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

Lokalisation is der Prozess die Position von einem Objekt zu bestimmen. Die Positionsbestimmung ist integral für viele technische System, z. B. Tracking, Navigation oder Überwachung. Ein sehr bekanntes und vielseitig genutztes Positionsbestimmungssystem ist das Global Positioning System (GPS). GPS trianguliert die Position des anfragenden Geräts mit Hilfe von mehreren Satelliten (TODO Quelle). Im freien ist eine Genauigkeit von X(TODO) m möglich. Für die Positionsbestimmung innerhalb von Gebäuden ist die Genauigkeit aber nicht ausreichend, außerdem wird sie erschwert durch die dicke Wände und Interferenzen. Dadurch ist GPS oft nicht ausreichend für Trackingsysteme innerhalb von Gebäuden, z. B. Lagerhallen. Aus diesem Grund werden andere Systeme für diesen Zweck verwendet. Je nach Bedarf der Genauigkeit können Objekte mit Sendern, RFID Tags oder Barcodes markiert werden (TODO Quellen). Diese Ansätze bedürfen eine Infrastruktur, die in den Gebäuden installiert und gewartet werden muss.

In dieser Arbeit wird die diskrete Positionsbestimmung basierend auf Sensordaten untersucht. Dabei soll eine bestimmte Anzahl an Orten anhand von verschiedenen 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 (TODO Quelle). 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.

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

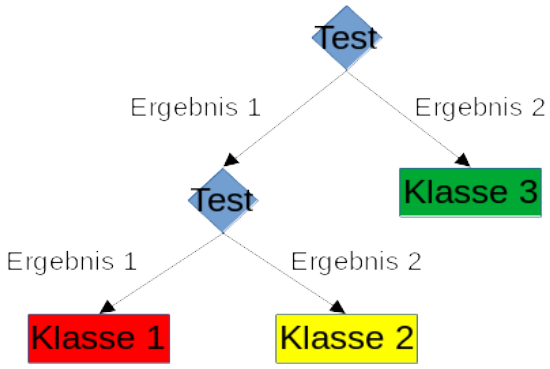

Abbildung 2.1: Beispiel eines binären Entscheidungsbaums mit 3 möglichen Ergebnissen.

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.

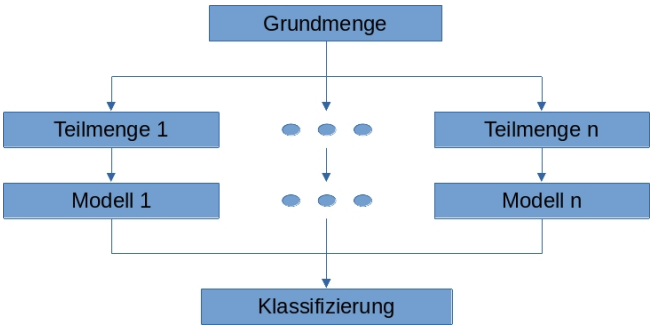

Abbildung 2.2: Klassifizierungsprozess mit der Bagging-Methode.

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.

