Bachelorarbeit

Entwicklung eines eingebetteten Systems zur Erkennung akustischer Warnsignale von Fahrradfahrern unter Einsatz künstlicher neuronaler Netzwerke

Clemens Kubach

Universität Potsdam

5. Juli 2022

Bachelorarbeit

Entwicklung eines eingebetteten Systems zur Erkennung akustischer Warnsignale von Fahrradfahrern unter Einsatz künstlicher neuronaler Netzwerke

Development of an Embedded System for the Detection of Cyclists' Acoustic Warning Signals Using Artificial Neural Networks

von
Clemens Kubach

Betreuung
Prof. Dr.-Ing. Benno Stabernack, Michał Steć
Architekturen eingebetteter Systeme für die Signalverarbeitung

Universität Potsdam

5. Juli 2022

Zusammenfassung

Die Arbeit hat die Zielsetzung, ein eingebettetes System zur Erkennung der akustischen Warnsignale von Fahrradklingeln zu entwickeln. Es sollen dabei künstliche neuronale Netzwerke für die Erkennung des Soundevents des Fahrradklingelns verwendet werden.

Drei verschiedene Architekturen neuronaler Netze werden definiert, und entsprechende Modelle trainiert und evaluiert. Außerdem wird die verwendete Hardware betrachtet und ein Softwaresystem zur Integration von Modellen und Hardware implementiert. Die Audiodaten werden live über ein Mikrofon aufgenommen, Soundevent-Klassifikationen mit neuronalen Netzen auf einem NVIDIA Jetson Nano in Echtzeit durchgeführt und zu Detektionen im zeitlichen Verlauf aggregiert.

Zwei der drei vorgestellten Modelle aus der Arbeit konnten ähnliche Performanz wie andere state-of-the-art Modelle anderer Soundevents erreichen: Ein Modell basierend auf dem vortrainierten YAMNet unter Einsatz eines Transfer Learning-Ansatzes und ein Modell basierend auf einem Convolutional Recurrent Neural Network. Das umgesetzte Softwaresystem wurde als Kommandozeilen-Tool, mit Möglichkeiten zur Untersuchung von Konfigurationen und Performanz des Gesamtsystems, entwickelt. Eine Individualisierbarkeit der Ergebnisausgabe ermöglicht die Integration in andere Systeme und bietet auf diese Weise eine flexible Anwendungsmöglichkeit.

Erste Tests des Gesamtsystems zeigten, dass eine Verarbeitungszeit von circa 32 ms für eine Sekunde Audiodaten mit dem Convolutional Recurrent Neural Network auf dem Jetson Nano erreicht und die Erkennung von Fahrradklingel-Sounds erfolgreich umgesetzt werden konnten.

Inhaltsverzeichnis

1 Einleitung

Mit 9,995 Milliarden beförderten Personen im Jahr 2019 fallen laut dem Bundesministerium für Verkehr und digitale Infrastruktur über 10 Prozent des Verkehrsaufkommens auf den Fahrradverkehr [42]. Damit ist er nach dem motorisierten Individualverkehr und dem Fußverkehr unter den meistgenutzten Arten des Personenverkehrs in Deutschland. Radfahren bietet viele Vorteile, sowohl für die Umwelt als auch für die eigene Fitness. Auch die Bundesregierung fördert den Radverkehr, wobei die Sicherheit der Radfahrer ein wichtiger Aspekt ist [30]. Radfahrer sind bei Verkehrsunfällen insbesondere gegenüber Kraftfahrzeugen besonders gefährdet. Allein in Deutschland gab es im Jahr 2020 laut dem Statistischen Bundesamt 91 533 Fahrradunfälle mit Personenschaden, wobei in über 71 Prozent der Unfälle ein Personenkraftwagen beteiligt war [19].

Im Rahmen dieser Arbeit soll ein System zur Erkennung der Signale von Fahrradklingeln implementiert werden. Als Prototyp wird es mit dem Hintergrund entwickelt, in einem Kraftfahrzeug integriert zu werden und damit Fahrradfahrern die Möglichkeit zu geben, den Fahrer akustisch über ihre Präsenz zu informieren. Es gibt bereits verschiedene Systeme zum Schutz von Fahrradfahrern in gefährlichen Positionen relativ zum Fahrzeug, wie den toten Winkeln. In der Literatur gibt es dazu verschiedene Ansätze. Beispiele dafür sind die Erkennung von Fahrradfahrern mit verschiedenen Sensoren oder die Verwendung von externen Airbags bei Kollisionen. Soweit bekannt, gibt es jedoch noch kein System für Kraftfahrzeuge, welches Fahrradfahrern eine akustische und aktive Möglichkeit zur Warnung gibt. Dieser neue Ansatz ist besonders als eine Komponente für Mehrsensorsysteme interessant, wie das von Steć, Stabernack und Kubach [38]. Diverse Sensoren werden hierbei für ein optimales Rechtsabbiegeassistenzsystem erforscht, um entsprechende Unfälle mit schwachen Verkehrsteilnehmern, insbesondere Fahrradfahrern, zu verhindern. Ein Prototyp zur Untersuchung des Ansatzes wird auf Basis von Mikrofon-Sensoren mit dieser Arbeit umgesetzt.

Die Verwendung von Fahrradklingeln ist zur Warnsignalisierung besonders geeignet. Eine Montage zusätzlicher Hardware am Fahrrad ist nicht erforderlich, weil in Deutschland laut der Straßenverkehrs-Zulassungs-Ordnung jedes Fahrrad im Straßenverkehr „mit mindestens einer helltönenden Glocke ausgerüstet sein“ (§ 64a StVZO) muss. Die Möglichkeit des Fahrers des Kraftfahrzeugs, ein Fahrradklingeln wahrzunehmen, ist jedoch durch eine Vielzahl von Einflüssen geprägt, weshalb das hier vorgestellte System notwendig ist.

Zur Erkennung der Geräusche von Fahrradklingeln in den aufgenommenen Live-Audiodaten werden künstliche neuronale Netzwerke[1] erstellt und angewendet. In vielen Bereichen, wie der Computer Vision oder dem Natural Language Processing, werden tiefe neuronale Netzwerke zur Lösung von Aufgaben eingesetzt, die ohne diese Techniken nicht automatisiert in solcher Qualität lösbar wären. Forscher konnten auch

[1] Als Synonym für den Begriff *künstliches neuronales Netzwerk* werden ebenso die Bezeichnungen *neuronales Netzwerk*, *neuronales Netz* und *Neural Network* verwendet.

für Aufgaben zu Audiosignalen neue state-of-the-art Modelle entwickeln, wie für die Soundevent-Klassifizierung [28] oder die Soundevent-Detektion [21]. Angesichts dessen stellt die Verwendung von künstlichen neuronalen Netzwerken eine vielversprechende Erkennungsmethode dar, auf welche diese Arbeit eingegrenzt wird.

Es wird ein vollständiges eingebettetes System in Form eines ersten lauffähigen Prototyps, mit Hardware und Software, zur Erkennung der akustischen Warnsignale entwickelt. Es ist dabei nicht die Absicht dieser Arbeit, ein marktfähiges oder alleinstehendes Produkt zu entwickeln oder den Ansatz der aktiven akustischen Warnung mit anderen Ansätzen zu vergleichen. Das eingebettete System soll die Implementierung einer performanten Software umfassen, welche auf einem Einplatinencomputer die Audio-Sensordaten mittels eines Mikrofons aufnimmt, mit einem neuronalen Netzwerk verarbeitet und das Erkennungsergebnis zurückgibt. Nachfolgende Untersuchungen, Integrierungen und Weiterentwicklungen sollen mit dem Gesamtsystem ermöglicht werden.

Im nächsten Kapitel geht es zuerst um die Klärung und Abgrenzung bestimmter Begriffe. Dabei werden einige bestehende Ansätze aus der Literatur aufgezeigt, um den Grundgedanken der Arbeit einzuordnen. Der Hauptteil der Arbeit befasst sich mit den drei Kernaufgaben zur Umsetzung des Systems: die Beschreibung und Vorbereitung der Hardware, das Modell zur Erkennung der Fahrradklingel-Sounds und die Software, die auf der Zielplattform bereitgestellt wird, und damit Hardware und Modell vereint. Abschließend folgt ein Fazit zur Arbeit mit einer Zusammenfassung und einem Ausblick auf mögliche Ansätze zur Weiterentwicklung und Verbesserung des Systems.

2 Grundlagen

2.1 Audiosignale

2.1.1 Metriken

Audiosignale sind eine Darstellung von Geräuschen und werden in Form von Schallwellen mit Sensoren, den Mikrofonen, empfangen. Das ursprünglich analoge Signal wird in ein digitales Signal umgewandelt. Bei der Entwicklung von Softwarekomponenten mit Bezug zu Audiodaten können verschiedene Eigenschaften der digitalen Signale relevant sein. Für diese Arbeit zählen dazu insbesondere die Samplingrate, die Samplingtiefe, die Lautstärke und die Kanalanzahl.

Samplingrate. Das ursprüngliche analoge, also zeit- und wertekontinuierliche, Signal wird durch Abtastung in ein zeitdiskretes Signal umgewandelt. Die Samplingrate, auch Abtastrate genannt, gibt die Anzahl der Messwerte eines Signales innerhalb eines bestimmten Zeitintervalls an. Die Messwerte werden auch als Samples bezeichnet. Angegeben wird die Rate typischerweise in der Einheit Hertz (Hz) als die Anzahl der Abtastungen pro Sekunde. Für die Aufnahme eines Audiosignales mit einer Samplingrate von 44 kHz, also 44 000 Hz, werden dementsprechend 44 000 einzelne Samples durch die Messung der Amplituden erfasst.

Samplingtiefe. Durch Quantisierung kann nach der Abtastung aus dem zeitdiskreten, wertekontinuierlichen Signal ein zeit- und wertediskretes Signal ermittelt werden. Damit liegt ein digitales Signal vor. Wie viele Bits für die Darstellung eines Amplitudenwertes zur Verfügung stehen, gibt die Samplingtiefe an. Unter Verwendung von mehr Bits zur Quantisierung sind mehr mögliche Zustände zum Speichern eines jeweiligen Amplitudenwertes verfügbar. Es ist dementsprechend eine feinere Abbildung des Signales möglich. Eine Samplingtiefe von 2 Bits würde nur 4 Zustände zulassen, wodurch die Qualität sehr eingeschränkt wäre. Häufig wird eine solche Umwandlung mit 8 Bits bis 32 Bits durchgeführt, was bei 32 Bits bereits $2^{32} = 4\,294\,967\,296$ Abstufungen der Amplitude zulässt.

Lautstärke. Audiosignale mit verschiedenen Amplituden können unterschiedlich laut wahrgenommen werden. Eine Information über die Lautstärke eines Audiosignales wird meist in der Maßeinheit Dezibel (dB) ausgedrückt. Dazu werden die Amplitudenwerte S relativ zu einem Referenzwert r logarithmisch durch den Ausdruck $20 * \log_{10}(\frac{S}{r})$ skaliert [23]. Das ist für die große Spanne der menschlichen Wahrnehmbarkeit von unterschiedlichen Lautstärken hilfreich.

Kanalanzahl. Werden ausschließlich die Daten von einem Sensor bezogen, wird von Einkanal-Audiodaten gesprochen. Diese werden auch als Monokanal-Audio bezeichnet. In aktuellen Alltagsgeräten sind häufig mehrere Mikrofone integriert. Beispielsweise in Sprachassistenzsystemen: Zur Erkennung von Sprachkommandos wird häufig Mehrkanal-Audio für eine höhere Aufnahmequalität durch Lokalisierung der wichtigen Soundquelle genutzt. Jedes Mikrofon liefert simultan die Daten eines einzelnen Kanals.

2.1.2 Darstellungsformen

Um Audiodaten zu betrachten, wird oftmalig eine der zwei Darstellungsformen genutzt: die Wellenform oder das Spektrogramm. Beide Varianten werden dazu verwendet, den zeitlichen Verlauf der akustischen Signale darzustellen. Die Formen können als Feature-Input für die Verarbeitung durch künstliche neuronale Netzwerke genutzt werden, wie von Li u. a. [20] ausführlich untersucht wurde.

Wellenform. Die Wellenform ist ein Array von ganzen Zahlen oder Gleitkommazahlen, die den zeitlichen Verlauf der Amplituden eines Audiosignals repräsentiert. Ein einzelnes Sample des Signales ist dabei ein Wert in dem Array. In Abbildung 2.1 ist eine Wellenform von einem Beispielsound einer Fahrradklingel als Hüllkurve der Amplitudenwerte visualisiert.

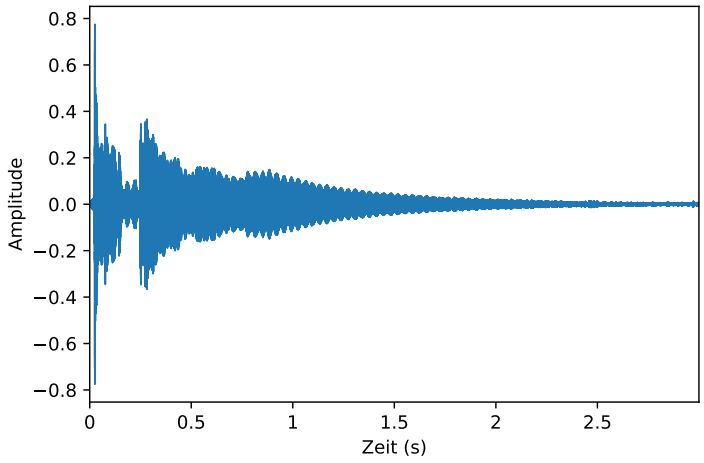


Abbildung 2.1: Wellenform des Geräusches einer Fahrradklingel. Die Amplituden werden als Gleitkommazahlen zwischen -1 bis 1 über einen zeitlichen Verlauf dargestellt.

Spektrogramm. Das Spektrogramm betrachtet den zeitlichen Verlauf der Amplitudenverteilung über einen gegebenen Frequenzbereich. Für Monokanal-Audio sind die Audiodaten in einem zweidimensionalen Array als Amplitudenwerte für jedes Zeit-Frequenz-Paar gespeichert. Dabei werden meist diverse Skalierungen ausgeführt, um die Werte signifikanter zu machen. Beispiele dafür sind eine logarithmische Skalierung der Frequenzen

oder Amplituden, oder die Verwendung der Mel-Skala. Unter Einsatz aller genannten Skalierungen wird das sogenannte Log-Mel-Spektrogramm in der Abbildung 2.2 für denselben Beispielsound einer Fahrradklingel visualisiert, der bereits als Wellenform in Abbildung 2.1 gezeigt wurde. Häufig werden die Werte des Arrays farblich für eine Visualisierung kodiert. Die positiven Effekte auf die Erkennungsperformanz in diversen Aufgaben konnten mehrfach nachgewiesen werden und sind auch durch Domänenwissen zum menschlichen Gehörsystem gestützt [9]. Menschen können Unterschiede zwischen zwei hochfrequenten Geräuschen schlechter wahrnehmen als den Unterschied zweier niederfrequenten Geräuschen bei jeweils gleichem Abstand. Das Vorgehen einer solchen Skalierung zum Log-Mel-Spektrogramm wird in 3.3.2.2 detailliert behandelt.

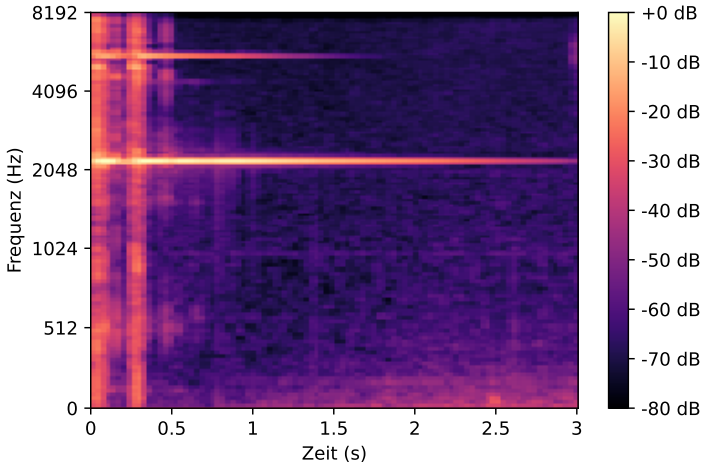


Abbildung 2.2: Spektrogramm von einem Beispielsound einer Fahrradklingel. Die Amplituden werden in Dezibel, relativ zu dem maximalen Wert, angegeben. Die farblich hellsten Pixel im Bild repräsentieren die Zeit-Frequenz-Punkte mit dem höchsten Amplitudenwert.

2.1.3 Speicherform

Das WAVE-Dateiformat wird oft für Aufgaben der Audioverarbeitung zur persistenten Speicherung von Audio-Rohdaten genutzt. In einer WAVE-Datei liegen häufig die Rohdaten als ganze Zahlen oder Gleitkommazahlen in dem Format der Puls-Code-Modulation (PCM) oder dem Float-Format vor. Außerdem sieht das Dateiformat mehrere Abschnitte zur Speicherung vor. Neben dem Abschnitt mit den Audiorohdaten können beim Lesen der Datei gegebenenfalls Metadaten wie das verwendete Datenformat, die Samplingrate, die Samplingtiefe oder die Kanalanzahl aus einer WAVE-Datei extrahiert werden.

2.2 Aufgabenarten bei der Erkennung von Soundevents

Bei der Erkennung von Soundevents in Audiodaten sind besonders drei Aufgabenarten grundlegend: die Klassifizierung, das Tagging und die Detektion [24], [29]. Alle drei

Begriffe liegen sehr nah beieinander, weshalb eine Abgrenzung an dieser Stelle wichtig ist. Ein Audiobeispiel wird ebenso als Sample bezeichnet. Der Begriff Sample ist in seiner Bedeutung somit durch den Kontext von einem einzelnen Sample in den Audiodaten zu unterscheiden.

Soundevent-Klassifizierung. Die Aufgabe der Soundevent-Klassifizierung (engl. Sound Event Classification – SEC) beschreibt die Zuweisung einer einzigen Event-Klasse zu einem gesamten Audio-Clip. Wie von Mesaros, Heittola und Virtanen [24] definiert wird, können zusätzlich zum Ziel-Soundevent weitere Geräusche in den Audio-Clips vorkommen, in den meisten Fällen sind die Soundevents jedoch isoliert. Nach der Definition ist es außerdem kein Bestandteil der Klassifizierung, die Anfangs- und Endzeiten des gefundenen Soundevents zu bestimmen. Der Ansatz der SEC ist die Grundlage der Lösung vieler Probleme bezüglich akustischer Events. Er wird beispielsweise in den Arbeiten [28], [46] und [26] eingesetzt.

Soundevent-Tagging. In der Arbeit von Choi, Fazekas und Sandler [9] wird ein Algorithmus basierend auf neuronalen Netzen für ein automatisches Tagging von Audio-Clips entwickelt. Hierbei kann ein Audiosample mehrere Ziel-Klassen beinhalten, die durch das Soundevent-Tagging extrahiert werden sollen. Die möglichen Geräusche können überlappend stattfinden und entsprechende Clips jeweils mit mehreren Events klassifiziert werden. Wie bei der SEC, gehört das Extrahieren von zeitlichen Informationen nicht zu dieser Aufgabe [29].

Soundevent-Detektion. Die Soundevent-Detektion (engl. Sound Event Detection – SED) befasst sich zusätzlich zu der Erkennung der Sound-Klassen mit der Bestimmung der zeitlichen Lokalisierung in dem Audio-Clip [24]. Dabei kann es für mehrere überlappende Ziel-Sounds, wie beim Tagging, oder nur für eine Klasse pro Sample, wie bei der Klassifizierung, ausgeführt werden. Es müssen im zeitlichen Verlauf Informationen zu dem Auftreten der jeweiligen Soundevents gegeben sein.

2.3 Neuronale Netzwerke für die Erkennung von Soundevents

Künstliche neuronale Netzwerke (engl. Artificial Neural Networks), genauer tiefe neuronale Netzwerke (engl. Deep Neural Networks – DNN), haben durch höhere Performanz in vielen Anwendungsbereichen die klassischen Machine Learning-Verfahren abgelöst [33].

In der Abbildung 2.3 ist ein einfaches künstliches neuronales Netz mit zwei Schichten von künstlichen Neuronen (Units) visualisiert. Die Eingabewerte auf der linken Seite werden über die Layer verarbeitet und ein Ausgabewert erzeugt. In dem Beispiel wurden dazu zwei Layer verwendet. Sie werden als Fully-connected Layer oder Dense-Layer bezeichnet, da jedes Neuron eines solchen Layers mit jedem Ergebniswert der vorhergehenden Schicht oder, wie im betrachteten Fall, mit jedem der Eingabewerte verbunden ist.

Ein vereinfachtes, dem biologischen Vorbild nachempfundenes, künstliches Neuron[2] ist aufgebaut aus lernbaren Parametern, einer Propagierungsfunktion Σ und einer Akti-

[2] Die Struktur des in diesem Absatz beschriebenen Modell eines Neurons basiert auf den Ausführungen von Ruland [32].

vierungsfunktion ϕ. Zu den Parametern gehören die Gewichtungen w_i der Eingaben x_i und der Schwellenwert Θ für den internen Zustand des Neurons. Die Ausgabe o ist durch Gleichung 2.1, als Anwendung der Aktivierungsfunktion auf v und den Schwellenwert, modelliert. Wie in Gleichung 2.2 notiert ist, wird dabei die Propagierungsfunktion zur Verknüpfung der Eingaben mit Gewichtungen ausgeführt.

$$o = \phi(v, \Theta) \qquad (2.1) \qquad\qquad v = \sum_{i=1}^{n} x_i * w_i \qquad (2.2)$$

Mithilfe von Lernverfahren werden die lernbaren Parameter der Neuronen unter Verwendung von Trainingsdaten schrittweise angepasst. Das Netz kann dadurch eine Mustererkennung erlernen und so eine Abbildung von Eingabewerten auf Ausgabewerte erreichen. Bei der Inferenz, also der Anwendung der trainierten neuronalen Netzwerke, können die entstandenen Modelle für gleichartige, aber neue Eingabedaten eingesetzt werden.

