Masterprojektarbeit

Vergleich von CNN und Embedding Klassifikatoren auf dem PhysioNet2016 Datensatz

An der Fachhochschule Dortmund
im Fachbereich Informatik
Studiengang Informatik
Vertiefungsrichtung Medizinische Informatik
erstellte Masterprojektarbeit
zur Erlangung des akademischen Grades
Master of Science

von
Martin Sondermann
geb. am 26.04.1997
Matr.-Nr. 7212408

Betreuer:
Prof. Dr.-Ing. Christoph Friedrich
Zweitbetreuer: M.Sc. Niklas Tschorn
Dortmund, 20. Dezember 2023

Überblick

Kurzfassung

In dieser Masterprojektarbeit wird der Vergleich zwischen CNN und Embedding-basierten Klassifikatoren zur Erkennung von Herzklappenerkrankungen auf Basis des PhysioNet2016 Datensatzes vorgestellt. Der Fokus liegt auf der Analyse und Bewertung der Leistungsfähigkeit von YAMNet und VGGish Modellen als Embedding-Extraktoren im Vergleich zu einem eigenständigen CNN. Das CNN ist speziell für die Anwendung erstellt und basiert auf Mel-Spektrogramm Features. Die Embedding werden mit einem kNN Klassifikator verarbeitet. Die Ergebnisse zeigen, dass das CNN in Bezug auf Genauigkeit, MCC und AUROC eindeutig überlegen ist. Es zeichnet sich durch eine effektivere Klassifikationsleistung aus, trotz einer leichten Neigung zum Overfitting. Im Gegensatz dazu haben die Embedding-Modelle, insbesondere YAMNet, Schwierigkeiten bei der genauen Klassifizierung positiver Fälle, zeigten jedoch eine hohe Spezifität. Das YAMNet-Modell schneidet marginal besser ab als das VGGish-Modell, beide sind jedoch effizient in der Erkennung negativer Klassen. Die Untersuchung legt nahe, dass eine Kombination der Stärken von CNN und Embedding-Methoden in klinischen Anwendungen vorteilhaft sein könnte. Diese Arbeit unterstreicht die Bedeutung der richtigen Wahl von Metriken und der Berücksichtigung von Klassenungleichgewichten. Die erzielten Erkenntnisse bieten wertvolle Ansätze für die Weiterentwicklung von Klassifikationsmethoden in der medizinischen Diagnostik.

Abstract

This study explores the effectiveness of CNNs and Embedding-based classifiers in identifying heart valve diseases using the PhysioNet2016 dataset. The research focuses on comparing the performance of YAMNet and VGGish models, employed as Embedding extractors followed by a kNN classifier, against a standalone CNN model. The findings reveal that the CNN model consistently outperforms the Embedding classifiers in terms of accuracy metrics such as MCC, Accuracy, and AUROC. While the CNN demonstrates balanced performance in identifying true positives and negatives, a slight inclination towards overfitting is observed. The Embedding models, especially YAMNet, exhibit high specificity but lower recall and precision, indicating challenges in accurate classification. A notable distinction is observed between the YAMNet and VGGish models, with YAMNet showing marginally better performance. Both models are effective in identifying negative cases but face difficulties in accurately classifying positive cases, potentially due to limitations in capturing the complexity of audio data and addressing dataset class imbalance. The study underscores the potential advantages of combining the of both CNN and Embedding methodologies to enhance effectiveness in various clinical applications. Future research might focus on optimizing these models to further improve their accuracy and sensitivity, particularly considering the temporal dependencies in the data.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen

AUC Area-under-the-Curve

AUROC *Area-under-the-Receiver-Operating-Characteristic-Curve*

BLSTM Bidirectional Long Short-Term Memory

CM confusion matrix („Verwechslungsmatrix“)

CNN Convolutional Neural Network („Gefaltetes Neuronales Netzwerk“)

dict Python dictionary Objekt

FC-Layer Fully-Connected-Layer

FN False Negative

FP False Positive

FPR False positive r ate

kNN k-nearest Neighbour

MCC Matthews Correlation Coefficient

NLR negative likelihood ratio

NN Nearest neighbour („Nächster Nachbar“)

PCG Phonocardiogramm

PLR positive likelihood ratio

RNN Recurrent Neural Network

ROC-Kurve *Receiver Operating Characteristic Curve*

SGD Stochastic Gradient Descent

STFT Short-Term-Fourier-Transformation („Kurzzeit-Fourier-Transformation“)

TN True Negative

TNR true negative rate

TP True Positive

TPR true positive rate

WANDB *Weights and Biases*

YAMNet *Yet Another Mobile Audio Network*

Kapitel 1

Einleitung

1.1 Motivation

Nach Clifford et al. (2016) sind Herzkrankheiten ein weltweit verbreitetes Thema, viele Menschen leiden unter ihnen und sie sind eine der häufigsten Todesursachen weltweit. Diese werden von einem Mediziner traditionell durch Abhören der Herzgeräusche mittels Stethoskop auf der Patientenbrustwand diagnostiziert. Dies sei aber insbesondere in abgelegenen Regionen, wo wenig Ärzte auf einen Patienten kommen und schlecht verfügbar sind, nicht ideal. Ziel sei eine automatischen Detektion von auffälligen Herzgeräuschen, wodurch der Patient untersucht werden kann, ohne dass ein Arzt anwesend sein muss.

Mit der fortschreitenden Digitalisierung im Gesundheitswesen sind digitale Auskultationsgeräte entwickelt worden, die ein Phonocardiogramm (PCG) erstellen (Raghu et al. , 2015). Diese Geräte ermöglichen eine genaue Aufzeichnung der Herzgeräusche, welche dann für die weitere Analyse verwendet werden können (Chowdhury et al. , 2019). In jüngerer Zeit haben Machine Learning Anwendungen, insbesondere tiefe Lernmodelle mit einem Convolutional Neural Network (CNN, „Gefaltetes Neuronales Netzwerk“), erhebliche Fortschritte in der automatischen Analyse von PCG-Daten erzielt (Chen et al. , 2021). Aufbauend auf diesen, gibt es moderne Verfahren die eine Audioklassifikation mittels Embedding extrahierenden Modellen anstreben (Maity et al. , 2023).

Ziel dieser hier vorgestellten Arbeit ist es mehrere Klassifikatoren für solche abnormale Herztöne zu erstellen und diese einzuordnen. Hierbei ist der Vergleich von gewöhnlichen CNN Modellen mit modernen Embedding Extraktoren besonders interessant und wird hier genauer betrachtet.

1.2 Gliederung der Arbeit

Zunächst werden in Kapitel 2 Abschnitt 2.1 die medizinischen Grundlagen genauer erklärt. Gemeinsam mit den sich hieraus ergebenden Komplikationen zur Detektion von malignen Herzgeräuschen. Weitergehend werden in Abschnitt 2.2 die Hintergründe hinter den verwendeten Technologien detaillierter erklärt. Anschließend wird in Kapitel 3 der Datensatz und die generelle Struktur der Anwendung, zusammen mit Erweiterungen die den Trainings- und Experiment-Verlauf vereinfachen, vorgestellt. Im darauffolgenden Abschnitt 3.6 werden die erarbeiteten Modelle mit den dazugehörigen Features erklärt. Im Kapitel 4 werden die Klassifikationsergebnisse jedes Modells bewertet und anschließend untereinander verglichen. Anschließend findet in Kapitel 5 eine Diskussion statt, wo Vor und Nachteile der jeweiligen Modelle besprochen werden. Zudem aus der Motivationsfrage hervorgehend, wird ein Vergleich von CNN Modellen gegen Embedding Extraktoren aufgestellt. Generelle Schwachstellen der Anwendung und des Datensatzes werden angesprochen und ein Ausblick in Kapitel 6 für weitere Arbeiten wird in Aussicht gestellt.

Kapitel 2

Hintergrund

2.1 Medizinische Grundlagen

2.1.1 Herztöne

Herzphasen und Geräuschherkunft

Von Liu et al. (2016) werden die verschiedenen Herzphasen, welche relevant für eine Beurteilung durch Geräusche sind, beschrieben und hier wiedergegeben. Die Herztöne entstehen primär durch das Öffnen und Schließen der vier Herzklappen: Mitralklappe, Trikuspidalklappe, Aortenklappe und Pulmonalklappe. Neben der Klappenbewegung tragen auch der turbulente Blutfluss und das Zusammenspiel von Herz- und Gefäßwänden zur Klangentstehung bei. Die so genannten primären und sekundären Fundamentalen Herztöne werden S1 und S2 genannt. Der S1-Ton ist das Ergebnis des Schließens der Atrioventrikularklappen, nämlich der Mitralklappe und der Trikuspidalklappe, und markiert den Beginn der Systole, der Kontraktionsphase des Herzens. Der S2-Ton erfolgt durch das Schließen der Semilunarklappen, der Aortenklappe und der Pulmonalklappe, und markiert den Beginn der Diastole. Seltener sind die S3- und S4-Töne, die in bestimmten Krankheitsbildern auftreten und oft als pathologisch angesehen werden könnten.

2.1.2 Herzklappenstörung

Herzklappenstörungen können meist in zwei Hauptkategorien unterteilt werden: Stenosen und Regurgitation (Dominguez-Morales et al. , 2018). Bei einer Stenose ist die Öffnung der Klappe verengt, was den Blutfluss behindert und Turbulenzen erzeugt (Singh et al. , 2020). Eine Regurgitation tritt auf, wenn die Klappe nicht vollständig schließt, was zu einem ungewollten Rückfluss des Blutes führt und ebenfalls die normale Funktionsweise des Herzens stört (Zoghbi et al. , 2017). In beiden Fällen entstehen charakteristische Herzgeräusche, welche von den üblichen Tönen abweichen, die als Störgeräusche bezeichnet werden und von einem erfahrenen Arzt erkannt werden können (Mishra et al. , 2019).

Der S3-Ton tritt kurz nach dem S2-Ton auf, entsteht wenn das Blut in die Ventrikel fließt und wird oft als *ventrikuläres Füllungsgeräusch* bezeichnet (Mondal et al. , 2013). Bei jüngeren Personen kann ein S3-Ton normal sein, jedoch ist er bei älteren Patienten oft ein Zeichen für eine Überladung des Herzen oder eine generelle Dysfunktion (Chowdhury et al. , 2019). Der S4-Ton erfolgt unmittelbar vor dem S1-Ton und entsteht durch die Kontraktion der Vorhöfe eines erkrankten Herzens (Mondal et al. , 2013).

Die CDC (2019) beschreibt die Diagnose und Therapie der Herzklappenstörung. Konkret seien die Ursachen vielfältig und die Symptome unterschiedlich. Hierzu gehören unter anderem Atemlosigkeit, Brustschmerzen, Schlappheit und Schwindel. Diverse Formen der Herzklappenstörung erzeugen unterschiedliche Störgeräusche am Herzschlag. In allen Fällen ist die genaue Identifizierung dieser Töne entscheidend, da sie wichtige diagnostische Hinweise auf den Zustand des Herzens liefern können. Die Analyse, insbesondere in Verbindung mit anderen Symptomen und Untersuchungsergebnissen, spielt eine wesentliche Rolle in der Diagnose und dem Management von Herzerkrankungen. Je nach Lokalisation und Schweregrad der Störung werden verschiedene therapeutische Maßnahmen, von Medikamenten bis hin zu chirurgischen Eingriffen, erforderlich sein.

2.1.3 Diagnoseschwierigkeiten

Nachdem die grundlegenden Aspekte der Herztöne und Herzklappenstörungen erörtert wurden, ist es wichtig, die daraus resultierenden diagnostischen Herausforderungen zu verstehen. Dieser Abschnitt 2.1.3 ist aus der zuvor eingereichten Hausarbeit über den *"PhysioNet2016 Datensatz"* entnommen (Sondermann , 2023). Durch Liu et al. (2016) werden die Herausforderungen der Herzton-Klassifikation erläutert. Diese ist eine komplexe Aufgabe, da verschiedene Komponenten des Herzschlags in den aufgenommenen Audiosignalen identifiziert und unterschieden werden müssen. [...] Konkret stellen die Autoren Liu et al. (2016) dar, dass die normalen Frequenzen der fundamentalen Herztöne gut bekannt sind. S1 hat eine Frequenz von etwa 20 bis 100 Hz, während S2 im Bereich von 50 bis 120 Hz liegt. Störgeräusche durch Herzklappenfehler werden Murmur (zu Deutsch „Murmeln“) genannt. Murmurgeräusche manifestieren sich in bestimmten Frequenzbereichen und können bis zu 600 Hz erreichen. Respiratorische Geräusche, die durch die Atmung verursacht werden, treten typischerweise im Frequenzbereich von 200 bis 700 Hz auf. Nach Maity et al. (2023) ist das der Grund, dass Herzton, Murmur und Atemgeräusche in der Frequenzdomäne nicht immer eindeutig trennbar sind und dies zu Problem bei der Analyse führt.

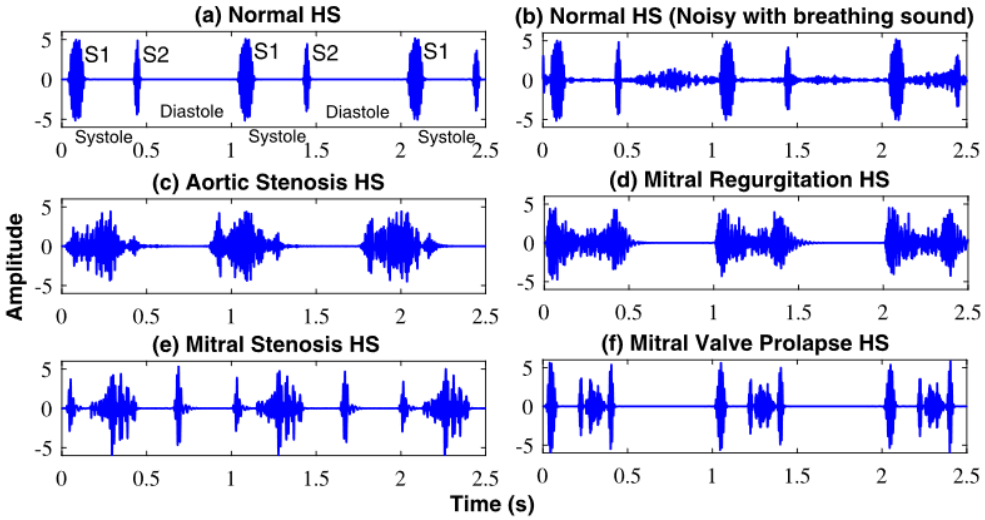

Abbildung 2.1: Vergleich der Audio Amplituden von normalen, abnormalen und Störgeräuschbehafteten PCG Signalen (Maity et al. , 2023)

Aus der Abbildung 2.1 wird erkenntlich, dass beispielsweise der Unterschied der Audiokurven zwischen normal (b) und abnormal (e) (f) relativ gering ausfällt. Durch reines Betrachten der Daten kann eine sichere Unterscheidung zwischen Atemgeräuschen und Herzklappenstörung nicht gegeben werden. Die morphologische Ähnlichkeit von verschiedenen Herzgeräuschen erschwert auch die Trennung in der Zeitdomäne (Liu et al. , 2016). Mehrere Studien haben sich daher mit der automatischen Segmentierung und Klassifizierung von Herzgeräuschen befasst. Eine Herausforderung besteht jedoch darin, dass diese Algorithmen aufgrund des Fehlens eines einheitlichen und validierten Datensatzes nur schwer zu verifizieren sind (Liu et al. , 2016). In diesem Zusammenhang wurde die PhysioNet2016 Challenge initiiert, um das Problem der Herzton-Klassifikation anzugehen (Liu et al. , 2016). Als Motivationsgrund wird von den Autoren unter anderem die geringe Anzahl an verfügbaren Datensätzen, die für die Forschung verwenden werden könnten, angegeben und erklärt: Die „Michigan Heart Sound and Murmur Database (MHSDB)“ enthalte beispielsweise nur 23 Aufnahmen mit einer Gesamtdauer von 1496 Sekunden. Der PASCAL-Datensatz enthält 656 Aufnahmen, jedoch wurden bei der Aufnahme ein Tiefpassfilter bei 195 Hz angewendet, was bestimmte Komponenten eliminiert. Die *„Cardiac Auscultation of Heart Murmurs Database“* enthält nur 64 Aufnahmen und ist zudem nicht öffentlich zugänglich und erfordere Bezahlung. Die Challenge Autoren Liu et al. (2016) erläutern zu den vorhandenen Datenbanken weitere Nachteile. So sind die Aufnahmen oft idealisiert, um Studenten das Auskultieren von Herztönen beizubringen. In der realen Welt sind die aufgenommenen Audiosignale oft von geringerer Qualität und mit Störgeräuschen behaftet. Daher wurde die Erstellung eines umfangreicheren und realistischeren Datensatzes erforderlich, um die Entwicklung und Evaluierung von Herzton-Klassifikationsalgorithmen zu verbessern.

2.2 Machine Learning

2.2.1 CNN-basierte Klassifikation

Von O'Shea und Nash (2015) wird ein umfassender Überblick über die technischen Hintergründe von CNN Klassifikatoren gegeben und in diesem gesamten Abschnitt 2.2.1 wiedergegeben. Sie sind eine spezielle Kategorie von neuronalen Netzwerken, die üblicherweise in der Bild- und Signalverarbeitung eingesetzt werden. Im Gegensatz zu herkömmlichen Methoden, wie einem gewöhnlichem Neuronalen Netzwerk, können CNNs besser lokale räumliche Abhängigkeiten in Daten erfassen, was für die Analyse von Bildern und Sequenzen mit daraus abgeleiteten Bildern sehr nützlich ist. Sie sind daher besonders geeignet für Aufgaben wie Bilderkennung, Textklassifikation und sogar medizinische Diagnoseverfahren, die Bild- oder Audiodaten nutzen.

Arten von Schichten

CNNs bestehen im Allgemeinen aus einer Kombination verschiedener Arten von Schichten. Die Convolutional-Layer sind darauf spezialisiert, lokale Muster in der Eingabematrix (oftmals ein Bild) zu erkennen. Pooling-Layer dienen der Reduzierung zur Dimensionalität der Eingabedaten, was den Rechenaufwand des Netzwerks reduziert. Schließlich sind die Fully-Connected-Layer (FC-Layer) dafür verantwortlich, die extrahierten Merkmale für die abschließende Klassifikation oder Regression zu nutzen.

Vorteile von CNNs sind ihre Fähigkeit zur automatischen Feature-Extraktion. Sie können komplexe Eigenschaften in den Daten erkennen, die für Menschen meist nicht direkt logisch erkennbar sind, und für die Klassifikation nutzen. Ihre Architektur erlaubt es ihnen, sowohl in der Bild- als auch in der Text- und Audiodatenverarbeitung effizient zu sein. Die integrierten Pooling-Layer reduzieren die Dimensionalität der Daten, was die Rechenzeit und den Bedarf an Rechenressourcen verringert. Dadurch werden komplexere Trainingsläufe ermöglicht und tiefere Ver-

bindungen innerhalb der Daten können gelernt werden.

Nachteile von CNNs sind unter anderem der hoher Rechenbedarf, besonders wenn das Netzwerk viele Schichten hat. Sie sind ebenso anfällig für Overfitting wie andere Verfahren, besonders wenn die Datenmenge für das Training klein ist. Overfitting führt dazu, dass das Modell die Trainingsdaten zu gut lernt und nicht gut auf unbekannte Daten generalisiert. Dies kann mit Regularisierungsmethoden angegangen werden.

2.2.2 Trainingsablauf mit CNN

Der Trainingsprozess eines CNN ist komplex und erfordert sorgfältige Datenaufbereitung und Selektion. Zu Beginn wird der gesamte Datensatz in die jeweiligen Trainings-, Validierungs- und Testdaten aufgeteilt (Goodfellow et al. , 2016, Kap. 5). Verschiedene Optimierungsverfahren wie Stochastic Gradient Descent (SGD) oder Adam und deren Variationen können als Optimierer verwendet werden (Kingma und Ba , 2015). Zusätzlich sind Lernpläne (Scheduler) sinnvoll, um die Lernrate während des Trainings anzupassen (Jepkoech et al. , 2021). Frühes Anhalten des Trainings (Early Stopping) und Regularisierungstechniken wie Dropout und L_1/L_2-Regularisierung können implementiert werden, um das Overfitting-Risiko zu verringern (Goodfellow et al. , 2016, Kap. 7) (Srivastava et al. , 2014). Transferlernen ermöglicht es, ein vortrainiertes Modell zu nutzen und nur bestimmte Schichten für die spezifische Aufgabe neu zu trainieren, was sowohl Zeit spart als auch den benötigten Datenaufwand verringert (Maity et al. , 2023).

2.2.3 Embeddings

Embeddings dienen der Umwandlung von hochdimensionalen Daten in einen niedrigerdimensionalen Raum (Almeida und Xexéo , 2023). Zusätzlich können sie dazu verwendet werden, eine verbesserte Klassifikationsleistung zu erzielen, indem sie ein tieferes Verständnis der Datenstruktur ermöglichen (Jepkoech et al. , 2021).

Diese reduzierten Darstellungen seien nützlich für die Transformation von rohen Daten und können die Klassifikationsleistung signifikant verbessern (Hershey et al. , 2017). Sie werden meist durch CNN Modelle erstellt und sind daher in den gleichen Domänen wie Text-, Bild- und Audiobearbeitung nützlich, wo ebenfalls gewöhnliche CNNs Anwendung finden (Alzubaidi et al. , 2021).

Arten von Embeddings

Es gibt verschiedene Arten von Embeddings. Wort-Embeddings für Textklassifikation wie Word2Vec überführen Textuelle Daten in einen Vektorraum und werden unter anderem für Sprachmodelle verwendet (Mikolov et al. , 2013). Von Dogan et al. (2022) werden zudem Bild und Audio Embeddings untersucht. Aus Bildern lassen sich demnach ebenfalls Embeddings extrahieren, welche die relevanten Informationen beinhalten. Diese sind gemeinsam mit Embeddings aus Audiodateien verwendbar, welche wiederum Geräusche in einer kompakten Repräsentierung beschreiben. Zusammen können sie als Semantic Embeddings eingesetzt werden und die Klassifizierungsergebnisse von Audioklassifikatoren verbessern.

Für dieses Projekt sind Audioembeddings von besonderem Interesse da hier untersucht wird, inwieweit sich für medizinische Fragestellungen aus den PCG Daten nutzbare Informationen ableiten lassen. Wie dies konkret funktioniert wird im Folgenden Abschnitt 2.2.4 beschrieben.

2.2.4 Vorstellung der Embedding Extraktoren

Sowohl *Yet Another Mobile Audio Network* (YAMNet) als auch VGGish Feature-Embedding Extraktoren sind durch ein Google Research Team mit TensorFlow entwickelt worden und basieren auf den YouTube-Audioset Datensätzen (Hershey et al. , 2017).

YAMNet

Das YAMNet ist ein vortrainiertes, tiefes CNN-Modell für Klassifizierung von Audiosignalen (Maity et al. , 2023). Es wird unter der Adresse https://github.com/tensorflow/models/tree/master/research/audioset/yamnet (Stand 01.11.2023) zur Verfügung gestellt. Das Modell basiert auf einer MobileNet_v1 Architektur welche mit dem AudioSet Datensatz trainiert wurde und kann 521 verschiedene Label vergeben (Howard et al. , 2017). Es besteht aus 86 Layern mit einer Tiefe von 28 und insgesamt aus 3.75 Millionen Parametern (Maity et al. , 2023). AudioSet ist ein Datensatz, bestehend aus $1,789,621$ 10 Sekunden langen Audiosegmenten von YouTube Videos, etwa 4971 Stunden umfassend und mit 632 verschiedenen Labeln versehen (Gemmeke et al. , 2017). Auf der verlinkten GitHub Seite wird die Funktionsweise und Vorverarbeitung bei YAMNet erklärt:

1. Das Audioinput wird auf 16 kHz mono resampled.
2. Ein Spektrogramm, basierend auf der Magnitude der Short-Term-Fourier-Transformation (STFT, „Kurzzeit-Fourier-Transformation“), mit einer window size von 25ms, window hop von 10ms und einer periodischen Hann-window, wird erstellt.
3. Dieses wird in ein 64 bin Mel-log Spektrogramm übertragen.
4. Input Beispiele für das Netzwerk werden erstellt. Diese bestehen aus 50% überlappenden, 0.96 Sekunden Fenster dieses Spektrogramms. Die Inputgröße eines Beispieles beträgt aufgrund den Einstellungen 96×64.
5. Dieser Input wird in das MobileNet_v1 Modell gegeben und verarbeitet, sodass ein 1024 dimensionales Embedding entsteht.
6. Das Embedding wird optional mit einem Logistic Layer verarbeitet um Ausgabewahrscheinlichkeiten der 521 verschiedenen Klassen zu erhalten.

Das YAMNet Modell ist also in der Lage selbständig Embeddings zu extrahieren und eine Klassifizierung durchzuführen. Oftmals, wie auch in diesem Kontext, bein-

halten die vergebenen Klassen keine medizinische Label wie ein abnormaler Herzton und könnten daher nicht verwendet werden (Howard et al. , 2017). Eine Möglichkeit, wie von den Tensorflow Maintainern angeregt, besteht das Model nachträglich weiter zu trainieren und neue Klassen als möglichen Output festzulegen. Eine andere Option ist die Embeddings so wie sie sind zu verwenden und einer nachträglichen Klassifikation zu unterziehen, die auf einer anderen Methode basiert. In dieser Anwendung wird dies so mit einem k-nearest Neighbour (kNN) Klassifikator durchgeführt. Das optionale Logistic Layer wird verworfen.

VGGish

Grundsätzlich ist das VGGish Netzwerk ähnlich wie das YAMNet aufgebaut, hat allerdings dennoch einige signifikante Unterschiede. Von Koh und Dubnov (2021) wird es genauer vorgestellt: Das VGGish Netzwerk ist eine Variante des zuvor entwickelten VGG Netzwerkes und wurde auf dem YouTube-8M Datensatz trainiert. Es wird unter https://github.com/tensorflow/models/tree/master/research/audioset/vggish (Stand 01.11.2023) zur Verfügung gestellt. Dieser Datensatz umfasst 500.000 Stunden Video und Audiomaterial von YouTube, extrahiert ungefähr 1.9 Milliarden Frames und vergibt ihnen 4800 verschiedene Label (Abu-El-Haija et al. , 2016). Das VGGish Netzwerk ist nach Hershey et al. (2017) eine vereinfachte Version des VGG Netzwerkes, welches auf lediglich einem Auszug des Trainingssatzes trainiert und optimiert wurde. Der generelle Ablauf sei dem von YAMNet sehr ähnlich, allerdings werden die ebenfalls 96×64 großen und 0.96 Sekunden langen Beispielframes nicht überlappend aus dem Mel-Spektrogramm gebildet. Die Ausgabe sind 1×128 große Embeddings, welche hier ebenfalls mit einem kNN Klassifikator weiter verwendet werden.

Ein merklicher Unterschied ist die deutlich höhere Modellgröße und Komplexität des VGGish Extraktors im Vergleich zu YAMNet. So beinhaltet das Modell für den VGGish Extraktor 72.1 Millionen Parameter, während YAMNet lediglich 3.7 Millionen aufweist (Tsalera et al. , 2021).

2.2.5 Integration von Embeddings

Von Maity et al. (2023) wird beschrieben wie Embeddings nahtlos in CNN-Architekturen integriert werden können. So erfolge dies indem an die Embedding Ausgabe neue Schichten eines CNN angehangen werden. Durch das Einbetten der Eingangsdaten in einen niedrigerdimensionalen Embedding Raum wird die Berechnungseffizienz erhöht, ohne signifikante Informationen zu verlieren. Darüber hinaus ist das Fine-Tuning der Embedding-Schichten möglich, wo manche oder alle Schichten nachtrainiert werden, um die spezifischen Anforderungen der jeweiligen Anwendung besser zu erfüllen . Dies ist besonders nützlich, wenn Transferlernen angewendet wird, da die vortrainierten Embedding-Extraktoren dann für die spezifische Aufgabe angepasst sind. Ein Beispiel hierfür wäre die in Maity et al. (2023) vorgestellte Anwendung eines ursprünglich für die Erkennung allgemeiner Audiosignale entwickelten Modells zur spezifischen Erkennung von Herzgeräuschen in medizinischen Anwendungen. In dieser Anwendung wird die Integration anders durchgeführt und im folgenden Abschnitt 2.2.5 vorgestellt.

Nearest Neighbour Clustering

Das Nearest neighbour (NN, „Nächster Nachbar“)-Clustering wird ursprünglich von (Cover und Hart , 1967) beschrieben und von Bhatia und Author (2010) weiter ausgeführt. Die Erkenntnisse werden hier in diesem Abschnitt 2.2.5 wiedergegeben. Es ist eine Technik im Bereich des Machine Learning um ähnliche Datenpunkte in Clustern zu gruppieren. Dies habe den Vorteil, dass die Modellleistung durch die Betrachtung von Clustern, die dadurch ähnliche Eigenschaften aufweisen, verbessert werden kann. Nach der Berechnung der Abstände zwischen den Nachbarpunkten, muss ein Voting durchgeführt werden,zum bestimmen welche Label für den Datenpunkt verwendet werden. Der Parameter der angibt wie viele der nächsten Nachbarpunkte einfließen sollen, wird k genannt. Übliche Werte können $k = 3$ oder beispielsweise $k = 7$ sein. Der Algorithmus wird in dieser Variante deshalb oft kNN genannt. Die Autoren bekräftigen dass es wichtig ist, die Parameter sorgfältig zu wählen und den Clustering-Prozess im Hinblick auf die

spezifischen Anforderungen der Aufgabe zu optimieren. Hierbei werden in deren Ausarbeitung erarbeitet und beschrieben.

Diese Eigenschaften kann man sich bei der Embedding Klassifizierung zu nutze machen. Da der kNN Klassifikator mit beliebig großer Dimensionalität der Inputwerte funktioniert, können die großen Embeddings der Extraktoren direkt als Datenpunkte in den Klassifikator zum trainieren gegeben werden (Cover und Hart , 1967). Nach der zuvor gestellten Annahme, dass die Embeddings die Eigenschaften der Audiosignale kondensiert beschreiben, sollten die kNN Klassifikatoren in der Lage sein diese Informationen sinnvoll zu verwenden. Inwieweit dies zutrifft und im Vergleich zu CNN Netzwerken funktioniert wird hier in diesem Projekt untersucht. Der konkrete Code der Implementierung ist in Quellcode 3.6.1 angegeben.

Kapitel 3

Material und Methoden

3.1 Verwendete Entwicklungsumgebung

Die Entwicklung sowie das Training und die Validierung wurden mit folgendem System durchgeführt:

CPU AMD Ryzen 5600x
GPU Nvidia RTX 3070 FE
OS Windows 11 23H1
RAM 32 GB @ 3600 Mhz.
Speicher 2 Tb. NVMe SSD

Die folgende Tabelle 3.1 gibt einen Überblick über alle individuell installierten Softwarepakete. Hinzu kommen Bibliotheken die jeweils bei der Installation mitgeladen werden. Eine exakte Auflistung der verwendeten Pakete liegt als conda Environment Export im Quellcode B.0.5 vor.

Tabelle 3.1: Auflistung der relevanten verwendeten Software

Name	**Versionsnummer** **Lizenz**	**Verwendung** **URL**
Python	3.10.12	Hauptprogrammiersprache, die für die Implementierung der Anwendungslogik genutzt wird
	PSFL	https://www.python.org/
Numpy	1.23.5	Mathematische Operationen, spezialisiert auf effiziente Matrixmanipulationen
	BSD-3-Clause	https://www.numpy.org/
Pandas	2.1.1	Werkzeug für Datenverwaltung, Datenstrukturen für effiziente Datenanalyse
	BSD-3-Clause	https://pandas.pydata.org/
PyTorch	2.0.1	Framework für maschinelles Lernen, Algorithmen für das Trainieren von Modellen
	BSD-3-Clause	https://pytorch.org/
cuda	11.7	Schnittstelle zu Tensorrechenkernen auf der Grafikkarte
	CUDA EULA	https://developer.nvidia.com/cuda-zone
wandb	0.15.11	Metrik Tracker und Auswertung, Hyperparameteroptimierung
	MIT	https://wandb.ai/site
audiomentations	0.33.0	Augmentierung von Audiosignalen und Spektrogrammen
	MIT	https://github.com/iver56/audiomentations
matplotlib	3.8.0	Bibliothek für die Erstellung von Plots und grafischen Darstellungen von Daten
	PSFL	https://matplotlib.org/
scikit-learn	1.3.1	Trainingsdaten manipulation wie Test splits und k-fold Implementierung
	BSD-3-Clause	https://scikit-learn.org/
librosa	0.10.1	Einladen von Audiosignalen und Featureberechnungen auf ihnen
	ISC	https://librosa.org/
torchaudio	2.0.2	Handhabung von Audiodaten, optimiert für die Integration in PyTorch
	MIT	https://pytorch.org/audio/
torchmetrics	1.1.1	Softwarepaket für die präzise und effiziente Berechnung von Metriken in ML Projekten
	Apache-2.0	https://lightning.ai/torchmetrics
Pillow	9.4.0	Bildverarbeitung
	HPND	https://python-pillow.org/
resampy	0.4.2	Audio resampling
	ISC	https://github.com/bmcfee/resampy
jupyter	1.0.0	Interaktive Python Code Umgebung
	BSD-3-Clause	https://jupyter.org/
ipykernel	6.19.2	Kernel für die Jupyter Umgebung
	BSD-3-Clause	https://ipython.org/
scipy	1.11.2	Bibliothek für Wissenschaftliche Berechnungen
	BSD-3-Clause	https://scipy.org/
seaborn	1.13.0	Visuell ansprechende Darstellungen von Daten
	BSD-3-Clause	https://seaborn.pydata.org/
tabulate	0.9.0	Darstellung von tabellarischen Daten in der Konsolenausgabe
	MIT	https://github.com/astanin/python-tabulate/
tqdm	4.66.1	Fortschrittsbalken und Informationen in Schleifen
	MIT	https://tqdm.github.io/
YAMNet		Feature Extraktor YAMNet von TensorHub
	Apache-2	https://github.com/tensorflow/models/tree/master/research/audioset/yamnet
VGGish		Feature Extraktor VGGish von TensorHub
	Apache-2	https://github.com/tensorflow/models/tree/master/research/audioset/vggish

3.2 PhysioNet Datensatz

In diesem Abschnitt 3.2 wird der verwendete Datensatz für das Training etwas genauer erklärt. Dieser Abschnitt 3.2 enthält ebenfalls größtenteils Inhalte aus der zuvor eingereichten Hausarbeit (Sondermann , 2023). Verbesserungen der Satzstruktur und Fehler sind korrigiert.