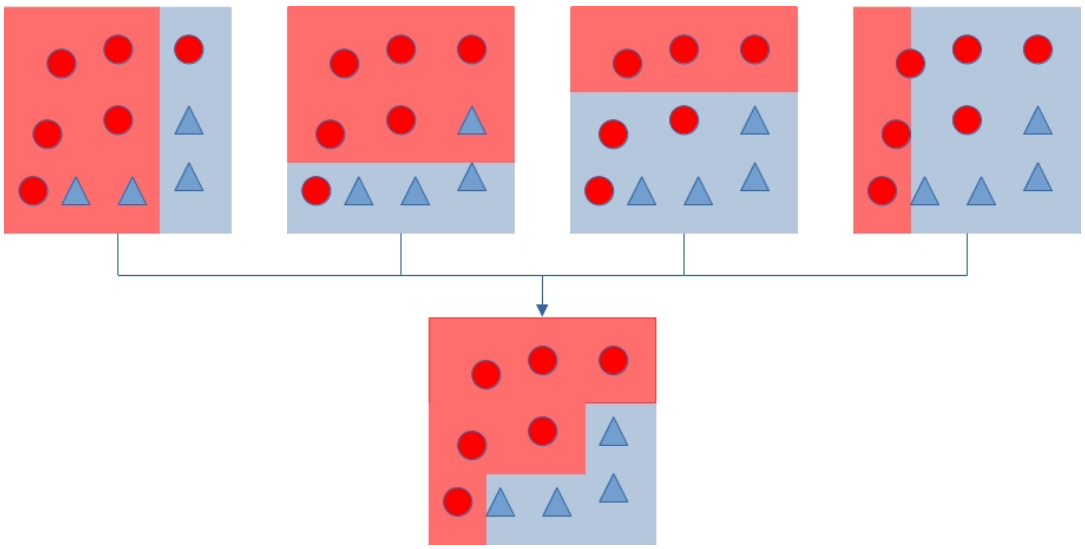

Abbildung 2.3: Klassifizierungsprozess mit der Boosting-Methode.

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 finden hätten 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. Man unterscheided beim biologischen Neuronen zwischen *Afferent*-Neuronen, *Efferent*-Neuronen und *Inter*-Neuronen (TODO: Quelle). 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 (TODO: Quelle). Gleichung 3.1 stellt diesen Zusammenhang dar.

$$y = a(\mathbf{w}^\intercal \mathbf{x} + b) \tag{3.1}$$

Die Vergleichsoperation ist die *Aktivierungsfunktion* $a : \mathbb{R} \mapsto \mathbb{R}$, die in diesem Fall ist es die Stufenfunktion (TODO: Quelle). Die Eingabe $\mathbf{x} \in \mathbb{R}^n$ wird mit $\mathbf{w} \in [0,1]$ 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 Approximationsfunktion f abgebildet wird.

Das KNN ist in Schichten organisiert. Analog zu den biologischen Neuron, gibt es eine *Eingabeschicht (engl. Input-Layer)*, *Ausgabeschicht (engl. Output-Layer)* und *verdeckte Schicht (engl. Hidden-Layer)* (TODO: Quelle). Analog zur Aktivierung eines einzelnen Neuronen,

dargestellt in Gleichung 3.1, stellt Gleichung 3.2 die Aktivierung einer Schicht dar.

$$\mathbf{a}_{i+1} = a_i(\mathbf{z}_i), \qquad \mathbf{z}_i := \mathbf{W}_i\mathbf{a}_i + \mathbf{b}_i \tag{3.2}$$

Das allgemeine KNN verfügt über $m \in \mathbb{N}$ Schichten. Jede Schicht i verfügt über $n_i \in \mathbb{N}$ Neuronen. Die Aktivierungsfunktion $a_i : \mathbb{R}^{n_{i+1}} \mapsto \mathbb{R}^{n_{i+1}}$ berechnet die Aktivierung mit der gewichteten Summe $\mathbf{z}_i$. Die gewichtete Summe setzt sich zusammen aus der Aktivierung der vorherigen Schicht $\mathbf{a}_i$ die mit $W_i \in \mathbb{R}^{n_{i+1}xn_i}$ gewichtet wird. Die Schwellenwerte jedes Neuronen werden durch die Biase $\mathbf{b}_i \in \mathbb{R}^{n_{i+1}}$ dargestellt. Gleichung 3.3 zeigt, wie das allgemeine KNN mit einer rekursiven Funktion modelliert werden kann.

$$\mathbf{a}_1 := \mathbf{x}, \qquad \mathbf{z}_i := \mathbf{W}_i\mathbf{a}_i + \mathbf{b}_i, \qquad \mathbf{a}_{i+1} := a_i(\mathbf{z}_i), \qquad \mathbf{f(x)} := \mathbf{a}_m \tag{3.3}$$

Diese Arbeit nutzt ausschließelich *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.

