

Forschungsprojekt und Seminar

Computer Science

Handgestenerkennung mit Entscheidungsbäumen

von

Tom Dymel

Februar 2021

Betreut von

Dr. Venzke
Institute of Telematics, Hamburg University of Technology

Erstprüfer Prof. Dr. Volker Turau

Institute of Telematics
Hamburg University of Technology

Zweitprüfer Dr. Venzke

Institute of Telematics
Hamburg University of Technology

Inhaltsverzeichnis

1 Einleitung	1
2 Methoden	3
2.1 Entscheidungsbäume	3
2.1.1 Konstruktion	4
2.2 Ensemble-Methoden	4
2.2.1 Wahlen	5
2.2.2 Bagging	5
2.2.3 Random Forest	5
2.2.4 Extremely Randomized Trees	6
2.2.5 Boosting	6
3 Stand der Forschung	7
3.1 Ähnliche Arbeiten	7
3.2 Optische Handgestenerkennung	8
3.2.1 Exrahierung von Gestenkandidaten	9
3.2.2 Skalierung des Gestenkandidaten	11
3.2.3 Trainings- und Testdaten	11
3.2.4 Gestenerkennung mit künstlichen neuronalen Netzen	12
4 Evaluierung von Entscheidungsbäumen	15
4.1 Infrastruktur	15
4.1.1 lib_gesture	16
4.1.2 lib_feature	16
4.1.3 lib_data_set	16
4.1.4 lib_evaluation	16
4.1.5 Simulator	17
4.1.6 Extractor	17
4.1.7 Reader	17
4.1.8 Recorder	17
4.2 DymelData	17
4.2.1 Konfigurationen	18
4.2.2 NullGesten	18
4.2.3 Synthetische Helligkeitstestmenge	19
4.3 Entscheidungsbaum basiertes Model	19
4.3.1 Training	19
4.3.2 C-Code Generierung eines Entscheidungsbaumes	20
4.3.3 C-Code Generierung eines Entscheidungswaldes	21
4.4 Features	22
4.4.1 Requirements	23
4.4.2 Curse of dimensionality	23

INHALTSVERZEICHNIS

4.4.3	Unpromising features	23
4.4.4	Brightness distribution	23
4.4.5	Motion History	23
4.4.6	Center of Gravity Distribution	23
4.5	Performance Evaluation	23
4.5.1	Testsets	24
4.5.2	Feasible solution	24
4.5.3	Considering NullGestures	24
4.5.4	Brightness Distribution	24
4.5.5	Motion History	24
4.5.6	Brightness Distribution and Motion History	24
4.5.7	Center of Gravity Distribution Float Ansatz	24
4.5.8	Center of Gravity Distribution Integer Ansatz	24
4.5.9	Center of Gravity Distribution Float and Integer Ansatz	25
4.5.10	Comparison to previous work	25
4.6	Execution Time Evaluation	25
4.6.1	WCEP and WCET	25
4.6.2	AVR Compiler for AtMega328p	25
4.6.3	Optimization Level	25
4.6.4	Feature extraction	25
4.6.5	Tree Evaluation	25
4.6.6	Forest Evaluation	25
4.6.7	Total Execution Time	25
4.7	Size Evaluation	25
4.7.1	CCP (TODO: Abrev.)	25
4.7.2	Minimum Leaf Sample Size	25
5	Conclusion	27
Literaturverzeichnis		29
A	Content of the DVD	31

Einleitung

Maschinelles Lernen (ML) gewann in den vergangenen Jahren an Popularität, u.a. durch die Fortschritte in parallelen Rechnen, sinkende Speicherpreise, schnelleren Speicher und den Zugang zu Bibliotheken, wie zum Beispiel Scikit-Learn, Keras und PyTorch, welche den Einstieg in maschinellen Lernen erleichtern (TODO: Quelle?). Ein namhaftes Beispiel für das Potential von maschinellen Lernen ist die AlphaGo-KI, die einen Sieg gegen den besten menschlichen Spieler erringen konnte in dem Spiel Go, welches als besonders schwierig für Computer zu meistern galt durch den enormen Suchraum von möglichen Aktionen (TODO: Quelle).

Ein häufiges Anwendungsgebiet in eingebetteten Systemen ist die optische Gestenerkennung, die zur kontaktlosen Interaktion mit technischen Geräten u. a. genutzt wird (Todo: Quelle). Die eingesetzten Micro-Controller sind jedoch häufig nicht ausreichend leistungsfest, um ein trainiertes Model in passabler Zeit auszuführen (TODO: Quelle). Gründe dafür sind Kosten oder Anforderungen an die Batterielanglebigkeit (TODO: Quelle). Eine Lösung ist das Auslagern der Modelle in einen leistungsstarken Rechenkluster, indem die nötigen Eingabedaten auf dem Mikro-Controller gesammelt werden und anschließend an den Rechenkluster gesendet werden (TODO: Quelle). Dies erzeugt allerdings Latenz und eine Abhängigkeit zu einer solchen Infrastruktur. Ein alternativer Ansatz ist das lokale Ausführen der Modelle, was allerdings die Komplexität des Modells limitiert um eine passable Ausführungszeit zu gewährleisten.

In dieser Arbeit wird die Handgestenerkennung mit Entscheidungsbäumen auf dem Arduino-Board ATmega328P untersucht, das mit 9 Fotowiderständen ausgestattet ist, die in einer 3x3 Matrix angeordnet sind. Der ATmega328P verfügt über eine 8-Bit APU, 2 kB RAM, 32 kB Flash-Speicher und operiert unter 16 MHz. Im Vergleich zu vorherigen Arbeiten, die sich mit Künstlichen Neuronalen Netzen (KNN) auseinander gesetzt haben, versprechen Entschei-

1 EINLEITUNG

dungsbäume einen geringeren Rechenaufwand.

Um dieses Ziel zu erreichen, werden Entscheidungsbaum basierte Klassifizierer auf einem leistungsstarken Computer trainiert und anschließend in eine bestehende Infrastruktur eingebettet, die Gestenkandidaten als Folge von Bildern identifiziert. Es werden insgesamt zwei Komponenten hinzugefügt. Die erste Komponente extrahiert Features, welche an die zweite Komponente, der Klassifizierer, weitergegeben werden um die Handgesten zu erkennen. Der Klassifizierer wird mit den vorherigen Arbeiten verglichen im Hinblick auf Ausführungszeit, Resourcenverbrauch und Erkennungsgenauigkeit unter verschiedenen Verhältnissen, wie Geschwindigkeit, Licht und Entfernung. Zusätzlich wird untersucht inwiefern Nullgesten, i. e. invalide Gesten, erkannt werden können und welche Konsequenzen es auf die Erkennungsgenauigkeit hat.

Zunächst werden in Kapitel 2 Entscheidungsbäume eingeführt, verschiedene Ensemble-Methoden erläutert und auf die Generierung von den Modellen eingegangen. In Kapitel 3 wird erläutert, wie Gestenkandidaten extrahiert werden und welche Ansätze vorherige Arbeiten bereits verfolgt hatten. Anschließend präsentiert Kapitel 4 den Kern der Arbeit. Zuallererst wird die Infrastruktur vorgestellt, welche im Rahmen dieser Arbeit angefertigt wurde, um Klassifizierer zu trainieren und zu evaluieren. Anschließend wird das Datenset Dymel vorgestellt, was zusätzlich zu dem bestehenden Datenset von Klisch erstellt wurde, um Nullgesten zu und verschiedene Helligkeitsstufen zu untersuchen. Danach werden die Features vorgestellt, die im Laufe dieser Arbeit untersucht wurden. Zuletzt wird die Erkennungsgenauigkeit, die Ausführungsgeschwindigkeit und der Resourcenverbrauch evaluiert. In Kapitel 5 werden die Schlussfolgerungen dargestellt.

Methoden

Dieses Kapitel verschafft einen Überblick über die genutzten Methoden und erläutert in dem Zusammenhang Begrifflichkeiten die für das Verständnis dieser Arbeit relevant sind. Zuerst wird der Entscheidungsbaum vorgestellt und wie man ihn trainiert und anschließend die in dieser Arbeit genutzten Ensemble-Methoden.

