

Forschungsprojekt und Seminar

Computer Science

# Handgestenerkennung mit Entscheidungsbäumen

von

Tom Dymel

Februar 2021

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

Zweitprüfer	Dr. Marcus Venzke Institute of Telematics Hamburg University of Technology
-------------	--



## Eidesstattliche Erklärung

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

Hamburg, 1. Februar 2021

Tom Dymel



# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>1</b>
<b>2 Entscheidungsbäume</b>	<b>3</b>
2.1 Einzelne Entscheidungsbäume . . . . .	4
2.2 Ensemble-Methoden . . . . .	5
<b>3 Gestenerkennung</b>	<b>9</b>
3.1 Gestenerkennung mit Entscheidungsbäumen . . . . .	9
3.2 Optische Handgestenerkennung . . . . .	10
3.2.1 Exrahieren von Gestenkandidaten . . . . .	11
3.2.2 Skalieren des Gestenkandidaten . . . . .	13
3.2.3 Trainings- und Testdaten . . . . .	13
3.2.4 Gestenerkennung mit künstlichen neuronalen Netzen . . . . .	14
<b>4 Entscheidungsbaum basierte Handgestenerkennung</b>	<b>17</b>
4.1 Das Modell . . . . .	17
4.1.1 Training . . . . .	18
4.1.2 C-Code Generierung eines Entscheidungsbaumes . . . . .	19
4.1.3 C-Code Generierung eines Entscheidungswaldes . . . . .	20
4.2 Features . . . . .	21
4.2.1 Feature Verbesserungen . . . . .	21
4.2.2 Featureauswahl . . . . .	22
4.3 Infrastruktur . . . . .	25
4.4 Aufgenommene Datenmenge . . . . .	27
<b>5 Evaluation</b>	<b>29</b>
5.1 Erkennungsgenauigkeit . . . . .	29
5.1.1 Helligkeitsverteilung . . . . .	30
5.1.2 Motion History . . . . .	31
5.1.3 Schwerpunktverteilung mit Gleitkommazahlen . . . . .	33
5.1.4 Schwerpunktverteilung mit Integer . . . . .	34
5.1.5 Kombinierte Schwerpunktverteilung . . . . .	36
5.1.6 Robustheit gegenüber Lichtverhältnisse . . . . .	37
5.2 Ausführungszeit . . . . .	39
5.2.1 Feature-Extrahierung . . . . .	39
5.2.2 Evaluation eines Entscheidungsbaumes . . . . .	40
5.2.3 Evaluation eines Entscheidungswaldes . . . . .	41
5.3 Programmgröße . . . . .	41
5.3.1 Maximierung des Zuwachses der Erkennungsgenauigkeit . . . . .	42
5.3.2 Minimierung der Instruktionen eines Vergleichs . . . . .	43
5.3.3 Minimierung der Instruktionen einer Rückgabe . . . . .	44

## INHALTSVERZEICHNIS

<b>6 Diskussion</b>	<b>47</b>
<b>7 Schlussfolgerungen</b>	<b>49</b>
<b>A Inhalt des USB-Sticks</b>	<b>53</b>
<b>Literaturverzeichnis</b>	<b>55</b>

# Einleitung

Maschinelles Lernen (ML) gewann in den vergangenen Jahren an Popularität, u.a. durch die Fortschritte in parallelen Rechnen, sinkende Speicherpreise und schnelleren Speicher. Zudem sind sehr gute ML-Bibliotheken frei verfügbar, wie Scikit-Learn, Keras oder PyTorch, die den Einstieg in maschinellen Lernen erleichtern erleichtern (TODO: Quelle?). Ein namhaftes Beispiel für das Potential von maschinellen Lernen ist *AlphaZero*, die einen Sieg gegen den besten menschlichen Spieler im Brettspiel Go erringen konnte. Das galt als besonders schwierig für Computer zu meistern, da der Suchraum von möglichen Aktionen sehr groß ist (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 Mikrocontroller sind jedoch häufig nicht ausreichend leistungsstark, um ein trainiertes Modell in passabler Zeit auszuführen (TODO: Quelle). Gründe dafür sind Kosten oder Anforderungen an die Batterielanglebigkeit (TODO: Quelle). Häufig wird dieses Problem umgangen, indem die Modelle in leistungsstarken Rechen-Clustern ausgeführt werden. Dabei werden die nötigen Daten auf dem Mikrocontroller gesammelt und an den Rechen-Cluster gesendet. Nachteile dieses Ansatzes sind einerseits die Abhängigkeit zu dieser Infrastruktur und andererseits kommt dadurch eine Latenz hinzu. Alternativ können die Modelle lokal ausgeführt werden. Dies erfordert aber, dass die Komplexität des Modells reduziert wird, sodass eine passable Ausführungszeit gewährleistet wird.

In dieser Arbeit wird die Entscheidungsbaum basierte Handgestenerkennung auf kleinen Mikrocontrollern untersucht. Vermutet wird, dass Entscheidungsbäume schneller sind als neuronale Netze (NN) und trotzdem eine passable Erkennungsgenauigkeit erzielen. Untersucht werden muss welcher Entscheidungsbaum basierte Klassifizierer und welche Feature sich am besten eignen. Maßgeblich dafür ist die Leistung im Hinblick auf Erkennungsgenauigkeit,

## 1 EINLEITUNG

Ausführungszeit und Resourcenverbrauch des Modells auf dem Mikrocontroller. Dafür ist ein Konzept zur Übersetzung des Modells auf den Mikrocontroller nötig.

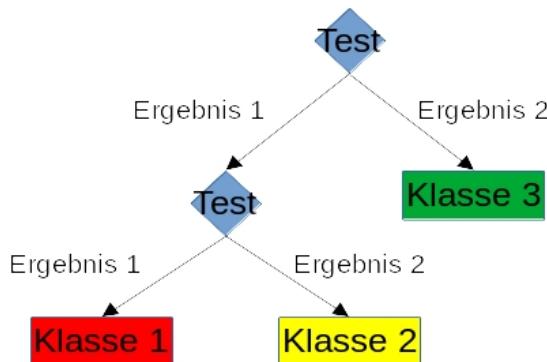
Insgesamt wurden 28 Varianten von Feature analysiert. Die Feature mit der besten Tauglichkeit wurden mit verschiedenen Ensemble-Methoden und Parametern für Entscheidungsbäume kombiniert. Daraus sind 22528 verschiedene Konfigurationen entstanden, die auf ihre Erkennungsgenauigkeit hin verglichen wurden. Die beste Konfiguration wurde auf ihre Worst-Case-Execution-Time (WCET) und auf den Resourcenverbrauch hin analysiert und mit den vorherigen Arbeiten verglichen. Dabei wurden zusätzliche Optimierungen diskutiert um den WCET und Resourcenverbrauch zu minimieren. Währenddessen ist eine komplexe Infrastruktur entstanden, die in Rust und Python geschrieben ist. Sie umfasst das Handgesten Modell und stellt Code-Bibliotheken bereit, um die verschiedenen Konfigurationen zu generieren, zu trainieren und zu validieren. Zudem stellt sie verschiedene Werkzeuge zur Verfügung. Ein wichtiges Werkzeug ist ein Codegenerator der aus den Entscheidungsbäumen basierten Modellen C-Code erzeugt. Mit einem anderen Werkzeug wurden innerhalb kürzester Zeit 14410 weitere Handgesten erfasst.

Kapitel 2 führt in Entscheidungsbäume und Ensemble-Methoden ein. Kapitel 3 erläutert die bisherigen Arbeiten zur Handgestenerkennung. In Kapitel 4 wird auf die Generierung des Models, die Tauglichkeit von Features und die Infrastruktur eingegangen, sowie die neu erstellte Trainings- und Testmenge erläutert. Darauf folgt die Evaluation der Erkennungsgenauigkeit, Ausführungszeit und des Resourcenverbrauchs in Kapitel 5. Kapitel 6 enthält einen kritischen Rückblick auf die Entscheidungen dieser Arbeit bevor Kapitel 6 Schlussfolgerungen zieht.

## Entscheidungsbäume

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

Die Konstruktion eines optimalen binären Entscheidungsbaumes ist NP-Vollständig [LR76]. Aus diesem Grund werden bei der Konstruktion Heuristiken verwendet, die nur lokal die beste Entscheidung treffen. Folglich ist es sehr aufwändig den optimalen Entscheidungsbaumklassifizierer zu finden. Ensemble-Methoden konstruieren eine Menge von Klassifizierern, dessen Ergebnisse zusammengefasst werden um die finale Entscheidung zu treffen [D<sup>+</sup>02].



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

## 2 ENTSCHEIDUNGSBÄUME

### 2.1 Einzelne Entscheidungsbäume

Der einzelne Entscheidungsbaum ist eine rekursive Datenstruktur um Entscheidungsregeln darzustellen. Jedem inneren Knoten ist ein *Test* zugeordnet, der eine arbiträre Anzahl von sich gegenseitig ausschließenden Ergebnissen hat. Das Ergebnis bestimmt mit welchem Kindknoten fortgefahrene wird [Qui90]. Abbildung 2.1 zeigt einen Entscheidungsbaum, indem jeder Test zwei mögliche Ergebnisse hat. Das wird als binärer Entscheidungsbaum bezeichnet.

Beim maschinellen Lernen werden aus mit Klassen beschrifteten Trainingsmengen Entscheidungsbäume generiert. Dabei wird die Trainingsmenge bestmöglich partitioniert, sodass die Blätter möglichst nur Einträge enthält die mit der gleichen Klasse beschriftet sind. Dabei wird erhofft, dass der Entscheidungsbaum möglichst gut generalisieren kann, d. h. möglichst allgemeingültige Tests hat, die auf alle möglichen Daten zutreffen und nicht nur auf die Trainingsmenge (TODO: Quelle).

Die Fähigkeit zu Generalisieren ist stark abhängig wie repräsentativ die Trainingsmenge ist und die Art und Weise, wie verschiedene Klassen in der Gesamtmenge unterschieden wird. Die Basis zum Unterscheiden bieten sogenannte *Feature*. Ein Feature kann ein Attribut sein oder eine berechnete Konsequenz aus mehreren Attributen der Rohdaten, z. B. der Durchschnitt oder das Maximum. Es ist nicht Aufgabe des Entscheidungsbaums aus den Rohdaten diese Feature zu extrahieren, sondern der Entscheidungsbaum nutzt eine vorgegebene Featuremenge die mit einer Klasse beschriftet ist (TODO: Quelle).

Es gibt verschiedene Algorithmen um Entscheidungsbäume zu erzeugen: ID3, C4.5, C5, CART, CHAID, QUEST, GUIDE, CRUISE and CTREE [SG14](TODO: Quellen für alle Algos im Detail). Das Grundprinzip der Partitionierung ist bei allen das Gleiche. Sie unterscheiden sich aber in den verwendeten Metriken (TODO: Ist das wirklich so?).

In dieser Arbeit wird die Python ML-Bibliothek *Scikit-Learn* verwendet. Sie implementiert eine optimierte Version des CART (Classification and Regression Trees) Algorithmus [Ent20a] und eine große Anzahl von Ensemble-Methoden [PVG<sup>+</sup>11].

CART ist ein Greedy-Algorithmus, d. h. ein Algorithmus der lokal immer, auf basis einer Bewertungsfunktion, die beste Entscheidung wählt. CART partitioniert die Trainingsmenge und wählt dabei immer lokal die beste Teilung aus.

```
BEGIN:  
Assign all training data to the root node  
Define the root node as a terminal node
```

```

SPLIT:
New_splits=0
FOR every terminal node in the tree:
    If the terminal node sample size is too small or all instances in the node ↘
        belong to the same target class goto GETNEXT
    Find the attribute that best separates the node into two child nodes using ↘
        an allowable splitting rule
    New_splits+=1

GETNEXT:
NEXT

```

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

Listing 2.1 skizziert den vereinfachten Baumwachstumsalgorithmus von CART. Der Algorithmus teilt die Trainingsmenge solange, bis keine weitere Teilung mehr möglich ist oder alle Einträge der mit der gleichen Klasse beschriftet sind. Folgend werden sukzessiv Teilbäume entfernt, die nach einer Bewertungsfunktion, z. B. Zuwachs der Erkennungsgenauigkeit, unterhalb eines vordefinierten Schwellenwert liegen [Ste09].

Scikit-Learn bietet zusätzlich noch weitere Parameter an um die Konstruktion zu steuern, wie eine Maximalhöhe, Minimale Anzahl von Einträgen pro Blatt oder Teilung, oder daer minimale Anteil einer Klasse um ein Blatt zu bilden [Ent20b].