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, mit dem Fokus auf effizienten Training der Modelle [ABC+16]. Dafür nutzt es die Multikernarchitektur von CPUs, GPUs und spezialisierte Hardware TPUs (**T**ensor **P**rocessing **U**nit). 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 zu approximieren ist eine Kostenfunktion nötig, die den Abstand von der approximierten Funktion zu der Zielfunktion angibt. Im Optimierungsprozess wird die

Kostenfunktion minimiert. Oft wird die Kostenfunktion auch als Verlustfunktion bezeichnet. Die Kostenfunktion kann aber zusätzlich noch Regulurisierungsterme besitzen. Üblicherweise wird Kreuzentropie für die Verlustfunktion verwendet, da sich diese experimentell als Beste Verlustfunktion für flache KNN erwiesen hat.

Das KNN wird für eine endliche Anzahl an Trainingsdurläufen trainiert. Ein Trainingsdurchlauf einer Trainingsmenge wird als Epoche bezeichnet. Mit jeder Epoche werden die Kosten erhoben und der Optimierer wird ausgeführt. Der Fehler wird dabei rückwerts durch das KNN propagiert, wodurch die Parameter $\boldsymbol{\theta}$ sich verändern. Dieser Vorgang wird als *Backpropagation* bezeichnet. Zu den Parametern gehört die Struktur des neuronalen Netzwerks, Aktivierungsfunktionen, Gewichte und Bias. Mit Fehler ist der Unterschied von dem des KNN ermittelten Wertes zu dem Zielwert gemeint.

Abhängig von der Strategie des Optimierers müssen die Gradienten rückwerts, 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 Ergebnises 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.

TODO: Batchsize

3.3 Optimierer

Die Strategie im Optimierungsprozess wird als Optimierer bezeichnet. Diese Algorithmen steuern, wie die Parameter $\boldsymbol{\theta}$ aktualisiert werden. Üblicherweise wird *ADAM* verwendet. ADAM ist eine Kombination aus *SGD* (**S**tochastic **G**radient **D**escent) mit Momentum und *RMSprop*.

SGD ist eine Approximation von *Gradient Descent* (GD), oder *Steepest Descent*. GD ist

ein iterativer Algorithmus, der den Gradienten in Richtung des Extrema folgt und dementsprechend die Eingabeparameter aktualisiert.

$$x_{k+1} := \begin{cases} x_k - C'(x_k)\alpha_k & \text{, wenn } C(x_k - C'(x_k)\eta_k) < C(x_k) \\ (1+\alpha_k)x_k & \text{, ansonsten} \end{cases} \tag{3.4}$$

Gleichung 3.4 illustriert diesen iterativen Prozess für Minimierung im eindimensionalen Fall, wobei $\eta_k > 0$ eine angemessene *Lernrate* ist, x der Eingabeparameter und C die Kostenfunktion. 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.

Im mehrdimensionalen Fall wird für jede Komponente des Eingabevektors dieser Prozess durchgeführt, sodass für jede Komponente die Richtung des Extrema verfolgt wird. Dies impliziert, dass GD sehr aufwendig zu berechnen ist für Eingabevektoren mit hohen Dimensionen. Gleichung 3.5 zeigt die iterative Berechnung im mehrdimensionalen Fall.

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \bigtriangledown C(\mathbf{x}_k)\eta_k \tag{3.5}$$

Zur Berechnung muss der Gradient des Eingabevektors berechnet werden. Je größer die Dimension des Eingabevektor ist, desto aufwendiger ist die Berechnung.

SGD nimmt an, dass eine Verbesserung wahrscheinlich ist, wenn eine Komponente des Eingabevektors in Richtung des Extrema aktualisiert wird. Aus diesem Grund werden die Parameter aktualisiert, nachdem jeweils nur eine Komponente des Eingabevektors aktualisiert wurde. Dadurch bedarf das neuronale Netzwerk zur Konvergenz mehr Epochen, muss aber weniger Berechnungen durchführen.