3.2.1 Herkunft der Daten

Die Datenbankerstellung ist ein entscheidender Schritt für die Herzton-Klassifikation, da sie die Grundlage für die Analyse von Herztönen und die Entwicklung von Klassifikationsalgorithmen bildet. Von Liu et al. (2016) werden die Daten, die im Rahmen der PhysioNet/CinC-Challenge-2016 erstellt wurden, weiter erklärt. Insgesamt acht bestehende Datensätze wurden hierfür ausgewählt. Diese Daten wurden den Teilnehmern der Challenge zur Verfügung gestellt, um die Wissenschaftliche Gemeinschaft anzuregen, neue und bessere Klassifikatoren zu entwickeln. Im Anschluss an die Challenge wurden die eingereichten Methoden mit einem speziellen Score der Organisatoren bewertet und veröffentlicht. Von Clifford et al. (2016) werden diese Ergebnisse und die Zusammenstellung der Daten am Ende der Challenge, ausführlich beschrieben. Die verwendeten Audiodateien wurden auf eine Abtastrate von 2000 Hz resampled und als .WAV Dateien zur Verfügung gestellt, um eine einheitliche Grundlage für die Analyse zu gewährleisten.

3.2.2 Umfang des Datensatzes

Insgesamt sind 4430 Aufnahmen für die Challenge erstellt worden. Die unterschiedliche Anzahl als aus den Summen der Zuvor angegebenen Zahlen der einzelnen Datensätze, beruht darauf, dass einige Aufnahmen aus dem DLUTHSDB Datensatz (in training-e vertreten) eine zu große Länge aufwiesen und in einzelne Sub-Teile getrennt wurden (Liu et al. , 2016). Die Publikation von Clifford et al. (2016), nach Ende der Challenge, erklärt die erstellten Datensätze und deren Aufteilung genauer.

Die Einträge der Datensätze wurden in einem 70:30 Training-Testsplit aufgeteilt, wobei vier Datensätze sowohl für das Training als auch für das Testen verwendet wurden und die anderen vier ausschließlich entweder für das Training oder für das Testen bereitgestellt wurden. Diese Datensätze sind in Subsets training-a/b/c/d/e/f sowie test-b/c/d/e/g/i unterteilt worden. Der gesamte Testdatensatz ist nie der Öffentlichkeit zur Verfügung gestellt worden und wurde intern bei der Challenge zur Bestimmung der Genauigkeitsmetriken benutzt. Hierfür wurde ein konkreter Score erstellt, der sich aus Sensitivität und Spezifität zu normalen Audiodateien und zusätzlich über Dateien mit schlechter Audioqualität, zusammensetzt. Alle Trainingsdaten und Annotationen, zusammen mit den Algorithmen der Teilnehmer und deren Publikationen, werden unter https://physionet.org/content/challenge-2016/1.0.0 (Stand 01.09.2023) zur Verfügung gestellt. Zusätzlich ist ein Validierungsdatensatz mitgeliefert, welcher aus 301 Kopien bestehender Daten ist und somit keine neuen Informationen bietet (Clifford et al. , 2016; Liu et al. , 2016). Er wird daher hier auch nicht näher betrachtet. Allerdings stimmt die Anzahl der gelieferten Trainings .WAV-Dateien nicht mit den in der Publikation angegebenen Daten überein. Es sollten eigentlich 3153 Dateien vorhanden sein, aber tatsächlich liegen 3240 Dateien vor (Liu et al. , 2016). Es fällt auf, dass der Datensatz a-f korrekt angelegt ist, jedoch enthält der Datensatz-e 2141 Dateien anstelle der von Liu et al. angekündigten 2054 Dateien. Neben den eigentlichen Dateien liegen weiter Metadaten vor. Eine Liste enthält `Challenge Record Name`, `Original Record name`, den Namen des Datensatzes, Diagnose, Klasse, Anzahl der Herzschläge und Anzahl der Herzschläge die per Hand korrigiert wurden. Es gibt auch Spalten für das Alter, das Geschlecht und verschiedene Störgeräusche, aber diese Informationen sind nur jeweils für einen kleinen Teil der Daten verfügbar und werden daher hier nicht weiter betrachtet. Die Liste mit den mitgelieferten Annotationen und Metadaten enthält die erwarteten 3153 Einträge. Es zeigt sich, dass es eine zusätzliche Liste mit Klassenzugehörigkeit und einer Audio Qualitätseinstufung gibt, die Daten für 3240 Audiodateien enthält. Die Audioqualität wurde im Verlauf der Challenge, zusammen mit einer neuen Scoring Methode. nachgereicht. Es gibt 87 Einträge, die in der ursprünglichen Metadatenliste nicht benannt sind. Diese

Einträge stammen alle aus dem "training-e"Subset. Es fällt auf, dass der tatsächliche Dateiname in der Spalte `Challenge Record Name` stehen sollte, während in der Spalte `Original Record Name` oft andere Bezeichnungen zu finden sind, die anscheinend nicht auf eine vorliegende Datei schließen lassen. Mit Ausnahme der zuvor genannten zusätzlichen Dateien in der Zusatzliste. Zum Beispiel existiert die Datei "e00001.wav"tatsächlich. In der Annotationsliste erscheint dieser Name jedoch nicht unter der üblichen Spalte `Challenge Record Name`, sondern in der Spalte `Original Record Name`. Allerdings ist für diese Datei der Challenge Record Name "e00137"angegeben, obwohl keine Datei namens "e00137.wav"vorhanden ist. Mutmaßlich lässt sich dieser Umstand durch die erwähnte Trennung in einzelne Sub-Dateien innerhalb von Subset-E erklären, was allerdings nicht transparent beschrieben wird. Eine Analyse aller vorhandenen Dateien und der beiden Annotationslisten zeigt, dass für jede Datei, inklusive der überschüssigen Dateien, ein Eintrag in der zusätzlichen Qualitätsliste mit einer Klassifizierung vorliegt. Wenn diese Datei ebenfalls in der originalen Standardliste vorhanden ist, was bis auf 16 Dateien der Fall ist, stimmen die Klassenangeben überein. Somit ist die nachgereichte Qualitätsliste bis auf diese Ausnahme, allein für eine Klassifikation hinreichend. Schlussendlich liegen 3240 Audiodateien vor.

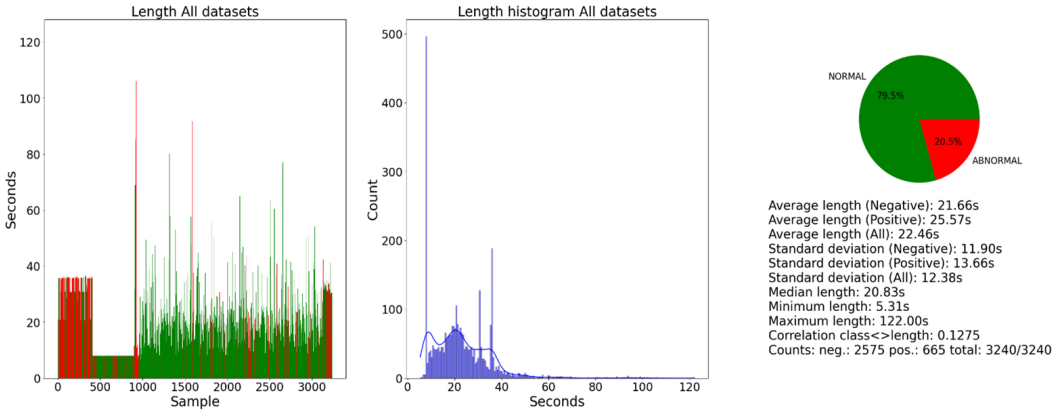

Abbildung 3.1: Visualisierung der Audiolänge und Klassenverteilung aller vorliegender Daten (Sondermann , 2023)

3.3 Features

Features beschreiben nach Alzubaidi et al. (2021) bestimmte Merkmale von Daten. Es wird erklärt, dass die Feature Wahl und Berechnung starken Einfluss auf Klassifikationsergebnisse eines Modells hat und daher sorgfältig gewählt werden sollten. Die Autoren beschreiben mehrere Unterschiedliche Arten von Features. In diesem Projekt sind Audiofeatures von besonderem Interesse und werden daher nun gesondert beschrieben.

3.3.1 Spektrogramme für das CNN

Von Thalmayer et al. (2020) werden Spektrogramme genauer verwendet und hier beschrieben. Ein Spektrogramm werde typischerweise durch die STFT eines Signals erzeugt, welche das Signal in seine Frequenzkomponenten zerlegt. Die kontinuierliche STFT eines Signals $x(t)$ sei definiert als:

$$X_c(\tau, f) = \int_{-\infty}^{+\infty} x(t) \cdot w(t-\tau) \cdot e^{-j2\pi f t}\, dt \tag{3.1}$$

Dabei ist $x(t)$ das kontinuierliche Zeit-Signal, $w(t-\tau)$ das Zeitfenster an der Stelle τ und f die Frequenz in Hertz. Gemäß Thalmayer et al. (2020) eignet sich das Hamming-Fenster ideal als Fensterfunktion für PCG Signale und wird daher auch in dieser Anwendung verwendet. Zur Verwendung in Algorithmen wird die diskrete Berechnung verwendet. Von den Autoren wird erklärt wie aus der STFT Funktion eines diskreten Signals $x[n]$ das Spektrogramm berechnet wird:

$$S[m,n] = |X[m,n]|^2 \tag{3.2}$$

Wobei $S[m,n]$ die Energie oder Leistung des Signals bei der diskreten Zeit m und der diskreten Frequenz n darstellt. Die diskrete STFT $X[m,n]$ ist nach Khan et al. (2021) definiert als:

$$X[m,n] = \sum_{k=0}^{N-1} x[k] \cdot w[k-m] \cdot e^{-\frac{j2\pi nk}{N}} \tag{3.3}$$

Wobei N die Anzahl der Punkte im Signal ist, $x[k]$ das diskrete Signal und $w[k-m]$ das diskrete Fenster zur Zeit m, das entlang des Signals verschoben wird.

Spektrogramme sind für die Verarbeitung durch ein CNN nützlich, da sie eine klare, visuelle Strukturierung der wichtigsten Merkmale eines Audiosignals bieten und direkt als 2D-Feature verarbeitet werden können Khan et al. (2021). Sie bieten einen guten Kompromiss aus temporalen und Frequenz-Zusammenhängen, welche sonst nicht gegeben wären (Maity et al. , 2023). Die Intensität der Werte in einem Spektrogramm entspricht dem Quadrat der Amplitude der Frequenz-Zeit-Komponenten Transformation des Signals und beschreibt somit eine Dichte-Funktion der Soundenergie (Sejdic et al. , 2008). Diese Eigenschaften das Signal zu repräsentieren werden hier bei der Klassifizierung von Herztönen zu Nutze gemacht und als Input für das CNN verwendet.

3.3.2 Mel-Spektrogramme

Von Maity et al. (2023) werden diese Sonderform von Spektrogrammen verwendet und in diesem Abschnitt 3.3.2 wiedergegeben. Die linearen Frequenzskalen eines gewöhnlichen Spektrogramms werden entsprechend der Mel-Skala in eine logarithmische Skala transformiert. Dies passt besser zur menschlichen Wahrnehmung von Geräuschen, die ebenfalls logarithmisch ist, und daher werde angenommen, dass sich diese Skalierung besser für die Geräuschklassifikation eignet. Ein Mel-Spektrogramm wird durch Anwendung der STFT auf ein Audiosignal und anschließender Abbildung der Frequenzen auf die Mel-Skala erhalten. Diese Skalierung mit $mel(f)$ der Frequenz f ist definiert als:

$$mel(f) = 2595 \log_{10}\left(1 + \frac{f}{700}\right) \tag{3.4}$$

Die Werte dieser Mel-Skala werden mit einer Filterbank verwendet. Diese Filterbank spiele laut den Autoren eine entscheidende Rolle bei der Umwandlung des Frequenzspektrums in die Mel-Skala. Jeder Filter in dieser Bank ist eine Dreiecksfunktion, die sich über einen bestimmten Bereich der Mel-Skala erstreckt.

Die Filterbank-Funktion $H_k(f)$ für den k-ten Filter ist definiert als:

$$H_k(f) = \begin{cases} 0 & f < f(k-1) \\ \dfrac{f - f(k-1)}{f(k) - f(k-1)} & f(k-1) \leq f < f(k) \\ \dfrac{f(k+1) - f}{f(k+1) - f(k)} & f(k) \leq f \leq f(k+1) \\ 0 & f > f(k+1) \end{cases} \tag{3.5}$$

Wobei $f(k)$ die Frequenz an der k-ten Mel-Frequenzband-Grenze ist. Die Anzahl der Filter bestimmt, wie feingranular das Mel-Spektrogramm ist; mehr Filter führen zu einer detaillierteren Darstellung des Frequenzspektrums. Das Mel-Spektrogramm wird gebildet, indem das Spektrogramm $S[m,n]$ mit der Mel-Filterbank verarbeitet wird:

$$S_{mel}[m,k] = \sum_{n=0}^{N-1} S[m,n] \cdot H_k(mel(n)) \tag{3.6}$$

Hierbei entspricht m der Zeit und $S[m,n]$ dem generierten STFT-Spektrogramm aus Gleichung 3.2. Die Anwendung der Filterbank transformiert die linearen Frequenzkomponenten in das Mel-Frequenzband, das die menschliche Wahrnehmung von Klang besser widerspiegelt (Maity et al. , 2023). Zuletzt werden die berechneten Mel-Spektrogramme zu der logarithmischen Dezibel-Skala skaliert:

$$S_{dB} = 10 \cdot \log_{10}(S_{mel}) \tag{3.7}$$

Diese Gleichung 3.7 transformiert die Werte des Mel-Spektrogramms zu log-Mel-Spektrogrammen und verbessert die Repräsentation der Daten, was wiederum ein effizienteres Training mit CNNs ermöglicht (Choi et al. , 2018).

3.4 Verwendung des Datensatzes

Filterung

Audiosignale sind häufig durch Störgeräusche wie Atemgeräusche, Hintergrundlärm, Bewegungen oder elektronische Interferenzen kontaminiert (Chowdhury et al. , 2019) (Thalmayer et al. , 2020). So können die normalen Geräusche der Atmung unter Umständen fälschlicherweise als Murmur eingestuft werden (Maity et al. , 2023). Bandpassfilter werden eingesetzt, um diese unerwünschten Frequenzkomponenten zu eliminieren (Singh et al. , 2020). Im Rahmen dieses Projekts kommt ein Butterworth-Bandpassfilter vierter Ordnung zum Einsatz. Dieser konkrete Filter entfernt Frequenzen unterhalb von 25 Hz und oberhalb von 400 Hz. Diese Werte entsprechen dem Bereich wie auch von Singh et al. (2020) angegeben. Zudem sind diese Einstellungen in Experimenten empirisch bestätigt. Andere Publikationen verwenden diesen Filter ebenfalls für Herzton Klassifizierung und Segmentierung, teils mit unterschiedlichen Konfigurationen (Deng et al. , 2020; Maity et al. , 2023; Thalmayer et al. , 2020).

Augmentierung

Durch Xie et al. (2021) wird eine breite Übersicht über Augmentierung gegeben und in diesem Absatz vorgestellt. Data Augmentation wird oft bei Machine Learning Projekten eingesetzt um die Diversität innerhalb der Trainingsdaten zu erhöhen. Ziel ist die Erhöhung der Robustheit des trainierten Modells, die alternativ durch eine Vergrößerung des Datensatzes erreicht werden würde. Kleine zufällige Veränderungen werden in die Trainingsdaten eingefügt. In der Bildklassifikation sind typische Methoden der Augmentation beispielsweise Rotationen, Skalierungen und Schattierungen.

Von Zhou et al. (2022) wird Augmentierung von Spektrogramm Daten gesondert untersucht. Viele mögliche Methoden zur Augmentierung die üblicherweise bei Bildern verwenden werden seien so nicht anwendbar. Ein Spektrogramm, obwohl es formal ein 2D-Feature ist, beinhaltet andere semantische Daten, welche nicht

wie ein Bild betrachtet werden können. Zudem wird erklärt, dass gängige Methoden wie Noise-Injection oder Farbfilter ebenfalls nicht anwendbar seien. Bei der Klassifikation von gesunden und abnormalen Audiosignalen, bei der die Grenzen bereits unklar sind, wäre die Zugabe von künstlichem Rauschen kontraproduktiv.

In diesem Projekt ist der Umfang an möglicher Augmentierung daher geringer. Wenn ein Signal Ausschnitt zur Augmentierung ausgewählt ist, dann wird die Wiedergabegeschwindigkeit auf einen zufälligen Wert zwischen 0.85 bis 1.15 geändert und das Signal auf die Ursprungslänge skaliert. Zudem wird der Pitch um maximal 2 Halbtöne nach oben oder unten verschoben. Hierbei ist laut den Autoren der genannten Studie wichtig nicht die physiologischen Grenzen eines Herztons zu verlassen, um keine negativen Effekte für die Klassifizierung zu induzieren. Weitergehend wird das errechnete Spektrogramm für einen zufälligen Zeitraum, maximal 25%, geschwärzt.

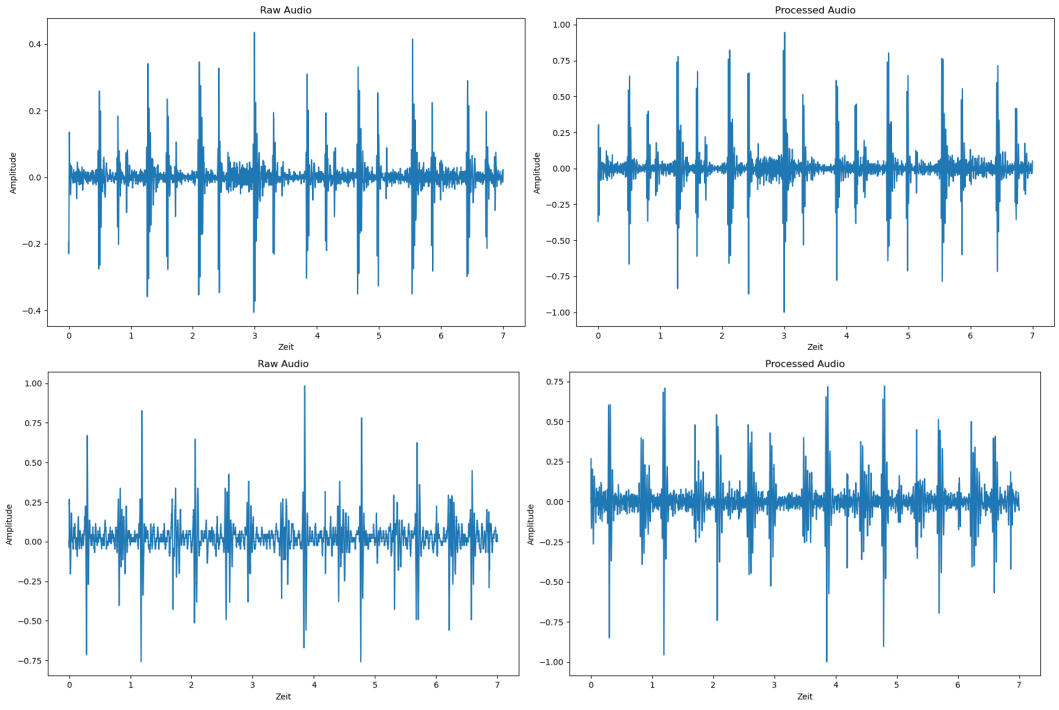

Abbildung 3.2: Audiosignal Darstellung von Chunks im Roh Zustand und nach der Augmentierung Oben: Negative Klasse, Unten: Positive Klasse

Verwendete Beispiele

Aufgrund der zuvor gestellten Annahmen, dass log-Mel-Spektrogramme besser für maschinelles Lernen geeignet sind und sie mehr den Frequenzwahrnehmungen des menschlichen Gehörs ähneln, werden diese in diesem Projekt ebenfalls verwendet.

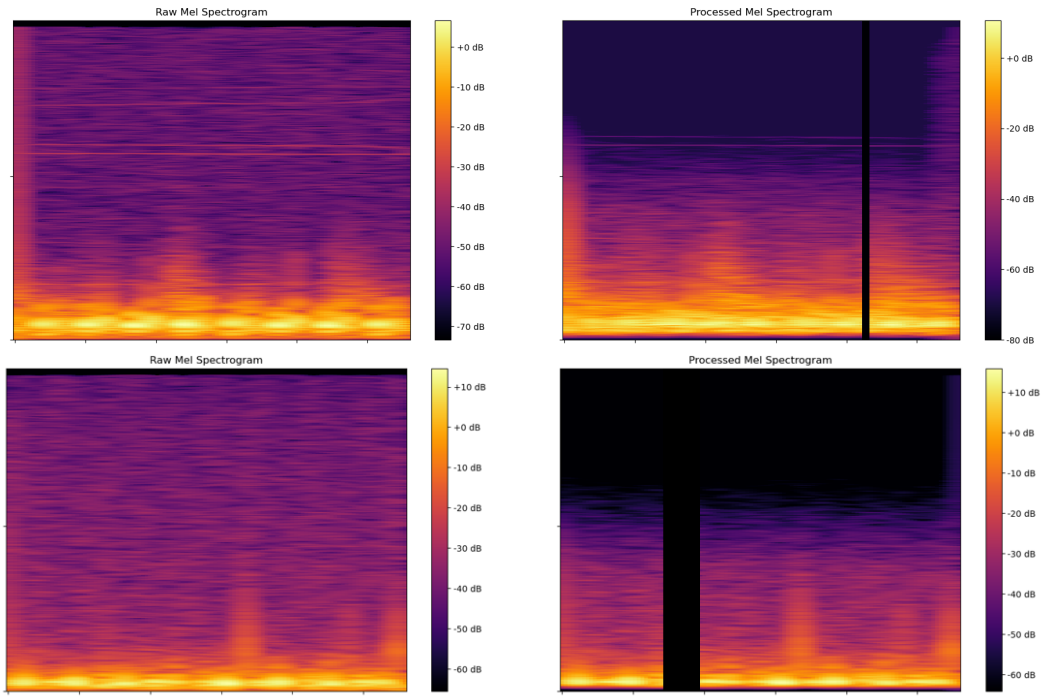

Abbildung 3.3: Generiertes Mel-Spektrogramm von Audiosignalen jeweils vor und nach der Vorverarbeitung (Audio Filterung und Normalisierung, Audio-Augmentation und Spektrogramm-Augmentation) Oben: negatives Beispiel, Unten: positives Beispiel

Diese Abbildung 3.3 zeigt vier Spektrogramme, dargestellt als Bild mit farblich eingefärbter Amplitude. Sie stammen jeweils von zwei verschiedenen Beispielen, links vor und rechts nach Filterung sowie Augmentation. Sie zeigen Beispiele welche in dem CNN Training verwendet werden.

3.5 Trainingszyklus

Das Training beginnt mit einem Python dictionary Objekt (dict). Dort werden alle Parameter die für einen Durchlauf relevant sein können hinterlegt. Zudem kann dieses dict durch einen so genannten "Sweep"durch das Tool *Weights and Biases* (WANDB) angepasst werden (Siehe Abschnitt 3.7.2). Sinn dieses dict ist es, immer in jedem Code Abschnitt verfügbar zu sein und universell anwendbar zu sein. Dies fördert die Reproduzierbarkeit und erleichtert es diverse Experimente durchzuführen.

Tabelle 3.2: Auszug an Elementen des Config-dicts

Spaltenname	Beschreibung
epochs	Anzahl der Epochen eines Durchlaufes
k-fold	Wie viele k-fold splits erstellt werden
augmentation_rate	Prozentsatz der zu Augmentierenden Inputsamples
l1_weight	Faktor für die L1 Regularisation
head_pct	Prozentsatz der Daten die für das Training verwendet werden
activation	Welche Aktivierungsfunktion (relu, silu, ..)
train_split	Training-Validierungs Split, wenn k-fold=0
criterion	Welche Verlustfunktion verwendet werden soll
optimizer	Wahl der Optimizers (adam, sgd)
sheduler	Welcher Sheduler (ReduceLROnPlateu, CosineAnnealing, StepLR)
learnrate	Initiale lernrate
drop0	Dropout an Position 1
drop1	Dropout an Position 2

Weitergehend sind Parameter für die Spektrogramm Featureberechnung hinterlegt:

Tabelle 3.3: Auszug an Audio Einstellungen des Config-dicts

Spaltenname	Verwendeter Wert	Beschreibung
samplerate	2000	Ziel Samplerate der Audiodaten
n_fft	1024	Bestimmt die Anzahl der Punkte für die FFT, beeinflusst die Frequenzauflösung
hop_length	128	Schrittgröße zwischen FFT-Fenstern, beeinflusst die Zeitauflösung.
n_mels	512	Größe der Mel-Filterbank
top_db	80	Referenzwert für die Skalierung zu Dezibel
butterpass_low	25	Hochpassfilter Frequenz
butterpass_high	400	Tiefpassfilter Frequenz

Diese Parameter haben einen signifikanten Einfluss in die Trainingsschleife und Featureberechnung. Die Sorgfältige Auswahl und Einstellung ist somit notwendig. In dieser Tabelle 3.3 gelisteten Parameter sind die in dieser Anwendung für am besten befundenen Parameter. Die Auswahl erfolgt empirisch durch Experimente oder sind durch diverse WANDB Sweeps ermittelt worden. Ein vollständige Auflistung der Parameter befindet sich in Quellcode B.0.1

3.5.1 Modularisierung

Durch Modularisierung des Codes können ähnliche Aufgaben effizient durchgeführt werden, ohne Redundanzen erstellen zu müssen. Dies ist insbesondere in dieser Anwendung hilfreich, da mehrere Verfahren verglichen werden, diese aber auf dem selben Datensatz beruhen und gemeinsame Vorverarbeitung erfahren.

Eine Erweiterung mit anderen Verfahren oder Modellen wäre von geringem Aufwand, da die Strukturen bereits gegeben sind und das config-dict überall durchgereicht wird. Die CNN, YAMNet oder VGGish Modelle befinden sich in ihrem jeweils eigenem Python Modul und werden durch eine einheitliche Funktion aufgerufen. Da jedes Modul eine abgegrenzte Funktionseinheit darstellt, können Fehler leichter lokalisiert und behoben werden, ohne andere Teile des Systems zu beeinträchtigen und dort ebenfalls Änderungen notwendig zu machen.

3.5.2 Datenverwaltung und Einladen

Die Metadaten des gegebenen Datensatzes werden in eine gemeinsame Zwischenliste überführt. Dort sind für jede Datei die notwendigen Metadaten sowie Pfadangaben zur vorliegenden Audiofile eingetragen. Dieses Überführen ermöglicht ein vereinfachtes Einlesen und verbessert die Übersicht, entgegensetzt der roh-Metadaten welche für jeden Unterdatensatz einzeln angelegt sind. Zudem sind die nachgereichten Qualitätsangaben (Clifford et al. , 2016) sowie Labeländerungen bereits übernommen und müssen nicht im Trainingscode nachgereicht werden. Weitere Datensätze oder Subsets für andere Experimente können auf diese Weise

dem Trainingscode zugänglich gemacht werden, ohne dass dieser verändert werden müsste. Es genügt die neuen Daten in gleicher Struktur einmalig vorzubereiten.

Tabelle 3.4: Spaltennamen und Beschreibung der erstellten Zwischendatei

Spaltenname	**Beschreibung**
id	Eindeutige ID des Eintrages
path	Relativert Pfas zur Audiodatei innerhalb des Projektes
name	Bezeichung des Eintrages, hier der Dateiname ohne Endung
dataset	Unterdatensatz Zugehörigkeit
diagnosis	Zusätzliche Label aus dem Roh-Datensatz
quality	Qualitätseinstufung anhand Einstufung der Challenge Autoren
sr	Native Samplerate der Datei
channels	Anzahl der Channels der Datei
length	Länge in einzelnen Samples
bits	Bitrate der Datei
label_1	Die Einstufung Normal/Abnormal

Im weiteren Schritt werden die Audiodateien alle in einer Schleife geladen und deren Länge ermittelt. In dem config-dict ist das `seconds` Feld relevant. Da manche Audiofiles teils über 100 Sekunden lang sind, werden diese in einzelne Teile, hier **Chunks** genannt, unterteilt. Der `seconds` Parameter stellt die gewünschte Länge ein. Empirische Versuchen haben gezeigt dass ein Wert von 7 (Sekunden) am Besten ist. Diese Länge von 7 Sekunden liegt gerinfügig unter den meisten am häufigsten vorliegenden Audiolängen im Datensatz. (Siehe Abbildung 3.1). Für jede Datei aus der Zwischendatei wird die Länge ermittelt und errechnet, wie oft ein Chunk der Länge `seconds` erstellt werden kann. Dann wird anschließend geprüft, ob der letzte Chunk der Liste an Chunks einer Datei kürzer ist als `seconds`. Dies ist realistischerweise praktisch immer der Fall. Es wird geprüft, ob die Fehlende Länge zu einem nächsten kompletten `seconds` Chunk, weniger als ein konfigurierbarer Anteil von `seconds` ist. In diesem Experiment wird ein Chunk verworfen, wenn die Länge kürzer als 65% eines 7 Sekunden Chunks beträgt. Ist die Länge über dem Schwellwert werden zu Beginn und Ende Daten angefügt. Als Wert hierfür wird das Minimum der vorliegenden Amplitude des Chunks

verwendet. Die Einträge der Zwischendatei werden für alle erkannten Chunks dupliziert und `range_start` und `range_end` als zusätzliche Spalten angefügt. Dort werden die Indexposition der jeweiligen Chunk-Fenster des Audiosignals gespeichert. Alle vorherigen Metadaten wie Label und Pfad bleiben erhalten. Bei einem 7 Sekunden Split mit der hier vorgestellten Methode werden aus den 3211 Dateien insgesamt 9850 Chunks erstellt.

Es werden zu kurze überschüssige Chunks verworfen, welche meist am Ende einer längeren Audiodatei anfallen, die knapp nicht in exakte 7 Sekunden Chunks unterteilt werden kann. Die Länge der verworfenen Chunks beträgt unter der Betrachtung des vollständigen Datensatzes ungefähr 7.43%. Dieser Wert ist je nach k-fold Trainingsliste leicht abweichend (Siehe Abschnitt 3.5.3). Die Klassenverteilung ändert sich von 20.5% positiven Dateien zu 23.6% positiv gelabelten Chunks, wenn diese wiederum betrachtet werden. Zu beachten ist, dass diese Angaben für den gesamten Datensatz gelten. Durch die Trennung der Daten für einen Training-Validierungssplit und unter verschiedenen Folds ändern sich die Verteilungen zum Teil um bis zu 2 Prozentpunkte. Eine etwaige Ausgleichung der Klassenimbalance, wie durch eine FocalLoss Lossfunktion, wird daher für jeden k-fold Split separat erstellt.

3.5.3 Trainingsschleife

k-fold Validierung

Durch Combrisson und Jerbi (2015) werden die hier verwendeten und in Abschnitt 3.5.3 vorgestellten Methoden zu k-fold und Stratifizierung beschrieben. Die k-fold-Validierung ist eine Technik zur Beurteilung der Leistungsfähigkeit eines Modells, die im Kontext der Kreuzvalidierung (Cross-Validation) verwendet wird. Kreuzvalidierung ist ein Verfahren zur Evaluierung eines Klassifikators und hat den Vorteil, dass sie den gesamten Datensatz sowohl für das Training als auch für die Validierung nutzen kann. Alle Daten werden in k Folds, also Unterteilungen, getrennt. Alle bis auf einen werden für das Training verwendet. Das Modell, basierend

auf diesen Daten, wird mit dem übrig gebliebenen Teil validiert. Dies wird k male durchgeführt, wobei immer ein anderer Fold zum Validieren verwendet wird. Varianzen werden dadurch unterdrückt und jedes Inputsample wird garantiert einmal für das Training verwendet. Im Anschluss werden die Metriken für jeden Durchlauf gemittelt. Für dieses Verfahren gibt es diverse Varianten. In dieser Anwendung wird Stratified-Group-k-fold von dem Modul sklearn verwendet. Die Verwendung wird unter https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedGroupKFold.html (Abgerufen 14.12.2023) genauer beschrieben. Die Stratifizierung bestrebt zwischen den verschiedenen Folds eine möglichst ausgeglichene Klassenverteilung zu erhalten und dass die Verteilung etwa dem des gesamten Datensatzes entspricht. Dies ist bei dem PhysioNet2016 Datensatz insbesondere relevant, da er mit den ungefähr 24% positiven Fällen stark unausgeglichen ist. Das grouping ist notwendig um Informationleakage zwischen den Folds zu verhindern. Der von skikit-learn geforderte Parameter für das Gruppieren ist in dieser Anwendung der Dateiname des jeweiligen Chunks. Effektiv wird damit also verhindert, dass Chunks einer selben Datei in unterschiedlichen Splits vorhanden sind. Erscheint der selbe Patient in beispielsweise sowohl dem Trainings als auch dem Validierungs Fold, so stellt dies eine Möglichkeit für Information-Leakage dar und die Sets sind nicht mehr komplett unabhängig. Der Datensatz beinhaltet bekanntermaßen mehrere Einträge des selben Patienten, ohne dass dies in den Metadaten mitgeteilt wird Liu et al. (2016). Dies wird durch das Stratified-Group-k-fold nicht verhindert, aber eine Verschärfung dieser Situation mit Chunks einer selben Aufnahme wird unterbunden.

