

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 . . . . .	18
4.1.3 C-Code Generierung eines Entscheidungswaldes . . . . .	20
4.2 Features . . . . .	20
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 Ganzzahlen . . . . .	34
5.1.5 Kombinierte Schwerpunktverteilung . . . . .	36
5.1.6 Robustheit gegenüber Lichtverhältnisse . . . . .	37
5.2 Ausführungszeit . . . . .	40
5.2.1 Operationen mit Gleitkommazahlen . . . . .	41
5.2.2 Feature-Extrahierung . . . . .	41
5.2.3 Ausführung eines Entscheidungbaumes . . . . .	44
5.2.4 Ausführung eines Entscheidungswaldes . . . . .	44
5.2.5 Gesamtausführungszeit und Optimierung . . . . .	44
5.3 Programmgröße . . . . .	45
5.3.1 Maximierung des Zuwachses der Erkennungsgenauigkeit . . . . .	46

## **INHALTSVERZEICHNIS**

5.3.2	Minimierung der Instruktionen eines Vergleichs . . . . .	46
5.3.3	Minimierung der Instruktionen einer Rückgabe . . . . .	48
<b>6</b>	<b>Diskussion</b>	<b>51</b>
<b>7</b>	<b>Schlussfolgerungen</b>	<b>53</b>
<b>A</b>	<b>Inhalt des USB-Sticks</b>	<b>57</b>
	<b>Literaturverzeichnis</b>	<b>59</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 [ABC<sup>+</sup>16]. Ein namhaftes Beispiel für das Potential von maschinellen Lernen ist *AlphaGo Zero*, 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 [SSS<sup>+</sup>17].

Ein häufiges Anwendungsgebiet in eingebetteten Systemen ist die optische Gestenerkennung, die zur kontaktlosen Interaktion, u. a. mit technischen Geräten, genutzt wird [PSH97]. Die eingesetzten Mikrocontroller sind jedoch häufig nicht ausreichend leistungsstark, um ein trainiertes Modell in passabler Zeit auszuführen. Gründe dafür sind Kosten oder Anforderungen an die Batterielanglebigkeit. 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 das Rechen-Cluster gesendet [VKK<sup>+</sup>20]. Nachteile dieses Ansatzes sind einerseits die Abhängigkeit zu dieser Infrastruktur und andererseits vergrößert sich die Latenz. 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 basierter 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, Ausführungszeit und Resourcenverbrauch des Modells auf dem Mikrocontroller. Dafür ist ein

## 1 EINLEITUNG

Konzept zur Übersetzung des Modells auf den Mikrocontroller nötig.

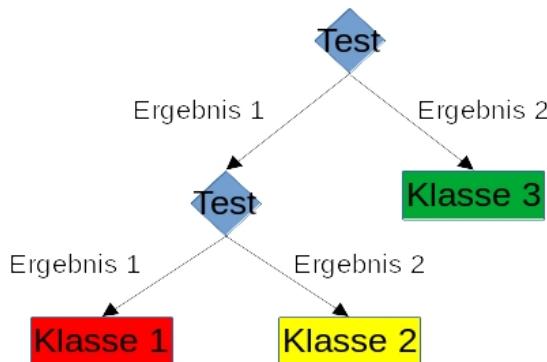
Insgesamt wurden 28 Varianten an Features analysiert. Die Features mit der besten Tauglichkeit wurden mit verschiedenen Ensemble-Methoden und Parametern für Entscheidungsbäume kombiniert. Daraus sind 22528 verschiedene Konfigurationen entstanden, die auf ihre Erkennungsgenauigkeit hin verglichen wurden. Die beste Konfiguration wurde auf ihre Worst-Case-Execution-Time (WCET) und auf den Resourcenverbrauch hin analysiert und mit den vorherigen Arbeiten, aus der gleichen Fallstudie [VKK<sup>+</sup>20], verglichen. Dabei wurden zusätzlich 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 mit solchen 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 auschließenden Ergebnissen hat. Das Ergebnis bestimmt mit welchem Kindknoten fortgefahrene wird [Qui90]. Abbildung 2.1 zeigt einen Entscheidungsbaum, indem jeder Test zwei mögliche Ergebnisse hat. Dieser 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 enthalten, die mit der gleichen Klasse beschriftet sind [Ste09].

Die Fähigkeit zu Generalisieren ist stark davon abhängig, wie repräsentativ die Trainingsmenge ist und die Art und Weise, wie verschiedene Klassen in der Gesamtmenge unterschieden wird [Ste09]. 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. Die Konstruktion findet auf Basis von einer Trainingsmenge statt. Die Effektivität des Lerners hängt dabei stark von der Qualität dieser Menge ab. Im schlimmsten Fall können irrelevante Feature die Effektivität stark beeinträchtigen. Aus diesem Grund ist es wichtig die richtige Featuremenge auszuwählen [PGP98].

Es gibt verschiedene Algorithmen um Entscheidungsbäume zu erzeugen. Am häufigsten referenziert sind ID3 [Qui86], C4.5 [Qui14] und CART [BFSO84]. Das Grundprinzip ist dabei immer das gleiche. Partitioniere die Trainingsmenge, sodass möglichst nur Einträge der Trainingsmenge mit der gleichen Beschriftung in einer Partitionierung sind. Die Algorithmen unterscheiden sich dabei in ihrer Strategie. Der naive Ansatz ist alle möglichen Entscheidungsbäume zu generieren und davon den besten auszuwählen. Das ist aber bei großen Feature- und Trainingsmengen sehr rechenaufwändig [Qui86].

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, die 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 der minimale Anteil einer Klasse um ein Blatt zu bilden [Ent20b].

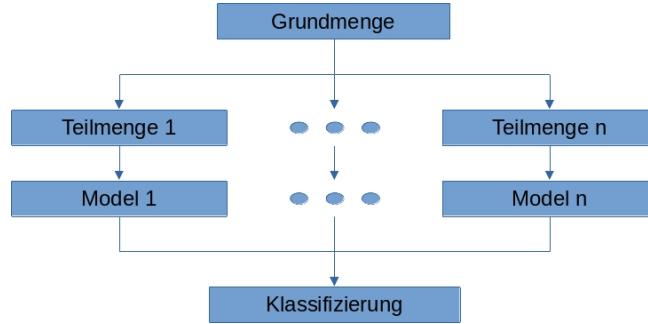
## 2.2 Ensemble-Methoden

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 Wahlklassifizierer  $H(x) = w_1h_1(x) + \dots + w_Kh_K(x)$  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

## 2 ENTSCHEIDUNGSBÄUME



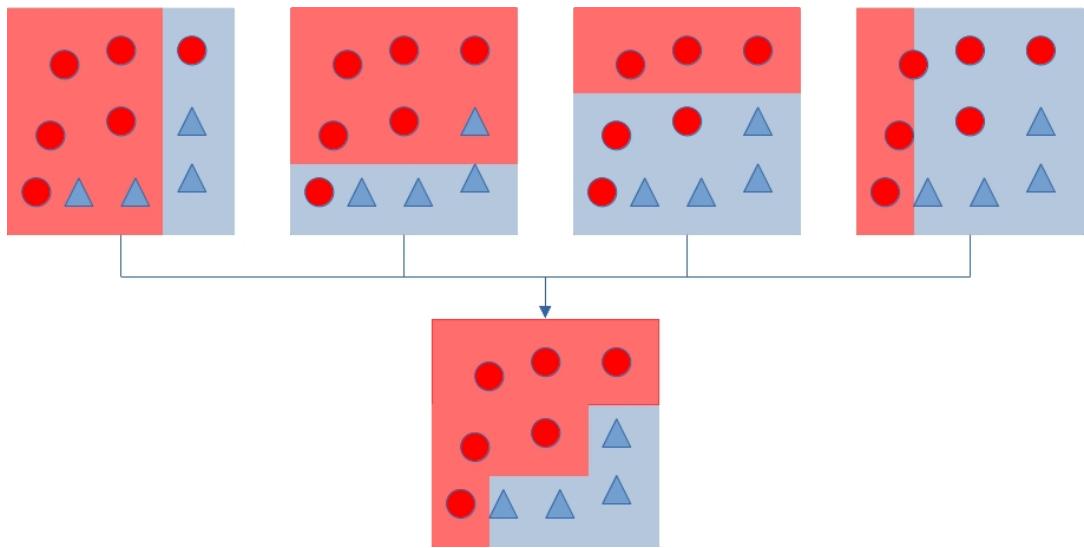
**Abbildung 2.2:** Klassifizierungsprozess mit der Bagging-Methode.

jedem dieser Mengen einen Klassifizierer zu trainieren und schließlich alle Klassifizierer, z. B. durch Wahlen, 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 (ExtraTrees) gehen im Vergleich zu Random Forest einen Schritt weiter. Anstatt den besten Teilungspunkt für die ausgewählten Features zu suchen, werden zufällig ein Teilungspunkte ausgewählt, aus denen der beste genutzt wird. Das soll die Varianz reduzieren. Außerdem wird nicht, wie bei der Bagging-Methode, auf Teilmengen trainiert sondern auf dem gesamten Set. Dies soll den Bias reduzieren [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. Die optischen Ansätze nutzen einen oder mehrere Kameras um eine Folge von Bildern aufzunehmen. Dieser Ansatz ist allerdings empfindlich gegenüber Lichtverhältnissen 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 100% korrekt klassifiziert werden. Auch Nullgesten, d. h. invalide Gesten, konnten zuverlässig erkannt werden. Das verwendete neuronale Netz benötigte  $6,8\text{ ms}$  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 mit Hilfe eines tragbaren SEMG Recorders die elektrischen Signale der Muskelaktivitäten erfasst. Als Eingabe für den Entscheidungsbaum wählten sie 9 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].