2.1 Entscheidungsbäume

Der Entscheidungsbaum ist eine rekursive Datenstruktur um Entscheidungsregeln darzustellen. Jedem Blatt ist eine Klasse zugeordnet und allen anderen Knoten ist ein *Test* zugeordnet. Der Test hat eine Reihe von sich gegenseitig ausschließenden Ergebnissen. Die Zuordnung einer Klasse zu einem Objekt wird durch das Traversieren dieses Baumes bestimmt bis ein Blatt erreicht wird [Qui90]. Abbildung 2.1 zeigt einen Entscheidungsbaum, wo jeder Test zwei mögliche Ergebnisse hat, i. e. einen binären Entscheidungsbaum. Möglich wäre aber auch das jeder Test eine arbiträre Anzahl an möglichen Ergebnissen hätte.

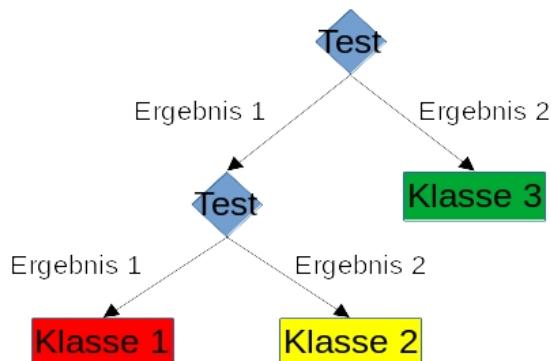


Abbildung 2.1: Beispiel eines binären Entscheidungsbaums mit 3 möglichen Klassen, die klassifizierbar sind.

2 METHODEN

2.1.1 Konstruktion

Es gibt viele verschiedene Algorithmen um Entscheidungsbäume zu erzeugen: ID3, C4.5, C5, CART, CHAID, QUEST, GUIDE, CRUISE and CTREE. Am häufigsten wird ID3 (Iterative Dichotomizer 3), bzw. C4.5, welches eine Weiterentwicklung von ID3 ist, und CART (Classification and Regression Trees) verwendet [PVG⁺11, SG14]. Diese Arbeit verwendet die Implementierung für Entscheidungsbaumklassifizierer von *Scikit-Learn*, die eine optimierte Version von CART nutzt [Ent20]. Scikit-Learn ist ein Python-Modul, das eine große Auswahl von Algorithmen zum maschinellen Lernen implementiert [PVG⁺11].

CART: Classification and Regression Trees

Die Konstruktion eines optimalen binären Entscheidungsbaum ist NP-Komplett [LR76]. CART ist ein Greedy-Algorithmus, der lokal immer die beste Teilung wählt.

```
Weise dem Wurzelknoten alle Trainingsdaten zu.  
Definiere den Wurzelknoten als Blatt.  
WHILE True:  
    Neue_Teilungen = 0  
    FOR jedes Blatt:  
        IF die Größe der zugewiesenen Trainingsdaten zu klein ist oder alle ↘  
            Einträge der Trainingsdaten zur gleichen Klasse gehören:  
            CONTINUE  
        Finde das Attribut, das am besten den Knoten in zwei Kindesknoten ↘  
            unterteilt mit einer erlaubten Teilungsregel.  
        Neue_Teilungen += 1  
    IF Neue_Teilungen == 0:  
        break
```

■ Listing 2.1: Skizze von vereinfachten Baumwachstumsalgorithmus [Ste09].

Listing 2.1 skizziert wie CART initial einen maximal großen Baum generiert indem die Trainingsdaten solange geteilt werden bis keine weitere Teilung mehr möglich ist oder alle Einträge der gleichen Klasse zugeordnet sind. Zuletzt beginnt der Reduzierungsprozess indem Teilbäume gelöscht werden, die die Klassifizierungsgenauigkeit nicht erhöhen oder der Zuwachs unter einem Nutzer definierten Schwellenwert liegt [Ste09].

2.2 Ensemble-Methoden

Oftmals ist der Suchraum für das Problem zu groß, als das es möglich wäre in tolerabler Zeit die optimale Lösung zu finden [LR76]. Konstruktionsalgorithmen für Entscheidungsbäume arbeiten aus diesem Grund auf Basis von Heuristiken um die lokal optimale Teilung zu bestimmen. Im Gegensatz zu diesen Algorithmen versuchen Ensemble-Methoden nicht die

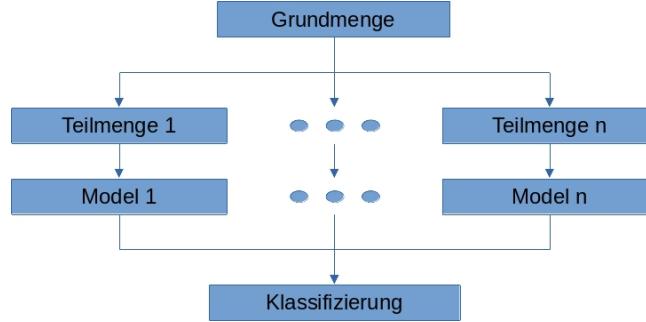


Abbildung 2.2: Klassifizierungsprozess mit der Bagging-Methode.

beste Lösung, sondern konstruieren eine Menge von Lösungen unter denen anschließend gewählt wird, was die finale Lösung für ein Problem ist [D⁺02].

2.2.1 Wählen

Formal wird also ein „Wahl“-Klassifizierer $H(x) = w_1 h_1(x) + \dots + w_K h_K(x)$ geschaffen, mit Hilfe von einer Menge von Lösungen $\{h_1, \dots, h_K\}$ und einer Menge von Gewichten $\{w_1, \dots, w_K\}$, wobei $\sum_i w_i = 1$. Eine Lösung $h_i : D^n \mapsto \mathbb{R}^m$ weist einer arbiträren, n -dimensionalen Menge D^n jeder der m möglichen Klassen eine Wahrscheinlichkeit zu. Die Summe einer Lösung ist immer 1. Die Klassifizierung einer Lösung ist die Klasse mit der höchsten Wahrscheinlichkeit. Dementsprechend ist analog dazu $H : D^n \mapsto \mathbb{R}^m$ definiert. Für gewöhnlich hat jeder Teilnehmer einer Wahl das gleiche Gewicht [D⁺02].

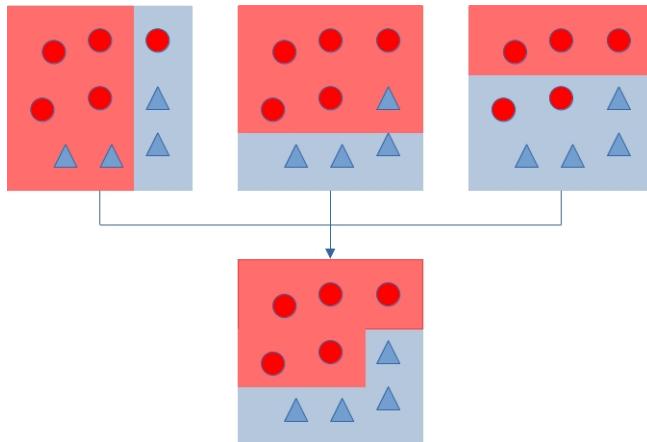
2.2.2 Bagging

Bagging ist ein Acronym für „Bootstrap aggregating“. Die Idee ist aus einer großen Menge von Trainingsdaten, eine Menge von Mengen von Trainingsdaten zu generieren, folgend mit jedem dieser Mengen einen Klassifizierer zu trainieren und schließlich alle Klassifizierer, e.g. durch Wählen, zu aggregieren (siehe Abbildung 2.2) [Bre96]. Die Methode die dahinter steht nennt sich „Bootstrap sampling“, welche einen Prozess beschreibt aus einer Grundmenge m mal jeweils n Einträge zu ziehen, die eine Teilmenge bilden [Efr92]. Der Name ist folglich aus der Methode und dem Aggregierungsprozess abgeleitet.

2.2.3 Random Forest

Random Forest ist eine Erweiterung der Bagging-Methode. Zusätzlich zu der zufällig ausgewählten Menge an Trainingsdaten wird auch zufällig eine Menge von Features ausgewählt. Auf dieser Basis wird ein Menge von Entscheidungsbäumen generiert die anschließend aggregiert werden [Bre01].

2 METHODEN



■ Abbildung 2.3: Klassifizierungsprozess mit der Boosting-Methode.