Epochenzyklus

Der Epochenzyklus ist die konkrete Umsetzung im Code, wie die Daten aus dem Datensatz geladen und trainiert werden. Als Framework hierfür wird PyTorch verwendet und nach der offiziellen Dokumentation (https://pytorch.org/docs/stable/; Stand 20.12.2023) implementiert. Während jeder Epoche werden so genannte Minibatches erzeugt, welche eine Sammlung an Features sind und für das Training oder die Validierung verwendet werden. Sie werden auch Beispiele genannt, welche dem

Modell präsentiert werden. Die Featureextraktion geschieht in den für die jeweilige Methode konzipierten Dataloader. Bei allen Methoden wird zunächst das Audiosignal anhand der zugewiesenen Chunkgrößen eingeladen und ggf. resampled. Im nächsten Schritt wird die Audiosignal-Filterung durchgeführt und das Signal Peak-Normalisiert. Weitergehend wird unterscheiden ob der Dataloader Beispiele für das Training oder die Validierung erzeugt. Nur beim ersteren wird das Audiosignal Augmentiert. Anschließend werden die für die jeweiligen Methode gewählten Features extrahiert. Im CNN Fall werden die Mel-Spektrogramme anhand der gefilterten Audiosignal Ausschnitte erstellt. Im Trainingsfall wird zudem das Spektrogramm augmentiert. Anhand der aus Tabelle 3.3 verwendeten Parameter, entstehen pro Audiochunk 512×110 große Spektrogamme. Die Embedding Modelle besitzen ihre eigene feste Spektrogramm Extraktionsmethoden, welche genau für das Modell konzipiert wurden und im Modellquellcode von deren Entwicklern mitgeliefert werden.

Optimierer und Scheduler

Das Training des CNN wird durch einen Optimieren und Scheduler unterstützt. Ein Scheduler sorgt für eine dynamische Änderung der Lernrate. Hierbei sind diverse Implementierungen in der Anwendung verfügbar:

ReduceLROnPlateu sobald der Loss während des Validierung über *patience=4* Epochen stagniert, wird die Lernrate um *gamma=0.3* verrechnet und somit verringert.

CosineAnnealing Der Verlauf der Lernrate wird jede Trainings-Epoche angepasst, sodass über 10 Epochen ein Verlauf der Cosinus Kurve abgebildet wird. Die Lernrate ist also zyklisch sinkend und steigend.

StepLR Hierbei wird alle *10* Epochen die Lernrate mit *gamma=0.25* verrechnet und somit verringert. Es bildet sich ein Treppen-stufiger Verlauf.

Als Optimierer ist, nach durchgeführten Versuchen mit adam, der SGD zum Einsatz bestimmt worden. Im folgenden Quellcode 3.5.1 werd ein Ausschnitt des erarbeiteten Codes abgebildet. Es handelt sich hierbei um einen Auszug, aus welchem nicht

relevante Teile, wie Progressbar und das Tracken der Metriken, ausgelassen sind. Die Reihenfolge der Methoden entspricht der logischen Ausführungsreihenfolge innerhalb der Trainingsschleife. Er dient als Referenz für die Umsetzung der Loops, dem k-fold splitting und dem Aufrufen von Optimizer und Sheduler sowie der Trennung in Training und Validierung.

```
config: dict = load_run_config() # current config of the run
datalist: pd.DataFrame = load_datalist() # all files and labels
# start of the training loop
def train_loop():
  kfold = sklearn.model_selection.StratifiedGroupKFold(n_splits
   =10, shuffle=True, random_state=SEED)
  label_list = datalist.get(config['label_name'])
  name_list = datalist.get('name')
  # "name" is the unique filename - used for grouping
  # because of chunk splitting, the same file could otherweise be
   in multiple folds
  for train_index, val_index in kfold.split(datalist, label_list,
   name_list):
    perform_fold(train_index, val_index)

# train and validate a specific fold
def perform_fold(train_index=None, val_index=None):
  prepare_cnn() # prepare optimizer, sheduler, criterion
  trainloader,validloader=get_dataloader(train_index,val_index)
  for epoch in range(run_config['epochs']):
    train_epoch(trainloader)
    validation_epoch(validloader)
    # save the model, early stopping check, saving metrics

def train_epoch(dataloader):
  model.train()
  dl_it = iter(dataloader)
  for _, (data_batch, labels) in enumerate(dl_it):
    process_batch(data_batch, labels, validation=False)

def validation_epoch(dataloader):
  model.eval()
  with torch.no_grad():
    dl_it = iter(dataloader)
    for _, (data, labels) in enumerate(dl_it):
      process_batch(data, labels, validation=True)

# prepare a minibatch for training
def process_batch(data_batch, labels, validation):
  data_batch = data_batch.to(device)
  labels = labels.to(device)
  loss, probabilities = predict_step(data_batch, labels)
  sheduler_step(loss, validation)
  if not validation: # step optimizer in training mode
    optimizer_step(loss)
  return loss, probabilities
```

```
44
45 # prediction of one minibatch
46 def predict_step(inputs, labels):
47   with autocast(): # mixed precision
48     outputs = model(inputs)
49     loss = criterion(outputs, labels)
50   l1_regulation = config['l1_weight']
51   if l1_regulation > 0:
52     regularization_loss = 0
53     for param in model.parameters():
54       regularization_loss += torch.norm(param, p=1)
55     loss += l1_regulation * regularization_loss
56   probabilities = torch.softmax(outputs, dim=1)
57   return loss, probabilities
58
59 def optimizer_step(loss):
60   optimizer.zero_grad()
61   scaler.scale(loss).backward()
62   scaler.unscale_(optimizer)
63   nn.utils.clip_grad_norm_(model.parameters(), 1.0)
64   scaler.step(optimizer)
65   scaler.update()
66
67 def scheduler_step(loss, validation: bool):
68   # check selected sheduler
69   # step StepLR and CosineAnnealing if not validation
70   # step ReduceLROnPlateau if validation
71   scheduler.step(loss)
```

Quellcode 3.5.1: Auszug des verwendeten Codes für das CNN Training

Die Trainingschleifen für die YAMNet und VGGish basierenden Klassifikatoren verlaufen beim k-fold splitting analog wie hier in Quellcode 3.5.1. Allerdings fallen Schritte wie Optimizer und Sheduler, sowie generell dem Training des CNNs weg. Stattdessen werden die Extraktoren, YAMNet oder VGGish, geladen und durch ihnen die Embeddings erstellt. Diese Schritte werden im folgenden Quellcode 3.5.2 in Auszügen dargestellt:

```
config: dict = load_run_config() # current config of the run
datalist: pd.DataFrame = load_datalist() # list of files and
    labels

def load_extractor(self):
  if self.mode == cfg.modes['yamnet-nn']:
    self.extractor = yamnet_models.get_model(base_config)
  elif self.mode == cfg.modes['vggish-nn']:
    self.extractor = vggish_models.get_model(base_config)
  state = torch.load(extractor_path)
  extractor.load_state_dict(state['model_state_dict'])
  extractor.eval()
  return extractor

def train_loop(self):
  num_splits = config['kfold'] if config['kfold'] > 1 else 1
  if num_splits > 1:
    data_kfold_object = sklearn.model_selection.
    StratifiedGroupKFold(n_splits=num_splits, shuffle=True,
    random_state=cfg.SEED)
    current_fold_number = 0
    label_list = datalist.get(config['label_name'])
    name_list = datalist.get('name')  # unique identifier for each
    file and thus hopefully patient
    kfold_splits = data_kfold_object.split(datalist, label_list,
    name_list)
    for train_index, val_index in kfold_splits:
      perform_fold(train_index, val_index, current_fold_number)
  else:
    perform_fold()

def train_nearestneighbor(features, labels):
  model = Model_NearestNeighbor_classifier()
  model.set_extractor(extractor)        # Set the same extractor
  model.add_neighbor_data(features, labels) # Fill embedding data
  model.build_nn_classifier(n_neighbors=config['n_neighbors'],
    distance_metric=config['knn_distance'], \
      assume_positive_p=config['assume_positive_p'],
    embedding_mode=config['embedding_mode_valid'])
  return model

def perform_fold(train_index=None, val_index=None):
  features, labels = train_epoch(trainloader)
  nearestneighbor_model = train_nearestneighbor(features, labels)
  valid_metrics = validation_epoch(validloader,
    nearestneighbor_model)
  # save metrics, save model

def train_epoch(self, trainloader):
  features = []
  labels = []
  self.extractor.eval()
  with torch.no_grad():
    for batch_idx, (data, target) in enumerate(trainloader):
      embeddings = extractor(data)
      embeddings = Model_NearestNeighbor_classifier.
```

```
    combine_embeddings(embeddings, config['embedding_mode'])
49      batch_size, num_embeddings, embedding_size = embeddings.
    shape
50      embeddings = embeddings.reshape(batch_size*num_embeddings,
    embedding_size)
51      target = target.repeat(1, num_embeddings).reshape(batch_size
    *num_embeddings)
52
53      features.append(embeddings)
54      labels.append(target)
55      self.pbars.update(3)
56  features = np.concatenate(features, axis=0)
57  labels = np.concatenate(labels, axis=0)
58
59  return features, labels
60
61def validation_epoch(validationloader, model,):
62  model.eval()
63  for batch_idx, (data, target) in enumerate(validationloader):
64    with torch.no_grad():
65      preds, probas = model.forward(data)
66      metrics.update_step(probas, target, validation=True)
67  epoch_metrics: dict = calculate_metrics(metrics)
68  return epoch_metrics
```

Quellcode 3.5.2: Auszug des verwendeten Codes für das das Embedding Training

3.6 Modellarchitekturen

Alle Modelle verwenden die PyTorch Struktur und die Module daraus. Ebenfalls anhand der Dokumentation unter https://pytorch.org/docs/stable/ (Abgerufen 15.12.2023) werden die Konzepte und die Struktur der Modelle hier beschrieben.

3.6.1 CNN Aufbau

Modellstruktur

Das Modell verfügt über vier aufeinanderfolgende Convolutional Blöcke. Ein solcher Block hat immer den folgenden Aufbau in der hier beschriebenen Reihenfolge:

Convolutional-Layer

Die Faltungsoperation und Dimensionserhöhung geschieht in dieser Schicht. Variable Parameter für Stride und Padding ändern die Kantenverarbeitung und Größe der Kernel-Operationen (Alzubaidi et al. , 2021).

Batchnorm Layer

Batchnormalisierung wird durchgeführt. Ziel ist es die Konvergenz zu beschleunigen und einem Kovarianz Shift entgegenzuwirken, indem die Ausgaben der vorherigen Schicht normalisiert werden (Ioffe und Szegedy , 2015).

Aktivierungs Funktion

Die gewählte Aktivierungsfunktion (ReLU, SiLU, tanh) zwischen den verschalteten Neuronen um eine Nicht-Linearität herzustellen (Elfwing et al. , 2018)].

Max Pooing Layer

Die Größe der Feature Map wird verringert, was die Rechenkomplexität verringert und die Merkmalsextraktion von manchen Features beeinflussen kann (Alzubaidi et al. , 2021).

Die erarbeitete Struktur der Blöcke ist in Tabelle 3.5 dargestellt.

Tabelle 3.5: Definition der einzelnen Convolutional-Blöcke

Name	Bestandteil	Input Channels	Output Channels	Kernel size	Stride
Block 1	Conv2D-Layer 1	1	8	(3, 3)	(1, 1)
	BatchNorm2D 1	8	8	–	–
	Aktivierungsfunktion				
	Max-Pool2D 1	8	8	(2, 2)	(2, 2)
Block 2	Conv2D-Layer 2	8	16	(3, 3)	(1, 1)
	BatchNorm2D 2	16	16	–	–
	Aktivierungsfunktion				
	Max-Pool2D 2	16	16	(2, 2)	(2, 2)
Block 3	Conv2D-Layer 3	16	32	(3, 3)	(1, 1)
	BatchNorm2D 3	32	32	–	–
	Aktivierungsfunktion				
	Max-Pool2D3	32	32	(2, 2)	(2, 2)
Block 4	Conv2D-Layer 4	32	64	(3, 3)	(1, 1)
	BatchNorm2D 4	64	64	–	–
	Aktivierungsfunktion				
	Max-Pool2D 4	64	64	(2, 2)	(2, 2)

Die Anzahl der Ausgangskanäle der Convolutional-Layer verdoppelt sich mit jedem Block, beginnend bei 8 und endend bei 64. Die Padding- und Dilation-Parameter sind nicht geändert und verbleiben beim Standardwert Padding=(0, 0) und Dilation=(1, 1). Nach den vier verwendeten Blöcken folgt ein AdaptiveAveragePool-Layer. Dieser mittelt die Schichten auf eine Feste Layergröße von 8×8. Anschließend gehen die 2D-Layer in Fully-Connected Blöcke über.

- FC 1:
  Linear-Layer 4096 zu 1024, Batchnorm1D 1024, Aktivierungsfunktion, optionaler Dropout-Layer Position 1
- FC 2:
  Linear-Layer 1024 zu 128, Batchnorm1D 128, Aktivierungsfunktion, optionaler Dropout-Layer Position 2
- FC 3:
  Linear-Layer 128 zu den 2 Ausgabe Klassen, softmax folgt außerhalb des Modells zur weiteren Verarbeitung.

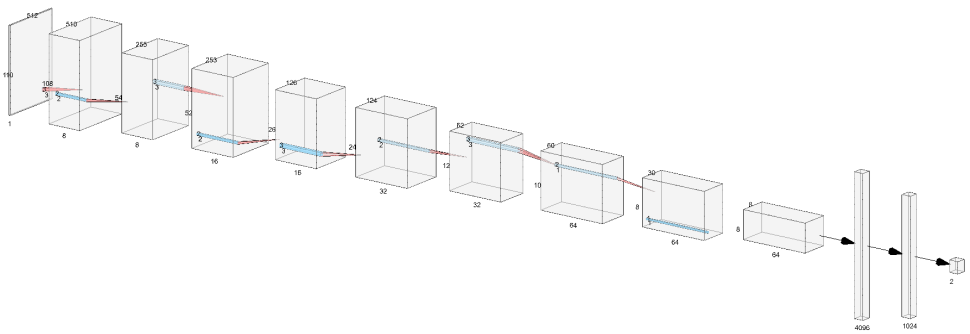

Abbildung 3.4: Graphische Darstellung des erstellen Modells

Diese Abbildung 3.4 zeigt den Modellaufbau des Netzwerkes in einer AlexNet-Style Grafik. Zu erkennen ist die sukzessive Verkleinerung der Inputgröße bei steigender Dimension. Dieser beschriebene Aufbau ist experimentell ermittelt worden und zeige die beste Performance.

3.6.2 Embedding Klassifikatoren Implementierung

Die beiden verwendeten Embedding Extraktoren erzeugen mehrere Embeddings für eingegebenes Audiosignal welche weiterverarbeitet werden müssen. Dies beruht auf der Eigenschaft immer nur für 0.96 Sekunden Blöcke ein Spektrogramm und eine Generierung durchzuführen. (Siehe Abschnitt 2.2.3) Es werden verschiedene Varianten, wie die Roh-Embeddings verarbeitet können, vorgestellt:

mean Die Elemente der Embeddings sind über ihre Dimension gemittelt. Aus einem Stack an 10×1024 Embeddings wird ein einzelnes 1x1024 Embedding erstellt.

plain Die Embeddings werden nicht weiter verarbeitet und jedes Fenster der Embeddingextraktion geht in den Datensatz

max Es wird aus jeder Dimension der maximale Wert verwendet.

sum Es wird die Summe entlang jeder Dimension des Embeddings gebildet. Aus 10×1024 entsteht 1×1024, wobei jedes Element die Summe aller Werte der selben Position ist.

tripplets_mean Die Embeddings werden ebenfalls gemittelt, allerdings nur jeweils 3 Stück hintereinander. Bleibt ein Rest übrig, wird dieser unter sich ebenfalls gemittelt. Ein Stack 10×1024 wird demnach zu 4×1024 umgeformt. Stack aus 6×1024 zu 2×1024. Ziel hierbei ist es, die Anzahl der Embeddings zu reduzieren, allerdings noch einen gewissen temporären Zusammenhang zu behalten.

Diese Verschaltung geschieht sowohl im Training des kNN-Klassifizierers als auch in der Inferenz. Hierbei muss die Variante allerdings nicht immer übereinstimmen. Im Zuge der Inferenz einer Inputdatei, ob in einem Test oder dem Modell-Validierungsschritt, werden für eine gegebene Audiodatei mehrere Embeddings extrahiert. Jedes davon muss für das Klassifikationsergebniss in Betracht gezogen werden. Abhängig von der Verknüpfungsmethode der Embeddings in der Validierung, entstehen ein oder mehrere Embeddings die zur Klassifizierung herangezogen

werden müssen. Im Falle von *plain* und *tripplets_mean* können mehrere entstehen, welche in einem Voting Verfahren behandelt werden müssen. , Dort wird die Klasse verwendet, wo die Anzahl der einzelnen Klassifizierungen über einem eingestellten Schwellenwert liegt. Im Falle der *mean* Methode wird kein prozentuales Voting durchgeführt, da naturgemäß nur jeweils ein Embedding nach der Verarbeitung übrig bleibt. Empirisch werden die Parameter *k* für den kNN, der Schwellwert für das Voting und die Embedding Verknüpfungsmethode gewählt. Die Wahl der Verschaltungsmethode ist ein Hyperparameter und wird für Training des kNN und Modelinferenz separat eingestellt.

kNN Implementierung

Der kNN Algorithmus bietet sich für diese Problemstellung gut an, da er mit vielen und hochdimensionalen Daten gut umgehen kann und diese Daten in gleichartige Cluster gruppiert (Bhatia und Author , 2010). Je nach der gewählten Verknüpfungsmethode für das Trainieren der Embeddings werden diese in den Datensatz des kNN-Klassifikators geladen. Jede Dimension eines Embeddings wird als neue Dimension im Klassifikator verwendet. Die Distanzmetrik ist die Standardmäßige Euklidische-Distanz welche gewichtet verwendet wird. Die verwendeten Funktionen um die Embeddings zu verschalten und den kNN-Klassifikator zu erzeugen sind in Quellcode 3.6.1 dargestellt:

```
@staticmethod
def combine_embeddings(embeddings, embedding_mode):
  if embedding_mode == "mean":
    embeddings = torch.mean(embeddings, dim=1)
  elif embedding_mode == "tripplets_mean": # mean the 3 successive
    embeddings
    if embeddings.shape[1] % 3 != 0:
      remainder = embeddings.shape[1] % 3
      # mean of the remaining embeddings
      last_embeddings = torch.mean(embeddings[:, -remainder:], dim
   =1, keepdim=True)
      embeddings = embeddings[:, :-remainder]
      # Reshape und Mittelung der 3er-Blöcke
      embeddings = embeddings.reshape(embeddings.shape[0], -1, 3,
   embeddings.size()[2])
      embeddings = torch.mean(embeddings, dim=2)
      embeddings = torch.cat([embeddings, last_embeddings], dim=1)
    else:
      embeddings = embeddings.reshape(embeddings.shape[0], -1, 3,
   embeddings.size()[2])
```

```
17        embeddings = torch.mean(embeddings, dim=2)
18    elif embedding_mode == "max":
19      embeddings = torch.max(embeddings, dim=1)[0]
20    elif embedding_mode == "sum":
21      embeddings = torch.sum(embeddings, dim=1)
22    elif embedding_mode == "plain":
23      pass # do not change anything
24    if len(embeddings.shape) == 2:
25      embeddings = embeddings[:, None, :]
26    return embeddings
27
28  def build_nn_classifier(n_neighbors=3, distance_metric='euclidean'
      , embedding_mode='mean', assume_positive_p=0.3):
29    classifier = sklearn.neighbors.KNeighborsClassifier(n_neighbors=
      n_neighbors, weights='distance', metric=distance_metric)
30    n_neighbors = n_neighbors
31    embedding_mode = embedding_mode    # used later in embedding
      extraction
32    assume_positive_p = assume_positive_p    # used later for the
      positive class assumption
33    classifier.fit(neighbor_data, neighbor_labels)
```

Quellcode 3.6.1: Funktionen der Embedding Verknüpfung und Erzeugung des Klassifikators

Dieser hier gezeigten Ausschnitt in Quellcode 3.6.1 ist in die PyTorch Model Struktur eingearbeitet. Vorteil hierbei ist, dass dieses PyTorch Modell den kNN-Klassifikator, zusammen mit den trainierten Embeddings beinhaltet. Zudem ist ebenfalls der Embedding Extraktor vorhanden. Hierdurch kann das Model als eigenständiges Objekt verteilt werden und die üblichen Model forward Funktionen sind identisch zu anderen Modellen. Dadurch wird für die Inferenz eine Austauschbarkeit der Modelle zwischen Embedding und CNN Klassifikatoren ermöglicht.

3.6.3 Beispiel Ablauf

Gegebene Annahmen:

- 10 Audiodateien, 22 Sekunden Länge, 1 Inferenz Audiodatei
- Chunklänge: 10 Sekunden
- YAMNet Klassifikator
- Training Verschaltungsmethode plain
- Inferenz Verschaltugnsmethode mean

Methodischer Ablauf:

Splitting in Chunks

Jede Audiodatei wird in 10 Sekunden Chunks unterteilt. Es entstehen pro Datei drei Chunks mit den Längen 10, 10 und 2 Sekunden. Der letzte Chunk wird verworfen da er nicht über den Threshold eines erlaubten unvollständigen Chunks kommt. Ansonsten wäre padding durchgeführt worden.
Resultat: 20 Chunks mit 10 Sekunden Länge.

Vorverarbeitung

Die Samplerate wird angeglichen, der Butterpass Filter wird angewendet und das Signal normalisiert. Im Trainingsmodus wird eine Augmentierung des Signals durchgeführt. **Resultat:** 20 Chunks, jeder mit einer Länge von 10 Sekunden.

YAMNet Examples extrahieren

Die YAMNet spezifische Vorverarbeitung wird durchgeführt. Erneutes Resampling und transformieren des Inputsignals in das Mel-Spektrogramm. Berechnung der YAMNet spezifischen Patches. 0.96 Sekunden mit 0.48 Sekunden Überlappung. **Resultat:** 19 Patches pro 10 Sekunden Chunk. Total $19 \times 20 = 380$ Input Beispiele.

Embedding extraktion

Eingabe der Spektrogramm-Patches in das Netzwerk. Ausgabe: 1 Embedding pro Patch. **Resultat:** 19 Embeddings pro 10 Sekunden Chunk. $20 \times$ ein19×1024 Embedding.

Verarbeitung mit kNN

Keine Zusammenführung der 19×1024 Embeddings da *plain* angenommen wurde. Hinzufügen in kNN Klassifikator Datensatz. **Resultat:** ein kNN Klassifikator mit 380 Datenpunkten über 1024 Dimensionen und einem Label

Inferenz Vorbereitung

Die Datei für die Inferenz wird ebenso vorverarbeitet und die Embeddings extrahiert. Bei der Inferenz wird das 19×1024 Embedding durch die Anwendung der *mean* Methode zu einem 1×1024 Elementweise gemittelt.

Inferenz

Das einzelne Embedding wird mit kNN Klassifiziert. Für jede Dimension des Embeddings wird die Distanz im kNN Datensatz berechnet und die k nächsten Nachbarn ermittelt. Labelzuweisung durch Voting der am häufigsten vertretenen Label. **Resultat:** Wahrscheinlichkeiten für jede Klasse, Schlussendlich ein einziges Label.

Diese Auflistung ziel ab einen generellen Überblick über die Abläufe zu zeigen und zeigt wie sich die Dimensionen verändern. Folgende Abbildung Abbildung 3.5 zeigt VGGish Embeddings einer Audiodatei vor und nach der Mittelung mit der *mean* Methode:

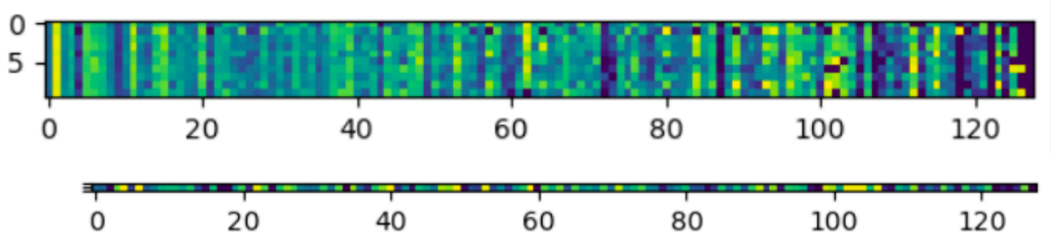

Abbildung 3.5: Oben: Ein 10×128 Embedding Stack eines Chunks. Unten: Beispielhaft gemitteltes 1×128 Embedding eines anderen Stacks

VGGish Anpassungen

Der Ablauf von Training und Inferenz des VGGish Klassifizierers verläuft sehr ähnlich wie bei dem YAMNet Modell. Durch die zuvor beschriebene modulare Codeaufteilung ermöglicht Sie nahezu vollständige Wiederverwendbarkeit und es müssen lediglich Konfigurationsobjekte angepasst werden. Die Dimension der VGGish Embeddings beträgt nur 1×128 entgegen der von YAMNet genutzten 1×1024. Die Trainingspipeline und vor allem das Klassifikationsmodell sind aber so konzipiert, dass eine unterschiedliche Inputgröße nicht den Ablauf beeinträchtigt und in jedem Fall Training und Inferenz stattfinden können. Aufgrund der unterschiedlichen Form sind die Modelle zwischen Embeddingextraktion und kNN Klassifikation aber nicht untereinander austauschbar. Also kann ein mit YAMNet-Embeddings erstellter Klassifikator keine Inputs von VGGish Extraktoren verwenden.

3.7 Weitere Implementierungsdetails

Im Rahmen der Entwicklung dieser Anwendung wurden zahlreiche zusätzliche Funktionen implementiert, die insbesondere zur Überwachung und Optimierung des Trainingsprozesses beitragen. Diese Funktionen greifen nicht direkt in den Trainingsverlauf ein und beeinflussen somit nicht die Ergebnisse der Klassifizierung. Dennoch tragen sie erheblich dazu bei, einen Einblick in den Trainingsfortschritt zu gewähren und die Effizienz des Trainingsprozesses zu steigern. Ein wichtiger Aspekt dieser Zusatzfunktionen ist ihre universelle Anwendbarkeit in verschiedenen Projektkontexten, was durch eine modulare Gestaltung mit vielseitigen Fallunterscheidungen erreicht wird. So ist es möglich, diese Funktionen auch in Projekten mit Multiclass-Klassifizierung ohne zusätzliche Anpassungen zu verwenden.

3.7.1 Metric Tracker

Um einen detaillierten Überblick über die Leistungsfähigkeit des Modells während des Trainings und der Validierung zu erhalten, wird ein spezieller Metric Tracker

verwendet. Dieser Tracker ist so konzipiert, dass er sich flexibel an verschiedene Metrik-Typen anpassen kann und nicht ausschließlich auf die Module von PyTorch torchmetrics angewiesen ist. Ein zukünftiges Ziel ist es, den Metric Tracker so zu erweitern, dass er auch Metriken für andere Arten von Klassifikatoren, wie beispielsweise XGBoost, verwalten kann. In solchen Fällen könnten die Metriken mit Werkzeugen wie *sklearn* berechnet und vom Tracker überwacht werden. Dies würde den Metric Tracker zu einem universell einsetzbaren Werkzeug machen.

3.7.2 WANDB Tracking

Weights and Biases, vertrieben unter https://wandb.ai/ (abgerufen am 15.12.2023), ist ein modernes Entwicklertool, das darauf abzielt, den Machine Learning-Workflow zu optimieren. Es bietet eine umfangreiche Plattform für das Tracking von Experimenten, die Visualisierung von Daten und die Analyse von Modellleistungen. WANDB wird insbesondere für die detaillierte Überwachung und Optimierung von Machine Learning-Projekten verwendet.

Vorteile

Der Einsatz von WANDB bietet zahlreiche Vorteile für das Tracking und die Analyse von Trainingsdaten. Durch die einfache Integration in den Trainingsprozess ermöglicht WANDB eine effiziente Datenerhebung und -verarbeitung. Die generierten Daten lassen sich leicht filtern, sortieren und analysieren. Ein besonders nützliches Feature von WANDB sind die sogenannten *Sweeps*, die eine automatisierte Optimierung von Hyperparametern ermöglichen. Dabei können relevante Einstellungen an bayesische, random oder grid-basierte Optimierungsalgorithmen übergeben werden, deren Ergebnisse anschließend grafisch visualisiert werden. Die verschiedenen Parameter werden durchprobiert und die Wichtigkeit der einzelnen Werte berechnet. WANDB bietet zudem Unterstützung bei der Verwaltung von Datensätzen und Modellen. In sogenannten Artefakten können beispielsweise Bilder, Overlays oder ganze Modelle gespeichert und über eine API abgerufen werden.

Nachteile

Trotz der vielen Vorteile gibt es auch einige Nachteile bei der Verwendung von WANDB. Ein wesentlicher Nachteil ist, dass das Tracking tief in die Trainingsroutine eingebettet ist, was es schwierig macht, Trainingsläufe ohne Tracking durchzuführen. Die Konfiguration der Trainingsparameter ist stark von der Integration mit WANDB abhängig. Ein weiteres Problem kann eine schlechte Internetverbindung sein, da WANDB eine Serververbindung benötigt, um Tracking-Daten zu übermitteln. Zwar bietet WANDB die Möglichkeit, Trainingseinheiten offline durchzuführen, jedoch hat sich dies in der Praxis als nicht immer zuverlässig erwiesen. Die Komplexität der WANDB-Implementierung steigt insbesondere bei der Verwendung über mehrere k-folds hinweg oder bei der Kombination mit Sweeps und Epochen. Dadurch kann die Flexibilität eingeschränkt werden und es müssen Kompromisse eingegangen werden. Besonders problematisch ist die Kombination von Sweeps und k-fold, die vermieden werden sollte. Ein Fehler in dem Service verhindert das Gruppieren der einzelnen Durchläufe und das weiterreichen der Sweep-config. Dies wurde im Zuge dieser Einwicklung hier als GitHub-Issue eröffnet und das Problem anerkannt, allerdings bis heute ohne Lösung https://github.com/wandb/wandb/issues/5119 (Stand 17.12.2023). Eine mögliche Lösung ist es, die Ergebnisse einzelner k-fold Läufe zu mitteln und nur einen einzelnen großen WANDB Run zu erstellen. Alternativ können Parameter mittels Sweeps in einem normalen Run geprüft und die Validierung dann in einem separaten Run unter Verwendung von k-fold durchgeführt werden. In dieser Anwendung wird das Tracking der Metriken unter anderem mittels WANDB realisiert. Erstellte Grafiken werden gespeichert, jedoch wird die Modellarchivierungsfunktion von WANDB nicht genutzt. Das Filtern und Analysieren verschiedener Experimente hat es ermöglicht, effektive Parameterkombinationen zu identifizieren und weniger relevante Einstellungen zu verwerfen. Dies geschah teilweise manuell durch tabellarische Auswertungen und teilweise automatisiert durch Sweeps.

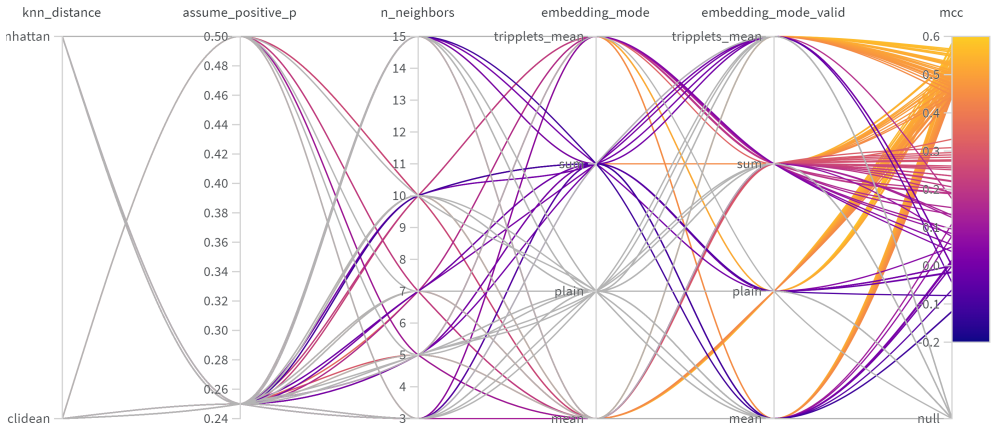

Abbildung 3.6: Ein beispielhaftes Ergebnis einer Sweep-Optimierung zur Parameterfindung des kNN Klassifikators.

In Abbildung 3.6 wird die Ausgabe einer Sweep-Optimierung dargestellt, bei der verschiedene Einstellungen für den Parameter *k* und unterschiedliche Embedding-Verknüpfungsmethoden getestet wurden. Eine wichtige Erkenntnis aus diesem Sweep ist, dass die *plain* Methode zu diesem Zeitpunkt nicht korrekt implementiert zu sein scheint, da sie *null* als Ergebnis liefert. Weiterhin zeigt sich, dass die *sum* Methode nur mittelmäßige Ergebnisse erzielt, während die *mean* Methoden tendenziell besser abschneiden.

