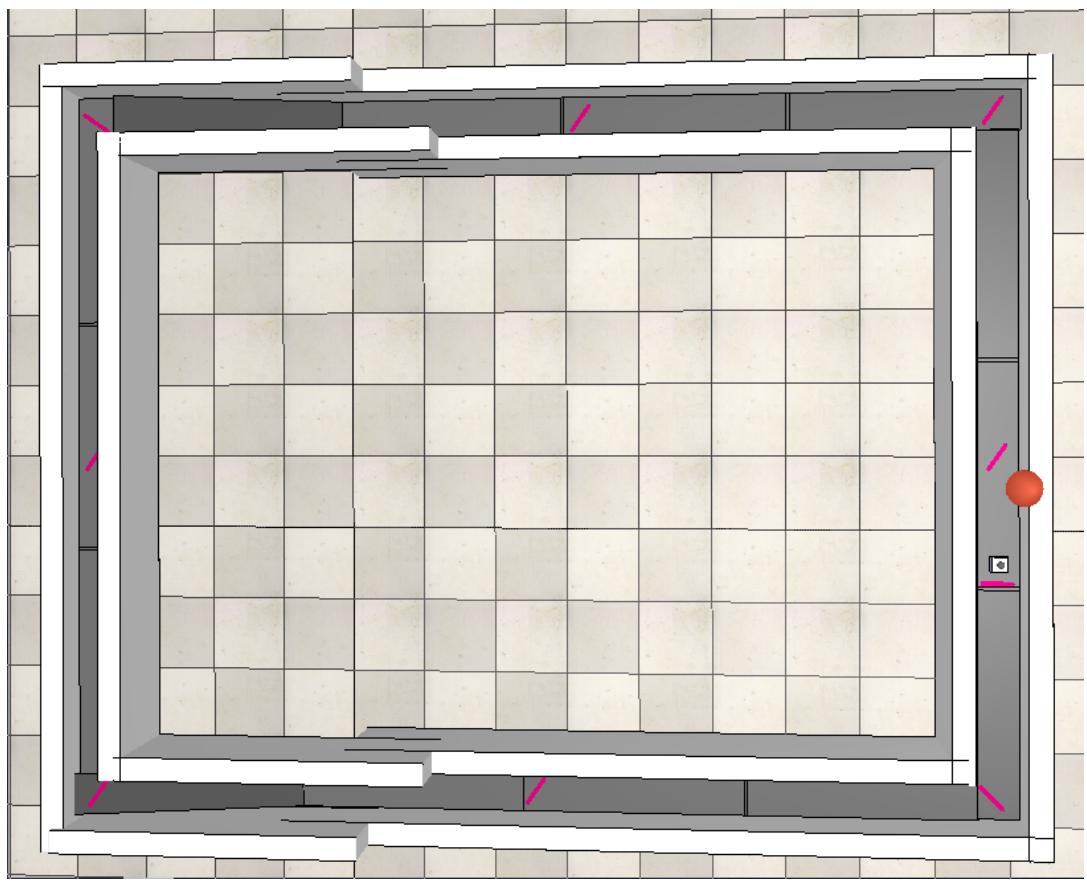


■ Abbildung B.1: Modell der Route „long_rectangle“ in CoppeliaSim.