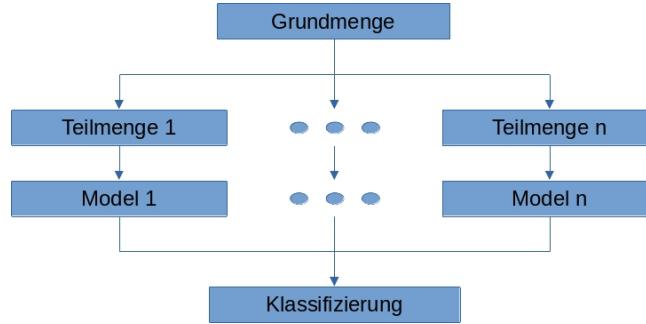
## 2.2 Ensemble-Methoden

Die Ensemble-Methoden beschreiben wie verschiedene Entscheidungsbäume trainiert werden, um eine möglichst hohe Diversität zu erzielen. Die Klassifizierungsergebnisse der einzelnen Entscheidungsbäume werden dann zu einem Ergebnis zusammengefasst [D<sup>+</sup>02].

Der „Wahl“-Klassifizierer  $H(x) = w_1 h_1(x) + \dots + w_K h_K(x)$  fasst eine Menge von Lösungen  $\{h_1, \dots, h_K\}$  zusammen mit Hilfe einer Menge von Gewichten  $\{w_1, \dots, w_K\}$ , die in der Summe 1 ergeben. 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<sup>+</sup>02].

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.

## 2 ENTSCHEIDUNGSBÄUME



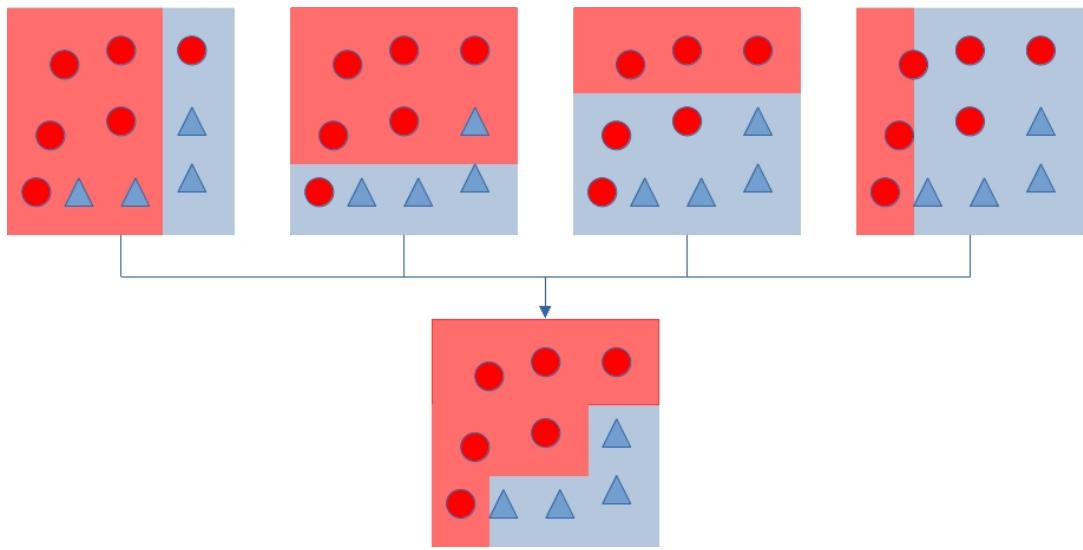
**Abbildung 2.2:** Klassifizierungsprozess mit der Bagging-Methode.

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.

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].

Extremely Randomized Trees gehen im Vergleich zu Random Forest 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].

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 mit einer Wahrscheinlichkeit  $1 - \delta$  und einem Fehler von bis zu  $\epsilon$  klassifizieren kann, wobei  $\epsilon, \delta > 0$ . 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].



■ **Abbildung 2.3:** Klassifizierungsprozess mit der Boosting-Methode.

In Abbildung 2.3 wird illustriert wie vier 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 ensteht. In dieser Arbeit wird im speziellen der Boosting Algorithmus **AdaBoost** von Freund und Schapire verwendet [FS97].

## 2 ENTSCHEIDUNGSBÄUME

# Gestenerkennung

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. Nicht-optische Ansätze bedienen sich anderen Sensoren, z. B. Infrarot Abstandssensoren, oder nutzen technische Hilfsmittel um zusätzliche Daten zu erfassen.

Im Bereich der optischen Handgestenerkennung auf kleinen Mikrocontrollern hat das Institut für Telematik von der Technischen Universität Hamburg Harburg (TUHH) bereits mehrere ML Ansätze untersucht. Es konnten Gestenkandidaten zuverlässig erkannt werden und davon bis 98% korrekt klassifiziert werden. Auch Nullgesten, d. h. invalide Gesten, konnten zuverlässig erkannt werden. Das verwendete neuronale Netz benötigte 6,8 ms zur Evaluierung auf dem Arduino Board ATmega328P.

## 3.1 Gestenerkennung mit Entscheidungsbäumen

Song et al. [SHL<sup>+</sup>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 Entscheidungsbauw 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

### 3 GESTENERKENNUNG

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<sup>+</sup>19].

dominate Bewegungen die kürzlich verarbeitet wurden heller angezeigt als nicht dominate Bewegungen oder Bewegungen die schon länger zurück liegen.

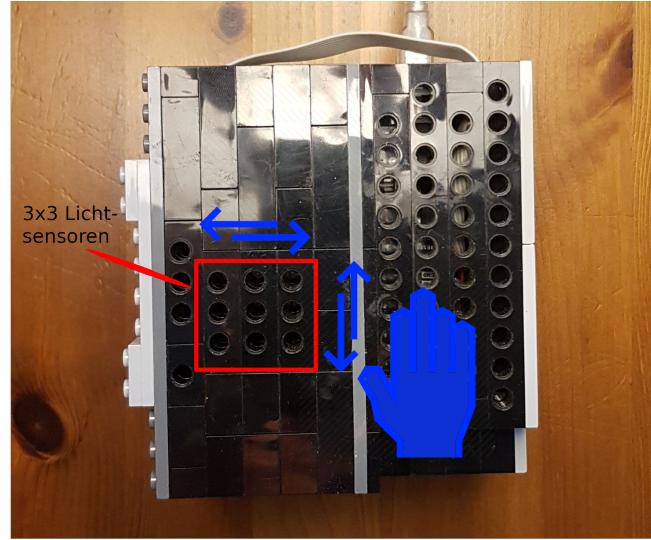
$$H_\tau(x, y, t) = \begin{cases} \tau & \text{if } \psi(x, y, t) = 1 \\ \max(0, H_\tau(x, y, t-1) - \delta) & \text{otherwise} \end{cases} \quad (3.1)$$

Das MHI kann sequentiell berechnet werden. Initial sind alle Werte 0. Wenn  $\psi(x, y, t)$  eine dominate Bewegung in einem Pixel  $(x, y)$  zu einem Zeitpunkt  $t$  signalisiert, dann wird der Pixel zum Maximalwert  $\tau$  gesetzt. Mit jedem Bild in denen keine dominate Bewegung im Pixel  $(x, y)$  stattgefunden hat, wird der Wert um den Zerfallswert  $\delta$  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  $\psi$ ,  $\tau$  und  $\delta$ . MHI ist besonders anfällig für Bildfolgen mit verschiedener Länge. Jenachdem wie  $\tau$  und  $\delta$  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 Mikro-Controllern von dem Institut für Telematik an der TUHH [VKK<sup>+</sup>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



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

nach Rechts, Rechts nach Links, Oben nach Unten, Unten nach Oben und NullGeste, i. 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 Exrahieren 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 Helligkeit 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 veränderten 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

### 3 GESTENERKENNUNG



■ **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“.

diesem Grund schlug Kubik zusätzlich vor am Anfang und Ende weitere 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 dem 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-ENDING-FRAMES** über geht, indem die letzten 3 Bilder an den Puffer angehängt werden. Sobald 3 Bilder angehängt wurde, wird der Puffer dem Klassifizierer übergeben und anschließend der Zustandsautomat zurückgesetzt, woraufhin der Automat in den initialen Zustand wieder übergeht.

### 3.2.2 Skalieren 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 des NN 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, wenn weniger als 20 Bilder vorhanden sind. 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].

