KLATSCHERKENNUNG

Dokumentation für den Projektauftrag von glaL4

Abstract

We present a lightweight, real-time clap detection system running on a Raspberry Pi. A simple 1D convolutional neural network classifies audio data from a USB microphone, achieving somewhat robust performance despite constrained computational resources.

Antony Stang, Fiona Eberhard

Antony.Stang@students.fhnw.ch

Contents

I. Einleitung

Motivation und Kontext

In vielen alltäglichen Situationen kann es unpraktisch sein, den PC manuell einzuschalten, wie zum Beispiel, wenn man auf dem Bett liegt und keine Lust hat aufzustehen. Eine einfache, kontaktlose Möglichkeit zur Aktivierung des Rechners wäre daher von großem Komfortgewinn.

Im Rahmen dieses Projekts wird dieses Problem mithilfe eines akustischen Triggers gelöst: Ein energieeffizienter Raspberry Pi wird so eingesetzt, dass er auf Klatschgeräusche reagiert und bei deren Erkennung ein Wake-on-LAN Signal an den PC sendet. Damit lässt sich der Rechner bequem per Klatschen einschalten – ohne physische Anstrengung.

Zielsetzung und Vorgehen

Ziel dieses Projekts ist es, im Rahmen des glaL4-Projekts ein neuronales Netz so zu trainieren, dass es Klatschgeräusche zuverlässig erkennen kann. Dabei soll das Modell leichtgewichtig genug bleiben, um auf einem ressourcenschwachen Raspberry Pi in Echtzeit zu laufen und ein Wake-on-LAN Signal auszulösen.

Zur Erreichung dieses Ziels werden drei unterschiedliche CNN Ansätze getestet und verglichen:

1. Basisnetz
   Ein klassisches 1D-Convolutional Neural Network mit zwei Faltungsschichten, das eine große Menge extrahierter Merkmale an einen einfachen Klassifikationsteil mit zwei Fully-Connected Schichten übergibt.

2. Frequenzbasiertes Netz
   In einem zweiten Ansatz wird das Audiosignal vor dem Netz mit einer schnellen Fourier-Transformation (FFT) in den Frequenzbereich überführt. Anschließend erfolgt die Verarbeitung ähnlich wie im ersten Modell.

3. Merkmal-reduziertes Netz mit verstärktem FC-Anteil
   Ein alternativer Ansatz, bei dem die convolution Komponente bewusst klein gehalten wird. Stattdessen werden mehr Ressourcen in den Fully-Connected-Bereich investiert, um trotz reduzierter Merkmalvielfalt eine robuste Entscheidung zu ermöglichen.

Die genaue Anzahl an Hidden Layers, Neuronen und Trainings-Epochen wird im Verlauf der Entwicklung flexibel angepasst und iterativ optimiert, um eine gute Balance zwischen Genauigkeit und Effizienz zu erreichen.

II. Projektaufbau und Hardware-Setup

Hardware-Beschreibung

Bauteil	Beschreibung
Raspberry Pi 5	Leistungseffizienter Rechner welcher Passiv durchlaufen kann
USB Mikrofon	44,1kHz Sampling rate. Kostengünstig in einer Schublade gefunden, aufgrund der mysteriösen Herkunft gibt es keine weiteren Daten.
LAN Kabel	0,5 m Ethernetkabel zur Verbindung des Raspberrys mit dem PC

Folgendes Hardware-Setup wird empfohlen:

Ein Raspberry Pi 5 ist über USB mit einem Mikrofon verbunden, das in einer Position platziert wird, um den gesamten Raum akustisch abzudecken. Für das Wake-on-LAN Signal wird der Pi per LAN-Kabel direkt mit dem Zielgerät verbunden. Die Stromversorgung erfolgt über das offizielle Raspberry-Pi-Netzteil.

Software-Umgebung

Bibliothek	Beschreibung	Typ
torch	PyTorch – verwendetes Framework für das Machine Learning	extern
torchaudio	Audioerweiterung für PyTorch	extern
os	Erlaubt zugriff auf das Dateisystem zum Importieren der Daten	buildin

Folgendes Software-Setup wird empfohlen:

Auf dem Raspberry Pi wird zunächst Python 3 installiert, gefolgt von der Erstellung einer virtuellen Umgebung via „python3 -m venv venv“. Innerhalb dieser Umgebung lässt sich per „pip install torch torchaudio“ das benötigte PyTorch-Audio-Framework einrichten. Anschließend wird das gewählte trainierte Modell aus dem Git-Projekt „Antony-St/glaL4“ heruntergeladen und lokal ausgeführt.