(S)GD mit Momentum versucht zu vermeiden, dass lokale Extrema gefunden werden anstatt globale Extrema, indem Momentum aus vorherigen Gradienten beibehalten wird, um aus lokalen Extrema wieder raus zu finden. Gleichung 3.6 zeigt die iterative Berechnung.

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

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 zum GD (3.5) ist der Anteil vergangener Gradienten.

RMSprop ist eine Variante von *Adagrad*. Adagrad passt die Lernrate η an, denn typischerweise

wird zuerst eine hohe Lernrate benötigt und je näher sich dem Extrema angenähert wird, sollte diese Lernrate sinken. Gleichung 3.7 zeigt, wie sich iterativ die Lernrate antiproportional zur kummulierten Norm der Gradienten der Kostenfunktion verringert. Dabei wird für ϵ eine kleine Zahl gewählt, um Teilen durch 0 zu vermeiden aber keinen signifikanten Einfluss auf die Berechnung zu haben.

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

Das Problem an Adagrad ist, dass die Lernrate zu schnell gegen 0 konvergieren kann, wodurch das Zielextrema nicht erreicht wird. RMSprop (3.8) löst dieses Problem, indem in jeder Iteration ein Zerfallsfaktor $\gamma < 1$ auf den Hilfsvektor $\mathbf{w}$ angewendet wird und Anteilweise das Hadamard-Produkt des Gradienten der Kostenfunktion addiert wird.

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

Gleichung 3.9 zeigt, wie Adam RMSprop und SGD mit Momentum vereint, wobei $\gamma_1 < \gamma_2 < 1$. Adam nutzt zwei Hilfsvektoren $\mathbf{v}$ und $\mathbf{w}$, die mit einer Zerfallsrate wachsen und beim Lernen sowohl Momentum nutzt, um lokale Extrema zu überbrücken, und passt die Lernrate im Laufe des Trainingsprozesses an.

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

3.4 Aktivierungsfunktionen

Die Aktivierungsfunktion trennt die einzelnen Schichten von einander, indem sie nichtlinearität einführt (TODO: Quelle). Aktivierungsfunktion sollten kontinuierlich und differenzierbar sein. Dies ist wichtig für den *Backpropagation*-Prozess beim Trainieren. Sie unterscheiden sich in ihren Eigenschaften und Berechnungskosten, was eine besondere Rolle für Mikrocontroller spielt. Es gibt viele verschiedene Aktivierungsfunktionen, eingeteilt in Familien.

In der *Heaviside*-Familie sind die Funktionen *Heaviside*, *modifizierte Heaviside* und die *Logistik* enthalten. Die Heaviside-Funktion, oder auch Stufenfunktion, ist die Aktivierungs-

funktion des biologischen Modells. Sie ist aber nicht kontinuierlich bei 0 und damit nicht differenzierbar. Dies stellt ein Problem für neuronale Netze dar, da einerseits sie einerseits unsymmetrisch ist und andererseits die Information über die Distanz zur Entscheidungsgrenze verloren geht. Die modifizierte Heavisde-Funktion löst das erste Problem, indem 0 der Wert $\frac{1}{2}$ zugeordnet wird. Das zweite Problem kann durch *sigmoidal* Funktionen gelöst werden. Eine Variante ist die Logistikfunktion, welche in der frühen Geschichte der neuronalen Netzwerke verwendet wurde. Sie ist kontinuierlich und differenzierbar. Der Wert der Ableitung ist in der Rechweite $[0, \frac{1}{4}]$. Dadurch ist sie ungeeignet für tiefe neuronale Netzwerke, da dadurch der Gradient bereits nach wenigen Schichten gegen 0 geht. Eine Generalisierung der Logistikfunktion ist die *SoftMax*-Funktion oder normalisierte Exponentialfunktion. Diese wird häufig in der Ausgabeschicht, da ihre Ausgabewerte in der Summe 1 ergeben und sie aus diesem Grund als Wahrscheinlichkeit für jede Klasse betrachtet werden können.