B BILDANHÄNGE

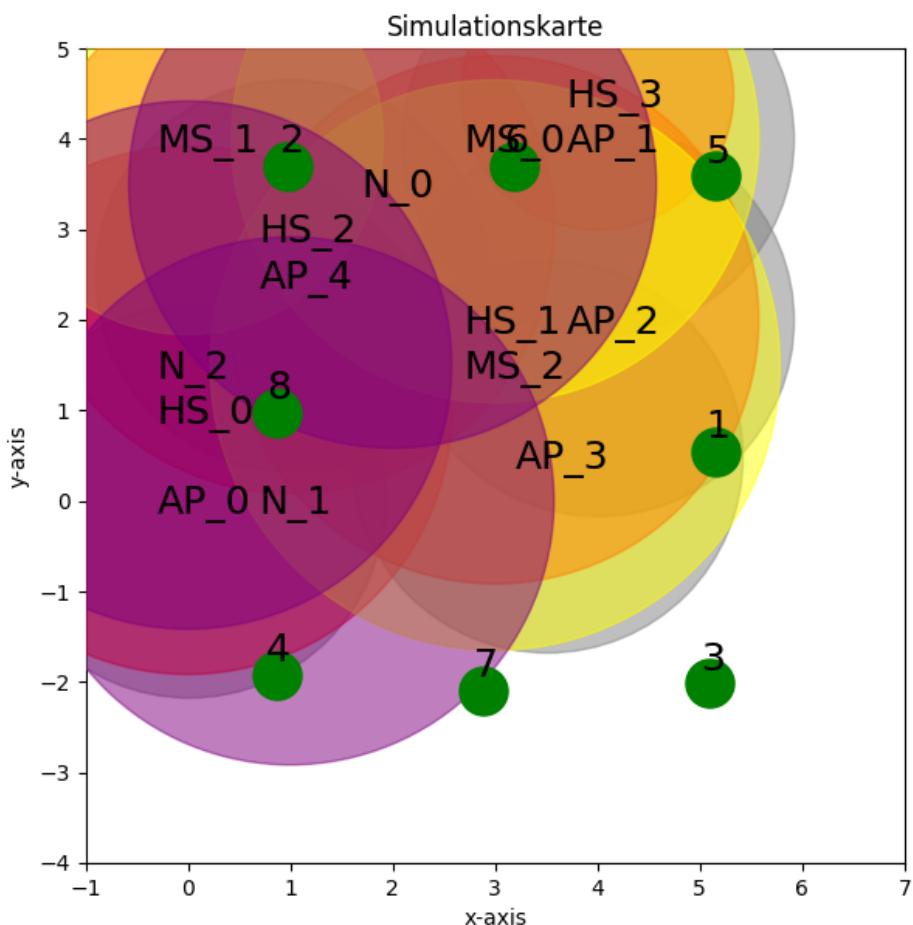


■ **Abbildung B.2:** Karte der Route „long_rectangle“ mit eingezeichneten Einflussbereichen der Objekte, die Einfluss auf modellierte Sensoren haben. **Magnetic Source** (Gelb), **Noise Source** (Lila), **Access Point** (Grau), **Heat Source** (Rot) und Standorte (Grün).



■ **Abbildung B.3:** Modell der Route „rectangle_with_ramp“ in CoppeliaSim.