(die Wake-on-LAN Funktion ist noch in Arbeit)

III. Datenerfassung und Vorverarbeitung

Datensammlung

Die Datenaufnahme erfolgte in Form mehrerer 60-sekündiger Audioaufnahmen, die anschließend manuell bearbeitet wurden. Zur Erzeugung unterschiedlicher Hintergrundgeräusche wurden lautstarke Musikvideos sowie normal lautes Audio von Serien genutzt.

Im Nachgang wurden die jeweiligen Klangereignisse in kurze Abschnitte geschnitten: Klatscher mit einer Länge von 50 bis 150 ms sowie Nicht-Klatscher (Hintergrundgeräusche) mit etwa 750 ms Länge. Insgesamt stehen damit 51 Klatsch-Samples und 80 Nicht-Klatsch-Samples für das Training und die Tests zur Verfügung.

Vorverarbeitungsschritte

Die Audioverarbeitung erfolgt in drei Teilschritten: Zunächst wird das Stereosignal in ein Monosignal umgewandelt.

Falls Audiodaten einer anderen Quelle verwendet werden, welche eine andere Samplingrate hat, werden diese noch auf 44,1kHz transformiert. (Diese Funktion findet im Rahmen von glaL4 keine Verwendung, wurde aber bereits für die weitere Entwickelung erstellt)

Anschließend werden alle Dateien auf den Bereich [-1, 1] normalisiert und auf exakt 4410 Samples (entsprechend 100 ms bei 44,1 kHz) gekürzt oder mit Nullen aufgefüllt.

Bei dem Frequenzbasiertem Netz wird hier ebenfalls eine FFT durchgeführt, um Frequenzinformationen zu gewinnen.

IV. Modellentwicklung

Architekturüberblick

1. Versuch 1: 2 Conv-Layer
   4410 Features werden in der ersten Faltung mit MaxPool reduziert auf 1102. In der zweiten Faltung werden die Features mit MaxPool auf 275 reduziert. Nach dem Flattening werden die Features durch die erste Fully Connected Layer auf 128 und schliesslich durch die zweite Fully Connected Layer auf 2 Features reduziert (clap und noclap)
2. Versuch 2: FFT
   Das ursprüngliche Signal wird mit einer FFT in ihre Frequenzkomponenten zerlegt, wobei die Anzahl Frequenzkomponenten meist deutlich kleiner ist als die Anzahl Samples im Zeitbereich. Die Frequenzkomponenten werden anschliessend gleich wie in Versuch 1 verarbeitet.
3. Versuch 3: 1 Conv-Layer
   Zuerst gehen die Daten durch einen Convolution Layer mit 32 Kanäle. Danach wird in jedem Kanal der Durchschnitt über die Zeit berechnet. Die übrig gebliebenen 32 Features werden durch 3 Fully Connected Layers geleitet, welche insgesamt die Features auf die 2 wichtigsten (clap und noclap) reduzieren.

Die tabellarische Darstellung der Feature-Reduktion des 1. Versuch soll zur allgemeinen Veranschaulichung dienen.

Schritt	Größe der Daten	Reduktionsmethode
Eingabe	(1, 4410)	-
Conv1	(16, 1102)	Max-Pooling (Faktor 4)
Conv2	(32, 275)	Max-Pooling (Faktor 4)
Flattening	(8800)	Lineare Umwandlung
FC1	(128)	Lineare Umwandlung
FC2	(2)	Klassifikation

Training

Das Training wird in Epochen durchgeführt. In jeder Epoche wird das neuronale Netz Batchweise trainiert. Jeder Batch wird in das Netz gespiessen, daraus wird der Gradient berechnet und die Gewichte angepasst. Für jeden Batch wird der Loss zwischen Vorhersage und echtem Label aufaddiert und es wird, falls korrekt klassifiziert, zu einem Counter dazugezählt. Sobald die Epoche abgeschlossen ist, wird der Durchschnittliche Loss und Accuracy berechnet.