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

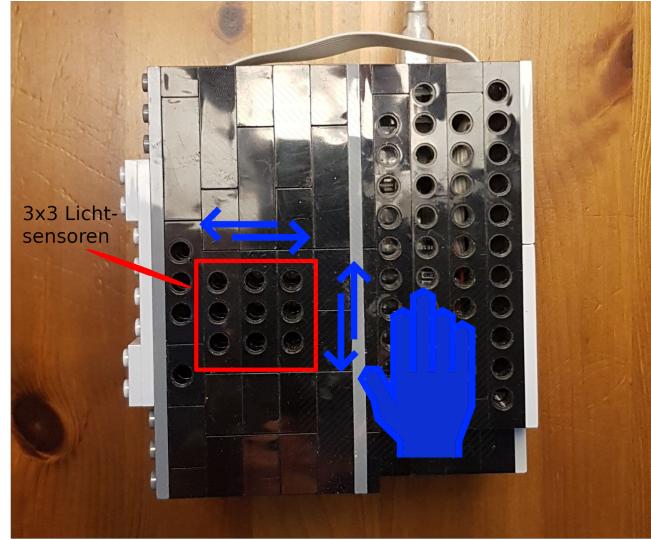
$$H_\tau(x, y, t) = \begin{cases} \tau & \text{if } \psi(x, y, t) = 1 \\ \max(0, H_\tau(x, y, t - 1) - \delta) & \text{otherwise} \end{cases} \quad (3.1)$$

Das MHI kann sequentiell berechnet werden. Initial sind alle Werte 0. Wenn  $\psi(x, y, t)$  eine dominante Bewegung in einem Pixel  $(x, y)$  zu einem Zeitpunkt  $t$  signalisiert, dann wird der Pixel zum Maximalwert  $\tau$  gesetzt. Mit jedem Bild in dem keine dominante 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. Je nach dem, 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 Mikrocontrollern 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 nach



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

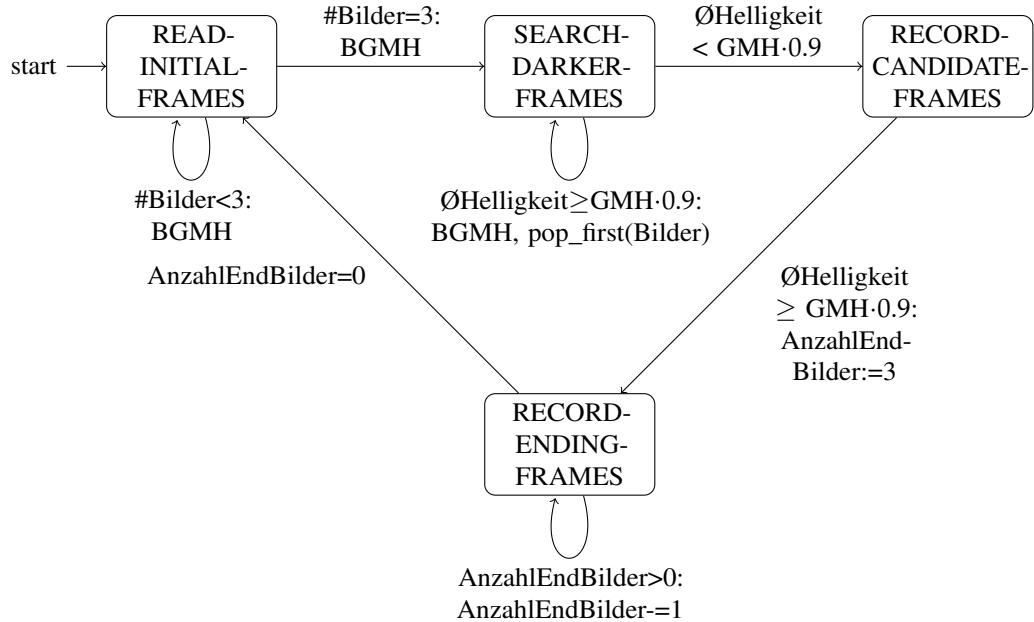
Rechts, Rechts nach Links, Oben nach Unten, Unten nach Oben (siehe Abbildung 3.1) und Nullgeste, d. h. eine invalide Geste. Die bisherigen Arbeiten haben sich mit künstlichen neuronalen Netzen beschäftigt. Dessen Prozessablauf zur Gestenerkennung lässt sich im Grunde auf 3 Schritte zusammenfassen.

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, d. h. 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 sind. 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, der von Dr. Marcus Venzke entwickelt wurde. 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 90% des gleitenden Mittelwerts unterschreitet wird die Aufnahme begonnen. Der Automat geht in den Zustand `RECORD-CANDIDATE-FRAMES` über. Dort verbleibt der Automat solange bis die Durchschnittshelligkeit 90% des gleitenden Mittelwerts überschreitet, woraufhin der Automat in den Zustand `RECORD-END-ENDING-FRAMES` über geht, indem die letzten 3 Bilder an den Puffer angehängt werden. Sobald 3 Bilder angehängt wurden, 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].

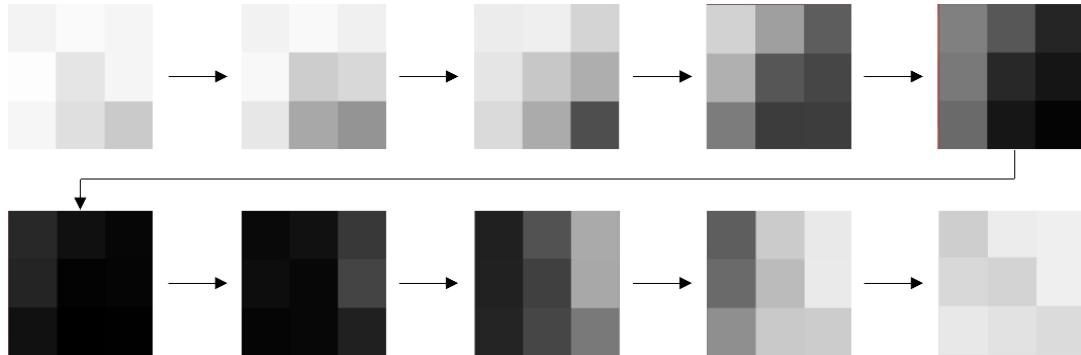
### 3.2.3 Trainings- und Testdaten

```
...
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 dem Finger in verschiedenen Geschwindigkeiten ausgeführt.

### 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 werden 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 Fingern unter verschiedenen Helligkeiten aus. Damit hat Engelhardt seine Modelle trainiert und validiert. Er argumentiert, dass rekurrente neuronale Netze (RNN), Feedforward neuronale Netze (FFNN) und Long-Short-Term Memory neuronale Netze (LSTMNN) für temporale Probleme am besten geeignet seien. Convolutional neuronale Netze (CNN) verwarf Engelhardt aufgrund der geringen Auflösung der Gesten und da die Faltung extrem rechenaufwendig sei. Des Weiteren 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 und auf Werte zwischen 0 und 1 normalisiert wurden. Als

### 3.2 OPTISCHE HANDGESTENERKENNUNG

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 in verschiedenen Distanzen zur Kamera, unter guten, und schlechten Lichtverhältnissen ausgeführt werden. Neben der Facettenkamera, die Engelhardt ebenfalls genutzt hatte, untersuchte Kubik ebenfalls eine Lochkamera. Er stellte fest, dass diese aber wesentlich auszuleuchten schwerer ist, was sich auf die Erkennungsgenauigkeit auswirkte. 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 aber 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 Anbetracht des RAMs von dem Microcontroller. Um die Erkennungsgenauigkeit zu erhöhen verwendete Kubik 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 unter guten und verhältnismäßig schlechten Lichtverhältnissen eine Erkennungsgenauigkeit von 71%, welches

### 3 GESTENERKENNUNG

eine Verbesserung zu dem Ergebnis von Engelhardt ist. Klisch stellte fest, dass 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 auf 100%. Die beste Ausführungszeit wurde mit dem CSC-MA-Bit Format erzielt, dass unnötige Multiplikationen verhindert und die kleinste Programmgröße wurde mit dem CSC-Centroid Format erzielt. SeeDot hat im Vergleich zum Ausgangsmodell sowohl Ausführungszeit, als auch Programmgröße verringert, hat aber die Erkennungsgenauigkeit signifikant gesenkt. 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

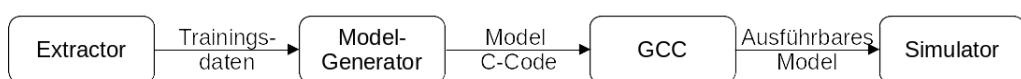
Entscheidungsbäume sind sehr schnell im Vergleich zu neuronalen Netzen. Allerdings eignen sich neuronale Netze oft besser für komplexe ML Probleme, da sie leicht zu konstruieren sind und bereits auf Rohdaten gut generalisieren. Das untersuchte Problem weiß aber nur eine geringe Komplexität. Aus diesem 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 dem 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. Anschließend wird aus dem Model ausführbarer C-Code generiert und kompiliert. Zuletzt



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

## 4 ENTSCHEIDUNGSBAUM BASIERTE HANDGESTENERKENNUNG

wird die Erkennungsgenauigkeit des ausführbaren Models auf der Testmenge von Klisch, der Gestentestmenge und Nullgestentestmenge 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 und 16 betrachtet und Maximalhöhen von den einzelnen Entscheidungsbäumen zwischen 1 und 22. Außerdem werden die Blattgrößen, d. h. die minimale Anzahl von Trainingsdateneinträgen pro Blatt, *1, 2, 4 und 8* untersucht.

Der letzte Parameter verringert potentiell die Erkennungsgenauigkeit von den einzelnen Entscheidungsbäumen, verringert aber auch die Größe des Baumes signifikant. Dadurch können 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 den gleich guten Teilungen 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 ein `random_state` zugewiesen werden, 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.

### 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 Ergebnis 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 linkes Kind...
} else {
    Traversiere rechtes Kind...
}

```

**■ 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.

Der C-Code im Blatt ist abhängig von dem Wahlklassifizierer. Es können entweder die Klasse ausgewählt werden, die von den meisten Bäumen klassifiziert wurde, oder die Erkennungswahrscheinlichkeiten jedes Baumes im Ensemble 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 ENTSCHEIDUNGSBAUM BASIERTE HANDGESTENERKENNUNG

### 4.1.3 C-Code Generierung eines Entscheidungswaldes

Ein Entscheidungswald besteht aus einem Ensemble von Entscheidungsbäumen. Bei der Evaluierung eines Entscheidungswaldes wird 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 Ensemble bestehend aus  $K$  Entscheidungsbäumen, die jeweils  $N$  mögliche Klassifizierungsergebnisse zurückgeben. Diese werden mit einem Wahlklassifizierer zusammengefasst. 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 eindeutig trennbar sind. Ist eine Trennung nicht möglich, so ist auch keine gute Erkennungsgenauigkeit 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 der 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 Feature für einzelne Spalten oder Zeilen.

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

### 4.2.2 Featureauswahl

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

Das *Mean absolute value* Feature ermöglich 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 bei Handgesten mit verschwieder Geschwindigkeit.

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

Feature 2, 5 und 7 bis 9 wurden nicht weiter untersucht, weil sie zu komplexe Berechnungen bedürfen für ein kleines eingebette System ohne Modul zur Gleitkommazahlberechnung.

### Motion History

Die Motion History zeigt eine Bewegungshistorie, indem eine 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 der Geschwindigkeit. Als Lösung wurde  $\delta$  abhngig von der Gestenlnge 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.

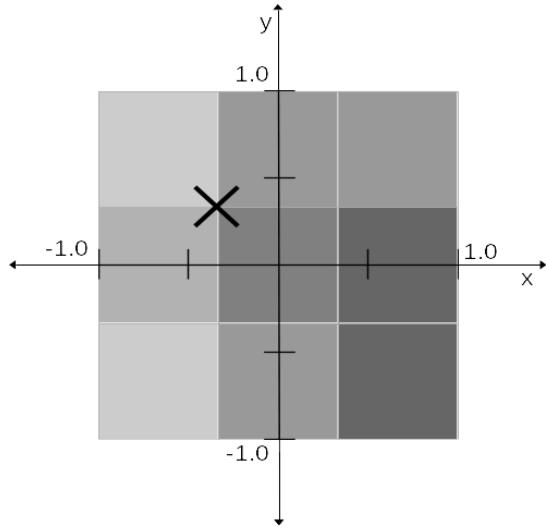
Außerdem können die Anzahl der Zeitfenster variiert werden und Pixel zu Gruppen zusammengefasst werden, d. h. Spalten und Zeilen.

