

Forschungsprojekt und Seminar

Computer Science

# Handgestenerkennung mit Entscheidungsbäumen

von

Tom Dymel

Februar 2021

Betreut von

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

Erstprüfer

Prof. Dr. Volker Turau

Institute of Telematics  
Hamburg University of Technology



# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>1</b>
<b>2 Entscheidungsbäume</b>	<b>3</b>
2.1 Scikit-Learn . . . . .	3
2.2 Einzelne Entscheidungsbäume . . . . .	4
2.3 Ensemble-Methoden . . . . .	5
<b>3 Gestenerkennung</b>	<b>9</b>
3.1 Gestenerkennung mit Entscheidungsbäumen . . . . .	9
3.2 Optische Handgestenerkennung . . . . .	11
3.3 Extrahieren von Gestenkandidaten . . . . .	11
3.4 Skalieren des Gestenkandidaten . . . . .	13
3.5 Trainings- und Testmenge . . . . .	14
3.6 Handgestenerkennung mit künstliche neuronalen Netzen . . . . .	15
<b>4 Handgestenerkennung mit Entscheidungsbäumen</b>	<b>19</b>
4.1 Modell . . . . .	19
4.1.1 Training . . . . .	20
4.1.2 C-Code Generierung eines Entscheidungsbaumes . . . . .	21
4.1.3 C-Code Generierung eines Entscheidungswaldes . . . . .	22
4.2 Features . . . . .	23
4.2.1 Feature Verbesserungen . . . . .	24
4.2.2 Featureauswahl . . . . .	24
4.3 Erstellte Werkzeuge und Code-Bibliotheken . . . . .	28
4.4 Aufgenommene Trainings- und Testmengen . . . . .	31
<b>5 Evaluation</b>	<b>35</b>
5.1 Klassifizierungsgenauigkeit . . . . .	35
5.1.1 Helligkeitsverteilung . . . . .	36
5.1.2 Motion History . . . . .	37
5.1.3 Schwerpunktverteilung mit Gleitkommazahlen . . . . .	39
5.1.4 Schwerpunktverteilung mit Ganzzahlen . . . . .	40
5.1.5 Kombinierte Schwerpunktverteilung . . . . .	42
5.1.6 Robustheit gegenüber Lichtverhältnissen . . . . .	43
5.2 Ausführungszeit . . . . .	47
5.2.1 Operationen mit Gleitkommazahlen . . . . .	47
5.2.2 Feature-Extrahierung . . . . .	48
5.2.3 Ausführung eines Entscheidungsbaumes . . . . .	50
5.2.4 Ausführung eines Entscheidungswaldes . . . . .	51
5.2.5 Gesamtausführungszeit und Optimierung . . . . .	51

## INHALTSVERZEICHNIS

5.3	Programmgröße . . . . .	52
5.3.1	Maximierung des Zuwachses der Klassifizierungsgenauigkeit . . . . .	52
5.3.2	Minimierung der Instruktionen eines Vergleichs . . . . .	53
5.3.3	Minimierung der Instruktionen einer Rückgabe . . . . .	55
<b>6</b>	<b>Diskussion</b>	<b>59</b>
<b>7</b>	<b>Schlussfolgerungen</b>	<b>61</b>
<b>A</b>	<b>Inhalt des USB-Sticks</b>	<b>63</b>
	<b>Literaturverzeichnis</b>	<b>65</b>

# Einleitung

Maschinelles Lernen (ML) gewann in den vergangenen Jahren an Popularität, u.a. durch die Fortschritte im parallelen Rechnen, sinkende Speicherpreise und schnelleren Speicher. Zudem sind gute ML-Bibliotheken frei verfügbar, wie Scikit-Learn [PVG<sup>+</sup>11] und TensorFlow [ABC<sup>+</sup>16], die den Einstieg in maschinelles Lernen erleichtern. Ein namhaftes Beispiel für das Potenzial maschinellen Lernens ist *AlphaGo Zero*, das einen Sieg gegen den besten menschlichen Spieler im Brettspiel Go erringen konnte. Das galt als besonders schwierig für Computer, da der Suchraum möglicher Aktionen sehr groß ist [SSS<sup>+</sup>17].

Ein Anwendungsgebiet von ML in eingebetteten Systemen ist die optische Gestenerkennung, die zur kontaktlosen Interaktion genutzt wird [PSH97]. Die eingesetzten Mikrocontroller sind jedoch häufig nicht ausreichend leistungsstark, um ein trainiertes Modell in akzeptabler 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 zum Rechen-Cluster gesendet [VKK<sup>+</sup>20]. Nachteile dieses Ansatzes sind einerseits die Abhängigkeit zu dieser Infrastruktur und andererseits vergrößert sich die Latenz durch die zusätzliche Kommunikation. Alternativ können die Modelle lokal ausgeführt werden. Dies erfordert aber, dass die Komplexität des Modells reduziert wird, sodass eine akzeptable Ausführungszeit gewährleistet werden kann.

In dieser Arbeit wird untersucht, wie Handgesten mit Entscheidungsbäumen auf kleinen Mikrocontrollern erkannt werden können. Es wird vermutet, dass Entscheidungsbäume schneller sind als künstliche neuronale Netze (KNN) und trotzdem eine akzeptable Klassifizierungsgenauigkeit erzielen. Betrachtet werden muss die Leistung im Hinblick auf Klassifizierungsgenauigkeit, Ausführungszeit und Ressourcenverbrauch. Dafür ist ein Konzept zur Übersetzung von Entscheidungsbäumen in Programmcode auf dem Mikrocontroller nötig. Es muss

## 1 EINLEITUNG

analysiert werden, welche Methode am besten zur Konstruktion eines Klassifizierers mit Entscheidungsbäumen geeignet ist. Als Eingabe für die Entscheidungsbäume müssen Features gefunden werden, die die einzelnen Handgesten unterscheidbar machen.

Insgesamt wurden 22528 verschiedene Konfigurationen analysiert, bestehend aus Kombinationen von Feature-Mengen, Hyperparametern und Ensemble-Methoden für das Training von Entscheidungsbäumen. Betrachtet wurden 30 Variationen von Features. Jede Konfiguration wurde auf Klassifizierungsgenauigkeit und Ressourcenverbrauch hin analysiert. Von jeder Feature-Menge wurden die Konfigurationen mit der höchsten Klassifizierungsgenauigkeit ausgewertet, die innerhalb der Speicherrestriktionen des Mikrocontrollers möglich waren. Die beste Konfiguration wurde auf ihre Worst-Case-Execution-Time (WCET) auf dem Mikrocontroller untersucht. Dabei wurden verschiedene Optimierungen diskutiert, um den Ressourcenverbrauch und die WCET zu minimieren. Bei diesen Arbeiten ist eine komplexe Infrastruktur entstanden, die in Rust und Python implementiert wurde. Diese stellt Code-Bibliotheken und Werkzeuge bereit, um die Rohdaten der Handgesten zu verarbeiten und ML Modelle zu trainieren und zu validieren. Es wurde ein Codegenerator implementiert, der C-Code für ein ML Modell mit Entscheidungsbäumen generiert. Außerdem wurden 14410 Handgesten aufgenommen, zwei neue Trainingsmengen und fünf neue Testmengen erzeugt.

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 ML Modells mit Entscheidungsbäumen, die Tauglichkeit von Features und die Infrastruktur eingegangen, sowie die neu erstellte Trainings- und Testmenge erläutert. Darauf folgt die Evaluation der Klassifizierungsgenauigkeit, Ausführungszeit und des Ressourcenverbrauchs in Kapitel 5. Kapitel 6 enthält einen kritischen Rückblick auf die Entscheidungen dieser Arbeit, bevor Kapitel 6 Schlussfolgerungen zieht.

# Entscheidungsbäume

Ein 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 Entscheidungbaums. Es wird unterschieden zwischen Bäumen, die versuchen eine der vordefinierten Klassen zu klassifizieren (Klassifizierer), und solchen, die versuchen den nächsten Wert vorherzusagen (Regressoren).

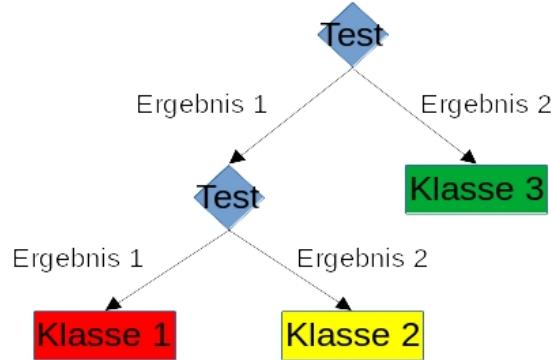
Die Konstruktion eines optimalen binären Entscheidungsbäumes ist NP-Vollständig [LR76]. Aus diesem Grund werden bei der Konstruktion Heuristiken verwendet, die nur lokal die beste Entscheidung treffen. Folglich ist es sehr aufwändig den optimalen Klassifizierer mit Entscheidungsbäumen zu finden. Ensemble-Methoden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

## 2.1 Scikit-Learn

In dieser Arbeit wird die Python ML-Bibliothek *Scikit-Learn* verwendet. Scikit-Learn bietet verschiedene ML Algorithmen mit einem High-Level Interface an [PVG<sup>+</sup>11]. Unter anderem konstruiert Scikit-Learn Entscheidungsbäume mit dem Algorithmus von CART [Ent20a]. Scikit-Learn kann Klassifizierer (`DecisionTreeClassifier`) und Regressoren (`DecisionTreeRegressor`) generieren.

Relevant für diese Arbeit ist nur der `DecisionTreeClassifier`. Dieser bietet eine Reihe an Hyperparametern an, um die Konstruktion des Entscheidungsbäumes zu steuern. Verwendet werden `max_depth` und `min_samples_leaf`. Der Hyperparameter `max_depth` beschränkt die maximale Baumhöhe und `min_samples_leaf` beschränkt die minimale Blattgröße. Die Blattgröße ist die kleinste Anzahl von Einträgen, bei der ein Knoten noch

## 2 ENTSCHEIDUNGSBÄUME



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

geteilt werden darf [Ent20c]. Diese Parameter sind relevant, weil sie die Größe der Entscheidungsbäume beeinflussen. Je größer ein einzelner Entscheidungsbaum ist, desto mehr Speicher verbraucht er. Das heißt, wenn der Speicher begrenzt ist, kann man weniger Entscheidungsbäume im Ensemble haben.

Scikit-Learn implementiert viele Ensemble-Methoden, die in Kombination mit dem Klassifizierer mit Entscheidungsbäumen genutzt werden können. Ihr Interface ist sehr ähnlich. Alle bieten `n_estimators` an, welche die Größe des Ensembles bestimmt, bzw. die Waldgröße. Denn ein Ensemble von Entscheidungsbäumen bildet einen Entscheidungswald [Ent20b].

## 2.2 Einzelne Entscheidungsbäume

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

Beim maschinellen Lernen werden Entscheidungsbäume aus Trainingsmengen generiert. Die Trainingsmenge besteht aus Feature-Mengen, die mit Klassen beschriftet sind [Ste09]. Sie sollte möglichst repräsentativ für die Gesamtmenge des Trainierten sein, d. h. im Falle der Handgesten, sollten möglichst alle Handgesten, Lichtverhältnisse und Geschwindigkeiten abgedeckt sein. Daneben müssen die Features eine eindeutige Partitionierung aller Klassen ermöglichen. Außerdem sollte die Feature-Menge keine irrelevanten Features enthalten, da diese die Effektivität stark beeinträchtigen können. Sind alle Anforderungen erfüllt, ist eine gute Generalisierung möglich [PGP98].

Es gibt verschiedene Algorithmen, um Entscheidungsbäume zu erzeugen, z. B. ID3 [Qui86], C4.5 [Qui14] oder CART [BFSO84]. Das Grundprinzip ist dabei immer das gleiche. Partitioniere die Trainingsmenge, sodass möglichst nur Einträge mit der gleichen Beschriftung in einer Partitionierung enthalten sind. Die Algorithmen unterscheiden sich 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. Aus diesem Grund werden Heuristiken verwendet, die schnell sind aber nicht optimal sein müssen [Qui86].

Scikit-Learn implementiert eine optimierte Version des CART (Classification and Regression Trees) Algorithmus [Ent20a]. CART partitioniert die Trainingsmenge und wählt dabei immer lokal die beste Teilung aus, d. h. es wird der beste Schwellenwert ausgewählt, der die Menge teilt.

```

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

```

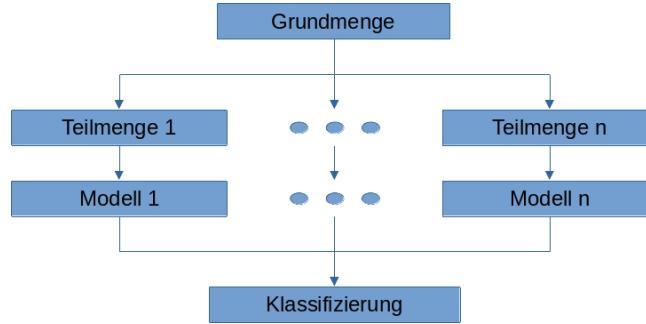
■ **Listing 2.1:** Skizze vereinfachten Baumwachstumsalgorithmus [Ste09].

Listing 2.1 skizziert den vereinfachten Baumwachstumsalgorithmus von CART. Der Algorithmus teilt solange die Trainingsmenge, bis keine weitere Teilung mehr möglich ist oder in allen Knoten nur Einträge mit der gleichen Beschriftung enthalten sind. Danach werden sukzessiv Teilbäume entfernt, die nach einer Bewertungsfunktion, z. B. Zuwachs der Klassifizierungsgenauigkeit, unterhalb eines vordefinierten Schwellenwert liegen [Ste09].

## 2.3 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].

## 2 ENTSCHEIDUNGSBÄUME



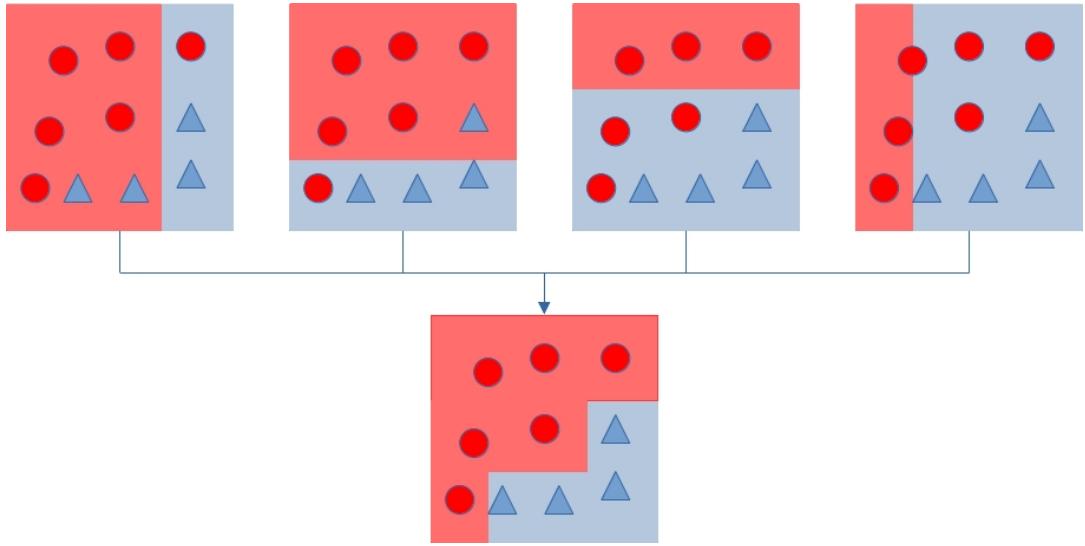
**Abbildung 2.2:** Klassifizierungsprozess mit der Bagging-Methode.

Ensemble-Methoden haben einen Wahlklassifizierer  $H(x) = w_1 h_1(x) + \dots + w_K h_K(x)$ , der die Menge von Lösungen  $\{h_1, \dots, h_K\}$  zusammenfasst 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 (**Bootstrap aggregating**) ist ein Verfahren, das Entscheidungsbäume mit unterschiedlichen Trainingsdaten konstruiert. Abbildung 2.2 illustriert die Bagging Methode für  $n$  Klassifizierer. Zuerst wird die Trainingsmenge aufgeteilt in  $n$  Teilmengen. Auf jeder Teilmenge wird ein Klassifizierer trainiert. Schließlich werden die Ergebnisse jedes Klassifizierers aggregiert, z. B. mit Hilfe des Wahlklassifizierers [Bre96]. Die Methode, die dahinter steht, nennt sich „Bootstrap sampling“, welche einen Prozess beschreibt aus einer Grundmenge,  $l$ -mal jeweils  $k$  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 an Features ausgewählt. Auf dieser Basis wird eine Menge an Entscheidungsbäumen generiert die anschließend aggregiert werden [Bre01].