Daten werden in Train-/Validation Daten mit 80/20 aufgeteilt. Die Loss Function CrossEntropyLoss wird von der PyTorch Library bereitgestellt. Der Vorteil ist, dass die Funktion das Modell bestraft, wenn es überzeugt aber falsch und unsicher aber richtig ist. Die Optimizer Funktion Adam (Adaptive Moment Estimation) wurde gewählt, da diese oft „out of the box“ ohne Finetuning der Hyperparameter funktioniert. Das Training wird anhand vergangener Gradienten beschleunigt und passt die Lernrate für jede Parameterdimension adaptiv an.

Das Training wurde auf dem Raspberry Pi durchgeführt, da die Zeit pro Trainingsschritt nicht nennenswert gross war (im Bereich 1 – 3 Sekunden).

Um das Training zu steuern, können Learning Rate, Anzahl Epochen angepasst werden und es kann ein explizites Abbruchkriterium bestimmt werden. Aufgrund der geringen Datenmenge wird val_acc >= 1 verwendet, dies sollte für größere Datenmengen auf >= 0,98 umgestellt werden.

V. Ergebnisse und Auswertung

Trainingsergebnisse

In folgenden Screenshots sind die Trainingsverläufe der drei Versuche zu sehen. Für jede Epoche wurde der Train Loss, die Train Accuracy, die Validation Loss, Validation Accuracy, False Positives und False Negatives an der Konsole ausgegeben.

Die Training/Validation Loss gibt an, wie gut oder schlecht das Netz während dem Training/Validation abschneidet. Die Training/Validation Accuracy berechnet den Anteil richtig klassifizierter Training/Validation Daten.

```
(venv) pi@raspi:~/clap/src $ python3 ML_CNN_wav_faltung.py
Device: cpu
Epoch [1/100] Train Loss: 0.7169, Train Acc: 0.59 | Val Loss: 0.5583, Val Acc: 0.63 | FP: 0/27, FN: 10/27
Epoch [2/100] Train Loss: 0.5050, Train Acc: 0.61 | Val Loss: 0.3798, Val Acc: 0.63 | FP: 0/27, FN: 10/27
Epoch [3/100] Train Loss: 0.3843, Train Acc: 0.76 | Val Loss: 0.3195, Val Acc: 0.81 | FP: 0/27, FN: 5/27
Epoch [4/100] Train Loss: 0.3406, Train Acc: 0.86 | Val Loss: 0.3076, Val Acc: 0.96 | FP: 1/27, FN: 0/27
Epoch [5/100] Train Loss: 0.3154, Train Acc: 0.93 | Val Loss: 0.2820, Val Acc: 0.93 | FP: 0/27, FN: 2/27
Epoch [6/100] Train Loss: 0.2956, Train Acc: 0.94 | Val Loss: 0.2862, Val Acc: 1.00 | FP: 0/27, FN: 0/27
Val Acc 100% erreicht. Breche Training ab...
Training abgeschlossen und Modell gespeichert.
```

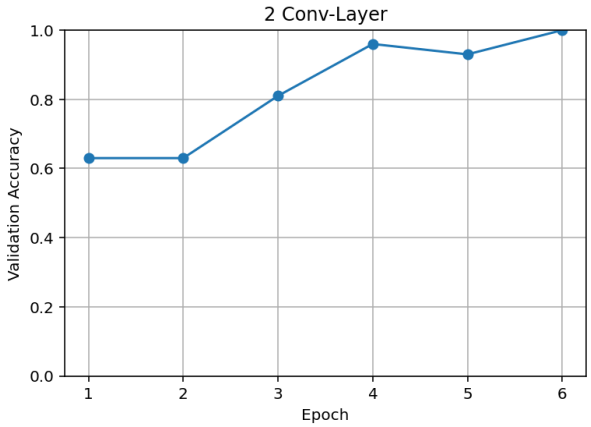
```
(venv) pi@raspi:~/clap/src $ python3 ML_CNN_wav_FF_faltung.py
Device: cpu
Epoch [1/100] Train Loss: 0.2513, Train Acc: 0.88 | Val Loss: 0.8422, Val Acc: 0.81 | FP: 0/27, FN: 5/27
Epoch [2/100] Train Loss: 0.3253, Train Acc: 0.90 | Val Loss: 0.3443, Val Acc: 0.93 | FP: 2/27, FN: 0/27
Epoch [3/100] Train Loss: 0.1040, Train Acc: 0.96 | Val Loss: 0.4726, Val Acc: 0.89 | FP: 3/27, FN: 0/27
Epoch [4/100] Train Loss: 0.0232, Train Acc: 0.99 | Val Loss: 0.0809, Val Acc: 1.00 | FP: 0/27, FN: 0/27
Val Acc 100% erreicht. Breche Training ab...
Training abgeschlossen und Modell gespeichert.
```

