Masterarbeit

Computer Science

Sensorbasierter Orientierungssinn mit künstlichen neuronalen Netzen und Entscheidungsbäumen

von

Tom Dymel

Juli 2021

Betreut von

Dr. Marcus Venzke
Institute of Telematics, Hamburg University of Technology

Erstprüfer	Prof. Dr. Volker Turau Institute of Telematics Hamburg University of Technology
Zweitprüfer	Dr. ToDo Institute of Telematics Hamburg University of Technology

Institut für Telematik
Technische Universität Hamburg
TUHH

Inhaltsverzeichnis

Kapitel 1

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.

GPS bedarf aber direkten Kontakt zu den Satelliten, ansonsten ist die Signalstärke zu den Satelliten extrem eingeschränkt. 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.

In dieser Arbeit wird die diskrete Positionsbestimmung basierend auf Sensordaten untersucht. Dabei soll eine bestimmte Anzahl an Orten anhand verschiedener Sensorwerten unterschieden werden. Dies ist vergleichbar mit dem Orientierungssinn von Tieren und Menschen. Zum Beispiel navigieren Honigbienen auf Basis von gelernten Orientierungspunkten, um Nahrungsquelle und Nest zu finden [MGC+96]. 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 keine Infrastruktur muss aber für jedes Gebäude individuell trainiert werden.

TODO: Was wurde in dieser Arbeit gemacht.
TODO: Was unterscheided diese Arbeit von anderen Arbeiten?

Kapitel 2 führt Entscheidungsbäume und Ensemble-Methoden ein. Kapitel 3 führt künstliche neuronale Netze (KNN) ein mit dem Fokus auf Feed Forward neuronale Netze (FFNN). In Kapitel 4 wird auf den Stand der Forschung für Standortbestimmung mit Hilfe von maschinellen Lernen (ML) eingegangen. Die Generierung von Trainings- und Validationsdaten auf Basis von Simulationen wird in Kapitel 5 erläutert. Kapitel 6 stellt die trainierten ML Modelle mit Entscheidungsbäumen und FFNN vor. 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.

Kapitel 2

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 fortgefahren 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.

Relevant ist jedoch lediglich der Klassifizierer, da in dieser Arbeit ein Klassifizierungsproblem gelöst werden soll. Dieser bietet zahlreiche Hyperparameter an, um die Konstruktion des Entscheidungsbaumes zu steuern. In dieser Arbeit wird der Hyperparameter `max_depth` verwendet. Dieser Hyperparameter beschränkt die maximale Baumhöhe und begrenzt somit den Programmspeicherverbrauch.

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

2.2 Einzelne Entscheidungsbäume

Der einzelne 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 fortgefahren 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

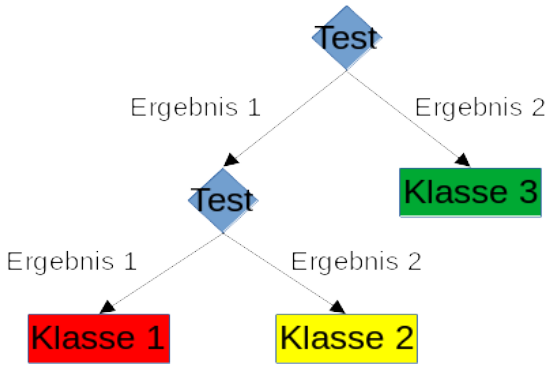


Abbildung 2.1: Beispiel eines binären Entscheidungsbaums mit 3 möglichen Ergebnissen.

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 Entscheidungsbaumes 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* (**C**lassification **A**nd **R**egression **T**rees) Algorithmus [Ent20a]. CART partitioniert die Trainingsmenge indem lokal immer die beste Teilung ausgewählt wird, d. h. es wird für die momentane Teilmenge immer die beste Teilungsregel ausgewählt. 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 $H(x) = w_1h_1(x) + ... + w_Kh_K(x)$ ist eine Möglichkeit die Einzelergebnisse $\{h_1, ..., h_K\}$ gewichtet mit $\{w_1, ..., w_K\}$ 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. Andererseits 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 (**B**ootstrap **agg**regat**ing**) 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.

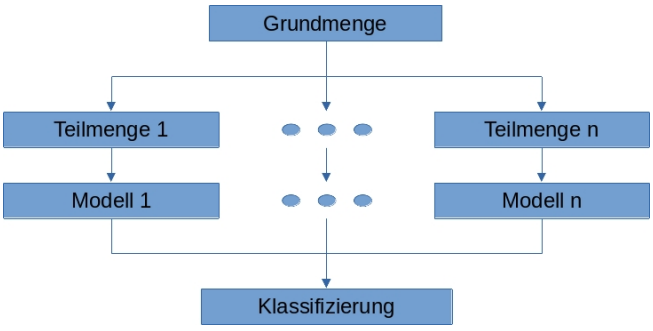


Abbildung 2.2: Klassifizierungsprozess mit der Bagging-Methode.

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

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 `AdaBoost` [FS97] von Freund und Schapire.

