Unicornn: Eine Unity Sprachsteuerung

Alexander Pech (736825)
alexander.pech@study.hs-duesseldorf.de
Hochschule Düsseldorf

Lara Bertram (736057)
lara.bertram@study.hs-duesseldorf.de
Hochschule Düsseldorf

Alexander Mikulaschek (736306)
alexander.mikulaschek@study.hs-duesseldorf.de
Hochschule Düsseldorf

Philipp Jonas Knohl (710050)
philipp.jonas.knohl@study.hs-duesseldorf.de
Hochschule Düsseldorf

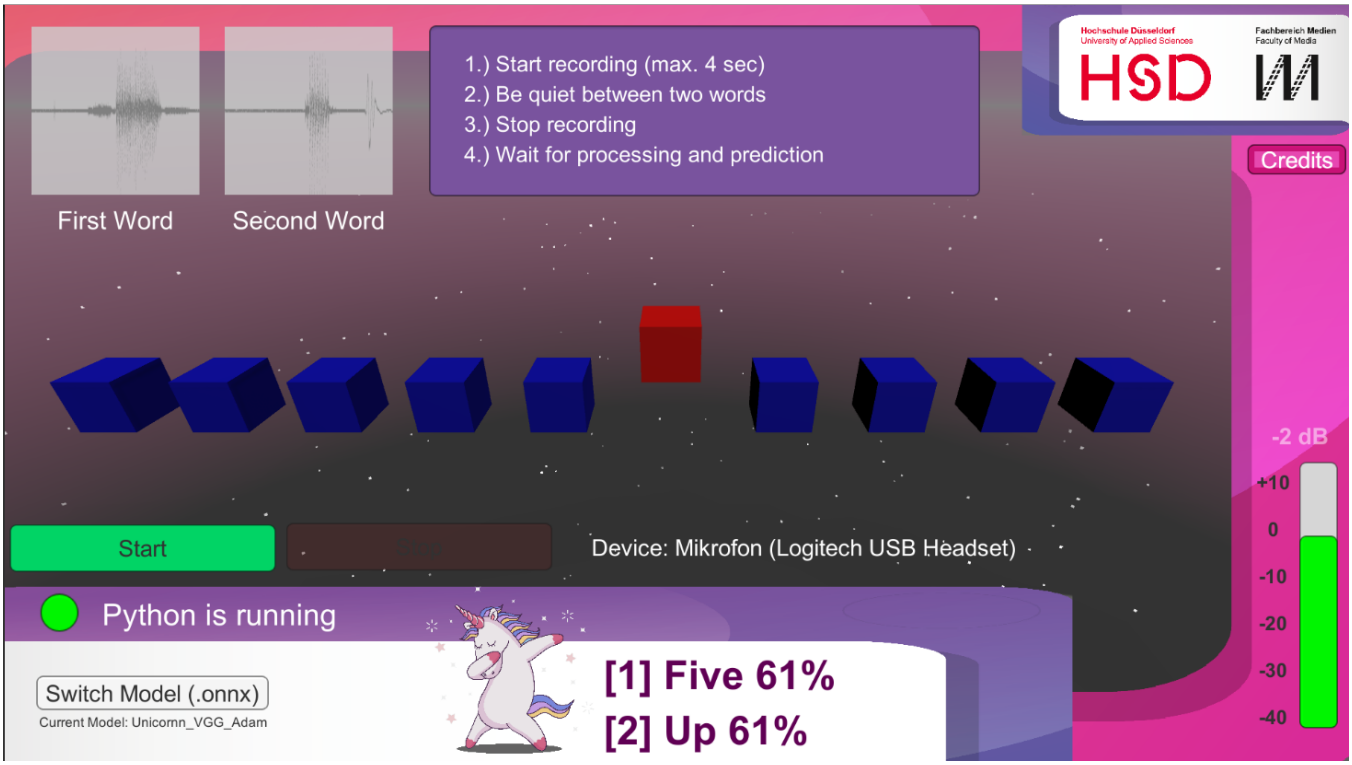

Abbildung 1: Screenshot des Interface der Unityanwendung.

Zusammenfassung

Wir präsentieren eine einfache Sprachsteuerung für die Game Engine Unity 3D. Die Sprachsteuerung erkennt einzelne Wörter und klassifiziert diese mittels eines neuronalen Netzes. Bestimmte Kombinationen bei der Klassifizierung erlauben eine direkte Zuordnung zu Unity-Befehlen. Ein Beispiel wäre der Sprachbefehl „One“ + „Forward“, welcher in Unity „transform.position += Vector3.forward“ entspricht.