### Schwerpunktverteilung

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

Der Schwerpunkt  $(X_Q, Y_Q)$  in einem Bild  $Q$  (Formel 4.2) ist über die Helligkeit in den einzelnen Pixel definiert. Der Pixel  $q_{11}$  bildet den Nullpunkt des Koordinatensystems. Dann ist

#### 4 ENTScheidungsbaum basierte Handgestenerkennung



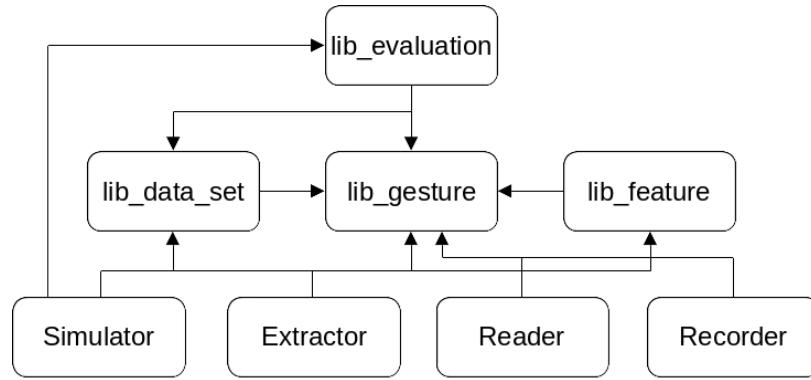
**Abbildung 4.2:** Illustration des Schwerpunktes im 3x3 Fotowiderstand-Array.

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 (Kapitel 5.3) und die Feature-Extrahierung ist schneller (Kapitel 5.2). Die Schwerpunktverteilung mit Integer ist durch das weglassen von  $P$  invariant gegenüber einem Offset  $O$ , da  $\sum_{i=0}^2 (q_{i,2} + O) - \sum_{i=0}^2 (q_{i,0} + O) = \sum_{i=0}^2 q_{i,2} - \sum_{i=0}^2 q_{i,0} = X_Q$  ist und analog für  $Y_Q$ . Der Ansatz mit den Ganzzahlen produziert Schwerpunkte in  $[-3072, 3072]^2$  und der Ansatz mit Gleitkommazahlen produziert Schwerpunkte in  $[-1, 1]^2$ .



■ Abbildung 4.3: Abhängigkeiten der einzelnen Module.

### 4.3 Infrastruktur

In dieser Arbeit mussten viele Features 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 Features hinzuzufügen, ohne Kompatibilitätsprobleme zu verursachen. Alle Funktionalitäten sind in Code-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 Handgesten 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 Handgesten.
- Rotation der äußeren Pixel um Diagonale Handgesten 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 Handgeste (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 entweder durch einen Offset oder indem er skaliert wird.

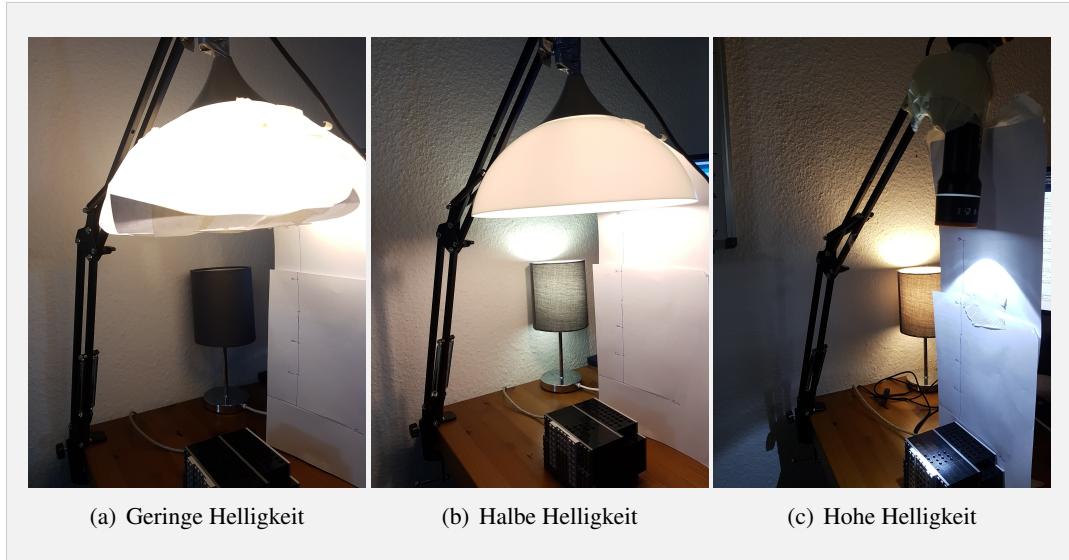
`lib_evaluation` bietet ein Hilfsobjekt an, dass Datenmengen nach Erkennungsgenauigkeit auswertet und Berichte daraus generiert.

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 Parsingmethode von Kubik um Gestenkandidaten zu erkennen. Diese Information wird genutzt, um in eine vordefinierte Datei die Handgesten reinzuschreiben. Um effizient Gesten aufzunehmen wurde der Ansatz von Kubik aufgegriffen mit einem Gestentyp zu starten



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

und folgend immer zwischen dem Inverstyp hin und her zu wechseln [VKK<sup>+</sup>20].

Das Programm wurde um zwei weitere Optionen erweitert. Mit der ersten Option wird immer nur eine bestimmte Handgeste hintereinander aufgenommen. Mit der zweiten Option wird jedes mal wenn eine Handgeste erkannt wurde, der Gestentyp zur manuellen Eingabe erfragt. 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. Die Datenmenge 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 ist nur ein geringer Anteil an Nullgesten enthalten.

Jede Handgeste wurde unter jeder Konfiguration ca. 100 mal aufgenommen bei 90 Bildern pro Sekunde. Insgesamt wurden in 3 Lichtverhältnissen 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 Handgesten wurden in den Abständen 5 cm,

## 4 ENTSCHEIDUNGSBAUM BASIERTE HANDGESTENERKENNUNG

10 cm, 20 cm und 25 cm aufgenommen.

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 mit zunehmender Distanz der Kontrast 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 Datenmenge ohne Nullgesten „Gestenmenge“ genannt. Die ersten 25% werden zum Trainieren verwendet, die hinteren 75% zum Testen. Die Testmenge wird im weiteren „Gestentestmenge“ genannt. Da die Gestenmenge sehr groß ist, wird zum Trainieren relativ zu den bestehenden Trainingsdaten von Feng und Kubik nur ein kleiner Teil der Gestenmenge verwendet, damit kein Bias erzeugt wird.

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 Datenmenge mit ausschließlich Nullgesten „Nullgestenmenge“ genannt. Die ersten 12,5% werden zum Trainieren verwendet, die hinteren 87,5% zum Testen. Die Testmenge wird im weiteren „Nullgestentestmenge“ genannt.

Um zu testen wie gut das Modell sich gegenüber verschiedenen Lichtverhältnissen generalisiert hat, ist es nötig mehr als nur 3 Helligkeitsstufen zu testen. Aus diesem Grund wurde aus dem Anteil der Gestenmenge 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 zusammengeführt. 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 dem Anteil der Gestenmenge 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 lauffähig sein auf einem kleinen eingebetteten System, d. h. nicht mehr als den verfügbaren RAM und Programmspeicher nutzen und in einer passablen Zeit terminieren.

Dieses Kapitel untersucht zuerst die Erkennungsgenauigkeit der besten Konfigurationen, die gefunden wurden, je Featuremenge. 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 senken.

## 5.1 Erkennungsgenauigkeit

Es werden 3 Features näher betrachtet. Motion History, Helligkeitsverteilung und Schwerpunktverteilung. Daraus wurden 4 Featuremengen generiert, die zum Trainieren genutzt werden. Insgesamt wurden 22528 verschiedene Konfigurationen trainiert und getestet, die in Kapitel 4.1.1 beschrieben wurden. Jede Konfiguration nutzt zum Trainieren eine Kombination aus der Trainingsmenge von Feng und Kubik, sowie die Gestentestmenge und die Nullgestenmenge, die in Kapitel 4.4 beschrieben wurden. Die Trainingsmenge beinhaltet insgesamt 7629 Handgesten. Davon wird 50% zum Trainieren und 50% zum Validieren und optimieren auf Basis der Monte Carlo Methode benutzt.

Unter jeder Featuremenge werden jeweils 3 Kategorien analysiert. Die erste Kategorie zeigt die beste Konfiguration, die ohne Restriktion des Programmspeichers, gefunden wurde. Die zweite Kategorie hat eine Restriktion von 48 kB und die dritte Kategorie hat eine Restriktion

## 5 EVALUATION

von 32 kB. Diese beziehen sich auf den Programmspeicher des ATmega4809 und ATmega328P, die im Rahmen der Fallstudie verwendet werden [VKK<sup>+</sup>20]. Dabei werden immer 4 kB abgezogen, da diese für andere Systemrelevante Funktionen reserviert sind. Die beste Konfiguration einer Kategorie maximiert die Summe der Klassifizierungsgenauigkeiten der Testmenge von Klisch, der Gestentestmenge und der Nullgestentestmenge. Dabei wird stets die optimierte Programmgröße der Konfiguration betrachtet, nachdem alle Optimierungen aus Kapitel 5.3 angewendet wurden.

In der Analyse werden die verschiedenen Featuremengen im Hinblick auf die Klassifizierungsgenauigkeit der Testmenge von Klisch, der Gestentestmenge, der Nullgestentestmenge und den synthetischen Helligkeitstestmengen untereinander verglichen und mit den Ergebnissen von Giese verglichen. Es wird ausschließlich mit Giese verglichen, da seine Ergebnisse die beste Erkennungsgenauigkeit, geringste Ausführungszeit und geringsten Resourcenverbrauch erzielte. Außerdem wird die Auswirkung von verschiedenen Waldgrößen auf die Erkennungsgenauigkeit untersucht. Anzumerken ist, dass lediglich die Testmenge von Klisch vergleichbar mit den Ergebnissen von Giese ist, da die Gestentestmenge, Nullgestentestmenge und Helligkeitstestmengen erst im Laufe dieser Arbeit entstanden sind.