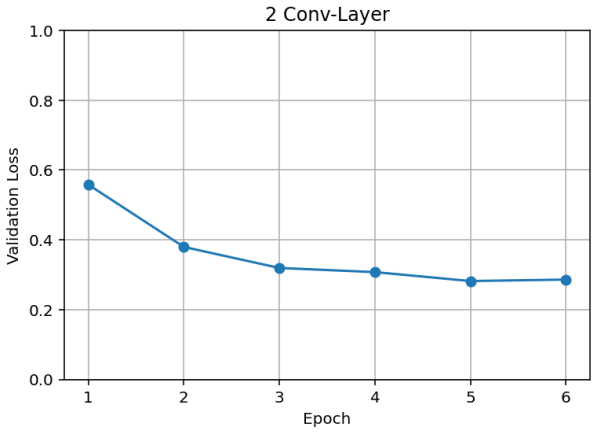
```
(venv) pi@raspi:~/clap/src $ python3 ML_CNN_wav_min_faltung.py
Device: cpu
Epoch [1/100] Train Loss: 0.6662, Train Acc: 0.64 | Val Loss: 0.7740, Val Acc: 0.48 | FP: 0/27, FN: 14/27
Epoch [2/100] Train Loss: 0.6352, Train Acc: 0.64 | Val Loss: 0.7212, Val Acc: 0.48 | FP: 0/27, FN: 14/27
Epoch [3/100] Train Loss: 0.6176, Train Acc: 0.64 | Val Loss: 0.6826, Val Acc: 0.48 | FP: 0/27, FN: 14/27
Epoch [4/100] Train Loss: 0.5917, Train Acc: 0.64 | Val Loss: 0.6718, Val Acc: 0.48 | FP: 0/27, FN: 14/27
Epoch [5/100] Train Loss: 0.5440, Train Acc: 0.64 | Val Loss: 0.5574, Val Acc: 0.48 | FP: 0/27, FN: 14/27
Epoch [6/100] Train Loss: 0.4684, Train Acc: 0.66 | Val Loss: 0.4622, Val Acc: 0.59 | FP: 0/27, FN: 11/27
Epoch [7/100] Train Loss: 0.3893, Train Acc: 0.87 | Val Loss: 0.3960, Val Acc: 0.78 | FP: 0/27, FN: 6/27
Epoch [8/100] Train Loss: 0.3347, Train Acc: 0.87 | Val Loss: 0.3727, Val Acc: 0.85 | FP: 0/27, FN: 4/27
Epoch [9/100] Train Loss: 0.3178, Train Acc: 0.90 | Val Loss: 0.3042, Val Acc: 0.96 | FP: 0/27, FN: 1/27
Epoch [10/100] Train Loss: 0.2883, Train Acc: 0.88 | Val Loss: 0.2552, Val Acc: 0.96 | FP: 0/27, FN: 1/27
Epoch [11/100] Train Loss: 0.2723, Train Acc: 0.89 | Val Loss: 0.2250, Val Acc: 0.96 | FP: 0/27, FN: 1/27
Epoch [12/100] Train Loss: 0.2648, Train Acc: 0.88 | Val Loss: 0.2354, Val Acc: 0.96 | FP: 0/27, FN: 1/27
Epoch [13/100] Train Loss: 0.2500, Train Acc: 0.90 | Val Loss: 0.2787, Val Acc: 0.96 | FP: 0/27, FN: 1/27
Epoch [14/100] Train Loss: 0.2497, Train Acc: 0.89 | Val Loss: 0.2084, Val Acc: 0.96 | FP: 0/27, FN: 1/27
Epoch [15/100] Train Loss: 0.2432, Train Acc: 0.91 | Val Loss: 0.1993, Val Acc: 0.96 | FP: 0/27, FN: 1/27
Epoch [16/100] Train Loss: 0.2339, Train Acc: 0.91 | Val Loss: 0.2139, Val Acc: 0.96 | FP: 0/27, FN: 1/27
Epoch [17/100] Train Loss: 0.2300, Train Acc: 0.91 | Val Loss: 0.1885, Val Acc: 0.96 | FP: 0/27, FN: 1/27
Epoch [18/100] Train Loss: 0.2356, Train Acc: 0.90 | Val Loss: 0.2777, Val Acc: 0.96 | FP: 0/27, FN: 1/27
Epoch [19/100] Train Loss: 0.2265, Train Acc: 0.91 | Val Loss: 0.1501, Val Acc: 0.96 | FP: 0/27, FN: 1/27