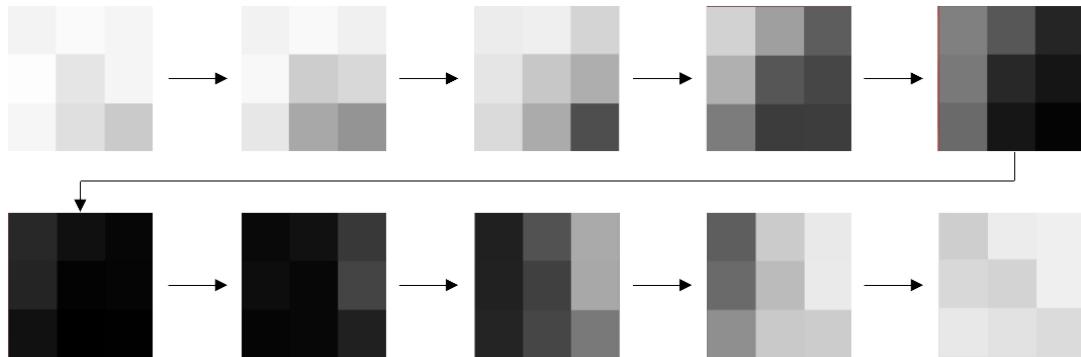
### 3.2.3 Trainings- und Testdaten

```
...
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.

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.

### 3 GESTENERKENNTUNG



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

Synthetische Daten können erzeugt werden, um die Datenmenge zu vergößern. Dabei wird aus einer aufgenommenen Geste Variationen durch Rotation und Rauschen generiert. Außerdem können Helligkeiten, Kontraste und Gamma verändert werden [VKK<sup>+</sup>20].

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].

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

#### 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 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) verwarf Engelhardt aufgrund der geringen Auflösung der Gesten und da die Faltung extrem Rechenaufwendig sei. Desweiteren verwarf 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.

### 3.2 OPTISCHE HANDGESTENERKENNUNG

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 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 nutzte 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 hat einen Ansatz von Venzke untersucht. Von Engelhardt motiviert schlug Venzke vor, dass man ein RNN als mehrere FFNNs trainieren könnte. Engelhardt stellte fest, dass RNNs schlechtere Erkennungsgenauigkeiten erzielen als FFNNs, da sie schwerer zu trainieren sind [Eng18]. Aus diesem Grund hat Klisch verschiedene Konfigurationen getestet und stellte 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 von 10% darstellt.

### 3 GESTENERKENNUNG

serung zu dem Ergebnis von Engelhardt ist. Klisch stellte fest, dass das sein Model schnell genug ist, um 50 Hz zu unterstützen [Kli20].

Der Fokus von Giese's 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 bessere 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 vermied und den 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].

# Entscheidungsbaum basierte Handgestenerkennung

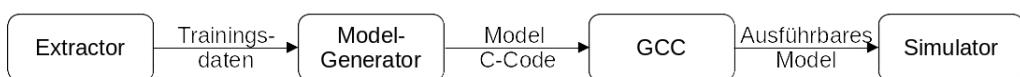
Entscheidungsbäume sind sehr schnell und ressourcenschonend im Vergleich zu neuronalen Netzen. Allerdings eignen sich neuronale Netze oft besser für komplexe ML Probleme, da sie leicht zu konstruieren sind und bereits auf Rohdaten gut generalisieren. Das untersuchte Problem weißt aber nur eine geringe Komplexität auf und es wird eine schnelle und ressourcenschonende Lösung gesucht. Aus diesem Grund wird untersucht, ob Entscheidungsbaum basierte Klassifizierer eine ausreichende Erkennungsgenauigkeit erzielen.

## 4.1 Das Model

Insgesamt werden 22528 verschiedene Konfigurationen getestet. Sie unterscheiden sich in der Baumgröße, Waldgröße, Featureauswahl, Ensemble-Methode und Blattgröße.

Jeder Entscheidungswald wird mit dem Python-Modul `Scikit-Learn` trainiert. Scikit-Learn implementiert den Konstruktionsalgorithmus CART (siehe Sektion 2.1) für den Entscheidungsbaum und bietet zusätzlich zahlreiche Ensemble-Methoden an. Jede Konfiguration folgt den in Abbildung 4.1 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.



■ **Abbildung 4.1:** Arbeitsablauf um ein Model zu trainieren und zu validieren.

## 4 ENTSCHEIDUNGSBAUM BASIERTE HANDGESTENERKENNUNG

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.

### 4.1.1 Training

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

Der letzte Parameter verringert potentiell die Erkennungsgenauigkeit von den einzelnen Entscheidungsbäumen, verringert 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 5.3).

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 Methode 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.

Zum trainieren wird eine Kombination aus der Trainingsmenge von Fey und Kubik verwendet, sowie 25% der Gestenmenge und 12,5% der Nullgestenmenge von Dymel (siehe Sektion 4.4). Insgesamt 7629 Ausführungen der Handgesten. Davon werden 50% zufällig zum trainieren genutzt und die überbleibenden 50% werden genutzt, um den besten Entscheidungswald aus den 140 verschiedenen `random_state` zu ermitteln.

### 4.1.2 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 4.1:** 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 4.1). 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 4.2:** C-Code eines Elternknotens.

Ein Elternknoten wird dementsprechend als ein `if (test) { ... } else { ... }` Ausdruck modelliert (siehe Listing 4.2). 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 4.3:** C-Code eines Blattes.

## 4 ENTScheidungsbaum basierte Handgestenerkennung

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). 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 4.3).

### 4.1.3 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 4.4:** C-Code Funktionskopf eines Baumes  $i$ .

Zunächst wird jeder Entscheidungsbaum als Funktion isoliert (siehe Listing 4.4). Als Eingabe-parameter 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 4.5:** C-Code des Wahlklassifizierers mit  $N$  Klassen und  $K$  Bäumen.

Listing 4.5 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.

## 4.2 Features

Der Entscheidungsbaum nutzt Features um die einzelnen Klassen voneinander zu unterscheiden. Dies funktioniert aber nur, wenn die Klassen klar trennbar sind. Ist eine Trennung nicht möglich, so ist auch keine gute Generalisierung von dem Entscheidungsbaum zu erwarten.

In dieser Arbeit muss die Richtung der Handbewegung erkannt werden. Die Bewegung kann mit verschiedenen Geschwindigkeiten durchgeführt werden. Aus diesem Grund sollte das Feature invariant gegenüber Geschwindigkeit sein. Außerdem wird die Handbewegung aus verschiedenen Distanzen zur Kamera durchgeführt. Dadurch variiert der Kontrast und die Helligkeit. Dementsprechend sollte das Feature invariant zu den Lichtverhältnissen sein. Die Richtung der Bewegung ist eine Kombination aus der derzeitigen Position der Hand und dem Zeitpunkt. Das Feature soll also Auskunft über die Entwicklung über die Zeit geben und die Position die die Hand zu diesen Zeitpunkten hatte.

Erschwerend ist, dass die Bewegung nie exakt gleich ausgeführt wird. Die Bewegung kann eine Kreisform haben, etwas schräg sein oder einige Fotowiderstände nicht verdecken. Das Feature muss dem gegenüber robust sein.

### 4.2.1 Feature Verbesserungen

Einige Anforderungen an ein Feature können durch Änderungen hinzugefügt werden. Relative Helligkeitsunterschiede können durch Normalisierung eliminiert werden, Positionen durch das Argument oder partielle Anwendung inferiert werden und die Entwicklung über Zeit durch die Duplizierung von Feature über Zeitfenster dargestellt werden.

Normalisierung ersetzt die Aussage über die absolute gegen die lokale Gesamthelligkeit. Dies erzeugt eine Invarianz gegenüber Skalierung der Helligkeiten jedoch nicht über einen Offset. Die Skalierung passt den Kontrast zwischen hellen und dunklen Stellen mit an, der Offset jedoch nicht.

Informationen über die Positionen können einerseits direkt aus dem Argument einer Funktion als Feature bereitgestellt werden, z. B.  $\arg(\max X)$ . Andererseits indirekt, indem das Feature dupliziert wird und auf Teilmengen der Definitionsmenge angewendet wird, z. B. die Berechnung eines Features für einzelne Spalten oder Zeilen.

Ähnlich zur Position kann auch die Entwicklung über Zeit durch das Dupizieren von Feature dargestellt werden. Dabei wird die Geste in eine bestimmte Anzahl von gleich großen Zeitfenstern eingeteilt. Für jedes Zeitfenster wird das Feature berechnet.

### 4.2.2 Featureauswahl

Insgesamt wurden 28 Varianten von Feature untersucht und davon 3 Feature verstärkt aus denen 20 Varianten entstanden sind. Die Features, die von Song et al. genutzt wurden (siehe Tabelle 3.1) eignen sich ohne Änderungen nicht, da sie mindestens eine Anforderung nicht erfüllen.

Mit dem *Mean absolute value* ermöglicht es die einzelnen Handgesten zu unterscheiden, wenn das Feature in verschiedene Zeitfenster aufgeteilt wird. Zusätzlich kann die Helligkeit normalisiert werden. Um die Featuremenge zu verringern, können Spalten und Zeilen zusammengefasst werden. Allerdings generalisierte der Ansatz nicht gut, da die Varianz sehr groß ist durch die fehlende Invarianz zur Geschwindigkeit.

*Average amplitude change* eignet sich gut um horizontale und vertikale Bewegungen zu unterscheiden. Allerdings ist es nicht möglich symmetrische Bewegungen zu unterscheiden. Nicht untersucht wurden Änderungen, die beim *Mean absolute value* durchgeführt wurden.

Feature 2, 5, 7 bis 9 wurden nicht weiter untersucht weil sie zu komplexe Berechnungen bedürfen für das Arduino Board.

### Motion History

Die Motion History zeigt eine Bewegunghistorie, indem kürzlich stattgefundene Bewegung heller ist als länger zurückliegende. Es ist invariant gegenüber Lichtverhältnisse, hat jedoch 2 große Schwachpunkte. Einerseits kann es überlappende Bewegungen nicht richtig anzeigen, da eine kürzlich detektierte Bewegung den Wert auf den Maximalwert  $\tau$  setzt. Dies stellt in diesem Anwendungsfall kein Problem dar, da die definierten Handgesten keine Überlappung erzeugen.

Andererseits ist das Feature bei konstantem  $\tau$  und  $\delta$  nicht Invariant gegenüber Geschwindigkeit. Als Lösung wurde  $\delta$  abhängig von der Gestenlänge gemacht, d. h.  $\delta = \frac{\tau}{\#Bilder}$ . Mit dieser Konfiguration ist die Bewegung nicht unvollständig, wenn sie langsam ausgeführt wird. Allerdings geht dieser Ansatz von einer konstanten Ausführungsgeschwindigkeit der Handgeste aus.

Eine Bewegung in einem Pixel  $q$  wird durch die Funktion 4.1 signalisiert, d. h. die Bewegung in  $q$  findet statt, wenn eine Veränderung oberhalb des Durchschnitts detektiert wird.

$$\phi(q, t) = \begin{cases} 1 & \text{if } \Delta_{q,t} \geq \frac{1}{N} \sum_{n=1}^N \Delta_{q,n} \\ 0 & \text{otherwise} \end{cases} \quad \text{where } \Delta_{q,t} = |q_t - q_{t-1}| \quad (4.1)$$

### Helligkeitsverteilung

Ein Pixel  $q$  ist am hellsten unter allen Pixeln in einem Bild  $Q$ , wenn  $q$  den höchsten Wert hat. Analog ist der dunkelste Pixel, der mit dem geringsten Wert. Folglich kann der hellste Pixel als  $q' = \arg(\max Q)$ , bzw. der dunkelste Pixel als  $q' = \arg(\min Q)$  definiert werden.

Weiterhin wird die Bildsequenz in eine bestimmte Anzahl von gleich großen Zeitfenstern aufgeteilt. In jedem Zeitfenster wird der hellste bzw. dunkelste Pixel ermittelt. Aus der daraus resultierenden Featuremenge kann jede definierte Handgeste inferiert werden. Sie ist invariant zu Lichtverhältnissen und Geschwindigkeiten. Per Definition gibt sie Auskunft über die Entwicklung über Zeit und die Position.

Es gibt mehrere Möglichkeiten die einzelnen Pixel in einem Zeitfenster zusammenzufassen.

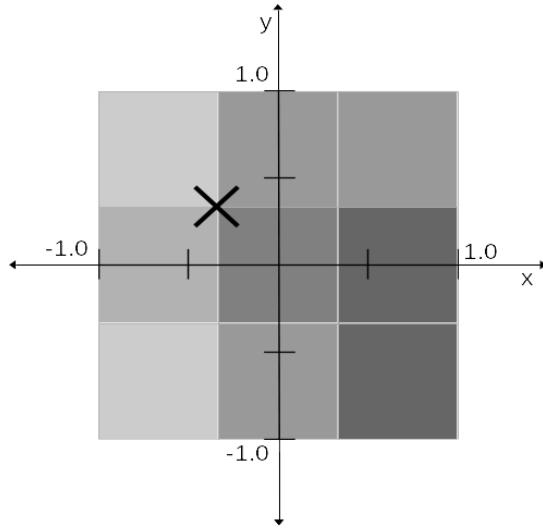
- Wähle das Minimum bzw. Maximum.
- Projizierte die Pixel auf ein kartesisches Koordinatensystem und fasse die Punkte über eine Abstandsmetrik zusammen, z. B. über den euklidischen Abstand.
- Unterteile die Pixel in Quadranten und wähle den Quadranten, der die meisten Einträge hat.

Außerdem können die Anzahl der Zeitfenster variiert werden und Pixel zu Gruppen zusammengefasst werden, d. h. Spalten und Zeilen.

### Schwerpunktverteilung

$$Q = \begin{pmatrix} q_{00} & q_{01} & q_{02} \\ q_{10} & q_{11} & q_{12} \\ q_{20} & q_{21} & q_{22} \end{pmatrix} \quad (4.2)$$

## 4 ENTSCHEIDUNGSBAUM BASIERTE HANDGESTENERKENNUNG



**Abbildung 4.2:** Illustration des Schwerpunktes im 3x3 Fotowiderstand-Array.

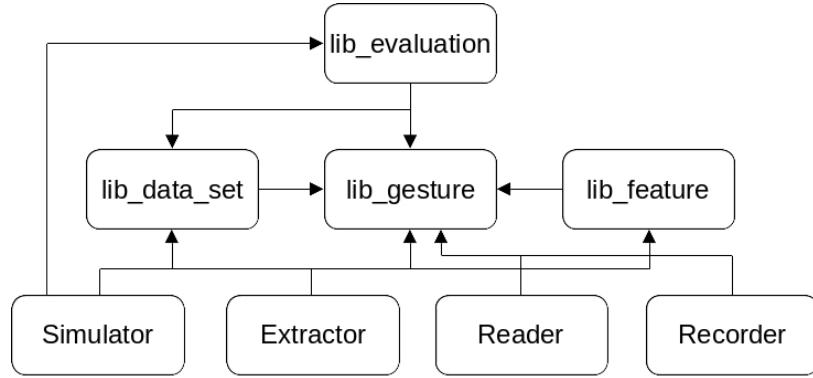
Der Schwerpunkt ( $X_Q, Y_Q$ ) in einem Bild  $Q$  (4.2) ist über die Helligkeit in den einzelnen Pixel definiert. Der Pixel  $q_{11}$  bildet den Nullpunkt des Koordinatensystems. Dann ist relativ zur Gesamthelligkeit  $P = \sum_{i,j} q_{i,j}$ ,  $X_Q = \frac{\sum_{i=0}^2 q_{i,2} - \sum_{i=0}^2 q_{i,0}}{P}$  die horizontale Komponente und  $Y_Q = \frac{\sum_{i=0}^2 q_{0,i} - \sum_{i=0}^2 q_{2,i}}{P}$  die vertikale Komponente des Schwerpunktes [VT20].