Bei dem neuronalen Netz handelt es sich um ein Convolutional Neural Network (kurz: CNN), da diese sich gut für Klassifizierungsaufgaben eignen. Teil dieser Arbeit ist dabei auch der Performanzvergleich zwischen VGG19 [4] und ResNet [2] sowie die Verwendung des SGD- und Adam-Optimizers beim Trainingsprozess. Zur Demonstration des Projektes wurde eine Demoszene in Unity 3D erstellt, welche den Agenten enthält und es ermöglicht via Spracheingabe Objekt im Raum zu bewegen.

Keywords

datasets, neural networks, speech recognition, Unity, Voice command

Intelligente Systeme:
Alexander Pech (736825), Lara Bertram (736057), Alexander Mikulaschek (736306), and Philipp Jonas Knohl (710050). 2020. Unicornn: Eine Unity Sprachsteuerung.

Deutschland, NRW, Düsseldorf,
2020.

1 Einleitung

1.1 Motivation

Sprachsteuerungen erhalten eine immer größere Relevanz im alltäglichen Leben (vgl. Alexa & Siri). Bislang ist es jedoch so, dass Sprachsteuerungen in den meisten Anwendungsfällen eine Alternative zu herkömmlichen Steuerungsinteraktionen darstellen, die oft als etwas „spielerisch“ wahrgenommen werden. Wir wollen diesen Aspekt aufgreifen und eine Sprachsteuerung in einer Umgebung implementieren, in welcher dieser verstärkt zur Geltung kommt.

1.2 Ziel

Ziel dieses Projektes war es eine Sprachsteuerung für die GameEngine Unity zu entwickeln, mit der man einfache Sprachbefehle einsprechen kann und diese dann korrekt verarbeitet und ausgeführt werden. Unity soll imstande sein Objekte in der Szene zu verändern, während diese sich im Playmode befindet.

2 Environment

Möchte man den Aufbau des Projektes in Agent und Umgebung aufteilen, ist zu beachten, dass es eine Trainings- und eine Anwendungsumgebung gibt. Der Agent wird durch das CNN repräsentiert.

2.1 Trainingsumgebung

Die Trainingsumgebung wurde in einem Jupyter-Notebook mit dem Machine Learning Framework PyTorch geschrieben. Innerhalb der Umgebung mussten zwei grundlegende Aufgaben realisiert werden. Die erste Aufgabe bestand darin, den Datensatz einzulesen, aufzubereiten und die Einträge mit entsprechendem Label für den Trainingsprozess bereit zu stellen. Die zweite Aufgabe war der Trainingsprozess selbst. Details zum Trainingsprozess werden in Kapitel 4 beschrieben.

2.1.1 Trainingsdaten

Die Trainingsdaten stammen aus dem Speech Command Dataset v0.02 von Warden[5]. Dieser Datensatz enthält tausende von Aufnahmen einzelner Worte. Die verschiedenen Worte repräsentieren dabei die Klassen. Für unsere Anwendung sind lediglich die folgenden Worte/Aufnahmen der Klassen relevant:

„one“, „two“, „three“, „four“, „five“, „six“, „seven“, „eight“, „nine“, „forward“, „backward“, „left“, „right“, „up“ und „down“.

Die relevanten Daten bilden unseren Datensatz, welcher mit einem Verhältnis von 80/20 in Trainings- und Testdaten aufgeteilt wird (siehe Abb. 2).

Die Trainingsdaten liegen im .wav Format vor. Für die Eingabe in die Netze werden die Aufnahmen auf eine Dauer von einer Sekunde beschränkt bzw. erweitert (hinzufügen von Stille). Bei einer Samplerate von 16kHz (mono) würde dies einer Eingabe von 16000 Werten in das Netz entsprechen. Um die Input Size zu verringern wird ein Spektrogramm der Aufnahmen erzeugt. Ein weiterer Vorteil der Verwendung von Spektrogrammen ist, dass es sich um eine drei dimensionale Repräsentation handelt (Zeit, Frequenz, Amplitude). Bei der Verwendung von RAW-Audio gäbe es nur die Dimensionen Zeit und Amplitude. Die 3D Repräsentation ermöglicht einen detaillierteren Einblick in die Strukturen innerhalb der Aufnahmen, die vom Netz erlernt werden sollen. Gespeichert werden die Spektrogramme in Form von Graustufen-Bildern (x-Ache: Zeit, y-Achse: Frequenz, Pixelwert: Amplitude). X und Y-Dimensionen werden dabei auf je 32 Werte beschränkt und die Amplitude wird normalisiert.

