Rhythmusgenerierung mit künstlicher Intelligenz

Machbarkeit auf eingebetteten Systemen

Roman Kravanja

Bachelor-Thesis

Erstgutachter: Prof. Dr. Robert Manzke
Zeitgutachter: Prof. Dr.-Ing. Gunnar Eisenberg

Institut für Informatik und Elektrotechnik
FH-Kiel
2020

Inhaltsverzeichnis

Abstrakt

Die Schlagzeugsektion ist ein wichtiger Teil der meisten Musikstücke aus allen Genres. Oft fehlt jedoch ein Schlagzeuger, um die Musiker bei ihrer Arbeit zu begleiten. Um dennoch nicht auf den Schlagzeugsektion verzichten zu müssen, kann entweder eine statische, zuvor eingestellte Spur eingespielt, oder eine neue dynamisch generiert werden. Letzteres hat den Vorteil, sich automatisch entsprechend der Anforderungen anpassen zu können. Eine solche Lösung ist weniger aufwendig in der Anwendung und kann zudem bessere Ergebnisse erzeugen. Ein Ansatz zur automatischen Erzeugung solcher Schlagzeugsektionen ist die Generierung mithilfe künstlicher Intelligenz, die zuvor mit Testdaten trainiert wurde. Es gibt bereits wissenschaftliche Arbeiten und erste Produkte, die sich mit dieser Thematik befassen. Die Umsetzung dieser Methoden auf eingebetteten Systemen ist aber bisher nur begrenzt erfolgt. In dieser Thesis wird daher untersucht, inwiefern ein automatisches System zur Rhythmusgenerierung mittels moderner Verfahren der künstlichen Intelligenz auf eingebetteten Systemen umsetzbar ist. Diese Untersuchung ist insbesondere auch aufgrund des stetigen Zuwachses an Rechenleistung bei sinkenden Kosten für moderne Mikrocontroller von hohem Interesse. Dabei soll die Einstellung des Schlagzeugers durch den Anwender erfolgen. Zum Umfang dieser Aufgabe gehört das Suchen und Ausprobieren möglicher KI-Ansetze sowie deren Beurteilung im Hinblick auf die Fragestellung.

Einleitung

Ohne einen Schlagzeuger kann das Musizieren schwierig sein. Eine gute Schlagzeugsektion gibt den Takt vor und verleiht dem Stück Struktur. Wenn kein Schlagzeuger verfügbar ist und ein Musiker nicht auf diese Begleitung verzichten möchte, kann auf eine statische Spur zurückgegriffen werden. Diese hat allerdings den Nachteil, dass sie manuell ausgewählt und angepasst werden muss. Auch ist sie statisch und kann nicht dynamisch auf wechselnde Anforderungen reagieren. Die Lösung für dieses Problem bieten automatische Schlagzeuger. Abhängig von der Implementierung kann ein solcher die gewünschte Spur spontan und angepasst an Einstellungen und Situation generieren. Diese Lösung ist auch im Rahmen des Hardware Sequencer Projektes interessant. Für diesen speziellen Anforderungsfall soll eine mikrocontrollerbasierte Hardware einen Schlagzeuger imitieren. Dabei soll der Output des Gerätes durch Eingaben des Endnutzers eingestellt werden können.
In dieser Thesis werden die einzelnen Teilherausforderungen auf Umsetzbarkeit untersucht. Auch werden verschiedene Ansätze zur Umsetzung in Experimenten getestet. Anschließend wird ein Modell mithilfe maschinellen Lernens erstellt und optimiert, welches letztlich auf der Hardware prototypisch implementiert wird.

Technischer Hintergrund

3.1 Hardware Sequencer Projekt

Das Hardware Sequencer Projekt wurde im Rahmen der Bachelor Thesis von René Herrmann erstellt. Die, für diese Arbeit wichtige, Hardware bestand aus einem ESP32-basiertem Gerät mit acht zu benutzenden Ausgängen. Der verbaute ESP32-Mikrocontroller lässt sich über eine, in das Gerät eingebaute, Mikro-USB Schnittstelle flashen.[Her20]

3.2 Mikrocontroller Plattform

Der Microcontroller, der für diese Arbeit als Zielplattform angestrebt wird, ist der ESP32-WROVER-B, im Folgenden ESP32 genannt. Es handelt sich dabei um eine Microcontroller Unit des Herstellers Espressif. Der ESP32 hat eine Taktrate von 40 Megahertz, 4 Megabyte Flash-Speicher und zusätzliche 8 MB statischen Arbeitsspeicher. Darüber hinaus verfügt er über Bluetooth- und WLAN-Antennen.
Die eingeschränkten Ressourcen dieser Hardware stellen eine Herausforderung für die Implementierung der Software dar, die in dieser Arbeit untersucht wird. [Esp20]

3.3 Formen künstlicher Intelligenz

Als Künstliche Intelligenz oder auch KI bezeichnet man einen Algorithmus, welcher menschenähnlich Entscheidungen treffen kann. Die Komplexität dieser Entscheidungen reicht von einfachen, statischen Algorithmen, bis hin zu selbst lernenden neuralen Netzen, wie sie für das autonome Fahren eingesetzt werden. Eine solche KI bietet sich auch an, um die Herausforderungen dieser Thesis anzugehen. Eine genaue Abgrenzung, ab wann ein Algorithmus als intelligent bezeichnet wird, gibt es nicht, da bereits die genaue Definition der Intelligenz umstritten ist[LH07][BPB17].

3.3.1 Statische Algorithmen

Eine statischer Algorithmus ist ein Programm, dessen Entscheidungsprozess vollständig vom Programmierer bestimmt wird. Eine solche Software lernt weder selbstständig dazu, noch kann sie dynamisch auf Probleme reagieren, für die sie nicht geschaffen wurde. Ein Beispiel hierfür ist ein einfacher Chatbot, für den der Entwickler Antworten zu entsprechenden Nachrichten vorgegeben hat. Wird eine Nachricht empfangen, die nicht vorgesehen war, oder zu der keine Antwort in der Datenbank steht, kann das Programm nicht dynamisch reagieren. Statische Algorithmen können viele Probleme lösen, doch kommen an ihre Grenzen, wenn die Datenmenge divers ist und Mustern folgt, die für den Erschaffer des Programmes nicht zu erschließen sind.[Lea20]

3.3.2 Maschinelles Lernen

Maschinelles Lernen bezeichnet ein Feld von Algorithmen, die aus Datenmengen lernen können, um Lösungen für Probleme zu finden, die sich auf Daten ähnlicher Art beziehen. Diese können beispielsweise in der Form von Datenbanken oder Bildern vorliegen. Dadurch eignen sie sich grundsätzlich vor allem für Aufgaben, die schwierig für Menschen zu erfassen sind. Dies ist beispielsweise der Fall, wenn Menschen die Muster in Daten nicht oder nur schwer erkennen können. Die Herangehensweisen an einen solchen Lernprozess lassen sich in vier grundlegenden Kategorien einordnen, die in den folgenden Abschnitten erläutert sind.[TS10]

Supervised Learning

Supervised Learning beschreibt einen Vorgang, bei dem ein Algorithmus einen Trainingsdatensatz mit Ausgangsdaten und dazu erwünschten Lösungen erhält.

$$(X_{train}, Y_{train}) = \{(x_1, y_1), ..., (x_n, y_n)\}\text{[Zha20, pg.224]}$$

Das Programm lernt zu diesen Daten eine Funktion, welche aus den Eingaben die gegebenen Ausgaben erzeugt. Für diesen Anlauf müssen also bereits Lösungen für das Set an Problemen existieren. Ein Beispiel für einen solchen Algorithmus sind neurale Netze, die Gegenstände auf Bildern erkennen sollen und als Trainingsmaterial eine Menge an Bildern mit Gegenständen oder Lebewesen und deren Zuordnung zu Kategorien enthalten, beispielsweise 'Hund' oder 'Katze'[Zha20].

Unsupervised Learning

Beim Unsupervised Learning wird einem Programm ein Datensatz gegeben, zu dem keine gewünschten Lösungen vorliegen.

$$X_{train} = Xu = \{x1, ..., xu\}[\text{Zha20, pg.224}]$$

Bei diesem Ansatz lernt der Algorithmus selbstständig, die Muster zu erkennen, die in den Eingabedaten zu finden sind. Angewendet werden kann dies, um Datenmengen einzuordnen, die für Menschen zu unübersichtlich sind.[Zha20]

Reinforcement Learning

Reinforcement Learning ist eine Teilmenge des maschinellen Lernens, bei der ein Programm selbstständig Daten generieren soll. Die einzigen Eingaben, die von außen erfolgen, erhält der Algorithmus in der Form von positiver oder negativer Rückmeldung zu seinen Ausgaben. Dabei wird in einer dynamischen Umgebung agiert, aus denen das Programm mit den gelernten Funktionen Daten erstellt, die positive Rückmeldung erzeugen.[Zha20]

Transfer Learning

Transfer Learning beschreibt das Übertragen von erlentem Wissen auf andere Probleme, die sich mit eben jenem Wissen lösen lassen. Im maschinellen Lernen ist diese Methodik nützlich, um Programme auf Situationen reagieren zu lassen, für die sie so nicht trainiert wurden. Durch Transfer Learning werden solche Programme flexibler und können eigenständig neue Lösungen zu Problemen entwickeln.[TS10]

3.3.3 Neurale Netzwerke

Künstliche neurale Netze sind digitale Strukturen, die Neuronen aus der Natur nachempfunden sind und deren Verhalten emulieren. Natürliche Neuronen, wie sie beispielsweise im menschlichen Gehirn zu finden sind, stehen in Verbindung mit anderen Neuronen und können solche Verbindungen ausbauen, abbauen oder neue anlegen. Dadurch werden Signale, die durch das Netz laufen, unterschiedlich verarbeitet und durch das Modifizieren der Verbindungen wird der Prozess optimiert. Das Verändern des Netzes wird als Lernen bezeichnet und künstliche, neurale Netzwerke somit dem maschinellen Lernen zugeordnet. Bereits 1942 untersuchte Warren S. McCulloch die Struktur neuraler Netze im Gehirn und erstellte theoretische Modelle solcher Strukturen.[Pic04] Ein Beispiel für eine soche Struktur ist in Abbildung 3.1 dargestellt.

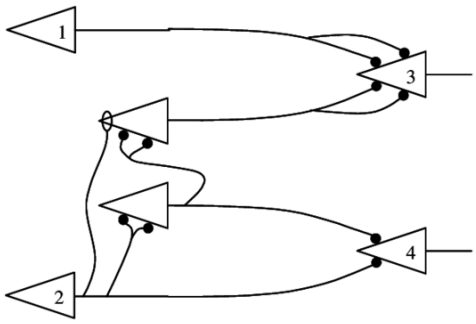

Abbildung 3.1: Darstellung der neuronalen Verbindungen, welche das Hitzeempfinden steuern[Pic04]

In den letzten Jahren nahm die Entwicklung im Bereich der neuralen Netze verstärkt Fahrt auf. Neue Techniken und immer leistungsfähigere Hardware sorgten für mehr Möglichkeiten in diesem Bereich der Informationstechnologie.

Künstliche neurale Netze werden immer für einen bestimmten Anwendungsfall entworfen. Die entstehende Struktur nennt man Topografie. Dabei werden Neuronen in Schichten mit Verknüpfungen zu vorherigen und nachfolgenden Schichten verbunden. Meist gibt es dazu eine Eingabe- und eine Ausgabeschicht. Die Eingabeschicht erhält die Signale, auf die das Netz reagieren soll, während die Ausgabeschicht die Antwort des Netzes auf die Stimulation ausgibt. Die Schichten dazwischen verarbeiten die Signale und bestimmen letztlich die Ausgabe und werden Hidden Layer genannt. Auch bei künstlichen neuronalen Netzen wird das Lernen durch Modifikation der Verbindungen zwischen den Neuronen einzelner Schichten realisiert. Dabei kann man die einzelnen Neuronen als Funktionen verstehen, die aus den gegebenen Parametern entsprechend Ergebnisse berechnen und weiterreichen. Verändert sich die Funktion, so ändert sich auch das Ergebnis und das Netzwerk lernt. Die Veränderungen der Verknüpfungen erfolgen dabei zufällig. Verbessert sich das Ergebnis, wird die Änderung in der Regel beibehalten und die KI hat etwas neues erlernt. Verschlechtert sich das Ergebnis, kann die Änderung verworfen werden. Welche Änderungen genau an wie vielen Verbindungen vorgenommen werden, hängt von der konkreten Implementation ab. Generell wird zunächst ein sogenannter Fehler berechnet, welcher angibt, wie weit das ausgegebene Ergebnis von dem erwünschten entfernt ist. Durch Lernen wird versucht, diesen Fehler zu minimieren. Während die Anzahl möglicher Topografien theoretisch unbegrenzt ist, gibt es einige Modellkategorien, die die meisten Anwendungsfälle grundlegend abdecken.[Wan03]
Abbildung 3.2 zeigt ein einfaches Neurales Netz.

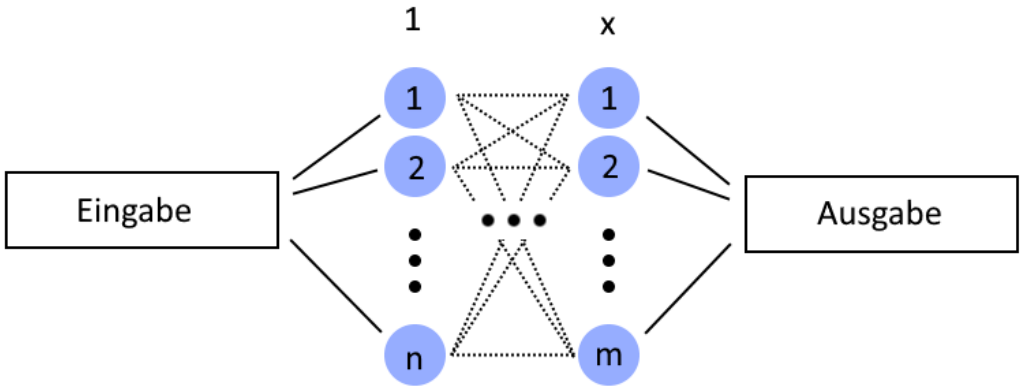

Abbildung 3.2: Beispiel eines neuralen Netzes. Dargestellt werden Ein- und Ausgabe, sowie die Schichten mit verarbeitenden Neuronen dazwischen.

Feed Forward Neural Network

Feed Forward Netzwerke sind eine einfache Form neuraler Netze. Die Hauptfunktion besteht im Annähern einer mathematischen Funktion. Dabei stellen die Eingaben in das Netz die Parameter der Funktion dar, während die Ausgaben sich dem Ergebnis der Funktion annähern sollen. Der Name Feed Forward kommt von der Flussrichtung der Informationen durch das Netzwerk. Die Signale werden von der Eingabe geradewegs durch die Neuronen zur Ausgabe verarbeitet, ohne dass Ergebnisse and vorherige Neuronen rückgemeldet werden.[Upa19] Abbildung 3.3 zeigt, wie eine solche Struktur aussehen kann.

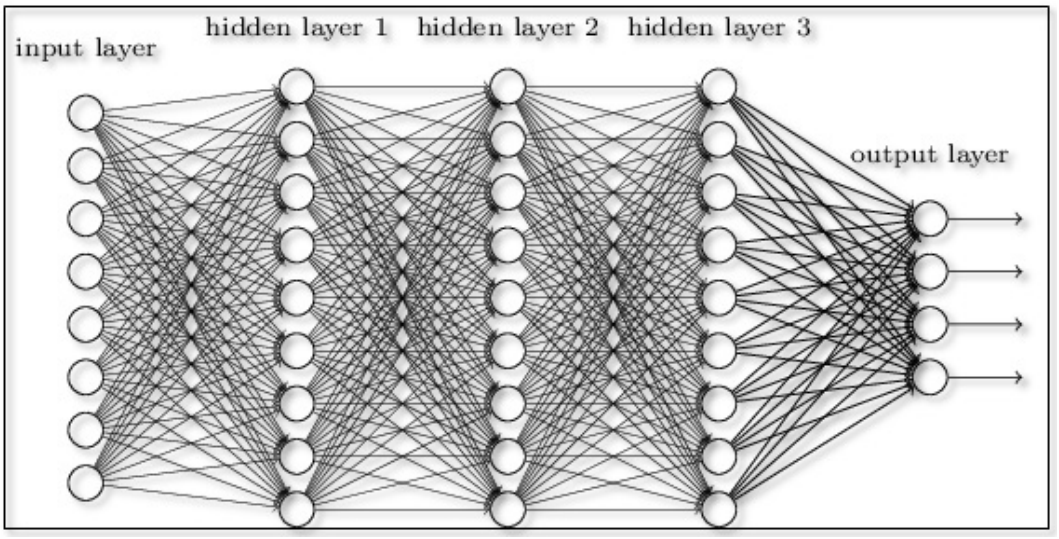

Abbildung 3.3: Ein Feed Forward Neural Network mit drei Hidden Layern.[Gup17]

Recurrent Neural Networks

In einem Recurrent Neural Network werden die Ausgaben von Neuronen als Rückmeldung für die Neuronen in vorherigen Schichten genutzt. Abbildung 3.4 zeigt die Darstellung eines Neurons, welches auch seine eigenen Ausgaben verarbeiten kann. Dadurch können zustandsbasierte Modelle entstehen. Ein solches Vorgehen erlaubt das Verarbeiten beliebig großer Eingaben, die in Abschnitten eingegeben und verrechnet werden können. Jeder Datenabschnitt verändert dabei den Zustand des Netzes und setzt so den Kontext für die Verarbeitung nachfolgender Daten. Ein weiterer Vorteil dieser Architektur ist, dass die Größe des Modells nicht mit der Größe der Eingabe skalieren muss. Nachteilig an dieser Herangehensweise ist die aufwendige Berechnung und die Schwierigkeit, auf Informationen

zuzugreifen, die das Netzwerk in der Vergangenheit verarbeitete.[AAntb]

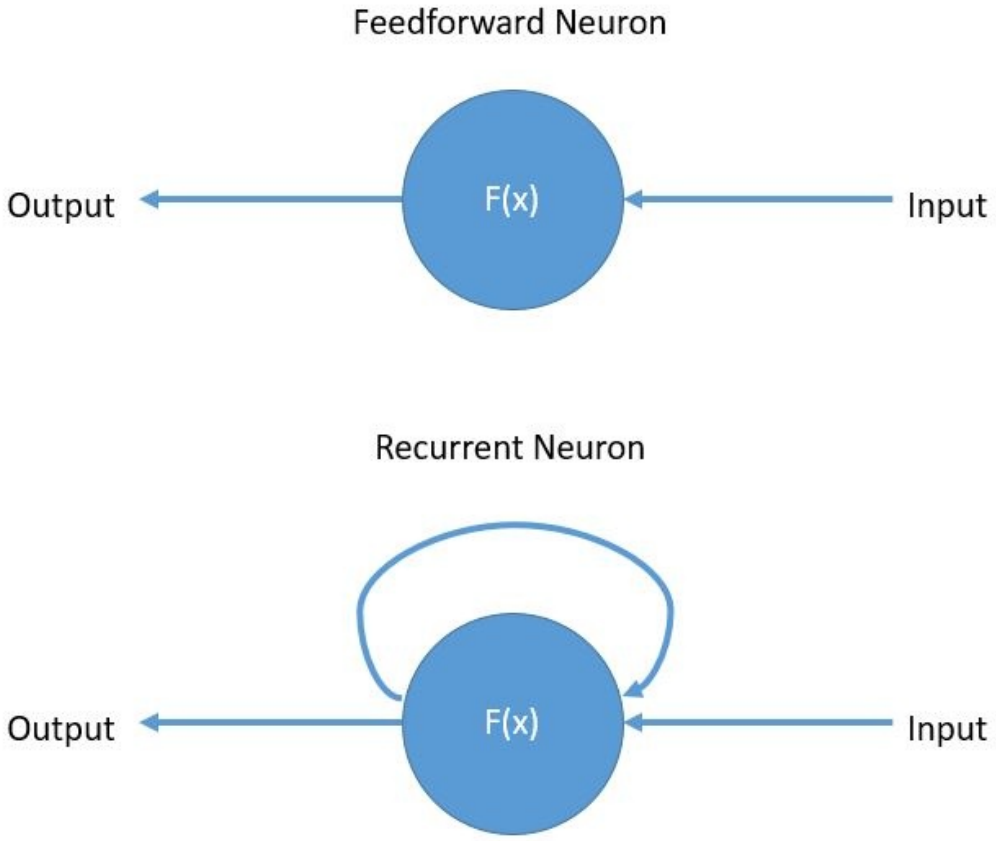

Abbildung 3.4: Beispiel eines Feedforward Neurons im Vergleich zu einem Recurrent Neuron.[Wie17]

Convolutional Neural Networks

Convolutional Neural Networks sind eine Klasse von neuralen Netzen, die hauptsächlich zur Verarbeitung von Bildern eingesetzt werden. Sie beste-

hen in der Regel aus einer Eingabeschicht, welche die zu verarbeitenden Bilder entgegennimmt, dann einer Reihe sogenannter Convolution-Layer, welche Filter auf Teilabschnitte der Bilder anwenden und die gefilterten Datenabschnitte nach Verarbeitung auf einer sogenannten Activation Map darstellen. Diese Activation Map wird als Ausgabe an die nachfolgenden Schichten weitergegeben.
Meist folgen dann auf diese Convolution-Layer sogenannte Pooling-Layer, welche die zuvor ausgegebenen Activation Maps auf eine kleinere Auflösung herunterskalieren. Die Ausgabe dieser Layer wird dann als eindimensionaler Array von Fully-Connected Layern weiter bearbeitet. Ein Beispiel für eine Verarbeitungskette in einem Convolutional Neural Network ist auf Abbildung 3.5 zu sehen. [AAnta]

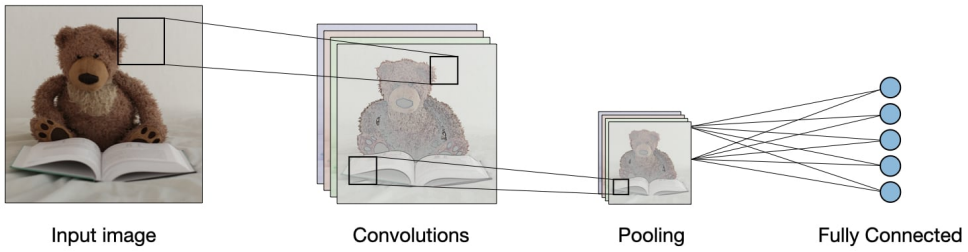

Abbildung 3.5: Beispielhafter Ablauf der Bildverarbeitung innerhalb eines Convolutional Neural Networks.[AAnta]

Generativ Adversarial Networks / GAN

Generativ Adversarial Networks sind neurale Netze, die aus zwei Teilen bestehen. Beide diese Teile sind selbst neurale Netze,von denen eines Generator und das andere Discriminator genannt wird. Dem Generator werden dabei Zufallsdaten als Eingaben gegeben, aus denen er, unter günstigen

Umständen, zunehmend genauere Ausgabedaten erzeugt. Der Discriminator erhält diese Ausgaben als Eingabedaten. Zusätzlich wird der Discriminator mit Daten aus einem realen Datensatz gefüttert. In Abbildung 3.6 ist die Struktur eines GANs zu sehen.