Extremely Randomized Trees (ExtraTrees) gehen im Vergleich zu Random Forest einen Schritt weiter. Anstatt den Schwellenwert zu finden, welcher für die ausgewählten Features die Trainingsmenge bestmöglich teilt, werden zufällig Schwellenwerte 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 der gesamten Trainingsmenge. Dies soll den Bias reduzieren [GEW06].



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

Boosting bezeichnet das Konvertieren eines „schwachen“ PAC-Algorithmus (**Probably Approximately Correct**), d. h. ein Algorithmus der nur leicht besser als Raten ist, 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  $\frac{1}{\epsilon}, \frac{1}{\delta}$  sein. Für einen schwachen PAC-Algorithmus gilt das Gleiche mit dem Unterschied, dass  $\epsilon \geq \frac{1}{2} - \gamma$ , wobei  $\delta > 0$  [FS97]. In Abbildung 2.3 wird illustriert, wie vier schwache Lerner jeweils auf eine Teilmenge nacheinander trainiert werden, wobei die Teilmenge des jeweils nächsten von dem Fehler des vorherigen Lerners abhängt. Schlussendlich werden alle schwachen Lerner gewichtet aggregiert, wodurch ein starker Lerner entsteht. In dieser Arbeit wird im speziellen der Boosting Algorithmus **AdaBoost** von Freund und Schapire verwendet [FS97].

## 2 ENTSCHEIDUNGSBÄUME

# Gestenerkennung

Handgesten gelten als Kommunikationsmittel, besonders bei gehörlosen Menschen. Sie werden auch in der kontaktlosen Interaktion mit Geräten verwendet. Dafür ist ein Ansatz nötig, um Handgesten zu erkennen. Es wird zwischen optischen und nicht-optischen Ansätzen unterschieden. Die optischen Ansätze nutzen einen oder mehrere Kameras, um eine Folge von Bildern aufzunehmen. Dieser Ansatz ist aber empfindlich gegenüber den vorherrschenden Lichtverhältnissen. Nicht-optische Ansätze bedienen sich anderen Sensoren, z. B. Infrarot Abstandssensoren, oder nutzen technische Hilfsmittel, z. B. ein sEMG Recorder, 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 und davon bis zu 98,98% korrekt klassifiziert werden. Auch invalide Gesten (Nullgesten) konnten zuverlässig erkannt werden. Das verwendete neuronale Netz benötigte 19,8 ms zur Ausführung auf dem Arduino Uno Board [Gie20].

## 3.1 Gestenerkennung mit Entscheidungsbäumen

Song et al. [SHL<sup>+</sup>19] haben die Handgestenerkennung mit Gradient Boosting Entscheidungsbäumen (GBDT) untersucht. Gradient Boosting definiert boosting als numerisches Optimierungsproblem, indem eine Kostenfunktion minimiert wird durch die iterative Summierung von schwachen Lernern, die durch den Gradient Descent Algorithmus gefunden wird [Fri99]. 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 neun Features (siehe Tabelle 3.1). Damit erzielten sie eine Klassifizierungsgenauigkeit von 91% unter zwölf verschiedenen Handgesten.

### 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].

Ahad et al. [ATKI12] diskutieren den Motion History Image (MHI) Ansatz. MHI ist ein optischer Ansatz, der eine Sequenz von Bildern in ein einziges Bild komprimiert. Dabei werden Bewegungen, die nach anderen Bewegungen stattgefunden haben heller angezeigt. Die Funktion  $\psi(x, y, t)$  bestimmt, ob eine Bewegung in einem Pixel  $(x, y)$  zu einem Zeitpunkt  $t$  stattfindet.

Formel 3.1 beschreibt die rekursive Berechnung des MHI. Das MHI kann aber auch sequenziell berechnet werden. Initial sind alle Werte 0. Wenn  $\psi(x, y, t)$  eine Bewegung in einem Pixel  $(x, y)$  zu einem Zeitpunkt  $t$  signalisiert, dann wird der Pixel auf den Maximalwert  $\tau$  gesetzt. Mit jedem Bild, indem keine Bewegung im Pixel  $(x, y)$  durch  $\psi$  signalisiert wurde, wird der Wert um den Zerfallswert  $\delta$  dekrementiert bis zu einem Minimum von 0.

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

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 nachdem, wie  $\tau$  und  $\delta$  gewählt sind, ist die Bewegungshistorie nicht sichtbar oder der Einfluss der ersten Bilder geht verloren.

Abbildung 3.1 zeigt ein Motion History Image einer Bildsequenz. In der Bildsequenz wird eine Armbewegung ausgeführt. Unter jedem Bild ist das resultierende MHI zu dem Zeitpunkt zu sehen. Zu sehen ist, dass Pixel die sich verändert haben weiß sind und die anderen Pixel schwarz. Mit der fortlaufenden Bildsequenz wird das Bild erweitert und der Arm scheint sich zu duplizieren und seine Vergangenheit hinter sich her zu ziehen. Diese vergangene Bewegung



■ **Abbildung 3.1:** Die Übersetzung von einer Bildsequenz zu einem Motion History Image.  
Übernommen aus der Arbeit von Ahad et al. [ATKI12].

ist aber dunkler als die kürzlich ausgeführten Bewegungen. Dies wird als die Historie des MHI bezeichnet. Durch den Zerfallswert sind vergangene Bewegungen dunkler.

## 3.2 Optische Handgestenerkennung

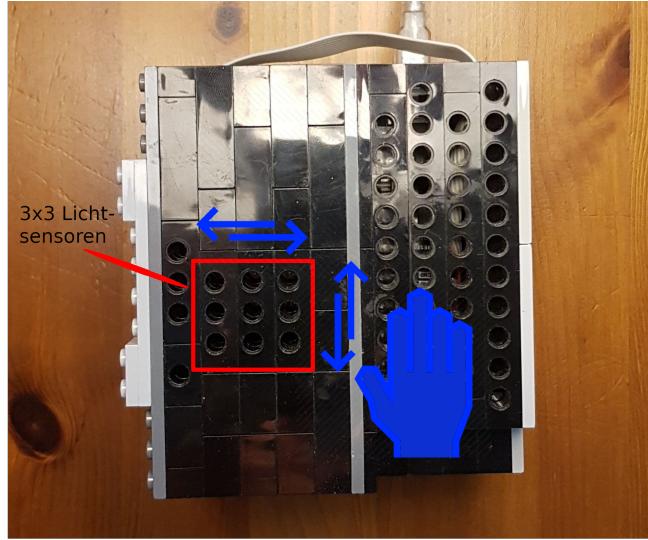
Diese Arbeit ist Teil einer Fallstudie zur Handgestenerkennung auf Low-End Mikrocontrollern von dem Institut für Telematik an der TUHH [VKK<sup>+</sup>20]. Das Ziel ist die Handgestenerkennung in Echtzeit mit so wenig Ressourcen wie möglich, damit die Produktion der einzelnen Module so kostengünstig wie möglich ist. Als Eingabe dient, je nach Modul, eine 3x3, bzw. 4x4, Matrix von Lichtsensoren. Abbildung 3.2 illustriert vier Typen von Handgesten, die durch den Mikrocontroller erkannt werden: Links nach Rechts, Rechts nach Links, Oben nach Unten, Unten nach Oben. Zudem wird zwischen Handgesten und Nullgesten, d. h. invalide Handgesten, unterschieden. In den bisherigen Arbeiten wurden ML Modelle mit künstlichen neuronalen Netzwerken erstellt. Deren Prozessablauf zur Handgestenerkennung lässt sich mit drei Schritten zusammenfassen [VKK<sup>+</sup>20].

1. Extrahiere einen Gestenkandidaten.
2. Vorverarbeite den Gestenkandidaten.
3. Wende das Modell auf den vorverarbeiteten Gestenkandidaten an.

## 3.3 Extrahieren von Gestenkandidaten

Die Lichtsensorenmatrix liefert einen kontinuierlichen Strom an Bildern. 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. Sobald der

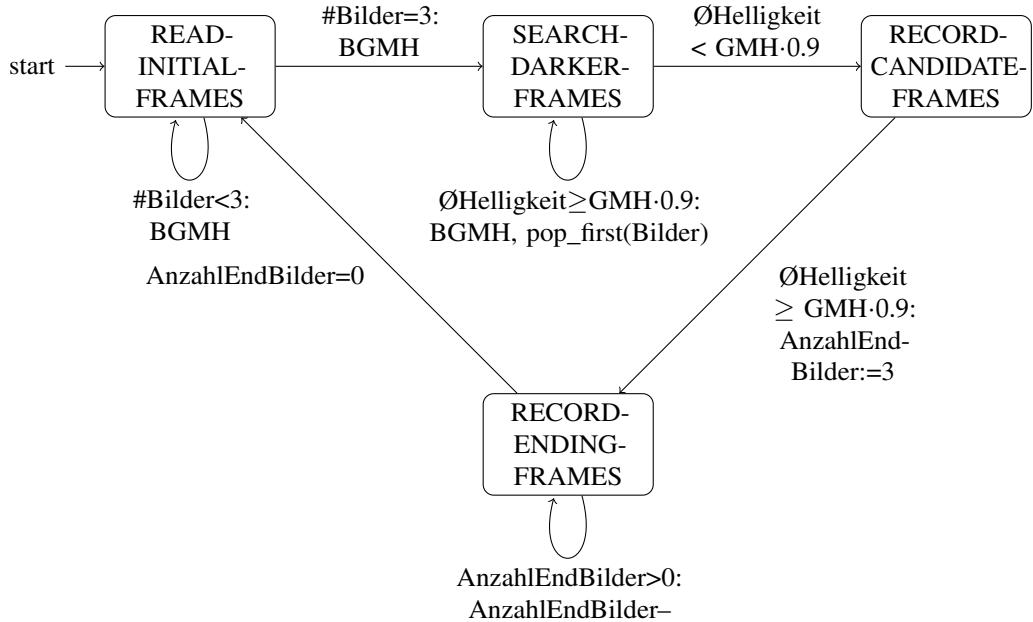
### 3 GESTENERKENNTUNG



**Abbildung 3.2:** Das Arduino-Board ATmega328P mit 3x3 Matrix von Lichtsensoren in Lego-Verpackung. Illustriert werden die möglichen Handgestentypen mit Ausnahme der Nullgeste.

gleitende Mittelwert unterschritten wird, wird angefangen einen Gestenkandidaten aufzunehmen. Sobald die Lichtverhältnisse zu dem Wert zurückkehren, wird die Aufnahme beendet. Der gleitende Mittelwert wird immer angepasst, wenn kein Gestenkandidat aufgenommen wird, um sich den verändernden Lichtverhältnissen anzupassen. Da leichte Veränderungen natürlich sind, muss eine Toleranzgrenze von 10% unter- und überschritten werden, damit die Aufnahme gestartet, bzw. beendet wird. Die Folge ist, dass der Anfang und das Ende nicht vollständig ist. Aus diesem Grund schlug Kubik zusätzlich vor am Anfang und Ende weitere Bilder anzufügen [Kub19].

Abbildung 3.3 zeigt die konkrete Implementierung dieses Algorithmus mit einem Zustandsautomaten, der von Venzke entwickelt wurde. In jedem Zustand wird das aktuelle Bild einem Puffer angefügt. Der Automat verbleibt im Zustand READ-INITIAL-FRAMES bis der Puffer drei Bilder enthält. Dabei wird stets der gleitende Mittelwert der Helligkeit angepasst. Anschließend geht der Automat in den Zustand SEARCH-DARKER-FRAMES über, indem immer das erste Bild aus dem Puffer entfernt wird, da lediglich drei Bilder jeweils vor dem Aufnehmen und nach dem Aufnehmen des Gestenkandidaten angefügt werden sollen. Außerdem wird weiterhin der gleitende Mittelwert angepasst. 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 überschritten wird, woraufhin der Automat in den Zustand RECORD-ENDING-FRAMES über geht. Dort werden die letzten



**Abbildung 3.3:** Implementierung des Kubik-Algorithmus zur Gestenerkennung von Venzke. BGMH steht für die Aktion „**Berechnung Gleitender Mittelwert der Helligkeit**“ und GMH steht für die Variable „**Gleitender Mittelwert der Helligkeit**“.

drei Bilder dem Puffer angehängt. Sobald drei Bilder angehängt wurden, wird der Puffer dem Klassifizierer übergeben und anschließend der Zustandsautomat zurückgesetzt. Daraufhin geht der Automat wieder in den initialen Zustand über.

### 3.4 Skalieren des Gistenkandidaten

Ein Gistenkandidat besteht aus einer variablen Anzahl von Bildern. Durch die künstlich angefügten Bilder am Anfang und Ende sind es mindestens acht Bilder. Kubik erkannte, dass ein neuronales Netz eine feste Anzahl an Eingaben benötigt und diskutierte verschiedene Ansätze, um mit einer variablen Länge des Input umzugehen. Er verwarf die Idee den Puffer mit irrelevanten Bildern oder Nullen auszufüllen, Bilder zu duplizieren oder Teile des Gistenkandidaten zu verwerfen. Er befürchtete, dass dadurch nicht die vollständige Handgeste auf die Eingangs-Ebene des neuronalen Netzes (NN) abgebildet werden würde, oder dass die Handgeste womöglich verzerrt wäre [Kub19].

Kubik schlug vor, dass lineare Interpolation angewendet werden soll. Dabei werden die vorhandenen Bilder uniform auf 20 Pseudoindexe verteilt, sodass das erste und letzte Bild auch jeweils den ersten und letzten Pseudoindex einnimmt. Die übrigen Indize sind gleich verteilt. Aus diesem Grund ergeben sie nicht immer natürliche Zahlen. In diesem Fall wird

### 3 GESTENERKENNUNG

das Bild durch Interpolation des vorherigen Bildes und des nächsten Bildes erzeugt. Dieser Ansatz wurde auch von Giese aufgegriffen, der sich in diesem Zusammenhang ebenfalls mit künstlichen neuronalen Netzen beschäftigt hatte [Gie20].

#### 3.5 Trainings- und Testmenge

Die Modelle werden auf Basis von aufgenommenen Daten trainiert und getestet. Die Aufnahmen beinhalten die fünf Handgestentypen, die mit unterschiedlichen Lichtverhältnissen aufgenommen wurden. Die Lichtverhältnisse sind durch verschiedene Lichtquellen und Ausführungsdistanzen der Handgeste zur Kamera entstanden. Die Handgesten wurden mit der flachen Hand oder mit dem Finger in verschiedenen Geschwindigkeiten ausgeführt. Die Datenmenge umfasst insgesamt 4792 Handgesten.