Epoch [20/100] Train Loss: 0.2204, Train Acc: 0.91 | Val Loss: 0.2384, Val Acc: 0.96 | FP: 0/27, FN: 1/27
Epoch [21/100] Train Loss: 0.2167, Train Acc: 0.91 | Val Loss: 0.1296, Val Acc: 0.96 | FP: 0/27, FN: 1/27
Epoch [22/100] Train Loss: 0.2064, Train Acc: 0.92 | Val Loss: 0.2531, Val Acc: 0.96 | FP: 0/27, FN: 1/27
Epoch [23/100] Train Loss: 0.2119, Train Acc: 0.91 | Val Loss: 0.1444, Val Acc: 0.96 | FP: 0/27, FN: 1/27
Epoch [24/100] Train Loss: 0.2336, Train Acc: 0.91 | Val Loss: 0.1281, Val Acc: 0.96 | FP: 0/27, FN: 1/27
Epoch [25/100] Train Loss: 0.1775, Train Acc: 0.94 | Val Loss: 0.3761, Val Acc: 0.78 | FP: 0/27, FN: 6/27
Epoch [26/100] Train Loss: 0.2012, Train Acc: 0.93 | Val Loss: 0.1187, Val Acc: 0.96 | FP: 0/27, FN: 1/27
Epoch [27/100] Train Loss: 0.1917, Train Acc: 0.93 | Val Loss: 0.1807, Val Acc: 0.96 | FP: 0/27, FN: 1/27
Epoch [28/100] Train Loss: 0.1754, Train Acc: 0.93 | Val Loss: 0.1622, Val Acc: 0.96 | FP: 0/27, FN: 1/27
Epoch [29/100] Train Loss: 0.1680, Train Acc: 0.95 | Val Loss: 0.1529, Val Acc: 0.96 | FP: 0/27, FN: 1/27
Epoch [30/100] Train Loss: 0.1594, Train Acc: 0.95 | Val Loss: 0.1530, Val Acc: 0.96 | FP: 0/27, FN: 1/27
Epoch [31/100] Train Loss: 0.1572, Train Acc: 0.94 | Val Loss: 0.1531, Val Acc: 0.96 | FP: 0/27, FN: 1/27
Epoch [32/100] Train Loss: 0.1588, Train Acc: 0.94 | Val Loss: 0.2054, Val Acc: 0.96 | FP: 0/27, FN: 1/27
Epoch [33/100] Train Loss: 0.1460, Train Acc: 0.95 | Val Loss: 0.1253, Val Acc: 0.96 | FP: 0/27, FN: 1/27
Epoch [34/100] Train Loss: 0.1413, Train Acc: 0.95 | Val Loss: 0.1695, Val Acc: 0.96 | FP: 0/27, FN: 1/27
Epoch [35/100] Train Loss: 0.1429, Train Acc: 0.92 | Val Loss: 0.1132, Val Acc: 0.96 | FP: 1/27, FN: 0/27
Epoch [36/100] Train Loss: 0.2241, Train Acc: 0.92 | Val Loss: 0.1499, Val Acc: 0.96 | FP: 0/27, FN: 1/27
Epoch [37/100] Train Loss: 0.2182, Train Acc: 0.91 | Val Loss: 0.1904, Val Acc: 0.96 | FP: 0/27, FN: 1/27
Epoch [38/100] Train Loss: 0.1655, Train Acc: 0.93 | Val Loss: 0.0613, Val Acc: 1.00 | FP: 0/27, FN: 0/27
Val Acc 100% erreicht. Breche Training ab...
Training abgeschlossen und Modell gespeichert.
```