Kapitel 4

Experimente und Ergebnisse

4.1 Methodik

Die Auswahl der Metriken für die Evaluierung der Modelle muss sorgfältig erfolgen, um eine präzise und aussagekräftige Einschätzung zu ermöglichen. Um eine umfassende Bewertung der verschiedenen Klassifikatoren vorzunehmen, werden diverse Evaluationsmetriken herangezogen. Sie werden wie folgt definiert:

Confusion Matrix

Die confusion matrix (CM, „Verwechslungsmatrix“) bietet eine detaillierte Darstellung der Klassifikationsleistung und wird unter anderem in Chicco (2017) genauer beschrieben und hier wiedergegeben. In der CM werden die Ergebnisse einer binären Klassifikation in vier Hauptkategorien unterteilt, die jeweils verschiedene Aspekte der Leistung des Modells darstellen. Diese Kategorien sind:

- **True Positive (TP)**: Dies sind die Fälle, in denen das Modell korrekt vorhersagt, dass die positive Klasse zutrifft.
- **False Positive (FP)**: In diesen Fällen sagt das Modell fälschlicherweise voraus, dass die positive Klasse zutrifft, obwohl dies nicht der Fall ist.

- **True Negative (TN)**: Hier hat das Modell korrekt erkannt, dass die negative Klasse zutrifft.
- **False Negative (FN)**: Das Modell sagt fälschlicherweise voraus, dass die negative Klasse zutrifft, obwohl die positive Klasse zutreffen würde.

Diese vier Kategorien bieten eine ganzheitliche Sicht auf die Leistung des Klassifikationsmodells und ermöglichen eine detaillierte Analyse von dessen Stärken und Schwächen.

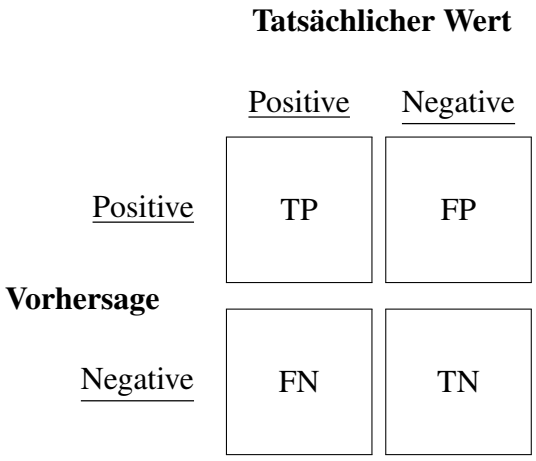

Abbildung 4.1: Aufbau einer binären CM

Diese Werte werden im binären Fall zu einer 2×2 Matrix zusammengeführt. Die Anordnung ist variabel, muss aber deutlich beschriftet werden. In Abbildung 4.1 wird die hier verwendete Form dargestellt.

Accuracy

Die Accuracy, oder Genauigkeit, ist eine weitverbreitete Metrik zur Bewertung von Klassifikatoren und beschreibt die Wahrscheinlichkeit eines Klassifikators, die Vorhersage korrekt zu treffen (Chicco , 2017). Problematisch ist eine Klassenimbalance der Daten, wodurch die Accuracy leicht falsch geschätzt werden kann (Stoica und Babu , 2023).

$$\text{Accuracy} = \frac{TP+TN}{TP+TN+FP+FN} \tag{4.1}$$

Specificity

Die Specificity, oder true negative rate (TNR), gibt an, wie gut das Modell die negative Klasse identifiziert und ist daher insbesondere dann relevant, wenn die Kosten für falsch positive Ergebnisse hoch sind (Weiß und Rzany , 2013).

$$\text{Spezifität} = \frac{TN}{TN+FP} \tag{4.2}$$

Recall

Der Recall, in der Statistik Sensitivität oder alternativ true positive rate (TPR) genannt, gibt die Wahrscheinlichkeit für einen Klassifikator an, die positive Klasse auch korrekt zu erkennen (Hajian-Tilaki , 2013).

$$\text{Recall} = \frac{TP}{TP+FN} \tag{4.3}$$

Precision

Die Precision bewertet die Genauigkeit bei der als positiv klassifizierten Instanzen Chicco und Jurman (2023).

$$\text{Precision} = \frac{TP}{TP+FP} \tag{4.4}$$

F1-Score

Der F1-Score beschreibt nach Chicco und Jurman (2023) das Verhältnis von TP und TN über alle Elemente sowie Precision und Recall. Er könne allerdings zu Optimistischen Ergebnissen verleiten, was insbesondere bei positiv unbalancierten Datensätzen auftrete.

$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4.5}$$

Matthews Correlation Coefficient (MCC)

Der MCC ist eine komplexe aber robuste Metrik und wird in Chicco und Jurman (2023) genau erklärt. Vorteil sei, dass diese Metrik selbst bei unausgeglichenen Klassenverteilungen zuverlässige Ergebnisse liefert. Dies wird erreicht, indem entgegen zu anderen Metriken jede Feld der CM betrachtet wird. Die positive und negativen Klassenkomponente wird gleichwertig einbezogen und es ist nicht relevant, entgegen dem F1-Score, welche Klasse in der Berechnung als positiv angesehen wird.

$$\text{MCC} = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP) \times (TP+FN) \times (TN+FP) \times (TN+FN)}} \tag{4.6}$$

Höhere Werte zeigen laut dem Autor eine bessere Vorhersagegenauigkeit des Modells an. Die Robustheit und ganzheitliche Betrachtung kommt allerdings mit dem Nachteil, dass die Interpretation geringfügig komplizierter ist. Die Werte befinden sich nicht wie üblich im Prozentbereich 0 bis 1, sondern reichen von -1 bis $+1$. Wobei ein Wert von 0 ein zufälliges Raten beschreibt, -1 konstant gegensätzliche Klassifizierung und +1 perfekte Ergebnisse bedeutet. Somit ist diese Metrik nicht direkt mit anderen wie zum Beispiel dem F1-Score vergleichbar.

positive likelihood ratio (PLR)

Die PLR nach (Weiß und Rzany , 2013, S.261f) gibt an, wie viel wahrscheinlicher ein positives Testergebnis bei einem tatsächlich positiven Fall im Vergleich zu einem negativen Fall ist. Eine höhere PLR deutet auf eine bessere Leistung des Modells bei der Identifikation der positiven Klasse hin. Für einen leistungsfähigen Test sollte der Wert über 3 liegen.

$$\text{PLR} = \frac{\text{Sensitivität}}{1 - \text{Specificity}} \tag{4.7}$$

negative likelihood ratio (NLR)

Die NLR beschreibt, wie viel wahrscheinlicher ein negatives Testergebnis bei einem tatsächlich negativen Fall im Vergleich zu einem positiven Fall ist. Eine niedrige NLR deutet auf eine bessere Leistung des Modells bei der Identifikation der negativen Klasse hin. Liegt der Wert unter $1/3$, wird der Test als Leistungsfähig bezeichnet (Weiß und Rzany , 2013, S.261f).

$$\text{NLR} = \frac{1 - \text{Sensitivität}}{\text{Specificity}} \tag{4.8}$$

ROC-Kurve

Die *Receiver Operating Characteristic Curve* (ROC-Kurve) ist ein leistungsstarkes Werkzeug zur Bewertung und zum Vergleich von binären Klassifikatoren und wird von Hajian-Tilaki (2013) genauer erklärt. So sei Hauptvorteil die Fähigkeit, die Leistung eines Modells unter verschiedenen Schwellwerten für eine positive Klassifizierung zu visualisieren. Auf der x-Achse der ROC-Kurve wird die False positive r ate (FPR) aufgetragen, und auf der y-Achse die TPR. Ein idealer Klassifikator hätte eine TPR von 1 und eine FPR von 0 und würde daher den oberen linken Punkt des Diagramms treffen. Die ROC-Kurve ermöglicht eine Visualisierung der Sensitivität und Spezifität, bei unterschiedlichen Schwellwerten.

AUROC

Weitergehend wird ebenfalls in Hajian-Tilaki (2013) das *Area-under-the-Receiver-Operating-Characteristic-Curve* (AUROC), dort Area-under-the-Curve (AUC) genannt, als abgeleitete Metrik beschrieben und hier folgend erklärt. AUROC setzt sich aus der Fläche unter der ROC-Kurve zusammen. Ein AUROC-Wert von 1.0 bedeutet eine perfekte Vorhersage, während ein Wert von 0.5 nicht besser ist als ein zufälliges Raten. Vorteilhaft sei hier die Quantifizierung der ganzen ROC-Kurve in einer einzelnen Zahl.

Aufgrund der hier vorliegenden starken Klassenimbalance wird der MCC als primäre Vergleichsmetrik der Modelle herangezogen. Dennoch sollte die Arbeit von Zhu (2020) beachtet werden, wo untersucht wird, dass auch der MCC bei unausgeglichen Datensätzen geringe Aussagekraft hat, insbesondere bei extremem Klassenimbalancen.

Die hier vorgestellten Metriken werden sowohl für das CNN-Modell als auch für die YAMNet und VGGish Modelle erhoben. Für das CNN-Modell werden die Metriken für jede Epoche als Mittelwert über alle Folds dargestellt. Bei den YAMNet und VGGish Modellen wird lediglich der Mittelwert über alle Folds berücksichtigt, da diese Modelle nicht epochenspezifisch trainiert werden. Alle ermittelten Durchschnittswerte werden zusätzlich durch ihre Stichprobenstandardabweichung ergänzt. Zudem wird die Inferenzgeschwindigkeit der Modelle gemessen. Eine Bewertung der Metriken dort wird nicht durchgeführt, da kein unabhängiger Testdatensatz vorliegt und lediglich zu Demonstrationszwecken auf den Trainingsdaten die Geschwindigkeit getestet wird. Diese ist lediglich als Orientierung zu sehen, da das Experiemnt nicht statistisch signifikant oft ausgeführt ist um Schwankungen der verwendeten Maschine auszugleichen. Alle Messungen verlaufen unter gleichen Bedingungen. Die Modellgröße ist geschätzt, da unter verschiedenen Folds die Embedding Modelle unterschiedliche Größen aufweisen. Die Ergebnisse werden in Tabelle 5.1 zusammengefasst.

4.2 Ergebnisse des CNN-Modells

Folgende Einstellungen des Trainingsablaufes für den CNN Klassifikator sind empirisch ermittelt und verwendet:

- Epochen: 70, Folds: 10
- Augmentation: 65%
- Audiochunk Länge: 7s
- Dropouts: Pos1 0.5 und Pos2 0.2
- Batchsize: 90
- initiale Lernrate: 0.01
- Loss-Regularisierung: Keine
- Batchnormalisation: Ja
- Aktivierung: SiLU
- Criterion: FocalLoss mit $\gamma = 2$
- Filterung: Butterworth Bandpass mit 4th Order 25 Hz-400 Hz
- Optimierer: SGD
- Scheduler: StepLR mit patience=10, factor=0.2

Tabelle 4.1: Evaluationsmetriken gemittelt über alle 10 Folds in Epoche 70

Metrik	Training	Validierung
Accuracy	0.9124 ± 0.0056	0.9032 ± 0.0156
Specificity	0.9415 ± 0.0038	0.9095 ± 0.0133
Recall	0.8180 ± 0.0138	0.8804 ± 0.0463
Precision	0.8125 ± 0.0130	0.7472 ± 0.0498
F1	0.8151 ± 0.0128	0.8077 ± 0.0421
MCC	0.7578 ± 0.0164	0.7485 ± 0.0508
NLR	0.1933 ± 0.0152	0.1314 ± 0.0517
PLR	14.0494 ± 1.1177	9.9599 ± 1.5729
AUROC	0.9688 ± 0.0039	0.9647 ± 0.0095

Die Verteilung der positiven Klassen beträgt: mean = 0.2363, (min, max) = $(0.2318, 0.2397)$ über alle Folds. Durchschnittliche Inferenzeit: 66.11 Chunks/Sekunde. Komplette Modellgröße im PyTorch Speicherformat: 33.2 Mb

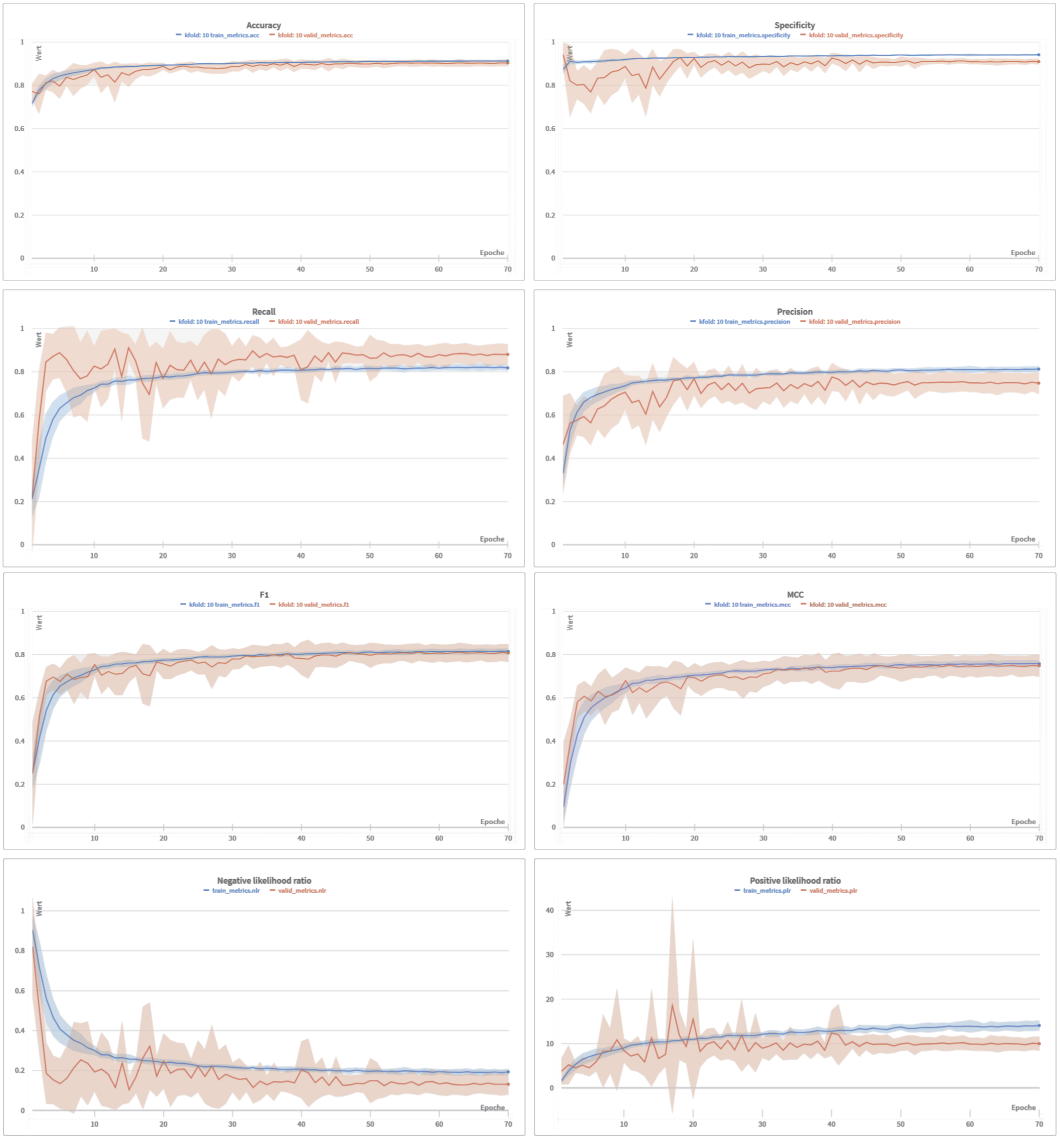

Abbildung 4.2: Verlauf aller aufgezeichneten Metriken während Training und Validierung über alle 10 Folds gemittelt

Diese Abbildung 4.2 zeigt den Verlauf der Metriken, über alle Epochen hinweg und über alle Folds gemittelt, graphisch auf. Die Linien beschreiben die durchschnittlichen Metrikwerte und die farbliche Schattierung beschreibt die Unsicherheit der

1± Standardabweichung. Die Graphen sind durch das Tracking mit WANDB erstellt worden. In jedem Graph zeigt sich der schnell stabilisierte Trainingsverlauf. Der Validierungsverlauf ist in jedem Graphen mit stärkeren Ausreißern behaftet was durch eine ungelmäßigerere und breitere Schraffierung der Fläche ersichtlich wird. Insbesondere beim Recall hat das Modell anfangs stärkere Probleme die Klassifizierung auf dem Validierungsdatensatz korrekt durchzuführen. Andere Metriken wie die Accuracy hingegen pendeln sich schnell und höher ein, was allerdings aufgrund der starken Klasseninbalance nicht überraschend ist.

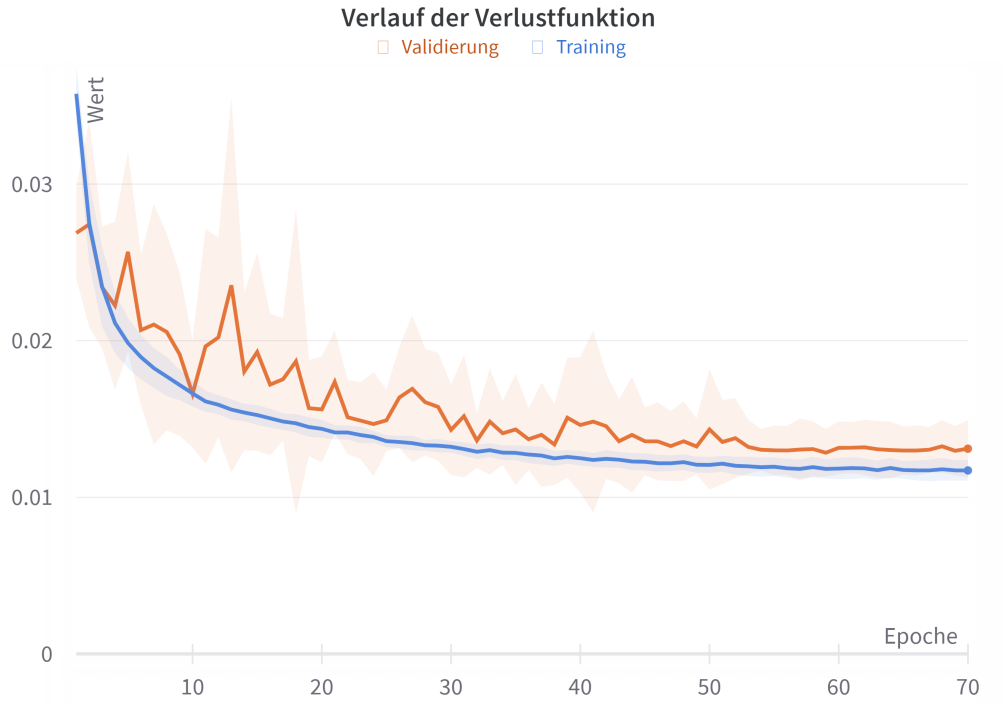

Abbildung 4.3: Verlauf der FocalLoss Verlustfunktion für Training (blau) und Validierung (orange) über alle 10 Folds gemittelt.

In dieser Abbildung 4.3 ist der Verlauf der Verlust-Funktion ersichtlich. Man erkennt den schnellen, starken Abfall der nach etwa 30 Epochen beginnt ein Plateau zu bilden. Dies geschieht im Einklang mit den zuvor erkannten Plateaus der anderen Metriken in dem selben Bereich. Ungefähr ab Epoche 55 verringert sich die durch Schattierung dargestellte Standardabweichung merklich. Die Werte werden robuster und nur geringe Abweichungen sind zu verzeichnen.

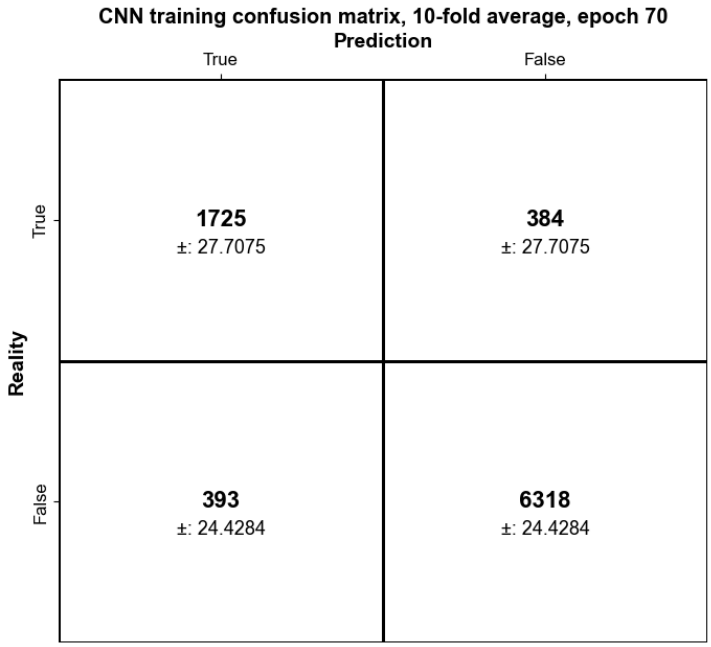

Abbildung 4.4: Über alle Folds gemittelte CM für Training

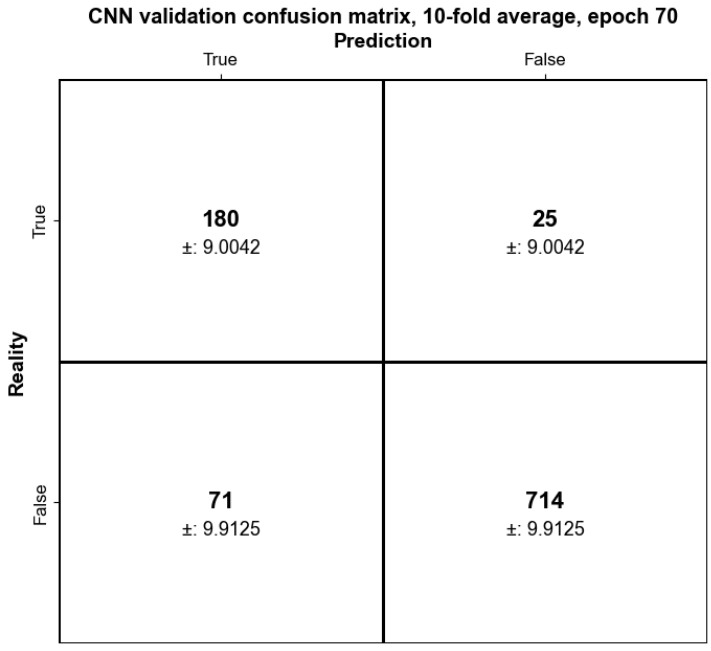

Abbildung 4.5: Über alle Folds gemittelte CM für Validierung

Die CMs für Training und Validierung in Abbildung 4.4 und Abbildung 4.5 zeigen eine ähnliche TPR und TNR, was auf eine konsistente Leistung des Modells hindeutet. Zu beachten ist die identische Standardabweichung bei TP/FP sowie FN/TN Feldern. Diese beruhen auf die Berechnungsweise welche die Werte über zuvor normalisierte Werte, Sensitivity, Specificity und der Anzahl der Label pro Klasse verwendet. Die Werte der Felder sind dem Klassifikator entsprechend, allerdings im Zusammenhang mit der Mittlung über alle Folds, entsteht diese Eigenschaft und sollte bei der Interpretation der Standardabweichungen beachtet werden. Dies ist lediglich bei dem CNN Modell relevant.

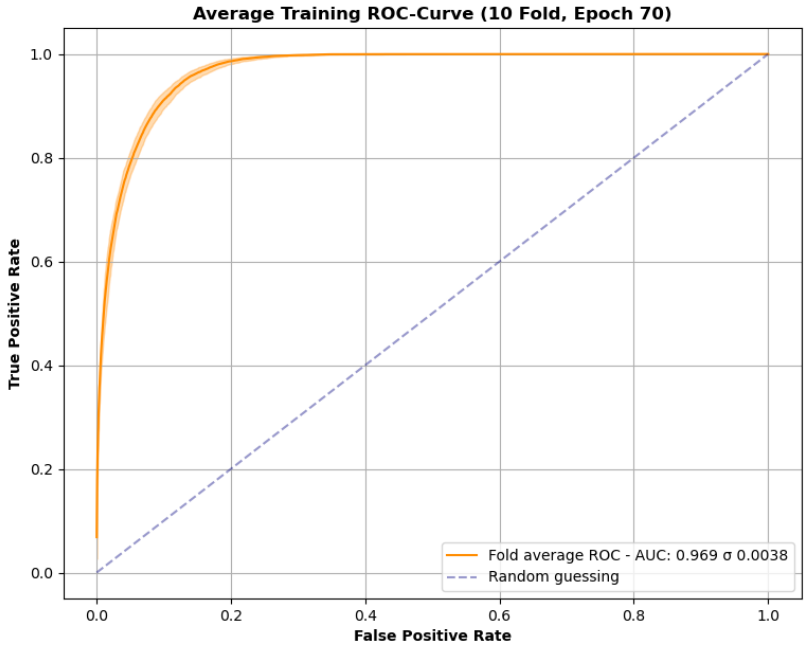

Abbildung 4.6: ROC-Kurve des Modells vom Training in Epoche 70, gemittelt über alle Folds

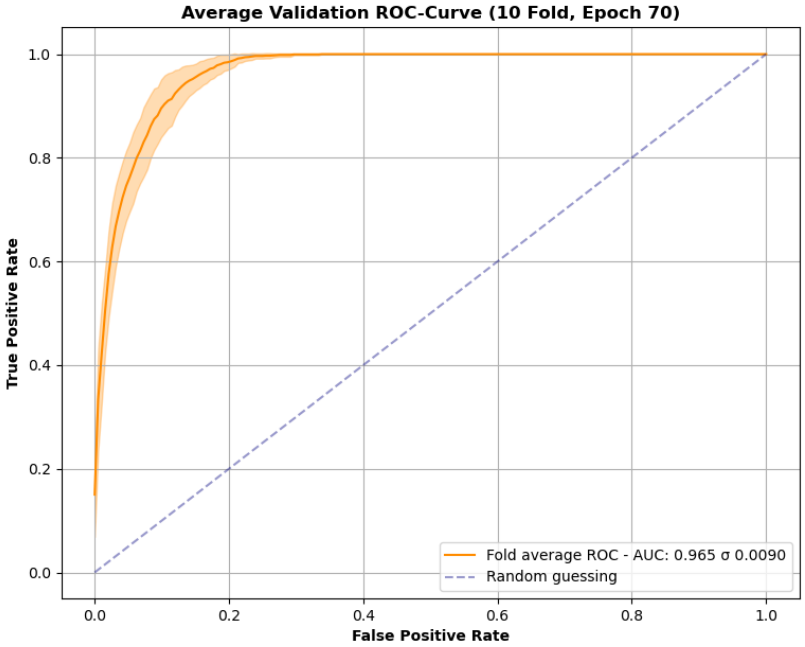

Abbildung 4.7: ROC-Kurve des CNN Modells vom Validieren in Epoche 70, gemittelt über alle Folds

Die in Abbildung Abbildung 4.6 dargestellte ROC-Kurve zeigt das durchschnittliche Verhalten des Modells über alle Folds während des Trainingsprozesses in der 70. Epoche. Die Schattierungen zeigen die Standardabweichung an. Die AUROC von ca. 0.969 zeigt an, dass das Modell eine auf den Trainingsdaten gut zwischen den beiden Klassen unterscheiden kann.

In Abbildung 4.7 wird das Verhalten des Modells während der Validierung in der gleichen Epoche dargestellt. Ein AUROC Wert von ca. 0.965 ist ebenfalls hoch und nur geringfügig schlechter. Das deutet darauf hin, dass das Modell nicht nur auf den Trainingsdaten, sondern auch auf den Validierungsdaten gut funktioniert. Diese hohen Werte deuteten also darauf hin, dass das Modell eine ausgezeichnete Fähigkeit hat, zwischen positiven und negativen Klassen zu unterscheiden.

4.3 Ergebnisse des YAMNet-Modells

Der kNN Parameter wird auf $k = 3$ festgelegt. Sowohl während des Trainings als auch während der Inferenz wird der Mittelwert der Embeddings (*mean* Methode) als Eingabe für die kNN-Klassifikation verwendet. Das zuvor beschriebene Voting-Schwellwert fällt also weg. Tabelle 4.2 zeigt die durchschnittlichen Metriken über alle 10 Folds hinweg für die Validierung.

Tabelle 4.2: Durchschnittliche Evaluationsmetriken des YAMNet-Modells über alle 10 Folds

Metrik	Validierung
Accuracy	0.8494 ± 0.0158
Specificity	0.9172 ± 0.0137
Recall	0.7746 ± 0.0237
Precision	0.7955 ± 0.0245
F1-Score	0.7833 ± 0.0231
MCC	0.5695 ± 0.0450
PLR	9.5848 ± 1.5943
NLR	0.2459 ± 0.0273
AUROC	0.8769 ± 0.0225

Der MCC als bevorzugte Metrik mit einem Wert von 0.5695 kann so als ausreichend eingestuft werden. Das Modell zeigt eine hohe Spezifität, jedoch geringere Werte für Recall und Precision, was auf bestimmte Herausforderungen bei der Klassifizierung hindeuten könnte.

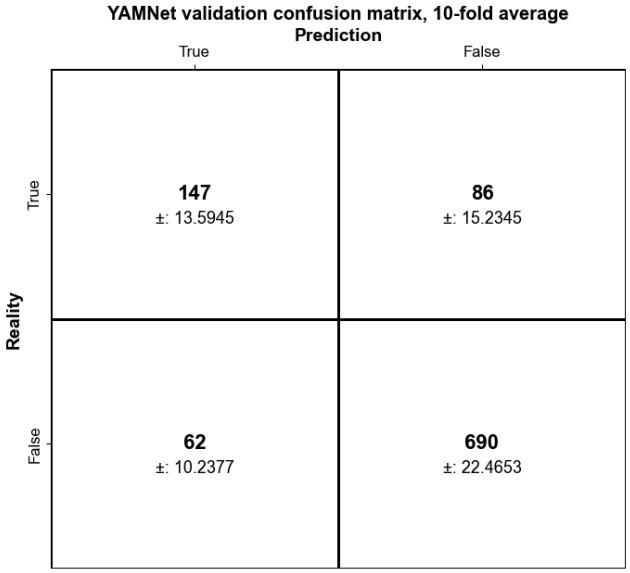

Abbildung 4.8: Über alle Folds gemittelte CM für die Validierung des YAMNet Klassifikators

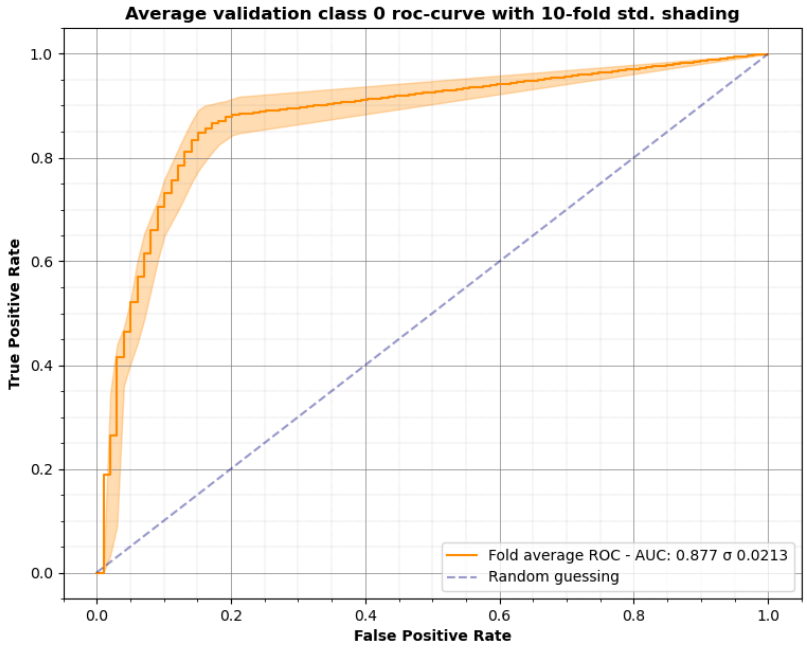

Abbildung 4.9: ROC-Kurve des YAMNet Klassifikators, gemittelt über alle Folds

Die CM in Abbildung 4.8 zeigt ähnliche Ergebnisse wie die Metriken und weist auf eine gute Spezifität und ausreichenden Recall hin. Die ROC-Kurve in Abbildung 4.9 verläuft anders als die der CNN-Modelle. Allerdings im Einklang mit den anderen Metriken erwartet schlechter, was sich auch im AUROC zeigt. Durchschnittliche Inferenzzeit: 15.76 Chunks/Sekunde. Komplette Modellgröße im PyTorch Speicherformat, inklusive Embededdings und Extraktor: ca. 49 Mb.

4.4 Ergebnisse des VGGish-Modells

Der kNN Parameter wird auf $k = 5$ festgelegt. Wie beim YAMNet-Modell wird sowohl während des Trainings als auch der Inferenz der Mittelwert der Embeddings für die kNN-Klassifikation genutzt. Tabelle 4.3 zeigt die durchschnittlichen Metriken über alle 10 Folds für die Validierung.