Die Aufgabe des Discriminators besteht darin, zu entscheiden, ob die eingehenden Datensätze aus dem realen Datensatz kommen, oder vom Generator erzeugt worden sind. Beide der neuralen Netze lernen im Verlauf des Trainings. Dadurch entsteht eine Dynamik, durch die der Generator lernt, die Eigenschaften der Originaldaten immer besser zu imitieren, während der Discriminator lernt, echte Beispiele von Fälschungen zu unterscheiden. Diese Art von Modell wird oft für die Generierung von Bildern eingesetzt.[Goo+20]

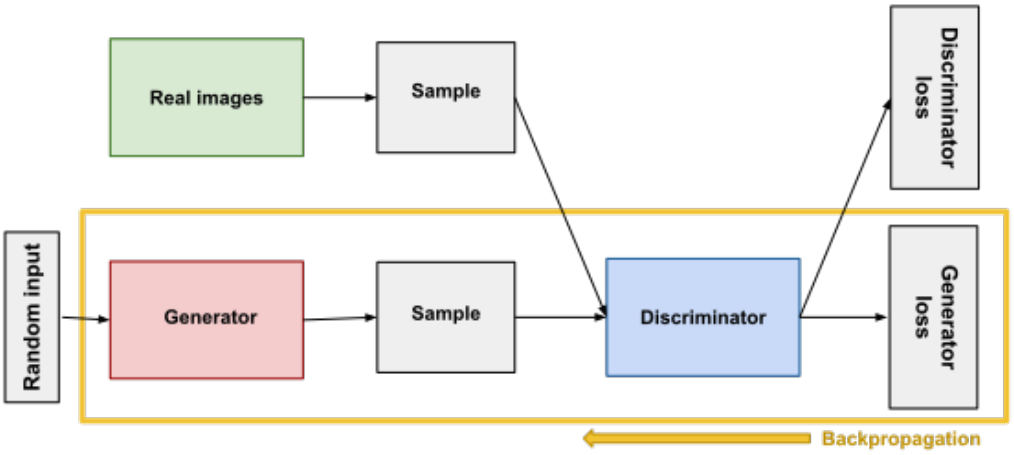

Abbildung 3.6: Strunktur eines einfachen GAN.[Goont]

Conditional Generative Adversarial Netorks / cGAN

Conditional GANs, cGANs oder auch bedingte GANs sind ähnlich strukturiert wie die zuvor erläuterten GANs, erlauben aber, das Füttern des

Generators mit zusätzlichen Eckdaten, um die Ausgabe zu steuern. Dies erlaubt, das gezielte Generieren von Daten einer angegebenen Kategorie, statt zufällig aus dem gesamten Datenraum. Dabei ersetzen die gezielten Eingabedaten entweder die zufälligen Eingaben des Generators, oder werden mit diesen verwoben. Ein Beispiel für ein solches Zusammenlegen von Daten ist das Hinzufügen eines weiteren Farbkanals in ein zufällig generiertes Eingabebild, in dem die bestimmenden Parameter codiert sind. Genau diese Daten werden dann erneut in die Ausgabe des Generators eingebettet, ohne dass das Modell diese selber beeinflussen kann. Dadurch wird sichergestellt, dass der Discriminator Zugriff auf die Parameter hat, zu denen der eingehende Datensatz zuzuordnen ist. Letztlich müssen die Parameter auch in die Originaldatensätze eingebettet werden, die genutzt werden, um den Discriminator zu trainieren. Ein cGAN mit vielen versteckten Schichten wird auch als "Conditional Deep Convolutional Generative Adversarial Networköder cDCGAN bezeichnet.[MO14][Bro19a]

Autoencoder

Autoencoder sind aus zwei Hälften bestehende neurale Netze. Sie werden häufig benutzt, um Datensets in Kategorien einzuordnen und die Datenmenge dabei auf die wichtigsten Informationen zu minimieren. Dazu wird das Modell meist symmetrisch aufgebaut. Die erste Hälfte, der sogenannte Encoder, erhält Originaldaten als Eingabe und reduziert diese auf die wichtigsten Eigenschaften, die er dann ausgibt. Diese nimmt die zweite Hälfte, der Decoder, dann an, und versucht daraus die Quelldaten zu rekonstruieren. Das gesamte Modell wird dann bewertet, indem die Ausgabe des Decoders mit

der Eingabe des Encoders verglichen wird. Ein Beispiel für einen solchen Autoencoder zeigt die Abbildung 3.7.
Encoder und Decoder können nach dem Training als seperate Modelle verwendet werden, um entweder neue Daten zu erzeugen, oder existierende einzuordnen. Dabei kann man die Eingabe für den Decoder variieren, um die Ausgabe gezielt zu steuern.
Das Autoencoder-Design ist für diese Thesis insbesondere interessant, da ein entsprechender Decoder die Anforderungen erfüllen würde.[Ngnt] In den Experimenten in 4.2 wird dieser Ansatz weiter verfolgt.

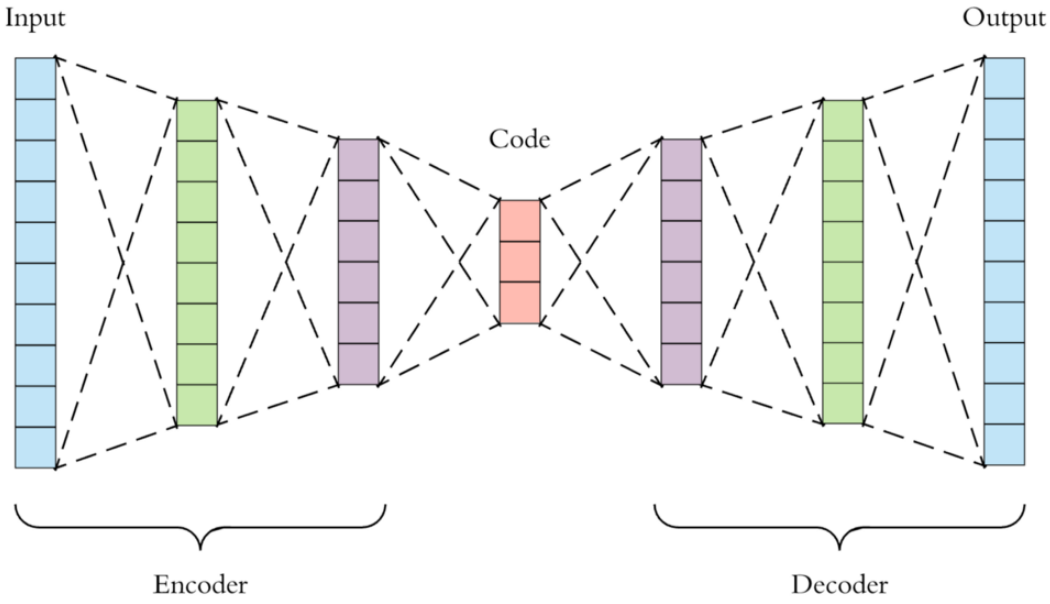

Abbildung 3.7: Beispielhafter Aufbau eines Autoencoders.[Der17]

3.4 Rhythmen durch künstliche Intelligenz - Nachforschung

Die Erzeugung von musikalischen Elementen im Allgemeinen und von Schlagzeugspuren im Besonderen ist keine neue Idee. Es gibt bereits einige, frei verfügbare Projekte und wissenschaftliche Arbeiten, die sich mit diesem Thema beschäftigen. Dieses Kapitel widmet sich einer Auseinandersetzung mit diesen Werken und es werden gewonnenne Ideen und Erkenntnisse erläutert.

3.4.1 Sony CSL

DrumNet von Sony CSL Paris ist ein neurales Netz, welches zu bestehenden Musikstücken Schlagzeugspuren erzeugen kann. Dazu wurde das Modell mit Daten trainiert, die in einem 16-dimensionalen Raum eingeordnet sind. So konnten Zusammenhänge zwischen verschiedenen Instrumenten erlernt werden. Dabei bestanden die Eingabedaten aus Audiodateien, statt, wie bei ähnlichen Projekten meistens der Fall, aus abstrakten Versionen der Musikstücke. Im Zeitraum dieser Recherche, war DrumNet nur für das Arbeiten mit kick-drum Rhythmen trainiert.
In der Anwendung kann DrumNet entweder eigenständig und dynamisch Spuren zu Live-Musik generieren, manuell gesteuert werden oder zu bestehenden Musikstücken Spuren erzeugen.[Son19]

3.4.2 Mutable Instruments - Grids

Der topografische Drum Sequencer von Mutable Instruments ist ein Schlagzeug-Generator, der auf Basis eines mit maschinellen Lernens erstell-

ten Programms funktioniert. Drei Steuereingänge dienen dem Programm als Eingabe. Die ersten beiden lassen den Sequencer in einem zweidimensionalen Raum interpolieren, während der dritte Steuereingang die 'event density' kontrolliert.
Die Ausgabe der erzeugten Schlagzeugsektionen erfolgt über drei Hauptausgänge und drei weitere Hilfsausgänge, die zusätzliche Signale zur Ergänzung des jeweiligen Hauptausganges geben können.[Insnt]

3.4.3 Magenta - GrooVAE

GrooVae ist eine Klasse von Modellen aus dem maschinellen Lernen, die bestehende Schlagzeugsektionen mit zusätzlichen Tönen ausschmücken. Die hinzugefügten Elemente sollen dem Stück einen besseren musikalischen Fluss geben, und dadurch steife Schlagzeugspuren verbessern. Andersherum können so trainierte Modelle auch eine Schlagzeugsektion zu einem gegebenen Rhythmus erzeugen.
Das GrooVAE Modellen zugrunde liegende Modell ist MusicVAE, welches wiederum auf dem sogenannten Variational Autoencoder basiert.[GRE19][Raf+18] Diese Art von Modellen ist im Kapitel 3.3.3 weiter erläutert.

3.4.4 Musenet

Musenet ist ein Projekt, welches mit Hilfe künstlicher Intelligenz in Form eines trainierten, neuralen Netzes vollständige musikalische Werke komponieren kann. Trainiert wurde das Netz mit hundertausenden MIDI Dateien aus vielen Musikrichtungen und kann daher auch vielfältige Stile imitieren.

Eingesetzt wurde Musenet, indem die ersten Töne einer Sequenz als Eingabe gegeben werden und das Programm dann den weiteren Verlauf erzeugt. Bis zu 10 Instrumente können parallel vom Projekt dirigiert werden.
Das Modell für Musenet ist ein tiefes neurales Netz, welches auf der GPT-2 Technologie basiert.[Ope19]
GPT-2 ist ein generelles Modell, welches ursprünglich für Sprachgenerierung erschaffen wurde und auch ohne aufgabenspezifisches Training angewendet werden kann. Dabei bestimmt es das nächste Zeichen einer Zeichenfolge, basierend auf allen vorherigen Zeichen der Folge. Für Musenet wurde diese Funktionalität auf musikalische Daten angewendet. Das Originalmodel GPT-2 wurde wegen Missbrauchsbedenken nicht veröffentlicht.[Rad+19]

3.4.5 Drumbot

Drumbot ist ein Programm, welches mit Hilfe eines trainierten neuralen Netzes zu einer gegebenen Melodie eine Schlagzeugsektion erzeugen kann. Dieses Modell ist frei verfügbar und wird, zusammen mit einer Umgebung, um es zu nutzen, auf github zum Download bereitgestellt. Für diese Thesis ist der Drumbot allerdings nicht das passende Modell, da die Schlagzeugspuren unabhängig von der Melodie erzeugt werden sollen.[DPnt]

3.4.6 Drum Patterns from Latent Space

Das Projekt Drum Patterns from Latent Space von Aleksey Tikhonov bearbeitet die Erstellung und das Training eines neuralen Netzes, welches Schlagzeugspuren erzeugt. Gesteuert wird das Modell dabei durch Koordinaten im Latent Space. Um die Ergebnisse zu visualisieren ordnete Tikhonov existie-

rende Schlagzeugsektionen mit dem Encoder des erzeugten Autoencoders in den Latent Space ein und übertrug diesen mittels T-SNE auf eine zweidimensionale Darstellung. Dabei ließen sich in dieser Darstellung verschiedene Musikrichtungen an Koordinaten ausrichten.
T-SNE steht für 't-Distributed Stochastic Neighbor Embedding' und ist eine Technik für die Reduktion der Dimensionalität von Daten, welche sich für die Visualisierung multidimensionaler Daten eignet. [MH08]
Viele Ziele dieses Projektes stimmen mit denen der vorliegenden Thesis überein. Allerdings ist die gezielte Steuerung der Ausgaben nicht möglich, da mithilfe von T-SNE zwar vom Latent Space in die zweidimensinale Ebene übertragen werden kann, aber nicht umgekehrt.

3.5 Tensorflow

Tensorflow ist ein Open-Source System für maschinelles Lernen. Es erlaubt die Verarbeitung großer Mengen an Daten über mehrere Gerät verteilt und ist für das Definieren, Trainieren und Nutzen von neuralen Netzwerken entworfen worden. Erstellt und veröffentlicht wurde Tensorflow vom Google Brain Team.[Aba+16]
Eine Teilmenge der Funktionen von Tensorflow lässt sich mittels integrierten Konvertierungsfunktionen zu sogenannten Tensorflow-Lite Modellen umwandeln. Diese wurden ursprünglich für den Einsatz auf mobilen Endgeräten entwickelt. Um Tensorflowmodelle auf Microcontrollern und anderen Geräten mit weiter eingeschränkten Ressourcen nutzen zu können, wurde Tensorflow Micro entwickelt. Das Framework bietet die Möglichkeit, zuvor trainierte Tensorflow Lite Modelle in C-Arrays umzuwandeln und auf den Zielsyste-

men zu nutzen. Der ESP32-Chipsatz, der im Hardware Sequencer Projekt genutzt wird, wird seit 2020 von Tensorflow Micro unterstützt.[Dav+20]
Zum Zeitpunkt der Erstellung dieser Arbeit konnte keine andere Möglichkeit ermittelt werden, Modelle auf dem ESP32 zu nutzen, welche mit maschinellem Lernen erstellt worden sind. Theoretisch wäre eine eigene Implementation umsetzbar. Dieser Ansatz würde jedoch weit über den Umfang dieser Arbeit hinausgehen.

3.5.1 Genutzte Tensorflow-Funktionalität

Für die Experimente in dieser Thesis wurde hauptsächlich die Schnittstelle Keras des Tensoflow Frameworks verwendet. Diese bietet einen high-level Zugriff auf die Funktionen von Tensorflow, erlaubt aber auch feinere Modifikationen der Abläufe durch direkte Tensorflow-Zugriffe.
Für das Arbeiten mit Keras ist das Verständnis einiger Begriffe notwendig, die oft keine direkte Übersetzung in die deutsche Sprache erlauben. Nachfolgend sind die, in diesem Kontext wichtigen, Begriffe aufgeführt.

'Labels'

Ein Label ist eine Kategorie, in die neurale Netzwerke die gegebenen Daten einordnen sollen. In einfachen Fällen ist die Entscheidung binär, das heißt, die Anfrage wird als Teil der Kategorie erkannt, oder nicht. Eine solche Aufgabe ist beispielsweise das Identifizieren eines Gesichtes auf einem Foto. Dabei gibt es nur zwei Möglichkeiten. Entweder die KI erkennt das Gesicht, oder hält es für fremd. Für kompliziertere Anforderungen können auch mehrere Label erforderlich sein. So kann beispielsweise auf dem Bild eines Kleidungsstücks nicht nur die Art des Kleidungsstücks, sondern auch die Farbe des Objektes

erkannt werden. Dies kann entweder durch seperate Modelle und Trainigsvorgänge realisiert werden, oder in einem kombinierten Projekt. Ein Modell würde dann beide Labels ausgeben, in diesem Beispiel also möglicherweise 'Kleid' und 'Blau'.[Ros17]

'Layer'

In dem Kontext des maschinellen Lernens werden einzelne Schichten eines neuralen Netzwerkes als Layer bezeichnet. Die Konfiguration und Anordnung dieser Layer bestimmt das Verhalten des gesamten Modells. In der Regel ist die erste Schicht eine Eingabeschicht - sie nimmt also die Eingaben an. Diese Eingaben bilden die Ausgangssituation ab, für die das Netz eine Entscheidung fällen soll. Die Eingabeschicht kann eine erste Verarbeitung der Daten vornehmen, und reicht die Ergebnisse anschließend an die tieferen Schichten weiter. Die letzte Schicht in einem Modell ist die Ausgabeschicht. Über diese Schicht lassen sich die Ergebnisse nach der Verarbeitung abrufen. Die Anzahl der Neuronen in einzelnen Schichten kann variieren, ebenso wie das Verhalten der Neuronen in diesen Schichten. Die letzte Schicht ist in der Regel eine Ausgabeschicht, die das Resultat aus den Berechnugen ausgibt.[Kerh]
Keras bietet viele vordefinierte Layer an, die in Modellen genutzt werden können; es lassen sich aber auch nutzerdefinierte Schichten erstellen. Die folgende Tabelle 3.1 stellt eine Übersicht der, im Verlauf dieser Arbeit genutzten, Layer dar. Dabei werden Parameter, die nicht in den Experimenten verwendet werden, aus Gründen der Übersichtlichkeit ausgelassen.

Name	Beschreibung	Parameter
Reshape	Dieser Layer formatiert die Eingabe in das angegebene Format. Dabei bleibt die Anzahl der Neuronen unverändert. Wird ein mit der Neuronenzahl inkompatibles, Format angegeben, wird ein Fehler ausgegeben.[Kero]	'input_shape': Das Format, zu dem die eingehenden Daten konvertiert werden sollen.
Dense	Dense Layer verarbeitet die eingehenden Daten via Skalarmultiplikation mit dem eigenen Kernel. Auf das Ergebnis dieser Berechnung werden dann Activation Funktion und Bias angewendet, falls aktiviert. Das Endresult wird dann als Ausgabe an die nächste Schicht weitergegeben.[Kerd]	'units': Anzahl der Neuronen im Kernel des Layers. 'activation': Activation function die auf die Ausgabe der Kernel-Operation angewendet werden soll. 'kernel_initializer': Algorithmus zur Initialisierung der Kernel-Neuronen zu Beginn des Trainings.
Flatten	Dieser Layer formatiert die Eingabe in ein eindimensionales Format. Dabei bleibt die Anzahl der Neuronen unverändert.[Kerg]	

Conv2D	Convolution Layer verschränken den eigenen Kernel mit den Eingabedaten. Dabei werden zunächst die angegebenen Filter angewendet, um die Datenmenge möglichst auf die wichtigen Aspekte zu reduzieren. Dann werden die Daten in Ausschnitten mit dem Kernel verrechnet. Zu dem Resultat wird dann der Bias addiert, falls eingestellt. Anschließend wird das Ergebnis durch die Aktivierungsfunktion verarbeitet und dann an nachfolgende Schichten ausgegeben. Der Conv2D-Layer eignet sich vor allem für die Verarbeitung von Bildern.[Kerb]	'filters': Die Filter, die auf die eingehenden Daten angewendet werden sollen. 'kernel_size': Anzahl der Neuronen im Kernel des Layers. 'strides': Bestimmt die Größe der Daten-Ausschnitte, die mit dem Kernel verrechnet werden. 'padding': Das Padding wird an den Ränder der Eingabematrix angefügt, damit auch Ausschnitte erstellt und verarbeitet werden können, die am Rand der Matrix liegen.

Conv2DTranspose	Dieser Layer geht ähnlich vor wie der Conv2D Layer, mit dem Unterschied, dass die Ausgabe andere Dimensionen haben kann. Dadurch kann diese Schicht genutzt werden, Bilder hoch zu skalieren.[Kerc]	'shape': Das Format, zu dem die eingehenden Daten konvertiert werden sollen.
LeakyRelu	Dieser Layer wendet die Formel $f(x) = alpha * x$ für $x < 0$ $f(x) = x$ für $x >= 0$ auf jede Neurone an. Die Ausgabe bewegt sich also nicht zwischen 0 und 1 wie bei dem normalen Relu-Layer, sondern kann auch negative Ausgaben erzeugen.[Kerj]	'input_shape': Das Format der Eigabedaten. 'alpha': Multiplikator für Ausgabe bei $x < 0$.
Dropout	Der Dropout Layer setzt mit anpassbarer Rate zufällige Eingaben auf 0. Dadurch kann vermieden werden, dass sich ein Modell in einem lokalen Minimum festfährt.[Kere]	'rate': Die Rate, mit der Eingaben auf 0 gesetzt werden.

Embedding	Embedding Layer transformieren Eingaben mit positiven Integer-Werten in Float-Vektoren fester Größe. Dieser Layer kann nur als erste Schicht in einem Modell verwendet werden.[Kerf]	'input_dim:' Das Format der eingehenden Daten. 'output_dim': Das Format, zu dem die eingehenden Daten konvertiert werden sollen.
Concatenate	Dieser Layer vereint eine Liste an Eingabetensoren. Die Eingaben müssen dabei in allen Dimensionen außer der Verbindungsebene die selben Längen haben.[Kera]	'axis': Index der Dimension, die als Verbindungsebene angesehen werden soll.

BatchNormalization	BatchNormalization Layer normalisieren die Werte im Eingabe-Tensor. Der Durchschnittswert der Ausgabe liegt nahe an 0, während die Standardabweichung sich 1 annähert. Dabei lernt der Layer während des Trainings, wie die eingehenden Daten zu verarbeiten sind, um das Erlernte bei späterer Verwendung im finalen Modell entsprechend anzuwenden. Daher kann sich das Ergebnis dieses Layers während des Trainings gegenüber der finalen Version unterscheiden.[Keri]	'axis': Index der Dimension, die als Verbindungsebene angesehen werden soll.

Tabelle 3.1: In Keras integrierte Layer.

'kernel'

Kernel werden die Neuronen eines Layers genannt. Diese Neuronen werden in der Regel für die Verarbeitung der eingehenden Daten genutzt. Die Anzahl dieser Neuronen bestimmen die Möglichkeiten des entsprechenden Layers, da die Menge der Variablen, welche durch den Lernprozess angepasst werden können, davon abhängt.[Tena]

'loss' Funktionen

Sogenannte Loss-Functions berechnen den Loss-Wert des Modells. Dies geschieht nach jedem Durchgang des Trainingsvorgangs und während des Lernprozesses wird versucht, den Fehler bei folgenden Durchgängen zu verringern.[Bro20a] Die Tabelle 3.2 listet die Loss-Funktionen auf, welche in den Experimenten verwendet wurden.[Kerk]

Loss-Function	Beschreibung
MeanSquaredError	Diese Loss-Function bestimmt den Loss eines Modells, indem sie den Durchschnitt der quadrierten Abweichungen aller einzelnen Ergebniswerte errechnet.[Kerm]
BinaryCrossentropy	BinaryCrossentropy wird angewendet, wenn das neurale Netz nur ein binäres Label zu vergeben hat. Der Loss-Wert wird dann aus der Differenz zwischen den errechneten Wahrscheinlichkeiten der Antwortmöglichkeiten und den tatsächlich richtigen Werten bestimmt.[Kerl]
MeanAbsoluteError	MeanAbsoluteError berechnet den Loss als die durchschnittliche, absolute Abweichung aller Ergebniswerte von den tatsächlichen Ergebnissen.[Kern]

Tabelle 3.2: In Keras integrierte Loss-Funktionen.

'activation' Funktionen

Activation-Functions bestimmen in neuralen Netzen die Ausgabe jedes einzelnen Neurons. Sie können die Ausgabe normalisieren und deaktivieren oder aktivieren, entsprechend der Konfiguration und dem zuvor Gelernten.

Da Activation-Functions bei jeder Anwendung des Netzes für jede Neurone ausgeführt werden, müssen sie effizient gestaltet sein, um die Gesamtdauer der Ausführung entsprechend niedrig zu halten. Beispiele für häufig verwendete Activation-Functions sind 'ReLu' und 'Sigmoid', welche auch in Keras enthalten sind und genutzt werden könenn. [Misnta]

'bias'

Das sogenannte Bias-Neuron verschiebt die Ausgabe einer Activation-Function und wird einzeln auf jedes Ausgabeelement angewendet. Meist wird der Wert der Ausgabe dabei um eins erhöht. Durch das Bias soll in der Regel der Ausgabewert 0 verhindert werden. In diesem Fall wäre die Ausgabe der meisten Aktivierungsfunktionen immer 0, wenn Multiplikation verwendet wird. Durch das Hinzufügen des Bias, kann mit dem Ergebnis sinnvoll weitergerechnet werden. Der Begriff Bias bezeichnet allerdings auch eine generelle Abweichung der Aussagen des Netzes von dem gewünschten Gebiet. Sind die Aussagen allesamt in die selbe Richtung verschoben, so spricht man von einem Bias des Netzes in genau diese Richtung.[Misntb]

'Overfitting' und 'Underfitting'

Wenn die Trainingsdaten ungünstig gewählt sind, kann das resultierende Netzwerk nicht die gewünschten Anforderungen erfüllen. Sind die Trainingsdaten beispielsweise nur aus einem besonderen Teilbereich des Problems, hat das Netzwerk außerhalb dieses Bereiches Probleme beim Verarbeiten der Daten, denn das Vorliegende passt nicht zu dem Erlernten. In diesem Fall spricht man von Overfitting. Sind die Daten, die zum Trainieren des Netzes

verwendet werden, schlecht gefiltert, so kann dies die Fähigkeit des Netzes einschränken, präzise Aussagen zu treffen. Durch das Auslassen weiter Teile des Problems, können keine generalisierten Aussagen bestimmt werden. Dies wird Underfitting genannt.[Misntb]
Abbildung 3.8 zeigt Visualisierungen der beiden Phänomene im Vergleich zu einem erfolgreichen Beispiel.