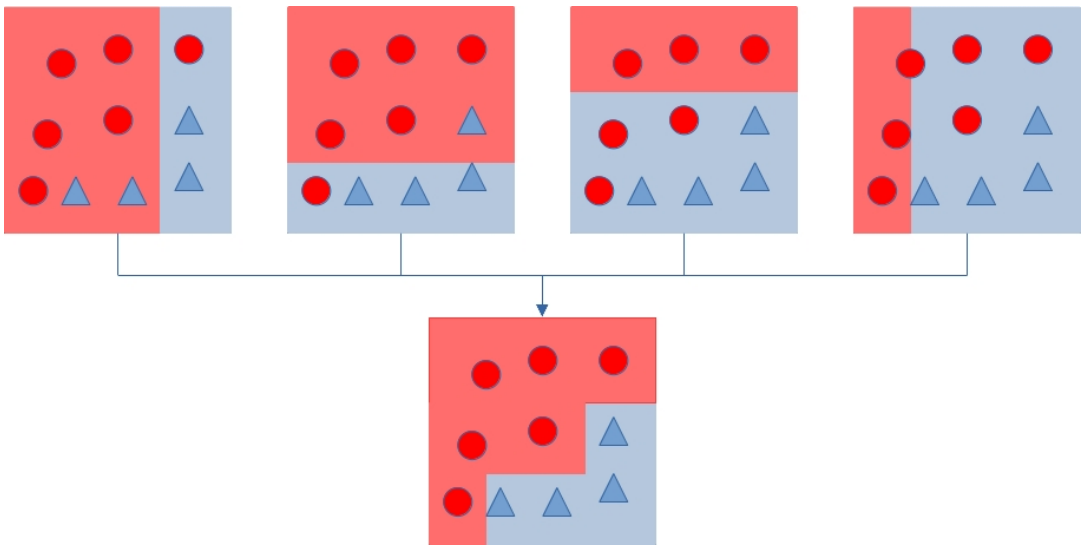


Abbildung 2.3: Klassifizierungsprozess mit der Boosting-Methode.

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 verschidenen 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.

2.5.1 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 Entscheidungsbaumes 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

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.

Kapitel 3

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 unterscheided 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]. Gleichung 3.1 stellt diesen Zusammenhang dar.

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

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.

Das künstliche neuronale Netz approximiert eine arbiträre Funktion f^*. Dazu findet es eine Menge von Parametern $\boldsymbol{\theta}$, wodurch $f^*(\mathbf{x}) \approx f(\mathbf{x}, \boldsymbol{\theta})$ möglichst gut von der Approximations-

funktion f abgebildet wird [BGC17].

Das KNN ist in Schichten organisiert. Analog zu den biologischen Neuron, 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 Gleichung 3.1, stellt Gleichung 3.2 die Aktivierung einer Schicht dar [HH19].

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

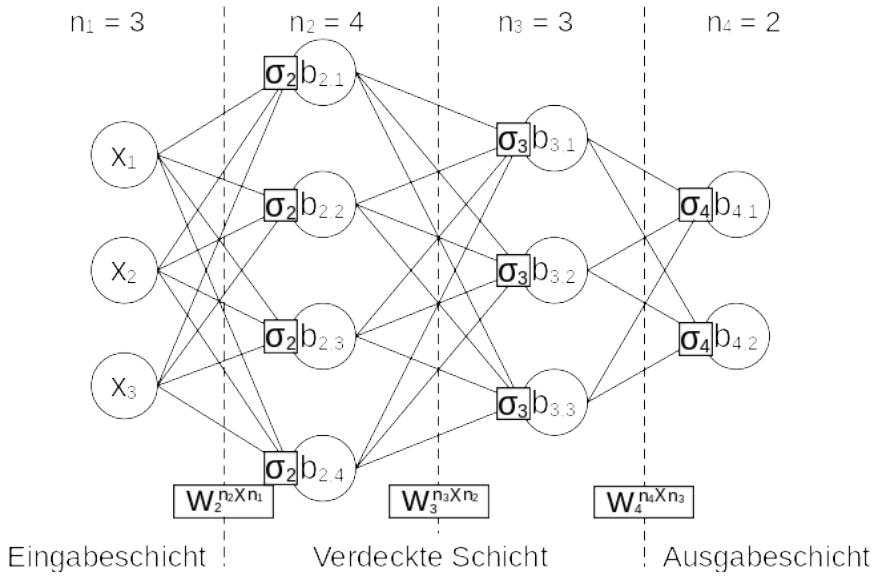


Abbildung 3.1: Beispiel eines FFNN mit vier Schichten und einer binären Ausgabeschicht.

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. Gleichung 3.3 zeigt, wie das allgemeine KNN mit einer rekursiven Funktion modelliert werden kann.

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

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 (**T**ensor **P**rocessing **U**nit), 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 einiger 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 $\boldsymbol{\theta}$ 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.

Abhängig von der Strategie des Lernalgorithmuses 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 $\boldsymbol{\theta}$ des KNN aktualisiert werden [HH19]. In dieser Arbeit wird ADAM verwendet, da dieser Algorithmus die Vorteile von *(S)GD* (**S**tochastic **G**radient **D**escent) mit Momentum und *RMSprop* vereinigt [KB14, HH19].

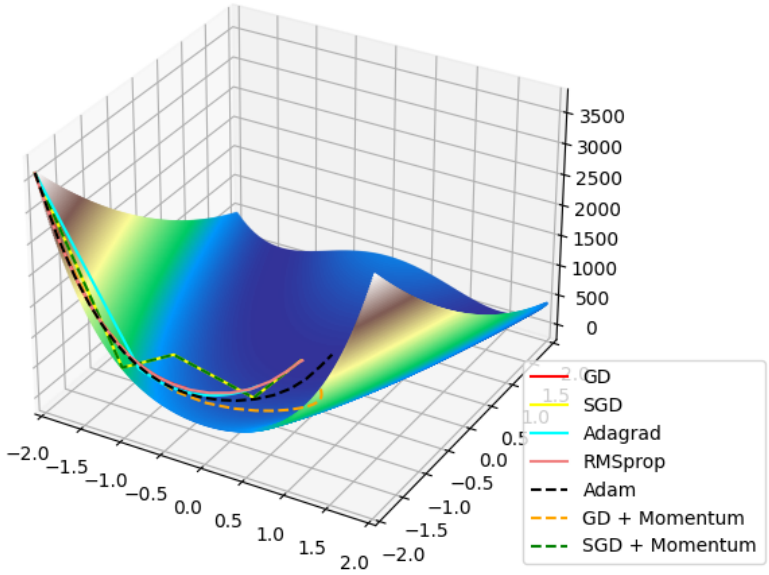


Abbildung 3.2: Verschiedene Lernalgorithmen auf der Rosenbrock-Funktion [Ros60].

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.