Tabelle 4.3: Durchschnittliche Evaluationsmetriken des VGGish-Modells über alle 10 Folds

Metrik	Validierung
Accuracy	0.8406 ± 0.0102
Specificity	0.9213 ± 0.0125
Recall	0.7504 ± 0.0170
Precision	0.7851 ± 0.0256
F1-Score	0.7645 ± 0.0195
MCC	0.5342 ± 0.0404
PLR	9.7439 ± 1.5926
NLR	0.2711 ± 0.0209
AUROC	0.8819 ± 0.0089

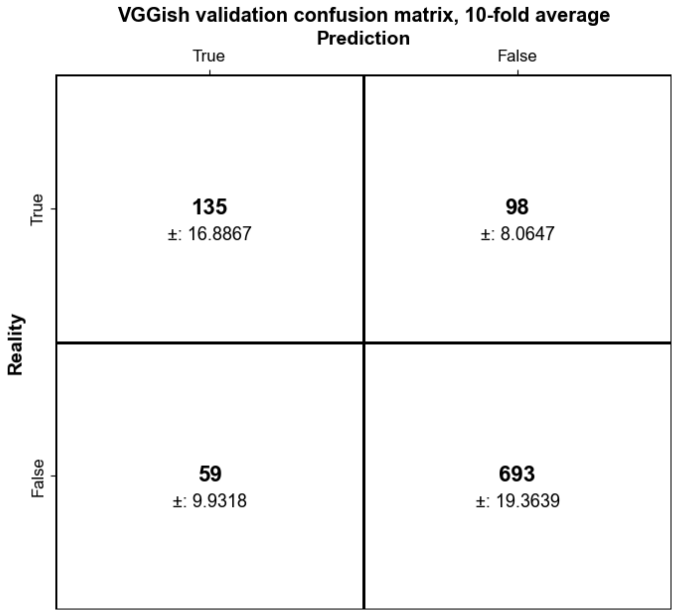

Abbildung 4.10: Über alle Folds gemittelte CM für die Validierung des VGGish Klassifikators

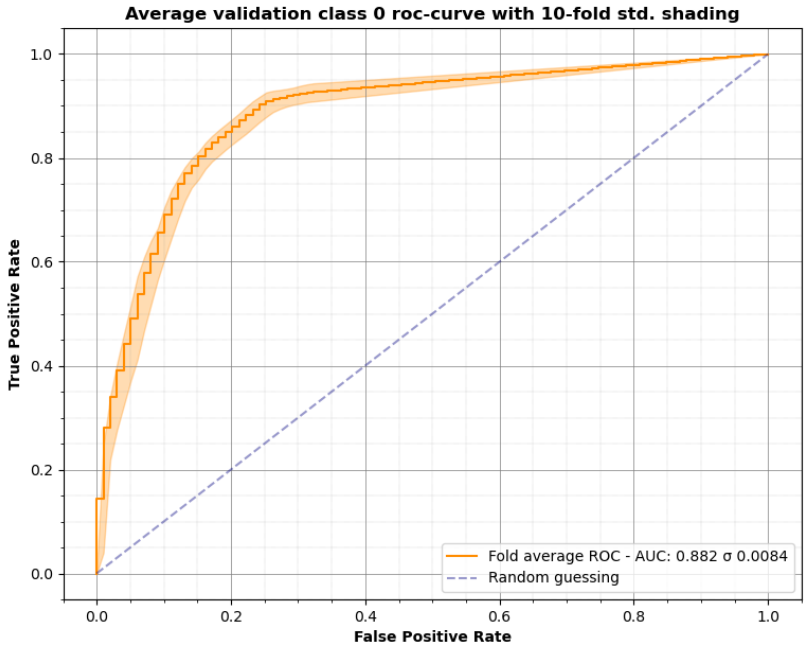

Abbildung 4.11: ROC-Kurve des VGGish Klassifikators, gemittelt über alle Folds

Die durchschnittlichen Metriken für das VGGish-Modell über alle 10 Folds zeigen eine robuste Performance in der Validierung. Die Accuracy des Modells beträgt 0.8406, was auf eine solide Gesamtleistung hinweist. Erneut gut ist die Specificity mit einem Wert von 0.9213, die die Fähigkeit des Modells unterstreicht, negative Fälle zuverlässig zu identifizieren. Der Recall-Wert von 0.7504 zeigt, dass das Modell einen angemessenen Anteil der positiven Fälle korrekt erkennt, obwohl hier noch Verbesserungspotenzial besteht. Die Precision mit 0.7851 verdeutlicht, dass viele der als positiv klassifizierten Fälle tatsächlich positiv sind. Der bevorzugte MCC Score von 0.5342 zeigt eine moderate Korrelation zwischen den vorhergesagten und den tatsächlichen Klassifikationen. Die PLR von 9.7439 und die NLR von 0.2711 bieten weitere Einblicke in die Leistungsfähigkeit des Modells bei der Unterscheidung zwischen positiven und negativen Fällen. Schließlich deutet der AUROC-Wert von 0.8819 auf eine sehr gute Fähigkeit des Modells hin, zwischen den Klassen zu unterscheiden. Die CM und ROC-Kurve in Abbildung 4.10 und Abbildung 4.11 bilden ein ähnliches Bild wie beim YAMNet Klassifikator und sind ebenfalls im Einklang mit anderen Metriken. Durchschnittliche Inferenzzeit: 10.56 Chunks/Sekunde. Komplette Modellgröße im PyTorch Speicherformat, inklusive Embeddings und Extraktor: ca. 279 Mb.

4.5 Vergleich der Ergebnisse

4.5.1 Vergleich aller Klassifikatoren untereinander

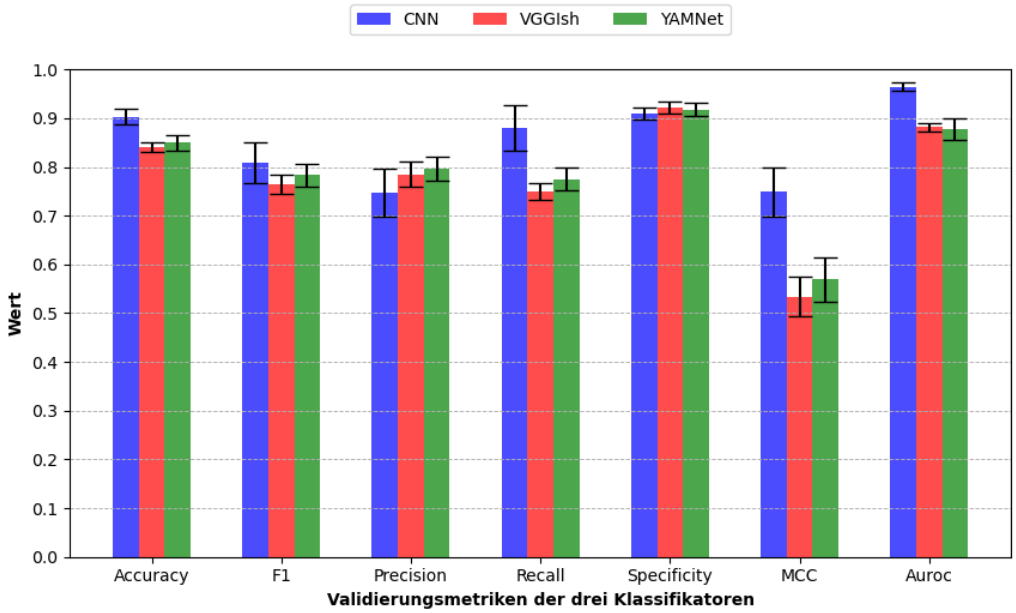

Abbildung 4.12: Balkendiagramm zum Vergleich der Evaluationsmetriken zwischen CNN, YAMNet und VGGish mit eingezeichneten Bereichen der Standardabweichung

Diese Abbildung 4.12 zeigt die finalen Validierungsmetriken, des Durchschnittes über alle 10 Folds, für jeden der drei Klassifikatoren. Zusätzlich sind die Standardabweichungen durch schwarze Striche eingezeichnet.

Embedding Klassifikatoren

Aus dieser Abbildung 4.12 wird neben den zuvor beschrieben Metriken ersichtlich, dass der YAMNet Klassifikator dem VGGish Klassifikator in Accuracy, F1, Precision, Recall und MCC überlegen ist. In den Metriken Specificity und AUROC ist das VGGish Modell geringfügig besser. Die Unterschiede sind allerdings nur geringfügig und unterscheiden sich nur um wenige Prozentpunkte. Zudem liegen die Unterschiede der Durchschnitte immer innerhalb der Reichweite der Standard-

abweichung. Insbesondere bei der AUROC Metrik ist die Standardabweichung des YAMNet Modells merklich größer als bei dem VGGish Modells. Ansonsten sind die AUROC vergleichbar ausgeprägt.

Wird die bevorzugte MCC Metrik betrachtet, beim YAMNet 0.5695 ± 0.0450 und bei VGGish 0.5342 ± 0.0404, wird die Annahme untermauert, dass der YAMNet Klassifikator marginal besser ist.

Vergleich zum CNN Klassifikator

Werden die Embedding Klassifikatoren mit dem CNN vergleichen, zeigt sich dass das CNN in fast allen Metriken überlegen ist. Nur bei der Precision liegt die Leistung unterhalb der anderen Modelle, wobei die Obergrenze der relativ großen Standardabweichung immer noch über den Durchschnitten der Embedding Modellen liegt. Die Unterschiede sind teilweise signifikant. So sind bei Accuracy, Recall MCC und AUROC die Metriken des CNN erheblich besser und liegen nicht mehr im Bereich einer Standardabweichung.

In der bevorzugten MCC Metrik zeigt sich dies deutlich. Das CNN Modell mit 0.7485 ± 0.0508 performt im Vergleich zu den Embedding Modellen, unter YAMNet 0.5695 ± 0.0450 und VGGish 0.5342 ± 0.0404, markant besser. Auch wenn die CNN Standardabweichung größer ausfällt, ist der Unterschied dennoch gering und die Durchschnittswerte liegen nicht in dessen Reichweite.

Kapitel 5

Diskussion

5.1 Interpretation der Ergebnisse

5.1.1 Bewertung der Metriken

CNN Modell

Das CNN-Modell zeigt über 70 Epochen eine insgesamt hohe Leistung, insbesondere in den Trainingsdaten. Der MCC von 0.7578 im Training und 0.7485 in der Validierung weisen auf eine höhere Korrelation zwischen den vorhergesagten und tatsächlichen Klassen hin. Die Standardabweichungen der Metriken sind generell gering, was für eine konsistente Klassifizierung über alle Folds hinweg hindeutet. Die Accuracy liegt sowohl in der Trainings- als auch in der Validierungsphase über 90%, was die Effektivität des Modells unterstreicht. Ähnlich verhält es sich mit der Spezifität, die sogar über 95% in der Trainingsphase erreicht. Dies deutet darauf hin, dass das Modell False-Positives effizient minimieren konnte. Aufgrund der starken Klassenimbalance ist dieses Verhalten allerdings zu erwarten. Werden komplexere Metriken wie der F1-Score oder der hier bevorzugte MCC betrachtet, so zeigt sich ein ausgewogeneres Bild. Die Metriken liegen nicht mehr teilweise im +90% Bereich. Durch die ganzheitliche Betrachtung aller Felder der CM durch den MCC Score fällt insbesondere die etwas schwächere Precision ins Gewicht und relativiert die ansonsten hohen anderen Metriken.

Die Trainingsmetriken sind sehr stabil und liegen auch unter den verschiedenen Folds sehr nah beieinander. Die Validierungsmetriken sind erwartungsgemäß geringfügig schlechter, aber auch die Standardabweichung ist merklich größer. Das könnte auf eine leichte Tendenz zum Overfitting hinweisen, die jedoch durch den eingesetzten Scheduler, die Batchnorm-Layer, einem Dropout-Layer und den Einsatz des FocalLoss-Kriteriums gemildert werden sollte. Allerdings sind trotzdem die vorliegenden Unterschiede, sowohl zwischen Training und Validierung, als auch zwischen den Folds, wie an der Standardabweichung zu erkennen, relativ gering. Jedoch ist zu beachten, dass die False Negatives in der Validierungsphase höher sind als die False Positives, was in klinischen Anwendungen problematisch sein könnte. Insgesamt zeigt das CNN-Modell eine vielversprechende Leistung, hat jedoch Raum für Verbesserungen, insbesondere in der Minimierung der False Negatives. Die Ergebnisse zeigen, dass das Modell eine zufriedenstellende Leistung bei der Klassifizierung der Abnormalität des Audiosignals erreicht. Obwohl die MCC-Werte auf dem Trainingsdatensatz höher sind als auf dem Validierungsdatensatz, ist die Differenz gering. Dies kann auf ein gut generalisierendes Modell hindeuten. Die Standardabweichung der Validierungsmetriken ist ebenfalls niedrig, was auf eine robuste Leistung des Modells hindeutet. Weitere Experimente könnten mit anderen Architekturen oder Hyperparametern durchgeführt werden, um die Leistung weiter zu optimieren.

Es ist möglich, dass das Modell sich in einem lokalen Minimum befindet und das Potential nicht ausschöpft. Die Metrikkurven zeigen dies durch einen sehr konstanten Verlauf, mit sehr geringen Änderungen ab Epoche 30 bis zum Trainingsende in Epoche 60, an. Die Lernrate ist durch den Scheduler stark verringert worden und möglicherweise ist die Anzahl der Epochen nicht ausreichend um die best möglichste Leistung zu erhalten.

YAMNet Modell

Die Evaluationsmetriken des YAMNet Modells weisen auf eine annehmbare Leistung hin, insbesondere in Bezug auf die Spezifität. Allerdings scheint es, dass das Modell mit Herausforderungen bei der Klassifizierung, insbesondere in Bezug auf Recall und Precision, konfrontiert ist. Der erreichte MCC-Wert von 0.5695 kann als *zufriedenstellend* bezeichnet werden. Dies legt nahe, dass das Modell eine akzeptable Balance zwischen Sensitivität und Spezifität erreicht hat, obwohl ein höherer MCC-Wert wünschenswert wäre. Die hohe Spezifität deutet darauf hin, dass das Modell sehr effektiv ist, wenn es darum geht, Negative Fälle korrekt zu identifizieren. Dies könnte jedoch zu Lasten des Recalls gehen, was in bestimmten Anwendungen problematisch sein könnte. Die korrekte Klassifizierung von negativen ist mit diesem Datensatz besonders einfach, da die Klassenbalance lediglich 24% positiv beträgt. Regularisierende Techniken gegen die starken Imbalance, wie sie im CNN Modell mit FocalLoss angewandt werden, sind in den Embedding Extraktor Methoden nicht zum Einsatz gekommen. Das könnte die schlechtere Performance erklären, die positiven Klasse zuverlässig zu erkennen.

VGGIsh Modell

Die Evaluationsmetriken des VGGish-Modells zeigen durchwegs sehr ähnliche Metriken wie das YAMNet-Modell. Der MCC-Wert von 0.5342 ist geringfügig schlechter als der des YAMNet-Modells und kann ebenfalls nur als zufriedenstellend interpretiert werden, da auch hier Verbesserungsmöglichkeiten bestehen. Das VGGish-Modell zeigt eine hohe Spezifität von 0.9213 und einen ausreichenden Recall von 0.7504, was darauf hindeutet, dass das Modell sowohl für die Identifikation von Negativen als auch für die von Positiven effizient ist. Beim letzteren allerdings Schwierigkeiten entstehen dies konsistent durchführen zu können.

5.1.2 Vergleich zwischen VGGish und YAMNet

Der Vergleich zwischen VGGish und YAMNet bietet interessante Einblicke, da beide Modelle auf die Extraktion von Audiomerkmalen spezialisiert sind, aber auf unterschiedlichen Architekturen basieren (siehe Abschnitt 2.2.4). Erneut zusammengefasst, VGGish ist eine Adaption der VGG-Architektur, die ursprünglich für visuelle Objekterkennung entwickelt wurde, und nutzt diese bewährte Struktur zur Analyse von Audiodaten. Im Gegensatz dazu basiert YAMNet auf der MobileNetV1-Architektur, die für mobile Anwendungen konzipiert ist und daher auf Effizienz und Geschwindigkeit optimiert wurde (Howard et al. , 2017). Diese architektonischen Unterschiede führen zu verschiedenen Leistungsmerkmalen in der Audioklassifikation. VGGish performt in einer etwas breiteren Untersuchung marginal besser als das YAMNet (Tsalera et al. , 2021). YAMNet, optimiert für Geschwindigkeit und Effizienz, ist hingegen besser für Anwendungen geeignet, in denen Ressourcenbeschränkungen eine Rolle spielen. Ebenso ist die Vorverarbeitung leicht unterschiedlich. Die Spektrogramme für YAMNet werden überlappend gebildet, während VGGish-Spektrogramme direkt nacheinander angeknüpft werden und somit keine Informationen zwischen den Patches bestehen bleibt. Die Leistungsunterschiede zwischen den beiden Modellen könnten auch durch die unterschiedlichen Trainingsdatensätze beeinflusst sein: VGGish wurde mit dem YouTube-8M-Datensatz und YAMNet mit dem AudioSet trainiert. Diese unterschiedlichen Trainingsbedingungen könnten zu Variationen in der Art und Weise führen, wie jedes Modell Audiomuster erkennt und klassifiziert. In dieser Anwendung hat sich gezeigt, dass das YAMNet geringfügig besser ist. Betrachtet man zudem die Geschwindigkeitsvorteile und den Speicherbedarf, resultierend aus der einfacheren Architektur, so fällt eine Entscheidung klar zu Gunsten des YAMNet Modell. Dies sollte auf den zuvor genannten Gründen aber nicht zwingend auf andere Fragestellungen und Implementierungen übertragen werden.

5.1.3 Vergleich zwischen CNN und Embedding-Methoden

CNN und Embedding-Methoden repräsentieren zwei grundlegend verschiedene Ansätze in der Verarbeitung und Klassifikation von Audiodaten. CNNs werden oft als end-to-end Modelle eingesetzt und integrieren sowohl die Merkmalsextraktion als auch die Klassifikation in einem einzigen Workflow (Pati et al. , 2023). Im Gegensatz dazu konzentrieren sich Embedding-Methoden, wie VGGish und YAMNet, primär auf die Merkmalsextraktion, wobei die Klassifikation durch nachfolgende Modelle, beispielsweise wie hier einem kNN, erfolgt (Tsalera et al. , 2021). Hierbei liegt der Vorteil in der Flexibilität andere Methoden auszuprobieren, ohne ein komplettes CNN neu zu trainieren. Embedding-Modelle seien hingegen, in der Theorie, für das Training effizienter und weniger rechenintensiv, da sie wie zuvor beschrieben speziell für die Extraktion von Merkmalen optimiert sind und bereits korrekt eingestellt sind. Optimierungsbedarf betrifft hierbei überwiegend nur die nachfolgenden Schritte, die vom Umfang her geringer ausfallen.

Tabelle 5.1: Zusammenfessung der gemittelten, geschätzten Modellgeschwindigkeiten und Größen

Modell	Chunks pro Sekunde	Modellgröße	Erreichter Validierungs MCC
CNN	66.11	33.2 Mb.	0.748
YAMNet	15.76	49 Mb.	0.5695
VGGish	10.56	279 Mb.	0.5342

In der Inferenzmessung hat sich herausgestellt, dass das CNN dennoch weit überlegen ist was die Geschwindigkeit angeht. Dies gepaart mit der besten Performance und dem kleinsten Modell, macht es zum bevorzugten Klassifikator für die meisten Fälle.

Zudem wird hier erneut der Umstand der einfachen Modellstruktur von YAMNet verdeutlicht im Vergleich zu VGGish. Relativ gleichwertige Ergebnisse, oft im Rahmen der Standardabweichung, allerdings ist das YAMNet markant schneller. Zudem sticht auch die Modellgröße des VGGish Extraktor negativ heraus.

Medizinische Anwendung

Es gibt Szenarien mit hohen Anforderungen an den Recall/Sensitivität oder Specificity. Insbesondere bei medizinischen Fragestellen die als Ziel die Detektion von Krankheiten aus einem Anfangsverdacht heraus haben, ist eine hohe Sensitivität wichtig (Weiß und Rzany , 2013, S. 266). Es ist wünschenswert möglichst wenig TP Fälle zu verpassen, wobei FP Fälle weniger relevant sind, da in so einem Kontext der Anwendung wie hier vorgestellt, sowieso nachfolgende Untersuchungen durch einen Arzt folgen würden. Die Metrik PLR beschreibt die Fähigkeit eben jene positiven Fälle auch korrekt zu identifizieren, ohne dass jemand gesundes als krank klassifiziert wird (Weiß und Rzany , 2013, S. 261). Der CNN Klassifikator hat eine PLR von 9.96 erreicht. YAMNet und VGGish liegen bei 9.585 und 9.744 respektive. Für einen leistungsfähigen Test sollte die PLR über 3 liegen und die NLR unter $1/3$ (Weiß und Rzany , 2013, S. 262). Alle Modelle erreichen diese Schwelle mit Abstand und daher könnten theoretisch diese auch als leistungsfähiger Test Nutzen in der medizinischen Anwendung finden.

5.1.4 Vergleich mit Stand der Forschung

In der Metastudie über Herztonklassifikatoren von Chen et al. (2021) werden verschiedene Verfahren miteinander verglichen und detaillierter erklärt. Eines der besten Verfahren welches dort beschrieben wird, basiert auf einem Recurrent Neural Network (RNN) von Latif et al. (2018) unter Verwendung von spezifischen Bidirectional Long Short-Term Memory (BLSTM) Zellen. Die folgende Tabelle zeigt einen Auszug von fünft Klassifikatoren dieser Metastudie basierend auf den verfügbaren Leistungsmetriken.

Tabelle 5.2: Top-5-Klassifikatoren aus der Metastudie (Chen et al. , 2021)

Referenz	Methode	Input Features	Recall, Specificity
Latif et al. (2018)	BLSTM	MFCC	0.9886, 0.9836
Chen et al. (2018)	2D-CNN	Wavelet + HHT	0.98, 0.885
Abduh et al. (2020)	2D-DNN	MFCC	0.893, 0.970
Dominguez-Morales et al. (2018)	2D-CNN	Spektrogramme	0.932, 0.951
Oh et al. (2020)	1D-CNN	Wavelets	0.925, 0.981

Die Tabelle 5.2 zeigt, dass die beste Leistung von dem RNN-Klassifikator von Latif et al. erzielt wurde, insbesondere mit der BLSTM-Architektur, die hervorragende Ergebnisse in allen vier Metriken erzielte. Verglichen mit unserem eigenen CNN-Modell, das in Tabelle 4.1 dargestellt ist, zeigen die Metriken von Latif et al. eine höhere Genauigkeit und Spezifität. Das CNN-Modell aus dieser Ausarbeitung zeigt jedoch eine Konkurrenzfähigkeit mit den Modellen, da die Werte für Specificity und Recall nicht sonderlich groß abweichen. Der Unterschied beträgt in allen Fällen weniger als 0.1 in den Metriken. Weitergehend zeigt sich, dass zum Zeitpunkt der Publikation, sich kein Eindeutig bestes Verfahren etabliert hat, da zumindest bei den 5 besten Anwendungen, drei verschiedene Herangehensweisen gewählt werden.

Diese Studie von Chen et al. untersucht keine Embedding Extraktoren und daher können diese aus dieser Ausarbeitung nicht eingeordnet werden. Dass die generelle Qualität, bis auf Ausnahmen, unter der des vorgestellten CNN liegt wurde bereits erklärt und spiegelt sich daher in einem potentiellen Vergleich ebenso wieder. Die Untersuchungen von Maity et al. (2023) nutzen YAMNet Klassifikatoren nach Transferlearning und Nachtrainieren. Sie erzielen dort außergewöhnlich gute Ergebnisse, allerdings lassen sich diese ebenfalls nicht auf diese Arbeit übertragen. Es wird ein anderer Datensatz verwendet, welcher speziell auf die Fragestellung zugeschnitten ist, verschiedene konkrete Herzkrankheiten zu unterscheiden. Deren Ausarbeitung ist aus diesem Jahr 2023 und somit aktuell. Da auch in der geringfügig jüngeren

Publikation von Chen et al. (2021) diese Verfahren keine Erwähnung finden, muss weitere Forschung abgewartet werden, bis es gut vergleichbare Metastudien gibt, welche auch Audio-Embedding Klassifikatoren für PCG Signale untersuchen.

5.2 Einschränkungen dieser Ausarbeitung

In dieser Ausarbeitung wurden wichtige Erkenntnisse über die Leistungsfähigkeit von CNN und Embedding-basierten Klassifikatoren gewonnen. Dennoch gibt es einige Einschränkungen, die bei der Interpretation der Ergebnisse berücksichtigt werden sollten. Zunächst ist die Trainingsdauer und -konfiguration zu erwähnen. Die Wahl der Hyperparameter und die Dauer des Trainingsprozesses haben einen erheblichen Einfluss auf die Leistung der Modelle (Alzubaidi et al. , 2021). In dieser Studie wurden die Modelle möglicherweise nicht bis zu ihrem vollen Potenzial trainiert, was die Ergebnisse beeinflussen könnte. Es besteht die Möglichkeit, dass die Modelle in einem lokalen Minimum verharren und dadurch nicht die optimale Leistung erbringen. Bestimmte Parameter wie *hop_length* oder *n_ftt* haben einen großen Einfluss und sind nicht zwingend optimal eingestellt worden. Weitergehend kann die Wahl der Features optimiert werden. Nach Chen et al. (2021) sei die diskrete Wavelet-Transformation den hier verwendeten STFT basierenden Mel-Spektrogrammen überlegen.

Ein weiterer Aspekt hinsichtlich Einschränkungen dieser Ergebnisse ist, dass es keine klinische Bewertung der Modelle gibt. Die Untersuchung basiert ausschließlich auf dem PhysioNet2016 Datensatz und verwendet Kreuzvalidierung zur Bewertung der Modelle. Dies kann zu einer gewissen Verzerrung führen, da die Modelle nicht an unabhängigen, externen klinischen Daten getestet wurden. Die Ergebnisse spiegeln daher möglicherweise nicht vollständig die tatsächliche Leistungsfähigkeit der Modelle in einem realen klinischen Umfeld wider.

Insgesamt bieten die Ergebnisse dieser Ausarbeitung wertvolle Einblicke, doch es ist wichtig, die genannten Einschränkungen bei der weiteren Forschung und Ent-

wicklung von Klassifikationsmodellen für medizinische Anwendungen zu berücksichtigen.

5.2.1 Bewertung des Datensatzes

Eine der Haupteinschränkungen dieser Arbeit könnte die Größe und Vielfältigkeit des verwendeten Datensatzes sein. Obwohl der Datensatz komplex aufgebaut ist, könnten bestimmte Biase vorliegen. (Dieser folgende Abschnitt ist aus der zuvor eingereichten Hausarbeit übernommen (Sondermann , 2023)).
Auch wenn der PhysioNet-Datensatz eine starke Erweiterung der bestehenden Datensätze bedeutet, zeigen sich dennoch Schwächen durch geringen Datenumfang. Insbesondere Deeplearning Modelle profitieren von einer ausgeglichenen und umfangreichen Datenlage (Chen et al. , 2021). Stetige Forschung ist notwendig, um den hohen Ansprüchen moderner Medizin gerecht zu werden und Patienten eine sichere Methodik zur Behandlung und Vorsorge zu ermöglichen. Eine kontinuierliche Weiterentwicklung der Modelle kann einen medizinischen Durchbruch ermögliche und die Mensch Medizin Interaktion vereinfachen Chen et al. (2021). Eine mögliche systematische Neuauflage, insbesondere mit neuen Audioaufnahmen von erkrankten Patienten, des Datensatzes würde die Datenqualität weiter steigern und folglich bessere Prädiktoren versprechen.

Kapitel 6

Zusammenfassung und Ausblick

In dieser Arbeit wird eine eingehende Untersuchung und ein Vergleich zwischen Embedding-basierten Klassifikatoren und CNN im Kontext der Erkennung von Herzklappenerkrankungen durchgeführt. Die Untersuchung konzentriert sich auf die Analyse und Bewertung der Effektivität beider Methoden unter Verwendung des umfassenden PhysioNet2016 Datensatzes. Die Forschungsergebnisse verdeutlichen, dass sowohl Embedding-basierte Klassifikatoren als auch CNNs jeweils einzigartige Stärken und Schwächen aufweisen, die sie für verschiedene Anwendungen im Bereich der klinischen Diagnostik geeignet machen. Während die CNN-Modelle insbesondere in der Genauigkeit und im MCC herausstechen, zeigen die Embedding-Modelle wie YAMNet und VGGish, auch wenn der Unterschied sehr gering ist, eine höhere Spezifität bei der Identifizierung negativer Fälle. Dies deutet darauf hin, dass eine Methode nicht eindeutig allen anderen überlegen ist, sondern dass die Auswahl des geeigneten Ansatzes von den spezifischen Anforderungen der klinischen Anwendung abhängen sollte. Zudem haben die Embedding Methoden mehr Potential in künftigen Arbeiten verbessert zu werden. Weitergehend besteht eine Abwägung hinschlich Implementierungsgeschwindigkeit und Aufwand. Das trainieren und optimieren des CNN ist aufwändiger als ein kNN-Klassifikator mit den Embeddings zu erstellen.

Das entscheidende Fazit dieser Arbeit ist, dass eine integrierte Herangehensweise, welche die Vorteile beider Technologien kombiniert, in vielen Fällen die effektivs-

te Lösung darstellen könnte. Eine solche Kombination könnte das Potenzial haben, die diagnostische Genauigkeit zu erhöhen und gleichzeitig die Limitationen jeder einzelnen Methode zu überwinden. Dieser Ansatz könnte besonders vorteilhaft sein, um die zeitlichen und räumlichen Muster in den Herzklappensignalen effizienter zu analysieren und zu interpretieren. So kann ein Embedding Extraktor diese Merkmale nicht nur direkt an einen Klassifikator weiterreichen, sondern an ein nachfolgendes mehrschichtiges CNN, welches speziell auf diese Merkmale weiter trainiert wird. Zusammen mit dem Nachtrainieren des Embedding Extraktors, konkret mit den vorliegenden Daten, würde dies die Qualität deutlich verbessern Tsalera et al. (2021). Dies, verbunden mit fortschrittlichen Feature Berechnungen wie Beispielsweise eine Wavelet-Transformation könnte zu verbessern Klassifizierungseigenschaften führen Chen et al. (2021).

6.1 Persönliche Erkenntnisse

Dieses Projekt ist zum aktuellen Zeitpunkt das persönlich größte und aufwändigste je durchgeführte Projekt. Im Zuge dessen kommen natürlich diverse Erkenntnisse zusammen, welche bei neuen Projekten hilfreich sein werden. Generelle Erfahrungen über Priorisierung, Fokus auf bestimmte Aspekte und Zeitrahmen sind aufgetreten. Weiterhin sind viele methodische Konzepte, insbesondere im Thema der Metrikberechnung und Datenaufbereitung, angeeignet worden. Im Zuge der Entwicklung war stets Wiederverwendbarkeit die Priorität, auch wenn dies mit einem erhöhtem zeitlichen Entwicklungsaufwand verbunden war und sich dieser nicht direkt in den Ergebnissen widerspiegelt. Dies ermöglicht in künftigen Projekten unabhängiger zu agieren und die Ergebnisse in der konkret gewünschten Form zu erhalten. Beispielsweise werden so die Vorteile des WANDB Service nachgeahmt aber die zuvor hier beschriebenen Nachteile nicht erfahren.

Während der Projektdauer sind zudem Probleme und Ineffizienzen aufgetreten. Diese reichen von kleinen Auswirkungen, wie die unerwartete Umständlichkeit des Latexmoduls für diese Ausarbeitung *acronym* bis hin zu komplexeren Problemen. Das

Projekt war zu beginn teils in Abstimmung mit einem anderen Projekt entwickelt, welche sich gegenseitig blockiert haben. Künftig würden solche großen Projekte klarer definiert und von nicht unabdingbaren abgegrenzt werden, mit klaren messbaren Meilensteinen. Die Erwartungen sind konstant gewachsen was dieses Projekt in die Länge zieht und den Umfang über das vorher geplante steigen lässt. Solche erkannten Probleme gehören zur persönlichen Erfahrung und helfen in künftigen Projekten effizienter zu arbeiten.