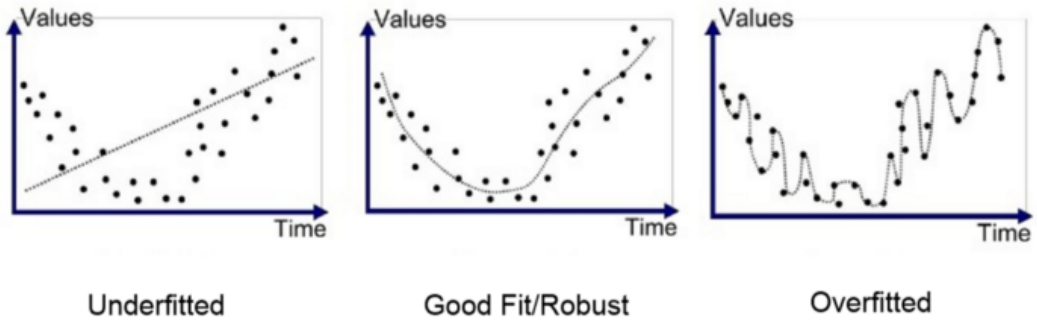

Abbildung 3.8: Over- und Underfitting[Bha18]

'Epoch'

Beim Trainingsvorgang kann derselbe Datensatz mehrfach hintereinander für dasselbe Model verwendet werden. Auch wenn dadurch keine neuen Daten eingeführt werden, können die Lernerfolge dadurch optimiert werden. Einen einzelnen dieser Durchläufe nennt man Epoch. Mit mehreren Epochs in einem Durchgang können manchmal die Resultate des Netzes verbessert werden. Allerdings erhöht sich auch der notwendige Rechenaufwand. Die Effizienz der Durchläufe nimmt mit steigender Anzahl der Epochs meist ab, daher ist es selten sinnvoll, eine sehr große Anzahl Epochs durchzuführen.[All+]

Vorbetrachtung für die Machbarkeit

In diesem Kapitel wird die Lösbarkeit der Teilprobleme im Sinne der Eingangs gestellten Fragestellung betrachtet.

4.1 Nutzung trainierter Modelle auf Mikrocontrollern

Im Rahmen dieser Thesis soll die Umsetzbarkeit eines automatischen Schlagzeugers auf einem Mikrocontroller untersucht werden. Dieses Kapitel widmet sich dem Ansatz, ein durch maschinelles Lernen erstelltes Modell auf einem Mikrocontroller anzuwenden.

4.1.1 Tensorflow Lite auf Microcontrollern

Um Tensorflow Modelle auf einem unterstützten Mikrocontroller einzusetzen, müssen sie zunächst zu Tensorflow Lite Modellen konvertiert werden. Dabei wird nur eine Teilmenge der Tensorflow Funktionalität unterstützt. Das umgewandelte Modell muss dann erneut in einen C-Array konvertiert werden, welcher von Tensorflow Micro auf dem Mikrocontroller interpretiert werden kann. Auch bei dieser Operation wird nur eine Teilmenge der Tensorflow-Lite Operationen unterstützt.[Tenb]

4.2 Experimente

Um Lösungsansätze auf Umsetzbarkeit zu testen und Resultate zu optimieren wurde eine Reihe an Experimenten vorgenommen. Die Trainingsdaten für die Tensorflow Modelle wurden aus dem Datensatz gewonnen, welcher von Aleksey Tikhonov[Tik19] zusammengestellt wurde.

4.2.1 Trainieren eines Modells und Portieren für Tensorflow Lite

Dieses Experiment wurde durchgeführt, um zu prüfen, ob es möglich ist, ein Modell erst von Tensoflow zu Tensorflow Lite und dann zu Tensoflow Micro zu konvertieren und anschließend auf der Hardware des Sequencer Projektes zu implementieren. Dafür wurde ein sehr simples Modell entwickelt.

Aufbau und Vorgehen

Als Modell wurde ein einfaches Netzwerk entworfen, welches aus drei Dense-Layern mit den Kernelgrößen 2, 32 und 1 besteht. Als Eingabe wird ein Tensor mit zwei Float-Werten erwartet. Die Ausgabe des letzten Layers, und damit des gesamten Modells, hat genau einen Float-Wert.
Die Trainingsdaten des Modells wurden mit einer einfachen Funktion generiert, mit zwei zufälligen Floats zwischen 0 und 100000 als Eingabe und der Summe aus den Eingängen als Lösung.
Trainiert wurde das Modell in 20 Epochs und einem Datensatz von 10000 Float-Tupeln. Als Loss-Funktion wurde 'mean_absolute_error' genutzt und 'adam' als Optimizer.

Das resultierende Modell hatte einen hohen Loss-Wert, welcher aber für das Experiment uninteressant ist. Das Resultat wurde mit dem, in Tensorflow integrierten, Tensoflow Lite Konverter als tflite-Datei gespeichert. Dieses wurde mit dem Konvertierer-Skript 'to_c_array.sh' in einen C-Array umgewandelt, welcher von Tensorflow Micro genutzt werden kann.
Für die Implementierung auf dem ESP32 wurde das 'Hello-World' Beispielprojekt aus dem offiziellen Tensorflow Micro Repository modifiziert.[Teant] Das Modell wird in das Programm geladen und in regelmäßigen Abständen mit zufälligen Floats aufgerufen. Dabei wurde die Zeit gemessen, die das Modell zur Berechnung brauchte.
In die modifizierte Version wurde dann der generierte C-Array eingefügt und das Programm direkt auf die Hardware aufgespielt.
Die verwendeten Quelldateien und das erzeugte Modell sind in den beigelegten Dateien unter 'experiments/initial_test' hinterlegt.

Ergebnisse und Erkenntnisse

Auch wenn das resultierende Programm die Lösungsfunktion nicht gut imitiert, ist das Experiment erfolgreich. Die Arbeitsschritte vom Tensorflow-Modell bis zur Ausführung auf der Hardware funktionierten einwandfrei für dieses einfache Beispiel. Die Ausführung dieses Modells erfolgte innerhalb eines Ticks, was bei einer Taktrate von 40MHz weniger als einer Millisekunde entspricht. Nachfolgende Experimente konnten auf dieser Basis aufbauen.

4.2.2 Flaches Netzwerk und boolean Darstellung im Latent Space

Ziel dieses Experiments war das Generieren von Schlagzeugsektionen durch ein Modell. Dafür mussten entsprechende Testdaten gefunden und konvertiert werden. Daher wurde der Datensatz von Aleksey Tikhonov[Tik19] verwendet, in dem Integer-codierte Sektionen mit dazugehörigen Latent-Space und 2-dimensionalen Koordinaten enthalten sind.

Aufbau und Vorgehen

Für dieses Experiment wurden die Trainingsdaten vorbereitet, indem die Integer-Arrays aus der Darstellung zu Arrays mit deren Binärdarstellung umgewandelt wurden. Auf jede soche Zeile folgt dann eine mit der zugehörigen Latent-Space Darstellung. Die zweidimensionalen Koordinaten wurden verworfen. Dazu wurde das Programm 'boolean_flat_latent_generator.py' genutzt.
Das Modell wurde aus drei Dense-Layern mit den Kernelgrößen 4, 256 und 448 gestaltet. Zwischen den Layern und für die Ausgabe wurde die 'relu' Aktivierungsfunktion genutzt. Der zweite und der dritte Layer wurden mit dem 'RandomNormal' Kernel-Initializer initialsiert.
Das Training wurde über 100 Epochs mit 'adam' als Optimizer und 'mean_squared_error' als Loss-Funktion ausgeführt. Dazu wurde 'acc' als zusätzliche Metrik verwendet, um den Trainingsverlauf und die Antworten des Modells besser bewerten zu können. Zur Übersichtlichkeit und Dokumentation wurden anschließend an das Training je ein Bild der Graphen zur Entwicklung der Loss- und Accuracy-Kurven erstellt. Die Quelldateien sowie

einige beispielhafte Resultate sind in den beiligenden Dateien unter 'experiments/bool_flat_latent' zu finden.

Ergebnisse und Erkenntnisse

Das fertig trainierte Modell liefert oft gute Ergebnisse zu zufälligen Werten aus dem Latent Space. Manchmal werden jedoch vollständig leere Spuren ausgegeben.
Die Entwicklung des Loss-Wertes im Verlauf des Trainings lief gut. Die Skala der Grafik, welche auf Abbildung 4.1 zu finden ist, ist durch den starken Rückgang in den ersten Epochs skaliert, sodass die kleineren Entwicklungen im weiteren Verlauf des Trainings nicht gut zu erkennen sind. Über den ganzen Trainingsverlauf ist aber ein eindeutiger Abwärtstrend zu sehen.

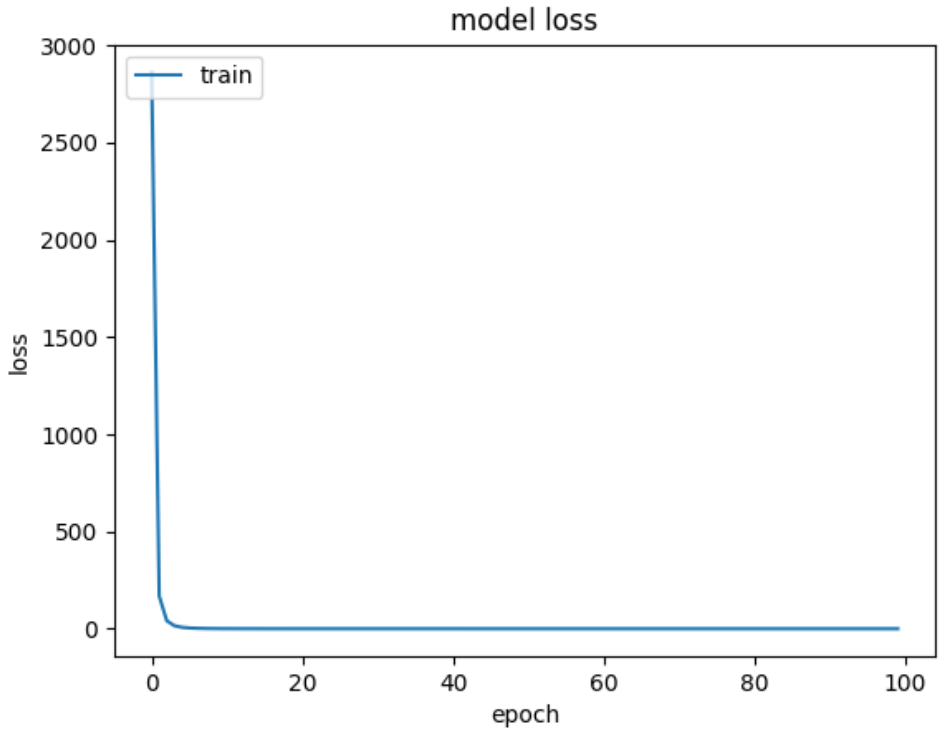

Abbildung 4.1: Entwicklung des Loss-Wertes über das Training mit 100 Epochs

Die Accuracy-Kurve, zu sehen in Abbildung 4.2, entwickelte sich auch bei mehreren Trainingsdurchläufen immer mit einem auffälligen Einbruch ab ungefähr 30 Epochs, von dem sie sich nach 20-40 Epochs schlagartig wieder erholte.

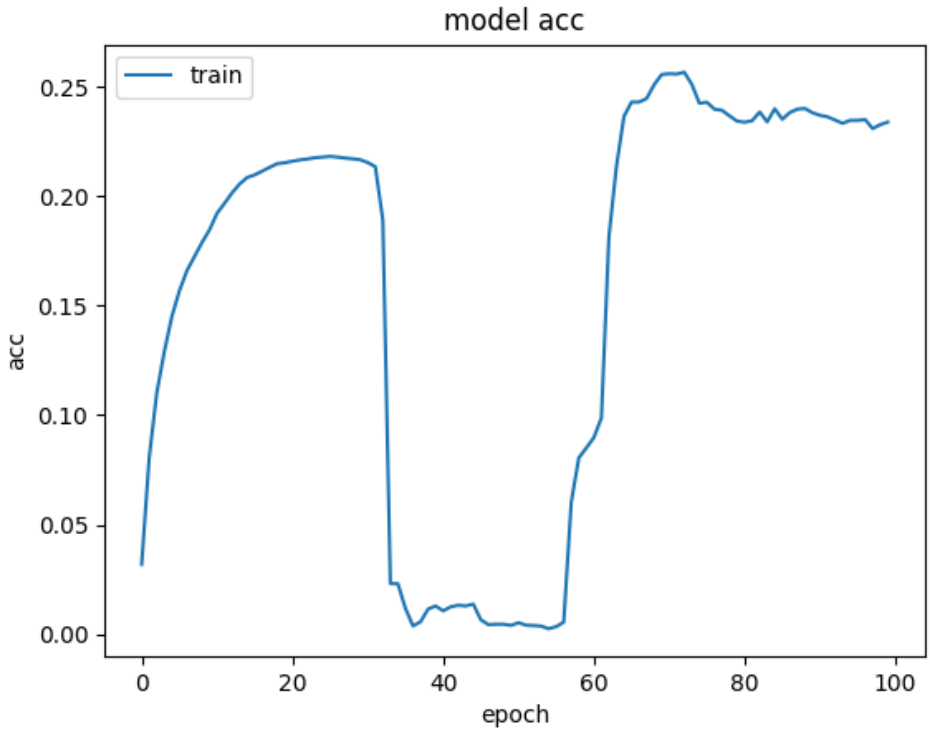

Abbildung 4.2: Entwicklung des Accuracy-Wertes über das Training mit 100 Epochs

Merkwürdig dabei ist, dass es keine Anzeichen auf diesen Einbruch in der Loss-Kurve gibt.

Letztlich lässt sich aus den Ergebnissen dieses Experiments einerseits schlussfolgern, dass ein neurales Netzwerk in der Lage ist, Schlagzeugsektionen aus dem Latent Space zu generieren. Andererseits muss das Modell sowie die Methoden angepasst werden, um die Resultate zu optimieren.

4.2.3 Tiefes Netzwerk und Boolean Darstellung

Dieses Experiment wurde entworfen, um die Auswirkungen eines komplexeren Modells auf die Ergebnisse zu untersuchen. Ausgegangen wird dabei von dem Versuch 4.2.2.

Aufbau und Vorgehen

Das Modell wurde aus einem Dense-Layer mit der Kerrnelgröße vier und sieben weiteren Dense layern mit je 448 Neuronen erstellt. Der letzte Layer hat dabei die Activation-Function 'linear', während der Rest 'relu' nutzt. Initialisiert werden alle, abgesehen vom ersten Layer mit dem 'RandomNormal' Initializer. Als Eingabe werden hier wieder vier Floats erwartet.
Kompiliert wurde das Netzwerk mit dem 'adam' Optimizer und 'mean_squared_error' als Loss-Funktion. Die Testdaten wurden mit dem Skript 'boolean_deep_latent_generator.py' erstellt. Um die Ergebnisse zu optimieren, wurde die Lernrate zwischen den Trainingsabläufen variiert und Dropout-Layer eingefügt. Die Quelldateien und einige Ergebnisse dieses Experiments sind in den beigelegten Dateien unter 'experiments/bool_deep_latent' abgelegt.

Ergebnisse und Erkenntnisse

Im ersten Durchlauf mit der standardmäßigen Lernrate von 0,001 und ohne Dropout-Layer waren die Testergebnisse wenig interessant. Es ließen sich in den Pattern zwar Rhytmen erkennen, jedoch waren nur wenige Steps gesetzt und auch nur auf wenigen Instrument-Spuren. Die Loss-Kurve für den Trainingsvorgang hatte einen vielversprechenden Verlauf, doch die Genauigkeit

der Netzes sank über die Epochs erheblich. Grafen zu beiden Verläufen sind auf den Abbildungen 4.3 und 4.4 zu sehen.

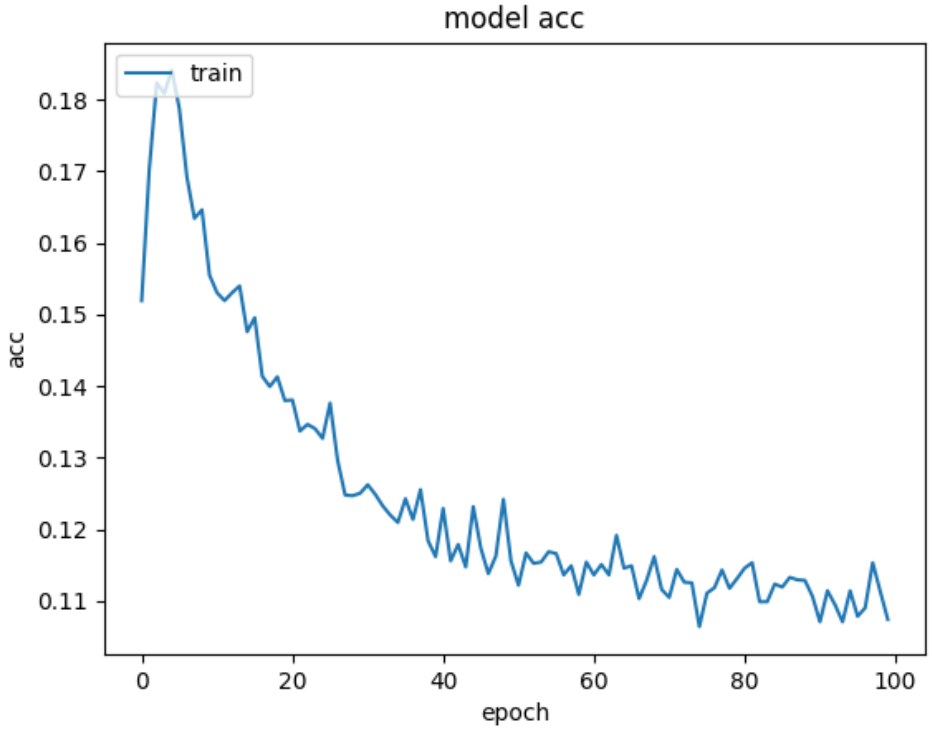

Abbildung 4.3: Entwicklung des Accuracy-Wertes über das Training mit 100 Epochs

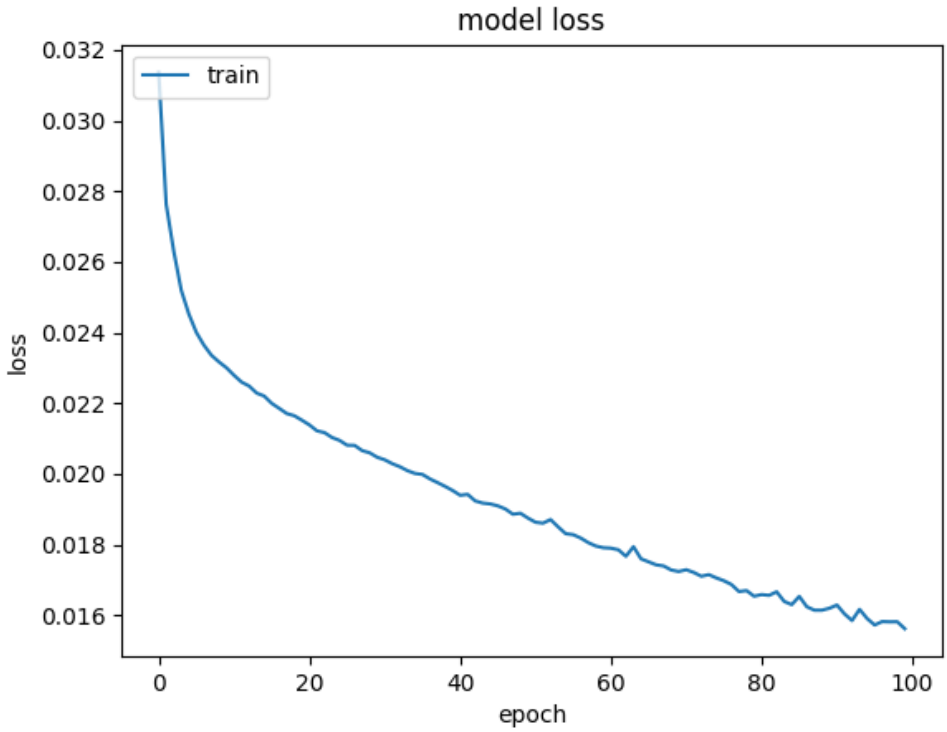

Abbildung 4.4: Entwicklung des Loss-Wertes über das Training mit 100 Epochs

Für den zweiten Durchlauf wurde nach jedem Dense-Layer mit Ausnahme des letzten ein Dropout-Layer mit der Rate 0.2 hinzugefügt. Außerdem wurde die Lernrate auf 0.0001 gesetzt.
Dies resultierte in einer besseren Genauigkeit bei ähnlichem Loss-Wert, welche auf Abbildung 4.5 zu finden ist. Die generierten Pattern waren teils aber immernoch eintönig. Leere Pattern wie zuvor wurden in den Tests aber nicht erzeugt.

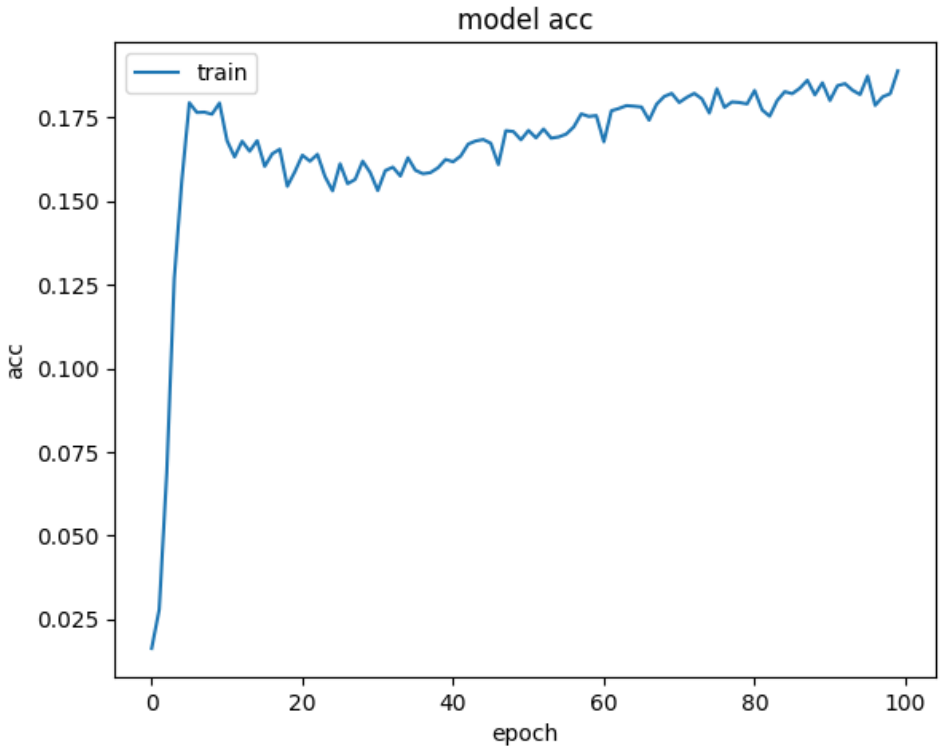

Abbildung 4.5: Entwicklung des Accuracy-Wertes über das Training mit 100 Epochs

Weitere Durchläufe mit Modifikation der Dropoutraten und Lernrate brachten keine erkennbare Verbesserung der Ergebnisse. Dieser Ansatz produzierte also im Vergleich bessere Resultate, aber erreichte immer noch nicht die gewünschte Qualität.

4.2.4 Flaches Netzwerk und Integer-Spalten-Darstellung

Das Ziel dieses Versuches ist es, die Auswirkung einer Veränderung der Struktur der Eingabedaten auf den Lernprozess und die Resultate des Modells zu ermitteln. Inspiriert wurde dieses Format der Eingabedaten durch das Projekt 'Drup Patterns from Latent Space' von Aleksey Tikhonov. In seinem Projekt nutzt Tikhonov diese Daten, um Schlagzeug-Pattern zu erzeugen und aus dem Latent Space in eine zweidimensionale Matrix einzuordnen.

Aufbau und Vorgehen

Für das Experiment wurden die Daten mit je einem Integer pro Spalte formatiert. Dabei stehen die ersten 14 Bit der binären Darstellung der Integer für je einen Step in der Spalte. Das Modell für diesen Testlauf wurde aus 5 Dense-Layern mit 4, 16, 32, 64 und wieder 32 Neuronen gestaltet, wobei überall die Activation-Function 'linear' verwendet wurde. Die Kernel-Initializer sind in allen Layern abgesehen vom ersten auf 'random_normal' gesetzt. Kompiliert wurde auch dieses Model mit dem Optimizer 'adam' und der Loss-Function 'mean_squared_error'. Das Training wurde mit 1000 Epochs ausgeführt. In den beigelegten Dateien unter 'experiments/int_flat_cols_latent' sind die Quelldateien und einige Ergebnisse hinterlegt.

Ergebnisse und Erkenntnisse

Die in den Tests generierten Schlagzeugsektionen lassen zwar teilweise Rhythmen erkennen, allerdings sind zu viele Steps gesetzt und die Anordnung ist chaotisch. Bei Betrachtung der Accuracy-Kurve in Abbildung 4.6 wird klar,

dass kaum eine Verbesserung über die letzten 350 Epochs eingetreten ist. Auch die vorherigen Steigerungen sind sehr gering und die Loss-Kurve in Abbildung 4.7 bleibt beinahe konstant.