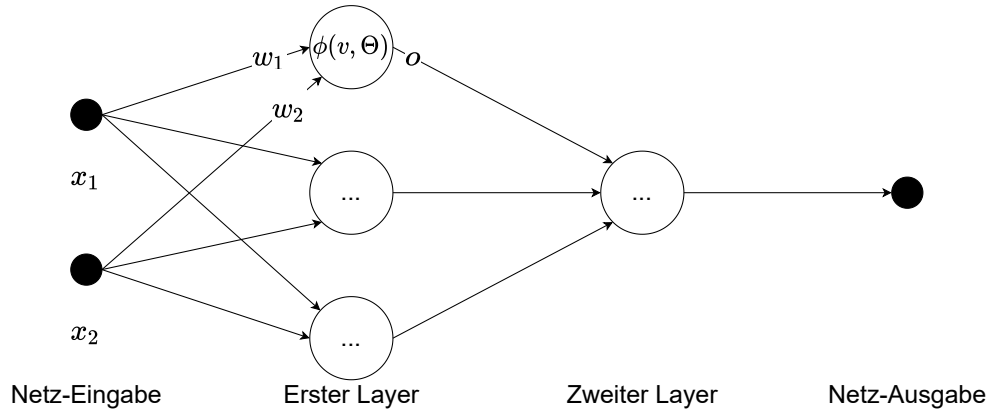


Abbildung 2.3: Darstellung eines einfachen künstlichen neuronalen Netzes mit zwei Layern und einer Verarbeitung innerhalb eines Neurons.

Tiefe neuronale Netzwerke zeichnen sich durch die Verbindung vieler Layer und eine hohe Parameteranzahl aus [29]. Für ein erfolgreiches Trainieren der DNN Modelle sind meist möglichst viele Daten erforderlich. Es werden automatisierte Ergebnisse mit einer hohen Qualität ermöglicht, die zuvor nicht ohne menschliche Arbeit erreicht werden konnten. Auch für diverse Aufgaben bezüglich Soundevents bei der Audiosignalverarbeitung konnte die hohe Performanz von neuronalen Netzwerken vielmals nachgewiesen werden [29], [7] und die Performanz klassischer Lösungen wie der Gaussian-Mixture-Modelle signifikant überbieten [6].

Grundlegende Architekturen. Besonders verbreitet für die SEC und SED sind künstliche neuronale Netzwerke, die auf Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN) oder einer Verbindung beider Ansätze, den Convolutional Recurrent Neural Networks (CRNN), basieren.

Convolutional Neural Networks. Convolutional Neural Networks werden häufig für Aufgaben in dem Bereich der Computer Vision angewendet. Sie sind typischerweise aus

einem oder mehreren Convolutional-Layern, Pooling-Layern und Fully-connected-Layern aufgebaut [46].

Ein Convolutional-Layer basiert auf der Operation der Faltung (engl. Convolution) und dient der Feature-Extraktion. Über die Daten, in der Form einer Matrix, wird ein trainierbarer Filterkernel bewegt und jeweils für jede Position ein Skalarprodukt gebildet. Damit können sich einzelne Muster aus den Eingabedaten herausstellen. Häufig wird Max-Pooling im Pooling-Layer eingesetzt, um die Daten anschließend auf die wichtigen Charakteristiken zu reduzieren. Im Fall von Audiodaten in der Form eines Spektrogramms und einer 1D-Convolution kann der Kernel als 1D-Array als verschiebbares Fenster über einen Frame, also über die Amplitudenverteilung eines kurzen Zeitabschnitts, bewegt werden. Ebenfalls sind andere Kombinationen von Filterkernel und Darstellungsformen üblich und werden von Purwins u. a. [29] ausführlich beschrieben. Die Fully-connected-Layer im CNN sorgen für die eigentliche Klassifizierung der Features und die abschließende Ausgabe. Der Ausgabe-Layer des CNN hat ein Neuron für jede Klasse und gibt, durch die Verwendung einer bestimmten Aktivierungsfunktion, Wahrscheinlichkeitswerte für die Klassen zurück.

Insbesondere die Arbeit von Piczak [28] hat den signifikanten Fortschritt von CNNs für die Sound-Klassifikation von Umgebungsgeräuschen aufgezeigt. Mehrere Autoren nutzen für ihre Modelle ein log-skaliertes Mel-Spektrogramm und erreichten damit vielversprechende Ergebnisse [47], [4]. Doch auch die Anwendung auf der Wellenform oder hybride Verfahren, wie es von Li u. a. [20] vorgestellt wird, können Vorteile aufweisen.

Recurrent Neural Networks. Ein Ansatz mit einem RNN für Aufgaben mit Audiodaten ist häufig damit begründet, insbesondere größeren temporalen Kontext bei der Entscheidung einzubeziehen. Es können Informationen von vorherigen Zeitschritten gespeichert und bei Neuronen späterer Schritte wieder aufgegriffen werden. Dazu sind insbesondere RNNs basierend auf Long Short-Term Memory (LSTM) [13] -Layern hilfreich. Nach der Aussage von Adavanne u. a. [2] und eigener Literaturrecherche zeigt sich das LSTM als häufigste Art eines Recurrent Neural Networks. Es wird in mehreren Arbeiten für die SED verwendet [2], [27].

Convolutional Recurrent Neural Networks. Als State-of-the-Art für die Soundevent-Detektion gilt aktuell die Verwendung von CRNNs mit dem Mel-Spektrogramm mit logarithmischer Amplitudenskalierung als Feature-Input[4]. Dabei geht es speziell um den Ansatz von Lim, Park und Han [21] für die Detektion von seltenen Soundevents mit 1D-Convolutions und zwei LSTM-Layern. Die Autoren haben einen durchschnittlichen F_1-Score von 93,1 % erreicht. Mit der Verwendung von Convolutional Recurrent Neural Networks sollen beide Vorteile der Teilnetzwerke, CNN und RNN, kombiniert werden. Zeitlich lokale Features sollen mittels des CNN extrahiert und anschließend in den LSTM-Layern in einen größeren zeitlichen Kontext zueinander gebracht werden.

Transfer Learning. Da Deep Neural Networks eine große Datenmenge erfordern, um Modelle effektiv zu trainieren, ist es häufig sinnvoll Transfer Learning zu nutzen. Dabei wird ein Modell verwendet, welches zuvor ausgiebig auf einem großen Datensatz für eine andere Aufgabe trainiert wurde. Anschließend kann das vortrainierte Modell für die

spezielle Aufgabestellung angepasst werden. Im vortrainierten Modell können Parameter bestimmter Layer durch weiteres Training, dem Finetuning, angepasst werden. Alternativ kann das vortrainierte Modell zur Feature-Extraktion genutzt werden. Beide Ansätze werden in [40] ausführlich für die Bildverarbeitung beschrieben und können ebenfalls auf neuronale Netzwerke für die Audioverarbeitung angewendet werden. Bei der Feature-Extraktion braucht nicht mehr das gesamte Netzwerk trainiert werden. Es werden die Schichten des Klassifizierers gegen einen aufgabenspezifischen Klassifizierer ausgetauscht. Somit wird nur noch der neue Klassifizierer trainiert und die Parameter der anderen Layer bleiben unverändert. Die angelernten grundlegenden Fähigkeiten zum Verständnis von Audiodaten werden wiederverwendet und für die Klassifizierung von ausgewählten Soundevents spezialisiert. Ein solches vortrainiertes neuronales Netz für die SEC ist das in [45] definierte YAMNet.

2.4 Systeme zum Schutz von Fahrradfahrern

Es wurden verschiedene Ansätze zum Schutz von Fahrradfahrern in der Forschung untersucht. In der Literatur wurden drei Hauptansätze gefunden: den Fahrradfahrer durch ein Hilfssystem am Fahrrad unterstützen, den Fahrradfahrer bei Kollision schützen oder die Erkennung von Fahrradfahrern durch Sensoren am Kraftfahrzeug (Kfz) durchführen.

Fahrradfahrer durch ein Hilfssystem am Fahrrad unterstützen. Smaldone u. a. [36] stellen ein System vor, bei welchem das Fahrrad durch ein System für die Audio- und Video-Wahrnehmung von sich von hinten annähernden Fahrzeugen erweitert wird. Die kognitive Last soll für den Radfahrer reduziert werden, wodurch die Sicherheit gesteigert werden soll.

Fahrradfahrer bei Kollision schützen. Das System von Kiang [17] sieht zum einen ein System zur Erkennung von Fußgängern in toten Winkeln auf Basis mehrerer Sensoren vor. Zum anderen gehört zusätzlich der Schutz der Unfallpartner durch extern am Fahrzeug montierte Airbags dazu. Mit diesem Gesamtkonzept soll eine herausragende Wirkung für den Schutz von Straßenteilnehmern geboten sein.

Fahrradfahrer durch ein Hilfssystem am Kraftfahrzeug erkennen. Der Grundgedanke, Fahrradfahrer durch ein Hilfssystem am Kraftfahrzeug zu erkennen, kann gegliedert werden in Lösungen mit passiver und aktiver Präsenzmitteilung. Dabei geht es um die Art, wie der Radfahrer dem System am Kfz seine Präsenz signalisiert. In die Gruppe der passiven Präsenzmitteilung werden Ansätze eingeordnet, die keine Aktion des Radfahrers erfordern und ermöglichen. Mit der vorliegenden Arbeit wurde ein Prototyp-System entwickelt, welches den Fahrradfahrern eine aktive Möglichkeit zur Präsenzsignalisierung bieten kann. Dadurch grenzt es sich von den anderen Ansätzen ab.

Passive Präsenzmitteilung. Lidar-, Kamera-, Radar- und Ultraschall-Sensoren wurden von Gale Bagi u. a. [11] untersucht und eine Kombination aus den letzten drei Sensoren vorgeschlagen. Sie werden in einer dargestellten Anordnung außen am Personenkraftwagen

angebracht. Rangesh und Trivedi [31] stellen ein weiteres System für einen Rundumblick um das Fahrzeug vor. Es basiert auf Lidar- und Kamera-Sensoren. Beide Systeme sollen potenzielle Gefahren erkennen und eine entsprechende Warnung an den Fahrer oder an das Fahrzeug aussenden.

Dieser Grundgedanke wird unter Verwendung verschiedener Sensoren auch in dem Projekt von Steć, Stabernack und Kubach [38] erweitert. Bei diesem Mehrsensorsystem soll nicht nur der Fahrer des Kfz gewarnt werden, sondern ebenso der gefährdete Fahrradfahrer, wenn er im toten Winkel potenziell gefährdet ist.

Aktive Präsenzmitteilung. In dem Projekt ist ebenso eine Umsetzung einer aktiven Präsenzsignalisierung durch den Fahrradfahrer geplant [38]. Aus diesem Ziel ist die vorliegende Arbeit entstanden. Um den Fahrradfahrern eine aktive Möglichkeit zu geben, den Autofahrer zu warnen, ist ein eingebettetes System für die Erkennung von akustischen Warnsignalen von Fahrradfahrern entwickelt worden. Als Warnsignal werden hier Töne von Fahrradklingeln betrachtet, wodurch kein zusätzliches System am Fahrrad erforderlich ist. Die Entwicklung eines eingebetteten Systems bedeutet für diese Arbeit eine aufgabenspezifische Integration von Hardware und Software in ein gemeinsames Gesamtsystem. Der Fokus dieser Arbeit liegt auf der Erstellung eines Prototyps für die Erkennung von Fahrradklingeln in einem fortlaufend aufzunehmenden Audiosignal.

Liss [22] hat die Implementierbarkeit von neuronalen Netzwerken auf einem Mikrocontroller zur Erkennung von Warnsignalen gezeigt. Das System wurde als Proof of Concept für Kopfhörer mit aktiver Geräuschunterdrückung entwickelt. Neben dem Warngeräusch einer Autohupe wurde sekundär das Warnsignal der Fahrradklingel untersucht. Durch die unterschiedlichen technischen Voraussetzungen und der geringen Performanz der Erkennungsmodelle ist das vom Autor vorgeschlagene System jedoch nicht für dieses Projekt anwendbar und ein Vergleich nicht relevant.

Die Recherche hat ergeben, dass kein weiterer Ansatz existiert, bei dem ein Gesamtsystem unter Verwendung von künstlichen neuronalen Netzen zur Erkennung von Fahrradklingel-Sounds untersucht oder umgesetzt wurde.

3 Eingebettetes System zur Soundevent-Detektion

3.1 System-Überblick

Das entwickelte System soll akustische Warnsignale von Fahrradfahrern erkennen und auf solche reagieren können. Da in Deutschland laut Gesetz (§ 64a StVZO) jedes Fahrrad im Straßenverkehr mit einer „helltönigen Glocke“ ausgestattet sein muss, eignet sich die Fokussierung auf die Sounds von Fahrradklingeln. Dadurch kann bei Anwendung des Systems ein ausgestattetes Fahrzeug auf dieselbe Weise gewarnt werden, wie es schon lange für andere schwache Verkehrsteilnehmer üblich ist. Ohne das System ist eine sichere Wahrnehmbarkeit der Fahrradklingel-Töne im Fahrzeug von diversen Faktoren abhängig. Dazu gehören Eigenschaften der Umgebungsgeräusche außerhalb und innerhalb des Fahrzeuges, sowie der Geräusch-Isolation vom Fahrzeug. Das erschwert die Wahrnehmbarkeit durch den Fahrzeugführer oder macht sie sogar unmöglich.

Das aus dieser Arbeit entstandene eingebettete System verfolgt zur Lösung dieses Problems den Ansatz, ein Mikrofon zur Aufnahme der Audiosignale zu verwenden, das Soundevent des Fahrradklingelns darin zu erkennen und anschließend die Information über eine potenzielle Erkennung zurückzugeben. Der Audio-Sensor könnte bei einer Integration in ein Testfahrzeug beispielsweise an dem hinteren rechten Ende des Fahrzeuges platziert werden, um ankommende Radfahrer zu erfassen.

Es wird sich mit der Entwicklung eines ersten Prototyps befasst, der Weiterentwicklungen und Untersuchungen des Ansatzes ermöglicht. Das vollständige, eingebettete System setzt sich aus drei Hauptbestandteilen zusammen: der Hardware, dem Modell zur Erkennung des Soundevents und der Software. Zur Hardware wird auf die verwendeten Geräte eingegangen, die für das eingebettete System verwendet wurden. Zu den betrachteten Erkennungsmodellen gehört die Definition der Architekturen der neuronalen Netze, sowie die Vorgehensweise bei der Erstellung der entsprechenden Modelle. Beide Teilsysteme müssen in einer ausführbaren Anwendung, der Software, miteinander integriert werden. Mit jedem dieser Bestandteile befassen sich die nächsten Abschnitte.

3.2 Hardware

Für die Entwicklung eines Prototyps ist ein Zielsystem mit einem relevanten Sensor erforderlich. Ein Jetson Nano Developer Kit von NVIDIA wurde für das eingebettete System als Zielplattform gewählt. Der Einplatinencomputer bietet vielfältige Einstellungs- und Installationsmöglichkeiten über das entsprechende Betriebssystem JetPack.

Das verwendete Modell besitzt einen Vier-Kern ARM Cortex-A57 MPCore Prozessor, eine NVIDIA Maxwell GPU mit 128 NVIDIA CUDA Kernen, 4 GB Arbeitsspeicher und

mehrere Anschluss-Möglichkeiten [15]. Für dieses Projekt ist insbesondere der 40-Pins Erweiterungs-Header und der microSD Anschluss zu erwähnen. Das Linux-basierte Betriebssystem JetPack 4.6.1 wurde auf eine 32 GB microSD geflasht, wobei der übrige Speicher für zusätzliche Software genutzt wird. Zum Betrieb sind 5 W bis 10 W an elektrischer Leistung erforderlich. Der Jetson Nano kann nach einem ersten Setup sowohl über SSH als auch über die Betriebssystem-integrierte grafische Benutzeroberfläche (GUI) mit angeschlossenen Peripheriegeräten angesteuert werden.

Als Audiogerät wird der Seeed ReSpeaker 2-Mics Pi HAT [34] über den 40-Pins Erweiterungs-Header an das Jetson Nano Board angeschlossen. Dieser basiert auf dem WM8960 Stereo Codec, der für portable digitale Audioanwendungen entwickelt wurde und verschiedene Samplingraten von 8 kHz bis 48 kHz ermöglicht [37]. Auf der Soundkarte von Seeed sind zwei Mikrofone integriert, über die jedoch keine genaueren Herstellerangaben auffindbar waren. Zusätzlich werden die Treiber für den Betrieb des ReSpeakers installiert. Das erforderte eine Einarbeitung in den Quellcode, wodurch ein zuvor bestandenes Kompatibilitätsproblem der Treiber und der aktuellen JetPack-Version gefunden wurde. Nachdem das Problem durch den Hersteller behoben wurde und die korrekte Pinkonfiguration vorgenommen wurde, konnte die Soundkarte erfolgreich ansteuert werden.

Die Verwendung des Jetson Nano Developer Kits mit dem ReSpeaker 2-Mics Pi HAT bietet über das Linux-Betriebssystem eine gewohnte und entwicklungsnahe Umgebung. Das lässt ein einfaches Prototyping und Testen des eingebetteten Systems zu. Als leistungsstarker Einplatinencomputer liegt eine geeignete Hardware für den Betrieb von tiefen neuronalen Netzwerk-Modellen vor.

3.3 Erkennungsmodelle

3.3.1 Architekturen der neuronalen Netzwerke

Künstliche neuronale Netzwerke sind in vielen Aufgaben deutlich den klassischen Verfahren des maschinellen Lernens überlegen [33]. Dabei geht es insbesondere um den Erfolg der DNNs, wie bereits im Kapitel der Grundlagen beschrieben wurde. Im Rahmen dieser Arbeit werden drei Modelle für die Erkennung des Fahrradklingel-Soundevents erstellt, jeweils basierend auf tiefen neuronalen Netzwerken. Es sind zwei Transfer Learning-Modelle und ein Modell mit einem Convolutional Recurrent Neural Network definiert. Implementiert wurden die Architekturen mit TensorFlow [1] in Python. Das Repository[3] zum Quellcode zu den Architekturen der neuronalen Netzwerke wurde in dem Repository zu den Modellen bereitgestellt. Im Folgenden sind die Strukturen der Netze beschrieben und im Anhang unter „Darstellungen der neuronalen Netzwerk-Architekturen“ visualisiert. Zusätzlich zu den Beschreibungen können die Architekturen im referenzierten Repository gefunden werden. Die drei entstehenden Modelle können anschließend trainiert und auf ihre Performanz evaluiert werden. In der Tabelle 3.1 ist die resultierende Parameteranzahl und die erforderliche Speichergröße der neuronalen Netzwerke zusammengefasst.

[3] Quellcode: `https://github.com/ClemensKubach/bicycle-bell-sed-models` (29. Juni 2022)

YAMNet. Das YAMNet wurde für die Soundevent-Klassifizierung von Umgebungsgeräuschen erstellt. Es wurde für die Erkennung von Audio-Events aus 521 Klassen auf dem AudioSet Datensatz [12] trainiert und kann mittels des vortrainierten Modells von [39] für Transfer Learning durch Feature-Extraktion genutzt werden [41]. Die Architektur basiert auf der von Howard u. a. [14] entworfenen MobileNet v1 Architektur. MobileNets wurden ursprünglich für mobile und eingebettete Computer Vision-Anwendungen entwickelt und deren Effektivität in einer Vielzahl von Vision-Aufgaben in [14] demonstriert. Mit Vorverarbeitung der Daten kann genannte Architektur ebenso auf Audiodaten angewendet werden, was mit dem YAMNet realisiert wurde. Als Eingabe wird ein Monokanal-Audio von beliebiger Länge mit einer Samplingrate von 16 kHz in der Wellenform erwartet. Intern wird die Wellenform in ein Log-Mel-Spektrogramm mit 64 Mel-Bins umgewandelt. Es wird in 960 ms Abschnitte mit 50-prozentiger Überlappung geteilt. Der Kern des YAMNets, das MobileNet, wird anschließend jeweils auf solch einen Abschnitt angewendet und ein Ergebnis erzeugt. Somit gibt es abhängig von der Audiolänge N Abschnitte und Ergebnisse. Ausgegeben wird ein Tupel aus N Scores, Embeddings und dem Log-Mel-Spektrogramm. Die Scores enthalten Vorhersagen für jede der 521 Klassen über den jeweiligen Abschnitt. Embeddings umfassen hier die Ausgabe des Layers vor dem Klassifizierer für jeden Abschnitt und können insofern für das Transfer Learning durch Feature-Extraktion genutzt werden. Das YAMNet ist als vortrainiertes Modell mit 3,7 Millionen Parametern das Basismodell von zwei der drei Erkennungsmodelle: YAMNet-Base und YAMNet-Extended.

YAMNet-Base. YAMNet-Base basiert auf dem vortrainierten YAMNet Modell. Es ist in Abbildung A.1 und A.2 visualisiert. Dabei unterstützt die erste Abbildung bei dem Verständnis der im Folgenden beschriebenen Struktur des Modells und die zweite Abbildung enthält die von TensorFlow generierte Darstellung des Layer-Aufbaus. Bei diesem werden die Scores vom Ausgabetupel des YAMNets verwendet. Die Architektur des YAMNet-Base passt dabei lediglich die Eingabe und Ausgabe auf die hier bestehende Aufgabe an. Daher werden nur Layer für die Formatanpassung benötigt. Es soll für den gesamten Audio-Clip nur ein einzelner Wert für das relevante Soundevent ausgegeben werden. Dafür werden keine trainierbaren Parameter hinzugefügt, sodass das Modell ohne Training verwendet werden kann.

Das akzeptierte Eingabeformat bleibt unverändert bei dem vom YAMNet definierten Format. Monokanal-Audiodaten mit einer Samplingrate von 16 kHz werden in der Wellenform als eine endlich-lange eindimensionale Folge von 32 Bits Gleitkommazahlen erwartet. Das vortrainierte Modell wird damit in das YAMNet-Base Modell eingebunden. Die relevanten Scores, bezüglich des Fahrradklingelns, werden aus den 521 Klassen der YAMNet-Ausgabe extrahiert. Anschließend wird die zeitliche Folge von N Scores zu einem einzelnen Wert für das Vorhandensein des Sounds einer Fahrradklingel im Audio-Clip aggregiert. Dazu wird der maximale Score ermittelt und als Ausgabe des Netzwerkes definiert.

YAMNet-Extended. Für das YAMNet-Extended werden die Embeddings des YAMNets für einen Transfer Learning-Ansatz genutzt. Es ist ebenso wie das YAMNet-Base in zwei Darstellungen visualisiert worden: Abbildung A.3 und A.4. Die Eingaben und Ausgaben des Netzwerkes haben das gleiche Format wie das YAMNet-Base. Nach dem eingebetteten

YAMNet Modell werden die Embeddings extrahiert und an ein Klassifizierer-Netz weitergegeben. Dieses setzt sich aus einem LSTM-Layer und einem Fully-connected-Network (FNN) zusammen. Der LSTM-Layer besteht aus 16 Neuronen und dem Hyperbeltangens (*tanh*) als Aktivierungsfunktion. Anschließend folgt ein zeitlich verteiltes FNN mit drei Dense-Layern und zwei Dropout-Layern mit einer Rate von 0,3 zwischen den einzelnen Dense-Layern. Die Dropout-Layer dienen der Reduzierung des Overfittings. Jede Schicht des beschriebenen Netzwerkes wird jeweils auf jedem der N Embeddings angewendet. Die Neuronen-Anzahl wird schrittweise in den aufeinanderfolgenden Dense-Layern von 16, auf 8 und abschließend auf eine einzelne Unit reduziert. Intern wird bei den ersten beiden Dense-Layern eine ReLu-Aktivierungsfunktion und nur im Letzten des Fully-connected-Netzes eine Sigmoidfunktion verwendet. Auf diese Weise wird ein Wahrscheinlichkeitswert zwischen 0 und 1 für jeden der N Abschnitte berechnet. Zur Reduzierung auf einen gemeinsamen Ausgabewert für das gesamte Audiosample werden die N Werte durch Anwendung des arithmetischen Mittels aggregiert. Das YAMNet-Extended Modell hat zusätzlich zu den 3,7 M nicht trainierbaren Parametern des YAMNets noch 67 041 trainierbare Parameter.