B BILDANHÄNGE

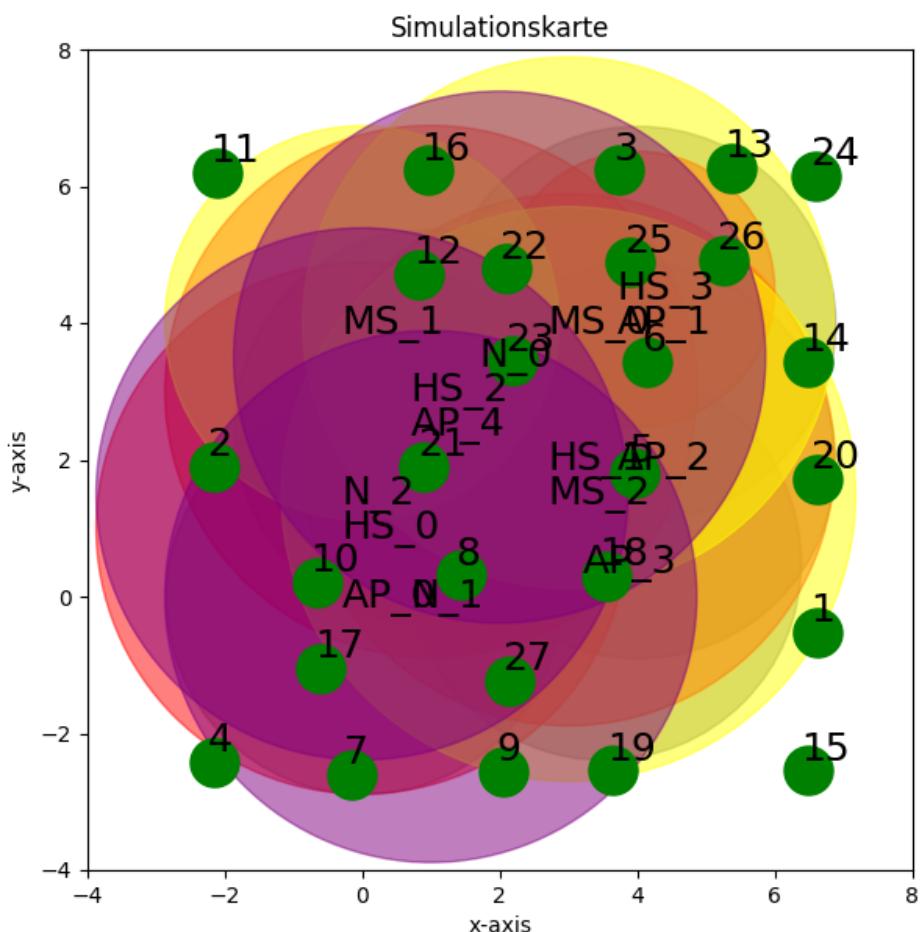


■ **Abbildung B.4:** Karte der Route „rectangle_with_ramp“ mit eingezeichneten Einflussbereichen der Objekte, die Einfluss auf modellierte Sensoren haben. *Magnetic Source* (Gelb), *Noise Source* (Lila), *Access Point* (Grau), *Heat Source* (Rot) und Standorte (Grün).



■ Abbildung B.5: Modell der Route „many_corners“ in CoppeliaSim.

B BILDANHÄNGE



■ **Abbildung B.6:** Karte der Route „many_corners“ mit eingezeichneten Einflussbereichen der Objekte, die Einfluss auf modellierte Sensoren haben. *Magnetic Source* (Gelb), *Noise Source* (Lila), *Access Point* (Grau), *Heat Source* (Rot) und Standorte (Grün).

Literaturverzeichnis

- [ABC⁺16] ABADI, Martín ; BARHAM, Paul ; CHEN, Jianmin ; CHEN, Zhifeng ; DAVIS, Andy ; DEAN, Jeffrey ; DEVIN, Matthieu ; GHEMAWAT, Sanjay ; IRVING, Geoffrey ; ISARD, Michael u. a.: Tensorflow: A system for large-scale machine learning. In: *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, S. 265–283
- [ÅBLM14] ÅKESSON, Susanne ; BOSTRÖM, Jannika ; LIEDVOGEL, Miriam ; MUHEIM, Rachel: Animal navigation. In: *Animal movement across scales* 21 (2014), S. 151–178
- [ADIP21] APICELLA, Andrea ; DONNARUMMA, Francesco ; ISGRÒ, Francesco ; PREVETE, Roberto: A survey on modern trainable activation functions. In: *Neural Networks* (2021)
- [Alc18] ALCAIDE, Eric: E-swish: Adjusting activations to different network depths. In: *arXiv preprint arXiv:1801.07145* (2018)
- [BFSO84] BREIMAN, Leo ; FRIEDMAN, Jerome ; STONE, Charles J. ; OLSHEN, Richard A.: *Classification and regression trees*. CRC press, 1984
- [BGC17] BENGIO, Yoshua ; GOODFELLOW, Ian ; COURVILLE, Aaron: *Deep learning*. Bd. 1. MIT press Massachusetts, USA:, 2017
- [BHE00] BULUSU, Nirupama ; HEIDEMANN, John ; ESTRIN, Deborah: GPS-less low-cost outdoor localization for very small devices. In: *IEEE personal communications* 7 (2000), Nr. 5, S. 28–34
- [Bre96] BREIMAN, Leo: Bagging predictors. In: *Machine learning* 24 (1996), Nr. 2, S. 123–140
- [Bre01] BREIMAN, Leo: Random forests. In: *Machine learning* 45 (2001), Nr. 1, S. 5–32
- [CGSS14] CORRAL, Luis ; GEORGIEV, Anton B. ; SILLITTI, Alberto ; SUCCI, Giancarlo: Can execution time describe accurately the energy consumption of mobile apps? an experiment in android. In: *Proceedings of the 3rd International Workshop on Green and Sustainable Software*, 2014, S. 31–37
- [Cor12] CORPORATION, Atmel: *AT32UC3C datasheet*. <https://ww1.microchip.com/downloads/en/DeviceDoc/doc32117.pdf>. Version: 2012
- [CPC⁺11] CUNHA, Joao ; PEDROSA, Eurico ; CRUZ, Cristóvao ; NEVES, António JR ; LAU, Nuno: Using a depth camera for indoor robot localization and navigation. In: *DETI/IEETA-University of Aveiro, Portugal* (2011), S. 6
- [CUH15] CLEVERT, Djork-Arné ; UNTERTHINER, Thomas ; HOCHREITER, Sepp: Fast and accurate deep network learning by exponential linear units (elus). In: *arXiv preprint arXiv:1511.07289* (2015)
- [D⁺02] DIETTERICH, Thomas G. u. a.: Ensemble learning. In: *The handbook of brain theory and neural networks* 2 (2002), S. 110–125