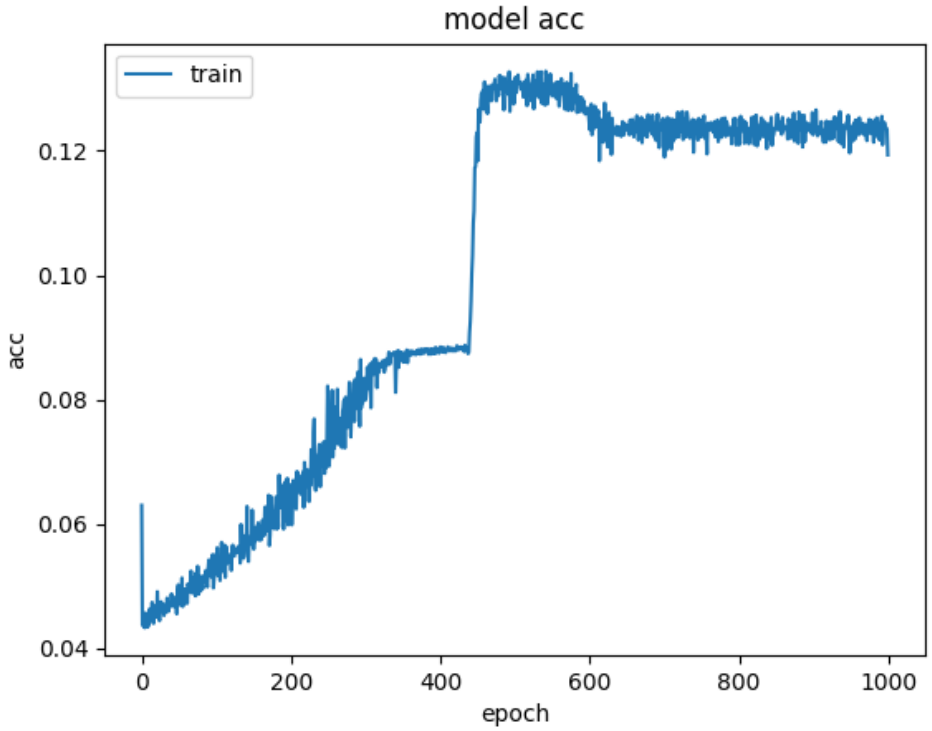

Abbildung 4.6: Entwicklung des Accuracy-Wertes über das Training mit 1000 Epochs

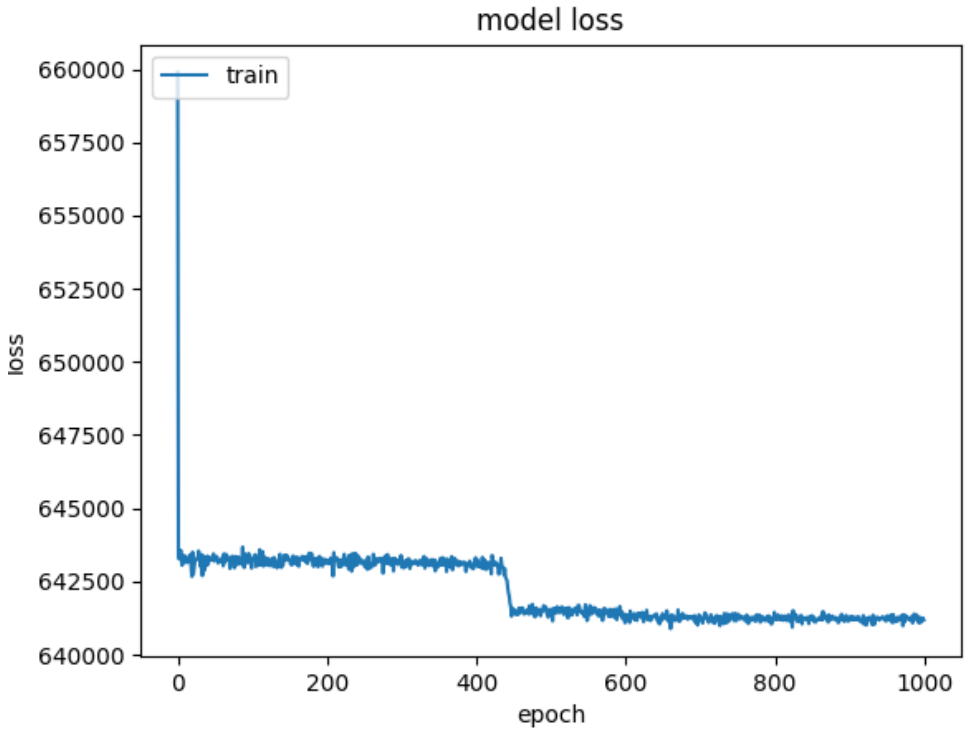

Abbildung 4.7: Entwicklung des Loss-Wertes über das Training mit 1000 Epochs

Diese Beobachtungen legen nahe, dass auch weitere Epochs das Ergebnis nicht signifikant verbessern würden. Es lässt sich schlussfolgern, dass das Model in dieser Form ungeeignet ist, Muster in der gewählten Darstellung zu erkennen. Damit scheint diese Art der Repräsentation der Daten im Zusammenspiel mit dem getesteten Modell nicht zielführend.

4.2.5 Tiefes Netzwerk und Integer-Spalten-Darstellung

Auch dieser Versuch beschäftigt sich mit einer anderen Art der Formatierung der Trainingsdaten. Im Gegensatz zu 4.2.4 wurde das Netzwerk allerdings wesentlich tiefer gebildet, um zu ermitteln, ob dieses Vorgehen die Ergebnisse verbessert.

Aufbau und Vorgehen

Die Formatierung der Daten erfolgte wie im vorausgegangenen Experiment 4.2.4. Das Modell allerdings wurde aus 9 Dense-Layern erstellt. Diese wurden mit der Activation-Function 'linear' versehen und alle, mit Ausnahme des ersten, mit dem 'random_normal' Initializer erstellt. Anschließend wurde das Modell mit dem optimizer 'adam' und der Loss-Function 'mean_squared_error' kompiliert. Das Training erfolgte in 1000 Epochs. Die Quelldateien und einige Ergebnisse befinden sich in den beiligenden Dateien unter 'experiments/int_deep_cols_latent'.

Ergebnisse und Erkenntnisse

Die erzeugten Pattern zeigen ähnliche Eigenschaften auf wie bei dem flachen Modell 4.2.4. Auch hier ist in den Loss- und Accuracy-Kurven, welche in den Grafiken 4.8 und 4.9 abgebiltdet sind, ein Trend abzulesen, der zeigt, dass eine bemerkenswerte Verbesserung der Ergebnisse auch bei einer größeren Anzahl an Epochs nicht zu erwarten ist.

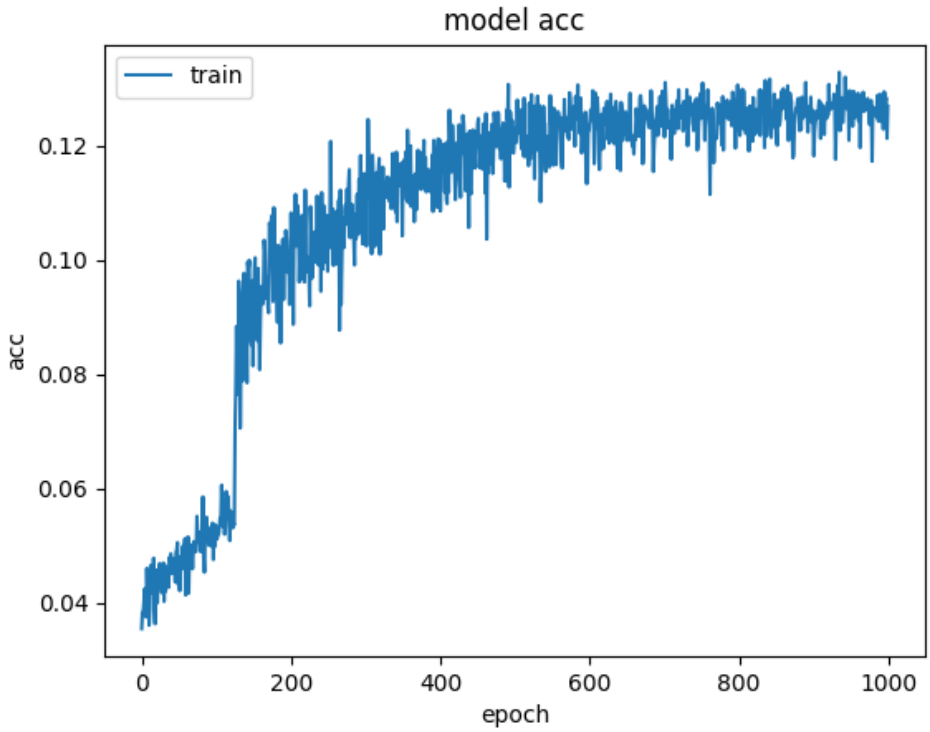

Abbildung 4.8: Entwicklung des Accuracy-Wertes über das Training mit 1000 Epochs

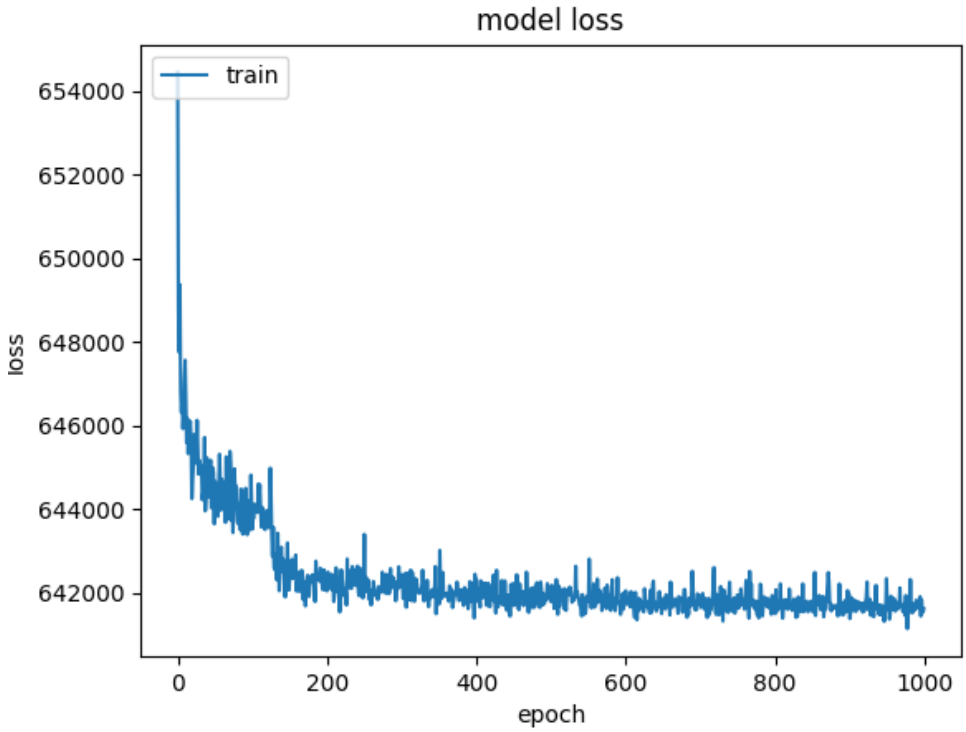

Abbildung 4.9: Entwicklung des Loss-Wertes über das Training mit 1000 Epochs

Diese Experimente zeigen, dass die integerweise Darstellung der Spalten nicht für Modelle dieser Art geeignet ist.

4.2.6 Flaches Netzwerk mit Integer-8-Zeilen-Darstellung und Latent Space Koordinaten

Grundlage für dieses Experiment ist die Beschränkung der Pattern in den Daten auf acht Instrument-Spuren. Diese Minimierung wurde durchgeführt, um den acht möglichen Ausgabekanälen der Hardware des Sequencer Projektes

zu entsprechen. Die resultierenden Pattern sollten also realistischen Resultaten für das Endprodukt ähneln. Zudem wurde die Integer-Darstellung der Zeilen gewählt, statt, wie in den Experimenten 4.2.4 und 4.2.5, der Spalten. Es galt zu prüfen, ob die Minimierung der Größe der Eingabedaten zusammen mit der zeilenweisen Darstellung einem Netzwerk die Erkennung von Mustern erlauben würde.

Aufbau und Vorgehen

Die Trainingsdaten wurden zunächst mit dem Skript '8_int_flat_latent_generator.py' generiert. Dabei wurden erst alle leeren Zeilen aus den Originaldaten ausgelassen. Die nicht-leeren Zeilen wurden dann nachheinander in der Integer-Darstellung aufgereiht. Dann wurden so viele Nullen angehängt, bis die Anzahl der Zeilen acht betrug. Waren in den Originaldaten mehr als acht nicht-leere Zeilen, so wurden die Überschüssigen nicht in die Trainingsdaten aufgenommen. Die so manipulierten Daten, besaßen, im Vergleich zu den Originaldaten weniger Informationen. So ging einerseits die Information verloren, welche Zeile des Patterns welchem Instrument zugeordnet war, da die Reihenfolge abgeändert wurde. Andererseits wurden manche Zeilen vollständig ausgelassen. Dies könnte die Qualität der Ergebnisse der trainierten Netzes negativ beeinflussen oder deren Interpretation erschweren.
Das Modell für diesen Versuch besteht aus vier Dense-Layern mit jeweils 4, 128, 256 und 8 Neuronen im Kernel. Als Activation-Function zwischen den Layern wurde 'relu' verwendet. Für das Kompilieren kamen 'adam' als Optimizer und 'mean_squared_error' als Loss-Function zum Einsatz. Das Training erfolgte über 1000 Epochs. Einige Ergebnisse und

die Quelldateien zu diesem Versuch sind in den beiligenden Dateien unter 'experiments/8_int_flat_rows_latent' hinterlegt.

Ergebnisse und Erkenntnisse

Die Sektionen, die das trainierte Modell bei Tests ausgab, hatten alle genau drei nicht-leere Zeilen. In diesen Zeilen ließen sich teils Ansätze von Rhythmen erkennen, doch waren die Abfolgen nicht melodisch. Die wenigen sichtbaren Rhythmen waren zwischen chaotisch verteilten Steps angeordnet. Diese Beobachtung wurden durch die sehr geringen und über den Verlauf des Trainings sogar abnehmenden Werte für die Genauigkeit der Resultate bestätigt, wie aus Abbildung 4.10 hervorgeht. Auch die Loss-Kurve veränderte sich nach Erfolgen in den ersten Epochs kaum noch merklich und sowohl Genauigkeit als auch Loss-Werte blieben insgesamt schlecht. Die Loss-Kurve ist in Abbildung 4.11 dargestellt. Der Verlauf der Graphen ließ zudem nicht auf eine Verbesserung durch zusätzliche Epochs hoffen. Damit ist klar, dass auch dieses Format der Daten mit dieser Art von Modell nicht zielführend im Hinblick auf das Endprodukt ist.

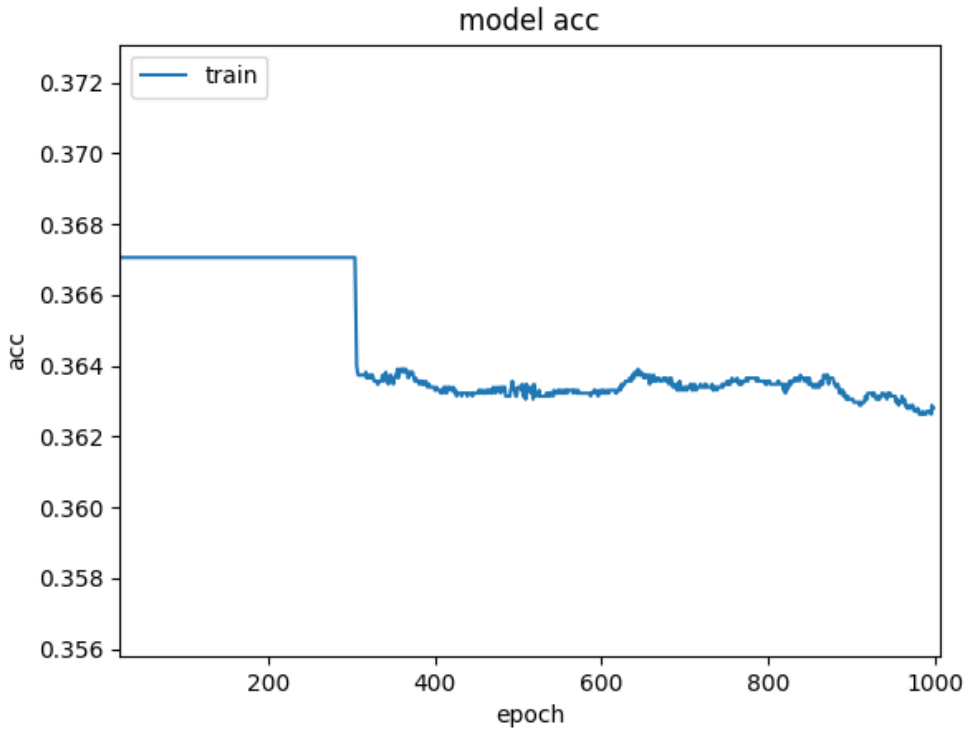

Abbildung 4.10: Entwicklung des Accuracy-Wertes über das Training mit 100 Epochs

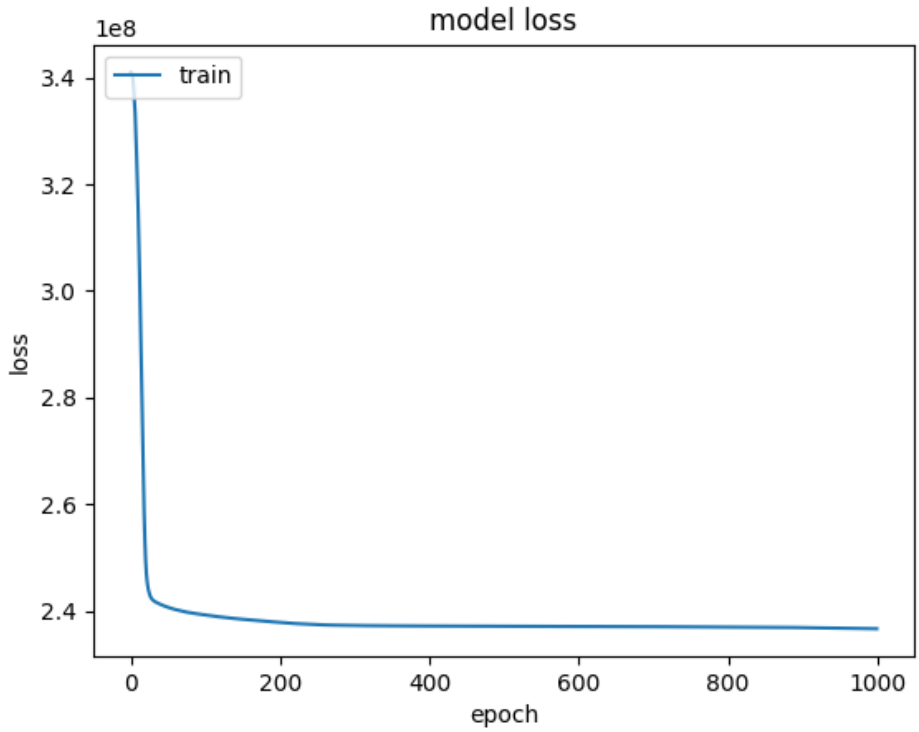

Abbildung 4.11: Entwicklung des Loss-Wertes über das Training mit 100 Epochs

4.2.7 Modifiziertes GAN mit Convolution Layern und 12-Zeilen-Boolean-Darstellung

Die GAN-Architektur sollte sich theoretisch für die Generierung von Schlagzeugsektionen eigenen, da sie auf das Erzeugen von Daten einer bestimmten Kategorie geschaffen ist. Die zufällige Natur der Eingabedaten, also des Seeds für den Generator, die normalerweise in GANs verwendet werden, erlaubt jedoch keine gezielte Steuerung der Ausgabe. Um dieses Problem in GANs

generell anzugehen, wurden cGANs und cDCGANs entwickelt. Ein solches cGAN zu erstellen und zu testen, ist Ziel dieses Versuches. Als Basis für den Programmcode und die Topografie dieses Netzes dient die Vorlage von Jason Brownlee für ein GAN zur Bildgenerierung. [Bro20b]

Aufbau und Vorgehen

Um zusätzlich die wichtigen Eckdaten durch das GAN zu führen, wurden erst die Originaldaten in binärer Darstellung wie in Experiment 4.2.2 formatiert. Dabei wurde die Anzahl der Zeilen wieder auf acht reduziert und, wenn möglich, leere Zeilen verworfen. Dann wurden die dazugehörigen vier Latent Space Koordinaten in binärer 32-Bit Darstellung als zusätzliche Zeilen angehängt, sodass sich eine 12 Zeilen und 32 Spalten große Matrix ergab. Diese Daten wurden dem Discriminator als Originale gegeben.
Als Eingabe für den Generator diente eine Matrix mit 4 Zeilen und 32 Spalten. Jede dieser Zeilen entsprach der binären 32-Bit Darstellung eines zufälligen Floats zwischen -8 und 8. Diese Floats sollten Koordinaten aus dem vierdimensionalen Latent Space darstellen, in den sich die Schlagzeugsektionen einordnen lassen sollten. Diese Matrix wurde durch den Generator dann auf eine Sektion mit den Dimensionen 8 und 32 erweitert. Anschließend wurden die vier Zeilen aus der Eingabe, die die kodierten Latent Space Koordinaten enthielten, an die Ausgabe angehängt. Somit hatte der Discriminator Sektionen mit den dazugehörigen Koordinaten zu beurteilen. Diese Herangehensweise sollte den Discriminator dazu bringen, theoretisch legitime Ausgaben des Generators, die jedoch nicht zu den Koordinaten passten, abzulehnen. Es würde sich zeigen müssen, ob das Format der Daten eine Erkennung von Mustern durch die Modelle begünstigte oder erschwerte.

In der Topografie des Generators folgt auf die Eingabeschicht ein Reshape-Layer, welcher sicherstellt, dass die Daten im beschriebenen Format sind. Die dritte Schicht ist ein LeakyRelu-Layer mit einem 'alpha'-Parameter von 0,2. Dessen Ausgabe wird in einen Conv2DTranspose-Layer gespeist, welcher die Matrix auf 8 Zeilen und 32 Spalten hochskaliert. Dieser Layer hat einen 4 mal 4 großen Kernel und verwendet 256 Filter. Die daraus resultierenden Daten werden durch zwei weitere LeakyRelu-Layer mit denselben Parametern wie der Vorherige geleitet. Anschließend daran verarbeitet ein Conv2D-Layer die Daten mit einer Kernelgröße von 5 mal 5 und 128 Filtern. Dieser Layer nutzt 'tanh' als Activation-Function und behält die Form der Eingabe durch die Verwendung von 'same' als Padding bei. Es folgt ein weiterer LeakyRelu-Layer mit einem 'alpha' Wert von 0,2 und ein Dense-Layer mit der Kernelgröße 1. Dieser hat die Activation-Function 'tanh' und bildet die letzte lernende Schicht des Generators. Der abschließende Layer ist ein Concatenate-Layer, welcher die Ausgabe des letzten Dense-layer mit der Eingabe des Input-Layers zu Beginn zusammenlegt.
Die erste Schicht des Discriminators bildet ein Conv2D-Layer mit 64 Filtern und einer Kernelgröße von 4 mal 4, dazu Strides mit einer Größe von 2 mal 2, 'same' als Wert für die Padding Einstellung und 'random_normal' als 'kernel_initializer'. Dessen Ausgaben werden von einem BatchNomalization-Layer normalisiert und an einen LeakyRelu-Layer mit einem 'alpha' von 0,2 weitergeleitet. Darauf folgt ein Flatten-Layer und darauf wiederum zwei Dense Layer mit 58 und 1 als Kernelgröße sowie 'relu' und 'sigmoid' als Activation function. Der Discriminator wurde mit 'binary_crossentropy' als Loss-Function und 'adam' mit einer modifizierten Lernrate von 0,0002 als Optimizer kompiliert. Um den Trainingsverlauf besser nachvollziehen zu können,

wurde 'accuracy' zu den Metrics des Modells hinzugefügt.
Das vollständige cGAN-Modell wurde dann als eigenes Modell definiert, mit dem Generator-Modell als erstes und dem Discriminator-Modell als zweites Element. Der 'trainable'-Parameter des Discriminators wurde in dieser Konstallation auf 'False' gesetzt, um so sicherzustellen, dass der Discriminator nicht absichtlich Ausgaben des Generator als Original kennzeichnet, um die Ergebnisse des gesamten cGAN-Modells zu verbessern. Dies würde dem Generator eine falsche positive Rückmeldung geben und könnte dadurch die Ergebnisse in ungewollter Art verfälschen.
Das Training verlief anschließend in zwei Phasen pro Epoch. In der ersten wurde eine Menge von Daten erzeugt, die sich zur Hälfte aus Originaldaten und zur anderen Hälfte aus Daten des Generators zusammensetzte. Mit diesem Datensatz wurde in jedem Epoch der Discriminator trainiert. Anschließend wurde mit dem so aktualisierten Discriminator das gesamte cGAN-Modell trainiert. In dieser Phase war es aus dem zuvor beschriebenen Grund wichtig, dass nur der Generator trainiert wurde. Insgesamt wurde das cGAN über 50 Epochs und mit denselben Einstellungen wie der Discriminator trainiert.
Die erstellten Quelldateien und einige Ergebnisse sind in den beiliegenden Dateien unter 'experiments/12_bool_rows_conv_cgan' zu finden.