2.2.4 Extremely Randomized Trees

Im Vergleich zu Random Forest gehen Extremely Randomized Trees einen Schritt weiter. Anstatt den besten Teilungspunkt zu suchen für die ausgewählten Features, werden zufällig ein Teilungspunkte ausgewählt, aus denen der beste genutzt wird. Dies soll die Varianz reduzieren. Außerdem wird nicht wie bei der Bagging-Methode auf Teilmengen trainiert sondern auf dem gesamten Set, was den Bias reduzieren soll [GEW06].

2.2.5 Boosting

Boosting bezeichnet das Konvertieren eines „schwachen“ PAC-Algorithmus (**P**robably **A**pproximately **C**orrect), welcher nur leicht besser ist als Raten, in einen „starken“ PAC-Algorithmus. Ein starker PAC-Algorithmus ist ein Algorithmus der, gegeben $\epsilon, \delta > 0$ und zufällige Beispiele der Trainingsdaten, mit einer Wahrscheinlichkeit $1 - \delta$ klassifiziert mit einem Fehler bis zu ϵ und die Laufzeit muss polynomial in $\frac{1}{\epsilon}, \frac{1}{\delta}$ und anderen relevanten Parametern sein. Für einen schwachen PAC-Algorithmus gilt das Gleiche mit dem Unterschied, dass $\epsilon \geq \frac{1}{2} - \gamma$, wobei $\delta > 0$ [FS97].

In Abbildung 2.3 wird illustriert wie drei schwache Lerner jeweils auf eine Teilmenge nacheinander trainiert werden, wobei die Teilmenge des jeweils nächsten von dem Fehler des vorherigen Models abhängt. Schlussendlich werden alle schwachen Lerner gewichtet aggregiert woraus ein starker Lerner entsteht. In dieser Arbeit wird im speziellen der Boosting Algorithmus **AdaBoost** von Freund und Schapire verwendet [FS97].

Stand der Forschung

Dieses Kapitel verschafft einen Überblick über bisherige Arbeiten im Bereich der Gestenerkennung. Zuerst wird allgemein auf die Gestenerkennung mit Entscheidungsbäumen eingegangen. Anschließend wird der Ansatz zur optischen Gestenerkennung vorgestellt, der im Institut für Telematik an der TUHH entwickelt wurde. Zuletzt wird auf die Ergebnisse von bisherigen Arbeiten mit künstlichen neuronalen Netzen mit diesem Ansatz eingegangen.

3.1 Ähnliche Arbeiten

Es gibt viele Ansätze, die sich mit der Gestenerkennung beschäftigen. Es wird unterschieden zwischen optischen und nicht-optischen Ansätzen. Der optische Ansatz nutzt einen oder mehrere Kameras um eine Folge von Bildern aufzunehmen. Dieser Ansatz ist allerdings empfindlich gegenüber Lichtverhältnisse und der Distanz die der Nutzer zu den Kameras hat [SHL⁺19]. Nicht-optische Ansätze bedienen sich anderen Sensoren, z. B. Infrarot Abstands-sensoren [CCRB11], oder nutzen technische Hilfsmittel um zusätzliche Daten zu erfassen.

Song et al. [SHL⁺19] haben die Handgestenerkennung mit Gradient Boosting Entscheidungsbäumen untersucht. Sie wählten einen nicht-optischen Ansatz, der aus einem tragbaren sEMG Recorder der die elektrischen Signale der Muskelaktivitäten erfasst. Als Eingabe für den Entscheidungsbaum wählten sie neun Features die in die Kategorie von zeitabhängigen Features einzuordnen sind (siehe Tabelle 3.1). Damit erzielten sie eine Erkennungsgenauigkeit von 91% unter 12 verschiedenen Handgesten.

Ahad et al. [ATKI12] diskutieren den Motion History Image (MHI) Ansatz. MHI ist ein optischer Ansatz, der eine Sequenz von Bildern in ein einziges komprimiert. Dabei werden dominante Bewegungen die kürzlich verarbeitet wurden heller angezeigt als nicht dominante

3 STAND DER FORSCHUNG

1	Mean absolute value	$\frac{1}{N} \sum_{t=1}^N x_t $
2	Simple square integral	$\sum_{t=1}^N x_t ^2$
3	Minimum value	$\min x_t$
4	Maximum value	$\max x_t$
5	Standard deviation	$\sqrt{\frac{1}{N} \sum_{t=1}^N (x_t - \tilde{x})^2}$
6	Average amplitude change	$\frac{1}{N-1} \sum_{t=1}^{N-1} x_{t+1} - x_t $
7	Zero crossing	$\sum_{t=1}^{N-1} \text{diff}(\text{sgn}(x_{t+1}), \text{sgn}(x_t))$
8	Slope sign change	$\sum_{t=1}^{N-2} \text{diff}(\text{sgn}(x_{t+1} - x_t), \text{sgn}(x_t - x_{t-1}))$
9	Willison amplitude	$\sum_{t=1}^{N-1} u(x_{t+1} - x_t - \text{threshold})$

■ **Tabelle 3.1:** Die von Song et al. genutzten Features [SHL⁺19].

Bewegungen oder Bewegungen die schon länger zurück liegen.

$$H_\tau(x, y, t) = \begin{cases} \tau & \text{if } \psi(x, y, t) = 1 \\ \max(0, H_\tau(x, y, t - 1) - \delta) & \text{otherwise} \end{cases} \quad (3.1)$$

Das MHI kann sequentiell berechnet werden. Initial sind alle Werte 0. Wenn $\psi(x, y, t)$ eine dominante Bewegung in einem Pixel (x, y) zu einem Zeitpunkt t signalisiert, dann wird der Pixel zum Maximalwert τ gesetzt. Mit jedem Bild in denen keine dominante Bewegung im Pixel (x, y) stattgefunden hat, wird der Wert um den Zerfallswert δ dekrementiert bis zu einem Minimum von 0 (siehe Formel 3.1).

MHI ist leicht zu berechnen und Invariant zu Lichtverhältnissen. Allerdings ist die Leistung stark abhängig von ψ , τ und δ . MHI ist besonders anfällig für Bildfolgen mit verschiedener Länge. Jenachdem wie τ und δ gewählt sind, ist die Bewegungshistorie nicht sichtbar oder verloren gegangen.

3.2 Optische Handgestenerkennung

Diese Arbeit ist Teil einer Fallstudie zur Handgestenerkennung auf Low-End Mikrocontrollern von dem Institut für Telematik an der TUHH [VKK⁺20]. Das Ziel ist die Handgestenerkennung in Echtzeit mit so wenig Ressourcen wie möglich, damit die Produktion der einzelnen Module so kostengünstig wie möglich ist. Als Eingabe dient, je nach Modul, eine 3x3, bzw. 4x4, Matrix von Lichtsensoren. Dabei werden 5 Typen von Handgesten untersucht: Links nach Rechts, Rechts nach Links, Oben nach Unten, Unten nach Oben und NullGeste, i.

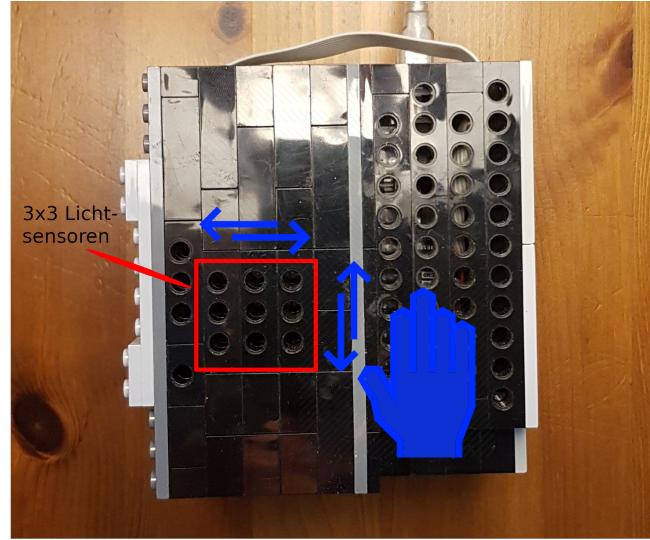


Abbildung 3.1: Das Arduino-Board ATmega328P mit 3x3 Matrix von Lichtsensoren in Lego-Verpackung. Illustriert werden die möglichen Handgestentypen mit Ausnahme der Nullgeste.

e. eine invalide Geste (siehe Abbildung 3.1). Die bisherigen Arbeiten haben sich mit künstlichen neuronalen Netzen beschäftigt. Dessen Prozessablauf zur Gestenerkennung lässt sich im Grunde auf 3 Schritte zusammenfassen.