LITERATURVERZEICHNIS

- [DBB⁺01] DUGAS, Charles ; BENGIO, Yoshua ; BÉLISLE, François ; NADEAU, Claude ; GARCIA, René: Incorporating second-order functional knowledge for better option pricing. In: *Advances in neural information processing systems* (2001), S. 472–478
- [DHS11] DUCHI, John ; HAZAN, Elad ; SINGER, Yoram: Adaptive subgradient methods for online learning and stochastic optimization. In: *Journal of machine learning research* 12 (2011), Nr. 7
- [Dym21] DYMEL, Tom: Handgestenerkennung mit Entscheidungsbäumen. (2021), 02, S. 1–71
- [Efr92] EFRON, Bradley: Bootstrap methods: another look at the jackknife. In: *Breakthroughs in statistics*. Springer, 1992, S. 569–593
- [Eng18] ENGELHARDT, Sebastian: *Optische Gestenerkennung mit künstlichen neuronalen Netzen für kleine eingebettete Systeme*. 10 2018
- [Ent20a] ENTWICKLER scikit-learn: *1.10.6. Tree algorithms: ID3, C4.5, C5.0 and CART*. <https://scikit-learn.org/stable/modules/tree.html#tree-algorithms>. Version: 2020
- [Ent20b] ENTWICKLER scikit-learn: *1.11. Ensemble methods*. <https://scikit-learn.org/stable/modules/ensemble.html#ensemble>. Version: 2020
- [EUD18] ELFWING, Stefan ; UCHIBE, Eiji ; DOYA, Kenji: Sigmoid-weighted linear units for neural network function approximation in reinforcement learning. In: *Neural Networks* 107 (2018), S. 3–11
- [FS97] FREUND, Yoav ; SCHAPIRE, Robert E.: A decision-theoretic generalization of on-line learning and an application to boosting. In: *Journal of computer and system sciences* 55 (1997), Nr. 1, S. 119–139
- [GBB11] GLOROT, Xavier ; BORDES, Antoine ; BENGIO, Yoshua: Deep sparse rectifier neural networks. In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics* JMLR Workshop and Conference Proceedings, 2011, S. 315–323
- [GEW06] GEURTS, Pierre ; ERNST, Damien ; WEHENKEL, Louis: Extremely randomized trees. In: *Machine learning* 63 (2006), Nr. 1, S. 3–42
- [Gie20] GIESE, Anton: *Compression of Artificial Neural Networks for Hand Gesture Recognition*. 10 2020
- [GL09] GHOSH, Joydeep ; LIU, Alexander: K-Means. In: *The top ten algorithms in data mining* 9 (2009), S. 21–22
- [HH19] HIGHAM, Catherine F. ; HIGHAM, Desmond J.: Deep learning: An introduction for applied mathematicians. In: *SIAM Review* 61 (2019), Nr. 4, S. 860–891
- [HSS12] HINTON, Geoffrey ; SRIVASTAVA, Nitish ; SWERSKY, Kevin: Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. (2012)
- [JCC⁺13] JURDAK, Raja ; CORKE, Peter ; COTILLON, Alban ; DHARMAN, Dhinesh ; CROSSMAN, Chris ; SALAGNAC, Guillaume: Energy-efficient localization: GPS duty cycling with radio ranging. In: *ACM Transactions on Sensor Networks (TOSN)* 9 (2013), Nr. 2, S. 1–33
- [Jia17] JIADONG, Yao: *Performance of Artificial Neural Networks on an AVR Microcontroller*. 2017
- [JLP06] JIN, Guang-yao ; LU, Xiao-yi ; PARK, Myong-Soon: An indoor localization mechanism using active RFID tag. In: *IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC'06)* Bd. 1 IEEE, 2006, S. 4–pp