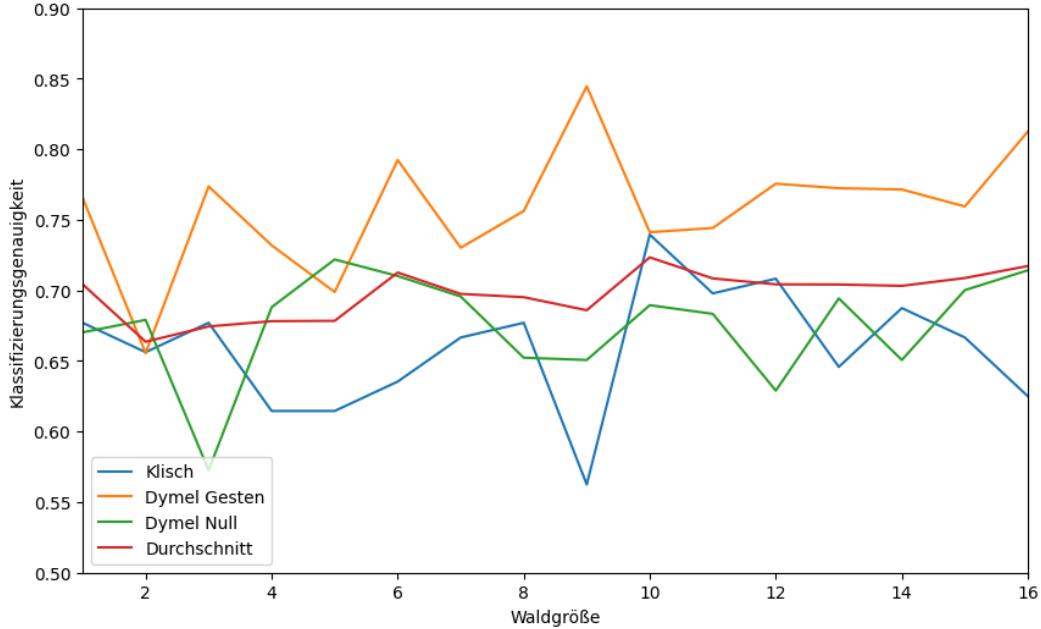
### 5.1.1 Helligkeitsverteilung

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:** Die beste Konfigurationen 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.

Aus der Tabelle 5.1 sind die besten Konfigurationen jeder Kategorie zu entnehmen. Die



■ Abbildung 5.1: Die besten Konfigurationen pro Waldgröße mit der Helligkeitsverteilung.

beste Konfiguration wurde mit der Ensemble-Methode *ExtraTrees* gefunden. Sie erzielt eine Erkennungsgenauigkeit von 74% auf der Testmenge von Klisch und ist damit 25,2% schlechter als das neuronale Netzwerk von Giese [Gie20]. Außerdem wird 74% der Gestentestmenge und 69% der Nullgestentestmenge korrekt klassifiziert.

Wird die Kategorie *Beste* und die Kategorie *Unter 28 kB* verglichen, nimmt die Gesamtklassifizierungsgenauigkeit nur um 1,94% ab. Dabei reduziert sich die Programmgröße um 87,8%. Ein ähnliches Verhalten ist auch in Abbildung 5.1 zu erkennen. Dort ist nur eine geringer Zuwachs der Gesamtklassifizierungsgenauigkeit mit der zunehmenden Waldgröße zu beobachten.

### 5.1.2 Motion History

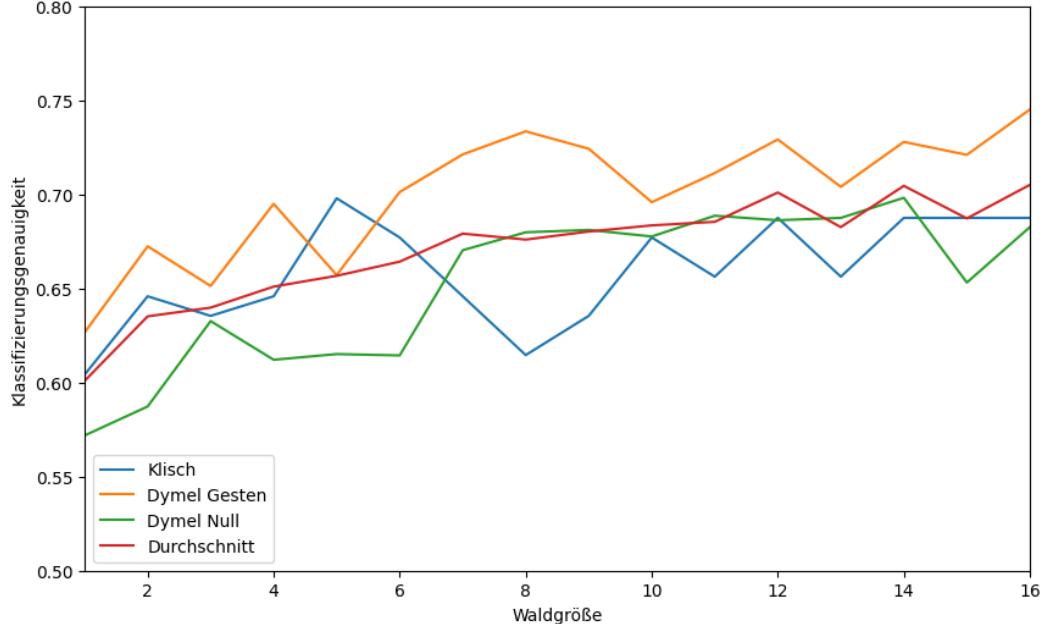
Die Featuremenge von Motion History beinhaltet für jeden Pixel einen Eintrag, die der Definition des Motion History Image folgen (Formel 3.1), wobei  $\tau = 100$  und  $\delta = \frac{\tau}{\#Bilder}$  ist.

Aus der Tabelle 5.1 sind die besten Konfigurationen jeder Kategorie zu entnehmen. Die beste Konfiguration wurde mit der Ensemble-Methode *ExtraTrees* gefunden. Sie erzielt eine Klassifizierungsgenauigkeit von 68,8% auf der Testmenge von Klisch, 74% auf der Gestentestmenge und 69% auf der Nullgestentestmenge. Im Vergleich zu der Helligkeitsverteilung wird

## 5 EVALUATION

Konfiguration	Beste	Unter 60 kB	Unter 28 kB
Ensemble-Methode	ExtraTrees	ExtraTrees	ExtraTrees
Maximalhöhe	10	11	9
Waldgröße	16	10	6
min_samples_leaf	1	4	1
Programmgröße in Bytes	84200	57436	22804
Genauigkeit Testmenge von Klisch	68,8%	67,7%	62,5%
Genauigkeit Gestentestmenge	74,5%	69,6%	68,5%
Genauigkeit Nullgestentestmenge	68,3%	67,8%	67,6%

■ **Tabelle 5.2:** Die besten Konfigurationen der Motion History.



■ **Abbildung 5.2:** Die besten Konfigurationen pro Waldgröße mit Motion History.

mehr Programmspeicher benötigt und die Gesamterkennungsgenauigkeit ist 1,84% geringer.

Wird die Kategorie *Beste* mit der Kategorie *Unter 28 kB* verglichen, nimmt die Gesamtklassifizierungsgenauigkeit nur um 4,3% ab. Dabei reduziert sich die Programmgröße um 72,9%. Abbildung 5.2 zeigt, dass die Klassifizierungsgenauigkeit im Durchschnitt sich mit zunehmender Waldgröße erhöht. Im Vergleich zur Helligkeitsverteilung, ist der Zuwachs größer. Wenn der Suchraum nicht auf eine Waldgröße von 16 Bäumen begrenzt wäre, würde die

beste Konfiguration vermutlich besser sein. Allerdings würde sich auch die Programmgröße signifikant erhöhen.

Die Motion History kann mit ausschließlich 8-Bit Integer implementiert werden und hat damit den geringsten WCET und Programmgröße pro Baum, weswegen die beste Konfiguration mit einer Waldgröße von 16 Bäumen nicht deutlich größer ist, als die der Helligkeitsverteilung mit einer Waldgröße von 10 Bäumen.

### 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
Genaugkeit Testmenge von Klisch	94,8%	94,8%	89,6%	87,5%
Genaugigkeit Gestentestmenge	97,0%	96,1%	95,6%	94,1%
Genaugigkeit Nullgestentestmenge	92,2%	91,1%	88,8%	89,9%

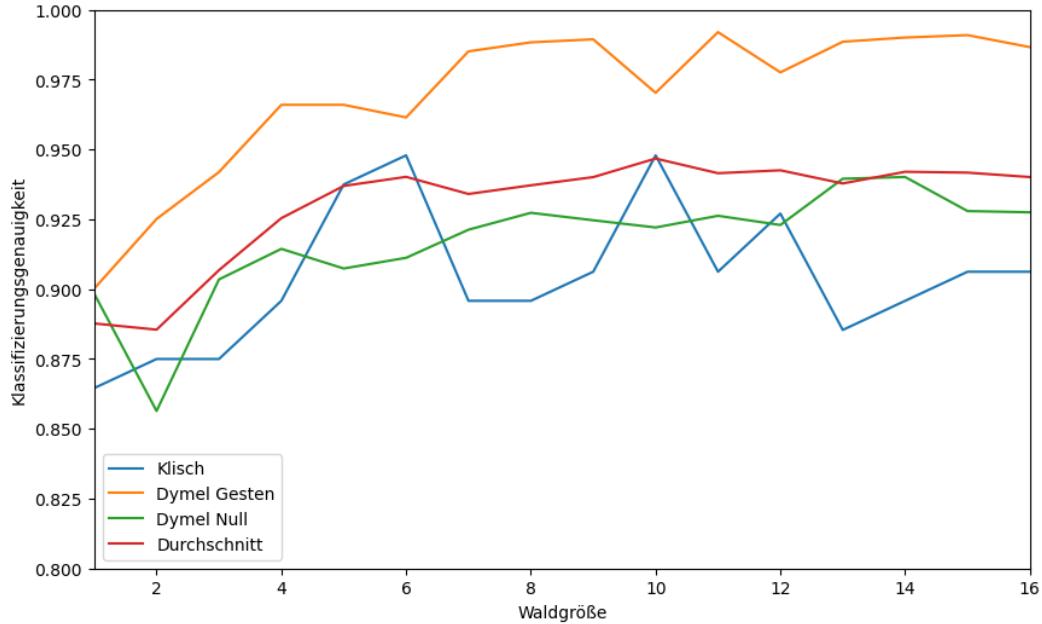
■ **Tabelle 5.3:** Die besten Konfigurationen der Schwerpunktverteilung mit Gleitkommazahlen.

Die Featuremenge Schwerpunktverteilung mit Gleitkommazahlen folgt der Definition aus Kapitel 4.2.2 und beinhaltet insgesamt 10 Einträge. Jeweils 2 Einträge bilden die X und Y Koordinate des Schwerpunktes. Damit spiegeln 10 Einträge insgesamt 5 Zeitfenster wieder.

Aus der Tabelle 5.3 sind die besten Konfigurationen jeder Kategorie zu entnehmen. Die beste Konfiguration wurde mit der Ensemble-Methode Boosting gefunden. Mit einer Klassifizierungsgenaugkeit von 94,8% auf der Testmenge von Klisch ist dieser Ansatz nur 4,2% schlechter als das neuronale Netz von Giese [Gie20]. Es ist anzumerken, dass mit einer kleineren Trainingsmenge ohne die Gestentrainingsmenge und Nullgestentrainingsmenge eine Lösung gefunden wurde, die 97,9% erzielte und damit nur 1,1% schlechter ist. Außerdem werden 97% der Gestentestmenge und 92,2% der Nullgestentestmenge korrekt klassifiziert.