1. Extrahiere einen Gestenkandidaten.
2. Vorverarbeite den Gestenkandidaten.
3. Wende das Model auf die vorverarbeiteten Daten an.

3.2.1 Extrahierung von Gestenkandidaten

Die Lichtsensorenmatrix liefert einen kontinuierlichen Strom an Bildern. Dabei limitiert die Verarbeitungszeit eines Bildes die Anzahl an Bilder pro Sekunde. Als Gestenkandidat wird eine Folge von Bildern definiert, die ein Ereignis einschließt. In diesem Fall wird das Ereignis durch die Veränderung im gleitenden Mittelwert der Lichtverhältnisse definiert, i. e. sobald der gleitende Mittelwert unterschritten wird ein Gestenkandidat angefangen aufgenommen zu werden und sobald die Lichtverhältnisse zu dem Wert zurückkehren wird die Aufnahme beendet. Der gleitende Mittelwert wird dabei immer angepasst, wenn kein Gestenkandidat aufgenommen wird, um sich den verändernden Lichtverhältnissen anzupassen. Da leichte Veränderungen natürlich sind, muss eine Toleranzgrenze von 10% unterschritten werden, damit die Aufnahme gestartet wird. Dies hat als Folge, dass der Anfang und das Ende nicht vollständig ist. Aus diesem Grund schlug Kubik zusätzlich vor am Anfang und Ende weitere

3 STAND DER FORSCHUNG

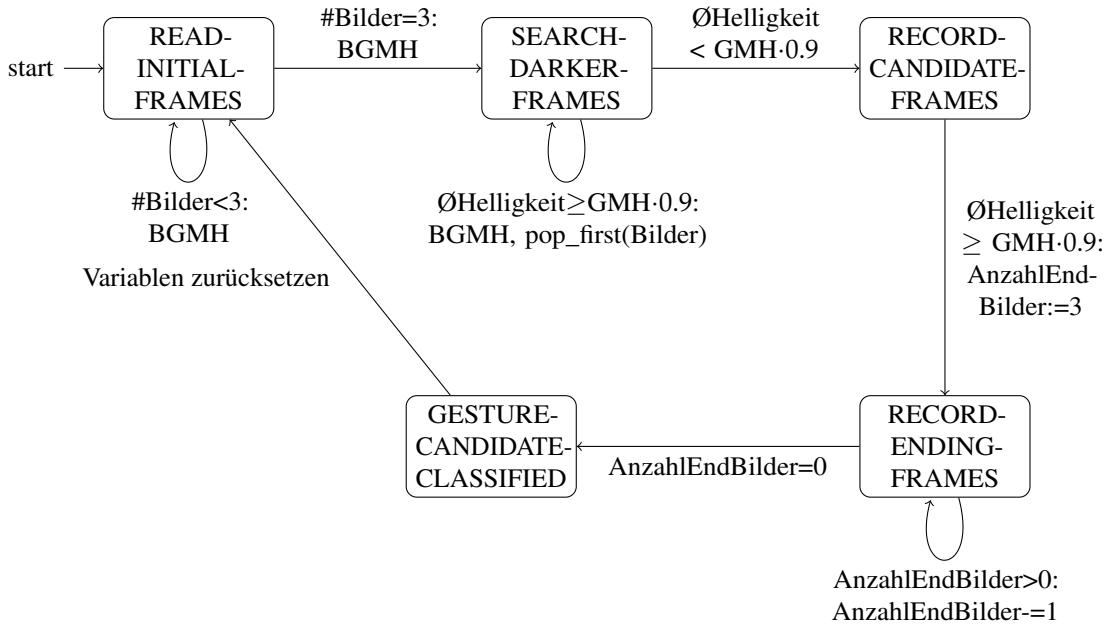
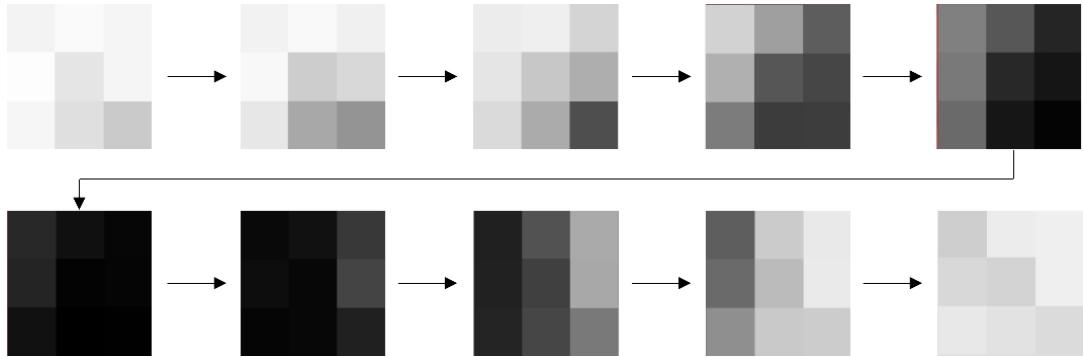


Abbildung 3.2: Implementierung von Kubik's Algorithmus um Gestenkandidaten zu erkennen von Dr. Marcus Venzke. BGMH steht für die Aktion „Berechnung Gleitender Mittelwert der Helligkeit“ und GMH steht für die Variable „Gleitender Mittelwert der Helligkeit“.

Bilder anzufügen [Kub19].

Abbildung 3.2 zeigt die konkrete Implementierung dieses Algorithmus mit einem Zustandsautomaten von Dr. Marcus Venzke. In jedem Zustand wird das aktuelle Bild dem Puffer angefügt. Der Automat verbleibt im Zustand **READ-INITIAL-FRAMES** bis der Puffer 3 Bilder enthält und passt stets den gleitenden Mittelwert der Helligkeit an. Anschließend geht der Automat in den Zustand **SEARCH-DARKER-FRAMES** über, indem er weiterhin den gleitenden Mittelwert anpasst und immer das erste Bild aus dem Puffer entfernt, da lediglich 3 Bilder jeweils vor Aufnehmen und nach dem Aufnehmen des Gestenkandidaten angefügt werden sollen. Sobald die Durchschnittshelligkeit den 90% des gleitenden Mittelwerts unterschreitet wird die Aufnahme begonnen. Der Automat geht in den Zustand **RECORD-CANDIDATE-FRAMES** über. Dort verbleibt der Automat solange bis die Durchschnittshelligkeit 90% des gleitenden Mittelwerts überschreitet, woraufhin der Automat in den Zustand **RECORD-END-FRAMES** über geht, indem die letzten 3 Bilder an den Puffer angehängt werden. Der Zustand des Puffers im Zustand **GESTURE-CANDIDATE-CLASSIFIED** repräsentiert den Gestenkandidaten. Zuletzt werden alle nötigen Variablen zurückgesetzt und der Automat geht in den initialen Zustand wieder über.



■ Abbildung 3.3: Illustration der Handgeste von Links nach Rechts aus Listing 3.1.

3.2.2 Skalierung des Gestenkandidaten

Ein Gestenkandidat besteht aus einer variablen Anzahl an Bildern. Durch die künstlich angefügten Bilder am Anfang und Ende sind es mindestens 8 Bilder. Kubik erkannte, dass ein neuronales Netz eine feste Anzahl an Eingaben hat und diskutierte verschiedene Ansätze. Er verwarf die Idee den Puffer mit irrelevanten Bildern oder Nullen auszufüllen, Bilder zu duplizieren oder Teile des Gestenkandidaten zu verwerfen, da dadurch nicht die vollständige Geste auf die Eingangs-Ebene abgebildet werden würde, oder dass die Geste womöglich verzerrt wäre. Aus diesem Grund hat Kubik sich für lineare Interpolation auf eine fixe Anzahl von 20 Bildern entschieden im Falle wenn weniger als 20 Bilder vorhanden sind. Im Falle, wenn mehr als 20 Bilder vorhanden sind, werden 20 Bilder gleichverteilt ausgewählt [Kub19]. Dieser Ansatz wurde auch von Anton Giese aufgegriffen, der sich in diesem Zusammenhang ebenfalls mit künstlichen neuronalen Netzen beschäftigt hatte [Gie20].