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

Gleichung 3.4 illustriert diesen iterativen Prozess für Minimierung im eindimensionalen Fall, 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.

Im mehrdimensionalen Fall 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. Gleichung 3.5 zeigt die iterative Berechnung im mehrdimensionalen Fall.

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \bigtriangledown C(\boldsymbol{\theta}_k)\eta \tag{3.5}$$

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 Approximierung ist eine Laufzeitverbesserung.

(S)GD mit Momentum 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]. Gleichung 3.6 zeigt die iterative Berechnung.

$$\mathbf{v}_{-1} = \mathbf{0}, \qquad \mathbf{v}_k = \mathbf{v}_{k-1}\gamma + \bigtriangledown C(\boldsymbol{\theta}_k), \qquad \boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \mathbf{v}_k\eta \tag{3.6}$$

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.

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].

Gleichung 3.7 zeigt, wie sich iterativ die Lernrate antiproportional zur kummulierten 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.

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

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].

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

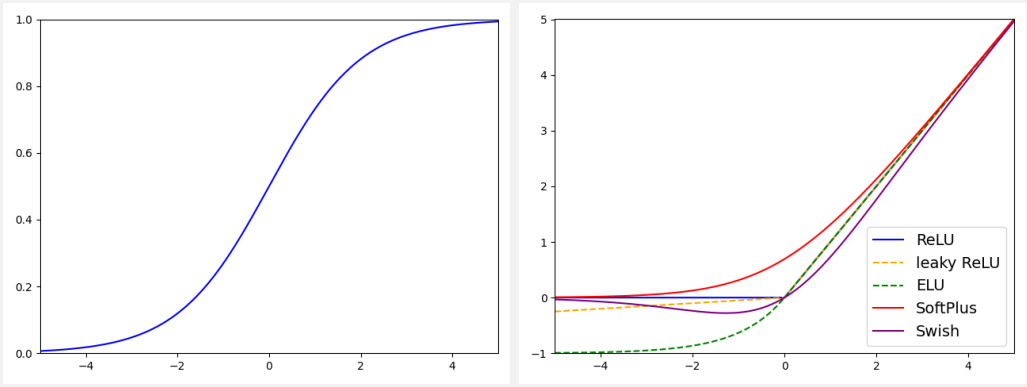
Gleichung 3.9 zeigt, wie Adam RMSprop und (S)GD mit Momentum vereint, wobei $\gamma_1 < \gamma_2 < 1$ [KB14] ist. Adam beinhaltet RMSprop und hat eine Momentum Komponente, indem das Moment der ersten Ordnung mit exponentieller Gewichtung approximiert wird [BGC17]. Zudem korrigiert Adam den Bias das Momente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

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

3.4 Aktivierungsfunktionen

Die Aktivierungsfunktion entscheided 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



(a) Sigmoid/SoftMax (b) ReLU Varianten

Abbildung 3.3: Verschiedene Aktivierungsfunktionen.

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.

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

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

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.

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

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.

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

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

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

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

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.

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

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 $n_1 n_2 + n_2 n_3$ 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 Re-

gister 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 hardwareuntersützten 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.

Kapitel 4

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 der Infrastruktur bestimmt wird.

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

4.1 Indoor- und Outdoor-Lokalisierung

GPS ist eine weit verbreites Positionsbestimmungssystem 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 Nutzer, da es ein Device-Based Ansatz ist. Die Geräte berechnen aus den empfangenden Signalen von mehreren Satelliten ihre Position. Dabei ist die Positionsbestimmung bis zu 5 m akkurat [SS18]. Das GPS Modul ist aber sehr teuer und bedarf viel Energie. Außerdem kann es im Indoor-Kontext nicht eingesetzt werden, da die Signalstärke zu den Satelliten stark beeinträchtigt ist. Aus diesem Grund ist es für Mikrocontroller im Indoor-Bereich ungeeignet.

Indoor-Lokalisierung bedarf meist eine hohe 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. Beispielsweise können visuelle Features mit der Kamera extrahiert werden, RSS Messungen des WiFi Netzwerkes durchgeführt werden, oder Bewegungs- und Lichtsensoren verwendet werden. Der Vorteil sind die geringen Infrastrukturkosten.

Device-Free Ansätze bedürfen Infrastruktur, um Objekte im Interessebereich wahrzunehmen. Dafür können zum einen TOA (**T**ime **O**f **A**rrival) basierte System verwendet werden, d. h. die Position wird über die Reflektion von Radio- oder Schallwellen ermittelt. Zum Anderen kommen auch *Tag*-Systeme zum Einsatz, in denen die RFID-Tags des Objektes an vorinstallierten Orten gelesen wird. Dadurch kann die Position approximiert werden. Dabei ist die Auflösung abhängig von der Verteilung und Anzahl der lesenden Geräte im Interessebereich.

4.2 WiFi basierte Indoor-Lokalisierung mit Transfer Lernen

Pan et al. untersuchten Indoor-Lokaliserung basierend auf WiFi RSS Daten [PZYH08]. *Received Signal Strength* (RSS) wird von dem Empfänger von mehreren Sendern gemessen. Dadurch steht ein Vektor von Signalstärken zur Verfügung aus denen die Position approximiert werden kann.

Sie bemerken, 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.

Beispielsweise können sich Daten über die Zeit ändern, 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 erheblichen Kalibrierungsaufwand der ML Modelle.

Die Autoren 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.

4.3 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 zu einer bimodalen Abbildung. 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 (**C**hannel **S**tate **I**nformation) 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. 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 Senario 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.

4.4 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 verschieden 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 über fünf Stichproben über den Median geglättet. Aus den resultierenden Sensorwerten wird die Veränderung zum vorherigen Sensorwert für jeden Sensor ermittelt. Daraus wird der Betrag gebildet, welches als Feature für jeden Sensor jeweils verwendet wird. Zudem werden die zuletzt besuchten Standorte in Form einer exponentiell fallenenden Funktion über Zeit als weiteres Feature hinzugefügt. 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 konkatiniert werden.