- [KB14] KINGMA, Diederik P. ; BA, Jimmy: Adam: A method for stochastic optimization. In: *arXiv preprint arXiv:1412.6980* (2014)
- [KBP⁺19] KIM, Chelhwon ; BHATT, Chidansh ; PATEL, Mitesh ; KIMBER, Don ; TJAHHADI, Yulius: InFo: indoor localization using fusion of visual information from static and dynamic cameras. In: *2019 International Conference on Indoor Positioning and Indoor Navigation (IPIN)* IEEE, 2019, S. 1–8
- [ker21] *Keras Dokumentation*. <https://keras.io/about/>. Version: 2021
- [KH05] KAPLAN, Elliott ; HEGARTY, Christopher: *Understanding GPS: principles and applications*. Artech house, 2005
- [KH10] KEMPER, Jürgen ; HAUSCHILD, Daniel: Passive infrared localization with a probability hypothesis density filter. In: *2010 7th Workshop on Positioning, Navigation and Communication* IEEE, 2010, S. 68–76
- [KMK14] KONDA, Kishore ; MEMISEVIC, Roland ; KRUEGER, David: Zero-bias autoencoders and the benefits of co-adapting features. In: *arXiv preprint arXiv:1402.3337* (2014)
- [KSA17] KRUHKAR, Jakub ; SCHWERING, Angela ; ANACTA, Vanessa J.: *Landmark-based navigation in cognitive systems*. 2017
- [Kub19] KUBIK, Philipp: *Zuverlässige Handgestenerkennung mit künstlichen neuronalen Netzen*. 04 2019
- [LF19] LYDIA, Agnes ; FRANCIS, Sagayaraj: Adagrad—An optimizer for stochastic gradient descent. In: *Int. J. Inf. Comput. Sci.* 6 (2019), Nr. 5
- [LM99] LEUPERS, Rainer ; MARWEDEL, Peter: Function inlining under code size constraints for embedded processors. In: *1999 IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers (Cat. No. 99CH37051)* IEEE, 1999, S. 253–256
- [LR76] LAURENT, Hyafil ; RIVEST, Ronald L.: Constructing optimal binary decision trees is NP-complete. In: *Information processing letters* 5 (1976), Nr. 1, S. 15–17
- [MGC⁺96] MENZEL, Randolph ; GEIGER, Karl ; CHITTKA, Lars ; JOERGES, Jasdan ; KUNZE, Jan ; MÜLLER, Uli: The knowledge base of bee navigation. In: *Journal of Experimental Biology* 199 (1996), Nr. 1, S. 141–146
- [MH17] MUKKAMALA, Mahesh C. ; HEIN, Matthias: Variants of rmsprop and adagrad with logarithmic regret bounds. In: *International Conference on Machine Learning* PMLR, 2017, S. 2545–2553
- [MHN13] MAAS, Andrew L. ; HANNUN, Awni Y. ; NG, Andrew Y.: Rectifier nonlinearities improve neural network acoustic models. In: *Proc. icml* Bd. 30 Citeseer, 2013, S. 3
- [Mia21] MIAN, Naveed A.: *Sensor based Location Awareness with Artificial Neural Networks*. 04 2021
- [MP43] MCCULLOCH, Warren S. ; PITTS, Walter: A logical calculus of the ideas immanent in nervous activity. In: *The bulletin of mathematical biophysics* 5 (1943), Nr. 4, S. 115–133
- [Nie] NIELSEN, Michael: Neural Networks and Deep Learning.
- [NIGM18] NWANKPA, Chigozie ; IJOMAH, Winifred ; GACHAGAN, Anthony ; MARSHALL, Stephen: Activation functions: Comparison of trends in practice and research for deep learning. In: *arXiv preprint arXiv:1811.03378* (2018)
- [PE96] PARKINSON, Bradford W. ; ENGE, Per K.: Differential gps. In: *Global Positioning System: Theory and applications*. 2 (1996), S. 3–50