Convolutional Recurrent Neural Networks (CRNN). Die Anwendung von Convolutional Neural Networks und Recurrent Neural Networks für die Soundevent-Klassifizierung und -Detektion wurden jeweils in mehreren Arbeiten untersucht, wie in [28], [8] und in[27], [2]. Hybride Architekturen aus beiden Ansätzen haben sich als besonders gute Lösung erwiesen. In mehreren Werken, wie in [6], [44], [48], [5], [3], wurde die Überlegenheit von Convolutional Recurrent Neural Networks für die Soundevent-Detektion herausgearbeitet. Purwins u. a. [29] bezeichnen die Frage der Wahl, welche der drei Ansätze in welcher Situation anzuwenden ist, als offene Forschungsfrage. Von Autoren wird aber ebenfalls erwähnt, dass CRNNs einen Kompromiss zwischen den jeweiligen Vor- und Nachteilen der CNN und RNN Modellen darstellen. Ein in der Literatur häufig referenzierter Wettbewerb für Aufgaben zu Soundevents ist der „IEEE AASP Challenge on Detection and Classification of Acoustic Scenes and Events“-Wettbewerb (DCASE). Aufgrund der Vielzahl von akustischen Signalen und Umgebungsgeräuschen im Straßenverkehr, sowie der häufigen Erfolge von CRNN Architekturen bei solchen Wettbewerben, wie von [21], wird als drittes Modell ein CRNN untersucht.

Das für diese Arbeit erstellte CRNN Modell, dargestellt in Abbildung A.5 und A.4, basiert auf der Grundlage der Architektur von Lim, Park und Han [21]. Zusammengefasst ist der Ansatz von den Autoren: die Audiodaten in ein Spektrogramm umzuwandeln, lokale Informationen aus zeitlich kurzen Frames über das Frequenzspektrum mit 1D-CNNs zu extrahieren, diese mittels eines RNN-LSTM-Netzwerks in zeitliche Beziehung zueinander zu bringen und abschließend eine frameweise Klassifizierung für den Ziel-Sound mit einem FNN zu ermitteln.

Für das in dieser Arbeit behandelte Problem wurden einige Konfigurationen vollzogen. Insbesondere wurden die Eingabe- und Ausgabeformate von dem in [21] vorgeschlagenen Modell angepasst. Es wird auf ein einheitliches Format zu den beiden YAMNet Varianten an den Schnittstellen des CRNN geachtet. Eine eindimensionale Wellenform von Audiodaten beliebiger Länge in 16 kHz wird als Eingabe erwartet und eine Wahrscheinlichkeit zum Vorhandensein des Ziel-Soundevents ausgegeben. Auf diese Weise werden mit dem

Modell verschiedene Konfigurationen in dem Softwaresystem realisierbar. Es besteht aus Layern zur Vorverarbeitung, einem CNN, RNN, FNN und aggregierenden Layern zur Ausgabe.

Der Wellenform-Input wird in ein Log-Mel-Spektrogramm transformiert und hat je nach Länge des Audios die Form: (N Frames, #Mel-Bins). Das Spektrogramm wird um eine zusätzliche Dimension zu (N Frames, #Mel-Bins, 1) erweitert. Ein 1D Convolutional Network wird für die frameweise Analyse der Audiodaten genutzt. Wie Lim, Park und Han [21] erklären, erlaubt das gegenüber einer chunkweisen Analyse eine präzisere Schätzung der Startzeit des Soundevents. Lokale Features werden durch das CNN extrahiert und mit dem RNN in einen zeitlichen Kontext gesetzt.

Der Aufbau des CNN ähnelt dem des in [21] beschriebenen CNNs, aber mit einer Anpassung der Filteranzahl auf die Anzahl von 64 Mel-Bins. Ein Convolutional Netz ist somit aufgebaut aus einem 1D-Convolutional-Layer mit 64 Filtern, einer Kernel-Größe von 32 und einer ReLu-Aktivierungsfunktion. Es folgen eine Batch-Normalisierung, ein 1D-Max-Pooling-Layer mit einer Pool-Größe von $33 = (64 - 32 + 1)$ und ein Dropout-Layer mit einer Dropoutrate von 0,3. Für den Input in das darauffolgende RNN wird eine Reduktion der Dimension durchgeführt. Das CNN wird zeitlich verteilt auf jedem der N Frames ausgeführt. Es wird nur ein einzelnes CNN auf allen Frames angewendet. Dadurch werden Audiodaten von dynamischer Länge akzeptiert. Im Vergleich zur Verwendung einer festen, endlichen Zahl an parallelen CNNs, müssen mit der hier vorgestellten Architektur weniger Parameter trainiert werden. Indem nur ein einzelnes CNN auf allen Frames angewendet und keine erstellt wird, werden in

Wie in [21] beschrieben, werden zwei rückwärts-orientierte LSTM-Layer verwendet, hierbei jedoch der Anzahl an Mel-Bins entsprechend mit nur 64 Units. Eine *tanh*-Aktivierungsfunktion und ein Dropout von 0,3 wird angewendet. Die Ausgabe über 64 Features je Frame wird an das zeitlich verteilte Fully-connected Neural Network weitergeleitet. Ein Dense-Layer mit 64 Neuronen und ReLu-Aktivierungsfunktion wird gefolgt von einer Batch-Normalisierung und einem Dense-Layer mit nur einer Unit und Sigmoid-Aktivierungsfunktion zur Zuordnung einer Wahrscheinlichkeit. Für jeden Frame wird eine Klassifizierung des einzelnen Ziel-Soundevents erzeugt. Die N Wahrscheinlichkeiten werden anschließend durch Anwendung des arithmetischen Mittels für die Ausgabe auf einen einzelnen Wert reduziert.

Mit nur 72 641 trainierbaren Parametern und insgesamt 72 897 Parametern ist das Netzwerk insgesamt deutlich kleiner als die vortrainierten YAMNet Modelle. Das CRNN muss vollständig trainiert werden und basiert nicht auf Transfer Learning-Techniken.

Modell	YAMNet-Base	YAMNet-Extended	CRNN
Parameteranzahl[4]	3.7M + 0	3.7M + 67041	72897
Speichergröße[5]	15 029 KB	13 172 KB	410 KB

Tabelle 3.1: Größen der neuronalen Netzwerke. Die Parameteranzahl der YAMNet-Varianten setzt sich aus den Parametern des vortrainierten YAMNets und der hinzugefügten Layer zusammen.

[4]Die Parameteranzahl für das YAMNet folgt der Angabe aus [45], die übrigen Angaben entsprechen den automatisierten Berechnungen von TensorFlow.

3.3.2 Machine Learning-Prozess

Die in TensorFlow definierten Neural Network-Architekturen aus dem vorhergehenden Abschnitt wurden mit der hier beschriebenen Vorgehensweise zu trainierten, evaluierten und anwendbaren Modellen vervollständigt. Die Implementierungen zu den folgenden Themen können im entsprechenden Repository[6] gefunden werden.

3.3.2.1 Datensatz

Für das überwachte Training und das Testen der entstehenden Modelle wird ein Datensatz von annotierten Audiodaten benötigt. Für diese Arbeit wurde der Freesound FSD50K Datensatz [10] gewählt. Er enthält 51 197 Audio-Clips mit insgesamt über 100 Stunden an Audiomaterial. Die Clips wurden von den Autoren des FSD50K manuell unter Verwendung von 200 Klassen der AudioSet-Ontologie [12] annotiert. Bei der Recherche nach einem geeigneten Datensatz musste die Voraussetzung erfüllt sein, dass annotierte Audiosamples des Ziel-Soundevents enthalten sind. Im FSD50K Datensatz sind 129 Audiobeispiele von Fahrradklingeln enthalten. Die Samples sind aufgeteilt in einen Entwicklungs- und in einen Evaluations-Datensatz (Dev- und Eval-Set). Zu beiden Gruppen gibt es eine Sammlung von WAVE-Dateien und eine CSV-Datei, die jeweils für jeden Dateinamen die zugehörigen Label-Informationen enthält.

Für das Training und die Evaluation der Modelle wird sich auf eine Teilmenge des vollständigen Datensatzes begrenzt. Zum einen kann so einfacher ein ausgeglicheneres Verhältnis zwischen der Menge der Samples des Ziel-Soundevents (positive Klasse) und dessen Komplementärmenge (negative Klasse) hergestellt werden. Zum anderen wird die zu investierende Gesamtzeit des Trainings der Modelle verringert. Es wurden einige der Soundevents betrachtet, die tendenziell im Straßenverkehr häufiger vorkommen können. Aus der Ontologie wurden die folgenden acht Klassen gewählt und deren Samples zur negativen Klasse zugeordnet: „Car“, „Truck“, „Bus“, „Siren“, „Wind“, „Rain“, „Human voice“ und „Traffic noise, roadway noise“. Nach dem Training soll das Modell möglichst nicht auf Geräusche des Straßenverkehrs reagieren, wie dem Hupen von anderen Fahrzeugen, Sirenen von Rettungsfahrzeugen oder Umgebungsgeräuschen.

Der Dev-Datensatz ist nochmals unterteilt in eine Trainings-Untermenge (Train-Subset) und eine Validierungs-Untermenge (Val-Subset). Für das Trainieren des Netzwerkes sind im Train-Subset 59 Samples in der positiven und 5315 Samples in der negativen Klasse. Zur Validierung während des Trainingsprozesses kann das Val-Subset genutzt werden. Es enthält 14 Samples in der positiven und 601 Samples in der negativen Klasse. Für eine separate Evaluation sind nach der Reduzierung auf die erwähnten Soundevents noch 2360 negative Samples und unverändert 56 positive Samples im Eval-Set verfügbar.

Die Gesamtanzahl wurde mit der Auswahl an negativen Events auf 8405 Audio-Clips, und somit auf unter 17 % der ursprünglichen Menge, reduziert. Bei der Betrachtung des Verhältnisses zwischen der Anzahl der positiven und der negativen Samples wird immer noch deutlich, dass der Datensatz stark unbalanciert ist. Die Anzahl von Sounds von Fahrradklingeln ist in nur relativ geringem Maße enthalten. Für das angestrebte Erkennungsmodell ist die Aufgabe, zu entscheiden, ob ein Fahrradklingeln im Audio-Beispiel

[5] Die Speichergröße bezieht sich auf die Inferenz-optimierten Modelle durch TensorFlow Lite, siehe 3.4.2.
[6] Quellcode: `https://github.com/ClemensKubach/bicycle-bell-sed-pipeline` (30. Juni 2022)

vorhanden ist oder nicht. Es muss demnach eine binäre Klassifizierung durchgeführt werden. Die positive Klasse ist die Minderheitsklasse und kann als *ein Fahrradklingeln ist enthalten* formuliert werden. Dem gegenüber steht die negative Klasse als Mehrheitsklasse und wird als *ein Fahrradklingeln ist nicht enthalten* formuliert. Eine Lösung des Klassen-Ungleichgewichts ist Teil des nächsten Abschnittes.

3.3.2.2 Daten-Vorverarbeitung

Die Vorverarbeitung der Daten ist ein wichtiger Schritt für die Trainingsphase und für die Inferenz-Phase. Dabei sind bestimmte Verfahren für beide Phasen relevant, wogegen andere nur für die Vorbereitung des Datensatzes im Trainingsprozess angewendet werden.

Wie zu den Architekturen beschrieben, finden auch Vorverarbeitungsschritte in Layern der neuronalen Netzwerke intern statt. Insbesondere ist die Transformation der Audiodaten von der Wellenform in das Log-Mel-Spektrogramm von zentraler Bedeutung. Im Unterabschnitt 2.1.2 sind die beiden Darstellungstypen in Abbildung 2.1 und Abbildung 2.2 für dasselbe Audio dargestellt. Jedes Geräusch hat eine individuelle Darstellung und ein Muster der Charakteristiken von gleichartigen Geräuschen kann durch neuronale Netze gelernt werden. Für die Umwandlung in ein Spektrogramm wird das zeitdiskrete 16 kHz Monokanal-Audiosignal mittels der Kurzzeit-Fourier-Transformation (engl. Short-Time-Fourier-Tranform – STFT) aus der Wellenform berechnet. Dabei wird auf kleinen Zeitfenstern eine Fourier-Transformation ausgeführt, sodass für jedes dieser T kurzen Zeitfenster das Signal aus der Zeitdomäne in die Frequenzdomäne umgewandelt wird. Damit ist eine Amplitudenverteilung über den F Frequenz-Intervallen (FFT-Bins) für das jeweilige Zeitfenster gegeben. Im Spektrogramm mit den Maßen (T, F) liegt für jedes Zeit-Frequenz-Paar ein Amplitudenwert vor. Außerdem werden die Frequenzen des Spektrogramms in die Mel-Skala überführt, um der menschlichen akustischen Wahrnehmung zu ähneln, wie bereits im Unterabschnitt 2.1.2 beschrieben wurde. Das Mel-Spektrogramm wird mit 64 Mel-Bins über den Frequenzbereich von 125 Hz bis 7500 Hz erstellt und anschließend eine logarithmische Skalierung der Amplitudenwerte durchgeführt. Bei dieser wird ein Offset zur Stabilisierung addiert, um die Berechnung von $log(0)$ zu verhindern. Die beschriebene Transformation zum Log-Mel-Spektrogramm wurde in [45] für das YAMNet implementiert und wird für die Eingaben aller drei Modelle dieser Arbeit angewendet.

Vorverarbeitung in der Trainingsphase. Folgende Schritte erfolgen in der Trainingsphase zur Datenvorbehandlung:

1. Formatierung des Datensatzes: Die Wellenform der Audiodaten stellt die Feature-Matrix dar, mit jeweils einem zugehörigen Wahrheitswert zum Vorhandensein des Ziel-Soundevents im Zielvektor.
2. Resampling der Audiosignale: Diese können in den WAVE-Dateien in unterschiedlichen Samplingraten vorliegen. Erforderlich ist eine für das Modell vorgesehene Samplingrate.
3. Daten-Augmentation: Diese Technik wird gesondert im nächsten Absatz 3.3.2.2 behandelt.

4. Shuffling der Samples: Die Daten sollen beim Training in möglichst zufälliger Reihenfolge vorkommen und werden dafür vermischt.
5. Batching der Samples: Die Audiosamples werden zu Stapeln einer festen Größe verbunden. Beim Training von neuronalen Netzen wird in einer Trainingsiteration typischerweise eine Batch von Daten verwendet, um die Hardware-Ressourcen möglichst vollständig zu nutzen und das Training zu beschleunigen.

Da die Verteilung der Klassen im Datensatz sehr stark unbalanciert ist, benötigt es entsprechende Lösungen. Oversampling und Undersampling sind zwei Arten von Techniken zur Angleichung der Klassenverteilung. Bei den Methoden des Undersamplings wird die Menge von Samples der Mehrheitsklasse reduziert, bei denen des Oversamplings wird die Anzahl an Elementen der Minderheitsklasse um neue Samples erweitert. Häufig werden balancierte Datensätze für Klassifizierungsprobleme betrachtet. Wang u. a. [43] und [35] stimmen zu, dass ein Klassen-Ungleichgewicht die Performanz eines Klassifizierers stark beeinflussen kann und betrachtet werden muss.

Undersampling wurde bereits durch die Eingrenzung der negativen Samples des Datensatzes auf bestimmte Klassen durchgeführt. Bei weiterem Undersampling bis zu einem ausgeglichenen Verhältnis zwischen den beiden Klassen, wäre der übrige Datensatz deutlich zu klein. Das Trainieren von Deep Neural Networks erfordert möglichst viele Daten. In dem Train-Subset sind nur 59 positive Samples vorhanden, diese Anzahl wird für ein effektives Training mittels Oversampling erhöht. Ein naheliegender Ansatz ist es, diese Samples zu vervielfachen. Für eine höhere Variation der Fahrradklingel-Audiosamples ist jedoch die Anwendung von Daten-Augmentation hilfreich.

Daten-Augmentation. Augmentation kann als Oversampling-Methode betrachtet werden, welche aus existierenden Daten neue künstliche Daten generiert. Wie Mesaros u. a. [25] beschreiben, können dadurch mehr Trainingsdaten der Minderheitsklasse erzeugt und ebenso die Diversität im Datensatz erhöht werden. Die Audiosamples müssen dabei so verändert werden, dass sie noch der ursprünglichen Kategorie charakteristisch zuordenbar sind.

Die künstlichen Veränderungen der Samples werden zu unterschiedlichen Anteilen auf beide Klassen angewendet. In der Implementierung werden zuerst die Samples der Minderheitsklasse um einen Faktor vervielfacht, sodass der augmentierte Datensatz möglichst balanciert ist. Anschließend wird für jedes Sample je nach Klassenzugehörigkeit die eigentliche synthetische Veränderung der Audiodaten vorgenommen. Ein Sample der Minderheitsklasse wird mit einer Wahrscheinlichkeit von 90 Prozent variiert und eines der Mehrheitsklasse nur zu 10 Prozent. Damit soll die Wahrscheinlichkeit gesenkt werden, dass entstehende Augmentation-Muster gelernt werden.

Soll ein Audiosample verändert werden, wird die Python-Bibliothek Audiomentations [16] zur Augmentation von Audiodaten eingesetzt. Nacheinander werden die folgenden Manipulations-Schritte zu einer Wahrscheinlichkeit von jeweils 80 Prozent auf der Wellenform der Audiosignale ausgeführt:

1. Veränderung des Pitches,
2. Streckung und Stauchung des Signales zu verschiedenen Raten,
3. Verdeckung von zufälligen Zeiten,

4. Verdeckung von zufälligen Frequenzen,
5. Hinzufügung von Gauß'schem Rauschen mit Variabilität vom Signal-Rausch-Verhältnis von 5 dB bis 40 dB (engl. Signal to Noise Ratio – SNR).

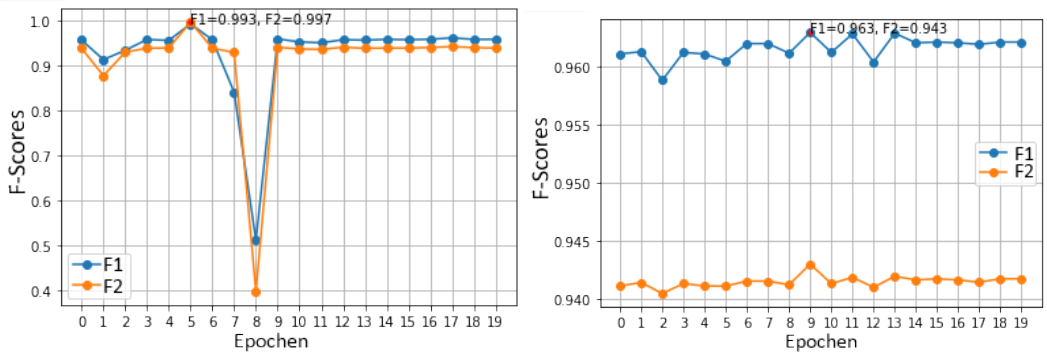
Auf diese Weise konnte die Klassenverteilung innerhalb der jeweiligen Teil-Datensätze Train, Val und Eval balanciert werden. Zugleich wurden neue Fahrradklingel-Sounds künstlich erzeugt. Die Robustheit von Modellen wird durch die zusätzliche akustische Variabilität im Trainingsprozess gesteigert [25].

3.3.2.3 Modell-Vorbereitung und -Training

Die bereits definierten Modelle YAMNet-Base, YAMNet-Extended und das CRNN werden für die Erkennung von Fahrradklingel-Sounds trainiert. Wie Lim, Park und Han [21] vorgeschlagen haben, wird die binäre Kreuzentropie für die binäre Klassifizierungsaufgabe als Verlustfunktion (engl. Loss) verwendet. Zur Minimierung der Verlustfunktion wird für jedes der Modelle der Adam-Optimierer [18] verwendet, im Gegensatz zu [21], hier mit einer unveränderten Lernrate von 0,001. Das CRNN und das YAMNet-Extended werden über 20 Epochen trainiert. Eine Epoche umfasst hierbei eine vollständige Iteration über alle Batches der Trainingsdaten und eine anschließende Validierung der Modellperformanz auf den Validierungsdaten. Das YAMNet-Base erfordert kein Training, da dem YAMNet keine zusätzlichen Parameter hinzugefügt wurden.

Zur Validierung des Trainingsfortschrittes werden nach jeder Epoche verschiedene Metriken genutzt und die Modelle entsprechend vor dem Training kompiliert. Dazu zählen die Genauigkeit (engl. Precision) P, die Trefferquote (engl. Recall) R und der F_1-Score. In der Literatur werden häufig Bewertungen einer Klassifikation auf Basis dieser Metriken vorgenommen, wie auch in [21] und [6]. Die Berechnungsvorschriften zu den Metriken werden im nächsten Abschnitt (3.3.2.4) definiert. F-Scores kombinieren die Precision und den Recall auf Basis des gewichteten harmonischen Mittels. Die allgemeine Gleichung 3.3 des F-Scores wird dabei mit einem α definiert, dessen Wert die Gewichtung ausdrückt. Im F_2-Score wird der Recall mit $\alpha = 2$ stärker als die Precision gewichtet. Dagegen bildet der F_1-Score ein Gleichgewicht zwischen den beiden Maßen mit $\alpha = 1$. Für das Klassifizierungsproblem bedeutet eine höhere Gewichtung des Recalls mit der F_2 Metrik eine höhere Wichtigkeit der Minimierung von unerkannten Fahrradklingel-Signalen im Vergleich zu der Minimierung falscher Erkennungen als Fahrradklingel. Ein Modell wird nach dem Durchlauf einer Epoche auf dem Val-Subset validiert und die Modell-Version mit dem höchsten F_1-Score gespeichert.