Ergebnisse und Erkenntnisse

Die Qualität der vom Netzwerk erzeugten Schlagzeugsektionen ist nicht überzeugend. Auffällig ist die Verteilung gesetzter Steps. Meistens gruppieren sich mehrere Steps nahe beieinander über mehrere Zeilen und Spalten hinweg. Ein Rhythmus ist nicht in den Ausgaben zu erkennen. Diese Beobachtungen

legen die Vermutung nahe, dass die Conv2D und Conv2DTranspose-Layer sich nicht für diese Art von Daten eignen. Ursprünglich sind diese Layer für die Verarbeitung von Bildern entworfen. Die vorliegenden Pattern bestehen aber in der Regel aus einzelnen Tönen, die selten an andere, gesetzte Steps angrenzen. Dies könnte die betroffenen Layer im Generator aus der Bahn geworfen haben. Unterstützt wird diese These durch die Loss-Werte des Generators und des Discriminators. Während der Discriminator zum Ende des Trainings einen Loss-Wert von fast 0 aufweist, liegt der des Generators bei mehr als 10. Da dieses Phänomen bereits nach wenigen Epochs zu beobachten ist, könnte der Grund für das Versagen auch sogenannte 'Convergence Failure' sein. [Bro19b] Im nachfolgenden Experiment 4.2.8 werden diese Ansätze weiter verfolgt.

4.2.8 Modifiziertes GAN und 12-Zeilen-Boolean-Darstellung

Nachdem das vorherige Experiment 4.2.7 mit einem modifizierten GAN-Netzwerk keine guten Ergebnisse geliefert hatte, stellte sich die Frage, ob die Schwächen des Netzes durch veränderte Topografie oder Parameter ausgeglichen werden könnten. Dazu sollten die Convolution-Layer im Generator, die eigentlich für die Verarbeitung von Bilddaten erstellt worden sind, durch alternative Layer ersetzt werden. Auch sollten die üblichen Fehlerquellen, die laut Jason Brownlee in seinem Artikel 'How to Identify and Diagnose GAN Failure Modes' bei einem 'Convergence Failure' auftreten können, untersucht werden.[Bro19b]

Aufbau und Vorgehen

Die Eingabedaten wurden genau so formatiert wie in Abschnitt 4.2.7. Der Discriminator funktionierte einwnadfrei und wurde daher auch übernommen. Geändert wurde die Topografie des Generators. Alle Conv2D- und Conv2DTranspose-Layer wurden ausgelassen und stattdessen mehrere Dense-Layer eingefügt. Nach dem Input-Layer werden die Daten durch einen Reshape-Layer in einen eindimensionalen Array umgeformt. Darauf folgen dann 5 Dense Layer, deren Kernelgrößen über mehrere Testläufe variiert wurden. Die Ausgabe des letzten Layers wurde dann durch einen Reshape-Layer in eine Matrix mit 8 Zeilen und 32 Spalten gebracht, und anschließend vor der Ausgabe noch durch einen Concatenate-Layer mit den Eingabewerten zu einer 12 mal 32 Matrix zusammengeführt. Auch geändert wurde der Optimizer des GANs. Statt 'adam' wurde Adagrad eingesetzt. Das Training erfolgte über 50 Epochs.

Ergebnisse und Erkenntnisse

Zuerst wurden die Kernelgrößen der fünf versteckten Schichten auf 256, 512, 1024, 512 und 256 festgelegt. Die Resultate des Durchlaufes waren im Vergleich zum vorherigen GAN-Versuch besser. Es ließen sich einzelne, gesetzte Steps finden und an manchen Stellen ließen sich rhythmische Anordnungen erkennen. Zudem war der Loss des Generators nur bei 5,4 und damit deutlich geringer als beim vorherigen Versuch. Allerdings war der Großteil der Steps immer noch chaotisch angeordnet und ließ keine Muster erkennen. Auch waren nicht übermäßig viele Steps gesetzt.
Im nächsten Aufbau wurden die Kernelgrößen der versteckten Layer auf 512, 512, 256, 256 und 256 gesetzt, da auch zu viele Neuronen diese Art von

Fehler verursachen können.[Bro19b] Zudem wurde die Anzahl der Epochs auf 250 erhöht, um die Auswirkung auf die Ergebnisse zu beobachten. Die resultierenden Schlagzeugsektionen wiesen eine große Ähnlichkeit auf. Das GAN schien, unabhängig von den Eingaben, wenig Variationen in die Daten einzubauen. Dementsprechend waren die Loss-Werte für den Generator im Vergleich zum Discriminator entsprechend hoch. Tatsächlich lagen sie für den Discriminator um 0,001 und beim Generator um 4,8. Diese Werte sind besser als die Ergebnisse des vorherigen Durchlaufs, aber noch nicht optimal. Insgesamt konnte dieser Ansatz zwar die Probleme des Netzes verbessern, aber nicht beheben.
In den beigelegten Dateien unter 'experiments/12_bool_rows_dense_cgan' sind einige Ergebnisse sowie die Quelldateien zu diesem Versuch hinterlegt.

4.2.9 Autoencoder mit vierdimensionalem Latent Space und Integer-Spalten-Darstellung

Dieses Experiment soll die Tauglichkeit eines Autoencoders für die Generierung von Schlagzeugsektionen testen. Dabei sollte der Encoder die Originaldaten in den vierdimensionalen Latent Space einordnen und der Decoder sie aus diesen Daten zu rekonstruieren versuchen.

Aufbau und Vorgehen

Die Eingabedaten wurden zunächst direkt aus den Originaldaten des 'Drum Patterns from Latent Space' Projektes ausgelesen, also wurden die einzelnen Spalten der Sektionen mit Integern dargestellt. Dabei wurden nur die Daten für die Schlagzeugsektionen selber gespeichert, die zugehörigen zwei-

und vierdimensionalen Latent Space Koordinaten wurden verworfen.
Der Encoder wurde aus drei Dense-Layern erstellt, die alle die 'relu' Activation-Function nutzen und je 64, 32 und 4 Neuronen im Kernel haben. Die Eingabewerte werden zu Floats konvertiert, um die Verarbeitung durch die Dense-Layer zu erlauben.
Der Decoder beginnt danach und besteht ebenfalls aus drei Dense-Layern mit 'relu' Aktivierung. Diese haben je eine Kernelgröße von 32, 64 und wieder 32.
Die beiden Modelle wurden zusammen als Autoencoder mit 'adam' als Optimizer und 'mean_squared_error' als loss-function kompiliert. Das Training erfolgte über 250 Epochs. Quellcode und einige Ergebnisse dieses Experiments sind in den beiliegenden Dateien unter 'experiments/int_rows_autoencoder_latent_4D' zu finden.

Ergebnisse und Erkenntnisse

In den Tests zeigte sich, dass das Modell meist in der Lage war, die Spalten zu erkennen, in denen Steps gesetzt wurden. Allerdings waren die jeweils gesetzten Steps im Vergleich zur Eingabesektion nicht in den richtigen Zeilen oder es gab zu viele. Das Problem schien in der Wahl der Loss-Funktion im Zusammenhang mit der Darstellung der Daten zu liegen. Durch 'mean_squared_error' wird der Loss exponentiell geringer, sobald der reelle Unterschied zwischen dem vorhergesagten Wert und dem in den Originaldaten unter 1 liegt. Damit wird ein Unterschied von beispielsweise 0,6 bereits als geringer Loss verrechnet. Problematisch ist, dass, bei der Darstellung der Spalten als Integer, ein solcher Unterschied durch die Rundung das tatsächliche Ergebnis enorm verschlechtern kann. Dadurch können Rundun-

gen eintreten, die bei einem geringen Unterschied der Ausgabezahl eine große Auswirkung auf die resultierenden Sektionen haben können. Beispielsweise könnte eine Spalte mit dem tatsächlichen Wert 1024 auf den Wert 1023,4 geschätzt werden. Der quadrierte Fehler wäre hier mit 0,12 sehr gering. Die Binärrepräsentationen jedoch unterscheiden sich stark, da 1023, 4 auf 1023 gerundet würde.

$$1024 = 00000000001$$
$$1023 = 11111111110$$

Diese Theorie wird durch die Accuracy- und und Loss-Graphen auf den Abbildungen 4.12 und 4.13 gestützt. Beide stellen einen guten Trainingsverlauf mit akzeptablen Werten dar, trotz der mangelhaften tatsächlichen Qualität der Ergebnissse.

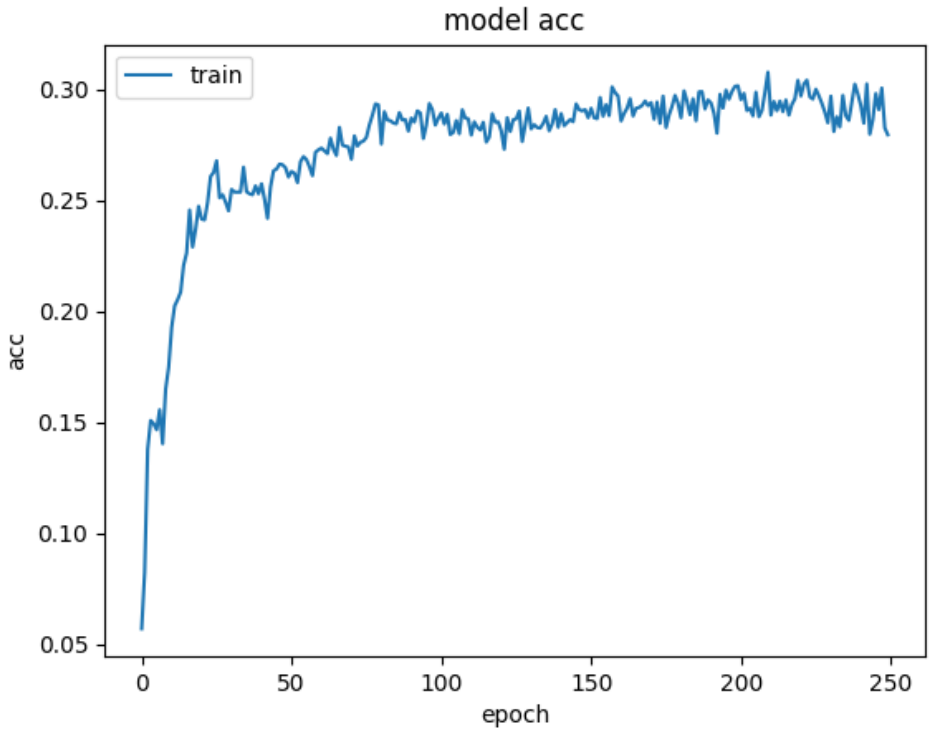

Abbildung 4.12: Entwicklung des Accuracy-Wertes über das Training mit 250 Epochs

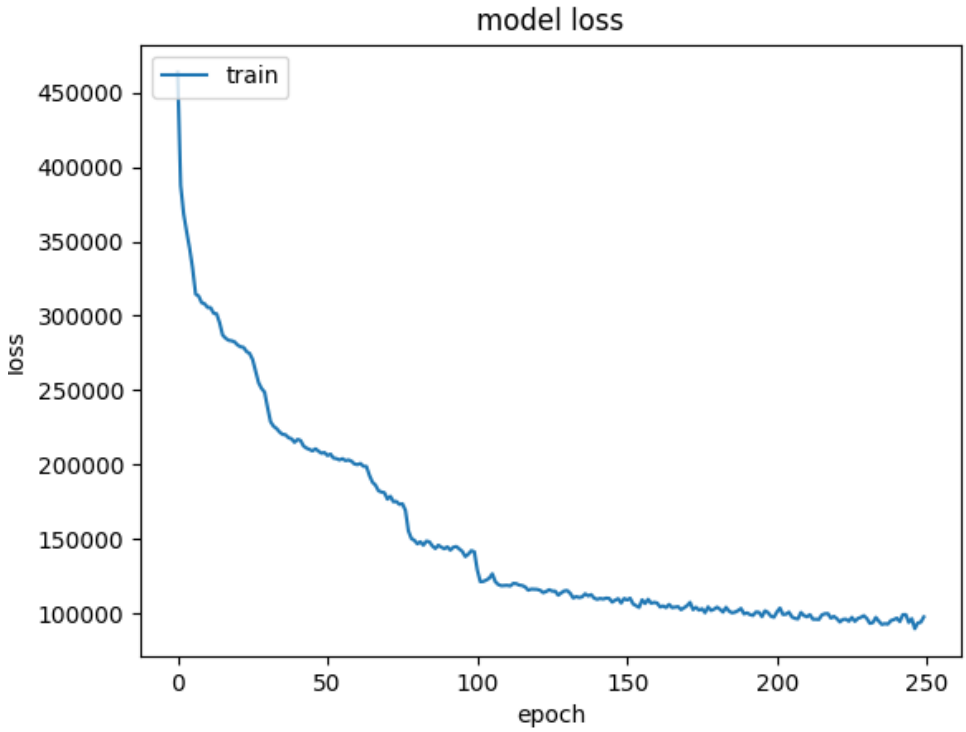

Abbildung 4.13: Entwicklung des Loss-Wertes über das Training mit 250 Epochs

4.2.10 Autoencoder mit zweidimensionalem Latent Space und Integer-Spalten-Darstellung

Um die in Experiment 4.2.9 aufgestellte Theorie bezüglich der Integer-Darstellung im Zusammenhang mit der 'mean_squared_error' Loss-Function zu untersuchen, wurde in diesem Versuch der Aufbau des Experiments variiert. So wurden die Schlagzeugsektionen der Trainingsdaten statt in den vierdimensionalen, in den zweidimensionalen Raum eingeordnet und die Anzahl

der Neuronen in den Kerneln der Dense-Layer variiert. Sollten die Ergebnisse dieselben Schwächen aufweisen wie im ursprünglichen Experiment, so würde dies die aufgestellte Theorie bekräftigen.

Aufbau und Vorgehen

Die Eingabedaten wurde genauso formatiert wie im Referenzexperiment. Die Modelle wurden jedoch verändert. Bei gleichbleibender Anzahl an Dense-Layern, wurden deren Kernelgrößen für den Encoder auf 64, 128 und 2 gesetzt und für den Decoder auf 128, 64 und 32. Der Autoencoder wurde wieder aus beiden Modellen zusammengesetzt und mit 'adam' als Optimizer und 'mean_squared_error' als Loss-Function kompiliert. Das Training erfolgte erneut über 250 Epochs. Der Quellcode sowie einige Ergebnisse dieses Versuchs sind in den beiliegenden Dateien unter 'experiments/int_cols_autoencoder_latent_2D' zu finden.

Ergebnisse und Erkenntnisse

Die resultierenden Ausgaben bestätigten die Theorie, dass das Runden der Integer in den Ausgabedaten die Pattern unbrauchbar machte. In der Ausgabe waren klar die zuvor bereits beobachteten Muster zu erkennen. Die Graphen für Loss und Accruacy, zu sehen auf den Abbildungen 4.14 und 4.15, waren auch in diesem Versuch vielversprechend, ebenso wie die absoluten Werte.

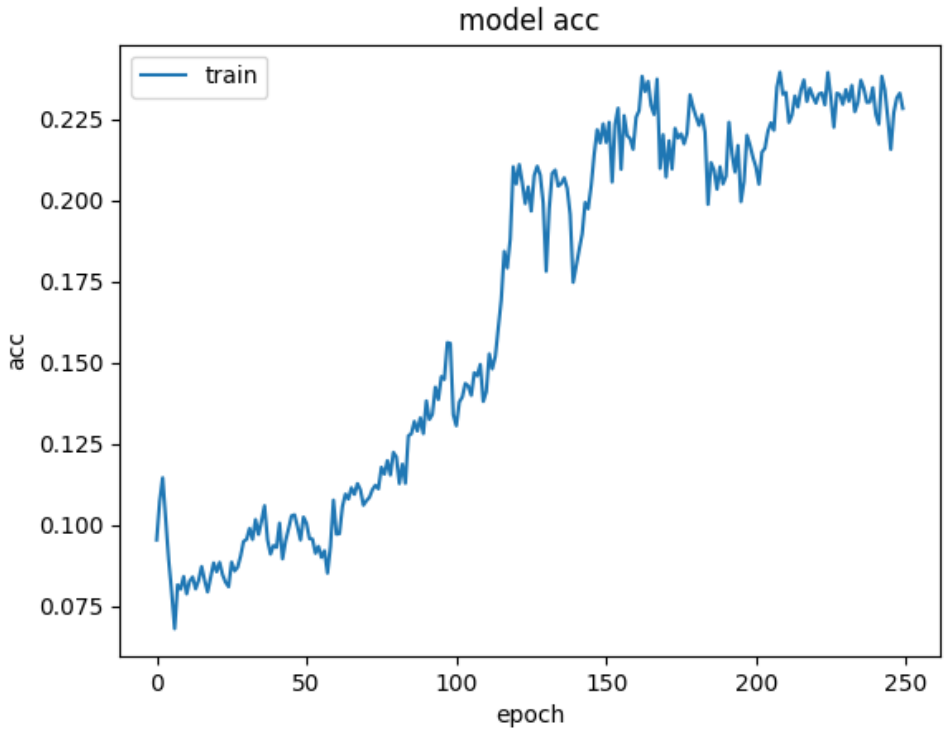

Abbildung 4.14: Entwicklung des Accuracy-Wertes über das Training mit 250 Epochs

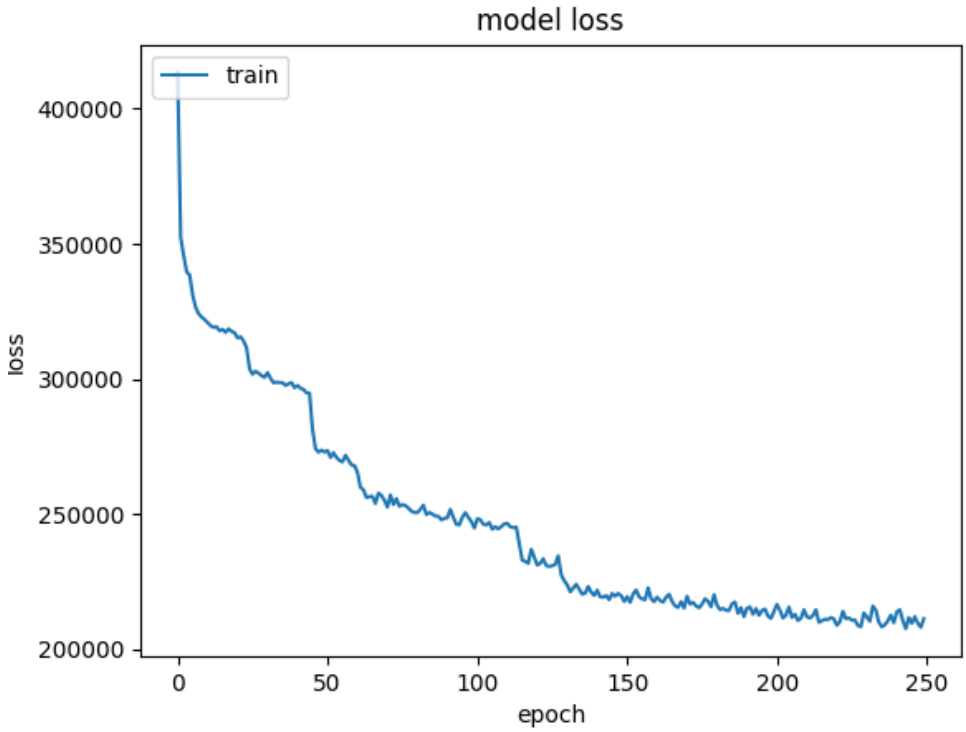

Abbildung 4.15: Entwicklung des Loss-Wertes über das Training mit 250 Epochs

Zusätzlich wurden die besten Ergebnisse erzeugt, wenn das Originalpattern nur die oberen Reihen nutzte, welche die least-significant Bits der Integer abbildeten. Da Rundungsfehler in diesem Bereich am wenigsten schwere Auswirkungen haben, stützt auch diese Beobachtung die Theorie. Es lässt sich also schlussfolgern, dass die Integer-Darstellung für diese Art von Modell nicht geeignet ist.

4.2.11 Autoencoder mit zweidimensionalem Latent Space und Binärdarstellung

Dieses Experiment sollte einen Ansatz testen,, um gegen die Schwächen der Modelle in den vorherigen Experimenten 4.2.9 und 4.2.10 vorzugehen. Der Ansatz bestand darin, die Originaldaten in die Boolean-Form wie in Experiment 4.2.2 zu bringen, um die Problematik des Rundens der Integer zu umgehen. So sollte die Loss-Function 'mean_squared_error' den Fehler für jeden einzelnen Step berechnen und eine bessere Rückmeldung für das Modell erzeugen, aus der dieses dann besser lernen könnte.

Aufbau und Vorgehen

Das Modell für den Encoder wurde für diesen Versuch mit drei Dense-Layern erstellt. Diese wurden mit den Kernelgrößen 512, 1024 und 2 erstellt. Der Input wurde als eindimensionaler Array erwartet, in dem alle Steps als einzelner Float-Wert hinterlegt sind. Der Decoder besteht seinerseits aus drei Dense-Layern mit ja 1024, 512 und 256 Neuronen im Kernel. Alle verwendeten Dense-Layer, abgesehen vom letzten, wurden mit 'relu' als Activation-Function initialisiert. Die Ausgabeschicht des Decoders verwendete hingegen 'sigmoid'. Der Autoencoder wurde durch das Verbinden der beiden Modelle erstellt und mit 'adam' als Optimizier sowie 'mean_squared_error' als Activation-Function kompiliert.
Um die Aussagekraft der Einordnung der Sektionen in den zweidimensionalen Latent-Space durch den Encoder zu visualisieren, wurde im Anschluss an das Training eine Grafik erzeugt, die Paare von Punkten in einer Ebene erzeugte. Erzeugt wurden diese Punkte, indem zwei zufällige Floats zwischen -8 und 8

erzeugt wurden. Diese wurden dann als Eingabe für den Decoder genutzt, um eine Sektion zu erstellen. Diese Schlagzeugsektionen wurde dann durch den Encoder verarbeitet. Die daraus resultierenden Koordinaten wurden in der Grafik mit roten Punkten markiert, die Ursprungswerte mit je einem Grünen. Dies wurde mit 100 Datenpaaren wiederholt und zusammengehörende Punkte mit einer blauen Linie verbunden. Dies sollte die Verzerrung des Latent Space durch das Modell visualisieren. Zudem wurde die durchschnittliche Distanz der Punktepaare berechnet und ausgegeben. Quelldateien sowie einige Ergebnisse zu diesem Versuch sind in den beiligenden Dateien unter 'experiments/bool_autoencoder_latent_2D' hinterlegt.

Ergebnisse und Erkenntnisse

Die Resultate, die der Autoencoder ausgibt, zeigen sehr große Übereinstimmungen mit den Ursprungssektionen, mit meist leichten Variationen. Die Ausgaben zeigen Rhythmus und Variation und sind somit zufriedenstellend.
Die Abweichung der Latent-Space Koordinaten von den Ursprungskoordinaten in der ausgegebenen Grafik ist mit ungefähr 8,25, trotz der guten Ergebnisse, hoch. Auffällig ist hier aber auch, dass Punkte, die nahe beieinander liegen, auch auf eine ähnliche Weise transformiert werden. Zu sehen ist dies auf Abbildung 4.16.