Ähnlich zur Helligkeitsverteilung wird das Feature mit einer Zeithistorie erweitert durch die multiple Anwendung auf verschiedene Zeitfenster, wobei die Anzahl der Zeitfenster variiert werden kann. Die einzelnen Schwerpunkte innerhalb eines Zeitfensters werden über den Durchschnitt zusammengefasst.

Sollte die Anzahl der Bilder einer Handgeste ein Vielfaches von der Anzahl der Zeitfenster sein, wird die gleiche Anzahl an Bildern auf jedes Zeitfenster verteilt. Ansonsten werden Überschüsse einem Muster nach bestimmten Zeitfenstern zugeordnet. Bei 5 Zeitfenstern wird der erste Überschuss dem letzten Zeitfenster zugeordnet, der zweite dem ersten, der dritte dem dritten, der vierte dem zweiten.

Die Schwerpunktverteilung ist durch die Dividierung mit  $P$  invariant gegenüber Skalierung der Helligkeit, jedoch nicht gegenüber einem Offset. Alternativ kann  $P$  weggelassen werden, damit ausschließlich mit Integer gearbeitet wird. Dadurch können größere Bäume generiert werden (siehe Sektion 5.3) und die Feature-Extrahierung ist schneller (siehe Sektion 5.2).

TODO: Erkläre Invarianz gegenüber Offset aber nicht Skalierung



■ Abbildung 4.3: Abhängigkeiten der einzelnen Module.

### 4.3 Infrastruktur

In dieser Arbeit mussten viele Feature und Konfigurationen der Entscheidungsbäume untersucht und getestet werden. Aus diesem Grund wurde eine umfangreiche Infrastruktur geschaffen, die die Auswertung von ML Modellen mit den Handgestendaten vereinfacht. Die Infrastruktur umfasst ein Datenmodell für Handgesten und kann die Datenmengen mit verschiedenen Parsingmethoden einlesen.

Außerdem können synthetischen Daten auf verschiedene Arten generiert werden. Die Architektur der Infrastruktur erlaubt es weitere Feature hinzuzufügen, ohne Kompatibilitätsprobleme zu verursachen. Alle Funktionalitäten sind in Bibliotheken isoliert, um die Integration in Hilfsprogramme zu vereinfachen (siehe Abbildung 4.3).

Im folgenden wird die Funktionalität der einzelnen Module vorgestellt und die daraus erstellten Hilfsprogramme.

`lib_gesture` definiert die Handgeste und die vorhandenen Gestentypen. Außerdem implementiert sie zwei Parsing-Methoden. Die erste Methode parsed Gesten nach Annotation und die zweite nach Kubiks Algorithmus (siehe Sektion 3.2.1). Die Handgeste 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.

## 4 ENTScheidungsbaum basierte Handgestenerkennung

- 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.

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

■ **Listing 4.6:** Interface, um ein Feature zu implementieren.

`lib_feature` bietet ein einfaches Interface an um Feature mit einer Geste (siehe Listing 4.6) zu implementieren. Zurzeit sind 28 verschiedene Feature implementiert.

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

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

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.

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.

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

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.



**Abbildung 4.4:** Verschiedene Helligkeitsstufen unter denen die Gesten von DymelData aufgenommen wurden.

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<sup>+</sup>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 4.4).

## 4.4 Aufgenommene Datenmenge

DymelData ist eine Datenmenge, die mit dem Recorder (siehe Sektion 4.3) erstellt wurde. Sie umfasst insgesamt 14410 Gesten in unterschiedlichen Konfigurationen. Sie wurde einerseits aufgenommen, um unter den vorort bestehenden Lichtverhältnissen die Modelle miteinander vergleichen zu können und andererseits, um Test- und Trainingsdaten für Nullgesten bereitzustellen. In den bisherigen Datenmengen enthält nur ein geringer Anteil Nullgesten.

Jede Handgeste 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. Die Gesten wurden in den Abständen 5 cm, 10 cm, 20 cm und 25 cm aufgenommen.

## 4 ENTScheidungsbaum basierte Handgestenerkennung

Die „Geringe“ Helligkeit war im Durchschnitt bei ca. 140, „Halbe“ Helligkeit bei ca. 659, „Hohe“ Helligkeit bei ca. 908. Alle Helligkeiten haben das 3x3-Array relativ gleichmäßig ausgeleuchtet. Bei den Lichtquellen 4.4(a) und 4.4(b) wurde eine Schirmlampe verwendet. Dadurch wurde das Licht relativ breit gestreut, wodurch der Kontrast mit vergrößender Distanz abgenommen hat. Bei 4.4(c) wurde eine Punktlichtquelle verwendet, wodurch der Kontrast über alle Distanzen sehr stark ist. Im weiteren Verlauf dieser Arbeit wird diese Testmenge ohne Nullgesten „Gestentestmenge“ genannt.

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. Die resultierenden Handgesten werden anschließend um 90°, 180° und 270° rotiert, um die äquivalenten Nullgesten aus den anderen Richtungen zu inferieren. Insgesamt entstehen dadurch 19400 Nullgesten. Im weiteren Verlauf dieser Arbeit wird diese Testmenge nur mit Nullgesten „Nullgestentestmenge“ genannt.

Um zu testen wie gut das Modell sich gegenüber verschiedene Lichtverhältnisse generalisiert hat, ist es nötig mehr als nur 3 Helligkeitsstufen zu testen. Aus diesem Grund wurde aus der Gestentestmenge mit der Helligkeit „Gering“ eine synthetische Testmenge generiert. Dabei wurden jeweils 16 Duplikate der Datenmenge erstellt mit einem Helligkeitsoffset zwischen 50 und 800 und einer Skalierung zwischen 1 und 7. Diese Datenmengen wurden zu einer Testmenge zusammengefügt. Im weiteren Verlauf dieser Arbeit wird diese Testmenge „Helligkeitstestmenge 1“ genannt.

Zusätzlich zu einer Testmenge, die den Kontrast immer weiter erhöht, bedarf es einer Testmenge, die bei gleichbleibender Helligkeit den Kontrast verringert. Aus diesem Grund wurde aus der Gestentestmenge mit der Helligkeit „Medium“ eine synthetische Testmenge generiert. Dabei wurden 19 Duplikate erstellt, die in 0,05 Schritten die Helligkeit runterskalieren. In jedem Schritt wird in gleichen Anteilen die Gesamthelligkeit auf jeden Pixel addiert. Dadurch wird der Kontrast zwischen dunklen und hellen Pixeln immer geringer. Im weiteren Verlauf dieser Arbeit wird diese Testmenge „Helligkeitstestmenge 2“ genannt.

# Evaluation

Kleine eingebettete Systeme weisen eine stark limitierte Hardware auf. Dafür sind sie aber sehr klein und verbrauchen wenig Energie. Verschiedene Konfigurationen wurden in dieser Arbeit untersucht. Die gefundene Lösung muss nicht nur eine hohe Erkennungsgenauigkeit erzielen, sondern auch auf luaffähig sein auf einem kleinen eingebetteten System.

Dieses Kapitel untersucht zuerst die Erkennungsgenauigkeit der besten Konfigurationen. Die beste Konfiguration wird anschließend auf die Ausführungszeit und Resourcenverbrauch auf dem ATmega328P hin analysiert. Dabei wird auf mögliche Optimierungen eingegangen, um die Ausführungszeit und den Resourcenverbrauch zu verringern.

## 5.1 Erkennungsgenauigkeit

Es werden 3 Features näher betrachtet. Motion History, Helligkeitsverteilung und Schwerpunktverteilung. Aus denen werden 4 Featuremengen generiert, die zum Trainieren genutzt werden. Insgesamt wurden 22528 verschiedene Konfigurationen trainiert und getestet (siehe Sektion 4.1.1).

Betrachtet wird die beste Konfiguration, die innerhalb der Restriktion des Programmspeichers, nach möglichen Optimierungen (siehe Sektion 5.3), die Summe der Erkennungsgenauigkeiten der Testmenge von Klisch, der Gestentestmenge und der Nullgestentestmenge maximiert.

Die verschiedenen Featuremengen werden im Hinblick auf die Erkennungsgenauigkeit auf der Testmenge von Klisch, der Nullgestentestmenge, der neuerstellten Gestentestmenge und den synthetischen Helligkeitstestmengen untereinander verglichen und mit den Ergebnissen von Giese. Außerdem wird die Auswirkung von verschiedenen Waldgrößen untersucht. Anzumerken ist, dass lediglich die Testmenge von Klisch vergleichbar ist mit den vorherigen

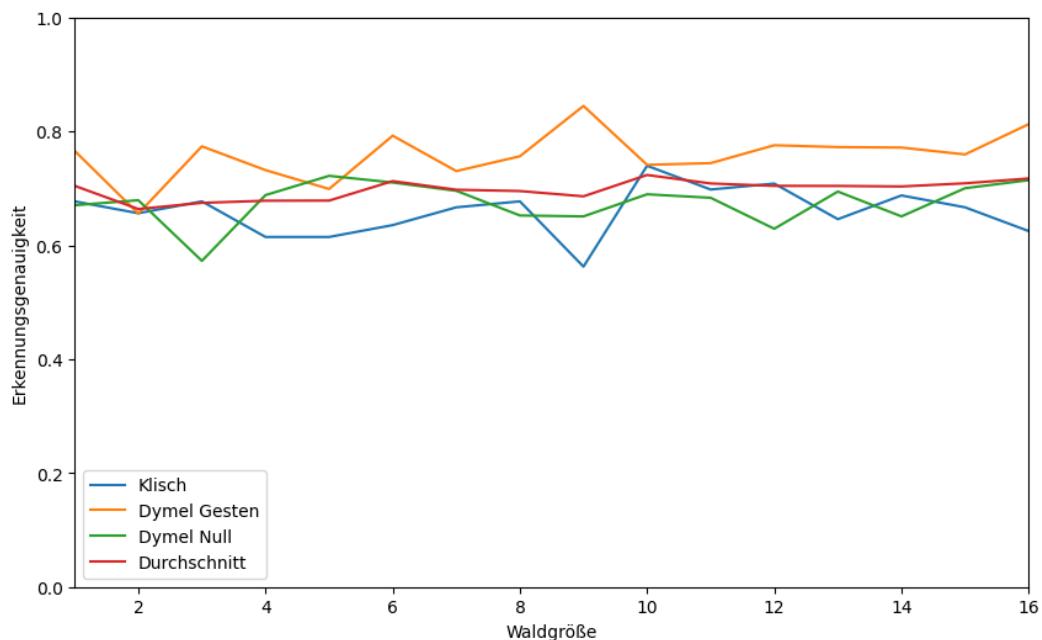
## 5 EVALUATION

Arbeiten. Die Gestentestmenge, Nullgestentestmenge und Helligkeitstestmengen sind erst im Laufe dieser Arbeit entstanden.

### 5.1.1 Helligkeitsverteilung

Konfiguration	Beste	Unter 60 kB	Unter 28 kB
Ensemble-Methode	ExtraTrees	ExtraTrees	ExtraTrees
Maximalhöhe	14	10	15
Waldgröße	10	6	1
min_samples_leaf	4	4	4
Programmgröße in Bytes	76628	33284	9364
Genauigkeit Testmenge von Klisch	74,0%	63,5%	67,7%
Genauigkeit Gestentestmenge	74,1%	79,2%	76,6%
Genauigkeit Nullgestentestmenge	69,0%	71,0%	67,0%

■ **Tabelle 5.1:** Beste Konfigurationen der Helligkeitsverteilung.



■ **Abbildung 5.1:** Die beste summierte Erkennungsgenauigkeit pro Waldgröße mit der Helligkeitsverteilung.

Die Featuremenge der Helligkeitsverteilung beinhaltet insgesamt 12 Features. Jeweils 6 Feature repräsentieren Zeitfenster der Minimalen Helligkeit und der Maximalen Helligkeit. Die Zeitfenster wurden geometrisch zusammengefasst.

Die beste Konfiguration wurde mit der ExtraTrees Ensemble-Methode gefunden (siehe Tabelle 5.1). Sie erzielt eine Erkennungsgenauigkeit von 82,3% auf der Testmenge von Klisch und ist damit nur 17,7% schlechter als das neuronale Netzwerk von Giese [Gie20]. Außerdem klassifiziert diese Konfiguration 61% aller Nullgesten aus der Testmenge von Dymel korrekt und ist damit in Summe deutlich besser als die anderen Ensemble-Methoden. Allerdings ist die Erkennungsgenauigkeit auf der Gestentestmenge von Dymel nur bei 39,2%, wobei RandomForest deutlich besser ist mit einer Erkennungsgenauigkeit von 55,4%. Insgesamt hat RandomForest damit ein besseres balanciertes Ergebnis, da die Erkennungsgenauigkeit auf der Nulltestmenge und Testmenge von Klisch nur leicht schlechter sind als die der Extra-Tress Methode. Mit einer Programmgröße von 4560 Bytes und 25844 Bytes passen beide Konfigurationen, ohne Optimierungen vorzunehmen, auf das Arduino Board.

Abbildung 5.1 zeigt keinen nennenswerten Anstieg der Erkennungsgenauigkeit durch eine größere Waldgröße.

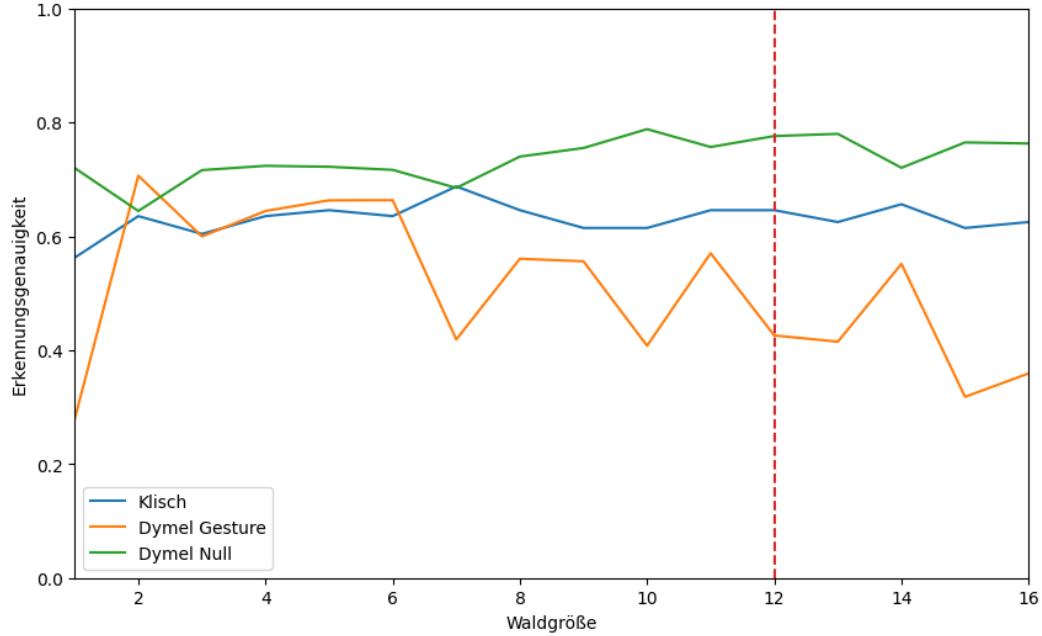
### 5.1.2 Motion History

Konfiguration	Beste Unter 50 kB	Unter 28 kB
Ensemble-Methode	ExtraTrees	
Maximalhöhe	10	
Waldgröße	11	
ccp_alpha	0.0	
min_samples_leaf	1	
Programmgröße in Bytes	-1	
Genauigkeit Testmenge von Klisch	68.8%	
Genauigkeit Gestentestmenge	71.5%	
Genauigkeit Nullgestentestmenge	69.4%	

■ **Tabelle 5.2:** Beste Konfigurationen der TODO.

Die Featuremenge von Motion History beinhaltet für jeden Pixel einen Eintrag, die der Definition des Motion History Image folgen (siehe Formel 3.1).

## 5 EVALUATION



**Abbildung 5.2:** Die beste summierte Erkennungsgenauigkeit pro Waldgröße mit Motion History.

Zu erwarten war, dass Motion History am besten von allen Konfigurationen abschneidet, da es alle Anforderungen an ein Feature erfüllt. In der Praxis schneidet die Motion History allerdings am schlechtesten im Hinblick auf die Erkennungsgenauigkeit der Testmenge von Klisch ab (siehe Tabelle ??). Die beste Ensemble-Methode ist ExtraTrees mit dieser Featuermenge. Allerdings ähnlich, wie die Helligkeitsverteilung, ist der RandomForest insgesamt balancierter im Hinblick auf die Gestentestmenge von Dymel. Die Bagging Ensemble-Methode verursachte Fehler und konnte nicht evaluiert werden.

Motion History ist mit Gleitkommazahlen implementiert. Allerdings sollte es keine Probleme bereiten sie mit 8-Bit Integer zu reimplementieren. Damit sollte die Programmgröße deutlich sinken. Obwohl die Konfiguration mit ExtraTrees am besten abschneidet, konnte die Konfiguration mit dem RandomForest nicht für den ATmega328P kompiliert werden aufgrund der Programmgröße. Es ist zu erwarten, dass dieser nach der Reimplementierung klein genug ist.