In Abbildung 3.1 sind die Trainingsverläufe von CRNN und YAMNet-Extended dargestellt. Um implizit eine Aussage über das Verhältnis von P und R zu haben, wird in den Abbildungen auch der F_2-Score betrachtet. Beide Modelle erreichen hohe Ergebnisse von über 0,9 für den F_1- und F_2-Score auf den Validierungsdaten, wobei 1,0 ein optimales Ergebnis ist. In der sechsten Epoche erreicht das CRNN, mit einem F_1-Score von 0,993 und einem F_2-Score von 0,997, ein höheres Ergebnis als das YAMNet-Extended Modell, mit einem maximalen F_1/F_2-Score von 0,963/0,943 nach der zehnten Epoche. Das YAMNet-Extended erkennt im Vergleich zum CRNN auf dem Val-Subset leicht mehr der Fahrradklingeln nicht als solche, was auch in den Abständen zwischen den Maximalwerten der beiden F-Scores zuerkennen ist. Nach dem Trainingsprozess wird auf einem



(a) CRNN Modell.

(b) YAMNet-Extended Modell.

Abbildung 3.1: Verlauf der *F*-Scores auf den Validierungsdaten über die Epochen in der Trainingsphase.

größeren, getrennten Datensatz eine ausführliche Evaluation der resultierenden Modelle vorgenommen und im nächsten Abschnitt 3.3.2.4 beschrieben.

Zur Beschleunigung des Trainingsprozesses wurde das Training auf einer Instanz von Google Colab[7] durchgeführt. Google Colab erlaubt, Python-Code im Internet-Browser zu schreiben und auf Cloud-Servern von Google auszuführen. Die virtuelle Maschine war ausgestattet mit einer NVIDIA Tesla P100 GPU und 27 GB Arbeitsspeicher.

Gespeichert wurden die Modelle im TensorFlow SavedModel-Format. Es beinhaltet alle notwendigen Modell-Informationen für die Inferenz, sowie für ein mögliches Fortfahren mit dem Trainingsprozess. Für jedes Modell wird eine persistente Modellinstanz erstellt, die für die spätere Bereitstellung in der Software genutzt wird.

3.3.2.4 Evaluation

Die Evaluation des YAMNet-Base, YAMNet-Extended und CRNN Modells wird auf Hold-out-Testdaten vorgenommen. Hold-out-Daten sind hierbei Audio-Clips, die vom Modell noch nicht während der Trainingsphase gesehen wurden. Dazu wird das vorbereitete Eval-Subset von der ausgewählten Teilmenge des FSD50K Datensatzes genutzt. Es wird in zwei Varianten erzeugt: balanciert und unbalanciert.

Für den balancierten Eval-Datensatz wurde Oversampling mit Daten-Augmentation durchgeführt, wie im Abschnitt der Daten-Vorverarbeitung beschrieben wurde. In diesem Fall gleicht der Anteil von Audio-Clips mit Fahrradklingel-Sound dem Anteil ohne Ziel-Soundevent. Für den unbalancierten Eval-Datensatz wurde kein Oversampling eingesetzt, sodass die Anzahl von Samples der positiven Klasse unverändert bei 56 bleibt.

Um die Performanz der Modelle zu messen, wird die F_1 Metrik verwendet, welche bereits im Training in Abschnitt 3.3.2.3 zur Validierung nach einer jeden Epoche verwendet wurde. Der F_1-Score wird aus der allgemeinen Berechnungsvorschrift der F Metrik in Gleichung 3.3 mit $\alpha = 1$ definiert. Demnach wird der F_1-Score mittels der Gleichung 3.4 berechnet. Precision P und Recall R sind dabei mit den Gleichungen 3.1 und 3.2 definiert. Es sei TP die Anzahl der korrekt positiven Klassifikationen (engl. True Positives), FP die

[7]Colab: `https://colab.research.google.com/` (30. Juni 2022)

Anzahl an Samples, die falsch als positiv klassifiziert wurden (engl. False Positives), und FN die Anzahl der Samples mit Fahrradklingel-Sound, die aber nicht als solche erkannt wurden (engl. False Negatives).

$$P = \frac{TP}{TP + FP} \tag{3.1}$$

$$R = \frac{TP}{TP + FN} \tag{3.2}$$

$$F_\alpha = (1 + \alpha^2)\frac{P * R}{\alpha^2 * P + R} \tag{3.3}$$

$$F_1 = 2\frac{P * R}{P + R} \tag{3.4}$$

Die Werte für TP, FP und FN sind jeweils natürliche Zahlen, wobei die Häufigkeit der falschen Klassifikationen möglichst gering sein sollte. Für die P, R und F Metriken ergeben sich Werte zwischen 0 und 1, wobei jeweils ein höherer Wert besser ist als ein niedriger. Bei der Evaluation wird hier das Minimieren der False Positives und das Minimieren der False Negatives als gleich wichtig verstanden und somit insbesondere der F_1-Score betrachtet. Das ist damit zu begründen, dass beide Fälle in der Praxis negative Auswirkungen haben, wie auch in [38] beschrieben wurde. Findet eine Erkennung der Fahrradklingel-Sounds in einer bestimmten Gefahrensituation zwischen Kraftfahrzeug und Fahrradfahrer nicht statt, kann ein Unfall durch das System nicht verhindert werden. Warnt das System den Kraftfahrer aber zu häufig durch fälschliche Einordnung von Umgebungsgeräuschen als Fahrradklingeln, steigt die Gefahr einer Ignorierung der Warnung in einem Ernstfall.

Datensatz	Modell	TP	FP	FN	P	R
Eval balanciert	YAMNet-Base	1118	1	1290	0,999	0,464
	YAMNet-Extended	2064	9	344	0,996	0,857
	CRNN	2150	44	258	0,980	0,893
Eval unbalanciert	YAMNet-Base	26	1	30	0,963	0,464
	YAMNet-Extended	47	9	8	0,839	0,855
	CRNN	50	44	5	0,532	0,909

Tabelle 3.2: Ergebnisse der Evaluation der vorgestellten Modelle zur binären Klassifizierung von Fahrradklingel-Sounds auf den balancierten und den unbalancierten Evaluationsdaten.[8]

In der Tabelle 3.2 sind die Ergebnisse der Evaluation der drei Modelle auf den beiden Eval-Subsets mit den jeweiligen Werten für TP, FP, FN, P und R aufgeschlüsselt. Wie in der Tabelle zu erkennen ist, sind die meisten Samples der positiven Klasse vom CRNN und vom YAMNet-Extended korrekt erkannt worden. Da das auch der Fall für das unbalancierte Eval-Subset ist, kann geschlussfolgert werden, dass bei dem Training der Modelle zumindest nicht ausschließlich die Erkennung von Daten-Augmentation-Mustern gelernt wurde.

Die Werte der Precision auf dem unbalancierten Datensatz sind ohne Berücksichtigung der Gesamtanzahl von 2360 negativen Samples nur wenig aussagekräftig. Durch die sehr

[8] Es ist zu beachten, dass die Evaluation mit einer Batch-Size von 32 durchgeführt wurde. Dadurch kann es vorkommen, dass je nach Gesamtanzahl bis zu 31 Samples vom jeweiligen Eval-Subset nicht zur Evaluation genutzt wurden. Beispielsweise ist daher für das CRNN im unbalancierten Eval-Subset $TP + FN = 55$ und nicht die erwartete Gesamtanzahl an positiven Samples von 56.

ungleiche Klassenverteilung haben hierbei schon wenige falsche Erkennungen als Klingel-Sound (FP) große Auswirkungen auf die Precision P. Somit ist für den unbalancierten Eval-Subset eine zusätzliche Betrachtung des F_1-Scores nicht sinnvoll.

Eval balanciert	YAMNet-Base	YAMNet-Extended	CRNN
F_1-Score	0,634	0,921	0,934

Tabelle 3.3: Performanz der vorgestellten Modelle zur binären Klassifizierung von Fahrradklingel-Sounds bei der Evaluation auf dem balancierten Evaluations-Datensatz, gemessen mit der F_1 Metrik.

Die F_1-Scores der Evaluation auf der balancierten Variante sind in der Tabelle 3.3 zusammengefasst. Das CRNN ist mit $F_1 = 0{,}934$ knapp besser als das YAMNet-Extended mit $F_1 = 0{,}921$. Das YAMNet hat hier zwar eine leicht bessere Precision, denn es werden nur 9 statt 44 negative Samples als positiv klassifiziert, aber das CRNN bietet einen leicht größeren Vorteil in den Recall-Werten. Das YAMNet-Base Modell schnitt insgesamt am schlechtesten von den drei Modellen ab. Es erreichte auf dem Eval-Subset nur einen F_1-Score von rund 0,63, was mit der hohen Zahl an falschen negativen Klassifikationen zu begründen ist.

Im Vergleich zu dem originalen CRNN von Lim, Park und Han [21] konnte in dieser Arbeit ein ähnlich gutes Ergebnis erreicht werden. Die Autoren ermittelten einen Durchschnittswert von 0,931 für den F_1-Score der drei untersuchten Soundevents „baby crying“, „glass breaking“ und „gunshot“. Bei deren Evaluation wurde eine maximale Verzögerung der Startzeit von 500 ms für eine korrekte Klassifizierung vorausgesetzt. Dieser zeitliche Aspekt wurde hier nicht als Zusatz in den F_1-Score einbezogen. Ein Vergleich ist trotzdem sinnvoll, da die CRNNs beide eine ähnliche Architektur mit frameweiser Verarbeitung haben. Somit bleibt der von Lim, Park und Han [21] beschriebene Vorteil einer präzisen Startzeit-Erkennung erhalten. Der F_1-Score des hier vorgestellten CRNNs ist mit 0,003 leicht über dem Ergebnis der Autoren. Es wurde gezeigt, dass die in dieser Arbeit entwickelten Modelle YAMNet-Extended und CRNN vielversprechende Lösungen für die Erkennung von Fahrradklingel-Sounds sind und mit state-of-the-art Lösungen mithalten können.

3.4 Software

Die Implementierung der in diesem Abschnitt beschriebenen Software wurde in Python realisiert und ist im entsprechenden Repository[9] unter dem Namen „Bicycle Bell Sound Event Detection System“ zu finden.

3.4.1 Anforderungen

Um die trainierten Erkennungsmodelle auf der gegebenen Hardware für die Live-Audio-Erkennung einzusetzen, benötigt das eingebettete System eine Komponente zur Integrati-

[9] Quellcode: `https://github.com/ClemensKubach/bicycle-bell-sed-software` (1. Juli 2022)

on der Erkennungsmodelle auf dem Zielgerät. Auf der Zielplattform soll ein Programm installiert werden, welches zur Ausübung der Zielaufgabe kontinuierlich läuft. Dazu wurde eine integrierende Komponente entwickelt, die als Software bezeichnet wird und mit welcher sich Abschnitt 3.4 befasst.

Im Abschnitt 3.3 wurden neuronale Netzwerke definiert und für eine Soundevent-Klassifikation trainiert. Es benötigt für die Aufgabe der Soundevent-Detektion weitere Prozesse, um eine zusätzliche, zeitliche Information zu gewinnen. Um zeitliche Angaben zum Auftreten des Soundevents zu liefern, kann eine Nachverarbeitung angewendet werden, wie sie von Lim, Park und Han [21] beschrieben wurde. In dieser Arbeit wird das Konzept einer fortlaufenden Ausführung der Klassifikationsmodelle auf verschiebbaren Bereichen (engl. Sliding Windows) der Audiodaten und anschließender Durchschnittsbildung der berechneten Wahrscheinlichkeiten aufgegriffen.

Die Software muss nach einer Installation auf dem Zielgerät, den in Echtzeit aufgenommenen Audiostream mit den Modellen verarbeiten. Anschließend sollen in zeitlich hoher Auflösung Informationen über das Vorhandensein eines Fahrradklingelns ausgegeben werden. Dadurch soll das Erkennen von Fahrradklingeln ebenso in Echtzeit erreicht werden. Die Software soll auf der beschriebenen Hardware (3.2) lauffähig sein und weiteres Prototyping ermöglichen. Dazu sollte sie allgemein für diverse Modell-Anpassungen und einfache Weiterentwicklung konzipiert sein.

3.4.2 Eigenschaften und Funktionalitäten

Aus vorgestellten Anforderungen ist die Software „Bicycle Bell Sound Event Detection System“ entstanden: ein Soundevent-Detektionssystem (engl. Sound Event Detection System – SEDS) von Fahrradklingeln (engl. Bicycle Bell). Zu dem Softwaresystem wurde ein Kommandozeilen-Tool als Schnittstelle der Ein- und Ausgabe zwischen Mensch und Maschine (engl. Command-Line-Interface – CLI) geschaffen, welche die Software über die Kommandozeile ausführbar macht. Daher wird die Software ebenso als Bicycle-Bell-Seds-Cli bezeichnet.

Die Software wurde in der Programmiersprache Python implementiert, da es insbesondere für das Machine Learning und Deep Learning weit verbreitet ist. Auf diese Weise wird die Integration der erstellten Erkennungsmodelle, deren Umsetzung ebenfalls in Python vorgenommen wurde, erleichtert. Insbesondere basieren die neuronalen Netzwerk-Modelle auf dem TensorFlow Framework. Die Software bietet eine Unterstützung für ARM64 und x86 Architekturen, TensorFlow 2.6 und höher, sowie Python Versionen ab 3.6, getestet bis Python 3.9. Damit ist das System für eine Vielzahl von Plattformen verfügbar.

Die Software ist als CLI entwickelt, um den Betrieb des Programmes auch ohne zusätzliche Peripheriegeräte sicherzustellen und zu erleichtern. Sie kann leicht um neue Funktionen erweitert und für neue Anforderungen abgewandelt werden.

Funktionsgruppen. Es gibt insgesamt vier Funktionsgruppen, die durch Spezifikation des entsprechenden Kommandos gewählt werden können. Dazu gehört insbesondere die Hauptfunktion `run` zur Ausführung eines Vorgangs zur Soundevent-Detektion. Die anderen drei Funktionalitäten sind Tools, dazu zählen:

- `conversion` für eine nachgehende Umwandlung von Aufnahmen des Audiostreams eines vorherigen SED-Prozesses in WAVE-Dateien,
- `devices` für die Durchführung von Tests der angeschlossenen Audiogeräte und deren Konfiguration und
- `resources` für das Lokalisieren des Installationsspeicherortes des Ressourcenordners der Software. Er beinhaltet unter anderem die vorinstallierten Erkennungsmodelle, Logdateien und Aufnahmen, die für Untersuchungen der Ergebnisse nützlich sein können.

Modi des Detektionssystems. Das System umfasst in der Hauptfunktion `run` die zwei Modi `production` und `evaluation`. Beide Modi bieten Funktionalitäten wie das Logging der Ausgaben und die persistenten Aufzeichnung der Audiodaten. In einer eigenen Callback-Funktion kann definiert werden, welche Daten in welcher Form nach einer jeden Vorhersage ausgegeben werden sollen. Entsprechende Konfigurationen können bei der Ausführung der Hauptfunktion `run` spezifiziert werden.

Produktionsmodus. Der Produktionsmodus `production` realisiert die Lösung der in den Anforderungen beschriebenen Aspekte zur Soundevent-Detektion. Die aufgenommenen Audiodaten werden in Echtzeit vom System verarbeitet und eine Ausgabe wird erzeugt. Eine Erkennung von Fahrradklingel-Sounds wird somit auf dem Live-Audiostream ausgeführt. Informationen über die Entscheidung der Vorhersage und die zugrunde liegende Wahrscheinlichkeit, sowie über die Verzögerungen in der Verarbeitung im System, werden auf dem Konsolenfenster und nach Wunsch in Logdateien ausgegeben.

Evaluationsmodus. Ergänzend zu dem Hauptmodus `production` gibt es den Modus der Evaluation, `evaluation`. Er bietet eine Möglichkeit zur Untersuchung des Zusammenspiels von Mikrofon und Softwaresystem. Mithilfe einer Testaudiodatei im WAVE-Format und einer CSV-Datei mit zugehörigen Labels, zu den Zeiten des Vorkommens des Ziel-Soundevents, kann das Zusammenspiel von Mikrofon und Erkennungsmodell live überprüft werden. Dazu wird der Testaudio-Clip über einen angeschlossenen Lautsprecher abgespielt und über das Mikrofon zusammen mit den vorherrschenden Umgebungsgeräuschen aufgenommen. Fortlaufend werden die Live-Aufnahmen und die aus der Datei eingelesenen Audiodaten durch das gewählte Erkennungsmodell verarbeitet. Die Ergebnisse werden zusammen mit den jeweiligen Ground-Truth-Daten zurückgegeben. Diese ergänzende Funktionalität ermöglicht verschiedene Konfigurationen von unterschiedlichen Modellen und Mikrofonen und wertet auf diese Weise den Nutzen des Systems für das Prototyping weiter auf.

Modellintegration. Die im Rahmen dieser Arbeit entwickelten Modelle aus Abschnitt 3.3 für die Erkennung von Fahrradklingel-Sounds werden bei der Installation der Software mitinstalliert. Das YAMNet-Base, YAMNet-Extended und CRNN können in der Software direkt für die Detektion der Warnsignale von Fahrradfahrern ausgewählt werden. Ohne zusätzliche Programmierung beschränkt sich die Austauschbarkeit aktuell auf neuronale

Netzwerke mit einem gleichen Eingabe- und Ausgabeformat wie dem der drei beschriebenen Modelle. Da deren Format nur wenige Einschränkungen macht, kann das Softwaresystem ohne Weiteres mit einer Vielzahl weiterer Modelle betrieben werden. Somit lässt sich das System sogar für andere Soundevents einsetzen, wenn ein dementsprechend trainiertes Erkennungsmodell existiert. Die Software wurde so entworfen, dass auch weitere Formate durch einfache Erweiterungen im Programmcode integriert werden können. Vor- und Nachverarbeitung können unabhängig vom restlichen Programmcode auf die Modell-Schnittstellen angepasst werden. Insgesamt wurde auf einen hohen Grad der Modularisierung gelegt, um die Weiterentwicklung des Softwaresystems zu vereinfachen.

Zeitliche Performanz. Für schnelle Ergebnisse bei der Inferenz werden die Modelle bei dem Programmstart in ein Inferenz-Format konvertiert. Dadurch sollen Vorhersagen mit möglichst geringen Verzögerungen durchgeführt werden. Die Optimierung wird individuell für das jeweilige neuronale Netzwerk und für die spezielle Hardware des Zielgerätes vorgenommen. In der Software integriert sind aktuell die zwei Optimierungslösungen von TensorFlow Lite (TFLite) und TensorFlow-TensorRT (TF-TRT). Sie profitieren von festen Eingabeformaten der Daten, weshalb eine feste Kontextgröße von Audiodaten bei einer Ausführung der Software festzulegen ist.

Mit der Konvertierung des CRNN Modells in ein TFLite Inferenz-Modell, können Vorhersagen auf dem Jetson Nano über 1000 ms an Audiodaten innerhalb von etwa 32 ms verarbeitet werden. Bei Einbeziehung der Pufferungsdauer werden im Höchstfall insgesamt circa 61 ms benötigt, für den Fall des ältesten einzelnen Samples. Im besten Fall, für das aktuellste Sample im Buffer, entsteht keine zusätzliche Pufferungszeit. Hierbei werden lediglich die circa 32 ms der Verarbeitungsdauer benötigt. Für YAMNet-Extended und YAMNet-Base erfordert es etwa 54 ms bis 106 ms und 58 ms bis 114 ms für die Verarbeitung und Pufferung. Die unterschiedlichen Laufzeiten sind durch die unterschiedlichen Verarbeitungsoperationen und Größen der Architekturen zu begründen. Eine Echtzeitverarbeitung ist bei allen drei Modellen gegeben, da in gleicher Zeit vom System mehr Audiosamples verarbeitet werden können, als erzeugt werden. Der Buffer kann sich nicht fortlaufend vergrößern und infolgedessen kann die Zeit bis zum Erhalt der Detektionsergebnisse nicht wachsen.

Als zusätzliche Maßnahme zur Einhaltung der Echtzeitbedingung des Systems wurde eine weitere Funktionalität integriert. Sollte es durch spontane Systemauslastung oder andere kurzzeitige Umstände zu einer großen Ansammlung von Audiodaten im Buffer kommen, werden alte Daten ignoriert und mit den aktuellen fortgefahren. Dadurch wird eine dauerhafte Verlängerung der Pufferungsdauer als Folge einer kurzzeitigen Verzögerung ausgeschlossen.

Detektions-Performanz. YAMNet-Extended und das CRNN sind entsprechend der Klassifikations-Performanz vielversprechend für den Einsatz im System, wie in 3.3.2.4 gezeigt wurde. Erste Tests mit digital abgespielten Fahrradklingeltönen und mit einer vorliegenden Fahrradklingel haben gezeigt, dass beide Modelle integriert im System ebenfalls gut auf die Signale reagieren. Dabei wurde herausgefunden, dass Erkennungen des Ziel-Sounds in Situationen mit geringen Signal-Rausch-Verhältnissen schwieriger sind. Eine Anpassung dieser Eigenschaft kann durch Training mit weiteren Daten mit gerin-

geren SNR vorgenommen werden. Außerdem kann die Sensibilität der Detektion durch die Einstellung eines Schwellenwerts (engl. Threshold) im Softwaresystem für die bereits bestehenden Modelle beeinflusst werden.

Eine ausführliche empirische Evaluation des Gesamtsystems bezüglich der Verzögerung der Detektion ist nicht Bestandteil dieser Arbeit. Wie im Absatz 3.4.4 gezeigt wird, können Detektionsergebnisse einer Ausführung mithilfe der Konsolenausgaben oder Logdateien beobachtet werden. Insbesondere mit dem Modus `evaluation` wurden bereits automatisierte Möglichkeiten zur Auswertung des Softwaresystems und gewählten Mikrofons geschaffen.

3.4.3 Komponenten und Funktionsweise

Die Bicycle-Bell-Seds-Cli ist in zwei Varianten ausführbar: in der Seds-Cli und in der JN-Seds-Cli. Von zentraler Bedeutung ist die Seds-Cli zur Soundevent-Detektion. Auf dieser basierend wurde die JN-Seds-Cli für die Anpassung auf das Ziel des Gesamtsystems aufgesetzt, dem Erkennen von Fahrradklingel-Tönen auf dem Jetson Nano (JN) mit dem Seeed ReSpeaker 2-Mics Pi HAT als Mikrofon. In dieser Variante wurden einige Voreinstellungen der veränderbaren Parameter vorgenommen.

Die Seds-Cli ist der eigentliche Hauptbestandteil des Softwaresystems. Mit den Erkennungsmodellen wird die Soundevent-Detektion ermöglicht und über eine CLI steuerbar gemacht. Die trainierten Modelle zur Erkennung von dem Soundevent des Fahrradklingelns sind dabei in einem Ressourcen-Verzeichnis enthalten und werden mit der Installation direkt bereitgestellt. Für andere Soundevents ist die Seds-Cli ebenfalls problemlos einsetzbar, sofern ein entsprechendes Erkennungsmodell trainiert und auf dem Zielgerät bereitgestellt wird. Die CLI ermöglicht Einstellungen für die Anpassung an das gegebene Zielsystem, welche in der JN-Seds-Cli für die Hardware des Gesamtsystems bereits vorgenommen wurde. Außerdem sind weitere Konfigurationen bei der Initiierung der Funktionsausführung durch Parametrisierung möglich.