Zur *ReLU*-Familie gehört die ReLU-Funktion und ihre Varianten, sowie die *SoftPlus*- und *Swish*-Funktion. ReLU steht für „*rectified linear function*“ und wird häufig in modernen neuronalen Netzwerken verwendet. Sie ist nicht differenzierbar bei 0 und die Ableitung für negative Eingaben ist 0. Dafür ist die Ableitung der Funktion ansonsten 1, was sehr gut für den Backpropagation-Prozess ist, da dadurch der Gradient unverändert ist. Zudem ist sie sehr leicht zu berechnen. Varianten sind *leaky ReLU* und *ELU (exponential linear unit)* welche versuchen das Problem bei 0 zu lösen. *SoftPlus* ist analytisch, dafür aber aufwendiger zu berechnen im Vergleich zu ReLU. Eine weitere Variante ist *Swish*. 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.

Daneben gibt es noch die *Sign*-Familie und die *Abs*-Familie, die in dieser Arbeit nicht verwendet werden (TODO: Quelle).

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. Ein FFNN mit 3 Schichten der Größe $n_1, n_2 und n_3$ hat $n_1 n_2 + n_2 n_3$ Gewichte und $n_1 + n_2 + n_3$ Biase. Die Gewichte und Biase sind für gewöhnlich Gleitkommazahlen. Dieser kann mit 4 und 8 Byte dargestellt werden. Als Alternative können auch Festkommazahlen mit verwendet werden, die 2 Byte benötigen.

Die Ausführungszeit des KNN ist stark abhängig von der Hardware. Das KNN kann sowohl im RAM abgelegt werden oder im Flash-Speicher. 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 es die Ausführungszeit um bis zu 74% verlangsamt.

Außerdem wird Multiplikation und Division zur Evaluierung des KNNs benötigt. Wenn die Hardware keine Unterstützung dafür hat, müssen diese Operationen durch Software ergänzt werden. Diese sind dann signifikant komplexer zu berechnen im Vergleich zu Hardwareuntersützten Operationen. Moderne Prozessoren besitzen Hardware Instruktionen, um Matrix Multiplikation durchzuführen anstatt die Matrix Multiplikation in Software durchzuführen.

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

Es gibt verschiedene Optimierungen, die die Ausführungszeit und 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 und der Ausführungszeit zeigt. 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. (TODO: WER?) hat durch geschickte Registerallokation und nutzen des Scratchpad Memory (SPM) signifikante Verbesserungen der Ausführungszeit und Energiebedarf gezeigt.

Kapitel 4

Standortbestimmung

- Location Awareness
- Anwendungen?

4.1 Lokalisierung

- Indoor
- Outdoor
- Network vs. Device based

4.2 Anwendung von Lokalisation

- GPS
- Triangulierung
- Navigation, Monitoring, Tracking

4.3 Standortbestimmung mit maschinellen Lernen

- Vision Based

4.4 Orientierungssinn

- Beispiel Honigbienen

Kapitel 5

Trainings- und Validationsdaten

- Daten aufgenommen mit CoppeliaSim(?)

5.1 Simulierten Sensordaten

- Wie wurde es aufgenommen?
- Welche Sensoren
- Aufgenommene Routen

5.2 Künstlichen Sensordaten

- Motivation: Warum ist das nötig?

5.2.1 Magnetfeld

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

5.2.2 Temperatur

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

5.2.3 Lautstärke

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

5.2.4 WLAN Zugangspunkte

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

5.3 Simulation von Interrupts

- Motivation => Energieverbrauch, Spiegelung der echten Datenaufnahme, Reduzierung der Trainingsdaten
- Wie funktioniert ist?
- Wie und Wann bei der Datenverarbeitung wird es gemacht?

5.4 Standortenkodierung

- Wie werden Orte modelliert?
- Warum wurden die Orte so modelliert?
- Wie werden Pfade zwischen Orten modelliert?
- Sollte das Modell ausgeben können, dass kein Ort erkannt wurde?
- Sollten Pfade erkannt werden können?