Beim letzten Versuch mit einer Convolution Layer dauert das Training länger als bei den anderen, allerdings ist diese Methode auch deutlich weniger anfällig auf False Positives. Die Accuracy und Validation Accuracy sind bei allen Netzen sehr ähnlich.

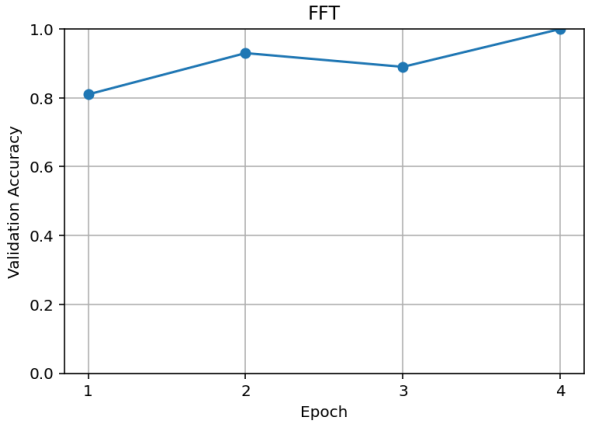
Fehlerraten und Beispielanalysen

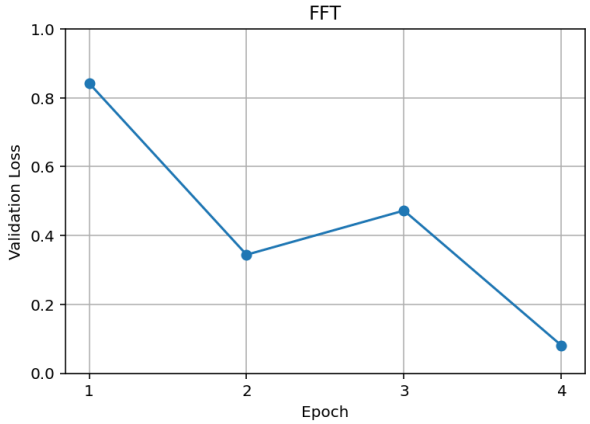
1. Versuch 1: 2 Conv Layers
   Validation Accuracy ist von Anfang an hoch aber Validation Loss verändert sich ab Epoche 3 nur noch geringfügig.



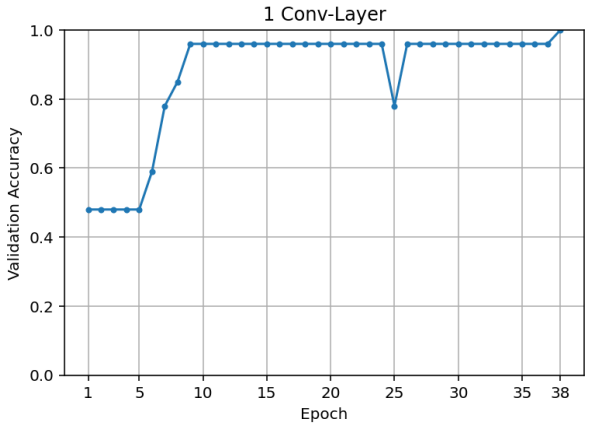


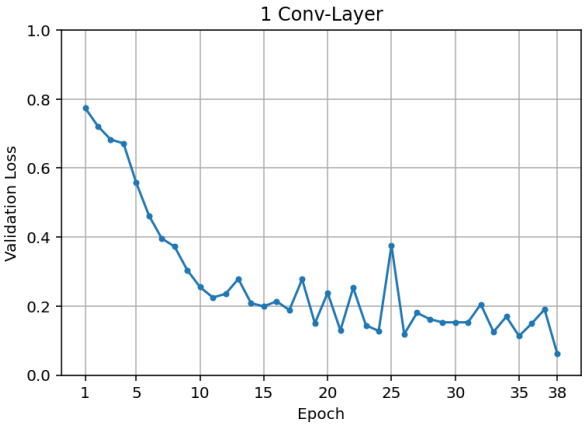
1. <u>Versuch 2: FFT</u>
   Validation Accuracy ist von Anfang an hoch und Validation Loss fällt sehr schnell auf unter 0,2.





2. <u>Versuch 3: 1 Conv-Layer</u>
   Validation Accuracy verändert sich ab Epoche 10 fast nicht mehr und Validation Loss fällt am Anfang sehr steil und schwankt um die 0,2. Bemerkenswert ist ein plötzlicher Einbruch in der Validation Accuracy nach der Epoche 25. Eine definitive Erklärung gibt es hierfür nicht.