In Abbildung 5.2 ist ein Anstieg der Erkennungsgenauigkeit mit zunehmender Waldgröße für die Testmenge von Klisch und die Nullgestentestmenge von Dymel zu erkennen. Auffällig ist die Erkennungsgenauigkeit der Gestentestmenge von Dymel, die zunehmend abnimmt. Die Abbildung zeigt pro Waldgröße immer die beste Konfiguration im Hinblick auf die summierte

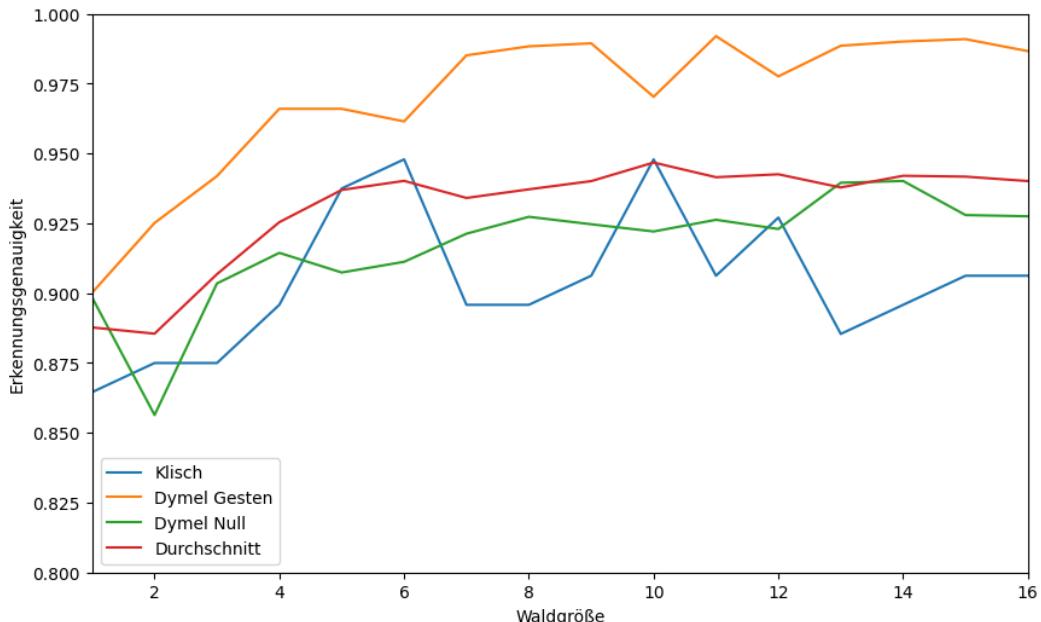
## 5.1 ERKENNUNGSGENAUIGKEIT

Erkennungsgenauigkeit der Testmenge von Klisch und der Nullgestentestmenge von Dymel an. Dementsprechend nimmt sie für die ExtraTrees Methode ab, nicht aber für RandomForest und Boosting.

### 5.1.3 Schwerpunktverteilung mit Gleitkommazahlen

Konfiguration	Beste	Unter 60 kB	Unter 28 kB	Unter 14 kB
Ensemble-Methode	Boosting	Boosting	RandomForest	Bagging
Maximalhöhe	20	19	10	7
Waldgröße	10	6	4	3
min_samples_leaf	8	8	2	8
Programmgröße in Bytes	83304	43678	20188	6656
Genauigkeit Testmenge von Klisch	94,8%	94,8%	89,6%	87,5%
Genauigkeit Gestentestmenge	97,0%	96,1%	95,6%	94,1%
Genauigkeit Nullgestentestmenge	92,2%	91,1%	88,8%	89,9%

■ **Tabelle 5.3:** Beste Konfigurationen der Schwerpunktverteilung mit Gleitkommazahlen.



■ **Abbildung 5.3:** Die beste summierte Erkennungsgenauigkeit pro Waldgröße der Schwerpunktverteilung mit Gleitkommazahlen.

## 5 EVALUATION

Die Featuremenge Schwerpunktverteilung mit Gleitkommazahlen folgt der Definition aus Sektion 4.2.2 und beinhaltet insgesamt 10 Einträge, wobei jeweils 2 Einträge die X und Y Koordinate des Schwerpunktes darstellen in insgesamt 5 Zeitfenstern.

Die beste Konfiguration wurde mit der Boosting Ensemble-Methode erzielt (siehe Tabelle 5.3). Mit einer Erkennungsgenauigkeit von 94,8% auf der Testmenge von Klisch ist dieser Ansatz nur 5,2% schlechter als das neuronale Netz von Giese [Gie20]. Es ist anzumerken, dass mit einer kleineren Trainingsmenge ohne Nullgesten eine Lösung gefunden wurde, die 97,9% erzielte und damit nur 2,1% schlechter ist. Außerdem werden 92,2% der Nullgestentestmenge von Dymel korrekt klassifiziert und 97% der Gestentestmenge von Dymel. Die anderen Ensemble-Methoden sind nur marginal schlechter.

Alle Konfigurationen sind ohne Optimierung aber zu groß um kompakt zu werden. Aus diesem Grund wurden die gleichen Konfigurationen mit Festkommazahlen und dem diskreten Wahlklassifizierer reevaluiert. TODO: Evaluieren, evtl. kleinere Waldgröße nutzen, e.g. 6 und mit Graphen begründen.

In Abbildung 5.3 kann man einen Anstieg der Erkennungsgenauigkeit mit zunehmender Waldgröße auf den Testmengen von Dymel erkennen. Bis zu einer Waldgröße von 6 gilt das auch für die Testmenge von Klisch, allerdings fängt sie ab dort an zu schwanken.

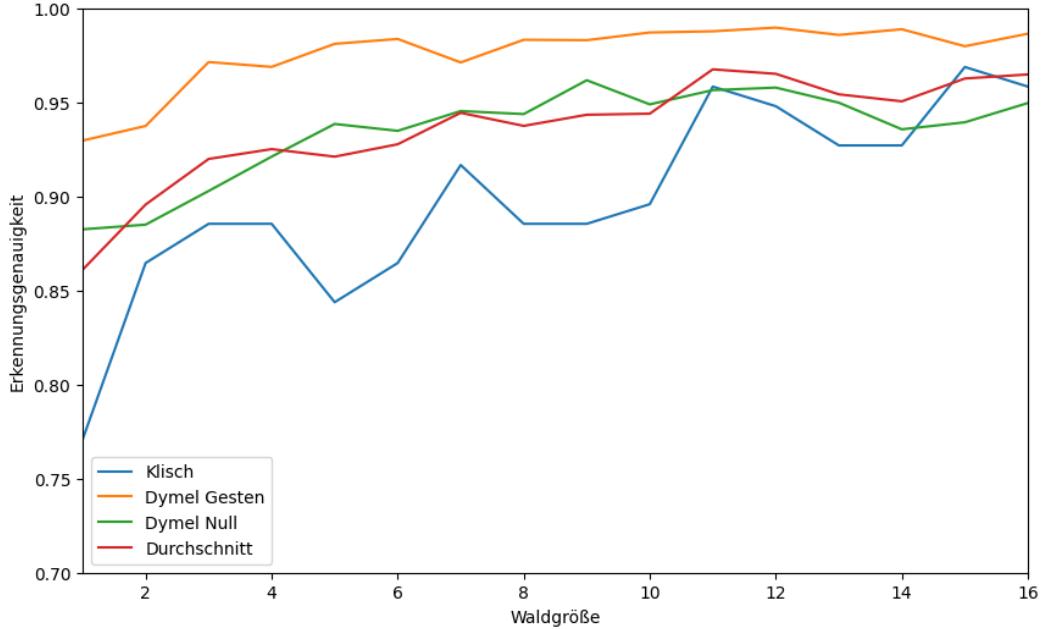
### 5.1.4 Schwerpunktverteilung mit Integer

Konfiguration	Beste	Unter 60 kB	Unter 28 kB	Unter 14 kB
Ensemble-Methode	ExtraTrees	RandomForest	RandomForest	RandomForest
Maximalhöhe	21	13	12	12
Waldgröße	11	15	8	3
min_samples_leaf	2	4	8	1
Programmgröße in Bytes	76200	51156	20476	11012
Genauigkeit Testmenge von Klisch	95,8%	96,9%	91,7%	86,5%
Genauigkeit Gestentestmenge	98,8%	97,9%	97,8%	95,5%
Genauigkeit Nullgestentestmenge	95,6%	93,9%	91,5%	88,9%

 **Tabelle 5.4:** Beste Konfigurationen der Schwerpunktverteilung mit Integer.

Die Featuremenge Schwerpunktverteilung mit Integer folgt der Definition aus Sektion 4.2.2 und beinhaltet insgesamt 10 Einträge, wobei jeweils 2 Einträge die X und Y Koordinate des

## 5.1 ERKENNUNGSGENAUIGKEIT



**Abbildung 5.4:** Die beste summierte Erkennungsgenauigkeit pro Waldgröße der Schwerpunktverteilung mit Integer.

Schwerpunktes darstellen in insgesamt 5 Zeitfenstern. Anders als die Schwerpunktverteilung mit Gleitkommazahlen wird diese nicht durch die Summe aller Pixel dividiert, wodurch X und Y zwischen -3072 und 3072 liegen anstatt -1 und 1.

Auch bei diesem Ansatz, weisen alle Ensemble-Methoden nur marginale Unterschiede auf (siehe Tabelle 5.4) im Hinblick auf die Erkennungsgenauigkeit. Insgesamt ist die Konfigurationen mit ExtraTrees am besten, da die summierte Erkennungsgenauigkeit dort maximal ist. Der RandomForest hat aber die höchste Erkennungsgenauigkeit auf der Testmenge von Klisch. Mit 96,9% ist das noch besser als die Schwerpunktverteilung mit Gleitkommazahlen und 0,1% besser als die beste Lösung die mit einer kleineren Trainingsmenge ohne Nullgesten gefunden wurde.

Leider ist die Programmgröße aller Konfigurationen, ohne Optimierungen, zu groß um kompiliert zu werden. Zurzeit nutzt der Ansatz 32-Bit Integer. Als Optimierung wurde der Datentyp auf 16-Bit Integer bzw. 8-Bit Integer reduziert und der hybride Wahlklassifizierer wurde genutzt. TODO: Reevaluierung.

Abbildung 5.4 zeigt, dass die Erkennungsgenauigkeit der Testmengen mit zunehmender Waldgröße ebenfalls zunimmt. Wobei besonders die Erkennungsgenauigkeit der Testmenge

## 5 EVALUATION

von Klisch stark schwankt.

### 5.1.5 Kombinierte Schwerpunktverteilung

Konfiguration	Beste	Unter 60 kB	Unter 28 kB
Schwerpunktverteilung Gleitkommazahl	Beste	Unter 28 kB	Unter 14 kB
Schwerpunktverteilung Integer	Beste	Unter 28 kB	Unter 14 kB
Programmgröße in Bytes	-	48040	20252
Genauigkeit Testmenge von Klisch	94,8%	90,6%	87,5%
Genauigkeit Gestentestmenge	99,0%	98,3%	96,9%
Genauigkeit Nullgestentestmenge	95,8%	92,3%	92,5%

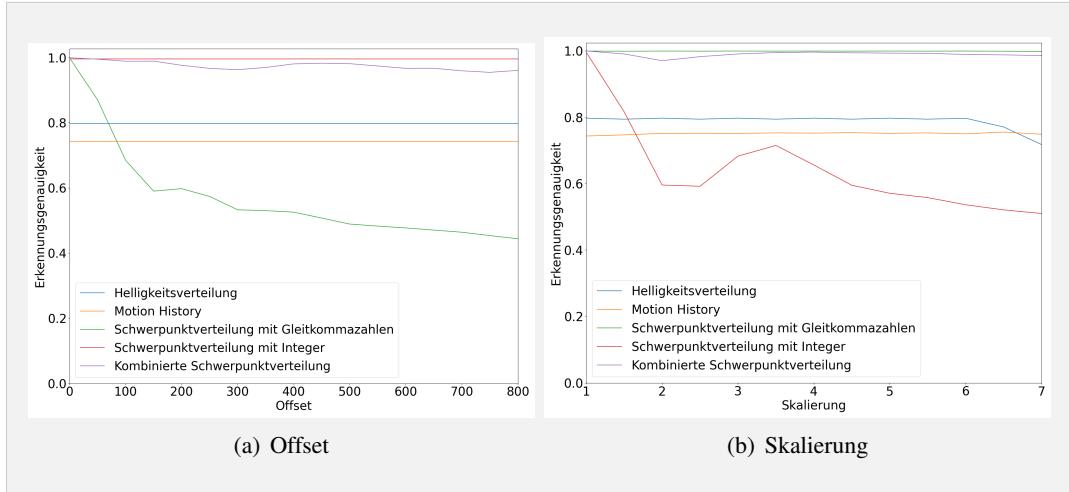
■ **Tabelle 5.5:** Beste Konfigurationen der kombinierten Schwerpunktverteilung.

Sektion 5.1.6 zeigt, dass die Schwerpunktverteilung mit Gleitkommazahlen invariant zu Skalierten Helligkeiten ist und die Schwerpunktverteilung mit Integer invariant gegenüber einen Offset in der Helligkeit. Aus diesem Grund erscheint ein kombinierter Klassifizierer sinnvoll. Es wird angenommen, dass die jeweiligen Ansätze in den Lichtverhältnissen, in denen ihre Stärken liegen, einen besonderen Fokus auf eine Klasse haben und andernfalls keinen besonderen Fokus.

Die jeweiligen Klassifizierer werden ebenfalls mit einem Wahlklassifizierer kombiniert, indem die Wahrscheinlichkeiten addiert werden zu gleichen Anteilen. Zu erwarten ist, dass sich die Erkennungswahrscheinlichkeit auf den Testmengen von Klisch und Dymel auf den Durchschnitt der beiden Ansätze angleicht. Außerdem wird dieser Klassifizierer nicht invariant gegenüber Skalierung und Offset sein. Trotzdem ist davon auszugehen, dass dieser Ansatz robuster gegenüber Lichtverhältnisse ist.

Die naive Implementierung, d. h. die besten Konfigurationen des Gleitkommazahl und Integer Ansatzes zu vereinen, ist zu groß für das Arduino Board. Sie erzielt jedoch 96,9% auf der Testmenge von Klisch, 99,4% auf der Gestentestmenge von Dymel und 95,2% auf der Nullgestentestmenge von Dymel. Eine klein genügende Lösung wird zwar eine geringere Erkennungsgenauigkeit haben, aber dafür durch die Kombination keine Regression zu den Ausgangsklassifizierern erfahren.

Der Nachteil dieses Ansatzes ist, dass sowohl die Featuremenge mit Gleitkommazahlen, als auch für Integer berechnet werden muss.



**Abbildung 5.5:** Ergebnisse der Helligkeitstestmenge 1 je Ansatz.

### 5.1.6 Robustheit gegenüber Lichtverhältnisse

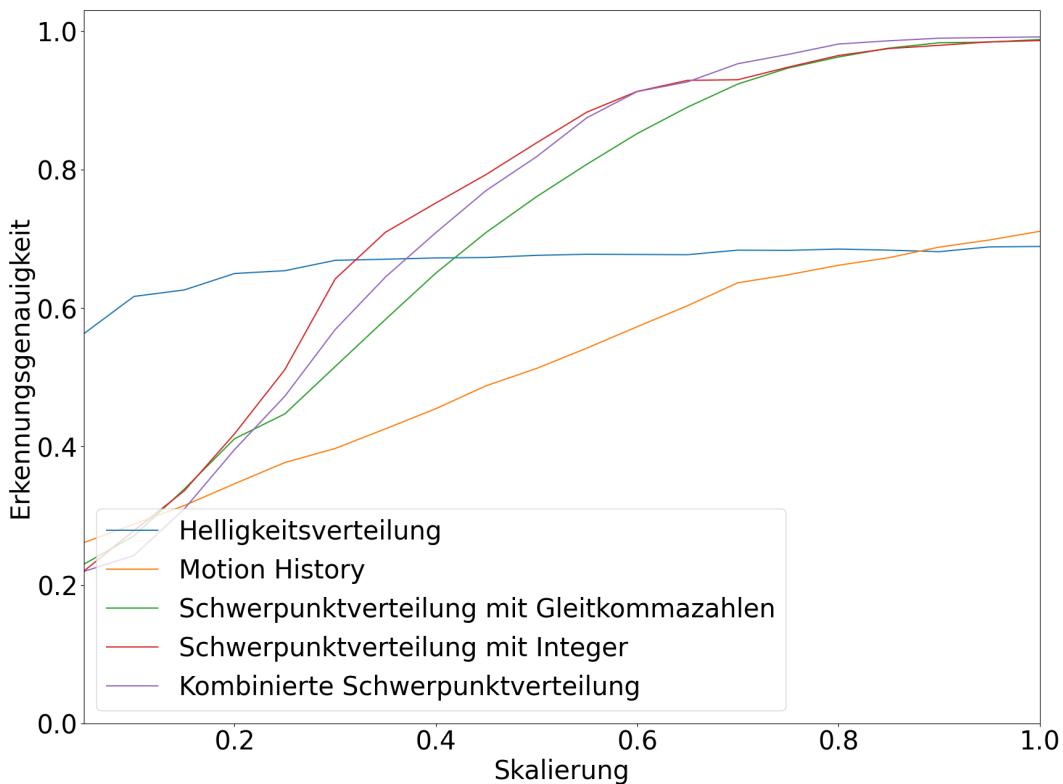
In Sektion 4.4 wurde die Helligkeitstestmenge 1 vorgestellt. Diese Testmenge modifiziert einen bestehenden Datensatz aus der Gestentestmenge indem Offset hinzugefügt oder die Helligkeiten skaliert werden. Bei dem Offset verändert sich der Kontrast nicht, aber die Gesamthelligkeit steigt. Bei der Skalierung steigt die Gesamthelligkeit und der Kontrast wird stärker. Mit dieser Testmenge sollen die Invarianzen der einzelnen Ansätze bewertet werden. Für die Helligkeitsverteilung und Motion History wurden die Konfigurationen gewählt, die insgesamt am besten waren, da die Helligkeitstestmenge 1 auf der Gestentestmenge basiert.

Abbildung 5.5(a) zeigt, dass die Helligkeitsverteilung, Motion History und die Schwerpunktverteilung mit Integer invariant gegenüber einem offset sind, wohingegen die Schwerpunktverteilung mit Gleitkommazahlen starke Defizite aufweist. Die kombinierte Schwerpunktverteilung ist nicht invariant und hat mit zunehmenden Offset eine schlechtere Erkennungsgenauigkeit.

Abbildung 5.5(b) zeigt, dass die Schwerpunktverteilung mit Gleitkommazahlen invariant gegenüber Skalierung ist. Die Helligkeitsverteilung und Motion History weisen weitgehend keine Defizite auf. Die Helligkeitsverteilung schwankt zwischen 6.5 und 7, allerdings findet ab dieser Skalierung der resultierende Wert bereits so groß, dass er teilweise auf den Maximalwert 1023 gesetzt wird. Die Schwerpunktverteilung mit Integer ist nicht invariant. Die kombinierte Schwerpunktverteilung weist keine Invarianz auf. Die Erkennungsgenauigkeit sinkt mit zunehmender Skalierung. Insgesamt verhalten sich alle Ansätze wie erwartet.

Die Helligkeitstestmenge 2 ist eine Mischung zwischen Skalierung und Offset. Die Gesamt-

## 5 EVALUATION



**Abbildung 5.6:** Ergebnisse der Helligkeitstestmenge 2 je Ansatz.

helligkeit bleibt erhalten und der Kontrast zwischen dunklen und hellen Pixeln verringert sich. Diese Situation ist vergleichbar mit einer zunehmenden Distanz zur Kamera, da dort der Kontrast durch Streulicht auch geringer wird.

Abbildung 5.6 zeigt, dass ausschließlich die Helligkeitsverteilung eine Invarianz aufweist. Sie zeigt zwar auch Defizite zwischen 0,05 und 0,3 auf, aber in diesem Intervall ist der Kontrast bereits sehr gering. Die Schwerpunktverteilungen verhalten sich der Erwartung gemäß, da keine invariant gegenüber Offset und Skalierung ist. Entgegen der Erwartung verhält sich die Motion History.

Abschließend ist die Helligkeitsverteilung am robustesten gegenüber den Lichtverhältnissen. Trotzdem erweisen sich die Schwerpunktverteilungen am besten, da die Erkennungsgenauigkeit deutlich höher ist in den meisten Fällen.

## 5.2 Ausführungszeit

Die Ausführungszeit der Feature-Extrahierung und Klassifizierung ist ausschlaggebend für die mögliche FPS. Diese ermöglicht die Wahrnehmung von schnellen Handgesten. Ist die FPS bereits ausreichend können leistungs schwächere Module verwendet werden, wodurch die Batterielaufzeit verlängert wird oder die Kosten verringert werden.

Das in dieser Arbeit verwendete Arduino Board ATmega328P verfügt nicht über ein Modul zur Verarbeitung von Gleitkommazahlen. Aus diesem Grund sind Operationen mit Gleitkommazahlen besonders teuer und zu vermeiden. Der ATmega328P verfügt über eine 8-Bit APU, 2 kB RAM, 32 kB Flash-Speicher und operiert unter 16 MHz [Cor15].

Es wird ausschließlich die *Worst-Case-Execution-Time* (WCET) betrachtet. Ausschlaggebend dafür ist der *Worst-Case-Execution-Path* (WCEP) im Kontrollflussgraph [FL10]. Der WCEP setzt sich zusammen aus dem lesen des aktuellen Bildes, der Extrahierung der Features und der Evaluation des Entscheidungswaldes.

Ausgewertet werden die Instruktionen des Programms das durch den AVR *GCC* mit der Optimierungsstufe O2 kompiliert wurde. Aus dem ATmega328P Handbuch [Cor15] können für jede Instruktion die Anzahl der Zyklen, die im schlimmsten Fall benötigt werden, entnommen werden. Die Gesamtausführungszeit berechnet sich aus der Anzahl der Zyklen multipliziert mit der Zeit pro Zyklus, d. h. bei 16 MHz,  $0,0625\mu s$ .

### 5.2.1 Feature-Extrahierung

Die Ausführungszeit um das Bild zu lesen wird als konstant angenommen. Diskutiert wird nur die Schwerpunktverteilung, da diese die beste Erfolgsgenauigkeit erzielte.

Jedes mal wenn ein Bild aufgenommen wird der Schwerpunkt des Bildes berechnet und gespeichert. In den WCEP fließt also nur ein mal die Schwerpunktberechnung eines Bildes ein. Insgesamt sind das jedoch bis zu  $116,5625\mu s$ . Diese setzen sich zusammen aus 201 Zyklen für die einzelnen Instruktionen ( $12,5625\mu s$ ),  $6 \cdot 4\mu s = 24\mu s$  für die Konvertierungsfunktion `_floatsif`,  $2 \cdot 36\mu s = 72\mu s$  für die Dividierungsfunktion `_divsf3` und  $2 \cdot 4\mu s = 8\mu s$  für die Vergleichsfunktion `_lesf2`.