Listing 3.1 zeigt ein Beispiel einer gespeicherten Handgeste von Rechts nach Links. Abbildung 3.4 illustriert die Handgeste als Graustufenbilder. Jedes Bild wird durch einen Komma separierten Vektor von Zahlen dargestellt. Die letzte Zahl ist die Beschriftung der Handgeste.

```
...
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 Rechts nach Links.

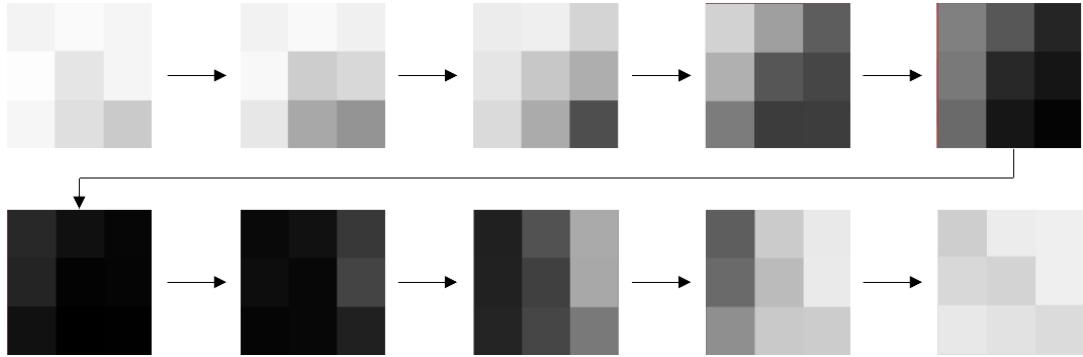
Um die Datenmenge zu vergrößern, können synthetische Daten erzeugt werden. Dabei werden aus einer aufgenommenen Handgeste Variationen durch Rotation und Rauschen generiert. Außerdem können Helligkeiten, Kontraste und Gamma verändert werden [VKK<sup>+</sup>20].

Als Testmenge wird ein Teil der Datenmenge bezeichnet, der zum Testen des trainierten Modells verwendet wird. Kubik hat Testdaten unter verschiedenen Lichtverhältnissen und Entfernungen zur Kamera aufgenommen. Klisch hat daraus eine Testmenge erstellt, die von Klisch und Giese zum Ermitteln der Klassifizierungsgenauigkeit verwendet wurden [Kli20, Gie20]. Klisch definiert die Klassifizierungsgenauigkeit gemäß 3.2. Die Klassifizierungsgenauigkeit

### 3.6 HANDGESTENERKENNTUNG MIT KÜNSTLICHE NEURONALEN NETZEN

ist das Verhältnis zwischen der Anzahl an korrekt klassifizierten Handgesten und der Anzahl der Einträge in der Testmenge.

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



**Abbildung 3.4:** Illustration der Handgeste von Rechts nach Links aus Listing 3.1.

## 3.6 Handgestenerkennung mit künstliche neuronalen Netzen

Insgesamt gingen dieser Arbeit sieben Arbeiten voraus, von denen vier für diese Arbeit relevant waren [Eng18, Kub19, Kli20, Gie20]. Diese Arbeiten haben sich im Zusammenhang mit dieser Fallstudie mit künstlichen neuronalen Netzen beschäftigt. Es wurden rekurrente neuronale Netze (RNN), Feedforward neuronale Netze (FFNN) und Long-Short-Term Memory neuronale Netze (LSTMNN) näher betrachtet, sowie die Optimierung von FFNN.

Engelhardt führte die in 3.2 definierten Handgesten mit der Hand, einem Finger und zwei Fingern unter verschiedenen Helligkeiten aus. Damit hat Engelhardt seine ML Modelle trainiert und validiert. Er argumentiert, dass RNN, FFNN und LSTMNN für temporale Probleme am besten geeignet seien. Convolutional neuronale Netze (CNN) verwarf Engelhardt aufgrund der geringen Auflösung der Handgesten und da die Faltung 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 Bilder, die zu 180 Werten konkateniert und auf Werte zwischen 0 und 1 normalisiert wurden. Als bestes ML Modell stellte sich eines seiner FFNNs heraus, das auf seinen Testdaten bis zu 99% Klassifizierungsgenauigkeit erzielte. Außerdem erwies es sich als robust gegenüber Rauschen und Helligkeitsveränderungen als RNNs. Die Ausführungszeit des FFNN belief sich auf 11,54 ms, bei einer Programmgröße von 11 kB und 573 Byte RAM [Eng18].

### 3 GESTENERKENNUNG

Kubik hat in seiner Arbeit den FFNN Ansatz von Engelhardt aufgegriffen. Er untersuchte Handgesten, die mit der Hand in verschiedenen Distanzen zur Kamera, unter guten und schlechten Lichtverhältnissen ausgeführt wurden. Neben der Facettenkamera untersuchte Kubik ebenfalls eine Lochkamera. Er stellte fest, dass diese aber wesentlich schwerer auszuleuchten ist, was sich auf die Klassifizierungsgenauigkeit auswirkte [Kub19].

Als Eingabe nutzte Kubik ebenfalls 180 Werte, die 20 Bilder repräsentieren. Um mit der variablen Länge von Handgesten umzugehen schlug Kubik vor, die Bildsequenzen auf 20 Bilder zu skalieren (Kapitel 3.4). Um die Skalierung durchzuführen, musste aber der Anfang und das Ende der Handgeste bekannt sein. Aus diesem Grund war es nötig Gestenkandidaten erkennen zu können (Kapitel 3.3). Er stellte fest, dass dies die Gesamtlänge der Handgeste limitierte in Anbetracht des RAMs des Mikrocontrollers [Kub19].

Um die Klassifizierungsgenauigkeit zu erhöhen, verwendete Kubik zusätzlich synthetische Trainingsdaten, die er aus der bestehenden Trainingsmenge durch Rotation generierte (Kapitel 3.5). Dies erhöhte die Klassifizierungsgenauigkeit erheblich. Kubik erstellte eine Testmenge (Kapitel 3.5) und evaluierte sein Modell damit. Im Allgemeinen stellte er fest, dass sich mit zunehmender Distanz zur Kamera die Klassifizierungsgenauigkeit verschlechtert. Dies erwies sich besonders als ein Problem für die Lochkamera. Bei guten Lichtverhältnissen konnte sein Ansatz mit der Facettenkamera bis 30 cm eine Klassifizierungsgenauigkeit von 97,2% erzielen. Bei schlechten Lichtverhältnissen war die Klassifizierungsgenauigkeit 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 Handgesten aufweisen. Kubiks Ansatz hat insgesamt 36 ms benötigt, um das Modell auszuführen, und 11 ms für die Skalierung [Kub19].

Klisch hat einen Ansatz von Venzke untersucht, nach dem man ein RNN als mehrere FFNNs trainieren könnte. Engelhardt stellte vorher fest, dass RNNs sonst schlechtere Klassifizierungsgenauigkeiten erzielten als FFNNs, da sie schwerer zu trainieren sind [Eng18]. Aus diesem Grund hat Klisch verschiedene Konfigurationen getestet und stellte fest, dass ein RNN bessere Ergebnisse erzielt, wenn es als ein Netzwerk trainiert wird, als wenn ein RNN vorher in mehrere FFNNs zerlegt wird. Mit seinem RNN erzielte Klisch mit einer Mischung aus guten und verhältnismäßig schlechten Lichtverhältnissen eine Klassifizierungsgenauigkeit von 71%. Das ist eine Verbesserung zu dem Ergebnis, das Engelhardt mit RNNs erzielte. Klisch stellte fest, dass sein Modell schnell genug ist, um eine Bildrate 50 Hz zu unterstützen [Kli20].

### 3.6 HANDGESTENERKENNUNG MIT KÜNSTLICHE NEURONALEN NETZEN

Der Fokus von Giese's Arbeit lag auf Kompression der Größe von FFNNs und Optimierung der Ausführungszeit. Er trainierte ein FFNN und erzielte eine Klassifizierungsgenauigkeit von 98,96%. Dies ist signifikant besser als das FFNN von Kubik, welches lediglich 83% auf der Testmenge von Klisch erzielte. Giese geht davon aus, dass sein FFNN bessere Ergebnisse erzielte, da er ca. 19-mal mehr Trainingsdaten zur Verfügung hatte als Kubik. Er untersuchte die Auswirkungen von Pruning, Quantisierung, Sparse Matrix Formaten, SeeDot und den Optimierungsparametern von GCC. Mit Pruning und erneutem Trainieren konnte Giese 72% aller Verbindungen entfernen ohne signifikanten Verlust in der Klassifizierungsgenauigkeit. Das wiederholte Ausführen von Quantisierung und erneutem Trainieren erhöhte die Klassifizierungsgenauigkeit. Die beste Ausführungszeit wurde mit dem Sparse Matrix Format CSC-MA-Bit erzielt, dass unnötige Multiplikationen verhindert. Die kleinste Programmgröße wurde mit dem Sparse Matrix Format CSC-Centroid erzielt.

SeeDot hat im Vergleich zum Ausgangsmodell sowohl Ausführungszeit als auch Programmgröße verringert. SeeDot verringerte die Klassifizierungsgenauigkeit aber signifikant. Der Vorteil von SeeDot ist die geringe Entwicklungszeit, die diese Optimierung benötigt. Der Optimierungsparameter O2 hat den besten Kompromiss zwischen Programmgröße und Ausführungszeit erzielt. Insgesamt hat die beste Lösung 35,7% weniger Programmspeicher benötigt und die Ausführungszeit wurde von 26,1 ms auf 6,8 ms reduziert. Eine Skalierung ist immer noch notwendig. Dafür gibt Giese 13 ms an. [Gie20].

### 3 GESTENERKENNUNG

# Handgestenerkennung mit Entscheidungsbäumen

Mit *Scikit-Learn* wurden verschiedene ML Modelle auf Basis von Entscheidungsbäumen trainiert, um zu untersuchen, wie gut sie sich für die Klassifizierung von Handgesten eignen. Die Modelle unterscheiden sich in der Ensemble-Methode, Hyperparametern und der Feature-Menge mit denen sie konstruiert wurden. Ein Modell ist ein Ensemble mit einer arbiträren Anzahl von Entscheidungsbäumen. Jede ausgewählte Feature-Menge erfüllt eine Reihe an Anforderungen (z. B. Invarianz gegenüber der Geschwindigkeit), die nötig sind, damit das ML Modell gut generalisieren kann.

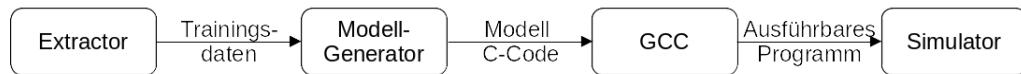
Das Modell wird innerhalb einer komplexen Infrastruktur generiert. Diese stellt Werkzeuge und Code-Bibliotheken bereit, um die Rohdaten der Handgeste zu Feature-Mengen zu verarbeiten und auf vordefinierten Testmengen zu validieren. Für diese Arbeit wurden 14410 Handgesten aufgenommen, um einerseits die verfügbaren Trainingsdaten mit Nullgesten zu ergänzen und andererseits, um Testmengen zur Validierung von Nullgesten und Lichtverhältnissen zu generieren.

## 4.1 Modell

Als Modell wird ein Klassifizierer bezeichnet, das mit ML konstruiert wurde. In diesem Fall besteht der Klassifizierer aus Entscheidungsbäumen. Jede Konfiguration eines Modells besteht aus einer Ensemble-Methode, einer Feature-Menge und Hyperparametern. Die Hyperparameter bestimmen die maximale Baumhöhe, die Waldgröße und die Blattgröße.

Jede Konfiguration wird mit dem in Abbildung 4.1 illustrierten Arbeitsablauf verarbeitet. Zunächst werden die Rohdaten verarbeitet, wobei eine beschriftete Trainingsmenge mit der

## 4 HANDESTENERKENNTUNG MIT ENTSCHEIDUNGSBÄUMEN



■ Abbildung 4.1: Arbeitsablauf, um ein Modell zu trainieren und zu validieren.

definierten Feature-Menge extrahiert wird. Dann wird das Modell trainiert und als C-Code exportiert. Der C-Code wird mit *GCC* kompiliert, wodurch der C-Code zu einem ausführbaren Programm konvertiert wird. Das Programm wird anschließend mit dem Werkzeug *Simulator* auf den definierten Trainingsmengen validiert.

### 4.1.1 Training

Das Modell wird mit *Scikit-Learn* trainiert. Die Konstruktion eines Entscheidungsbaumes ist aber nicht deterministisch, da es bei der Konstruktion mit CART Teilungen geben kann, die gleich gut sind (Kapitel 2.2). In diesem Fall wählt Scikit-Learn zufällig eine Teilung aus. Dies beeinflusst die folgenden Teilungen und somit die Klassifizierungsgenauigkeit auf der Trainings- und Testmenge. Der Zufall kann gesteuert werden, indem der Startwert des Zufallsgenerators auf einen vordefinierten Wert gesetzt wird. Mit einem konstanten Startwert ist Scikit-Learn deterministisch.

Daraus folgt, dass einige Startwerte bessere Modelle erzeugen als andere, obwohl die Konfiguration identisch ist. Folglich wurde festgestellt, dass die Konstruktion mit Scikit-Learn als Monte Carlo Methode verstanden werden kann. Durch wiederholtes Trainieren mit unterschiedlichen Startwerten, erhöht sich die Wahrscheinlichkeit, dass das beste Modell gefunden wurde. Aus diesem Grund wird das Modell mit 140 verschiedenen Startwerten für den Zufallsgenerator trainiert. Beim manuellen testen wurden im schlechtesten Fall nach 138 verschiedenen Startwerten keine Verbesserungen im Modell mehr festgestellt, weswegen sich für etwas mehr, d. h. 140, entschieden wurde.

Um die 140 Modelle der gleichen Konfiguration miteinander zu vergleichen, wurde die Trainingsmenge zufällig in zwei Mengen unterteilt. Die eine Hälfte wurde zum Trainieren des Modells verwendet und die andere Hälfte wurde zum Validieren des Modells verwendet. Aus den 140 trainierten Modellen wurde das Modell mit der höchsten Klassifizierungsgenauigkeit auf der Validationsmenge ausgewählt.

Es wurde jede Ensemble-Methode getestet, die in Kapitel 2.3 genannt wurde und mit dem Wahlklassifizier zusammengefasst. Die Waldgrößen sind zwischen 1 und 16, die maximalen Baumhöhen zwischen 1 und 22 und die Blattgrößen 1, 2, 4 oder 8.

### 4.1.2 C-Code Generierung eines Entscheidungsbaumes

Zur Ausführung eines Entscheidungsbaumes wird C-Code erzeugt. Dies ist erforderlich, weil der Entscheidungsbaum auf einem kleinen eingebetteten System ausgeführt werden soll. Die Toolchain, um die Firmware dafür zu generieren, bedarf meistens, dass der Quellcode in der Programmiersprache C implementiert ist. Dies trifft auch auf das für dieses Projekt verwendete Arduino Board ATmega328P zu.

Für jeden Entscheidungsbaum wird eine Funktion generiert. Listing 4.1 zeigt den Funktionskopf eines Entscheidungsbaumes. Als Eingabe wird ein Zeiger auf die Feature-Menge `features` übergeben und ein Zeiger auf das Rückgabe-Array `result`, dass die Wahrscheinlichkeitsverteilung des Entscheidungsbaumes als Ergebnis speichert.

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

■ **Listing 4.1:** C-Code Funktionskopf eines Baumes *i*.

Das Modell, dass von Scikit-Learn generiert wird, hat eine interne Datenstruktur, um den Entscheidungsbaum darzustellen. Aus dieser Datenstruktur wird der C-Code generiert. Listing 4.2 skizziert eine vereinfachte Darstellung dieser Datenstruktur, wobei  $T$  der Datentyp der Feature-Menge ist. Jeder Knoten ist entweder ein Blattknoten oder ein innerer Knoten. Jeder Blattknoten beinhaltet die Wahrscheinlichkeitsverteilung, mit der jede Klasse in diesem Knoten anzutreffen ist. Jeder innere Knoten verfügt über einen Schwellenwert und den Index zum Feature, dass er nutzt, um im Test den Schwellenwert mit dem Feature zu vergleichen. Außerdem enthält er seine Kindknoten. Scikit-Learn konstruiert binäre Entscheidungsbäume, weswegen es genau zwei Kindknoten gibt.

```
enum Knoten<T> {
    Blattknoten {
        klassen_wahrscheinlichkeiten: Vec<f64>
    },
    InnererKnoten {
        feature_index: usize,
        schwellenwert: T,
        knoten_links: Knoten<T>,
        knoten_rechts: Knoten<T>
    }
}
```

■ **Listing 4.2:** Vereinfachte Datenstruktur von Scikit-Learn für Entscheidungsbäume.

Listing 4.3 zeigt den C-Code, der für einen inneren Knoten generiert wird. Von der Wurzel aus, wird bis zu einem Blattknoten traversiert. Dabei wird rekursiv für jeden inneren Knoten ein `if (Test) { ... } else { ... }` Ausdruck generiert. Der *Test* ist ein Vergleich

## 4 HANDESTENERKENNUNG MIT ENTSCHEIDUNGSBÄUMEN

des ausgewählten Features und einem Schwellenwert. Aus der Feature-Menge `features` wird das Feature an der Stelle `feature_index` mit dem `schwellenwert` verglichen. Im Fall, indem der Ausdruck wahr ist wird der linke Kindknoten traversiert, ansonsten der rechte Kindknoten. Die Ausdrücke `feature_index` und `schwellenwert` werden im in Listing 4.3 durch die in der Datenstruktur eines inneren Knotens vorzufindenden Konstanten ersetzt.

```
if (features[feature_index] <= schwellenwert) {  
    Traversiere Kindknoten links...  
} else {  
    Traversiere Kindknoten rechts...  
}
```

■ **Listing 4.3:** C-Code eines inneren Knotens.

Für einen Blattknoten muss der C-Code für die Rückgabe generiert werden. Listing 4.4 zeigt, dass im Rückgabeparameter `result` für alle  $M$  Klassen die Wahrscheinlichkeit jeder Klasse gespeichert wird. Diese Rückgabe ist notwendig, da der Wahlklassifizierer, der diese Funktion ausführt, eine Wahrscheinlichkeitsverteilung erwartet. Die Ausdrücke `klassen_wahrscheinlichkeiten[i]` werden durch die Konstanten ersetzt, die in dieser Variable in der Datenstruktur eines Blattknotens gespeichert sind.

```
result[0] = klassen_wahrscheinlichkeiten[0];  
...  
result[M - 1] = klassen_wahrscheinlichkeiten[M-1];  
return;
```

■ **Listing 4.4:** C-Code eines Blattknotens.

### 4.1.3 C-Code Generierung eines Entscheidungswaldes

Ein Entscheidungswald besteht aus einem Ensemble von Entscheidungsbäumen. Bei der Ausführung eines Entscheidungswaldes wird jeder enthaltende Entscheidungsbaum ausgeführt. Die Ergebnisse jedes Entscheidungsbaumes werden mit dem Wahlklassifizierer (Kapitel 2.3) zusammengefasst. Das heißt, die Wahrscheinlichkeiten jeder Klasse werden addiert und die Klasse mit der größten Summe wird ausgewählt.

Listing 4.5 zeigt den C-Code, der zum Zusammenfassen eines Entscheidungswaldes mit einem Wahlklassifizierer generiert wird. Zunächst wird ein Array `total_res` erstellt, dass die Summe der Ergebnisse der einzelnen Entscheidungsbäume speichert. Die Anzahl der Ergebnisse pro Entscheidungsbaum, d. h. die Anzahl an Rückgabeklassen, ist  $M$ . Als Speicher für die Ergebnisse eines Entscheidungsbaumes wird `tree_res` erstellt. In Zeile 6-9 wird ein

Entscheidungsbaum  $i$  ausgeführt und sein Rückgabe wert auf `total_res` addiert. Dieser Codeblock wird für alle  $N$  Entscheidungsbäume wiederholt, die im Entscheidungswald enthalten sind. In Zeile 11-18 wird die Klasse ermittelt, die die höchste summierte Wahrscheinlichkeit hat. In Zeile 19 wird die Klasse an die aufrufende Funktion zurückgegeben.