Im Vergleich zu der Helligkeitsverteilung und Motion History ist die Klassifizierungsgenaugkeit dieses Ansatzes signifikant besser, sogar wenn nur 6656 Byte Programmspeicher verwendet werden. Wird die Kategorie *Beste* mit der Kategorie *Unter 14 kB* verglichen, nimmt

## 5 EVALUATION



**Abbildung 5.3:** Die besten Konfigurationen pro Waldgröße der Schwerpunktverteilung mit Gleitkommazahlen.

die Gesamtklassifizierungsgenauigkeit nur um 4,17% ab. Dabei wird die Programmgröße um 92% reduziert. Dies verspricht, dass mit zunehmender Waldgröße die Klassifizierungsgenauigkeit steigt. Abbildung 5.3 zeigt, dass zwar der Fall ist, aber schon ab einer Waldgröße von 6 Bäumen verzeichnet die durchschnittliche Klassifizierungsgenauigkeit keinen signifikanten Zuwachs mehr. Dementsprechend ist der Unterschied der Gesamtklassifizierungsgenauigkeit der Kategorie *Beste* und der Kategorie *Unter 44 kB* mit 0,7% nicht groß. Damit eignet sich dieses Feature gut für kleine eingebettete Systeme, da nur wenige Bäume benötigt werden.

### 5.1.4 Schwerpunktverteilung mit Ganzzahlen

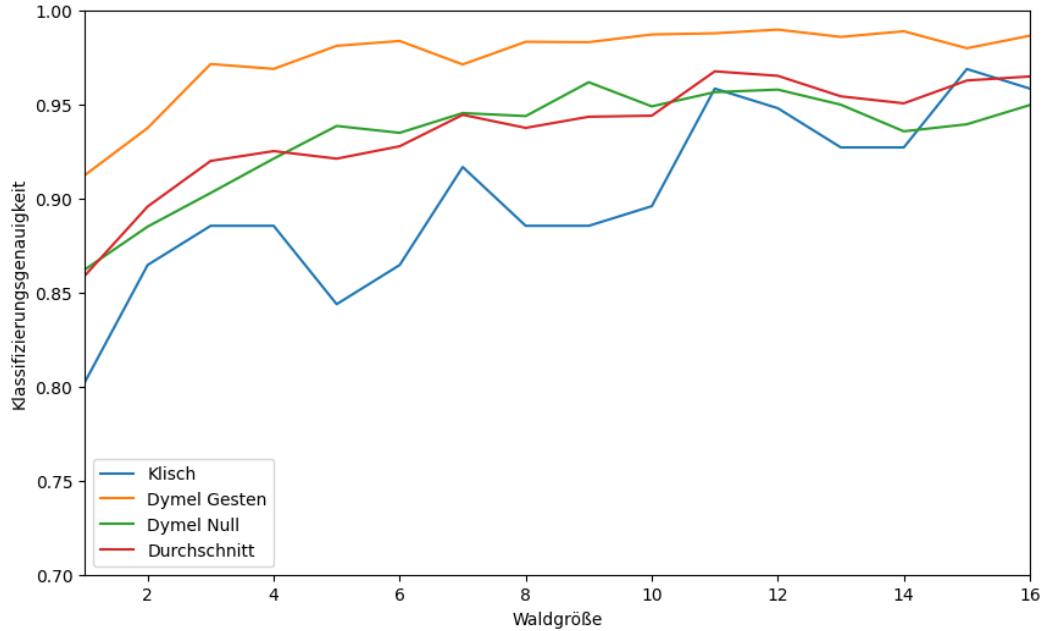
Die Featuremenge Schwerpunktverteilung mit Ganzzahlen folgt der Definition aus Kapitel 4.2.2 und beinhaltet insgesamt 10 Einträge. Jeweils 2 Einträge bilden die X und Y Koordinate des Schwerpunktes. Damit spiegeln 10 Einträge insgesamt 5 Zeitfenster wieder.

Aus der Tabelle 5.4 sind die besten Konfigurationen jeder Kategorie zu entnehmen. Die beste Konfiguration wurde mit der Ensemble-Methode ExtraTrees gefunden. Mit einer Klassifizierungsgenauigkeit von 95,8% auf der Testmenge von Klisch ist dieser Ansatz nur 3,2% schlechter als das neuronale Netz von Giese [Gie20]. Es wurde aber auch eine Konfiguration gefunden, die 96,9% der Testmenge von Klisch korrekt klassifiziert und damit nur 2,1% schlechter ist. Diese maximiert aber in keiner Kategorie die Gesamtklassifizierungsgenauigkeit.

## 5.1 ERKENNUNGSGENAUIGKEIT

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:** Die besten Konfigurationen der Schwerpunktverteilung mit Ganzzahlen.



■ **Abbildung 5.4:** Die besten Konfigurationen pro Waldgröße der Schwerpunktverteilung mit Ganzzahlen.

Außerdem werden 98,8% der Gestentestmenge und 95,6% der Nullgestentestmenge korrekt klassifiziert.

Der Ansatz mit Ganzzahlen erzielte eine 2,1% höhere Gesamtklassifizierungsgenauigkeit als der Ansatz mit Gleitkommazahlen. Der 16-Bit Integer Datentyp erlaubt der Schwerpunktverteilung mit Ganzzahlen unter jeder Restriktion größere Entscheidungswälder zu bilden,

## 5 EVALUATION

als die Schwerpunktverteilung mit Gleitkommazahlen. Abbildung 5.4 zeigt einen Zuwachs der durchschnittlichen Klassifizierungsgenauigkeit mit zunehmender Waldgröße. Es ist auszugehen, dass eine noch bessere Konfiguration gefunden werden könnte, wenn der Suchraum auf eine größere Waldgröße erweitert wird. Ähnlich wie Schwerpunktverteilung mit Gleitkommazahlen ist der Zuwachs der durchschnittlichen Klassifizierungsgenauigkeit ab einer Waldgröße von 7 Bäumen gering. Somit kann bereits bei einer geringen Programmgröße eine hohe Klassifizierungsgenauigkeit erzielt werden. Damit eignet sich die Schwerpunktverteilung mit Ganzzahlen ebenfalls für kleine eingebettete Systeme.

### 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:** Die besten Konfigurationen der kombinierten Schwerpunktverteilung.

Die kombinierte Schwerpunktverteilung vereint die Schwerpunktverteilung mit Ganzzahlen und Gleitkommazahlen. Das erscheint sinnvoll, da der Ansatz mit Ganzzahlen invariant zu einem Offset in der Helligkeit ist und der Ansatz mit Gleitkommazahlen invariant zur Skalierung der Helligkeit.

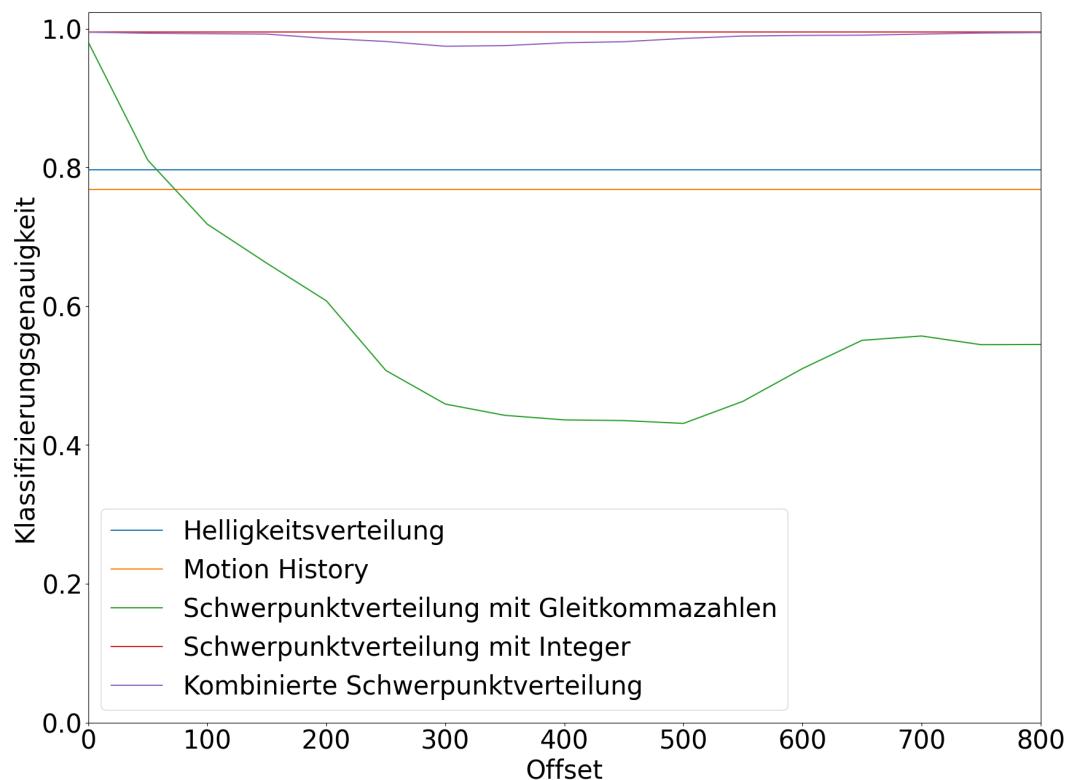
Es wird davon ausgegangen, dass der jeweilige Klassifizierer entweder ein Ergebnis mit einer deutlichen Mehrheit zurückgibt oder ein Ergebnis mit einer knappen Mehrheit. Für jedes Lichtverhältniss hat mindestens ein Klassifizierer eine deutliche Mehrheit. Damit erzielt die Kombination eine Mehrheit bei der korrekten Klasse. Die besten Konfigurationen der beiden Ansätze werden mit einem Wahlklassifizierer vereint. Das heißt, die Wahrscheinlichkeitsverteilungen der Wahlklassifizierer der jeweiligen Ansätze werden zu gleichen Anteilen addiert.

Die Kombination der besten Konfigurationen beider Ansätze erzielt eine Klassifizierungsgenauigkeit von 94,8% auf der Testmenge von Klisch. Dies entspricht der Klassifizierungsgenauigkeit der Schwerpunktverteilung mit Gleitkommazahlen. 99% der Gestentestmenge wird korrekt klassifiziert. Das ist besser als beide Ansätze. Die Nullgestentestmenge wurde

zu 95,8% korrekt klassifiziert. Dies entspricht der Klassifizierungsgenauigkeit der Schwerpunktverteilung mit Ganzzahlen. Bei der Kategorie *Unter 44 kB* und *Unter 28 kB* wurde der kombinierte Klassifizierer nie schlechter als der schlechteste Ansatz auf dem die kombinierte Schwerpunktverteilung basiert.