3.2.3 Trainings- und Testdaten

```

...
665,683,669,690,627,670,672,611,557,1
662,679,657,676,564,592,633,467,415,1
645,653,583,627,549,483,598,474,230,1
576,444,269,488,251,209,352,184,187,1
361,254,123,343,130,82,304,83,36,1
131,69,41,120,34,39,72,25,30,1
49,71,174,61,45,206,40,45,110,1
111,242,473,113,195,467,122,210,343,1
272,559,637,304,518,639,401,553,562,1
566,646,654,592,580,654,634,618,602,1
...

```

■ Listing 3.1: Beispiel einer gespeicherten Handgeste von Links nach Rechts.

3 STAND DER FORSCHUNG

Die Modelle werden auf Basis von aufgenommenen Daten trainiert und getestet. Jedes aufgenommene Bild wird durch einen Komma separierten Vektor von Zahlen dargestellt gefolgt von einer Annotation für den Gestentyp. Eine Geste ist eine Folge von Bildern, die die gleiche Annotation teilen (siehe Listing 3.1). Insgesamt gibt es 4792 Aufnahmen von validen Handgesten, die unter verschiedenen Lichtverhältnissen und Distanzen zur Kamera aufgenommen wurden. Dabei wurde die Gesten mit der Hand und Finger ausgeführt in verschiedenen Geschwindigkeiten.

Synthetische Daten

Um die Datenmenge zu erhöhen, können synthetisch Daten aus den bestehenden Daten erzeugt werden indem sie rotiert werden, Rauschen hinzugefügt wird oder die Helligkeit, Kontraste oder Gamma verändert wird [VKK⁺20].

Testdaten und Metrik von Klisch

Als Testdaten wird ein Teil der Datenmenge bezeichnet, die nicht zum trainieren verwendet wurde. Kubik hat Testdaten unter verschiedenen Lichtverhältnissen und Entfernung zur Kamera aufgenommen. Klisch hat daraus eine Testmenge erstellt die von Klisch und Giese zur Verifikation verwendet wurden [Kli20, Gie20]. Klisch definiert die Erkennungsgenauigkeit als Verhältnis zwischen der Anzahl an korrekt erkannten Gesten und der Gesamtanzahl (siehe Formel 3.2) [Kli20].

$$\text{accuracy} = \frac{\#\text{true positives}}{\#\text{total gestures}} \quad (3.2)$$

3.2.4 Gestenerkennung mit künstlichen neuronalen Netzen

Insgesamt gingen dieser Arbeit 4 Arbeiten voraus, die sich mit künstlichen neuronalen Netzen im Zusammenhang dieser Fallstudie beschäftigt hatten.

Engelhardt

Engelhardt führte die in 3.2 definierten Handgesten mit der Hand, einem Finger und 2 Finger unter verschiedenen Helligkeiten aus, auf Basis dessen seine Modelle trainiert und validiert wurden. Er argumentiert, dass rekurrente neuronale Netze (RNN), Feedforward neuronale Netze (FFNN) und Long-Short-Term Memory neuronale Netze (LSTMNN) am besten geeignet für temporale Probleme seien. Convolutional neuronale Netze (CNN) verwirft Engelhardt aufgrund der geringen Auflösung der Gesten und da die Faltung extrem Rechenaufwendig sei. Des Weiteren verwirft er LSTMNN, da diese zu viel Rechenleistung und Speicherplatz benötigen. Als Eingabewerte zu seinen RNNs und FFNNs diente eine Sequenz von 20 Bildern.

die zu 180 Werten konkatiniert wurden und auf Werte zwischen 0 und 1 normalisiert wurden. Als bestes Model stellte sich eines seiner FFNNs heraus, das auf seinen Testdaten bis zu 99% Erkennungsgenauigkeit erzielte. Außerdem erwies es sich als robust gegenüber Rauschen und Helligkeitsveränderungen im Vergleich zum RNN. Die Ausführungszeit des FFNN belief sich auf 11,54 ms mit einem Verbrauch von 11 kB Flash-Speicher und 573 bytes RAM [Eng18].

Kubik

Kubik hat in seiner Arbeit den FFNN Ansatz von Engelhardt aufgegriffen. Er untersuchte Gesten die mit der Hand ausgeführt werden mit verschiedenen Distanzen zur Kamera und unter guten und schlechten Lichtverhältnissen. Neben der Facettenkamera, die Engelhardt ebenfalls genutzt hatte, untersuchte Kubik ebenfalls eine Lochkamera. Er stellte fest, dass diese aber wesentlich schwerer war auszuleuchten, was sich auch bei der Erkennungsgenauigkeit bemerkbar machte. Als Eingabe nutze Kubik ebenfalls 180 Werte, die 20 Bilder repräsentieren. Um mit der variablen Länge von Gesten umzugehen schlug Kubik vor die Bildsequenzen auf 20 Bilder zu skalieren (siehe Sektion 3.2.2). Um die Skalierung durchzuführen musste allerdings der Anfang und das Ende der Geste bekannt sein. Aus diesem Grund war es nötig Gestenkandidaten erkennen zu können (siehe Sektion 3.2.1). Er stellte fest, dass dies die Gesamtlänge der Geste limitierte in Abetracht des RAMs von dem Arduino. Um die Erkennungsgenauigkeit zu erhöhen verwendete er synthetische Trainingsdaten, die er aus bestehenden Daten durch Rotation generierte (siehe Sektion 3.2.3). Dies erhöhte die Erkennungsgenauigkeit erheblich. Kubik erstellte Testdaten (siehe Sektion 3.2.3) und evaluierte sein Model darauf. Im allgemeinen stellte er fest, dass mit zunehmender Distanz zur Kamera die Erkennungsgenauigkeit sich verschlechtert. Dies erwies sich besonders als ein Problem für die Lochkamera. Bei guten Lichtverhältnissen konnte sein Ansatz mit der Facettenkamera bis 30 cm eine Erkennungsgenauigkeit von 97,2% erreichen. Bei schlechten Lichtverhältnissen war die Erkennungsgenauigkeit bereits ab 20 cm nur noch bei 83%. Zusätzlich zu den 4 Grundgesten, untersuchte Kubik Nullgesten. Er stellte fest, dass ruckartige Veränderungen der Lichtverhältnisse mit 92% erkannt wurden und Handbewegungen die wieder zurück gezogen wurden mit 96%. Schwierigkeiten hat die Erkennung von diagonalen Bewegungen als Nullgeste bereitet, da diese eine hohe Ähnlichkeit zu den benachbarten horizontalen und vertikalen Gesten hat. Kubiks Ansatz hat insgesamt 36 ms benötigt um das Model auszuwerten und 11 ms für die Skalierung [Kub19].

Klisch

Engelhardt stellte fest, dass RNNs schlechtere Erkennungsgenauigkeiten ergaben, als FFNNs, da sie schwerer zu trainieren sind [Eng18]. Aus diesem Grund schlug Venzke vor, dass man ein RNN als mehrere FFNNs trainieren könnte. Diesen Ansatz hat Klisch in seiner Arbeit auf-

3 STAND DER FORSCHUNG

gegriffen. Klisch hat verschiedene Konfigurationen getestet und stellte am Ende fest, dass ein RNN als einzelnes Netzwerk zu trainieren bessere Ergebnisse liefert als ein RNN als mehrere FFNNs zu trainieren. Mit seinem RNN erzielte Klisch eine Erkennungsgenauigkeit von 71% unter guten und verhältnismäßig schlechten Lichtverhältnissen, welches eine Verbesserung zu dem Ergebnis von Engelhardt ist. Klisch stellte fest, dass das sein Model schnell genug ist, um 50 Hz zu unterstützen [Kli20].