```

01: unsigned char execute_decision_forest(float* features) {
02:     float total_res[M] = { 0.0, ..., 0.0 };
03:     float tree_res[M] = { 0.0, ..., 0.0 };
04:
05:     // Die folgenden 4 Zeilen werden für alle N Bäume wiederholt.
06:     tree_i(features, tree_res);
07:     total_res[0] += tree_res[0];
08:     ...
09:     total_res[M-1] += tree_res[M-1];
10:
11:     unsigned char max_index = 0;
12:     float max_value = 0;
13:     for (unsigned char i = 0; i < M; ++i) {
14:         if (max_value < total_res[i]) {
15:             max_value = total_res[i];
16:             max_index = i;
17:         }
18:     }
19:     return max_index;
20: }
```

**■ Listing 4.5:** C-Code des Wahlklassifizierers mit  $M$  Klassen und  $N$  Bäumen.

## 4.2 Features

Features bilden die Eingabe für einen Entscheidungsbaum. Gute Features sind integral, damit der Entscheidungsbaum gut generalisiert. Wenn die Feature-Menge keine eindeutige Trennung der Klassen in der Trainingsmenge zulässt, so ist auch keine gute Generalisierung zu erwarten.

In dieser Arbeit muss die Richtung der Handgeste klassifiziert werden. Dafür muss ein Feature eine Aussage über die Richtung machen können und zusätzlich Anforderungen erfüllen, um eine gute Generalisierung zu gewährleisten. Die Handgeste kann mit verschiedenen Geschwindigkeiten und unterschiedlichen Distanzen zur Kamera durchgeführt werden. Mit zunehmender Entfernung nimmt der Kontrast ab, da Streulicht einen größeren Einfluss hat. Für eine gute Generalisierung sollten die Features Invarianten zur Geschwindigkeit und den Lichtverhältnissen haben. Erschwerend ist, dass die Handgeste nie exakt gleich ausgeführt wird. Sie kann leichte Kreisbewegungen aufweisen oder schräg durchgeführt werden, sodass einige Fotowiderstände nicht verdeckt werden.

## 4 HANDESTENERKENNUNG MIT ENTSCHEIDUNGSBÄUMEN

### 4.2.1 Feature Verbesserungen

Einige Anforderungen, können durch spezielle Änderungen an einem Feature ergänzt werden. Dazu gehören relative Helligkeitsunterschiede, Positionsinformationen und Entwicklung über die Zeit.

Relative Helligkeitsunterschiede können durch Normalisierung über die lokale Gesamthelligkeit eliminiert werden. Dadurch ändert sich aber die Art der Aussage über die absolute Helligkeit zu einer Aussage über die relative Helligkeit. Das heißt, jeder Pixel in einem Bild wird durch die Summe der Pixel im Bild geteilt. Dies erzeugt eine Invarianz gegenüber skalierten Helligkeiten, jedoch nicht gegenüber Helligkeiten auf denen ein Offset addiert wurde.

Informationen über die Position können einerseits aus dem Argument des Features und andererseits durch partielle Anwendung inferiert werden. Beim Argument eines Features wird das Argument als Feature bereitgestellt, indem das Feature bestimmte Bedingungen erfüllt. Bei  $\arg(\max X)$  zum Beispiel, wird der Index bereitgestellt, an dem die Menge  $X$  maximal ist. Bei der partiellen Anwendung wird das Feature auf Teilmengen der Definitionsmenge angewendet und damit mehrfach zur Feature-Menge hinzugefügt, z. B. bei der Fotowiderstandsmatrix könnten Zeilen und Spalten Teilmengen sein.

Die Entwicklung über Zeit kann ebenfalls über die Duplizierung des Features dargestellt werden. Anstatt einen einzelnen Bild zu unterteilen, wird die Handgeste in Zeitfenster aufgeteilt. Jedes Zeitfenster fasst die einzelnen Bilder zu einem Bild zusammen. Für jedes Zeitfenster wird das Feature berechnet.

### 4.2.2 Featureauswahl

Untersucht wurden die Features aus Tabelle 3.1 die von Song et al. genutzt wurden, das Motion History Image und zwei selbst entwickelte Features. Die Features aus Tabelle 3.1 erfüllen ohne Änderungen nicht ausreichend Anforderungen. Die Features 3 und 4 wurden in abgewandelter Form in der Helligkeitsverteilung verwendet. Die Features 2, 5 und 7 bis 9 wurden nicht getestet, da sie zu komplexe Berechnungen für kleine eingebettete Systeme bedürfen.

Das *Mean absolute value* Feature von Song et al. ermöglicht die einzelnen Handgesten zu unterscheiden, wenn das Feature auf verschiedene Zeitfenster dupliziert wird. Zusätzlich kann die Helligkeit normalisiert werden. Um die Feature-Menge zu verringern, können Spalten und Zeilen zusammengefasst werden. In der Praxis generalisierte der Ansatz aber nicht gut. Es wird vermutet, dass die Varianz sehr groß ist, wenn die Handgeste mit verschiedenen Geschwindigkeiten ausgeführt wird.

*Average amplitude change* von Song et al. eignet sich gut, um horizontale und vertikale Bewegungen zu unterscheiden. Allerdings ist es nicht möglich symmetrische Bewegungen zu unterscheiden, da die Berechnung unabhängig von der Richtung ist. Dadurch wäre zum Beispiel eine Links nach Rechts Bewegung nicht von einer Rechts nach Links Bewegung zu unterscheiden. Aus diesem Grund wurde dieses Feature nicht weiter untersucht.

### Motion History

Das Motion History Feature komprimiert die Bewegung vieler Bilder in ein Bild, indem kürzlich stattgefundene Bewegungen heller erscheinen als länger zurückliegende Bewegungen (Kapitel 3.1). Das Feature ist invariant gegenüber unterschiedlichen Lichtverhältnissen, solange die Funktion  $\psi$  invariant ist, um Bewegungen zu detektieren. Überlappende Bewegungen können nicht dargestellt werden, da die Historie überschrieben wird. In dieser Arbeit ist das bei den validen Handgesten kein Problem, da diese keine überlappenden Bewegungen beinhalten.

Die Aussagekraft des Features ist abhängig von den Parametern  $\tau$  und  $\delta$ . Je nach dem, ist eine Bewegungshistorie nicht sichtbar, wenn  $\delta$  zu gering ist oder  $\tau$  zu groß, oder ein Teil der Bewegungshistorie ist abgeschnitten, wenn  $\delta$  zu groß ist oder  $\tau$  zu klein. Damit die vollständige Handgeste abgebildet wird, ist  $\delta$  abhängig von  $\tau$  und der Handgestenlänge, d. h.  $\delta = \frac{\tau}{\#Bilder}$ .

Eine Bewegung in einem Pixel  $q$  wird durch die Funktion 4.1 signalisiert. Die Bewegung in  $q$  findet statt, wenn eine absolute Veränderung oberhalb des Durchschnitts der absoluten Veränderung in der Helligkeit 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

Die Helligkeitsverteilung stellt die Pixel mit Extrema in der Helligkeit über Zeitfenster dar. Die Extrema der Helligkeit sind entweder der hellste oder dunkelste Pixel in einem oder mehrerer Bilder. Pixel sind heller, wenn ihre Werte größer sind und dunkler, wenn ihre Werte geringer sind. Folglich können die Pixel mit den Extrema über die Komposition der Funktionen  $\arg$  und  $\max$  bzw.  $\min$  definiert werden, d. h. für ein Bild  $Q$  ist der hellste Pixel  $q' = \arg(\max Q)$  und der dunkelste Pixel  $q' = \arg(\min Q)$ .

Eine Handgeste besteht aus einer Sequenz von Bildern. Diese wird in Zeitfenster unterteilt, so dass möglichst gleich viele Bilder in jedem Zeitfenster enthalten sind. Ist die Anzahl der Bilder

## 4 HANDESTENERKENNUNG MIT ENTSCHEIDUNGSBÄUMEN

nicht ohne Rest teilbar mit der Anzahl der Zeitfenster, so werden die überschüssigen Bilder uniform auf die Zeitfenster verteilt. Anschließend wird jedes Zeitfenster zusammengefasst. Es gibt mehrere Möglichkeiten die einzelnen Bilder in einem Zeitfenster zusammenzufassen.

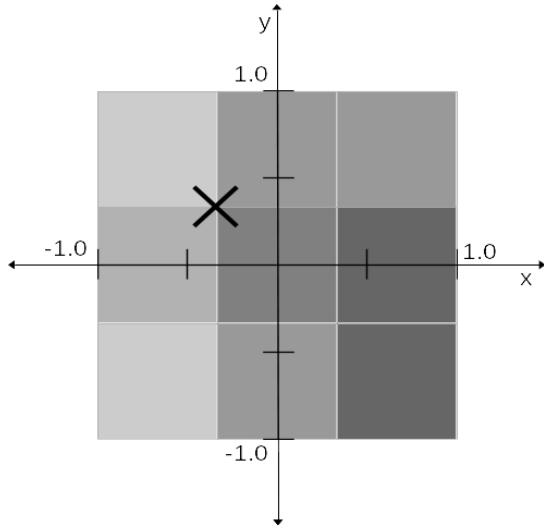
- Für jedes Zeitfenster wähle das Maximum bzw. das Minimum aus. Das heißt, jedes Zeitfenster wird von dem Pixel repräsentiert, welches den Maximalen bzw. Minimalen Wert hat.
- Projiziere das Frame auf ein kartesisches Koordinatensystem, sodass der untere linke Pixel die Koordinaten  $(0, 0)$  hat und der obere rechte Pixel  $(2, 2)$ . Wähle aus jeden Frame im Zeitfenster den Pixel mit dem maximalen bzw. minimalen Wert aus. Fasse anschließend die Koordinaten der Punkte über eine Abstandsmetrik zusammen, um den Mittelpunkt zu ermitteln. Projiziere den Mittelpunkt auf den nächsten diskreten Punkt im Koordinatensystem und gebe ihn als Wert den Durchschnitt der akkumulierten Pixel. Im weiteren Verlauf dieser Arbeit wird dies als geometrische Helligkeitsverteilung bezeichnet.
- Weise jedem Pixel zu einem oder mehrere Quadranten zu in einem Pixel. Dabei bilden benachbarte Pixel einen Quadranten mit einer maximalen Größe von vier. Für jeden Frame im Zeitfenster wird der Pixel mit dem maximalen bzw. minimalen Wert ermittelt. Wähle den Quadranten aus, der die meisten Maxima bzw. Minima beinhaltet.

Desweiteren können die Anzahl der Zeitfenster variiert werden und Pixel zu Gruppen zusammengefasst werden. Das heißt, anstatt jeden Pixel im einzelnen zu betrachten, können Gruppen von Pixeln zueinander betrachtet werden, z. B. Spalten und Zeilen des Frames. Die evaluierte Helligkeitsverteilung in dieser Arbeit nimmt keine Gruppierung vor. Es werden sechs Zeitfenster extrahiert die geometrisch zusammengefasst werden.

Es wird davon ausgegangen, dass dieses Feature invariant gegenüber unterschiedlichen Lichtverhältnissen ist, da nur relative Helligkeitsunterschiede relevant sind und Kontraste irrelevant bei der Berechnung sind.

### Schwerpunktverteilung

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



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

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

Die Handgeste wird in Zeitfenster aufgeteilt. Jedes Zeitfenster beinhaltet gleich viele Bilder. Sollte die Anzahl der Bilder nicht ohne Rest mit der Anzahl der Zeitfenster teilbar sein, werden die überschüssigen Bilder auf die Zeitfenster verteilt. Listing 4.6 zeigt die Abbildung, die in der Implementierung verwendet wird für eine nicht teilbare Anzahl von Frames für fünf Zeitfenster.

Sei  $X$  die Anzahl der Frames. Dann ist die Anzahl pro Fenster ohne Rest  $Y = \lfloor \frac{X}{5} \rfloor$ . Jedes Zeitfenster enthält für die Reste 0, 1, 2, 3, 4 folgende Anzahl an Frames:

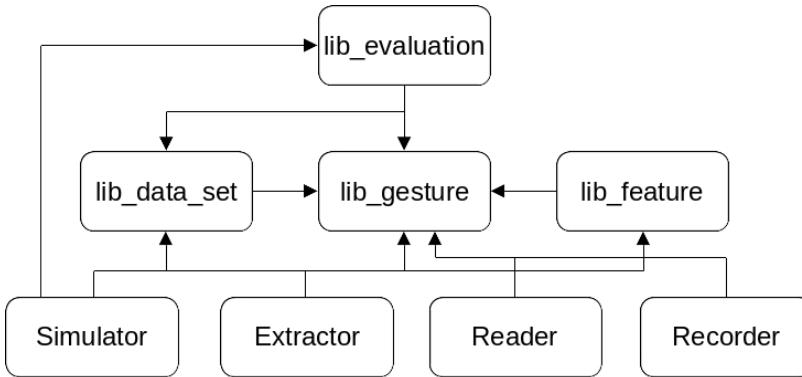
```