Der Nachteil dieses Ansatzes ist, dass sowohl die Featuremenge mit Gleitkommazahlen, als auch für Ganzzahlen benötigt wird. Zum einen müssen immer beide Features berechnet werden und zum anderen kann jeder Klassifizierer nur halb soviel Speicher nutzen. Dadurch ist die Klassifizierungsgenauigkeit jedes Klassifizierers potentiell geringer, als wenn es den vollständigen Speicher zur Verfügung hätte. Bei der Schwerpunktverteilungen ist aber bereits ab einer geringen Waldgröße kein signifikanter Zuwachs der Klassifizierungsgenauigkeit zu verzeichnen. Deswegen eignet sich die Schwerpunktverteilung besonders gut. Der Vorteil ist, dass der kombinierte Klassifizierer potentiell robuster ist.

### 5.1.6 Robustheit gegenüber Lichtverhältnisse

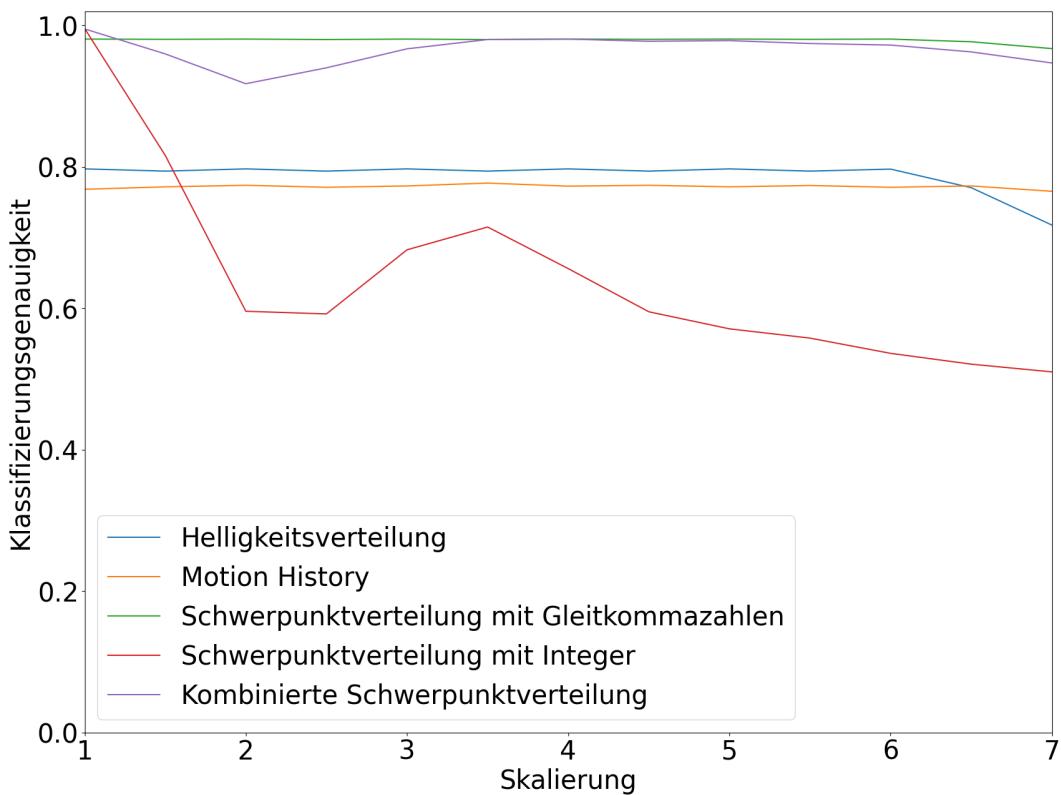


■ Abbildung 5.5: Ergebnisse des Offset der Helligkeitstestmenge 1 je Ansatz.

## 5 EVALUATION

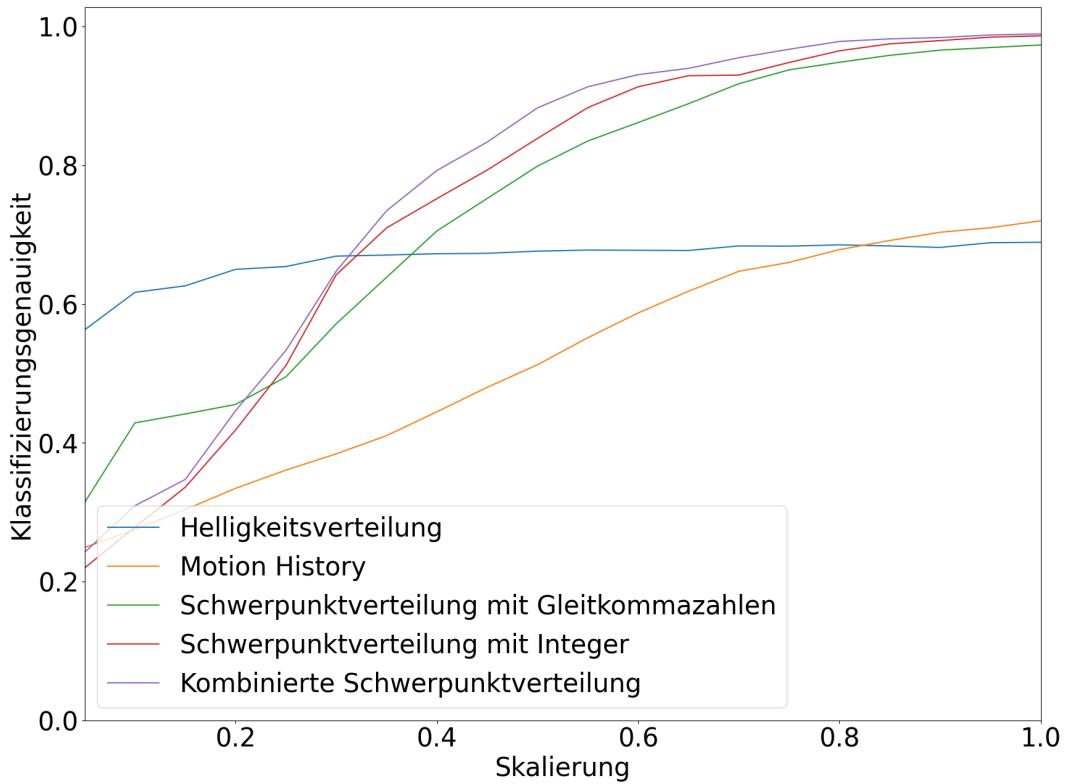
Dieses Kapitel validiert die Invarianzen der einzelnen Feature, die in Kapitel 4.2.2 diskutiert wurden. Mit Hilfe der Helligkeitstestmengen werden die Features validiert. Diese synthetischen Testmengen verändern die Helligkeit und den Kontrast einer bestehenden Testmenge durch Skalierung und Offsets.

Abbildung 5.5 zeigt, wie sich ein zunehmender Offset zwischen 0 und 800 auf die Featuremengen auswirkt. Dabei erhöht sich die Gesamthelligkeit, aber der Kontrast bleibt gleich. Die Helligkeitsverteilung, Motion History und Schwerpunktverteilung mit Ganzzahlen bleiben konstant. Daher wird geschlossen, dass diese invariant gegenüber Offset sind. Die kombinierte Schwerpunktverteilung ist sehr robust gegenüber einem Offset aber nicht invariant. Bei einem Offset zwischen 300 und 500 ist der stärkste Einbruch der Klassifizierungsgenauigkeit zu beobachten. Das ist auf die Schwerpunktverteilung mit Gleitkommazahlen zurückzuführen, die massive Einbrüche der Klassifizierungsgenauigkeit mit zunehmenden Offset erfährt. Zwischen 300 und 500 ist der Einbruch am größten. Die Schwerpunktverteilung mit Gleitkommazahlen ist sehr anfällig gegenüber einem Offset.



■ **Abbildung 5.6:** Ergebnisse der Skalierung der Helligkeitstestmenge 1 je Ansatz.

Abbildung 5.6 zeigt, wie sich ein zunehmender Skalierungsfaktor auf die Featuremengen auswirkt. Dabei erhöht sich sowohl die Gesamthelligkeit, als auch der Kontrast. Die Schwerpunktverteilung mit Gleitkommazahlen, die Helligkeitsverteilung und die Motion History zeigen eine Invarianz gegenüber Skalierung. Ab einem Faktor von 6,5 und 7 sind Einbrüche zu erkennen. Die Trainingsmenge enthält aber auch Einträge mit einer Helligkeiten oberhalb 160, sodass eine Skalierung mit 6,5 Clipping-Effekte erzeugt, d. h. Pixel mit einer Helligkeit über 1023 werden auf 1023 gesetzt. Die Schwerpunktverteilung mit Ganzzahlen erfährt starke Einbrüche der Klassifizierungsgenauigkeit, ähnlich wie Schwerpunktverteilung mit Gleitkommazahlen im Vergleich zum Offset, allerdings ist der maximale Einbruch der Klassifizierungsgenauigkeit geringer. Sie ist aber trotzdem sehr anfällig gegenüber der Skalierung. Die kombinierte Schwerpunktverteilung ist sehr robust gegenüber der Skalierung aber nicht invariant. Sie weist ebenfalls Einbrüche der Klassifizierungsgenauigkeit zwischen 6,5 und 7 auf aber auch zwischen 1,5 und 3,5. Dies ist auf die Schwerpunktverteilung mit Ganzzahlen zurückzuführen.



**Abbildung 5.7:** Ergebnisse der Helligkeitstestmenge 2 je Ansatz.

Abbildung 5.7 zeigt, wie sich sich ein abnehmender Kontrast bei gleichbleibender Helligkeit auf die Featuremengen auswirkt. Diese Situation ist vergleichbar mit einer zunehmenden

## 5 EVALUATION

Distanz zur Kamera, da dort der Kontrast durch Streulicht ebenfalls geringer wird. Keiner der Featuremengen weist eine Invarianz dem gegenüber auf. Die Helligkeitsverteilung ist bis zu einem Faktor von 0,3 sehr stabil. Ab 0,3 nimmt die Klassifizierungsgenauigkeit leicht ab. Bei einem initialen Kontrastunterschied von 100% ist ab 0,3 der Kontrast nurnoch 18% des ursprünglichen Kontrasts. Dementsprechend ist die Helligkeitsverteilung extrem robust gegenüber die Lichtverhältnisse. Im Vergleich sind die anderen Featuremengen sehr sensibel gegenüber der abnehmenden Skalierung. Die Schwerpunktverteilungen verhalten sich vom Trend her gleich, aber die kombinierte Schwerpunktverteilung erzielt wie Erwarten, fast durchweg, die beste Klassifizierungsgenauigkeit. Da keiner der Schwerpunktverteilungen invariant gegenüber einem Offset und der Skalerierung waren, war auch nicht zu erwarten, dass sie eine Invarianz in diesem Test zeigen. Trotzdem erzielen sie die beste Klassifizierungsgenauigkeit bis zu einem Faktor von 0,3. Von der Motion History war ein ähnliches Verhalten wie die Helligkeitsverteilung zu erwarten. Vermutet wird, dass die unterliegende Funktion um eine Bewegung zu detektieren zu restriktiv wird, wenn der Kontrast sinkt.