Giese

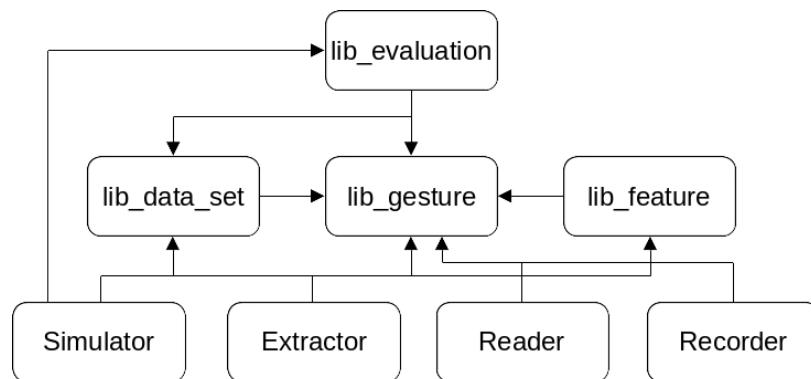
Der Fokus von Gieses Arbeit lag auf Kompression und Optimierung. Er trainierte ein FFNN und erzielte eine Erkennungsgenauigkeit von 98,96%. Dies ist signifikant besser als das FFNN von Kubik, welches lediglich 83% erzielte. Giese geht davon aus, dass sein FFNN besere Ergebnisse lieferte, da ca. 19x mehr Trainingsdaten zur Verfügung hatte als Kubik. Er untersuchte die Auswirkungen von Pruning, Quantisierung, Sparse Matrix Format, SeeDot und den Optimierungsparametern von GCC. Mit Pruning und Retraining konnte Giese 72% aller Verbindungen entfernen ohne signifikanten Verlust in Erkennungsgenauigkeit. Das wiederholte ausführen von Quantisierung und Retrainieren erhöhte die Erkennungsgenauigkeit sogar etwas. Die beste Ausführungszeit wurde mit dem CSC-MA-Bit Format erzielt, dass unnötige Multiplikationen verhindert und die kleinste Programmgröße wurde mit dem CSC-Centroid Format erzielt. SeeDot hat im Vergleich zum Ausgangsmodell sowohl Ausführungszeit, als auch Programmgröße verringert, hat aber die Erkennungsgenauigkeit signifikant verringert. Der Vorteil von SeeDot ist die geringe Zeit, die diese Optimierung benötigt. Der Optimierungsparameter O2 hat den besten Kompromiss zwischen Programmgröße und Ausführungszeit erzielt. Insgesamt hat die beste Lösung 35,7% weniger Speicher benötigt und die Ausführungszeit wurde von 26,1 ms auf 6,8 ms reduziert [Gie20].

Evaluierung von Entscheidungsbäumen

TODO Also talk somewhere about the training sets. Here we can also mention cherry picking, since I cant find any source that mentions something along those lines.

4.1 Infrastruktur

Der Kern der Arbeit war es viele Verschiedene Feature und Konfigurationen der Entscheidungsbäume zu untersuchen und zu testen. Aus diesem Grund habe ich es als nötig erachtet eine umfangreiche und dokumentierte Infrastruktur zu schaffen, die die Auswertung meiner Arbeit und folgenden Arbeiten vereinfacht. Die Infrastruktur umfasst die Erstellung eines Datenmodells und das Auswerten von Datenmengen unter verschiedenen Parsingmethoden. Außerdem wird die Generierung von synthetischen Daten vereinfacht und eine leicht zu erweiternde Architektur bereitgestellt um Feature zu definieren. All diese Funktionalitäten sind in Bibliotheken verpackt, die leicht in eigene Projekte zu integrieren sind (siehe Abbildung)



■ Abbildung 4.1: Abhängigkeiten der einzelnen Module.

4 EVALUIERUNG VON ENTSCHEIDUNGSBÄUMEN

4.1). Darauf aufbauend wurden außerdem einige Programme erstellt, um den Arbeitsablauf zu vereinfachen.

4.1.1 lib_gesture

lib_gesture definiert was eine Geste ist und die vorhandenen Gestentypen. Außerdem implementiert sie zwei Parsing-Methoden. Die erste parsed Gesten nach Annotation und die zweite nach Kubiks Algorithmus (siehe Sektion 3.2.1). Die Geste selber implementiert Methoden um synthetische Daten zu generieren.

- Rotation um 90°, 180° und 270°.
- NullGesten durch das Kombinieren der ersten Hälfte der Ausgangsgeste und der zweiten Hälfte von dessen Rotationen.
- Verschiebung der Pixel nach oben und unten für eine Links nach Rechts bzw. Rechts nach Links Geste und analog dazu eine Verschiebung nach links und rechts für die restlichen Gesten.
- Rotation der äußeren Pixel um Diagonale Gesten zu generieren.

4.1.2 lib_feature

```
pub trait Feature {  
    fn calculate(gesture: &Gesture) -> Self where Self: Sized;  
    fn marshal(&self) -> String;  
}
```

- **Listing 4.1:** Interface, um ein Feature zu implementieren.

lib_feature bietet ein einfaches Interface an um Feature mit einer Geste (siehe Listing 4.1) zu implementieren. Zurzeit sind 28 verschiedene Feature implementiert.

4.1.3 lib_data_set

lib_data_set stellt alle verfügbaren Datenmengen als statische Importe bereit. Einträge sind bereits nach Distanz zur Kamera, Helligkeit, Verdeckungsobjekt und Ausführungs geschwindigkeit klassifiziert. Ein Eintrag kann in der Helligkeit verändert werden durch einen Offset oder indem er skaliert wird.

4.1.4 lib_evaluation

lib_evaluation bietet ein Hilfsobjekt an, dass Datenmengen nach Erkennungsgenauigkeit auswertet und Berichte daraus generiert.

4.1.5 Simulator

Der `Simulator` ist zweigeteilt. Der aktive Teil nutzt die die Gestenkandidatenerkennungsmethode nach Kubik, die in `lib_gesture` implementiert ist, um den seriellen Datenstrom des Arduino zu parsen. Der Gestenkandidat wird anschließend durch das hinterlegte Model klassifiziert und das Ergebnis ausgegeben. Der passive Teil evaluiert die Erkennungsgenauigkeit aller definierten Datenmengen.

4.1.6 Extractor

Der `Extractor` extrahiert aus spezifizierten Datenmengen die definierten Features und exportiert diese in Dateien, sodass sie von dem Model zum trainieren genutzt werden können. Optional kann die Datenmenge durch synthetische Daten erweitert werden.

4.1.7 Reader

Der `Reader` gibt den seriellen Datenstrom des Arduino aus.

4.1.8 Recorder

Der `Recorder` nutzt ähnlich wie der `Simulator` den seriellen Datenstrom des Arduino und die Gestenkandidaten-Parsingmethode von Kubik um Gestenkandidaten zu erkennen. Diese Information wird genutzt, um in eine vordefinierte Datei die Gesten reinzuschreiben. Um effizient Gesten aufzunehmen wurde der Ansatz von Kubik aufgegriffen mit einem Gestentyp zu starten und folgend immer zwischen dem Inverstyp hin und her zu wechseln [VKK⁺20]. Erweitert wurde das Programm um eine Option immer nur eine bestimmte Geste hintereinander aufzunehmen oder, jedes mal wenn eine Geste erkannt wurde, manuell den Gestentyp anzugeben. Mit diesem Programm wurde die Datenmenge `DymelData` in wenigen Stunden erstellt (siehe Sektion 4.2).

4.2 DymelData

`DymelData` ist eine Datenmenge, die mit dem `Recorder` (siehe Sektion 4.1.8) erstellt wurde. Sie umfasst insgesamt 14410 Gesten in unterschiedlichen Konfigurationen. Ich habe diese Datenmenge einerseits aufgenommen, um unter den in meinem Zimmer vorliegenden Lichtverhältnissen die Modelle miteinander vergleichen zu können und andererseits, um Test- und Trainingsdaten für NullGesten bereitzustellen. In den bisherigen Datenmengen waren nur wenige Einträge mit NullGesten hinterlegt.

4 EVALUIERUNG VON ENTSCHEIDUNGSBÄUMEN



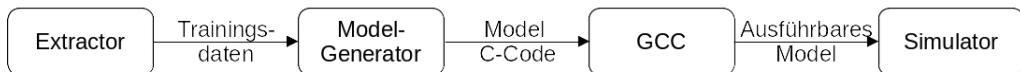
Abbildung 4.2: Verschiedene Helligkeitsstufen unter denen die Gesten von DymelData aufgenommen wurden.

4.2.1 Konfigurationen

Jede Geste wurde unter jeder Konfiguration ca. 100 mal aufgenommen bei 90 Bildern pro Sekunde. Insgesamt wurden in 3 Lichtverhältnisse und 4 Distanzen, 6 verschiedene Gesten (Links nach Rechts, Rechts nach Links, Oben nach Unten, Unten nach Oben und 2 NullGesten) jeweils schnell und langsam aufgenommen. Geringe Helligkeit war im Durchschnitt bei ca. 140, Halbe Helligkeit bei ca. 659, Hohe Helligkeit bei ca. 908. Alle waren relativ gleichmäßig ausgeleuchtet. Der Unterschied liegt in der Art der Lichtquelle. Während bei den Lichtquellen 4.2(a) und 4.2(b) relativ breit Licht gestreut hatten, war 4.2(c) eine Punktlichtquelle, wodurch besonders dort der Kontrast sehr stark ist. Die Gesten wurden in den Abständen 5 cm, 10 cm, 20 cm und 25 cm aufgenommen.