```
Initialisierung.
for (char i = 0; i < 5; ++i) {
    features[i] = 0;
    for (char j = 0; j < zusammenfass_muster[i]; ++j)
        features[i] += * (schwerpunkt_buffer++);
```

## 5 EVALUATION

```
    features[i] /= ((float)zusammenfass_muster[i]);  
}
```

■ **Listing 5.1:** Algorithmus um die Schwerpunktverteilung zu berechnen.

Verbleibend ist nur für jedes der, in diesem Fall 5, Zeitfenster den zusammenfassenden Schwerpunkt durch den Durchschnitt zu ermitteln (siehe Listing 5.1). Die innere Schleife wird im schlimmsten Fall für die gesamte Schwerpunktbuffergröße durchlaufen, die zurzeit 125 ist. Die äußere Schleife wird exakt 5 mal durchlaufen.

Jeder Durchlauf der inneren Schleife bedarf im schlimmsten Fall bis zu  $13,6875 \mu s$ . Dies setzt sich zusammen aus 27 Zyklen für die einzelnen Instruktionen ( $1,6875 \mu s$ ), und  $12 \mu s$  für die Addierungsfunktion `_addsf3`. Bei 125 Durchläufen beläuft sich die Gesamtausführungszeit der inneren Schleife auf bis zu  $1710,9375 \mu s \approx 1,7 ms$ .

Jeder Durchlauf der äußeren Schleife bedarf im Schlimmsten Fall bis zu  $43,5625 \mu s$ . Dies setzt sich zusammen aus 57 Zyklen für die einzelnen Instruktionen ( $3,5625 \mu s$ ), 4  $\mu s$  für die Funktion `_floatsisf` und  $36 \mu s$  für die Funktion `_divsf3`. Bei 5 Durchläufen beläuft sich die Gesamtausführungszeit der äußeren Schleife auf bis zu  $217,8125 \mu s \approx 0,2 ms$ .

Für die Initialisierung werden weitere  $27,6875 \mu s$  benötigt. Die Schwerpunktverteilung wird in horizontaler und vertikaler Richtung berechnet. Zusammenfassend beläuft sich die WCET zur Berechnung der Schwerpunktverteilung mit Gleitkommazahlen auf bis zu  $4029,4375 \mu s \approx 4 ms$ .

### 5.2.2 Evaluation eines Entscheidungbaumes

Der WCEP eines Entscheidungbaumes ist der längste Pfad. Entlang des Pfades werden Vergleiche durchgeführt, bis ein Blatt des erreicht wird. Insgesamt sind die Anzahl der Vergleiche gleich der Höhe des Baumes.

Ein Vergleich in dem Entscheidungsbaum setzt sich aus bis zu 19 Zyklen für die Instruktionen `LDI`, `LDD`, `CALL`, `CP` und `BRLT` ( $1,1875 \mu s$ ) und  $4 \mu s$  für die Vergleichsfunktion `_lesf2`. Insgesamt  $5,1875 \mu s$  für einen Vergleich.

Zusätzlich kommt noch Funktionsoverhead von bis zu 11 Zyklen hinzu ( $0,6875 \mu s$ ). Bei einer maximalen Entscheidungsbauhöhe von 15 mit Gleitkommazahlenvergleichen beläuft sich die WCET eines einzigen Entscheidungbaumes auf bis zu  $78,5 \mu s$ .

### 5.2.3 Evaluation eines Entscheidungswaldes

Der WCEP eines Entscheidungswaldes setzt sich auf dem WCEP des Wahlklassifizierungsalgorithmus und dem WCEP jedes Entscheidungsbaumes zusammen, der im Wald enthalten ist.

Der in Listing 4.5 gezeigte Code implementiert den Wahlvorgang. Die Komplexität ist abhängig von der Anzahl der Features  $N$  und der Anzahl der Entscheidungsbäume  $K$ . In dieser Analyse wird für die Anzahl der Features  $N = 5$  angenommen.

Jede Stimme eines Entscheidungsbaumes bedarf 18 Zyklen ( $1,125 \mu s$ ), um die Evaluation aufzurufen und das Ergebnis auf die Gesamtsumme zu addieren. Die restlichen Instruktionen bedürfen 64 Zyklen ( $4 \mu s$ ). Bei 8 Bäumen ist die WCET bis zu  $13 \mu s$ .

Zusammenfassend ist die WCET für einen Entscheidungswald von 8 Bäumen, die jeweils eine Maximalhöhe von 15 haben, und einer Buffergröße von 125 mit der Gleitkommazahl basierten Schwerpunktverteilung als Feature  $4670,4375 \mu s \approx 4,7 ms$ . Dies ist  $1,7 ms$ , bzw.  $36,66\%$ , schneller als das beste neuronale Netz von Giese. Die Maximalhöhe des Entscheidungsbaumes und die Größe des Waldes haben dabei nur einen minimalen Einfluss auf die WCET, wodurch dieser Ansatz gut skaliert.

Potentiell kann die Ausführungszeit durch Festkommazahlarithmetik verbessert werden oder durch die Verwendung eines anderen Features. Momentan bedarf die Hardware knapp  $10 ms$  um ein Bild auszulesen. Damit sind FPS von bis zu 68 möglich.

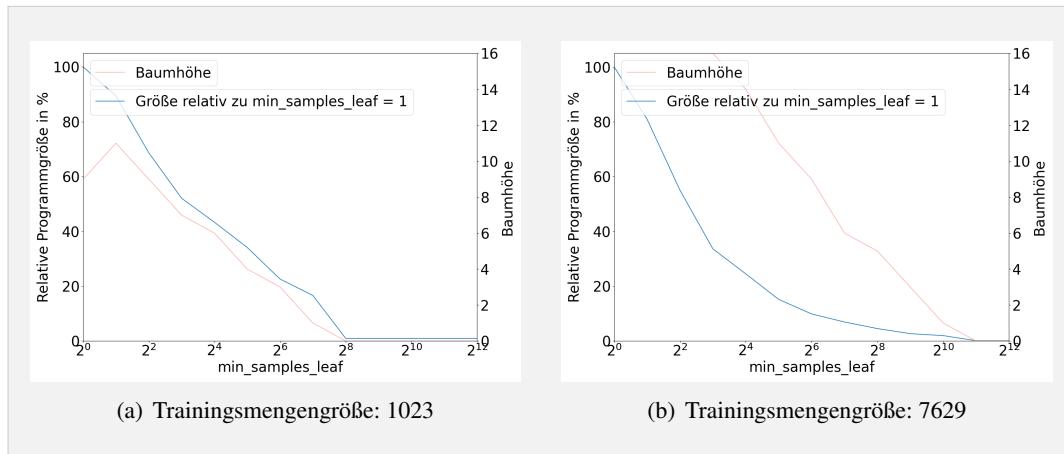
Die Gleitkommazahlfunktionen nehmen den Großteil der Ausführungszeit in Anspruch. Ein Feature, dass ausschließlich native 8-Bit Integer verwenden würde, würde die Gesamtausführungszeit deutlich reduzieren.

## 5.3 Programmgröße

Generell gilt, je größer und dichter der Entscheidungswald ist, desto höher ist die Erkennungsgenauigkeit. Aus diesem Grund sollte immer der vollständige Programmspeicher ausgenutzt werden. Allerdings nimmt der Zuwachs an Erkennungsgenauigkeit mit jeder weiteren Teilung ab bei der Konstruktion eines Entscheidungsbaumes.

Scikit-Learn bietet viele Parameter, um die Teilung zu steuern. Dies mag die potentielle Erkennungsgenauigkeit eines Baumes leicht verringern, kann dafür aber die Größe stark verringern. Dadurch können tiefere Bäume verwendet werden oder mehr Bäume in einem

## 5 EVALUATION



**Abbildung 5.7:** Auswirkung von `min_samples_leaf` Parameter auf resultierende Programmgröße und Baumhöhe.

Entscheidungswald, was potentiell einen größeren Zuwachs der Erkennungsgenauigkeit verspricht.

Die Größe und Dichte eines Entscheidungswaldes haben einen direkten Einfluss auf die generierten Instruktionen. Je größer und dichter, desto mehr Instruktionen. Aus diesem Grund soll einerseits der Zuwachs der Erkennungsgenauigkeit pro Vergleich maximiert werden und andererseits die Instruktionskosten pro Vergleich und Blattkosten minimiert werden.

### 5.3.1 Maximierung des Zuwachses der Erkennungsgenauigkeit

Standardmäßig wächst ein Entscheidungsbaum solange es mindestens eine Teilung gibt, die die Erkennungsgenauigkeit erhöht. Das führt zu sehr großen und sehr granularen Entscheidungsbäumen, die die Trainingsmenge auswendig lernen. Zudem sind die Bäume meistens unbalanciert es gibt viele Teilungen, die die Erkennungsgenauigkeit nur sehr geringfügig erhöhen.

Scikit-Learn bietet eine Reihe an Hyperparametern um dieses Verhalten zu steuern. Eine vollständige Analyse aller Parameter würde den Rahmen dieser Arbeit sprengen. Aus diesem Grund wird sich auf den Parameter `min_samples_leaf` beschränkt. Es wird immer eine feste Maximalhöhe von 16 betrachtet und mit der Optimierungsstufe `O`s kompiliert.

Damit eine Teilung stattfinden kann, muss der Knoten mindestens `min_samples_leaf` Einträge enthalten. Standardmäßig ist der Wert 1 [Ent20b]. Abbildung 5.7 zeigt die relative Programmgröße zu der Konfiguration mit `min_samples_leaf = 1` und die resultierende

maximale Baumhöhe die erreicht wurde. Zu sehen ist, dass bereits kleine Werte die Programmgröße signifikant verringern. Dieser Effekt ist größer bei großen Trainingsmengen und kleiner bei kleinen Trainingsmengen. Gleichzeitig verringert sich aber auch die maximale Baumhöhe die erreicht wird. Bei der großen Trainingsmenge hat sich die Programmgröße um 66,6% verringert bei gleichbleibender Baumhöhe mit `min_samples_leaf = 16`. `min_samples_leaf` eignet sich dementsprechend am besten für große Trainingsmengen.

### 5.3.2 Minimierung der Instruktionen eines Vergleichs

Das Ziel ist mit so wenig Instruktionen wie möglich einen Vergleich darzustellen. In Sektion 4.1.2 wurde bereits gezeigt, wie C-Code aus den Entscheidungsbäumen generiert wird.