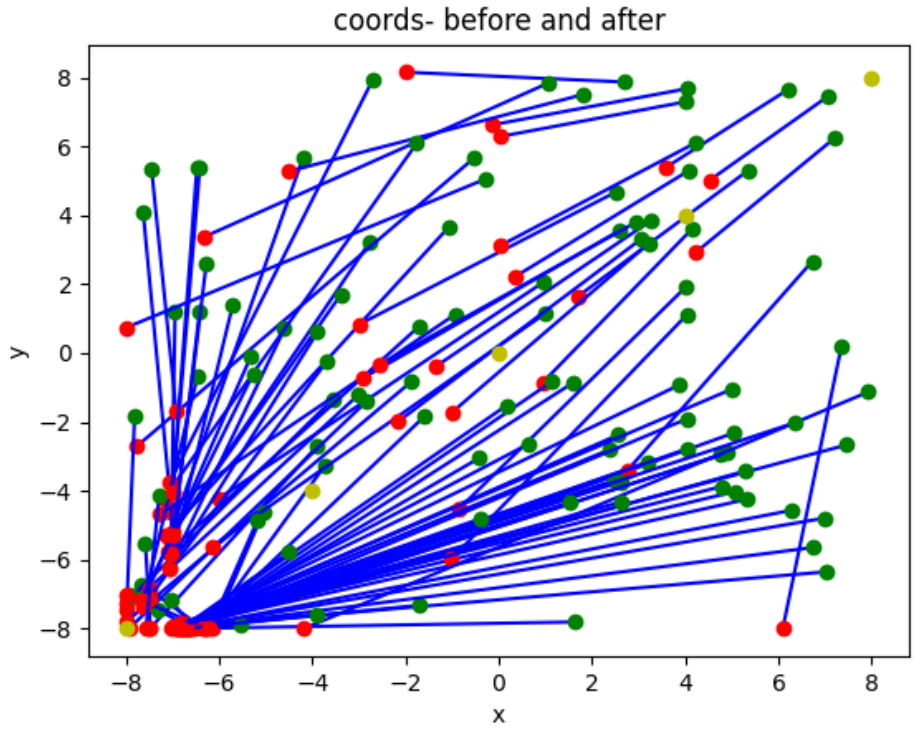

Abbildung 4.16: Entwicklung des Accuracy-Wertes über das Training mit 250 Epochs

Die Loss-Kurve für den Trainingsvorgang auf Abbildung 4.17 zeigt einen guten Verlauf, was auf einen passendes Design der Modelle, der Eingabedaten und des Trainings selber hindeutet.

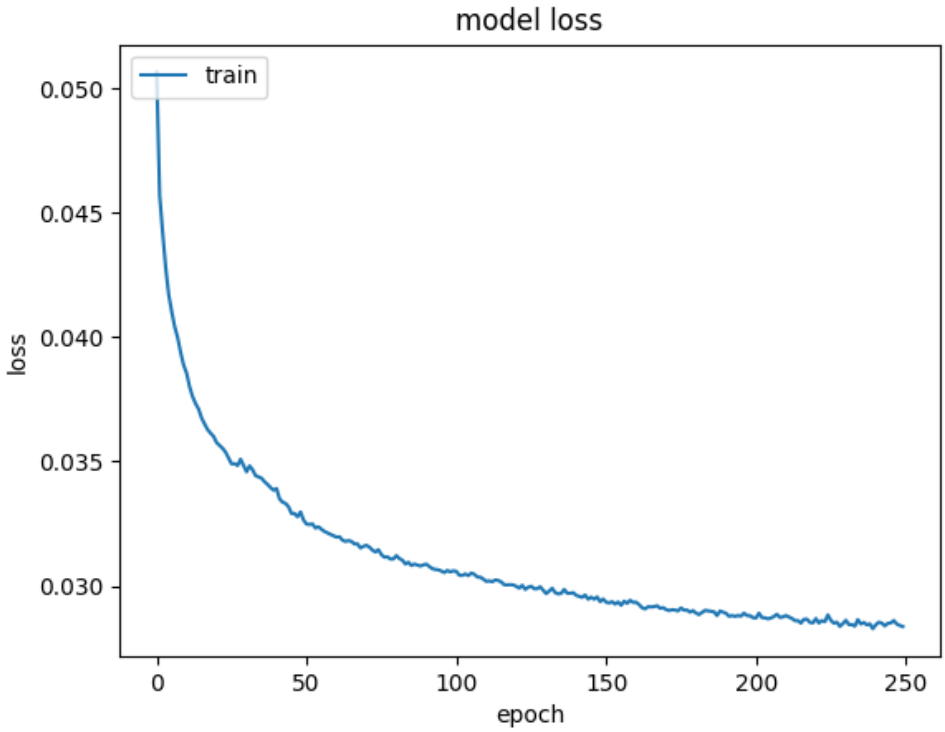

Abbildung 4.17: Entwicklung des Loss-Wertes über das Training mit 250 Epochs

Die Accuracy-Kurve auf Abbildung 4.18 hingegen verläuft, im Hinblick auf die guten Resultate, ungewöhnlich. In den ersten Epochs steig der Accuracy-Wert schnell auf ungefähr 0,225 an, um dann knapp 50 Epochs lang wieder bis auf ungefähr 0,1 abzufallen. Ab da erholte sich der Wert wieder über ungefähr 100 Epochs auf eine Höhe von knapp 0,125.

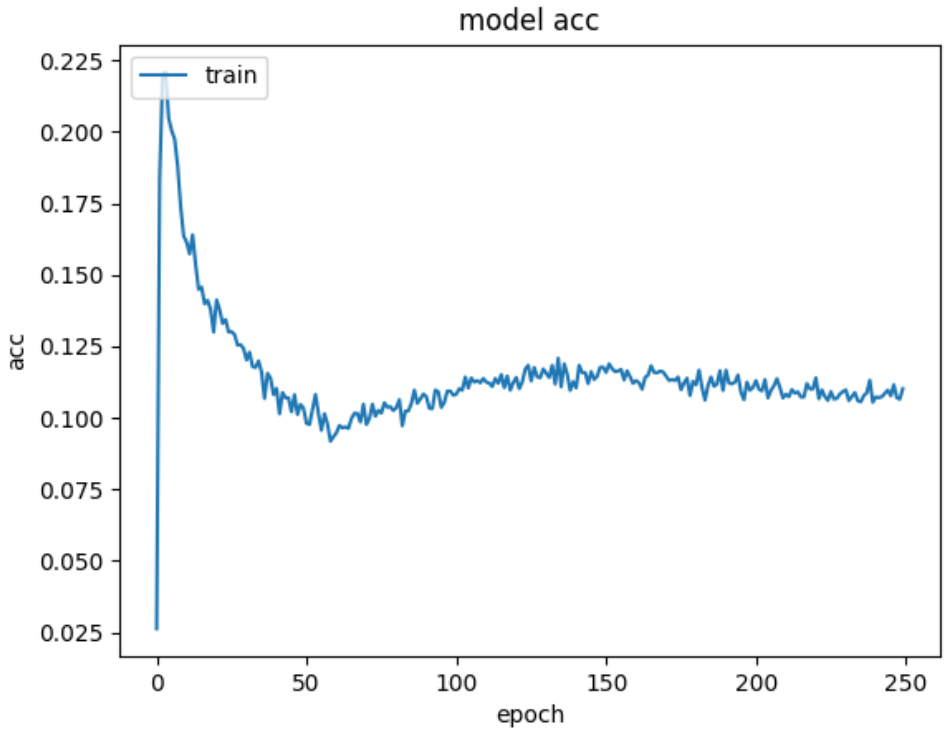

Abbildung 4.18: Entwicklung des Accuracy-Wertes über das Training mit 250 Epochs

Dies sind, absolut betrachtet, keine hohen Werte. Wichtiger für die Bewertung des Modells, sind jedoch Loss-Werte und vor allem die Ergebnisse, daher ist dieses Experiment dennoch als Erfolg zu werten.
Das gute Abschneiden dieses Modells, macht den Ansatz eines Autoencoders mit binären Daten und zweidimensionalem Latent Space zu einem aussichtsreichen Kandidaten für das Endprodukt.

Implementierung und prototypische Ergebnisse

Dieses Kapitel behandelt die Optimierung eines Modells, dessen Konvertierung zu einem C-Array sowie die finale Implementierung auf einem Mikrocontroller.

5.1 Modelloptimierung

Im Kapitel Experimente 4.2 wurden verschiedene Arten neuraler Netzwerke auf ihre Tauglichkeit bezüglich der Fragestellung dieser Thesis geprüft. Die besten Resultate lieferte dabei der Autoencoder in Abschnitt 4.2.11.
Für ein bestmögliches Endprodukt wurde dieser Ansatz zu verbessern versucht. Dafür wurden die Kernelgrößen der Dense-Layer sowohl im Decoder, als auch im Encoder variiert. Die Trainingsdaten wurden genau so formatiert und verwendet wie im Referenzexperiment. Für die Beurteilung der Ergebnisse, wurden Graphen zur Entwicklung der Loss- und Accuracy-Werte über die Epochs erstellt, ebenso wie ein Graph mit der Verschiebung der Koordinaten innerhalb des Latent Space analog zu 4.2.11. Auch wurden zu jeder Variation einige Beispielsektionen generiert.

5.1.1 Topografie mit vergrößerten Kerneln

Die Kernelgrößen der Dense-Layer im Encoder wurden auf 2048, 1024 und 2 festgelegt. Für den Decoder wurden sie auf 2048, 1024 und 256 angepasst. Das kombinierte Modell wurde dann mit 'adam' als Optimizer und 'mean_squared_error' als Loss-Function kompiliert und anschließend über 1000 Epochs trainiert.

Die Resultate des trainierten Netzes waren meistens leer. Die zusätzlichen Neuronen schienen den Lernprozess fast vollständig zu verhindern. Dazu passend veränderte sich der Loss-Wert nach anfänglicher Einstellung nicht mehr, während die Accuracy-Werte über die ersten 250 Epochs insgesamt besser wurden, sich jedoch in den Folgenden 750 Epochs keine Berbesserung mehr einstellte. Diese Entwicklung ist in den Abbildungen 5.1 und 5.2 zu erkennen.

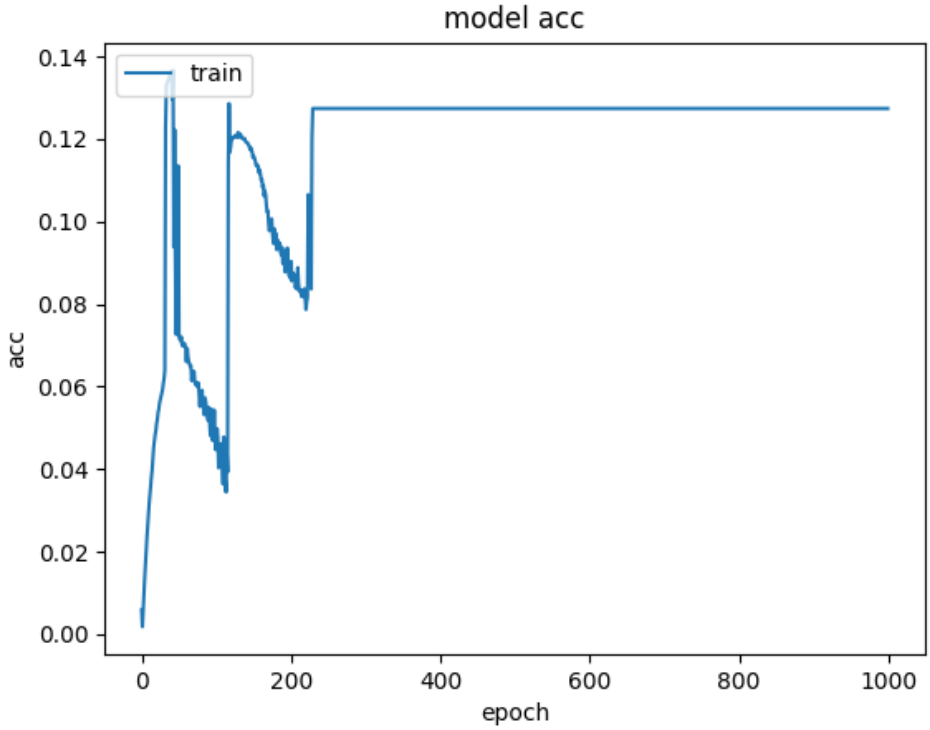

Abbildung 5.1: Entwicklung des Accuracy-Wertes über das Training mit 1000 Epochs

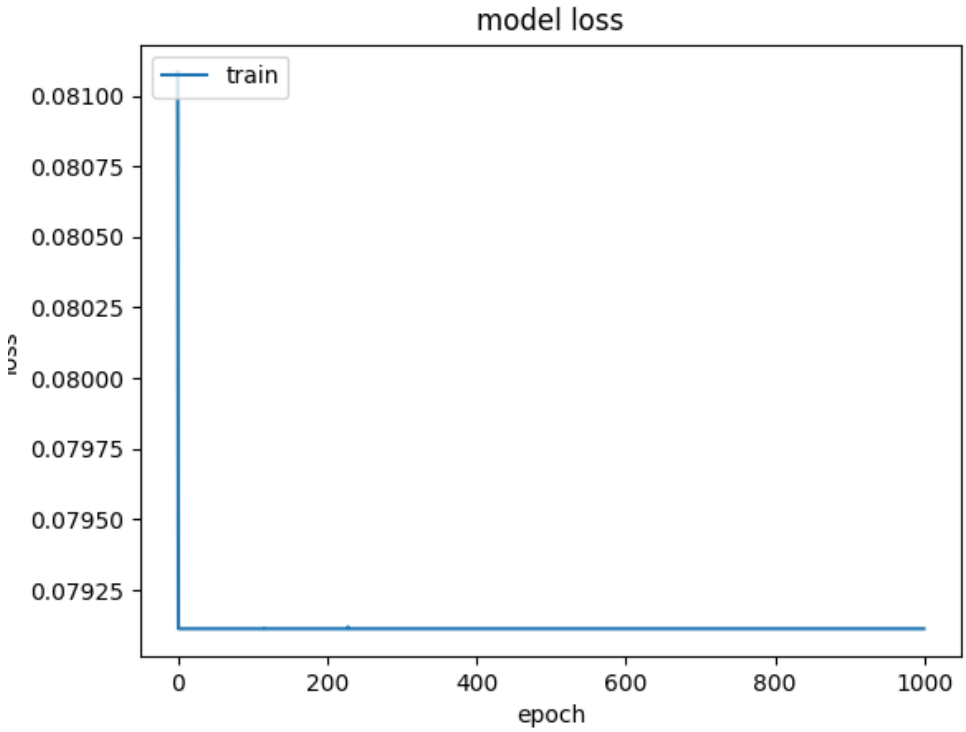

Abbildung 5.2: Entwicklung des Loss-Wertes über das Training mit 1000 Epochs

Auch die starke Verschiebung der Koordinaten im zweidimensionalen Raum, nachdem daraus eine Schlagzeugsektion generiert und diese wieder in den Raum eingeordnet war, deutete auf eine mangelhafte Deutung der Sektionen durch das Modell hin. Visualisiert ist diese Verschiebung in der Abbildung 5.3.

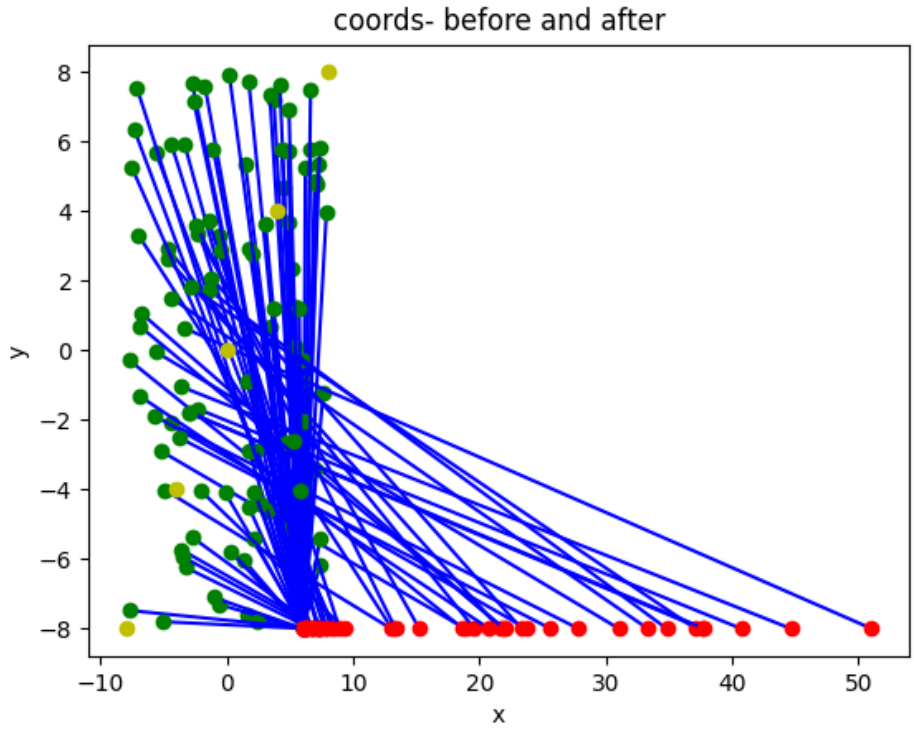

Abbildung 5.3: In blau sind die ursprünglichen Koordinaten zu sehen. Rot sind die Koordinaten nach dem Generieren einer Schlagzeugsektion und dem Einordnen zurück in den Latent Space. Zusammengehörende Punkte sind mit Linien verbunden. Die hellgrünen Punkte sind entlang der Diagonalen von -8 -8 bis 8 8 angeordnet und dienen der Übersichtlichkeit.

Die genannten Probleme zeigten, dass dieser Ansatz ungeeignet ist. Quelldateien und einige Ergebnisse sind in den beigelegten Dateien unter 'implementation/bigger_kernel' zu finden.

5.1.2 Topografie mit verkleinerten Kerneln

Nach der Beobachtung, dass eine deutlich größere Anzahl an Neuronen in den Kerneln den Lernerfolg des Netzes beträchtlich schmälerte, war der Test des gegenteiligen Ansatzes naheliegend. Dafür wurden die Kernelgrößen des Encoder auf 512, 512 und 2 festgelegt, und die des Decoders auf 512, 512 und 256. Kompiliert und trainiert wurde das Modell genauso wie in 5.1.1.
Die resultierenden Schlagzeugsektionen waren von guter Qualität, welche mit der des ursprünglichen Experimentes vergleichbar war. Die Accuracy- und Loss-Werte hatten auch einen ähnlichen Verlauf, zu sehen in den Abbildungen 5.4 und 5.5.

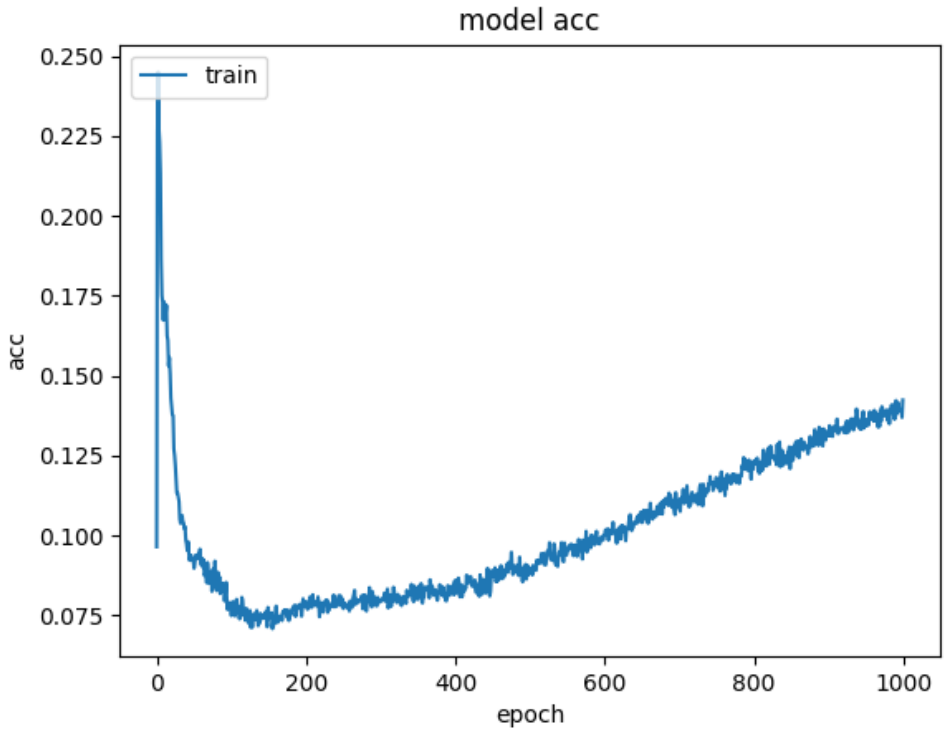

Abbildung 5.4: Entwicklung des Accuracy-Wertes über das Training mit 1000 Epochs

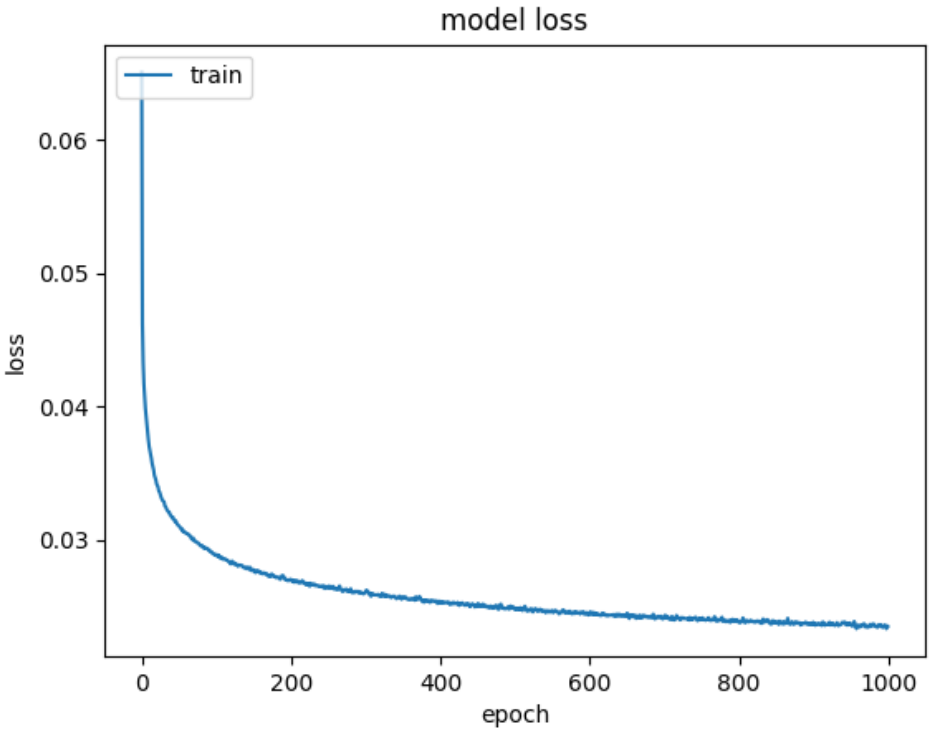

Abbildung 5.5: Entwicklung des Loss-Wertes über das Training mit 1000 Epochs

Die Verzerrung der Koordinaten, wie sie in der Abbildung 5.6 zu sehen ist, hielt sich im Vergleich zu Experiment 5.1.1 auch in Grenzen.

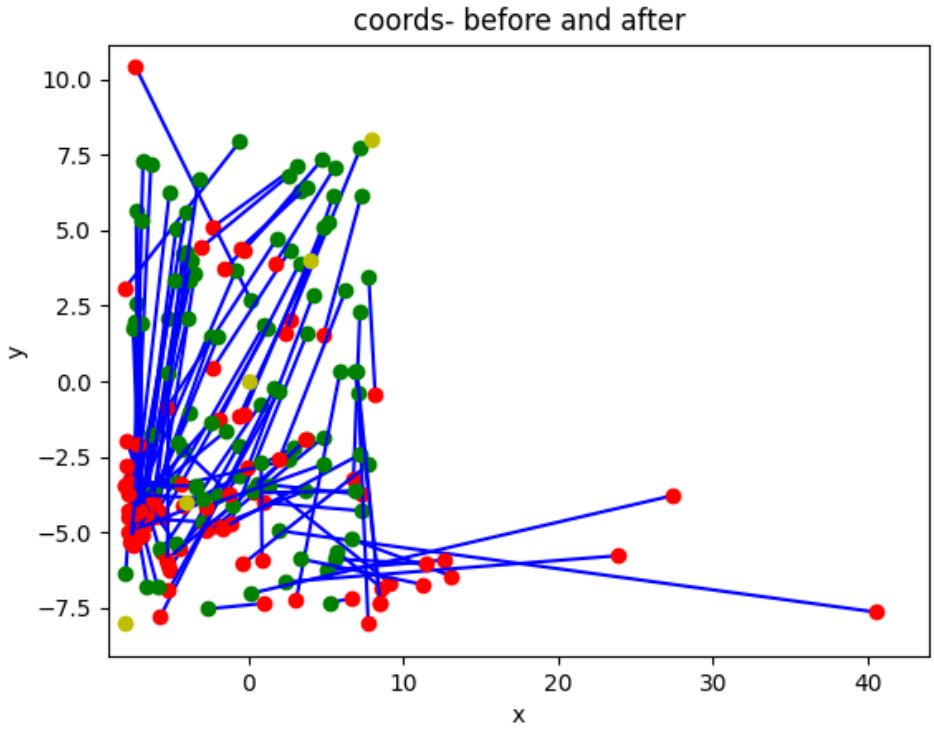

Abbildung 5.6: In blau sind die ursprünglichen Koordinaten zu sehen. Rot sind die Koordinaten nach dem Generieren einer Schlagzeugsektion und dem Einordnen zurück in den Latent Space. Zusammengehörende Punkte sind mit Linien verbunden. Die hellgrünen Punkte sind entlang der Diagonalen von -8 -8 bis 8 8 angeordnet und dienen der Übersichtlichkeit.