4.2.2 NullGesten

Insgesamt wurden 2 Typen von NullGesten aufgenommen. Die erste NullGeste geht *Oben* rein, verschieden weit in Richtung *Unten* und kehrt anschließend um, um bei *Oben* wieder rauszukommen. Die zweite NullGeste geht *Oben* rein, verschieden weit in Richtung *Unten* und anschließend *Rechts* wieder raus. Durch das Rotieren dieser Gesten wurden die Equivalente aus den anderen Richtungen inferiert. Insgesamt entstehen dadurch 19400 NullGesten.



■ Abbildung 4.3: Arbeitsablauf um ein Model zu trainieren und zu validieren.

4.2.3 Synthetische Helligkeitstestmenge

Um zu testen wie gut das Model gegenüber den Lichtverhältnissen sich generalisiert hat, ist es nötig mehr als nur 3 Helligkeitsstufen zu testen. Aus diesem Grund wurde aus der Gestenmenge mit den Lichtverhältnissen „Gering“ eine synthetische Testmenge generiert. Dabei wurden jeweils 20 Duplikate der Datenmenge erstellt mit einem Helligkeitsoffset zwischen 50 und 1000 und einer Skalierung zwischen 0,5 und 10 und zu einer Testmenge zusammengefügt.

4.3 Entscheidungsbaum basiertes Model

Entscheidungsbäume sind sehr schnell und ressourcenschonend im Vergleich zu neuronalen Netzen. Allerdings eignen sich neuronale Netze oft besser für ML Probleme, da sie bessere Ergebnisse erzielen. Aus diesem Grund wird untersucht, wie gut Entscheidungsbaum basierte Klassifizierer im Vergleich zu neuronalen Netzen sind. Insgesamt werden 67584 verschiedene Konfigurationen getestet. Sie unterscheiden sich in der Baumgröße, Waldgröße, Featureauswahl, Ensemble-Methode, sowie in Blattgröße und Quantifizierungsschwellenwert.

Jeder Entscheidungswald wird mit dem Python-Modul Scikit-Learn trainiert. Scikit-Learn implementiert den Konstruktionsalgorithmus CART (siehe Sektion 2.1.1) für den Entscheidungsbaum und bietet zusätzlich zahlreiche Ensemble-Methoden an. Jede Konfiguration folgt den in Abbildung 4.3 illustrierten Arbeitsablauf.

Zunächst wird die Trainingsmenge vorverarbeitet. Dabei werden die verschiedenen Features extrahiert, die während der Konstruktion eines Entscheidungsbaumes benötigt werden. Dann wird das Model mit Scikit-Learn und den gegebenen Konfigurationsparametern generiert. Anschließend wird aus dem Model ausführbarer C-Code generiert und kompiliert. Zuletzt wird die Erkennungsgenauigkeit des ausführbaren Models auf der Testmenge von Klisch und Dymel ermittelt.

4.3.1 Training

Mit Scikit-Learn werden Wahl basierte Entscheidungswaldklassifizierer mit den in Sektion 2.2 genannten Ensemble-Methoden trainiert. Insgesamt werden Waldgrößen zwischen 1 bis

4 EVALUIERUNG VON ENTSCHEIDUNGSBÄUMEN

16 betrachtet und Maximalhöhen von den einzelnen Entscheidungsbäumen zwischen 1 bis 22. Außerdem werden die Blattgrößen, d. h. die minimale Anzahl von Trainingsdateneinträgen pro Blatt, 1, 2, 4 *und* 8 sowie die Quantifizierungsschwellenwerte, d. h. Zuwachs der Entscheidungsgenauigkeit pro Teilbaum, 0, 10^{-3} , 10^{-2} *und* 10^{-1} betrachtet.

Die letzten beiden Parameter verringern potentiell die Erkennungsgenauigkeit von den einzelnen Entscheidungsbäumen, sie verringern aber auch die Größe des Baumes signifikant. Dadurch können bessere Entscheidungswälder gefunden werden, die zuvor nicht in den Programmspeicher eines Arduino Boards gepasst haben (siehe Sektion 4.7).

Die Konstruktion der Entscheidungswälder ist nicht deterministisch. Zuallererst muss die Konstruktion eines einzelnen Entscheidungsbaumes nicht deterministisch sein, da selbst wenn immer die beste Teilung ausgewählt wird, können mehrere Teilungen auch gleich gut sein. Aus diesen müsste zufällig eine ausgewählt werden. Folglich ist jede Ensemble-Methode zufällig. Außerdem wählt die RandomForest-Methode zufällig eine Featureauswahl, und die Bagging-Methode partitioniert die Trainingsmenge zufällig. Aus diesem Grund kann in Scikit-Learn einen `random_state` zuweisen, der den *Seed* des unterliegenden Zufallszahlengenerators setzt.

Dementsprechend kann man die Konstruktion als Monte Carlo Algorithmus betrachten, d. h. wiederholte Ausführungen erhöhen die Wahrscheinlichkeit, dass das beste Ergebnis dieser Konfiguration gefunden wurde. Jede Konfiguration wird aus diesem Grund mit 140 verschiedenen `random_state` ausgeführt.

4.3.2 C-Code Generierung eines Entscheidungsbaumes

Das Model soll auf kleinen eingebetteten Systemen ausgeführt werden. Diese haben eine Toolchain um die Firmware zu generieren, die meistens auf der Programmiersprache C basiert. Dies trifft auch auf das in dieser Arbeit benutzten Arduino Board zu.

```
enum Knoten<T> {
    Blatt(Vec<usize>),
    Elternknoten {
        test: (features: Vec<T>) -> bool,
        knoten_links: Knoten<T>,
        knoten_rechts: Knoten<T>
    }
}
```

■ **Listing 4.2:** Skizze der rekursiven Datenstruktur für Entscheidungsbäume die von Scikit-Learn genutzt wird.

Das Model wird mit dem Python-Modul Scikit-Learn generiert. Dementsprechend muss aus der internen Repräsentierung von Scikit-Learn das Model extrahiert werden. Scikit-Learn definiert eine rekursive Datenstruktur in der jedes Blatt für jede Klassifizierungsklasse die Anzahl der Trainingsdateneinträge enthält, die nach dem traversieren aller Test in diesem Blatt eingeordnet werden. Jeder Elternknoten besteht aus einem Test der ein Feature mit einem Schwellenwert vergleicht und einen Knoten für jedes Ergebniss dieses Tests (siehe Listing 4.2). Im Falle von Scikit-Learn ist der der Test immer ein \leq Vergleich, weswegen es genau zwei Kindknoten gibt.

```
if (features[k] <= X) {
    Traversiere Kind Links...
} else {
    Traversiere Kind Rechts...
}
```

■ Listing 4.3: C-Code eines Elternknotens.

Ein Elternknoten wird dementsprechend als ein `if (test) { ... } else { ... }` Ausdruck modelliert (siehe Listing 4.3). Dabei ist der `test` ein \leq Vergleich eines Features mit einem Schwellenwert und der Inhalt der einzelnen Blöcke ist abhängig von den Kindesknoten.

```
result[0] = (Anzahl Klasse 1) / (Gesamtanzahl im Blatt);
...
result[N] = (Anzahl Klasse N) / (Gesamtanzahl im Blatt);
return;
```

■ Listing 4.4: C-Code eines Blattes.

Der C-Code im Blatt ist abhängig von dem Wahlklassifizierer. Man kann entweder die Klasse auswählen, die von den meisten Bäumen klassifiziert wurde, oder die Erkennungswahrscheinlichkeiten jedes Baumes im Ensamble wird summiert und davon wird die Klasse mit der größten Summe ausgewählt (siehe Sektion 2.2.1). In dieser Arbeit wurde sich für letzteres entschieden. Im C-Code wird das modelliert durch die Zuweisung der Wahrscheinlichkeiten der einzelnen Klassen im Blatt zu dem Ergebnisparameter `result` (siehe Listing 4.4).