Mit dieser Eingabe trainierte Mian ein FFNN mit einer Rückwärtskante, um den zuletzt bestimmten Standort als Feature 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 validierte sein Modell auf einer Testmenge die aus sechs distinkten Standorten besteht über fünf Zyklen. Mit einem Datenfenster ab 25 konnten Klassifizierungsgenauigkeiten von bis zu 96% erreicht werden.

Kapitel 5

ML-Modelle

In dieser Arbeit wird die Device-Based Indoor-Lokalisation auf Basis von Sensorwerten untersucht. Dieser 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 von dem 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. Im Vergleich zu der Architektur von Mian [Mia21] können die Klassifizierungsergebnisse bei der Feature-Extrahierung weiter verarbeitet werden.

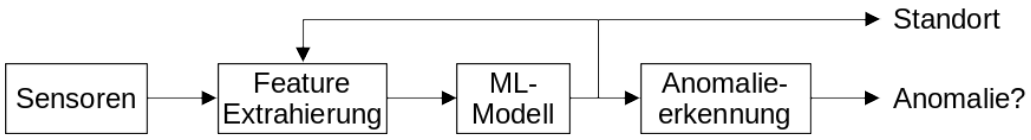


Abbildung 5.1: Architektur des verfolgten Ansatzes.

In dieser Arbeit werden Entscheidungsbaum basierte Klassifizierer mit KNN verglichen, insbesondere den von Mian verwendeten Ansatz mit FFNN. Entscheidungsbäume sind deutlich effizienter als KNN, weswegen diese bei vergleichbarer Klassifizierungsgenauigkeit und Fehlertoleranz die preferierte Wahl sind.

5.1 Standortenkodierung

Als Standort wird ein einzigartiger diskreter Ort im Indoor-Szenario bezeichnet. Bei der Klassifizierung können Standorte auf verschiedenen Arten im ML-Modell enkodiert werden.

Mian enkodierte die Pfade zwischen Interessepunkten als Standorte, d. h. das Klassifizierungsergebnis ist der Pfad auf dem sich der Mikrocontroller befindet [Mia21].

Daneben wurden in dieser Arbeit noch zwei weitere Ansätze untersucht. Im ersten Ansatz werden alle Punkte im Umkreis von bestimmten Interessepunkten als Standorte definiert und alle restlichen als *unbekannten Standort*. Der zweite Ansatz kombiniert die beiden anderen Ansätze. Zunächst werden, wie im ersten Ansatz, alle Punkte im Umkreis von bestimmten Interessepunkten als Standorte definiert. Dann werden die übrigen Punkte, also die Pfade zwischen zwei Standorten, als Standorte definiert.

Der Zusammenhang dieser Ansätze wird deutlich, wenn man eine Route als zyklischen Graphen betrachtet, wie in Abbildung 5.2. Einzigartige diskrete Interessepunkte, bilden die Knoten und die Pfade zwischen diesen Punkten, die Kanten. Mian's Ansatz nutzt nur die Kanten und folglich die anderen Ansätze jeweils die Knoten und die Knoten und Kanten.

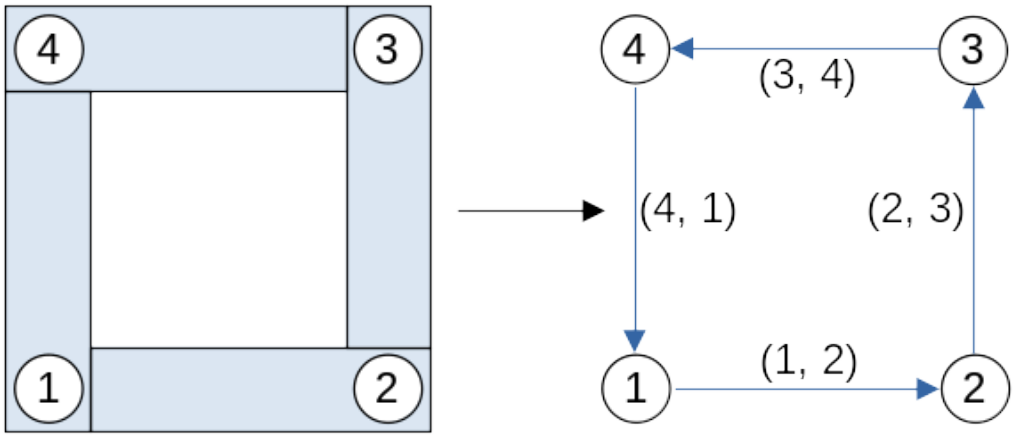


Abbildung 5.2: Standortenkodierung der Orte und Pfade.

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 enkodiert soviele Standorte wie beide Ansätze zusammen, wodurch dieser Ansatz am komplexisten 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.

Bei der Evaluation in Kapitel 7.4 hat sich gezeigt, dass Anomalien besser erkannt werden können, wenn die Auflösung des Enkodierungsansatzes hoch ist.

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 verfasst wird, sind die Limitierungen des Mikrocontrollers noch nicht bekannt. Für gewöhnlich sind die 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.

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 normalisert. Die Größe der Ausgabeschicht ist die Anzahl der verschieden 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 kategorische Enkodiert, 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 *kategorische 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 ist unterteilt in Partitionen basierend auf deren Zyklusbeschriftung. 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 beschriftet sind. Zunächst werden 50% der Partition *i* vom ML-Modell beschriftet, bis beim 13. Zyklus schließlich 100% beschriftet wird. Die Beschriftung ist zufällig, damit der Klassifizierungsfehler auf allen Teilstücken der Route gelernt werden kann.