Trotz der Reduzierung der Kernelgröße vergleichbare Ergebnisse zu erhalten, bedeutete, dass diese Topografie weniger rechen- und speicherintensiv war. Dies würde für das Endprodukt wichtig sein. Auf dieser Basis wurden weitere Tests zur Optimierung durchgeführt. Einige Ergebnisse dieses Versuchs sowie die Quelldateien sind in den beiliegenden Dateien unter 'implementation/smaller_kernel' hinterlegt.

5.1.3 Topografie mit weniger Schichten und kleineren Kerneln

Um die Größe des Netzwerkes weiter zu reduzieren, wurde je ein Layer aus dem Encoder und Decoder gestrichen. Sollten die Resultate weiterhin qualitativ hochwertig sein, wäre dies eine sinvolle Optimierung des Modells.
Der Encoder bestand nach der Anpassung aus zwei Dense-Layern mit 512 und 2 Neuronen und der Decoder aus ebenfalls Dense-Layern mit den Kernelgrößen 512 und 256. Am Kompiliervorgang wurde im Vergleich zu vorausgegangenen Versuchen nichts verändert und der Autoencoder über 1000 Epochs trainiert.
Die, von diesem Modell erzeugten Schlagzeugsektionen, waren nicht so genau wie noch im Versuch 5.1.2. Im direkten Vergleich mit den Eingaben, waren die Resultate vereinfacht und wiesen weniger interessante Variationen auf als das Ausgangsmodell. Zudem wurden manchmal vollständig leere Schlagzeugsektionen ausgegeben. Die Einordnung der Sektionen in den zweidimensionalen Raum war ebenfalls weniger erfolgreich, wie in der Abbildung 5.7 erkennbar.

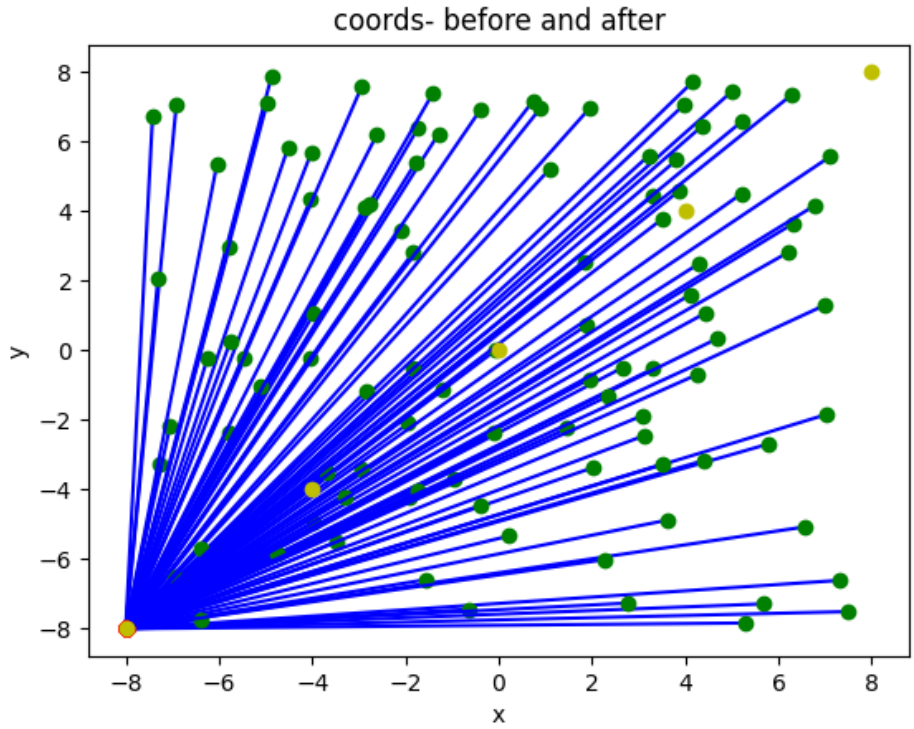

Abbildung 5.7: In blau sind die ursprünglichen Koordinaten zu sehen. Rot sind die Koordinaten nach dem Generieren einer Schlagzeugsektion und dem Einordnen zurück in den Latent Space. Zusammengehörende Punkte sind mit Linien verbunden. Die hellgrünen Punkte sind entlang der Diagonalen von -8 -8 bis 8 8 angeordnet und dienen der Übersichtlichkeit.

Auch aus den Verläufen von Accuracy- und Loss-Werten, zu sehen in Abbildungen 5.9 und 5.8, geht hervor, dass der Lernfortschritt des Modells ab ungefähr dem 200. Epoch nur langsam voranschritt.

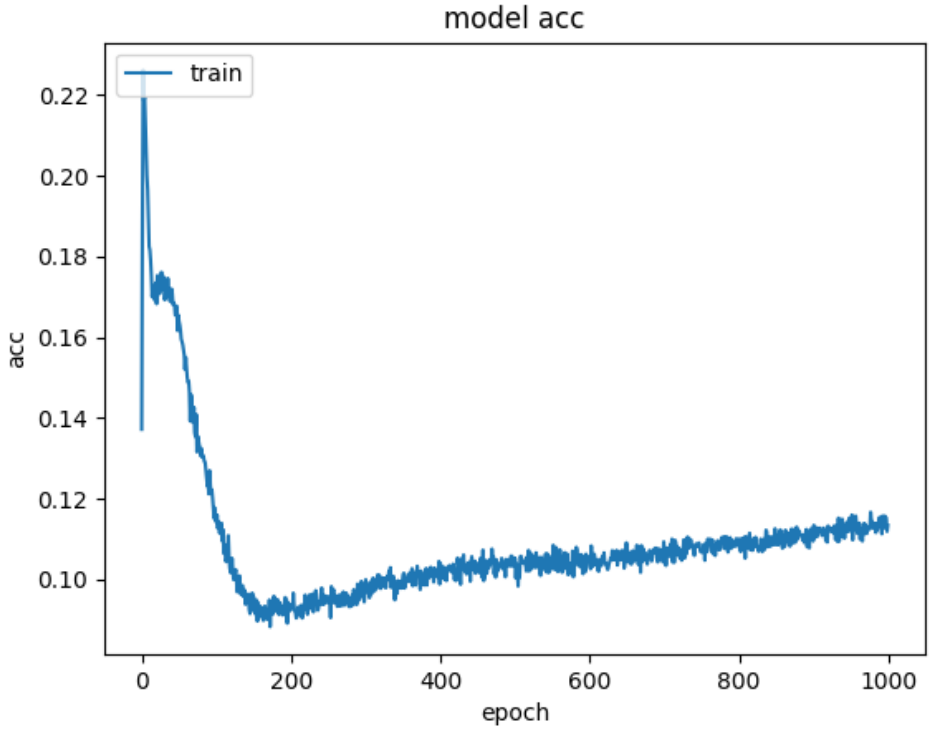

Abbildung 5.8: Entwicklung des Accuracy-Wertes über das Training mit 1000 Epochs

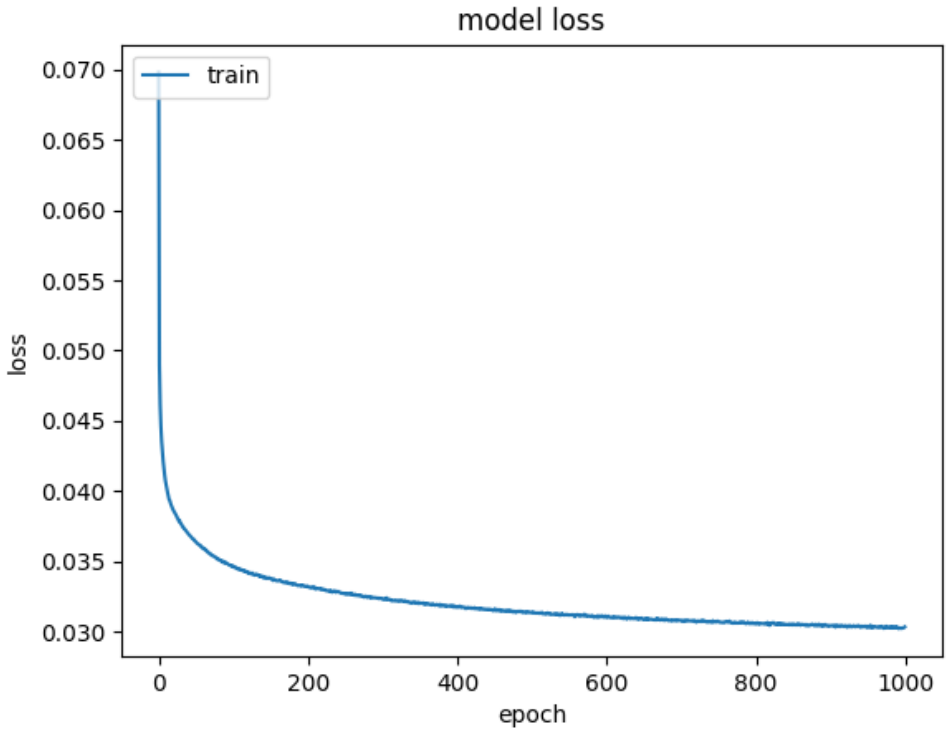

Abbildung 5.9: In blau sind die ursprünglichen Koordinaten zu sehen. Rot sind die Koordinaten nach dem Generieren einer Schlagzeugsektion und dem Einordnen zurück in den Latent Space. Zusammengehörende Punkte sind mit Linien verbunden. Die hellgrünen Punkte sind entlang der Diagonalen von -8 -8 bis 8 8 angeordnet und dienen der Übersichtlichkeit.

Diese Beobachtungen legen nahe, dass ein, in dieser Form verkleinertes, Modell über zu wenige Kapazitäten verfügt, um die Muster in den Daten angemessen zu erkennen und anzuwenden. Eine Modifizierung der Kernelgrößen könnte dieses Problem eventuell beheben. Dieser Ansatz wird im folgenden Abschnitt 5.1.4 geprüft.

Ergebnisse und Quellen dieses Versuchs sind unter 'implementati-

on/less_layers_smaller_kernel' in den beiliegenden Dateien zu finden.

5.1.4 Topografie mit weniger Schichten und ursprünglichen Kernelgrößen

Mit dieser Architektur wurden die Effekte einer größeren Menge an Neuronen in den Kerneln der Dense-Layer bei einer geringeren Gesamtzahl an Schichten getestet. Dazu wurde der Encoder aus zwei Dense-Layern mit 1024 und 2 Neuronen im Kernel konstruiert. Der Decoder erhielt zwei Dense-Layer mit Kernelgrößen von 1024 und 256. Erneut wurde der Autoencoder nach der Kompilierung mit 1000 Epochs trainiert.
Die Schlagzeugsektionen, die das trainierte Modell ausgab, waren näher an den Originalen als noch beim vorherigen Versuch 5.1.3. Doch auch hier ließ sich eine Vereinfachung der Sektionen beobachten, wenn auch weniger ausgeprägt. Die Reduzierung der Anzahl der Layer wirkte sich negativ auf die Fähigkeit der Netze aus, komplexere Details der Sektionen zu erkennen und diese auf die Erzeugnisse anzuwenden.
Verläufe von Accuracy- und Loss-Kurven zeigen, dass der Trainingsprozess besser verlief als beim vorherigen Versuchsaufbau. Dennoch blieben die Werte hinter denen des Modells mit mehr Schichten in Abschnitt 5.1.2 zurück. Die Kurven sind auf den Abbildungen 5.10 und 5.11 zu sehen.

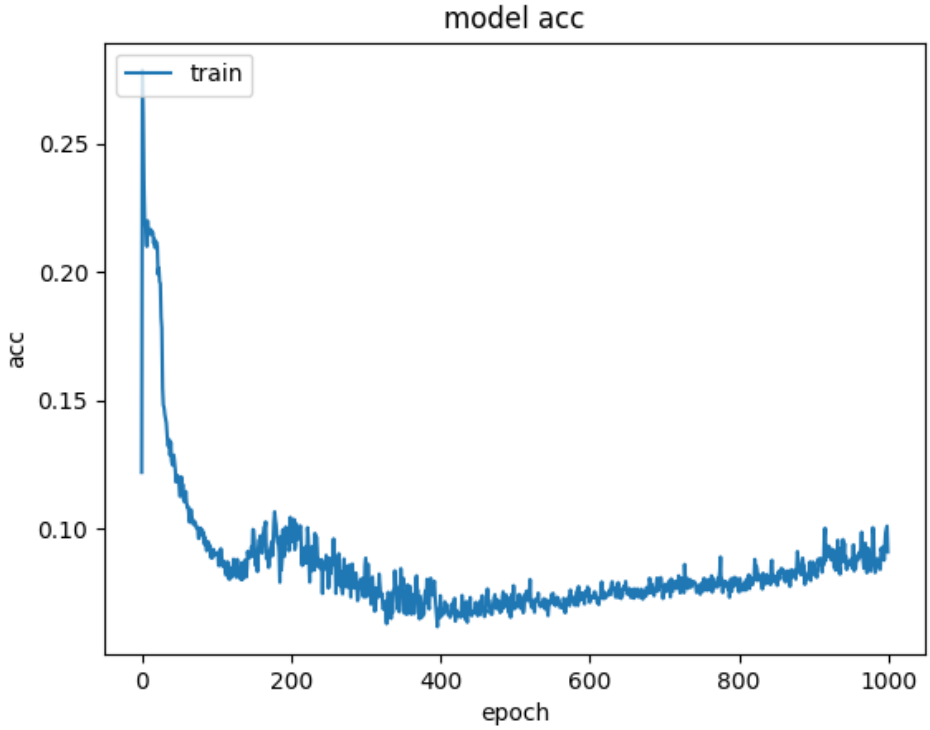

Abbildung 5.10: Entwicklung des Accuracy-Wertes über das Training mit 1000 Epochs

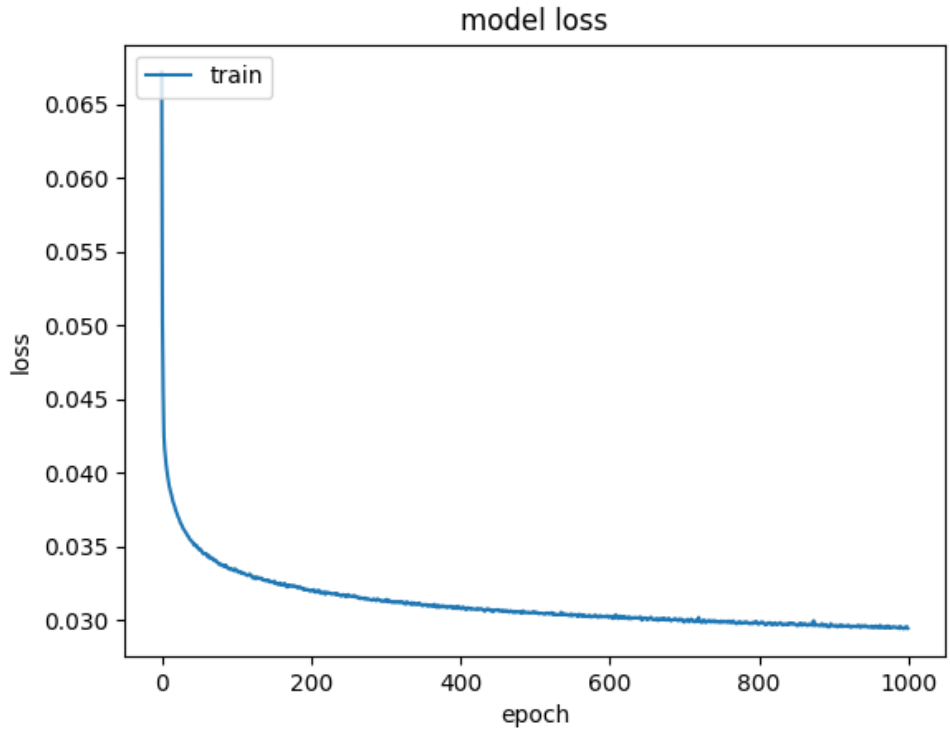

Abbildung 5.11: Entwicklung des Accuracy-Wertes über das Training mit 1000 Epochs

Bei der Einordnung von Schlagzeugsektionen in den Latent Space und dem Erzeugen von Sektionen mit Koordinaten aus eben diesem zweidimensionalen Raum, ließ sich eine Verbesserung der Genauigkeit im Vergleich zum Versuchsaufbau mit geringerer Kernelgröße feststellen. Abbildung 5.12 visualiesiert diese Daten.

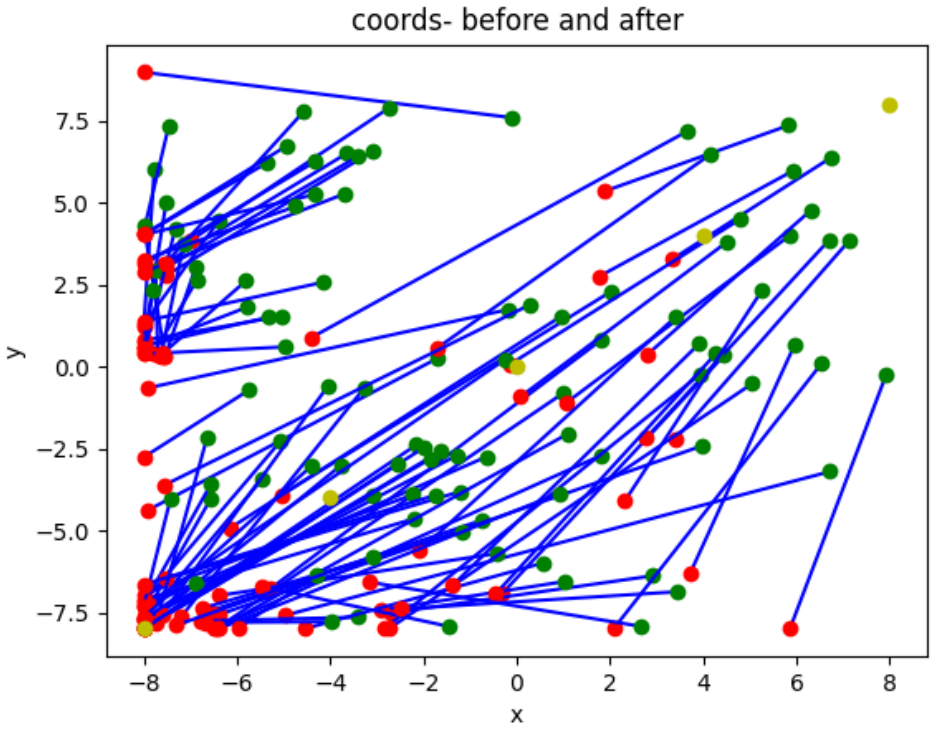

Abbildung 5.12: Entwicklung des Accuracy-Wertes über das Training mit 1000 Epochs

Die Beobachtungen aus diesem Experiment machten deutlich, dass die Reduzierung der Anzahl der Schichten im Autoencoder für die Reduzierung des Rechen- und Speicheraufwandes aufgrund mangelnder Qualität der Ergebnisse nicht sinnvoll ist.
Einige Ergebnisse sowie die Quelldateien zu diesem Test sind unter 'implementation/less_layers_orig_kernel' in den beiliegenden Dateien zu finden.

5.1.5 Topografie mit mehr Schichten und kleineren Kerneln

Der letzte Ansatz zur Optimierung der Topografie des Modells war die Erhöhung der Anzahl der Schichten bei gleichzeitiger Verringerung der Kernelgröße. Dieser Aufbau hatte das Potential, die Speicherbelastung auf der Hardware zu verringern. Für diesen Aufbau wurde die Anzahl an Dense-Layern im Encoder auf vier mit Kernelgrößen von 512, 128, 64 und 2 erhöht. Symmetrisch dazu wurde der Decoder aus vier Layern mit 64, 128, 512 und 256 Neuronen im Kernel konstruiert. Die Kompilierung erfolgte erneut mit 'mean_squared_error' als Loss-Function und adam als Optimizer. Trainiert wurde das Modell über 1000 Epochs.
Die Sektionen, welche das Modell nach dem Training in der Testphase ausgab, waren teils stark vereinfachte Versionen der Eingabedaten. Oft wurden nur wenige Steps gesetzt. Rhythmen ließen sich zwar erkennen, doch im Vergleich zu anderen Ergebnissen waren sie hier weniger interessant gestaltet.
Beim Betrachten der Graphen fiel eine Anomalie zu Beginn des Trainings in den ersten 200 Epochs auf. Diese Auffälligkeit war sowohl in den Loss- wie auch in den Accuracy-Werten zu erkennen. Danach erinnerten die Verläufe der Kurven, welche in den Abbildungen 5.13 und 5.14 zu sehen sind, sehr an die, der vorherigen Versuche.

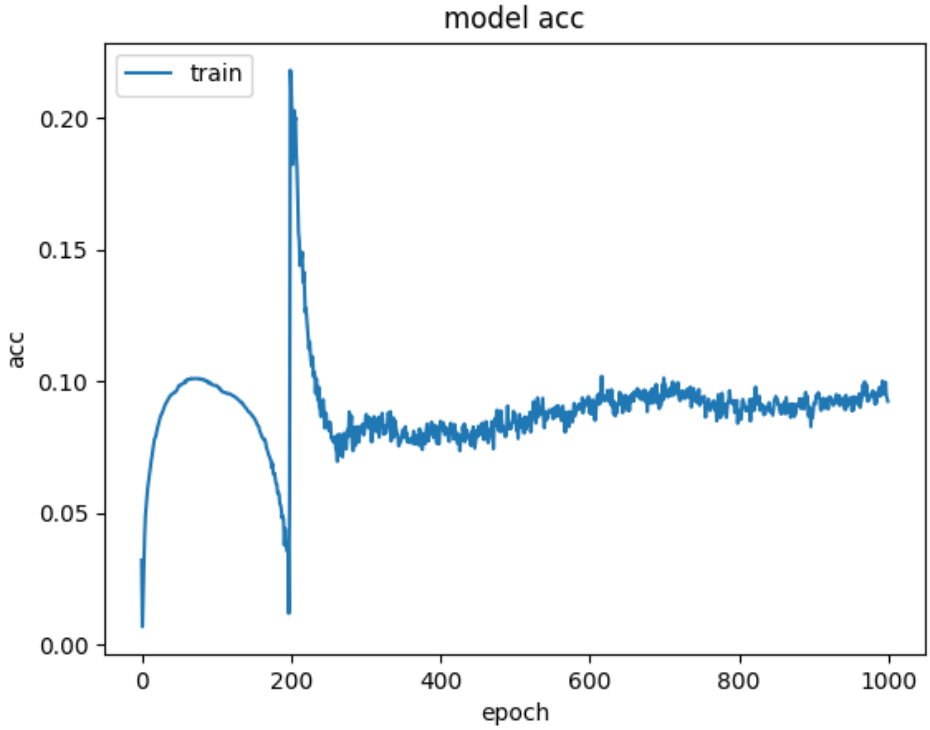

Abbildung 5.13: Entwicklung des Accuracy-Wertes über das Training mit 1000 Epochs

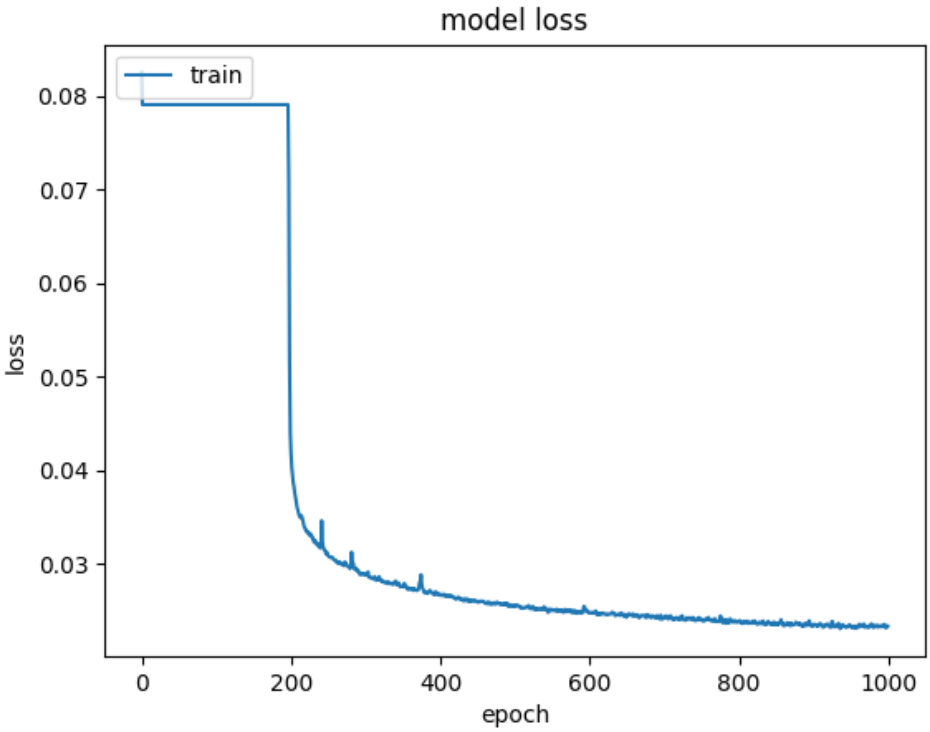

Abbildung 5.14: Entwicklung des Accuracy-Wertes über das Training mit 1000 Epochs

Die Resultate dieses Versuches zeigen, dass die Erhöhung der Anzahl der Layer in Verbindung mit geringeren Kernelgrößen zur Optimierung der Speicherbelastung nicht zielführend ist, da die Qualität der Resultate darunter leidet.
Testergebnisse sowie Quelldateien sind in den beiliegenden Dateien unter 'implementation/more_layers_smaller_kernel' hinterlegt.