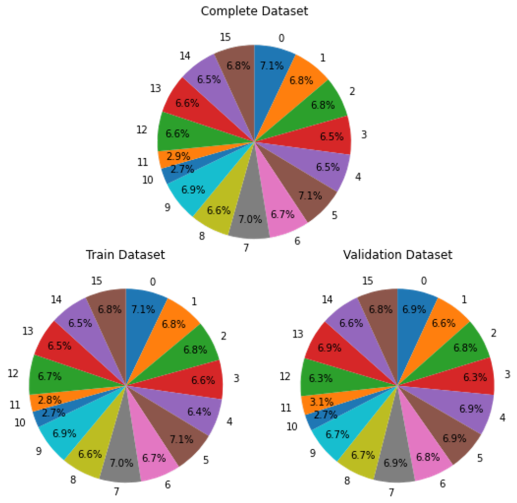

Abbildung 2: Die verschiedenen Datensätze

2.1.2 Mel-Spektrogramm

Ein Mel-Spektrogramm ist eine Repräsentation des Kurzzeitenergiespektrums eines Tons in Abhängigkeit des menschlichen Gehörs (siehe Abb. 3). Damit diese Abhängigkeit simuliert werden kann, wird die Frequenz an die Mel-Skala[1] angepasst (siehe Formel 1). Es simuliert durch Filterbänke wie das menschliche Gehör einen Ton wahrnimmt und hebt somit die wichtigen Frequenzen an und schwächt jene Frequenzen ab, die nicht wahrgenommen werden.

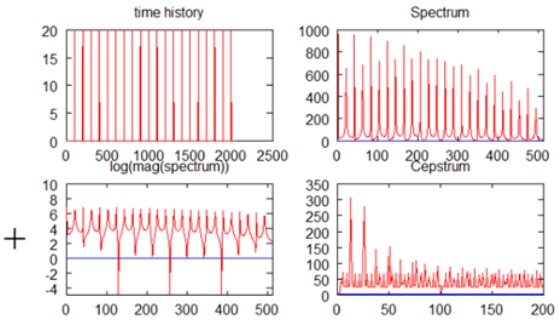

Abbildung 3: Unterschiedliche Darstellungsmöglichkeiten von Audio

$$M(f) = 1125ln(1 + \frac{f}{700}) \quad (1)$$

Beim Cepstrum wird auf der x-Achse die „quefrency“ dargestellt. Die Quefrenz hat als Einheit die Zeit und kann als Maß für die Verschiebung von Mustern im Zeitbereich interpretiert werden.
Das fertige Mel-Spektrogramm kann schließlich auch als eine Art Textur dargestellt werden, siehe Abb.4. Hier beschreibt die Farbe die Intensität, die X-Achse die Quefrenz und die Y-Achse die Frequenzbänder.

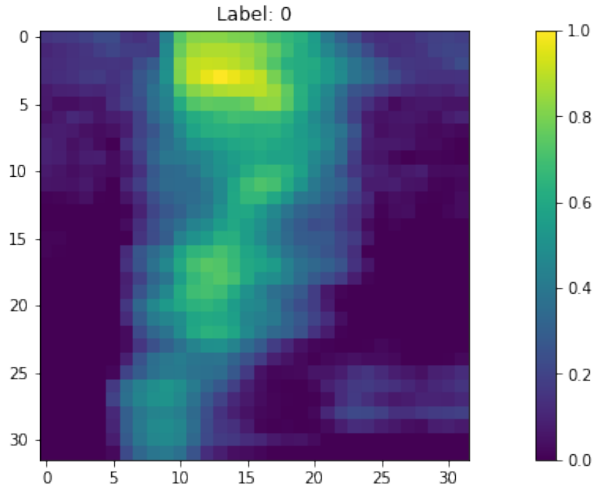

Abbildung 4: Mel-Spektrogramm als Textur

2.2 Anwendungsumgebung

Die Anwendungsumgebung in Unity stellt die Daten in dem gleichen Format wie die Trainingsumgebung bereit. Statt die Ausgabe, wie beim Trainingsprozess, mit Labeln zu vergleichen, werden die Ergebnisse hier interpretiert, sodass in Unity ein Befehl ausgeführt werden kann. Die Daten werden in Unity mit einem Mircrophone Input erzeugt. Die Aufnahmen werden nach Worten getrennt, wodurch es möglich ist, die Worte einzeln zu klassifizieren. Dies ist vergleichbar damit, dass Worte im Datensatz als einzelne Dateien hinterlegt sind.