LITERATURVERZEICHNIS

- [PGP98] PEI, M ; GOODMAN, ED ; PUNCH, WF: Feature extraction using genetic algorithms. In: *Proceedings of the 1st International Symposium on Intelligent Data Engineering and Learning, IDEAL* Bd. 98, 1998, S. 371–384
- [PH19] POULOSE, Alwin ; HAN, Dong S.: Hybrid indoor localization using IMU sensors and smartphone camera. In: *Sensors* 19 (2019), Nr. 23, S. 5084
- [PVG⁺11] PEDREGOSA, F. ; VAROQUAUX, G. ; GRAMFORT, A. ; MICHEL, V. ; THIRION, B. ; GRISEL, O. ; BLONDEL, M. ; PRETTENHOFER, P. ; WEISS, R. ; DUBOURG, V. ; VANDERPLAS, J. ; PASSOS, A. ; COURNAPEAU, D. ; BRUCHER, M. ; PERROT, M. ; DUCHESNAY, E.: Scikit-learn: Machine Learning in Python. In: *Journal of Machine Learning Research* 12 (2011), S. 2825–2830
- [PZYH08] PAN, Sinno J. ; ZHENG, Vincent W. ; YANG, Qiang ; HU, Derek H.: Transfer learning for wifi-based indoor localization. In: *Association for the advancement of artificial intelligence (AAAI) workshop* Bd. 6 The Association for the Advancement of Artificial Intelligence Palo Alto, 2008
- [Qui86] QUINLAN, J. R.: Induction of decision trees. In: *Machine learning* 1 (1986), Nr. 1, S. 81–106
- [Qui90] QUINLAN, J R.: Decision trees and decision-making. In: *IEEE Transactions on Systems, Man, and Cybernetics* 20 (1990), Nr. 2, S. 339–346
- [Qui14] QUINLAN, J R.: *C4. 5: programs for machine learning*. Elsevier, 2014
- [QWZ⁺18] QIAN, Kun ; WU, Chenshu ; ZHANG, Yi ; ZHANG, Guidong ; YANG, Zheng ; LIU, Yunhao: Widar2. 0: Passive human tracking with a single wi-fi link. In: *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, 2018, S. 350–361
- [RLF18] ROTH, Mikko ; LUPOLD, Arno ; FALK, Heiko: Measuring and modeling energy consumption of embedded systems for optimizing compilers. In: *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*, 2018, S. 86–89
- [Ros60] ROSENBROCK, H. H.: An Automatic Method for Finding the Greatest or Least Value of a Function. In: *The Computer Journal* 3 (1960), 01, Nr. 3, 175–184. <http://dx.doi.org/10.1093/comjnl/3.3.175>. – DOI 10.1093/comjnl/3.3.175. – ISSN 0010–4620
- [Ros61] ROSENBLATT, Frank: Principles of neurodynamics. perceptrons and the theory of brain mechanisms / Cornell Aeronautical Lab Inc Buffalo NY. 1961. – Forschungsbericht
- [RSF13] ROHMER, E. ; SINGH, S. P. N. ; FREESE, M.: CoppeliaSim (formerly V-REP): a Versatile and Scalable Robot Simulation Framework. In: *Proc. of The International Conference on Intelligent Robots and Systems (IROS)*, 2013. – www.coppeliarobotics.com
- [RZL17] RAMACHANDRAN, Prajit ; ZOPH, Barret ; LE, Quoc V.: Searching for activation functions. In: *arXiv preprint arXiv:1710.05941* (2017)
- [Ses91] SESSLER, GM: Acoustic sensors. In: *Sensors and Actuators A: Physical* 26 (1991), Nr. 1-3, S. 323–330
- [SLCY11] SEIDE, Frank ; LI, Gang ; CHEN, Xie ; YU, Dong: Feature engineering in context-dependent deep neural networks for conversational speech transcription. In: *2011 IEEE Workshop on Automatic Speech Recognition & Understanding* IEEE, 2011, S. 24–29
- [SS18] SADOWSKI, Sebastian ; SPACHOS, Petros: Rssi-based indoor localization with the internet of things. In: *IEEE Access* 6 (2018), S. 30149–30161