Es wurde sehr darauf geachtet, die Logik, Daten und Funktionalitäten des Systems getrennt von der I/O-Schnittstelle zu entwickeln. Auf diese Weise kann in Zukunft auf dem System aufbauend auch eine grafische Benutzeroberfläche (engl. Graphical User Interface – GUI) zur Steuerung und Auswertung entwickelt werden. Ebenso wäre eine Anbindung des Softwaresystems an ein Gesamtsystem aus mehreren Sensoren zur Erkennung von Fahrradfahrern, wie das von Steć, Stabernack und Kubach [38], über Netzwerkprotokolle der Maschine-zu-Maschine-Kommunikation denkbar. Für eine individuelle Ergebnisausgabe kann die bereits integrierte Möglichkeit einer konfigurierbaren Callback-Funktion genutzt werden. Das kann das Prototyping vereinfachen.

Die Software wurde weitgehend typisiert und objektorientiert implementiert, da während der Entwicklung ein besonderer Fokus auf die Modularisierung und Option einer einfachen Weiterentwicklung gelegt wurde. Um die Funktionsweise der Seds-Cli zu erklären, benötigt es die Betrachtung der Hauptkomponenten. Die zentrale Struktur des Kernsystems mit den Komponenten Receiver, Buffer und Predictor sind in der Abbildung 3.2 visualisiert.

Receiver. Der Receiver kümmert sich um das Erhalten der Live-Audiodaten. Dazu empfängt er die Samples mit einer Samplingrate von 16 000 Samples pro Sekunde und als

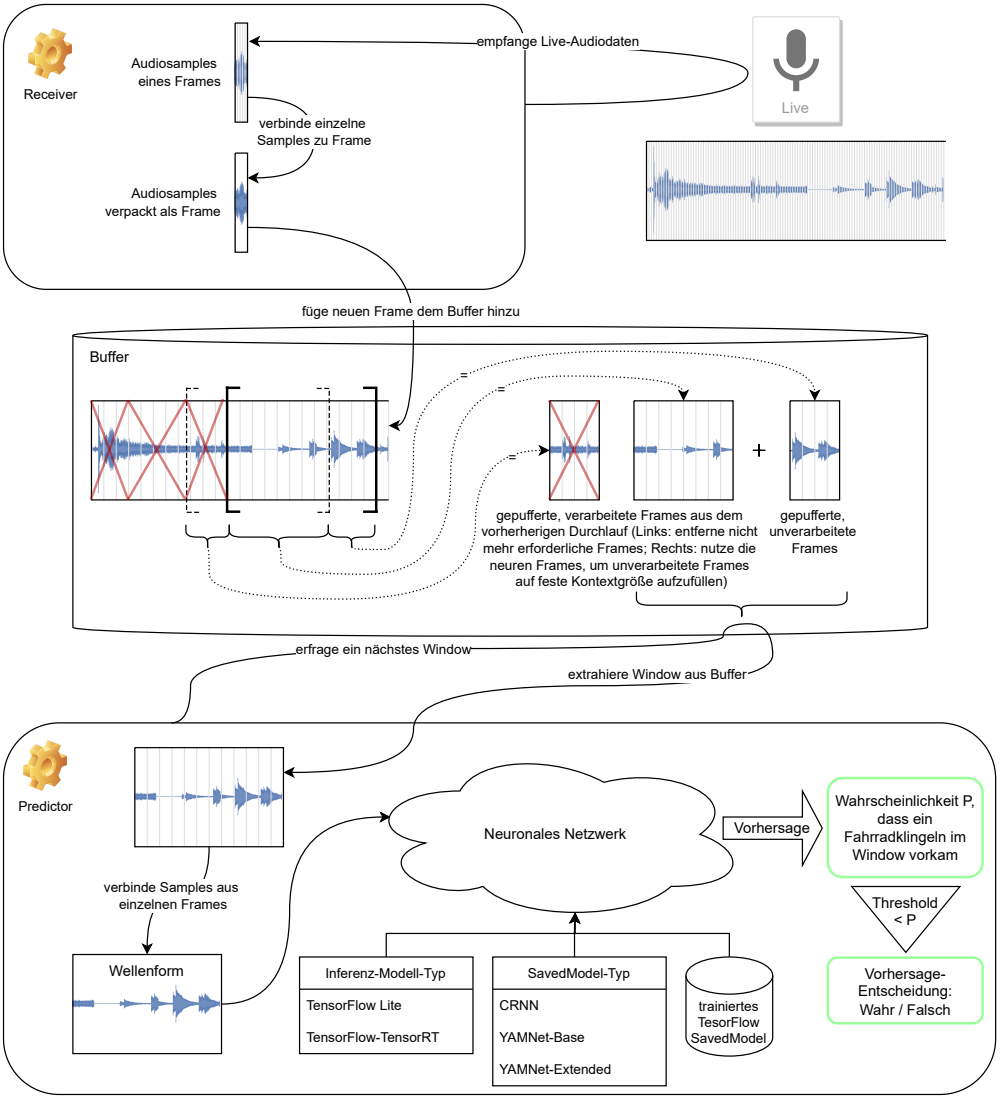


Abbildung 3.2: Übersicht der Hauptkomponenten des Softwaresystems und deren Funktionsweise.

Gleitkommazahlen mit einer Samplingtiefe von 32 Bits. Da der Seeed ReSpeaker zwei Mikrofone hat, wird die Kanalanzahl entsprechend auf genannte Mikrofonanzahl eingestellt und die Daten werden intern auf den ersten Kanal reduziert. Mit den in der JN-Seds-Cli voreingestellten Parametern werden, für 1 ms, Folgen von jeweils 16 Samples geliefert. Diese Folge von Audiosamples wird zusammengesetzt als Frame bezeichnet. Ein neuer Frame der Länge einer Millisekunde wird nach dem Erhalt dem Buffer hinzugefügt.

Predictor. Der Predictor (engl. Vorhersager) hat die Aufgabe, die Daten aus dem Buffer zu verarbeiten und Vorhersageergebnisse zu erstellen. Nebenläufig zum regelmäßigen Hinzufügen von Daten in den Buffer durch den Receiver, findet die Verarbeitung durch den Predictor statt.

Ist der Predictor nach der Fertigstellung der vorhergehenden Verarbeitung $d-1$ bereit, erfragt er aus dem Buffer ein neues Window an Daten. Ein Window bezeichnet hier die vom Erkennungsmodell zu verwendende Menge an aufeinanderfolgenden Audiodaten für einen Durchlauf d. Der Kontext, vor- oder nach dem Erklingen des Soundevents, kann relevant für eine gute Vorhersage für ein solches sein. Daher entnimmt der Predictor nicht nur die einzelnen gepufferten Frames, die noch nicht verarbeitet wurden, sondern ergänzt sie bis zum Erreichen der Window-Größe mit bereits verarbeiteten Frames des vorhergehenden Durchlaufs $d-1$. Voreingestellt ist in der JN-Seds-Cli eine Sekunde als Größe eines solchen Kontextfensters, also sind in einem Window 1000 Frames enthalten, bei einer Frame-Länge von einer Millisekunde. Die für den aktuellen Durchlauf d nicht mehr benötigten Frames werden aus dem Buffer entfernt und nur noch die 1000 Frames des aktuellen Windows werden für den folgenden Durchlauf $d+1$ gespeichert. Die jeweils 16 Samples der 1000 Frames werden vom Predictor anschließend zu einer fortlaufenden Wellenform der Audiodaten zusammengesetzt und dem neuronalen Netzwerk als Eingabe übergeben. Es berechnet den Wahrscheinlichkeitswert für das Vorkommen eines Fahrradklingel-Sounds in dem Window und entscheidet in der Nachverarbeitung anhand des eingestellten Thresholds, ob die Wahrscheinlichkeit hoch genug ist, um es als den Ziel-Sound zu bezeichnen. Abschließend führt der Predictor die in der CLI angegebene Callback-Funktion aus und erzeugt damit eine Ausgabe.

Am Anfang der Ausführung des Detektionsprozesses, überprüft das Softwaresystem die angegebenen Modelleinstellungen und bereitet das Modell vor. Die trainierten Erkennungsmodelle aus Abschnitt 3.3 sind im TensorFlow SavedModel-Format gespeichert und werden zu Beginn in das gewählte Inferenz-Modell konvertiert. In der JN-Seds-Cli ist das standardmäßig TensorFlow Lite. Die Größen der damit optimierten Modelle wurden im Abschnitt 3.3 in der Tabelle 3.1 aufgezeigt. Ein SavedModel-Typ, wie es in der Abbildung und in der Implementierung bezeichnet ist, wird zur Modell-spezifischen Vor- und Nachverarbeitung der Daten gewählt. Mithilfe des SavedModel-Typs wird dementsprechend das Eingabe- und Ausgabeformat der jeweiligen Modelle für die Software festgelegt. Wie in dem Abschnitt zu den Architekturen 3.3.1 beschrieben, sind die Schnittstellen der drei vorgestellten Modelle YAMNet-Base, YAMNet-Extended und CRNN gleich. Zur Aufzeigung der Anpassbarkeit dieser Schnittstelle wurden die drei SavedModel-Typen trotzdem getrennt aufgelistet. Das Softwaresystem ist vollständig unabhängig von den internen Eigenschaften der Modelle implementiert.

Das System mit seinen Komponenten funktioniert nach dem Erzeuger-Verbraucher-Schema (engl. Producer-Consumer-Pattern). Receiver und Predictor laufen in jeweils einem eigenständigen Thread, sodass sie nebenläufig arbeiten können. Mit dieser Struktur des Systems ist eine performante Soundevent-Detektion in Live-Audiodaten realisiert worden.

3.4.4 Bereitstellung auf dem Zielgerät

Der Jetson Nano ist das Zielgerät, für welches die Bicycle-Bell-Seds-Cli primär bereitgestellt wird. Dadurch soll das Gesamtsystem zu einem eingebetteten System für den Zweck der Erkennung von akustischen Warnsignalen von Fahrradfahrern vervollständigt werden. Nachdem die Hardware wie im Abschnitt 3.2 vorbereitet wurde, kann die Soft-

ware mithilfe von pip,[10] dem Paketverwaltungsprogramm für Python, installiert werden. Dazu kann das Paket direkt aus dem Python Package Index (PyPi),[11] ein Repository für Python-Software, bezogen werden. Alternativ können die Programmdateien direkt aus dem Projekt-Repository der Software bezogen und installiert werden, was für Entwicklungszwecke nützlich sein kann. In beiden Fällen kann die Software sowohl auf dem Jetson Nano als auch auf anderen Systemen betrieben werden. Dadurch wird die Entwicklung auf den verbreiteten x86 Plattformen ermöglicht.

Installation. Für die Bereitstellung der Software für JetPack 4.6.1 auf dem Jetson Nano gab es während der Entwicklung einige Voraussetzungen zu beachten. Die Software hat bestimmte Abhängigkeiten zu anderen Python 3.6 Paketen. Dazu gehört insbesondere PyAudio, TensorFlow und TensorFlow IO.

PyAudio wird intern für die Integration der angeschlossenen Audiogeräte verwendet. Es basiert auf der plattformübergreifenden und quelloffenen Audio-I/O-Bibliothek PortAudio ,[12] weshalb eine gesonderte Installation erforderlich ist. NVIDIA bietet eine für den Jetson Nano ausgelegte Version von TensorFlow 2.7 mit JetPack 4.6.1 an, die zur Zeit der Ausarbeitung die aktuellste stabile Version von JetPack ist. Die genannte Version von TensorFlow ist ausschließlich für das vorinstallierte Python 3.6 verfügbar und bietet eine Unterstützung der GPU des Einplatinencomputers. Aus dieser Gegebenheit heraus entstand jedoch eine Kompatibilitätseinschränkung zu TensorFlow IO. Bei der Entwicklung der Software musste sich auf die Sprachfeatures von Python 3.6 und Bibliotheksversionen für die ARM64 Architektur beschränkt werden. Für ARM64 mit den eben benannten anderen Installationen gab es zur Zeit der Ausarbeitung keine kompatible TensorFlow IO Version in PyPi. Eine Lösung zur Installation von TensorFlow IO in Version 0.21.0 wurde gefunden, jedoch ist dafür eine andere TensorFlow Installation erforderlich. Unter Verwendung einer alternativen TensorFlow Version entfällt die GPU-Unterstützung unter JetPack 4.6.1. Mit der geplanten Einführung von JetPack 5.0 soll eine höhere Python-Version vorinstalliert sein. Dementsprechend wird das benannte Problem wahrscheinlich in JetPack 5.0 vollständig gelöst sein. Zur Entwicklungszeit konnte dadurch jedoch noch keine Ausführung der Software mit GPU-Unterstützung getestet werden. Deshalb konnte die Optimierungstechnik von TensorFlow-TensorRT bisher nicht angewendet werden. Zur Ausführung der Software muss die Konvertierung in ein TensorFlow Lite Modell auf dem Zielgerät geschehen und kann anschließend auf den Kernen des Prozessors vom Jetson Nano laufen. Die Verzögerungen sind mit TensorFlow Lite schon sehr gering, wie in 3.4.2 gezeigt, sodass TensorFlow Lite eine für das Prototyping geeignete Möglichkeit bietet.

Eine detaillierte Schritt-für-Schritt-Anleitung zur Installation der Software auf dem Zielgerät ist im Repository unter Beachtung der hier beschriebenen Kompatibilitätsvoraussetzungen dokumentiert. Ebenfalls sind dort Hinweise und Beispiele für die Programmausführung zu finden.

Ausführung. Eine Ausführung des Programmes findet über die Kommandozeile statt. Mit der Installation der Bicycle-Bell-Seds-Cli werden zwei bereits beschriebenen Varianten

[10] Link zur pip Webseite: `https://pip.pypa.io` (1. Juli 2022)
[11] Link zur PyPi Webseite: `https://pypi.org` (1. Juli 2022)
[12] Link zur PortAudio Webseite: `http://www.portaudio.com` (1. Juli 2022)

installiert: die `seds-cli` und die `jn-seds-cli`. Beide Varianten umfassen alle Funktionalitäten des Softwaresystems. Da auch an dieser Stelle wieder das Softwaresystem in dem Gesamtsystem betrachtet wird, wird im Folgenden von der JN-Seds-Cli `jn-seds-cli` gesprochen.

Sehr zentral für den Einsatz der Software sind die beiden Kommandos `jn-seds-cli --help` und `jn-seds-cli run --tfmodel='!crnn' production`. Mit `jn-seds-cli --help` sind alle beschriebenen Funktionsgruppen aus 3.4.2 bei der Ausführung der Software nachlesbar. Für die jeweiligen Funktionen und Unterfunktionen kann jeweils rekursiv `--help` genutzt werden. Wird beispielsweise `jn-seds-cli run --help` ausgeführt, werden alle Informationen zur Ausführung des Detektionsprozesses und zu den verfügbaren Parametrisierungen ausgegeben. Dadurch können zu jedem Feature notwendige und detaillierte Informationen gefunden werden. Die Ausführung des Produktionsmodus wird mit dem Befehl `jn-seds-cli run --tfmodel='!crnn' production` gestartet. Mithilfe des `tfmodel`-Parameters und des `!`-Symbols wird das angegebene vorinstallierte Modell zur Fahrradklingel-Erkennung ohne Pfadangabe ausgewählt. In dem Fall des notierten Beispielbefehls wird dementsprechend das CRNN geladen. Ebenso sind demnach `--tfmodel='!yamnet_extended'` und `--tfmodel='!yamnet_base'` verwendbar.

Der Produktionsmodus wird mit dem Kommando `production` gewählt, sodass das System zur Live-Erkennung mit den am Jetson Nano verbundenen Standardaudiogeräten gestartet wird. Eine beispielhafte Ausgabe auf der Konsole kann wie folgt aussehen:

```
$ jn-seds-cli run --tfmodel='!crnn' production
...
2022-06-26 21:13:35,048 INFO: Prediction for the past 1.000sec: False [0.38] |
    ↪ delay: 0.034-0.064sec
2022-06-26 21:13:35,078 INFO: Prediction for the past 1.000sec: False [0.47] |
    ↪ delay: 0.030-0.061sec
2022-06-26 21:13:35,107 INFO: Prediction for the past 1.000sec: True [0.95] |
    ↪ delay: 0.030-0.058sec
...
```

Wie in Unterabsatz 3.4.2 beschrieben und im Ausschnitt zu erkennen, wird zu jedem Vorhersageergebnis eine Zeile als Ausgabe erzeugt. Eine Zeile enthält einen Wahrscheinlichkeitswert und das Label der Klassifikation, sowie eine Angabe zur entstandenen Verzögerung. Im Anhang unter „Ausgabe einer Ausführung im Produktionsmodus“ ist eine Ausgabe einer gesamten Beispielausführung des Gesamtsystems dargestellt. Die gesamte Parametrisierung einer Ausführungsinstanz wird bei dem Start des Detektionssystems als Ausgabe erzeugt. In dem Beispiel-Fall wird nur ein expliziter Parameter angegeben, und zwar zum zu verwendenen, gespeicherten Modell mit `--tfmodel='!crnn'`. Jedoch werden ebenso die erzeugten, impliziten Parameter ausgegeben, wie etwa die Standardeinstellung des Thresholds mit 0,5. Bei diesem Funktionsnachweis wurde eine Fahrradklingel ein einzelnes Mal betätigt, was auch korrekt in den Vorhersageergebnissen im Anhang wiederzuerkennen ist. Es ist zudem ersichtlich, dass ein Threshold von 0,1 eine frühere Erkennung von knapp über 200 ms ermöglichen würde. Andererseits kann eine Senkung des Schwellenwertes die Zahl von unerwünschten Erkennungen nachsichziehen.

Zusätzlich zum Funktionsnachweis des Gesamtsystems durch den Produktionsmodus befinden sich die Ergebnisse einer Beispielausführung des Evaluationsmodus im Anhang unter „Ausgabe einer Ausführung im Evaluationsmodus“. Dieser Modus wurde auf einer

x86 Plattform mit einem anderen Mikrofon testweise ausgeführt und kann in nächsten Schritten für den Betrieb auf dem Jetson Nano und empirischen Untersuchungen angepasst werden. Wie in dem folgenden Ausschnitt zu sehen ist, kann das Softwaresystem durch die Angabe `--channels` auf die Kanalanzahl des gewählten Audiogerätes abgestimmt werden:

```
$ jn-seds-cli run --tfmodel='!crnn' evaluation --channels=1
...
2022-06-29 11:09:42,463 INFO: Prediction of the past 1.000sec: True [0.60]
    ↪ received, True [0.96] played, True [1.00] ground-truth | delay: 0.016-
    ↪ 0.055sec
...
```

Es entstehen jeweils drei Vorhersagen, wobei das als `ground-truth` bezeichnete Ergebnis direkt aus den Annotationen der CSV-Datei abgeleitet ist. Zur Erzeugung jeder Ausgabezeile wurde das Erkennungsmodell zum einen direkt auf den abgespielten Audiodaten der WAVE-Datei (`played`) und zum anderen unabhängig auf den aufgenommenen Mikrofon-Daten ausgeführt. Bei der Betrachtung der Ausgaben ist es wichtig zu beachten, dass die verwendeten Testdaten mit WAVE- und CSV-Datei synthetisch aus verschiedenen Audio-Clips zusammengesetzt wurden. Ein Teil der Verschiebung zwischen den Ergebnissen in `ground-truth` und `played` ist demnach begründet in dem tatsächlichen Timing des Fahrradklingelns innerhalb der Audio-Clips. So kann es vorkommen, dass das Fahrradklingeln nicht direkt am Anfang der WAVE-Datei vorkommt, was aber bei der automatisierten Annotation nicht berücksichtigt ist. An den Verzögerungen und Wahrscheinlichkeitsdifferenzen zwischen `played` und `received` ist der Einfluss von Softwareexternen Faktoren erkennbar. Dazu gehören die Aufnahmequalität des Mikrofons, die Umgebungsgeräusche und das bereits erwähnte Signal-Rausch-Verhältnis. Es findet eine deutlich spätere und unsicherere Erkennung bei den aufgenommenen Audiodaten statt als bei den abgespielten. Somit konnte mithilfe des Modus bereits das Zusammenspiel von Softwaresystem, Erkennungsmodell und einem Mikrofon getestet werden. Für Schlussfolgerungen zur Detektions-Performanz des Gesamtsystems sind jedoch weitere Untersuchungen notwendig.

Mithilfe der beschriebenen Ausgaben des Produktionsmodus konnte gezeigt werden, dass das eingebettete System mit all seinen Komponenten funktioniert. Das Softwaresystem aus Abschnitt 3.4 läuft zusammen mit der Hardware aus Abschnitt 3.2 und kann mithilfe der Erkennungsmodelle aus Abschnitt 3.3 ein Fahrradklingeln detektieren und bildet so das Gesamtsystem aus 3.1.

4 Fazit

Mit dieser Arbeit wurde ein neuartiges System umgesetzt, welches der Erkennung der akustischen Warnsignale von Fahrradfahrern dient. Dazu wurden verschiedene künstliche neuronale Netzwerke für die Wahrnehmung von Fahrradklingeln erstellt. Sie wurden in einer entwickelten Software für die Echtzeitverarbeitung von Live-Audiodaten integriert und ein NVIDIA Jetson Nano mit einem Mikrofon des Seeed ReSpeaker 2-Mics Pi HAT verwendet. Abschließend konnte das Softwaresystem in das Hardwaresystem eingebettet werden, sodass im Rahmen dieser Arbeit ein Prototyp für die Fahrradklingel-Erkennung in Live-Audioaufnahmen entstand.

Zusammenfassung. Die in dieser Arbeit vorgestellten neuronalen Netzwerke, YAMNet-Base, YAMNet-Extended und CRNN, wurden für die Soundevent-Klassifikation trainiert und evaluiert. Eine state-of-the-art Performanz wurde mit den Erkennungsmodellen YAMNet-Extended und insbesondere dem CRNN erreicht. Mit der stetigen Ausführung eines Erkennungsmodells auf den aufgenommenen Audiosamples wird die Detektion der Fahrradklingel-Sounds im Softwaresystem realisiert. Ein Funktionsnachweis des ausführbaren Gesamtsystems für die Soundevent-Detektion konnte erbracht werden.