Partition i:	1 - 5	6 - 15	16 - 20
Phase:	Aufwärmen	Beschriftung mit Modell	Validieren
Beschriftet in %:	0	$100 \bullet \min(1, 0.5 + (i - 6)^2 \div 128)$	-

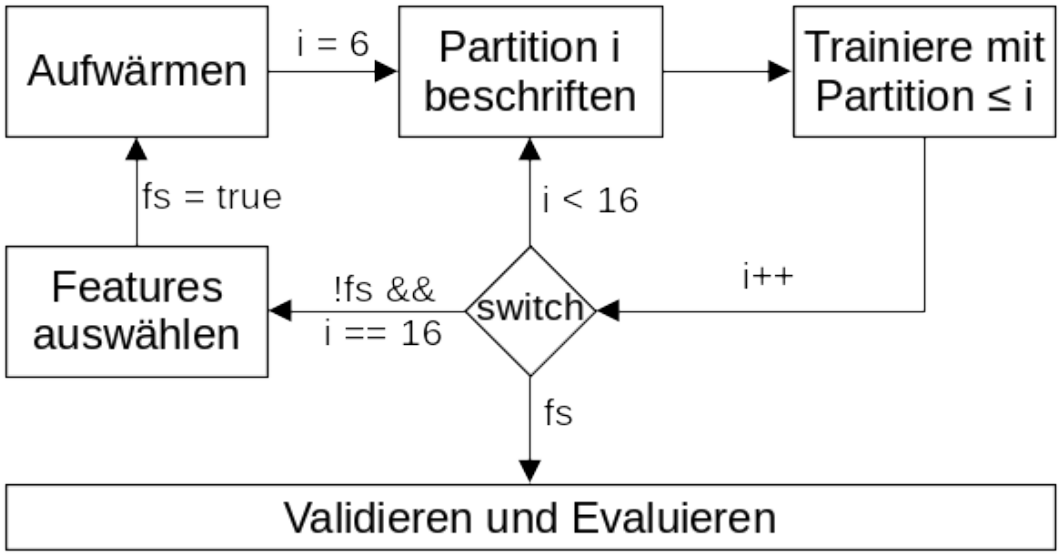


Abbildung 5.3: Der Trainingsablauf des verfolgten Ansatzes.

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. 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 das Objekt 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 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 seperat 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 verschieden 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 6.5 detailiert erläutert werden.

Kapitel 6

Trainings- und Validationsdaten

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- und Validationsdaten unterteilt.

6.1 Simulierte Sensordaten

Insgesamt wurden vier Routen über 20 Zyklen erfasst. Dabei wurden alle 50 ms die xyz-, Koordinaten, Beschleunigung 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.

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 die Datensätze mit deren 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 Routenabschnitt auf den Wert der Markierung gesetzt. Dabei werden alle aufgenommenen Datensätze immer mit dem derzeitigen Wert für den Routenabschnitt markiert.

Neben „simple_square “ gibt es noch drei weitere Routen (siehe Abbildungen B.1, B.2

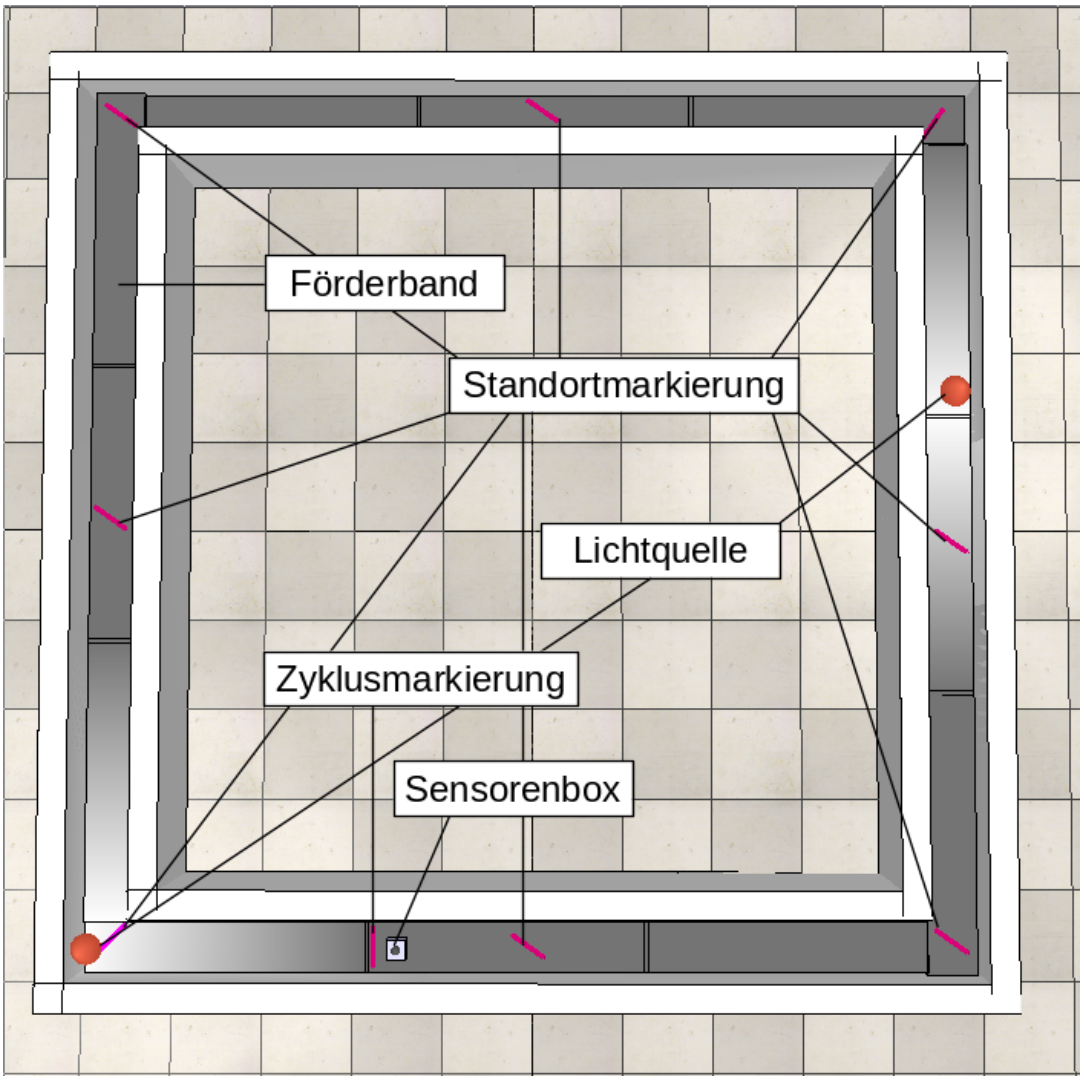


Abbildung 6.1: Modell der Route „simple_square“ in CoppeliaSim mit Beschriftungen.