0 => [Y,      Y,      Y,      Y,      Y]
1 => [Y + 1,  Y,      Y,      Y,      Y]
2 => [Y + 1,  Y,      Y,      Y,  Y + 1]
3 => [Y + 1,  Y,  Y + 1,  Y,  Y + 1]
4 => [Y + 1,  Y + 1,  Y + 1,  Y,  Y + 1]

```

**Listing 4.6:** Anzahl der Frames pro Zeitfenster, je nach Rest für genau 5 Zeitfenster.

Es wurden unterschiedliche Anzahlen an Zeitfenster getestet. Entschieden wurde sich letztlich um fünf Zeitfenster, da einerseits die Berechnung des Features mit zunehmender Zeitfens-teranzahl komplexer wird (Kapitel 5.2) und andererseits zu viele Zeitfenster redundant sein können.



**■ Abbildung 4.3:** Abhängigkeiten der einzelnen Module.

Die Schwerpunktverteilung ist durch das Dividieren durch die Gesamthelligkeit  $P$  invariant gegenüber Skalierung der Helligkeit, jedoch nicht gegenüber einem Offset. Alternativ kann die Division durch  $P$  weggelassen werden, damit ausschließlich mit Ganzzahlen gerechnet wird. Dadurch können größere Bäume generiert werden, weil ganze Zahlen weniger Speicher benötigen als Fließkommazahlen (Kapitel 5.3), und die Feature-Extrahierung ist schneller (Kapitel 5.2). Die Schwerpunktverteilung mit Ganzzahlen ist durch das Weglassen der Division durch  $P$  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 konstruiert Schwerpunkte in  $[-3069, 3069]^2$ , da Sensorenwerte in  $[0, 1023]$  liegen, und der Ansatz mit Gleitkommazahlen konstruiert Schwerpunkte in  $[-1, 1]^2$ , da mit der Gesamthelligkeit  $P$  normalisiert wurde.

### 4.3 Erstellte Werkzeuge und Code-Bibliotheken

In dieser Arbeit mussten viele Features und Konfigurationen der Entscheidungsbäume untersucht und getestet werden. Aus diesem Grund wurde eine umfangreiche Infrastruktur in Rust und Python geschaffen, die die Auswertung von ML Modellen mit den Handgestendaten vereinfacht. Die Infrastruktur umfasst ein Datenmodell für Handgesten und kann die Datenmengen mit verschiedenen Parsing-Methoden einlesen. Außerdem können synthetischen Daten auf verschiedene Arten generiert werden. Abbildung 4.3 zeigt ein Abhängigkeitsdiagramm der einzelnen Module. Alle Funktionalitäten wurden in Code-Bibliotheken extrahiert, um die Integration in Hilfsprogramme zu vereinfachen.

`lib_gesture` definiert die Handgeste und die vorhandenen Handgestentypen. Außerdem implementiert sie zwei Parsing-Methoden. Die erste Methode parsed Handgesten nach Anno-

tation und die zweite nach Kubiks Algorithmus (Kapitel 3.3). Die Handgeste implementiert Methoden, um synthetische Daten zu generieren.

- Rotation um  $90^\circ$ ,  $180^\circ$  und  $270^\circ$ .
- Nullgesten durch Kombination der ersten Hälfte der Ausgangshandgeste und der zweiten Hälfte von dessen Rotationen.
- Verschiebung um einen Pixel nach oben und unten für eine Handgeste von links nach rechts bzw. rechts nach links und analog eine Verschiebung nach links und rechts für eine Handgeste von oben nach unten bzw. unten nach oben.
- Rotation der äußeren Pixel, um diagonale Handgesten zu generieren.

`lib_feature` bietet ein einfaches Interface an, um Features aus einer Handgeste zu implementieren. Listing 4.7 beschreibt das Interface in Rust. Ein Feature kann aus einer Handgeste berechnet werden und deserialisiert werden. Zurzeit sind 30 verschiedene Variationen an Features implementiert (Tabelle 4.1).

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

■ **Listing 4.7:** Das Interface, um ein Feature zu implementieren.

`lib_data_set` stellt alle Trainings- und Testmengen, die im Laufe dieser Fallstudie aufgenommen wurden, als statische Importe bereit. Einträge sind bereits nach Distanz zur Kamera, Helligkeit, Verdeckungsobjekt (Hand und Finger) und Ausführungsgeschwindigkeit sortiert. Die Helligkeit eines Eintrags kann durch einen statischen Offset verändert werden, sodass die Helligkeit jedes Pixels um den Offset erhöht oder verringert wird, oder durch eine Skalierung verändert werden.

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

Der Simulator ist zweigeteilt. Ein Teil nutzt die Gistenkandidatenerkennungsmethode nach Kubik, die in `lib_gesture` implementiert ist, um den seriellen Datenstrom von der Kamera in Echtzeit zu verarbeiten. Wenn ein Gistenkandidat gefunden wird, wird er durch das hinterlegte Modell klassifiziert. Das Ergebnis wird auf der Konsole ausgegeben. Der andere Teil evaluiert Testmengen aus der Bibliothek `lib_data_set` mit dem hinterlegten Modell und gibt Statistiken zu der Klassifizierungsgenaugkeit auf der Konsole aus.

## 4 HANDESTENERKENNUNG MIT ENTSCHEIDUNGSBÄUMEN

Feature	Variation
Helligkeitsverteilung	Minimum/Maximum über alle Zeitfenster
Helligkeitsverteilung	Geometrisches Minimum/Maximum
Helligkeitsverteilung	Quadranten mit Minima/Maxima
Helligkeitsverteilung	Minima/Maxima Zeilenweise über 3 und 6 Zeitfenster
Helligkeitsverteilung	Minima/Maxima Spaltenweise über 3 und 6 Zeitfenster
Schwerpunktverteilung	Mit Ganzzahlen in horizontaler und vertikaler Richtung
Schwerpunktverteilung	Mit Fließkommazahlen in horizontaler und vertikaler Richtung
Motion History Image	8-Bit und 16-Bit
Standardabweichung	-
Durchschnitt	-
Maximum/Minimum	-
Summe der Gradienten	Von Frame zu Frame
Summe der Gradienten	Von benachbarten Pixel in Spaltenweise und Zeilenweise
Summe der Gradienten	Spaltenweise und Zeilenweise Summe der Gradienten zusammengefasst pro Zeitfenster
Durchschnittliche Änderung der Amplitude	-

■ **Tabelle 4.1:** Implementierte Features in lib\_feature.

Der Extractor extrahiert aus spezifizierten Datenmengen alle definierten Features und exportiert diese in Dateien. Die können bei der Konstruktion eines Modells eingelesen werden und zum Trainieren genutzt werden. Optional kann die Datenmenge durch synthetische Daten erweitert werden.

Der Reader gibt den seriellen Datenstrom von der Kamera auf der Konsole aus. Dies kann zum Fehler finden und Testen der programmierten Firmware genutzt werden.

Der Recorder nutzt, wie der Simulator, den seriellen Datenstrom der Kamera und die Parsing-Methode von Kubik, um Gestenkandidaten zu erkennen. Der Gestenkandidat wird dann in eine vordefinierte Datei geschrieben. Es gibt drei Aufnahmemechanismen, um effizient annotierte Trainings- und Testmengen aufzunehmen.

#### 4.4 AUFGENOMMENE TRAININGS- UND TESTMENGEN

1. Es wird immer zwischen dem ausgewählten Handgestentypen und seinem inversen Handgestentypen hin und her gewechselt. Dieser Ansatz wurde von Kubik vorgeschlagen [VKK<sup>+</sup>20].
2. Es kann ein fixer Handgestentyp ausgewählt werden, mit dem alle Gestenkandidaten beschriftet werden.
3. Jedes Mal, wenn ein Gestenkandidat erkannt wurde, wird erfragt welcher Handgestentyp es ist.

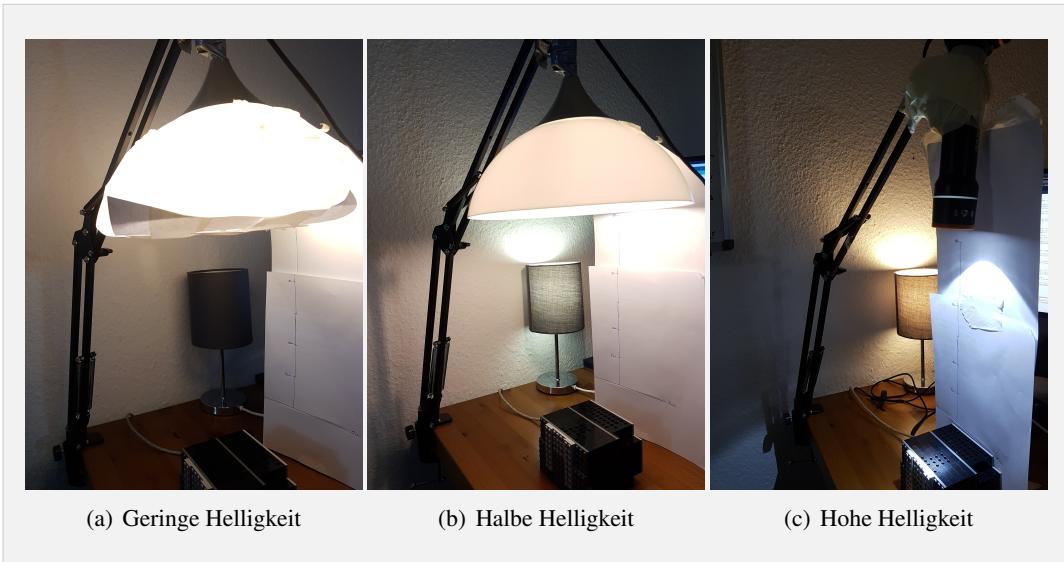
### 4.4 Aufgenommene Trainings- und Testmengen

Für diese Arbeit wurden zusätzliche Handgesten mit dem Recorder (Kapitel 4.3) aufgenommen und annotiert. Es wurden einerseits Handgesten aufgenommen, um die Modelle unter bestehenden Lichtverhältnissen 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. Insgesamt wurden 14410 Handgesten in unterschiedlichen Konfigurationen aufgenommen und annotiert.

Jede Handgeste wurde unter jeder Permutation ca. 100-mal aufgenommen bei 90 Bildern pro Sekunde. Insgesamt wurden in drei Lichtverhältnissen und vier Distanzen, sechs verschiedene Handgesten (links nach rechts, rechts nach links, oben nach unten, unten nach oben und zwei Nullgesten) jeweils schnell und langsam aufgenommen. Die Handgesten wurden in den Abständen 5 cm, 10 cm, 20 cm und 25 cm aufgenommen.

Die „Geringe“ Helligkeit hat im Durchschnitt bei ca. den Wert 140 auf dem Sensor gehabt, ohne Verdeckt zu sein. „Halbe“ Helligkeit bei ca. 659, „Hohe“ Helligkeit bei ca. 908. Alle Helligkeiten haben die 3x3-Fotowiderstandmatrix relativ gleichmäßig ausgeleuchtet. Bei den Lichtquellen 4.4(a) und 4.4(b) wurde eine Schirmlampe verwendet. Dadurch wurde das Licht 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 zum Trainieren verwendet, damit kein Bias erzeugt wird.

#### 4 HANDEKSTENERKENNUNG MIT ENTSCHEIDUNGSBÄUMEN



**Abbildung 4.4:** Verschiedene Helligkeitsstufen unter denen die Handgesten aufgenommen wurden.

Es wurden zwei Typen von Nullgesten aufgenommen. Die erste Nullgeste soll eine valide Handgeste ca. zur Hälfte ausführen und dann umkehren. Die aufgenommen Nullgesten beginnen *oben* und führen 20% bis 70% der Handgeste aus, bevor sie wieder *oben* enden. Die zweite Nullgeste soll eine Eckbewegung bzw. Kreisbewegung darstellen, indem bei einer Richtung eingedrungen wird und bei einer benachbarten Richtung ausgetreten wird. Die aufgenommen Nullgesten beginnen *oben* und führen die valide Handgeste zu 20% bis 70% aus, bevor anschließend *rechts* geendet wird. 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.

#### 4.4 AUFGENOMMENE TRAININGS- UND TESTMENGEN

Aus der Gestentestmenge wurden Testmengen erstellt, die testen wie gut ein Modell sich gegenüber unterschiedlichen Lichtverhältnissen generalisiert hat. Die Testmenge „Helligkeitstestmenge mit Offset und Skaliert“ nutzt den Anteil der Gestentestmenge mit der Helligkeit „Gering“ und ergänzt jeweils 16 Duplikate der Datenmenge mit einem Offset in der Helligkeit zwischen 0 und 800 und mit einer Skalierung zwischen 1 und 7. Die Testmenge „Helligkeitstestmenge mit konstantem Kontrast“ nutzt den Anteil der Gestentestmenge mit der Helligkeit „Halb“ und ergänzt 19 Duplikate. Diese Duplikate werden in 0,05 Schritten von 1 bis 0,05 skaliert und die Gesamthelligkeit die dadurch reduziert wird, wird auf alle Pixel in gleichen Teilen addiert. Dadurch ändert sich die Gesamthelligkeit nicht, aber der Kontrast zwischen hellen und dunklen Pixeln wird gesenkt.

## 4 HANDESTENERKENNUNG MIT ENTSCHEIDUNGSBÄUMEN

# 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 Klassifizierungsgenauigkeit erzielen, sondern auch lauffähig auf einem kleinen eingebetteten System sein, d. h. nicht mehr als den verfügbaren RAM und Programmspeicher nutzen und in einer akzeptablen Zeit terminieren.

Dieses Kapitel untersucht zuerst die Klassifizierungsgenauigkeit der besten Konfigurationen, die gefunden wurden, je Feature-Menge. Die beste Konfiguration wird anschließend auf Ausführungszeit und Ressourcenverbrauch auf dem ATmega328P hin analysiert. Dabei wird auf mögliche Optimierungen eingegangen, um die Ausführungszeit und den Ressourcenverbrauch zu senken.

## 5.1 Klassifizierungsgenauigkeit

Es werden drei Features näher betrachtet: Motion History, Helligkeitsverteilung und Schwerpunktverteilung. Daraus wurden vier Feature-Mengen 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 die gleiche 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 Feature-Menge werden jeweils drei 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 eine