Kubik hat beobachtet, dass bei zunehmender Distanz der Kontrast geringer wird. Seine KNN haben mit abnehmenden Kontrast geringere Klassifizierungsgenauigkeiten erzielt [Kub19]. Dieses Verhalten wird durch die Helligkeitstestmenge 2 bestätigt. Erstaunlich robust erweist sich die Helligkeitsverteilung. Sie erzielt sie aber trotzdem eine geringer Klassifizierungsgenauigkeit als die Schwerpunktverteilungen.

### 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 leistungsschwächere Module verwendet werden, wodurch die Batterielaufzeit verlängert wird oder die Kosten reduziert werden. In dieser Arbeit wird das Arduino Board ATmega328P genutzt. Dieses Board 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 Vorgang das aktuelle Bild auszulesen, der Extrahierung der Features und der Ausführung des Klassifizierers. Für das Auslesen des Bildes wird eine konstante Zeit von 10 ms angenommen.

Die Auswertung bezieht sich auf die Instruktionen des Programms, die bei der Überset-

zung der Firmware durch den AVR GCC mit der Optimierungsstufe `O8` entstehen. Aus dem Handbuch des ATmega328P [Cor15] können für jede Instruktion die maximale Anzahl der Zyklen entnommen werden, die im schlimmsten Fall benötigt werden. Die Gesamtausführungszeit berechnet sich aus der Anzahl der Zyklen multipliziert mit der Zeit pro Zyklus, d. h. bei 16 MHz bedarf ein Zyklus  $0,0625 \mu\text{s}$ .

Im folgenden wird nur die Schwerpunktverteilung diskutiert, da diese die besten Klassifizierungsgenauigkeiten erzielten und auf dem ATmega328P implementiert wurden. Dabei wird zuerst auf den Ansatz mit Gleitkommazahlen eingegangen, dann auf den Ansatz mit Ganzzahlen und zuletzt auf den kombinierten Ansatz.

### 5.2.1 Operationen mit Gleitkommazahlen

Der ATmega328P verfügt über keine Hardwareunterstützung um Gleitkommazahlen zu verarbeiten. Dementsprechend muss der Kompiler Gleitkommazahlunterstützung durch Software ersetzen. Dies hat zu folge, dass Operationen auf Gleitkommazahlen sehr viele Zyklen benötigen im Vergleich zu Operationen auf Ganzzahlen. Operationen sind zum Beispiel Addition, Dividierung, Vergleiche oder Typkonvertierung.

Die Operationen arbeiten mit dem Datentyp `Float`. Dieser benötigt 32 Bit, damit er dargestellt werden kann. Der ATmega328P verfügt aber nur über 8-Bit Register. Zur Darstellung wird der Float in 4 hintereinander liegende Register aufgeteilt. Das hat zur Folge, dass jede Operation mit Gleitkommazahlen 4 mal soviele Instruktionen als ein 8-Bit Datentyp benötigt um die Float Operatoren in Register zu laden und Float zu speichern.

Für jede Operation sind Algorithmen in Form von Funktionen hinterlegt. Diese werden von dem Kompiler automatisch bei der Übersetzung mit verlinkt. In Tabelle 5.6 ist der experimentell erprobte WCET der Funktionen zu sehen, die bei der Extrahierung der Features und Ausführung des Klassifizierers verwendet werden. Zum Beispiel, die Addition von 8-Bit Integer benötigt nur 1 Zyklus. Im Vergleich benötigt die Addition von Gleitkommazahlen im schlimmsten Fall `_addsf3` 192 Zyklen. Zudem kommt noch ein Overhead von bis zu 4 Zyklen hinzu um die Funktion aufzurufen. Dementsprechend sind die Gleitkommaoperationen besonders teuer im Vergleich zu Hardwareunterstützten Operationen, weswegen sie vermieden werden sollten auf Systemen ohne Hardwareunterstützung.

### 5.2.2 Feature-Extrahierung

Die Feature-Extrahierung implementiert die Berechnung der 5 Zeitfenster für die Schwerpunktverteilung. Einerseits muss aus jedem Bild der Schwerpunkt berechnet werden und

## 5 EVALUATION

Operation	Funktion	WCET	WCET in Zyklen
__lesf2	Kleiner oder gleich Vergleich	4 $\mu s$	64
__floatsisf	Konvertierung auf Float	4 $\mu s$	64
__divsf3	Dividierung	36 $\mu s$	576
__addsf3	Addition	12 $\mu s$	192

■ **Tabelle 5.6:** Experimentell ermittelte WCET von Gleitkommaoperationen auf dem ATmega328P.

andererseits müssen die Schwerpunkte auf 5 Schwerpunkte zusammengefasst werden, die die 5 Zeitfenster repräsentieren.

Jedes mal wenn ein Bild aufgenommen wird, wird der Schwerpunkt dieses Bildes berechnet und gespeichert. Dies reduziert einerseits die WCET, da im WCEP weniger Schwerpunkte berechnet werden müssen, und andererseits wird weniger Pufferspeicher benötigt pro Bild. Für Gleitkommazahlen reduziert sich der Verbrauch pro Bild von 18 Byte auf 8 Byte und für Ganzzahlen auf 4 Byte. Der kombinierte Ansatz muss beide Schwerpunkte speichern. Die jeweiligen Schwerpunktkoordinaten berechnen sich mit der in Kapitel 4.2.2 beschriebenen Formel. Dabei muss die Summe der Pixel einmalig berechnet werden pro Bild und jeweils die berechnete X und Y Koordinate im Puffer für den derzeitige Schwerpunkt gespeichert werden. Listing 5.1 zeigt, wie dies auf dem ATmega328P implementiert ist. Insgesamt werden bei der Schwerpunktverteilung mit Gleitkommazahlen 201 Zyklen für die einzelnen Instruktionen benötigt (12,5625  $\mu s$ ). Zusätzlich wird \_\_floatsisf 6 mal aufgerufen, \_\_lesf2 und \_\_divsf3 jeweils 2 mal aufgerufen. Der WCET zur Schwerpunktberechnung eines Bildes beläuft sich damit auf 116,5625  $\mu s$ , davon werden 104  $\mu s$  für Gleitkommaoperationen aufgewendet. Der Ansatz mit Ganzzahlen benötigt keine Gleitkommaoperationen und 57 Zyklen weniger, da die summe der Pixel nicht berechnet werden muss, d. h. es werden im WCET nur 8,875  $\mu s$  benötigt. Für den kombinierten Ansatz werden zusätzlich 4 Speicherinstruktionen benötigt, die einen Overhead von 0,25  $\mu s$  erzeugen, d. h. es werden im WCET 116,8125  $\mu s$  benötigt.

```
short helligkeits_summe = 0;
for (char i = 0; i < 9; ++i)
    helligkeits_summe += bild_puffer[i];
schwerpunkt_puffer_x[anzahl_bilder_im_puffer] = (float)(bild_puffer[0] + \
    bild_puffer[3] + bild_puffer[6] - bild_puffer[2] - bild_puffer[5] - \
    bild_puffer[8]) / ((float)helligkeits_summe);
schwerpunkt_puffer_y[anzahl_bilder_im_puffer] = (float)(bild_puffer[0] + \
    bild_puffer[1] + bild_puffer[2] - bild_puffer[6] - bild_puffer[7] - \
    bild_puffer[8]) / ((float)helligkeits_summe);
```

■ **Listing 5.1:** Implementierung um den Schwerpunkt für ein Bild zu berechnen.

Wenn ein Handgestenkandidat detektiert wurde, wird für jedes Zeitfenster der Durchschnitt der darin enthaltenden Schwerpunkte berechnet. Die daraus berechneten Schwerpunkte werden als Schwerpunktverteilung bezeichnet. Listing 5.2 zeigt den Algorithmus, um die Schwerpunktverteilung aus den Schwerpunkten im Puffer zu berechnen. Zunächst wird bei der Initialisierungsphase das `zusammenfass_muster` berechnet (Kap 4.2.2). Dafür werden im schlimmsten Fall 123 Zyklen für die einzelnen Instruktionen benötigt ( $7,6875 \mu s$ ) und  $20 \mu s$  für die Ganzzahldividierung `_divmodhi4`. Insgesamt  $27,6875 \mu s$ . Dieser Teil wird genau 1 mal für alle Richtungen durchgeführt und Schwerpunktverteilungen durchgeführt. Die innere Schleife wird im schlimmsten Fall für die Gesamtgröße des Schwerpunktpuffers durchlaufen. Jeder Durchlauf benötigt im schlimmsten Fall 27 Zyklen für die einzelnen Instruktionen ( $1,6875 \mu s$ ) und führt `_addsf3` einmal aus. Der WCET für einen Durchlauf beläuft sich damit auf  $13,6875 \mu s$ . Der Ansatz mit Ganzzahlen benötigt im schlimmsten Fall 17 Zyklen ( $1,125 \mu s$ ). Bei einer Gesamtpuffergröße von 125 wird für den Teil der inneren Schleife für die Schwerpunktverteilung mit Gleitkommazahlen  $1710,9375 \mu s$  benötigt, für die Schwerpunktverteilung mit Ganzzahlen  $140,625 \mu s$  und für den kombinierten Ansatz  $1851,5625 \mu s$ . Die äußere Schleife benötigt im schlimmsten Fall 57 Zyklen für die einzelnen Instruktionen ( $3,5625 \mu s$ ) und ruft im Ansatz mit Gleitkommazahlen 5 mal `_floatsisf` und `_divsf3` auf und im Ansatz mit Ganzzahlen 5 mal `_divmodhi4`. Damit beläuft sich der WCET bei 5 Durchläufen der äußeren Schleife für den Ansatz mit Gleitkommazahlen auf  $217,8125 \mu s$ , für den Ansatz mit Ganzzahlen auf  $117,8125 \mu s$  und für den kombinierten Ansatz  $335,625 \mu s$ .

```
Initialisierung.
for (char i = 0; i < 5; ++i) { // Äußere Schleife
    features[i] = 0;
    for (char j = 0; j < zusammenfass_muster[i]; ++j) // Innere Schleife
        features[i] += *(schwerpunkt_puffer_x++);
    features[i] /= ((float)zusammenfass_muster[i]);
}
```

- **Listing 5.2:** Algorithmus um die Schwerpunktverteilung in horizontaler Richtung zu berechnen.

Der Schwerpunkt wird jeweils für die horizontale und vertikale Richtung berechnet. Der kombinierte Ansatz berechnet sowohl den Schwerpunkt für Gleitkommazahlen als auch für Ganzzahlen. Die WCET für die Feature-Extrahierung der Schwerpunktverteilung mit Gleitkommazahlen beläuft sich auf  $4001,75 \mu s \approx 4 \text{ ms}$ . Die WCET der Schwerpunktverteilung mit Ganzzahlen beläuft sich auf  $553,4375 \mu s \approx 0,6 \text{ ms}$ . Die WCET der kombinierten Schwerpunktverteilung beläuft sich auf  $4518,875 \mu s \approx 4,5 \text{ ms}$ .