Mit der Implementierung des Prototyps wurden mehrere Möglichkeiten geschaffen. Dazu gehören Untersuchungen des akustisch, aktiven Ansatzes zum Schutz von Fahrradfahrern, Weiterentwicklungen der einzelnen Systembestandteile und eine Einbettung der Lösung in andere Systeme. Das Softwaresystem wurde mit einem hohen Grad an Modularität umgesetzt und eine Anpassung an diverse Hardwaresysteme durch Parametrisierung über eine CLI geschaffen. Das ermöglicht eine einfache Bereitstellung auf alternativen Plattformen mit anderen Mikrofon-Sensoren. Zudem lassen sich die Erkennungsmodelle unmittelbar auf bestimmte Konfigurationen in der Software anpassen oder über geschaffene Schnittstellen austauschen.

Neben dem Produktionsmodus zur Live-Erkennung der Fahrradklingeln im akustischen Umfeld wird zusätzlich der Evaluationsmodus angeboten. Zusammen mit einer individualisierbaren Callback-Funktion, sind Untersuchungen des Zusammenwirkens von Software-Parametrisierungen und Mikrofonen realisierbar. Auf diese Weise kann der in der Arbeit behandelte Ansatz einer Fahrradklingel-Erkennung zu einem Teil eines Mehrsensorsystems weiterentwickelt werden und als zusätzlicher Faktor dem Schutz von Fahrradfahrern dienen.

Ausblick. Für weiterführende Arbeiten ist insbesondere die Durchführung von empirischen Studien zur Detektion im Gesamtsystem relevant. Außerdem können weitere Performanz-Steigerungen für das Ziel eines Produktiveinsatzes in Kraftfahrzeugen vorgenommen werden.

Der Evaluationsmodus sollte für den Einsatz im Gesamtsystem angepasst werden. Anschließend sind empirische Untersuchungen des Detektionssystems unter Verwendung der Callback-Funktion durchführbar. Dazu kann sowohl mit Tests auf synthetischen Daten als auch mit ausführlichen Praxistests evaluiert werden.

Die Performanz ist sowohl zeitlich als auch qualitativ verbesserbar. Um die Robustheit der neuronalen Netzwerke zu steigern, können weitere Variationen der Daten durch spezifischere Daten-Augmentation erzeugt werden. Das kann die Performanz der Detektion bei geringeren Lautstärke-Unterschieden zwischen dem Warnsignal und Umgebungsgeräuschen erhöhen. Um die Verzögerungen bei der Verarbeitung zu verkürzen, kann an der Unterstützung von TensorFlow-TensorRT oder anderen Techniken zur Inferenzoptimierung mit GPU-Unterstützung gearbeitet werden. Außerdem ist eine Anpassung der Softwareimplementierung der Predictor-Komponente auf die Verwendung mehrerer Prozessor-Kerne ein möglicher Ansatz, um die Pufferungsdauer zu reduzieren.

Es sollte eine Weiterentwicklung für die Unterstützung von Mehrkanal-Audiodaten betrachtet werden. Mehrere Mikrofone können der Lokalisierung des Soundevents und der anschließenden Bestimmung der Relevanz des erkannten Fahrradklingelns dienen. Das kann den effektiven Einsatz in Kraftfahrzeugen als zusätzlicher Schutzansatz für Fahrradfahrer ermöglichen.

Literaturverzeichnis

[1] Martín Abadi u. a. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[2] Sharath Adavanne, Giambattista Parascandolo, Pasi Pertilä, Toni Heittola und Tuomas Virtanen. „Sound event detection in multichannel audio using spatial and harmonic features“. In: *arXiv preprint arXiv:1706.02293* (2017).

[3] Sharath Adavanne, Pasi Pertilä und Tuomas Virtanen. „Sound event detection using spatial features and convolutional recurrent neural network“. In: *2017 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE. 2017, Seiten 771–775.

[4] Amirul Sadikin Md Affendi und Marina Yusoff. „Review of anomalous sound event detection approaches“. In: *IAES International Journal of Artificial Intelligence* 8.3 (2019), Seite 264.

[5] Shahin Amiriparian, N Cummins, S Julka und BW Schuller. „Deep convolutional recurrent neural network for rare acoustic event detection“. In: *Proc. DAGA*. 2018, Seiten 1522–1525.

[6] Emre Cakir. *Deep neural networks for sound event detection*. Tampere University, 2019.

[7] Emre Cakır und Tuomas Virtanen. „Convolutional recurrent neural networks for rare sound event detection“. In: *Deep Neural Networks for Sound Event Detection* 12 (2019).

[8] Yanping Chen und Hongxia Jin. „Rare Sound Event Detection Using Deep Learning and Data Augmentation.“ In: *INTERSPEECH*. 2019, Seiten 619–623.

[9] Keunwoo Choi, George Fazekas und Mark Sandler. „Automatic tagging using deep convolutional neural networks“. In: *arXiv preprint arXiv:1606.00298* (2016).

[10] Eduardo Fonseca, Xavier Favory, Jordi Pons, Frederic Font und Xavier Serra. „Fsd50k: an open dataset of human-labeled sound events“. In: *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 30 (2021), Seiten 829–852.

[11] Shayan Shirahmad Gale Bagi, Hossein Gharaee Garakani, Behzad Moshiri und Mohammad Khoshnevisan. „Sensing Structure for Blind Spot Detection System in Vehicles“. In: *2019 International Conference on Control, Automation and Information Sciences (ICCAIS)*. 2019, Seiten 1–6. DOI: 10.1109/ICCAIS46528.2019.9074580.

[12] Jort F. Gemmeke, Daniel P. W. Ellis, Dylan Freedman, Aren Jansen, Wade Lawrence, R. Channing Moore, Manoj Plakal und Marvin Ritter. „Audio Set: An ontology and human-labeled dataset for audio events“. In: *Proc. IEEE ICASSP 2017*. New Orleans, LA, 2017.

[13] Sepp Hochreiter und Jürgen Schmidhuber. „Long short-term memory“. In: *Neural computation* 9.8 (1997), Seiten 1735–1780.

[14] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto und Hartwig Adam. „Mobilenets: Efficient convolutional neural networks for mobile vision applications“. In: *arXiv preprint arXiv:1704.04861* (2017).

[15] *Jetson Modules*. NVIDIA. URL: https://developer.nvidia.com/embedded/jetson-modules (besucht am 29. Juni 2022).

[16] Iver Jordal, Araik Tamazian, Emmanouil Theofanis Chourdakis, Céline Angonin, askskro, Nikolay Karpov, Omer Sarioglu, kvilouras, Enis Berk Çoban, Florian Mirus, Jeong-Yoon Lee, Kwanghee Choi, MarvinLvn, SolomidHero und Tanel Alumäe. *iver56/audiomentations: v0.25.1*. Version v0.25.1. Juni 2022. DOI: 10.5281/zenodo.6645998. URL: https://doi.org/10.5281/zenodo.6645998.

[17] Tsz Laam Kiang. „A Novel Multi-Sensor Crash Safety System Targeting Heavy Duty Vehicle Blind Spots“. In: *2021 IEEE Asia-Pacific Conference on Image Processing, Electronics and Computers (IPEC)*. 2021, Seiten 852–859. DOI: 10.1109/IPEC51340.2021.9421130.

[18] Diederik P Kingma und Jimmy Ba. „Adam: A method for stochastic optimization“. In: *arXiv preprint arXiv:1412.6980* (2014).

[19] *Kraftrad- und Fahrradunfälle im Straßenverkehr 2020*. Seite 8. Statistisches Bundesamt (Destatis). 2021. URL: https://www.destatis.de/DE/Themen/Gesellschaft-Umwelt/Verkehrsunfaelle/Publikationen/Downloads-Verkehrsunfaelle/unfaelle-zweirad-5462408207004.pdf?__blob=publicationFile (besucht am 28. Juni 2022).

[20] Shaobo Li, Yong Yao, Jie Hu, Guokai Liu, Xuemei Yao und Jianjun Hu. „An ensemble stacked convolutional neural network model for environmental event sound recognition“. In: *Applied Sciences* 8.7 (2018), Seite 1152.

[21] Hyungui Lim, Jeongsoo Park und Yoonchang Han. „Rare sound event detection using 1D convolutional recurrent neural networks“. In: *Proceedings of the Detection and Classification of Acoustic Scenes and Events 2017 Workshop*. 2017, Seiten 80–84.

[22] Anders Liss. „Implementation of a neural network on a microcontroller for recognition of warning signals“. Uppsala universitet, 2020.

[23] Brian McFee u. a. *librosa/librosa: 0.9.2*. Version 0.9.2. https://librosa.org/doc/main/generated/librosa.amplitude_to_db.html. Juni 2022. DOI: 10.5281/zenodo.6759664. URL: https://doi.org/10.5281/zenodo.6759664 (besucht am 29. Juni 2022).

[24] Annamaria Mesaros, Toni Heittola und Tuomas Virtanen. „Metrics for polyphonic sound event detection“. In: *Applied Sciences* 6.6 (2016), Seite 162.

[25] Annamaria Mesaros, Toni Heittola, Tuomas Virtanen und Mark D Plumbley. „Sound event detection: A tutorial“. In: *IEEE Signal Processing Magazine* 38.5 (2021), Seiten 67–83.

[26] Loris Nanni, Gianluca Maguolo, Sheryl Brahnam und Michelangelo Paci. „An ensemble of convolutional neural networks for audio classification“. In: *Applied Sciences* 11.13 (2021), Seite 5796.

[27] Giambattista Parascandolo, Heikki Huttunen und Tuomas Virtanen. „Recurrent neural networks for polyphonic sound event detection in real life recordings“. In: *2016 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE. 2016, Seiten 6440–6444.

[28] Karol J Piczak. „Environmental sound classification with convolutional neural networks“. In: *2015 IEEE 25th international workshop on machine learning for signal processing (MLSP)*. IEEE. 2015, Seiten 1–6.

[29] Hendrik Purwins, Bo Li, Tuomas Virtanen, Jan Schlüter, Shuo-Yiin Chang und Tara Sainath. „Deep learning for audio signal processing“. In: *IEEE Journal of Selected Topics in Signal Processing* 13.2 (2019), Seiten 206–219.

[30] *Radverkehr*. Bundesministerium für Digitales und Verkehr. 2022. URL: https://www.bmvi.de/SharedDocs/DE/Artikel/StV/fahrrad-uebersicht.html?https=1 (besucht am 28. Juni 2022).

[31] Akshay Rangesh und Mohan Manubhai Trivedi. „No Blind Spots: Full-Surround Multi-Object Tracking for Autonomous Vehicles Using Cameras and LiDARs“. In: *IEEE Transactions on Intelligent Vehicles* 4.4 (2019), Seiten 588–599. DOI: 10.1109/TIV.2019.2938110.

[32] Thomas Ruland. „Einführung in Neuronale Netze“. In: (2004).

[33] Jürgen Schmidhuber. „Deep learning in neural networks: An overview“. In: *Neural networks* 61 (2015), Seiten 85–117.

[34] *Seeed ReSpeaker 2-Mics Pi HAT*. Seeed Studio. URL: https://wiki.seeedstudio.com/ReSpeaker_2_Mics_Pi_HAT/ (besucht am 29. Juni 2022).

[35] Mayuri S Shelke, Prashant R Deshmukh und Vijaya K Shandilya. „A review on imbalanced data handling using undersampling and oversampling technique“. In: *Int. J. Recent Trends Eng. Res* 3.4 (2017), Seiten 444–449.

[36] Stephen Smaldone, Chetan Tonde, Vancheswaran Ananthanarayanan, Ahmed Elgammal und Liviu Iftode. „Improving bicycle safety through automated real-time vehicle detection“. In: (2010).

[37] *Stereo CODEC with 1W Stereo Class D Speaker Drivers and Headphone Drivers for Portable Audio Applications*. Wolfson Microelectronics. URL: https://www.waveshare.com/w/upload/1/18/WM8960_v4.2.pdf (besucht am 29. Juni 2022).

[38] Michal Steć, Benno Stabernack und Clemens Kubach. „Towards an optimal right-turn assistant system to avoid accidents with vulnerable traffic participants“. In: *2022 11th Mediterranean Conference on Embedded Computing (MECO)*. 2022, Seiten 1–5. DOI: 10.1109/MECO55406.2022.9797083.

[39] *TensorFlow Hub: YAMNet*. Google. URL: https://tfhub.dev/google/yamnet/1 (besucht am 28. Juni 2022).

[40] *Transfer learning and fine-tuning*. Google. URL: https://www.tensorflow.org/tutorials/images/transfer_learning (besucht am 28. Juni 2022).

[41] *Transfer learning with YAMNet for environmental sound classification*. TensorFlow. URL: https://www.tensorflow.org/tutorials/audio/transfer_learning_audio (besucht am 28. Juni 2022).

[42] *Verkehr in Zahlen 2021/2022*. Seiten 226-227. Bundesministerium für Verkehr und digitale Infrastruktur. 2022. URL: https://www.bmvi.de/SharedDocs/DE/Publikationen/G/verkehr-in-zahlen-2021-2022-pdf.pdf?__blob=publicationFile (besucht am 28. Juni 2022).

[43] Shoujin Wang, Wei Liu, Jia Wu, Longbing Cao, Qinxue Meng und Paul J Kennedy. „Training deep neural networks on imbalanced data sets". In: *2016 international joint conference on neural networks (IJCNN)*. IEEE. 2016, Seiten 4368–4374.

[44] Yong Xu, Qiuqiang Kong, Wenwu Wang und Mark D Plumbley. „Large-scale weakly supervised audio classification using gated convolutional neural network". In: *2018 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE. 2018, Seiten 121–125.

[45] *YAMNet*. TensorFlow. URL: https://github.com/tensorflow/models/tree/master/research/audioset/yamnet (besucht am 28. Juni 2022).

[46] Haomin Zhang, Ian McLoughlin und Yan Song. „Robust sound event recognition using convolutional neural networks". In: *2015 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE. 2015, Seiten 559–563.

[47] Xiaohu Zhang, Yuexian Zou und Wei Shi. „Dilated convolution neural network with LeakyReLU for environmental sound classification". In: *2017 22nd International Conference on Digital Signal Processing (DSP)*. IEEE. 2017, Seiten 1–5.

[48] Zhichao Zhang, Shugong Xu, Shunqing Zhang, Tianhao Qiao und Shan Cao. „Attention based convolutional recurrent neural network for environmental sound classification". In: *Neurocomputing* 453 (2021), Seiten 896–903.

A Anhang

Darstellungen der neuronalen Netzwerk-Architekturen

YAMNet-Base

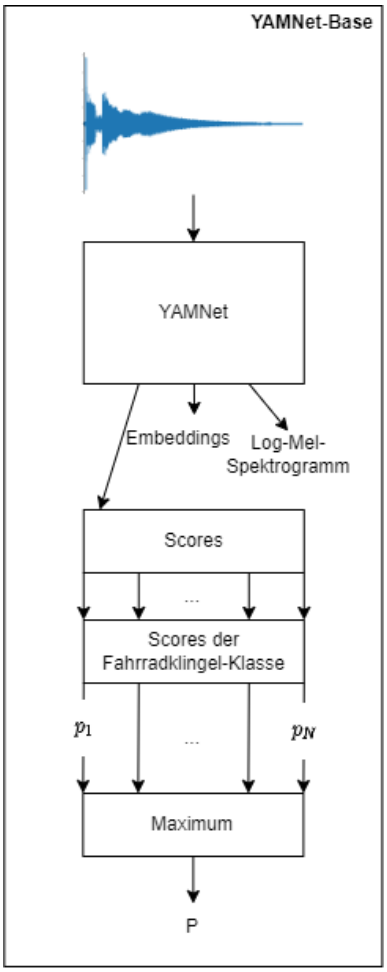


Abbildung A.1: Darstellung der YAMNet-Base Architektur.

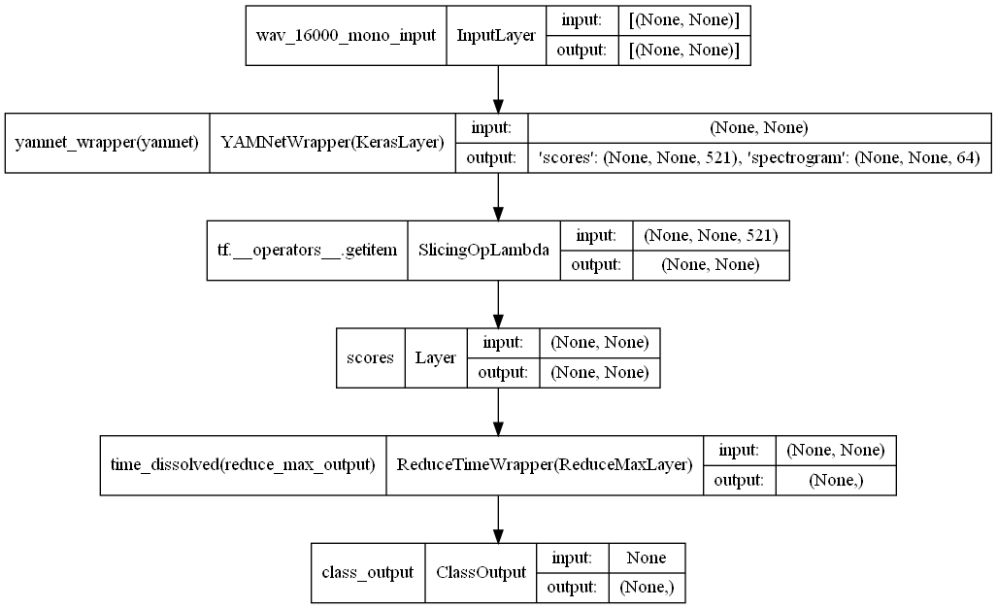


Abbildung A.2: Darstellung des Layer-Aufbaus der YAMNet-Base Architektur.

YAMNet-Extended

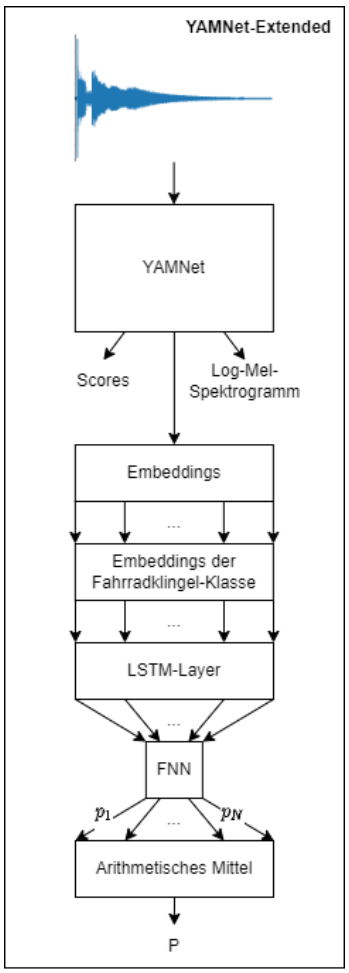


Abbildung A.3: Darstellung der YAMNet-Extended Architektur.

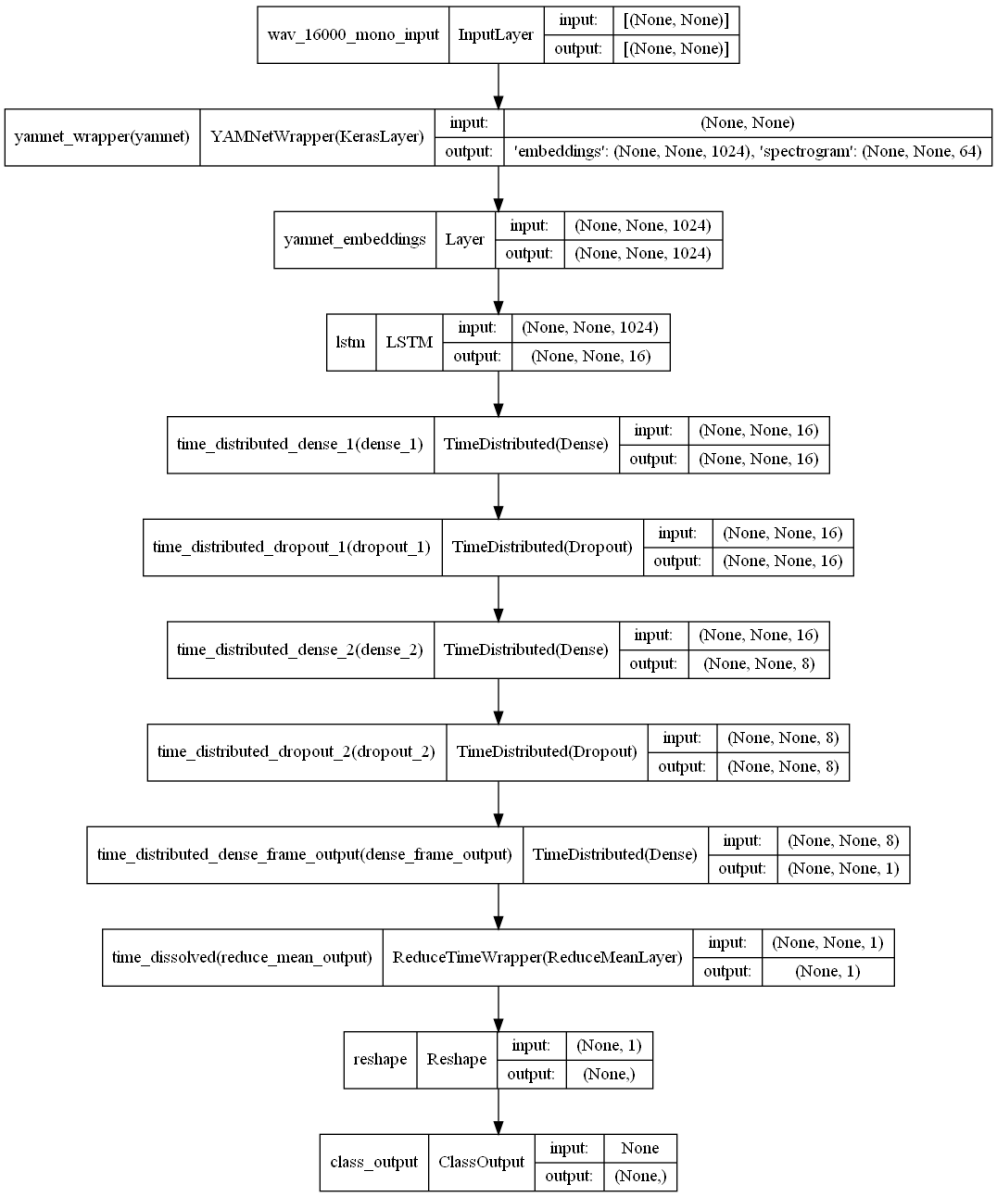


Abbildung A.4: Darstellung des Layer-Aufbaus der YAMNet-Extended Architektur.