2.2.1 Deployment

Mit dem Barracuda Package ist es möglich, neuronale Netze in Unity zu verwenden. Das Plugin kann Netze im eigenen Format .nn, sowie im Austauschformat .onnx einlesen und anwenden. Dabei werden noch nicht alle Funktionalitäten des ONNX-Formats komplett unterstützt. Für die Anwendung einfacher feedforward Netze ist die Komptabilität allerdings weitestgehend gegeben. Den Trainingsprozess konnten wir so mit dem Machine Learning Framework PyTorch durchführen und die Netze mit den entsprechenden Trainingständen als ONNX exportieren. Abb. 1 zeigt die Demo-Anwendung in Unity, welche die Anwendungsumgebung aus Kapitel 2.1 darstellt.

2.2.2 Audio Slicing

In dem Notebook “FeatureDetection” [1] ist der Vorgang zur Verdeutlichung in Python umgesetzt worden. In der Anwendung läuft dies über ein C#-Script.

Der Algorithmus:

- Einlesen der .wav
- Konvertieren in eine Liste aus Buffern (Custom Class)
- Iterieren durch die Buffer-Liste und eine Serie aus "lauten" Buffern finden
- Die Serie an "lauten" Buffern in vorherige und folgende Buffer einbetten
- Die Buffer-Liste wieder in eine Float-Liste konvertieren

Es gibt Einstellungsmöglichkeiten, mit denen man die Genauigkeit anpassen könnte. Zum Beispiel die Größe der Buffer verringern oder den Threshold verändern. Dies funktioniert relativ zuverlässig und hat nur bei den Wörter "forward” und "backward" Probleme, wahrscheinlich weil dort beim Sprechen eine kleine Pause im Wort vorhanden ist. Dem kann man entgegenwirken indem man die Buffersize erhöht.
Abb.5 zeigt die gesamte Audiospur, Abb.6 und Abb.7 zeigen die einzelnen getrennten Worte.

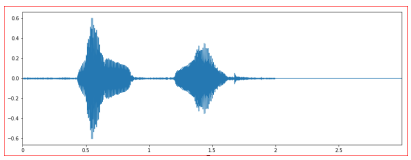

Abbildung 5: Waveform der gesamten Audiospur.

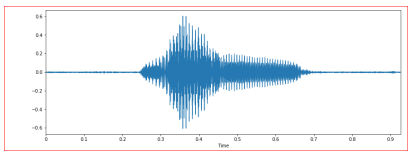

Abbildung 6: Waveform des ersten Wortes.

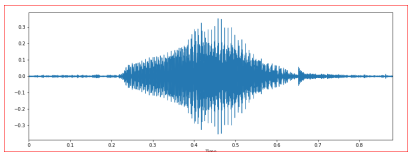

Abbildung 7: Waveform des zweiten Wortes.

[1] https://github.com/Alpe6825/Unicornn/blob/master/slicingShowcase/FeatureDetection.ipynb

3 Agent

Da sich das Mel-Spektrogramm als Bild handhaben lässt, bietet sich die Verwendung von 2D-Convolutional Neural Networks an. Grundlegend lässt sich die Verarbeitung der Bilder durch CNNs in zwei Abschnitte einteilen: Feature Erkennung und Klassifizierung. Die Feature Erkennung dient dazu durch zweidimensionale Faltungen markante Strukturen zu erkennen, welche für bestimmte Klassen typisch sind. Die Werte der Faltungsmatrizen werden im Training erlernt. Zur Reduzierung der Auflösung von Featuremaps wird der Maxpooling Algorithmus mit einer Kernelgröße von 2x2 (Halbierung der Auflösung) verwendet. Von vier Pixeln wird dabei lediglich der Pixel mit dem größten Wert übernommen. Als Aktivierungsfunktion wird typischerweise die ReLu-Funktion verwendet (siehe Formel 2).