Anhang A

Diagramme und Tabellen

CNN Training	**Epoche 70**								
Fold	Accuracy	F1	Precision	Recall	Specificity	MCC	PLR	NLR	Auroc
0	0.919	0.832	0.828	0.837	0.945	0.779	15.196	0.173	0.9726
1	0.91	0.811	0.805	0.817	0.938	0.752	13.285	0.195	0.9666
2	0.913	0.817	0.808	0.826	0.94	0.76	13.735	0.185	0.9702
3	0.909	0.802	0.806	0.798	0.942	0.743	13.757	0.214	0.9648
4	0.906	0.802	0.799	0.805	0.937	0.74	12.773	0.208	0.9664
5	0.916	0.821	0.82	0.822	0.945	0.766	14.839	0.188	0.9711
6	0.92	0.833	0.829	0.837	0.946	0.781	15.566	0.172	0.974
7	0.905	0.799	0.794	0.805	0.936	0.737	12.509	0.209	0.9621
8	0.918	0.827	0.829	0.826	0.946	0.773	15.408	0.184	0.973
9	0.908	0.807	0.807	0.807	0.94	0.747	13.426	0.205	0.9676
Average	0.9124	0.8151	0.8125	0.818	0.9415	0.7578	14.0494	0.1933	0.96884
STD	0.005561774	0.012784105	0.012972192	0.013848385	0.003836955	0.016362559	1.117702922	0.015202704	0.003933955
CNN Validation	Epoche 70								
Fold	Accuracy	F1	Precision	Recall	Specificity	MCC	PLR	NLR	Auroc
0	0.89	0.767	0.681	0.877	0.893	0.706	8.21	0.137	0.9594
1	0.896	0.787	0.725	0.861	0.905	0.723	9.11	0.153	0.96
2	0.905	0.827	0.759	0.907	0.904	0.768	9.49	0.103	0.971
3	0.911	0.844	0.818	0.872	0.926	0.783	11.794	0.138	0.966
4	0.91	0.815	0.76	0.879	0.919	0.76	10.873	0.132	0.9706
5	0.917	0.848	0.778	0.933	0.911	0.798	10.5	0.073	0.971
6	0.909	0.814	0.789	0.84	0.93	0.754	12.028	0.172	0.9662
7	0.921	0.853	0.777	0.946	0.913	0.807	10.913	0.059	0.9793
8	0.867	0.716	0.657	0.786	0.888	0.634	7.048	0.24	0.9449
9	0.906	0.806	0.728	0.903	0.906	0.752	9.633	0.107	0.959
Average	0.9032	0.8077	0.7472	0.8804	0.9095	0.7485	9.9599	0.1314	0.96474
STD	0.015619077	0.042105819	0.049754508	0.046299028	0.013310397	0.050754857	1.572900043	0.051681073	0.009463286
VGGish Validation									
Fold	Accuracy	F1	Precision	Recall	Specificity	MCC	PLR	NLR	Auroc
0	0.827	0.731	0.735	0.727	0.897	0.462	7.069	0.305	0.8773
1	0.848	0.778	0.788	0.771	0.914	0.558	8.946	0.251	0.8809
2	0.84	0.776	0.794	0.762	0.919	0.555	9.379	0.259	0.8747
3	0.841	0.788	0.814	0.771	0.928	0.583	10.647	0.247	0.8955
4	0.833	0.747	0.768	0.733	0.918	0.499	8.937	0.291	0.8749
5	0.834	0.761	0.791	0.743	0.927	0.532	10.164	0.277	0.8912
6	0.855	0.78	0.812	0.759	0.939	0.569	12.436	0.256	0.8839
7	0.845	0.776	0.802	0.759	0.929	0.559	10.665	0.26	0.8722
8	0.855	0.77	0.794	0.752	0.933	0.545	11.198	0.265	0.8943
9	0.828	0.738	0.753	0.727	0.909	0.48	7.998	0.3	0.8738
Average	0.8406	0.7645	0.7851	0.7504	0.9213	0.5342	9.7439	0.2711	0.88187
STD	0.010189319	0.019472202	0.025557995	0.01695222	0.012463725	0.040438705	1.592567769	0.020909594	0.008893824
YAMNet Validation									
Fold	Accuracy	F1	Precision	Recall	Specificity	MCC	PLR	NLR	Auroc
0	0.845	0.767	0.763	0.771	0.897	0.534	7.496	0.256	0.8696
1	0.869	0.808	0.82	0.797	0.93	0.617	11.385	0.218	0.9064
2	0.86	0.803	0.823	0.788	0.932	0.611	11.565	0.227	0.8921
3	0.832	0.778	0.798	0.765	0.915	0.562	9.002	0.257	0.8749
4	0.841	0.762	0.779	0.75	0.918	0.528	9.145	0.273	0.8485
5	0.825	0.748	0.777	0.73	0.921	0.505	9.286	0.293	0.8492
6	0.874	0.822	0.829	0.815	0.927	0.644	11.125	0.199	0.9169
7	0.855	0.794	0.814	0.779	0.929	0.592	10.948	0.238	0.8728
8	0.854	0.782	0.785	0.78	0.91	0.564	8.703	0.242	0.8755
9	0.839	0.769	0.767	0.771	0.893	0.538	7.193	0.256	0.8629
Average	0.8494	0.7833	0.7955	0.7746	0.9172	0.5695	9.5848	0.2459	0.87688
STD	0.015812794	0.023118295	0.024541348	0.023744941	0.013677232	0.045038872	1.594301226	0.027270049	0.0224777633

Tabelle A.1: Metriken und STD pro Fold

Anhang B

Quellcode

Der komplette Projekt wird zusammen mit dem erarbeiteten Quellcode, den Daten und erstellten Modellen unter https://fh-dortmund.sciebo.de/s/gdfD6ffw6rNnVrU und https://github.com/MarSond/PhysionetClassify (beides Stand 19.12.2023) zur Verfügung gestellt.

```
# Parameters for audio augmentation and processing specific tasks
audio_human = {
  "samplerate": 2000,    # Expected File samplerate
  "n_fft": 1024,       # FFT window size
  "hop_length": 128,     # Hop length for FFT
  "n_mels": 512,       # Number of Mel bands to generate
  "top_db": 80,      # Top db for silence trimming
  "butterpass_low": 25, # Highpass frequenzy
  "butterpass_high": 400, # Lowpass frequenzy
}

vggish_params = {
  "batchsize": 10,       # Batchsize for the VGGish extractor
   training
  "learnrate": 0.0002,     # initial Learning rate
  "seconds": 7,        # Seconds of audio to use per chunk
  "drop0": 0.2,        # Dropout rate position 0
  "drop1": 0.5,        # Dropout rate position 1
  "freeze_extractor": False,  # Freeze VGGish layers
  # the previous parameters are for a potential fine tuning of the
    extractor
  "embedding_mode": "mean",    # Embedding mode for training data -
    feeding the kNN
  "embedding_mode_valid": "plain",# Embedding mode for validation
   data
  "plot_embeddings": False, # Plot embeddings using t-SNE and UMAP
  "n_neighbors": 5,      # Number of neighbors for KNN - list [ ]
   needed
  "assume_positive_p": 0.3, # Percentage from where on in MIL/
   Embedding merge to assume positive
```

```
  "knn_distance": "euclidean",# Distance metric for KNN
  "model_type": "embeddings", # Nearest Neighbor model type or CNN
  "model_path": VGGISH_EXTRACTOR, # Path to pretrained model
  "filter_anomaly": False,    # Filter anomaly class from training data using Isolation Forest
}

yamnet_params = {
  "batchsize": 10,      # Batchsize for the YAMnet extractor training
  "learnrate": 0.0002,    # initial Learning rate
  "seconds": 7,       # Seconds of audio to use per chunk
  "drop0": 0.5,       # Dropout rate position 0
  "drop1": 0.5,       # Dropout rate position 1
  "freeze_extractor": False,  # Freeze YAMNet layers
  # the previous parameters are for a potential fine tuning of the extractor
  "embedding_mode": "mean",   # Embedding mode for training data - feeding the kNN
  "embedding_mode_valid": "plain",# Embedding mode for validation data
  "plot_embeddings": False,   # Plot embeddings using t-SNE and UMAP
  "n_neighbors": 3,       # Number of neighbors for KNN - list [ ] needed
  "assume_positive_p": 0.5,   # Percentage from where on in MIL/ Embedding merge to assume positive
  "knn_distance": "euclidean",  # Distance metric for KNN
  "model_type": "embeddings",   # Nearest Neighbor model type or CNN
  "model_path": YAMNET_EXTRACTOR, # Path to pretrained model
  "filter_anomaly": False,    # Filter anomaly class from training data using Isolation Forest
}

# CNN Mode Parameters
cnn_params = {
  "method": "cnn",      # Method to use (cnn, yamnet, vggish)
  "optimizer": "sgd",     # Optimizer to use (adam, sgd)
  "use_sheduler": True,   # Use scheduler for learning rate
  "scheduler": "plateau",   # Scheduler type (plateau, step, cosine)
  "load_checkpoint": "",    # Path to checkpoint file to load, empty if none
  "learnrate": 0.01,      # initial Learning rate
  "target_samplerate": 2000,  # Target samplerate for cnn audio files
  "seconds": 7,       # Seconds of audio to use per chunk
  "batchsize": 90,      # Batchsize for the CNN training
  "drop0": 0.5,       # Dropout rate position 1
  "drop1": 0.2,       # Dropout rate position 2
  "model_type": 1,      # Model selection
  "enable_earlystop": False,  # Enable early stopping with rolling F1 (rudymentary)
  "earlystop_threshold": 0.15,# Early stopping threshold after 5 epochs
```

```
}

# General Parameters
train = {
  "dataset": "standard",      # Dataset selection (which datalist to use)
  "epochs": 200,          # Number of epochs
  "kfold": 0,           # Number of kfold splits
  "augmentation_rate": 0.65,    # Percentage of data to augment
  "l1_weight": 0,         # Weight for regulation loss L1
  "l2_weight": 0,         # Weight for regulation loss L2
  "head_pct": 1.0,        # Percentage of datalist to use
  "activation": "silu",     # Activation function]
  "train_split": 0.80,      # Percentage of data to use for training
  "num_workers": 0,       # Number of workers for dataloader
  "save_model": True,       # Save model after each epoch
  "tag": "",            # A name/tag/label for a experiment
  "criterion": "FocalLoss",   # Loss function [(weighted) CrossEntropyLoss, FocalLoss]
  "focalloss_gamma": 2,     # Gamma for focal loss. Unused in other loss functions
  "num_classes": len(CLASSES_1),  # Number of distinct classes
  "label_name": CLASS_LABEL_1,  # Name of the label column in the datalist
}
```

Quellcode B.0.1: Standardwerte der Parameter dict Objekte

```
import torch.nn as nn
import parameters as cfg
import logging

def get_model(run_config):
  type = run_config['model_type']
  if type == 1: # Als finale Model verwendet
    return CNN_Model_1(run_config)
  elif type == 2:
    return CNN_Model_2(run_config)
  elif type == 3:
    return CNN_Model_3(run_config)
  else:
    raise ValueError(f"Model type {type} not found")

class CNN_Base(nn.Module):
  def __init__(self, run_config):
    super().__init__()
    self.num_classes = run_config['num_classes']
    self.target_samplerate = run_config['target_samplerate']
    self.pDrop0 = run_config['drop0']
    self.pDrop1 = run_config['drop1']
    self.seconds = run_config['seconds']
    self.n_mels = run_config['n_mels']
    if run_config['activation'] == "relu":
      self.activation = nn.ReLU(inplace=True)
    elif run_config['activation'] == "l_relu":
      self.activation = nn.LeakyReLU(inplace=True)
```

```
29      elif run_config['activation'] == "silu":
30        self.activation = nn.SiLU(inplace=True)
31      else: raise ValueError(f"Activation {run_config['activation']} not found in YAMNET Model list")
32      self.softmax = nn.Softmax(dim=1)
33      self.tensor_logger = logging.getLogger(cfg.TENSOR_LOGGER)
34      self._initialize_weights()
35
36
37    def forward(self, x):
38      self.tensor_logger.debug(f"Raw Forward Input shape: {x.shape}")
39      self.batchsize = x.shape[0]
40      if len(x.shape) == 3: # missing channel dimension
41        x = x.unsqueeze(1)
42      self.tensor_logger.info(f"Modified Forward INIT Input shape: {x.shape}")
43      return x
44
45    def _initialize_weights(self):
46      for m in self.modules():
47        if isinstance(m, nn.Conv2d):
48          nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='silu')
49          if m.bias is not None:
50            nn.init.constant_(m.bias, 0)
51        elif isinstance(m, nn.BatchNorm2d):
52          nn.init.constant_(m.weight, 1)
53          nn.init.constant_(m.bias, 0)
54        elif isinstance(m, nn.Linear):
55          nn.init.normal_(m.weight, 0, 0.01)
56          nn.init.constant_(m.bias, 0)
57
58  class CNN_Model_1(CNN_Base):
59    def __init__(self, run_config):
60      super().__init__(run_config)
61      conv_layers = []
62
63      # First Convolutional Block
64      self.conv1 = nn.Conv2d(in_channels=1, out_channels=8, kernel_size=(3, 3))
65      self.bn1 = nn.BatchNorm2d(8)
66      self.maxpool2d1 = nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2))
67      conv_layers += [self.conv1, self.bn1, self.activation, self.maxpool2d1]
68
69      # Second Convolutional Block
70      self.conv2 = nn.Conv2d(in_channels=8, out_channels=16, kernel_size=(3, 3))
71      self.bn2 = nn.BatchNorm2d(16)
72      self.maxpool2d2 = nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2))
73      conv_layers += [self.conv2, self.bn2, self.activation, self.maxpool2d2]
74
```

```
75      # Third Convolutional Block
76      self.conv3 = nn.Conv2d(in_channels=16, out_channels=32,
        kernel_size=(3, 3))
77      self.bn3 = nn.BatchNorm2d(32)
78      self.maxpool2d3 = nn.MaxPool2d(kernel_size=(2, 2), stride=(2,
        2))
79      conv_layers += [self.conv3, self.bn3, self.activation, self.
        maxpool2d3]
80
81      # Fourth Convolutional Block
82      self.conv4 = nn.Conv2d(in_channels=32, out_channels=64,
        kernel_size=(3, 3))
83      self.bn4 = nn.BatchNorm2d(64)
84      self.maxpool2d4 = nn.MaxPool2d(kernel_size=(2, 2), stride=(2,
        2))
85      conv_layers += [self.conv4, self.bn4, self.activation, self.
        maxpool2d4]
86
87      self.ap = nn.AdaptiveAvgPool2d(output_size=(8,8))
88
89      # Flatten the output of the convolutional layers
90      self.flatten = nn.Flatten()
91
92      # First Fully-Connected (Linear) Layer
93      self.fc1 = nn.Linear(in_features=4096, out_features=1024)
94      self.bn5 = nn.BatchNorm1d(1024)
95      fc1_layers = [self.fc1, self.bn5, self.activation]
96      if self.pDrop0 > 0.0:
97        fc1_layers += [nn.Dropout(p=self.pDrop0)]
98
99      self.fc2 = nn.Linear(in_features=1024, out_features=128)
100     self.bn6 = nn.BatchNorm1d(128)
101     fc2_layers = [self.fc2, self.bn6, self.activation]
102     if self.pDrop1 > 0.0:
103       fc2_layers += [nn.Dropout(p=self.pDrop1)]
104
105     # Third Fully-Connected (Linear) Layer - Last layer with 2
        outputs
106     self.fc3 = nn.Linear(in_features=128, out_features=self.
        num_classes)
107
108     self.conv = nn.Sequential(*conv_layers)
109     self.fc1_block = nn.Sequential(*fc1_layers)
110     self.fc2_block = nn.Sequential(*fc2_layers)
111
112   def forward(self, x):
113     x = super().forward(x)
114     x = self.conv(x)
115     x = self.ap(x)
116     x = self.flatten(x)
117     x = self.fc1_block(x)
118     x = self.fc2_block(x)
119     x = self.fc3(x)
120     return x
121
122 class CNN_Model_2(CNN_Base):
```

```
123   def __init__(self, run_config):
124     super().__init__(run_config)
125
126     self.drop0 = self.pDrop0
127     self.drop1 = self.pDrop1
128
129     # First Convolutional Block
130     self.conv1 = nn.Conv2d(in_channels=1, out_channels=8,
      kernel_size=(3, 3))
131     self.bn1 = nn.BatchNorm2d(8)
132     self.maxpool1 = nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2)
      )
133
134     # Second Convolutional Block
135     self.conv2 = nn.Conv2d(in_channels=8, out_channels=16,
      kernel_size=(3, 3))
136     self.bn2 = nn.BatchNorm2d(16)
137     self.maxpool2 = nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2)
      )
138
139     # Third Convolutional Block
140     self.conv3 = nn.Conv2d(in_channels=16, out_channels=32,
      kernel_size=(3, 3))
141     self.bn3 = nn.BatchNorm2d(32)
142     self.maxpool3 = nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2)
      )
143
144     # Adaptive Pooling
145     self.ap = nn.AdaptiveAvgPool2d(output_size=(16, 16))  #
      Adjusted size
146
147     # Flatten Layer
148     self.flatten = nn.Flatten()
149
150     # First Fully-Connected Layer
151     self.fc1 = nn.Linear(in_features=8192, out_features=1024)  #
      Adjusted input size
152     self.bn4 = nn.BatchNorm1d(1024)
153     self.fc1_block = nn.Sequential(self.fc1, self.bn4, self.
      activation)
154     if self.pDrop0 > 0:
155       self.fc1_block.add_module('dropout0', nn.Dropout(p=self.
      pDrop0))
156
157     # Second Fully-Connected Layer
158     self.fc2 = nn.Linear(in_features=1024, out_features=128)
159     self.bn5 = nn.BatchNorm1d(128)
160     self.fc2_block = nn.Sequential(self.fc2, self.bn5, self.
      activation)
161     if self.pDrop1 > 0:
162       self.fc2_block.add_module('dropout1', nn.Dropout(p=self.
      pDrop1))
163
164     # Output Layer
165     self.fc3 = nn.Linear(in_features=128, out_features=self.
      num_classes)
```

```
166
167   def forward(self, x):
168     x = super().forward(x)
169     x = self.conv1(x)
170     x = self.bn1(x)
171     x = self.activation(x)
172     x = self.maxpool1(x)
173
174     x = self.conv2(x)
175     x = self.bn2(x)
176     x = self.activation(x)
177     x = self.maxpool2(x)
178
179     x = self.conv3(x)
180     x = self.bn3(x)
181     x = self.activation(x)
182     x = self.maxpool3(x)
183
184     x = self.ap(x)
185
186     x = self.flatten(x)
187
188     x = self.fc1_block(x)
189     x = self.fc2_block(x)
190
191     x = self.fc3(x)
192
193     return x
194
195 class CNN_Model_3(CNN_Base):
196   def __init__(self, run_config):
197     super().__init__(run_config)
198     conv_layers = []
199
200     # First Convolutional Block
201     self.conv1 = nn.Conv2d(in_channels=1, out_channels=8,
      kernel_size=(3, 3))
202     self.bn1 = nn.BatchNorm2d(8)
203     self.maxpool2d1 = nn.MaxPool2d(kernel_size=(2, 2), stride=(2,
      2))
204     conv_layers += [self.conv1, self.bn1, self.activation, self.
      maxpool2d1]
205
206     # Second Convolutional Block
207     self.conv2 = nn.Conv2d(in_channels=8, out_channels=16,
      kernel_size=(3, 3))
208     self.bn2 = nn.BatchNorm2d(16)
209     self.maxpool2d2 = nn.MaxPool2d(kernel_size=(2, 2), stride=(2,
      2))
210     conv_layers += [self.conv2, self.bn2, self.activation, self.
      maxpool2d2]
211
212     # Third Convolutional Block
213     self.conv3 = nn.Conv2d(in_channels=16, out_channels=32,
      kernel_size=(3, 3))
214     self.bn3 = nn.BatchNorm2d(32)
```

```
215     self.maxpool2d3 = nn.MaxPool2d(kernel_size=(2, 2), stride=(2,
        2))
216     conv_layers += [self.conv3, self.bn3, self.activation, self.
        maxpool2d3]
217
218     # Fourth Convolutional Block
219     self.conv4 = nn.Conv2d(in_channels=32, out_channels=64,
        kernel_size=(3, 3))
220     self.bn4 = nn.BatchNorm2d(64)
221     self.maxpool2d4 = nn.MaxPool2d(kernel_size=(2, 2), stride=(2,
        2))
222     conv_layers += [self.conv4, self.bn4, self.activation, self.
        maxpool2d4]
223
224     self.ap = nn.AdaptiveAvgPool2d(output_size=(8,8))
225
226     # Flatten the output of the convolutional layers
227     self.flatten = nn.Flatten()
228
229     # First Fully-Connected (Linear) Layer
230     self.fc1 = nn.Linear(in_features=4096, out_features=128)
231     self.bn5 = nn.BatchNorm1d(128)
232     fc1_layers = [self.fc1, self.bn5, self.activation]
233     if self.pDrop0 > 0.0:
234       fc1_layers += [nn.Dropout(p=self.pDrop0)]
235
236     # Second Fully-Connected (Linear) Layer - Last layer with 2
        outputs
237     self.fc3 = nn.Linear(in_features=128, out_features=self.
        num_classes)
238
239     self.conv = nn.Sequential(*conv_layers)
240     self.fc1_block = nn.Sequential(*fc1_layers)
241
242   def forward(self, x):
243     x = super().forward(x)
244     x = self.conv(x)
245     x = self.ap(x)
246     x = self.flatten(x)
247     x = self.fc1_block(x)
248     x = self.fc3(x)
249     return x
```

Quellcode B.0.2: Modelklasse für das CNN Model

```
1  import torch
2  import torch.nn
3  import logging
4  import parameters as cfg
5  import numpy as np
6  from sklearn.neighbors import KNeighborsClassifier
7  import torch.nn as nn
8
9  # This model gets loaded with embeddings tensors, together with
       their labels. It then uses sklearn KNeighborsClassifier to
       perform classification on th CPU
10 class Model_NearestNeighbor_classifier(nn.Module):
```

```
11
12    def __init__(self, device):
13      super().__init__()
14      self.tensor_logger = logging.getLogger(cfg.TENSOR_LOGGER)
15      self.extractor = nn.Module()
16      self.device = device
17      self.neighbor_data = torch.Tensor()
18      self.neighbor_labels = torch.Tensor()
19      self.classifier = None
20      self.n_neighbors = 3
21      self.knn_distance = 'euclidean'
22      self.assume_positive_p = 0.3
23      self.embedding_mode = 'mean'
24
25
26    def add_neighbor_data(self, data, labels):
27      self.neighbor_data = torch.cat([self.neighbor_data, data], dim
        =0)
28      self.neighbor_labels = torch.cat([self.neighbor_labels, labels
        ], dim=0)
29
30    def set_extractor(self, extractor: nn.Module):
31      self.extractor = extractor
32
33    def load_state_dict(self, state_dict):
34      self.extractor = state_dict['extractor']
35      super().load_state_dict(state_dict['model_state_dict'])
36      self.build_nn_classifier(n_neighbors=state_dict['n_neighbors'
        ], distance_metric=state_dict['distance_metric'], \
37          embedding_mode=state_dict['embedding_mode'],
        assume_positive_p=state_dict['assume_positive_p'])  # Rebuild
        the nearest neighbor classifier
38
39    def build_nn_classifier(self, n_neighbors=3, distance_metric='
        euclidean', embedding_mode='mean', assume_positive_p=0.3):
40      self.classifier = KNeighborsClassifier(n_neighbors=n_neighbors
        , weights='distance', metric=distance_metric)
41      self.n_neighbors = n_neighbors
42      self.knn_distance = distance_metric
43      self.embedding_mode = embedding_mode
44      self.knn_distance = distance_metric
45      self.assume_positive_p = assume_positive_p
46      neighbor_data_cpu = self.neighbor_data.cpu().detach().numpy()
47      neighbor_labels_cpu = self.neighbor_labels.cpu().detach().
        numpy()
48      self.tensor_logger.warning(f"Building nearest neighbor
        classifier with {n_neighbors} neighbors, distance metric {
        distance_metric} and embedding mode {embedding_mode} and
        assume_positive_p {assume_positive_p}")
49      self.classifier.fit(neighbor_data_cpu, neighbor_labels_cpu)
50
51    @staticmethod
52    def combine_embeddings(embeddings, embedding_mode):
53      if embedding_mode == "mean":
54        embeddings = torch.mean(embeddings, dim=1)
55      elif embedding_mode == "tripplets_mean": # mean the 3
```

```
    successive embeddings
56      if embeddings.shape[1] % 3 != 0:
57        remainder = embeddings.shape[1] % 3
58        # Mitteln der übrigen Embeddings
59        last_embeddings = torch.mean(embeddings[:, -remainder:],
    dim=1, keepdim=True)
60        # Entfernen der übrigen Embeddings vor dem Reshape
61        embeddings = embeddings[:, :-remainder]
62        # Reshape und Mittelung der 3er-Blöcke
63        embeddings = embeddings.reshape(embeddings.shape[0], -1,
    3, embeddings.size()[2])
64        embeddings = torch.mean(embeddings, dim=2)
65        embeddings = torch.cat([embeddings, last_embeddings], dim
    =1)
66      else:
67        embeddings = embeddings.reshape(embeddings.shape[0], -1,
    3, embeddings.size()[2])
68        embeddings = torch.mean(embeddings, dim=2)
69    elif embedding_mode == "max":
70      embeddings = torch.max(embeddings, dim=1)[0]
71    elif embedding_mode == "sum":
72      embeddings = torch.sum(embeddings, dim=1)
73    elif embedding_mode == "plain":
74      pass
75    if len(embeddings.shape) == 2:
76      embeddings = embeddings[:, None, :]
77    return embeddings
78
79  '''
80  First, check if the classifier is available. If not, raise an
     exception.
81  Then extract all embeddings for a new sample.
82  The embeddings get processed. Either mean, max, summed or
     concatenated into a long embedding.
83  self.classifier is the pretrained sklearn KNeighborsClassifier.
84  Output is the output of the classifier.
85  If the output are multiple embeddings, then a assume_positive_p
     is the percentage of embeddings required is checked.
86  '''
87  def forward(self, x):
88    if self.classifier is None or len(self.neighbor_data) == 0:
89      raise Exception("No nearest neighbor classifier found.
    Please add neighbor data first.")
90    if self.extractor is None:
91      raise Exception("No embedding extractor found. Please add
    extractor first.")
92
93    self.tensor_logger.debug(f"Raw Input shape: {x.shape}")
94    embeddings = self.extractor.forward(x)
95    self.tensor_logger.info(f"Embeddings shape: {embeddings.shape}
    ")
96
97    embeddings = Model_NearestNeighbor_classifier.
    combine_embeddings(embeddings, self.embedding_mode)
98    # Fügen Sie eine zusätzliche Dimension hinzu
99    batch_size, num_embeddings, embedding_size = embeddings.shape
```

```
100     embeddings = embeddings.reshape(batch_size*num_embeddings,
    embedding_size)
101     embeddings = embeddings.cpu().detach().numpy()
102 
103     # Klassifikation und Wahrscheinlichkeiten
104     preds = self.classifier.predict(embeddings)
105     preds_proba = self.classifier.predict_proba(embeddings)
106     preds_reshape = preds.reshape(batch_size, num_embeddings)
107     preds_proba_reshape = preds_proba.reshape(batch_size,
    num_embeddings, -1)  # Anzahl der Klassen
108 
109     # Schwellenwert-Logik
110     if self.embedding_mode in ["plain", "tripplets_mean"]:
111       positive_counts = np.sum(preds_reshape, axis=1)
112       threshold = num_embeddings * self.assume_positive_p
113       instance_preds = np.where(positive_counts >= threshold, 1,
    0)
114     else:
115       instance_preds = np.argmax(np.mean(preds_proba_reshape, axis
    =1), axis=-1)  # Multiclass
116 
117     # Wahrscheinlichkeiten für ROC
118     instance_proba = np.mean(preds_proba_reshape, axis=1)
119 
120     return instance_preds, instance_proba
```