5.5 Feature-Extrahierung

- Datenfenster: Realzeit vs. Diskret mittels Wakeups
- Relevanz von Zeit
- Welche Feature werden genutzt?

- Wie werden diese extrahiert?
- Welchen Mehrwert verschaffen diese Features?
- Welchen Einfluss haben Sie im Hinblick auf Ressourcenbedarf(?), Klassifizierungsgenauigkeit(?), Fehlertoleranz(?)

5.6 Aufteilung der Daten

- Kurz und knapp wie und warum werden die Daten aufgeteilt.
- Sollten Trainingsdaten um synthetische Daten ergänzt werden?
  - Fault Daten, um das Modell Robuster zu machen
  - Synthetische Routen
- Wie viele Trainingsdaten werden benötigt?
  - Um KNN zu trainieren?
  - Um Entscheidungsbaum zu trainieren?
  - Ggf. Unterschiede klären

Kapitel 6

ML-Modelle

TODO

6.1 Entscheidungswald

- Wie wird automatisch das beste Modell gefunden?
- Warum trainieren wir es so => Um die beste Konfiguration zu finden unter den bestehenden Restriktionen.
- Welche Restriktionen werden eingefordert?
- Warum gibt es diese Restriktionen?
- Welchen Einfluss hätte es, gäbe es diese nicht?

6.2 Feed Forward Neuronales Netzwerk

- Wie viele Hidden Layers und Neuronen sollte es haben?
- Welche Aktivierungsfunktionen werden verwendet?
- Braucht man mehr Neuronen/Hidden Layer mit steigender Ort Anzahl?

6.3 Feedback Kanten

- Was ist das?
- Wofür ist das Relevant?
- Wie funktionier es?

6.4 Training der Modelle

- Trainieren in Phasen => Erklären wie es genau funktioniert?
- Warum trainieren wir in Phasen?
- Sag das wir die Zyklen der einzelnen Pfade nutzen
- Werden mehr Trainingsdaten benötigt mit steigender Ort Anzahl? Wenn ja wie viel?

Kapitel 7

Evaluation

TODO

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(?)
- 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?

7.2 Fehlertoleranz

- Metriken => Insbesondere continued_predict hier erwähnen
- Wenn falscher Ort erkannt wurde, wie lange dauert es um wieder den korrekten Ort zu finden?
- Was passiert wenn Sensoren ausfallen?
- Transportbox nicht dem trainierten Pfad folgt?

- Änderungen der Fabrik, e.g. Licht, Wärme, Magnet, Schnelligkeit der Fließbänder
- Einfluss von einzelnen Features für die Fehlertoleranz(?)

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(?)

Kapitel 8

Diskussion

TODO

Kapitel 9

Schlussfolgerungen

TODO

- Kurze, knappe und gut formulierte Schlussfolgerung
  - Was ist besser Entscheidungsbaum oder Entscheidungswald? Welche Vor- und Nachteile?
  - Sollten Entscheidungswälder verwendet werden?
  - Sollten KNNs verwendet werden?
  - Is die Realzeit relevant für die Ortung(?)
- Kurze Zusammenfassung wichtigster Dinge
- Zukünftige Arbeit

Inhalt des USB-Sticks

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

[BFSO84] BREIMAN, Leo ; FRIEDMAN, Jerome ; STONE, Charles J. ; OLSHEN, Richard A.: *Classification and regression trees*. CRC press, 1984

[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

[D+02] DIETTERICH, Thomas G. u. a.: Ensemble learning. In: *The handbook of brain theory and neural networks* 2 (2002), S. 110–125

[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

[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

[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

[GEW06] GEURTS, Pierre ; ERNST, Damien ; WEHENKEL, Louis: Extremely randomized trees. In: *Machine learning* 63 (2006), Nr. 1, S. 3–42

[GL09] GHOSH, Joydeep ; LIU, Alexander: K-Means. In: *The top ten algorithms in data mining* 9 (2009), S. 21–22

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

[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

[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

[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

[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

[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