$$ReLu(x) = max(0, x) \tag{2}$$

Die ReLu-Funktion besitzt keine Obergrenze. Um zu verhindern, dass Werte im Laufe des Netzes stark gestreut werden, kommen Batch-Normalisierungs-Layer [3] zwischen den Faltungen zum Einsatz. Die Klassifizierung besteht typischerweise aus vollverbundenen Schichten. Aufgabe ist es auf Basis von vielen gering aufgelösten Featuremaps (Ausgabe des ersten Abschnittes) für jede Klasse einen Score zu errechnen. Letztendlich wird sich bei der Klassenvorhersage dann für die Klasse mit dem höchsten Score entschieden. Für die Klassifikation wurden ein VGG19 und ein ResNet verwendet. Beim VGG handelt es sich um ein neuronales Netzwerk, bei dem die Output-Werte eines Layers als Input-Werte in das darauffolgende Layer gegeben werden. Im Unterschied dazu können bei einem ResNet vor der Eingabe dieser Werte in ein Layer noch Input-Werte vorheriger Layer aufsummiert werden, wodurch ein tieferes Netzwerk entsteht.

4 Training

Beide Netze wurden über 20 Epochen trainiert. Als Loss Funktion wurde der, für Klassifizierungsprobleme oft verwendende, CrossEntropy-Loss verwendet. Der Loss-Wert wurde, bei einer Batchgröße von 128, pro Batch gemittelt. Um den Trainingsprozess qualitativ bewerten zu können wird nach jeder Epoche der Testdatensatz durch das Netz gespeist. Auf Basis dessen wird, neben der Berechnung eines Loss-Wertes zur Anpassung der Learningrate, überprüft, in wie vielen Fällen die Klassifizierung korrekt liegt. Ein Accuracy-Wert von 1 repräsentiert dabei den Fall, dass die Klassifizierung für alle Testdaten erfolgreich war. Ein Wert von 0 das keine Klassifizierung korrekt war. Der Trainingsprozess wurde pro Netz mit dem SGD- sowie dem Adam-Optimizer durchlaufen. Letzterer führte bei beiden Netzen zu besseren Ergebnissen. Die Accuracy stieg bei beiden Netzen mit dem Adam Optimizer innerhalb von nur weniger Epochen auf über 90%, siehe Abb. 13 und Abb. 15. Mit dem SGD-Optimizer konnte in keinem der beiden Netze eine Accuracy von über 90% innerhalb von 20 Epochen erreicht werden, siehe Abb. 12 und Abb. 14.

Die Learningrate war zu Beginn der Trainingsdurchläufe je 0,001. Für den Fall, dass sich der Loss-Wert für die Testdaten über 5 Epochen nicht markant ändert (<0.0001) wurde die Learningrate um den Faktor 10 verkleinert.

5 Auswertung

Basierend auf den Loss und Accuracy-Verläufen der Testdaten im Anhang A schien die Verwendung des Adam-Optimizers zu besseren Ergebnissen beider Netze, sowohl bei den Loss- als auch bei den Accuracy-Werten zu führen. Insgesamt schien das VGG19 Netz mit Batch Normalisierung die besten Ergebnisse liefern zu können.

Die folgende Confusion-Matrix in Abb.9 zeigt die entsprechenden Ergebnisse der Testdaten. In den meisten Fällen verteilen sich alle Fehlklassifizierungen gleichmäßig auf alle anderen Klassen. Ausnahme bildet das Wort „four“ dies neigt bei einer Fehleinschätzung tendenziell etwas näher zur Klassifizierung als „forward“. Grund dafür ist mit hoher Wahrscheinlichkeit die phonetische Ähnlichkeit der beiden Worte.