```

01: ldi r18,lo8(33)
02: ldi r19,lo8(-92)
03: ldi r20,lo8(69)
04: ldi r21,lo8(60)
05: ldd r26,Y+5
06: ldd r27,Y+6
07: adiw r26,36
08: ld r22,X+
09: ld r23,X+
10: ld r24,X+
11: ld r25,X
12: sbiw r26,36+3
13: call __lesf2
14: cp __zero_reg__,r24
15: brge .+2

```

**Listing 5.2:** Vergleich von Gleitkommazahl-Feature mit konstanter Gleitkommazahl.

Listing 5.2 zeigt die Komplexität eines einzigen Vergleiches in Instruktionen. Zeile 1 bis 4 lädt die konstante Gleitkommazahl in 4 hintereinander liegende 8-Bit Register. Zeile 5 bis 7 lädt den Zeiger auf die Featuremenge und inkrementiert sie um 36 um auf das 9. Feature zuzugreifen. In Zeile 8 bis 11 wird das Feature in die Register geladen. Zeile 12 bis 15 führen die Vergleichsfunktion aus. Insgesamt sind es 15 Instruktionen, was ziemlich teuer für einen einzigen Vergleich ist.

Zu Vermeiden sind Zeile 5 bis 11 indem alle Features nur einmal in Register geladen werden. Dies ist allerdings nur möglich, wenn die gesamte Featuremenge in die Register reinpassen und zusätzlich noch Register verfügbar sind, um die Konstanten zu laden. Der Kompiler übernimmt diese Optimierung schon automatisch. Der ATmega328P verfügt allerdings nur über 32 Register. Bei einer Featuremenge von 10 werden dementsprechend regelmäßig Register verdrängt wodurch zusätzliche Instruktionen entstehen. Die Anzahl der Instruktionen können

## 5 EVALUATION

also reduziert werden, indem die Featuremenge reduziert wird bei der gleichen Anzahl von Vergleichen.

```
01: adiw r26,4
02: ld r24,X
03: sbiw r26,4
04: cpi r24,lo8(124)
05: brge .L3
```

■ Listing 5.3: Vergleich von 8-Bit-Feature mit konstanter 8-Bit Zahl.

Zusätzlich ist die Gleitkommazahl sehr teuer für einen 8-Bit Prozessor. Es werden immer 4 Register benötigt und zusätzliche Funktionen die die fehlende Hardwareunterstützung ausgleichen. Idealerweise sollte für die Featuremenge und die Vergleiche ein 8-Bit Datentyp gewählt werden. Damit werden einerseits weniger Register benötigt, wodurch wiederum die Featuremenge größer sein kann, und andererseits können native Vergleichsinstruktionen benutzt werden. Dies verringert die Anzahl der Instruktionen signifikant und eliminiert die teure Gleitkommazahlvergleichsfunktion. Mit einem kleineren Datentyp können dementsprechend Instruktionen vermieden werden. Listing 5.3 zeigt einen Vergleich mit einem 8-Bit Datentyp. Insgesamt werden 66,6% weniger Instruktionen benötigt.

### 5.3.3 Minimierung der Instruktionen einer Rückgabe

Der Wahlklassifizierer addiert die Wahrscheinlichkeiten für jede Klasse jedes Entscheidungsbaumes. Dadurch muss der Entscheidungsbaum eine derartige Wahrscheinlichkeitsverteilung als Ergebnis zurückgeben. In Sektion 4.1.2 wurde das durch die Zuweisung zu einem Parameter gelöst.