und B.3). 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 beschriftet ist, d. h. Datensätze mit der gleichen Standortbeschriftung, die sich nicht im

Umkreis des initialen Datensatzes befinden, gelten nicht als dieser Standort. Die Knoten werden mit dem Wert der Standortbeschriftung beschriftet. Die übrigen Datensätze werden entweder als unbekannten Standort beschriftet oder erhalten einen diskreten Standortwert, der die Beziehung der Kante zwischen zwei Knoten enkodiert.

6.2 Künstlichen Sensordaten

- Motivation: Warum ist das nötig?

6.2.1 Magnetfeld

- Welchen Sensor spiegelt das wieder?
- Wie funktioniert das Modell?
- Was und Wie wurden Daten ergänzt?

6.2.2 Temperatur

- Welchen Sensor spiegelt das wieder?
- Wie funktioniert das Modell?
- Was und Wie wurden Daten ergänzt?

6.2.3 Lautstärke

- Welchen Sensor spiegelt das wieder?
- Wie funktioniert das Modell?
- Was und Wie wurden Daten ergänzt?

6.2.4 WLAN Zugangspunkte

- Welchen Sensor spiegelt das wieder?
- Wie funktioniert das Modell?
- Was und Wie wurden Daten ergänzt?

6.3 Simulation von Interrupts

- Motivation => Energieverbrauch, Spiegelung der echten Datenaufnahme, Reduzierung der Trainingsdaten
- Wie funktioniert ist?
- Wie und Wann bei der Datenverarbeitung wird es gemacht?
- So wird bei einer Idle Box auch kein Interrupt erzeugt. (Sneaky Beispiel)
- Vorteile bei der Trainingszeit, da weniger Trainingsdaten
- Wie gut repräsentiert ist jeder Standort danach? => Graph vorher vs. nachher mit training sampling rate
- => Problem: Locations können einfach verpasst werden => Sampling rate?
- => Damit alle locations ausreichend in den Trainingsdaten repräsentiert sind, wird eine training sampling rate eingeführt

6.4 Feature-Extrahierung

- Datenfenster: Realzeit vs. Diskret mittels Wakeups => Es werden immer die letzten 3 Behalten über die die Werte geglättet werden
- Relevanz von Zeit => Interrupts, Zeit als Feature, Feature können Zeit Abhängig und Unabhängig sein
- Welche Feature werden genutzt? => Abhängig von Feature Importance und wie günstig zu berechnen
- Wie werden diese extrahiert?
- Welchen Mehrwert verschaffen diese Features?
- Welchen Einfluss haben Sie im Hinblick auf Ressourcenbedarf(?), Klassifizierungsgenauigkeit(?), Fehlertoleranz(?)
- Rede über Feature Importance, insb. über Permutation Importance => Wie nützlich ist ein Sensor?
- Diskrete Distinct Location wird außerhalb verarbeitet => Modell verändern, dass die prev loc in Feature Processing step mit rein geht?

- Previous Location, Prev. Distinct Location vs. exponentiell abfallene letzte Orte diskutieren

6.5 Fehlerhafte/Anomalie Daten

- Warum und Wieso?
- Welche Fehlerdaten werden eingebaut, was ist deren Begründung?
- Was ist eine Anomaly?
- Welche Anomalydaten wurden eingebaut?
- Welche Features werden zur Anomalierkennung extrahiert? Wie werden sie extrahiert? Wie funktioniert das in der Praxis?
- window deviation zu no anamoly data vs normal deviation zum avg

6.6 Aufteilung der Daten

- Kurz und knapp wie und warum werden die Daten aufgeteilt. => Zyklen
- Sollten Trainingsdaten um synthetische Daten ergänzt werden?
  - Fault Daten, um das Modell Robuster zu machen
  - Synthetische Routen => Was ist das? Wie werden sie erzeugt?
- 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?)

Kapitel 7

Evaluation

TODO: Erkläre irgendwo die Metriken, die verwendet werden

7.1 Klassifizierungsgenauigkeit

- Metriken
- Wie groß ist die Wahrscheinlichkeit, dass die Orte korrekt erkannt werden?
- Entscheidungsbaum vs KNN
- Skalierung mit Anzahl der Orte
- 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
- 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?
- Werden mehr Trainingsdaten benötigt mit steigender Ort Anzahl? Wenn ja wie viel? (Hier oder bei ML-Modell Training)
- Irgendwo drüber reden wie wir evaluieren, also continued predict vs predict und die ganzen Testsets

7.2 Fehlertoleranz

- Irgendwas über Robustheit
- Metriken => Insbesondere continued_predict hier erwähnen
- 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? => Sezenarien enumiereren, 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?

7.3 Ressourcennutzung

- Metriken(?)
- Entscheidungsbaum Ausführung
- FFNN Ausführung
- Feature extrahierung
  - Verhältnis von Kosten zu Nutzen
- Daten Sammlung => Interrupts (Wakeups)
- Einfluss von einzelnen Features für den Ressourcenverbrauch(?)
- Braucht man mehr Neuronen/Hidden Layer mit steigender Ort Anzahl? (Hier oder bei ML-Modell FFNN)
- Irgendwo über den impact von Interrupts reden. Bei Simulationsdaten konnte damit bis zu 95% der Events eingespart werden.
- Wie viel Speicherverbrauch spart man durch die Feature-Selection zusätzlich ein?