CRNN

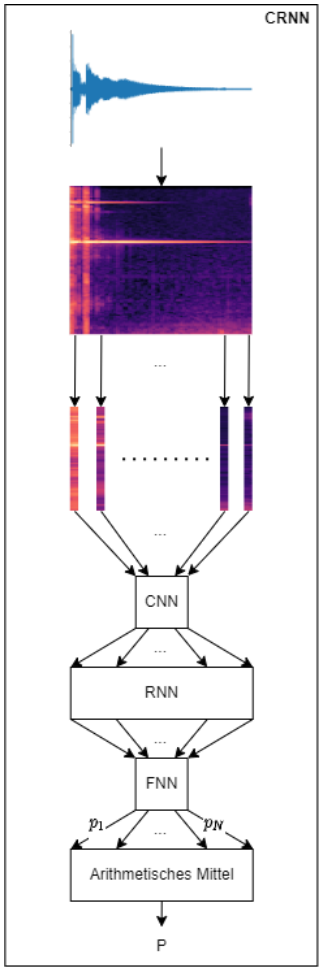


Abbildung A.5: Darstellung der CRNN Architektur.

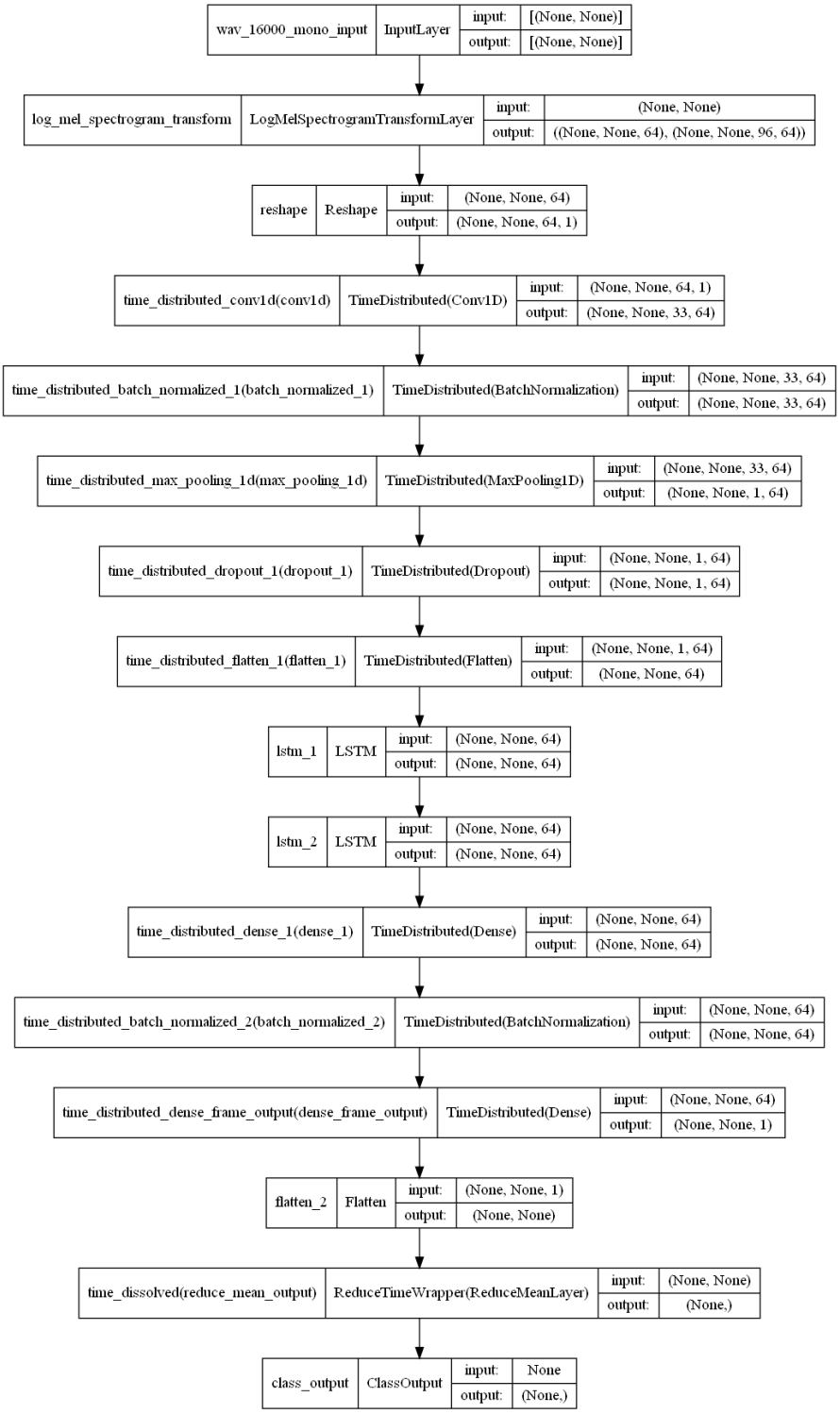


Abbildung A.6: Darstellung des Layer-Aufbaus der CRNN Architektur.

Ausgabe einer Ausführung im Produktionsmodus

Die hier notierte Beispielausführung des Softwaresystems fand auf der vorgestellten Jetson Nano Plattform unter Python 3.6 statt. Es wurden einige initiale Zeilen mit ... rausgekürzt, um das Beispiel auf Ausgaben zu beschränken, die für den Funktionsnachweis des Gesamtsystems bedeutend sind.

```
$ jn-seds-cli run --tfmodel='!crnn' production
Parameters: {'tfmodel_path': '/home/jetson/venv/lib/python3.6/site-packages/
    ↪ seds_cli/res/models/crnn', 'threshold': 0.5, 'channels': 2, 'gpu': False,
    ↪  'infer_model': <InferenceModels.TFLITE: <class 'seds_cli.seds_lib.models
    ↪ .inference_models.TFLiteInferenceModel'>>, 'saved_model': <SavedModels.
    ↪ CRNN: <class 'seds_cli.seds_lib.models.saved_models.CrnnSavedModel'>>, '
    ↪ storage_length': 0, 'save_records': False, 'input_device': None, '
    ↪ output_device': None, 'sample_rate': 16000, 'window_length': 1.0, '
    ↪ frame_length': 0.001, 'callback': <function get_custom_logging_callback.<
    ↪ locals>.custom_callback at 0x7f753a96a8>, 'loglevel': <LogLevels.INFO: '
    ↪ info'>, 'save_log': False, 'mode': None, 'silent': False, 'use_input':
    ↪ True, 'use_output': False, 'wav_file': None, 'annotation_file': None}
2022-06-26 21:10:26,874 INFO: Production mode selected
2022-06-26 21:10:26,874 INFO: Environment variable CUDA_VISIBLE_DEVICES is set to
    ↪  -1
2022-06-26 21:10:26,874 INFO: AudioStorage was initialized for always storing the
    ↪  last 0 seconds of received audio samples.
...
2022-06-26 21:10:26,975 INFO: Number of installed sound devices: 18
...
2022-06-26 21:10:26,981 INFO: Default input sound device info: {'index': 17, '
    ↪ structVersion': 2, 'name': 'default', 'hostApi': 0, 'maxInputChannels':
    ↪ 32, 'maxOutputChannels': 32, 'defaultLowInputLatency':
    ↪ 0.008707482993197279, 'defaultLowOutputLatency': 0.008707482993197279, '
    ↪ defaultHighInputLatency': 0.034829931972789115, 'defaultHighOutputLatency
    ↪ ': 0.034829931972789115, 'defaultSampleRate': 44100.0}
2022-06-26 21:10:26,982 INFO: Default output sound device info: {'index': 17, '
    ↪ structVersion': 2, 'name': 'default', 'hostApi': 0, 'maxInputChannels':
    ↪ 32, 'maxOutputChannels': 32, 'defaultLowInputLatency':
    ↪ 0.008707482993197279, 'defaultLowOutputLatency': 0.008707482993197279, '
    ↪ defaultHighInputLatency': 0.034829931972789115, 'defaultHighOutputLatency
    ↪ ': 0.034829931972789115, 'defaultSampleRate': 44100.0}
2022-06-26 21:10:26,982 INFO: Selected input sound device index (None=default
    ↪ device): None
2022-06-26 21:10:26,982 INFO: Selected output sound device index (None=default
    ↪ device): None
2022-06-26 21:10:26,983 INFO: Num GPUs Available: 0
2022-06-26 21:10:26,983 INFO: Processing device 0: PhysicalDevice(name='/
    ↪ physical_device:CPU:0', device_type='CPU')
2022-06-26 21:10:26,984 INFO: Loading crnn model from: /home/jetson/venv/lib/
    ↪ python3.6/site-packages/seds_cli/res/models/crnn with threshold: 0.5
2022-06-26 21:10:26,984 INFO: Converting model ... (may take a few minutes)
2022-06-26 21:13:26.182169: W tensorflow/compiler/mlir/lite/python/
    ↪ tf_tfl_flatbuffer_helpers.cc:351] Ignored output_format.
2022-06-26 21:13:26.182246: W tensorflow/compiler/mlir/lite/python/
    ↪ tf_tfl_flatbuffer_helpers.cc:354] Ignored drop_control_dependency.
```

```
2022-06-26 21:13:26.182271: W tensorflow/compiler/mlir/lite/python/
    ↪ tf_tfl_flatbuffer_helpers.cc:360] Ignored change_concat_input_ranges.
2022-06-26 21:13:26.185527: I tensorflow/cc/saved_model/reader.cc:38] Reading
    ↪ SavedModel from: /home/jetson/venv/lib/python3.6/site-packages/seds_cli/
    ↪ res/models/crnn
2022-06-26 21:13:26.358033: I tensorflow/cc/saved_model/reader.cc:90] Reading
    ↪ meta graph with tags { serve }
2022-06-26 21:13:26.358135: I tensorflow/cc/saved_model/reader.cc:132] Reading
    ↪ SavedModel debug info (if present) from: /home/jetson/venv/lib/python3.6/
    ↪ site-packages/seds_cli/res/models/crnn
2022-06-26 21:13:26.971375: I tensorflow/cc/saved_model/loader.cc:229] Restoring
    ↪ SavedModel bundle.
2022-06-26 21:13:27.854494: I tensorflow/cc/saved_model/loader.cc:213] Running
    ↪ initialization op on SavedModel bundle at path: /home/jetson/venv/lib/
    ↪ python3.6/site-packages/seds_cli/res/models/crnn
2022-06-26 21:13:28.408461: I tensorflow/cc/saved_model/loader.cc:301] SavedModel
    ↪  load for tags { serve }; Status: success: OK. Took 2222942 microseconds.
2022-06-26 21:13:30.057044: I tensorflow/compiler/mlir/tensorflow/utils/
    ↪ dump_mlir_util.cc:210] disabling MLIR crash reproducer, set env var `
    ↪ MLIR_CRASH_REPRODUCER_DIRECTORY` to enable.
2022-06-26 21:13:32,085 INFO: Converted model of inference type Tensorflow Lite
    ↪ was created and saved in /home/jetson/venv/lib/python3.6/site-packages/
    ↪ seds_cli/res/models/crnn/converted-model.tflite.
2022-06-26 21:13:32,089 INFO: Press Ctrl+C or Interrupt the Kernel
2022-06-26 21:13:33,854 WARNING: The buffer was full and 0.002 seconds were
    ↪ skipped from 1.000 seconds ago to avoid increasing the delay permanently.
2022-06-26 21:13:33,896 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.042-1.042sec
2022-06-26 21:13:33,932 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.036-0.263sec
2022-06-26 21:13:33,965 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.033-0.227sec
2022-06-26 21:13:33,997 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.032-0.215sec
2022-06-26 21:13:34,027 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.029-0.211sec
2022-06-26 21:13:34,058 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.031-0.062sec
2022-06-26 21:13:34,090 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.032-0.062sec
2022-06-26 21:13:34,123 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.033-0.061sec
2022-06-26 21:13:34,155 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.033-0.066sec
2022-06-26 21:13:34,188 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.033-0.065sec
2022-06-26 21:13:34,220 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.032-0.060sec
2022-06-26 21:13:34,253 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.033-0.062sec
2022-06-26 21:13:34,284 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.031-0.065sec
2022-06-26 21:13:34,314 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.030-0.059sec
```

```
2022-06-26 21:13:34,345 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.031-0.059sec
2022-06-26 21:13:34,377 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.030-0.060sec
2022-06-26 21:13:34,407 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.030-0.058sec
2022-06-26 21:13:34,436 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.030-0.059sec
2022-06-26 21:13:34,466 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.030-0.058sec
2022-06-26 21:13:34,496 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.030-0.057sec
2022-06-26 21:13:34,526 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.030-0.059sec
2022-06-26 21:13:34,555 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.030-0.055sec
2022-06-26 21:13:34,586 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.030-0.061sec
2022-06-26 21:13:34,615 INFO: Prediction for the past 1.000sec: False [0.01] |
    ↪ delay: 0.030-0.058sec
2022-06-26 21:13:34,646 INFO: Prediction for the past 1.000sec: False [0.01] |
    ↪ delay: 0.030-0.058sec
2022-06-26 21:13:34,678 INFO: Prediction for the past 1.000sec: False [0.02] |
    ↪ delay: 0.032-0.061sec
2022-06-26 21:13:34,708 INFO: Prediction for the past 1.000sec: False [0.02] |
    ↪ delay: 0.030-0.061sec
2022-06-26 21:13:34,738 INFO: Prediction for the past 1.000sec: False [0.02] |
    ↪ delay: 0.030-0.058sec
2022-06-26 21:13:34,769 INFO: Prediction for the past 1.000sec: False [0.02] |
    ↪ delay: 0.031-0.057sec
2022-06-26 21:13:34,800 INFO: Prediction for the past 1.000sec: False [0.02] |
    ↪ delay: 0.030-0.052sec
2022-06-26 21:13:34,831 INFO: Prediction for the past 1.000sec: False [0.02] |
    ↪ delay: 0.031-0.069sec
2022-06-26 21:13:34,861 INFO: Prediction for the past 1.000sec: False [0.03] |
    ↪ delay: 0.030-0.059sec
2022-06-26 21:13:34,893 INFO: Prediction for the past 1.000sec: False [0.10] |
    ↪ delay: 0.032-0.061sec
2022-06-26 21:13:34,925 INFO: Prediction for the past 1.000sec: False [0.18] |
    ↪ delay: 0.032-0.062sec
2022-06-26 21:13:34,955 INFO: Prediction for the past 1.000sec: False [0.23] |
    ↪ delay: 0.030-0.059sec
2022-06-26 21:13:34,984 INFO: Prediction for the past 1.000sec: False [0.27] |
    ↪ delay: 0.030-0.058sec
2022-06-26 21:13:35,014 INFO: Prediction for the past 1.000sec: False [0.32] |
    ↪ delay: 0.030-0.057sec
2022-06-26 21:13:35,048 INFO: Prediction for the past 1.000sec: False [0.38] |
    ↪ delay: 0.034-0.064sec
2022-06-26 21:13:35,078 INFO: Prediction for the past 1.000sec: False [0.47] |
    ↪ delay: 0.030-0.061sec
2022-06-26 21:13:35,107 INFO: Prediction for the past 1.000sec: True [0.95] |
    ↪ delay: 0.030-0.058sec
2022-06-26 21:13:35,139 INFO: Prediction for the past 1.000sec: True [0.96] |
    ↪ delay: 0.032-0.060sec
```

```
2022-06-26 21:13:35,169 INFO: Prediction for the past 1.000sec: True [0.98] |
    ↪ delay: 0.030-0.060sec
2022-06-26 21:13:35,199 INFO: Prediction for the past 1.000sec: True [0.98] |
    ↪ delay: 0.030-0.058sec
2022-06-26 21:13:35,228 INFO: Prediction for the past 1.000sec: True [0.98] |
    ↪ delay: 0.030-0.057sec
2022-06-26 21:13:35,258 INFO: Prediction for the past 1.000sec: True [0.99] |
    ↪ delay: 0.030-0.059sec
2022-06-26 21:13:35,293 INFO: Prediction for the past 1.000sec: True [0.99] |
    ↪ delay: 0.035-0.060sec
2022-06-26 21:13:35,323 INFO: Prediction for the past 1.000sec: True [0.99] |
    ↪ delay: 0.030-0.066sec
2022-06-26 21:13:35,354 INFO: Prediction for the past 1.000sec: True [0.99] |
    ↪ delay: 0.031-0.059sec
2022-06-26 21:13:35,386 INFO: Prediction for the past 1.000sec: True [0.99] |
    ↪ delay: 0.032-0.060sec
2022-06-26 21:13:35,415 INFO: Prediction for the past 1.000sec: True [0.99] |
    ↪ delay: 0.030-0.059sec
2022-06-26 21:13:35,445 INFO: Prediction for the past 1.000sec: True [0.99] |
    ↪ delay: 0.030-0.058sec
2022-06-26 21:13:35,475 INFO: Prediction for the past 1.000sec: True [0.99] |
    ↪ delay: 0.030-0.059sec
2022-06-26 21:13:35,505 INFO: Prediction for the past 1.000sec: True [0.99] |
    ↪ delay: 0.030-0.060sec
2022-06-26 21:13:35,537 INFO: Prediction for the past 1.000sec: True [0.99] |
    ↪ delay: 0.032-0.058sec
2022-06-26 21:13:35,567 INFO: Prediction for the past 1.000sec: True [0.99] |
    ↪ delay: 0.030-0.060sec
2022-06-26 21:13:35,599 INFO: Prediction for the past 1.000sec: True [0.99] |
    ↪ delay: 0.031-0.061sec
2022-06-26 21:13:35,629 INFO: Prediction for the past 1.000sec: True [1.00] |
    ↪ delay: 0.030-0.060sec
2022-06-26 21:13:35,659 INFO: Prediction for the past 1.000sec: True [0.99] |
    ↪ delay: 0.030-0.058sec
2022-06-26 21:13:35,690 INFO: Prediction for the past 1.000sec: True [0.99] |
    ↪ delay: 0.030-0.059sec
2022-06-26 21:13:35,720 INFO: Prediction for the past 1.000sec: True [1.00] |
    ↪ delay: 0.030-0.058sec
2022-06-26 21:13:35,750 INFO: Prediction for the past 1.000sec: True [1.00] |
    ↪ delay: 0.030-0.057sec
2022-06-26 21:13:35,780 INFO: Prediction for the past 1.000sec: True [1.00] |
    ↪ delay: 0.030-0.058sec
2022-06-26 21:13:35,809 INFO: Prediction for the past 1.000sec: True [1.00] |
    ↪ delay: 0.030-0.057sec
2022-06-26 21:13:35,839 INFO: Prediction for the past 1.000sec: True [1.00] |
    ↪ delay: 0.030-0.059sec
2022-06-26 21:13:35,869 INFO: Prediction for the past 1.000sec: True [1.00] |
    ↪ delay: 0.030-0.059sec
2022-06-26 21:13:35,900 INFO: Prediction for the past 1.000sec: True [1.00] |
    ↪ delay: 0.030-0.059sec
2022-06-26 21:13:35,930 INFO: Prediction for the past 1.000sec: True [1.00] |
    ↪ delay: 0.030-0.058sec
2022-06-26 21:13:35,960 INFO: Prediction for the past 1.000sec: True [1.00] |
    ↪ delay: 0.030-0.059sec
```

```
2022-06-26 21:13:35,990 INFO: Prediction for the past 1.000sec: True [1.00] |
    ↪ delay: 0.030-0.058sec
2022-06-26 21:13:36,019 INFO: Prediction for the past 1.000sec: True [1.00] |
    ↪ delay: 0.030-0.056sec
2022-06-26 21:13:36,049 INFO: Prediction for the past 1.000sec: True [1.00] |
    ↪ delay: 0.030-0.058sec
2022-06-26 21:13:36,082 INFO: Prediction for the past 1.000sec: True [1.00] |
    ↪ delay: 0.033-0.062sec
2022-06-26 21:13:36,114 INFO: Prediction for the past 1.000sec: True [0.99] |
    ↪ delay: 0.032-0.063sec
2022-06-26 21:13:36,146 INFO: Prediction for the past 1.000sec: True [0.99] |
    ↪ delay: 0.032-0.062sec
2022-06-26 21:13:36,179 INFO: Prediction for the past 1.000sec: True [0.99] |
    ↪ delay: 0.032-0.063sec
2022-06-26 21:13:36,211 INFO: Prediction for the past 1.000sec: True [0.99] |
    ↪ delay: 0.032-0.061sec
2022-06-26 21:13:36,243 INFO: Prediction for the past 1.000sec: True [0.99] |
    ↪ delay: 0.032-0.063sec
2022-06-26 21:13:36,276 INFO: Prediction for the past 1.000sec: True [0.99] |
    ↪ delay: 0.032-0.061sec
2022-06-26 21:13:36,308 INFO: Prediction for the past 1.000sec: True [0.99] |
    ↪ delay: 0.032-0.065sec
2022-06-26 21:13:36,343 INFO: Prediction for the past 1.000sec: True [0.99] |
    ↪ delay: 0.034-0.062sec
2022-06-26 21:13:36,376 INFO: Prediction for the past 1.000sec: True [0.99] |
    ↪ delay: 0.033-0.067sec
2022-06-26 21:13:36,408 INFO: Prediction for the past 1.000sec: True [0.99] |
    ↪ delay: 0.032-0.063sec
2022-06-26 21:13:36,440 INFO: Prediction for the past 1.000sec: True [0.99] |
    ↪ delay: 0.032-0.063sec
2022-06-26 21:13:36,473 INFO: Prediction for the past 1.000sec: False [0.08] |
    ↪ delay: 0.032-0.062sec
2022-06-26 21:13:36,505 INFO: Prediction for the past 1.000sec: False [0.03] |
    ↪ delay: 0.032-0.063sec
2022-06-26 21:13:36,538 INFO: Prediction for the past 1.000sec: False [0.02] |
    ↪ delay: 0.032-0.061sec
2022-06-26 21:13:36,570 INFO: Prediction for the past 1.000sec: False [0.01] |
    ↪ delay: 0.032-0.063sec
2022-06-26 21:13:36,603 INFO: Prediction for the past 1.000sec: False [0.01] |
    ↪ delay: 0.033-0.064sec
2022-06-26 21:13:36,635 INFO: Prediction for the past 1.000sec: False [0.01] |
    ↪ delay: 0.032-0.063sec
2022-06-26 21:13:36,668 INFO: Prediction for the past 1.000sec: False [0.01] |
    ↪ delay: 0.033-0.061sec
2022-06-26 21:13:36,701 INFO: Prediction for the past 1.000sec: False [0.01] |
    ↪ delay: 0.033-0.067sec
2022-06-26 21:13:36,736 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.034-0.065sec
2022-06-26 21:13:36,768 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.032-0.064sec
2022-06-26 21:13:36,801 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.033-0.064sec
2022-06-26 21:13:36,834 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.032-0.061sec
```

```
2022-06-26 21:13:36,865 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.030-0.064sec
2022-06-26 21:13:36,894 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.029-0.056sec
2022-06-26 21:13:36,924 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.029-0.056sec
2022-06-26 21:13:36,954 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.030-0.058sec
2022-06-26 21:13:36,984 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.030-0.059sec
2022-06-26 21:13:37,014 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.030-0.058sec
2022-06-26 21:13:37,044 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.030-0.059sec
2022-06-26 21:13:37,075 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.030-0.059sec
2022-06-26 21:13:37,105 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.030-0.058sec
2022-06-26 21:13:37,135 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.031-0.058sec
2022-06-26 21:13:37,166 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.030-0.060sec
2022-06-26 21:13:37,196 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.030-0.059sec
2022-06-26 21:13:37,228 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.032-0.060sec
2022-06-26 21:13:37,261 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.032-0.062sec
2022-06-26 21:13:37,294 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.032-0.063sec
2022-06-26 21:13:37,326 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.032-0.062sec
2022-06-26 21:13:37,359 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.033-0.065sec
2022-06-26 21:13:37,391 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.032-0.063sec
2022-06-26 21:13:37,425 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.033-0.062sec
2022-06-26 21:13:37,459 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.033-0.066sec
2022-06-26 21:13:37,491 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.032-0.063sec
2022-06-26 21:13:37,524 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.032-0.063sec
2022-06-26 21:13:37,558 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.034-0.063sec
2022-06-26 21:13:37,590 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.032-0.065sec
2022-06-26 21:13:37,624 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.033-0.064sec
2022-06-26 21:13:37,656 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.032-0.061sec
2022-06-26 21:13:37,688 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.032-0.062sec
```

```
2022-06-26 21:13:37,721 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.032-0.064sec
2022-06-26 21:13:37,753 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.032-0.062sec
2022-06-26 21:13:37,786 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.032-0.062sec
2022-06-26 21:13:37,819 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.032-0.065sec
2022-06-26 21:13:37,851 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.032-0.062sec
2022-06-26 21:13:37,883 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.032-0.060sec
2022-06-26 21:13:37,916 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.033-0.066sec
2022-06-26 21:13:37,950 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.034-0.065sec
2022-06-26 21:13:37,968 WARNING: Caught KeyboardInterrupt
2022-06-26 21:13:37,984 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.033-0.065sec
2022-06-26 21:13:38,013 INFO: Prediction for the past 1.000sec: False [0.00] |
    ↪ delay: 0.029-0.048sec
2022-06-26 21:13:39,138 INFO: Stopped gracefully
```