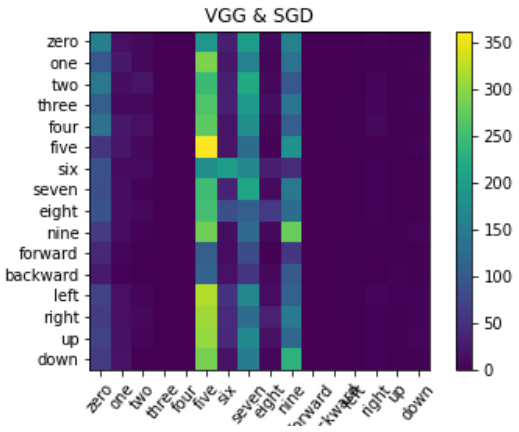

Abbildung 8: VGG und SGD

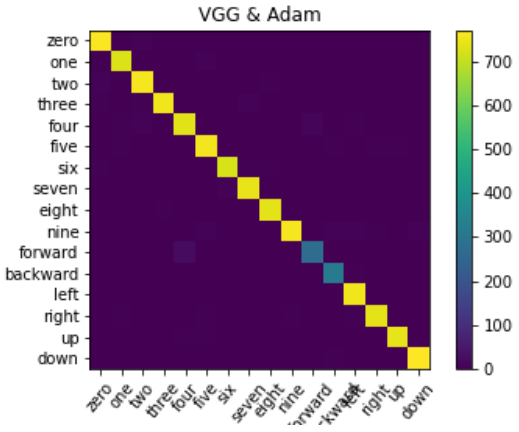

Abbildung 9: VGG und Adam

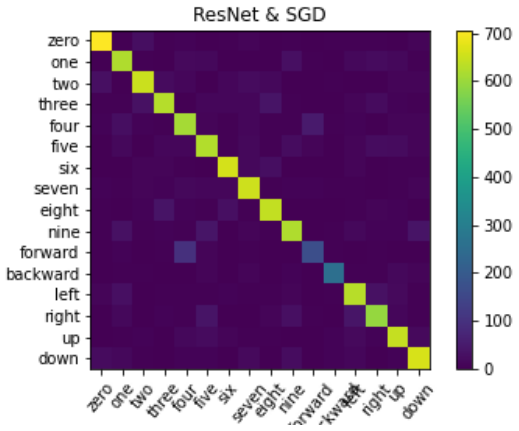

Abbildung 10: ResNet und SGD

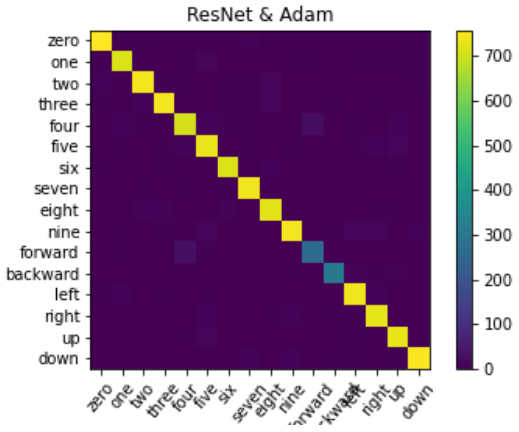

Abbildung 11: ResNet und Adam

Des Weiteren scheint es als würden die Worte „forward“ und „backward“ weniger gut erkannt werden als die restlichen Wörter. In Tabelle 1 wird deutlich, dass für zwei der Klassen weniger Beispiele vorliegen. Bei diesen handelt es sich um die Worte „forward“ und „backward“.

zero	4052	eight	3787
one	3890	nine	3934
two	3880	forward	1557
three	3727	backward	1664
four	3728	left	3801
five	4052	Right	3779
six	3860	up	3723
seven	3998	down	3917

Tabelle 1: Anzahl der Daten im Datensatz von Warden[5].

Ursprünglich war eine größere Varianz an Wortkombinationsmöglichkeiten geplant. Dazu wurden bereits Sprachaufnahmen von den Worten der folgenden Wörten gesammelt:

„create“, „delete“, „select“, „color“, „move“, „cube“, „sphere“, „plane“, „red“, „green“, „blue“, „white“.

Pro Wort kamen insgesamt 147 Aufnahmen zusammen. Daraufhin war geplant diese mit Hilfe von Audio-Modulationen (u.a. Hall, Pitch, Verschiebung) zu verändern und so auf ca. 1500 Aufnahmen zu erweitern. Durch die Verwendung von zufälligen Werten bei den Modulationen sollten Korrelationen, die durch gleiche Modulations-Funktionen entstehen und zu Fehlklassifikationen führen könnten, verhindert werden.
Bislang gelang es nicht eine Verzehnfachung ohne Korrelation zu erzeugen.

6 Fazit & Ausblick

Wir haben einen Sprachassistent für Unity 3D entwickelt, der in seiner Funktionsweise derzeit noch stark begrenzt ist, dafür aber solide funktioniert. Eine Skalierung ist bei entsprechender Anzahl von Trainingsdaten durchaus denkbar. Eine mögliche Erweiterung könnte zukünftig darin bestehen, RNNs oder LSTMs für die Verarbeitung von kontinuierlichen Inputsequenzen zu verwenden.