7.4 Anomalieerkennung

- 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 Confididence, Fraction Zero
- Sag Motivation, warum diese Metriken (Indikatoren)
- Beispiele
- Wie zuverlässig können Anomalien erkannt werden?
- Vergleich mit Coin-Toss, True und False

Kapitel 8

Diskussion

- Daten Generation über simulierte Teilstücke für gezielte Szenarien generierung. => Mehr Pfade für gleiche Anzahl der Orte dann möglich, Anzahl der Orte beliebig hoch skalierbar => Bessere Aussagen über Performance zu Orten und Pfaden
- Bei Entscheidungsbäume kann man das Cherry-Pickung über verschiedene Ensemble Methoden erweitern
- Man könnte RNN untersuchen, diese sollten sehr gut für diese Aufgabe geeignet sein. Außerdem spielt Evaluierungszeit nicht wirklich eine Rolle, solange es nicht häufig evaluiert wird. Es wäre interessant zu sehen, wie gut man werden kann.

Kapitel 9

Schlussfolgerungen

TODO

- 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 könenn Anomalien erkannt werden?
- Kurze Zusammenfassung wichtigster Dinge
- Zukünftige Arbeit
- Future Work:
  - Weitere Optimierung der Modelle damit Größe und Laufzeit verringert werden kann, e. g. greedy Verkleinerung guter Modelle oder GA für die Suche guter Parameter

Anhang A

Inhalt des USB-Sticks

Bildanhänge

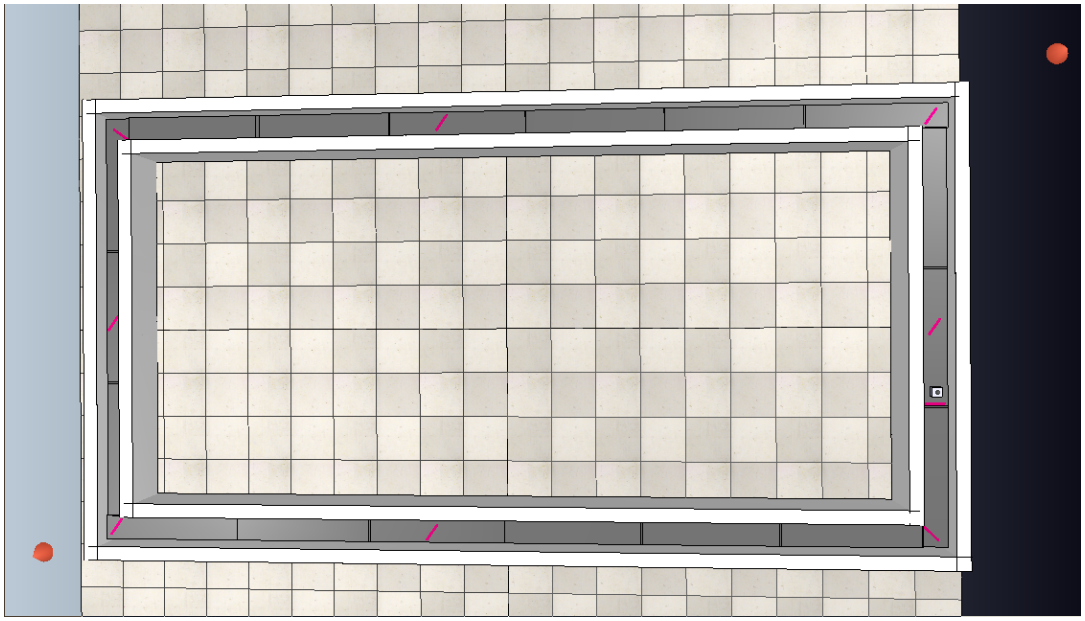


Abbildung B.1: Modell der Route „long_rectangle“ in CoppeliaSim.