## 5 EVALUATION

### 5.2.3 Ausführung eines Entscheidungbaumes

Der WCEP eines Entscheidungbaumes ist der längste Pfad. Entlang des Pfades werden Vergleiche durchgeführt, bis im Blatt das Klassifizierungsergebnis zurückgegeben wird. Insgesamt sind die Anzahl der Vergleiche gleich der Höhe des Entscheidungbaumes.

Jeder Vergleich besteht aus 3 Teilen. Der erste Teil ist der Vergleichsoperation. Der zweite das Laden der Operatoren, d. h. das Feature und der Schwellenwert und der dritte Teil sind die Abzweigungsinstruktionen. Für die Schwerpunktverteilung mit Gleitkommazahlen werden 19 Zyklen für das Laden der Operatoren, den Aufruf der Vergleichsfunktion und die Abzweigungsinstruktionen benötigt ( $1,1875 \mu s$ ). Die Vergleichsfunktion ist `_lesf2`. Insgesamt beläuft sich die WCET für ein Vergleich auf  $5,1875 \mu s$ . Die Schwerpunktverteilung mit Ganzzahlen benötigt für alle Teile insgesamt 15 Zyklen, d. h. die WCET beläuft sich auf  $0,9375 \mu s$  pro Vergleich.

Zusätzlich kommt noch Overhead hinzu, der durch den Funktionsaufruf entsteht und die Rückgabe des Ergebnisses im Blatt. Im schlimmsten Fall sind das 27 Zyklen ( $1,6875 \mu s$ ). Damit beläuft sich die WCET auf  $1,6875 \mu s + \text{Maximale Baumhöhe} \cdot 5,1875 \mu s$  bzw.  $0,9375 \mu s$ .

### 5.2.4 Ausführung eines Entscheidungswaldes

Der WCEP eines Entscheidungswaldes setzt sich aus dem WCEP des Wahlklassifizierungsalgorithmus und dem WCEP jedes Entscheidungsbäumes 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 Entscheidungsbäumes bedarf 18 Zyklen ( $1,125 \mu s$ ), um die Funktion, die den Entscheidungsbäum ausführt, aufzurufen und das Ergebnis des ausgeführten Entscheidungsbäumes auf die Gesamtsumme zu addieren. Die restlichen Instruktionen bedürfen 64 Zyklen ( $4 \mu s$ ).

Die WCET eines Entscheidungswaldes beläuft sich damit auf  $4 \mu s + \#\text{Entscheidungsbäume} \cdot (1,125 \mu s + \text{WCET der Entscheidungsbäume})$ .

### 5.2.5 Gesamtausführungszeit und Optimierung

Der beste Klassifizierer der Schwerpunktverteilung mit Gleitkommazahlen mit einer Programmgröße unterhalb von 28 kB hat eine Maximalhöhe von 10 und eine Waldgröße von 4

Bäumen. Die WCET dieses Entscheidungswaldes beläuft sich damit auf  $4224,5 \mu s \approx 4,2$  ms. Das ist 2,2 ms, bzw. 34,4% schneller als das schnellste neuronale Netz von Giese [Gie20]. Die Ausführungszeit könnte deutlich reduziert werden, indem Festkommaarithmetik mit 16-Bit Festkommazahlen verwendet werden würde.

Der beste Klassifizierer der Schwerpunktverteilung mit Ganzzahlen mit einer Programmgröße unterhalb von 28 kB hat eine Maximalhöhe von 12 und eine Waldgröße von 8 Bäumen. Die WCET dieses Entscheidungswaldes beläuft sich damit auf  $669,9375 \mu s \approx 0,7$  ms. Das ist 5,7 ms, bzw. 89,1% schneller als das schnellste neuronale Netz von Giese. Es ist anzumerken, dass in der Praxis die Puffergröße bei diesem Ansatz größer sein kann, da nur 4 Bytes pro Schwerpunkt benötigt werden. Dadurch erhöht sich die WCET.

Der beste Klassifizierer der kombinierten Schwerpunktverteilung mit einer Programmgröße unterhalb von 28 kB vereint die Klassifizierer der Kategorie *Unter 14 kB*. Der Entscheidungswald der Schwerpunktverteilung mit Gleitkommazahlen hat eine Maximalhöhe von 7 und eine Waldgröße von 3 Bäumen. Der Entscheidungswald der Schwerpunktverteilung mit Ganzzahlen hat eine Maximalhöhe von 12 und eine Waldgröße von 3 Bäumen. Bei der Berechnung für die WCET muss insgesamt 3 mal der Wahlklassifizierer angewendet werden, wobei nur der letzte den Overhead von  $4 \mu s$  hat, um die Klasse mit der höchsten Wahrscheinlichkeit zu identifizieren. Damit beläuft sich die WCET auf  $4684,6875 \mu s \approx 4,7$  ms. Das ist 1,7 ms, bzw. 26,6% schneller als das schnellste neuronale Netz von Giese. Es ist anzumerken, dass in der Praxis die Puffergröße bei diesem Ansatz geringer ist, da 12 Bytes pro Schwerpunkt benötigt werden. Dadurch reduziert sich die WCET.

Der größte Anteil der WCET ist die Feature-Extrahierung. Dementsprechend können Entscheidungswälder beliebig groß skaliert werden, solange es der Programmspeicher zulässt. Der Klassifizierer mit der Schwerpunktverteilung mit Ganzzahlen ist 85,1% schneller als der Klassifizierer mit der Schwerpunktverteilung mit Gleitkommazahlen. Dadurch ist der kombinierte Klassifizierer nur insignifikant langsamer als der Ansatz mit Gleitkommazahlen.

## 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.

## 5 EVALUATION

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 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

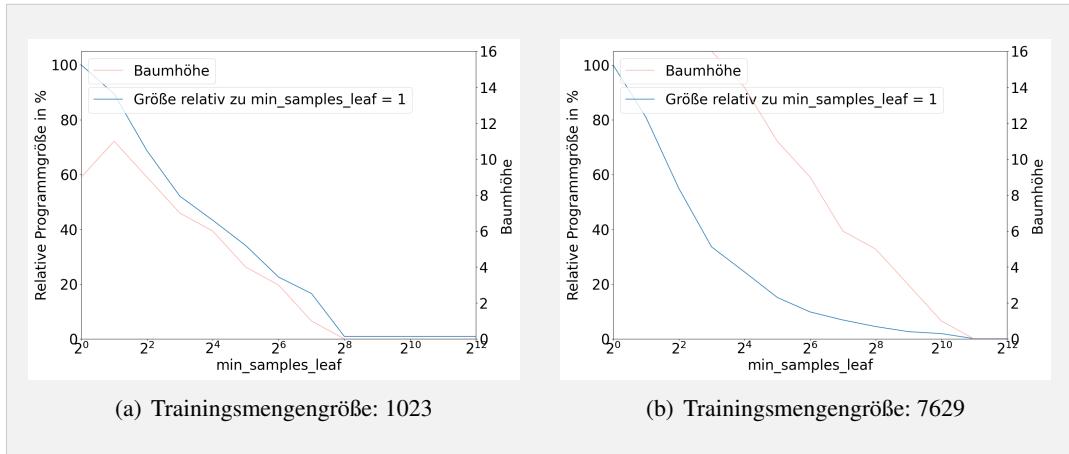
Standardmäßig wächst ein Entscheidungsbaum solange es mindestens eine Teilung gibt, die die Erkennungsgenauigkeit erhöht. Das führt zu sehr großen und sehr granularen Entscheidungsbäumen, die die Trainingsmenge auswendig lernen. Zudem sind die Bäume meistens unbalanciert es gibt viele Teilungen, die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

Damit eine Teilung stattfinden kann, muss der Knoten mindestens `min_samples_leaf` Einträge enthalten. Standardmäßig ist der Wert 1 [Ent20b]. Abbildung 5.8 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.



**Abbildung 5.8:** Auswirkung von `min_samples_leaf` Parameter auf resultierende Programmgröße und Baumhöhe.

```

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 5.3:** Vergleich von Gleitkommazahl-Feature mit konstanter Gleitkommazahl.

Listing 5.3 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

## 5 EVALUATION

32 Register. Bei einer Featuremenge von 10 werden dementsprechend regelmäßig Register verdrängt wodurch zusätzliche Instruktionen entstehen. Die Anzahl der Instruktionen können 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 5.4:** 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 teuere Gleitkommazahlvergleichsfunktion. Mit einem kleineren Datentyp können dementsprechend Instruktionen vermieden werden. Listing 5.4 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 5.5:** Beispiel des Assemblycodes der Rückgabe der Wahrscheinlichkeitsverteilung eines Entscheidungsbaumes.

Listing 5.5 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 5.6:** 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 5.6). 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

- [ABC<sup>+</sup>16] ABADI, Martín ; BARHAM, Paul ; CHEN, Jianmin ; CHEN, Zhifeng ; DAVIS, Andy ; DEAN, Jeffrey ; DEVIN, Matthieu ; GHEMAWAT, Sanjay ; IRVING, Geoffrey ; ISARD, Michael u. a.: Tensorflow: A system for large-scale machine learning. In: *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, S. 265–283
- [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
- [BFSO84] BREIMAN, Leo ; FRIEDMAN, Jerome ; STONE, Charles J. ; OLSHEN, Richard A.: *Classification and regression trees*. CRC press, 1984
- [Bre96] BREIMAN, Leo: Bagging predictors. In: *Machine learning* 24 (1996), Nr. 2, S. 123–140
- [Bre01] BREIMAN, Leo: Random forests. In: *Machine learning* 45 (2001), Nr. 1, S. 5–32
- [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. <http://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

## LITERATURVERZEICHNIS

- [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 as Multiple Feed Forward Networks. (2020), 01, S. 1–39
- [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
- [PGP98] PEI, M ; GOODMAN, ED ; PUNCH, WF: Feature extraction using genetic algorithms. In: *Proceedings of the 1st International Symposium on Intelligent Data Engineering and Learning, IDEAL* Bd. 98, 1998, S. 371–384
- [PSH97] PAVLOVIC, Vladimir I. ; SHARMA, Rajeev ; HUANG, Thomas S.: Visual interpretation of hand gestures for human-computer interaction: A review. In: *IEEE Transactions on pattern analysis and machine intelligence* 19 (1997), Nr. 7, S. 677–695
- [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
- [Qui86] QUINLAN, J. R.: Induction of decision trees. In: *Machine learning* 1 (1986), Nr. 1, S. 81–106
- [Qui90] QUINLAN, J R.: Decision trees and decision-making. In: *IEEE Transactions on Systems, Man, and Cybernetics* 20 (1990), Nr. 2, S. 339–346
- [Qui14] QUINLAN, J R.: *C4. 5: programs for machine learning*. Elsevier, 2014
- [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
- [SSS<sup>+</sup>17] SILVER, David ; SCHRITTWIESER, Julian ; SIMONYAN, Karen ; ANTONOGLOU, Ioannis ; HUANG, Aja ; GUEZ, Arthur ; HUBERT, Thomas ; BAKER, Lucas ; LAI, Matthew ; BOLTON, Adrian u. a.: Mastering the game of go without human knowledge. In: *nature* 550 (2017), Nr. 7676, S. 354–359
- [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