4.3.3 C-Code Generierung eines Entscheidungswaldes

Ein Entscheidungswald besteht aus einem Ensamble von Entscheidungsbäumen. Bei der Evaluierung eines Entscheidungswaldes wird der jeder Entscheidungsbaum evaluiert und die Ergebnisse zusammengefasst, z. B. durch einen Wahlklassifizierer.

```
function tree_i(float* features, float* result);
```

■ Listing 4.5: C-Code Funktionskopf eines Baumes i .

4 EVALUIERUNG VON ENTSCHEIDUNGSBÄUMEN

Zunächst wird jeder Entscheidungsbaum als Funktion isoliert (siehe Listing 4.5). Als Eingabeparameter dienen die extrahierten Features und ein Float-Array `result`, dass das Ergebnis speichert.

```
float tree_res[N] = { 0.0, ..., 0.0 };
float total_res[N] = { 0.0, ..., 0.0 };
unsigned char result_map[N] = { ... };

// Wiederhole dies für K Bäume
tree_i(features, tree_res);
total_res[0] += tree_res[0];
...
total_res[N-1] += tree_res[N-1];

unsigned char max_index = 0;
float max_value = 0;
for (unsigned char i = 0; i < N; ++i) {
    if (max_value < total_res[i]) {
        max_value = total_res[i];
        max_index = i;
    }
}
return result_map[max_index];
```

■ **Listing 4.6:** C-Code des Wahlklassifizierers mit N Klassen und K Bäumen.

Listing 4.6 zeigt wie ein Ensamble bestehend aus K Entscheidungsbäumen, die jeweils N mögliche Klassifizierungsergebnisse zurückgeben, mit der Wahlklassifizierungsmethode evaluiert wird. Zunächst wird jeder Baum evaluiert und die Ergebnisse summiert. Anschließend wird die Klasse mit dem maximalen Wert zurückgegeben.

4.4 Features

* Manuelle Feature selection (TODO: Lieber bei features rein?) * RandomForest und ExtraTree machen das schon * Eigentlich sind unsere Featuresets keine Featuresets sondern einzelne Feature

4.4.1 Requirements

Invariance to velocity

Development over time

Sense of direction

Sense of position

4.4.2 Curse of dimensionality

4.4.3 Unpromising features

Explain that they had been tried out and did work out. Explain why

4.4.4 Brightness distribution

Explain what it is. Note this as "discretized" position. Explain what requirements it fulfills. Explain different variants.

4.4.5 Motion History

Explain origin and how it works. Explain what requirements it fulfills and why.

Implementation

4.4.6 Center of Gravity Distribution

Motivate.?

Center of Gravity

Integer variant

Adding development over time

Window size

Also talk about the squishing?

4.5 Performance Evaluation

Talk about what was searched for, i.e. not all sizes had been tried out. Talk about different testset sizes? Talk about the greedy nature of the training algorithm? (Was that done in Cherry Picking?)

4 EVALUIERUNG VON ENTSCHEIDUNGSBÄUMEN

4.5.1 Testsets

Motivate. Explain what they are used for. Explain whats in there?

Metrics

Talk about used metrics

Klisch

DymelNull

DymelGesture

4.5.2 Feasible solution

4.5.3 Considering NullGestures

4.5.4 Brightness Distribution

Show graphs about: Best solution, Best feasible solution, With and WithOUT considering null gestures. Talk when it starts to generalize more poorly(?)

4.5.5 Motion History

Show graphs about: Best solution, Best feasible solution, With and WithOUT considering null gestures. Talk when it starts to generalize more poorly(?)

4.5.6 Brightness Distribution and Motion History

Show graphs about: Best solution, Best feasible solution, With and WithOUT considering null gestures. Talk when it starts to generalize more poorly(?)

4.5.7 Center of Gravity Distribution Float Ansatz

Show graphs about: Best solution, Best feasible solution, With and WithOUT considering null gestures. Talk when it starts to generalize more poorly(?)

Talk about brightness distribution!!

4.5.8 Center of Gravity Distribution Integer Ansatz

Show graphs about: Best solution, Best feasible solution, With and WithOUT considering null gestures. Talk when it starts to generalize more poorly(?)

Talk about brightness distribution!!

4.5.9 Center of Gravity Distribution Float and Integer Ansatz

Show graphs about: Best solution, Best feasible solution, With and WithOUT considering null gestures. Talk when it starts to generalize more poorly(?)

Here state what was expected and state how it worked out! Talk about brightness distribution!! ==> Alternative approach => Stacking (?)

4.5.10 Comparison to previous work

4.6 Execution Time Evaluation

Talk about constrained micro controller. Say that more frames potentially give more insight about fast gestures or that a longer battery life can be achieved. Say that we only consider COCD here, because that has the best feasible solution.

4.6.1 WCEP and WCET

4.6.2 AVR Compiler for AtMega328p

4.6.3 Optimization Level

Say that we considered O2 since the feasible solution can be compiled in O2. And why its a good Kompromiss between O3 and Os

4.6.4 Feature extraction

4.6.5 Tree Evaluation

4.6.6 Forest Evaluation

Here mainly talk about voting.

4.6.7 Total Execution Time

4.7 Size Evaluation

Explain that scikit learn provides some hyperparameters that can be changed. Elaborate which exist. Say that I considered here two parameters because all ensemble methods supported that. Maybe other reasons.

4.7.1 CCP (TODO: Abrev.)

4.7.2 Minimum Leaf Sample Size

4 EVALUIERUNG VON ENTSCHEIDUNGSBÄUMEN

Conclusion

Do a summary. Should be brief but also name everything and its result more or less. Say the improvement compared to the old approach. (Or downsides).

Mention future work. => Stacking

5 CONCLUSION

Literaturverzeichnis

- [ATKI12] AHAD, Md Atiqur R. ; TAN, Joo K. ; KIM, Hyoungseop ; ISHIKAWA, Seiji: Motion history image: its variants and applications. In: *Machine Vision and Applications* 23 (2012), Nr. 2, S. 255–281
- [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
- [CCRB11] CHENG, Heng-Tze ; CHEN, An M. ; RAZDAN, Ashu ; BULLER, Elliot: Contactless gesture recognition system using proximity sensors. In: *2011 IEEE International Conference on Consumer Electronics (ICCE)* IEEE, 2011, S. 149–150
- [D⁺02] DIETTERICH, Thomas G. u. a.: Ensemble learning. In: *The handbook of brain theory and neural networks* 2 (2002), S. 110–125
- [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. (2018), 10, S. 1–53
- [Ent20] 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
- [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
- [Gie20] GIESE, Anton: Compression of Artificial Neural Networks for Hand Gesture Recognition. (2020), 10, S. 1–46
- [Kli20] KLISCH, Daniel: Training of Recurrent Neural Networks asMultiple Feed Forward Networks. (2020), 01, S. 1–39
- [Kub19] KUBIK, Philipp: Zuverlässige Handgestenerkennungmit künstlichen neuronalen Netzen. (2019), 04, S. 1–47
- [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
- [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

LITERATURVERZEICHNIS

- [Qui90] QUINLAN, J R.: Decision trees and decision-making. In: *IEEE Transactions on Systems, Man, and Cybernetics* 20 (1990), Nr. 2, S. 339–346
- [SG14] SINGH, Sonia ; GUPTA, Priyanka: Comparative study ID3, cart and C4. 5 decision tree algorithm: a survey. In: *International Journal of Advanced Information Science and Technology (IJAIST)* 27 (2014), Nr. 27, S. 97–103
- [SHL⁺19] SONG, Wei ; HAN, Qingquan ; LIN, Zhonghang ; YAN, Nan ; LUO, Deng ; LIAO, Yiqiao ; ZHANG, Milin ; WANG, Zhihua ; XIE, Xiang ; WANG, Anhe u. a.: Design of a flexible wearable smart sEMG recorder integrated gradient boosting decision tree based hand gesture recognition. In: *IEEE Transactions on Biomedical Circuits and Systems* 13 (2019), Nr. 6, S. 1563–1574
- [Ste09] STEINBERG, Dan: Chapter 10 CART: Classification and Regression Trees. (2009), 01, S. 179–201
- [VKK⁺20] VENZKE, Marcus ; KLISCH, Daniel ; KUBIK, Philipp ; ALI, Asad ; DELL MISSIER, Jesper ; VOLKER, Turau: Artificial Neural Networks for Sensor Data Classification on Small Embedded Systems. (2020), 08, S. 1–25

Content of the DVD

In this chapter, you should explain the content of your DVD.