## 5 EVALUATION

Restriktion 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 des Modells betrachtet, nachdem alle Optimierungen aus Kapitel 5.3 angewendet wurden.

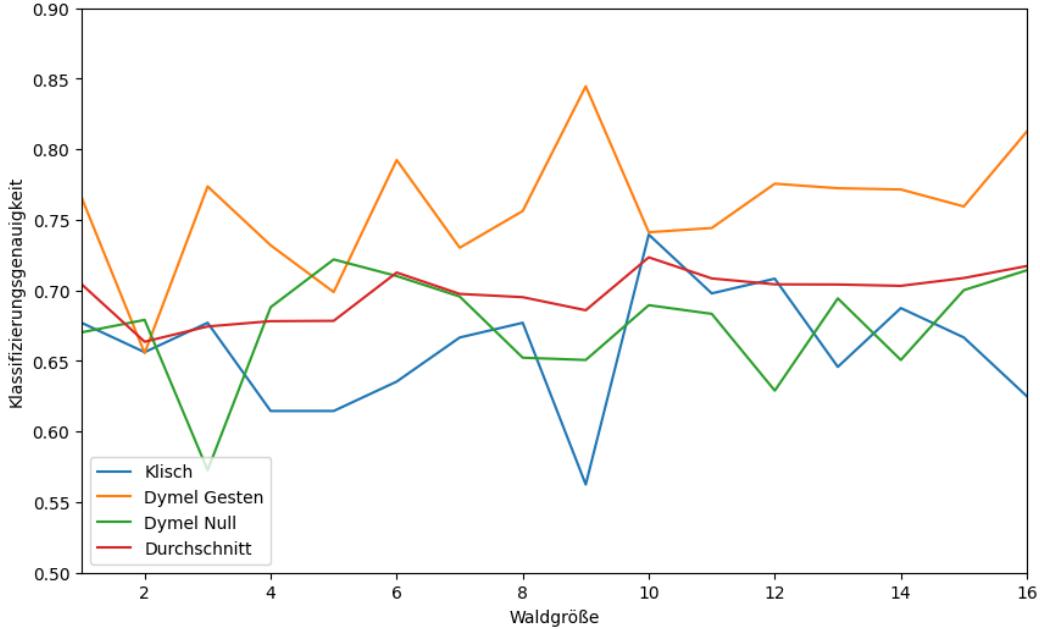
In der Analyse werden die verschiedenen Feature-Mengen im Hinblick auf die Klassifizierungsgenauigkeit der Testmenge von Klisch, der Gestentestmenge, der Nullgestentestmenge und den synthetischen Helligkeitstestmengen untereinander verglichen und mit den Ergebnissen für KNNs von Giese verglichen. Es wird ausschließlich mit Giese verglichen, da seine Ergebnisse für KNNs die beste Klassifizierungsgenauigkeit, geringste Ausführungszeit und geringsten Ressourcenverbrauch erzielte. Außerdem wird die Auswirkung von verschiedenen Waldgrößen auf die Klassifizierungsgenauigkeit untersucht. Anzumerken ist, dass lediglich die Testmenge von Klisch vergleichbar mit den Ergebnissen von Giese ist, da die Gestentestmenge, Nullgestentestmenge und Helligkeitstestmengen Teil dieser Arbeit sind. Der vollständige Datensatz der Evaluierung ist als Datei auf dem USB-Stick hinterlegt.

### 5.1.1 Helligkeitsverteilung

Konfiguration	Beste	Unter 44 kB	Unter 28 kB
Ensemble-Methode	ExtraTrees	ExtraTrees	ExtraTrees
Maximalhöhe	14	10	15
Waldgröße	10	6	1
Blattgröße (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 Feature-Menge Helligkeitsverteilung.

Die Feature-Menge der Helligkeitsverteilung beinhaltet insgesamt zwölf Features. Jeweils sechs Features repräsentieren Zeitfenster der minimalen Helligkeit und der maximalen Helligkeit. Die Zeitfenster wurden geometrisch zusammengefasst.



■ Abbildung 5.1: Die besten Modelle pro Waldgröße der Feature-Menge Helligkeitsverteilung.

Aus der Tabelle 5.1 sind die besten Konfigurationen jeder Kategorie zu entnehmen. Die beste Konfiguration wurde mit der Ensemble-Methode *ExtraTrees* gefunden. Das Modell erzielt eine Klassifizierungsgenauigkeit 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 ein geringer Zuwachs der Gesamtklassifizierungsgenauigkeit mit der zunehmenden Waldgröße zu beobachten.

### 5.1.2 Motion History

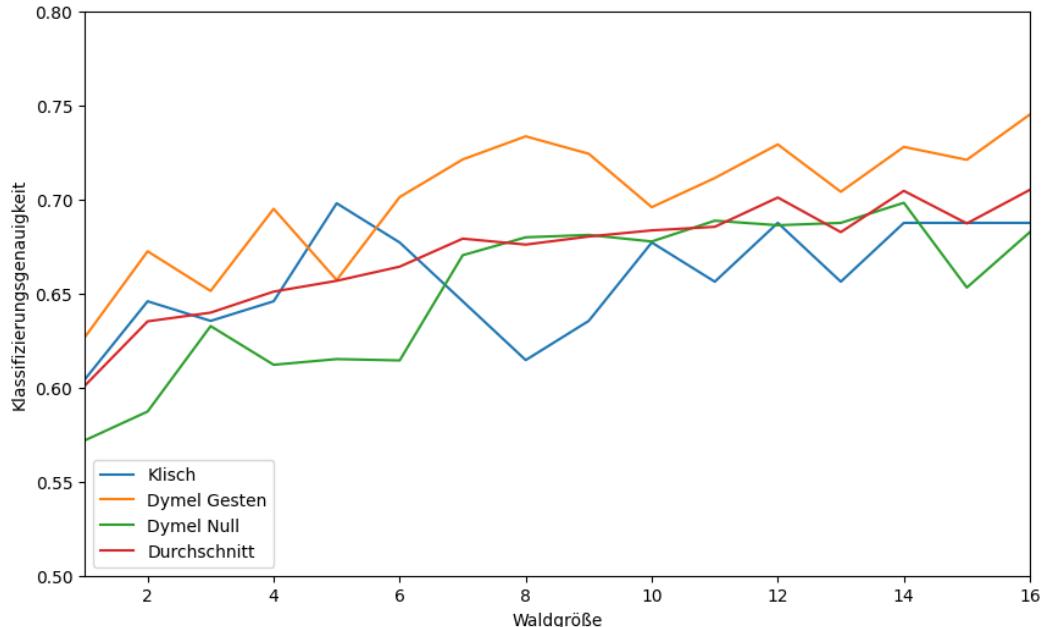
Die Feature-Menge Motion History beinhaltet für jeden Pixel ein Feature, das der Definition des Motion History Image folgt (Formel 3.1), wobei  $\tau = 100$  und  $\delta = \frac{\tau}{\#Bilder}$  ist.

Aus Tabelle 5.1 sind die besten Konfigurationen jeder Kategorie zu entnehmen. Die beste Konfiguration wurde wieder mit der Ensemble-Methode *ExtraTrees* gefunden. Das Modell erzielt eine Klassifizierungsgenauigkeit von 68,8% auf der Testmenge von Klisch, 74% auf

## 5 EVALUATION

Konfiguration	Beste	Unter 44 kB	Unter 28 kB
Ensemble-Methode	ExtraTrees	ExtraTrees	ExtraTrees
Maximalhöhe	10	11	9
Waldgröße	16	7	6
Blattgröße (min_samples_leaf)	1	2	1
Programmgröße in Bytes	84200	40456	22804
Genauigkeit Testmenge von Klisch	68,8%	67,7%	62,5%
Genauigkeit Gestentestmenge	74,5%	67,6%	68,5%
Genauigkeit Nullgestentestmenge	68,3%	64,4%	67,6%

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



■ **Abbildung 5.2:** Die besten Modelle pro Waldgröße der Feature-Menge Motion History.

der Gestentestmenge und 69% auf der Nullgestentestmenge. Im Vergleich zu der Helligkeitsverteilung wird mehr Programmspeicher benötigt und die Gesamtklassifizierungsgenauigkeit ist 1,84% schlechter.

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 die geringste 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

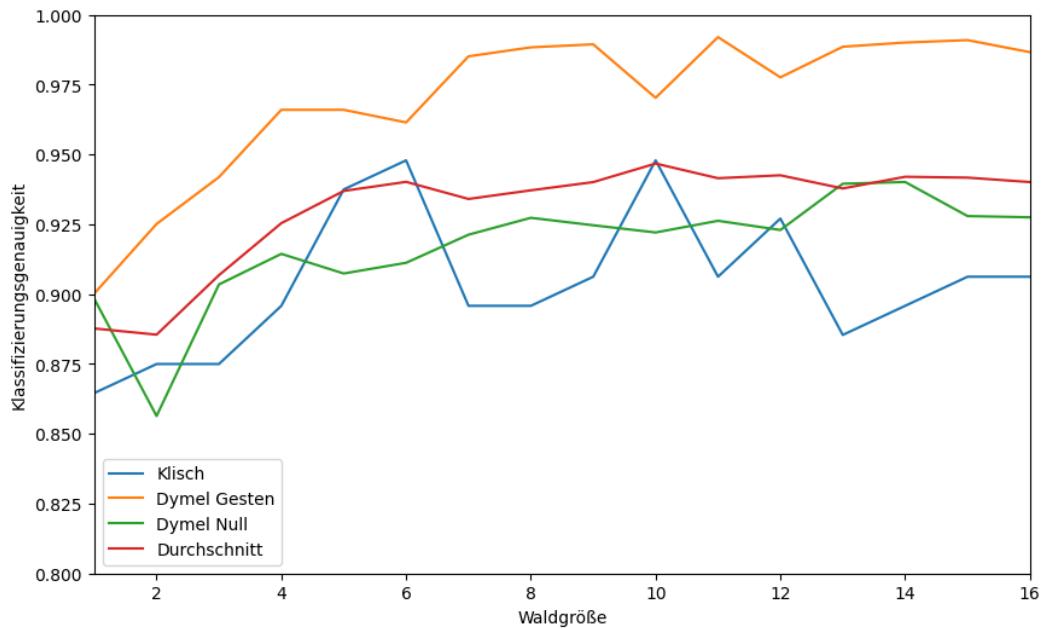
Konfiguration	Beste	Unter 44 kB	Unter 28 kB	Unter 14 kB
Ensemble-Methode	Boosting	Random Forest	Random Forest	Bagging
Maximalhöhe	20	12	10	7
Waldgröße	10	7	4	3
Blattgröße (min_samples_leaf)	8	1	2	8
Programmgröße in Bytes	83304	43668	20188	6656
Genauigkeit Testmenge von Klisch	94,8%	89,6%	89,6%	87,5%
Genauigkeit Gestentestmenge	97,0%	96,5%	95,6%	94,1%
Genauigkeit Nullgestentestmenge	92,2%	92,4%	88,8%	89,9%

**Tabelle 5.3:** Die besten Konfigurationen der Feature-Menge Schwerpunktverteilung mit Gleitkommazahlen.

Die Feature-Menge Schwerpunktverteilung mit Gleitkommazahlen folgt der Definition aus Kapitel 4.2.2 und beinhaltet insgesamt zehn Einträge. Jeweils zwei Einträge bilden die X- und Y-Koordinate des Schwerpunktes. Damit spiegeln zehn Einträge insgesamt fünf 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 Klassifizierungsgenauigkeit 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.

## 5 EVALUATION



■ **Abbildung 5.3:** Die besten Modelle pro Waldgröße der Feature-Menge Schwerpunktverteilung mit Gleitkommazahlen.

Im Vergleich zu der Helligkeitsverteilung und Motion History ist die Klassifizierungsgenauigkeit dieses Ansatzes signifikant besser, sogar wenn nur 6656 Byte Programmspeicher verwendet werden. Wird die Kategorie *Beste* mit der Kategorie *Unter 14 kB* verglichen, nimmt 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 dies 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 1,8% nicht groß. Damit eignet sich diese Feature-Menge gut für kleine eingebettete Systeme, da nur wenige Bäume benötigt werden.

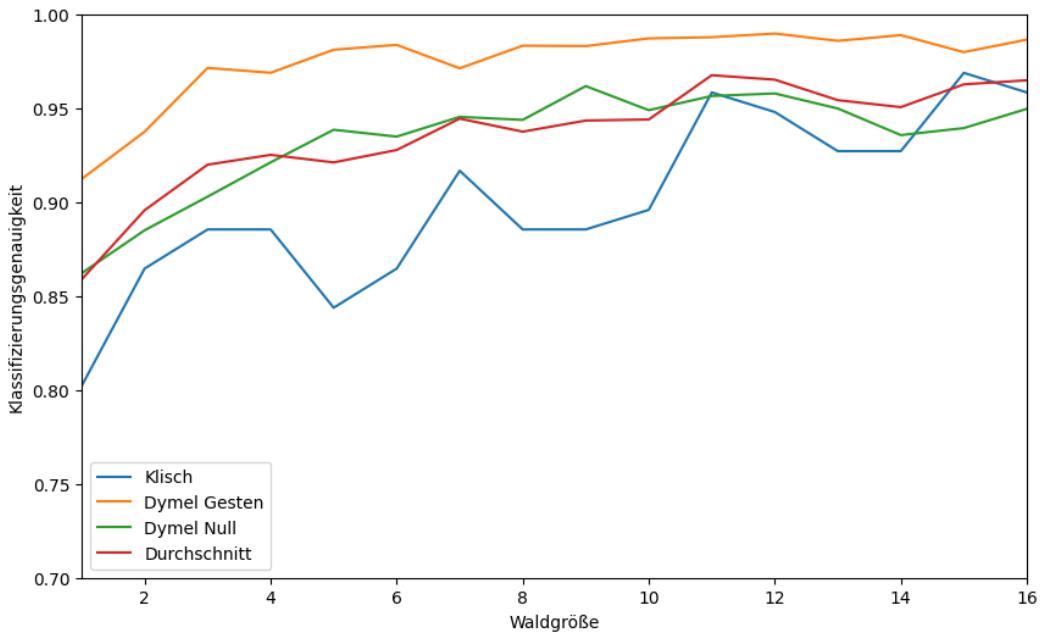
### 5.1.4 Schwerpunktverteilung mit Ganzzahlen

Die Feature-Menge Schwerpunktverteilung mit Ganzzahlen folgt der Definition aus Kapitel 4.2.2 und beinhaltet insgesamt zehn Einträge. Jeweils zwei Einträge bilden die X- und Y-Koordinate des Schwerpunktes. Damit spiegeln zehn Einträge insgesamt fünf Zeitfenster wieder.

## 5.1 Klassifizierungsgenauigkeit

Konfiguration	Beste	Unter 44 kB & 28 kB	Unter 14 kB
Ensemble-Methode	ExtraTrees	Random Forest	Random Forest
Maximalhöhe	21	13	12
Waldgröße	11	7	3
Blattgröße (min_samples_leaf)	2	4	1
Programmgröße in Bytes	76200	21532	11012
Genaugkeit Testmenge von Klisch	95,8%	91,7%	86,5%
Genaugigkeit Gestentestmenge	98,8%	97,1%	95,5%
Genaugigkeit Nullgestentestmenge	95,6%	94,5%	88,9%

■ **Tabelle 5.4:** Die besten Konfigurationen der Feature-Menge Schwerpunktverteilung mit Ganzzahlen.



■ **Abbildung 5.4:** Die besten Modelle pro Waldgröße der Feature-Menge Schwerpunktverteilung mit Ganzzahlen.

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, wo das Modell 96,9% der Testmenge von Klisch korrekt klassifizierte und damit nur

## 5 EVALUATION

2,1% schlechter ist. Diese maximiert aber in keiner Kategorie die Gesamtklassifizierungsgenauigkeit. Außerdem werden 98,8% der Gestentestmenge und 95,6% der Nullgestentestmenge korrekt klassifiziert. Es wurde kein Entscheidungswald gefunden, der weniger als 44 kB Programmspeicher benötigt und besser ist als die Konfiguration in der Kategorie *Unter 28 kB*.

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, 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 die 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.

Zu erwarten war, dass der Ansatz mit Gleitkommazahlen bessere Ergebnisse als der Ansatz mit Ganzzahlen erzielt. Da im Training ohne Nullgestentestmenge eine besseres Modell gefunden wurde, ist davon auszugehen, dass das äquivalente Modell im Training mit der Nullgestentestmenge nicht gefunden wurde, durch den größeren Suchraum.

### 5.1.5 Kombinierte 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 Klassifizierungs-

Konfiguration	Beste	Unter 44 kB	Unter 28 kB
Schwerpunktverteilung Gleitkommazahl	Beste	Unter 14 kB	Unter 14 kB
Schwerpunktverteilung Ganzzahlen	Beste	Unter 28 kB	Unter 14 kB
Programmgröße in Bytes	-	33276	20252
Genauigkeit Testmenge von Klisch	94,8%	87,5%	87,5%
Genauigkeit Gestentestmenge	99,0%	97,7%	96,9%
Genauigkeit Nullgestentestmenge	95,8%	92,9%	92,5%

■ **Tabelle 5.5:** Die besten Konfigurationen der kombinierten Schwerpunktverteilung.

genauigkeit 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.

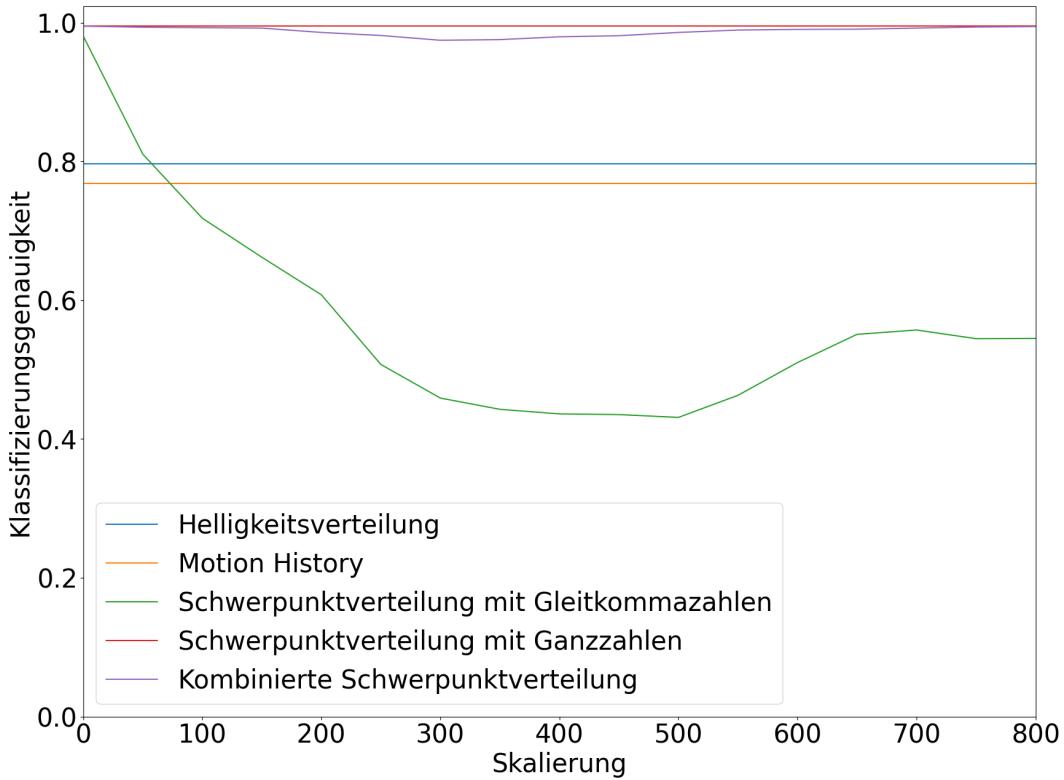
Der Nachteil dieses Ansatzes ist, dass sowohl die Feature-Menge mit Gleitkommazahlen, als auch für Ganzzahlen benötigt wird. Zum einen müssen immer beide Features berechnet werden und zum anderen kann jeder Klassifizierer nur halb so viel Speicher nutzen. Dadurch ist die Klassifizierungsgenauigkeit jedes Klassifizierers potenziell 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 potenziell robuster gegenüber unterschiedlicher Lichtverhältnisse ist.

### 5.1.6 Robustheit gegenüber Lichtverhältnissen

Dieses Kapitel validiert die Invarianzen der einzelnen Features, 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 Klassifizierer auswirkt. Dabei erhöht sich die Gesamthelligkeit, aber der Kontrast bleibt gleich. Die Helligkeitsverteilung, Motion History und Schwerpunktverteilung mit Ganzzahlen bleiben

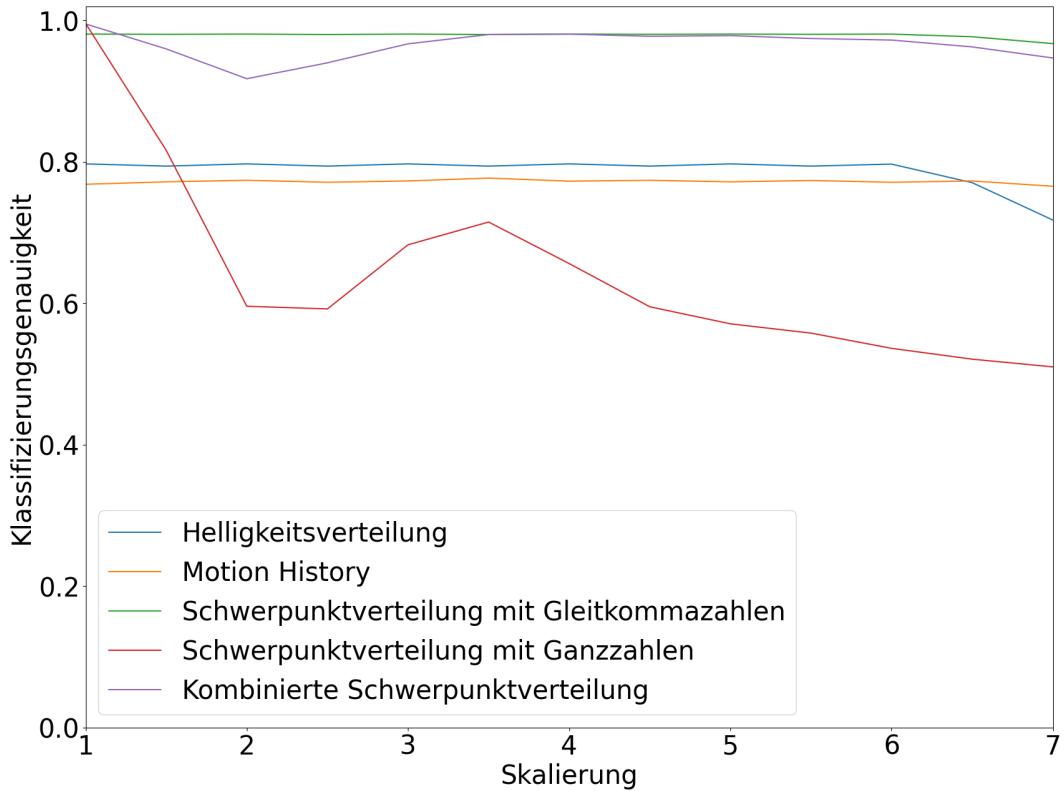
## 5 EVALUATION



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

konstant. Daher wird geschlossen, dass diese invariant gegenüber einem 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 zunehmendem Offset erfährt. Zwischen 300 und 500 ist der Einbruch am größten. Die Schwerpunktverteilung mit Gleitkommazahlen ist sehr anfällig gegenüber einem Offset.

Abbildung 5.6 zeigt, wie sich ein zunehmender Skalierungsfaktor auf die Klassifizierer 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, auf der die Helligkeitstestmenge 1 basiert, enthält aber auch Einträge mit 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 in der Klassifizierungsgenauigkeit, ähnlich wie die Schwerpunktverteilung mit Gleitkommazahlen im Vergleich zum Offset.

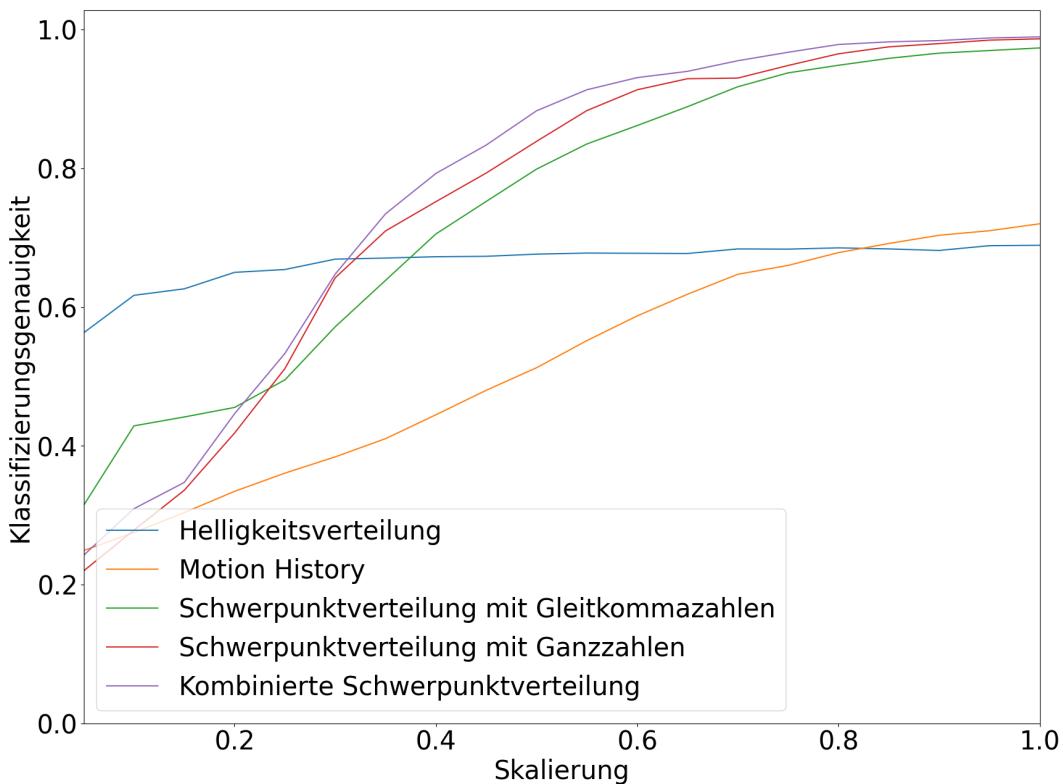


■ Abbildung 5.6: Ergebnisse der Skalierung der Helligkeitstestmenge 1 je Ansatz.

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 zeigt, wie sich ein abnehmender Kontrast bei gleichbleibender Helligkeit auf die Klassifizierer auswirkt. Diese Situation ist vergleichbar mit einer zunehmenden Distanz zur Kamera, da dort der Kontrast durch Streulicht ebenfalls geringer wird. Keiner der Feature-Mengen 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 nur noch 18% des ursprünglichen Kontrasts. Dementsprechend ist die Helligkeitsverteilung extrem robust gegenüber schlechten Lichtverhältnissen. Im Vergleich sind die anderen Klassifizierer sehr sensibel gegenüber der abnehmenden Skalierung bei gleichbleibender Gesamthelligkeit.

## 5 EVALUATION



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