```
01: ldi r24,0
02: ldi r25,0
03: ldi r26,lo8(-128)
04: ldi r27,lo8(63)
05: st Z,r24
06: std Z+1,r25
07: std Z+2,r26
08: std Z+3,r27
09: std Z+4,__zero_reg__
10: std Z+5,__zero_reg__
11: std Z+6,__zero_reg__
12: std Z+7,__zero_reg__
13: std Z+8,__zero_reg__
14: std Z+9,__zero_reg__
15: std Z+10,__zero_reg__
16: std Z+11,__zero_reg__
17: std Z+12,__zero_reg__
18: std Z+13,__zero_reg__
```

```

19: std z+14,__zero_reg__
20: std z+15,__zero_reg__
21: ret

```

- **Listing 5.4:** Beispiel des Assemblycodes der Rückgabe der Wahrscheinlichkeitsverteilung eines Entscheidungsbaumes.

Listing 5.4 zeigt das Assembly das generiert wird für die Zuweisung von 4 Klassen 1.0, 0.0, 0.0 und 0.0. Der Kompiler optimiert das bereits, indem für jedes Ergebnis ein eigener Basic block erzeugt wird. Zusätzlich könnte die C-Code Generierung für die Fälle indem das Ergebnis 0 ist keine Zuweisung erzeugen. Wenn allerdings von dem Worst-Case Szenario ausgegangen wird hat diese Optimierung keine Wirkung, da jeder Klasse eine Wahrscheinlichkeit zugeordnet wird.

```

01: ldi r24,lo8(1)
02: ret

```

- **Listing 5.5:** Beispiel des Assemblycodes der Rückgabe eines diskreten Wahlklassifizierers.

Ein weiterer Ansatz ist den Wahlklassifizierer diskret zu modellieren. Anstatt die Wahrscheinlichkeiten zu addieren, werden die erkannten Klassen zu gleichen Teilen gezählt und addiert. Der Rückgabeblock verringert sich dann auf genau 2 Instruktionen (siehe Listing 5.5). Auch diese Rückgabe kann der Kompiler in einzelne Basic blocks extrahieren. Diese Optimierung ist hier sogar noch effektiver, da es nur genau  $N$  verschiedene Rückgabewerte gibt, für  $N$  mögliche Klassen. Tests haben ergeben, dass mindestens 66% Reduzierung der Programmgröße damit möglich ist.

Der Nachteil ist, dass die Ergebnisse instabil werden können, wenn viele Rückgaben nicht klar Eindeutig sind, sondern die Klasse nur eine knappe Mehrheit haben. Das ist insbesonders der Fall in Kombination mit einem hohen Wert für `min_samples_leaf`. Dennoch kann auf dieser Art und Weise die Programmgröße von einem Teil des Waldes oder der ganze Wald reduziert werden und die Stabilität des gefundenen Baumes mit den Testmengen revalidiert werden. Tests haben ergeben, dass die Erkennungsgenauigkeit der Testmengen nur geringfügig schwankt.

Denkbar wäre ein hybrider Ansatz, der bei einem eindeutigen Ergebnis die Klasse zurück gibt im `return` und ansonsten die Wahrscheinlichkeitsverteilung. Die „Eindeutigkeit“ kann über einen Schwellenwert definiert sein. Ein Schwellenwert von 0 würde an der Korrektheit und damit an der Stabilität nichts ändern.

## 5 EVALUATION

## Diskussion

Es wurde früh angefangen verschiedene Konfigurationen zu erproben und der Fokus lag schnell auf die Evaluierung verschiedener Ensemble-Methoden. Im Verlauf der Arbeit wurde aber klar, dass der Ansatz nicht optimal ist. Die Featuremengen, die generiert werden sind an sich keine Featuremengen, sondern eigene Features. Aus diesem Grund können einige Ensemble-Methoden, die die Featureauswahl varrieren, z. B. ExtraTrees, nicht ihr volles Potential ausschöpfen. Außerdem, wuchsen damit die Anforderungen an ein einzelnes Feature, was die Suche erheblich erschwerte. Vermutlich wäre ein besserer Ansatz Stacking gewesen mit Klassifizierern auf verschiedenen Featuremengen. Dieser Ansatz kombiniert verschiedene Klassifizierer, dessen Ergebnisse in jeweils nächsten mit einbezogen werden. Es wird vermutet, dass das Resultat deutlich simpler wäre bei trotzdem hoher Erkennungsgenauigkeit.

Die momentanen Trainingsdaten wurden weitestgehend unter den gleichen Lichtverhältnissen aufgenommen. Damit sind aber nicht alle möglichen Lichtverhältnisse gut repräsentiert. Wo möglich könnte ein Teil der synthetischen Daten zur Überprüfung der Lichtverhältnisse dazu genutzt werden, um Modelle zu generieren die robuster gegenüber verschiedenen Kontrasten und Helligkeiten sind.

## 6 DISKUSSION

## Schlussfolgerungen

Diese Arbeit hat gezeigt, dass sich Entscheidungsbaum basierte Klassifizierer sehr gut für die Handgestenerkennung eignen. Sie können sehr hohe Erkennungsgenauigkeiten erzielen, sowohl unter guten, als auch relativ schlechten Lichtverhältnissen. Nullgesten können von validen Handgesten unterschieden werden. Dabei sind sie signifikant schneller als KNNs und benötigen keinen RAM zur Evaluierung.

Die beste Konfiguration, der kombinierte Schwerpunktverteilungsklassifizierer, erzielte 94,8% auf der Testmenge von Klisch, 99% auf der Gestentestmenge und 95,8% auf der Nullgestentestmenge. Damit ist der Ansatz nur marginal schlechter als die beste Konfiguration der vorherigen Arbeiten, die 100% erzielte. Die kombinierte Schwerpunktverteilung erwies sich als äußerst Robust gegenüber skalierte Helligkeiten und Helligkeiten mit einem Offset. Selbst bei geringem Kontrast erzielt der Ansatz eine hohe Erkennungsgenauigkeit. Die WCET beläuft sich auf X(TODO). Der Großteil Y(TODO) wird davon für die Berechnung der Features benötigt. Damit ist man Z(TODO) schneller, als der beste vorherige Ansatz. Nach Anwendung aller Optimierungen und unter Annahme, dass Festkommazahlen keine Auswirkung auf die Entscheidungsgenauigkeit haben, ist der Klassifizierer zu groß für den ATmega328P.

Mit einer Beschränkung von 64kB kann der kombinierte Schwerpunktverteilungsklassifizierer 90,6% auf der Testmenge von Klisch erzielen, 98,3% auf der Gestentestmenge und 92,3% auf der Nullgestentestmenge bei einer Programmgröße von 48040 Byte. Mit der Beschränkung des ATmega328P von 32 kB, 87,5% auf der Testmenge von Klisch erzielen, 96,9% auf der Gestentestmenge und 92,5% auf der Nullgestentestmenge bei einer Programmgröße von 205252 Byte. Beide haben damit noch genug Raum, um andere Funktionen unter zu bringen.

Die Schwerpunktverteilung mit ausschließlich Integer hat eine WCET von X(TODO) und ist damit X%(TODO) schneller als die kombinierte Schwerpunktverteilung und X%(TODO) schneller als der beste vorherige Ansatz. Dieser Ansatz erzielt marginal schlechtere Ergebnisse

## 7 SCHLUSSFOLGERUNGEN

auf den Testmengen, als der Kombinierte Ansatz. Gegenüber skalierten Helligkeiten ist der Ansatz nicht robust.

Insgesamt benötigt die Implementierung der kombinierten Schwerpunktverteilung auf dem ATmega328P X(TODO) RAM, wobei nur X(TODO) zur Berechnung der Feature benötigt wird, X(TODO) für den Puffer und X(TODO) für die restliche Firmware. Im Puffer werden keine Bilder mehr gespeichert, sondern partiell ausgerechnete Feature, d. h. Im Fall der Schwerpunktverteilung, den Schwerpunkt jedes Bildes. Dafür wird weniger Speicher pro Bild verwendet, wodurch der Puffer größer sein kann als bei den KNNs. Die Programmgröße beträgt X(TODO). Davon wird X für den Klassifizierer benötigt.

Der Entscheidungsbaum bietet viel Optimierungspotential gegenüber der naiven Implementierung. Nur der Datentyp hat einen großen Einfluss sowohl auf die Programmgröße, als auch die Ausführungsgeschwindigkeit. Das ist motiviert aus der Spezifizierung des ATmega328P, der über einen 8-Bit Prozessor ohne Gleitkommazahlmodul verfügt. Gleitkommazahlen sind dementsprechend sehr teuer und 8-Bit Integer am günstigsten. Weitere Optimierungen sind Festkommazahlen und die Verwendung eines hybriden bzw. diskreten Wahlklassifizierers. Diese vergrößern jedoch den Suchraum für den besten Klassifizierer, da sie sowohl die Programmgröße verringern als auch einen Einfluss auf die Erkennungsgenauigkeit haben.

Insgesamt ist es sehr aufwändig den potentiell besten Klassifizierer zu finden, da es viele Parameter gibt die in Kombination untereinander unterschiedliche Klassifizierer produzieren. Zudem ist die Konstruktion nicht immer deterministisch, weswegen sie als Monte Carlo Methode betrachtet werden kann. Insgesamt wurden 22528 verschiedene Konfigurationen untersucht und 28 Variationen von Feature. Darunter wurden neben der Schwerpunktverteilung die Helligkeitsverteilung und Motion History betrachtet, die wesentlich schlechtere Erkennungsgenauigkeiten auf die Testmengen erzielen.

Im Laufe dieser Arbeit ist eine komplexe Infrastruktur entstanden, die die Evaluierung von Modellen zur Handgestenerkennung erleichtert. Die Infrastruktur stellt nützliche Code-Bibliotheken und verschiedene Werkzeuge bereit. Mit einem Werkzeug wurden in kürzester Zeit 14410 verschiedene Handgesten aufgenommen. Aus diesen Handgesten wurden 3 synthetische Testmengen erstellt. Die Nullgestentestmenge und die Helligkeitstestmengen, die Kontraste und Skalierung testen.

In folgenden Arbeiten sollte untersucht werden, ob Stacking oder hierarchische Klassifizierer nicht besser geeignet sind für Entscheidungsbaum basierte Klassifizierer auf kleinen eingebetteten Systemen. Der momentane Ansatz erzeugt sehr große Entscheidungsbäume. Das

hängt einerseits mit der Größe der Trainingsmenge zusammen und andererseits mit der Größe der Featuremenge. Dadurch können andere Feature verwendet werden, die lediglich weitere Feature für den nächsten Klassifizierer generiert.

Außerdem könnte untersucht werden, ob KNNs nicht wesentlich kleiner sein können, wenn die Feature dieser Arbeit verwendet werden, anstatt die Rohdaten der Geste. Damit müsste die Geste nicht mehr 20 Bilder skaliert werden, wie es in den vorherigen Arbeiten der Fall ist.

## **7 SCHLUSSFOLGERUNGEN**

## Inhalt des USB-Sticks

- Latex-Quellcode und PDF dieses Dokuments
- Latex-Quellcode und PDF des Antrittsvortrags
- Latex-Quellcode und PDF des Abschlussvortrags
- Quellcode der gesamten Infrastruktur
- Dokumentation der Infrastruktur
- Hilfsskripte zum trainieren, validieren und generieren von Grafiken
- Verschiedene Versionen der Arduino-Firmware
- Trainingsdaten
- Testdaten
- Ergebnisse der Modelle auf den Testdaten in Rohform

Weitere Informationen können dem README.md entnommen werden.

## A INHALT DES USB-STICKS

# 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
- [Cor15] CORPORATION, Atmel: *ATmega328P*. [http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P\\_Datasheet.pdf](http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf). Version: 2015
- [D<sup>+</sup>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
- [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: *sklearn.tree.DecisionTreeClassifier*. <https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html>. Version: 2020
- [FL10] FALK, Heiko ; LOKUCIEJEWSKI, Paul: A compiler framework for the reduction of worst-case execution times. In: *Real-Time Systems* 46 (2010), Oct, Nr. 2, 251–300. <https://dx.doi.org/10.1007/s11241-010-9101-x>. – DOI 10.1007/s11241-010-9101-x. – ISSN 1573–1383
- [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

## LITERATURVERZEICHNIS

- [Kub19] KUBIK, Philipp: Zuverlässige Handgestenerkennung mit 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<sup>+</sup>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
- [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<sup>+</sup>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<sup>+</sup>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
- [VT20] VENZKE, Marcus ; TURAU, Volker: Ansatz: Schwerpunkt der Pixel. (2020), 11, S. 1