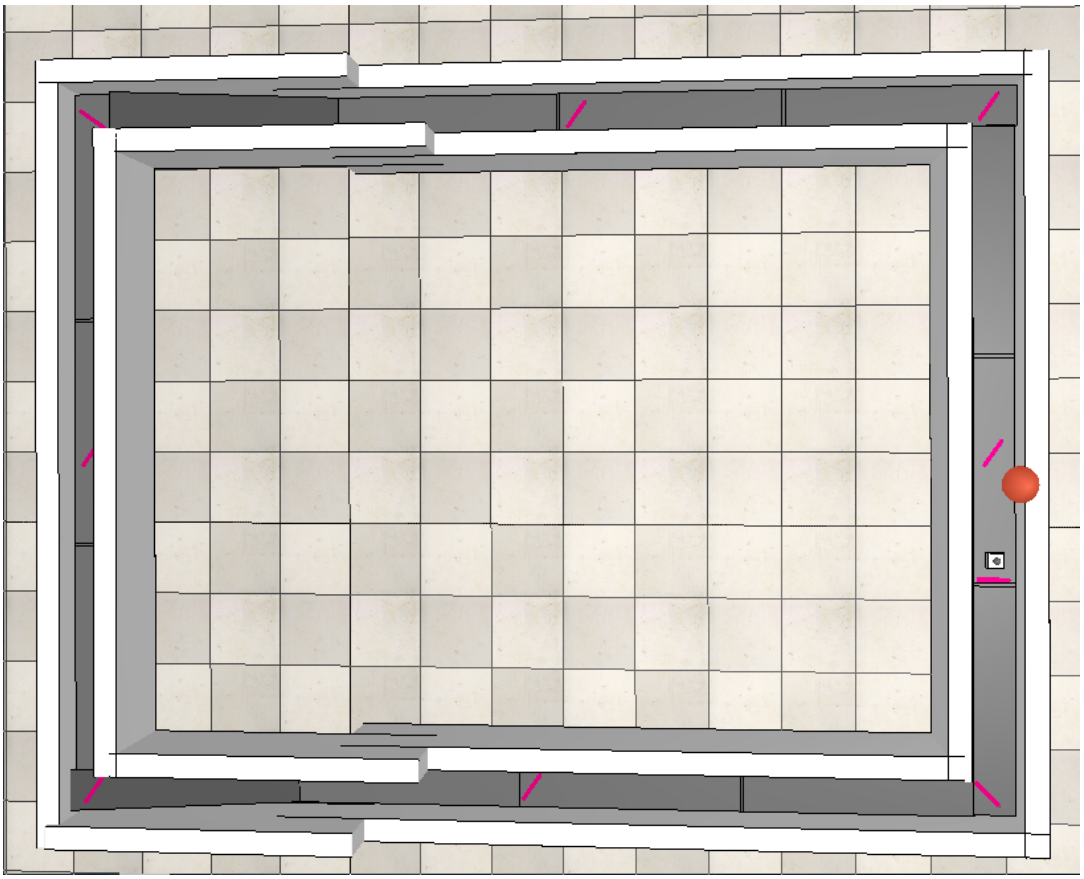


Abbildung B.2: Modell der Route „rectangle_with_ramp“ in CoppeliaSim.

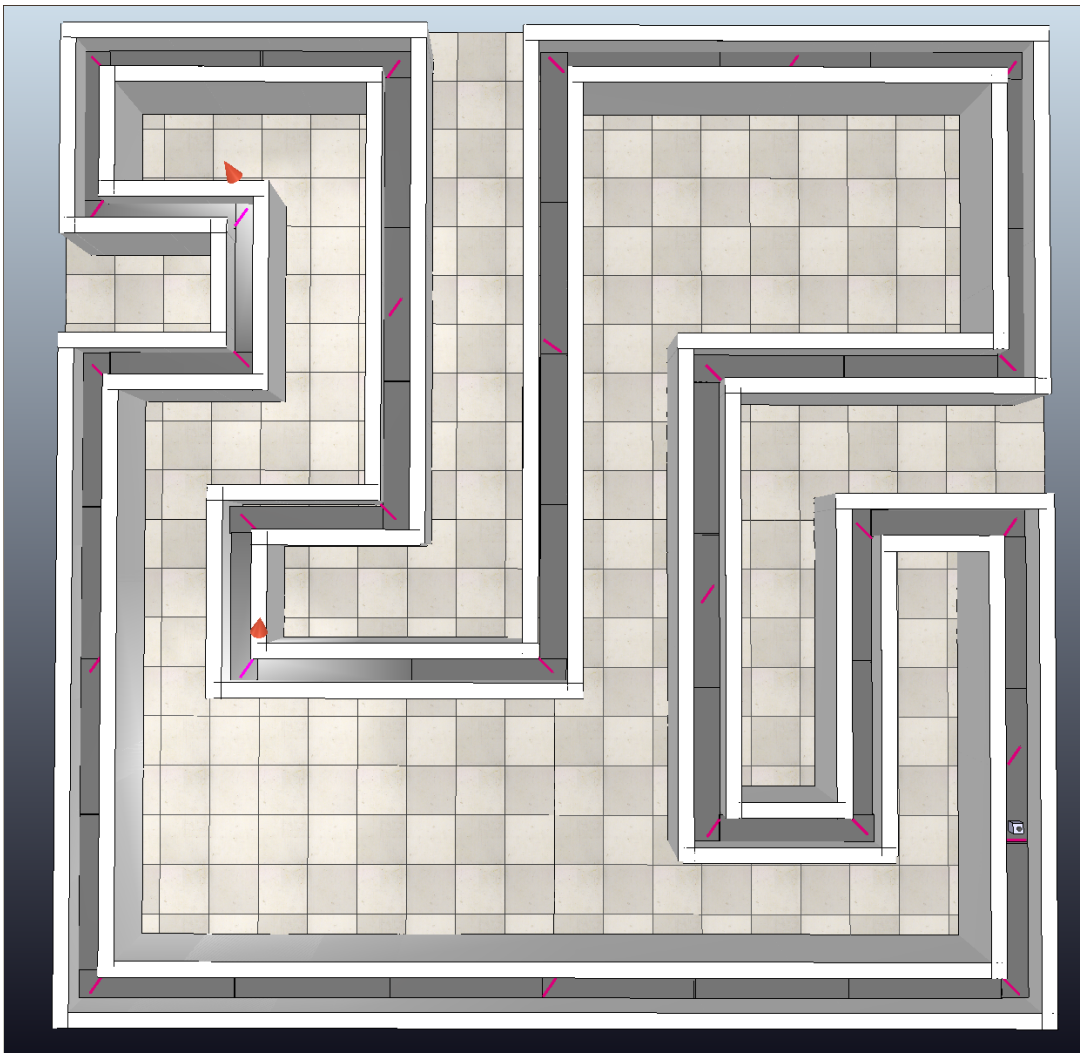


Abbildung B.3: Modell der Route „many_corners“ in CoppeliaSim.

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

[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

[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

[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)

[Jia17] JIADONG, Yao: *Performance of Artificial Neural Networks on an AVR Microcontroller*. 2017

[KB14] KINGMA, Diederik P. ; BA, Jimmy: Adam: A method for stochastic optimization. In: *arXiv preprint arXiv:1412.6980* (2014)

[ker21] *Keras Dokumentation*. `https://keras.io/about/`. Version: 2021

[KH05] KAPLAN, Elliott ; HEGARTY, Christopher: *Understanding GPS: principles and applications*. Artech house, 2005

[KMK14] KONDA, Kishore ; MEMISEVIC, Roland ; KRUEGER, David: Zero-bias autoencoders and the benefits of co-adapting features. In: *arXiv preprint arXiv:1402.3337* (2014)

[Kub19] KUBIK, Philipp: *Zuverlässige Handgestenerkennungmit 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, Randolf ; 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)

[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

[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

[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)

[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

[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

[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