Die Schwerpunktverteilungen verhalten sich vom Trend her gleich, aber die kombinierte Schwerpunktverteilung erzielt wie erwartet, fast durchweg, die beste Klassifizierungsgenauigkeit. Da keiner der Schwerpunktverteilungen invariant gegenüber einem Offset und der Skalierung 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 bei der 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 geringere Klassifizierungsgenauigkeit als die Schwerpunktverteilungen.

## 5.2 Ausführungszeit

Die Ausführungszeit der Feature-Extrahierung und Klassifizierung ist ausschlaggebend für die mögliche Bildrate. Diese ermöglicht die Wahrnehmung von schnellen Handgesten. Ist die Bildrate 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. Es verfügt über eine 8-Bit CPU, 2 kB RAM, 32 kB Flash-Speicher und verwendet eine Taktfrequenz von 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. Dies ist die Zeit die benötigt wird, um die Werte der Kamera auf dem in dieser Arbeit verwendeten Boards auszulesen.

Die Auswertung bezieht sich auf die Instruktionen des Programms, die bei der Übersetzung der Firmware durch den AVR GCC mit der Optimierungsstufe 0s 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. Bei 16 MHz bedarf ein Zyklus 0,0625  $\mu$ s.

Im Folgenden wird nur die Schwerpunktverteilung diskutiert, da die 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 Compiler Gleitkommazahlunterstützung durch Software realisieren. Dies hat zu folge, dass Operationen auf Gleitkommazahlen sehr viele Zyklen benötigen im Vergleich zu Operationen auf Ganzzahlen. Operationen sind zum Beispiel Addition, Dividieren, Vergleiche oder Typkonvertierungen.

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

## 5 EVALUATION

wird ein Zahl daher in 4 hintereinander liegende Register aufgeteilt. Das hat zur Folge, dass jede Operation mit Gleitkommazahlen viermal so viele Instruktionen wie ein 8-Bit Datentyp benötigt um die Float Operatoren in Register zu laden oder die Float-Zahl zu speichern.

Für jede Operation sind Algorithmen in Form von Funktionen hinterlegt. Diese werden vom Compiler automatisch zugelinkt. Tabelle 5.6 zeigt die im schlechtesten Fall gemessene Ausführungszeit der Funktionen, die bei der Extrahierung der Features und Ausführung des Klassifizierers verwendet werden. Im Folgenden wird diese Zeit als der WCET der Funktionen angenommen. Zum Vergleich, die Addition von 8-Bit Integer benötigt nur ein Zyklus. Die Addition von Gleitkommazahlen mit `_addsf3` benötigt dagegen im schlimmsten Fall 192 Zyklen. Zudem kommt noch ein Overhead von bis zu vier 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.

Operation	Funktion	WCET	WCET in Zyklen
<code>_lesf2</code>	Kleiner oder gleich Vergleich	$4 \mu s$	64
<code>_floatsisf</code>	Konvertierung von 32-Bit Integer nach Float	$4 \mu s$	64
<code>_divsf3</code>	Division	$36 \mu s$	576
<code>_addsf3</code>	Addition	$12 \mu s$	192

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

### 5.2.2 Feature-Extrahierung

Die Feature-Extrahierung implementiert die Berechnung der fünf Zeitfenster für die Schwerpunktverteilung. Einerseits muss aus jedem Bild der Schwerpunkt berechnet werden und andererseits müssen die Schwerpunkte auf fünf Schwerpunkte zusammengefasst werden, die die fünf 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 pro Bild berechnet werden 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` sechs mal aufgerufen, `_lesf2` und `_divsf3` jeweils zweimal aufgerufen. Die 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 für die WCET nur  $8,875 \mu s$  benötigt. Für den kombinierten Ansatz werden zusätzlich vier Speicherinstruktionen benötigt, die einen Overhead von  $0,25 \mu s$  erzeugen, d. h. es werden für die 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 einmal für alle Richtungen und Schwerpunktverteilungen durchgeführt.

Die innere Schleife wird im schlimmsten Fall für die Gesamtgröße des Schwerpunktbuffers 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 fünf mal `_floatsisf` und

## 5 EVALUATION

`_divsf3` auf und im Ansatz mit Ganzzahlen fünf mal `_divmodhi4`. Damit beläuft sich der WCET bei fünf 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 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  $4518,875 \mu s \approx 4,5 \text{ ms}$ .

### 5.2.3 Ausführung eines Entscheidungsbaumes

Der WCEP eines Entscheidungsbaumes 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 Entscheidungsbaumes.

Jeder Vergleich besteht aus drei Teilen. Der erste Teil ist die Vergleichsoperation. Der zweite Teil das Laden der Operatoren, d. h. des Features und des Schwellenwertes. 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 44 Zyklen ( $2,75 \mu s$ ). Damit beläuft sich die WCET auf  $2,75 \mu s + \text{maximale Baumhöhe} \cdot 5,1875 \mu s$  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 Entscheidungsbaumes zusammen, der im Wald enthalten ist.

Der in Listing 4.5 gezeigte Code implementiert den Wahlvorgang. Die Komplexität ist abhängig von der Anzahl der Features  $N$  und der Anzahl der Entscheidungsbäume  $K$ . In dieser Analyse wird für die Anzahl der Features  $N = 5$  angenommen. Jede Stimme eines Entscheidungsbaumes bedarf 18 Zyklen ( $1,125 \mu s$ ), um die Funktion, die den Entscheidungsbaum ausführt, aufzurufen und das Ergebnis des ausgeführten Entscheidungsbaumes 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 vier Bäumen. Die WCET dieses Entscheidungswaldes beläuft sich damit auf  $4225,5625 \mu s \approx 4,2 \text{ ms}$ . Das ist 15,2 ms, bzw. 78,4% schneller als das schnellste neuronale Netz von Giese mit Skalierung [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 13 und eine Waldgröße von sieben Bäumen. Die WCET dieses Entscheidungswaldes beläuft sich damit auf  $651,6875 \mu s \approx 0,7 \text{ ms}$ . Das ist 18,7 ms, bzw. 96,4% schneller als das schnellste neuronale Netz von Giese mit Skalierung. Es ist anzumerken, dass in der Praxis die Puffergröße bei diesem Ansatz größer sein kann, da nur vier Byte 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 Klassifizierer der Schwerpunktverteilung mit Gleitkommazahlen hat eine Maximalhöhe von sieben und eine Waldgröße von drei Bäumen. Der Klassifizierer der Schwerpunktverteilung mit Ganzzahlen hat eine Maximalhöhe von 12 und eine Waldgröße von drei Bäumen. Bei der Berechnung für die WCET muss insgesamt dreimal 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  $4686,8125 \mu s \approx 4,7 \text{ ms}$ . Das ist 14,7 ms,

## 5 EVALUATION

bzw. 75,8% schneller als das schnellste neuronale Netz von Giese mit Skalierung. Es ist anzumerken, dass in der Praxis die Puffergröße bei diesem Ansatz geringer ist, da 12 Byte 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.

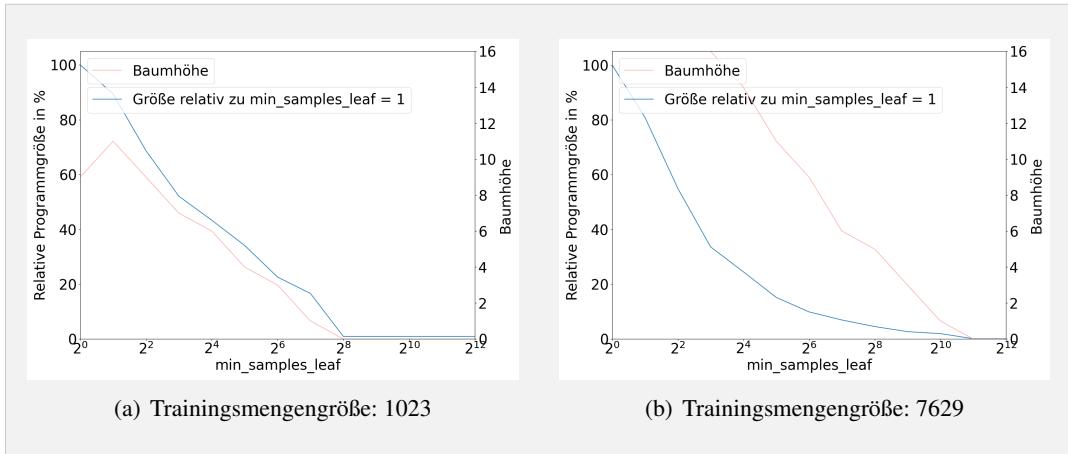
### 5.3 Programmgröße

Bei der Konstruktion eines Entscheidungsbaumes erhöht jede Teilung die Klassifizierungsgenauigkeit auf der Trainingsmenge und die Programmgröße die der Entscheidungsbaum benötigt. Folglich, sollte der verfügbare Programmspeicher vollständig ausgenutzt werden. Die Klassifizierungsgenauigkeit muss folglich maximiert werden, unter der Bedingung, dass der Programmspeicher nicht überschritten wird. Dementsprechend müssen die Teilungen bevorzugt werden, die den größten Zuwachs der Klassifizierungsgenauigkeit versprechen. Ein weiterer Lösungsansatz ist die Anzahl der Instruktionen pro Vergleich zu minimieren, damit mehr Vergleiche möglich sind.

#### 5.3.1 Maximierung des Zuwachses der Klassifizierungsgenauigkeit

Scikit-Learn bietet viele Parameter an, um den Teilungsprozess bei der Konstruktion zu steuern. Einer dieser Parameter ist `min_samples_leaf`, d. h. die minimale Anzahl an Einträgen der Trainingsmenge in einem Blatt. Dieser steuert die minimale Anzahl an Einträgen die in einem Kindknoten enthalten sein müssen, nachdem ein Knoten geteilt wurde. Standardmäßig ist der Wert 1. Dadurch entsteht ein sehr fein granularer Entscheidungsbaum, der viele Blätter mit nur einem Eintrag hat. Das heißt, es wurden Teilungen durchgeführt, die nur zwei Einträge unterteilt haben. Der Zuwachs der Klassifizierungsgenauigkeit ist durch diese Teilungen sehr gering. Eine Erhöhung in der Blattgröße verspricht, dass Entscheidungsbäume gefunden werden können, die zwar tiefer sind, aber dafür eine bessere Klassifizierungsgenauigkeit pro Vergleich erzielen. Der Baum kann tiefer werden, da die Grenze des Programmspeichers noch nicht erreicht wurde.

Abbildung 5.8 zeigt, wie sich der Parameter auf verschieden große Trainingsmengen auswirkt. Betrachtet wird eine Maximalhöhe von 16. Besonders bei großen Trainingsmengen (Abbildung 5.8(b)) kann die Programmgröße mit einer Blattgröße von 8 signifikant sinken,



**Abbildung 5.8:** Auswirkung der minimalen Blattgröße auf Programmgröße und Baumhöhe.

ohne die maximale Baumhöhe zu beeinflussen. Bei kleinen Trainingsmengen (Abbildung 5.8(a)) reduziert bereits bei kleinen Blattgrößen sich die Programmgröße ebenfalls signifikant. Allerdings wird die maximale Baumhöhe beeinflusst. Besonders bei großen Trainingsmengen wirkt sich der Parameter stärker auf die Programmgröße aus. Vermutlich, da die Anzahl der Blätter sich signifikant erhöht, je tiefer der Baum ist.

Es kann nicht argumentiert werden, dass ein Wert für die Blattgröße besser ist als ein Anderer, da die Klassifizierungsgenauigkeit auf der Trainingsmenge keine Aussage über die Klassifizierungsgenauigkeit auf der Testmenge treffen kann. Eine hohe Klassifizierungsgenauigkeit auf Trainingsmenge muss nicht unbedingt eine hohe Klassifizierungsgenauigkeit auf der Testmenge implizieren. Allerdings können so vermeintlich höhere Entscheidungsbäume ausgewählt werden, da diese weniger Programmspeicher benötigen im Vergleich zu gleich hohen Entscheidungsbäumen mit einer geringeren Blattgröße. Dieser Parameter vergrößert folglich den Suchraum. Bei kleinen Trainingsmengen ist eine Blattgröße über 1 nur sinnvoll, wenn der größte Entscheidungsbaum, der generiert werden kann, nicht innerhalb der Restriktionen des Programmspeichers liegt.

### 5.3.2 Minimierung der Instruktionen eines Vergleichs

Ein Vergleich in einem Entscheidungsbaum wurde in Kapitel 4.1.2 als Abzweigungsexpression mit einem Test definiert. Der Compiler erzeugt für den gleichen Programmcode verschiedene Instruktionen je nach Wahl des Datentyps.

## 5 EVALUATION

Listing 5.3 zeigt die Komplexität eines einzigen Vergleichs in Instruktionen eines Gleitkommazahlvergleichs. Zeile 1 bis 4 lädt die konstante Gleitkommazahl in 4 hintereinander liegende 8-Bit Register. Zeile 5 bis 7 lädt den Zeiger, der auf die Feature-Menge zeigt, und inkrementiert ihn 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. Damit benötigt ein Vergleich insgesamt 15 Instruktionen.