Derzeit wird die Berechnung des Spektrogramms noch in Python durchgeführt, da in Unity 3D keine entsprechenden Funktionen bereitstehen. Der Umweg über ein Python-Script im Hintergrund kann derzeit noch als Bottleneck der Pipeline betrachtet werden und wäre für eine Auslieferung des Programms bislang nicht optimal. Eine vollständige Verarbeitung der Daten in Unity steht noch aus.

Die Beschaffung und notwendige Menge an Trainingsdaten hat sich als problematischster Punkt des Projektes herausgestellt.

Literatur

[1] Todor Ganchev, Nikos Fakotakis, and George Kokkinakis. 2005. Comparative evaluation of various MFCC implementations on the speaker verification task. In *Proceedings of the SPECOM*, Vol. 1. 191–194.
[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. arXiv:1512.03385 [cs.CV]
[3] Sergey Ioffe and Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. arXiv:1502.03167 [cs.LG]
[4] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv:1409.1556 [cs.CV]
[5] P. Warden. 2018. Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition. *ArXiv e-prints* (April 2018). arXiv:1804.03209 [cs.CL] https://arxiv.org/abs/1804.03209

A Anhang

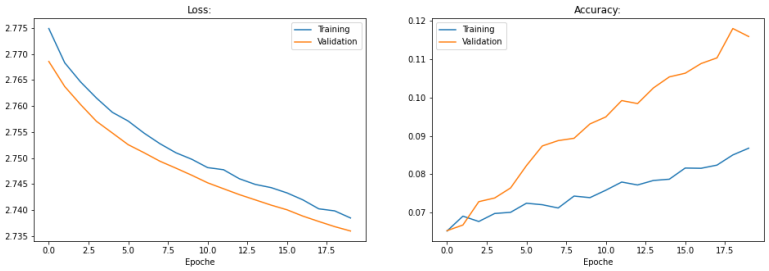

Abbildung 12: Loss Accuracy von VGG & SGD

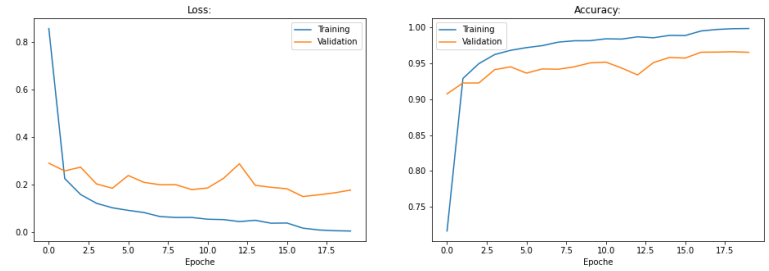

Abbildung 13: Loss Accuracy von VGG & Adam

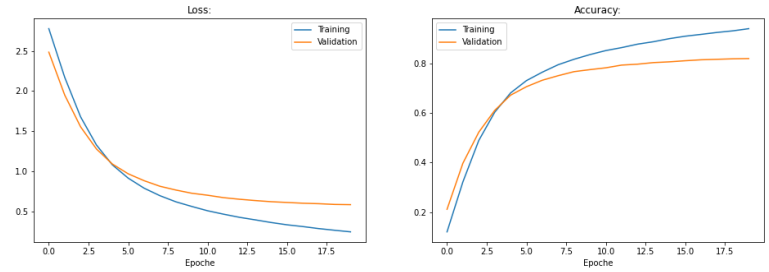

Abbildung 14: Loss Accuracy von ResNet & SGD

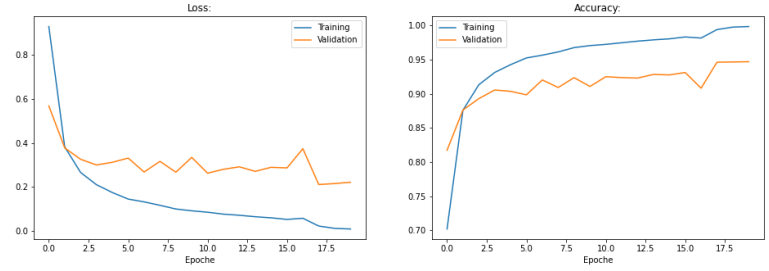

Abbildung 15: Loss Accuracy von ResNet & Adam