Quellcode B.0.3: Modelklasse für das KNN Model in PyTorch eingebettet

```
1  import torch.nn as nn
2  from ml_helper import metrics, utils, ml_base
3  from ml_helper.utils import MLUtil, MLPbar
4  import parameters as cfg
5  import torch.optim as optim
6  from sklearn.model_selection import StratifiedGroupKFold
7  from torch.cuda.amp.grad_scaler import GradScaler
8  from torch.cuda.amp import autocast
9  import torch, wandb, logging
10 from os import path
11 import numpy as np
12 from ml_helper.utils import MLLogging
13 import warnings
14 
15 class CNN_base(ml_base.MLBase):
16 
17   def __init__(self, base_config, device, datalist, pbars: MLPbar
     = None):
18     self.base_config = base_config
19     self.datalist = datalist
20     self.metrics = metrics.MetricsTracker(config=base_config,
    device=device, metrics_class=metrics.TorchMetricsAdapter)
21     self.batchsize = base_config['batchsize']
22     self.device = device
23     self.pbars = pbars
24     self.current_epoch = 0
25     self.current_kfold = 0
26     self.logger, self.tensor_logger = MLLogging.getLogger([cfg.
    MAIN_LOGGER, cfg.TENSOR_LOGGER])
```

```
27      self.first_wandb_run = True
28      self.valid_f1_history = []    # used for early stopping
29      self.do_stop_early_now = False  # if set to true, this epoch
      will be the last one
30
31    # prepare the model, optimizer, scheduler and loss function
32    def prepare_cnn(self):
33      model = self.get_model()
34      self.model= model.to(self.device)
35      if self.logger.isEnabledFor(logging.DEBUG):
36        self.logger.debug(MLUtil.get_model_table(self.model))
37        #MLUtil.(self.model, (self.base_config['batchsize'], 1, 72,
      157))
38      if self.run_config['optimizer'] == 'adam':
39        self.optimizer = optim.Adam(filter(lambda p: p.requires_grad
      , self.model.parameters()), \
40                    lr = self.base_config['learnrate'], weight_decay=
      self.base_config['l2_weight'])
41      elif self.run_config['optimizer'] == 'sgd':
42        self.optimizer = optim.SGD(filter(lambda p: p.requires_grad,
       self.model.parameters()), \
43                      lr = self.base_config['learnrate'], momentum
      =0.9, weight_decay=self.base_config['l2_weight'])
44      self.scaler = GradScaler()
45      if self.run_config['scheduler'] == 'plateau' :
46        self.scheduler = optim.lr_scheduler.ReduceLROnPlateau(self.
      optimizer, mode='min', factor=0.2, patience=10, verbose=True)
47      elif self.run_config['scheduler'] == 'step':
48        self.scheduler = optim.lr_scheduler.StepLR(self.optimizer,
      step_size=10, gamma=0.2)
49      elif self.run_config['scheduler'] == 'cosine':
50        self.scheduler = optim.lr_scheduler.CosineAnnealingLR(self.
      optimizer, T_max=40, eta_min=0)
51
52    # complete training loop
53    def train_loop(self):
54      self.num_splits = self.base_config['kfold'] if self.
      base_config['kfold'] > 1 else 1
55      self.pbars.update_fold_parameter(self.num_splits)
56      self.pbars.reset_fold()
57      if self.num_splits > 1:
58        data_kfold_object = StratifiedGroupKFold(n_splits=self.
      num_splits, shuffle=True, random_state=cfg.SEED)
59        current_fold_number = 0
60
61        label_list = self.datalist.get(self.base_config['label_name'
      ])
62        name_list = self.datalist.get('name') # unique identifier
      for each file and thus hopefully patient
63        for train_index, val_index in data_kfold_object.split(self.
      datalist, label_list, name_list):
64          current_fold_number += 1 # fold number starts with 1
65          self.perform_fold(train_index, val_index,
      current_fold_number)
66      else:
67        self.perform_fold()
```

```
68      if self.logger.isEnabledFor(logging.INFO):
69        self.metrics.plot_metrics()
70      self._end_train_loop()
71
72    # one single fold
73    def perform_fold(self, train_index=None, val_index=None,
      current_split=0):
74      with warnings.catch_warnings():
75        self.pbars.update(1)
76        warnings.filterwarnings("ignore", category=UserWarning,
      module='wandb.sdk')
77        current_wandb_run = self.get_current_wandb_run(current_split
      )
78        self.prepare_cnn()
79        current_wandb_run.watch(self.model)
80        trainloader, validationloader = self.get_dataloader(
      train_index, val_index)
81        current_wandb_run.log({"class_weights": utils.MLUtil.
      convert_classweights(self.class_weights), "kfold":
      current_split})
82        self.logger.debug(f"len trainloader {len(trainloader)}")
83        self.logger.debug(f"len validationloader {len(
      validationloader)}")
84        self.pbars.reset_epoch()
85        self.pbars.update_epoch_parameter(self.run_config['epochs'])
86
87        for epoch in range(self.run_config['epochs']):
88          self.current_epoch = epoch+1
89          self.logger.info("-"*25)
90
91          self.logger.info(f"Start Training Epoch {self.
      current_epoch}")
92          self.train_epoch(trainloader, current_wandb_run)
93
94          self.logger.info(f"Start Validation Epoch {self.
      current_epoch}")
95          self.validation_epoch(validationloader, current_wandb_run)
96
97          self.pbars.reset_batch()
98          self.pbars.reset_valid_batch()
99          self.pbars.update(2, postfix=self.metrics.
      get_last_validation_postfix())
100         if self.run_config['save_model']:
101           self.save_model(model=self.model, folder=self.
      job_base_path)
102         # check early stopping
103         # TODO if self.metrics.check_early_stopping(): --> in
      metrics tracker verlagern ob early stopping
104         if self.do_stop_early_now:
105           self.logger.info("Early stopping")
106           current_wandb_run.log({"stopped_early": True})
107           break
108       self.metrics.finish_fold()
109
110       if self.run_config['save_model']:
111         self.save_model()
```

```
112         current_wandb_run.save()
113         current_wandb_run.finish()
114         try:
115           del current_wandb_run # potential bug workaround fix
    attempt
116         except:
117           self.logger.info("No current_wandb_run to delete end
    perform fold")
118         self.logger.info(f"Finish WANDB KFold {current_split}")
119
120     # training the model
121     def train_epoch(self, dataloader, current_wandb_run):
122       self.model.train()
123       self.metrics.reset_epoch_metrics(validation=False)
124       dl_it = iter(dataloader)
125       num_batches = len(dataloader)
126       assert num_batches > 0, "Train Dataloader is empty"
127       self.logger.info(f"num_batches in train loader: {num_batches}"
    )
128       self.pbars.update_batch_parameter(num_batches)
129       current_wandb_run.log({"learnrate": self.optimizer.
    param_groups[0]['lr']})
130
131       for _, (data_batch, labels) in enumerate(dl_it):
132         loss, probabilities = self.process_batch(data_batch, labels,
     validation=False)
133
134       self.process_epoch_after(validation=False, current_wandb_run=
    current_wandb_run)
135
136     # validating the model
137     def validation_epoch(self, dataloader, current_wandb_run):
138       self.metrics.reset_epoch_metrics(validation=True)
139       self.model.eval()
140       with torch.no_grad():
141         dl_it = iter(dataloader)
142         num_batches = len(dataloader)
143         assert num_batches > 0, "Validation Dataloader is empty"
144         self.logger.info(f"num_batches in validation loader: {
    num_batches}")
145         self.pbars.update_valid_batch_parameter(len(dataloader), "
    Validation batch")
146
147         for _, (data_batch, labels) in enumerate(dl_it):
148           self.process_batch(data_batch, labels, validation=True)
149         epoch_metric = self.process_epoch_after(validation=True,
    current_wandb_run=current_wandb_run)
150
151         # create rolling F1 and check early stopping
152         self.valid_f1_history.append(epoch_metric)
153         if len(self.valid_f1_history) >= 5: # atleast 5 epochs
154           rolling_f1 = np.mean(self.valid_f1_history[-4:])
155           self._check_early_stopping(rolling_f1)
156         else:
157           rolling_f1 = np.mean(self.valid_f1_history)
158         current_wandb_run.log({"valid_rolling_f1": rolling_f1})
```

```
159
160   # process a single batch
161   def process_batch(self, data_batch, labels, validation):
162     try:
163       if not validation:
164         prefix = "train_"
165         pbar_number = 3
166       else:
167         prefix = "valid_"
168         pbar_number = 4
169       if self.base_config['mode'] == cfg.modes['vggish']:
170         data_batch = data_batch.squeeze(0)
171       data_batch = data_batch.to(self.device)
172       labels = labels.to(self.device)
173       self.tensor_logger.info(f"labels shape {labels.shape}")
174       self.tensor_logger.info(f"data_batch shape {data_batch.shape
    }")
175       self.tensor_logger.info(f"data_batch type {type(data_batch)}
    ")
176       self.tensor_logger.debug(f"labels {labels}")
177       loss, probabilities  = self.predict_step(data_batch, labels)
178       if not validation:
179         self.optimizer_step(loss)
180       self.metrics.update_step(probabilities=probabilities, labels
    =labels, loss=loss, validation=validation)
181       loss_message = {f"{prefix}_loss": loss.item()}
182       self.pbars.update(pbar_number, postfix=loss_message)
183       return loss, probabilities
184     except Exception as e:
185       self.logger.error(f"Failed to process batch: {e}")
186       raise
187
188   # prediction of one batch
189   def predict_step(self, inputs, labels):
190     with autocast():
191       self.tensor_logger.info(f"predict_step inputs shape {inputs.
    shape}")
192       outputs = self.model(inputs)
193       loss = self.criterion(outputs, labels)
194     l1_regulation = self.run_config['l1_weight']
195     if l1_regulation > 0:
196       regularization_loss = 0
197       for param in self.model.parameters():
198         regularization_loss += torch.norm(param, p=1)
199       loss += l1_regulation * regularization_loss
200
201     probabilities = torch.softmax(outputs, dim=1)
202     return loss, probabilities
203
204   # called after each epoch, after all batches
205   def process_epoch_after(self, validation, current_wandb_run):
206     epoch_metrics: dict = self.metrics.save_epoch_metrics(
    validation=validation)
207     self.scheduler_step(epoch_metrics.get('mean_loss'), validation
    =validation)
208     if validation:
```

```
209         prefix = "valid_"
210         self.pbars.reset_valid_batch()
211       else:
212         prefix = "train_"
213         self.pbars.reset_batch()
214       message = {f"{prefix}metrics": epoch_metrics.copy(), "epoch":
      self.current_epoch, "kfold": self.current_kfold}
215       message.get(f"{prefix}metrics").pop(metrics.Names.ROC_DATA)
216       current_wandb_run.log(message)
217
218       cm_name = f"cm_{prefix}epoch_{self.current_epoch}_fold_{self.
      current_kfold}.png"
219       roc_name = f"roc_{prefix}epoch_{self.current_epoch}_fold_{self
      .current_kfold}.png"
220
221       cm_path = path.join(self.job_base_path, "cm", cm_name)
222       roc_path = path.join(self.job_base_path, "roc", roc_name)
223       fold_cm_image = utils.Plotting.create_confusion_matrix(
      epoch_metrics.get('confusion'), title=cm_name, normalize=True)
224       roc_fig, roc_ax = utils.Plotting.create_roc(roc_data=
      epoch_metrics.get(metrics.Names.ROC_DATA), fold=self.
      current_kfold, epoch=self.current_epoch)
225
226       utils.Plotting.show_save(fold_cm_image, cm_path, show=self.
      logger.isEnabledFor(logging.DEBUG))
227       utils.Plotting.show_save(roc_fig, roc_path, show=self.logger.
      isEnabledFor(logging.DEBUG))
228
229       if validation:
230         self.logger.warn(f"Epoch validation metrics: \n{message}")
231         current_wandb_run.log({"cm_valid": wandb.Image(cm_path,
      caption=cm_name)})
232         current_wandb_run.log({"roc_train": wandb.Image(roc_path,
      caption=roc_name)})
233       else:
234         self.logger.info(f"Epoch train metrics:\n{message}")
235         current_wandb_run.log({"cm_valid": wandb.Image(cm_path,
      caption=cm_name)})
236         current_wandb_run.log({"roc_valid": wandb.Image(roc_path,
      caption=roc_name)})
237       return epoch_metrics.get('f1') # return best representative
      metric for this epoch
238
239   # stepping the optimizer - called in training after each batch
240   def optimizer_step(self, loss):
241     try:
242       self.optimizer.zero_grad()
243       self.scaler.scale(loss).backward()
244       self.scaler.unscale_(self.optimizer)
245       nn.utils.clip_grad_norm_(self.model.parameters(), 1.0)
246       self.scaler.step(self.optimizer)
247       self.scaler.update()
248     except Exception as e:
249       self.logger.error("Error during optimizer step", exc_info=
      True)
250       self.logger.error(f"Loss: {loss}")
```

```
251
252   # stepping the scheduler - called in training and validation
       after all batches
253   def scheduler_step(self, loss, validation=False):
254     try:
255       if loss != type(torch.Tensor):
256         loss = torch.tensor(loss, requires_grad=True).to(self.
      device)
257       if validation and isinstance(self.scheduler, torch.optim.
      lr_scheduler.ReduceLROnPlateau):
258         self.scheduler.step(loss)
259         self.logger.debug(f"ReduceLROnPlateau stepped with loss {
      loss}")
260       elif not validation and isinstance(self.scheduler, torch.
      optim.lr_scheduler.StepLR):
261         self.scheduler.step()
262         self.logger.debug(f"StepLR stepped")
263       elif not validation and isinstance(self.scheduler, torch.
      optim.lr_scheduler.CosineAnnealingLR):
264         self.scheduler.step()
265         self.logger.debug(f"CosineAnnealingLR stepped")
266       else:
267         self.logger.debug("No scheduler step")
268     except Exception as e:
269       self.logger.error("Error during scheduler step")
270       self.logger.error(e)
271       self.logger.error(f"Loss: {loss}")
272
273   # check if early stopping is enabled and if the rolling F1 is
       below the threshold
274   def _check_early_stopping(self, rolling_f1):
275     if self.run_config['enable_earlystop']:
276       if rolling_f1 < self.run_config['earlystop_threshold']:
277         self.logger.warning(f"F1 score is {rolling_f1}. Check your
       data and labels")
278         self.do_stop_early_now = True
279
280   # reset the state to state_dict of model
281   def load_checkpoint(self, path):
282     checkpoint = torch.load(path)
283     self.current_epoch = checkpoint['epoch']
284     self.current_kfold = checkpoint['kfold']
285     self.model.load_state_dict(checkpoint['model_state_dict'])
286     self.optimizer.load_state_dict(checkpoint['
      optimizer_state_dict'])
287     for i in range(0,self.current_kfold):
288       self.pbars.update(1)
289     for i in range(0,self.current_epoch):
290       self.pbars.update(2)
291     self.logger.info(f"Loaded checkpoint from {path}")
```

Quellcode B.0.4: Training und Validierung

```
1 name: physionet
2 channels:
3   - anaconda
4   - conda-forge
```

```
  - pytorch
  - nvidia
  - defaults
dependencies:
  - anyio=3.5.0=py310haa95532_0
  - appdirs=1.4.4=pyh9f0ad1d_0
  - argon2-cffi=21.3.0=pyhd3eb1b0_0
  - argon2-cffi-bindings=21.2.0=py310h2bbff1b_0
  - asttokens=2.0.5=pyhd3eb1b0_0
  - attrs=22.1.0=py310haa95532_0
  - audioread=3.0.0=py310h5588dad_1
  - babel=2.11.0=py310haa95532_0
  - backcall=0.2.0=pyhd3eb1b0_0
  - beautifulsoup4=4.11.1=py310haa95532_0
  - blas=2.118=mkl
  - blas-devel=3.9.0=18_win64_mkl
  - bleach=4.1.0=pyhd3eb1b0_0
  - brotli=1.1.0=hcfcfb64_0
  - brotli-bin=1.1.0=hcfcfb64_0
  - brotli-python=1.1.0=py310h00ffb61_0
  - bzip2=1.0.8=h8ffe710_4
  - ca-certificates=2023.7.22=h56e8100_0
  - certifi=2023.7.22=pyhd8ed1ab_0
  - cffi=1.15.1=py310h8d17308_5
  - charset-normalizer=3.2.0=pyhd8ed1ab_0
  - click=8.1.7=win_pyh7428d3b_0
  - colorama=0.4.6=pyhd8ed1ab_0
  - comm=0.1.2=py310haa95532_0
  - contourpy=1.1.1=py310h232114e_0
  - cuda-cccl=12.2.140=0
  - cuda-cudart=11.7.99=0
  - cuda-cudart-dev=11.7.99=0
  - cuda-cupti=11.7.101=0
  - cuda-libraries=11.7.1=0
  - cuda-libraries-dev=11.7.1=0
  - cuda-nvrtc=11.7.99=0
  - cuda-nvrtc-dev=11.7.99=0
  - cuda-nvtx=11.7.91=0
  - cuda-runtime=11.7.1=0
  - cycler=0.11.0=pyhd8ed1ab_0
  - debugpy=1.5.1=py310hd77b12b_0
  - decorator=5.1.1=pyhd8ed1ab_0
  - defusedxml=0.7.1=pyhd3eb1b0_0
  - dill=0.3.7=pyhd8ed1ab_0
  - docker-pycreds=0.4.0=py_0
  - entrypoints=0.4=py310haa95532_0
  - executing=0.8.3=pyhd3eb1b0_0
  - filelock=3.12.4=pyhd8ed1ab_0
  - fonttools=4.42.1=py310h8d17308_0
  - freetype=2.12.1=hdaf720e_2
  - gettext=0.21.1=h5728263_0
  - gitdb=4.0.10=pyhd8ed1ab_0
  - gitpython=3.1.37=pyhd8ed1ab_0
  - glib=2.78.0=h12be248_0
  - glib-tools=2.78.0=h12be248_0
  - gst-plugins-base=1.22.6=h001b923_0
```

```
  - gstreamer=1.22.6=hb4038d2_0
  - icu=70.1=h0e60522_0
  - idna=3.4=pyhd8ed1ab_0
  - intel-openmp=2023.2.0=h57928b3_49503
  - ipykernel=6.19.2=py310h9909e9c_0
  - ipython=8.8.0=py310haa95532_0
  - ipython_genutils=0.2.0=pyhd3eb1b0_1
  - ipywidgets=7.6.5=pyhd3eb1b0_1
  - jedi=0.18.1=py310haa95532_1
  - jinja2=3.1.2=pyhd8ed1ab_1
  - joblib=1.3.2=pyhd8ed1ab_0
  - jpeg=9e=hcfcfb64_3
  - json5=0.9.6=pyhd3eb1b0_0
  - jsonschema=4.16.0=py310haa95532_0
  - jupyter=1.0.0=py310haa95532_8
  - jupyter_client=7.4.8=py310haa95532_0
  - jupyter_console=6.4.4=py310haa95532_0
  - jupyter_core=5.1.1=py310haa95532_0
  - jupyter_server=1.23.4=py310haa95532_0
  - jupyterlab=3.5.3=py310haa95532_0
  - jupyterlab_pygments=0.1.2=py_0
  - jupyterlab_server=2.16.5=py310haa95532_0
  - jupyterlab_widgets=1.0.0=pyhd3eb1b0_1
  - kiwisolver=1.4.5=py310h232114e_1
  - krb5=1.20.1=heb0366b_0
  - lame=3.100=hcfcfb64_1003
  - lazy_loader=0.3=pyhd8ed1ab_0
  - lcms2=2.15=ha5c8aab_0
  - lerc=4.0.0=h63175ca_0
  - libabseil=20230802.1=cxx17_h63175ca_0
  - libblas=3.9.0=18_win64_mkl
  - libbrotlicommon=1.1.0=hcfcfb64_0
  - libbrotlidec=1.1.0=hcfcfb64_0
  - libbrotlienc=1.1.0=hcfcfb64_0
  - libcblas=3.9.0=18_win64_mkl
  - libclang=15.0.7=default_h77d9078_3
  - libclang13=15.0.7=default_h77d9078_3
  - libcublas=11.10.3.66=0
  - libcublas-dev=11.10.3.66=0
  - libcufft=10.7.2.124=0
  - libcufft-dev=10.7.2.124=0
  - libcurand=10.3.3.141=0
  - libcurand-dev=10.3.3.141=0
  - libcusolver=11.4.0.1=0
  - libcusolver-dev=11.4.0.1=0
  - libcusparse=11.7.4.91=0
  - libcusparse-dev=11.7.4.91=0
  - libdeflate=1.17=hcfcfb64_0
  - libexpat=2.5.0=h63175ca_1
  - libffi=3.4.2=h8ffe710_5
  - libflac=1.4.3=h63175ca_0
  - libglib=2.78.0=he8f3873_0
  - libhwloc=2.9.2=default_haede6df_1009
  - libiconv=1.17=h8ffe710_0
  - liblapack=3.9.0=18_win64_mkl
  - liblapacke=3.9.0=18_win64_mkl
```

```
117   - libnpp=11.7.4.75=0
118   - libnpp-dev=11.7.4.75=0
119   - libnvjpeg=11.8.0.2=0
120   - libnvjpeg-dev=11.8.0.2=0
121   - libogg=1.3.4=h8ffe710_1
122   - libopus=1.3.1=h8ffe710_1
123   - libpng=1.6.39=h19919ed_0
124   - libprotobuf=4.23.4=hb8276f3_6
125   - librosa=0.10.1=pyhd8ed1ab_0
126   - libsndfile=1.2.2=h2628c91_0
127   - libsodium=1.0.18=h62dcd97_0
128   - libsqlite=3.43.0=hcfcfb64_0
129   - libtiff=4.5.0=hf8721a0_2
130   - libuv=1.44.2=hcfcfb64_1
131   - libvorbis=1.3.7=h0e60522_0
132   - libwebp-base=1.3.2=hcfcfb64_0
133   - libxcb=1.13=hcd874cb_1004
134   - libxml2=2.11.5=hc3477c8_1
135   - libzlib=1.2.13=hcfcfb64_5
136   - lightning-utilities=0.9.0=pyhd8ed1ab_0
137   - llvmlite=0.40.1=py310hb84602e_0
138   - m2w64-gcc-libgfortran=5.3.0=6
139   - m2w64-gcc-libs=5.3.0=7
140   - m2w64-gcc-libs-core=5.3.0=7
141   - m2w64-gmp=6.1.0=2
142   - m2w64-libwinpthread-git=5.0.0.4634.697f757=2
143   - markupsafe=2.1.3=py310h8d17308_1
144   - matplotlib=3.8.0=py310h5588dad_1
145   - matplotlib-base=3.8.0=py310hc9baf74_1
146   - matplotlib-inline=0.1.6=py310haa95532_0
147   - mistune=0.8.4=py310h2bbff1b_1000
148   - mkl=2022.1.0=h6a75c08_874
149   - mkl-devel=2022.1.0=h57928b3_875
150   - mkl-include=2022.1.0=h6a75c08_874
151   - mpg123=1.31.3=h63175ca_0
152   - mpmath=1.3.0=pyhd8ed1ab_0
153   - msgpack-python=1.0.6=py310h232114e_0
154   - msys2-conda-epoch=20160418=1
155   - munkres=1.1.4=pyh9f0ad1d_0
156   - nbclassic=0.4.8=py310haa95532_0
157   - nbclient=0.5.13=py310haa95532_0
158   - nbconvert=6.4.4=py310haa95532_0
159   - nbformat=5.7.0=py310haa95532_0
160   - nest-asyncio=1.5.6=py310haa95532_0
161   - networkx=3.1=pyhd8ed1ab_0
162   - notebook=6.5.2=py310haa95532_0
163   - notebook-shim=0.2.2=py310haa95532_0
164   - numba=0.57.1=py310h19bcfe9_0
165   - numpy=1.23.5=py310h4a8f9c9_0
166   - openjpeg=2.5.0=ha2aaf27_2
167   - openssl=3.1.3=hcfcfb64_0
168   - packaging=23.1=pyhd8ed1ab_0
169   - pandas=2.1.1=py310hecd3228_0
170   - pandocfilters=1.5.0=pyhd3eb1b0_0
171   - parso=0.8.3=pyhd3eb1b0_0
172   - pathtools=0.1.2=py_1
```

```
  - patsy=0.5.3=pyhd8ed1ab_0
  - pcre2=10.40=h17e33f8_0
  - pickleshare=0.7.5=pyhd3eb1b0_1003
  - pillow=9.4.0=py310hdbb7713_1
  - pip=23.2.1=pyhd8ed1ab_0
  - platformdirs=3.10.0=pyhd8ed1ab_0
  - ply=3.11=py_1
  - pooch=1.7.0=pyha770c72_3
  - portaudio=19.6.0=h63175ca_9
  - prometheus_client=0.14.1=py310haa95532_0
  - prompt-toolkit=3.0.20=pyhd3eb1b0_0
  - prompt_toolkit=3.0.20=hd3eb1b0_0
  - protobuf=4.23.4=py310h19be30a_3
  - psutil=5.9.5=py310h8d17308_1
  - pthread-stubs=0.4=hcd874cb_1001
  - pthreads-win32=2.9.1=hfa6e2cd_3
  - pure_eval=0.2.2=pyhd3eb1b0_0
  - pycparser=2.21=pyhd8ed1ab_0
  - pygments=2.11.2=pyhd3eb1b0_0
  - pynndescent=0.5.10=pyh1a96a4e_0
  - pyparsing=3.0.9=pyhd8ed1ab_0
  - pyqt=5.15.9=py310h1fd54f2_5
  - pyqt5-sip=12.12.2=py310h00ffb61_5
  - pyrsistent=0.18.0=py310h2bbff1b_0
  - pysocks=1.7.1=pyh0701188_6
  - pysoundfile=0.12.1=pyhd8ed1ab_0
  - python=3.10.12=h4de0772_0_cpython
  - python-dateutil=2.8.2=pyhd8ed1ab_0
  - python-fastjsonschema=2.16.2=py310haa95532_0
  - python-sounddevice=0.4.6=pyhd8ed1ab_0
  - python-tzdata=2023.3=pyhd8ed1ab_0
  - python_abi=3.10=4_cp310
  - pytorch=2.0.1=py3.10_cuda11.7_cudnn8_0
  - pytorch-cuda=11.7=h16d0643_5
  - pytorch-mutex=1.0=cuda
  - pytz=2023.3.post1=pyhd8ed1ab_0
  - pywin32=305=py310h2bbff1b_0
  - pywinpty=2.0.2=py310h5da7b33_0
  - pyyaml=6.0.1=py310h8d17308_1
  - pyzmq=23.2.0=py310hd77b12b_0
  - qt-main=5.15.8=h720456b_6
  - qtpy=2.2.0=py310haa95532_0
  - requests=2.31.0=pyhd8ed1ab_0
  - resampy=0.4.2=pyhd8ed1ab_0
  - scikit-learn=1.3.1=py310hfd2573f_0
  - scipy=1.11.2=py310h70e3499_1
  - seaborn=0.13.0=hd8ed1ab_0
  - seaborn-base=0.13.0=pyhd8ed1ab_0
  - send2trash=1.8.0=pyhd3eb1b0_1
  - sentry-sdk=1.31.0=pyhd8ed1ab_0
  - setproctitle=1.3.2=py310h8d17308_2
  - setuptools=68.2.2=pyhd8ed1ab_0
  - sip=6.7.11=py310h00ffb61_1
  - six=1.16.0=pyh6c4a22f_0
  - smmap=3.0.5=pyh44b312d_0
  - sniffio=1.2.0=py310haa95532_1
```

```
  - soupsieve=2.3.2.post1=py310haa95532_0
  - soxr=0.1.3=hcfcfb64_3
  - soxr-python=0.3.5=py310h9b08ddd_0
  - stack_data=0.2.0=pyhd3eb1b0_0
  - statsmodels=0.14.0=py310h9b08ddd_1
  - sympy=1.12=pyh04b8f61_3
  - tabulate=0.9.0=pyhd8ed1ab_1
  - tbb=2021.10.0=h91493d7_0
  - terminado=0.17.1=py310haa95532_0
  - testpath=0.6.0=py310haa95532_0
  - threadpoolctl=3.2.0=pyha21a80b_0
  - tk=8.6.13=hcfcfb64_0
  - toml=0.10.2=pyhd8ed1ab_0
  - tomli=2.0.1=pyhd8ed1ab_0
  - torchinfo=1.8.0=pyhd8ed1ab_0
  - torchmetrics=1.1.1=pyhd8ed1ab_0
  - tornado=6.3.3=py310h8d17308_1
  - tqdm=4.66.1=pyhd8ed1ab_0
  - traitlets=5.7.1=py310haa95532_0
  - typing-extensions=4.8.0=hd8ed1ab_0
  - typing_extensions=4.8.0=pyha770c72_0
  - tzdata=2023c=h71feb2d_0
  - ucrt=10.0.22621.0=h57928b3_0
  - umap-learn=0.5.4=py310h5588dad_0
  - unicodedata2=15.0.0=py310h8d17308_1
  - urllib3=2.0.5=pyhd8ed1ab_0
  - vc=14.3=h64f974e_17
  - vc14_runtime=14.36.32532=hdcecf7f_17
  - vs2015_runtime=14.36.32532=h05e6639_17
  - wandb=0.15.11=pyhd8ed1ab_0
  - wcwidth=0.2.5=pyhd3eb1b0_0
  - webencodings=0.5.1=py310haa95532_1
  - websocket-client=0.58.0=py310haa95532_4
  - wheel=0.41.2=pyhd8ed1ab_0
  - widgetsnbextension=3.5.2=py310haa95532_0
  - win_inet_pton=1.1.0=pyhd8ed1ab_6
  - winpty=0.4.3=4
  - xorg-libxau=1.0.11=hcd874cb_0
  - xorg-libxdmcp=1.1.3=hcd874cb_0
  - xz=5.2.6=h8d14728_0
  - yaml=0.2.5=h8ffe710_2
  - zeromq=4.3.4=hd77b12b_0
  - zstd=1.5.5=h12be248_0
  - pip:
      - audiomentations==0.33.0
      - qtconsole==5.4.0
      - torchaudio==2.0.2
      - torchvision==0.15.2
prefix: E:\Work\mambaforge\envs\physionet
```