```
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 Feature als Gleitkommazahl mit konstanter Gleitkommazahl.

Zu vermeiden sind Zeile 5 bis 11, indem alle Features nur einmal in Register geladen werden. Dies ist allerdings nur möglich, wenn die Größe der Feature-Menge nicht 32 Byte übersteigt bei dem ATmega328P. Zusätzlich müssten noch Bytes verfügbar sein, um die Konstanten zu laden. Die Feature-Menge der Schwerpunktverteilung beinhaltet 10 Einträge. Der Ansatz mit Gleitkommazahlen ist mit 40 Bytes zu groß. Der Ansatz mit Ganzzahlen kann diese Optimierung mit 20 Byte ausnutzen. Der Compiler führt diese Optimierung bereits automatisch durch. Wenn die Feature-Menge zu groß ist, werden aus diesem Grund regelmäßig Register verdrängt, wodurch zusätzlich Instruktionen entstehen. Die Anzahl der Instruktionen können reduziert werden, indem die Größe der Feature-Menge reduziert wird, sodass die Feature-Menge und eine zusätzliche Konstante des gleichen Datentyps den Registerspeicher nicht übersteigen.

Der Datentyp `Float` ist sehr teuer für einen 8-Bit Prozessor, da immer 4 Register benötigt werden und gegebenenfalls zusätzliche Funktionen, die die fehlende Hardwareunterstützung ergänzen. Idealerweise sollte für die Feature-Menge und die Vergleiche ein 8-Bit Datentyp gewählt werden. Damit werden einerseits weniger Register benötigt, wodurch wiederum die Feature-Menge größer sein kann, und andererseits können hardwareunterstützte Vergleichsinstruktionen benutzt werden. Dies verringert die Anzahl der Instruktionen signifikant. Folglich vermindert ein kleinerer Datentyp die Anzahl der Instruktionen signifikant. Listing 5.4 zeigt

einen Vergleich von einem 8-Bit Datentyp. Im Kontrast zum Vergleich mit Gleitkommazahlen, werden 66,6% weniger Instruktionen benötigt.

```

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.

### 5.3.3 Minimierung der Instruktionen einer Rückgabe

Die Rückgabe der Klassifizierung in einem Entscheidungsbaum kann auf zwei Arten stattfinden. Einerseits kann lediglich die Klasse mit der höchsten Wahrscheinlichkeit zurückgegeben werden. Andererseits kann die Wahrscheinlichkeitsverteilung zurückgegeben werden, sodass die nächste Ebene die Entscheidung trifft. In Kapitel 4.1.2 wurde letzteres vorgestellt, da in der nächsten Ebene der Wahlklassifizierer die Entscheidungsbäume im Ensemble mit Hilfe ihrer Wahrscheinlichkeitsverteilungen zusammenfasst.

Listing 5.5 zeigt die Instruktionen die für die Zuweisung zu vier Klassen von 1.0, 0.0, 0.0 und 0.0 generiert werden. Für jede Klasse wird die Konstante Wahrscheinlichkeit in Register geladen und anschließend in den Rückgabeparameter gespeichert. In diesem Fall muss nur für die erste Klasse eine Konstante geladen werden, da jede andere Klasse 0 ist. Das heißt, dass im schlimmsten Fall 33 Instruktionen benötigt werden, anstatt 21. Der Compiler führt hier bereits eine Optimierung aus, indem für jedes Ergebnis ein eigener *Basic block* (Eine Datenstruktur die Instruktionen mit einer Annotation zusammenfasst) erzeugt wird. Zusätzlich könnte kein C-Code generiert werden für eine Zuweisungen mit 0. Dies erfordert aber, dass der Rückgabeparameter mit 0 vorinitialisiert ist.

## 5 EVALUATION

```
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 der Instruktionen einer Rückgabe der Wahrscheinlichkeitsverteilung eines Entscheidungsbaumes mit 4 Klassen.

Eine weitere Optimierung ist den Wahlklassifizierer diskret zu modellieren. Dabei wird für jede Rückgabe des Entscheidungsbaumes ein einstimmiges Ergebnis angenommen, d. h. es wird die Klasse mit der höchsten Wahrscheinlichkeit in jedem Baum zurückgegeben und nicht mehr die Wahrscheinlichkeitsverteilung. Dadurch werden lediglich die erkannten Klassen gezählt, anstatt die Wahrscheinlichkeitsverteilungen zu addieren. Listing 5.6 zeigt, dass sich die Anzahl der Instruktionen für eine Rückgabe auf genau 2 Instruktionen reduzieren. Zusätzlich kann der Compiler diese Rückgabe in Basic blocks extrahieren, wodurch lediglich eine Sprunginstruktion benötigt wird. Diese Optimierung ist bei dem diskreten Wahlklassifizierer noch effektiver, da es genau  $N$  verschiedene Rückgabewerte gibt, für  $N$  mögliche Klassen. Im schlimmsten Fall reduzieren sich die Anzahl der Instruktionen pro Rückgabe um  $\frac{100}{1+4N}\%$  und im besten Fall um  $\frac{100}{1+8N}\%$ .

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

■ **Listing 5.6:** Beispiel des Assemblycodes der Rückgabe eines diskreten Wahlklassifizierers.

Der Nachteil dieses Ansatzes ist, dass die Ergebnisse instabil werden können, wenn viele Rückgaben nur über eine knappe Mehrheit verfügen. Das ist insbesondere der Fall in Kombination mit einem hohen Wert für die Blattgröße, da dieser die Anzahl der Blattknoten mit Einträgen aus verschiedenen Klassen potenziell erhöht. Diese Optimierung kann auf eine

### 5.3 PROGRAMMGRÖSSE

gefundene Lösung angewendet werden, die zu groß für den Programmspeicher ist. Anschließend sollte die Klassifizierungsgenauigkeit revalidiert werden. Tests haben ergeben, dass die Klassifizierungsgenauigkeit geringfügig schwankt. Folglich kann sich die Klassifizierungsgenauigkeit auf der Testmenge auch erhöhen.

Denkbar wäre ein hybrider Ansatz, der bei einem eindeutigen Ergebnis die Klasse zurück gibt und ansonsten die Wahrscheinlichkeitsverteilung. Die „Eindeutigkeit“ kann über einen Schwellenwert  $\delta$  definiert sein. Ein Schwellenwert von  $\delta = 0$  würde an der Korrektheit nichts ändern, würde aber im schlimmsten Fall die Programmgröße nicht verringern. Tests haben ergeben, dass es immer eindeutige Ergebnisse gibt, weswegen diese Optimierung immer angewendet werden sollte.

## 5 EVALUATION

## Diskussion

Der Fokus der Arbeit lag sehr früh auf der Evaluierung von verschiedenen Ensemble-Methoden und Hyperparametern. Dieser Ansatz ist aber nicht optimal, da die Ensemble-Methoden ihr Potenzial nicht vollständig ausnutzen konnten. Der Grund dafür ist die Wahl der Feature-Mengen. Im engeren Sinne sind die für diese Arbeit verwendeten Feature-Mengen keine Feature-Mengen, sondern Features die aus mehreren Werten bestehen. Dementsprechend können Ensemble-Methoden, wie Random Forests oder ExtraTrees keinen Vorteil daraus ziehen, da sie nur Teilmengen der Feature-Menge nutzen. Diese Ansätze funktionieren besser mit eigenständigen Features.

Vermutlich wäre es sinnvoll gewesen Stacking zu untersuchen. Dieser Ansatz baut iterativ einen Entscheidungsbaum nach dem anderen, auf Basis der Ergebnisse der vorherigen Entscheidungsbäume und unterschiedlichen Feature-Mengen. Der Vorteil ist, dass die einzelnen Feature-Mengen nicht so viele Anforderungen erfüllen müssen, solange alle unterschiedlichen Feature-Mengen zusammen alle Anforderungen erfüllen. Dies erleichtert die Suche nach Features signifikant. Mit jedem Entscheidungsbaum können einfache Attribute inferiert werden, wie z. B. dass es eine horizontale Bewegung ist. Im folgenden Entscheidungsbaum müsste dann lediglich die Richtung unterschieden werden. Es wird vermutet, dass der resultierende Ansatz simpler ist und kleinere Entscheidungswälder generiert.

Insgesamt wurden von 4 Personen Handgesten unter unterschiedlichen Lichtverhältnissen erfasst. Dies repräsentiert allerdings nicht alle möglichen Lichtverhältnisse. Aus diesem Grund könnte ein Teil der synthetischen Daten zur Überprüfung der Lichtverhältnisse dazu genutzt werden, um Modelle zu generieren die robuster gegenüber bisher nicht erfassten Lichtverhältnissen in der Trainingsmenge sind.

## 6 DISKUSSION

## Schlussfolgerungen

Diese Arbeit hat gezeigt, dass sich Klassifizierer mit Entscheidungsbäumen sehr gut für die Handgestenerkennung eignen. Sie können, sowohl unter guten als auch relativ schlechten Lichtverhältnissen, sehr hohe Klassifizierungsgenauigkeiten erzielen. Nullgesten können von validen Handgesten unterschieden werden. Dabei sind diese Klassifizierer signifikant schneller als KNNs und benötigen kaum RAM zur Ausführung. Die Klassifizierungsgenauigkeit ist abhängig vom Programmspeicher der genutzt werden darf.

Das beste Modell, der Klassifizierer der kombinierten Schwerpunktverteilung, erzielte eine Klassifizierungsgenauigkeit von 94,8% auf der Testmenge von Klisch, 99% auf der Gestentestmenge und 95,8% auf der Nullgestentestmenge. Damit ist der Ansatz 4,2% schlechter als der beste Klassifizierer der vorherigen Arbeiten, der 98,96% auf der Testmenge von Klisch 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 Klassifizierungsgenauigkeit. Die Worst Case Execution Time (WCET) beläuft sich auf 5,8 ms. Der Großteil, ca. 4,5 ms, wird davon für die Berechnung der Features benötigt. Damit benötigt der Ansatz nur 29,9% der Ausführungszeit des schnellsten vorherigen Ansatzes. Nach Anwendung aller Optimierungen und unter Annahme, dass Festkommazahlen keine Auswirkung auf die Klassifizierungsgenauigkeit haben, ist der Klassifizierer trotzdem zu groß für den ATmega328P.

Mit einer Beschränkung von 48 kB Programmspeicher kann der Klassifizierer der kombinierten Schwerpunktverteilung 87,5% auf der Testmenge von Klisch erzielen, 97,7% auf der Gestentestmenge und 92,9% auf der Nullgestentestmenge bei einer Programmgröße von 33276 Byte und einer WCET von 4,7 ms. Mit der Beschränkung auf 32 kB, kann 87,5% auf der Testmenge von Klisch erzielt werden, 96,9% auf der Gestentestmenge und 92,5% auf der Nullgestentestmenge bei einer Programmgröße von 20552 Byte und einer WCET von 4,7 ms.

## 7 SCHLUSSFOLGERUNGEN

Die schnellste Ausführungszeit hatte der Klassifizierer Schwerpunktverteilung mit Ganzzahlen. Mit einer Programmgröße unter 28 kB, hat dieser eine WCET von 0,7 ms und benötigt damit nur 3,6% der Ausführungszeit des schnellsten KNN von Giese mit Skalierung [Gie20]. Dieser Ansatz erzielt bessere Ergebnisse als der kombinierte Ansatz auf den Testmengen unter gleichen Restriktionen, ist aber nicht robust gegenüber skalierten Helligkeiten.

Insgesamt benötigt die Implementierung der kombinierten Schwerpunktverteilung auf dem ATmega328P 1640 Bytes RAM. Dieser setzt sich zusammen aus 60 Bytes zur Berechnung der Features, 1200 Bytes für den Puffer mit einer Größe von 100 Einträgen und 380 Bytes für Variablen der restlichen Firmware. Im Puffer werden im Gegensatz zu bisherigen Arbeiten keine Bilder gespeichert, sondern partiell ausgerechnete Features, d. h. im Fall der Schwerpunktverteilung, der Schwerpunkt jedes Bildes. Dafür wird weniger Speicher benötigt als pro Bild, wodurch Gestenkandidaten mit mehr Bildern möglich sind als bei den KNNs. Die Programmgröße beträgt 31520 Bytes.

Der Entscheidungsbaum bietet viel Optimierungspotenzial gegenüber der naiven Implementierung. Zum Beispiel hat der für Features gewählte Datentyp einen großen Einfluss sowohl auf die Programmgröße, als auch die Ausführungsgeschwindigkeit. Dies liegt am ATmega328P, der als 8-Bit Prozessor über kein Modul zur Verarbeitung von Gleitkommazahlen verfügt. Die Verarbeitung von Gleitkommazahlen erfordert daher viel Ausführungszeit im Vergleich zum 8-Bit Integer. Weitere Optimierungen sind Festkommazahlen und die Verwendung eines hybriden bzw. diskreten Wahlklassifizierers.

Insgesamt ist es sehr aufwändig den potenziell besten Klassifizierer zu finden, da es viele Parameter gibt, die in Kombination zu unterschiedlichen Klassifizierern führen. Zudem ist die Konstruktion nicht immer deterministisch, weswegen sie als Monte Carlo Methode betrachtet werden kann. Daher wurden für diese Arbeit insgesamt 22528 verschiedene Konfigurationen mit 140 verschiedenen Startwerten für den Zufallsgenerator untersucht und 30 Variationen an Features.

In folgenden Arbeiten sollte untersucht werden, ob Stacking oder hierarchische Klassifizierer nicht für Klassifizierer mit Entscheidungsbäumen auf kleinen eingebetteten Systemen besser geeignet sind. Es wird vermutet, dass dadurch simplere Klassifizierer generiert werden können, die simplere Feature-Mengen verwenden können. Außerdem könnte untersucht werden, ob KNNs nicht wesentlich kleiner sein können, wenn die Features dieser Arbeit verwendet werden, anstatt die Rohdaten der Handgeste.

## Inhalt des USB-Sticks

- Latex-Quellcode und PDF dieses Dokuments (/thesis/thesis.pdf)
- Quellcode der gesamten Infrastruktur (/feature\_extractor, /gesture\_recorder, /lib\_data\_set, /lib\_evaluation, /lib\_feature, /lib\_gesture, /model, /serial\_reader, /simulation)
- Dokumentation der Infrastruktur (/README.md, /doc/feature\_extractor/index.html)
- Hilfsskripte zum trainieren, validieren und generieren von Grafiken (/evaluation, /plotting, /test\_env, /test\_env\_size, /test\_env)
- Verschiedene Versionen der Arduino-Firmware (/ino\_tree, /ino\_tree2, /ino\_tree3, /ino\_tree4)
- Trainings- und Testmengen (/data/export\_of\_dymel\_data\_sets.zip, /data/dymel\_data)
- Ergebnisse der Modelle auf den Testdaten in Rohform (/eval\_data.csv, /ccp\_alpha\_big.csv, /ccp\_alpha\_small.csv, /data\_min\_sample\_leaf\_big.csv, /data\_min\_sample\_leaf\_small.csv, /light\_eval.csv, /light\_eval2.csv)
- Arduino C-Code der besten Konfiguration jeder Feature-Menge die in der Speicher des ATmega328P passen (/model\_export)

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: *1.11. Ensemble methods.* <https://scikit-learn.org/stable/modules/ensemble.html#ensemble>. Version: 2020
- [Ent20c] 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

## LITERATURVERZEICHNIS

- [Fri99] FRIEDMAN, Jerome H.: Greedy Function Approximation: A Gradient Boosting Machine. (1999), S. 34
- [FS97] FREUND, Yoav ; SCHAPIRE, Robert E.: A decision-theoretic generalization of on-line learning and an application to boosting. In: *Journal of computer and system sciences* 55 (1997), Nr. 1, S. 119–139
- [GEW06] GEURTS, Pierre ; ERNST, Damien ; WEHENKEL, Louis: Extremely randomized trees. In: *Machine learning* 63 (2006), Nr. 1, S. 3–42
- [Gie20] GIESE, Anton: *Compression of Artificial Neural Networks for Hand Gesture Recognition*, Bachelor's Thesis, 10 2020. – 1–46 S.
- [Kli20] KLISCH, Daniel: *Training of Recurrent Neural Networks as Multiple Feed Forward Networks*, Research Project and Seminar, 01 2020. – 1–39 S.
- [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: CART: classification and regression trees. In: *The top ten algorithms in data mining* 9 (2009), S. 179–201

## LITERATURVERZEICHNIS

- [VKK<sup>+</sup>20] VENZKE, Marcus ; KLISCH, Daniel ; KUBIK, Philipp ; ALI, Asad ; MISSIER, Jesper D. ; TURAU, Volker: *Artificial Neural Networks for Sensor Data Classification on Small Embedded Systems*. 2020
- [VT20] VENZKE, Marcus ; TURAU, Volker: Ansatz: Schwerpunkt der Pixel. (2020), 11, S. 1