Ausgabe einer Ausführung im Evaluationsmodus

Die hier notierte Beispielausführung des Softwaresystems fand auf einem Computer mit Intel i7 der vierten Generation (x86 Architektur) unter Windows 10 und Python 3.9 statt. Es wurde ein Audiogerät mit einem einzelnen Mikrofon-Sensor verwendet. Das Beispiel wurde mit `...` auf Ausgabezeilen gekürzt, die für den Funktionsnachweis des Evaluationsmodus bedeutend sind.

```
$ jn-seds-cli run --tfmodel='!crnn' evaluation --channels=1
Parameters: {'tfmodel_path': 'C:\\Users\\User\\AppData\\Local\\Programs\\Python\\
    ↪ Python39\\lib\\site-packages\\seds_cli\\res\\models\\crnn', 'threshold':
    ↪ 0.5, 'channels': 1, 'gpu': False, 'infer_model': <InferenceModels.TFLITE:
    ↪  <class 'seds_cli.seds_lib.models.inference_models.TFLiteInferenceModel'
    ↪ >>, 'saved_model': <SavedModels.CRNN: <class 'seds_cli.seds_lib.models.
    ↪ saved_models.CrnnSavedModel'>>, 'storage_length': 0, 'save_records':
    ↪ False, 'input_device': None, 'output_device': None, 'sample_rate': 16000,
    ↪  'window_length': 1.0, 'frame_length': 0.001, 'callback': None, 'loglevel
    ↪ ': <LogLevels.INFO: 'info'>, 'save_log': False, 'mode': None, 'silent':
    ↪ False, 'use_input': True, 'use_output': False, 'wav_file': None, '
    ↪ annotation_file': None}
2022-06-29 11:08:26,750 INFO: Evaluation mode selected
2022-06-29 11:08:26,752 INFO: Environment variable CUDA_VISIBLE_DEVICES is set to
    ↪  -1
2022-06-29 11:08:26,752 INFO: Evaluation mode selected without using the direct
    ↪ input of the wave file
2022-06-29 11:08:26,753 INFO: AudioStorage was initialized for always storing the
    ↪  last 0 seconds of received audio samples.
2022-06-29 11:08:26,893 INFO: Number of installed sound devices: 28
...
```

```
2022-06-29 11:08:26,911 INFO: Reading wave and csv file... (may take a few
    ↪ seconds)
...
2022-06-29 11:08:27,059 INFO: Num GPUs Available: 0
2022-06-29 11:08:27,059 INFO: Processing device 0: PhysicalDevice(name='/
    ↪ physical_device:CPU:0', device_type='CPU')
2022-06-29 11:08:27,060 INFO: Loading crnn model from: C:\Users\User\AppData\
    ↪ Local\Programs\Python\Python39\lib\site-packages\seds_cli\res\models\crnn
    ↪  with threshold: 0.5
2022-06-29 11:08:27,061 INFO: The corresponding saved model was already converted
    ↪  into the desired inference format in a previous execution. If frame and
    ↪ window configuration is not changed, it can be used seamlessly without re
    ↪ -converting.
> Use previously converted tflite model? (Y/n)
y
2022-06-29 11:08:28,895 INFO: Existing model C:\Users\User\AppData\Local\Programs
    ↪ \Python\Python39\lib\site-packages\seds_cli\res\models\crnn\converted-
    ↪ model.tflite will be used.
INFO: Created TensorFlow Lite delegate for select TF ops.
INFO: TfLiteFlexDelegate delegate: 6 nodes delegated out of 157 nodes with 4
    ↪ partitions.

2022-06-29 11:08:28,899 INFO: Press Ctrl+C or Interrupt the Kernel
...
2022-06-29 11:09:40,218 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.00] played, False [0.00] ground-truth | delay: 0.016-
    ↪ 0.055sec
2022-06-29 11:09:40,251 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.00] played, False [0.00] ground-truth | delay: 0.018-
    ↪ 0.044sec
2022-06-29 11:09:40,281 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.00] played, False [0.00] ground-truth | delay: 0.015-
    ↪ 0.051sec
2022-06-29 11:09:40,311 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.00] played, False [0.00] ground-truth | delay: 0.016-
    ↪ 0.045sec
2022-06-29 11:09:40,343 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.00] played, False [0.00] ground-truth | delay: 0.016-
    ↪ 0.054sec
2022-06-29 11:09:40,372 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.00] played, False [0.00] ground-truth | delay: 0.016-
    ↪ 0.043sec
2022-06-29 11:09:40,406 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.00] played, False [0.00] ground-truth | delay: 0.019-
    ↪ 0.045sec
2022-06-29 11:09:40,435 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.00] played, False [0.00] ground-truth | delay: 0.017-
    ↪ 0.056sec
2022-06-29 11:09:40,466 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.00] played, False [0.00] ground-truth | delay: 0.018-
    ↪ 0.044sec
2022-06-29 11:09:40,496 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.00] played, False [0.00] ground-truth | delay: 0.017-
    ↪ 0.043sec
```

```
2022-06-29 11:09:40,525 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.00] played, True [0.02] ground-truth | delay: 0.016-
    ↪ 0.042sec
2022-06-29 11:09:40,555 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.00] played, True [0.06] ground-truth | delay: 0.017-
    ↪ 0.056sec
2022-06-29 11:09:40,587 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.01] played, True [0.09] ground-truth | delay: 0.018-
    ↪ 0.044sec
2022-06-29 11:09:40,620 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.03] played, True [0.11] ground-truth | delay: 0.020-
    ↪ 0.046sec
2022-06-29 11:09:40,652 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.08] played, True [0.15] ground-truth | delay: 0.017-
    ↪ 0.056sec
2022-06-29 11:09:40,682 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.09] played, True [0.19] ground-truth | delay: 0.016-
    ↪ 0.055sec
2022-06-29 11:09:40,716 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.09] played, True [0.22] ground-truth | delay: 0.019-
    ↪ 0.045sec
2022-06-29 11:09:40,746 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.01] played, True [0.24] ground-truth | delay: 0.017-
    ↪ 0.043sec
2022-06-29 11:09:40,776 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.01] played, True [0.28] ground-truth | delay: 0.016-
    ↪ 0.055sec
2022-06-29 11:09:40,805 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.01] played, True [0.31] ground-truth | delay: 0.016-
    ↪ 0.042sec
2022-06-29 11:09:40,835 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.01] played, True [0.34] ground-truth | delay: 0.017-
    ↪ 0.043sec
2022-06-29 11:09:40,865 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.01] played, True [0.37] ground-truth | delay: 0.016-
    ↪ 0.055sec
2022-06-29 11:09:40,893 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.01] played, True [0.40] ground-truth | delay: 0.015-
    ↪ 0.041sec
2022-06-29 11:09:40,922 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.01] played, True [0.43] ground-truth | delay: 0.016-
    ↪ 0.042sec
2022-06-29 11:09:40,952 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.01] played, True [0.47] ground-truth | delay: 0.015-
    ↪ 0.054sec
2022-06-29 11:09:40,981 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.01] played, True [0.49] ground-truth | delay: 0.016-
    ↪ 0.042sec
2022-06-29 11:09:41,010 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.01] played, True [0.51] ground-truth | delay: 0.015-
    ↪ 0.038sec
2022-06-29 11:09:41,038 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.01] played, True [0.54] ground-truth | delay: 0.015-
    ↪ 0.044sec
```

```
2022-06-29 11:09:41,069 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.01] played, True [0.57] ground-truth | delay: 0.018-
    ↪ 0.044sec
2022-06-29 11:09:41,100 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.01] played, True [0.60] ground-truth | delay: 0.018-
    ↪ 0.045sec
2022-06-29 11:09:41,128 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.02] played, True [0.63] ground-truth | delay: 0.015-
    ↪ 0.053sec
2022-06-29 11:09:41,160 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.02] played, True [0.66] ground-truth | delay: 0.018-
    ↪ 0.044sec
2022-06-29 11:09:41,192 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.03] played, True [0.69] ground-truth | delay: 0.017-
    ↪ 0.047sec
2022-06-29 11:09:41,224 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.09] played, True [0.72] ground-truth | delay: 0.016-
    ↪ 0.051sec
2022-06-29 11:09:41,257 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, False [0.17] played, True [0.76] ground-truth | delay: 0.017-
    ↪ 0.056sec
2022-06-29 11:09:41,290 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, True [0.68] played, True [0.79] ground-truth | delay: 0.019-
    ↪ 0.045sec
2022-06-29 11:09:41,321 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, True [0.78] played, True [0.82] ground-truth | delay: 0.017-
    ↪ 0.049sec
2022-06-29 11:09:41,352 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, True [0.85] played, True [0.85] ground-truth | delay: 0.016-
    ↪ 0.049sec
2022-06-29 11:09:41,383 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, True [0.97] played, True [0.89] ground-truth | delay: 0.017-
    ↪ 0.053sec
2022-06-29 11:09:41,412 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, True [0.99] played, True [0.92] ground-truth | delay: 0.016-
    ↪ 0.045sec
2022-06-29 11:09:41,443 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, True [0.99] played, True [0.95] ground-truth | delay: 0.016-
    ↪ 0.042sec
2022-06-29 11:09:41,474 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, True [1.00] played, True [0.98] ground-truth | delay: 0.016-
    ↪ 0.055sec
2022-06-29 11:09:41,504 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, True [0.99] played, True [1.00] ground-truth | delay: 0.016-
    ↪ 0.042sec
2022-06-29 11:09:41,533 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, True [0.99] played, True [1.00] ground-truth | delay: 0.016-
    ↪ 0.042sec
2022-06-29 11:09:41,563 INFO: Prediction of the past 1.000sec: False [0.01]
    ↪ received, True [0.99] played, True [1.00] ground-truth | delay: 0.015-
    ↪ 0.041sec
2022-06-29 11:09:41,593 INFO: Prediction of the past 1.000sec: False [0.01]
    ↪ received, True [0.55] played, True [1.00] ground-truth | delay: 0.016-
    ↪ 0.055sec
```

```
2022-06-29 11:09:41,625 INFO: Prediction of the past 1.000sec: False [0.01]
    ↪ received, True [0.58] played, True [1.00] ground-truth | delay: 0.017-
    ↪ 0.043sec
2022-06-29 11:09:41,655 INFO: Prediction of the past 1.000sec: False [0.01]
    ↪ received, True [0.63] played, True [1.00] ground-truth | delay: 0.016-
    ↪ 0.042sec
2022-06-29 11:09:41,685 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, True [0.64] played, True [1.00] ground-truth | delay: 0.017-
    ↪ 0.056sec
2022-06-29 11:09:41,716 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, True [0.69] played, True [1.00] ground-truth | delay: 0.016-
    ↪ 0.042sec
2022-06-29 11:09:41,746 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, True [0.83] played, True [1.00] ground-truth | delay: 0.017-
    ↪ 0.043sec
2022-06-29 11:09:41,776 INFO: Prediction of the past 1.000sec: False [0.00]
    ↪ received, True [0.94] played, True [1.00] ground-truth | delay: 0.016-
    ↪ 0.055sec
2022-06-29 11:09:41,804 INFO: Prediction of the past 1.000sec: False [0.01]
    ↪ received, True [0.95] played, True [1.00] ground-truth | delay: 0.015-
    ↪ 0.041sec
2022-06-29 11:09:41,834 INFO: Prediction of the past 1.000sec: False [0.01]
    ↪ received, True [0.98] played, True [1.00] ground-truth | delay: 0.015-
    ↪ 0.041sec
2022-06-29 11:09:41,862 INFO: Prediction of the past 1.000sec: False [0.01]
    ↪ received, True [0.95] played, True [1.00] ground-truth | delay: 0.015-
    ↪ 0.054sec
2022-06-29 11:09:41,892 INFO: Prediction of the past 1.000sec: False [0.01]
    ↪ received, True [0.91] played, True [1.00] ground-truth | delay: 0.015-
    ↪ 0.041sec
2022-06-29 11:09:41,920 INFO: Prediction of the past 1.000sec: False [0.02]
    ↪ received, True [0.91] played, True [1.00] ground-truth | delay: 0.015-
    ↪ 0.038sec
2022-06-29 11:09:41,948 INFO: Prediction of the past 1.000sec: False [0.04]
    ↪ received, True [0.93] played, True [1.00] ground-truth | delay: 0.015-
    ↪ 0.044sec
2022-06-29 11:09:41,980 INFO: Prediction of the past 1.000sec: False [0.10]
    ↪ received, True [0.96] played, True [1.00] ground-truth | delay: 0.018-
    ↪ 0.044sec
2022-06-29 11:09:42,010 INFO: Prediction of the past 1.000sec: True [0.93]
    ↪ received, True [0.99] played, True [1.00] ground-truth | delay: 0.017-
    ↪ 0.047sec
2022-06-29 11:09:42,038 INFO: Prediction of the past 1.000sec: True [0.97]
    ↪ received, True [0.99] played, True [1.00] ground-truth | delay: 0.015-
    ↪ 0.050sec
2022-06-29 11:09:42,069 INFO: Prediction of the past 1.000sec: True [0.90]
    ↪ received, True [1.00] played, True [1.00] ground-truth | delay: 0.018-
    ↪ 0.044sec
2022-06-29 11:09:42,099 INFO: Prediction of the past 1.000sec: False [0.30]
    ↪ received, True [1.00] played, True [1.00] ground-truth | delay: 0.018-
    ↪ 0.044sec
2022-06-29 11:09:42,127 INFO: Prediction of the past 1.000sec: False [0.32]
    ↪ received, True [1.00] played, True [1.00] ground-truth | delay: 0.015-
    ↪ 0.054sec
```

```
2022-06-29 11:09:42,159 INFO: Prediction of the past 1.000sec: False [0.35]
    ↪ received, True [1.00] played, True [1.00] ground-truth | delay: 0.018-
    ↪ 0.044sec
2022-06-29 11:09:42,191 INFO: Prediction of the past 1.000sec: False [0.36]
    ↪ received, True [1.00] played, True [1.00] ground-truth | delay: 0.016-
    ↪ 0.048sec
2022-06-29 11:09:42,222 INFO: Prediction of the past 1.000sec: False [0.38]
    ↪ received, True [1.00] played, True [1.00] ground-truth | delay: 0.016-
    ↪ 0.045sec
2022-06-29 11:09:42,252 INFO: Prediction of the past 1.000sec: False [0.41]
    ↪ received, True [0.99] played, True [1.00] ground-truth | delay: 0.016-
    ↪ 0.059sec
2022-06-29 11:09:42,282 INFO: Prediction of the past 1.000sec: False [0.44]
    ↪ received, True [0.99] played, True [1.00] ground-truth | delay: 0.016-
    ↪ 0.042sec
2022-06-29 11:09:42,311 INFO: Prediction of the past 1.000sec: False [0.47]
    ↪ received, True [0.99] played, True [1.00] ground-truth | delay: 0.016-
    ↪ 0.042sec
2022-06-29 11:09:42,341 INFO: Prediction of the past 1.000sec: False [0.48]
    ↪ received, True [0.99] played, True [1.00] ground-truth | delay: 0.016-
    ↪ 0.042sec
2022-06-29 11:09:42,372 INFO: Prediction of the past 1.000sec: True [0.52]
    ↪ received, True [0.98] played, True [1.00] ground-truth | delay: 0.015-
    ↪ 0.054sec
2022-06-29 11:09:42,401 INFO: Prediction of the past 1.000sec: True [0.54]
    ↪ received, True [0.96] played, True [1.00] ground-truth | delay: 0.016-
    ↪ 0.040sec
2022-06-29 11:09:42,433 INFO: Prediction of the past 1.000sec: True [0.56]
    ↪ received, True [0.96] played, True [1.00] ground-truth | delay: 0.017-
    ↪ 0.045sec
2022-06-29 11:09:42,463 INFO: Prediction of the past 1.000sec: True [0.60]
    ↪ received, True [0.96] played, True [1.00] ground-truth | delay: 0.016-
    ↪ 0.055sec
2022-06-29 11:09:42,496 INFO: Prediction of the past 1.000sec: True [0.64]
    ↪ received, True [0.95] played, True [1.00] ground-truth | delay: 0.017-
    ↪ 0.043sec
2022-06-29 11:09:42,527 INFO: Prediction of the past 1.000sec: True [0.65]
    ↪ received, True [0.97] played, True [1.00] ground-truth | delay: 0.016-
    ↪ 0.042sec
2022-06-29 11:09:42,557 INFO: Prediction of the past 1.000sec: True [0.70]
    ↪ received, True [0.97] played, True [1.00] ground-truth | delay: 0.015-
    ↪ 0.054sec
2022-06-29 11:09:42,590 INFO: Prediction of the past 1.000sec: True [0.75]
    ↪ received, True [0.98] played, True [1.00] ground-truth | delay: 0.019-
    ↪ 0.045sec
2022-06-29 11:09:42,620 INFO: Prediction of the past 1.000sec: True [0.83]
    ↪ received, True [0.99] played, True [1.00] ground-truth | delay: 0.017-
    ↪ 0.049sec
2022-06-29 11:09:42,650 INFO: Prediction of the past 1.000sec: True [0.96]
    ↪ received, True [0.98] played, True [1.00] ground-truth | delay: 0.016-
    ↪ 0.049sec
2022-06-29 11:09:42,682 INFO: Prediction of the past 1.000sec: True [0.98]
    ↪ received, True [0.97] played, True [1.00] ground-truth | delay: 0.017-
    ↪ 0.047sec
```

```
2022-06-29 11:09:42,711 INFO: Prediction of the past 1.000sec: True [0.99]
    ↪ received, True [0.92] played, True [1.00] ground-truth | delay: 0.016-
    ↪ 0.049sec
2022-06-29 11:09:42,742 INFO: Prediction of the past 1.000sec: True [0.99]
    ↪ received, True [0.93] played, True [1.00] ground-truth | delay: 0.016-
    ↪ 0.041sec
2022-06-29 11:09:42,772 INFO: Prediction of the past 1.000sec: True [0.99]
    ↪ received, True [0.95] played, True [1.00] ground-truth | delay: 0.016-
    ↪ 0.058sec
2022-06-29 11:09:42,802 INFO: Prediction of the past 1.000sec: True [0.99]
    ↪ received, True [0.95] played, True [1.00] ground-truth | delay: 0.015-
    ↪ 0.041sec
2022-06-29 11:09:42,830 INFO: Prediction of the past 1.000sec: True [0.99]
    ↪ received, True [0.97] played, True [1.00] ground-truth | delay: 0.015-
    ↪ 0.040sec
2022-06-29 11:09:42,858 INFO: Prediction of the past 1.000sec: True [0.99]
    ↪ received, True [0.98] played, True [1.00] ground-truth | delay: 0.015-
    ↪ 0.042sec
2022-06-29 11:09:42,890 INFO: Prediction of the past 1.000sec: True [0.99]
    ↪ received, True [0.99] played, True [1.00] ground-truth | delay: 0.018-
    ↪ 0.044sec
2022-06-29 11:09:42,920 INFO: Prediction of the past 1.000sec: True [0.99]
    ↪ received, True [0.99] played, True [1.00] ground-truth | delay: 0.017-
    ↪ 0.048sec
2022-06-29 11:09:42,949 INFO: Prediction of the past 1.000sec: True [0.99]
    ↪ received, True [0.99] played, True [1.00] ground-truth | delay: 0.015-
    ↪ 0.049sec
2022-06-29 11:09:42,981 INFO: Prediction of the past 1.000sec: True [0.99]
    ↪ received, True [1.00] played, True [1.00] ground-truth | delay: 0.018-
    ↪ 0.045sec
2022-06-29 11:09:43,010 INFO: Prediction of the past 1.000sec: True [0.97]
    ↪ received, True [0.99] played, True [1.00] ground-truth | delay: 0.015-
    ↪ 0.051sec
...
```

Eidesstattliche Erklärung

Hiermit versichere ich, dass meine Bachelorarbeit „Entwicklung eines eingebetteten Systems zur Erkennung akustischer Warnsignale von Fahrradfahrern unter Einsatz künstlicher neuronaler Netzwerke“ („Development of an Embedded System for the Detection of Cyclists’ Acoustic Warning Signals Using Artificial Neural Networks“) selbständig verfasst wurde und dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt wurden. Diese Aussage trifft auch für alle Implementierungen und Dokumentationen im Rahmen dieses Projektes zu.

Potsdam, den 5. Juli 2022,

_____

(Clemens Kubach)