- [Ste09] STEINBERG, Dan: CART: classification and regression trees. In: *The top ten algorithms in data mining* 9 (2009), S. 179–201
- [TDS⁺14] TIETE, Jelmer ; DOMÍNGUEZ, Federico ; SILVA, Bruno d. ; SEGERS, Laurent ; STEENHAUT, Kris ; TOUHAFI, Abdellah: SoundCompass: a distributed MEMS microphone array-based sensor for sound source localization. In: *Sensors* 14 (2014), Nr. 2, S. 1918–1949
- [TH09] THOMPSON, Matthew J. ; HORSLEY, David A.: Resonant MEMS magnetometer with capacitive read-out. In: *SENSORS, 2009 IEEE*, 2009, S. 992–995
- [Ven21] VENZKE, Marcus: TODO: Antrag Forschungsprojekt. (2021), 04, S. 1–71
- [VKK⁺20] VENZKE, Marcus ; KLISCH, Daniel ; KUBIK, Philipp ; ALI, Asad ; MISSIER, Jesper D. ; TURAU, Volker: *Artificial Neural Networks for Sensor Data Classification on Small Embedded Systems*. 2020
- [WYM18] WANG, Xuyu ; YU, Zhitao ; MAO, Shiwen: DeepML: Deep LSTM for indoor localization with smartphone magnetic and light sensors. In: *2018 IEEE International Conference on Communications (ICC)* IEEE, 2018, S. 1–6
- [XZYN16] XIAO, Jiang ; ZHOU, Zimu ; YI, Youwen ; NI, Lionel M.: A survey on wireless indoor localization from the device perspective. In: *ACM Computing Surveys (CSUR)* 49 (2016), Nr. 2, S. 1–31
- [YLL⁺15] YANG, Lei ; LIN, Qiongzheng ; LI, Xiangyang ; LIU, Tianci ; LIU, Yunhao: See through walls with COTS RFID system! In: *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, 2015, S. 487–499
- [ZGL19] ZAFARI, Faheem ; GKELIAS, Athanasios ; LEUNG, Kin K.: A survey of indoor localization systems and technologies. In: *IEEE Communications Surveys & Tutorials* 21 (2019), Nr. 3, S. 2568–2599
- [ZLGZ16] ZHANG, Guojun ; LIU, Mengran ; GUO, Nan ; ZHANG, Wendong: Design of the MEMS piezoresistive electronic heart sound sensor. In: *Sensors* 16 (2016), Nr. 11, S. 1728