Echtzeitverhalten

Es wurden bis jetzt noch keine Langzeitanwendung umgesetzt, aber bei kurzen Tests des trainierten Netzes kann man bereits eine signifikante Menge False Positives feststellen.

VI. Diskussion

Interpretation der Resultate

Die ersten Beiden Versuche beginnen mit einer hohen Validation Accuracy. Bei der Version mit zwei Convolution Layer verändert sich die Validation Loss nach drei Epochen nur noch marginal, wobei sie bei der FFT Variante ziemlich schnell bis auf fast 0 geht. Alle Netze sind mit wenigen Epochen sehr akkurat geworden. Versuch 3 mit nur einer Convolution Layer konnte dasselbe Resultat erzielen, einfach mit mehr Trainingsepochen. Unsere Netze sind jedoch so klein, dass der grössere Trainingsaufwand beim Versuch 3 nicht ins Gewicht fällt.

Variante 3 ist für die finale Anwendung am besten geeignet, weil die Anzahl der false-positives hier am geringsten sind. Die FFT Variante kommt nicht infrage, da der zusätzliche Rechenaufwand nicht durch ein besseres Netz kompensiert wird.

Jedoch leiden alle Netze noch unter dem schlechten Datensatz, weshalb das Echtzeitverhalten bei keinem zufriedenstellend ist.

Limitationen und mögliche Verbesserungen

Obwohl die grundlegende Klatscherkennung bereits funktioniert, zeigen sich noch deutliche Schwächen im Hinblick auf die Daten.

Zum einen ist die Anzahl der Audiosamples (51 Klatscher, 80 Nicht-Klatscher) zu gering, um ein robustes Training zu gewährleisten. Zum anderen spiegelt das vorhandene Datenmaterial nicht alle realen Situationen wider: Es fehlen Extremfälle wie Bohren, Silvesterknaller oder lautes Lachen – Geräusche, die in der Praxis durchaus ähnlich erscheinen können, wie ein Klatscher.

Neben der aufwändigen Lösungsmöglichkeit mehr Audiodaten zu erzeugen sind andere Wege möglich:

- Es können Daten aus externen Datenbänken angepasst werden und für das Training verwendet werden.
- Die aktuellen Daten können mit künstlichen Zusätzen wie Rauschen oder Veränderung des Lautstärkepegels angepasst werden, um mehr Nutzen zu gewinnen.

VII. Fazit und Ausblick

Zusammenfassung

Das Ziel, ein funktionierendes Klatscherkennungssystem zu entwickeln, wurde im Wesentlichen erreicht. Um jedoch ein wirklich robustes System zu erhalten, ist eine größere und diversere Sammlung von Trainingsaufnahmen notwendig. Die bisherigen Ergebnisse zeigen, dass der gewählte Ansatz grundsätzlich vielversprechend ist, jedoch noch mehr Daten erfordert.

Wesentliche Erkenntnisse:

- **Reduzierte Feature-Menge vs. erweiterte Fully-Connected-Schichten**
  Bis zu einem gewissen Grad lässt sich ein kleinerer Convolution Teil durch mehr und komplexere Fully-Connected-Schichten kompensieren, ohne die Erkennungsleistung zu verschlechtern. Jedoch gibt es einen Punkt, ab welchem die FC-Schichten nicht mehr für die geringen Eingangsschichten kompensieren können.
- **FFT-Vorverarbeitung**
  Die Anwendung einer Fourier-Transformation hat das Training beschleunigt, führte jedoch in der fertigen Anwendung zu keinem entscheidenden Qualitätszuwachs. Der hinzugefügte Verarbeitungsschritt der FFT führt ebenfalls zu einer komplexeren Vorverarbeitung während der Verwendung des Netzes. Die Verwendung der FFT ist für diesen Anwendungsfall nicht von Nutzen.

Zukünftige Entwicklung

In einem nächsten Schritt soll eine Wake-on-LAN Funktion implementiert werden, die ein bereits trainiertes Netz zur Erkennung von Klatschsignalen heranzieht. Um die Präzision weiter zu erhöhen, wird außerdem erwogen, bestärkendes Lernen (Reinforcement Learning) in diesen Prozess einzubeziehen. Darüber hinaus sind keine weiteren Erweiterungen oder Anpassungen des Projekts geplant.