Quellcode B.0.5: YAML Conda Export

```
import json
import pandas as pd
import numpy as np
import logging
from abc import ABC, abstractmethod
```

```
6  import parameters as cfg
7  import torchmetrics as tm
8  import tabulate
9  from os.path import join
10 from sklearn.metrics import accuracy_score, f1_score,
       precision_score, recall_score, confusion_matrix,
       matthews_corrcoef, roc_curve, roc_auc_score
11 from torch import Tensor
12 from . import utils
13 import matplotlib.pyplot as plt
14
15 class Names:
16   ACCURACY  = "acc"
17   F1        = "f1"
18   PRECISION = "precision"
19   RECALL    = "recall"
20   CONFUSION = "confusion"
21   SPECIFICITY = "specificity"
22   MCC       = "mcc"
23   ROC_DATA  = "roc_data"
24   AUROC     = "auroc"
25   MEAN_LOSS = "mean_loss"
26   EPOCH     = "epoch"
27   FOLD      = "fold"
28   MULTICLASS  = "multiclass"
29   PLR       = "plr"
30   NLR       = "nlr"
31   BINARY    = "binary"
32   MODE      =   "mode"
33   FPR       = "fpr"
34   TPR       = "tpr"
35   THRESHOLDS  = "thresholds"
36   TRAINING_EPOCHS   = "training_epochs"
37   VALIDATION_EPOCHS = "validation_epochs"
38
39
40 class MetricsInterface(ABC):
41
42   def __init__(self, task_type, num_classes, device=None) -> None:
43     super().__init__()
44     self.task_type = task_type
45     self.num_classes = num_classes
46     self.device = device
47     self.predictions = []
48     self.labels = []
49     self.loss = []
50     self.probabilities = []
51     self.one_update_done = False
52
53   def calculate_and_store_roc(self):
54     desired_thresholds = np.linspace(0, 1, 100)  # feste
       Schwellenwerte der ROC Kurve
55     auroc_data = []
56     fpr_data = []
57     tpr_data = []
58
```

```
59      if self.num_classes == 2:  # Binary classification
60        y_score = np.array(self.probabilities)[:, 1]
61        fpr, tpr, thresholds = roc_curve(self.labels, y_score)
62
63        # Interpolation
64        interpolated_fpr = np.interp(desired_thresholds, thresholds
    [::-1], fpr[::-1])
65        interpolated_tpr = np.interp(desired_thresholds, thresholds
    [::-1], tpr[::-1])
66
67        fpr_data.append(interpolated_fpr.tolist())
68        tpr_data.append(interpolated_tpr.tolist())
69        auroc_data = [roc_auc_score(self.labels, y_score)]
70      else:  # Multi-class classification - untested
71        for i in range(self.num_classes):
72          y_true_i = (np.array(self.labels) == i).astype(int)
73          y_score_i = np.array(self.probabilities)[:, i]
74          fpr, tpr, thresholds = roc_curve(y_true_i, y_score_i)
75
76          # Interpolation
77          interpolated_fpr = np.interp(desired_thresholds,
    thresholds[::-1], fpr[::-1])
78          interpolated_tpr = np.interp(desired_thresholds,
    thresholds[::-1], tpr[::-1])
79
80          fpr_data.append(interpolated_fpr.tolist())
81          tpr_data.append(interpolated_tpr.tolist())
82          auroc_data.append(roc_auc_score(y_true_i, y_score_i))
83
84      roc_data = {
85        Names.FPR: fpr_data,
86        Names.TPR: tpr_data,
87        Names.THRESHOLDS: desired_thresholds.tolist(),
88        Names.AUROC: auroc_data,
89      }
90      return roc_data
91
92    def _get_fake_update_data(self, num=2):
93      if num == 3:
94        probs_0 = np.array([[0.6, 0.2, 0.2]] * 20)
95        probs_1 = np.array([[0.1, 0.7, 0.2]] * 30)
96        probs_2 = np.array([[0.1, 0.1, 0.8]] * 50)
97        fake_probabilities = np.vstack([probs_0, probs_1, probs_2])
98        # Für die Vorhersagen nehmen wir an, dass:
99        # - 10 der Klasse-0-Instanzen richtig klassifiziert wurden (
    TP für Klasse 0)
100       # - 20 der Klasse-1-Instanzen richtig klassifiziert wurden (
    TP für Klasse 1)
101       # - 40 der Klasse-2-Instanzen richtig klassifiziert wurden (
    TP für Klasse 2)
102
103       # Die restlichen Vorhersagen sind zufällig falsch, um die
    Matrix interessanter zu gestalten.
104       # Angenommen, die falschen Vorhersagen sind wie folgt
    verteilt:
105       # - 10 der Klasse-0-Instanzen wurden als Klasse 1
```

```
    klassifiziert (FP für Klasse 1)
106       # - 10 der Klasse-1-Instanzen wurden als Klasse 2
    klassifiziert (FP für Klasse 2)
107       # - 10 der Klasse-2-Instanzen wurden als Klasse 0
    klassifiziert (FP für Klasse 0)
108       fake_labels = [0]*20 + [1]*30 + [2]*50
109       fake_predictions = [0]*10 + [1]*10 + [1]*20 + [2]*10 +
    [2]*40 + [0]*10
110
111     elif num == 2:
112       fake_labels = [1]*10 + [0]*40 + [1]*20 + [0]*30
113       fake_predictions = [1]*10 + [1]*40 + [0]*20 + [0]*30
114       # sum = 100 # total 1: 30, total 0: 70
115       # tp = 10
116       # fp = 40
117       # fn = 20
118       # tn = 30
119       # Ziel: [TP, FP; FN, TN]
120       # [10, 40; 20, 30]
121       fake_probabilities = np.array([[0.6, 0.4]] * 10 + [[0.3,
    0.7]] * 40 + [[0.6, 0.4]] * 20 + [[0.3, 0.7]] * 30)
122     return fake_labels, fake_predictions, fake_probabilities
123
124
125   # update the history each batch with new predictions and labels
126   @abstractmethod
127   def update(self, probabilities, labels, loss=None):
128     pass
129
130   # compute the scores based on the history
131   @abstractmethod
132   def compute(self):
133     pass
134
135   # delete all history, ready for a new fold/epoch
136   def reset(self):
137     self.predictions = []
138     self.labels = []
139     self.loss = []
140     self.probabilities = []
141
142 class SKLearnMetricsAdapter(MetricsInterface):
143   def __init__(self, num_classes, task_type, device):
144     super().__init__(task_type, num_classes, device)
145
146   def update(self, probabilities, labels, loss=None, num_fake=0):
147     if self.one_update_done:
148       return
149
150     predictions = probabilities.argmax(axis=1)
151
152     if num_fake > 0:
153       fake_labels, fake_predictions, fake_probabilities = self.
    _get_fake_update_data(num_fake)
154       predictions = fake_predictions
155       labels = fake_labels
```

```
156         probabilities = fake_probabilities
157         self.one_update_done = True
158       self.predictions.extend(predictions)
159       self.labels.extend(labels)
160       self.probabilities.extend(probabilities)
161       if loss is not None:
162         self.loss.append(loss)
163
164     def compute(self):
165       metrics = {}
166       metrics[Names.ACCURACY] = accuracy_score(self.labels, self.
      predictions)
167       metrics[Names.F1] = f1_score(self.labels, self.predictions,
      average='macro')
168       metrics[Names.PRECISION] = precision_score(self.labels, self.
      predictions, average='macro')
169       metrics[Names.RECALL] = recall_score(self.labels, self.
      predictions, average='macro')
170       metrics[Names.CONFUSION] = confusion_matrix(self.labels, self.
      predictions, normalize=None)
171       tn, fp, fn, tp = metrics[Names.CONFUSION].ravel()
172       metrics[Names.SPECIFICITY] = tn / (tn + fp)
173       metrics[Names.MCC] = matthews_corrcoef(self.labels, self.
      predictions)
174
175       if self.task_type == Names.BINARY:
176         metrics[Names.CONFUSION] = metrics[Names.CONFUSION].T[:,
      ::-1][::-1, :] # Torchmetrics 2 class
177
178       metrics[Names.ROC_DATA] = self.calculate_and_store_roc()
179       metrics[Names.AUROC] = metrics[Names.ROC_DATA][Names.AUROC]
180       metrics[Names.PLR] = metrics[Names.RECALL] / (1 - metrics[
      Names.SPECIFICITY] + 1e-9)
181       metrics[Names.NLR] = (1 - metrics[Names.RECALL]) / (metrics[
      Names.SPECIFICITY] + 1e-9)
182
183       if len(self.loss) > 0:
184         metrics[Names.MEAN_LOSS] = np.mean(self.loss)
185       else:
186         metrics[Names.MEAN_LOSS] = 0.0
187       return metrics
188
189     def reset(self):
190       super().reset()
191
192 class TorchMetricsAdapter(MetricsInterface):
193     def __init__(self, num_classes, task_type, device):
194       super().__init__(task_type, num_classes, device)
195       assert self.device is not None, "Device must be set for
      TorchMetricsAdapter"
196       self.num_classes = 2
197       self.task_type = "binary"
198       metrics_collection = tm.MetricCollection({
199         Names.ACCURACY: tm.Accuracy(num_classes=self.num_classes,
      average='macro', task=self.task_type),
200         Names.F1: tm.F1Score(num_classes=self.num_classes, average='
```

```
    macro', task=self.task_type),
201       Names.PRECISION: tm.Precision(num_classes=self.num_classes,
    average='macro', task=self.task_type),
202       Names.RECALL: tm.Recall(num_classes=self.num_classes,
    average='macro', task=self.task_type),
203       Names.CONFUSION: tm.ConfusionMatrix(num_classes=self.
    num_classes, normalize="none", threshold=0.5, task=self.
    task_type),
204       Names.SPECIFICITY: tm.Specificity(num_classes=self.
    num_classes, average='macro', task=self.task_type),
205       Names.MCC: tm.MatthewsCorrCoef(num_classes=self.num_classes,
     task=self.task_type),
206     })
207     self.loss = []
208     self.labels = []
209     self.probabiities = []
210     self.metrics_collection = metrics_collection.to(self.device)
211
212   def _fake_step(self, num=2):
213     fake_labels, fake_predictions, fake_probabilities = self.
    _get_fake_update_data(num)
214
215     # Umwandlung der Listen in PyTorch-Tensoren
216     fake_labels = Tensor(fake_labels).to(self.device)
217     fake_predictions = Tensor(fake_predictions).to(self.device)
218     self.metrics_collection.update(fake_predictions, fake_labels)
219     self.labels.extend(fake_labels.cpu().numpy())
220     self.probabiities.extend(fake_probabilities)
221
222   def update(self, probabilities, labels, loss=None):
223     predictions = probabilities.argmax(dim=1)
224     probabilities = probabilities.detach()
225
226     fake = False # set to True to test the metrics with fake data
227     if fake:
228       self._fake_step(3)
229     else:
230       self.metrics_collection.update(predictions, labels)
231       self.labels.extend(labels.cpu().numpy())
232       self.probabilities.extend(probabilities.cpu().numpy())
233     if loss is not None:
234       if isinstance(loss, Tensor):
235         loss = loss.item()
236       self.loss.append(loss)
237
238   def compute(self):
239     metrics: dict = self.metrics_collection.compute()
240     # Umstellen der Elemente, um das gewünschte Format zu erhalten
     [TP, FP; FN, TN]
241     # https://torchmetrics.readthedocs.io/en/stable/classification
    /confusion_matrix.html#binaryconfusionmatrix
242     tm_conv = metrics[Names.CONFUSION].cpu().numpy() # CM von
    Torchmetrics
243     if self.task_type == Names.BINARY:
244       target_matrix = tm_conv.T[:, ::-1][::-1, :] # Torchmetrics 2
     class
```

```
245     else:
246       target_matrix = tm_conv
247     metrics[Names.CONFUSION] = target_matrix
248     metrics[Names.ROC_DATA] = self.calculate_and_store_roc()
249     metrics[Names.AUROC] = metrics[Names.ROC_DATA][Names.AUROC]
250     metrics[Names.PLR] = metrics[Names.RECALL] / (1 - metrics[
    Names.SPECIFICITY] + 1e-8)
251     metrics[Names.NLR] = (1 - metrics[Names.RECALL]) / (metrics[
    Names.SPECIFICITY] + 1e-8)
252
253     if len(self.loss) > 0:
254       metrics[Names.MEAN_LOSS] = np.mean(self.loss)
255     else:
256       metrics[Names.MEAN_LOSS] = 0.0
257     return metrics
258
259   def reset(self):
260     super().reset()
261     self.metrics_collection.reset()
262
263
264 class MetricsTracker:
265
266   def __init__(self, config, metrics_class, device = None, logger=
    None, fl_precision=3):
267     if logger is None:
268       self.logger = logging.getLogger(cfg.MAIN_LOGGER)
269     else:
270       self.logger = logger
271     self.run_config = config
272     if config['job_base_path'] is None:
273       self.job_base_path = "./temp_models/"
274     else:
275       self.job_base_path = config['job_base_path']
276     self.num_classes = config['num_classes']
277     self.epoch_train_history = []
278     self.epoch_valid_history = []
279     self.fold_history = []
280     self.fl_precision = fl_precision
281     self.all_data_df = pd.DataFrame()
282     if self.num_classes > 2:
283       self.task_type = Names.MULTICLASS
284     else:
285       self.task_type = Names.BINARY
286
287     self.train_metrics = metrics_class(self.num_classes, self.
    task_type, device)
288     self.valid_metrics = metrics_class(self.num_classes, self.
    task_type, device)
289
290   def update_step(self, probabilities, labels, loss=None,
    validation=False):
291     if validation:
292       self.valid_metrics.update(probabilities, labels, loss)
293     else:
294       self.train_metrics.update(probabilities, labels, loss)
```

```
295
296   def reset_epoch_metrics(self, validation=False):
297     if validation:
298       self.valid_metrics.reset()
299     else:
300       self.train_metrics.reset()
301
302   def save_epoch_metrics(self, validation=False) -> dict:
303     if validation:
304       epoch_metrics = self.valid_metrics.compute()
305       epoch_metrics_dict = self._get_dict(epoch_metrics)
306       epoch_metrics_dict[Names.EPOCH] = len(self.
    epoch_valid_history) + 1
307       self.epoch_valid_history.append(epoch_metrics_dict)
308     else:
309       epoch_metrics = self.train_metrics.compute()
310       epoch_metrics_dict = self._get_dict(epoch_metrics)
311       epoch_metrics_dict[Names.EPOCH] = len(self.
    epoch_train_history) + 1
312       self.epoch_train_history.append(epoch_metrics_dict)
313     return epoch_metrics_dict
314
315   def _reset_epoch_history(self):
316     self.epoch_train_history = []
317     self.epoch_valid_history = []
318     self.train_metrics.reset()
319     self.valid_metrics.reset()
320
321   def finish_fold(self):
322     train_valid_metrics = {
323       Names.TRAINING_EPOCHS: self.epoch_train_history,
324       Names.VALIDATION_EPOCHS: self.epoch_valid_history,
325       Names.FOLD: len(self.fold_history) + 1
326     }
327     self.fold_history.append(train_valid_metrics)
328
329     for epoch_metrics in self.epoch_train_history:
330       epoch_metrics[Names.FOLD] = len(self.fold_history)
331       epoch_metrics["mode"] = "train"
332       self.all_data_df = pd.concat([self.all_data_df, pd.DataFrame
    ([epoch_metrics])], axis=0)
333
334     for epoch_metrics in self.epoch_valid_history:
335       epoch_metrics[Names.FOLD] = len(self.fold_history)
336       epoch_metrics["mode"] = "valid"
337       self.all_data_df = pd.concat([self.all_data_df, pd.DataFrame
    ([epoch_metrics])], axis=0)
338     self.save_metrics_to_json()
339     self._reset_epoch_history()
340
341
342 ####################
343
344
345   def _truncate_floats(self, x, precision=4):
346     if isinstance(x, float):
```

```
347         return round(x, precision)
348       elif isinstance(x, list):
349         return [self._truncate_floats(xi, precision) for xi in x]
350       else:
351         return x
352
353     def _get_value(self, metrics, metric_name):
354       if isinstance(metrics[metric_name], Tensor):
355         return float(f"{metrics[metric_name].item():.{self.
        fl_precision}f}")
356       elif isinstance(metrics[metric_name], float):
357         return float(f"{metrics[metric_name]:.{self.fl_precision}f}"
        )
358       elif isinstance(metrics[metric_name], list):
359         return self._truncate_floats(metrics[metric_name], self.
        fl_precision)
360       else:
361         self.logger.error(f"Error getting value for {metric_name}.
        Type: {type(metrics[metric_name])}")
362
363     def _get_dict(self, metrics):
364       confusion_metric = metrics[Names.CONFUSION]
365       roc_data = metrics[Names.ROC_DATA]
366       roc_data = {k: self._truncate_floats(v, self.fl_precision+1)
        for k, v in roc_data.items()}
367       if isinstance(confusion_metric, Tensor):
368         confusion_metric = confusion_metric.cpu().numpy().astype(np.
        float32)
369
370       confusion_round = np.round(confusion_metric, self.fl_precision
        )
371       confusion_list  = self._truncate_floats(confusion_round.tolist
        (), self.fl_precision)
372       metric_dict = {
373         Names.ACCURACY    : self._get_value(metrics, Names.ACCURACY)
        ,
374         Names.F1      : self._get_value(metrics, Names.F1),
375         Names.PRECISION   : self._get_value(metrics, Names.PRECISION
        ),
376         Names.RECALL    : self._get_value(metrics, Names.RECALL),
377         Names.SPECIFICITY   : self._get_value(metrics, Names.
        SPECIFICITY),
378         Names.MCC       : self._get_value(metrics, Names.MCC),
379         Names.AUROC     : self._get_value(metrics, Names.AUROC),
380         Names.PLR       : self._get_value(metrics, Names.PLR),
381         Names.NLR       : self._get_value(metrics, Names.NLR),
382         Names.CONFUSION   : confusion_list,
383         Names.ROC_DATA    : roc_data,
384       }
385       if len(metrics[Names.CONFUSION]) > 2:
386         # Adjust format of confusion matrix for n classes
387         # Not tested!
388         n = len(metrics[Names.CONFUSION])
389         metric_dict[Names.CONFUSION] = [metrics[Names.CONFUSION][i].
        tolist() for i in range(n)]
390
```

```
391     if isinstance(metrics[Names.MEAN_LOSS], Tensor):
392       metric_dict[Names.MEAN_LOSS] = self.train_metrics.loss.
        compute().item()
393     elif isinstance(metrics[Names.MEAN_LOSS], float):
394       metric_dict[Names.MEAN_LOSS] = metrics[Names.MEAN_LOSS]
395     elif isinstance(metrics[Names.MEAN_LOSS], tm.MeanMetric):
396       metric_dict[Names.MEAN_LOSS] = metrics[Names.MEAN_LOSS].
        compute().item()
397
398     metric_dict[Names.EPOCH] = len(self.epoch_train_history) + 1
399     metric_dict[Names.FOLD] = len(self.fold_history) + 1
400     return metric_dict
401
402   # postfix to be used in tqmd postfix after an epoch
403   def get_last_validation_postfix(self):
404     if len(self.epoch_valid_history) > 0:
405       last_valid_epoch = self.epoch_valid_history[-1]
406       # drop the epoch and fold keys + ROC data
407       last_valid_epoch = {k:v for k,v in last_valid_epoch.items()
        if k not in [Names.EPOCH, Names.FOLD, Names.ROC_DATA]}
408       return last_valid_epoch
409     else:
410       return {}
411
412   def _prepare_result_table(self, dataframe, mode):
413     # select which metrics to show in table
414     metrics = [Names.ACCURACY, Names.F1, Names.PRECISION, Names.
        RECALL, Names.SPECIFICITY, Names.MCC, Names.MEAN_LOSS, Names.
        PLR, Names.NLR]
415     # TODO AUROC
416     aggfunc = {metric: 'mean' for metric in metrics}
417     dataframe = dataframe[dataframe['mode'] == mode]
418     dataframe = dataframe.drop(columns=[Names.MODE])
419     # drop all not in metrics
420     dataframe = dataframe.drop(columns=[col for col in dataframe.
        columns if col not in metrics and col not in [Names.EPOCH,
        Names.FOLD]])
421     dataframe[metrics] = dataframe[metrics].apply(pd.to_numeric)
422     dataframe = dataframe.pivot_table(index=Names.EPOCH, values=
        metrics, aggfunc=aggfunc)
423     return dataframe
424
425   def get_fold_averages_table(self, show_training=True):
426     df = self.all_data_df
427     if df.empty:
428       return pd.DataFrame()
429
430     try:
431       if show_training:
432         train_df = self._prepare_result_table(df, 'train')
433       else:
434         train_df = pd.DataFrame()
435       valid_df = self._prepare_result_table(df, 'valid')
436
437       # Hier verwenden wir pd.concat anstatt join
438       dfs = []
```

```
439         if not train_df.empty:
440           train_df.columns = pd.MultiIndex.from_product([['train'],
    train_df.columns])
441           dfs.append(train_df)
442         if not valid_df.empty:
443           valid_df.columns = pd.MultiIndex.from_product([['valid'],
    valid_df.columns])
444           dfs.append(valid_df)
445
446         result_df = pd.concat(dfs, axis=1, sort=True)
447         result_df.columns = result_df.columns.map('_'.join)
448         # Sortieren der Spalten
449         sorted_columns = sorted(result_df.columns, key=lambda x: (x.
    split('_')[-1], x.split('_')[0]))
450         result_df = result_df[sorted_columns]
451
452       except ValueError as e:
453         self.logger.error(f"Error computing epoch metrics: {e}")
454         result_df = pd.DataFrame()
455       return result_df
456
457     def save_metrics_to_json(self):
458       path = join(self.job_base_path, "metrics.json")
459       try:
460         with open(path, 'w') as f:
461           json.dump(self.fold_history, f, cls=utils.MLUtil.
    NumpyEncoder)
462         self.logger.info(f"Metrics saved to {path}")
463       except Exception as e:
464         self.logger.error(f"Error saving metrics: {e} to {path}")
465
466     # plot of each run, train and valid in one plot. If kfold, then
      plot the averages. Train=blue, validation=orange. Do one large
      plot with subplots. Do it for accuarcy and f1
467     def plot_metrics(self):
468
469       import seaborn as sns
470       sns.set_theme()
471       df = self.all_data_df
472       if len(df) == 0:
473         return
474
475       metrics = [Names.ACCURACY, Names.F1, Names.PRECISION, Names.
    RECALL, Names.SPECIFICITY, Names.MCC, Names.MEAN_LOSS, Names.
    AUROC, Names.PLR, Names.NLR]
476       aggfunc = {metric: 'mean' for metric in metrics}
477
478       try:
479         # TODO check if train is even filled - modular, allow
    missing values
480         train_df = df[df['mode'] == 'train']
481         valid_df = df[df['mode'] == 'valid']
482
483         train_df = train_df.drop(columns=['mode'])
484         valid_df = valid_df.drop(columns=['mode'])
485
```

```
486       train_df[metrics] = train_df[metrics].apply(pd.to_numeric)
487       valid_df[metrics] = valid_df[metrics].apply(pd.to_numeric)
488
489       result_df = train_df.pivot_table(index=Names.EPOCH, values=
    metrics, aggfunc=aggfunc)
490       result_df.columns = [f'{col}_train' for col in result_df.
    columns]
491
492       result_df = result_df.join(valid_df.pivot_table(index=Names.
    EPOCH, values=metrics, aggfunc=aggfunc))
493       result_df.columns = [f'{col}_valid' if col in metrics else
    col for col in result_df.columns]
494       result_df = result_df.sort_index(axis=1)
495     except ValueError as e:
496       self.logger.error(f"Error computing epoch metrics: {e}")
497       result_df = pd.DataFrame()
498     pRows = 2
499     pCols = 5
500     assert len(metrics) <= pRows * pCols, f"Too many metrics to
    plot. Max is {pRows * pCols}"
501     fig, axes = plt.subplots(pRows, pCols, figsize=(17, 10))
502     axes = axes.flatten()
503     for i, metric in enumerate(metrics):
504       train_metric = f"{metric}_train"
505       valid_metric = f"{metric}_valid"
506       axes[i].plot(result_df[train_metric], label=train_metric)
507       axes[i].set_xticks(range(len(result_df)+1))
508       #axes[i].set_xticklabels(result_df.index)
509       axes[i].set_xlim(1, len(result_df))
510       axes[i].set_xlabel("Epoch")
511       axes[i].set_ylim(0, 1)
512       if metric == Names.MCC:
513         axes[i].set_ylim(-1, 1)
514       if metric == Names.NLR or metric == Names.PLR:
515         # use auto scaling
516         axes[i].set_ylim(None, None)
517
518       axes[i].plot(result_df[valid_metric], label=valid_metric)
519       axes[i].set_title(metric)
520       axes[i].legend()
521
522       if i >= len(metrics):
523         break
524     plt.savefig(join(self.job_base_path, "metrics.png"))
525     plt.show()
526
527
528   def print_all_metrics(self, epoch=-1) -> None:
529     metrics = self.fold_history
530     """
531     Print a table of average metrics and their standard deviation
    across all folds.
532
533     Args:
534       metrics (List[Dict]): List of metric dictionaries for each
    fold.
```

```
535     """
536
537     # Initialisierung der Listen zur Speicherung der Metriken für
    jeden Fold
538     acc, f1, precision, recall, specificity, mcc, plr, nlr = [],
    [], [], [], [], [], [], []
539
540     mode = "validation_epochs"
541
542     # Sammle Metriken für jeden Fold
543     for fold in metrics:
544       fold_data = fold[mode]
545       last_epoch = fold_data[epoch]
546
547       acc.append(last_epoch["acc"])
548       f1.append(last_epoch["f1"])
549       precision.append(last_epoch["precision"])
550       recall.append(last_epoch["recall"])
551       specificity.append(last_epoch["specificity"])
552       mcc.append(last_epoch["mcc"])
553       plr.append(last_epoch["plr"])
554       nlr.append(last_epoch["nlr"])
555
556     # Erstelle einen DataFrame zur leichteren Darstellung
557     df = pd.DataFrame({
558       "Metric": ["Accuracy", "F1", "Precision", "Recall", "
    Specificity", "MCC", "PLR", "NLR"],
559       "Average": [np.mean(acc), np.mean(f1), np.mean(precision),
    np.mean(recall), np.mean(specificity), np.mean(mcc),  np.mean(
    plr), np.mean(nlr)],
560       "STD": [np.std(acc), np.std(f1), np.std(precision), np.std(
    recall), np.std(specificity), np.std(mcc), np.std(plr), np.std(
    nlr)]
561     })
562
563     # Drucke den DataFrame in Tabellenform
564     self.logger.error(df.to_string(index=False))
565
566   def print_end_summary(self):
567     try:
568       # set pd max float precision of self.fl_precision
569       with pd.option_context('display.float_format', f'{{:.{self.
    fl_precision}f}}'.format):
570         # pd.options.display.float_format = f'{{:.{self.
    fl_precision}f}}'.format
571         self.logger.info("#"*50)
572         #self.logger.debug(json.dumps(self.fold_history))
573
574         table = self.get_fold_averages_table(show_training=False)
575         self.logger.error("Averages over all folds, for every
    epoch:")
576         self.logger.error(tabulate.tabulate(table, headers='keys',
     tablefmt='grid'))
577         self.logger.info("#"*50)
578         self.print_all_metrics()
579     except Exception as e:
```

```
580         self.logger.error(f"Error printing metrics: {e}")
581       ########################
582
583     def load_state_from_json(self, path):
584       with open(path, 'r') as f:
585         self.fold_history = json.load(f)
586
```

Quellcode B.0.6: Metrics Tracker

Literatur

Abduh, Z., Nehary, E. A., Abdel Wahed, M., & Kadah, Y. M. (2020). Classification of Heart Sounds Using Fractional Fourier Transform Based Mel-Frequency Spectral Coefficients and Traditional Classifiers. *Biomedical Signal Processing and Control*, *57*, 101788. https://doi.org/10.1016/j.bspc.2019.101788

Abu-El-Haija, S., Kothari, N., Lee, J., Natsev, P., Toderici, G., Varadarajan, B., & Vijayanarasimhan, S. (2016). YouTube-8M: A Large-Scale Video Classification Benchmark. *Computing Research Repository*, 1–10. https://doi.org/10.48550/arXiv.1609.08675

Almeida, F., & Xexéo, G. (2023, 2. Mai). *Word Embeddings: A Survey*. arXiv: 1901.09069 [cs, stat]. Verfügbar 30. Oktober 2023 unter http://arxiv.org/abs/1901.09069v2

Alzubaidi, L., Zhang, J., Humaidi, A. J., Al-Dujaili, A., Duan, Y., Al-Shamma, O., Santamaría, J., Fadhel, M. A., Al-Amidie, M., & Farhan, L. (2021). Review of Deep Learning: Concepts, CNN Architectures, Challenges, Applications, Future Directions. *Journal of Big Data*, *8*(1), 53. https://doi.org/10.1186/s40537-021-00444-8

Bhatia, N., & Author, C. (2010). Survey of Nearest Neighbor Techniques. *International Journal of Computer Science and Information Security*, *8*(2), 302–305. https://doi.org/10.48550/arXiv.1007.0085

CDC. (2019, 9. Dezember). *Valvular Heart Disease | cdc.gov*. Centers for Disease Control and Prevention. Verfügbar 26. Mai 2023 unter https://www.cdc.gov/heartdisease/valvular_disease.htm

Chen, L., Ren, J., Hao, Y., & Hu, X. (2018). The Diagnosis for the Extrasystole Heart Sound Signals Based on the Deep Learning. *Journal of Medical Imaging*

and Health Informatics, *8*(5), 959–968. https://doi.org/10.1166/jmihi.2018.2394

Chen, W., Sun, Q., Chen, X., Xie, G., Wu, H., & Xu, C. (2021). Deep Learning Methods for Heart Sounds Classification: A Systematic Review. *Entropy*, *23*(6), 1–18. https://doi.org/10.3390/e23060667

Chicco, D. (2017). Ten Quick Tips for Machine Learning in Computational Biology. *BioData Mining*, *10*, 1–35. https://doi.org/10.1186/s13040-017-0155-3

Chicco, D., & Jurman, G. (2023). The Matthews Correlation Coefficient (MCC) Should Replace the ROC AUC as the Standard Metric for Assessing Binary Classification. *BioData Mining*, *16*(1), 1–4. https://doi.org/10.1186/s13040-023-00322-4

Choi, K., Fazekas, G., Sandler, M., & Cho, K. (2018). A Comparison of Audio Signal Preprocessing Methods for Deep Neural Networks on Music Tagging. *2018 26th European Signal Processing Conference (EUSIPCO)*, 1870–1874. https://doi.org/10.23919/EUSIPCO.2018.8553106

Chowdhury, M. E. H., Khandakar, A., Alzoubi, K., Mansoor, S., M. Tahir, A., Reaz, M. B. I., & Al-Emadi, N. (2019). Real-Time Smart-Digital Stethoscope System for Heart Diseases Monitoring. *Sensors*, *19*(12), 1–22. https://doi.org/10.3390/s19122781

Clifford, G., Liu, C., Springer, D., Moody, B., Li, Q., Abad, R., Millet, J., Silva, I., Johnson, A., & Mark, R. (2016). Classification of Normal/Abnormal Heart Sound Recordings: The PhysioNet/Computing in Cardiology Challenge 2016. *2016 Computing in Cardiology Conference (CinC)*, *43*, 609–612. https://doi.org/10.22489/CinC.2016.179-154

Combrisson, E., & Jerbi, K. (2015). Exceeding Chance Level by Chance: The Caveat of Theoretical Chance Levels in Brain Signal Classification and Statistical Assessment of Decoding Accuracy. *Journal of neuroscience methods*, *250*, 126–136. https://doi.org/10.1016/j.jneumeth.2015.01.010

Cover, T., & Hart, P. (1967). Nearest Neighbor Pattern Classification. *IEEE Transactions on Information Theory*, *13*(1), 21–27. https://doi.org/10.1109/TIT.1967.1053964

Deng, M., Meng, T., Cao, J., Wang, S., Zhang, J., & Fan, H. (2020). Heart sound classification based on improved MFCC features and convolutional recurrent neural networks. *Neural Networks*, *130*, 22–32. https://doi.org/10.1016/j.neunet.2020.06.015

Dogan, D., Xie, H., Heittola, T., & Virtanen, T. (2022). Zero-Shot Audio Classification Using Image Embeddings. *2022 30th European Signal Processing Conference (EUSIPCO)*, 1–5. https://doi.org/10.23919/EUSIPCO55093.2022.9909701

Dominguez-Morales, J. P., Jimenez-Fernandez, A. F., Dominguez-Morales, M. J., & Jimenez-Moreno, G. (2018). Deep Neural Networks for the Recognition and Classification of Heart Murmurs Using Neuromorphic Auditory Sensors. *IEEE Transactions on Biomedical Circuits and Systems*, *12*(1), 24–34. https://doi.org/10.1109/TBCAS.2017.2751545

Elfwing, S., Uchibe, E., & Doya, K. (2018). Sigmoid-Weighted Linear Units for Neural Network Function Approximation in Reinforcement Learning. *Neural Networks*, *107*, 3–11. https://doi.org/10.1016/j.neunet.2017.12.012

Gemmeke, J. F., Ellis, D. P. W., Freedman, D., Jansen, A., Lawrence, W., Moore, R. C., Plakal, M., & Ritter, M. (2017). Audio Set: An Ontology and Human-Labeled Dataset for Audio Events. *Proc. IEEE ICASSP 2017*, 776–780. https://doi.org/10.1109/ICASSP.2017.7952261

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press. Verfügbar 7. Dezember 2023 unter https://www.deeplearningbook.org/

Hajian-Tilaki, K. (2013). Receiver Operating Characteristic (ROC) Curve Analysis for Medical Diagnostic Test Evaluation. *Caspian Journal of Internal Medicine*, *4*(2), 627–635. Verfügbar 18. November 2023 unter https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3755824/

Hershey, S., Chaudhuri, S., Ellis, D. P. W., Gemmeke, J. F., Jansen, A., Moore, C., Plakal, M., Platt, D., Saurous, R. A., Seybold, B., Slaney, M., Weiss, R., & Wilson, K. (2017). CNN Architectures for Large-Scale Audio Classification. *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, *42*, 131–135. https://doi.org/10.1109/ICASSP.2017.7952132

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., & Adam, H. (2017, 16. April). *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. arXiv: 1704.04861 [cs]. Verfügbar 30. Oktober 2023 unter http://arxiv.org/abs/1704.04861

Ioffe, S., & Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, 448–456. Verfügbar 14. Dezember 2023 unter https://arxiv.org/abs/1502.03167v3

Jepkoech, J., Mugo, D. M., Kenduiywo, B. K., & Too, E. C. (2021). The Effect of Adaptive Learning Rate on the Accuracy of Neural Networks. *International Journal of Advanced Computer Science and Applications*, *12*(8), 1–16. https://doi.org/10.14569/IJACSA.2021.0120885

Khan, K. N., Khan, F. A., Abid, A., Olmez, T., Dokur, Z., Khandakar, A., Chowdhury, M. E. H., & Khan, M. S. (2021). Deep learning based classification of unsegmented phonocardiogram spectrograms leveraging transfer learning. *Physiological Measurement*, *42*(9), 1–22. https://doi.org/10.1088/1361-6579/ac1d59

Kingma, D. P., & Ba, J. (2015). Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations*, *1*, 1–15. Verfügbar 8. Dezember 2023 unter http://arxiv.org/abs/1412.6980v9

Koh, E., & Dubnov, S. (2021, 13. April). *Comparison and Analysis of Deep Audio Embeddings for Music Emotion Recognition*. arXiv: 2104.06517 [cs, eess]. Verfügbar 30. Oktober 2023 unter http://arxiv.org/abs/2104.06517

Latif, S., Usman, M., Rana, R., & Qadir, J. (2018). Phonocardiographic Sensing Using Deep Learning for Abnormal Heartbeat Detection. *IEEE Sensors*, *18*(22), 9393–9400. https://doi.org/10.1109/JSEN.2018.2870759

Liu, C., Springer, D., Li, Q., Moody, B., Juan, R. A., Chorro, F. J., Castells, F., Roig, J. M., Silva, I., Johnson, A. E. W., Syed, Z., Schmidt, S. E., Papadaniil, C. D., Hadjileontiadis, L., Naseri, H., Moukadem, A., Dieterlen, A., Brandt, C., Tang, H., … Clifford, G. D. (2016). An open access database for the

evaluation of heart sound algorithms. *Physiological Measurement*, *37*(12), 2181–2213. https://doi.org/10.1088/0967-3334/37/12/2181

Maity, A., Pathak, A., & Saha, G. (2023). Transfer learning based heart valve disease classification from Phonocardiogram signal. *Biomedical Signal Processing and Control*, *85*, 1–17. https://doi.org/10.1016/j.bspc.2023.104805

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. *International Conference on Learning Representations*. Verfügbar 8. Dezember 2023 unter http://arxiv.org/abs/1301.3781v3

Mishra, M., Menon, H., & Mukherjee, A. (2019). Characterization of S_1 and S_2 Heart Sounds Using Stacked Autoencoder and Convolutional Neural Network. *IEEE Transactions on Instrumentation and Measurement*, *68*(9), 3211–3220. https://doi.org/10.1109/TIM.2018.2872387

Mondal, A., Bhattacharyya, P., & Saha, G. (2013). An Automated Tool for Localization of Heart Sound Components S1, S2, S3 and S4 in Pulmonary Sounds Using Hilbert Transform and Heron's Formula. *SpringerPlus*, *2*, 512. https://doi.org/10.1186/2193-1801-2-512

Oh, S. L., Jahmunah, V., Ooi, C. P., Tan, R.-S., Ciaccio, E. J., Yamakawa, T., Tanabe, M., Kobayashi, M., & Rajendra Acharya, U. (2020). Classification of heart sound signals using a novel deep WaveNet model. *Computer Methods and Programs in Biomedicine*, *196*, 105604. https://doi.org/10.1016/j.cmpb.2020.105604

O'Shea, K., & Nash, R. (2015). An Introduction to Convolutional Neural Networks. *arXiv*, 1–10. https://doi.org/arXiv:1511.08458v2

Pati, S., Thakur, S. P., Hamamcı, İ. E., Baid, U., Baheti, B., Bhalerao, M., Güley, O., Mouchtaris, S., Lang, D., Thermos, S., Gotkowski, K., González, C., Grenko, C., Getka, A., Edwards, B., Sheller, M., Wu, J., Karkada, D., Panchumarthy, R., … Bakas, S. (2023). GaNDLF: The generally nuanced deep learning framework for scalable end-to-end clinical workflows. *Communications Engineering*, *2*(1), 1–17. https://doi.org/10.1038/s44172-023-00066-3

Raghu, A., Praveen, D., Peiris, D., Tarassenko, L., & Clifford, G. (2015). Engineering a mobile health tool for resource-poor settings to assess and manage cardiovascular disease risk: SMARThealth study. *BMC medical informatics and decision making*, *15*, 36. https://doi.org/10.1186/s12911-015-0148-4

Sejdic, E., Djurovic, I., & Stankovic, Lj. (2008). Quantitative Performance Analysis of Scalogram as Instantaneous Frequency Estimator. *IEEE Transactions on Signal Processing*, *56*(8), 3837–3845. https://doi.org/10.1109/TSP.2008.924856

Singh, S. A., Meitei, T. G., & Majumder, S. (2020, 1. Januar). 6 - Short PCG classification based on deep learning. In B. Agarwal, V. E. Balas, L. C. Jain, R. C. Poonia & Manisha (Hrsg.), *Deep Learning Techniques for Biomedical and Health Informatics* (S. 141–164). Academic Press. https://doi.org/10.1016/B978-0-12-819061-6.00006-9

Sondermann, M. (2023). *PhysioNet2016 Datensatz - Untersuchung des Datensatzes und dessen Anwendungen* [Masterseminar]. Fachhochschule Dortmund; Fachbereich Informatik.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *The Journal of Machine Learning Research*, *15*(1), 1929–1958. http://jmlr.org/papers/v15/srivastava14a.html

Stoica, P., & Babu, P. (2023, 10. Mai). *Pearson-Matthews Correlation Coefficients for Binary and Multinary Classification and Hypothesis Testing*. https://doi.org/10.48550/arXiv.2305.05974

Thalmayer, A., Zeising, S., Fischer, G., & Kirchner, J. (2020). A Robust and Real-Time Capable Envelope-Based Algorithm for Heart Sound Classification: Validation under Different Physiological Conditions. *Sensors (Basel, Switzerland)*, *20*(4), 972. https://doi.org/10.3390/s20040972

Tsalera, E., Papadakis, A., & Samarakou, M. (2021). Comparison of Pre-Trained CNNs for Audio Classification Using Transfer Learning. *Journal of Sensor and Actuator Networks*, *10*(4), 1–22. https://doi.org/10.3390/jsan10040072

Weiß, C., & Rzany, B. (2013). *Basiswissen Medizinische Statistik* (6. Aufl.). Springer. https://doi.org/10.1007/978-3-642-34261-5

Xie, T., Cheng, X., Liu, M., Deng, J., Wang, X., & Liu, M. (2021). Cut-Thumbnail: A Novel Data Augmentation for Convolutional Neural Network. *Proceedings of the 29th ACM International Conference on Multimedia*, 1627–1635. https://doi.org/10.1145/3474085.3475302

Zhou, G., Chen, Y., & Chien, C. (2022). On the Analysis of Data Augmentation Methods for Spectral Imaged Based Heart Sound Classification Using Convolutional Neural Networks. *BMC Medical Informatics and Decision Making*, *22*(1), 1–21. https://doi.org/10.1186/s12911-022-01942-2

Zhu, Q. (2020). On the performance of Matthews correlation coefficient (MCC) for imbalanced dataset. *Pattern Recognition Letters*, *136*, 71–80. https://doi.org/10.1016/j.patrec.2020.03.030

Zoghbi, W. A., Adams, D., Bonow, R. O., Enriquez-Sarano, M., Foster, E., Grayburn, P. A., Hahn, R. T., Han, Y., Hung, J., Lang, R. M., Little, S. H., Shah, D. J., Shernan, S., Thavendiranathan, P., Thomas, J. D., & Weissman, N. J. (2017). Recommendations for Noninvasive Evaluation of Native Valvular Regurgitation. *Journal of the American Society of Echocardiography*, *30*(4), 303–371. https://doi.org/10.1016/j.echo.2017.01.007

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig angefertigt und mich keiner fremden Hilfe bedient sowie keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen, die wörtlich oder sinngemäß veröffentlichten oder nicht veröffentlichten Schriften und anderen Quellen entnommen sind, habe ich als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Erklärung zu eingesetzten Hilfsmitteln

1. Korrekturservice der Fachhochschule bzw. des Fachbereichs genutzt: Ja ☐ Nein ☑
2. Einsatz eines externen (kommerziellen) Korrekturservice: Ja ☐ Nein ☑
   wenn ja, welcher
3. Folgende Personen haben die Arbeit zusätzlich Korrektur gelesen:
   Niklas Tschorn
   Malgorzata Sondermann
   Malgorzata Sondermann
4. Nutzung von Sprachmodellen für die Texterstellung (z.B. ChatGPT), Ja ☑ Nein ☐
   wenn ja, welche und in welchen Abschnitten:
   Alle Abschnitte, zur Ausformulierung eigener Gedanken, LaTex Erzeugung und Rechtschreibkontrolle
5. Sprachübersetzungstools (z.B. Google Übersetzer, DeepL), Ja ☐ Nein ☑
   wenn ja, welche und in welchen Abschnitten:
6. Einsatz von Software zur Sprachkorrektur (z.B. Grammarly), Ja ☐ Nein ☑
   wenn ja, welche und in welchen Abschnitten:
7. Einsatz anderer Hilfsmittel:
   Zotero, TexStudio, GitKraken

Ich nehme zur Kenntnis, dass meine Thesis mittels Software zur Plagiatserkennung überprüft werden kann.
Ich bestätige, dass obige Aussagen vollständig und nach bestem Wissen ausgefüllt wurden.

Dortmund, den 20. Dezember 2023

Unterschrift der verfassenden Person