5.2 Topografie des finalen neuralen Netzwerkes

Das, aus den durchgeführten Versuchen hervorgegangene, Netz hatte eine Topografie mit insgesamt 6 Dense Layern. Diese wurden mit Kernelgrößen von 512, 512, 2, 512, 512 und 256, gemäß des Versuches in Abschnitt 5.1.2 erstellt. Eine grafische Darstellung des Encoders sowie des Decoders dieses Modells sind auf den Abbildungen 5.15 und 5.16 zu sehen.

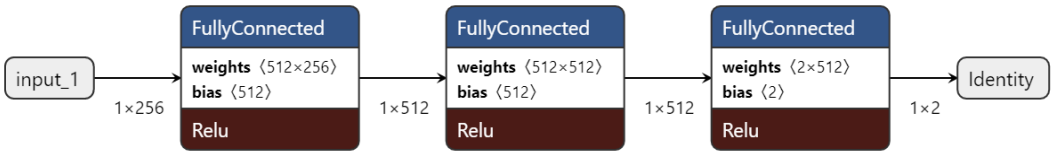

Abbildung 5.15: Modell des erstellten Encoders. Grafik erzeugt mit Netron[Roent]

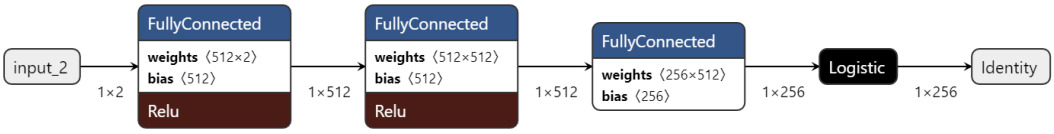

Abbildung 5.16: Modell des erstellten Decoders. Grafik erzeugt mit Netron[Roent]

5.3 Trainingsvorgang

Aus dem Verlauf des Experimentes in 5.1.2 wurde ersichtlich, dass die Accuracy- und Loss-Werte sich zum Ende des Trainings nach 1000 Epochs

noch in einer positiven Entwicklung befanden. Daher wurde die Anzahl der Epochs auf 1500 erhöht. Der Rest des Trainingsvorganges blieb für die finale Version unverändert.

Das Training des Modells verlief wie der vorherige Versuch 5.1.2. Die positive Entwicklung der Accuracy- und Loss-Werte setzte sich fort, wie auf den Abbildungen 5.17 und 5.18 erkennbar. Auch wenn die Entwicklung sich mit weiteren Epochs eventuell fortgesetzt hätte, wurde davon abgesehen, um Overfitting zu vermeiden.

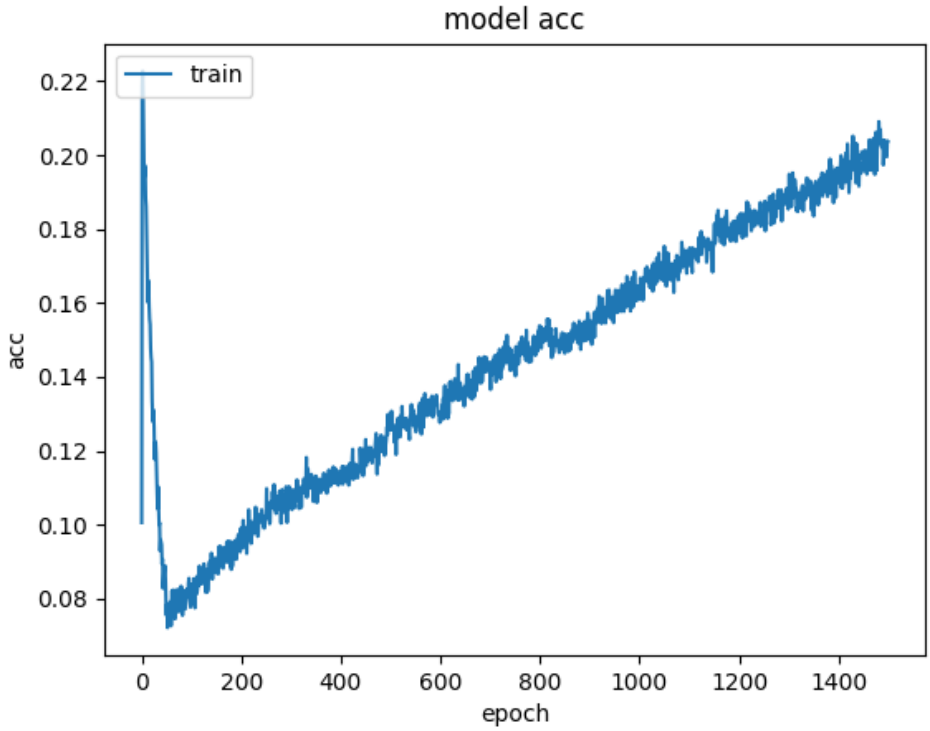

Abbildung 5.17: Entwicklung des Accuracy-Wertes über das Training mit 1000 Epochs

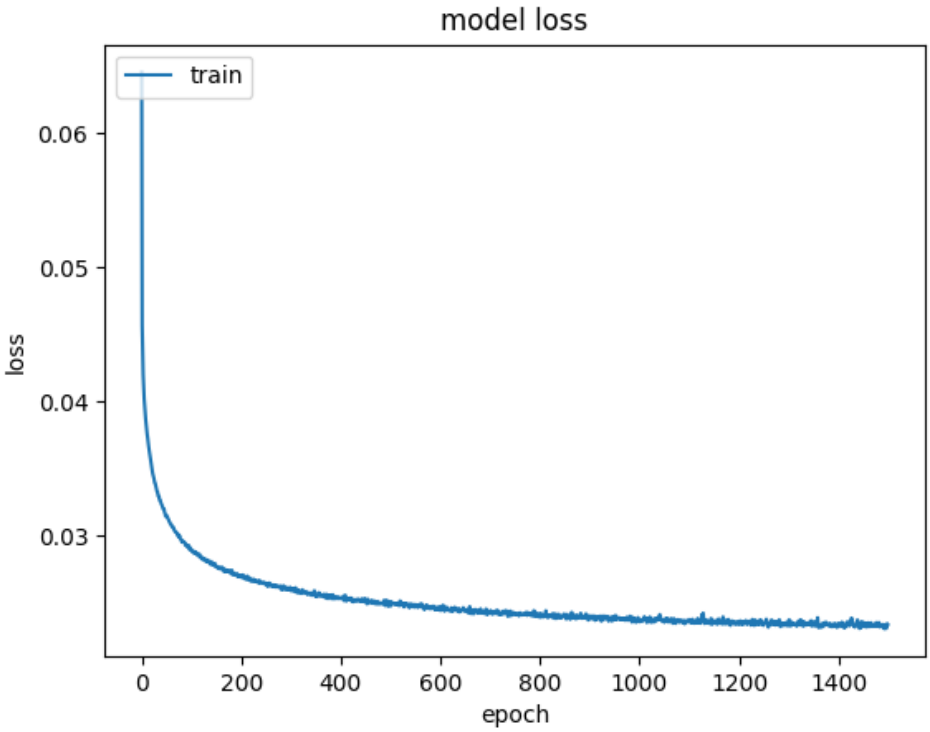

Abbildung 5.18: Entwicklung des Accuracy-Wertes über das Training mit 1000 Epochs

5.4 Test des resultierenden Modells

Der resultierende Decoder, welcher letztlich auf der Hardware des Sequencer Projektes laufen sollte, wurde mit dem Skript 'tflitetester_coord_input_ordered_output.py' getestet, indem Pattern erzeugt und abgespielt wurden.

Um zu visualisieren, wie das Modell den Latent Space nutzt, wurde mit dem Skript 'map_latent_space_usage.py' eine Grafik erzeugt, in der alle blauen

Punkte je ein Pattern aus dem Trainingsdatensatz repräsentieren, welcher vom Encoder-Modell eingeordnet wurde. Diese Grafik ist in Abbildung 5.19 zu sehen.

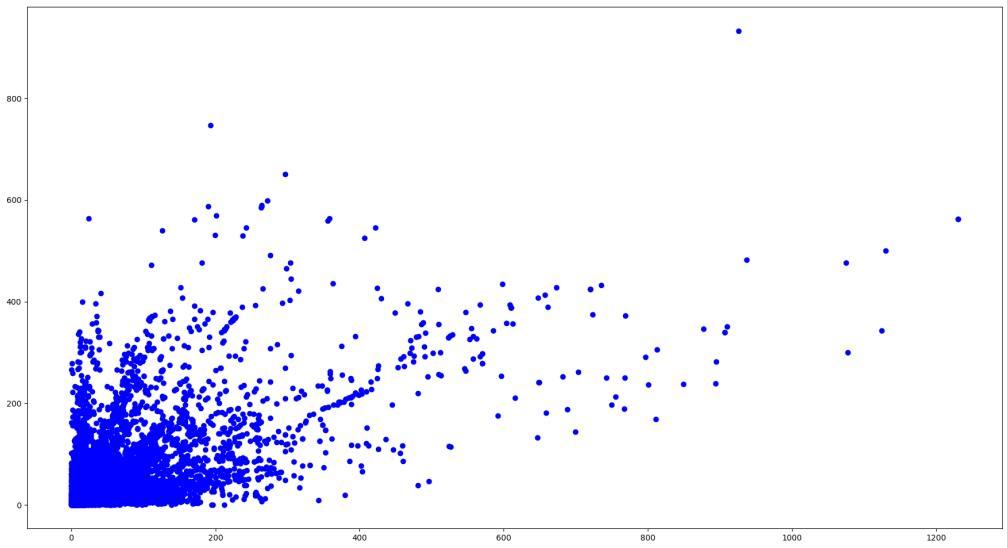

Abbildung 5.19: Jeder Punkt repräsentiert Koordinaten aus dem Latent space, die der Encoder aus den originalen Schlagzeugsektionen generiert hat.

Erkennbar ist eine deutliche Häufung nahe des Koordinatenursprungs. Auch lassen sich mehrere Strahlen, die von diesem Punkt ausgehen, erkennen. Bei der Erzeugung von Pattern durch Eingabe von Koordinaten mit dem Decoder, sollte diese Verteilung berücksichtigt werden. Eingaben, die weit außerhalb der genutzen Verteilung liegen, zwingen das Modell zu starker Interpolation und könnten daher in mangelhaften Ergebnissen resultieren.

5.5 Implementieren des Modells auf der Hardware

Um das erstellte Modell auf den ESP32 aufzuspielen, wurde die erzeugte Tensorflow-Lite Datei mit dem Skript 'to_c_array.sh' in einen C-Array umgewandelt. Dieser wurde dann in eine Kopie des Programmes eingefügt, welches bereits in Experiment 4.2.1 erstellt wurde. In der Flash-Konfiguration für den ESP32 wurde die Variable 'Flash size' auf 4 Megabyte gesetzt und die Partitionsgröße für das Modell auf 2 Megabyte erhöht.
Das Programm auf dem ESP32 generiert jede Sekunde ein neues Pattern und gibt dieses aus. Als Eingabe werden dabei zufällige Koordinaten zwischen 0 und 1250 auf der X-Achse sowie 0 und 950 auf der Y-Achse generiert. Dabei werden die Ticks berechnet, die der ESP32 zur Ausführung des Modells benötigt. Das Programm lief in den Tests wie geplant und benötigte in jedem Durchgang genau 22 Ticks für die Anwendung des Modells auf zufällige Latent Space Koordinaten.
Die Quelldateien und Ergebnisse sowie Zwischenergebnisse zum finalen Resultat sind in den beiliegenden Dateien unter 'implementation/final' hinterlegt.

Beurteilung der Ergebnisse

Die Ergebnisse, die über den Verlauf dieser Arbeit erzeugt wurden, liefern die Antwort auf die Eingangs gestellte Fragestellung. Demnach ist es möglich, mithilfe von maschinellem Lernen auf einem Mikrocontroller Schlagzeugsektionen aus Nutzereingaben zu generieren. Es wurden dafür alle Teilprobleme gelöst und ein funktionsfähiges Beispiel erstellt. Somit wurde das Ziel dieser Arbeit erreicht und die Thesis ist inhaltlich abgeschlossen. Die resultierenden Schlagzeugsektionen beurteilt der Autor im Rahmen seiner Möglichkeiten als angemessen.
Im Hinblick auf das Endprodukt sind für die Zukunft weitere Verbesserungen denkbar, welche im Kapitel 'Fazit und Ausblick' weiter erläutert werden.

Fazit und Ausblick

Das erstellte und implementierte Beispiel ist ausreichend, um die Fragestellung der Thesis in Hinblick auf Umsetzbarkeit zu beantworten. Das finale Produkt sowie dessen Implementation lassen jedoch Raum für Verbesserung. So ist es nicht möglich, eine Schlagzeugsektion zu einem bestimmten Musikstil zu erstellen. Auch andere Eingabeparameter, wie die Tondichte der resultierenden Instrumentspuren, lassen sich nicht durch den Anwender konfigurieren. Zusätzlich könnten die Ergebnisse des finalen Modells durch professionelle Musiker bewertet werden, um die Qualität beurteilen zu können und eventuell Verbesserungen daran vorzunehmen. Auch lassen sich viele der Experimente in dieser Arbeit noch weiterführen und optimieren.

Literatur

[AAnta] Afshine Amidi und Shervine Amide. *Convolutional Neural Networks cheatsheet*. https://stanford.edu/ shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks. Zugriff am: 19.12.2020 13:40. unbekannt.

[AAntb] Afshine Amidi und Shervine Amide. *Recurrent Neural Networks cheatsheet*. https://stanford.edu/ shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks. Zugriff am: 19.12.2020 13:40. unbekannt.

[Aba+16] Martín Abadi u. a. "TensorFlow: A System for Large-Scale Machine Learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, S. 265–283. ISBN: 978-1-931971-33-1. URL: `https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi`.

[All+] JJ Allaire u. a. *Train a Keras model*. https://keras.rstudio.com/reference/fit.html. Zugriff am: 27.12.2020 17:55.

[Bha18] Anup Bhande. *What is underfitting and overfitting in machine learning and how to deal with it.* https://medium.com/greyatom/what-is-underfitting-and-

overfitting-in-machine-learning-and-how-to-deal-with-it-6803a989c76. Zugriff am: 22.12.2020 21:45. 2018.

[BPB17] Johannes Scherk B.Sc, Mag. Gerlinde Pöchhacker-Tröscher und Karina Wagner B.Sc. "Künstliche Intelligenz - Artificial Intelligence". In: (2017).

[Bro19a] Jason Brownlee. *How to Develop a Conditional GAN (cGAN) From Scratch*. https://machinelearningmastery.com/how-to-develop-a-conditional-generative-adversarial-network-from-scratch/. Zugriff am: 21.12.2020 14:30. 2019.

[Bro19b] Jason Brownlee. *How to Identify and Diagnose GAN Failure Modes*. https://machinelearningmastery.com/practical-guide-to-gan-failure-modes/. 2019.

[Bro20a] Jason Brownlee. *How to Choose Loss Functions When Training Deep Learning Neural Networks*. https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/. Zugriff am: 22.12.2020 14:30. 2020.

[Bro20b] Jason Brownlee. *How to Develop a Conditional GAN (cGAN) From Scratch*. https://machinelearningmastery.com/how-to-develop-a-conditional-generative-adversarial-network-from-scratch/. 2020.

[Dav+20] Robert David u. a. *TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems*. 2020. arXiv: 2010.08678 [cs.LG].

[Der17] Arden Dertat. *Applied Deep Learning - Part 3: Autoencoders*. https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798. Zugriff am: 22.12.2020 14:25. 2017.

[DPnt] Monica Dinculescu und Jérôme Petazzoni. *DrumBot*. https://magenta.tensorflow.org/groovae. unbekannt.

[Esp20] Espressif. *ESP32 wrover-b datasheet*. https://www.espressif.com/sites/default/files/documentation/esp32-wrover-b_datasheet_en.pdf. Zugriff am: 21.12.2020. 2020.

[Goo+20] Ian Goodfellow u. a. "Generative adversarial networks". In: (2020).

[Goont] Google. *The Generator*. https://developers.google.com/machine-learning/gan/generator. Zugriff am: 21.12.2020 14:17. unbekannt.

[GRE19] John Gillick, Adam Roberts und Jesse Engel. *GrooVAE: Generating and Controlling Expressive Drum Performances*. https://magenta.tensorflow.org/groovae. 2019.

[Gup17] Rajendra Gupta. *What is Deep Learning and Neural Network*. https://www.houseofbots.com/news-detail/1443-1-what-is-deep-learning-and-neural-network. Zugriff am: 22.12.2020 14:00. 2017.

[Her20] René Herrmann. "Hardware Sequencer Projekt". In: (2020).

[Insnt] Mutable Instruments. *Grids*. https://www.modulargrid.net/e/mutable-instruments-grids. unbekannt.

[Kera] Keras. *Concatenate layer*. https://keras.io/api/layers/merging_layers/concatenate/. Zugriff am: 27.12.2020 17:20.

[Kerb] Keras. *Conv2D layer*. https://keras.io/api/layers/convolution_layers/convolution2d/. Zugriff am: 27.12.2020 17:20.

[Kerc] Keras. *Conv2DTranspose layer*. https://keras.io/api/layers/convolution_layers/convolution2d_transpose/. Zugriff am: 27.12.2020 17:20.

[Kerd] Keras. *Dense layer*. https://keras.io/api/layers/core_layers/dense. Zugriff am: 27.12.2020 17:20.

[Kere] Keras. *Dropout layer*. https://keras.io/api/layers/regularization_layers/dropout/. Zugriff am: 27.12.2020 17:20.

[Kerf] Keras. *Embedding layer*. https://keras.io/api/layers/core_layers/embedding/. Zugriff am: 27.12.2020 17:20.

[Kerg] Keras. *Flatten layer*. https://keras.io/api/layers/reshaping_layers/flatten/. Zugriff am: 27.12.2020 17:20.

[Kerh] Keras. *Keras layers API*. https://keras.io/api/layers/. Zugriff am: 27.12.2020 17:20.

[Keri] Keras. *Keras layers API*. https://keras.io/api/layers/normalization_layers/batch_normalization/. Zugriff am: 27.12.2020 17:20.

[Kerj] Keras. *LeakyReLU layer*. https://keras.io/api/layers/activation_layers/leaky_relu/. Zugriff am: 27.12.2020 17:20.

[Kerk] Keras. *Losses*. https://keras.io/api/losses/. Zugriff am: 27.12.2020 17:20.

[Kerl] Keras. *Probabilistic losses*. https://keras.io/api/losses/probabilistic_losses/. Zugriff am: 27.12.2020 17:45.

[Kerm] Keras. *Regression losses*. https://keras.io/api/losses/regression_losses/. Zugriff am: 27.12.2020 17:45.

[Kern] Keras. *Regression losses*. https://keras.io/api/losses/regression_losses/. Zugriff am: 27.12.2020 17:45.

[Kero] Keras. *Reshape layer*. https://keras.io/api/layers/reshaping_layers/reshape. Zugriff am: 27.12.2020 17:20.

[Lea20] Leah. *Künstliche Intelligenz im Kundenservice — der komplette Guide*. `https://www.userlike.com/de/blog/kuenstliche-intelligenz-kundenservice`. Zugriff am: 11.12.2020 14:50. 2020.

[LH07] Shane Legg und Marcus Hutter. "A Collection of Definitions of Intelligence". In: (2007).

[MH08] Laurens van der Maaten und Geoffrey Hinton. "Visualizing Data using t-SNE". In: *Journal of Machine Learning Research* 9.86 (2008), S. 2579–2605. URL: `http://jmlr.org/papers/v9/vandermaaten08a.html`.

[Misnta] Missinglink.ai. *7 Types of Neural Network Activation Functions: How to Choose?* https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/. Zugriff am: 22.12.2020 14:32. unbekannt.

[Misntb] Missinglink.ai. *Neural Network Bias: Bias Neuron, Overfitting and Underfitting*. https://missinglink.ai/guides/neural-network-concepts/neural-network-bias-bias-neuron-overfitting-underfitting/. Zugriff am: 22.12.2020 14:35. unbekannt.

[MO14] Mehdi Mirza und Simon Osindero. *Conditional Generative Adversarial Nets*. 2014. arXiv: `1411.1784 [cs.LG]`.

[Ngnt] Andrew Ng. *Sparse autoencoder*. http://ailab.chonbuk.ac.kr/seminar_board/pds1_files/sparseAutoencoder.pdf. unbekannt.

[Ope19] OpenAI. *MuseNet*. https://openai.com/blog/musenet/. 2019.

[Pic04] Gualtiero Piccinini. "The First Computational Theory of Mind and Brain: A Close Look at Mcculloch and Pitts's "Logical Calculus of Ideas Immanent in Nervous Activity"". In: *researchgate* (2004).

[Rad+19] Alec Radford u. a. "Language Models are Unsupervised Multitask Learners". In: (2019).

[Raf+18] Colin Raffel u. a. *MusicVAE: Creating a palette for musical scores with machine learning*. https://magenta.tensorflow.org/groovae. 2018.

[Roent] Lutz Roeder. *Netron*. https://github.com/lutzroeder/netron. Zugriff am: 22.12.2020 19:50. unbekannt.

[Ros17] Adrian Rosebrock. *Multi-label classification with Keras*. https://www.pyimagesearch.com/2018/05/07/multi-label-classification-with-keras/. Zugriff am: 11.12.2020 15:30. 2017.

[Son19] France Sony Computer Science Laboratories (Sony CSL) Paris. *DrumNet: High-Level Control of Drum Track Generation Using Learned Patterns of Rhythmic Interaction*. https://sites.google.com/view/drum-generation. 2019.

[Teant] Google Brain Team. *TensorFlow*. https://github.com/tensorflow/tensorflow. unbekannt.

[Tena] Tensorflow. *Create an op*. https://www.tensorflow.org/guide/create_op. Zugriff am: 27.12.2020 17:45.

[Tenb] Tensorflow. *Deploy machine learning models on mobile and IoT devices*. https://www.tensorflow.org/lite. Zugriff am: 27.12.2020 17:55.

[Tik19] Aleksey Tikhonov. *Drum Patterns from Latent Space*. https://towardsdatascience.com/drum-patterns-from-latent-space-23d59dd9d827. Zugriff am: 19.12.2020 14:03. 2019.

[TS10] Lisa Torrey und Jude Shavlik. *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*. IGI Global, 2010.

[Upa19] Yash Upadhyay. *Introduction to FeedForward Neural Networks*. https://towardsdatascience.com/feed-forward-neural-networks-c503faa46620. Zugriff am: 19.12.2020 13:35. 2019.

[Wan03] Sun-Chong Wang. "Artificial Neural Network". In: *Interdisciplinary Computing in Java Programming*. Boston, MA: Springer US, 2003, S. 81–100. ISBN: 978-1-4615-0377-4. DOI: 10.1007/

978-1-4615-0377-4_5. URL: https://doi.org/10.1007/978-1-4615-0377-4_5.

[Wie17] Lukas Wiest. *Recurrent Neural Networks - Combination of RNN and CNN.* https://wiki.tum.de/display/lfdv/Recurrent+Neural+Networks+-+Combination+of+RNN+and+CNN. Zugriff am: 21.12.2020 14:10. 2017.

[Zha20] Xian-Da Zhang. *A Matrix Algebra Approach to Artificial Intelligence.* Springer, 2020.

Abbildungsverzeichnis

Tabellenverzeichnis